FUJITSU

FUJITSU Software Interstage Business Process Manager V11.4.1

Developer's Guide

J2U3-0062-11ENZ0(00)
August 2017

Publication Date	August 2017
Revision	11

High Risk Activity

The Customer acknowledges and agrees that the Product is designed, developed and manufactured as contemplated for general use, including without limitation, general office use, personal use, household use, and ordinary industrial use, but is not designed, developed and manufactured as contemplated for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could lead directly to death, personal injury, severe physical damage or other loss (hereinafter "High Safety Required Use"), including without limitation, nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system. The Customer shall not use the Product without securing the sufficient safety required for the High Safety Required Use. In addition, Fujitsu (or other affiliate's name) shall not be liable against the Customer and/or any third party for any claims or damages arising in connection with the High Safety Required Use of the Product.

Table of Contents

About this Manual

This manual describes how to use the Model API to customize and extend Interstage Business Process Manager to fit the unique needs of your organization.

Intended Audience

This manual should be read by:

- IT professionals who are interested in the development of business processes
- Independent Software Vendors who are interested in empowering their products with workflow
- Systems integrators who are interested in building workflow-enabled applications or extending existing applications with workflow

Note: Any occurrence of 'administrators' in this guide refers to 'Interstage BPM Tenant Owners' and **not** 'Interstage BPM Super Users'.

This Manual Contains

Here is a list of what is in this manual:

Chapter	Title	Description
1	Introduction	Introduction to Interstage Business Process Manager.
2	Architecture Overview	Description of the interaction between the system components.
3	Concepts	An overview of the structure and operation of process definitions, process instances, work items, filters, process editing, subprocesses, etc.
4	Using the Model API	Describes the system environment for application development with the Model API.
5	Designing Process Definitions	Programming examples for simple and complex process definitions, as well as for process instances.
6	Enhancing Interstage Business Process Manager	Description of the various means of enhancing Interstage Business Process Manager, including Java Actions and Advanced Filtering & Sorting.
7	Administration	Programming examples for administrative activities.

Chapter	Title	Description
8	Retrieving History Information	Describes how to retrieve the history information of a workflow from the database.
Appendix A	Web Services Interfaces	Information about Command Interface Web Services
Appendix B	Using the Interstage Business Process Manager Samples	Lists the samples provided with Interstage Business Process Manager, their structure and how to use them.
Appendix C	Customizing Forms	Describes how to customize QuickForms.
Appendix D	Supported JavaScript Functions	Lists and describes the JavaScript functions supported by Interstage Business Process Manager.
Appendix E	Enabling SSO Authentication Using the OpenID Provider Application	Describe how to enable the SSO Authentication using the OpenID Provider application.
Appendix F	Troubleshooting	Describes the Interstage Business Process Manager log files and specific error situations and provides troubleshooting information.
	Glossary	Glossary of terms.

Typographical Conventions

The following conventions are used throughout this manual:

Example	Meaning
`command`	Text, which you are required to type at a command line, is identified by `Courier` font.
screen text	Text, which is visible in the user interface, is **bold**.
Reference	Reference material is in *italics*.
`Parameter`	A command parameter is identified by `Courier` font.

Other References

The following references for Interstage Business Process Manager are also available:

- *Release Notes*

  Contains an overview of Interstage Business Process Manager, setup tips, and late-breaking information that could not make it into the manuals.

- *Interstage Business Process Manager Server and Console Installation Guide*

  Describes software and hardware requirements, setup procedure for Interstage Business Process Manager Server and Console
- *Interstage Business Process Manager Server Administration Guide*

  Explains how to configure and administrate Interstage Business Process Manager Server. This guide also describes the configuration parameters of the Interstage BPM Server.
- *Interstage Business Process Manager Studio User's Guide*

  Explains how to model processes using the Interstage Business Process Manager Studio.
- *Interstage Business Process Manager Tenant Management Console Online Help*

  Explains how to use the Interstage Business Process Manager Tenant Management Console user interface.
- *Interstage Business Process Manager Console Online Help*

  Explains how to use the Interstage Business Process Manager Console user interface.
- *API Javadoc Documentation*

  This HTML documentation provides the API and syntax of the packages, interfaces and classes for developing custom applications or embedding Interstage Business Process Manager into other products.

Abbreviations

The products described in this manual are abbreviated as follows:

- "Interstage Business Process Manager" is abbreviated as "Interstage BPM".
- "Microsoft® Windows Server® 2008" and "Microsoft® Windows Server® 2012" are abbreviated as "Windows Server".
- "Oracle Solaris" might be described as "Solaris", "Solaris Operating System", and "Solaris OS" in this document.
- "Java Development Kit" and "Java SE Development Kit" is abbreviated as "JDK".

Export Controls

Exportation/release of this document may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

1 Introduction

Interstage Business Process Manager™ is a server-based workflow engine with APIs (Application Programming Interfaces) for workflow application development. It empowers developers or systems engineers to embed a workflow engine into their own products or systems which implement Interstage Business Process Manager. The target users of Interstage Business Process Manager are therefore system integrators and software product vendors.

1.1 Overview of Workflow

Workflow is a relatively new approach to automating business processes. The earliest attempts to automate business processes incorporated the logic for the flow of work in specialized applications. These specialized applications, however, were not easily updated every time the process changed. This typically resulted in a maintenance nightmare.

Workflow products and middleware were introduced to address this problem by encapsulating all the aspects of a process. This includes process-defining information, such as rules, routing paths, activities, data. More importantly, it includes the automation of the management of this information. An application that supports workflow could then implement any workflow without requiring intimate knowledge of the workflow process itself.

1.2 Overview of Interstage Business Process Manager

Interstage Business Process Manager (BPM) is a distributed Web-enabled workflow application development tool. It empowers business groups to collaboratively plan, automate, track, and improve business processes. More specifically, it allows an enterprise to design automated processes, which encapsulate the steps in the process, the responsible person(s) for each step, the order in which steps may take place, and the relevant data for the step or process.

Some key features include:

- Model API, which allows customized applications to communicate with the Interstage BPM Server or existing products to be workflow-enabled.
- Enterprise-wide, scalable infrastructure for handling processes of all types.
- Organizable and filterable universal to-do list.
- Central location for documents relevant to a process.

1.3 Advantages of Interstage Business Process Manager’s Design

The philosophy behind Interstage Business Process Manager is that an automated workflow system should minimize its users' involvement with the mundane and maximize their opportunities to use their creative capabilities. For instance, it enables a customer service or sales person to focus more effectively on servicing customers and less on processing the orders, complaints, or other issues.

In engineering, business development and R&D applications, the flexibility of the Interstage Business Process Manager allows it to monitor and enhance the creative efforts of an entire team, department, division or company. By particularizing the aspects of group creativity, the abilities and energies of each individual in the team can be optimally engaged. Development can be going on at the same time that testing is being performed without disturbing the running production processes, and the multi-tenant architecture allows all of this on a single server if desired.

To best achieve these goals, it allows you to extend the functionality of the clients and classes from which the product is built. This allows your organization to incorporate context-sensitive information in your implementation as well as customizing the application to the environment in which the work takes place.

1.4 Types of Workflow

There are the following industry-accepted types of workflow products: Production, Administrative, Ad Hoc, Process Discovery, Case Handling, Collaborative, and Component. Interstage Business Process Manager supports all of these types of workflow.

Production Workflow

Production Workflow was the first workflow technology to be designed because production workflow automates the core business processes that a company relies on to produce a profit. Production workflow ensures that even at high volumes, the business process reliably follows stringent predetermined rules that never change, controlling and monitoring that does specific tasks at what time and exactly what data they have access to.

An insurance company, for instance, might use production workflow to manage the work of numerous agents upgrading and selling a large variety of policies, and constantly processing claims. Production workflow can be used wherever sales, purchasing, manufacturing, accounting, and other core business processes require extensive, enterprise-wide coordination.

The earliest workflow products were software applications that were reengineered for specific industry needs. Unfortunately, since the workflow process was hard-coded, whenever the process changed, the changes also needed to be hard-coded into the application. Programmers therefore found it difficult to keep up with the evolution of the business. The development cycle, from program design and creating new code through testing and implementation, was so long that the workflow system sometimes became obsolete as soon as it was completed.

Administrative Workflow

Administrative Workflow is a low volume subtype of production workflow, used to ensure that regular business practices stringently adhere to company policy. Examples of administrative workflow include purchase orders, travel expense reports and reimbursements, vacation requests, and similar personnel and administrative functions.

Interstage Business Process Manager is capable of managing the stringent requirements of production or administrative workflow, and at the same time allows you to make changes almost instantly when the business requirements change.

Ad Hoc Workflow

Ad Hoc Workflow is designed as an unstructured, free form workflow, which puts few, if any, constraints on the process. A key feature required for ad hoc workflow is the ability to modify or alter a running process.

Ad hoc workflow can be used as a planning tool in which the user might only instantiate a process definition once, because the process is never the same from one time to the next. A company would use workflow in this way if it's impossible to predict how the end of a project will look or if it's only possible to partially specify where it's headed. In other words, as a user, you would create a partial process definition, run the process instance, and extend it as you go along.

A pharmaceutical company might use ad hoc project management workflow while testing a new drug to satisfy conditions set by the FDA (Food and Drug Administration in the U.S.) or by a software

development company while evolving a new product. These are both situations where the project's requirements are unpredictable and undeterminable. As the company acquires experience, however, it would build a knowledge base, and the process would become better defined; however, the nature of such situations may be that each process is unique and constantly mutable.

Process Discovery Workflow

Process Discovery is a type of workflow that has a goal of defining a repeatable workflow system. Process discovery uses workflow to record, organize, and refine existing business processes while they are occurring. Process discovery is appropriate in those situations where individuals have evolved their job description by the seat of their pants. It is also useful in situations where no procedure yet exists but where one is being created and refined on the fly.

Although process discovery uses a form of ad hoc workflow, the processes that are discovered and defined through the use of workflow can later become production processes. Thus, the use of workflow for process discovery enables the existing work of a company to be automated without stopping the existing flow of production.

Interstage Business Process Manager provides both subprocess and process editing capabilities, which enables users to use it for both ad hoc workflow and process discovery. Both of these types of workflow can lead to highly organized systems of dealing with unanticipated business events, evaluating the effectiveness of improvised responses and creating effective procedures from what works.

These capabilities also make it a good platform for "growing" complex case handling workflow, described below. Case handling workflow tends to involve a multifaceted set of variables, which must be handled both in a modular manner and as a single, unified case.

Case Handling Workflow

Case Handling Workflow is a subtype of production workflow. It is used to assist in complex decision-making situations where each case might be completely unique, and yet the entire case needs to be handled in the correct order and with due consideration of all dependencies within the system.

An example of case handling might be the workflow employed by a law office that specializes in divorce. Every divorce is different, yet there are many different common aspects, which might arise: division of assets, the house, the children, retirement funds, spousal support, and spousal protection. Such a complex set of details can easily become an unmanageable nightmare that is difficult to understand or handle properly.

By using subprocesses to break down the overall process, each component of such a process definition might be designed as a module, which is linked together through a master process definition.

Another example might be using workflow for case handling in a tax consultant business. When a corporation considers expansion, buying a smaller company, or moving to a new state, for instance, it needs to evaluate the impact of taxes fully. There are a huge number of factors and dependencies involved in such a decision making practice, including different types of taxes for real estate, tax structures around R&D, state laws, and corporate restrictions. Managing this amount of complexity would involve a carefully sequenced hierarchy of subprocesses so that all of the required information is available at every step of the process.

Collaborative Workflow

Collaborative Workflow has two meanings. The most common meaning is a system in which the users must continuously see proactively updated information in real time. In addition, fat clients are

typically used to provide complete information to whoever happens to be available to take the case. An example of a department that might use this type of workflow is a customer support center.

Collaborative workflow can also refer to a form of workflow that might be used for project management of a large and complicated process, in which you might plan your team's activities for the next two months, evaluating and modifying your plans periodically along the way. Using workflow software would enable you to efficiently share data, review documents in an organized fashion, and process requests for approval by linking software components such as a word processor, spreadsheet program, and electronic mail program to enable participants to work on shared data.

Component Workflow

Component Workflow is workflow that drives software and automated processes without human intervention. Many of the nodes in this type of workflow might be Simple Exclusive Gateway Nodes, which invoke human interaction only when a business exception occurs. But so long as everything is moving along smoothly, the processes are entirely automated.

You might think of component workflow as one of your goals in your business production processes. You might design a labor-intensive business process for the moment, and whenever new technology appears that can automate steps in the process, you simply change a few nodes in your process definition to integrate the features of your new automation into the process as it already exists. Interstage Business Process Manager provides the environment that makes this form of gradual automation straightforward and orderly.

Open Architecture

Interstage Business Process Manager uses an open architecture to support all of these types of workflow. This allows a company to create an enterprise-wide system, which coordinates the work of the company end to end. Since most companies have different departments and responsibility groups with diverse modus operandi, it is a tool that can unite the different types of workflow needed throughout.

Interstage Business Process Manager supports various operating system platforms, includes a Model API for applications to speak to the Interstage BPM Server with, supports RDBMS databases, and allows routing of any kind of work (forms, images, and executables). In addition, its flexible architecture is designed to facilitate changes to a workflow dynamically and to support integration with an existing infrastructure.

1.5 Creating Your Own Application

The purpose of this guide is to enable you to embed Interstage BPM into your own products or systems and extend delivered Interstage BPM EJBs. To do this most effectively and practically, we recommend that you first consult the examples provided in the `client/samples` subdirectory of `<engine directory>` and consider what is built into the sample classes, and that you create a strategy for extending those capabilities in the application that you design. You should start with the specific capabilities and orientation of your end users and tailor the application to them.

Here are the basic possibilities that you can work with:

- You can create your own Java GUI using the Model API. Documentation for this API can be found in the `docs/apidocs` subdirectory of product media.
- You can extend the functioning of the system by using Java Action that can automatically manipulate the data and make decisions. Thereafter, less user interaction is needed to maintain the flow of information through your organization.

The Interstage BPM engine is language independent. You can create a user interface in any language you choose, and even support a multi-lingual user interface that switches between languages. One need only employ the standard language features of Java and JavaScript in your application. Sorting and collating is done in the order defined by the database that Interstage BPM is using. Full (UTF-16) Unicode support is available for all data values. You can integrate to other systems (e.g. document management or directory servers) regardless of the language they present to the user. Interstage BPM was designed from the ground up to be independent of the language that your users speak.

Your application defines and controls the access to information. All users must authenticate to the system for any aaccess to any part of the system. Users, groups, and roles are defined and maintained by the applicatin administrator. Beyond that, your application can control access to information in any way necessary for the particular domain, preventing unauthorized users from seeing or updating information they should not have access to. All roles and security policies are strictly obeyed.

2 Architecture Overview

Interstage Business Process Manager (Interstage BPM) is a server-based workflow engine with APIs (Application Programming Interfaces) for workflow application development. It empowers developers or systems engineers to embed a workflow engine into their own products or systems which implement Interstage BPM.

Some key features of Interstage BPM include:

- API, which allows customized applications to communicate with the workflow engine or existing products to be workflow-enabled.
- Enterprise-wide, scalable infrastructure for handling processes of all types
- Organizable and filterable universal to-do list
- Central location for documents relevant to a process

Interstage BPM can run on multiple application servers providing load balancing and failover capabilities for non-stop operation with nearly 100% reliability. One customer installation ran a production server non-stop, 7x24, and using careful 6-Sigma methods measured a server up-time to be 99.996% and the only reason it was not 100% was a small computer hardware maintenance issue. Therefore, Interstage BPM is ideally suited for large mission critical applications deployed on the leading J2EE-compliant application servers. Refer to section *System Architecture* on page 20 for more information.

Interstage BPM can be used together with the following integration components:

- **UDDI Repository**: UDDI provides storage infrastructure for Web Services registries, Meta model repositories, as well as data necessary for auditing, access security and versioning. Refer to section *UDDI Repository* on page 24 for more information.

2.1 Configuration Overview

Interstage BPM can be run in the following configuration:

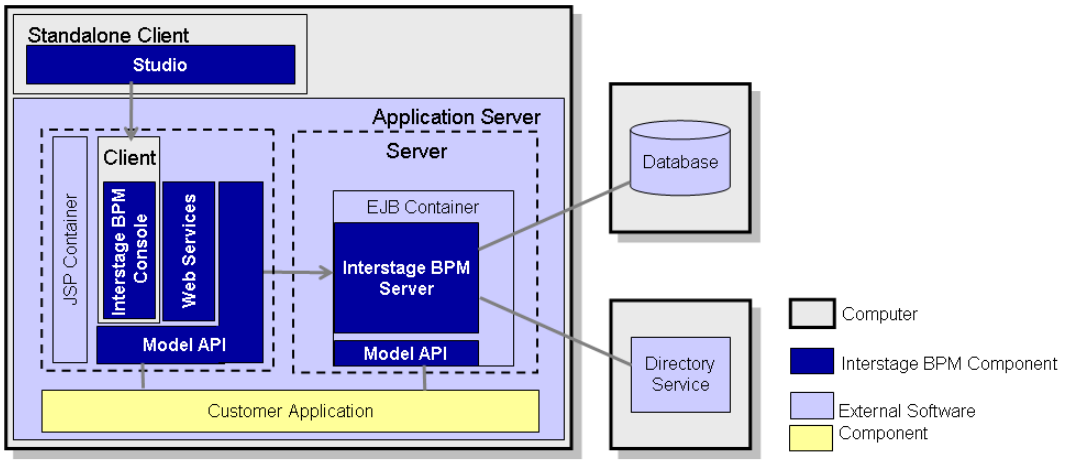

Figure 1: Configuration Overview

The Interstage BPM Server operates with a Database and optionally a Directory Service. The Interstage BPM Clients are used to access the Interstage BPM Server.

You can setup the components that make up a complete Interstage BPM setup in various configurations:

- All systems are setup one and the same computer
- One or several of the following is setup on separate computers:
  - Interstage BPM Server and Console
  - Database
  - Directory Service
  - Studio

Refer to the *Interstage Business Process Manager Server and Console Installation Guide* for details.

Note: Model API must refer the NamingService of Interstage BPM Server. Therefore, if the customer application is referring its own NamingService that is available on another machine, it is necessary to integrate that NamingService with the NamingService of Interstage BPM Server. The NamingService of the customer application cannot independently refer the NamingService of Interstage BPM Server.

2.2 System Architecture

Interstage BPM basically consists of a Server and a Model API. Several connectivity options allow for the integration of third party tools and other systems. This section provides an overview of the Interstage BPM components and their interaction.

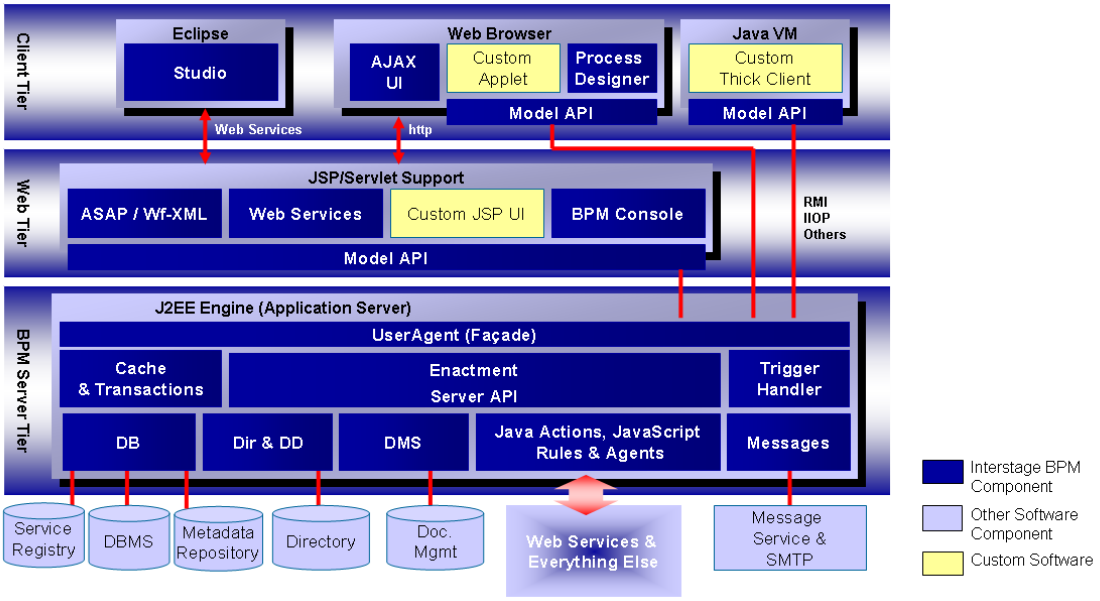

Figure 2: Architecture Overview

2.2.1 Interstage BPM Server Tier

The Interstage BPM Server is running inside an application server providing an Enterprise Java Bean (EJB) interface. The server negotiates interaction between users and other components, enacts

processes started by users, and notifies users of changes in status within a process. Interstage BPM can be configured using the standard capabilities of the application server, and managed using standard tools that work with application servers such as standard backup software, virus protection software, and distributed install software.

Interstage BPM is setup into a standard application server that isolates the application from operating system and database differences. Interstage BPM utilizes the facilities of the application server to provide, for example, for clustering, load balancing and failover capabilities. Standard application servers provide SNMP connectivity to allow monitoring of the server through SNMP. Interstage BPM can run on just about any operating system and in any hypervisor (e.g. VMware vSphere, Oracle VirtualBox, etc.) where the application server can be run. (The specific certified environments are listed in the release notes.)

The server is composed as a collection of EJBs that run in an application server, and make use of application server functionality. The Interstage BPM EJBs participate in container transactions so that the server and any client application can participate in the same transactions. Container-based transactions ensure a consistent state of the server. The only way to access the server is through the Model API, which allows for all forms of automated access: automated reporting, external system demand integration, and even automated testing of implemented processes.

The subsequent sections describe the EJBs in more detail.

User Agent (Façade)

The User Agent EJB (UA bean) enables the client to “log in” to the Interstage BPM system and validates the client for further interaction with Interstage BPM. First, the client requests a UA bean to be created; next the server creates the UA bean and returns a handle to it to the client. Then the client provides user name, password and server name for login to the server; this information is validated either through Interstage BPM's local user management capabilities or using the functionality of a connected Directory Service.

A UserAgent instance represents the login session to the server. It holds information for that particular login session. As the client’s agent, the UserAgent makes bean requests and method calls to the other Interstage BPM components on behalf of its client, i.e. it acts as a gateway for the model to access the process definition, process instance, work item, directory, and other objects. Therefore, the UA bean is also referred to as **Façade**, representing the interface between the server and the model.

Another function of the UA bean is to interpret the various filters on process definition, process instance and work item objects. Upon a client’s option to log out, the UserAgent will do all the necessary cleanup of resources held on behalf of the associated client. In addition, the UA bean implements session synchronization.

The server requests a unique instance of the UA bean for each client that logs into Interstage BPM. It requests these instances from the EJB container that is part of the application server. In a manner of speaking, the Server bean is a factory that “produces” instances of UA beans.

Enactment Engine - Process Definition Interpreter

The Process Definition Interpreter is the heart of the Interstage BPM Server. It is responsible for enacting a process defined with Interstage BPM. The server communicates with the Database adapter to maintain process state data, process instance and activity-related data, and process history information. The server controls database request queues.

There are two types of entity beans representing data objects holding the information about process definitions and instances. Upon enactment of a process, a Process Definition EJB is created, which

in turn creates Process Instance beans. Both beans implement the application server functionality of Bean-Managed-Persistence and Container Transactions.

They are not exposed to the model. All requests to the model pass through the UA bean(s).

Process data, e.g. information about the current state, is stored in and, on request, retrieved from the database. The server communicates with the database to maintain process state data, process and activity-relevant data.

Messaging

A combination of message-driven beans (MDBs) and a Java class library implement the Interstage BPM's type system (Meta model). Process enactment events are encapsulated in JMS messages that the MDBs process. Interstage BPM makes use of the default application server functionality.

In Interstage BPM, message-driven beans (MDBs) realize the flow of information between the server components by means of asynchronous messages.

For example, there are the following MDBs:

- Enactment Message bean: When a process instance is created, this bean generates a message so that the client is informed about this.
- Email Dispatcher bean: Handles email messages to the client
- Action Agent bean: handles Action Agents

JMS connectivity and MDBs are the most effective way to integrate with standard Message Oriented Middleware (MOM) also known as an Enterprise Service Bus (ESB). Because messaging is a native part of Interstage BPM it is particularly easy and effective to integrate process through an ESB to other enterprise services.

Custom EJBs

Any application can implement EJBs that run on the same application server setup as the Interstage BPM Server. Custom EJBs use the Model API, and Interstage BPM EJBs can call custom EJBs using Java Action.

User and Group Management

Every user that is to work with Interstage BPM needs a user account and must be assigned to one or more groups. Groups are used to determine who is responsible for carrying out a task in a process.

Interstage BPM comes with its own user and group management capabilities. Interstage BPM also allows for connecting to a Directory Service. Depending on your choices when setting up the server, users are managed either in Interstage BPM's local user store or in a Directory Service. Groups can be managed in Interstage BPM's local group store, in a Directory Service or in both systems.

Persistence

All process data and metadata is stored in a database, even when a process appears to the user to be running. The process instance can be in a running state for an unlimited amount of time. The number of processes instances that can be running at the same time is also nearly unlimited. The only limit to the number of processes that can run at once is the amount of disk space that the database needs to store the data. A relatively small process might take 100KBytes per process instance, and 10 million such process instances will take about 1TB of disk space. A more complicated process will take a proportional amount more, yet the real limitation on the number of processes remains only the space needed for the database.

The only time a process instance is read out of the database is when it needs to change state, for instance when a user completes a task, an event is received for that process instance, or a timer goes off causing an escalation or other timed activity. We refer to every such update as a transaction. The primary limit to scalability of a server is the limit to the number of transactions that can be performed in a unit time. The exact number of transactions per unit time depends upon the speed of the hardware, the speed of the IO, the latency of the network, the amount of data in each transaction, etc. As a rule of thumb, small servers can handle 10,000 to 20,000 transactions per hour, while large servers can reach 100,000 and possibly even 1 million transactions per hour. By counting the number of steps in a single process, you can estimate how many processes per hour you can process.

Persisting all process data in the database makes the servers stateless with regard to a given user. This means that there is no significant overhead caused by a user logging into the system. Adding users to the system is just a matter of adding the userids, and there is no critical dependency on the number of users. The system is limited instead by the number of transactions that it can handle. The exact number of transactions per unit time depends upon the speed of the hardware, the speed of the IO, the latency of the network, the amount of data in each transaction. For example, 100,000 users making one update per hour would produce virtually the same load as 100 users making 1000 updates per hour. The number of users, by itself, does not matter. Because the server does not hold any information outside of the database, a clustered environment can scale to any number of servers, and servers can be dynamically added and removed on demand.

The secret to the high reliability of the engine is the proper use of database transactions. All updates for a given operation are committed in a single quick database transaction. The process instance state is either completely updated, or not changed at all. This guarantees that the process instance is always in a consistent state, no matter what happens -- even unplugging the server in the middle of an update will not leave the process instance in an invalid state.

- **Database (DB)** adapter using the JDBC standard. The server provides the communication mechanism between the server and a database server. The database persistently stores and maintains all process information. The DB Adapter is responsible for the translation of server internal objects into persistent databse formats. Included with Interstage BPM is an adapter that persists the structures in a relational database using JDBC and some stored procedures. If a non-relational database, or other exotic database, is required it would be possible to write a custom DB adapter to meet this need.

Connectivity

The Interstage BPM architecture allows for the integration with third-party products. The server can communicate with the other components via "adapter classes". An adapter behaves as a converter that allows the server to speak to a common interface. Interstage BPM allows for connecting to the following:

- **Directory Service (Dir & DD)** adapter which implements an Interstage BPM specific interface to expand a user group into a list of individuals. The enactment engine uses this at runtime to determine who to give work items to. The Directory Adapter uses the LDAP standard. Currently, Microsoft® Active Directory and Oracle Directory Server Enterprise Edition is supported.

  The DD Framework Adapter is used by the User Agent at login time to authenticate users.

- **DMS** adapter which is used to interface the Interstage BPM system to external file systems using standard copy and transfer protocols. Forms, attachments, process definitions, etc. can be stored in a file system. A locator for such documents is stored in the attachments attributes of a process instance. An adapter for the most common procol, WebDAV, is included in Interstage BPM by default and can be used to access most documenet management system. Access to other document management systems may require a custom DMS Adapter to be installed. For example,

installing a CMIS adapter would allow access to document management systems that support CMIS protocol.

- **Messages** using the SMTP standard. Email can be sent from the server to the SMTP mail server as a response to Interstage BPM events.
- **External Systems**:
  - **Java Actions and JavaScript**: You can implement Java Actions for connecting to any external system, such as CRM or ERP systems. Java Actions are extensions to the workflow engine.

    Java Actions are data structures in the process definition that tell the Interstage BPM Server how to call a particular Java method during execution. Java Actions make application integration easier and calls to external applications and adapters faster.
  - **Agents**: Agents in Interstage BPM are set up to run automatically and act asynchronously on your behalf. You can use Agents to access external systems such as legacy systems or Web Services, both inside and outside of company firewalls. Using Agents, you can incorporate these external services into your Interstage BPM process instances. This mode of integrating Java is particularly convenient when multiple retries may be required.
  - **Rules**: The Rules Engine Bridge is a Java Action that is included in Interstage BPM in order to invoke rules engines like the iLog JRules Engine. Rules have all the same capabilities that are available from JavaScript, so you can think of the rules as a kind of scripting engine.

2.2.2 Interstage BPM Web and Client Tier

The **Model API** is an abstraction over the server and provides a single unified API to the server. The Model API runs in the client process, and handles all the communications to the server.

Interstage BPM comes with several client applications, for example, the Studio and the BPM Console.

Except for the Studio, the clients run in a servlet engine and are accessed using a Web browser. They are comprised of a combination of Java User Interface classes. Such client components are structured in two layers: a model layer (using the Model API) and a user interface layer (using Java User Interface classes). The model layer encapsulates the state of the client objects and interacts with the server. With using the Model API, you can develop your own clients and user applications.

The Studio is a standalone process design tool that can be installed separately. The Studio is independent of application server functionality. It interacts with the Interstage BPM Server through the Web tier.

Web Service capabilities is included in the web tier.

Interstage BPM supports a kind of Web Services Interface that is known as an Asynchronous Web Services Interface. This is an implementation of a standard way to access process instances and other long running programs. The standard is known as the Asynchronous Service Access Protocol (ASAP).

Refer to the *Interstage Business Process Manager Server and Console Installation Guide* for details on the Interstage BPM Web Services.

2.3 UDDI Repository

A UDDI repository provides storage infrastructure for Web Services registries, Meta model repositories, as well as for data necessary for auditing, access security and versioning. In addition it provides a web-based interface to visualize reports that analyze the usage of Web Services in process instances, orchestrations and information integration queries. As a result, business analysts, architects and developers can all collaborate, eliminate business risk related to change in IT assets and avoid the disruption of critical business processes.

From a functional point of view a UDDI repository manages metadata generated from integration software, Web Service descriptions, application specific data, and in general it serves as a central store for documents in native XML and non-XML formats.

UDDI is an industrial standard and serves the known registry functionality such as publicizing, discovering and staging consumption of Web Services. Publishing, discovering and retrieving Web Services capabilities provided by Interstage BPM is based on standard UDDI interfaces, and therefore you can use CentraSite from Interstage BPM as a UDDI registry implementation.

WebDAV is another industrial standard and can be used for storing and retrieving development artefacts, which are stored in standard formats such as XPDL. Interstage BPM provides the capability of publishing metadata into WebDAV and therefore you can use CentraSite from Interstage BPM as a WebDAV repository implementation.

2.4 Interstage BPM as a Service (SaaS)

Interstage Business Process Manager offers the option of being used in SaaS (Software as a Service) mode. If you use Interstage BPM in SaaS mode, you can create multiple tenants and lease out Interstage BPM to these tenant organizations, who will use it as a service. Note the following:

- When Interstage BPM is set up, it is automatically used in the SaaS mode.
- An organization that leases out Interstage BPM to other organizations for use as a service is called a service provider.
- An organization that uses Interstage BPM as a service from the service provider will use Interstage BPM as a 'tenant'.
- A service provider user who administrates tenants is called a Super User. Functionality of a Super User is limited to only managing tenants through the Interstage BPM Tenant Management Console, and managing the Interstage BPM Server.
  - Interstage BPM Tenant Management Console setup is automatically performed when you setup Interstage BPM Console.
  - Information about using the Interstage BPM Tenant Management Console is included in the Interstage BPM Tenant Management Console Online Help
  - Information about managing the Interstage BPM Server is included in the *Interstage BPM Administration Guide*
- The Super User role cannot use or administrate Interstage BPM workflows.
- Default tenant is automatically setup when you setup Interstage BPM. Setting up a default tenant also sets up the default `System` application.
- **Even if you use Interstage BPM in the non-SaaS mode**:
  - The role of this Super User will be limited to managing Interstage BPM Server.
  - You will use all Interstage BPM functionality as a default tenant. You will not be allowed to create more than one tenant.
- Irrespective of whether you use Interstage BPM in SaaS mode or not, any operation on a workflow element will always be in the specific context of an application. For example, before you create a process definition or process instance, you need to choose an application. However, choosing an application is optional. To know about the behavior when an application is not chosen, please refer the `ApplicationSecurityMode` parameter in the *Interstage BPM Administration Guide*.

3 Concepts

From start to finish, the developers and users of a typical workflow application may perform some or all of the following operations:

- Connect to the Interstage BPM Server
- Build a process definition
- Define and associate a form with a User Task Node
- Start a new process instance
- Modify a process instance
- Modify variables through Java Actions
- Associate an attachment with a process instance
- Execute an activity choice option
- Obtain a list of process definitions, process instances and work items
- Obtain the status of process instances and activities

The work performed by these operations can be best understood through the concepts that are discussed in this chapter.

Note: The Interstage BPM Server uses JMS messages for internal communication. JMS notification is used internally only. You cannot write external applications to receive JMS messages from the Interstage BPM Server.

3.1 Process Definitions and Process Instances

At its core, workflow is a medium of collaboration or coordination for any complex project. The basic building block of this collaboration is the process definition. Process definitions are like recipes in a cookbook: they provide you with the steps to follow, with references to the ingredients you must use, and criteria for deciding that the job is done. Like a recipe, a process definition is a static, reusable method, which guarantees a predictable result.

A process instance is the individual execution of the process definition, much like the specific act of baking a cake that you create by following a general recipe. In other words, the process instance provides a day-to-day context for coordinating the work on a project. It ensures that all of the cooks have the right ingredients at the right time, and that everyone is using those ingredients according to the agreed-upon recipe. In a business context, these ingredients generally consist of documents and data.

Since a process instance merely reflects the structure and functionality that was designed into the process definition, there is a close correspondence between the subordinate or dependent objects of process definitions and process instances. This chapter therefore focuses on process definition-related classes first to provide the background for understanding the whole system.

3.2 Process Definitions

Process definitions encapsulate all of the aspects of a process instance that can be set at design time. Aspects include predefined attributes, operations to be performed when the process instance is run, definitions for variable data that will be set at run-time, and the state of the process definition.

Process Definition Attributes

Process definition attributes include (but are not necessarily limited to) the following metadata about the process:

- Process definition state
- Process definition identifiers
  - ID
  - Version
  - Owner
  - Name (optional)
  - Title (optional)
  - Description (optional)
- Flow control
  - Nodes
  - Arrows
- Process definition ownership
- Variable data
  - User Defined Attributes
- Java Actions
  - Init Action Set
  - Commit Action Set
  - onSuspend Action Set
  - onResume Action Set
  - onAbort Action Set
  - Error Actions
- Timers including Timer Actions

Additional metadata about the process can be stored in Extended Attributes which store extra information that is not native to Interstage BPM. A process designer can define new extended attributes on process definitions, nodes, and arrows.

Conceptually, a process definition is one of a family of process definitions which have the same name. The family is collectively considered to be the process, and each member of this family is version of process. The first version of a process definition installed into an application is numbered 1.0. The second process definition installed with the same name into a server will be numbered 2.0, and so on. All versions of a process are available simultaneously for execution. A process instance that is started on version 5.0 may continue to execute version 5.0 even after version 6.0 has been installed. An unlimited number of versions of the process may be executing at the same time in the process engine.

3.2.1 Process Definition States

A process definition can be in one of the following states:

- Draft
- Published
- Private

- Obsolete
- Deleted

To control the usage of all the versions of a process, each definition is assigned a state: draft, published, private, obsolete, or deleted. An application can have any number of draft, private, obsolete, or deleted versions of the process.

Process definitions in draft state are considered to be in design stage. Brand new process definitions, when they are created or imported, will aways start in the draft state. Only the owner of a draft process definition can create a process instance from it, presumably for testing purposes.

When the draft process definition is ready for general use, the application administrator can publish it, making it visible so others can start process instances from it. Only one version of a given process may be designated as published at a time. When a user chooses to start a process by name, it will be the published version that will be started. When a new version of a process is published, the previous published process will be redesignated as obsolete.

An obsolete process definition can continue to be used just as well as it ever could be. Process instances running that version of the process continue without any diminishment. The only difference is that an obsolete process version is not as visible as a published process version, when being selected by name. An application should always select a process definition by name, so that new processes will always be started on the latest published version. Once a process is running, it remains on the same version it started on, unless the process administrator uses the operation of process migration to move it to the latest version.

If a user modifies a running process instance, a private process definition is created for just that one process instance. A private defintion can not be used to create new instances, it remains forever tied to the one instance that was edited. If you want to use private process definition in future new process instances, you must copy the definition a new draft process definition.

You cannot edit a published or obsolete process definition -- because they might have instances running on them -- but you can make a new copy of them. The new copy form a new draft version of the process. This new copy of the process definition can be edited until it is published, and the cycle continues.

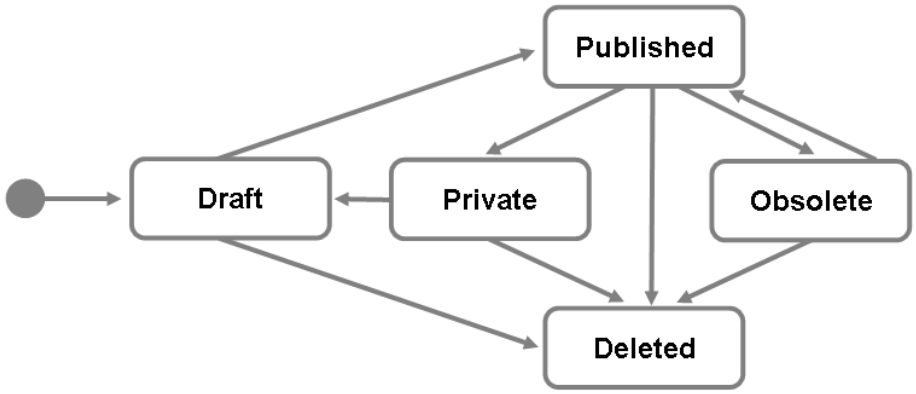

Figure 3: Process Definition State Transition Diagram

If you want to take a process definition out of circulation, an administrator can change the process definition state from published to obsolete. If you want to delete a process definition, an administrator can change the state to deleted. When draft and private process definitions are deleted, they are

dropped from the system. When published and obsolete process definitions are deleted, they are not dropped from the system, but you cannot modify their state again. You can create new versions from draft, private, published and obsolete process definitions, but not from deleted process definitions.

3.2.2 Process Definition Identifiers

A process definition is identified by the following attributes:

- ID
- Version
- Owner
- Name (optional)
- Title (optional)
- Description (optional)

The ID is a variable of data type Long, which is automatically assigned by the Interstage BPM Server to provide a unique handle for the process definition. The version number is automatically assigned, and the owner is the creator of the process definition.

The other three identifiers are optional. When designing an application, the `Plan` interface from the Model API gives you access to the name, title, and description if you desire.

Process Definition Versioning

New process definitions are assigned a version number of 1.0, for example `Purchase Order v1.0`. You can create a new version of a process definition without changing its name. If you copy a process definition the new version of the process definition will have the next whole number, for example `Purchase Order v2.0`.

Note: Versioning of a process is specific to the application to which it belongs. This implies you can have two processes with the same name without affecting their version number as long as the processes exist in separate applications.

3.2.3 Workflow Elements

A workflow process is a network of nodes connected by arrows. Nodes and arrows together are referred to as workflow elements.

Nodes represent the steps in a process. A step could be an activity where process participants are assigned specific tasks to be completed, and also be a place where a decision on the process flow is made. In the former case, including User Task Nodes, Voting User Task Nodes, Embedded Sub-Process Nodes and Call Activity Nodes, user actions are required or executable except for Call Activity Nodes. In the latter case, no user action is required.

Arrows are the connectors that guide the flow of the process instance from one node to another. The flow of events simply passes through the arrow on its way from one node to the next. They have little additional functionality except to change the state of their target nodes.

When first encountering it, application developers are usually surprised to learn that all nodes operate independently. An activity node is either "on" or "off". When it is "on", it has a work item in the work list, and it is available for receiving events from the user. In fact, all activities have the potential to be "on" at the same time. They reason they are not "on" all at the same time, is that the arrows carry a signal such that when one is completed, it starts the next one in the sequence. Each activity is triggered in sequence, just as the one before it is completed.

This has some practical consequences: The user has an tremendous amount of control over the running of the process. If an activity is completed by mistake, it is possible to go back and manually turn it on, so it will wait again for the user to complete it. If data had been incorrectly entered, causing the process to incorrectly go down the branch "A", it is possible for the user to turn "off" the activity on the incorrect branch, and to turn "on" the appropriate activity on branch "B". Even if the process is perfectly designed, and the users make all the right inputs, it is possible sometimes that the external events change the situation unexpectedly and the process needs to be set back to an earlier state. This is easy and straightforward for users to do.

Workflow elements have the following minimum attributes: ID, Name, Title and Description. ID is absolutely mandatory - the one attribute that all workflow elements must have. The Interstage BPM Server automatically assigns a unique ID to each element in a workflow process to identify it.

In addition to the attributes that all nodes have, specific node types have properties that are applicable to only them. In the discussion that follows, some properties are only applicable to certain individual node types - as required by their intended behavior.

Arrows have a simple set of identifiers: ID, Name, and Description. The ID is a long data type that is automatically assigned by the Interstage BPM Server to provide a unique handle for the process definition. The other two identifiers are optional.

In designing an application, the `Plan` interface from the Model API gives you access to the ID, Name, and Description fields, if desired.

3.2.4 Java Actions

Interstage BPM offers a "two level" design for modeling business processes. The upper level is the visible process diagram which contains nodes and arrows. The upper level can generally be designed by a business user sometimes known as a process analyst. The upper level is the important visible level which displays the coordination of actions by people, and it shows the branches and flow of the work.

The lower level is the additional parts that is used for integrating to the data resources and other servers. Lower level modeling is performed by adding Java Actions to the process diagram. While you don't need to be a programmer to add a Java Action, it is often the case that integration often requires someone who understands how the other data resources are designed and how they operate. There are relatively easy Java Actions that do things like send email, send text messages, or perform simple calculations. There are also quite sophisticated Java Actions that will access or update a database, and call a web service.

Technically speaking, a Java Action is just a piece of the Process Definition that calls out to a Java method. Java Actions can read or write UDA values, as well as perform any operation that is available in Java code. A Java Action has a user interface that helps in configuring the call that is to be made.

A Java Action is an instance of a Java Action Type. A single process might contain many instances of the same Java Action Type placed in various places in the Process Definition. For example there may be many points in a process that send email. The Java Action that sends a late notice to the owner of a bank account is one instance of the Send Email Java Action Type. Interstage BPM comes with a long list of predefined Java Actions Types for doing all the normally anticipated data integration and manipulation, but an application developer can install new, custom Java Actions Types. If it helps, you can think of the Java Action Type as a particular Java method that might be called, and the Java Action is the object in the process definition that says precisely how to call it.

Java Action (instances) are configured to specify where the inputs to the operation will come from, and what to do with the output if any. For example, a Java Action to store a value in a UDA will be configured to specify what value to store, and what UDA to store the value into. In the case of a web

service, the Java Action is configured to specify which web service to call, along with a lot of details on how to call it, what XML data to pass, and how to parse and store the returning information.

A Java Action Set is a place in the process definition that you can put Java Actions. Each set has a defined time or condition for which it will be triggered, and when triggered all the Java Actions in the Java Action Set will be executed. Nodes have a number of different Java Action Sets, and there are also Java Action Sets that exist on the process definition as a whole. For example:

- **Prologue Action Set**. Java Actions in the Prologue Action Set of a node are executed as the node is being activated. Java Actions in this set can be used to set up or initialize values associated with the node.
- **Epilogue Action Set**. Java Actions in the Epilogue Action Set are executed as the node is being completed and before the process instance moves on to another node. Java Actions in this set can be used to clean up values after the task is done, or communicate these values to external destinations after the intended work is finished.
- **Role Action Set**. Java Actions in the Role Action Set are evaluated as part of role resolution, after the group assigned to the activity is looked up in the directory server and before work items are created. Java Actions in this set can be used to dynamically compute a list of assignees for a task in conjunction with the assigned user group.
- **Error Action Set**. Java Actions in an Error Action Set are used for handling specific error situations that occur during execution. Error Action Sets exist for the entire process definitions, for individual nodes and for other Java Actions, i.e. when you want to react on errors occurring during the execution of another Java Action.
- **Compensate Action Set**. Java Actions in a Compensate Action Set can be used to "undo" a non-transactional system outside of Interstage BPM, e.g. a database when an error occurs and a previous Java Action needs to be undone (compensated). Compensate Actions are useful to ensure a consistent state of all systems which do not participate in a standard transaction.

Refer to section *Types of Java Actions* on page 92 for more information.

Java Actions in a Java Action Set will be referred to as `<Category> Java Action` or `<Category> Action`. For example, Java Actions in Prologue Action Set will be referred to as Prologue Java Action or Prologue Action.

3.2.5 User Groups & Organizational Model

Process definitions identify the right person for a task or notification through the use of an organizational model. The organization is composed by groups and relationships within the corporate LDAP directory server. If you are lucky you will be able to use the organizational units that already exist in the directory server as groups accessed directly by the process definition. There is no requirement to create new groups, however it is common to want to make new application-specific groups to represent roles for the application. Application specific groups can be created in the directory server, or they can be created in the application, whichever is more convenient.

Relationships defined between users in the directory server can also be used by a process definition. A person in the directory may be defined to have a manager, or an emergency contact. The process definition can use these relationships when it comes time to decide who to send a notification to, who to assign a task to, or who to escalate to. Through the use of organizational units and relationships, it is easy to escalate or send alerts to the appropriate organizational hierarchy depending upon who is responsible for a particular part of the process.

User Groups and assignee resolution are a basic, underlying feature of the system. A User Group is a name given to a set of organizational resources. Depending on your system configuration, user groups are defined in Interstage BPM's local group store, in a Directory Service or in both systems.

Using the Directory Service functionality, a User Group can also be mapped to any other organizational resources, including machines, software programs, parameterized function calls, or even queries. User groups might be used to identify experts on particular subjects, so that processes can access those group members in a process to notify them or to connect users who need their help.

Usually, a User Group is a name given to users who are grouped according to criteria that fit the needs of the business or organization. The most common of these criteria describe the kind of work performed by one or more persons in an organization. Users can also be grouped according to their authority, responsibility, skill, or profession: The workflow engine doesn't distinguish what the organizational resources are, so long as it can be identified with a string variable.

A node can be assigned a User Group if it represents a node that requires user involvement. In the current Interstage BPM model, only User Task Nodes, Voting User Task Nodes, and Embedded Sub-Process Nodes can be given an assignee.

3.2.6 Forms

Users access process information through some sort of user interface. A "Form" is a name we give to the concept of the mechanism that display that data, and allows the user to interact with it. There is no single specific form technology, and the application design is fairly free to use whatever user interface technology they are most familiar with. Technically, the form is a file which the front end client can use to display the data, and clearly the contents of this file depend greatly on the client program that reads and displays the form. Given the dominance today of web browsers being used as client platforms, Interstage BPM offers several different forms options based on HTML. Some customers have chosen to integrate sophisticated approaches like Adobe PDF Forms for displaying and manipulating process data. The most recent trend is the use of client UI frameworks, like AngularJS as the technology to use for forms. While Interstage BPM offers the ability to associate a form file with an activity node, the important thing to remember is that the Interstage BPM engine does not read or interpret the form file, it only delivers it to the client. The client, then, is solely responsible for interpreting the form, in order to display the data.

An attachment is a document which can be placed into the process instance, and is available for other users of that process instance. Forms are added to process definitions at design time, and attachments are added to running process instances. The server treats the attachment as an opaque file which is delivered to the client without any constraints on the contents. Processes can, however, be designed to open and read attachments if that is needed for the process.

Forms can be used to display or report data that is stored in User Defined Attributes and data from external data sources. Forms can also enable users to interact with, modify, or add data to the process instance or to an external data source. Refer to section *User Defined Attributes* on page 34 for using variable data in forms.

Forms are stored on a Web server and Interstage BPM only knows the URL with which the forms can be addressed. They are associated to a node by specifying the path to the form including the form name. Forms can be associated with Start Nodes, User Task Nodes, Embedded Sub-Process Nodes, and Voting User Task Nodes. Forms are presented to the user in starting a process instance or viewing a work item. Depending upon the client, forms can be used to provide rich user experience, including tables, lists, graphics, styled using the latest CSS and HTML5 coding techniques. The BPM engine does not limit the ability to use the web in its most powerful ways for making excellent user experiences.

3.2.7 Timers

Almost all business processes live and die according to timed criteria, including deadlines, interest costs, milestones, procrastination, project management, vacation schedules, personal organization - the list goes on forever.

The system environment allows you to create timers that trigger certain actions when they expire. These timers can be Node-Level timers that start running when the User Task Node is activated or Process-Level timers that start running whenever a new process instance is created from the process definition containing the timer.

Timers are controlled by time and action parameters, specified as User Defined Attributes, and set through Java Actions or forms. The expiration or firing time of a timer can be an absolute, specific time or relative to another event (such as the start of the activity or process instance). The timer can also be set to trigger actions periodically. Timers can also perform a variety of actions, including escalation (assigning the activity to an additional list of users) or sending emails.

The following timer types are available:

- **Relative timers**. A Relative timer calculates the time when they will expire based upon the time when the activity or process becomes active. They are only activated once.
- **Absolute timers**. An Absolute timer is set according to the absolute time when the timer will expire, measured in milliseconds since January 1, 1970, 00:00:00.
- **Periodic timers**. A Periodic timer is a Relative timer that operates repeatedly. The first occurrence of the timer is relative to the time when the activity or the process instance becomes active. Subsequent occurrences are relative to the last occurrence.
- **Business Periodic timers**. A Business Periodic timer is a Business Relative timer that operates repeatedly. The first occurrence of the timer is activated relative to the time the activity or a process instance becomes active, and subsequent occurrences are relative to the last occurrence.
- **Business Relative timers**. A Business Relative timer calculates the time that they will expire, based upon the time when the activity or a process instance becomes active, and using the 'business calendar' expression methods. They are only activated once.

The Business Periodic timers and the Business Relative timers are used within Business Calendars. For defining a timer with the Model API refer to section *Using Timers* on page 166 for details. For configuring a Business Calendar refer to the *Interstage Business Process Manager Server Administration Guide* for details.

3.2.8 Process Definition Ownership

The attributes for process ownership allow you to set permission levels in process definitions and process instances to individuals or groups. The purpose of process ownership is to identify users who are authorized to edit it structurally.

The owner of a process definition is the person who created it. When a process instance is created from the process definitions, this person is by default the owner of the process instance as well. However, process definition ownership and process instance ownership are two distinct attributes.

During process definition design, process ownership can also be directly assigned to members of a user group or programmatically assigned by using a Role Action. The Role Action interacts as described in the table below.

Set by the User Group	Set in the Role Action	The Process Owner Will Be
No	No	The process definition owner becomes the process owner.

Set by the User Group	Set in the Role Action	The Process Owner Will Be
Yes	No	The process instance owners are determined by the User Group.
No	Yes	The Role Action determines the process instance owners.
Yes	Yes	The process owners belong to the intersection between the assigned User Group and the Role Action. If this intersection is empty, the process definition owner becomes the process instance owner.

The Interstage BPM Server makes sure that users are authenticated and that their assigned Groups and their capabilities in accessing and manipulating process instances, activities and their attributes are identified. The use of Groups eliminates the need to modify every process definition when there is a personnel change. Instead, only the group definition in Interstage BPM's local group store or in the Directory Service needs to be updated.

3.2.9 User Defined Attributes

Another set of attributes assigned to the process definition consists of the User Defined Attribute definitions. These definitions describe data items that will be associated with process instances created from the process definitions. Process instance-specific values can therefore be stored in them. The values may be assigned either at the beginning or during the running of the process instance. This variable data is global to the process instance it is associated with. Any user may change it with access to it.

While designing a process definition, the designer determines what types of data will be used in the process instances and creates a strategy for assigning that data to variables. The data in the UDA variables can be displayed to the user in a form, and forms can also allow users to update those values. The values can then be used to produce output, (e.g. form letters to customers, invoices for accounting, or reports for management) and this output can be automatically sent via email, text messages, or printed and sent by post.

The data can also be modified through Java Actions. Refer to section *Configuring Java Actions, Agents and JavaScripts for External Applications* on page 88 for using Java Actions.

3.2.10 Node Types

Interstage BPM supports different node types. The following table gives an overview of the supported node types and their attributes. It does not contain attributes that are common to all node types.

Node Type	Assignee	Role Action Set	Prologue Action Set	Epilogue Action Set	onAbort, onSuspend, onResume Action Set	Error Action Set	Timers	Triggers	Forms
Start Node	-	-	-	-	-	-	-	-	Yes
End Node	-	-	-	Yes	-	-	-	-	-

Node Type	Assignee	Role Action Set	Prologue Action Set	Epilogue Action Set	onAbort, onSuspend, onResume Action Set	Error Action Set	Timers	Triggers	Forms
User Task Node	Yes	Yes	Yes	Yes	Yes	-	Yes	Yes	Yes
Voting User Task Node	Yes	Yes	Yes	Yes	Yes	-	Yes	-	Yes
Timer Node	-	-	Yes	Yes	Yes	-	Yes	-	-
Message Receive Node	-	-	Yes	Yes	Yes	-	Yes	Yes	-
Parallel Join Gateway Node	-	-	-	Yes	-	-	-	-	-
Parallel Split Gateway Node	-	-	-	Yes	-	-	-	-	-
Simple Exclusive Gateway Node	-	-	Yes	-	Yes	-	-	-	-
Call Activity Node	-	-	Yes	Yes	Yes	-	-	-	-
Remote Sub-Process Node	-	-	Yes	Yes	Yes	Yes	-	-	-
Chained-Process Node	-	-	Yes	Yes	Yes	-	-	-	-
Embedded Sub-Process Node	Yes	Yes	Yes	Yes	Yes	-	-	-	Yes

The following sections provide a detailed discussion of the different node types.

Note: There are some custom nodes that can be used in the Interstage BPM Studio. At API level, Service Task Nodes are equivalent to User Task Nodes. Similarly, Flexible Exclusive Gateway Nodes are equivalent to Simple Exclusive Gateway Nodes; Receive Task Nodes are equivalent to Message Receive Nodes; Send Task Nodes, Business Rule Task Nodes, Script Task Nodes, Email Nodes, DB Nodes and Web Service Nodes are equivalent to Parallel Split Gateway Nodes.

Start Node

The Start Node identifies the beginning of a process. Every process definition has one and only one Start Node.

Not supported: This node type does not have any assignees, timers or triggers.

Behavior:

- A process instance that is created from a process definition has to be explicitly started.
- Enactment of a process instance begins at the Start Node. When a process instance is started, an event is sent to the Start Node.
- The Start Node then immediately sends events to all the outgoing arrows.

End Node

An End Node identifies the end of a process branch.

Every process definition has at least one End Node. A process definition may have multiple End Nodes, which model different modes of completion of a process. With huge process definitions, multiple End Nodes can simplify the graphical representation of the process.

In a typical process definition, an End Node has one or more incoming arrows and no outgoing arrows.

Supported: Epilogue Action Set can be used as required.

Not supported: This node type does not support any kind of forms, assignees, timers or triggers.

Behavior: Enactment of the process instance stops at an End Node. When a process instance has multiple End Nodes, as soon as the process instance's enactment reaches one of the End Nodes it is deemed to be complete.

User Task Node

A User Task Node is an interactive node that represents one or more tasks performed by humans towards completion of a business process. Upon enactment of this node, this thread of events in the process instance pauses. It only continues after human intervention.

A well designed user task node will contain all of the information needed to help the novice user know what they should be doing. The description should be a complete description of the task to be accomplished. The description can contain links to reference documents and web resources that explain details on the legal or customary requirements of that task and how it fits together with all the other activities in the process. A well designed process will actually help train the user while supporting the correct flow of the process.

User Task Nodes can also stand-in for activities performed by external agents (such as other devices).

A process instance may have any number of User Task Nodes. A User Task Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Forms, as many as required, can be used. This node has a Prologue Action Set, Epilogue Action Set, assignee, Role Action Set, onAbort Action Set, onSuspend Action Set, onResume Action Set, timers and node-level triggers.

Behavior:

- The list of assignees using the assigned group is computed as follows:

Group Specified	Users Specified in Java Action	Assignees
Yes	No	User group belonging to this user group or individual members of user group
Yes	Yes, list intersects with user group	Members specified in both user group and Java Action
Yes	Yes, list does not intersect with user group	Process owner
No	Yes	Members specified in Java Action

- All timers defined for this node are activated.

- One group work item or individual work items are created for the assignees.
- When one of the assignees (not all of them) completes the task associated with the User Task Node, the assignee sets the process instance rolling again by indicating that the task is complete (e.g., by clicking a button). This is referred to as committing the work item.
- Each of the outgoing arrows of this node represents a direction in which the process instance can proceed from this node. Depending on the business process, an assignee can make the process instance continue in a particular direction by choosing the arrow that represents the direction.
- Once the work item is committed, the Epilogue Action is evaluated, any process timers that are still active are cancelled, and an event is sent to the arrow that has been chosen.

Voting User Task Node

A Voting User Task Node uses voting rules to determine the choice (node's outgoing arrow) that wins. For every Voting User Task Node the rules for voting must be specified. A voting rule specifies for one outgoing arrow the kind of controls to be used for voting, for example, TYPE_MAJORITY.

A process instance may have any number of Voting User Task Nodes. A Voting User Task Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Forms, as many as required, can be used. This node has a Prologue Action Set, Epilogue Action Set, Role Action Set, assignee, onAbort Action Set, onSuspend Action Set, onResume Action Set, and timer can be used as required.

Behavior:

- Upon receiving an event from any of its incoming arrows, a Voting User Task Node enters a running state. While in this state, it ignores any events that it receives from its incoming arrows. If the user group assigned to the node cannot be evaluated, the node and process instance containing it enter the error state.
- The Voting User Task Node then evaluates the Prologue Action.
- It then computes the list of assignees, using the assigned user group, as follows:

User Group Specified	Users Specified in Java Action	Assignees
Yes	No	Members of User Group
Yes	Yes, list intersects with User Group	Members specified in both User Group and Java Action
Yes	Yes, list does not intersect with User Group	Process owner
No	Yes	Members specified in Java Action

- All timers defined for this node are activated.
- Work items are created for all assignees.
- The voting rules defined for the Voting User Task Node are evaluated. A voting rule defines the kind of controls to be used for voting. The threshold of a voting rule defines the limit when the condition becomes true.
- When one of the voting rules is matched, it sets the process instance rolling again by indicating that the task is complete. This is referred to as committing the work item.

- Once the work item is committed, the Epilogue Action is evaluated, any process timers that are still active are cancelled, and an event is sent to the arrow that has been chosen.

Timer Node

A Timer Node represents a step in a process in which execution of the process instance (or a particular execution thread in a process instance) is suspended for a specified amount of time.

A process definition can have any number of Timer Nodes. A Timer Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Prologue Action Set, Epilogue Action Set, onAbort Action Set, onSuspend Action Set, onResume Action Set, and timer can be used as required. Although more than one timer can be specified on this node, only one is effective. This node has no forms, assignee or triggers.

Behavior:

- When a Timer Node receives an event from an incoming arrow, it evaluates any Prologue Action, activates all the timers specified, and waits. While waiting, it ignores additional events from incoming arrows.
- As soon as one active timer expires, the node cancels all other active timers, evaluates any Epilogue Action, and sends events to all outgoing arrows.

Message Receive Node

A Message Receive Node represents a step in a process which is driven by an external event. Each Message Receive Node has a trigger defined that is supposed to fire when data, typically XML files, comes in from an external system. Once the data arrives, the trigger moves the data into User Defined Attributes (UDAs) according to the trigger's definition. The trigger then makes a choice and process execution moves forward to the next node.

This node type does not involve any human interaction. Hence, no work items are generated when a node of this type becomes active. It can only be completed by a trigger defined on the node.

A process definition can have any number of Message Receive Nodes. A Message Receive Node can have one or more incoming arrows and exactly one outgoing arrow.

Supported: A Message Receive Node must have one or more node-level triggers (make choice triggers) defined. Prologue Action Set, Epilogue Action Set, onAbort Action Set, onSuspend Action Set, onResume Action Set, and timers can be used as required.

Unsupported: This node type has no assignee or Role Action Set.

Behavior:

- When a Message Receive Node receives an event from an incoming arrow, it executes the actions in the Prologue Action Set, activates all of its active triggers, and waits. While waiting, it ignores additional events from incoming arrows.
- As soon as one active trigger fires, the node evaluates any Epilogue Action, and sends an event to its outgoing arrow.

Parallel Join Gateway Node

A Parallel Join Gateway Node represents a step in a process where the process instance pauses to synchronize multiple threads of execution.

A process can have any number of Parallel Join Gateway Nodes. A Parallel Join Gateway Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Epilogue Action Set can be used as required.

Unsupported: This node has no Prologue Action Set, assignee, forms, timers or triggers.

Behavior:

- A Parallel Join Gateway Node waits till it receives at least one event on each of its incoming arrows. Any additional events received from the same incoming arrows are ignored while waiting to receive an event from each of the arrows.
- When each of the incoming arrows has received an event, all of the execution threads represented by the incoming arrows are synchronized. Any Epilogue Action is evaluated and events are sent to all the outgoing arrows.
- Thus, this node can also result in multiple branches.

Parallel Split Gateway Node

A Parallel Split Gateway Node represents a step in a process from where a single branch of control splits into multiple parallel branches. A process can have any number of Parallel Split Gateway Nodes.

This node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Epilogue Action Set can be used as required.

Unsupported: The semantics of this node type do not support Prologue Action Set, assignee, forms, timers or triggers.

Behavior: Every time this node type receives an event from one of its incoming arrows, it executes the Epilogue Action Set and sends events to all its outgoing arrows.

Simple Exclusive Gateway Node

A Simple Exclusive Gateway Node represents a step in a process where the process's execution proceeds in one of the many possible directions, depending upon the value of a certain User Defined Attribute. All of the outgoing arrows of this node represent various directions in which the process can proceed.

A process can have any number of Simple Exclusive Gateway Nodes. A Simple Exclusive Gateway Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Prologue Action Set, onAbort Action Set, onSuspend Action Set, and onResume Action Set can be used as required.

Unsupported: The semantics of this node type do not support Role Action Set, forms, timers or triggers.

Behavior:

- The decision that process instance should proceed with which arrow, depends on the value of the specified User Defined Attribute.
- The Simple Exclusive Gateway Node compares this value with the constant value that is associated with conditions specified by each outgoing arrow.
- When the Simple Exclusive Gateway Node receives an event from an incoming arrow, it first evaluates the Prologue Action and then the conditions on the outgoing arrows in a specified order.
- The process instance proceeds along the first single outgoing arrow whose condition is satisfied by the value of the specified User Defined Attribute. If the value of the User Defined Attribute satisfies none of the conditions the process instance proceeds along the outgoing arrow specified as the default.

Note: You can define the outgoing arrows of a Simple Exclusive Gateway Node to evaluate conditions in specified order using Interstage BPM Studio.

Call Activity Node

A Call Activity Node represents a step in a process where a task is accomplished passing data to an external process, invoking a process there, and ultimately receiving the results back.

Call Activity Nodes facilitate the reuse of existing process definitions. A single Call Activity Node can represent an entire process definition that is already designed, accessing all of the nodes, arrows, and other characteristics of the process definition.

With Call Activity Nodes, a user can organize a complex process into a simpler hierarchical network of subprocesses that hide the details from the top-level view. The processes and subprocesses are independent and can be designed simultaneously.

Call Activity Nodes can also be used to delegate details to other people.

A process can have any number of Call Activity Nodes. A Call Activity Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Prologue Action Set, Epilogue Action Set, onAbort Action Set, onSuspend Action Set, and onResume Action Set can be used as required.

Unsupported: This node has no assignee, forms, timers or triggers.

Behavior:

- Attributes specific to this node type include the ID of the process definition to be used to create a subprocess instance and data-mapping references that define what data is moved between the parent process instance and subprocess instance.
- Input data mapping initializes the subprocess's User Defined Attributes with the values from the parent process's User Defined Attributes before the execution of the subprocess.
- Output data-mapping copies values from the subprocess's User Defined Attributes to the parent process's User Defined Attributes after the subprocess is completed.
- Data mapping need not be specified if no data will be copied into or out of the subprocess.
- On receiving an event from an incoming arrow, a Call Activity Node executes the Prologue Action Set and then creates a process instance using the process definition who's ID has been specified. Input data mapping, if specified, initializes the User Defined Attributes of the subprocess.
- The subprocess then starts, and the Call Activity Node waits for it to complete. While waiting, the node ignores any events received from incoming arrows.
- Once the subprocess completes, output data mapping, if specified, copies data from the subprocess to the parent process, executes the Epilogue Action Set, and sends events to each of its outgoing arrows.

Remote Sub-Process Node

A Remote Sub-Process Node represents a step in a process where a task is accomplished passing the execution to a process running on a remote workflow server. A remote workflow server might be, for example, another Interstage BPM Server. Like Call Activity Nodes, Remote Sub-Process Nodes can be used to reuse existing process definitions within new ones, simplify complex business processes, and delegate details to other people.

A process can have any number of Remote Sub-Process Nodes. A Remote Sub-Process Node can have one or more incoming arrows and one or more outgoing arrows. However, the names of the outgoing arrows must match with the names of the End Nodes in the remote subprocess.

Supported: Prologue Action Set, Epilogue Action Set. onAbort Action Set, onSuspend Action Set, and onResume Action Set can be used as required. The semantics of this node type also support the assignment of an Error Java Action that is activated when the starting of the remote subprocess fails.

Unsupported: This node has no assignee, forms, timers or triggers.

Behavior:

- Attributes specific to this node type include the URI of the process definition to be used to create a remote subprocess instance and data-mapping references that define what data is moved between the parent process instance and the remote subprocess instance.
- Input data mapping initializes the remote subprocess's User Defined Attributes with the values from the parent process's User Defined Attributes before the enactment of the remote subprocess.
- Output data-mapping copies values from the remote subprocess's User Defined Attributes to the parent process's User Defined Attributes after the remote subprocess is completed.
- Data mapping need not be specified if no data will be copied into or out of the remote subprocess.
- On receiving an event from an incoming arrow, a Remote Sub-Process Node evaluates the Prologue Action and then creates a process instance using the process definition whose URI has been specified. Input data mapping, if specified, initializes the User Defined Attributes of the remote subprocess.
- The remote subprocess then starts, and the Remote Sub-Process Node waits for it to complete. While waiting, the node ignores any events received from incoming arrows.
- Once the remote subprocess completes, output data mapping, if specified, copies data from the remote subprocess to the parent process, evaluates the Epilogue Action, and sends events to each of its outgoing arrows.

Chained-Process Node

A Chained-Process Node represents a step in a process where a new process is created to accomplish a task independent of the parent process instance. This node enacts this independent process as a part of the execution of the parent process.

A process can have any number of Chained-Process Nodes. A Chained-Process Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Prologue Action Set, Epilogue Action Set, onAbort Action Set, onSuspend Action Set, and onResume Action Set can be used as required.

Unsupported: The semantics of this node type do not support assignee, forms, timers or triggers.

Behavior:

- Attributes specific to this node include the ID of the process definition that has to be used to create new process instance and data mapping that defines what data is to be copied to the new process instance.
- Input data mapping initializes the new process instance's User Defined Attributes with values from the User Defined Attributes of the parent process instance. A Chained-Process Node uses only input data mapping, since the flow of enactment does not return to the parent process instance. Data mapping need not be specified if no data will be copied to the new process instance.
- On receiving an event from an incoming arrow, a Chained-Process Node evaluates the Prologue Action and then creates a process instance using the process definition who's ID has been specified.
- Input data mapping, if specified, initializes the User Defined Attributes of the chained-process.
- The chained-process is independent of the parent process instance.

Embedded Sub-Process Node

An Embedded Sub-Process node encompasses a phase of a process and the milestones to be achieved at the end of the phase.

Inside Embedded Sub-Process node, there is one Start Node and one or more End Node, and any type of node except Embedded Sub-Process node can be included in it. Thus, each Embedded Sub-Process will represent a phase.

Supported:

On Embedded Sub-Process Node:

- Group/Expand Group
- Prologue, Epilogue, onAbort, onSuspend, onResume Actions Set
- Timers
- Due Date
- Forms
- Priority

On workitems created from Embedded Sub-Process Nodes:

- Make Choice
- Reassign
- Priority

Behavior:

- When the control reaches the Embedded Sub-Process node in a process, the child Start node is activated and the Embedded Sub-Process node state changes to `WaitingOnSubProcess`.
- Once the child nodes are complete and the child End node is reached, the Embedded Sub-Process node is closed and the outgoing arrow from the Embedded Sub-Process node that has the same name as the child End node, is activated.
- Milestone date can be defined on the Embedded Sub-Process by adding DueDate Timer.
- There might be a requirement that Phase needs to be forcefully completed. To achieve this, assignee can make choice on Workitems of Embedded Sub-Process Node and further Process execution can continue. This way any active child nodes will be forcefully aborted .

Refer to *Using Embedded Sub-Process Nodes* on page 184 for more information on how to create and use Embedded Sub-Process Nodes.

3.2.11 Modifying Process Definitions

Process definition operations pertain to how process definitions can be modified. To modify any process definition, the following prerequisites must be fulfilled:

- The process definition must be in Draft state and in edit-mode.
- There must not be an instance of the process definition.
- You must be process definition owner or an administrator.
- You must acquire a lock on the process definition from the Interstage BPM Server.

You can only commit the changes to the process definitions atomically; i.e. either all or none can be made. If you cancel or roll back the edit, no changes are made.

3.3 Process Instances

A process definition is a general model of a business process, using workflow elements to describe it. A process instance describes and organizes one specific execution of this process definition. A running process instance mimics the flow of events in the business process from its inception to its completion.

Once a process instance is created, the Interstage BPM Server (also referred to as the workflow engine) executes the process instance according to the design of the process definition. This execution of the process instance by the Interstage BPM Server is referred to as enactment.

Process instances are enacted in the Interstage BPM Server using the event-model mechanism. Events are like messages that are sent to the workflow elements in a process instance as the Interstage BPM Server enacts it. These events trigger the workflow elements into action. Nodes and arrows react to these events and perform tasks according to their type and definition.

Note: Arrows merely carry an event from its source to its target node.

Process instances have one Start Node and one or more End Nodes. Enactment of a process instance begins when the Start Node receives an event. Nodes react differently to the events, depending on their type and definition. Subsequent events are propagated from node to node in the process instance.

A process instance can have one or more branches of control flow, depending upon the number of paths that it has from its Start Node to its End Nodes. However, the process instance completes when one of the End Nodes receives an event. When this happens, all active nodes are canceled.

Process Instance Attributes

The structure of a process instance is exactly the same as the structure of the process definition on which it is based. Every object and every attribute within the process definition is mirrored by a corresponding process instance or attribute. In addition, process instance have additional "process-relevant data" beyond the process definition.

A process instance can have the same attributes as the originating process definition with certain exceptions. For instance, there is no relationship between the ID of a process definition and the ID of the process instance that is created from it. In addition, a process instance can have the same name, title, and description as the process definition on which it is based, or it can have its own identifiers.

Finally, process instances have a number of additional attributes that are not shared with (or mirrored from) process definitions in any way. These include attachments, User Defined Attributes, and attributes that are meaningless prior to run-time (e.g. process priority).

A process instance attribute could be a persistent data element or a link to an external information source such as a document. Process instance attributes include (but are not necessarily limited to) the following:

- Process instance state
- Process instance identifiers
  - Process ID
  - Process instance name
  - Process instance description
  - Process instance title
- Flow control
  - Node instances
  - Arrow instances
  - Java Action Set
- Process instance special users
  - Process Owner

    - Process initiator
- Variable data
    - Data item definitions
    - Creation time
    - Priority
    - Set of attachment references
- Timers including a Java Action Set

3.3.1 Process Instance States

A process instance can be in one of the following states that are stored on the Interstage BPM Server:

- Initial
- Running
- Suspended
- Completed

When a process instance is created, it is in the initial state until it is started. When the process instance reaches the End Node, it enters the completed state.

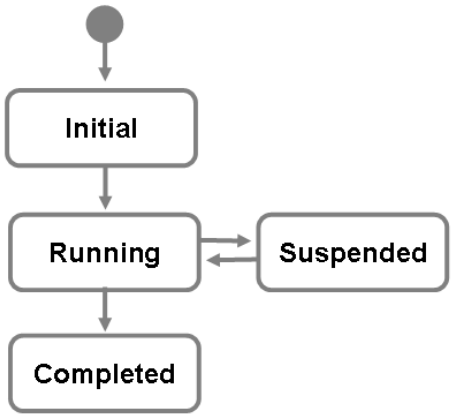

Figure 4: Process Instance State Transition Diagram

In practice, most process instances may be created and started at the same time, though it is possible to create a process instance without starting it. However, creating and starting process instances are two separate actions.

Suspending a process instance temporarily removes it from the running state. This temporarily inactive state is called the suspended state. If this process instance has any work items or running subprocesses, they are also put in the suspended state. Also, a suspended process instance cannot be modified.

A suspended process instance is, however, not in a completed state. You cannot modify the process definition from which a suspended process instance was created, you cannot reassign a suspended process instance, and resuming a suspended process instance activates it (puts it back in running state) along with any work items or subprocesses it has.

You can suspend or resume a process instance using the Model API. You must be an administrator to use the suspend/resume methods in the API. Refer to section *Suspending Process Instances* on page 233 for using the `suspend()` and `resume()` methods.

Persistence and State

The chief criteria for whether a state is persistent (i.e. stored on the Interstage BPM Server) is whether making this state persistent will drive the process instance towards completion.

The Interstage BPM Server does not guarantee persistence and might detract from the forward movement of the process instance (such as sessions and session data). For instance, a process instance can be in a locked or unlocked state. This state is volatile and is lost if the Interstage BPM Server goes down while the process instance is running, because in a situation where the Interstage BPM Server machine fails, a persistently locked state could interfere with recovery.

Interstage BPM supports ACID properties (atomic, consistent, isolated, and durable) for workflow process transactions, which do not include direct coordination across application participants. For mission critical applications, it is recommended that workflow processes be kept separate from application processes, and synchronized by the application based on the workflow steps. If the Interstage BPM Server is unable to complete a workflow process update and has to back it out, the application also can back out its processing to the same point. This coordination can be achieved by using the Model API. In this context, changes are visible to other transactions.

3.3.2 Node Instances and Arrow Instances

Node instances and arrow instances have a one-to-one correspondence with process definition nodes and arrows. Node instances are basically state holders for the specific nodes that they represent. Node instances and arrow instances can be in the following states:

- Initial
- Running
- Completed

Refer to section *Node Types* on page 34 for information about the input behavior for the different node types.

3.3.3 Process Instance Ownership

A process instance is owned by the person who begins the process instance (Process Initiator). It is not necessary for the process owner to be involved in creating a process instance; however, it can be useful for process owners to know who created each specific process instance that belongs to them.

3.3.4 Attachments

Attachments are references to documents that are created and accessed through other software. They could be images, spreadsheets, word processor files, or any other file that might need to be accessed or modified in accomplishing a business process.

Attachments consist of only references to the document's paths and file names.

Attachments are added, removed, and accessed through a work item. Attachments belong to the entire process instance which means that they are shared with User Task Nodes on other branches of the workflow. Attachments can be easily accessed and displayed by user, and even updated when that is allowed. By default, the work item allows the user to specify which programs to use to open the different file types.

3.4 Work Items

Once a User Task Node has finished a evaluating task assignment, it generates separate work items for each assigned user or user group. Work items are the users' window on the process instance; they are the mechanism by which the user can access all of the attributes of the process instance.

Work Item Attributes

- Identifiers
  - Process instance ID
  - Process instance name
  - Process instance description
  - Process instance title
  - Work item name
  - Work item title
  - Work item description
  - Set of choices
- Assignee
- Variable Data
  - Set of attachment references
  - Set of forms
  - Data items

3.4.1 Work Item Modes

When a User Task Node becomes active, work items are generated according to the associated assignee expression and possibly Role Actions.

The number of generated work items depends on the whether the User Task node is set for **individual work items** or **group work items**.

Group Work Item Mode

In this mode, one work item is generated for the entire user group assigned to a User Task Node. Group work items are useful in case of frequent changes to the members of a group and you want all current members of the group to always have access to workitems even when the work item has been created some time before the last group change.

Note: This mode is not applicable for work items associated with Voting User Task Nodes, and when a Role Action is used. If you try setting this for Voting User Task nodes or when a Role Action is used, a group work item is not generated; instead, work items for each user in the group are generated.

Individual Work Item Mode

In this mode, one work item is generated for every user in the User Group assigned to a User Task Node. Every user works on his/her own work item. Individual work items are required if you want to keep track of users who decline the task, and for activities that can be forwarded to other individual.

Changing the Work Item Mode

The type of workitems generated for a User Task Node is controlled by the `Activity_node.markForExpandGroups` parameter. The value `true` indicates that individual workitems will be created, and `false` indicates that group level workitems will be created. By default, individual work items are generated.

Additionally, there is a `SupportGroupWorkItem` parameter of the Interstage BPM Server which effects process definitions which have no setting on the nodes. This can only happen if the process definitions was created by a very old version of the product. When a process definition that does not specify markForExpandGroups in the User Task Nodes, then that server will set the flag for each node to the converse of this global parameter at the time that the process definition is installed into the server. Specifically, when a very old process definition is imported into the system, if SupportGroupWorkItem=true, then markForExpandGroups will be set to false, and vice versa. Refer to section *Designing a Process Definition with Start Node, User Task Node and End Node* on page 66 for a sample.

3.4.2 Work Item States

When a User Task Node becomes active, Interstage BPM creates one or more work items to the members of the specified user group. By default, individual work items are generated and assigned to every member of a user group; you can change this behavior and have Interstage BPM generate one group work item for the entire group assigned to the respective User Task Node. Two or more people can avoid duplicating each other's work by accepting the task thereby reserving the task to themselves.

In general, work items can be in the following states: active, read, accepted, declined, inactive (individual work items only), suspended, and completed.

- **Active**: Whenever a User Task Node becomes active, Interstage BPM starts by finding the list of user associated with the user group assigned to the User Task Node. Then this list of users can be manipulated by Java Actions in the Role Action Set adding or excluding users from the list. Then workitems are created for each of the remaining users in the list. Active work items allows the users to view, accept, suspend, reassign, or decline.
- **Read**: When a work item has been viewed by a user it is placed into a "Read" state. A work item in "Read" state can later be accepted, suspended, reassigned, or declined, just like a work item in the "Active" state.
- **Accepted**: When a work item has been accepted by a user it is placed into an "Accepted" state. At the same time other items associated with that node are placed into the "Inactive" state. Because only one work item can be in the "Accepted" state, this can be used to prevent two or more users duplicating each other's work.

  The user who has accepted a work item may later choose to decline it. When this happens, the other work items which had been "Inactive" are returned to the "Active" state. In addition, in Regular Reassignment Mode, a person who has accepted a work item may assign it to someone else.

  Refer to section *Reassignment Modes* on page 51 for information about the possible modes.
- **Declined**: A work item in the "Declined" state indicates that a user had declined the task to tell others they will not be doing the task. A declined work item no longer appears on a user’s list of active work items, though it still appears on the list of inactive work items. It is still available for the user to view or accept. The behavior of the system is different depending upon whether the node is set of 'individual' or 'group' work items.

If all of the assignees of a node decline their **individual** work items, new active work items are created for each of the process instance owners. If all process instance owners decline their work items, new active work items are created for them again.

If everyone declines a **group** work item, no work item is created for any process instance owner. The existing group members can work on the group work item after accepting it; any new member added to the group can work on it directly.

A user can accept a previously declined work item, so long as someone else has not already accepted it.

- **Inactive**: A work item becomes inactive for a user when someone else has accepted it.

  An inactive work item cannot be declined or accepted, though it can become active at a later time, when it can again be accepted or declined

- **Completed**: When the user makes an appropriate choice on the work item, it enters the Completed state. For most nodes, this completes the entire node, and all workitems are removed disappear from the lists of work items.

3.4.3 Choices

When a user has completed the tasks specified by the work item, he or she may choose one or more commit options to complete the activity and activate the next node in the process instance.

3.4.4 Future Work Items

Future Work Items are the work items that the users may be assigned in future. As a user, Future Work Items help you to be aware of work items that may be assigned to you later so you can plan your tasks in advance.

Future Work Items predict the most probably future path for the process but as the situation changes the actual path can vary from the prediction. Using this kind of prediction together with analytical history of all processes can yeild a powerful way to help users drive to the most optimal outcomes for their given process instance.

Note: Future Work Items are generated just as normal work items, with state as `WorkItem.STATE_FUTURE`. They are displayed as a list for a particular user. User cannot actually accept them or reject them or perform any kind of actions on them since these are displayed as a read-only list.

Following are the properties of Future Work Items:

- Future Work Items are generated only on the User Task nodes that have the `Activity_node.markForFutureWorkItems` flag set to `true`. By default, this flag is set to 'false'.

Note: Future Work Items can also be generated for User Task nodes in Embedded Sub-Process Node.

- Future Work Items are generated only on inactive (initial state) User Task node instances of a process instance.
- Future Work Items are generated in a process instance only when at least one User Task node instance is activated.
- When a User Task node instance is activated, it generates work items for current node, deletes Future Work Items for current node and generates Future Work Items for other User Task node instances that are in initial state.

- Future Work Items are generated using Assignee and Role Java Actions.

Note: Any process update done in Role Actions while evaluating assignees of Future Work Items is committed. Future Work Items are generated for a User Task node whenever any other User Task node instance is activated in the process instance. Hence, these Role Actions will be executed multiple times. Also, they will be executed again while generating normal work items. You need to make sure that execution of these actions multiple times, does not result in undesired behavior.

- When the Role Action Set of a User Task Node is not empty, Future Work Items are assigned to the users using the intersection of the group members and users specified by the Java Actions in the Role Action Set. If this intersection does not contain any value, Future Work Items are not generated for process owners (normal Work Items are generated for the process owners in such a case).
- Future Work Items are generated or deleted during the process enactment. This means if a User Task node instance A is in active state and waiting for the user to make choice and you set the Future Work Item flag of any other User Task node instance in the initial state (say, B), to 'true' then B will not generate Future Work Items until you make choice on A.
- Future Work Items cannot be generated for the User Task nodes that have Agents defined on them.
- Future Work Items cannot be generated for the User Task nodes configured as Iterator Nodes.
- Future Work Items cannot be generated for Voting User Task Nodes.
- Future Work Items cannot be generated for the User Task nodes having group-level work items.
- Future Work Items generated for a User Task node in 'initial' state are deleted once the state of the User Task node changes to 'active'.
- Any failure while generating Future Work Items is handled similar to the normal failure, that is, transaction is rolled back and exception is thrown to user or process instance is put into error state depending on inline/background enactment. Also, compensate and error actions are executed in normal fashion.

3.5 Filters

Filters in Interstage BPM operate similarly to a SQL SELECT statement: they allow you to access columns in a table in a database according to specific criteria. In other words, you can create a list from the table of process definitions, process instance, or work items; you can filter them by assignee, initiator or owner, by priority, by state, or by identifier.

Note: Advanced filtering and sorting capabilities are available through the Model API. Refer to section *Advanced Filtering & Sorting API* on page 111 for instructions in using these advanced capabilities.

3.6 The Purpose of Structural Process Editing

Process editing is the key feature required for several different types of workflow which require fewer constraints on the direction a process instance can take:

- Ad Hoc Workflow
- Process Discovery
- Collaborative Workflow

- Case Handling

Workflow users need structural process editing for the following purposes:

- **Design error**: After a process instance has been running for a period of time, users might discover that it is based on faulty assumptions or logic.
- **Maintenance**: Most long-running process instances will need to be tweaked periodically to keep them up-to-date and running smoothly.
- **Process discovery**: In the real world, many business processes are started without completely solidifying the business plan. Structural process editing allows a project team to fly by the seat of their pants, modifying the process definition from the feedback they receive, and recording their process instance for later replication.
- **Obtaining Experimental Results**: Structural process editing allows you to try out a new procedure along the way and later analyze the results of using two or more methods that are precisely recorded.
- **Adjusting to New Requirements**: Structural process editing enables you to modify your process definition whenever new government regulations or new market conditions occur.
- **Starting Over**: When a process is proceeding according to plan, but no one is happy with the results, structural process editing allows you to start it over with modifications.

3.7 The Purpose of Interstage BPM's Subprocess Capabilities

Subprocesses are an important element because of the following:

- Subprocesses enable users to break a project or process into easy to handle units of collaboration. In other words, a complex process definition might be made up of an extremely large number of nodes, making the process definition unreadable. If you reconstruct this process instance into a hierarchy of simple subprocesses, each level of the process definition can be designed as a separate task that fits easily into the whole process definition.
- Subprocesses also enable different departments with different processes to linking those processes together easily and appropriately. For instance, a purchasing department, an accounting department, and a new product development department will need completely separate process definitions to serve their individual internal needs. However, Interstage BPM's subprocesses enable them to create an interdepartmental process definition to trade information and coordinate collaborative tasks among all three seamlessly.
- Subprocesses enable an individual, a department, or a company to create a library of modular components, process definitions, or tasks that they can later paste together in a wide variety of ways.

Users can create subprocesses in two different ways: formally at design time and on the fly at run time. These two different implementation strategies involve slightly different structures in the APIs.

- **Design-time subprocess implementation**, for instance, requires the use of a Call Activity Node, a node type that is expressly constructed for this purpose. This node provides a uniform interface that users can plug into a process definition at design time, with properties that specify the subprocess definition and the method for mapping data that will be passed back and forth between the parent process instance and child process instance. This Call Activity Node becomes a member of a process definition in the same way that any other node is a member of a specific process definition.

  When the flow of an active process instance reaches a Call Activity Node, the node immediately instantiates the subprocess and then starts it running. While waiting for the subprocess to complete, the Call Activity Node enters a suspended state.

- **Run-time subprocess implementation** uses methods that are part of the User Task Node class. When a task is assigned to an individual who must delegate the task (or parts of it) to others, the person can invoke a run-time subprocess from the User Task Node. The delegating person can create a new process definition or use an existing one and can map data from the original process instance to the subprocess - just as if it had been done at design time. The original assigned work item enters into 'waiting for subprocess' state as soon as a subprocess is invoked. The node changes to a completed state, when the subprocess is complete and passes the workflow back to the work item.

The difference between run-time and design-time subprocess implementation is that run-time subprocess implementation is ephemeral and leaves the original process definition in its original state. The next time that the process definition is used to create a process, the original User Task Node is still only a User Task Node.

However, when subprocesses are used with process editing, it enables a powerful feature: **enterprise-wide process discovery workflow**. This enables an executive, for instance, to create a general statement of the work of the company, and use run-time subprocesses to get input from those below him in the hierarchy. As they develop and distribute their own subprocesses, a procedural description of the work of the entire business can evolve. Thus, enterprise-wide process discovery workflow provides a gateway into reengineering the entire business operation through the use of workflow.

3.8 Security and Reassignment Modes

The Security and Reassignment Modes address slightly different issues than the work item states. However, since all of these modes and states work together to control work item access, they must be addressed together. By default, the system is setup in Open Security mode and Regular Reassignment mode.

3.8.1 Security Modes

The Security mode can be modified by setting the `SecuritySwitch` parameter of the Interstage BPM Server. Refer to the *Interstage Business Process Manager Server Administration Guide* for more information on how to change the configuration of the Interstage BPM Server.

The major difference in work item access between Open Security Mode and Secure Mode is that in Open Security Mode, any Interstage BPM user can view a work item (read-only). In Secure Mode, only the work item assignees and the process owners can view the work item and modify process details or make a choice on the work item.

3.8.2 Reassignment Modes

The Reassignment mode can be modified by setting the `ServerReassignMode` parameter of the Interstage BPM Server. Refer to the *Interstage Business Process Manager Server Administration Guide* for more information on how to change the configuration of the Interstage BPM Server.

Reassignment allows activities to be assigned to a different set of people than those originally specified through the User Group. The access control created for reassignment operates differently than access control that is implemented by work item state. Reassignment acts on the activity, not on the work item.

Reassigning a work item causes to delete all active and inactive work items associated with the activity. Then the server creates new, active work items for the people specified in the reassignment.

Activities must be reassigned to individuals, not to Groups, because assignment is mainly appropriate for process definitions that will be reused for a period of time. Because of this, it does not verify that the people specified in the reassignment are valid users.

Interstage BPM can operate in three different reassignment modes: Regular Mode, Process-Owner-Only Mode or No-Reassignment Mode. These are configured by setting the `ServerReassignMode` parameter to regular, owner or none.

- **Regular Mode**: In this mode anyone who is a current assignee for the activity or a process owner can reassign the activity to a new set of users.
- **Process-Owner-Only Mode**: In this mode only a process owner can reassign an activity to a different set of users. The process owner doesn't have to be an assignee of a work item in order reassign the work item to someone else.
- **No-Reassignment Mode**: In this mode, reassignment is completely disabled, and no one can reassign an activity.

4 Using the Model API

This chapter describes the system environment for application development using the Model API and introduces the basic Model API architecture.

4.1 System Environment

You can develop and deploy applications using the Model API on all of the operating system platforms supported by Interstage BPM. Refer to the *Release Notes* for a list of supported operating systems.

On the system where you want to **develop** or **deploy** your applications using the Model API, the following is required:

- JDK of the version specified in Release Notes, is configured correctly.
- `iFlow.jar`. You must add its path to the `CLASSPATH` environment variable. In a typical setup, you can find this file at the following location:

  On Windows:

  `<engine directory>\client\lib\iFlow.jar`

  Here, `<engine directory>` is considered as `C:\fujitsu\InterstageBPM`

  On UNIX:

  `<engine directory>/client/lib/iFlow.jar`

  Here, `<engine directory>` is considered as `/opt/FJSVibpm`

Note: The `iFlow.jar` file contains methods that are used internally by Interstage BPM. These methods are not supported for application development and are not described in the Javadoc. For this reason, an alternative jar file is provided for development purposes: `iFlow_api.jar`, which does not contain any internally used methods. It is recommended that you compile your application with the `iFlow_api.jar` library so that you make sure that your application does not use any internal methods.

All other libraries (except `iFlow_api.jar`) contained in the `C:\Fujitsu\InterstageBPM\client\lib` or `/opt/FJSVibpm/client/lib` directory must also be added to the CLASSPATH, since they are used by the Model API implementation. Currently, there are two libraries in the `client/lib` directory that are needed at runtime:

- `js.jar`
- `log4j-1.2.15.jar`

- Make sure that you add and define the path to the required application-server specific `jar` files to the `CLASSPATH`.

4.1.1 Specifying Configuration Settings for WebLogic

Prerequisite: Set up your system environment as explained in section *System Environment* on page 53.

- If you want to develop applications using the Model API on the WebLogic Application Server, add the following JAR files to the `CLASSPATH` environment variable:

  a) **For WebLogic 10.3.2 and WebLogic 12.1.1:**

  `<MW_HOME>/wlserver_<version>/server/lib/wlclient.jar`

  `<MW_HOME>/wlserver_<version>/server/lib/wljmsclient.jar`

b) **For WebLogic 12.1.3 and above:**

`<MW_HOME>/wlserver/server/lib/wlclient.jar`

`<MW_HOME>/wlserver/server/lib/wljmsclient.jar`

4.1.2 Specifying Configuration Settings for JBoss(Local)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 53.

To develop applications using the Model API on JBoss EAP 6 Application Server, add the following files to the `CLASSPATH` environment variable:

- `jboss-client.jar` file

  `<JBoss Installation Directory>/bin/client/jboss-client.jar`

- Directory where`jboss-ejb-client.properties` exists

  `<engine directory>/client/`

4.1.3 Specifying Configuration Settings for JBoss for deploying Client J2EE Application (Local)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 53.

You can use Model API in your J2EE application on the same machine where the Interstage BPM Server is setup.

To deploy client J2EE applications for JBoss EAP 6

1. Add the `<engine directory>/client/lib/iFlow.jar` to the `CLASSPATH` of client J2EE application.
2. Create the structure of `jboss-deployment-structure.xml` file is as mentioned below and add it to the `CLASSPATH` of client J2EE application:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
 <deployment>
  <dependencies>
   <module name="org.jboss.remote-naming" />
   <module name="org.hornetq" />
  </dependencies>
 </deployment>
</jboss-deployment-structure>
```

4.1.4 Specifying Configuration Settings for JBoss(Remote)

Pre-requisite: Set up your system environment as explained in section *System Environment* on page 53.

To develop applications on a remote computer using the Model API on JBoss EAP 6 Application Server,

1. Copy the following files to the remote machine into a directory of your choice, for example, in this guide we are using `<IBPM JAR>`
   - `jboss-client.jar` file from `<JBoss Installation Directory>/bin/client/jboss-client.jar`
   - `jboss-ejb-client.properties` from `<engine directory>/client/`
2. Add the following path to the `CLASSPATH` environment variable:
   - `jboss-client.jar` file

     `<IBPM JAR>/jboss-client.jar`
   - Directory where`jboss-ejb-client.properties` exists

     `<IBPM JAR>/`

4.1.5 Specifying Configuration Settings for JBoss for deploying Client J2EE Application (Remote)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 53.

You can use Model API in your J2EE application on the remote computer (that is on a computer different from the one hosting the JBoss EAP 6):

Follow these steps:

1. Create an `outbound-socket-binding` on the "Client Server (remote)", as mentioned below:
   a) **In case of standalone server setup:** For connecting to an Interstage BPM server, update the `domain.xml` with the following `outbound-socket-binding` in the `socket-binding-group` section.

   For example, if your client J2EE application is using the profile `<profile name="full-ha">` in the file, then update the `socket-binding-group name="full-ha-sockets"` as shown below:

```
<socket-binding-group name="full-ha-sockets" default-interface="public"
 port-offset="${jboss.socket.binding.port-offset:0}">
        ...
      <outbound-socket-binding name="remote-ejb">
          <remote-destination host="HOST_IP" port="4447"/>
      </outbound-socket-binding>
</socket-binding-group>
```

   Replace `HOST_IP` with the IP address of the machine on which Interstage BPM server is running.

   b) **In case of cluster setup:** Update the `domain.xml` to specify the `outbound-socket-binding` setting for each of the cluster nodes, in the `socket-binding-group` section, as shown below.

   For example, if your client J2EE application is using the profile `<profile name="full-ha">`, then update the `socket-binding-group name="full-ha-sockets"` as shown below:

For example, if there are two nodes in the cluster setup, then this file will be updated as shown below. You can add more `outbound-socket-binding` settings depending on the number of nodes in your cluster setup.

```
<socket-binding-group name="full-ha-sockets" default-interface="public"
 port-offset="${jboss.socket.binding.port-offset:0}">
        ...
      <outbound-socket-binding name="remote-ejb1">
          <remote-destination host="HOST_IP_NODE1" port="4447"/>
      </outbound-socket-binding>
      <outbound-socket-binding name="remote-ejb2">
          <remote-destination host="HOST_IP_NODE2" port="4447"/>
      </outbound-socket-binding>
</socket-binding-group>
```

Replace `HOST_IP_NODE1` and `HOST_IP_NODE2` with the IP address of the machine that you have included in the Interstage BPM server cluster setup.

2. Create a "`remote-outbound-connection`", which uses the "`outbound-socket-binding`" that you have just created on the client server:
   a) **In case of standalone server setup:** update the domain.xml for "`remote-outbound-connection`" in the profile `<profile name="full-ha">` or `<profile name="full">` etc. that your client J2EE application uses, as below.

```
<subsystem xmlns="urn:jboss:domain:remoting:[version]">
  <connector name="remoting-connector" socket-binding="remoting"
security-realm="ApplicationRealm"/>
  <outbound-connections>
    <remote-outbound-connection name="remote-ejb-connection"
outbound-socket-binding-ref="remote-ejb"/>
  </outbound-connections>
</subsystem>
```

   b) **In case of cluster setup:** Update the `domain.xml` to specify the "`remote-outbound-connection`" setting for each of the cluster nodes, as shown below:

   For example, if there are two nodes in the cluster setup, then this file will be updated as shown below. You can add more `remote-outbound-connection` settings depending on the number of nodes in your cluster setup.

```
<subsystem xmlns="urn:jboss:domain:remoting:[version]">
  <connector name="remoting-connector" socket-binding="remoting"
security-realm="ApplicationRealm"/>
  <outbound-connections>
    <remote-outbound-connection name="remote-ejb-connection1"
outbound-socket-binding-ref="remote-ejb1"/>
    <remote-outbound-connection name="remote-ejb-connection2"
outbound-socket-binding-ref="remote-ejb2"/>
  </outbound-connections>
</subsystem>
```

3. Create `jboss-ejb-client.xml` and add it in the `CLASSPATH` of client J2EE application:
   a) **In case of standalone server setup:** create the structure of `jboss-ejb-client.xml` file as shown below:
      - Use remote outbound connection ("`remote-ejb-connection`") created in step 2 to specify in `<remoting-ejb-receiver>`.

- Make sure that the following properties are set to false:

  ```
  <property name="org.xnio.Options.SSL_ENABLED" value="false" />
  <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS" value="false"
  />
  ```

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:[version]">
  <client-context>
    <ejb-receivers>
      <remoting-ejb-receiver
outbound-connection-ref="remote-ejb-connection"/>
    </ejb-receivers>
    <clusters>
      <cluster name="ejb">
        <connection-creation-options>
         <property name="org.xnio.Options.SSL_ENABLED" value="false"
 />
          <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS"
value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
  </client-context>
</jboss-ejb-client>
```

b) **In case of cluster server setup:** create the structure of `jboss-ejb-client.xml` file as shown below:

- Use remote outbound connection ("`remote-ejb-connection1`" and "`remote-ejb-connection2`") created in step 2 to specify in `<remoting-ejb-receiver>`.

  For example, if there are two nodes in the cluster setup, then this file will be updated as shown below. You can add more `remoting-ejb-receiveroutbound-connection` settings depending on the number of nodes in your cluster setup.

- Make sure that the following properties are set to false:

  ```
  <property name="org.xnio.Options.SSL_ENABLED" value="false" />
  <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS" value="false"
  />
  ```

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:[version]">
  <client-context>
    <ejb-receivers>
      <remoting-ejb-receiver
outbound-connection-ref="remote-ejb-connection1"/>
      <remoting-ejb-receiver
outbound-connection-ref="remote-ejb-connection2"/>
    </ejb-receivers>
    <clusters>
      <cluster name="ejb">
        <connection-creation-options>
         <property name="org.xnio.Options.SSL_ENABLED" value="false"
 />
          <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS"
value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
```

```
  </client-context>
</jboss-ejb-client>
```

4. Create jboss-deployment-structure.xml as shown below and add it in CLASSPATH of client J2EE application.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.jboss.remote-naming" />
      <module name="org.hornetq" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

5. Add following jar file in the CLASSPATH of client J2EE application:

   `<engine directory>/client/lib/iFlow.jar`

6. Start Interstage BPM server and then client server. The Client J2EE application is ready for use.

4.2 Executing a Model API Application

Prerequisites:

- RMI access is possible to the computer where Interstage BPM is setup.
- The Interstage BPM Server has been started.

To execute your Model API application:

- In your `<engine directory>`, you find a script for starting the programming samples provided with Interstage BPM:

  On Windows `<engine directory>\client\samples\examples\bin\StartSamples.bat`

  On UNIX `<engine directory>/client/samples/examples/bin/StartSamples.sh`

- You need to ALWAYS execute the command specified in the "Start the Sample" section of this script BEFORE you can execute a client application. This script requires you to enter the Java class name of your application.

  For details on how to use the samples, refer to Appendix *Using the Interstage Business Process Manager Samples* on page 261.

4.3 Location of Properties Files

You can access properties files from within your Model API application. The following code fragment shows an example of how to load the `iFlowClient.properties` file:

```
Properties iflowProps = new Properties();
FileInputStream fin = null;
try {
   fin = new FileInputStream("iFlowClient.properties");
   iflowProps.load(fin);
}
finally {
   if (fin != null) {
     fin.close();
```

```
    }
}
```

As a default, properties files are loaded from the current runtime directory. You need to store your properties files there. The sample `iFlowClient.properties` file is located in the following directory:

`<engine directory>/client`

4.4 Model API Architecture

The Model API allows you to access and manipulate Interstage BPM, including its administration function. With the Model API you can build your own applications which you can integrate in your own graphical user interface.

The Model API contains, among others, the following packages. For detailed information, refer to the API Javadoc.

- **`com.fujitsu.iflow.model.event`**: Contains interfaces and classes that listen for changes in the Interstage BPM objects to provide proactive notification to the user.
- **`com.fujitsu.iflow.model.util`**: Contains low level utility classes that are commonly used by other classes and interfaces. It also contains Exceptions thrown by the Model classes themselves.
- **`com.fujitsu.iflow.model.wfadapter`**: Contains interfaces that manage the Document Management System (DMS). This includes retrieving and updating information about folders and attachments in the DMS, checking objects in or out from the DMS.
- **`com.fujitsu.iflow.model.workflow`**: Contains interfaces that manage information required by the workflow process definitions and process instances. This includes objects that represent nodes, arrows, forms, attachments, work items and permission levels. The programming examples explained in this manual, above all, use the following interfaces:
  - **`Arrow`** interface: Used for creating and manipulating arrows.
  - **`ArrowInstance`** interface: Used to access arrow attributes within a process instance.
  - **`AttachmentRef`** interface: Used to access attachments to a process instance in the DMS. There is no limit to the number of attachments that can be associated with a process instance, and there are no restrictions on the types of attachments that can be added.
  - **`DataItemRef`** interface: Used to hold the name, type, and initial value of User Defined Attributes (UDAs) defined in a process definition.
  - **`JavaActionSet`** interface: Provides methods for placing Java Action into a process definition.
  - **`Node`** interface: Defines all the functions that can be used to obtain information about any node type within a process definition.
  - **`Plan`** interface: Used for creating and manipulating process definitions.
  - **`ProcessInstance`** interface: Provides operations for creating and initiating a process instance.
  - **`WFAdminSession`** interface: Provides methods used by an administrator. Extends the `WFSession` interface.
  - **`WFDetailsList`** interface: Allows for retrieving information about multiple objects at a time.
  - **`WFObjectList`** interface: Allows for advanced sorting and filtering of process definitions, process instances or work items.
  - **`WFObjectSearch`** interface: Allows for searching a process instance based on a specified search text string.

- **`WFSession`** interface: Provides methods for establishing and maintaining access to the Interstage BPM Server for the entire time a user is logged into it.
- **`WorkItem`** interface: Provides access to all aspects of a work item, that is, an activity that is assigned to a particular user.

- **`com.fujitsu.iflow.server.intf`**: Contains an interface that provides access to workflow data. This interface is called ServerEnactmentContext. This interface also contains classes for implementing action agents.

4.5 Exception Handling

There are two levels of exceptions thrown by the Model API. The `ModelException` class is the super class. Use this class for your exception handling.

The `ModelException` class contains some sub classes. For reasons concerning the future compatibility, use the `ModelException` class only.

4.6 Model-side Notifications Used by Model API

The following two types of model-side notification listeners (JMS Listeners) are used by Model API to get notifications from the Interstage BPM Server:

- **Notification Listener:** Listens to notifications about addition, removal, or modification of workflow objects (process definition, process instance, and work items). These notifications are required for the functioning of `ProcessInstanceListener` and `WFObjectListListener` APIs.

  If you disable this listener, the Model API cannot use the `ProcessInstanceListener` and `WFObjectListListener` APIs which listen to the notifications about changes made to workflow objects.

- **SQNotification Listener:**

  Whenever process definitions or process instances are updated in the server, Model cache is updated with the latest versions of process definitions and process instances. This operation of keeping the process definitions/process instances up-to-date in the Model cache is performed by the sequence id notifications through SQNotification Listener.

  If you disable this listener, the process definitions and process instances in Model cache are not refreshed to store their latest versions as in the server. In this case, the Model cache will contain old versions of process definitions and process instances. Refer *Ensuring Latest Process Definitions and Process Instances in Model Cache* on page 61 for information about methods to ensure latest versions of process definitions/process instances in Model cache.

These notification listeners receive notifications from the server by subscribing to the JMS notification topics which are maintained by the server. The model creates the JMS topic subscriptions as part of the login process.

If you do not need these notifications, you can use options to disable them. It ensures that Model does not connect to the specified JMS topics to log into Interstage BPM Server. Note that there would be limitations to disabling the notifications as the functionality provided by these notifications would not work.

4.6.1 Disabling Model-side Notifications

You can disable model-side notifications at two levels: server-level and client-level.

Disabling model-side notifications at server-level:

You can disable notifications at server-level using Interstage BPM Server system configuration parameters `ProactiveNotificationEnabled` and `ProactiveSQNotificationEnabled`. You can disable Notifications and SQNotification respectively using these parameters. You need to set these parameters to `false` to disable the notifications.

After disabling notifications, Interstage BPM Server does not post any model notifications to the JMS topics. This ensures that the model-side JMS Listeners are not initialized. Thus, none of the clients receive any notifications.

Disabling model-side notifications at client-level:

You can also disable the notifications at client-level using a client-level configuration parameter `DisableNotification`. You need to set this parameter to `true` to disable the notifications. In this scenario, client does not connect to the JMS notification topics, even though the server continues posting notification messages in it.

Prior to login, `DisableNotification` parameter needs to be specified as a configuration parameter using the `WFSession.initForApplication(String[ ] args, Properties prop)` API.

The following table presents the relationship between the configuration parameters and model-side notifications:

DisableNotification	Proactive Notification Enabled	Proactive SQNotification Enabled	Notification	SQNotification
`true`	-	-	Disabled	Disabled
`false`	`true`	`false`	Enabled	Disabled
`false`	`true`	`true`	Enabled	Enabled
`false`	`false`	`false`	Disabled	Disabled
`false`	`false`	`true`	Disabled	Enabled

4.6.2 Ensuring Latest Process Definitions and Process Instances in Model Cache

When you disable SQNotification, there is a limitation that the Model cache would not be automatically updated with the latest process definitions and process instances that are present in Interstage BPM Server.

To ensure latest process definitions and process instances in Model cache, use the following APIs:

- `WFObjectFactory.getProcessInstance(processInstanceId, wfSession)`

  This API returns a process instance of the latest version and removes any old version of the process instance from the Model cache.

- `WFObjectFactory.getPlan(planId, wfSession)`

  This API returns a process definition of the latest version and removes any old version of the process definition from the Model cache.

- The `WFObjectList` APIs, which are used to fetch the list of workflow objects (process definition, process instance, and work items) from Interstage BPM Server ensure that the retrieved objects

represent the latest versions of the workflow objects present in Interstage BPM Server. The list of these APIs is as given below:

- `WFObjectList.getNextBatch(int batchSize)`
- `WFObjectList.getNextBatch(long startId, int howMany)`
- `WFObjectList.getPreviousBatch(long endId, int howMany)`

Note: If you disable the notifications using the configuration parameters, you will not be able to use the `ProcessInstanceListener` and `WFObjectListListener` APIs which use model notifications.

5 Designing Process Definitions

This chapter provides programming examples, using the Model API, for designing process definitions and starting process instances.

The examples include:

- Logging in/logging out a normal user
- Creating a process definition with a Start Node, User Task Node and End Node
- Starting a process instance
- Executing work items

5.1 Designing a Simple Process Definition

To get started you can build a simple process definition with a Start Node, a User Task Node and an End Node, using the Model API.

You can find the complete programming code of the examples presented in this section in the `SimplePlan.java` sample file.

Before creating a process definition, a user must be logged in, and a workflow application selected. Afterwards he/she can create a simple process definition, whose basic structure consists of one and only one Start Node, a User Task Node and at least one End Node. Logging in a user means to create a session, i.e. a `WFSession` object. This session is finished when a user is logged out again.

The following figure shows the necessary steps for designing a process definition using the Model API:

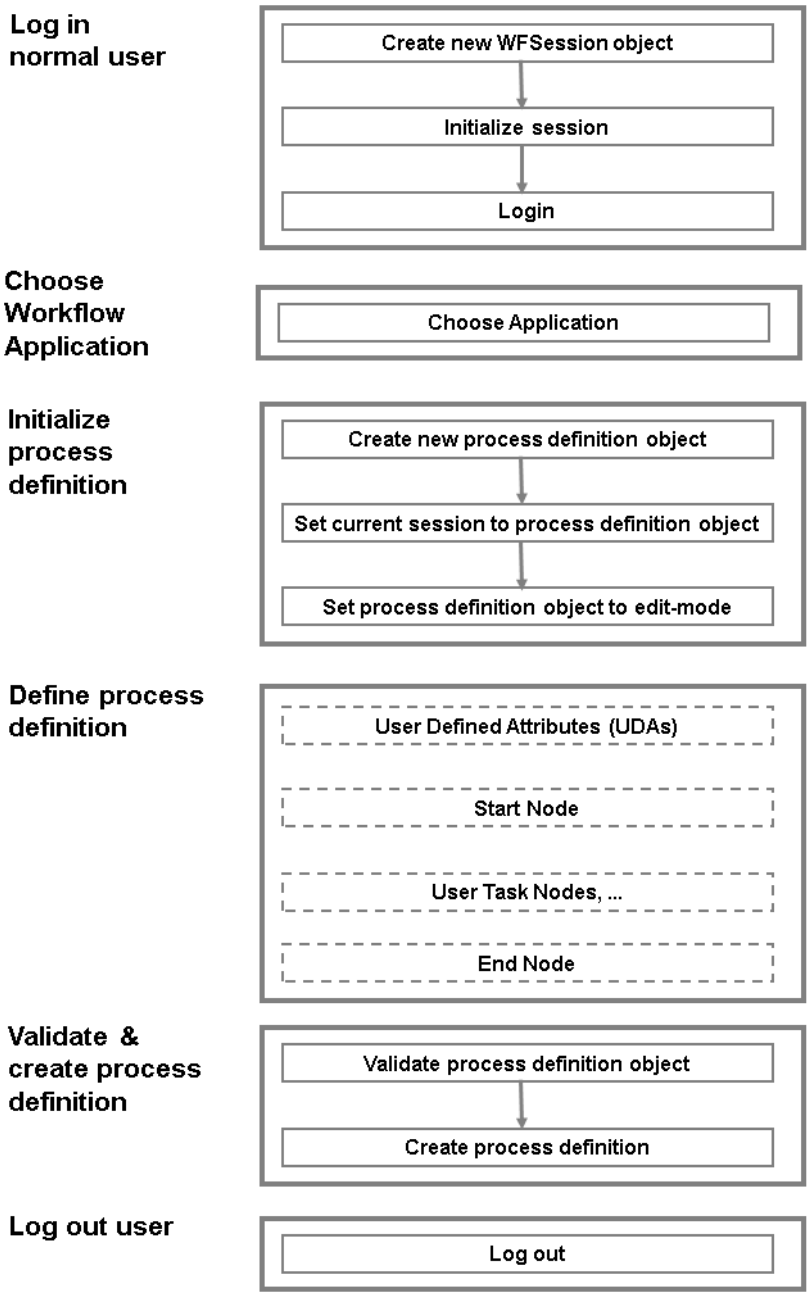

Figure 5: Designing a Process Definition Using the Model API

5.1.1 Logging in/Logging out a Normal User

Before working with process definitions, a tenant user must be logged in to the Interstage BPM Server. When working with process definitions is finished, he/she must log out again.

To log in/log out a tenant user:

1. Create a new `WFSession` object.

```
WFSession session;
session = WFObjectFactory.getWFSession();
```

   Use the `WFObjectFactory` class for accessing workflow objects. `getWFSession()` then creates a `WFSession` object.

2. Initialize the session with the appropriate configuration file.

   You can use the default `iFlowClient.properties` file located in `<engine directory>/client`.

   Either use this file or write the configuration parameters into a new file, which must be located in the current runtime directory. For the location of the current runtime directory, refer to section *Location of Properties Files* on page 58.

Note: Ensure you specify the tenant name logged in for the property file used. The tenant name is specified for value of `WFObjectFactory.TENANT_NAME(TenantName)`. `TenantName=Default` is specified for login to `Default` tenant.

Note: In the `iFlowClient.properties` file, any backslashes "\" or colons ":" are escaped by backslashes. For example, a server address is specified like this:

`ibpmhost\:49950`

When loading the `iFlowClient.properties` file using the `java.util.Properties.load()` method, escape characters will automatically be taken into account. If you use another way to load the properties, make sure that you handle any escape characters correctly. For details about escape sequences that may occur in the `iFlowClient.properties` file, refer to the Java documentation of the `java.util.Properties.store()` method.

   Load the configuration file `iFlowClient.properties` for the session:

```
Properties sessionProps = new Properties();
     sessionProps.load(new
FileInputStream("../classes/iFlowClient.properties"));
```

   Initialize the session:

```
session.initForApplication(null, sessionProps);
```

3. Log in a user.

```
session.logIn(server, userName, password);
```

   The parameter `server` is retained for compatibility purposes. When the user logs into Interstage BPM Server, this parameter is not evaluated. Its value should be set to a STRING but not null.

4. After work with the process definition is finished, the user has to log out from the `WFSession`. To log out a user:

```
if (session != null ) {
   session.logOut();
   }
```

5.1.2 Choosing a Workflow Application

After logging in, choose a workflow application within which you want to operate, using `WFSession.chooseApplication()`

```
session.chooseApplication(myApplicationID);
```

Note: Choosing an application is optional. To know about detailed behavior when an application is not chosen, refer the `ApplicationModeSecurity` parameter in the *Interstage BPM Server Administration Guide.*

5.1.3 Designing a Process Definition with Start Node, User Task Node and End Node

After a user is logged in, the user can design a new process definition. A simple process definition consists of only one Start Node, of one or more User Task Nodes and of one or more End Nodes.

Note: Before creating a process definition, if you do not choose an application:

- In `Relax` mode, the default `System` application is selected.
- In `Secure` mode, the default `System` application is selected. If `System` application does not exist, then an exception will be thrown.

A **Start Node** identifies the start of a process. Every process definition must have one and only one Start Node. A **User Task Node** represents a step in the process where human intervention is planned. A process definition may have any number of User Task Nodes. An **End Node** identifies the end of a process. Every process definition must have at least one End Node.

Each node can be connected to each other with arrows. The fundamental purpose of an arrow is to control the flow within a process. When a node completes, the process instance moves from the tail of an arrow to the head of another arrow. There is no limit to the number of arrows that can enter or leave a node (except for Start Nodes, which can only have an unlimited number of arrows leaving them, or End Nodes, which can only have an unlimited number of arrows entering them).

To design a process definition:

1. Create an empty process definition object with `WFObjectFactory.getPlan()`. Set the current session to the process definition object with `Plan.setWFSession()`.

```
Plan plan = null;
plan = WFObjectFactory.getPlan();
plan.setWFSession(session);
```

2. Before adding nodes and arrows to a process definition, change the current mode of the process definition to edit-mode with `startEdit()` from the `Plan` interface.

```
plan.startEdit();
```

3. You can add some general information to the process definition with `setName()`, `setTitle()`, and `setDesc()` from the `Plan` interface.

```
plan.setName("My Plan");
plan.setTitle("My first process definition");
plan.setDesc("Creation of a simple process definition");
```

4. Add the Start Node, User Task Node, and End Node to the process definition object.

   For adding nodes use `addNode(name, nodeType)` from the `Plan` interface. The constant `nodeType` defines the node type you want to add. You find the possible values for this constant in the `Node` interface.

   `setUpperLeftPoint()` from the `Node` interface defines the position of a node. Each node has an X and Y coordinate to show where it resides on the canvas if it is shown graphically. This positioning information is implementation-specific, so it will depend on how the node is represented. When setting this parameter, choose a coordinate system that is appropriate for your implementation.

   a) **To add a Start Node**:

```
Node startNode = plan.addNode("Start", Node.TYPE_START);
startNode.setUpperLeftPoint(new Point(100, 150));
```

   b) For User Task Nodes you can assign a User Group with `setRole()` from the `Node` interface. In addition, you can define whether the User Task Node will generate a group work item when it becomes active, or individual work items for every member of the Group assigned to the User Task node. This setting also depends on the value of the `SupportGroupWorkItem` parameter of the Interstage BPM Server. By default, this parameter is set to `false`. Refer to the *Interstage Business Process Manager Server Administration Guide* for details on the `SupportGroupWorkItem` parameter.

   To add a User Task Node :

```
Node activityNode = plan.addNode("Activity", Node.TYPE_ACTIVITY);
activityNode.setUpperLeftPoint(new Point(200, 250));
activityNode.setRole("SampleGroup");
activityNode.markForExpandGroups(false);
```

   Only User Task Nodes, Voting User Task Nodes and Embedded Sub-Process Nodes support assigning Groups.

   c) **To add an End Node**:

```
Node exitNode = plan.addNode("Exit", Node.TYPE_EXIT);
exitNode.setUpperLeftPoint(new Point(300, 350));
```

5. Connect nodes with arrows using `addArrow()` from the `Node` interface. When adding an arrow, specify the name, source and target of the arrow.

```
Arrow goArrow = plan.addArrow("go", startNode, activityNode);
Arrow stopArrow = plan.addArrow("stop", activityNode, exitNode);
```

6. After adding nodes and arrows to a process definition object, you need to validate it with `validatePlan()` from the `Plan` interface.

```
plan.validatePlan();
```

7. Create a process definition with `createProcessDef()` from the `Plan` interface.

```
plan = plan.createProcessDef();
```

Note: A process definition will not be created on the Interstage BPM Server until you use `createProcessDef()`. At this point, when the process definition object is committed to the Interstage BPM Server, the Interstage BPM Server assigns a process definition ID to it.

The figure below illustrates the resulting process definition.

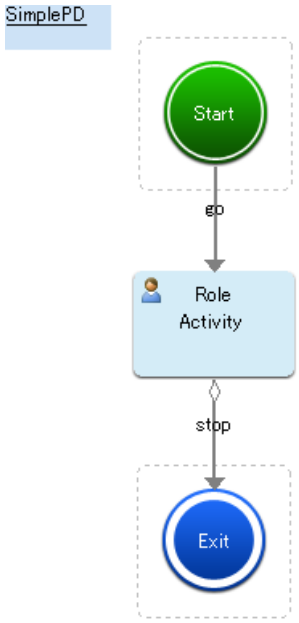

Figure 6: Simple Process Definition with a Start Node, User Task Node and End Node

5.2 Designing a Complex Process Definition

This section provides programming instructions for creating a complex process definition with nearly all possible components. You can find the complete programming code of the examples presented

in this section in the `ComplexPlan.java` sample file. The following figure illustrates how the workflow elements defined in the sample file interact.

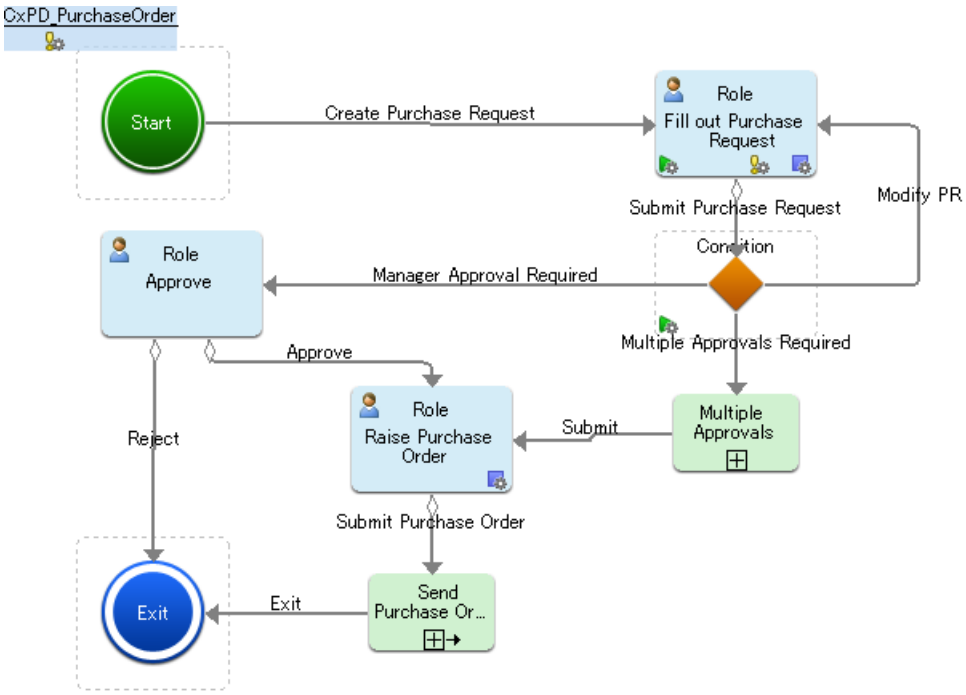

Figure 7: A Complex Process Definition with Different Node Types

In the previous section, you learned how to add basic workflow elements like Start Node, End Nodes, and User Task Nodes. The following sections explain how to add User Defined Attributes and how to add additional node types like Voting User Task Nodes, Parallel Join Gateway Nodes, Parallel Split Gateway Nodes, and so on.

To design a complex process definition:

1. Log in a user.

   Refer to section *Logging in/Logging out a Normal User* on page 65 for information about to how to log in and log out a user.

2. Use `startEdit()` from the `Plan` interface to set the edit-mode for a process definition.

   Once in edit-mode, the Interstage BPM Server locks the process definition so that no other user can make changes to it while an edit session is running. Only draft or private process definitions can be edited. Draft process definitions can be edited only if there are no current running process instances associated with them.

3. Add a Start Node, at least one End Node and the required User Task Nodes.

   Refer to section *Designing a Process Definition with Start Node, User Task Node and End Node* on page 66 for information about how to add these node types.

4. Add User-Defined Attributes to the process definition. Add the other node types like Voting User Task Nodes, Parallel Join Gateway Nodes, and so on.

   For details, refer to the following sections.

5. Connect the nodes with arrows.

   For details, refer to section *Designing a Process Definition with Start Node, User Task Node and End Node* on page 66.

6. Validate the process definition and create it.

   For details, refer to section *Designing a Process Definition with Start Node, User Task Node and End Node* on page 66.

7. Check if you want to make use of the extended features for enhancing your business processes, such as Java Actions, Extended attributes, agents, timers, error handling, etc.

   For details, refer to chapter *Enhancing Interstage Business Process Manager* on page 88.

5.2.1 Adding User Defined Attributes

The `DataItemRef` interface is used to hold the identifier, the name, type, and the initial values of User Defined Attributes (UDAs) defined in the process definitions.

UDAs are variables that are global to the process instance, so that all nodes within the process instance can access all of the UDAs. UDAs specify the behaviour of nodes and store data for process execution.

UDAs have one of the following data types: BIGDECIMAL, BOOLEAN, DATE, FLOAT, INTEGER, LONG, STRING, XML, Custom data types.

Note: A UDA has a name and an identifier.

The UDA name

- is user-defined
- may contain any character (maximum: 64 characters)
- must not start with two underscores (__), because two underscores are used as prefix of UDAs created and maintained by the system.

The UDA identifier

- is either user-defined or generated by the system. If you do not specify an identifier when creating a UDA, the identifier is automatically generated as follows: The name is taken as input; all special characters are removed. If this "sanitized name" (identifier) is longer than 32 characters, empty or not unique, the identifier will be composed of the prefix 'uda<number>', e.g. 'uda1'. The number is increased by one each time a new UDA is added that fulfills the above conditions.
- may consist of a maximum of 32 characters
- may not contain any special characters (all characters except 'a' - 'z', 'A' - 'Z', '0' - '9')
- must be unique throughout the entire process definition

For every user interaction, the UDA name is to be used. Whenever necessary, the name is automatically mapped to the identifier. Once created, the UDA identifier cannot be changed anymore; the UDA name can, however, be changed any time.

To add UDAs to a process definition:

1. Make sure that you have changed the current mode of the process definition to edit-mode with `startEdit()` from the `Plan` interface.
2. To add UDAs, use `addDataItemRef()` or `addDataItemRefWithId()` from the `Plan` interface.

   Using the first method will generate an identifier for the UDA; using the second method, you can define your own identifier.

Example

```
protected final static String WLUDA_PRICE = "Price";
protected final static String WLUDA_QTY = "Qty";
protected final static String WLUDA_TOTAL = "Total";

DataItemRef udaPrice = plan.addDataItemRef(WLUDA_PRICE,
             DataItemRef.TYPE_FLOAT, "0.0");
DataItemRef udaQty = plan.addDataItemRef(WLUDA_QTY,
             DataItemRef.TYPE_INTEGER, "0");
DataItemRef udaTotal = plan.addDataItemRef(WLUDA_TOTAL,
             DataItemRef.TYPE_FLOAT, "0.0");
DataItemRef udaJavaActionTest = plan.addDataItemRefWithId(
           "JavaActionTest", "My JA Test", DataItemRef.TYPE_STRING, "0");
DataItemRef udaMapping = plan.addDataItemRefWithId(
            "MapUDAParent", "Mapping UDA Parent",
             DataItemRef.TYPE_STRING,
             "This value is from parent process");
DataItemRef udaCondition = plan.addDataItemRef("Condition",
             DataItemRef.TYPE_STRING, "");
DataItemRef udaSec = plan.addDataItemRef("SEC",
             DataItemRef.TYPE_STRING, "sec");
```

Note: Use the `isIdentifierUnique(String ID)` method from the `Plan` interface for checking whether a user-defined identifier is unique within the entire process definition.

5.2.2 Using Voting User Task Nodes

A Voting User Task Node uses voting rules to determine the choice (node's outgoing arrow) that wins. For every Voting User Task Node you specify the rules for voting.

To add a Voting User Task Node:

1. Create a node using `addNode()`. Set the constant `nodeType` to `TYPE_VOTING_ACTIVITY`.

```
Node directorApproveNode = plan.addNode("Approve",
   Node.TYPE_VOTING_ACTIVITY);
directorApproveNode.setRole(userGroup);
directorApproveNode.setUpperLeftPoint(new Point(290, 350));
```

2. Define the rules and the threshold for the Voting User Task Node using `setVotingRule()` from the `Node` interface.

   Choose between the following types of rules:

   - `VotingRule.TYPE_NUMBER` (Specified Number Rule)
   - `VotingRule.TYPE_PERCENTAGE` (Specified Percentage Rule)
   - `VotingRule.TYPE_MAJORITY` (Majority Rule)

   The threshold defines the value when a vote is performed.

```
directorApproveNode.setVotingRule("Reject",
   VotingRule.TYPE_NUMBER, 1);
directorApproveNode.setVotingRule("Approve",
   VotingRule.TYPE_NUMBER, 1);
```

   The sample code specifies a voting rule of type `NUMBER` with the threshold 1 for both outgoing arrows. This means, if one user makes a choice, the action associated with the arrow is performed.

3. Specify the time when the votes are checked using `setEvaluateRulesMode()`.

You can choose between the following:

- `Node.ON_EVERY_VOTE`: Conditions will be checked after every vote.
- `Node.WHEN_ALL_VOTES_ARE_CAST`: Conditions will be checked after all users of the assigned Group have voted.

```
directorApproveNode.setEvaluateRulesMode(Node.ON_EVERY_VOTE);
```

4. Define the default choice for the voting rule with `setDefaultChoice()`.

```
directorApproveNode.setDefaultChoice("Reject");
```

5.2.3 Adding Parallel Join Gateway Nodes and Parallel Split Gateway Nodes

A Parallel Join Gateway Node represents a step in a process where the process instance pauses to synchronize multiple threads of execution. A process definition can have any number of Parallel Join Gateway Nodes.

A Parallel Split Gateway Node represents a step in a process from where a single branch of control splits into multiple parallel branches. A process definition can have any number of Parallel Split Gateway Nodes.

- **To add a Parallel Join Gateway Node:**

  Use `addNode()` and set the constant `nodeType` to `TYPE_AND`.

```
Node andNode = plan.addNode("And Node", Node.TYPE_AND);
andNode.setUpperLeftPoint(new Point(440, 460));
```

- **To add a Parallel Split Gateway Node:**

  Use `addNode()` and set the constant `nodeType` to `TYPE_OR`.

```
Node orNode = plan.addNode("Or Node", Node.TYPE_OR);
orNode.setUpperLeftPoint(new Point(430, 230));
```

5.2.4 Using Simple Exclusive Gateway Nodes

A Simple Exclusive Gateway Node is a Node with a Prologue Action. After adding the Simple Exclusive Gateway Node to the process definition you first have to define a Prologue Action and then you have to specify the condition for the node.

To add a Simple Exclusive Gateway node:

1. Use `addNode()` and set the constant `nodeType` to `TYPE_CONDITION`.

```
Node condNode = plan.addNode("Conditional Node",
                  Node.TYPE_CONDITION);
condNode.setUpperLeftPoint(new Point(430, 140));
```

2. Define the Prologue Actions. Refer to chapter *Enhancing Interstage Business Process Manager* on page 88 for details.
3. Define the conditions for the Simple Exclusive Gateway Node using `getConditionSpec()` and `setCondBranchSpecInfo()` from the `Node` interface.

Example

In the following example, three conditions for the outgoing arrows of a Simple Exclusive Gateway Node are defined. The values of the User Defined Attribute (UDA) `Total` represent the borders for a given budget:

- Modify PR: Total <= 0.0 (default)
- Manager Approval Required: Total > 0.0 and Total < = 1000.0
- Multiple Approvals Required: Total > 1000.0

It is not possible to use the UDA `Total` to decide which branch is used, because it contains two conditions, but only one condition can be defined per branch. Therefore a further User Defined Attribute `Condition` is introduced. Depending on the UDA `Total` the UDA `Condition` is set to one of three unique values. These values are required to control which arrow is used. A Java Action, named `SampleJavaActions` in this example, is used to wrap the current value of the UDA `Total` to the UDA `Condition`, which is then checked.

The conditions for the wrapping are:

```
if (Total <= 0.0) {
   Condition = "Modify PR";
} else if (Total > 0.0 && Total <= 1000.0) {
   Condition = "Manager Approval Required";
} else if (Total > 1000.0) {
   Condition = "Multiple Approvals Required";
} else {
   Condition = "Modify PR";
```

The `getConditionSpec()` from the `Node` interface which is defined for this Simple Exclusive Gateway Node uses this User Defined Attribute (UDA) to retrieve information about the arrow it has to choose. This is necessary because only one `ConditionSpec` object can be defined containing only one UDA. With `setCondBranchSpecInfo()` from the `ConditionSpec` interface you can add a new branch to the `ConditionSpec` object.

```
ConditionSpec conditionSpec = condNode.getConditionSpec();
conditionSpec.setConditionAttribute("Condition");

conditionSpec.setCondBranchSpecInfo("Modify PR",
   BranchSpec.EQUAL_OP, "Modify PR", true );
conditionSpec.setCondBranchSpecInfo("Multiple Approvals Required",
   BranchSpec.EQUAL_OP,
   "Multiple Approvals Required", false );
conditionSpec.setCondBranchSpecInfo("Manager Approval Required",
   BranchSpec.EQUAL_OP, "Manager Approval Required", false );
```

The Interstage BPM Model API also provides a method for accessing values of XML substructures: `setConditionAttribute(String udaName, String xPath)` as well as for retrieving the value of an xPath by calling the `getConditionAttributeXPath()` method. Refer to the API Javadoc for detailed information.

Note: The order of evaluation of outgoing arrows can be changed in Interstage BPM Studio.

5.2.5 Using Call Activity Nodes

A Call Activity Node represents a step in a process where a task is accomplished by invoking another process definition, and waiting for a result to come back. This node can be used to serve multiple purposes, especially for reusing existing process definitions within new ones.

A single Call Activity Node can represent an entire process definition that is already designed, accessing all of the nodes, arrows, and other characteristics of that process definition. The Call Activity Node can then call an already existing process definition via its name.

In the example file `ComplexPlan.java` a Call Activity Node calls a process definition with the name `CxPD_Approval`. The following figure illustrates this process definition:

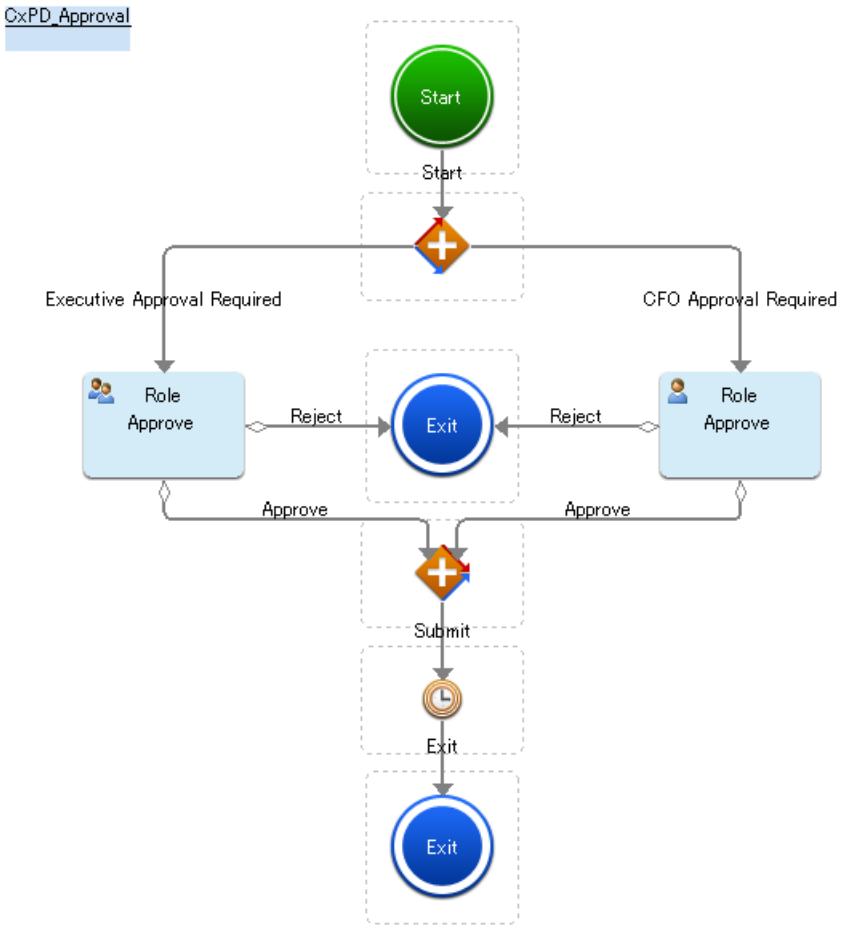

Figure 8: A Process Definition with Different Node Types called from a Call Activity Node

To use a Call Activity node:

1. Add a Call Activity Node to the process definition. To do so, use `addNode()` and set the constant `nodeType` to `TYPE_SUB_PROCESS`.

```
Node subProcessNode = plan.addNode("Multiple Approvals",
   Node.TYPE_SUB_PROCESS);
subProcessNode.setUpperLeftPoint( new Point(406, 237));
```

2. Specify the name of an existing process definition for which a process instance is to be generated when the Call Activity Node is called. To do so, use `setSubPlanName()` from the `Node` interface.

```
subProcessNode.setSubPlanName(subPlan.getName());
```

A new process instance of the **latest version** of this subprocess definition will automatically be generated.

Note: If you need to use a subprocess definition of the **same version** as that of the parent process definition, use the `setSameSubPlanVersion(boolean)` method of the `com.fujitsu.iflow.model.workflow.Plan` interface to do this. For example, for a parent process definition 'A' of version 2, you can use the method mentioned above to use only version 2 (and not the latest version) of all its subprocesses. Sample code to set such functionality for a parent process definition is as follows:

```
plan.startEdit();
plan.setSameSubPlanVersion(true);
plan.commitEdit();
```

If you set this feature for a parent process, ensure that for a version of a parent process definition, subprocess definitions having that same version exist. If the parent process instance tries to call a subprocess definition with a version number that does not exist, an error is thrown.

While updating an application using the `WFSession.updateApplication()` method, if you want to use the same versions of the subprocess definitions, you need to update the parent process definition as well as the child process definitions.

3. Map the flow of information between a User Defined Attribute (UDA) in the parent process definition and a UDA in the subprocess definition using `addDataMappingElement()`.

   A `DataItemMappingElement` that contains a UDA in a parent process definition mapped to a UDA in a subprocess definition is called "input data mapping element".

```
subProcessNode.addDataMappingElement("MappingUDAParent",
   "MappingUDAChild", DataItemMappingElement.INOUT);
```

Note: Be careful when designing process definitions that have recursive subprocesses. Check all process definitions involved and make sure that there are no infinite recursions.

5.2.6 Using Timer Nodes

A Timer Node represents a step in a process where process execution is suspended for a certain amount of time.

To use a Timer Node:

1. Add a Timer Node to the process definition. To do so, use `addNode()` and set the constant `nodeType` to `TYPE_DELAY`.

```
Node delayNode = plan.addNode("Delay Node", Node.TYPE_DELAY);
delayNode.setUpperLeftPoint(new Point(434, 500));
```

2. Add a timer to the Timer Node to specify how long the process execution will be suspended.

   Refer to section *Defining a Timer* on page 167 for an example.

5.2.7 Using Chained-Process Nodes

A Chained-Process Node represents a step in a workflow process where a new process instance is created to accomplish a task independent of the parent process instance. This node enacts this independent process instance as a part of the execution of the parent process instance. The Chained-Process Node can only call an already existing process definition via its name.

In the example file `ComplexPlan.java` a Chained-Process Node calls a process definition with the name `CxPD_Send`. The following figure illustrates this process definition.

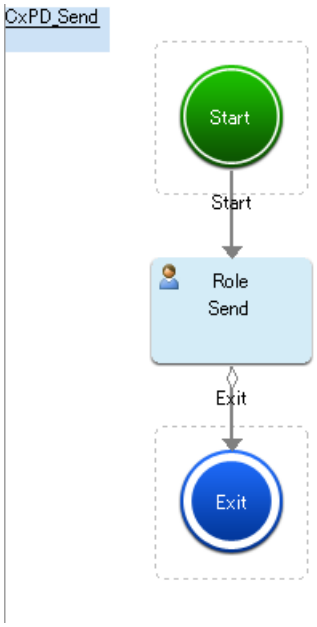

Figure 9: A Process Definition Called from a Chained-Process Node

To use a Chained-Process Node:

1. Add a Chained-Process Node to the process definition. To do so, use `addNode()` and set the constant `nodeType` to `TYPE_CHAINED_PROCESS`.

```
Node chainedProcessNode = plan.addNode("Send Purchase Order",
Node.TYPE_CHAINED_PROCESS);
chainedProcessNode.setUpperLeftPoint( new Point(212, 391));
```

2. Specify the name of an existing process definition for which a process instance is to be generated when the Chained-Process Node is called. To do so, use `setChainedPlanName()` from the `Node` interface.

```
chainedProcessNode.setChainedPlanName(chainedPlan.getName());
```

   A new process instance of the **latest** version of this chained-process definition will automatically be generated.

Note: If you need to use a chained-process definition of the **same version** as that of the parent process definition, use the `setSameSubPlanVersion(boolean)` method of the `com.fujitsu.iflow.model.workflow.Plan` interface to do this. For example, for a parent process definition 'A' of version 2, you can use the method mentioned above to use only version 2 (and not the latest version) of all its chained-processes. Sample code to set such functionality for a parent process definition is as follows:

```
plan.startEdit();
plan.setSameSubPlanVersion(true);
plan.commitEdit();
```

If you set this feature for a parent process, ensure that for a version of a parent process definition, chained-process definitions having that same version exist. If a parent process instance tries to call a chained-process with a version number that does not exist, an error is thrown.

While updating an application using the `WFSession.updateApplication()` method, if you want to use the same versions of the subprocess definitions, you need to update the parent process definition as well as the child process definitions.

3. Map the flow of information between a User Defined Attribute (UDA) in the parent process definition and a UDA in the chained-process definition using `addDataMappingElement()`.

   A `DataItemMappingElement` that contains a UDA in a parent process definition mapped to a UDA in a chained-process definition is called "input data mapping element".

```
chainedProcessNode.addDataMappingElement("MappingUDAParent",
   "MappingUDAChild", DataItemMappingElement.INOUT);
```

5.3 Working with Process Instances

The following sections provide instructions for retrieving a process definition and starting a process instance from it. In addition, the execution of work items, which are generated within a process instance, is shown. You can find the complete programming code of the examples presented in these sections in the sample file `ProcessExecution.java`.

Before doing any work on process instances and work items, the user must be logged in. Refer to section *Logging in/Logging out a Normal User* on page 65 for information about how to log in and log out a user.

The following figure shows the necessary steps when working with process instances:

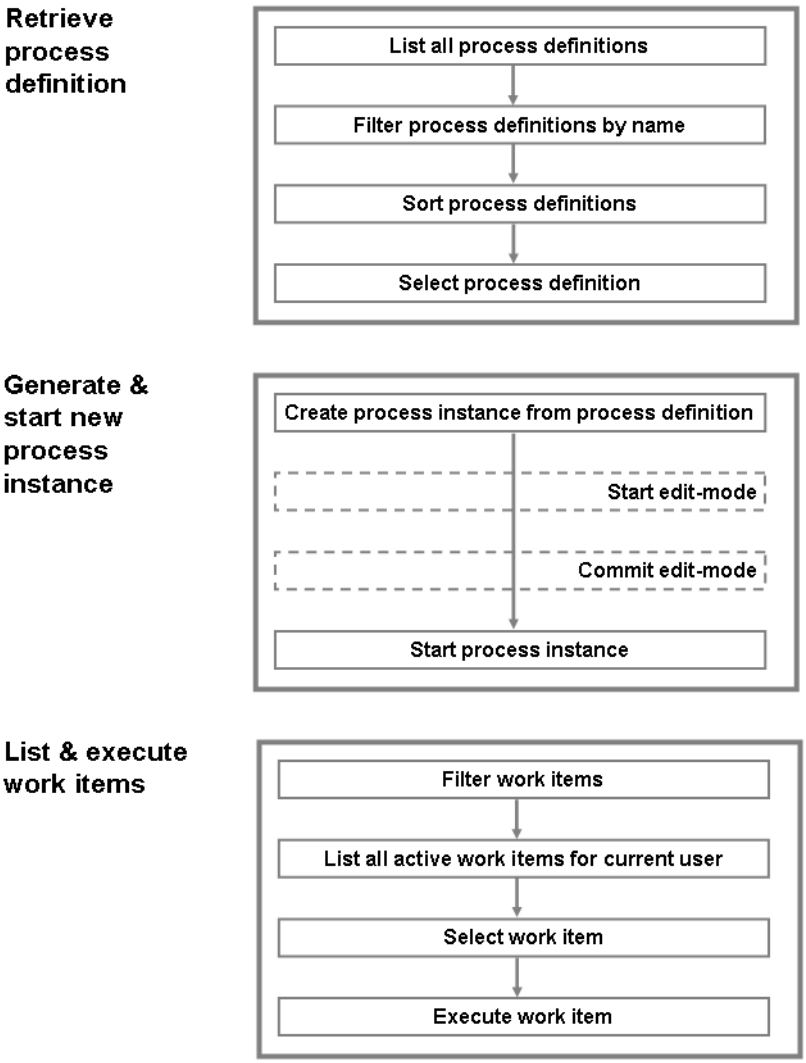

Figure 10: Working with Process Instances Using the Model API

5.3.1 Retrieving the Latest Version of a Process Definition

To retrieve the latest version of a process definition:

1. List all process definitions that are available on the Interstage BPM Server. To do so, use the `WFObjectList` interface from the package `com.fujitsu.iflow.model.workflow` to get a list of the existing `WFObjects` from the Interstage BPM Server.

   The `WFObjectFactory` class provides you with consistent, non-proprietary means of accessing the workflow objects.

```
Plan plan = null;
WFObjectList wfObjectList =
```

```
   WFObjectFactory.getWFObjectList(session);
Object[] planList = null;
```

2. Filter the process definitions according to their names. To do so, use `addFilter()` from the `WFObjectsList` interface to add filtering criteria to the list of the process definitions to be retrieved.

```
wfObjectList.addFilter(WFObjectList.LISTFIELD_PLAN_NAME,
WFObjectList.SQLOP_EQUALTO, "\'" + planName +"\'");
```

Note: Include the process definition name in single quotes `" ' "`, because it is directly included in the generated database query.

3. Sort the process definitions according to a desired criteria. Use `addSortOrder()` to add a field-based sort order to the list of the process definitions to be retrieved.

```
wfObjectList.addSortOrder(WFObjectList.LISTFIELD_PLAN_ID,
   WFObjectList.SORTORDER_DESCENDING);
wfObjectList.openBatchedList(Filter.AllPlans);

planList = wfObjectList.getNextBatch(1);
```

   In the code sample, process definitions are sorted by their ID. `openBatchedList()` returns a list of process definitions from the Interstage BPM Server matching the filter criteria. `getNextBatch(1)` retrieves the next entry only, because we only search for the latest version of the process definition.

4. Use the process definition with the latest version.

```
if (planList != null && planList.length > 0) {
   plan = (Plan) planList[0];
}
return plan;
```

   The first entry of the returned list is the searched process definition version.

5.3.2 Creating and Starting a New Process Instance

After you have retrieved a process definition, you can create a process instance from it and start the process instance.

To create and start a new process instance:

1. Create a process instance using `createProcessInstance()` from the `Plan` interface.

```
processInst =
WFObjectFactory.createProcessInstance(plan.getId(),session);
```

   `createProcessInstance()` copies the process definition's structure and attributes to a running process instance, making it appear as if it had been instantiated.

   When a process definition is copied to a new process instance, the components of the process definition are also copied as components of the new process instance. For example, a `StartNode` object in the process definition is copied to a `StartNodeInstance` object in the new process instance.

2. You can edit the parameters of the process instance by setting it to edit-mode with `startEdit()`.
3. When the editing is finished, stop edit-mode with `commitEdit()`.

   Otherwise the process instance is locked for further operations.

4. Start the process instance.

```
processInst.start();
```

5.3.3 Listing Work Items

For listing work items, use the `WFObjectsList` interface from the `com.fujitsu.iflow.model.workflow` package. This interface retrieves a list of existing `WFObjects` from the Interstage BPM Server.

To list work items:

1. Build a filter using the `Filter` class.

   Possible filter criteria for work items are:

   - `AllWorkItems`: Retrieves all work items. This is the main filter for retrieving group work items.
   - `MyAcceptedWorkItems`: Retrieves all work items including group work items accepted by the logged-in user.
   - `MyActiveWorkItems`: Retrieves all active work items belonging to the logged-in user.
   - `MyCompletedWorkItems`: Retrieves all work items completed by the logged-in user.
   - `MyDeclinedWorkItems`: Retrieves all work items declined by the logged-in user.
   - `MyWorkItems`: Retrieves all work items belonging to the logged-in user.

2. Use `openBatchedList()`.
3. Call `getNextBatch()`.

   `getNextBatch()` returns an array of workflow objects matching the filter criteria set with `openBatchedList()`.

Example

```
WorkItem[] workItemList = null;
if (filter == Filter.AllWorkItems
|| filter == Filter.MyAcceptedWorkItems
|| filter == Filter.MyActiveWorkItems
|| filter == Filter.MyCompletedWorkItems
|| filter == Filter.MyDeclinedWorkItems
|| filter == Filter.MyWorkItems) {
   WFObjectList wfObjectList =
   WFObjectFactory.getWFObjectList(session);
   wfObjectList.openBatchedList(filter);
   int batchSize = 50;
   Object[] elements = wfObjectList.getNextBatch(batchSize);
}
```

The filter `Filter.MyActiveWorkItems` is not supported for group work items. If the `SupportGroupWorkItem` parameter of the Interstage BPM Server is set to true, you cannot retrieve work items using this filter. To retrieve work items, use the filter `Filter.AllWorkItems` instead. For advanced filtering based on the name of the group to which the group work item belongs, use `LISTFIELD_WORKITEM_ASSIGNEE` together with SQL operators. Refer to the *Interstage Business Process Manager Server Administration Guide* for details on the `SupportGroupWorkItem` parameter.

Examples for Filtering Group Work Items

Consider UserX who is a member of two user groups: GroupA and GroupB.

Scenario 1: UserX wants to retrieve all group work items that are assigned to GroupA:

```
...
wfObjectList.addFilter (WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
WFObjectList.SQLOP_EQUALTO, "GroupA");
wfObjectList.openBatchedList(Filter.AllWorkItems);
...
```

Scenario 2: UserX wants to retrieve all group work items that are assigned to both groups:

```
...
wfObjectList.addFilter (WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
WFObjectList.SQLOP_IN, "(GroupA,GroupB)");
wfObjectList.openBatchedList(Filter.AllWorkItems);
...
```

Scenario 3: UserX wants to retrieve all group work items that are assigned to both groups as well as all individual work items assigned to himself/herself:

```
...
wfObjectList.addFilter (WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
WFObjectList.SQLOP_IN, "(GroupA,GroupB,UserX)");
wfObjectList.openBatchedList(Filter.AllWorkItems);
...
```

5.3.4 Executing Work Items

Only active work items can be executed.

You can execute a work item using any of the following two methods:

1. **To execute a work item without enactment edit mode:**
   a) Generate a list with all active work items, for example:

```
wfObjectList.addFilter(WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
   WFObjectList.SQLOP_IN, "User,Group");
WorkItem[] workItemList = listWorkItems(Filter.AllWorkItems);
```

   b) Retrieve the possible choices allowed for a work item using `getChoices()` from the `WorkItem` interface.

```
choices = workItem.getChoices();
```

   The choices are the names of each of the outgoing arrows attached to the node that the work item represents.

   c) Accept the work item by using `accept()` from the `WorkItem` interface.

```
if (choices != null ) {
   workItem.accept();
   int choiceIdx = 0;
   ...
}
```

   `accept()` changes the state of the work item to `STATE_ACCEPTED`, and all other active work items associated with this User Task change to `STATE_DEACTIVE`.

d) Execute the work item, i.e. make a choice for it using `makeChoice()` from the `WorkItem` interface.

```
workItem.makeChoice(choices[choiceIdx]);
```

`makeChoice()` captures the specified choice for this work item through a parameter. `makeChoice()` completes the work item.

2. **To execute a work item in enactment edit mode:**

You can execute a work item in enactment mode. You can commit edit operations such as updating UDA along with the work item execution operation.

a) Generate a list with all active work items, for example:

```
wfObjectList.addFilter(WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
   WFObjectList.SQLOP_IN, "User,Group");
WorkItem[] workItemList = listWorkItems(Filter.AllWorkItems);
```

b) Retrieve the possible choices allowed for a work item using `getChoices()` from the `WorkItem` interface.

```
choices = workItem.getChoices();
```

The choices are the names of each of the outgoing arrows attached to the node that the work item represents.

c) Accept the work item by using `accept()` from the `WorkItem` interface.

```
if (choices != null ) {
   workItem.accept();
   int choiceIdx = 0;
   ...
}
```

d) Start enactment edit on the work item.

```
workItem.startEdit();
```

Note: You can complete any operation in enactment edit mode such as updating UDA or adding additional history information.

e) Execute the work item.

```
workItem.makeChoice(choices[choiceIdx]);
```

Specified choice will be saved and executed within the `commitEdit` method.

Note: Only one work item execution operation is allowed in enactment edit mode.

f) Commit edit.

```
workItem.commitEdit()
```

This saves the enactment edit operations and also executes the work item.

5.3.5 Recalling Work Items

Only completed work items can be recalled.

The node that is the target of recall is called the **Recall Target**. The node that was active prior to the **Recall Target** is called the **Recall Source**.

The status of node instances in the recall process undergoes the following changes:

- Recalling a work item results in the deactivation of the target node and it goes back to the closed state, the only exception is the Parallel Join Gateway node which continues to remain active.
- The Recall Source then goes into the active state.

Note: Only single level recall of work item is possible. Multiple level recall of work items cannot be done.

The figures below illustrate a successfully recalled work item, and the change in the status of node instances before and after recall:

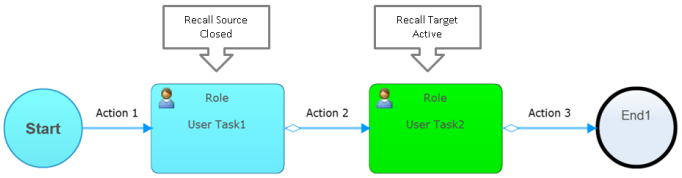

Figure 11: Status of Node Instances before Recall

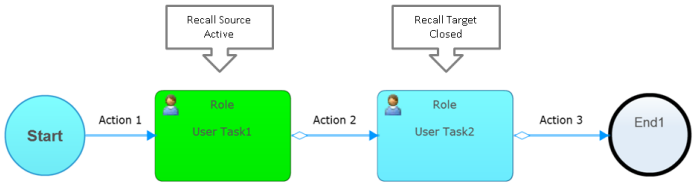

Figure 12: Status of Node Instances after Recall

In case of nodes connected through parallel branches to the Parallel Join Gateway or Parallel Split Gateway nodes or leading to a common node, both the parallelly connected nodes cannot be recalled simultaneously. Only one node can be recalled at a time. The work item for the recalled node has to be committed and only then can the second node be recalled.

For instance in the figures below, User Task 2 has been recalled. User Task 1 cannot be recalled until work item for User Task 2, that is, action 4 is committed.

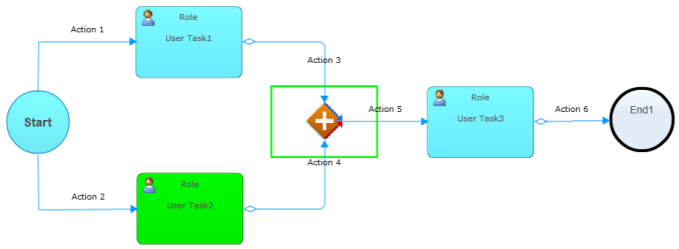

Figure 13: User Task Nodes connected parallelly to the Parallel Join Gateway Node

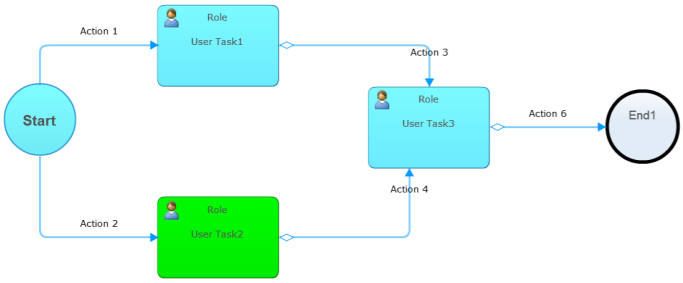

Figure 14: User Task Nodes connected parallelly leading to a common User Task node

When a work item is recalled, compensate java actions are executed to roll back operations done by java actions executed before recall.

Conditions for Recalling Work Item

To recall a work item, its imperative that the node types are supported by the recall functionality. However this support is provided only if the relevant conditions for the node types are met.

The following table gives an overview of the node types supported and not supported by the recall functionality and the conditions to be met by different node types in order to be supported as recall targets.

Node Type	Recall Source	Recall Targets	Conditions
User Task Node	Supported	Supported	• It is not an Agent • It has not been accepted • It is in the active state and not in the error or closed state • The process instance is not in the suspended or aborted state
Voting User Task Node	Not supported	Supported	• No voting has been done • The process instance is not in the suspended or aborted state
Parallel Join Gateway Node, Parallel Split Gateway Node (including Send Task, Business Rule Task, Script Task, Email, DB, WebService and Customized Nodes) and Simple Exclusive Gateway Node (including Flexible Exclusive Gateway Node)	Not supported	Not supported	-
Call Activity, Chained-process, and Remote Sub-Process Nodes	Not supported	Not supported	-
Timer Node	Not supported	Supported	• It is in the active state and not in the error or closed state • The process instance is not in the suspended or aborted state
Message Receive Node	Not supported	Supported	• It is in the active state and not in the error or closed state • The process instance is not in the suspended or aborted state
End Node	Not supported	Not supported	-
Start Node	Not supported	Not supported	-

Node Type	Recall Source	Recall Targets	Conditions
Iterator Node	Supported	Supported	• Only User Task Nodes are included in the Iterator Node, that is, no Call Activity or Agent Nodes are included in it • All the User Task Nodes included in the Iterator Node are active • User Task Nodes included in the Iterator Node have not been accepted • The process instance is not in the suspended or aborted state

Note: Parallel Join Gateway Node, Parallel Split Gateway Node (including Send Task, Business Rule Task, Script Task, Email, DB, WebService and Customized Nodes) and Simple Exclusive Gateway Node (including Flexible Exclusive Gateway Node) cannot became recall targets, however, when they exist between recall source and recall target, recall is possible.

The recall flag is used to disable or enable the recall functionality for an activity. If the recall flag is set to `false`, then recall is enabled, and if it is set to `true` then recall is disabled. To check the status of recall flag, use `isRecallDisabled()`. By default, recall flag is set to `false`. To set the recall flag to `true` or `false` use `setRecallDisabled(boolean)`.

To recall a work item:

1. Generate a list of Completed Work Items.

```
wfObjectList.openBatchedList(Filter.MyCompletedWorkItems);
```

2. Select the work Item you want to recall.

```
Object[] elements = wfObjectList.getNextBatch(100);
workItem =(WorkItem)elements[n];
```

3. Recall the source work item.

```
 workItem.recall();
```

   This results in the recall of the selected work item.

5.3.6 Handling Attachments

In the context of a process instance, the `AttachmentRef` interface allows you to access attachments in a specified Document Management System (DMS).

Note: The DMS must be one of the DMS directories specified in the Interstage BPM configuration.

An attachment is referenced by two attributes: name and path. The name attribute is a short descriptive value used for identification. The path attribute describes the fully qualified path to the attachment.

There is no limit to the number of attachments that can be associated with a process instance, and there are no restrictions on the types of attachments that can be added.

Attachments are global to the process instance; any activity in a process instance can access the attachment.

- **To add an attachment:**

  Make sure that the process instance is in enactment edit-mode or in structural edit-mode. Then, construct a new `AttachmentRef` object using `ProcessInstance.addAttachment`.

```
procInst.startEdit();
procInst.addAttachment(ATTACHMENT_NAMES[attachIdx],
   ATTACHMENT_FILES[attachIdx]);
procInst.commitEdit();
```

- **To retrieve all attachments:**

  Use `ProcessInstance.getProcessAttachments .`

```
currentAttachments = procInst.getProcessAttachments();
```

  `getProcessAttachments()` returns the references to all the attachments associated with a process instance. Attachments are added to a process instance while it is running.

- **To retrieve a particular attachment:**

  Use `ProcessInstance.getAttachment`.

```
newAttachment =
procInst.getAttachment(ATTACHMENT_NAMES[attachIdx]);
```

  `getAttachment` returns the `AttachmentRef` object of the attachment associated with the process instance.

The `ProcessExecution.java` sample contains examples for handling attachments. Refer to the provided sample source code and the API Javadoc for details.

6 Enhancing Interstage Business Process Manager

This chapter describes the means of enhancing and extending Interstage BPM so that you can exploit its full power.

6.1 Integrating Interstage BPM with External Applications

6.1.1 Configuring Java Actions, Agents and JavaScripts for External Applications

Interstage BPM is designed to be integrated with other applications and a number of built-in Java Action Types are provided for this, but this is not always enough. You can extend the server with your own Java Action Types in order to call methods on classes external to the Interstage BPM Server. Normally these custom classes are installed as part of a BPM application using the lib or classes folder in an application. The advantage of adding with an application is that they move around with the application, and the server automatically finds them without additional trouble. Also, the server properly handles classes that conflict across different applications, so that each application gets the version of a class that was installed with that application. You will find this to be the most convenient way to extend the server capabilities.

However, you can also add these classes to the server to be available to all applications if you take the following integration steps. When you extend a server in this way, keep in mind that you will have to extend all of your testing and staging servers in the same way if you have applications that depend upon these classes. You also need to be careful about having the right version of the class that matches the currently installed application.

Note: Actions, Agents, and JavaScripts use the `ServerEnactmentContext` interface to access process information. The interfaces in the Model API are not accessible to Actions, Agents, or JavaScripts .

To install external Java classes:

1. To include external libraries, so they can be used in Interstage BPM, copy your external classes or JARs to the following locations:
   - To be able to use the files across tenants
     - If you use classes separately, copy the class files to `<engine directory>/server/instance/default/classes`
     - If you use a custom library, copy the JAR files to the Interstage BPM library extensions directory `<engine directory>/server/instance/default/lib/ext`
   - To be able to use the files so that they are specific to a tenant
     - If you use classes separately, copy the class files to `<engine directory>/server/instance/default/tenants/<tenant name>/classes`
     - If you use a custom library, copy the JAR files to `<engine directory>/server/instance/default/tenants/<tenant name>/lib/ext`
   - To be able to use the files so that they are specific to an application

     (Usually, this operation is done by Interstage BPM Studio. For details, refer *Interstage Business Process Manager Studio User's Guide*.)
     - If you use classes separately, copy the class files to `<DMSRoot>/apps/<application ID>/engine_classes`

- If you use a custom library, copy the JAR files to `<DMSRoot>/apps/<application ID>/engine_lib`
- If you use JavaScript files, copy them to `<DMSRoot>/apps/<application ID>/engine_js`

2. To use your own custom classes in a JavaScript, add them to a JAR file and copy the JAR file into the appropriate directory as mentioned in the step above. Then, instantiate the class with the "Packages." notation.

Note: The API `JavaAction#setClassPath()` is deprecated. It has effect only when the Interstage BPM Server is in non-SaaS mode. Also, it should be used only when the Java Action classes are not present in the predefined locations as specified above.

Example of Using a Custom Class in a JavaScript

You want to use `test.class`, a special purpose custom Java class. You add it to a JAR file called `test.jar`, and copy the JAR file into `<engine directory>/server/instance/default/lib/ext`. Then, you instantiate `test.class` as follows in your JavaScript:

```
var test = new Packages.test();
```

The class `SalaryCommission` must be located in either of the following paths:

- In the global classpath (`<ServerRoot>/classes` or for libraries `<ServerRoot>/lib/ext`)
- In the tenant classpath (`<ServerSharedRoot>/tenants/<tenant name>/classes` or for libraries `<ServerSharedRoot>/tenants/<tenant name>/lib/ext`)
- In the application classpath (`<DMSRoot>/apps/<application ID>/engine_classes` or for libraries `<DMSRoot>/apps/<application ID>/engine_lib`)

But the javascript uses the classloader of the `ServerEnactmentContext` instance where it is executed in. This instance is created without the knowledge of the action classpath. JavaScriptUtil then uses this classloader and additionally adds the global and if needed the application classpath. Please correct the test and add SalaryCommission to one of the loaded classpaths.

6.1.2 Managing Interstage BPM Sessions in an External Web Based Application

If you have a web based application that interacts with Interstage BPM server, you should properly manage the lifecycle of Interstage BPM session with each HTTP session in the application. In simple cases such as when a user logs out from the application, this should internally log out the Interstage BPM session. For complex cases like HTTP session timeouts and invalidated HTTP sessions where the application will not have control to logout the associated Interstage BPM session, you need to implement the `HttpSessionListener` interface for your web application.

How to use `HttpSessionListener` interface:

The `HttpSessionListener` interface can be implemented to receive the notifications of the changes to the list of active sessions in a web application. It has two methods:

- `public void sessionCreated(HttpSessionEvent se)`: Notification that a session was created.
- `public void sessionDestroyed(HttpSessionEvent se)`: Notification that a session is about to be invalidated.

Follow the steps below to configure the `HttpSessionListener` in your application:

1. To receive notification events, configure the `HttpSessionListener` implementation in the deployment descriptor (`web.xml`) by adding the following tag:

```
<listener>
  <listener-class>package.MyHttpSessionListener</listener-class>
</listener>
```

   where, `MyHttpSessionListener` is your `HttpSessionListener` implementation.

2. Add the Interstage BPM session object as an attribute in the HTTP session when the user logs in into Interstage BPM server. Do this in the JSP where the web application login is handled:

```
WFSession wfSession = WFObjectFactory.getWFSession();
...
...
wfSession.logIn(server, userName, password);
session.setAttribute("IBPMSession", wfSession);
```

3. In the `sessionDestroyed()` method of `MyHttpSessionListener`, add the call to log out the Interstage BPM session. The sample code is as follows:

```
public void sessionDestroyed(HttpSessionEvent event) {
  try {
       HttpSession session = event.getSession();
      WFSession bpms = (WFSession)session.getAttribute("IBPMSession");

       if (bpms != null) {
           bpms.logOut();
       }
  } catch (Exception ex) {
     // handle the exception
  }
}
```

   Once you have configured the `HttpSessionListener` as given above, it will make sure that whenever the HTTP session expires, the corresponding Interstage BPM session is logged out.

6.2 Using Application Variables

Application Variables allow all users of a particular application to share data between processes in that application. With this ability, more dynamic behavior and various actions associated with it are possible in processes. For example, the Web Service Java Action can use an Application Variable to designate the Web Service location. This allows end users to change the Web Service location dynamically without changing the Java Action specification in the process definition.

These variables are defined as part of an application during application development in the Interstage BPM Studio and are available throughout the life cycle of the application. All of the Application Variables for an application are represented in the code as an XML file named Appvariable.xml. It can be found directly under the application. Appvariable.xml contains the variable names and values. The following is an example:

```
<properties>
    <entry key="VariableName1">Value1</entry>
    <entry key="VariableName2">Value2</entry>
    …
</properties>
```

Variables can be created only in the Interstage BPM Studio, but their values can be changed in the Interstage BPM Console. However, the value of an Application Variable can be updated only by the Application Administrator.

Application Variables are especially helpful to developers because they might need a particular setting or a variable to have a common value across all of the processes in that application. These variables are used in much the same way as user-defined attributes (UDAs). The only difference between them is that UDAs are limited to sharing data in a particular running process.

Model API

The following methods in the Model API WFSession package allow manipulation of Application Variables in runtime:

```
public java.util.Properties getApplicationVariables  (String appId) throws
ModelException;
```

This will return a Property object of Application Variables name, value pair.

```
public void setApplicationVariables(String appId, java.util.Properties properties)
 throws ModelException;
```

This method updates the Application Variables.

NOTE: You must be an Application Owner or System Administrator to set Application Variables.

6.3 Using Java Actions

Java Actions are extensions to the workflow engine. At its core, a Java Action Type is nothing more than an object paced in a node configured to call a static Java method. Custom Java Action Types can extend the workflow engine to do anything that is possible to do in Java. Typically, these Java Action Types are used to connect to and interact with external programs and services. Data can be moved from external programs and services to Interstage BPM and vice versa. One might use Java Actions to access a resource where business objects are shared across any number of processes. Java Actions can make use of standard protocols to access remote programs and services.

A Java Action is then an instance of a Java Action Type. A Java Action is a structure that gives the specifics on how to call the Java method represented by the Java Action Type. A specific Java Action tells how to send the input, and what to do with the output of the Java method. You can have any number of Java Action (instances) for any number of Java Action Types, in a given process.

The `JavaActionSet` interface of `com.fujitsu.iflow.model.workflow` is a container for `JavaActions`. You add Java Actions to an existing Java Action Set at design time. There are Java Action Sets at the process level, nodes, timers, etc. The set that you add the Java Action to determines when that Java Action will be executed.

Java Actions Sets exist for the following points of process execution:

- at process initialization (Init Action Set and Process Owner Action Set)
- at process completion (Commit Action Set)
- before an activity starts (Prologue Action Set and Role Action Set)
- at activity completion (Epilogue Action Set)
- when a timer expires (Timer Action Set)
- when a process instance is aborted, suspended, or resumed (onAbort, onSuspend, or onResume Action Sets)
- when an error occurs during process execution (Error Action Set)

- when an error occurs during the execution of a Java Action (Error Action Set and Compensate Action Set)
- when the starting of a remote subprocess fails (Error Action Set for a Remote Sub-Process Node)

Refer to section *Types of Java Actions* on page 92 for details.

Java Actions run within the context of a process instance. This means that they can read and update User Defined Attributes (UDAs) and other attributes of the particular process instance they are embedded within. When data is moved to a process instance, it is usually read from one or more external data sources and copied into UDAs. When data is moved to an external program or service, the values of UDAs are copied to one or more external data sources.

Note: Java Actions access the process information through the Server Enactment Context API.

You can, for example:

- Manipulate process flow through external programs or services. Refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93 for an example.
- Enhance Interstage BPM with functions that just read context data, e.g. to compare values. Refer to sections *Assigning Prologue Actions to a User Task Node* on page 94 and *Assigning Epilogue Actions to a User Task Node* on page 96 for examples.
- Manipulate process flow through existing Interstage BPM functionality (built-in Java Actions). Refer to section *Using Built-In Java Action Types* on page 96 for more information.
- Define specific error handling when an error occurs during the execution of a Java Action. Refer to section *Dealing with Errors in Java Actions* on page 104 for more information.
- Define specific error actions in case a remote subprocess cannot be started. Refer to section *Using Error Java Actions* on page 103 for more information.
- Handle the case when a process instance is aborted, suspended or resumed. Refer to section *Using onAbort, onSuspend, onResume Java Action Sets* on page 110 for an example.

6.3.1 Types of Java Actions

Java Action provides a way to call static Java methods during a process enactment. Java Actions can be added to the process definition at design time by placing the Java Action into a particular Java Action Set. There are a number of Java Action Sets for at the process level, such as Init, Process Owner, Commit, onAbort, onResume, onSuspend, or Error Java Action Sets. There are more Java Actions Sets for each Node such as Role, Prologue, Epilogue, onAbort, onResume, onSuspend, Error or Timer Java Action Sets. In addition, Compensate and Error Action Sets exist for every Java Action.

Interstage BPM offers the following Java Action Sets:

- **Init Action Set** and **Process Owner Action Set** are executed upon process initialization. Java Actions in these sets initialize User Defined Attribute data before the first activity is started.
- **Commit Action Set** contains Java Actions that are executed upon process completion. They can be used to clean up or analyze the data of an entire process instance at the very end of enactment.
- **Prologue Action Set**. Java Actions in the Prologue Action Set are executed as the node is being activated. These Java Actions can be used to set up or initialize values associated with the node before it does its work.
- **Epilogue Action Set**. Java Actions in an Epilogue Action Set are executed as the node is being completed and before the process instance moves on to the next node. These Java Actions can be used to clean up or analyze values associated with the node after the intended work is finished.

- **Role Action Set**. Java Actions in the Role Action Set are executed after assignee resolution and before work items are created. These Java Actions can be used to dynamically manipulate the list of assignees for a User Task Node.
- **Timer Action Set**: These Java Actions associated with a timer are executed when that timer expires.
- **Error Action Set**. Java Actions in Error Action Set are second level Java Action. Every first level Java Action from the other Java Action Sets has an associated Error Action Set. An Error Action can be used for handling specific error situations in the execution of a process. Error Actions can be defined for entire process definitions, for individual nodes and for other Java Actions, i.e. when you want to react on errors occurring during the execution of another Java Action.
- **Compensate Action Set**. Java Actions in Compensate Action Set are second level Java Action. Every first level Java Action from the other Java Action Sets has an associated Compensate Action Set. Compensate Action Set are useful to undo successful changes to external systems when the transaction is being rolled back because of some error in a different part of the process.
- A special type of action can be activated as soon as an Administrator issues the command to abort, suspend or resume the processing of a process instance. Such actions are stored in either of the following Action Sets:
  - **onAbort Action Set**
  - **onResume Action Set**
  - **onSuspend Action Set**

  Java Actions belonging to on* Actions Sets are to be performed before the state of a process instance is changed. They can be defined for individual activities, i.e. for individual nodes in a process definition, or for an entire process definition.

6.3.2 Accessing Workflow Data Using the Server Enactment Context Interface

You can use Java Actions to manipulate process information by using the Server Enactment Context interface (`com.fujitsu.iflow.server.intf.ServerEnactmentContext`). It provides, for example, the following methods:

- `addAttachment()`
- `getAttachment()`
- `getProcessOwners()`
- `setProcessDescription()`

Refer to the API Javadoc for complete details on the available classes and methods.

To access workflow data using the Server Enactment Context:

1. When creating the class for your Custom Java Action Type:
   a) Import the ServerEnactmentContext interface.
   b) Make sure that one parameter of the method that you want to call from a Java Action is of type `ServerEnactmentContext`.
2. When defining the Java Action, you specify the values to pass to your method. Use the `sec` identifier as value for the `ServerEnactmentContext` parameter.

   At run time, Interstage BPM will pass the ServerEnactmentContext object to your method.

Example

Here is an example of a Java method you might write that needs to access workflow data. The bold text shows how to import and use the Server Enactment Context interface:

```
import com.fujitsu.iflow.server.intf.ServerEnactmentContext;
public class MyClass {
   public void reassignIfTooExpensive(ServerEnactmentContext sec,
      int amount) {
      String[] employee = {"Employee1", "Employee2"};
      String[] managers = {"Manager1", "Manager2"};
      if (amount <= 5000)
         sec.setActivityAssignees(employee);
      else
         sec.setActivityAssignees(managers);
   }
}
```

This is how you might define the Java Action. The bold text shows how to pass the context data to your method:

```
JavaAction[] myAction = MyActionSet.createJavaActions(1);
myAction[0].setActionDescription("Decide on purchase requisition");
myAction[0].setActionName("RoutePurchaseRequisition");
myAction[0].setClassName("MyPackage.MyClass");
myAction[0].setMethodName("reassignIfTooExpensive(ServerEnactmentContext,
 int)");
String[] args = new String[2];
args[0] = "sec";
args[1] = "uda.amount";
String params = Utils.combineParametersToXML(args);
myAction[0].setArgumentsUDANames(params);
MyActionSet.setJavaActions(myAction);
```

6.3.3 Assigning Prologue Actions to a User Task Node

This section gives you an example of how to assign Prologue Actions to a User Task Node. The Java Actions used are part of the example class `ComplexPlan`. Use `setClassName()` from the `JavaAction` interface to refer to the example class `ComplexPlan`.

In the example, Prologue Actions are defined that return the initial values for the User Defined Attributes (UDAs) `Qty` and `Price`. This means that you implement your own Java class and add its methods as Java Actions without using the Server Enactment Context API.

Note: When programming Java Actions, you specify its type only when you set it for a process definition, a node or another Java Action. The methods available from the `JavaAction` interface can be used for any type of Java Action, however, you need to be aware that the type of Java Action determines which methods are executed at runtime. Refer to the API Javadoc for more information.

To assign Prologue Actions to a User Task Node:

1. Add a User Task Node. The name of the first node in the process definition is protected final static String NODE_FILL_OUT_PR = "Fill out Purchase Request";

```
protected final static String NODE_FILL_OUT_PR =
   "Fill out Purchase Request";
```

```
Node fillOutNode = plan.addNode("NODE_FILL_OUT_PR",
   Node.TYPE_ACTIVITY);
fillOutNode.setRole("SampleGroup");
fillOutNode.setUpperLeftPoint(new Point(450, 40));
```

2. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet foPJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

3. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, three Java Actions are generated.

```
JavaAction[] foPJavaActions =
   foPJavaActionSet.createJavaActions(3);
```

4. Define the Java Actions.

   These are the Java Actions that set the initial quantity and price:

```
foPJavaActions[0].setActionDescription("Sets initial value for Qty");
foPJavaActions[0].setActionName("load initial qty");
foPJavaActions[0].setMethodName("getInitialQty");
foPJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
foPJavaActions[0].setReturnValueUDAName("Qty");
...
foPJavaActions[1].setActionDescription("Sets initial value for Price");
foPJavaActions[1].setActionName("load initial price");
foPJavaActions[1].setMethodName("getInitialPrice");
foPJavaActions[1].setClassName(CLASS_NAME_JAVA_ACTION);
foPJavaActions[1].setReturnValueUDAName("Price");
...

foPJavaActionSet.setActionSetDescription("Operations to set"
   + " initial values of UDAs Qty and Price");
foPJavaActionSet.setActionSetName("Qty and Price Setter");
```

   In the example, another Prologue Action is defined for creating an attachment. To handle errors that might occur, Error Actions are defined for that Java Action. For details refer to section *Dealing with Errors in Java Actions* on page 104.

5. Assign `JavaActionSet` to the prologue part of the User Task Node.

```
foPJavaActionSet.setJavaActions(foPJavaActions);
fillOutNode.setJavaActionSet(foPJavaActionSet,JavaActionSet.NODE_PROLOGUE);
```

A copy of the Java Actions is made and saved with the Action Set in the User Task Node. Further changes of the Java Action Set object will have no effect on the Java Actions that are already copied

into the User Task Node. If you want to change the User Task Node, you must modify the Java Action Set, and then assign the entire set again to the User Task Node.

6.3.4 Assigning Epilogue Actions to a User Task Node

Prerequisite: You have defined a User Task Node as explained in section *Assigning Prologue Actions to a User Task Node* on page 94.

In this example, `ComplexPlan` defines a Jave Action which returns the calculated price `Total` for the User Defined Attributes (UDAs) `Qty` and `Price`. The Java Action is added to the Epilogue Action Set of a User Task Node.

To add a Java Action to the Epilogue Action Set of a User Task Node:

1. Use `getJavaActionset()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet foEJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] foEJavaActions =
   foEJavaActionSet.createJavaActions(1);
```

3. Define the Java Action.

```
foEJavaActions[0].
   setActionDescription("Sets calculated total amount to UDA 'Total'");
foEJavaActions[0].setActionName("calculate Total");
foEJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
foEJavaActions[0].setMethodName("getTotal(float,int)");
foEJavaActions[0].
   setArgumentsUDANames("<E>uda.Price</E><E>uda.Qty</E>");
foEJavaActions[0].setReturnValueUDAName("Total");
foEJavaActionSet.
   setActionSetDescription("Operation to calculate total");
foEJavaActionSet.setActionSetName("Total Setter");
```

4. Assign `JavaActionSet` to the epilogue part of the User Task Node.

```
foEJavaActionSet.setJavaActions(foEJavaActions);
fillOutNode.setJavaActionSet(foEJavaActionSet,JavaActionSet.NODE_EPILOGUE);
```

6.3.5 Using Built-In Java Action Types

Java Actions are the workhorses of the process engine, and Interstage BPM provides many types of Java actions for all the kinds of data manipulation you might need to do at any point in a process.

Here is a partial list of the built-in Java Action Types:

- Evaluate a JavaScript. The JavaScript is an easy way to perform any kind of freeform data testing or manipulation that you might need to do in the process. Refer to section *JavaScript Java Actions* on page 98 for an example.

- Modify process settings. You can change a process priority, activity priority, the title, name, or description of a process
- Read and update process data. You can test and set user defined attributes in the process instance.
- Balance Workload. You have control over who is assigned to the activity, how activities are reassigned, check who performed an activity in the past, assign according to user relationships in the directory, assure that two activities are always done by different people, distribute activities evenly across a group of people, find out who started the process, escalating an activity to a specific set of people, and assure that the current workload is evenly distributed.
- Manipulate XML. Powerful commands exist to extract parts of XML data, update parts, search for data in XML, parse files into XML, and transform XML into other formats using standards like XPath and XSLT.
- Invoke business rules in compatible business rules engines. Refer to section *Rules Java Actions* on page 98 for more information.
- Send notifications and alerts. You can compose and send email messages to notify people of tasks or other occurrences in a process. Custom java actions can send text messages, pager messages, messages to social platforms like Twitter, Linked-In, Facebook, and other instance messaging services.
- Retrieve and update external databases. SQL commands can be used to select, update, insert, or delete rows in any relational database that can be accessed by a JDBC interface, even remote databases.
- Call web services. You can read data from just about any enterprise or cloud system, as well as update and manipulate their contents. Conforms to all the well know web service standards of HTTP, WSDL, SOAP, XML, XPath, XSD, WS-Security, WS-Reliability, WS-Transaction, and UDDI. Extensions offer ready access to REST-oriented web services using both XML and JSON transfer formats.
- Call remote commands. A convenient way to call any command line operation on any server on the network.
- Generic actions. Call any static method on any JAva class you include in the application. This opens the door to just about anything you need to do at any point in the process. Simply include the compiled Java class with the application, and it will be loaded and called automatically at the right time.

Example

The following example shows how to specify a method of the `IflowActions` class:

```
JavaAction[] myAction = MyActionSet.createJavaActions(1);
myAction[0].setActionDescription("Decide on purchase requisition");
myAction[0].setActionName("MakeChoice");
myAction[0].setClassName("com.fujitsu.iflow.actions.IflowActions");
myAction[0].setMethodName("makeChoiceAction(String, long,
ServerEnactmentContext)");
String[] args = new String[3];
...
```

For more information on defining Java Actions, refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93.

6.3.6 JavaScript Java Actions

Java Actions are flexible in that they allow you to do anything that Java can do, but they have the drawback that you must compile your Java into a class that is accessible to the process engine. A JavaScript Java Action is a type of Java Action that can evaluate JavaScript. In JavaScript, you can do everything that you could do in Java; only the JavaScript does not need to be compiled ahead of time. Because the JavaScript is kept in source form inside the process definition, there is no associated class file that must be carried around externally.

JavaScript uses a syntax very much like Java, but there are a few differences, such as dynamically typed variables that make sense in a scripting environment. JavaScript is actually an ECMA standard described in the document ECMA-262.PDF that is provided as a printable file with the software. The same Server Enactment Context API is available to the JavaScripts.

In addition, you might want to create your own custom JavaScript extensions to effect script behavior that you will use over and over in your specific situation.

Note: The size that a method used in a JavaScript may have is limited by the Java Virtual Machine (JVM). Currently, the method byte code size is limited to 65535 bytes (64 KBytes). If you use a method with a larger size, the JVM will throw an error and you have to reduce the size of the method before executing the JavaScript again.

Calling Java Methods in JavaScript

Standard JavaScript has the ability to call any Java method you wish. It can even construct Java objects, and call methods on them, passing other Java objects as parameters if you wish.

Example of a call to a static method in ScriptPlugIn.class:

```
Packages.ScriptPlugIn.createFile("test.txt");
```

Example of a call to a member method in JDBCPlugIn.class:

```
x = new Packages.JDBCPlugIn();
var customer = uda.Customer;
var finish = x.findCustomer(customer);
```

6.3.7 Rules Java Actions

A Rules Java Action is a special type of Java Action that acts as an interface to a Rules Engine. This interface allows you to use advanced process logic in your processes at every point in the process where you could attach a Java Action Set.

You can apply these rules using the following Interstage BPM methods along with any of your own custom methods:

- `getCurrentProcessId`
- `getCurrentActivityId`
- `getCurrentActivityName`
- `getCurrentActionType`
- `getProcessDefinitionName`
- `getProcessDefinitionIdentifier`
- `getOwners`
- `getInitiator`
- `getMembers`

- `getActivityAssignees`
- `getActivityIteratorIndex`
- `getProcessAttribute`*
- `getProcessPriority`
- `getActivityPriority`
- `getProcessTitle`
- `getProcessOwners`
- `getProcessInitiator`
- `getProcessDescription`
- `getProcessName`
- `setProcessName`
- `getGroupMembers`
- `getActivityActor`
- `setProcessOwners`
- `setActivityAssignees`
- `setProcessAttribute`*
- `setProcessPriority`
- `setActivityPriority`
- `setOwners`
- `setProcessTitle`
- `setProcessDescription`

*For the `getProcessAttribute` and `setProcessAttribute` methods, the user-defined process attributes that you are getting or setting must be defined on the process definition on which the Rules Java Action is attached.

ILOG Configuration

Note: To use the ILOG Rules Engine, you must have the ILOG application installed on your system and copy its `rulesall.jar` file to the location as specified in *Configuring Java Actions, Agents and JavaScripts for External Applications* on page 88.

This section discusses the development of a business rules (`*.ilr`) file in the ILOG Rules Engine. It provides instructions for integrating Interstage BPM methods in ILOG but not for using ILOG. For instructions in using ILOG, please refer to your ILOG documentation.

With a business rules file you can create a Rules Java Action to implement the advanced process logic of ILOG.

You can write your business rules file in ILOG by importing the Rules Engine Interface class into ILOG. This class is called `RulesEngineIntf.class`, and it can be found in the `com.fujitsu.iflow.rules` package. Once you import this class, it will appear in your ILOG editor, and you will see all the Interstage BPM methods (mentioned above) under it.

If you want to use ILOG Rules in Java Actions, the following configuration step is required:

On the Interstage BPM Server

- Specify a rule file name by using the first parameter of InputUDAList.

- If in SaaS mode, or if only the file name was specified, copy the file to `<DMSRoot>/apps/<application ID>/dms/Attachments`.
- If in non-SaaS mode the complete file path was specified, copy the file to that location.

Blaze Advisor Configuration

If you want to use Blaze Advisor rules in Java Actions, the following configuration steps are required.

1. If it is the first time that you use Blaze Advisor with Interstage BPM, copy the following JAR files with the license directory from the Blaze Advisor `lib` directory into the `<Interstage BPM Domain>/lib` directory. For example, Blaze Advisor `lib` directory for WebLogic is `C:/Oracle/Middleware/user_projects/domains/base_domain/lib`:
   - `AdvCommon.jar`
   - `Advisor.jar`
   - `AdvisorSvr.jar`
   - `collections.jar`
   - `InnovatorRT.jar`
   - `jaxen.jar`
   - `Ndkjc-2.1A-bin.jar`
   - `OROMatcher.zip`
   - `saxpath.jar`
   - `iFlow.jar`. This file is located in the `<engine directory>/client/lib` folder.
2. On the Interstage BPM Server
   - Specify a server configuration file name by using the first parameter of InputUDAList.
   - If in SaaS mode, or if only the file name was specified, copy the file to `<DMSRoot>/apps/<application ID>/dms/Attachments`.
   - If in non-SaaS mode the complete file path was specified, copy the file to that location.
3. Include the Blaze license file in your CLASSPATH.

   If you are using Interstage BPM for WebLogic, update the script `setDomainEnv.cmd` or `setDomainEnv.sh` located in the `<Interstage BPM Domain>/bin` directory, for example, the path for WebLogic is `C:/Oracle/Middleware/user_projects/domains/base_domain/bin`:

   On **Windows**:

   Delete the line:

   ```
   set
   CLASSPATH=%PRE_CLASSPATH%;%WEBLOGIC_CLASSPATH%;
   %POST_CLASSPATH%;%WLP_POST_CLASSPATH%
   ```

   Add the following lines:

   ```
   set BLAZE_LICENSEPATH=<path to your license file>
   set
   CLASSPATH=%PRE_CLASSPATH%;%WEBLOGIC_CLASSPATH%;
   %POST_CLASSPATH%;%WLP_POST_CLASSPATH%;%BLAZE_LICENSEPATH%
   ```

   On **UNIX**:

Delete the line:

```
CLASSPATH="{PRE_CLASSPATH}${CLASSPATHSEP}
${WEBLOGIC_CLASSPATH}${CLASSPATHSEP}
${POST_CLASSPATH}${CLASSPATHSEP}
${WLP_POST_CLASSPATH}"
```

Add the following lines:

```
BLAZE_LICENSEPATH=<path to your license file>
export $BLAZE_LICENSEPATH
CLASSPATH="{PRE_CLASSPATH}${CLASSPATHSEP}
${WEBLOGIC_CLASSPATH}${CLASSPATHSEP}
${POST_CLASSPATH}${CLASSPATHSEP}
${WLP_POST_CLASSPATH}${CLASSPATHSEP}
${BLAZE_LICENSEPATH}"
```

4. Stop and start the Interstage BPM Server.

   For instructions, refer to the *Interstage Business Process Manager Server Administration Guide*.

Decision Tables Java Action

Decision Tables is a feature built into Interstage BPM that allows designing of advanced rules for decision making without programming. For details of how to create rules using Decision Tables, refer section *Decision Tables* on page 189.

To use a decision table in a process definition, a predefined Java Action to evaluate Decision Tables (called a Decision Table Java Action, a type of Rules Java Action) is provided. For further details refer section *Using a Decision Table within a Process Definition* on page 191, the *Interstage BPM Studio User's Guide* and/or the *Interstage BPM Console Online Help*.

6.3.8 Activity Actors and Relationships

Activity Actors

When an activity is assigned to a group, any user in that group can perform that activity. For instance, if an activity is assigned to the group called 'managers', any user who is a member of the 'managers' group will be allowed to access and complete that activity. But only one particular user will actually complete the activity. That user can be referred to as the actor for that activity.

After an activity has been performed, certain later operations may depend upon who the activity actor was. For instance, consider the activity 'Record customer issue' assigned to the 'customer support executive' group. One would want the same user to be assigned a later activity 'Escalate issue' in that process instance. This is accomplisehd by assigning the later activity, to the actor of the earlier activity.

Determining Activity Actors

The `ServerEnactmentContext` interface of the `com.fujitsu.iflow.server.intf` package contains the `getActivityActor()` method to determine activity actors.

The `getActivityActor()` method gets the actor of a completed User Task node using the name of the User Task node as a parameter. Voting User Task node is not supported by this method.

Note: To determine the activity actors using the name for a User Task node, please make sure that there is only one node in the process with that name.

The following sample gets the actor of 'Activity A' and assigns the current activity to that actor.

```
public void assignActivityActor(ServerEnactmentContext sec){
 String[] actor = new String[1];
 actor[0] = sec.getActivityActor("Activity A");
 sec.setActivityAssignees(actor);
}
```

Note: You can call this method in a custom Java Action Type (refer *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93) or as part of a JavaScript.

Refer to the API Javadoc for details of this method.

Relationships

Certain workflows may demand finding out the hierarchical relationships for an activity actor or a UDA value. For example, the UDA 'Company Name' may have a value 'Fujitsu', and one might need to find the 'account executive' for the company 'Fujitsu', so only that account executive (and not any other) may be assigned a particular activity. Or, an activity may have an actor called 'Jim', and the process might need the 'manager' for Jim assigned a particular activity.

Determining Relationships

The `ServerEnactmentContext` interface of the `com.fujitsu.iflow.server.intf` package contains the `resolveRelationship()` method to determine relationships.

The `resolveRelationship()` method returns a target value using a source value and a relationship as parameters. For this method to work, mappings of source values, relationships, and target values need to be pre-defined, so that this method can refer to the mappings and return appropriate values.

SourceValue	Relationship	TargetValue
`Jim`	`manager`	`Robert`
`Jim`	`assistant`	`Arthur`
`Fujitsu`	`executive`	`Bob`

For example, `resolveRelationship("assistant", "Jim")` returns the value `Arthur`.

SourceValue-Relationship-TargetValue mappings should be stored as user profiles in the Directory Service or local user store, where the source value is a user ID, relationship is a user attribute name, and target value is a user attribute value. To create a user profile, refer the `DirectoryServices` interface of the `com.fujitsu.iflow.model.workflow` package in the API Javadoc.

Note that SourceValues can be both human as well as non-human (such as company name or group name). User profiles of such non-human objects also need to be added to the directory server or local user store, taking care that such user profiles cannot be used for the purpose of logging in.

The following sample gets the actor of 'Activity A', gets the manager of that actor, and assigns the current activity to this manager.

```
public void assignManagerOfActivityActor(ServerEnactmentContext sec){
 String[] managers =
sec.resolveRelationship("Manager",sec.getActivityActor("Activity A"));
 sec.setActivityAssignees(managers);
}
```

Note: You need to either call this method in a Java Action defined by you (refer *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93) or as part of a JavaScript.

Refer to the API Javadoc for details of this method.

6.3.9 Using Error Java Actions

Error Java Actions can be used to handle specific errors and to determine the behavior of a process instance in case an error occurs. If you do not define any error handling, a process instance will go into error state as soon as an exception is thrown: e.g. if the starting of a Remote Subprocess fails, if an email cannot be sent, etc.

Error Java Actions can be defined on different levels:

- On **Process Definition level**: The Error Action Set will be executed in case of any error, independent of the activity in which an error occurs and independent of the severity of an error. Error Action Set defined on process definition level will be executed immediately before the process instance will go into error state. Note that such Error Actions cannot influence the behavior of the process instance. Error Actions on process definition level may, for example, be used for sending a notification email or for writing additional information into a log file.
- On **Node level** (for Remote Sub-Process Nodes only): An Error Action Set on this level will become active if the remote subprocess fails to start. Refer to section *Error Handling for Remote Subprocesses* on page 183 for an example.
- On **Java Action level**: An Error Action Set on this level will be executed when an exception is thrown in the related first level Java Action. Error Java Actions can be defined for any first level Java Actions (e.g. Prologue Action Set, Epilogue Action Set, etc.). Error Java Actions are themselves second level Java Actions, and Error Java Actions can not be assigned to second level Java Actions. Assigning Error Actions to Java Actions is described in section *Dealing with Errors in Java Actions* on page 104.

Below you find an example of how to assign an Error Java Action to a process definition. The Java Actions used are part of the example class `ComplexPlan`.

To define an Error Action Set on process definition level:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet plErrorJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] plErrorJavaAction =
   plErrorJavaActionSet.createJavaActions(1);
```

3. Define the Java Action Set.

```
plErrorJavaAction[0]
    .setActionName("WriteLogEntryWithException");
plErrorJavaAction[0]
    .setActionDescription("Writes a log entry with exception");
plErrorJavaAction[0]
    .setMethodName("writeLogEntryWithException(String,ServerEnactmentContext)");
plErrorJavaAction[0]
    .setClassName(CLASS_NAME_JAVA_ACTION);
plErrorJavaAction[0]
```

```
    .setArgumentsUDANames("<E>uda.ProcessLevelErrorLogEntry</E><E>sec</E>");
plErrorJavaActionSet.setJavaActions(plErrorJavaAction);
```

4. Copy the contents of this `JavaActionSet` to the process definition's Error Action Set.

```
plan.setJavaActionSet(plErrorJavaActionSet,
    JavaActionSet.PLAN_ERROR);
```

6.3.10 Dealing with Errors in Java Actions

When an error occurs during the execution of a Java Action, an exception is thrown. Interstage BPM allows you to define your own error handling for erroneous Java Actions. In this way, you can prevent that a process instance goes into error state when an exception is thrown.

In addition, you can define a set of actions for any Java Actions in order to perform a "cleanup" as the transaction is being rolled back. This allows you to perform general actions (e.g. sending notification emails in any error case), or executing some specific actions before setting the process instance to error state.

Pay attention to the whether the action is a 'first level' action or a 'second level' action. First level actions can have error and/or compensate actions. These error and/or compensate actions are themselves second level actions. There are no third level actions: second level actions can not have their own error or compensate actions. Any exceptions thrown from second level actions will always cause the process to enter error state.

Imagine that you have an action set with five Java Actions in it. Upon execution of the set, and exception is thrown from the fourth action. In this case, three Java Actions have successfully completed, but the fourth Java Action is going to cause the translaction to roll back. The Error Actions on the fourth Java Action could "catch" or handle the exception, and prevent it from causing the transaction to abort. If Error Actions do catch the exception, then execution will continue to the fifth Java Action. However, if the Error Actions for the fourth Java Actin do not catch the exception, then the transaction will roll back, and the process will enter the error state. Remember that Java Actions 1 thru 3 had already completed successfully, so they are given an opportunity to be "compensated". First, any Compensate Actions on Java Action 3 will be run, then Compensate Actions on Java Action 2 will be run, and finally the Compensate Actions for Java Action 1 will be run, before the transaction is terminated and rolled back. Keep in mind that Compensate Actions are run only for Java Actions that have already completed successfully, but for which a later Java Action is causing the transaction to be rolled back.

Interstage BPM provides the following options to handle exceptions from first level Java Actions:

- **Compensate Actions**: Compensate Actions are used to clean up the system and to ensure a consistent state of all systems involved in a transaction, e.g. external databases or mail servers. Compensate Actions are particularly useful whenever external systems are involved.

  If you do not define any error handling for a Java Action, the following happens: When an exception is thrown in this Java Action, the transaction will be rolled back. A rollback, however, is only possible for changes in the Interstage BPM Application Server context. Any transactions in external systems cannot be rolled back, for example, if a row has been added to an external database. Therefore, it is sometimes necessary to manually clean up external systems to ensure a consistent state of all systems used in the transaction. Optionally, you can use Compensate Action Sets.

  You can specify a set of Compensate Actions for every Java Action. You can use a Compensate Actions e.g. for removing a newly added row in a database or for sending out an additional email. If an exception occurs in a regular Java Action Set, all Compensate Actions defined for all Java

Actions that have been successfully executed before the exception was thrown, are invoked in reverse order.

Note: You cannot put a Compensate Action on any second level action. If a Compensate Action throws an exception, the process instance will immediately go to error state, and the execution of remaining Compensate Actions will be aborted.

Refer to the sample below for more information.

- **Error Actions**: Error Actions will be executed when an exception is thrown from the Java Action they are assigned to. Error Actions can be specified for any first level Java Action. Note that you cannot define an Error Action for any second level Java Action.

  Refer to section *Using Error Java Actions* on page 103 and the sample below for more information.

  The Continue Setting

  Error Actions allow for determining whether the error will be caught and process execution will continue or not. This behavior is defined by the Continue Setting "**catchException**":

  If set to `true`, the exception gets caught and the process instance continues its execution after the defined Error Actions have been executed.

  If set to `false`, any Compensate Action defined will be executed and afterwards the process instance will go into error state.

  In case several Error Actions are defined that have different "catchException" settings, `false` overrides `true` and the process instance will go into error state as soon as one Error Action has set the "catchException" to `false`.

  The Exception Class

  In addition to the Continue Setting, Error Actions on Node level and on Java Action level allow for determining on which exceptions the Error Action reacts: You can specify a concrete exception class (default: `java.lang.Exception`. In this case, an Error Action will only be executed when the exception thrown is an instance or a subclass of this specified exception class. For example:

  You specify the `java.io.FileNotFoundException`) class so that the Error Action will only be considered in case the occurred exception is an instance or a subclass of this class.

Note: In case an Error Java Action throws an exception, the transaction will be rolled back instantly and the process instance will go into error state. No Error Action can be defined for such a case.

For each first level Java Action, you can define Error Actions, as well as Compensate Actions. Refer to section *Java Action Structure and Execution Plan in Case of Errors* on page 107 for additional information.

Defining and Assigning Compensate Java Actions to a Java Action

The sample below defines a Compensate Java Action that is to be executed in case the execution of another Java Action fails: The administrator is to be notified by email in case the retrieval of the initial quantity fails. This retrieval is defined in another Java Action that is executed as Prologue Action. You can find the whole code of this sample in the `ComplexPlan.java` sample.

To define and assign a Compensate Java Action Set:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet compensateJavaActionSet =
    WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`. In the following example, one Java Action is generated.

```
JavaAction[] compensateJavaAction =
    compensateJavaActionSet.createJavaActions(1);
```

3. Define the Java Action. Note that the `sendEmail()` method is defined in the `SampleJavaActions` sample.

```
compensateJavaAction[0].setActionName("Send a notification mail");
compensateJavaAction[0]
    .setActionDescription("Send notification mail to administrator");
compensateJavaAction[0].setMethodName
    ("sendEmail(String, String, String, String, String, String,
      ServerEnactmentContext)");
compensateJavaAction[0].setClassName(CLASS_NAME_JAVA_ACTION);
compensateJavaAction[0].setArgumentsUDANames
     ("<E>uda.MailTo</E><E>uda.MailFrom</E>
       <E>uda.MailCc</E><E>uda.MailBcc</E><E>uda.MailSubject</E>
       <E>uda.MailBody</E><E>sec</E>");
```

4. Add the `compensateJavaAction` to the Java Action that retrieves the initial quantity. This Java Action is part of the Prologue Action Set. You can find the entire definition in the `ComplexPlan.java` sample.

```
....
JavaActionSet foPJavaActionSet = WFObjectFactory.getJavaActionSet();
JavaAction[] foPJavaActions = foPJavaActionSet.createJavaActions(3);
foPJavaActions[0]
        .setActionDescription("Sets initial value for Qty");
foPJavaActions[0].setActionName("load initial qty");
foPJavaActions[0].setMethodName("getInitialQty");
foPJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
foPJavaActions[0].setReturnValueUDAName("Qty");
foPJavaActions[0].setJavaActionSet(compensateJavaActionSet,
        JavaActionSet.ACTION_COMPENSATE);
....
```

Defining and Assigning Error Actions to a Java Action

The sample below defines an Error Action Set that is to be executed in case the execution of a related Java Action fails: An entry is to be written to the log file and the processing of the process instance is to be continued in case an attachment cannot be added. The method for adding an attachment is defined in the in the `SampleJavaActions` sample. You can find the whole code of this sample in the `ComplexPlan.java` sample.

To define and assign an Error Action Set:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet errorJavaActionSet =
    WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`. In the following example, two Java Actions are generated.

```
JavaAction[] errorJavaActions =
    errorJavaActionSet.createJavaActions(2);
```

3. Define the Java Actions. Note that the `addAttachment()` method is defined in the `SampleJavaActions` sample.

```
errorJavaActions[0]
    .setActionDescription("Writes a log entry with exception");
errorJavaActions[0]
    .setMethodName("writeLogEntryWithException(String,ServerEnactmentContext)");
errorJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
errorJavaActions[0]
    .setArgumentsUDANames("<E>uda.ProcessLevelErrorLogEntry</E><E>sec</E>");
errorJavaActions[0].setCatchException(true);

errorJavaActions[1].setActionName("Continue process");
errorJavaActions[1].setActionDescription("Catches FileNotFoundExceptions" +
    "and continues the execution of the process");
errorJavaActions[1].setMethodName("noop()");
errorJavaActions[1]
    .setClassName("com.fujitsu.iflow.actions.IflowActions");
errorJavaActions[1]
     .setEditorClassName("com.fujitsu.iflow.actions.IflowActions");
errorJavaActions[1]
      .setCatchException(true);
```

4. Add the above Error Action Set to the Java Action that is to add the attachment. This Java Action is part of the Prologue Action Set. You can find the entire definition in the `ComplexPlan.java` sample.

```
....
foPJavaActions[2].setJavaActionSet(errorJavaActionSet,
    JavaActionSet.ACTION_ERROR);
...
```

6.3.11 Java Action Structure and Execution Plan in Case of Errors

This section describes the possible structure of Java Action Sets including Compensate and Error Action Sets. In addition, for any error that occurs in a Java Action, it shows which actions will be executed in which order.

All first level Java Actions have an Error Action set for handling execptions thrown, and have a Compensate Action Set for undoing what they did in the case that the entire transaction is being rolled back.

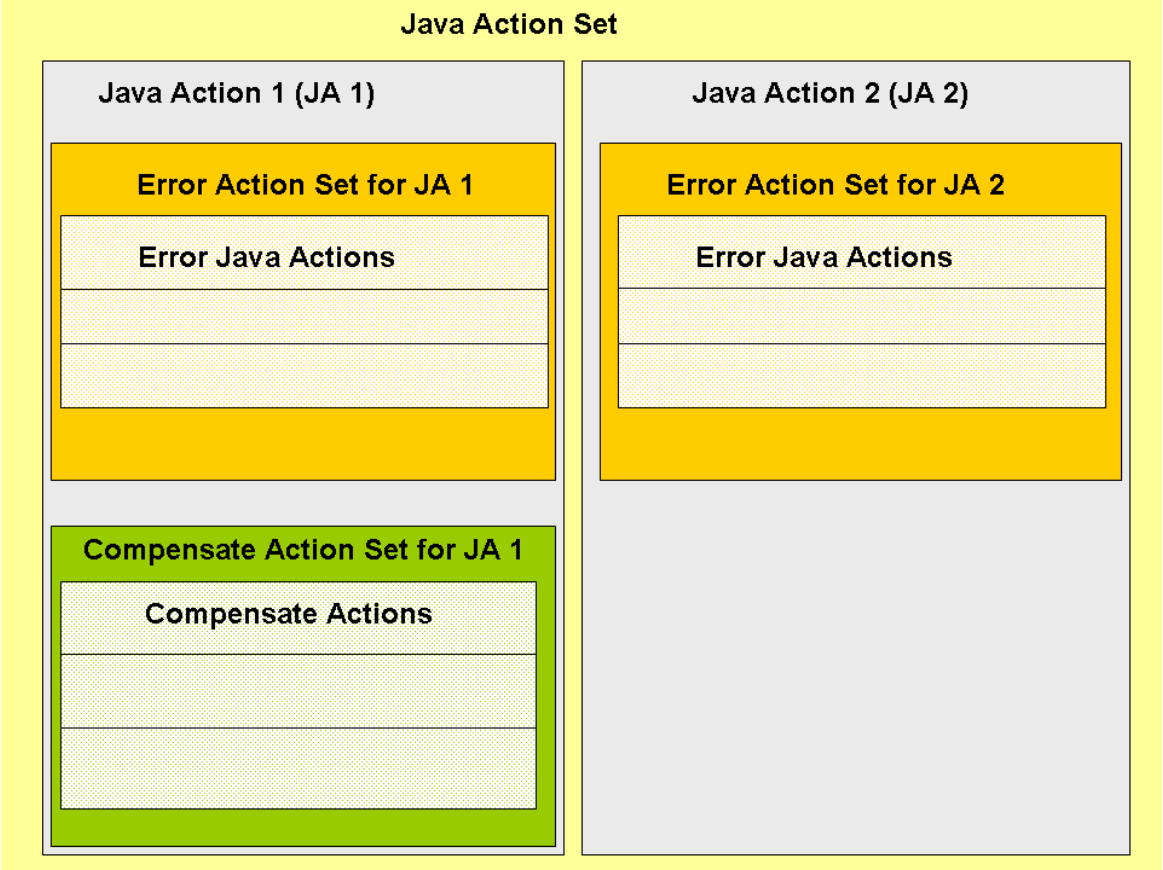

Figure 15: Java Action Set Structure

Example of a Regular Java Action Set

Assume you have defined a Prologue Action Set comprising two Java Actions (JA_1 and JA_2).

- **JA_1**: An Error Action Set with one Error Java Action (EJA_JA_1) and a Compensate Action Set with two Compensate Java Actions (CJA_1_JA_1 and CJA_2_JA_1) have been defined.

- **JA_2**: An Error Action Set with two Error Java Actions (EJA_1_JA_2 and EJA_2_JA_2) has been defined, but no Compensate Action Set is available:

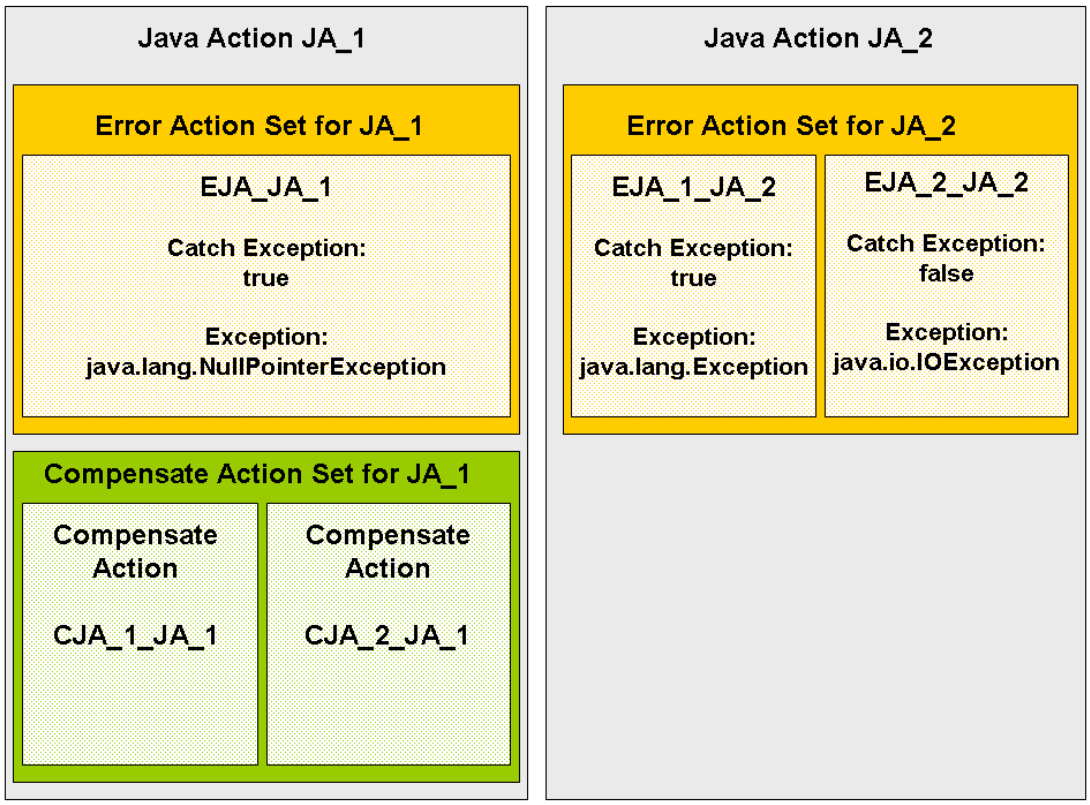

Figure 16: Regular Java Action Set

Execution Plan in Case of Errors

The table below lists several error situations and explains which Java Actions will be executed in case of an error:

Situation	Result
JA_1 throws a `FileNotFoundException`.	Since no Error Action is defined for this situation, the process instance will go into error state.
JA_1 throws a `NullPointerException`.	The exception gets caught and EJA_JA_1 will be executed. The process instance continues, i.e. JA_2 will be executed and afterwards the transaction will be committed.
JA_1 executes successfully. JA_2 throws a `NullPointerException`.	EJA_1_JA_2 will be executed, because `NullPointerException` is a subclass of `Exception`. The exception will be caught, the transaction will be committed and the process instance continues.

Situation	Result
JA_1 executes successfully. JA_2 throws a `FileNotFoundException`.	EJA_1_JA_2 and EJA_2_JA_2 will be executed, because `FileNotFoundException` is a subclass of `Exception` and of `IOException`. However, the exception will not be caught due to the `Catch Exception` being set to `false` in EJA_2_JA_2. This setting overrides the EJA_1_JA_2 setting. Therefore, after execution of the two Error Actions, the Compensate Actions CJA_1_JA_1 and CJA_2_JA_1 will be executed before the process instance goes into error state.
JA_1 and JA_2 are executed successfully.	No Error or Compensate Actions are executed. The process instance remains in its regular state.

Refer to sections *Dealing with Errors in Java Actions* on page 104 and *Using Error Java Actions* on page 103 for additional information.

6.3.12 Using onAbort, onSuspend, onResume Java Action Sets

The onSuspend, onResume, and onAbort Action Sets can be used to execute specific actions before the state of a process instance is changed because an administrator has called the suspend, resume, or abort command.

Similar to other Java Action Sets, the onAbort, onResume, and onSuspend Action Sets can contain several Java Actions. Each of these sets can be defined on node level as well as on process definition level. Node level Action Sets will always be executed first, but only if the respective activity is active when the abort, suspend, resume command has been issued. The process definition level Action Sets will be executed independent of the activities that are active when the command is being issued.

Below you find an example of how to assign an onAbort Java Action to a process definition. The Java Actions used are part of the example class `ComplexPlan`. Use `setClassName()` from the `JavaAction` interface to refer to the example class `ComplexPlan`.

To design and assign an onAbort Action Set on process definition level:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
onAbortJavaActionSet = WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] onAbortJavaAction =
     onAbortJavaActionSet.createJavaActions(1);
```

3. Define the Java Action Set.

```
onAbortJavaAction[0].setActionName("WriteLogEntry");
onAbortJavaAction[0]
    .setActionDescription("Writes a log entry if process is aborted");
onAbortJavaAction[0].setMethodName("writeLogEntry(String)");
onAbortJavaAction[0].setClassName(CLASS_NAME_JAVA_ACTION);
```

```
onAbortJavaAction[0]
    .setArgumentsUDANames("<E>uda.OnAbortLogEntry</E>");
onAbortJavaActionSet.setJavaActions(onAbortJavaAction);
```

4. Copy the actions from the `JavaActionSet` to the process definition's onAbort Action Set

```
plan.setJavaActionSet(onAbortJavaActionSet,
                      JavaActionSet.PLAN_ABORT);
```

6.4 Advanced Filtering & Sorting API

You can filter lists of process definitions, process instances and work items on a range of fields like the process definition name, process initiator, work item name, or any other. With this feature, you can retrieve lists that match certain filtering criteria like: get all process instances initiated by "userX", or get all work items originating from process instances with title "Purchase Order". You can also filter on multiple fields. A table of the fields and the lists with which they are valid is provided in the next section.

6.4.1 Interface WFObjectList (Package com.Fujitsu.iflow.model.workflow)

This interface offers constants that identify the fields that can be used for filtering/sorting lists along with the filters for which they are valid, for example:

Filter	listField

`Filter.AllPlans`
`Filter.MyPlans`
	`LISTFIELD_APPLICATION_IDENTIFIER`
	`LISTFIELD_PLAN_ID`
	`LISTFIELD_PLAN_NAME`
	`LISTFIELD_PLAN_OWNER`
	`LISTFIELD_PLAN_STATE`
	`LISTFIELD_PLAN_IDENTIFIER`

Filter	listField
Filter.AllProcesses Filter.AllActiveProcesses Filter.MyProcesses Filter.MyActiveProcesses Filter.MyInactiveProcesses Filter.AllInactiveProcesses Filter.AllProcessesInErrorState	LISTFIELD_APPLICATION_IDENTIFIER LISTFIELD_PLAN_ID LISTFIELD_PLAN_NAME LISTFIELD_PLAN_IDENTIFIER LISTFIELD_PROCESSINSTANCE_ID LISTFIELD_PROCESSINSTANCE_STATE LISTFIELD_PROCESSINSTANCE_INITIATOR LISTFIELD_PROCESSINSTANCE_PRIORITY LISTFIELD_PROCESSINSTANCE_NAME LISTFIELD_PROCESSINSTANCE_TITLE LISTFIELD_PROCESSINSTANCE_PARENTID LISTFIELD_PROCESSINSTANCE_CREATEDTIME LISTFIELD_PROCESSINSTANCE_CLOSEDTIME LISTFIELD_PROCESSINSTANCE_OWNER LISTFIELD_PROCESSINSTANCE_DUEDATE LISTFIELD_PROCESSINSTANCE_DESCRIPTION
Filter.AllWorkItems Filter.AllInactiveWorkItems Filter.MyWorkItems Filter.MyAcceptedWorkItems Filter.MyActiveWorkItems Filter.MyDeclinedWorkItems	LISTFIELD_APPLICATION_IDENTIFIER LISTFIELD_PLAN_ID LISTFIELD_PLAN_NAME LISTFIELD_PLAN_IDENTIFIER LISTFIELD_PROCESSINSTANCE_ID LISTFIELD_PROCESSINSTANCE_STATE LISTFIELD_PROCESSINSTANCE_NAME LISTFIELD_PROCESSINSTANCE_TITLE LISTFIELD_PROCESSINSTANCE_INITIATOR LISTFIELD_PROCESSINSTANCE_PRIORITY LISTFIELD_PROCESSINSTANCE_CREATEDTIME LISTFIELD_PROCESSINSTANCE_DESCRIPTION LISTFIELD_WORKITEM_ID LISTFIELD_WORKITEM_ASSIGNEE LISTFIELD_WORKITEM_STATE LISTFIELD_WORKITEM_CREATEDTIME LISTFIELD_WORKITEM_DUEDATE LISTFIELD_WORKITEM_PRIORITY LISTFIELD_WORKITEM_NAME LISTFIELD_WORKITEM_ACTIVITYINSTANCEID

Filter	listField
Filter.AllArchivedPlans	LISTFIELD_PLAN_ID LISTFIELD_PLAN_NAME
Filter.AllArchivedProcesses	LISTFIELD_PLAN_ID LISTFIELD_PROCESSINSTANCE_ID LISTFIELD_PROCESSINSTANCE_NAME
Filter.MyFutureWorkItems Filter.AllFutureWorkItems	LISTFIELD_APPLICATION_IDENTIFIER LISTFIELD_PLAN_ID LISTFIELD_PLAN_NAME LISTFIELD_PLAN_IDENTIFIER LISTFIELD_PROCESSINSTANCE_ID LISTFIELD_PROCESSINSTANCE_STATE LISTFIELD_PROCESSINSTANCE_NAME LISTFIELD_PROCESSINSTANCE_TITLE LISTFIELD_PROCESSINSTANCE_INITIATOR LISTFIELD_PROCESSINSTANCE_PRIORITY LISTFIELD_PROCESSINSTANCE_CREATEDTIME LISTFIELD_PROCESSINSTANCE_DESCRIPTION LISTFIELD_WORKITEM_ID LISTFIELD_WORKITEM_ASSIGNEE LISTFIELD_WORKITEM_PRIORITY LISTFIELD_WORKITEM_NAME LISTFIELD_WORKITEM_ACTIVITYINSTANCEID
Filter.MyCompletedWorkItems	LISTFIELD_APPLICATION_IDENTIFIER LISTFIELD_PLAN_IDENTIFIER LISTFIELD_PROCESSINSTANCE_ID
Filter.AllCompletedWorkItems	LISTFIELD_APPLICATION_IDENTIFIER LISTFIELD_WORKITEM_ASSIGNEE LISTFIELD_PROCESSINSTANCE_OWNER LISTFIELD_PROCESSINSTANCE_STATE LISTFIELD_WORKITEM_CREATEDTIME LISTFIELD_PLAN_IDENTIFIER LISTFIELD_PROCESSINSTANCE_ID

The following constants denote sorting order:

- SORTORDER_ASCENDING
- SORTORDER_DESCENDING

The following constants denote SQL relational operands:

- `SQLOP_EQUALTO`
- `SQLOP_GREATERTHAN`
- `SQLOP_LESSTHAN`
- `SQLOP_GREATERTHANOREQUALTO`
- `SQLOP_LESSTHANOREQUALTO`
- `SQLOP_NOTEQUALTO`
- `SQLOP_LIKE`
- `SQLOP_NOTLIKE`
- `SQLOP_IN`

You can find examples using filters in the following sections: *Listing Work Items* on page 80, *Listing Process Definitions* on page 227, and *Listing Process Instances* on page 231.

6.4.2 Methods for Filtering and Sorting

The `WFObjectList` interface provides the following methods for filtering and sorting:

- `addFilter(int, String, String)`: Adds a field-based filtering criteria for the list of objects to be retrieved. The values passed in the parameters conceptually translate to "where <actual-column-name-of-the-listField> <actual-value-of-the-SQL-operator> <value>".
- `addFilter(String, String, String, String)`: Adds a UDA-based filtering criteria for the list of objects to be retrieved. The values passed in the parameters conceptually translate to "where <udaName> <actual-value-of-the-SQL-operator> <value>".
- `addSortOrder(int, boolean)`: Adds a field-based sort order for the list of objects to be retrieved.
- `addSortOrder(String, String, boolean)`: Adds a UDA-based sort order for the list of objects to be retrieved.
- `getNextBatch[]`: Returns an array of list objects that are next in sequence. The returned array size is equal to `batchSize`. It may be less than `batchSize` for the last batch. The call returns `null` if there are no more elements in the list to return.

For details, refer to the API Javadoc.

6.4.3 Interface Semantics

The following sample represents the proper order in which the methods of this interface are to be invoked:

```
wfol = WFObjectFactory.getWfObjectList(session);
wfol.addFilter(…)
wfol.addSortOrder(…)
Object[] listElements = null;
int batchSize = 10;
wfol.openBatchedList(…)
while ((listElements = wfol.getNextBatch(batchSize)) != null) {
   // process elements in listElements array
}
```

Make sure that the `openBatchedList()` call follows `addFilter()` and/or `addSortOrder()` call(s) and precedes `getNextBatch()` calls. If the methods are invoked in any other order, the behavior of the `WFObjectList` is undefined.

6.4.4 Identifying UDAs to be Used With Lists

You can filter lists of process definitions, process instances and work items on values of User Defined Attributes (UDAs). Users can retrieve lists which match filtering criteria based on values of UDAs, for example "get all work items where transaction_amount > 5000.*". To do this, make sure, that the `WorkListUDA` flag is set on the UDA. This can be accomplished when you design the process definition by using `markAsWorkListUDA()` from the `DataItemRef` interface. Refer to section *Worklist UDAs* on page 133 for more information.

If UDAs that are not identified as described above are used to filter and sort lists, the behavior of the list will be undefined.

6.4.5 Note on Batching

You can retrieve lists in batches. If there were a large number of records in the database, opening a list would take a long time even if you needed only a few items of the list. You can use `openBatchedList` and `getNextBatch(int howMany)` to retrieve lists in batches. This batching works in conjunction with the filtering and sorting described here; i.e., Interstage BPM will retrieve a batch of the filtered and sorted list. However, the instructions provided in *Interface Semantics* on page 114 must be followed to obtain the desired result.

Note: If filtering or sorting is used, the relationship between the speed of retrieving a batch and the speed of retrieving the full list depends on the distribution of the values of the field/UDA used for filtering/sorting. The better the distribution of the values, the more advantage is derived from using batching.

For example if a list of process instances is sorted by process name, and if all the process instances have the same name, retrieving a batch internally degenerates to retrieving the full list. On the other hand if there are process instances with most process instances having different names, then the full advantage of using batching is derived.

6.4.6 Notification

Notification will be supported only in simple-list cases, i.e. without batch, filters or sort orders. If the list is opened with notification the following methods will throw a ModelException when invoked.

- `addFilter()`
- `addSortOrder()`
- `getNextBatch()`

6.5 Text-based Searching for Process Instance

You can search for a process instance that includes the specified text string in the selected fields like process instance name, description, comments, and user defined attributes. To search a particular process instance, you need to select the fields and specify the text to be searched in these selected fields. For example, if you specify the text as "sample" and select the search fields as process instance name and description, then it will search for the search string "sample" in the selected fields of all the process instances. A search result is retrieved as an array of matching process instances.

Searching rules are as given below:

- Searching is not case sensitive.
- The search string can contain only following special characters:
  - **Underscore (_)**: The underscore (_) character is treated as a literal character.

- **Asterisk (*)**: The asterisk (*) character will be treated as wildcard character. It can be substituted for any number of any characters. For example, the search string "`*id`" will match the words that contain "id", such as "`id`", "`userid`", "`PIid`","`012id`".
- **Question mark (?)**: The question mark (?) can be substitued for any one character. For example, the search string "`?id`" will match the words that contains any one character followed by "`id`", such as "`aid`", "`bid`", "`0id`".

If you specify any other special characters then they will be ignored.

- If you specify multiple words in the search string, then it will search for complete text string in the selected fields.

6.5.1 Interface WFObjectSearch (Package com.fujitsu.iflow.model.workflow)

This interface provides the methods by which you can search the process instance based on the specified text string.

Handle or reference of this interface can be obtained from `WFObjectFactory` class.

Interface Semantics

The following sample represents the proper order in which the methods of this interface are to be invoked:

```
WFSession session = WFObjectFactory.getWFSession();
// login to IBPM server and create a valid session
WFObjectSearch wfObjectSearch = WFObjectFactory.getWFObjectSearch(session,

      Filter.AllProcesses);
wfObjectSearch.setSearchFields(new int[] {WFObjectList.LISTFIELD_UDA,
      WFObjectList.LISTFIELD_PROCESSINSTANCE_NAME,
      WFObjectList.LISTFIELD_PROCESSINSTANCE_DESCRIPTION});
Object[] processes = wfObjectSearch.search("mobile phone", 0, 10);
int totalNumberOfObjects = wfObjectSearch.getCount();
for (int i = 0; i < processes.length; i++){
      ProcessInstance processInstance = (ProcessInstance) processes[i];
      String piName = (processInstance.getName().replaceAll(
            WFObjectFactory.SEARCH_HIGHLIGHT_START_DELIMITER, "<em>"))
            .replaceAll(WFObjectFactory.SEARCH_HIGHLIGHT_END_DELIMITER,
"</em>");
     String piDescription = (processInstance.getDescription().replaceAll(

             WFObjectFactory.SEARCH_HIGHLIGHT_START_DELIMITER, "<em>"))
            .replaceAll(WFObjectFactory.SEARCH_HIGHLIGHT_END_DELIMITER,
"</em>");
      System.out.println(processInstance.getId() + " - " + piName);
      System.out.println(piDescription);
     . . . . . .
     . . . . . .
}
```

Make sure that the `getCount()` call follows `search()` call and `setSearchFields()` call precedes `search()` call. If `setSearchFields()` is not called, then by default the given text will be searched in all the fields of the process instance.

You can also retrieve the result in batches by providing the start index and batch size.

For highlighting the search keywords in the returned process instance objects, highlight delimiters are used. Highlight delimiters can be replaced by some other values like tags for identifying them to

highlight in UI. In the above code snippet, the delimiters are replaced with some other tags i.e. . Thus any value enclosed between and tags can be highlighted in UI.

If you do not want to highlight the search keywords in the result then you can remove them programmatically. If not removed then they might give un-expected results.

6.6 Retrieving Information about Multiple Objects at a Time

The WFDetailsList interface allows you to efficiently retrieve information about many objects at a time. This capability is also referred to as bulk processing.

Currently, the interface provides methods to do the following:

- Retrieve User Defined Attributes (UDAs) of multiple process instances at a time
- Retrieve outgoing arrows of multiple work items at a time

This section explains the typical steps that you follow to use this interface. A programming sample is presented, which retrieves UDAs of multiple process instances. You can find the complete programming code in the BatchDetailsRetrieval.java sample file. The sample file also contains an example of how to retrieve outgoing arrows of multiple work items.

To retrieve UDAs of multiple process instances at a time:

1. Make sure that you have a list of process instance IDs for which you want to retrieve UDAs later. To do so, you typically use one of the process instance filters provided by the WFObjectList interface.

   In the programming sample, to prepare for the bulk processing functionality, the active process instances of the logged-in user are retrieved:

   ```
   WFObjectList list = WFObjectFactory.getWFObjectList(session);
   list.openBatchedList(Filter.MyActiveProcesses);
   Object[] batch = list.getNextBatch(10);
   ```

   For more information about filters, refer to section *Advanced Filtering & Sorting API* on page 111.

2. If process instances are found, assign their IDs to an array.

   ```
   if (batch == null) {
      < ... >
      return;
   }
   long[] idValues = new long[batch.length];
   for (int i = 0; i < batch.length; i++) {
      idValues[i] = ((ProcessInstance) batch[i]).getId();
   }
   ```

3. As the list of process instances is not needed for further processing, you are recommended to close it.

   ```
   list.closeList();
   ```

4. Retrieve the UDAs of all of the process instances that you previously retrieved. To do so, create a `WFDetailsList` instance and call `getUDAsForProcessInstances()`.

```
WFDetailsList detailsList = WFObjectFactory.getWFDetailsList(session);
ProcessInstancesUDASet udaInfo = detailsList
                .getUDAsForProcessInstances(idValues);
```

   The `udaInfo` object stores the UDAs of the process instances in a Java Map. The process instance ID is used as key, and another Java Map representing the UDAs of that process instance is used as value.

   Next, you will retrieve the UDAs of a particular process instance.

5. To retrieve the UDAs of a particular process instance, use `getUDAsForProcessInstance()` from the `ProcessInstancesUDASet` class:

```
for (int i = 0; i < idValues.length; i++) {
   Map udaData = udaInfo.getUDAsForProcessInstance(idValues[i]);
   < ... >
}
```

   In the sample, the UDAs of a process instance are assigned to a Java Map named `udaData`. This Map uses the UDA's name as key and an object of type `UDAData` as value. The `UDAData` object stores the name, data type and value of an individual UDA.

6. To process the individual UDAs of a process instance, use a Java Iterator to iterate through the UDAs of that process instance. You can retrieve the name, data type and value of a UDA, using `getUdaName()`, `getUdaType()` and `getUdaValue()` from the `UDAData` class.

   The following sample creates a Java Iterator, which is required to iterate through the elements of the `udaData` Map. Each object returned by `iterator.next()` is casted to a `UDAData` object. Finally, the UDA's name, data type and value are retrieved for further processing.

```
Iterator iterator = udaData.keySet().iterator();
while (iterator.hasNext()) {
   UDAData data = (UDAData) udaData.get(iterator.next());
   System.out.println("\tUDA name: '" + data.getUdaName() + "'");
   System.out.println("\tUDA type: '" + data.getUdaType() + "'");
   System.out.println("\tUDA value: '" + data.getUdaValue()
                      + "'\n");
}
```

6.7 Using Process Comments

To facilitate flexible communication among users, Interstage BPM allows users to add comments to the node or process instance.

You can add, retrieve, and delete comments on `ProcessInstance` and `NodeInstance` interfaces in the `com.fujitsu.iflow.model.workflow` package using the following APIs.

Note: The following APIs can be used for both process instance and node instance.

- To fetch a comment - `getComments()`. This API returns an array of comments of a process instance or node instance. Comments can be fetched by all users.

  For example, to fetch comments from

  - the node instance object `activity` of a User Task Node, use `activity.getComments();`

- the process instance object `pi`, use `pi.getComments();`

You can retrieve comments' details, using `Comment` interface in the `com.fujitsu.iflow.model.workflow` package. For example, the following APIs can be used.

- getMessage(): Returns message for this Comment
- getId(): Returns comment ID for this Comment
- getUserId(): Returns the user ID who has added this Comment
- getTimeStamp(): Returns the timestamp in milliseconds, for when this comment is added.

Refer to the API Javadoc for details on the available classes and methods.

The following sample code gives an example of retrieving individual comment information from a process instance object `pi`.

```
Comment [] comments = pi.getComments();
for(int count=0;count<comments.length;count++){
     System.out.println("CommentId: "+comments[count].getId());
     Date date = new Date(comments[count].getTimeStamp());
     System.out.println("Timestamp: "+ date);
     System.out.println("UserId: "+comments[count].getUserId());
     System.out.println("CommentMsg: "+comments[count].getMessage());
     System.out.println("Deleted flag: "+comments[count].isDeleted());
  }
```

Users can also fetch all comments of a process instances and comments from associated node instances using `ProcessInstance.getAllComments()` API.

The following sample code gives an example of fetching all comments from the process instance object `pi` and comments from all node instances of process instance object `pi`.

```
 Comment [] comments = pi.getAllComments();
```

Note: If no comments are present on the process instance or node instance, an empty array of comments is returned.

- To add a comment -`addComment()`. This API adds the user-provided comment to the process instance or node instance. Administrators, process instance owners, process instance initiator, and assignees of each work item can add comments on that process instance. Administrators, process instance owners, process instance initiator, and assignees of each work item on that node can add comments on that node instance.

Note: For a node instances, comments can be added to the User Task nodes, Voting User Task nodes, Embedded Sub-Process nodes and Dynamic User Task nodes.

Note: Comments can be added on a node instance and process instance, if the process instance is not locked by another user.

The following sample code gives an example of adding a comment to the node instance object `activity` of a User Task Node.

```
String comment = "This is a comment on the node instance";
activity.addComment(comment);
```

The following sample code gives an example of adding a comment to the process instance object `pi`.

```
String comment = "This is a comment on the process instance";
pi.addComment(comment);
```

- To delete a comment - `deleteComment()`. This API marks the given comment as deleted. Comments can be deleted by the user who has added the comment and administrators.

Note: A comment can be deleted from a process instance, only if the process instance is not locked by another user.

The following sample code gives example of deleting a comment from the node instance object `activity` of a User Task Node.

```
Comment[] comments = activity.getComments();
long commentId = comments[0].getId();
activity.deleteComment(commentId);
```

Note: Make sure that the CommentID for the comment that you want to delete, is specific to the node instance or process instance from which you want to delete the comment. If you try to delete a comment from the node instance using a CommentID of a comment belong to the process instance, you will get an error. Same holds true for process instance.

The following sample code gives an example of deleting a comment from the process instance object `pi`.

```
Comment[] comments = pi.getComments();
long commentId = comments[0].getId();
pi.deleteComment(commentId);
```

6.8 Using Additional History Information

Interstage BPM allows users to add additional information to all the operations they perform in enactment edit mode.

You can add any information about a certain operation. This additional information is stored along with the history information of the operation performed.

You need to define additional information for an operation as a key value pair in a hashtable and set this hashtable as additional history information for an operation. You can retrieve this additional history information in future through the history information of the operation.

Additional History Information feature provides the following functionalities:

- You can retrieve the added additional history information for an operation through process instance or node instance history.
- If you add additional history information only for a UDA update operation through work item, then the additional history information is stored with the associated activity history and you can retrieve this information with node instance history.
- If you add additional history information for both UDA update and work item execution operations through work item, then you can retrieve this information along with the work item execution history of the node instance.

- If you add additional history information only for UDA update operation through process instance, then you can retrieve this information along with process instance edit history.

6.8.1 Adding Additional History Information

Follow the steps given below to add additional history information for UDA update and work item execution operations together using `setAdditionalHistoryInfo(Hashtable)` method of the `com.fujitsu.iflow.model.workflow.WorkItem` interface:

1. Retrieve the required work item.
2. Start enactment edit on the work item.
3. Update the UDA.
4. Add the additional history information.
5. Execute the required work item.
6. Commit edit.

   Committing the work item:

   - updates the UDA
   - executes the work item
   - adds the additional history information with work item execution history

You can use the following sample code as basis to add additional history information for UDA update and work item execution operations together:

```
// Here, consider 'wi' is the required WorkItem object
// Start the enactment edit
wi.startEdit();
Properties udaProp = new Properties();
udaProp.put("UDAName", "newValue");
wi.setDataItemValues(udaProp);
Hashtable additionalHistInfo = new Hashtable();
// Add the additional history information
additionalHistInfo.put("OperatorName ", "Smith");
additionalHistInfo.put("OperationType",
           "Update customer's information");
wi.setAdditionalHistoryInfo(additionalHistInfo);
wi.makeChoice("choiceArrowName");
wi.commitEdit();
```

Note: For adding additional history information for enactment edit operation from process instance, use the method `setAdditionalHistoryInfo(Hashtable)` of the `com.fujitsu.iflow.model.workflow.ProcessInstance` interface.

6.8.2 Retrieving Additional History Information

You can retrieve additional history information through process instance history or node instance history using `ProcessInstance.HISTORY_ADDITIONAL_INFO` field.

You can use the code similar to given below for retrieving additional history information associated with node instance using `NodeInstance.getHistory()` instead of `ProcessInstance.getHistory()`.

You can use the following sample code as basis to retrieve additional history information:

```
// Here 'pi' is the required ProcessInstance object
int historyFields[] = { ProcessInstance.HISTORY_ID,
```

```
        ProcessInstance.HISTORY_TIME_STAMP,
        ProcessInstance.HISTORY_ADDITIONAL_INFO,
        ProcessInstance.HISTORY_EVENT_TYPE };
IflowEnumeration historyEnum = pi.getHistory(historyFields);
while (historyEnum.hasMoreElements()) {
   Hashtable htblEvent = (Hashtable) historyEnum.nextElement();
   Hashtable additionalHistoryInfo = (Hashtable) htblEvent.get(new Integer(

            ProcessInstance.HISTORY_ADDITIONAL_INFO));
   if (additionalHistoryInfo != null) {
      // get the information from Hashtable and use as required.
   }
```

6.9 Special User Defined Attribute Properties

6.9.1 Working with User Defined Attributes of Type XML

When modeling the data that will be used in a process, you might need to handle structured data in XML format. For example, an external system might pass an XML structure to Interstage BPM which needs to be modified during process execution and then passed back.

Interstage BPM provides a special data type for XML data. User Defined Attributes (UDAs) of type XML allow you to access and manipulate the entire XML structure as well as substructures, single elements and attributes. XPath expressions are used to select components from the XML data for further processing.

Before storing data in a UDA of type XML, the Interstage BPM Server checks whether the data is well-formed. Only well-formed XML data can be stored. Optionally, you can specify an XML schema for each UDA and validate the XML data against that schema.

UDAs of type XML can be created, updated, read and deleted like any other UDA. For UDAs of type XML, there are several additional methods provided for the following:

- setting and retrieving an XSD schema
- setting and retrieving the UDA value as file
- setting and retrieving the UDA value using an XPath expression, i.e. you can set and retrieve a value for the entire XML structure, for a substructure and for a single element.
- validating the XML content of a UDA

When mapping UDAs of type XML, the `DataItemMappingElement` class of the `WorkItem` interface allows for

- mapping a full XML structure to a full XML structure
- mapping a single element to another UDA of type XML
- mapping an XML substructure to another XML substructure
- mapping an XML substructure to a full XML structure

Below you find some examples for working with UDAs of type XML. Refer to the `ComplexPlan.java` sample file for the entire sample code.

To work with UDAs of type XML:

1. When adding UDAs, specify `TYPE_XML` as the data type:

```
DataItemRef dataRefOrder = procDef.addDataItemRef(XMLUDA_BOOKSTORE,
             DataItemRef.TYPE_XML, XMLVAL_BOOKSTORE);
```

The XMLUDA_BOOKSTORE constant defines the name of the UDA:

```
private final static String XMLUDA_BOOKSTORE = "XMLBookstore";
```

The XMLVAL_BOOKSTORE constant defines the value for the UDA in the following XML structure:

```
private final static String XMLVAL_BOOKSTORE =
     "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>"
     + "<bookstore>"
     + "<book category=\"COOKING\">"
     + "<title lang=\"en\">Everyday Italian</title>"
     + "<author>Giada De Laurentiis</author>"
     + "<year>2005</year>"
     + "<price>30.00</price>"
     + "</book>"
     + "<book category=\"CHILDREN\">"
     + "<title lang=\"en\">Harry Potter</title>"
     + "<author>J K. Rowling</author>"
     + "<year>2005</year>"
     + "<price>29.99</price>"
     + "</book>"
     + "<book category=\"WEB\">"
     + "<title lang=\"en\">XQuery Kick Start</title>"
     + "<author>James McGovern</author>"
     + "<author>Per Bothner</author>"
     + "<author>Kurt Cagle</author>"
     + "<author>James Linn</author>"
     + "<author>Vaidyanathan Nagarajan</author>"
     + "<year>2003</year>"
     + "<price>49.99</price>"
     + "</book>"
     + "<book category=\"WEB\">"
     + "<title lang=\"en\">Learning XML</title>"
     + "<author>Erik T. Ray</author>"
     + "<year>2003</year>"
     + "<price>39.95</price>" + "</book>" + "</bookstore>";
```

2. To read a single element of the XML structure, in this case the price for the first book, specify the element using an XPath expression:

```
DataItem di = pi.getDataItem(XMLUDA_BOOKSTORE);
// retrieve the price for the first book
String price = di.getElementValue("/bookstore/book[1]/price/text()");
            logger.log(Logger.DEBUG, "Price is: " + price);
```

3. To read the name of an XML node in the substructure, in this case the name of the XML node defining the second book:

```
org.w3c.dom.Node subNode = di.getSubTreeValue("/bookstore/book[2]");
          logger.log(Logger.DEBUG, "Node name is: " +
subNode.getNodeName());
```

   Note that you must use the `org.w3c.dom.Node` interface for addressing the XML node, because Interstage BPM also offers a `Node` interface.

4. To compare the value of a single XML element:

```
String xPath = "/bookstore/book[3]/author/text()";
String expAuthor = "Erik T. Ray";
logger.log(Logger.DEBUG, "Author: " + di.getElementValue(xPath));
if (di.getElementValue(xPath).equals(expAuthor)) {
  logger.log(Logger.DEBUG, "Expected author found");
}
```

6.9.2 Working with Pre-defined XML Data Structures ('Custom' Data Types)

When using User Defined Attributes (UDAs) of type XML, you need to define the corresponding XSD each time you want to define a new UDA of type XML. If you intend to re-use part of a data structure of XML data type, you need to create many UDA definitons of the same structure, which is a drawback. In such a case, you can choose to work with a new data type, called 'custom' data type.

A 'custom' data type is basically XML, but its structure is pre-defined (with an associated XSD) and named in Interstage BPM Studio. This 'named structure' then serves as a new data type, which can be used to create new UDAs of that type.

For example, the structure of 'address details' can be re-used for 'shipping address' as well as 'billing address'. To use 'address details' as a custom data type:

1. In Interstage BPM Studio, create (or import) an XSD that defines the data structure for address details.
2. Store the XSD in the schema folder of the workflow application. The address details will now become available as a custom data type.
3. In Interstage BPM Studio, for a process definition of the workflow application, add a new UDA. For its data type, select the address details from the list of custom data types. Name the custom UDA as 'Shipping Details'.
4. You can re-use the 'address details' structure by creating another UDA in a similar way, and name it 'Billing Address'.
5. To use custom UDAs by Model API, upload the workflow application (including the XSD) to Interstage BPM Server.

The scope of a custom data type is limited to the application it is defined in.

Custom data types defined by using XSD are shared in the same application project.

Custom Data Type Nomenclature

Custom data types are named using either of the following:

- Global elements and namespaces, in the format **<global element name>#<namespaceURI>**.
- XSDType and namespaces, in the format **<XSDType name>@<namespaceURI>**.

where `XSDType` means `xsd:complexType` and `xsd:simpleType`, and `<namespaceURI>` is the targetNamespace for the XSD in which the type is defined.

Consider the following XSD.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.com/PO1"
            xmlns:po="http://www.example.com/PO1">
     <xsd:element name="shipToPurchaseOrder" type="po:PurchaseOrder"/>
     <xsd:complexType name="PurchaseOrder">
         <xsd:sequence>
             <xsd:element name="customerName" type="xsd:string"/>
             :
         </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="Address">
         <xsd:sequence>
             <xsd:element name="street" type="xsd:string"/>
             :
         </xsd:sequence>
     </xsd:complexType>
</xsd:schema>
```

With reference to the XSD above,

- an example of custom data type name using a global element and namespace is `shipToPurchaseOrder#http://www.example.com/PO1`
- an example of custom data type name using an XSDType and namespace is `Address@http://www.example.com/PO1`

Internalization of Custom UDAs

When working with Custom UDA values, it is possible that the same short name is used in different XMLs to refer to a different namespace, or, the same namespace is referenced by different short names. This can cause conflicts during operations like copy, or when specifying path expressions.

To overcome this problem, Interstage BPM automatically 'internalizes' all Custom UDA values before they are set. Internalization involves removing all namespace declarations, prefixes for elements and attributes, and all processing instructions, comments. Also, attributes with prefix `xml` and `xsi` are removed.

Once the UDA value is set it is implicitly internalized, and all further use of the UDA value in Interstage BPM Server, Model, Console, Java Actions, and so on will be in the internalized format. When you retrieve any Custom UDA value within Interstage BPM, the value will be available in this 'internalized' form, that is, all items listed in the previous paragraph will have been removed.

It follows that when you add initial value of a custom UDA in a process definition, or set the value of a custom UDA in a process instance, you need not add short name declaration for any element even if `elementFormDefault` equals `qualified` in the schema.

Note: The internalize operation will fail if the UDA value provided is not a valid XML document or fragment.

Sample Internalization of a well-formed XML value

Before internalization

```
<?xml version="1.0" encoding="UTF-8"?>
<po:shipToAddress xmlns:po="http://www.example.com/PO1" >
    <!-- this is ship to address -->
```

```
    <po:street po:name="Oak Street" number="12-3-5"/>
    <po:city>Wonder City</po:city>
    <po:state>Timberland</po:state>
    <isbn:zip xmlns:isbn="http://www.example.com/ISBN">123456</isbn:zip>
</po:shipToAddress>
```

After internalization

```
<shipToAddress>
    <street name="Oak Street" number="12-3-5"/>
    <city>Wonder City</city>
    <state>Timberland</state>
    <zip>123456</zip>
</shipToAddress>
```

Sample Internalization of an XML fragement

Before internalization

```
<!-- this is ship to address -->
<po:street po:name="Oak Street" number="12-3-5"/>
<po:city>Wonder City</po:city>
<po:state>Timberland</po:state>
<isbn:zip xmlns:isbn="http://www.example.com/ISBN">123456</isbn:zip>
```

After internalization

```
<street name="Oak Street" number="12-3-5"/>
<city>Wonder City</city>
<state>Timberland</state>
<zip>123456</zip>
```

Note: If in a Custom UDA value there are two (or more) **sibling** elements that are **defined in different namespaces** but **have the same local name**, after internalization those elements will be considered as the same element, that is, as an array. For example, if a UDA value is provided as follows:

```
<bb:customer xmlns:bb="http://www.example.com/bb"
 xmlns:cc="http://www.example.com/cc"
 xmlns:dd="http://www.example.com/dd">
   <cc:Name>SomeName</cc:Name>
   <dd:Name>SomeOtherName</dd:Name>
</bb:customer>
```

Then Interstage BPM Server will internalize it and save it as follows, and both `Name` elements will be considered as the same element (as an array).

```
<customer>
   <Name>SomeName</Name>
   <Name>SomeOtherName</Name>
</customer>
```

Externalization of Custom UDAs

'Externalizing' a custom UDA means converting the 'internalized' UDA value to a namespace-aware format. During externalization, effectively, the short name prefix is added to all elements and attributes which are required to be 'qualified' as per the schema definition. Also, the namespace declarations

corresponding to the short name prefixed are added to the root tag. If the custom data type is not defined in any of the schema files or if any particular element or attribute in the UDA value is not defined in the schema, the externalize operation will fail, and an exception will be thrown.

You need to externalize a custom UDA only when it is passed outside the scope of Interstage BPM. For example, in a custom defined web service java action you may need to externalize the UDA value before it is sent as part of the web service request.

Use the following APIs for for externalizing a custom UDA value:

- `getExternalizedValue(Node targetNode)` method of the `DataItemRef` interface
- `getExternalizedProcessAttribute(String udaName, Node targetNode)` method of the `ServerEnactmentContext` interface

For more details of these methods, refer the API Javadoc.

Note that Interstage BPM also performs an 'externalize' operation prior to validating the UDA value against the schema files in the workflow application.

Sample Externalization

- Consider a custom UDA of type `Address@http://example.com/addr` containing the following value:

```
<street name="Oak Street" number="12-3-5"/>
<city>Wonder City</city>
<state>Timberland</state>
<zip>123456</zip>
```

- Also, the corresponding schema definition is as below:

```
<xsd:schema targetNamespace="http://example.com/addr"
   xmlns:addr="http://example.com/addr"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:complexType name="Address">
        <xsd:sequence>
            <xsd:element name="street" form="qualified">
                <xsd:complexType>
                    <xsd:attribute name="number"
                        type="xsd:integer" form="qualified" />
                    <xsd:attribute name="name"
                        type="xsd:string" form="unqualified" />
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="city" type="xsd:string" />
            <xsd:element name="state" type="xsd:string"
              form="qualified" />
            <xsd:element name="zip" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

- The target node into which the UDA value is to be externalized is an empty node, the value of which is `<udaValue/>`
- After the externalized UDA value is copied into the target node, its value will be as follows:

```
<udaValue>
    <addr:street xmlns:addr="http://example.com/addr"
        name="Oak Street" addr:number="12-3-5"/>
    <city>Wonder City</city>
    <addr:state
```

```
        xmlns:addr="http://example.com/addr">Timberland</addr:state>
    <zip>123456</zip>
</udaValue>
```

Custom UDA Manipulation

You can manipulate custom UDAs using the following:

- APIs: For details see *Custom UDA Manipulation using APIs* on page 128.
- Simple Java objects: For details see *Manipulating Custom UDAs using Java Objects* on page 130.
- JavaScript: For details see *Manipulating Custom UDAs with JavaScript* on page 276.

Custom UDA Manipulation using APIs

Note that no APIs exist to create custom UDA. They can be created by defining the XSD, only within Interstage BPM Studio.

However, following APIs exist for Custom UDA Manipulation:

- **Retrieving the list of custom data types that have been defined for an application**: to do this, use the `getApplicationCustomDataTypes(String applicationId)` method of the `com.fujitsu.iflow.model.workflow.WFSession` interface
- **Adding a UDA of a specified custom data type to a process definition**: to do this, use the `addDataItemRef(String dataItemRefName, String dataItemRefType)` method of the `com.fujitsu.iflow.model.workflow.Plan` interface. Here, `dataItemRefName` refers to the name of the UDA you want to add, and `dataItemRefType` refers to the name of the custom data type. Also note that you can use any of the existing `addDataItemRef()` methods to add data items to the process definition.
- **Adding a UDA of a specified custom data type to a process instance**: to do this, use the `addDataItem(String dataItemName, String dataItemType, String dataItemValue)` method of the `com.fujitsu.iflow.model.workflow.ProcessInstance` interface. Here, `dataItemName` refers to the name of the UDA you want to add, `dataItemType` refers to the name of the custom data type, and `dataItemValue` refers to the value to be assigned to this UDA. You can also use any of the existing `addDataItem()` methods to add data items to the process instance.
- **Searching based on custom data type UDAs**: to do this, use the `addFilterWithXPath(String udaName, String xPath, String sqlOperator, String value)` method of the `com.fujitsu.iflow.model.workflow.WFObjectList` interface. This method adds an XPath-based filtering criteria for custom data type UDAs for the list of the objects to be retrieved. The XPath must be specified till the leaf node.

Note: To be able to search based on custom UDAs, those UDAs should be marked as worklist UDAs. For details on how to do this, refer *Worklist UDAs* on page 133.

- **Validating the following in an application**: that all custom data type schemas are well-formed, no duplicate namespaces exist for the application BAR file, and each XSD file has a target namespace defined. This validation is done by the `validateApplication(String appBarFilePath)` method of the `com.fujitsu.iflow.model.workflow.WFSession` interface.
- **Validating the following for a process definition**: if all the custom UDA types exist, the initial values of custom UDAs, and that no invalid XPaths have been specified. This validation is done by the `validatePlan()` method of the `com.fujitsu.iflow.model.workflow.Plan` interface.

- **Manipulating custom UDAs with simple Java objects.** For details, refer *Manipulating Custom UDAs using Java Objects* on page 130.
- **Manipulating custom UDAs with the `ServerEnactmentContext` and `DataItemRef` interfaces.** The custom UDA value can be set and retrieved using an XPath expression, i.e. you can set and retrieve a value for the entire XML structure, for a substructure and for a single element.

For more details on the methods listed above, refer the API Javadoc.

XPath Support: Note that similar to XML type UDAs, the use of XPaths with custom UDAs are supported for the following:

- Specifying a condition on a Simple Exclusive Gateway node
- Data mapping for Call Activity, Chained-Process and Remote Sub-Process nodes
- Data mapping in Triggers
- Data mapping for Subprocess started from workitem
- Predefined Java Actions
  - XML Actions
  - Web Service Call
  - Decision Tables

For custom UDAs of XSDType (in which case the UDA value would be an XML fragment), XPaths can be specified without using a root element.

For example, the value of a custom UDA can be an XML fragment that represents a purchase order, as given below:

```
<customerName>Helmut Baer</customerName>
<dueDate>2010-12-12</dueDate>
<shipTo>
     <street>315 Elm Street</street>
     <city>Oakhurst</city>
     <state>AZ</state>
     <zip>87654</zip>
</shipTo>
<totalCost>1550.00</totalCost>
```

The XPath `/shipTo/zip/text()` can be used to get the text value of the `<zip>` node, which is `87654`.

Sample code

The following code retrieves names of custom data types within an application:

```
import com.fujitsu.iflow.model.workflow.WFSession;

//Create a new WFSession object
String applicationId = "SampleApplication";
WFSession wfs = WFObjectFactory.getWFSession();

// Note: No need to execute chooseApplication().
// Before, execute the following method.
String[] cdTypes = wfs.getApplicationCustomDataTypes(applicationId);
for(int cdtfCount = 0; cdtfCount<cdTypes.length(); cdtfCount++) {
   System.out.println("Custom Data Type Name: " +
```

```
            cdTypes [cdtfCount]);
}
```

The following code adds a filter to retrieve process instances based on the custom UDA 'shipOrders', whose price is greater than or equal to 20.

```
wfObjectList.addFilterWithXPath("shipOrder",
 "/bookstore/book[1]/price/text()", WFObjectList.SQLOP_GREATERTHANOREQUALTO,
 "'20'");
```

Note: If your database does not support XMLType, or, if you have upgraded from a pre-v11.2 version of Interstage BPM without updating the database schema, you can still use custom data types, except that you will be unable to use the search functionality associated with them.

Note: Do not use short names or any namespace information in any path expressions.

Manipulating Custom UDAs using Java Objects

You can manipulate custom UDAs using simple java objects (Plain Old Java Objects - POJO) with either of the following:

- methods of the `XData` interface
- methods of the `ProcessInstance`, `ProcessInstantiator`, `ServerEnactmentContext` and `WorkItem` interfaces.

Manipulating Custom UDAs using the `XData` interface

Objects of the `com.fujitsu.iflow.model.workflow.XData` interface hold references to the custom UDA or a part of the custom UDA which they represent. Calling `setValue(String, String)`, `copy(String, XData)` or `remove(String)` methods on an `XData` object alters the value of the underlying UDA. Custom UDA can be directly manipulated using this interface; XML parsing or DOM API are not required. This interface does not support XPath, but supports something similar to XPath, referred to as pathExpression. The pathExpression supports path expressions, attributes and element cardinality. Refer to the API Javadoc for more information about this interface and methods.

This interface does not validate the custom UDA value resulting after the manipulation with the corresponding schema. That value is validated during process definition validation.

Consider the following custom UDA which is referred by `XData`:

```
<shipToPurchaseOrder>
  <customerName>Helmut Baer</customerName>
  <dueDate>2010-12-12</dueDate>
  <shipTo>
     <street number="315">Elm Street</street>
     <city>Oakhurst</city>
     <state>AZ</state>
     <zip>87654</zip>
  </shipTo>
  <billTo>
     <street number="315">Elm Street</street>
     <city>Oakhurst</city>
     <state>AZ</state>
     <zip>87654</zip>
  </billTo>
```

```
  <totalCost>1550.00</totalCost>
</shipToPurchaseOrder>
```

The path expression specified to the methods of the `XData` interface should be relative to the current `XData` object. For example, if the current XData object represents the element for the `myPurchase/shipToPurchaseOrder` expression (where `myPurchase` is the UDA Id), and the value of the `city` element from the `shipTo` element is required, the code to use is `XData.getValue("shipTo/city");`.

Attributes should be specified at the end of the path expression starting with the `@` symbol. For example, `XData.getValue("street/@number");`.

Some `XData` methods include:

- `setValue(String pathExpression, String value)`: This method sets the specified value to the element specified by the pathExpression. It also sets the attribute of an element for the specified pathExpression. If the specified pathExpression element or attribute does not exist, a new element or attribute will be created and the specified value will be set.

Note: When a new element or attribute is created in this manner, the schema of this UDA remains unchanged, hence validation will fail.

  Examples of supported pathExpressions include:
  - simple pathExpression: `shipTo/city`
  - pathExpression with attribute: `shipTo/street/@number`
  - pathExpression with cardinality: `items/item[2]/price`

Note: This pathexpression is not the same as the actual XPath. Therefore, there is no need to add a `/` at the start of the pathExpression. And at the end of the pathExpression, there is no need to add `text()` to specify a text node.

- `getXData(String pathExpression)`: This method returns an `XData` object representing the content of an element specified by the pathExpression. This method does not support attributes in the pathExpression.
- `remove(String pathExpression)`: This method removes the element specified by the pathExpression. It can also be used to remove the attribute of an element by specifying the attribute at the end of the pathExpression starting with the `@` symbol. Examples of supported pathExpressions include:
  - simple pathExpression: `shipTo/city`
  - pathExpression with attribute: `shipTo/street/@number`
  - pathExpression with cardinality: `items/item[2]`

  For example, consider an array that contains 5 "item"s. The following code will remove the 4th item of the array.

```
uda.remove("myPurchase/shipToPurchaseOrder/items/item[4]");
```

  Of 5 items, since the 4th item has been removed, the array now contains only 4 items, and the erstwhile 5th item will become the 4th item.

Sample code for custom UDA manipulation using `XData`:

```
// Before setting a value, edit on ProcessInstance is required.
// Consider 'pi' is a ProcessInstance object.
```

```
pi.startEdit();
// XData object can be retrieved for a custom UDA.
// The code below retrieves the XData for 'myStreet' and 'myAddress' UDA.

XData streetObj = pi.getXData("myStreet");
XData address = pi.getXData("myAddress");

// Value can be set relative to current XData object
// using the relative XPath.
// The code below sets the 'street' value from
// another XData object 'streetObj'.
address.copy("street", streetObj);

// The code below sets the value for 'street', 'city', 'state'
//and 'zip' in 'myAddress' UDA.
address.setValue("street/@number", "200");
address.setValue("city", "Oakhurst");
address.setValue("state", "AZ");
address.setValue("zip", "87654");

// Commit the process instance
pi.commitEdit();
```

For a list and details of all methods part of the `XData` interface, refer the API Javadoc.

Manipulating custom UDAs using the `ProcessInstance`, `ProcessInstantiator`, `ServerEnactmentContext` and `WorkItem` interfaces

Custom UDAs can be directly manipulated using methods of `ProcesssInstance`, `ProcessInstantiator`, `ServerEnactmentContext` and `WorkItem` interfaces. These methods are similar to methods of the `XData` interface, but the pathExpression is of the format `<UDA ID>/<target pathExpression>`, and the path is absolute.

Sample code for custom UDA manipulation using `ProcessInstance`:

```
// Before setting a value, edit on ProcessInstance is required.
// Consider 'pi' is a ProcessInstance object.
pi.startEdit();
//setting value to a simple type custom UDA
pi.setValue("myStreet", "315 Elm Street");

// copy the content/value of myStreet UDA to the street element
// value of myAddress UDA
pi.copy("myAddress/street", pi.getXData("myStreet"));

// to update the elements value of a custom UDA, the code below
// updates 'city', 'state' and 'zip' elements of custom UDA 'myAddress'.

pi.setValue("myAddress/city", "Oakhurst");
pi.setValue("myAddress/state", "AZ");
pi.setValue("myAddress/zip", "87654");

// the code below updates 'customerName' and 'dueDate' elements
// of custom UDA 'myPurchase'
pi.setValue("myPurchase/customerName", "Helmut Baer");
pi.setValue("myPurchase/dueDate", "2010-12-12");

// One UDA value can be directly copied to another UDA or UDA elements.
// The code below copies 'myAddress' UDA value to 'shipTo' and 'billTo'
// element values of the custom UDA 'myPurchase'.
pi.copy("myPurchase/shipTo", pi.getXData("myAddress"));
```

```
pi.copy("myPurchase/billTo", pi.getXData("myAddress"));

// If setValue() is called for setting an array element, then
// an empty element will be added till the current element specified
// in Xpath, if it does not exist.
// In the case below, if 1st and 2nd element do not exist, the code will

// add empty elements.
pi.setValue("myPurchase/items[3]/quantity","15");
pi.setValue("myPurchase/items[3]/price","100.00");

// Commit the process instance
pi.commitEdit();
```

Note: Do not use short names or any namespace information in any path expressions.

6.9.3 Worklist UDAs

Many applications require that the work list display certain User Defined Attribute (UDA) values that are used to help determine which work item will be worked on next. Normal access to UDA values from the worklist can be slow - Worklist UDAs provide a faster way of access. In addition, marking a UDA as Worklist UDA allows you to sort and filter a list of workflow elements by this UDA.

Worklist UDAs are returned as an object within the worklist object. The same set of UDA values will be included regardless of the activity from which the worklist object is generated. These values will be included with the work item object, and can be immediately read without having to load the entire process instance into the model.

There are two methods that can be used to obtain UDA information on a particular work item using the Model API. You start by obtaining the work item object. Then, you can either obtain all of the UDAs (DataItems) of this work item with `getDataItems()` or a subset that consists of the work item `DataItems` with the `getWorklistDataItems()` method.

The prerequisite for using the `getWorklistDataItems()` method from the `WorkItem` interface is that you mark certain `DataItems` as Worklist UDAs. This can be accomplished when you design the process definition by using `markAsWorkListUDA()` from the `DataItemRef` interface.

There are the following worklist UDA-specific methods:

- `markAsWorkListUDA()`: Marks a UDA as a worklist UDA.
- `boolean isWorkListUDA()`: Checks, whether a UDA is a worklist UDA.

The following sample code demonstrates how you can define UDAs and mark them as Worklist UDAs when you create a process definition. Refer to the `ComplexPlan.java` sample file for the complete process definition.

```
//*** Definition of UDAs ***
DataItemRef udaPrice = plan.addDataItemRefWithId("Price",
      "WLUDA_PRICE", DataItemRef.TYPE_FLOAT, "0.0");
DataItemRef udaQty = plan.addDataItemRefWithId("Quantity",
      "WLUDA_QTY", DataItemRef.TYPE_INTEGER, "0");
DataItemRef udaTotal = plan.addDataItemRefWithId("Total",
      "WLUDA_TOTAL", DataItemRef.TYPE_FLOAT, "0.0");

//Mark the UDAs as Worklist UDAs.
udaPrice.markAsWorkListUDA(true);
udaQty.markAsWorkListUDA(true);
udaTotal.markAsWorkListUDA(true);
```

You can use these Worklist UDAs, for example, to retrieve process instances that match filtering criteria based on values of UDAs. For a programming sample, refer to the `listProcsByUDA()` method in the `ComplexPlan.java` sample file.

Note: You can mark UDAs of type XML and type 'custom' as worklist UDAs only if the database supports XMLType.

6.10 Using Extended Attributes

Interstage BPM allows you to define extended attributes. Extended attributes store extra information that is not native to Interstage BPM. You can assign extended attributes to process definitions, nodes, and arrows.

Extended attributes are specified as an XML document. Values of extended attributes are always of type String. The following XML schema defines the format of extended attributes:

```
<xsd:element name="ExtendedAttributes">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="xpdl:ExtendedAttribute" minOccurs="0"
            maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name="ExtendedAttribute">
   <xsd:complexType mixed="true">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
         <xsd:any minOccurs="0" maxOccurs="unbounded"/>
      </xsd:choice>
      <xsd:attribute name="Name" type="xsd:NMTOKEN" use="required"/>
      <xsd:attribute name="Value" type="xsd:string"/>
   </xsd:complexType>
</xsd:element>
```

When using extended attributes, make sure that your XML documents comply with this XML schema. The following sample is a valid instance:

```
<ExtendedAttributes>
   <ExtendedAttribute Name="SomeAttributeName" Value="Some value"/>
   <ExtendedAttribute Name="AnotherAttributeName" Value="200"/>
</ExtendedAttributes>
```

The `Plan`, `Node`, and `Arrow` interfaces provide the following methods related to extended attributes:

- `setExtendedAttributes (org.w3c.dom.Document attrs)`: Assigns extended attributes to the process definition, node or arrow. `attrs` is an XML document that contains the extended attributes to be assigned.
- `getExtendedAttributes()`: Returns the XML document that contains all extended attributes of the process definition, node or arrow.

`ComplexPlan.java` contains a sample that shows how to work with extended attributes. A sample process definition `CxPD_PurchaseOrder` is created that is supposed to be used by divisions in different countries. Therefore, certain strings like the process definition name and node names need to be available in different languages. Extended attributes are used to store the language-specific names.

The following sections explain how to assign and retrieve extended attributes.

6.10.1 Assigning Extended Attributes

To assign extended attributes to a process definition, node, or arrow:

1. Create an XML document containing the extended attributes that you want to assign to an element.

   The sample program `ComplexPlan.java` creates two XML documents: One for the language-specific names of the `CxPD_PurchaseOrder` process definition and another for the language-specific names of a User Task Node. The following sample code shows the XML document that is created for the language-specific names of the process definition:

```
StringBuffer sb_xmlProcDef = new StringBuffer();
sb_xmlProcDef.append("<ExtendedAttributes>");
sb_xmlProcDef.append("<ExtendedAttribute
   Name=\"DE\" Value=\"Kaufauftrag\"/>");
sb_xmlProcDef.append("<ExtendedAttribute
   Name=\"EN\" Value=\"Purchase Order\"/>");
sb_xmlProcDef.append("<ExtendedAttribute
   Name=\"FR\" Value=\"Contrat d'achat\"/>");
sb_xmlProcDef.append("</ExtendedAttributes>");
InputStream in_xmlProcDef = new
ByteArrayInputStream(sb_xmlProcDef.toString().getBytes("UTF-8"));
```

   This is the XML document that is created for the language-specific names of a User Task Node:

```
StringBuffer sb_xmlProcDef = new StringBuffer();
sb_xmlProcDef.append("<ExtendedAttributes>");
sb_xmlProcDef.append("<ExtendedAttribute Name=\"DE\"
   Value=\"Bestellformular ausfuellen\"/>");
sb_xmlProcDef.append("<ExtendedAttribute Name=\"EN\"
   Value=\"Fill out Purchase Requisition\"/>");
sb_xmlProcDef.append("<ExtendedAttribute Name=\"FR\"
   Value=\"Remplir le formulaire de commande\"/>");
sb_xmlProcDef.append("</ExtendedAttributes>");
InputStream in_xmlActivity = new
ByteArrayInputStream(sb_xmlProcDef.toString().getBytes("UTF-8"));
```

   There are some restrictions on the names you can use for extended attributes. For more information, refer to section *Names of Extended Attributes* on page 137.

2. Read the XML documents containing the extended attributes.

```
DocumentBuilderFactory dbf =
   DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbf.newDocumentBuilder();
Document extAttPlan = builder.parse(in_xmlProcDef);
Document extAttNode = builder.parse(in_xmlActivity);
```

3. Assign the extended attributes to the element.

   The following sample code shows how to assign extended attributes to a process definition and to a node:

```
procDef.startEdit();
procDef.setExtendedAttributes(extAttPlan);
nodePR.setExtendedAttributes(extAttNode);
procDef.commitEdit();
```

Once you have assigned extended attributes to an element, you can use `getExtendedAttributes()` to retrieve them for further processing.

6.10.2 Retrieving the Value of an Extended Attribute

Prerequisite: You have assigned extended attributes to an element as explained in section *Assigning Extended Attributes* on page 135.

To retrieve the value of a particular extended attribute:

1. Read the extended attributes that have been assigned to the element.

   The following sample shows how to read the extended attributes that have been assigned to a process definition:

   ```
   Document resExtAttPlan = procDef.getExtendedAttributes();
   ```

   Alternatively, you can use your own custom Java Action Type to read the extended attribute assigned to a process definition. Refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93 to create your own custom Java Action Type. Make sure that you use `ServerEnactmentContext.getProcessDefinitionExtendedAttributes()` API in your customized Java action.

2. Build an XPath expression that specifies the location of the extended attribute within the XML document.

   In the following sample, `exAttName` stores the name of an extended attribute that is to be retrieved.

   ```
   String xpath = "/ExtendedAttributes/ExtendedAttribute[@Name=\""
            + exAttName + "\"]";
   ```

   If `exAttName` has the value `EN`, the resulting XPath expression is:

   ```
   "/ExtendedAttributes/ExtendedAttribute[@Name=\"EN\"]"
   ```

3. Select the XML node that corresponds to the extended attribute:

   ```
   org.w3c.dom.Node ndLang =
                XPathAPI.selectSingleNode(resExtAttPlan, xpath);
   ```

   The resulting XML node might be, for example:

   ```
   <ExtendedAttribute Name="EN" Value="Purchase Order"/>
   ```

4. To access the value of the extended attribute:

   ```
   String strRes =
             ndLang.getAttributes().getNamedItem("Value").getNodeValue();
   ```

   If an element has no extended attributes assigned, `getExtendedAttributes()` or `selectSingleNode()` return a null value.

6.10.3 Retrieving all Extended Attributes of an Element

Prerequisite: You have assigned extended attributes to an element as explained in section *Assigning Extended Attributes* on page 135.

To retrieve all extended attributes of a particular element:

1. Build an XPath expression that specifies the location of the extended attributes within the XML document:

```
String xpath = "/ExtendedAttributes/ExtendedAttribute";
```

2. Read the extended attributes that have been assigned to the element.

   The following sample retrieves all nodes of a process definition and checks whether extended attributes have been assigned to a node. If a node has extended attributes, they are assigned to a list of XML nodes.

```
Node[] nodes = procDef.getNodes();
for (int iNodeCnt=0; iNodeCnt<nodes.length; iNodeCnt++) {
   Node tmpNode = nodes[iNodeCnt];
   Document resExtAttNode = tmpNode.getExtendedAttributes();
      if (resExtAttNode != null) {
      NodeList nodelistNode =
         XPathAPI.selectNodeList(resExtAttNode, xpath);
   . . . }
}
```

   Alternatively, you can use your own custom Java Action Type to read the extended attribute assigned to an activity definition. Refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93 to create your own custom Java Action Type. Make sure that you use `ServerEnactmentContext.getActivityExtendedAttributes()` API in your customized Java action.

3. To access a particular extended attribute within the list of XML nodes:

```
for (int iNListCnt=0; iNListCnt<nodelistNode.getLength();
          iNListCnt++) {
   org.w3c.dom.Node exAttNode = nodelistNode.item(iNListCnt);
. . . }
```

   Suppose `nodelistNode` contains the following XML nodes:

```
<ExtendedAttribute Name="DE"
   Value="Bestellformular ausfuellen"/>
<ExtendedAttribute Name="EN"
   Value="Fill out Purchase Requisition"/>
<ExtendedAttribute Name="FR"
   Value="Remplir le formulaire de commande"/>
```

   In this case, `nodelistNode.item(1)` returns the following XML node:

```
<ExtendedAttribute Name="EN"
   Value="Fill out Purchase Requisition"/>
```

6.10.4 Names of Extended Attributes

When defining extended attributes, make sure that their names do not interfere with names used by Interstage BPM or by the XPDL standard.

Extended Attributes Native to Interstage BPM

When converting a process definition to XPDL, Interstage BPM encodes some information that is native to Interstage BPM into extended attributes. Below is the list of names used by Interstage BPM for its native extended attributes.

Note: Do not use these names as extended attribute values.

XML nodes having these attribute values may be removed when the process definition is converted to XPDL. Interstage BPM uses some of these XML nodes to map internal data; they are extracted from the extended attributes using `getExtendedAttributes()`.

```
Associations
Artifacts
ChildPlan
ChildPlanId
CommitJavaActionSet
Coordinates
CustomNodeType
DataMapping
DataMappings
Description
EnableFutureWorkItems
EndPoint
EpilogueJavaActionSet
ExposedField
FormList
FormsList
IflowDataType
InitJavaActionSet
InTransaction
IsWorkItemUDA
IteratorCountUDA
LoopInfo
NodeType
Organization
ParentVersion
PrivateData
ProcessDefinitionId
ProcessOwnerRole
ProcessOwnerRoleJavaActionSet
ProcessTypeId
PrologueJavaActionSet
```

RoleJavaActionSet

SameSubPlanVersion

StartPoint

State

subProcessDefinitionURI

SWIM_LANES

TemplateIdentifier

TimerDefSet

Title

TriggerDefSet

VersionComment

ViewerScript

Example: The following XML fragment defines two extended attributes, `StartPoint` and `StartPoint1`. The definition of `StartPoint` is faulty because this name is used by Interstage BPM for one of its native extended attributes.

```
<ExtendedAttributes xmlns:x="http://example.com">
   <ExtendedAttribute Name="StartPoint">
      <x:StartPoint>
         PointA
      </x:StartPoint>
   </ExtendedAttribute>
   <ExtendedAttribute Name="StartPoint1">
      <x:StartPoint1>
         PointB
      </x:StartPoint1>
   </ExtendedAttribute>
</ExtendedAttributes>
```

When the process definition is converted to XPDL, the `StartPoint` XML node will be removed. The resulting XPDL fragment would look like this:

```
<ExtendedAttributes xmlns:x="http://example.com">
   <ExtendedAttribute Name="StartPoint1">
      <x:StartPoint1>
         PointB
      </x:StartPoint1>
   </ExtendedAttribute>
</ExtendedAttributes>
```

6.10.5 Namespace of Extended Attributes

Interstage BPM generates namespace-aware XPDL that fully complies with the XPDL 2.0 standard.

Note: If you are using extended attributes, you are recommended to prefix their names with valid namespaces. This ensures that the XPDL generated by Interstage BPM remains fully compliant with the XPDL standard.

If you do not use a namespace when defining extended attributes, Interstage BPM adds the default `ibpm` namespace to generate namespace-aware XPDL that fully complies with the XPDL 2.0 standard.

Example 1

The following sample defines extended attributes that are prefixed with the namespaces `x` and `y`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtendedAttributes xmlns:x="http://example.com"
   xmlns:y="http://example.net">
   <ExtendedAttribute Name="myAttribute">
      <x:myTag1>
         myData1
      </x:myTag1>
      <y:myTag2>
         <myTag3>
            myData2
         </myTag3>
      </y:myTag2>
   </ExtendedAttribute>
</ExtendedAttributes>
```

After importing such a process definition, `getExtendedAttributes()` returns the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtendedAttributes xmlns="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:xpdl="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:ibpm="http://fujitsu.com/ibpm1" xmlns:x="http://example.com"
xmlns:y="http://example.net">
   <ExtendedAttribute Name="myAttribute">
      <x:myTag1>
         myData1
      </x:myTag1>
      <y:myTag2>
         <myTag3>
            myData2
         </myTag3>
      </y:myTag2>
   </ExtendedAttribute>
</ExtendedAttributes>
```

Note that additional namespace attributes (the `ibpm` namespace and the XPDL 2.0 standard namespaces) have been added to the root element. The immediate children of the `ExtendedAttribute` element have not been prefixed with `ibpm` because appropriate namespaces were already present in the original XML document.

Example 2

This XML fragment defines some extended attributes without namespace. In the context of the XPDL standard, this XML is invalid because the children of the `ExtendedAttribute` element have no namespace prefix.

```
<!-- INVALID -->
<ExtendedAttributes>
   <ExtendedAttribute Name="myAttribute">
         <myTag1>
             myData1
         </myTag1>
         <myTag2>
            <myTag3>
               myData2
```

```
            </myTag3>
         </myTag2>
    </ExtendedAttribute>
</ExtendedAttributes>
```

After importing such a process definition, `getExtendedAttributes()` returns the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtendedAttributes xmlns="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:xpdl="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:ibpm="http://fujitsu.com/ibpm1">
   <ExtendedAttribute Name="myAttribute">
      <ibpm:myTag1>
         myData1
      </ibpm:myTag1>
      <ibpm:myTag2>
         <myTag3>
            myData2
         </myTag3>
      </ibpm:myTag2>
   </ExtendedAttribute>
</ExtendedAttributes>
```

The returned XML document has changed in two aspects:

- The `ibpm` namespace and the XPDL 2.0 standard namespace attributes have been added to the root element.
- The `ibpm` prefix has been added to the immediate children of the `ExtendedAttribute` element.

6.11 Transaction Control

The Interstage BPM transaction control mechanism allows for rolling back transactions to a specific point in process execution. This mechanism also improves the system's overall performance.

A process often consists of a sequence of nodes. You can decide whether or not the transaction stays open after node completion. If the transaction stays open, the next node in the process will be processed in the same transaction.

Consider the following:

- Performance: Committing a transaction saves everything to the database, and might force everything to be read from the database again to start the subsequent transaction. If you have a long chain of nodes that do very little computation (for example, a chain of branch nodes deciding among many possible directions) the overhead of saving to the database every time will be significant and unnecessary.
- Rollback upon errors: If a low level error occurs during processing, the current transaction will be rolled back to the last commit point. If four nodes are all part of the same transaction, all four nodes will be rolled back in case of an error, and upon restart all four will be executed again. Committing the transaction somewhere in the middle means that it will rollback only to that point.

Controlling Transactions with the Model API

There are two methods in the `com.fujitsu.iflow.model.workflow.Node` interface supporting the transaction control feature. These are:

- `setNodeInTxn()`: Sets the transaction flag on this node. This transaction flag is used to decide whether the transaction should be committed when this node completes.

- `getNodeTxnStatus()`: Gets the node transaction status. Depending upon whether the node is set to complete in the same transaction.

For more information on these methods or the Model API in general, refer to the API Javadoc.

For a node to automatically commit a transaction with the next commit operation (and not directly), set `setNodeInTxn()` to `true`, for example:

```
fillOutNode.setNodeInTxn(true);
```

For checking the transaction status of a node, use `getNodeTxnStatus()`, for example:

```
logger.log(Logger.INFO, "");
logger.log(Logger.INFO, "Get status of node transaction");
boolean nodeTxnStatus =
   workItem.getNodeInstance().getNodeTxnStatus();
logger.log(Logger.INFO, "Status: " + nodeTxnStatus);
```

Refer to the `ComplexPlan.java` and `ProcessExecution.java` samples for the complete node definition.

Rolling Back Transactions in External Systems

You can define actions for first level Java Actions which perform a "cleanup" before a transaction is rolled back and a process instance is set to the error state. This includes a rollback of all Java Actions in a Java Action Set, and allows you to perform general actions (e.g. sending notification emails in any error case), or executing some specific actions before setting the process instance to error state.

If you do not define any error handling actions for a Java Action, the following happens: When an exception is thrown in this Java Action, the transaction will be rolled back. A rollback only effects systems participating in the same container transaction. Any external systems not participating in the transaction will not be rolled back. External systems may already have been updated during this transaction, and in this case it can be necessary to manually undo the changes, to clean up external systems to ensure a consistent state of all systems. This is done using Compensate Action Sets.

Refer to section *Dealing with Errors in Java Actions* on page 104 for more information.

6.12 Using Triggers

The purpose of a trigger is to move data, typically XML files, coming from an external system's data source into Interstage BPM process instances. These process instances are either started or directed by the incoming XML data files, and the process instances perform functions that use the data source data. The XML data files are called event data files.

Triggers can be configured to recognize the data in the event data files, so that the data can be mapped to User Defined Attributes (UDAs) in the Interstage BPM process instances, or triggers use the event data files directly and write the data to the defined UDAs.

Triggers are very versatile and can be associated with a process definition, a Message Receive Node or a User Task Node. Each of these provides a different scope of behaviour.

- When defined on process definition level, the trigger creates a new process instance and starts it.
- When defined on node level, the trigger makes a choice so that process execution moves forward to the next node.

A trigger contains two components: the event that drives the trigger and the action that the trigger takes when it is fired.

6.12.1 How It Works

Event data files (XML files) are sent from an external data source to a directory configured in the Interstage BPM File Listener. The Trigger Handler checks all the triggers configured on Interstage BPM and activates them in case the File Listener notifies the Trigger Handler of incoming XML files. The trigger then acts according to its definition:

- If the trigger is defined on process definition level, it starts a process instance from the process definition in which the trigger is defined and moves the XML data into the User Defined Attributes (UDAs) of the process according to the data mapping specified in the trigger definition.
- If the trigger is defined on node level, it makes a choice on the node for which it is defined and moves the XML data into the UDAs of the process instance according to the data mapping specified in the trigger definition.

Trigger State

A trigger is allowed to be in one of three possible states: Default, Active and Inactive. The state of a trigger is a property at definition level, i.e. changing a trigger obtained on instance level, e.g. by calling `ProcessInstance.getTriggers()`, will affect all triggers of all process instances belonging to the same process definition as well. Changes to the state of the process definition or process instance have no effect on the state of the contained trigger(s).

Default

If the trigger is 'Start Process' trigger and its state is Default then it is fired if the process definition is in Published state; else it is not fired.

If the trigger is 'Make Choice' trigger and its state is Default then it is fired if the process definition is in Published, Obsolete, or Private state.

Active

If trigger is in Active state and if the trigger is a 'Start Process' trigger then it will not be fired if the process definition is in Obsolete or Private state.

If trigger is in Active state and if the trigger is a 'Make Choice' trigger then it will be fired irrespective of the state of the process definition.

Inactive

If trigger is in Inactive state, it will not be fired irrespective of the state of the process definition.

If a process definition is deleted, the state of its trigger is Inactive.

Control Conditions

Triggers may be designed to operate on certain process instances only. In such a case, you can provide control conditions to facilitate the trigger to locate the right process instance to operate on when the trigger is fired. If you do not specify control conditions, the trigger is always fired as soon as the File Listener detects a new XML file in the specified directory.

For example, for triggers defined on process definition level: If you define triggers for several process definitions, every time the File Listener finds an XML file, a process instance for every process definition will be created. This means that an XML file can create an arbitrary number of process instances of different process definitions. You can avoid this behavior by specifying control conditions so that the creation of a process instance depends on the content of the XML file. Only if the XML file content matches the control conditions, the trigger will be fired, i.e. a process instance will be created.

Control conditions operate on User Defined Attributes. For each attribute used, you specify an element of the incoming event data via an XPath expression. When the trigger is fired, only those processes

are selected in which the value of the specified attribute matches the value of the corresponding element in the incoming event.

Action Specifications

Action specifications define the operation that the trigger will perform. Since each trigger is defined to handle only one type of operation, all the actions defined in a trigger relate to the same operation. This means, for example, that a trigger defined to make a choice will not be able to start a process.

For triggers defined on node level (make choice triggers), it is required to specify the actions that can be performed using the method `addAction(String, String)`. The first parameter contains the name of the outgoing arrow that is to be chosen if the condition specified in the second parameter is true.

When defining a trigger on a Message Receive Node, exactly one action has to be defined because the node has exactly one outgoing arrow.

When defining a trigger on a User Task Node, the number of actions that can be defined depends on the number of outgoing arrows defined for the node. The sequence of evaluating the conditions, which is identical to the second parameter in the method, depends on the sequence of the actions. The condition of the action with `index [0]` is evaluated first. If its condition is true, the conditions of the other actions will not be evaluated. The process instance continues with the arrow defined in the action whose condition was found to be true.

6.12.2 Defining a Trigger

This section explains how to define a trigger on process definition level. You can find the complete programming code of the examples presented in this section in the sample file `TriggerTimerSample.java`.

Note: The `TriggerTimerSample.java` file contains an additional sample for adding a trigger to a User Task Node. For information about triggers defined on Message Receive Nodes, refer to section *Using Message Receive Nodes* on page 146.

To define a trigger:

1. Configure the File Listener that starts the triggering mechanism so that it looks into the directory where incoming event data files are stored. For instruction on configuring and testing the File Listener, refer to section *File Listeners* on page 147.
2. Create the relevant User Defined Attributes (UDAs) for your process definition.

   In the sample file, the following UDAs are created:

```
plan.addDataItemRef("OrderID",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("OrderPerson",
   DataItemRef.TYPE_STRING,"Initial Value");
plan.addDataItemRef("Name",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Address",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("City",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("State",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("ZIP",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Title",
```

```
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Note",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Quantity",
   DataItemRef.TYPE_INTEGER, "1");
plan.addDataItemRef("Price",
   DataItemRef.TYPE_BIGDECIMAL, "0.00");
```

3. Add the trigger to your process definition, define control conditions and set its state to active.

   In the sample below, two control conditions are defined. The first one requires that the value for the price of the article must be bigger than 10, and the second one states that the name of the adressee ("shipto" tag) must be "Pete Gagnet".

```
TriggerDef myTrigger = plan.addTrigger("myTrigger",
   TriggerDef.TYPE_START_PROCESS);
myTrigger.setDescription("Will start a process instance.");

// add control conditions
String[] conditions = new String[2];
conditions[0] = "toFloat(eventData.getXMLData
   ( \"/shiporder/item/price/text()\" ))  > 10";
conditions[1] = "eventData.getXMLData
   ( \"/shiporder/shipto/name/text()\" ) == \"Pete Gagnet\"";
myTrigger.setControlConditions(conditions);

// activate trigger
myTrigger.setState(TriggerDef.STATE_ACTIVE);
```

4. Specify the data mapping for extracting the values of the event elements of the XML file that will be used to fire the trigger, and write them to the UDAs, for example:

```
myTrigger.addDataMap("OrderID","shiporder/@orderid");
myTrigger.addDataMap("OrderPerson","shiporder/orderperson/text()");
myTrigger.addDataMap("Name","shiporder/shipto/name/text()");
myTrigger.addDataMap("Address","shiporder/shipto/address/text()");
myTrigger.addDataMap("City","shiporder/shipto/city/text()");
myTrigger.addDataMap("State","shiporder/shipto/state/text()");
myTrigger.addDataMap("ZIP","shiporder/shipto/zip/text()");
myTrigger.addDataMap("Title","shiporder/item/title/text()");
myTrigger.addDataMap("Note","shiporder/item/note/text()");
myTrigger.addDataMap("Quantity","shiporder/item/quantity/text()");
myTrigger.addDataMap("Price","shiporder/item/price/text()");
```

5. Create an XML file that will be used by the trigger to start an instance of a process definition and fill the file with values. You can either use an existing XML file or create it using the API, for example:

```
// Create XML file in the folder specified by the FILELISTENER_PATH
File xml = new File(FILELISTENER_PATH + "shiporder.xml");

// Write XML data into the file
FileWriter fw = new FileWriter(xml);
BufferedWriter bw = new BufferedWriter(fw);
bw.write("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>");
bw.newLine();
bw.write("<shiporder orderid=\"889923\">");
bw.newLine();
bw.write("<orderperson>Joan Smith</orderperson>");
```

```
bw.newLine();
bw.write("<shipto>");bw.newLine();
bw.write("<name>Pete Gagnet</name>");
bw.newLine();
bw.write("<address>325 Eastwood Ave</address>");
bw.newLine();
bw.write("<city>" + CITY_OF_ORDER + "</city>");
bw.newLine();
bw.write("<state>OH</state>");
bw.newLine();
bw.write("<zip>20131-1234</zip>");
bw.newLine();
bw.write("</shipto>");bw.newLine();
bw.write("<item>");bw.newLine();
String titleStr = "<title>" + ITEM_TITLE + "</title>";
bw.write(titleStr);bw.newLine();
bw.write("<note>Special Edition</note>");bw.newLine();
titleStr = "<quantity>" + QUANTITY_ORDER + "</quantity>";
bw.write(titleStr);bw.newLine();
bw.write("<price>39.99</price>");bw.newLine();
bw.write("</item>");bw.newLine();
bw.write("</shiporder>");bw.newLine();
bw.close();
```

Note: When specifying dates in the XML file, make sure to use a date format that matches the locale setting of the Interstage BPM Server.

6.13 Using Message Receive Nodes

A Message Receive Node represents a step in a process which is driven by an external event. Each Message Receive Node has a trigger defined that is supposed to fire when data, typically XML files, comes in from an external system. Once the data arrives, the trigger moves the data into User Defined Attributes (UDAs) according to the trigger's definition. The trigger then makes a choice and process execution moves forward to the next node. No human interaction is required.

This section explains the typical steps involved in working with Message Receive Nodes. You can find the complete programming code for the example presented in this section in the sample file `EventActivityNodeSample.java`.

For general information about triggers and how they work, refer to section *Using Triggers* on page 142.

To use Message Receive Nodes:

1. Create a Message Receive Node using `addNode()`. Set the constant `nodeType` to `TYPE_EVENT_ACTIVITY`.

```
Node activity = plan.addNode("EventActivity",
   Node.TYPE_EVENT_ACTIVITY);
```

2. Add exactly one outgoing arrow to the Message Receive Node, for example

```
plan.addArrow("Exit", activity, exit);
```

3. Define a make choice trigger for the Message Receive Node that you previously created.

```
TriggerDef trigger = activity.addTrigger("ChoiceTrigger",
   TriggerDef.TYPE_MAKE_CHOICE);
```

4. Optionally, add control conditions and correlation maps.

   Control conditions check whether the content of the incoming XML data is valid for the trigger. Correlation maps are used to select the process instances the trigger will operate on.

   The following sample defines that the trigger will fire if the `<doMakeChoice>` element in the incoming XML data has `true` as its value. The trigger will only fire for process instances where a particular UDA has the same value as the `<newValue>` element in the incoming XML data.

```
trigger.addControlCondition("eventData.getXMLData"
                + "('/root/doMakeChoice/text()[1]') == 'true'");
trigger.addCorrelationMap(UDA_NAME, "/root/currentValue/text()[1]");
```

5. Specify the outgoing arrow to be chosen by the trigger using the method `addAction(String, String)`.

   The first parameter contains the name of the outgoing arrow that is to be chosen if the condition specified in the second parameter is true.

   In the programming sample, the trigger chooses the outgoing arrow if the value of a particular UDA is less than 10.

```
trigger.addAction(exit.getName(), "uda." + UDA_NAME + " < 10");
```

6. Map any information of the incoming XML data that you require for further processing to UDAs.

   In the programming sample, a single data mapping is defined, which maps the value of the `<newValue>` element to a UDA.

```
trigger.addDataMap(UDA_NAME, "/root/newValue/text()[1]");
```

7. Set the trigger's state to active.

```
trigger.setState(TriggerDef.STATE_ACTIVE);
```

The definition of the Message Receive Node and the trigger is now complete. At runtime, for the trigger to fire, some incoming XML data has to be sent to the trigger. This data is created by the sample program and sent to the trigger using `WFObjectFactory.processTriggerEvent()`:

```
StringBuffer bw = new StringBuffer();
bw.append("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>");
bw.append("<root>");
bw.append("<currentValue>5</currentValue>");
bw.append("<doMakeChoice>true</doMakeChoice>");
bw.append("<newValue>7</newValue>");
bw.append("</root>");
WFObjectFactory.processTriggerEvent(bw.toString(), adminSession);
```

6.14 File Listeners

File Listeners monitor files in specified directory locations. When they detect new or newly modified files, they notify the Interstage BPM file handlers, so they can perform automatic functions like starting

Interstage BPM process instances or making choices on activities. File listeners are typically used for integrating Interstage BPM with other enterprise applications.

6.14.1 Using File Listeners

To use a file listener:

1. The File Listener configuration file `fileListenerConf.xml` is part of the workflow application project which you create and import from Interstage BPM Studio. It should be located in `<DMSRoot>/apps/<application ID>`.
2. If you changed your File Listener configuration file, restart your workflow application.
3. Create a process definition and add a trigger that will cause the File Listener to start a process instance. Refer to section *Using Triggers* on page 142 for instructions.
4. Create a File Listener XML file for your trigger and save it into the filelistener directory. Refer to section *Using Triggers* on page 142 for instructions.

   If the trigger is working, the XML file will disappear and a process instance will start from the process definition containing the trigger. If the trigger is not working, the XML file will move to the `filelistener\error` directory.

Note: When a new or newly modified file is detected:

- In `Relax` mode, triggers of all existing applications will be processed.
- In `Secure` mode, triggers of application having the file listener that acted on the new incoming file, will be processed.

6.14.2 Configuring File Listeners

The File Listener configuration file `fileListenerConf.xml` is located in `<DMSRoot>/apps/<application ID>`.

The following is an example of a `fileListenerConf.xml` configuration file:

```
<FileListener>
   <Directory>
      <ScanInterval>60000</ScanInterval>
      <StabilizationPeriod>2000</StabilizationPeriod>
      <PostProcessing>
         <onSuccess>
            <Delete></Delete>
         </onSuccess>
         <onError>
            <Move>
            </Move>
         </onError>
      </PostProcessing>
   </Directory>
</FileListener>
```

The following table describes the XML tags used in `fileListenerConf.xml`:

Tag	Description
`<Directory>`	Encloses configuration parameters for the directory to be monitored for new or newly modified files.

Tag	Description
<Path>	Path of the directory to be monitored. In non-SaaS mode, this can be any directory. If no path is specified, the default will automatically be set to <ServerSharedRoot>/tenants/<tenantname>/apps/<application ID>/filelistener. In SaaS mode, the application will automatically use <ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener as the default directory path. Hence <Path> does not need to be specified. If specified, it will be ignored.
<ScanInterval>	Time interval in milliseconds after which the directory is scanned for any new incoming file.
<StabilizationPeriod>	Time interval in milliseconds for which the size of the file is observed (to see if it has changed) before it is processed.
<PostProcessing><onSuccess>	Action to be taken if the file is successfully processed. Possible actions are <Delete> or <Move>.
<PostProcessing><onError>	Action to be taken if the file is not successfully processed. Possible actions are <Delete> or <Move>.
<Delete>	Deletes the file.

Tag	Description
<Move>	Moves the file to a directory. On success: • In non-SaaS mode, file is moved to the specified directory. If no path was specified, file is moved to the default directory <ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener/success • In SaaS mode, file will be automatically moved to default directory <ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener/success. Hence directory does not need to be specified. If specified, it will be ignored. On error: • In non-SaaS mode, file is moved to the specified directory. If no path was specified, file is moved to the default directory <ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener/error • In SaaS mode, file will be automatically moved to default directory <ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener/error. Hence directory does not need to be specified. If specified, it will be ignored.

6.15 Email Listeners

Email Listeners monitor incoming email messages containing data files intended for Interstage BPM. When they detect messages containing these files, they notify the Interstage BPM file handlers, so they can perform automatic functions like starting Interstage BPM process instances or making choices on activities. Email listeners are typically used for integrating Interstage BPM with other enterprise applications.

6.15.1 Using the Email Listener for Triggers

You must perform the following actions to use the Email Listener:

- You must configure the Email Listener.

Note: This configuration must be made by a Tenant Administrator.

To configure your Email Listener to listen for the email responses you will send, the following parameters must be set to configure the Email Listener:

- EmailListenerAutoReplyEnabled
- EmailListenerDeleteInvalidMessages
- EmailListenerEmailAddress
- EmailListenerEnabled
- EmailListenerPassword
- EmailListenerPollingInterval
- EmailListenerPOPPort
- EmailListenerPropertiesFile
- EmailListenerServerHost
- EmailListenerUserName
- EmailStyleSheetFile

For a description of these configuration parameters, see the *Interstage Business Process Manager Server Administration Guide*.

- Have a simple process definition with a Start Process or Make Choice (Node-level) trigger with or without data mapping available and enable its trigger.

The Email Listener works just like the File Listener with one exception. The File Listener transfers any XML data file that is copied into the File Listener directory to the trigger handler for processing. The Email Listener transfers any XML data file that is contained in a specially formatted email message to the trigger handler for processing. This data file is typically used by the trigger handler to trigger an Interstage BPM event. An application access key is the special formatting required.

Note: These instructions use the System application and the default tenant.

1. Obtain the application access key for the application upon which the prerequisite process definition is defined.

   Note: You must be a Tenant Administrator or Application Owner to obtain this key using the substeps given below, otherwise you must obtain this key from Tenant Administrator or Application Owner.

   a) Log in to Interstage BPM Console and open **Application Settings** under the **System Administration** tab for the application upon which the prerequisite process definition is defined.

   Following the example, you would open the System application.

   b) Click **Access Key**.

   The application access key will be displayed.

   c) Without editing the key, copy it to a secure location on your network.

   This key must be added unedited to the message body of email messages to be used to activate the Email Listener. If non-administrators will be creating these special Email Listener messages, you must distribute this key to them.

2. Open your email client and create a new message.
3. Address the message to the email address specified in the tenant property **EmailListenerEmailAddress**.

   If you do not know this information, see the application administrator mentioned in Step 1.

4. Attach to the email message the Data Event file that will activate your trigger and transfer data to the process containing it. The file must be in the XML format, and it must conform to the schema defined on your trigger.

   This Data Event file must be of the same format that you would use with the File Listener.

   Only one file should be attached. If more than one file is attached, only the first file attached will be used.

   In summary, the following is needed in email messages that will activate the Email Listener:

   - Application Access Key (see Step 1)
   - Email Listener Email Address (see Step 3)
   - XML Data Event File (see top part of Step 4, this step)

5. Send the Email message.

   Interstage BPM will process the message and send an email response back to you.

   That message will notify you of the pass or fail status of the Email Listener.

6. Check the results of sending the Email Listener message.

   This results obtained will depend on the type of trigger to be activated by the message and the success or failure of the message to activate the trigger.

   If the message activates a Start Process trigger, a new process instance will be started from the process definition defined in the trigger. Interstage BPM will also send you an email response indicating the successful operation.

   If the message activates a Make Choice trigger, the Node defined on the trigger will be completed along the path of the choice specified in the Data Event file. Interstage BPM will also send you an email response indicating the successful operation.

   If data was mapped in the trigger, it will be added to the process according to the trigger definition.

   If the message fails to activate the trigger specified in the Data Event file, Interstage BPM will send you an email response indicating the unsuccessful operation.

   If you send an email message containing an invalid or already expired application access key, Interstage BPM will discard the message and send you a response indicating that you are using an invalid key.

Note: When a new incoming email message is detected:

- In `Relax` mode, triggers of all existing applications will be processed.
- In `Secure` mode, triggers of application whose access key is mentioned in the email message, will be processed.

6.16 Using Email Notifications

You can configure Interstage BPM to send email notifications to users about any new workitems assigned to them. The default email notification contains Interstage BPM Console URL links to each of the choices for new workitems. You can create a custom email class by implementing the interface `com.fujitsu.iflow.server.customize.intf.GNEmailCustomizerInterface` and using the context object `com.fujitsu.iflow.server.customize.intf.IflowGNContextInterface` within the implementation class. The `IflowGNContextInterface` provides access to the Interstage BPM Server-related details and `GNEmailCustomizerInterface` provides methods to customize your own email message body, subject, etc.

This custom email class file can be placed at different levels in server for configuring email notifications. These different levels are as mentioned below:

To use custom email class across tenants:

- If you use classes separately, copy the class files to `<engine directory>/server/instance/default/classes`
- If you use a custom library, copy the JAR files to the Interstage BPM library extensions directory `<engine directory>/server/instance/default/lib/ext`

To use custom email class specific to a tenant:

- If you use classes separately, copy the class files to `<engine directory>/server/instance/default/tenants/<tenant name>/classes`
- If you use a custom library, copy the JAR files to `<engine directory>/server/instance/default/tenants/<tenant name>/lib/ext`

To use custom email class specific to an application:

- If you use classes separately, copy the class files to `<DMSRoot>/apps/<application ID>/engine_classes`
- If you use a custom library, copy the JAR files to `<DMSRoot>/apps/<application ID>/engine_lib`

6.16.1 Configuring Email Notifications for Applications

To configure email notifications at application level:

1. Add a new tag `gnEmailCustomizedClassName` to `appinfo.xml`. A sample `appinfo.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<appinfo>
       <displayname>System</displayname>
       <description>This is System application</description>
       <owner>Adminrole</owner>
       <editors>Fujitsu</editors>
       <guid>1</guid>
       <params></params>

<gnEmailCustomizedClassName>mypackage.email.CustomizedEmail</gnEmailCustomizedClassName>
</appinfo>
```

Note: Use the fully qualified class name of the custom class (`mypackage.email.CustomizedEmail`) in `gnEmailCustomizedClassName` tag.

2. Place the custom email class file at any of the following locations:
   - If you use classes separately, copy the class files to `<DMSRoot>/apps/<application ID>/engine_classes`
   - If you use a custom library, copy the JAR files to `<DMSRoot>/apps/<application ID>/engine_lib`

If application-level custom email class is not present (that is, if `<gnEmailCustomizedClassName>` tag is not present in `appinfo.xml` or the tag is empty), then tenant-level email custom class is searched. If it is not present at tenant level, then email custom class is searched across tenants. If a custom class is not defined even across tenants, then the default behavior is adopted which means the email will contain default email message body, default subject, etc.

Note: You can define custom email class across tenants in `ServerGNEmailCustomizerClass` parameter in `IBPMProperties` table. For more details, refer the *Interstage BPM Server Administration Guide*.

To configure email notifications at application level, use the following API:

```
package com.fujitsu.iflow.server.customize.intf;
public interface GNEmailCustomizerInterface
{
 public boolean
   isOkToSendEmail(IflowGNContextInterface iflowGNContext)
   throws Exception;
 public String[]
  getToList(IflowGNContextInterface iflowGNContext, String[] defaultToList)

   throws Exception;
 public String[]
   getCCList(IflowGNContextInterface iflowGNContext)
   throws Exception;
 public String[]
   getBCCList(IflowGNContextInterface iflowGNContext)
   throws Exception;
 public String
  getSubject(IflowGNContextInterface iflowGNContext, String defaultSubject)

   throws Exception;
 public String
   getMessageBody(IflowGNContextInterface iflowGNContext, String
defaultMsgBody)
   throws Exception;
 public String
   getMimeType(IflowGNContextInterface iflowGNContext, String
defaultMimeType)
   throws Exception;
}
```

You can use the following sample code for configuring email notifications for applications:

```
public class CustomizeEmail implements GNEmailCustomizerInterface {
   public String getMessageBody(IflowGNContextInterface iflowGNContext,
      String defaultMsgBody) throws ModelException{
      // Get the process name this workitem belongs to
      String processName = iflowGNContext.getProcessTitle();
      String messageBody = "You have received this email from process " +

       processName;
      // Return the message body
      return messageBody;
   }
   public String getSubject(IflowGNContextInterface iflowGNContext,
      String defaultSubject) throws ModelException{
      // Get the process name this workitem belongs to
      String activityName = iflowGNContext.getNodeName();
      String subject = "Task assigned for activity " + activityName;
      // Return the subject
      return subject;
   }
   public boolean
      isOkToSendEmail(IflowGNContextInterface iflowGNContext)
        throws Exception{
```

```
      return true;
   }
   public String[]
      getToList(IflowGNContextInterface iflowGNContext,
        String[] defaultToList) throws Exception{
      return defaultToList;
   }
   public String[]
     getCCList(IflowGNContextInterface iflowGNContext) throws Exception}{

      return defaultToList;
   }
   public String[]
     getCCList(IflowGNContextInterface iflowGNContext) throws Exception}{

      return null;
   }
   public String[]
     getBCCList(IflowGNContextInterface iflowGNContext) throws Exception}{

      return null;
   }
   public String
      getMimeType(IflowGNContextInterface iflowGNContext, String
defaultMimeType)
       throws Exception{
      return "text/plain";
   }
 }
```

6.16.2 Using Make Choice Key

You can customize you email notifications using the methods `getChoices()` and `getMakeChoiceKey()` of the `IflowGNContextInterface` interface of `com.fujitsu.iflow.server.customize.intf` package. Using these methods, you can provide all the possible choices within the workitem and generate corresponding keys for each of the choices.

`getChoices()`

You can use this method to get all the possible choices within the workitem.

`getMakeChoiceKey()`

This method returns a key for the choices provided. This key is an encoded String which when embedded in an email body and sent to the email address defined by `EmailListenerEmailAddress` property configured in Interstage BPM Server properties, will process this choice in the server. That is, it will perform the **make choice** operation on that workitem, specified by the choice. This key should be embedded in the email as it is and should not be modified.

For more information, refer *Email Listeners* on page 150.

6.17 JMS Listeners

JMS Listeners monitor incoming JMS messages containing data intended for Interstage BPM. When they detect messages containing this data, they notify the Interstage BPM trigger handler, so it can perform automatic functions like starting Interstage BPM process instances or making choices on activities. JMS listeners are typically used for integrating Interstage BPM with other enterprise applications.

6.17.1 Using the JMS Listener for Triggers

You must have the following to use the JMS Listener:

- The knowledge to create a simple JMS client application
- A simple process definition with a Start Process or Make Choice (Node-level) trigger with or without data mapping available and enable its trigger.

The JMS Listener works just like the File Listener with one exception. The File Listener transfers any XML data file that is copied into the File Listener directory to the trigger handler for processing. The JMS Listener transfers XML data from a JMS message to the trigger handler for processing. This data is typically used by the trigger handler to trigger an Interstage BPM event. An application access key must be included in the JMS message as indicated below.

Note: The instructions in Step 1 use the System application and the default tenant.

1. Obtain the the application access key for the application upon which the prerequisite process definition is defined.

   Note: You must be a Tenant Administrator or Application Owner to obtain this key using the substeps given below, otherwise you must obtain this key from Tenant Administrator or Application Owner.

   a) Log in to Interstage BPM Console and open **Application Settings** under the **System Administration** tab for the application upon which the prerequisite process definition is defined.

   Following the example, you would open the System application.

   b) Click **Access Key**.

   The application access key will be displayed.

   c) Without editing the key, copy it to a secure location on your network.

   This key must be added unedited to the JMS message to be used to activate the JMS Listener. If non-administrators will be creating these special JMS messages, you must distribute this key to them.

2. Write theData Event STRING that will activate your trigger and transfer data to the process containing it. The STRING must be in the XML format, and it must conform to the schema defined on your trigger.
3. Determine the name of the Interstage BPM Server to which you want to post the JMS message.
4. When an Interstage BPM Server is setup, it comes with the JMS Topics like CommandTopic and ResponseTopic. Interstage BPM server will listen on to the CommndTopic for any incoming JMS message to process, and once the processing is done, the Interstage BPM Server will post a response message to the ResponseTopic. The user application can listen to the ResponseTopic to get the response back from the Interstage BPM Server. Create a simple JMS client application to post a text message onto the CommandTopic. This will be your JMS Message. You must set the xmlString variable to the XML Data Event STRING (see Step 2), the APPLICATION_ACCESSKEY property to the Application Access Key STRING (see Step 1), and the SERVER_ID property to the name of the Interstage BPM Server to which you want to post the JMS message expressed as a STRING (see Step 3). The following is an example of how you would set these properties in the JMS message:

```
...
TextMessage msg = pubSession.createTextMessage();
```

```
String xmlString = [XML Data Event STRING];
String serverName = [name of Interstage BPM Server STRING];
String applicationAccessKey = [the Application Access Key STRING];

msg.setText(xmlString);
msg.setStringProperty("SERVER_ID", serverName);
msg.setStringProperty("APPLICATION_ACCESSKEY", applicationAccessKey);
...
```

5. Post the JMS message.

   Interstage BPM server will process the JMS message and send a JMS response to ResponseTopic about the pass or fail status of the JMS Listener. The response will be of type Text Message or Object Message. When there is an exception accessing the application, a Text Message will be posted back explaining the failure and if the XML Data Event STRING was successfully processed then, the com.fujitsu.mode.workflow.TriggerResult Object will be posted back in the form of Object Message.

6. Check the results of posting the JMS message.

   This results obtained will depend on the type of trigger to be activated by the message and the success or failure of the message to activate the trigger.

   If the message activates a Start Process trigger, a new process instance will be started from the process definition defined in the trigger. Interstage BPM will also send you a response indicating the successful operation.

   If the message activates a Make Choice trigger, the Activity defined on the trigger will be completed along the path of the choice specified in the Data Event file. Interstage BPM will also send you a response indicating the successful operation.

   If data was mapped in the trigger, it will be added to the process according to the trigger definition.

   If the message fails to activate the trigger specified in the Data Event file, Interstage BPM will send you a response, described above, indicating the unsuccessful operation.

   If you post a message containing an invalid or already expired application access key, Interstage BPM will discard the message and send you a response indicating that you are using an invalid key.

Note: When a new JMS message is detected:

- In `Relax` mode, triggers of all existing applications will be processed.
- In `Secure` mode, triggers of application whose access key is mentioned in the JMS message, will be processed.

6.18 Using Agents

Agents in Interstage BPM are set up to run automatically and act asynchronously on your behalf. You can use Agents to access systems that are external to Interstage BPM. The external systems to which you can connect might be legacy systems or Web Services, both inside and outside of company firewalls. Using Agents, you can incorporate these external services into your Interstage BPM process instances.

Note:
- To include external libraries, so they can be used in Interstage BPM Agents, copy your external classes or JARs to a directory as specified in *Configuring Java Actions, Agents and JavaScripts for External Applications* on page 88.
- If you are using Agents in a cluster setup, configure your Agents on each cluster node. The Agent's configuration must be identical on all cluster nodes.

6.18.1 Agents Overview

Agents are application-specific, and are configured using an XML file called `agentsConfig.xml`. This file is part of the workflow application project when you import it from Interstage BPM Studio, and can be found in `<DMSRoot>/apps/<application id>`.

Once an Agent is configured in Interstage BPM by adding it to `agentsConfig.xml`, it can be run as an activity in an Interstage BPM process instance by assigning it to the User Task Node. The agent is then run in place of the activity. The work item is actually assigned to a system user that has the same name as the agent.

Configuration File agentsConfig.xml

The following code listing is from `agentsConfig.xml`:

```
<ActionAgentList>
   <ActionAgent>
      <Name>@TestFrameAgent</Name>
      <Description>Use for Test, return first choice</Description>
      <RetryInterval>20</RetryInterval>
      <EscalationInterval>1</EscalationInterval>
      <ClassName>com.fujitsu.iflowqa.testframe.TestFrameAgent
      </ClassName>
      <ConfigFile></ConfigFile>
   </ActionAgent>
</ActionAgentList>
```

The following table describes the XML tags used in `agentsConfig.xml`:

Tag	Description
`<ActionAgentList>`	Contains a set of Agents.
`<ActionAgent>`	Contains a definition for a single Agent. Each Agent must have its own definition that is contained under this tag.
`<Name>`	Name of the Agent. The name must begin with "`@`". This designates it as an Agent. The name has to be specified as the assignee of a User Task Node, making the activity an "Agent" activity. As such, it no longer behaves as a "normal" Interstage BPM activity but as an Agent.
`<Description>`	Short description of the Agent. It typically explains form and function.

Tag	Description
<RetryInterval>	If the Agent fails to call its external service, the Agent retries to call it after a specified time. <RetryInterval> specifies the time interval in seconds between attempts.
<EscalationInterval>	Specifies the number of failures after which the System Administrator is notified via email. An email is sent after each escalation interval. For example, with a value of 1 the System Administrator is notified each time the Agent fails. The email is sent to the address specified in the ServerEmailAddress parameter of the Interstage BPM Server.
<ClassName>	Name of the Java class associated with the Agent. This class is the functional part of the Agent.
<ClassPath>	CLASSPATH of the Java class associated with the Agent. In SaaS mode, this value is ignored.
<ConfigFile>	It is the configuration file name

Agent Class

The class specified in the <ClassName> tag in agentsConfig.xml is the Agent's access to its external service. Henceforth, it is called the Agent Class. The Agent Class must implement the following interface:

```
package com.fujitsu.iflow.server.intf;
public interface ActionAgentInvoke
   {
      public String invokeService(
         ServerEnactmentContext sec, String configFile) throws Exception;
}
```

The Agent Class calls the external service using invokeService(), receives data from the service, and returns a STRING to Interstage BPM that indicates the action of the service.

There are three possible results of Agent Class execution. These are:

- Return of a NULL STRING or an empty STRING: If the Agent Class returns a NULL STRING or an empty STRING, the Agent repeats its attempt to execute its service. The time interval is specified in the <RetryInterval> tag. Each time, the agent attempts execution the number of times specified in the <EscalationInterval> tag, the Agent sends an email message to the address specified in the ServerEmailAddress parameter of the Interstage BPM Server
- Return of any other STRING: If the Agent Class returns any other STRING, the Agent evaluates it to see if it matches the name of one of the arrows outgoing from the Agent activity. If there is a match, the process instance continues with the matching arrow. If the return STRING does not match an arrow, the Agent throws a cannot-find-arrow exception, and the process instance goes into error state.

- Agent Class throws an exception: The Agent sends the exception to Interstage BPM, and the process instance goes into error state.

AgentSimulator Example

A sample process definition using an Agent is provided in the `<engine directory>/client/samples/examples/sources` directory. `AgentSimulator.xpdl` is a process definition that models a simple bank loan approval process. An Agent called `AgentSimulator` is used to simulate a loan decision maker.

For instructions on running the sample, refer to the comments in the source file `AgentSimulator.java`.

6.18.2 Configuring FTP Agents

FTP Agents automatically transfer files attached to process instances that contain them. Like all Agents, FTP Agents take the place of Interstage BPM activities. Process instances started from a process definition containing an FTP Agent activity automatically transfer the files that arge attached to them. They transfer the attached file(s) to the FTP Server of any machine to which the Interstage BPM Server machine can connect.

An FTP Agent, like all Agents, is configured with `agentsConfig.xml`. This configuration file contains an `<ActionAgent>` section with default settings for the FTP Agent. Usually, you don't need to change the FTP Agent settings in this file.

The FTP Agent uses Interstage BPM's `ServiceAgent` class. It also uses a special FTP Agent configuration file called `ftp.xml`. This configuration file defines FTP settings, e.g. the address of the FTP host. Check this configuration file and adapt it to your needs.

Configuration File ftp.xml

The FTP Agent configuration file `ftp.xml` is located in `<DMSRoot>/apps/<application id>`.

The following is a sample `ftp.xml` file that configures an FTP Agent:

```
<Services>
   <Service>
      <ServiceType>FTP</ServiceType>
      <ServiceStatusUDA>AgentServiceStatus</ServiceStatusUDA>
      <ServiceResultUDA>AgentServiceResult</ServiceResultUDA>
      <ServiceSpecificInfo>
         <FTPHost><HOSTNAME or IP Address></FTPHost>
         <FTPPort></FTPPort>
         <FTPUser>anonymous</FTPUser>
         <FTPPassword></FTPPassword>
         <FTPType>ASCII</FTPType>
         <FTPAppend>FALSE</FTPAppend>
         <FTPDirectory>%%REMOTE_FTP_DIRECTORY%%
            </FTPDirectory>
         <FTPFileNames>%%REMOTE_FTP_FILES%%</FTPFileNames>
         <Documents>%%OUTGOING_FILES%%</Documents>
      </ServiceSpecificInfo>
   </Service>
</Services>
```

Some of the tags in this file are used by all agents using the `ServiceAgent` class. Some of them are specific to FTP Agents.

The following table describes the XML tags used in `ftp.xml`:

Tag	Description
<ServiceType>	For the FTP Agent, the service type is always FTP. If there is no service type specified or the service type is invalid, the process instance goes into error state. This tag is used by all agents using the ServiceAgent class.
<ServiceStatusUDA>	Name of the User Defined Attribute (UDA) that stores the status of the FTP Agent activity. Possible values are Done or Failed. Removing this tag has no effect on the FTP Agent; however, the user cannot see the status of the FTP Agent activity.
<ServiceResultUDA>	Name of the UDA that stores the error message, if the FTP Agent throws an exception. Removing this tag has no effect on the FTP Agent; however, if the FTP Agent fails, the user cannot see the error that occurred.
<FTPHost>	Host name or IP address of the machine that receives the transferred file. The default value is 127.0.0.1, i.e. the local host. If there is no FTP host specified or the FTP host is invalid, files are transferred to the local FTP server, if available. This tag is specific to the FTP Agent.
<FTPPort>	Port used by the FTP server. If no port is specified, the default FTP port 21 is used. This tag is specific to the FTP Agent.
<FTPUser>	User name of the FTP user that transfers the file. If the user name is invalid, file transfer fails and the process instance goes into error state. This tag is specific to the FTP Agent.
<FTPPassword>	Encrypted password of the FTP user that transfers the file. This tag is specific to the FTP Agent. To encrypt the password, use the EncryptPassword.bat/EncryptPassword.sh located in <engine directory>/client/samples/configuration directory. For details on how to encrypt passwords, refer to section *Password Encryption* of the *Interstage BPM Server and Console Installation Guide.*
<FTPType>	Type of files to be transferred. Allowed values are ASCII or BINARY. Default value is BINARY. Use ASCII if you are transferring text files. Use BINARY if you are transferring binary files. If any other value than ASCII or BINARY is specified, the FTP Agent throws an exception. This tag is specific to the FTP Agent.

Tag	Description
<FTPAppend>	Specifies whether a file to be transferred is appended to a remote file. Allowed values are TRUE or FALSE. Default value is FALSE. Use TRUE if you want to append the contents of the local file to an already existing remote file. Use FALSE if you want to overwrite an already existing remote file. If any other value than TRUE or FALSE is specified, the FTP Agent throws an exception. This tag is specific to the FTP Agent.
<FTPDirectory>	Name of the User Defined Attribute (UDA) that specifies the directory to which files are transferred. A path relative to the FTP server is specified. If an invalid directory is specified, the process instance goes into error state. This tag is specific to the FTP Agent.
<FTPFileNames>	Name of the UDA that specifies the file names to be used for the remote files. This parameter is used to rename files while transferring them. For example, if LocalFile.txt is attached to the process instance and REMOTE_FTP_FILES has RemoteFile.txt as its value, LocalFile.txt is transferred to the FTP directory and renamed to RemoteFile.txt. REMOTE_FTP_FILES can contain several file names separated by semicolon (;). If REMOTE_FTP_FILES contains no file names, the original file names are used.
<Documents>	Name of the UDA that specifies the files that are transferred to the FTP directory. OUTGOING_FILES can contain several file names separated by semicolon (;). If OUTGOING_FILES contains no file names, all files attached to the process instance are transferred.

Note: Files that have a semicolon (;) in their names cannot be transferred to an FTP directory. Semicolons are considered as file name delimiters and are therefore not allowed within names of files that are to be transferred.

6.18.3 Using FTP Agents

To use an FTP Agent in your process definition:

1. Create a User Task Node that represents the FTP agent.
2. Assign "@FTP" to the User Task Node.
3. Add all User Defined Attributes (UDAs) that correspond to parameters in the FTP Agent configuration file ftp.xml to your process definition.

   These are the UDAs you might need to add:

   - AgentServiceStatus
   - AgentServiceResult
   - REMOTE_FTP_DIRECTORY

- REMOTE_FTP_FILES
- OUTGOING_FILES

If you don't want to add REMOTE_FTP_FILES or OUTGOING_FILES, remove the contents of the corresponding tag from the ftp.xml file. Example: You want to keep the original names of the files to be transferred. Therefore, you don't add the REMOTE_FTP_FILES UDA to your process definition. In this case, you need to remove the contents of the <FTPFileNames> tag from your ftp.xml file: <FTPFileNames></FTPFileNames>.

If you don't add REMOTE_FTP_FILES or OUTGOING_FILES while the corresponding parameter is specified in the ftp.xml file, the process instance goes into error state once the FTP Agent node becomes active.

Refer to section *Configuring FTP Agents* on page 160 for more information about the ftp.xml configuration file.

4. Add a Done arrow that originates from the FTP Agent node to activate the next activity.

The FTP Agent either returns Done or null:

- If it returns Done, the process instance continues with the Done arrow.
- If it returns null, the process instance goes into error state. The AgentServiceStatus UDA indicates the error state, and the AgentServiceResult UDA stores the error message that has been issued.

In the following cases, the process instance goes into error state, but AgentServiceStatus and AgentServiceResult UDAs are NOT updated with error information:

- The FTP Agent node has no outgoing Done arrow.
- Some data in agentsConfig.xml is incorrect or missing.

To see the error reason, the user needs to check the history of the process instance.

6.18.4 Configuring HTTP Agents

HTTP Agents are used to send data to external locations on the Internet and receive data from the Internet. An Agent sends data to the URL specified in its configuration and receives response data from that URL. It sends the value of the User Defined Attributes (UDAs) specified in the <HTTPRequestUDA> tag to the URL specified in the <HTTPBaseURL> tag and assigns the return data as the value of the UDA specified in the <HTTPReponseUDA> tag.

An HTTP Agent, like all Agents, is configured in the agentsConfig.xml file. This configuration file contains an <ActionAgent> section with default settings for the HTTP Agent. Usually, you don't need to change the HTTP Agent settings in this file.

To configure an HTTP Agent in Interstage BPM:

Create an HTTP Agent configuration file and place it in <DMSRoot>/apps/<application_id>.

Configuration File HTTPagent.xml

The following is a sample HTTP Agent configuration file:

```
<HTTPAgent>
   <HTTPBaseURL method="POST"
      followRedirects="true">{{Field HTTP_URL}}</HTTPBaseURL>
   <HTTPHeaderParams name="Content-Type">text/xml;
      charset=UTF-8</HTTPHeaderParams>
   <HTTPHeaderParams name="User-Agent">I-BPM HTTP Agent
   </HTTPHeaderParams>
```

```
   <HTTPHeaderParams name="accept-charset">UTF-8
   </HTTPHeaderParams>
   <QueryParams name= "ParamName">Param value</QueryParams>
   <HTTPRequestUDA>HTTP_REQUEST</HTTPRequestUDA>
   <HTTPResponseUDA>HTTP_RESPONSE</HTTPResponseUDA>
   <HTTPAgentStatusUDA>HTTP_STATUS</HTTPAgentStatusUDA>
</HTTPAgent>
```

The following table describes the XML tags used in `HTTPagent.xml`:

Tag	Description
`<HTTPBaseURL>`	URL end point. The value for the `<HTTPBaseURL>` tag specifies the external service to be called by the HTTP Agent, e.g. a Servlet. This value can also be defined to come from a process instance UDA as QuickForm expression, such as `{{Field HTTP_URL}}`. Attributes: `method="POST"` or `"GET"`. Defines the request method to be used for the HTTP request. If you do not specify the `method` attribute, the process instance will go into the error state. If you specify an attribute value other than `"POST"` or `"GET"`, the HTTP agent will fail. `followRedirects="true"`. Since the default value for `followRedirects` is `false`, you must specify this attribute.

Tag	Description
<HTTPHeaderParams>	Sets the HTTP header properties like the encoding style, the content type, etc. HTTP headers are used to define a variety of web object properties, for example, the properties of request and response objects for the current HTTP session. Attributes: name="Content-type" or "User-Agent". "Content-type" defines the type of requested content and takes the following value: value="text/xml", charset=UTF-8. This value can also be defined to come from process instance UDAs as QuickForm expression such as {{Field contentType}}. "User-Agent" contains information about the client (user agent) from where the request originates. Note that currently there is no restriction for the value that can be specified in the name attribute of the <HTTPHeaderParams> tag. Any valid values for this attribute are valid HTTP headers.
<QueryParams>	This tag is required in case the "GET" method is used with the HTTP Agent. The value of this tag is passed to the external service as Query String. Note that the value of the UDA specified in the <HTTPRequestUDA> tag is not used when the "GET" method is used. Correct usage of this tag is as follows: <QueryParams name= "ParamName">Param value </QueryParams> This will result in the following URL specification during HTTP Agent execution: URL?ParamName=Param Value You can specify any valid string for the name attribute of the <QueryParams> tag.
<HTTPRequestUDA>	Value of the UDA specified in this tag is part of the query string for the HTTP request.
<HTTPAgentStatusUDA>	UDA where the HTTP service status will be stored.
<HTTPResponseUDA>	UDA where the HTTP response will be stored.

6.18.5 Using HTTP Agents

To use an HTTP Agent in your process definition:

1. Create a User Task Node that represents the HTTP agent.
2. Assign `@HTTPAgent` to the User Task Node.
3. Add all the User Defined Attributes (UDAs) that correspond to parameters in the `HTTPAgent.xml` file to your process definition.

   These are the UDAs you might need to add:

   - `HTTP_URL`
   - `HTTP_REQUEST`
   - `HTTP_RESPONSE`
   - `HTTP_STATUS`
4. Add a `Done` and a `Failed` arrow that originate at the HTTP Agent node for activating the next node.

   The HTTP Agent either returns `Done` or `Failed`:

   - If it returns `Done`, the process instance continues with the `Done` arrow.
   - If it returns `Failed`, the process instance continues with the `Failed` arrow.

HTTP Agent Example

A pre-configured HTTP Example is provided for your use in the `<engine directory>/client/samples/examples/sources/HTTPAgentExample` directory. This example provides an easy-to-implement HTTP Agent. The example agent demonstrates the complete functionality of an HTTP Agent.

This example includes a `.war` file that needs to be deployed on the application server where the Interstage BPM server has been setup. Proceed with the implementation of the HTTP Agent as described in the `HTTPAgentExampleInstructions.txt` file located in the `HTTPAgentExample` directory.

Below, you find some additional explanations:

- The configuration files `agentsConfig.xml` and `HTTPAgent.xml` include the configuration of the HTTP Agent used in this example. The HTTP Agent settings are contained in the `HTTPAgent.xml` file. The `agentsConfig.xml` file specifies the `HTTPAgent.xml` file as an agent.
- You must restart the Interstage BPM Services to have your newly configured HTTP Agent take effect.
- The imported template `HTTP Agent Example V1.xpdl` contains the HTTP Agent. An agent is assigned to a User Task Node in the same manner that a User Group is assigned to a User Task Node. However, the special designation for the agent name is the `@` sign. In this case the agent name is `@HTTPAgent`.

6.19 Using Timers

Timers trigger certain actions when they expire. These timers can be Node-Level timers that start running when the User Task Node is activated or Process-Level timers that start running whenever a new process instance is created from the process definition containing the timer.

Timers are controlled by time and action parameters, specified as User Defined Attributes, and set through Java Actions or forms. The expiration or firing time of a timer can be an absolute, specific time or relative to another event (such as the start of the node or process instance). The timer can

also be set to trigger actions periodically. Timers can also perform a variety of actions, including escalation (assigning the activity to an additional list of users) or sending emails.

Execution and expiration of timers can also be controlled for the actions such as cancel, reschedule, expire or reset active timres while the timer is running.

Refer to sections *Controlling Execution of Timers* on page 169 and *Defining a Timer* on page 167.

Timers can be set with the following nodes:

- User Task Node
- Timer Node
- Voting User Task Node
- Embedded Sub-Process Node

Refer to section *Timers* on page 33 for an overview of available timer types.

6.19.1 Defining a Timer

In the following example a timer is added to a Timer Node, which waits 200 milliseconds and then sends an e-mail.

To add a timer to a node:

1. Define a Java Action and the User Defined Attributes (UDAs) for a `TimerAction` object.

```
JavaActionSet timerJASet = WFObjectFactory.getJavaActionSet();
timerJASet.setActionSetDescription("Timer controlled execution"
   + " of actions");
timerJASet.setActionSetName("TimerActions");
JavaAction[] timerJA = timerJASet.createJavaActions(1);

timerJA[0].setActionDescription("Sends an email ");
timerJA[0].setActionName("EmailSend");
timerJA[0].setClassName(CLASS_NAME_JAVA_ACTION);
timerJA[0].setMethodName("sendEmail(String,String,String,String,"
   + "String,String,ServerEnactmentContext)");
timerJA[0].setArgumentsUDANames("<E>uda.to</E><E>uda.from</E>"
   + "<E>uda.cc</E><E>uda.bcc</E><E>uda.subject</E>"
   + "<E>uda.body</E><E>sec</E>");
timerJASet.setJavaActions(timerJA);

plan.addDataItemRef("Timer_JavaActionSet",
DataItemRef.TYPE_STRING,
timerJASet.toString());
```

2. Define a `TimerAction` object. Use `getTimerDef()` from the `WFObjectFactory` class to create a timer of a specified type.

```
TimerAction timerAction = WFObjectFactory.getTimerAction();
timerAction.setJavaActionSet("Timer_JavaActionSet");
```

3. In this step, define the type of the used timer with `setType(timerType)`. You find the possible values of the `timerType` constant in the `TimerDef` interface:
   - `RELATIVE`: Defines a Relative timer.
   - `ABSOLUTE`: Defines an Absolute timer.
   - `PERIODIC`: Defines a Periodic timer.
   - `BUSINESSPERIODIC`: Defines a Business Periodic timer.

- `BUSINESSRELATIVE`: Defines a Business Relative timer.

```
TimerDef timer = WFObjectFactory.getTimerDef();
timer.setName("Send mail timer");
timer.setDescription("This timer waits 2000 miliseconds and then"
   + " sends an email.");
timer.setType(TimerDef.RELATIVE);
timer.setTime("TimeOfExpiration");
timer.setTimerTimeExpression("DateAdd(Packages.java.util.Date(), 4,
\"hh\").getTime();");
timer.addTimerAction(timerAction);
plan.addDataItemRef("TimeOfExpiration",
   DataItemRef.TYPE_LONG, "2000");
```

4. Define a `TimerDef` object, i.e. add the `TimerAction` object to `TimerDef` object.
5. Set the name of the UDA which contains the time of expiration.
6. You can also set a Javascript that gets evaluated to calculate the expiry date of a timer.

Note: You can set the Javascript for Absolute timers, Relative timers. You cannot set the Javascript for Periodic timers.

- In case of an Absolute timer, the result of script evaluation is considered as the expiry time. However, if script evaluation does not result in a `date-time` value in `Long` format, then the UDA value is considered as the expiry time. That is, expiry time = script evaluation result or UDA value.
- In case of relative timer, the result of script evaluation is considered as the reference time and time specified in UDA value is considered as relative time from the reference time. That is, expiry time = script evaluation result + UDA value.

This example sets the expiry time as four hours from the time the script is executed.

```
timer.setTimerTimeExpression("DateAdd(Packages.java.util.Date(), 4,
\"hh\").getTime();");
```

This example sets expiry time as two days from `RequestDate` in XML UDA.

```
timer.setTimerTimeExpression("sec.getProcessXMLAttributeElementValue(\"xmlUda\",
 \"//Request/RequestDate/text()\");");
 timer.setTime("timerUda"); // timerUda holds the value for 2 days in
long format
```

`TimerAction` objects and `TimerDef` objects are distinct from one another, and they do not need to be created in a specific order. However, to use a `TimerAction` object, it must be associated with the `TimerDef` object by using `timer.addTimerAction(TimerAction action)`.

7. Add the `TimerDef` object to the node.

```
delayNode.addTimer(timer);
```

6.19.2 Adding a Due Date

You can add due date timer at plan or node level by specifying the time by when the process or node should complete. The following sample code illustrates adding and retrieving DueDate on the Process Definition.

```
 //Adding DueDate on Plan:
plan.startEdit();
TimerDef timerDef = WFObjectFactory.getTimerDef();
timerDef.setName("DueDateTimer");
timerDef.setType(TimerDef.RELATIVE);
plan.addDataItemRef("DueDateExpTime",DataItemRef.TYPE_STRING, "300000");
timerDef.setTime("DueDateExpTime");

//To add DueDate on Plan
plan.setDueDateDef(timerDef);
plan.commitEdit();

//To get Due date on Plan
TimerDef dueDateDef = plan.getDueDateDef();
ProcessInstance pi = plan.createProcessInstance();

//To retrieve DueDate of Process
Date dueDate = pi.getDueDate();
```

6.19.3 Controlling Execution of Timers

Interstage BPM allows you to control the execution of active timers. You can cancel an active timer, re-schedule it to a new absolute time, reset it to recalculate the timer expiry time, or expire it immediately. By controlling the execution of a timer, you can either change the actual expiry of a timer or stop it from being executed.

Note: Timer control is only accessible to the process owners and the administrators.

You can control the execution of an active timer in the following ways:

- **Cancel a timer:** You can cancel an active timer. The cancelled timer will not execute even after the expiry time is reached. However, it can be reactivated by re-scheduling or resetting it.

Note: You can reshedule and reset a cancelled timer.

- **Re-schedule a timer:** You can re-schedule an active or cancelled timer by specifying an absolute time at which they should expire.

Note: Rescheduling any timer can only be achieved by mentioning an absolute time.

The following code samples show usage of `reschedule` API

```
Calendar cal = Calendar.getInstance();
cal.clear();
cal.set(2011, 7, 11, 14, 30, 00);
timerInstance.reschedule( cal.getTimeInMillis() );
//Assignee of an activity can also re-schedule a DueDate timer
//along with process owners and the administrators.
```

Note: Assignee of an activity can also re-schedule a DueDate timer along with process owners and the administrators.

- **Reset a timer:**When you reset an active or cancelled timer, the system gets the timer value again from the UDA or recalculates it from the script. This is useful when the UDA is changed after the timer is activated or cancelled. Once reset, the timer will expire at the new expiry time that is defined.
- **Expire a timer:** Active timers can be forcefully fired immediately, irrespective of their scheduled expiry time.

Note: A DueDate Timer cannot be forcefully fired.

Note: If a timer is rescheduled, reset, or expired, it is implicitly canceled and a new timer instance is started.

6.19.4 Timer Instance History

The timer instance history information is stored for audit purposes. Each time there is a change to a timer instance, the trail of the change is stored and can be retrieved to be viewed at a later time.

The following sample code shows how to retrieve the history of a timer.

```
TimerInstance[]  tInsts = processInstance.getTimerInstances(timerDefId);
 for (int i = 0; i < tInsts.length; i++) {
   TimerInstance tInstance = tInsts[i];
   System.out.println(tInstance.getId() + "\t"
       + tInstance.getName() + "\t"
       + tInstance.getDescription() + "\t"
       + tInstance.getTimerInstanceId() + "\t"
       + new Date(tInstance.getTimerExpirationTime()) + "\t"
       + tInstance.getState() + "\t" + tInstance.getActor());
  }
```

6.19.5 Using Business Calendars

Interstage BPM allows you to create timers that trigger certain events associated with the timer upon its expiration. The Business Calendars feature allows you to create business timers. A business timer is a special type of timer that will only "count" business hours and days and expires only during business hours. Business hours and days are specified in a Business Calendar.

The Business Periodic timers and the Business Relative timers are used with Business Calendars. Since Business Calendars are used in combination with timers, they can be used with the following types of nodes:

- User Task Node
- Timer Node
- Voting User Task Node
- Message Receive Node
- Embedded Sub-Process Node

Refer to section *Timers* on page 33 for an overview of available timer types.

You can find the complete programming code of the examples presented in this section in the sample file `TriggerTimerSample.java`.

Configuring Business Calendars

Interstage BPM allows you to use many different business calendars. If you haven't defined a business calendar for the process definition you are using, Interstage BPM uses the default business calendar setup with Interstage BPM.

You can create and use your own custom business calendars or modify the default business calendar to meet your needs. You may create as many Business Calendars as necessary to meet the needs of your organization and use a different Business Calendar for every process definition or process instance. The name of the calendar corresponds to the name of the calendar file. The calendar file is given a `.cal` file extension, so it can be recognized as a Business Calendar.

A default calendar (named `Default.cal`) is provided at setup time. If no other calendar is configured for use with Interstage BPM, you can still add a business timer to your Interstage BPM process instances, and the default calendar will specify the business days and hours. The default calendar also provides an example of a fully functional Business Calendar.

For instructions in creating and using your own custom business calendars or modifying the default business calendar, refer to the *Interstage Business Process Manager Server Administration Guide*.

Assigning Business Calendars

You can assign a particular calendar to a process definition or process instance by assigning its name (calendar file name without the `.cal` extension) to the `__businessCalendar` User Defined Attribute (UDA). This UDA is used exclusively for Business Timers and Due Dates.

For example, say you had a calendar that you wanted to use for process instances run in your German subsidiary and you called it `German.cal`. You want a certain activity to be assigned to one of your employees, and if this employee is not available after a certain period of time, you want the activity to be assigned to another employee. The time when the activity is reassigned depends on your definition of the Business Calendar and the timer you use.

To assign a business calendar to a process definition:

1. Add a UDA of type STRING to your process definitions and assign the name of the calendar file as its value.

   For example, you could add a STRING type UDA to your German subsidiary division process definitions and assign it a value of `German`:

   ```
   plan.addDataItemRef("__businessCalendar", DataItemRef.TYPE_STRING,
   "German");
   ```

2. Add a UDA with a given name, type, and initial value to the process definition. The value of the UDA is to be specified as STRING and represents the start time for the timer.

   In the example, the business time is set relative to the current time of the day by two minutes:

   ```
   plan.addDataItemRef("__udaStartTime", DataItemRef.TYPE_STRING,
   "BT(00:02:00)");
   ```

   For a list of time and day codes that you can use for the business time, refer to section *Time and Day Codes for Timers* on page 172.

3. Add a node to which the business periodic or business relative timer is to be assigned, for example, a User Task Node.

   ```
   Node orderActivity = plan.addNode("Order",Node.TYPE_ACTIVITY);
   ```

4. Create the timer by specifying its name, a description, the start time defined by the `__udaStartTime` UDA, and by defining whether it is to be a business relative or periodic timer.

   In the example, a business relative timer is used:

```
TimerDef busCalTimer = WFObjectFactory.getTimerDef();
busCalTimer.setName("busCalTimer");
busCalTimer.setDescription("Timer to show the business calendar
functionality");
busCalTimer.setTime("__udaStartTime");
busCalTimer.setType(TimerDef.BUSINESSRELATIVE);
```

5. Add the timer to the User Task Node.

```
TimerAction busCalTimerAction = WFObjectFactory.getTimerAction();
// Set the name of the UDA that contains the JavaActionSet to be
// executed when the timer expires.
busCalTimerAction.setJavaActionSet("BusCalActionSet");
// Add a TimerAction that specifies a JavaActionSet to this timer
// definition. The defined JavaActionSet of this action is executed
// when this timer expires.
busCalTimer.addTimerAction(busCalTimerAction);
//Add timer to the activity node
orderActivity.addTimer(busCalTimer);
```

You can also assign a business calendar to a particular Business Timer or Due Date. The Business Timer or Due Date is then calculated based on this business calendar. Use the following expression:

`UC(<business calendar>);`

For `<business calendar>`, specify the name of your business calendar without the .cal extension.

In the following example, a business calendar called `California.cal` is used for the calculation of the Business Timer:

```
plan.addDataItemRef("__udaStartTime", DataItemRef.TYPE_STRING,
     "UC(California);BT(00:02:00)");
. . .
TimerDef busCalTimer = WFObjectFactory.getTimerDef();
. . .
busCalTimer.setTime("__udaStartTime");
busCalTimer.setType(TimerDef.BUSINESSRELATIVE);
. . .
```

You can change calendars while a process instance is running using the `setProcessAttribute` JavaScript function and the `__businessCalendar` UDA. For example to change the business calendar to `England.cal`, you would use the following script:

```
sec.setProcessAttribute("__businessCalendar","England");
```

6.19.6 Time and Day Codes for Timers

When defining a timer, you can use time and/or day codes to specify the time. In the following example, a time code is used to set the business time relative to the current time. The time code is marked bold.

```
plan.addDataItemRef("__udaStartTime", DataItemRef.TYPE_STRING,
"BT(00:02:00)");
```

These are the time and day codes that you can use.

Time Codes

Code	Meaning	Example
AT	Sets the absolute time of the day.	AT(16:30:00): 4:30pm on that day.
CT	Sets the business time relative to the closing time of the day. Allowed values: 00 or negative hours. Typically you will use negative hours with closing time in order to calculate a relative time before closing time.	CT(00): Closing time. CT(-02:00:00): 2 hrs before the closing time.
OT	Sets the business time relative to the opening time of the day. Allowed values: 00 or positive hours.	OT(00): At the opening time. OT(02:00:00): 2 hrs after the opening time.
BT	Sets the business time relative to the current time of the day.	BT(04:30:00): After 4 & 1/2 business hours from the current time. BT(-02:00): 2 business hours earlier. BT(00): Use this to search forward to the next business time, without changing the time if the current time is not a business time. You may need this if different business days have different hours. BT(-00): Use this to search backward to the last previous business time. Has no effect if the current time is already during business time.

Day Codes

Code	Meaning	Example
BD	Sets a business day.	BD(4): Four business days from today. BD(0): Same day if it is a business day, else the next day. BD(-0): Same day if it is a business day, else the previous day.
RD	Sets a relative day from the current day.	RD(7): After one week. RD(-1): One day earlier.
WD	Sets a day of the week. Allowed values: 1 to 7.	WD(1): Sunday of that week. WD(7): Saturday of that week.

Code	Meaning	Example
WN	Sets the next weekday after today. Allowed values: 1 to 7.	WN(1): The next Sunday after today. WN(7): The next Saturday after today.
RM	Sets a relative month in the future. If the month does not have enough days to be the same day, the day will be the last day of the month.	RM(3): After 3 months.
DM	Sets an exact day of the month. Allowed values: Any number except 0.	DM(1): The first day of the month. DM(-1): The last day of the month.
BM	Sets an exact business day of the month. Allowed values: Any number except 0.	BM(1): The first business day of the month. BM(-1): The last business day of the month.
DY	Sets a day of the year. Allowed values: Any number except 0.	DY(1): The first day of the year. DY(-1): The last day of the year.
BY	Sets a business day of the year. Allowed values: Any number except 0.	BY(1): The first business day of the year. BY(-1): The last business day of the year.

6.20 Process Scheduler

Process Scheduler is a periodic, workflow application-level timer that starts process instances of specified process definitions contained in a workflow application based on the schedule set in the timer. The timer is configured using the ProcessScheduler.xml file, which should be placed in the <DMSRoot>/apps/<workflow applicationName>/ folder.

Apart from setting the schedule in ProcessScheduler.xml, you also specify the Business Calendar the timer should use. Optionally, you can also specify an expiration date after which the timer will no longer be valid. This expiration date can be extended and timer will execute till new extended period.

Note the following points about Process Schedulers:

- For a workflow application you can configure multiple timers using that workflow application's ProcessScheduler.xml file. Each timer can be configured to start multiple process definitions.
- A workflow application must be in online state for its Process Scheduler to work properly.
- Any process definitions specified within a timer must be in published state and present in the workflow application for which the ProcessScheduler.xml has been defined.
- A Process Scheduler logs exception, audit messages in the server log file. The states of the timers are also displayed in this log file.

6.20.1 Defining and Using a Process Scheduler

The only way to create Process Schedulers is through the ProcessScheduler.xml file. This file can either be created in Interstage BPM Studio, or it can be created manually.

Note: You cannot create Process Schedulers using APIs. You can, however, retrieve Process Scheduler timer instance information using APIs. For details, refer the **Retrieving Process Scheduler Timer Instance Information** section below.

To define Process Scheduler manually:

1. Create the `ProcessScheduler.xml` file using the format specified below. Note that the file name is case sensitive.
2. Place it in the `<DMSRoot>/apps/<workflow applicationName>/` folder
3. Start the workflow application.

To edit or delete a timer in the Process Scheduler:

1. Stop the workflow application that contains the timer.
2. Edit or delete the timer in the `ProcessScheduler.xml` file.
3. Re-start the workflow application.

When a workflow application is started, the process scheduler timer instance is:

- Created in the database if it has not been created earlier but exists in `ProcessScheduler.xml`.
- Updated in the database if it exists in both `ProcessScheduler.xml` and the database.
- Deleted from the database if it exists in the database and not in the ProcessScheduler.xml.

ProcessScheduler.xml Details

The format of the `ProcessScheduler.xml` file is as shown in the following sample.

Note: All tag names are case-sensitive.

```
<ProcessScheduler>
   <Timers>
      <Timer>
        <Name>CheckClearanceTimer</Name>
          <ProcessDefinitions>
            <ProcessDefinition>CheckClearanceProcess1</ProcessDefinition>

            <ProcessDefinition>CheckClearanceProcess2</ProcessDefinition>

          </ProcessDefinitions>
          <Calendar>MyCalendar</Calendar>
          <Schedule>BT(05:00:00)</Schedule>
          <ExpirationDate>2015/12/31</ExpirationDate>
      </Timer>
      <Timer>
        <Name>CashTransferTimer</Name>
          <ProcessDefinitions>
            <ProcessDefinition>CashTransferProcess</ProcessDefinition>
          </ProcessDefinitions>
          <Calendar>MyCalendar</Calendar>
          <Schedule>WN(6);CT(-01:00:00)</Schedule>
          <ExpirationDate>2015/12/31</ExpirationDate>
      </Timer>
   </Timers>
</ProcessScheduler>
```

The tag details are as below.

Tag	Description
<Name>	Unique name of the timer which must not contain any of following characters: \ | / : * ? " < > and must not be longer than 64 characters. For example, MyProcessScheduler. This is a mandatory tag.
<ProcessDefinition>	Name of the process definition in this workflow application whose process instance is to be started by the process scheduler. State of specified process definition must be published. For example, MyProcessDefinition. This is a mandatory tag.
<Calendar>	Name of the business calendar file, without the .cal extension. The system uses Default when no calendar is specified. For example, MyBusinessCalendar. This is an optional tag.
<Schedule>	Periodic timer specified using business calendar format. For example, WN(1);BT(01:00:00) Refer section *Time and Day Codes for Timers* on page 172 for a list of time and day codes. As listed in the restrictions below, certain time and day codes and combinations may lead to errors. This is a mandatory tag.
<ExpirationDate>	Expiration date of this timer. The timer will not be scheduled if this date has lapsed. Expiration date will be used from the time zone specified in the business calendar file. Format: yyyy/mm/dd For example 2020/12/31 This is an optional tag. If this tag is not specified, the timer will continue to be valid for infinite time.

Retrieving Process Scheduler Timer Instance Information

You can retrieve a list of all Process Scheduler timer instances for a workflow application using the getProcessSchedulerTimers(String applicationId) method of the com.fujitsu.iflow.model.workflow.WFSession interface.

For each timer instance, you can use the following methods of the `com.fujitsu.iflow.model.workflow.ProcessSchedulerTimer` interface:

- `getName()`
- `getState()`
- `getLastExecutionTime()`
- `getNextScheduledTime()`

The sample code to retrieve Process Scheduler timer instance information for a workflow application is as follows:

```
//Log in to Interstage BPM Server, that is, create a WFSession object.
//Consider 'wfSession' to be the WFSession object.

//Retrieve the list of Process Scheduler timer instances
ProcessSchedulerTimer []psTimers =
  wfSession.getProcessSchedulerTimers("LoanApplication");

//Retrieve information for each timer instance
   for (int i = 0; i < psTimers.length; i++) {
        System.out.println("Name: " + psTimers[i].getName());
        System.out.println("State: " + psTimers[i].getState());

        long lastExecutionTime = psTimers[i].getLastExecutionTime();
        if (lastExecutionTime != -1) {
            System.out.println("LastExecutionTime:" + lastExecutionTime);
        }

        long nextScheduledTime = psTimers[i].getNextScheduledTime();
        if (nextScheduledTime != -1) {
            System.out.println("NextScheduledTime: " + nextScheduledTime);

        }

   }
```

For details of these methods, refer the API Javadoc.

Restrictions

Note the following restrictions when using Process Schedulers:

1. If a periodic timer instance is configured in `ProcessScheduler.xml` for which the difference between the 'next-fire time' and 'previous-fire time' is less than 1 minute, then the first timer instance will be executed but the next timer instance will not be scheduled, and will be put into error state. The error message will be logged into the server log file.

   This error can be recovered by stopping the workflow application, updating the schedule to correct code which will result in more than one minute interval between two timer executions, and then restarting the workflow application.

2. Workflow applications will not be started and no timer instances created if `ProcessScheduler.xml`:
   - contains invalid parameters (such as timer name containing `\ | / : * ? " < >` characters, and so on)
   - does not contain mandatory tags
   - contain mandatory tags that are empty or contain an empty string
   - specifies a business calendar file that does not exist within the workflow application

3. If a process definition specified in `ProcessScheduler.xml` does not exist or does not have a published version, then this timer instance will log a WARN message in the server log file specifying that the process definition does not exist. The timer instance will start the process instance for those process definitions which are present and published and then schedule next timer instance till the Expiration date is not reached.
4. While all time and day codes mentioned in section *Time and Day Codes for Timers* on page 172 can be used for Process Scheduler, use them carefully since for some codes and combinations whenever the next expiration time is calculated using the business calendar, the next and previous expiration time will be same, leading to error. For example:
   - `WD(X);AT(XX:XX:XX)`
   - `CT(-XX:XX:XX)`

Missed Events

Missed events for timers are generated and the timers fired for these events in any of the following cases:

- If a (valid or expired) timer is updated and the workflow application is re-started.

  For example: A timer was last fired at 2:00 pm and is scheduled to fire at 4:00 pm. At 2:30 pm, the workflow application is stopped and timer is updated to fire every hour instead. If the workflow application is restarted at 5:30 pm, there will be three immediate timer executions for missed timer events corresponding to 3:00pm, 4:00 pm and 5:00 pm.
- If a workflow application is stopped (either manually, or for other reasons such as tenant-deactivation, server stop) and then restarted.

  For example: A timer was last fired at 2:00 pm and is scheduled to fire at 4:00 pm (every 2 hours). At 2:30 pm, the workflow application is stopped. Then the workflow application is started at 6:30 pm. In this case the timer will immediately be fired for each missed timer instance, that is, timer will be executed for 4:00 pm, 6:00 pm and then schedule for 8:00 pm.

Timer Instance States in Server Log File

Timer instance states (used in the log file) are as follows. This information is helpful for troubleshooting Process Scheduler-related errors.

- **`NotHandled`**: implies the Timer is configured and will fire at the set time.
- **`workflow applicationStopped`**: implies the Timer will not fire since the workflow application has been stopped.
- **`Error`**: implies the Timer will not fire due to an error condition, for example, when 'next evaluate time' - 'first evaluate time' is less than 1 minute.
- **`Handled`**: implies the Timer will not be scheduled and not be fired since it has expired. This state also occurs if a timer is updated and the new scheduled time is beyond the expiration date. A `Handled` timer can be re-used by updating its expiration date to be later than the current date.

6.21 Modeling Remote Subprocesses

Remote subprocesses are subprocesses that run on a remote workflow server. A remote workflow server might be, for example, another Interstage BPM Server.

A remote subprocess is represented by a Remote Sub-Process Node in the parent process definition. The interaction between parent and remote subprocess comprises the following steps:

1. When a Remote Sub-Process Node is reached, the local workflow server sends a Start Process request to the remote workflow server, carrying with it the User Defined Attribute (UDA) values that the remote subprocess instance will work on.
2. The remote workflow server must receive this request, and start a process instance.
3. When the remote process instance completes, its workflow server sends a Process Completed message back, carrying with it the results of the subprocess instance.
4. The local workflow server must receive this message, incorporate the results, and complete the Remote Sub-Process Node allowing the process instance to continue to the next node.

For a successful interaction, the following must be defined with the parent process definition:

- The URI of the remote process definition
- The identifiers of the UDAs that will be passed back and forth
- Return values of the subprocess definition.

6.21.1 Designing a Parent and Remote Subprocess Definition

This section explains how to design the parent and the subprocess definition. It explains the central steps based on the following sample process definitions:

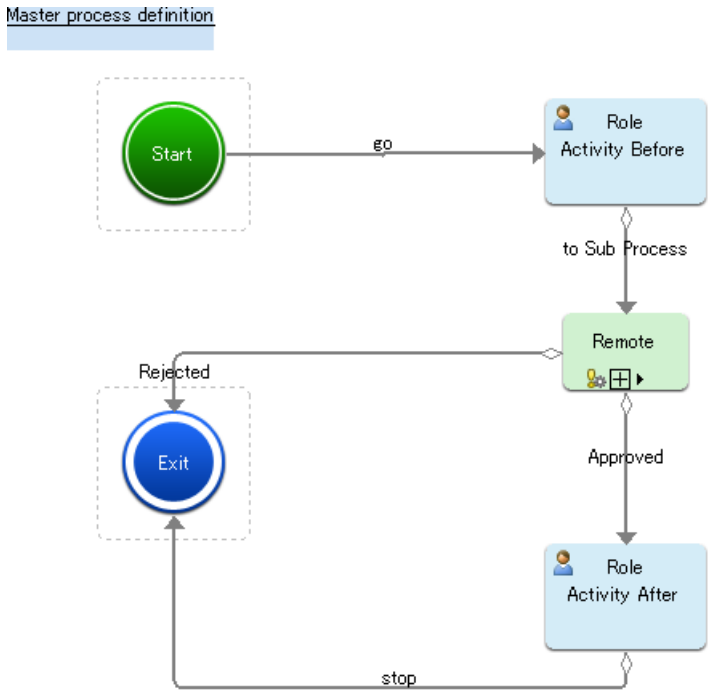

Figure 17: Parent Process Definition with a Remote Sub-Process Node

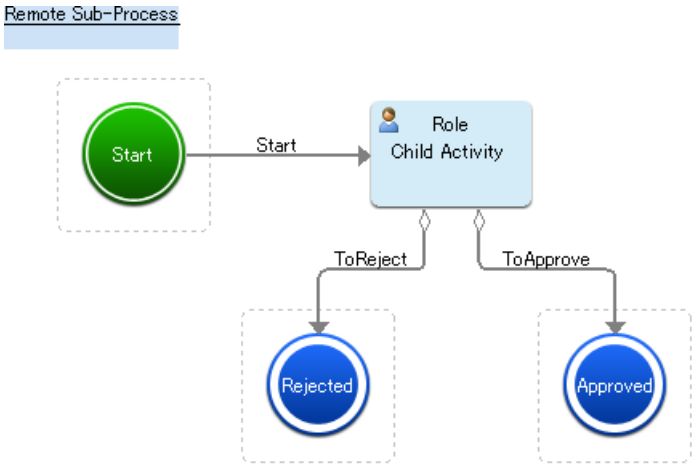

Figure 18: Remote Subprocess Definition

You can find the complete programming code in the sample file `RemoteSubProcess.java`.

To design a parent and a remote subprocess definition:

1. Create the subprocess definition. Add a Start Node, one or more End Nodes and the required User Task Nodes to the subprocess definition. Also, specify the UDAs that will be passed back and forth.

   In the sample program, `createRemoteSubPlan()` is called to create a subprocess definition:

   ```
   Plan subPlan = createRemoteSubPlan();
   ```

   The sample subprocess definition consists of a Start Node, a User Task Node and two End Nodes. The End Nodes define the possible results of the subprocess definition that need to be handled by the parent process definition:

   ```
   Node exitNodeApp = plan.addNode("Approved", Node.TYPE_EXIT);
   exitNodeApp.setUpperLeftPoint(new Point(500, 350));
   Node exitNodeRej = plan.addNode("Rejected", Node.TYPE_EXIT);
   exitNodeRej.setUpperLeftPoint(new Point(200, 350));
   ```

   The following UDAs are created in the sample subprocess definition:

   ```
   plan.addDataItemRef("Name",
         DataItemRef.TYPE_STRING, "John Smith");
   plan.addDataItemRef("Number",
         DataItemRef.TYPE_STRING, "0123456789");
   plan.addDataItemRef("Amount",
         DataItemRef.TYPE_FLOAT, "0.0");
   plan.addDataItemRef("Limit",
         DataItemRef.TYPE_FLOAT, "0.0");
   ```

2. Create the parent process definition and set it to edit-mode.

   ```
   plan = WFObjectFactory.getPlan();
   plan.setWFSession(adminSession);
   plan.startEdit();
   ```

3. In the parent process definition, create the UDAs that will be passed back and forth. In the sample program, the following UDAs are created:

   ```
   plan.addDataItemRefWithId("Name", "Name",
         DataItemRef.TYPE_STRING, "John Smith");
   plan.addDataItemRefWithId("SSN", "SSN",
         DataItemRef.TYPE_STRING, "0123456789");
   plan.addDataItemRefWithId("LoanAmount", "Loan Amount",
         DataItemRef.TYPE_FLOAT, "0.0");
   plan.addDataItemRefWithId("ApprovalLimit", "Approval Limit",
         DataItemRef.TYPE_FLOAT, "0.0");
   ```

   Note: You can use different names for the same UDA in the process definitions involved. However, the data type must be identical.

4. Add a Start Node, one or more End Nodes and the required User Task Nodes to the parent process definition.

5. Add a Remote Sub-Process Node to the parent process definition.

```
Node RemoteSubProcessNode = plan.addNode("Remote",
   Node.TYPE_REMOTE_SUB_PROCESS);
RemoteSubProcessNode.setUpperLeftPoint(new Point(400, 300));
```

6. Define data mappings for the UDAs that need to be passed back and forth. Use the following method of the `Node` interface:

```
addDataMappingElement(<UDA identifier in parent process definition>,
     <UDA identifier in subprocess definition>,<direction of data flow>)
```

   When the data value passes from the parent to the remote subprocess, specify `DataItemMappingElement.IN` as the direction of data flow. When the data value passes from the remote subprocess to the parent process, specify `DataItemMappingElement.OUT`. When the data value passes is both directions, specify `DataItemMappingElement.INOUT`.

   In the sample program, the following data mappings are defined:

```
RemoteSubProcessNode.addDataMappingElement("Name","Name",
      DataItemMappingElement.IN);
RemoteSubProcessNode.addDataMappingElement("SSN","Number",
      DataItemMappingElement.IN);
RemoteSubProcessNode.addDataMappingElement("LoanAmount", "Amount",
      DataItemMappingElement.IN);
RemoteSubProcessNode.addDataMappingElement("ApprovalLimit", "Limit",
      DataItemMappingElement.OUT);
```

7. Connect the parent and subprocess definition using the following method. To do so, specify the communication protocol to be used by the workflow servers involved and the URI of the remote subprocess definition:

```
RemoteSubProcessNode.setSubPlanURI("asap:" +
subPlan.getPlanURI());
```

   Interstage BPM supports two open protocols for communication between workflow servers: Simple Workflow Access Protocol (SWAP) and Asynchronous Service Access Protocol (ASAP). These protocols pass XML messages over HTTP between workflow servers.

   You are recommended to use ASAP (specified as `asap`) when the processes run on integrated Interstage BPM Servers. Use SWAP (specified as `swap`) with Collaboration Ring only. For details about Collaboration Ring integration, contact your local Fujitsu Support Organization.

8. Add arrows to connect the nodes in the parent process definition. Make sure to add an outgoing arrow to the Remote Sub-Process Node for each result that the subprocess might return.

   The result values correspond to the names of the End Nodes in the remote subprocess definition.

   A Remote Sub-Process Node may have one or more outgoing arrows. Only one arrow is chosen when the parent process resumes. The arrow that is chosen is the one that matches the result of the remote subprocess. In the sample program, the subprocess returns either `Approved` or `Rejected`.

```
Arrow ExitSubProcessApp = plan.addArrow("Approved", RemoteSubProcessNode,
 activityNodeAfter);
Arrow ExitSubProcessRej = plan.addArrow("Rejected", RemoteSubProcessNode,
 exitNode);
```

Note: The names of the End Nodes in the subprocess definition and the names of the outgoing arrows must be identical, also regarding uppercase/lowercase.

6.21.2 Running Remote Subprocess Definitions

To be able to run remote subprocess definitions, an Interstage BPM Linkage User must be configured on the local and the remote Interstage BPM Server:

- On the local Interstage BPM Server, a user must be specified that can be authenticated on the remote Interstage BPM Server. This user will create the subprocess instance on the remote Interstage BPM Server.
- On the remote Interstage BPM Server, a user must be specified that can be authenticated on the local Interstage BPM Server. This user will return the result of the remote subprocess instance to the local Interstage BPM Server.

Note: InterstageBPM will be able to start the remote subprocess only if the Interstage BPM Linkage User is present in the Interstage BPM tenant on the remote server as well as present in the Interstage BPM tenant on the local server.

The Interstage BPM Linkage User is configured in the parameters `SWAPLinkageUserName` and `SWAPLinkagePassword` of the Interstage BPM Server. For more information, refer to the *Interstage Business Process Manager Server Administration Guide*.

Only published subprocess definitions can be called remotely unless the subprocess definition belongs to the Interstage BPM Linkage User. The Interstage BPM Linkage User can call remote subprocess definitions that are in Draft state. Therefore, use one of the following options to test your subprocess definitions:

1. Publish the remote subprocess definition before running it.
2. Find out the Interstage BPM Linkage User configured for the local Interstage BPM Server; use it as the owner of the subprocess definition.
3. Change the Interstage BPM Linkage User configured for the local Interstage BPM Server; specify the owner of the subprocess definition as the Interstage BPM Linkage User.

6.21.3 Error Handling for Remote Subprocesses

You can handle the case when the starting of a remote subprocess fails (e.g. because the remote server is not started) by defining an Error Java Action for the Remote Sub-Process Node in the parent process definition. In this way you can avoid that the process instance goes into error state as soon as the subprocess cannot be started.

An Error Action Set on the Process level of remote subprocess will become executed if the remote subprocess fails to start. Refer to section *Using Error Java Actions* on page 103 for more information on this type of Java Action.

To design and assign an Error Action Set to a Remote Sub-Process Node:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet errorJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

In the following example, one Java Action is generated.

```
JavaAction[] errorJavaActions =
   errorJavaActionSet.createJavaActions(1);
```

3. Define the Java Action Set.

```
errorJavaActions[0]
    .setActionName("WriteLogEntry");
errorJavaActions[0]
    .setActionDescription("Writes a log entry in case the Remote
                                Subprocess could not be started");
errorJavaActions[0]
    .setMethodName("writeLogEntryWithException(String,ServerEnactmentContext)");
errorJavaActions[0]
    .setClassName(CLASS_NAME_JAVA_ACTION);
errorJavaActions[0]
    .setArgumentsUDANames("<E>uda.LogEntry</E><E>sec</E>");
errorJavaActionSet.setJavaActions(errorJavaActions);
```

4. Copy the contents of this `JavaActionSet` into the Remote Sub-Process Node.

```
RemoteSubProcessNode.setJavaActionSet(errorJavaActionSet,
    JavaActionSet.NODE_ERROR);
```

6.22 Using Embedded Sub-Process Nodes

An Embedded Sub-Process node encompasses a phase of a process and the milestones to be achieved at the end of the phase. A phase is a container that contains various nodes and arrow transitions. Once a phase is complete a milestone in the process is said to be achieved and the process can move to the next node or another phase. You can also define due date for a phases. This due date can be defined using different types of Timers.

The Embedded Sub-Process node has child nodes nested inside it. It should have Start and End nodes as child nodes. Refer to *Defining an Embedded Sub-Process Node* on page 185 for more information on how to define an Embedded Sub-Process node.

The Embedded Sub-Process node and the nodes nested inside an Embedded Sub-Process node will have the same Process Definition ID as the parent process. The nodes nested inside the Embedded Sub-Process node cannot be accessed outside this node.

The following figure illustrates an example of an Embedded Sub-Process node.

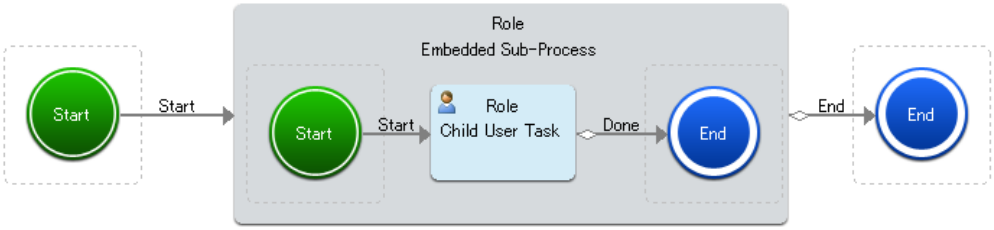

Figure 19: Embedded Sub-Process Node

Workflow for Embedded Sub-Process Nodes

When the control reaches the Embedded Sub-Process node in a process,

1. Workitem for the Embedded Sub-Process is created and WorkItem state is changed to `WaitingForSubProcess`.
2. The child Start node is activated.

Note: An Embedded Sub-Process node must have Start and End nodes inside it.

3. The status of Embedded Sub-Process Node is changed to `WaitingOnSubProcess` and the child nodes will be executed.
4. When the Child End node is reached
   - The Embedded Sub-Process Node is closed and its state is changed to `Completed`.
   - Workitems of the Embedded Sub-Process Node are deleted.
   - Outgoing arrow from the Embedded Sub-Process node that has the same name as child End node is activated.

Note: Make sure that the child End node name matches any one of the outgoing arrow names from the Embedded Sub-Process node.

Note: When the work item for Embedded Sub-Process is completed by making choice, then any active child node instance will be aborted, its Work items are deleted and the Embedded Sub-Process state is changed to `Completed`.

For more information on functions that are supported on Embedded Sub-Process node, refer to information in *Node Types* on page 34.

6.22.1 Defining an Embedded Sub-Process Node

Pre-requisites

The Process Definition to which you are adding an Embedded Sub-Process node should be in the edit mode.

To define an Embedded Sub-Process Node

1. Create an Embedded Sub-Process node using `addNode()`. Set the constant `nodeType` to `TYPE_COMPOUND_ACTIVITY`

```
 Node activity = plan.addNode("CompoundActivity",
Node.TYPE_COMPOUND_ACTIVITY);
```

   Following sample code adds Embedded Sub-Process `CompoundNode` to the process Definition `Plan`

```
 // Adding Embedded Sub-Process Node
Node compoundNode =
plan.addNode("CompoundNode",Node.TYPE_COMPOUND_ACTIVITY);
compoundNode.setRole("SampleGroup");
compoundNode.setSize(new Dimension(200,100));
```

2. Add child nodes using the `addChildNode()` API of the Interface `com.fujitsu.iflow.model.workflow.Node`

Following sample code adds child nodes to the Embedded Sub-Process node.

```
 //Adding child nodes
Node subStartNode = compoundNode.addChildNode("SubStart",
Node.TYPE_START);
Node subActivityNode =
compoundNode.addChildNode("SubActivity",Node.TYPE_ACTIVITY);
subActivityNode.setRole("SampleGroup");

//End Node name should be same as Embedded Sub-Process's outgoing Arrow
 name
Node subExitNode = compoundNode.addChildNode ("CommonName",
Node.TYPE_EXIT);
```

3. Add arrows for the child nodes. Make sure that name if an outgoing arrow is same as the End node name under the Embedded Sub-Process Node

   The following sample code adds arrows to connect various Child Nodes

```
 plan.addArrow("SubArrow1", subStartNode, subActivityNode);
 plan.addArrow("SubArrow2", subActivityNode, subExitNode);
```

   The following sample code adds an outgoing arrow with the same name as the End node.

```
 plan.addArrow("CommonName", compoundNode, actNode);
```

Note: You cannot define arrow transitions from child nodes inside Embedded Sub-Process to nodes outside Embedded Sub-Process.

6.23 Using Dynamic Subtasks

When a task is activated, it might be required to create subtasks to that task and assign these subtasks to different users, in order to complete the task. Creating dynamic subtasks to a task enables you to assign these subtasks to different users.

To add a subtask to an active task (Activity1), a dynamic child node instance needs to be created and assigned. The workitem that is generated for this dynamic child node instance becomes the subtask for Activity1.

Note: To add a subtask to a task, the task should be either in the `Active` or `WaitingForSubProcess` states.

When you add node instance to an activity, the task and the associated node instances states are changed to `WaitingforSubProcess`. The dynamic child node instance and associated subtasks states are changed to `Active`. When all the subtasks of a task are complete, the task is said to be complete.

Note: The workitems of dynamic child node instance is deleted when dynamic node instance is completed or aborted.

You can also add dynamic child node instances to existing dynamic child node instances thus, nesting dynamic child node instances.

A dynamic child node instance contains:

- Name

- Description
- Priority
- Comments
- Owner
- Order (The order of task that can be used when listing)
- Assignee
- Point (The position of task that can be used when displaying on screen)

You can set due date on a dynamic child node instance.

You cannot set the following on dynamic child node instances and subtasks:

- Java Action
- Javascript
- Timers
- Triggers
- Iterator Count
- Recall Flag
- Transaction Flag
- Subprocess
- Chained Process
- Remote Subprocess
- Arrows (When creating dynamic child node instance, the arrow named complete is created as default.)
- Forms

To add subtasks to tasks:

The following steps demonstrate adding of dynamic child node instance (newNodeInst) to a node instance (nodeInstance).

1. Start a process instance and activate a node instance. For example, nodeInstance is activated.
2. On an active node instance, create a dynamic child node instance (for example, newNodeInst) using the API addChildDynamicNodeInstance(). Refer to the sample code below.

```
// create dynamic node instance
ProcessInstance procInst = … // Get process instance
String[] assignees = {"user1","user2"};
procInst.startEdit();
NodeInstance newNodeInst = nodeInstance.addChildDynamicNodeInstance("Sub
 Task",assignees) ;
newNodeInst.setDesc("Description of this sub task");
newNodeInst.setOrder(10);
newNodeInst.setPriority(Node.PRIORITY_MEDIUM);
procInst.commitEdit();
```

Note: addChildDynamicNodeInstance() API belongs to the NodeInstance of com.fujitsu.iflow.model.workflow package. Refer to the API JavaDoc for more information.

Now, newNodeInst is in `Active` state and nodeInstance is in `WaitingForSubProcess` state.

The workitem that is generated for newNodeInst becomes the subtask for workitem generated for nodeInstance.

Note: You can get the parent node instances and dynamic child node instances using the APIs `getParentNodeInstance()` and `getChildNodeInstances()`.

6.24 Using Dynamic Processes

During execution of a process instance or a workitem, you might have the need to complete some activities that are not part of the original process definition. In other words, you might want to create user task nodes dynamically at run-time and complete their associated tasks when these activities are not part of the original plan.

Dynamic process instance is a process instance that is created without being associated with any process definition. Dynamic processes do not have any of arrow transitions. They are just a collection of tasks that need to be completed for the dynamic process to be completed.

To create a dynamic process instance use the `WFObjectFactory.createDynamicNodeInstance()` method.

Note: When you create an application space, application ID is defined when `WFAdminSession.createApplicationSpace(String applicationId)` is executed. To create dynamic process, it is necessary to know the application ID for the application in which dynamic process is to be created. You must use `WFSession.chooseApplication()` to determine the application ID before `WFObjectFactory.createDynamicNodeInstance()` is executed to create dynamic process.

When you execute `WFSession.chooseApplication()` method, it is necessary to specify the `applicationId` other than the `System`.

To create a dynamic process and generate dynamic node instances

The following steps demonstrate creating a dynamic process instance and generating node instances and tasks.

1. Create and start a dynamic process instance using `WFObjectFactory.createDynamicNodeInstance()` API. Refer to the following sample code that creates a dynamic node instance called Root Task.

```
 // create dynamic process instance
NodeInstance nodeInst =
WFObjectFactory.createDynamicNodeInstance(wfSession, "Root Task");
```

2. Create node instances for this dynamic process instance. The dynamic process instance name and node instance name are set as same value. You can modify name and other attributes. Refer to the sample code below.

```
 // create dynamic node instance
 ProcessInstance procInst = nodeInst.getProcessInstance();
procInst.startEdit();
nodeInst.setName("Modified task name");
nodeInst.setDesc("Description of this sub task");
nodeInst.setOrder(10);
nodeInst.setPriority(Node.PRIORITY_MEDIUM);
procInst.commitEdit();
```

Note: After a dynamic node instance is created, the process instance creator is set as node instance owner, and work items are assigned to the node instance owner.

3. On this active dynamic node instance, add child dynamic node instances. The dynamic node instance state changes to `WaitingForSubProcess` and the child dynamic node instance becomes `Active`. Refer to the sample code below.

```
 procInst.startEdit();
String[] assignees = {"user01","user02"};
nodeInst.addChildDynamicNodeInstance("NewTask", assignees);
nodeInst.setDesc("Description of this sub task");
nodeInst.setOrder(10);
nodeInst.setPriority(Node.PRIORITY_MEDIUM);
procInst.commitEdit();
```

6.25 Decision Tables

Decision Tables allow you to design advanced rules for decision making without programming. Decision Tables use a simple yet powerful table based approach for managing rules dynamically. They avoid the need to learn, develop, and support a complex rules engine infrastructure.

Decision tables (.dt files) can exist independently of any process definition. This allows the designing of decision tables to be done by business users using business terms, and as such, it can be abstracted out from the more technical activity of creating a process definition. A single decision table can be used by any number of processes, and a process can use any number of decision tables.

Decision tables are versioned independently from processes as well. A single version of a decision table might be used by any number of versions of a process instance. When this is the case, a change to the decision table will change the behavior of all versions of the process. You can also have multiple versions of a decision table by giving each version a slightly different name. In this case, when you install a new version of a process definition that uses a new decision table into the server, the old versions of the process will continue to use the old decision table. In this way, any number of versions of the decision table can be running at the same time in the server, each tied appropriately with the coordinated version of the process definition.

6.25.1 Decision Table Concepts

A Decision Table is defined using the following:

- A list of input variables, called Condition Criteria. For eg., '`Deal Amount`'
- A list of output variables, called Result Locators. For eg., '`Discount Rate`'
- Conditions, which are pre-defined values with conditions for an input variable. For eg., '`Deal Amount > 5000`' or '`Deal Amount <= 3000`'
- Results, which are pre-defined values for an output variable. For eg., '`Set Discount Rate to 35`'
- Decisions, or rules, which map different conditions to different results. For eg, "`If the condition 'Deal Amount is > 5000' is found to be true, the result 'Set the Discount Rate to 35' should be applied`".

Depending on what values the decision table receives for its input variables (or Condition criteria), it evaluates if any of the listed conditions is met. For the condition that is met, it selects its corresponding pre-defined result, that is, it sets the specified values for its output variables (the result locators).

Creating a decision table mainly involves defining its condition criteria, result locators, conditions, results, and mapping the conditions to results, that is, defining a decision.

A sample Decision Table can conceptually be illustrated as follows:

Decision Table Name	My_Table.dt		
Description	Table description		
Condition Criteria			
Name	Description	Type	
CUSTOMER_LEVEL	description	STRING	
DEAL_AMOUNT	description	INT	
Result Locators			
Name	Description	Type	
DISCOUNT_RATE	description	INT	
SALES_MAN_BONUS	description	INT	
Decisions			
Conditions		Results	
CUSTOMER_LEVEL	DEAL_AMOUNT	DISCOUNT_RATE	SALES_MAN_BONUS
GOLD	> 500000	35	35
SILVER	> 500000	25	25
REGULAR	>= 500000	15	15

Figure 20: Decision Table concept

In the example above, `CUSTOMER_LEVEL`, `DEAL_AMOUNT` are the input variables or condition criteria. `DISCOUNT_RATE`, `SALES_MAN_BONUS` are the pre-defined output variables or result locators. The different decisions listed are:

- If `CUSTOMER_LEVEL` is `GOLD` and `DEAL_AMOUNT` is greater than `500000`, set the `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `35`
- If `CUSTOMER_LEVEL` is `SILVER` and `DEAL_AMOUNT` is greater than `500000`, set the `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `25`
- If `CUSTOMER_LEVEL` is `REGULAR` and `DEAL_AMOUNT` is greater than or equal to `500000`, set the `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `15`

The decision table is evaluated in a top-down order. Result values are set as per the table for the first condition that is found to be met.

For example, if the `CUSTOMER_LEVEL` is `SILVER`, and the `DEAL_AMOUNT` is `800000`, the first condition is found not to be met. The second condition is found to be met, so the `DISCOUNT_RATE` and `SALES_MAN_BONUS` are set to `25`. Since the second decision has already been selected, the third and fourth decisions are not evaluated.

Data Dictionary

Condition criteria have a feature called the Data Dictionary. This allows storing of synonym values for any pre-defined value of a condition criteria. For example, consider a condition criterion called 'Country' and the condition created is 'Country = Japan'. Pre-defining allowable alternatives for 'Japan', such as 'JP','JPN', etc. will ensure the condition 'Country = Japan' is found to be met even if the end-user enters 'JP' or 'JPN' instead of 'Japan'.

The Data Dictionary feature makes Decision Tables more flexible.

6.25.2 Using a Decision Table within a Process Definition

To use a decision table in a process definition, a pre-defined Java Action to evaluate Decision Tables (called the Decision Table Java Action) is provided. The variables of the decision table (condition criteria and result locators) then need to be mapped to UDAs of the process definition. For details on how to do this, refer the *Interstage BPM Studio User's Guide* and/or the *Interstage BPM Console Online Help*.

In a process instance, when the node of the Decision Tables Java Action is active, if the input UDA values match any of the conditions specified in the Decision Table, the Decision Table Java Action will set the values of the output UDAs as per the results of the Decision Table.

6.25.3 Decision Table Specifications

ConditionCriterion and ResultLocator

ConditionCriterion and ResultLocator contain following information:

- **Name:** Abstract names of ConditionCriteria and ResultLocators. These names are mapped to actual UDA names when defining a Decision Table Java Action.
- **Description:** A brief explanation of ConditionCriteria and ResultLocators. It does not affect behavior of the decision table.
- The type of ConditionCriteria and ResultLocators. Following types are supported.
  - BOOLEAN
  - INTEGER
  - LONG
  - FLOAT
  - BIGDECIMAL
  - STRING
  - DATE

Decisions

Decisions have following information:

- conditions for each ConditionCriterion
- values to assign to each ResultLocator

The following operators can be used within conditions:

- `=`, `!=` for exact match of any data type
- `>`, `<`, `>=`, `<=`, for greater/less than comparisons of numeric types
- `in(a,b,c)` for comparison to one of a number of literals
- `between(x,y)` for numeric range testing
- `LIKE()`, `NOT_LIKE()` for pattern matching expressions; question mark (`?`) will match any single character, and asterisk (`*`) will match any number of characters. For example, if `LIKE(?OLD)` is used in a condition, all values such as `GOLD`, `BOLD`, `TOLD` will satisfy this condition; if `LIKE(JAP*)` is used in a condition, all values such as `JAPAN`, `JAPANESE` will satisfy this condition.

Different operators are supported depending on the type of the condition criterion. See the following table:

Operator	Boolean	Integer	Float	Long	BigDecimal	Date	String
<	No	Yes	Yes	Yes	Yes	Yes	No
>=	No	Yes	Yes	Yes	Yes	Yes	No
<=	No	Yes	Yes	Yes	Yes	Yes	No
>=	No	Yes	Yes	Yes	Yes	Yes	No
!=	Yes	Yes	Yes	Yes	Yes	Yes	Yes
=	Yes	Yes	Yes	Yes	Yes	Yes	Yes
in(a,b,c)	No	Yes	Yes	Yes	Yes	Yes	Yes
between(x,y)	No	Yes	Yes	Yes	Yes	Yes	No
like	No	No	No	No	No	No	Yes
not_like	No	No	No	No	No	No	Yes

6.25.4 Managing Decision Tables

To manage Decision Tables, use the `com.fujitsu.iflow.decisiontable` package. It contains APIs for managing Decision Tables. For a detailed description of this package and its APIs, refer to the API Javadoc. The following important operations are discussed:

1. Creating decision tables
2. Creating condition criteria, result locators, and decisions
3. Validating decision tables
4. Evaluating decision tables
5. Saving decision tables
6. Loading decision tables

Creating Decision Tables

You can create decision table files by either using the API, or by creating an XML file which conforms to the Decision Table XML Schema.

Creating Decision Tables using API

You can create a Decision Table by using the `createDecisionTable()` method of the `DecisionTableFactory` class. Refer to the API Javadoc for details of this method. For example:

```
DecisionTable dt = DecisionTableFactory.createDecisionTable();
dt.setName("DecisionTableName");
dt.setDescription("example of decision table");
```

Creating Decision Tables manually by conformance to the XML Schema

If you want to create a complete decision table without using the APIs, ensure the Decision Table XML file (.dt file) you create conforms to the XML Schema for the .dt file, `DecisionTable.xsd`, stored in the `<engine directory>/client/DecisionTable` directory.

Creating condition criteria, result locators, decisions

Creating Condition Criteria

Use the `createConditionCriterion()` method of the `DecisionTable` class to create a new condition criterion. Refer to the API Javadoc for details of this method.

In the following example, `CUSTOMER_LEVEL` and `DEAL_AMOUNT` are the two condition criteria created. Also, for `CUSTOMER_LEVEL`, the Data Dictionary feauture is used to create the synonyms 'normal' and 'none' for the main condition criterion value 'Regular'.

```
ConditionCriterion criterion1 = dt.createConditionCriterion();
criterion1.setName("CUSTOMER_LEVEL");
criterion1.setType(DataItemRef.TYPE_STRING);
Dictionary dic = criterion1.createDictionary();
DictionaryItem regularDictionaryItem = dic.createDictionaryItem();
regularDictionaryItem.setWord("Regular");
String synonyms[] = new String[2];
synonyms[0] = "Normal";
synonyms[1] = "None";
regularDictionaryItem.setSynonyms(synonyms);

ConditionCriterion criterion2 = dt.createConditionCriterion();
criterion2.setName("DEAL_AMOUNT");
criterion2.setType(DataItemRef.TYPE_INTEGER);

ConditionCriterion criteria[] = new ConditionCriterion[2];
criteria[0] = criterion1;
criteria[1] = criterion2;
dt.setConditionCriteria(criteria);
```

Creating Result locators

Use the `createResultLocator()` method of the `DecisionTable` class to create a new Result Locator. Refer to the API Javadoc for details of this method. The following example creates the `DISCOUNT_RATE` and `SALES_MAN_BONUS` result locators.

```
ResultLocator locator1 = dt.createResultLocator();
locator1.setName("DISCOUNT_RATE");
locator1.setType(DataItemRef.TYPE_STRING);

ResultLocator locator2 = dt.createResultLocator();
locator2.setName("SALES_MAN_BONUS");
locator2.setType(DataItemRef.TYPE_STRING);

ResultLocator locators[] = new ResultLocator[2];
locators[0] = locator1;
locators[1] = locator2;
dt.setResultLocators(locators);
```

Creating Decisions

The example below creates the following decisions:

- If `CUSTOMER_LEVEL` = `'GOLD'`, and, `DEAL_AMOUNT` > `'500000'`, set `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `'35'`
- If `CUSTOMER_LEVEL` = `'SILVER'`, and, `DEAL_AMOUNT` > `'500000'`, set `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `'25'`

```
Decision decision1 = dt.createDecision();
Condition condition1 = decision1.createCondition();
```

```
condition1.setCriterionName("CUSTOMER_LEVEL");
condition1.setOperator(Condition.CONDITION_OPERATOR_EQUAL);
condition1.setValue("Gold");
Condition condition2 = decision1.createCondition();
condition2.setCriterionName("DEAL_AMOUNT");
condition2.setOperator(Condition.CONDITION_OPERATOR_GREATER_THAN);
condition2.setValue("500000");
Condition conditions1[] = new Condition[2];
conditions1[0] = condition1;
conditions1[1] = condition2;
decision1.setConditions(conditions1);
Result result1 = decision1.createResult();
result1.setLocatorName("DISCOUNT_RATE");
result1.setValue("35");
Result result2 = decision1.createResult();
result2.setLocatorName("SALES_MAN_BONUS");
result2.setValue("35");
Result results1[] = new Result[2];
results1[0] = result1;
results1[1] = result2;
decision1.setResults(results1);

Decision decision2 = dt.createDecision();
... //add code for creating decision2, similar to creating decision1
...

Decision decisions[] = new Decision[2];
decisions[0] = decision1;
decisions[1] = decision2;
dt.setDecisions(decisions);
```

Validating Decision Tables

A decision table should be validated before it can be evaluated.

Validating a decision table involves some basic checks such as:

- Ensuring the Decision Table name is specified
- Ensuring ConditionCriterion name is specified
- Ensuring ConditionCriterion type is specified, and so on

Use the `validate()` method of the `DecisionTable` class to validate a decision table. Refer to the API Javadoc for details of these methods.

```
ValidationResult[] validationResults = dt.validate();
System.out.println(validationResults.length + " Problems detected.");
for (int i = 0; i < validationResults.length; i++) {
  ValidationResult validationResult = validationResults[i];
  System.out.println("Severity = " + validationResult.getSeverity());
  System.out.println("ID = " + validationResult.getIdentifier());
  System.out.println("Problem = " + validationResult.getProblem());
}
```

Evaluating a Decision Table

Use the `evaluate()` method of the `DecisionTable` class to evaluate or test a decision table with sample input values. Refer to the API Javadoc for details of this method.

Note that to successfully evaluate a decision table, the decision table must be valid. If `validate()` returns any results whose severity is `ERROR`, evaluation of the decision table fails.

The following example sets `CUSTOMER_LEVEL` to a sample value of '`SILVER`' and `DEAL_AMOUNT` to a sample value of '`800000`'. The decision table is then tested using these sample values, and the result values are set accordingly.

```
Properties conditionValues = new Properties();
conditionValues.setProperty("CUSTOMER_LEVEL","Silver");
conditionValues.setProperty("DEAL_AMOUNT","800000");
DecisionResult decisionResult = newDt.evaluate(conditionValues);

int decisonIndex = decisionResult.getExecutedDecision();
Decision allDecisions[] = newDt.getDecisions();
Decision selectedDecision = allDecisions[decisonIndex];

Result results[] = decisionResult.getResults();
for (int i = 0; i < results.length; i++){
  Result result = results[i];
  String locatorName = result.getLocatorName();
  String value = result.getValue();
  System.out.println("result : " + locatorName + " = " + value);
}
```

Saving a Decision Table

Use the `save()` method of the `DecisionTable` class to evaluate or test a decision table with sample input values. Refer to the API Javadoc for details of this method.

```
FileOutputStream outFile = new FileOutputStream("/decisiontable.dt");
dt.save(outFile);
```

Decision tables are application-specific, and are to be stored in the `<DMSRoot>/apps/<application ID>/rule/<category>` directory.

Loading an existing Decision Table

Use the `load()` method of the `DecisionTable` class to load an existing decision table. Refer to the API Javadoc for details of this method.

```
DecisionTable newDt = DecisionTableFactory.createDecisionTable();
FileInputStream inFile = new FileInputStream("/decisiontable.dt");
newDt.load(inFile);
```

API Sample for Decision Tables

For a complete sample of managing a Decision Table, refer section *Samples Related to Decision Tables* on page 264

6.26 Loop

Loop behavior can be enabled on nodes to create multiple node instances under specific conditions.

There are two types of loops:

- **Iterator Node:** The XPDL representation of Iterator Node is `LoopMultiInstance`. Enabling iteration on a node with an iteration count 'n' allows you to create 'n' instances of the same node during run-time. The loop execution in Iterator node takes place in parallel. Refer *Iterator Node* on page 196 for more information.

- **Sequential Loop Node:** The XPDL representation of Sequential Loop Node is `LoopStandard`. Enabling Sequential Loop on a node allows it to generate sequential instances. The loop execution in Sequential Loop takes place sequentially. Refer *Sequential Loop Node* on page 203 for more information.

6.26.1 Iterator Node

Normally, for each node created in a process definition, only a single node-instance is generated at run-time. Enabling iteration on a node with an iteration count 'n' allows you to create 'n' instances of the same node during run-time.

An iterator node has the following general features:

- Iteration is enabled during design-time.
- The number of iterations is specified using a UDA of Integer type. If a specified UDA for an iterator node does not exist, process validation will fail and no process instance can be started from the process definition until the UDA is added.
- You can enable only the following types of nodes for iteration:
  - User Task nodes
  - Call Activity nodes
  - Chained-Process nodes
- All iterated node instances are executed in parallel.
- To enable iteration, iteration count should be > 0.
- If the iteration count for a node is set to <=0:
  - Even though a single instance of a node would already have been created as soon as the process instance is created, that node instance will not be activated
  - Execution of the node instance will be skipped, and the next node will be activated.
- If a completed iterator node is reactivated, node instances created during the earlier execution of the iterator node are re-used by reactivating them. For example, consider during the first run of an iterator node 3 instances were created.
  - If during its second run iteration count is set to 10, then only 7 new instances will be created; 3 instances from the previous run will be re-used by reactivating them.
  - If during the second run iteration count is set to 2, no new instances will be created; of the 3 instances from the previous run, only 2 instances will be reactivated for re-use, and one instance will remain inactive.
- For ad hoc activation, deactivation, if some iterated instances of a node are running, and some are closed:
  - You cannot reactivate a closed instance.
  - If you deactivate a running instance, all running instances will be deactivated.
- Prologue and Epilogue Java Actions are executed only once irrespective of the number of iterations.
- You cannot change the source or target nodes of the outgoing arrows of an iterated node instance.
- If you delete an iterated node instance, all iterated instances for that node are deleted.
- If you delete the outgoing arrow instance of an iterated node instance, all outgoing arrow instances of all iterated instances for that node are deleted.
- If you add an outgoing arrow instance to an iterated node instance, outgoing arrow instances are added for all iterated instances of that node.

- Setting iterator count on a running node instance will not be effected during that run of the node. It will be effected only during the next run (if any) of that node instance.
- Each iterated node instance can be identified and accessed using its iterator index number. Index count ranges from 1 to 'n', where 'n' is the iteration count.
- From a running process instance containing an iterator node, the iterator count UDA specified for the iterator node cannot be deleted.

Iterated User Task Nodes

Iterated User Task nodes have the following features:

- For a User Task node, consider you set iteration count as 'n'. As soon as the process instance is created, the first instance of that node is created. After the process instance starts, when control reaches that node, the following occurs:
  - For an iterated User Task node, 'n-1' instances are created. (The first instance was created when the process instance is created.)
  - Work items are created for all users of an assigned group, for each of the 'n' node instances. For example, if there are 10 users for an assigned group, and iteration count is 5, 50 work items are generated.
- For each of the instances, if any user completes the work item, the work items for other users within that node instance are deleted, and that node instance is complete.
- When all iterated instances of a node will complete, the activity of that node is completed, and the next node is activated.
- The number of outgoing arrows from an activity iterator node is restricted to one.
- All instances of an iterator node share all the UDAs in the process. If one instance changes a UDA's value, the change is reflected for all instances immediately.
- You cannot use triggers on an iteration-enabled User Task node.
- Each iterated instance has the same properties (name, description, and so on).
- The iterator index number of an iterated instance can be retrieved using a Model API function, JavaScript function or a custom Java Action. For more information, refer *Retrieving the Iterator Index of a Node Instance using Model API* on page 199 and *Retrieving the Iterator Index of a User Task Node Instance using a Custom JavaAction Class* on page 199.
- You can assign different users to each iterated instance of a node. For more information, refer *Retrieving the Iterator Index of a User Task Node Instance using a Custom JavaAction Class* on page 199. The Role Action Set is executed for each iterated node instance.

Recalling Work items for Iterator Nodes

Iterator Nodes support the work item recall feature in the following scenarios.

- When an iterator node is a source node for recall, recall is possible only if at least one of the iterator node instances has been closed.
  - If all the iterator node instances are closed, and the next node instance is still active - On recall, the active target node instance is de-activated. The compensate Java Action defined for prologue Java Action of target node instance is executed, the compensate Java Action defined for epilogue Java Action of source node instance is executed. The recalled iterator node instance is activated, and a work item is created for the user that recalled the work item.

    - If all of the iterator node instances have not been closed - Recall is possible only for any of the closed iterator node instances. On recall, the recalled node instance is re-activated and a work item created for the user that recalled the work item.
- When an iterator node is a target node for recall, recall is possible only if all iterator node instances are running. On recall, all iterated node instances are de-activated. The compensate Java Action defined for prologue Java Action of target node instance will be executed, the compensate Java Action defined for epilogue Java Action of source node instance will be executed. The recalled node instance is re-activated, and a work item is created for the user that recalled the work item.

For more information about recalling work items, refer *Recalling Work Items* on page 83.

Iterated Call Activity and Chained-Process Nodes

Iterated Call Activity and Chained-Process nodes have the following features:

- For a Call Activity or Chained-Process node, consider you set iteration count as 'n'. As soon as the process instance is created, the first instance of that node is created. After the process instance starts, when control reaches that node, the following occurs:
    - For an **Iterated Call Activity Node**, 'n-1' node instances will be created. For each of the 'n' node instances, an instance of the linked subprocess will be created and started. Control will shift to the next node only after all iterated instances of the Call Activity node are complete.
    - For an **Iterated Chained-Process Node**, only the first node instance will be created. There will be no further node instances (n-1) as in the case of Iterated Call Activity Node.

      Only one Chained-Process Instance is generated in this scenario and its state is 'closed'.

      For the single node instance that was created when the process instance was created, 'n' instances of the linked child subprocess will be created and started. Control will shift to next node as soon the 'n' linked processes are started.
- If the process instance containing iterated Call Activity and Chained-Process nodes is aborted, all iterated subprocess instances are aborted.

Data Mapping for Call Activity and Chained-Process Iterator Nodes

- Each iterated instance of a linked process (Call Activity or Chained-Process) has its own UDAs. An instance cannot interact directly with UDAs of its sibling iterated instances. It can directly interact only with UDAs of its parent process.
- You can transmit different data between each iterated child process instance and its parent process instance (and vice-versa) by using an XML UDA and XPath with the predefined variable **`$index`**. The variable `$index` within the Xpath expression of a parent process XML UDA helps map different values from the parent XML UDA to XML UDAs of different iterated child instances. For details of how such data mapping can be implemented, refer *Data Mapping between Parent and Iterated Child Process Instances* on page 201.

Working with Iterator Nodes

Creating an Iterator Node

While designing a process definition, you can enable a node for iteration by using the `setIteratorCount(String udaName)` method of the `com.fujitsu.iflow.model.workflow.Node` interface. Note that the UDA should be of type integer.

You can also retrieve the name of an iterator UDA using the `getIteratorCount()` method of the `com.fujitsu.iflow.model.workflow.Node` interface.

For more information about these methods, refer the API Javadoc.

The following sample code sets the UDA `IteratorCount` as the iterator count UDA name for User Task node `Activty`, and also sets the iteration count to `5`.

```
//lock a Process definition before calling the API
plan.startEdit();
plan.addDataItemRef("IteratorCount", DataItemRef.TYPE_INTEGER, "5");
Node activityNode = plan.addNode("Activity", Node.TYPE_ACTIVITY);
activityNode.setIteratorCount("IteratorCount");
//when you save the plan, the Iterator count of
//the above node will also get saved
plan.createProcessDef();
//Now you can check the Iterator count set above
String iteratorCountUda = activityNode.getIteratorCount();
```

An iteration enabled node can be reverted back to a normal node by setting an empty ("") string as the iterator count UDA name.

Note: It is recommended to set iterator count of a node by taking into consideration the number of assignees for each Iterated node instance, so that it should not result in creation of more than 2000 work items per Process instance for the iterated node. Otherwise, depending upon the server environment setup and system load, the iterator node activation transaction may take too long and time out, causing the process to go into error state.

Retrieving the Iterator Index of a Node Instance using Model API

You can retrieve the iterator index of a node instance using the `getIteratorIndex()` method of the `com.fujitsu.iflow.model.workflow.NodeInstance` interface. For more information about this method, refer the API Javadoc.

Sample code:

```
 //retrieve the Node instance using Model API, then use code below
//for retrieving iterator index of node instance
//Let 'ni' represent retrieved Node Instance
int iteratorIndex ;
iteratorIndex = ni.getIteratorIndex();

//this iteratorIndex can be used for manipulation of an XML UDA as follows

//Create the XPath using above retrieved iteratorIndex
String xpath="//OrderIds/orderId[position()=" + iteratorIndex + "]/text()";


//Retrieve the XML UDA from the process instance
:
:
//Assume 'dataItem' is the retrieved XML UDA

//Set the new value of the XML UDA using the XPath
dataItem.setElementValue(XPath, "Updated");

//Retrieve the new value
String udaValue = dataItem.getElementValue(xpath);
```

Retrieving the Iterator Index of a User Task Node Instance using a Custom JavaAction Class

You can retrieve the iterator index of a node instance by calling the `getActivityIteratorIndex()` method of the `com.fujitsu.iflow.server.intf.ServerEnactmentContext` interface, and use

this index for reading values from an XML UDA for a particular node instance in a custom JavaAction class.

For example, you would need to define a custom JavaAction class to retrieve the iterator index of node instances if you want to set different assignees to each instance.

To set Node Instance Assignees, perform the following steps:

1. Create a custom JavaAction class that uses `getActivityIteratorIndex()` to define a custom method that can be used to set assignees.
2. The custom JavaAction class defined above can be used in a Role Action Set to assign different assignees to different iterations of a node instance.

1. **Creating a Custom JavaAction Class**

   When creating the class for a custom Java Action Type:

   a) Import the `ServerEnactmentContext` interface.

   b) Make sure that one parameter of the method that you want to call from a JavaAction is of type `ServerEnactmentContext`.

   c) When defining the Java Action, specify the values to pass to your method. Use the `sec` identifier as a value for the `ServerEnactmentContext` parameter. At run time, Interstage BPM will pass an instance of ServerEnactmentContext to the custom method.

   For example, consider the XML UDA `UdaAssignees`

```
<Assignees>
    <Assignee>ibpm_user1</Assignee>
    <Assignee>ibpm_user2</Assignee>
    <Assignee>ibpm_user3</Assignee>
</Assignees>
```

   The sample code below is for the custom JavaAction class `MyClass` that uses the `getActivityIteratorIndex()` method to define a custom method named `setAssignee` that can be used to set assignees.

```
import com.fujitsu.iflow.server.intf.ServerEnactmentContext;
public class MyClass {
    public void setAssignee(ServerEnactmentContext sec)
            throws Exception {
        String xpath = "//Assignees/Assignee[position()=" +
sec.getActivityIteratorIndex() + "]/text()";
        String assignee =
sec.getProcessXMLAttributeElementValue("UdaAssignees", xpath);
//here "UdaAssignees" is the name of the XML UDA containing the
assignees

        String[] assignees = { assignee };
        sec.setActivityAssignees(assignees);
    }
}
```

2. **Using Role Action Set to assign different assignees to different iterator instances**

   a) Add a User Task Node. The name of the first node in the process definition uses the '`protected`', '`final`', '`static`' modifiers.

```
String NODE_FILL_OUT_PR = "Fill out Purchase Request";
protected final static String NODE_FILL_OUT_PR = "Fill out Purchase
Request";
```

```
Node fillOutNode = plan.addNode("NODE_FILL_OUT_PR",Node.TYPE_ACTIVITY);
fillOutNode.setRole("SampleGroup");
fillOutNode.setUpperLeftPoint(new Point(450, 40));
```

b) Use `getJavaActionSet() WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet asRJavaActionSet = WFObjectFactory.getJavaActionSet();
```

c) Generate the required number of Java Actions for JavaActionSet.

```
JavaAction[] asRJavaActions = asRJavaActionSet.createJavaActions(1);
```

d) Define the JavaActions that set the assignee of a node instance

```
asRJavaActions[0].setActionDescription("Sets Activity assignee");
asRJavaActions[0].setActionName("Set assignee");
asRJavaActions[0].setMethodName("setAssignee(ServerEnactmentContext");
asRJavaActions[0].setClassName("MyClass");
```

e) Assign JavaActionSet contents to the Role Action Set of the User Task Node.

```
asRJavaActionSet.setJavaActions(asRJavaActions);
fillOutNode.setJavaActionSet(asRJavaActionSet,JavaActionSet.NODE_ROLE);
```

Note: You can also use the `getActivityIteratotIndex()` in a JavaScript as follows:

```
var iteratorIndex = sec.getActivityIteratorIndex();
```

You can further use `iteratorIndex` as required.

Data Mapping between Parent and Iterated Child Process Instances

You can transmit different data between each iterated child process instance and its parent process instance (and vice-versa) by using an XML UDA and XPath with the predefined variable **`$index`**.

Consider the following XML UDAs:

- Parent process XML UDA, named `ParentXMLUDA`

```
<Items>
    <Item>A1</Item>
    <Item>A2</Item>
</Items>
```

- Child instance #1 with XML UDA `ChildXMLUDA_P`

```
<ChildItems>
    <ChildItem>P</ChildItem>
</ChildItems>
```

- Child instance #2 with XML UDA `ChildXMLUDA_P`

```
<ChildItems>
    <ChildItem>P</ChildItem>
</ChildItems>
```

Note that since child instances come from the same child process, all child instances will have UDAs with same names and **initial** values.

You can implement data mapping between parent and child XML UDAs using the `addDataMappingElement()` method of the `com.fujitsu.iflow.model.workflow.Node` interface.

Syntax:

```
addDataMappingElement(java.lang.String parentDataItemRefName,java.lang.String
xPathForParentDataItem,java.lang.String childDataItemRefName,java.lang.String
xPathForChildDataItem,int direction)
```

Why `$index` is needed

Using the `addDataMappingElement()` method without `$index`, for example `addDataMappingElement("ParentXMLUDA", "//Items/Item[2]/text()", "ChildXMLUDA_P", "//ChildItems/ChildItem/text()", DataItemMappingElement.INOUT)` will cause the following to happen:

1. At the start of the child process, UDAs of **both** child instances will receive the **same** value (as specified by `"//Items/Item[2]/text()"`) that is, the **second** item of `ParentXMLUDA`, which is **`A2`**. The child instance UDAs will then be as follows:
   - UDA of child instance **#1**:

     ```
     <ChildItems>
         <ChildItem>A2</ChildItem>
     </ChildItems>
     ```

   - UDA of child instance **#2**:

     ```
     <ChildItems>
         <ChildItem>A2</ChildItem>
     </ChildItems>
     ```

2. At the end of the child instances, both instances will modify the **same** item in the parent instance (as specified by `"//Items/Item[2]/text()"`) that is, the **second** item of `ParentXMLUDA`, thus overwriting each others transferred data. For example, child instance #1 will change the second item of `ParentXMLUDA` to **P1**, and then child instance #2 will overwrite the second item of `ParentXMLUDA` to **P2**. The parent XML UDA would be as follows:

```
<Items>
    <Item>A1</Item>
    <Item>P2</Item>
</Items>
```

Essentially, this means you cannot map different values from a parent process to the different child instances. Such a problem will occur even when you use non-XML UDA data mapping.

To be able to map different values from a parent process to the different iterated child instances, you need to use XML UDAs and the `$index` pre-defined variable.

Using `$index`

Using the `addDataMappingElement()` method with `$index`, for example `addDataMappingElement("ParentXMLUDA", "//Items/Item[$index]/text()",`

"ChildXMLUDA_P", "//ChildItems/ChildItem/text()", DataItemMappingElement.INOUT)
will cause the following to happen:

1. UDAs of **both** child instances will receive **different** values (as specified by "//Items/Item**[$index]**/text()"). The $index variable in the Xpath expression of the parent UDA will automatically map the **first** item of the parent XML UDA to the XML UDA of the **first** child instance, the **second** item of the parent XML UDA to the XML UDA of the **second** child instance, and so on. The child instance UDAs will then be as follows:
   - UDA of child instance **#1**:

```
<ChildItems>
    <ChildItem>A1</ChildItem>
</ChildItems>
```

   - UDA of child instance **#2**:

```
<ChildItems>
    <ChildItem>A2</ChildItem>
</ChildItems>
```

2. At the end of the child instances, each child instance will have modified **different** items of the parent XML UDA. The $index variable in the Xpath expression of the parent UDA will automatically map the **first** item of the parent XML UDA to the XML UDA of the **first** child instance, the **second** item of the parent XML UDA to the XML UDA of the **second** child instance, and so on. The parent XML UDA will then be as follows:

```
<Items>
    <Item>P1</Item>
    <Item>P2</Item>
</Items>
```

Thus, using $index you can map different parent UDA values to UDAs of different iterated child instances.

Note:
- $index forces serial order mapping (first item to first child instance, second item to second child instance); you need to ensure that items in your parent XML UDA are ordered corresponding to the child instances to which they are to be mapped.
- $index can be used only in Xpath expressions of the **parent** process XML UDA.
- If $index is used for non-iterator node XML UDA XPaths, the variable will be replaced by '1' during processing.

For other information related to data mapping, refer *Designing a Parent and Remote Subprocess Definition* on page 180.

6.26.2 Sequential Loop Node

Normally, for each node created in a process definition, only a single instance is generated at run-time. A scenario might require a node to generate multiple instances sequentially 'while' a certain condition is being satisfied. Enabling Sequential Loop on a node allows it to generate sequential instances.

For example, in **Order Shipment** process, Sequential Loop Node enables an Agent to check the stock of ordered items. The Agent gets items (from **Order**) and automatically adds them to the

shipment *while* there are items available in **Order**. This Agent repeats the same process sequentially until there are no more items in the **Order**.

Sequential Loop Node Features

Sequential Loop Node has the following features:

- You can configure a node as a Sequential Loop Node during design time.
- You can configure Sequential Loop only on the following types of nodes:
  - User Task nodes
  - Call Activity nodes
  - Embedded Sub-Process nodes
- While designing a process definition, you can enable Sequential Loop on a node by specifying either Loop Condition or Max Loop Count or both.
- If a wrong or an invalid Loop Condition expression is specified then the error cannot be detected during design time. It will be a run-time error.

Note: If an operation on a process instance fails then either Model API returns an error or the same process instance goes into error state. This type of error is called a run-time error.

- The Loop Condition expression must evaluate into a Boolean type value, that is a `true` or a `false`. If it evaluates into a non-Boolean value, it will result in a run-time error.
- Max Loop Count can be specified as either an integer constant or UDA.
- Each looped instance will have a unique Loop Index. The Loop Index specifies the activation sequence of the loop node instances. You can choose to enable Sequential Loop in either auto-increment mode or auto-decrement mode.
  - If you set it in auto-increment mode, Loop Index will start from 1 to reach the Max Loop Count.
  - If you set it in auto-decrement mode, Loop Index will start from Max Loop Count to reach 1.

Note: By default, the behavior will be auto-increment.

- In auto-increment mode, when the Loop Index is less than or equal to the Max Loop Count and Loop Condition becomes true, the loop will continue and new looped node instance will be activated.
- In auto-decrement mode, when the Loop Index is greater than or equal to 1 and Loop Condition becomes true, the loop will continue and new looped node instance will be activated.

- Java Action and Agent on Sequential Loop Node are executed for each looped node instance, depending upon whether looped node types are supported for them.
- Loop behavior can be controlled by using error handling options (**Continue** or **Break**), either by using node-level setting or by handling loop behavior programmatically in Custom Java Action Type and Agent. These are available for Prologue/Role/Epilogue Action Sets and Agents.
- If a completed Sequential Loop Node is reactivated, looped node instances created during the earlier execution of the Sequential Loop Node are re-used by reactivating them. For example, consider during the first run of a Sequential Loop Node, 3 instances were created.
  - If during its second run, loop executes 10 times then only 7 new looped instances will be created; 3 looped instances from the previous run will be re-used by reactivating them.

    - If during the second run, if loop executes two times, no new looped instances will be created; of the 3 looped instances from the previous run, only 2 looped instances will be reactivated for re-use, and one looped instance will remain inactive.
- Ad-hoc activation of loop instance:
    - **For User Task Node:** In case of ad-hoc activation, if already running looped node instance exists then the running looped node instance will be aborted and the current (looped instance on which ad-hoc activation is called) looped node instance will be activated.
    - **For Call Activity Node:** In case of ad-hoc activation, if already running looped Call Activity node instance exists then the running looped Call Activity node instance will be aborted and the current (looped instance on which ad-hoc activation is called) looped Call Activity node instance will be activated. The subprocess linked to the aborted looped Call Activity node instance will also be aborted.
    - **For Embedded Sub-Process Node:** In case of ad-hoc activation, if already running looped Embedded Sub-Process node instance exists (the Embedded Sub-Process node instance is in 'waiting for subprocess' state) then the running looped Embedded Sub-Process node instance and its child node instances will be aborted and the current (looped instance on which ad-hoc activation is called) looped Embedded Sub-Process node instance will be activated.
- If the parent process instance containing looped subprocesses is aborted, then active subprocess looped instance will also be aborted.
- You can use the Loop Index for UDA manipulation and data mapping just as you can use the Iterator index. Refer section *Data Mapping for Subprocess and Chained-process Iterator Nodes* in section *Iterated Call Activity and Chained-Process Nodes* on page 198 for more information.

Sequential Loop Node Restrictions

Following are the restrictions for using Sequential Loop Node:

- Only 1 outgoing arrow is allowed from the Sequential Loop Node. If a Sequential Loop Node has 2 or more arrows, then process validation will fail.
- You cannot use Iteration and Sequential Looping features together on the same node.
- You cannot use Future Work Item feature on Sequential Loop Node.
- You cannot use Timer/Due Date on Sequential Loop Node.
- You cannot use Transaction Control (by using `setNodeInTxn()` method of `com.fujitsu.iflow.model.workflow.Node` interface) on Sequential Loop Node.
- You cannot archive the process definitions/process instances which have Sequential Loop Nodes.
- You cannot migrate the process instance which has Sequential Loop Node, to another process definition. Similarly, you cannot migrate a process instance to a process definition which has Sequential Loop Node.
- You cannot use `StructuralEdit` at run-time for a Sequential Loop Node. You cannot perform the following actions at run-time:
    - Adding, removing, and changing the Loop Condition
    - Adding, removing the Max Loop Count, and changing constant value or UDA which is used as Max Loop Count

Note: If a UDA is specified as Max Loop Count, the value can be changed and the value will be reflected as Max Loop Count.

    - Changing the increment/decrement counter option

  - Changing the option for error handling on node level
  - Deleting looped instance
  - Deleting outgoing arrow of looped instance
  - Changing source/target node instance of an outgoing arrow of looped instance
- If a Sequential Loop Node has a loop-back arrow (cyclic structure) and Sequential Loop is set in auto-decrement mode then, you can change Max Loop Count (in case the Max Loop count is set by UDA value) only before the first loop execution starts. Even if you change it after the first loop execution or before the second execution starts, it is not reflected and the previous Max Loop Count is used for the second execution.
- If a Sequential Loop Node is completed and the next node is activated, you cannot recall a completed workitem of the sequential loop node.
- If Sequential Looping is enabled on an Embedded Sub-Process Node, then you cannot use Iteration or Sequential Looping features on its child nodes.

Configuring Node as Sequential Loop Node

To configure Sequential Loop on a node, you can either specify Loop Condition or Max Loop Count or both on a User Task node, Call Activity node, or Embedded Sub-Process node. The following table shows the inter-relationship between the Loop Condition values and Max Loop Count along with the result when the Sequential Loop is in auto-increment mode:

Loop Condition	Max Loop Count	Result
True	index < = Max Count	Next looped node instance will be activated.
True	index > Max Count	Next looped node instance will not be activated.
False	index < = Max Count	Next looped node instance will not be activated.
False	index > Max Count	Next looped node instance will not be activated.
Not specified	index < = Max Count	Next looped node instance will be activated.
Not specified	index > Max Count	Next looped node instance will not be activated.
True	Not specified	Next looped node instance will be activated.
False	Not specified	Next looped node instance will not be activated.
Not specified	Not specified	Node will behave as a normal node.

The methods described below belong to the `com.fujitsu.iflow.model.workflow.Node` interface.

- Loop condition can be specified using the `setLoopCondition(String jsExpression)` method.
- Max Loop Count can be specified using `setMaxLoopCount(String maxCountJs)` method.
- Loop Condition can be retrieved using `getLoopCondition()` and for Max Loop Count, use `getMaxLoopCount()` method.
- After configuring Sequential Loop Node, you can set the loop count behavior to either auto-decrement or auto-increment by using `setLoopCountBehavior(int option)` method.

You can use the following sample code as basis to set the Loop Condition, Max Loop Count, and loop count behavior for User Task `loopActivity`.

```
// Get the process definition
Plan plan = WFObjectFactory.getPlan();
// Start edit
plan.startEdit();
```

```
// Add a User Task node
Node loopActivity = plan.addNode("LoopActivity", Node.TYPE_ACTIVITY);

// Define a loop condition, here balance and stop are 2 UDA names
String jsExpression = "(uda.get(\"balance\")  >  0  &&  uda.get(\"stop\")"

       + "  ==  false)";
// Add the defined loop condition to a node
loopActivity.setLoopCondition(jsExpression);

// For specifying the Max Loop Count, either add an integer type UDA.
plan.addDataItemRef("LoopMaxCount", DataItemRef.TYPE_INTEGER, "10");
// then specify this UDA name as Max Loop Count in form of Java Script
// expression
loopActivity.setMaxLoopCount("uda.get(\"LoopMaxCount\")");
// Or directly specify the integer constant as Max Loop Count
// loopActivity.setMaxLoopCount("10");

// Now set the loop behavior to auto decrement.
loopActivity.setLoopCountBehavior(Node.LOOP_COUNT_AUTO_DECREMENT);
// When you save the plan after completing the process definition
// design, all above changes to node will also get saved
plan.createProcessDef();

// Retrieving loop condition, Max loop count and loop behavior
jsExpression = loopActivity.getLoopCondition();
String maxLoopCountUDAName = loopActivity.getMaxLoopCount();
int loopCountBehavior = loopActivity.getLoopCountBehavior();
```

Enabling Error Handling on Sequential Loop Nodes

You can specify error handling options for a Sequential Loop Node.

Node-level error handling options will work for Prologue/Role/Epilogue Action Sets and Agent only if these throw error. Loop will break or continue according to the node-level error handling setting as soon as the error occurs.

The methods described below belong to the `com.fujitsu.iflow.model.workflow.Node` interface.

- Error handling option can be specified using the `setLoopErrorHandling(int option)` method.
- Use the constant `LOOP_ERROR_HANDLING_BREAK` for breaking the loop.
- Use the constant `LOOP_ERROR_HANDLING_CONTINUE` for ignoring the error and continue the loop.
- Use the constant `LOOP_ERROR_HANDLING_NONE` for using the default behavior.

You can use the following sample code as basis to set `LOOP_ERROR_HANDLING_BREAK` as the error handling option to a loop node `loopActivity`.

```
//lock a Process definition before calling the API
plan.startEdit();
// for break option
loopActivity.setLoopErrorHandling(Node.LOOP_ERROR_HANDLING_BREAK);
// for continue option
// loopActivity.setLoopErrorHandling(Node.LOOP_ERROR_HANDLING_CONTINUE);

// for disabling the error handling and use default behavior
// loopActivity.setLoopErrorHandling(Node.LOOP_ERROR_HANDLING_NONE);
// then commit the plan
plan.commitEdit();
```

Programmatically Handling Loop Behavior from Custom Java Action Types and Agent Class

You can use APIs in `com.fujitsu.iflow.server.intf.ServerEnactmentContext` interface to programmatically control the behavior of the loop to either continue or break it in your custom Java Action Types in Prologue/Role/Epilogue Action Sets or Custom Agents. The loop will break or continue as per the `ServerEnactmentContext` method called only after the complete execution of the current Java Action. Execution of the remaining Java Actions in this Java Action Set will be skipped.

Given below is the sample code to be used as basis for using the break and continue options from Custom Java Actions and Agent Classes:

```
package mypakage;
import com.fujitsu.iflow.server.intf.ServerEnactmentContext;
public class MyClass {
 public void readOrderData(ServerEnactmentContext sec,
 String orderFilePath) {
  try {
      // code for reading the order file.
      FileInputStream fInputStream = new FileInputStream(new File(
      orderFilePath));
      ........
      ........
      ........
      // if file is read, but some value is not there for this looped
      // instance then ignore the current loop and continue with
      // next looped instance.
      DataInputStream dataInStream = new DataInputStream(fInputStream);
      BufferedReader buffReader = new BufferedReader(
      new InputStreamReader(dataInStream));
      String messageFromFile = buffReader.readLine();
      if (messageFromFile == null || messageFromFile.trim().length() ==
0) {
      // The setContinueLoop() method call below will mark the loop
      // for ignoring the current looped instance and continue with
      // the next looped instance after the execution
      // of this Java Action.
      sec.setContinueLoop();
      return;
      }

   } catch (FileNotFoundException ex) {
     // code for exception handling and break the loop
     // The setBreakLoop() method call below will mark the loop
     // for breaking the loop after the execution
     // of this Java Action.
     sec.setBreakLoop();
   }
  }
}
```

Consider the Java Action Set below as an example for loop behavior:

JAS:

- JA1
  - JA1Err
  - JA1Cmp

- JA2
  - JA2Err
  - JA2Cmp
- JA3
  - JA3Err
  - JA3Cmp

where:

JAS: Java Action Set

JA: Java Action

JA*Err: Error Java Action

JA*Cmp: Compensate Java Action

Consider `setBreakLoop()` or `setContinueLoop()` method called from JA2, then the different scenarios and respective loop behaviors are as given below:

Scenario 1: If JA2 does not throw error then the behavior will be as given below:

- JA1 is executed successfully, JA1Err and JA1Cmp will not be executed.
- JA2 is executed successfully (as it does not throw error), JA2Err and JA2Cmp will not be executed.
- JA3 execution will be skipped.
- Then the loop will break or continue according to the called method after execution of JA2.

Scenario 2: If JA2 throws error and JA2Err handles the error and it does not throw the caught error then the behavior will be as given below:

- JA1 is executed successfully, JA1Err and JA1Cmp will not be executed.
- JA2 is executed but throws error, JA2Err is executed and handles the error, JA2Cmp will not be executed.
- JA3 execution will be skipped.
- Then the loop will break or continue according to the called method after execution of JA2.

Scenario 3: If JA2 throws error and JA2Err does not handle the error and throws the caught error then the behavior will be as given below:

- JA1 is executed successfully, JA1Err and JA1Cmp will not be executed.
- JA2 is executed but throws error, JA2Err is executed and throws error or there is no Error Java Action, JA2Cmp will not be executed.
- JA3 execution will be skipped.
- Then the loop will break or continue according to node-level loop error handling setting after execution of JA2.

Error Handling Options and their Behaviors

Following table details the states of the Loop, Process Instance, and Looped Node Instance when you select any of the error handling options (None, Continue, Break) on a looped User Task Node:

Error in Operation	Error Handling Option	Loop Behavior	Loop User Task Node Instance State	Process Instance State
• Prologue and Role Java Action Error • Agent Error	None	Stops at that particular looped node instance	Error	Error
	Continue	Loop will continue as per the Loop Condition evaluation and Max Loop Count	Closed	Running
	Break	Loop will break and next node will be activated	Aborted	Running
Epilogue Java Action Error	None	Make choice operation will fail	Running	Running
	Continue	Loop will continue as per the Loop Condition evaluation and Max Loop Count	Closed	Running
	Break	Loop will break and next node will be activated	Aborted	Running

Following table details the states of the Loop, Process Instance, and Looped Node Instance when you select any of the error handling options (None, Continue, Break) on a looped Call Activity Node:

Error in Operation	Error Handling Option	Loop Behavior	Loop Call Activity Node Instance State	Parent Process Instance State	Child Process Instance State
Prologue Java Action Error	None	Stops at that particular looped Call Activity node instance	Error	Error	Not created
	Continue	Loop will continue as per the Loop Condition evaluation and Max Loop Count	Closed	Running	Not created
	Break	Loop will break and next node will be activated	Aborted	Running	Not created

Error in Operation	Error Handling Option	Loop Behavior	Loop Call Activity Node Instance State	Parent Process Instance State	Child Process Instance State
Epilogue Java Action Error	None	Stops at that particular looped Call Activity node instance	Suspended	Error	Suspended
	Continue	Loop will continue as per the Loop Condition evaluation and Max Loop Count	Closed	Running	Closed
	Break	Loop will break and next node will be activated	Aborted	Running	Closed

Following table details the states of the Loop, Process Instance, and Looped Node Instance when you select any of the error handling options (None, Continue, Break) on a looped Embedded Sub-Process Node:

Error in Operation	Error Handling Option	Loop Behavior	Loop Embedded Sub-Process Node Instance State	Process Instance State	Child Node Instance State
Prologue and Role Java Action Error	None	Stops at that particular looped Embedded Sub-Process Node instance	Error	Error	Not activated
	Continue	Loop will continue as per the Loop Condition evaluation and Max Loop Count	Closed	Running	Not activated
	Break	Loop will break and next node will be activated	Aborted	Running	Not activated

Error in Operation	Error Handling Option	Loop Behavior	Loop Embedded Sub-Process Node Instance State	Process Instance State	Child Node Instance State
Epilogue Java Action Error	None	Stops at that particular looped Embedded Sub-Process Node instance	Suspended	Error	If Embedded Sub-Process Node is completed by completing its child node instances: • End child node instance goes into Error state. • States of other child node instances remain unaffected.
	Continue	Loop will continue as per the Loop Condition evaluation and Max Loop Count	Closed	Running	If Embedded Sub-Process Node is completed by completing its child node instances: • States of child node instances remain unaffected. If Embedded Sub-Process Node is completed by doing make choice operation on its workitem: • Active child node instances will be closed. • States of other child node instances remain unaffected.
	Break	Loop will break and next node will be activated	Aborted	Running	If Embedded Sub-Process Node is completed by completing its child node instances: • States of child node instances remain unaffected. If Embedded Sub-Process Node is completed by doing make choice operation on its workitem: • Active child node instances will be closed. • States of other child node instances remain unaffected.

Forcibly Breaking the Loop

Follow the steps given below to break the loop forcibly after a particular looped instance is completed:

1. Define an integer UDA named `flag` and assign an initial value 1.
2. Add one more condition to the actual Loop Condition as: `LoopCondition && uda.get("flag") != 0`.
3. Then change the UDA (`flag`) value to `0` if you want to break the loop.

Note: When you make choice in the current looped instance or when the subprocess completes, the loop will break and the next node will be activated.

6.27 Displaying BPMN View of Process Definitions, Process Instances in External Web Applications

You can display the BPMN view of process definitions, process instances in external web applications, outside Interstage BPM Console. To do this, you use the HTTP POST method (with required parameters) and call the process definition/process instance URL from the external application.

The view can be called either as an independent link or within a frame in the external web application.

Note: In Interstage BPM, if SSO authentication is enabled, this functionality is not supported. It may cause an error.

Note:
- The BPMN view displayed in the external web application will be read-only.
- You cannot display archived process definitions or process instances in external web applications.
- The approach on how to pass parameters to this URL via POST method can be decided by the external web application owner. For example, it can be as part of an HTML form having all parameters as listed in the sample implementation below.

HTTP POST Details

To display the BPMN view of the process definition/process instance, you need the following information to construct the POST request:

- The URL of the process definition/process instance
- The following POST parameters:
  - `pdID` / `piID` – the ID of the process definition/process instance to be displayed
  - `pdDetails` (only for process definitions) - the value for this is always set to `true`
  - `username` – the Console username for which the process definition/process instance is to be displayed
  - `password` – the password for the username
  - `fromExt` – the value for this is always `true`

Note: All parameter names and values above are case sensitive.

The URL for process definitions is:

`http://<hostname>:<port>/console/<tenant_name>/<application_id>/getProcessDefBPMN.page`

The URL for process instances is:

`http://<hostname>:<port>/console/<tenant_name>/<application_id>/getProcessInstanceBPMN.page`

Here,

- `hostname` is the machine name/IP on which Interstage BPM is setup
- `port` is the port for which Interstage BPM is configured
- `tenant_name` is name of the tenant
- `application_id` is name of the application that has the process definition/process instance

Note: If Interstage BPM has been configured for secure access, use '`https`' instead of '`http`' in all URLs above.

Sample Implementation

A sample implementation of using the POST method is shown below. This sample displays the process instance BPMN view inside a frame.

```
<FORM METHOD=POST
ACTION="http://<hostname>:<port>/console/<tenant_name>/<application_id>/
getProcessInstanceBPMN.page"
  name="test" id="test" target="frm">
    <INPUT TYPE="hidden" NAME="fromExt" value="true">
    <INPUT TYPE="hidden" NAME="password" value="<%=password%>">
    <INPUT TYPE="hidden" NAME="username" value="<%=username%>">
    <INPUT TYPE="hidden" NAME="piID" value="<%=piID%>">
</FORM>

<iframe name="frm" id="frm" frameborder="0" marginwidth="0" width="100%"

  height="1000px" marginheight="0" leftmargin="0" topmargin="0">
</iframe>
```

Restrictions

- If both the external web application as well as Interstage BPM Console are using the same domain (that is, having the same IP address while accessing the URL), and login parameters are missing or not passed correctly to the BPMN view URL, then:
  - The BPMN view will not load correctly when called inside frames.
  - No login screen will appear; the frame's parent page will reload.
- If both the external web application and Interstage BPM Console are using different domains (for example, the external application using local domain and Interstage BPM Console using a remote domain) and Interstage BPM Console is called inside the frame, then Interstage BPM Console does not operate properly. To avoid this:
  - Do not use frames in the scenario described above (different domains). If you need to use frames, either of the following settings is required for Internet Explorer:
    - Add Interstage BPM Console address to local Intranet or to **Trusted sites** in **Tools > Internet Options > Security**.
    - Add Interstage BPM Console domain to **Per Site Privacy Actions** as **Allow** in **Tools > Internet Options > Privacy > Sites**.

6.28 Displaying Forms in External Web Applications

You can display forms (*.jsp) in external web applications, outside Interstage BPM Console. To do this, you use the HTTP POST method (with required parameters) and call the QuickForm URL from the external application.

The form will display with its assigned values and users can save the updated value using the 'save' button.

The view can be called either as an independent link or within a frame in the external web application.

Note: In Interstage BPM, if SSO authentication is enabled, this functionality is not supported. It may cause an error.

Note: The approach on how to pass parameters to this URL via POST method can be decided by the external web application owner. For example, it can be as part of an HTML form having all parameters as listed in the sample implementation.

HTTP POST Details

To display the form, you need the following information to construct the POST request:

- The URL of the form
- The following POST parameters:
  - `workItemID` – the ID of task for which the form has to be displayed (the form should have been added as part of the task)
  - `username` – the Console username for which form is to be displayed
  - `password` – the password for the username
  - `fromExt` – the value for this is always `true`
  - `formName` – the name of the form that has to be displayed. This is optional.

Note: All parameter names and values above are case sensitive.

The URL for forms is:

`http://<hostname>:<port>/console/<tenant_name>/<application_id>/<relativePath>/<formName>.jsp`

Here,

- `hostname` is the machine name/IP on which Interstage BPM is setup
- `port` is the port for which Interstage BPM is configured
- `tenant_name` is name of the tenant
- `application_id` is name of the application that has the process instance
- `relativePath` represents the relative path of the form inside the application DMS Root's `web` folder. For example, if the form is inside `<DMSRoot>/web/AllForms/MyForm/` folder, you enter `AllForms/MyForm` in `<relativePath>`.

Note:
- If Interstage BPM has been configured for secure access, use '`https`' instead of '`http`' in all URLs above.
- If the URL has multi-byte characters (such as Japanese character set), then the character set (UTF-8) for the form and the JSP needs to be set, along with having the browser encoding set to UTF-8.

Sample Implementation

A sample implementation of using the POST method is shown below. This sample displays the form inside a frame.

```
<FORM METHOD=POST
ACTION="http://<hostname>:<port>/console/<tenant_name>/<application_id>/<relativePath>/
 <%=formName%>.jsp"
 name="QuickForm" id="QuickForm" target="frm">
    <INPUT TYPE="hidden" NAME="fromExt" value="true">
    <INPUT TYPE="hidden" NAME="password" value="<%=password%>">
    <INPUT TYPE="hidden" NAME="username" value="<%=username%>">
    <INPUT TYPE="hidden" NAME="workItemID" value="<%=workItemID%>">
    <INPUT TYPE="hidden" NAME="formName" value="<%=formName%>">
</FORM>

<iframe name="frm" id="frm" frameborder="0" marginwidth="0" width="100%"
  height="1000px" marginheight="0" leftmargin="0" topmargin="0">
</iframe>
```

Restrictions

- If both the external web application as well as Interstage BPM Console are using the same domain (that is, having the same IP address while accessing the URL), and login parameters are missing or not passed correctly to the Form URL, then:
  - Forms will not load correctly when called inside frames.
  - No login screen will appear; the frame's parent page will reload.
- If both the external web application and Interstage BPM Console are using different domains (for example, the external application using local domain and Interstage BPM Console using a remote domain) and Interstage BPM Console is called inside the frame, then Interstage BPM Console does not operate properly. To avoid this:
  - Do not use frames in the scenario described above (different domains). If you need to use frames, either of the following settings is required for Internet Explorer:
    - Add Interstage BPM Console address to local Intranet or to **Trusted sites** in **Tools > Internet Options > Security**.
    - Add Interstage BPM Console domain to **Per Site Privacy Actions** as **Allow** in **Tools > Internet Options > Privacy > Sites**.
- Even if an invalid `workItemID` is specified in the POST request, the form is still displayed in the external application, though without any UDA values in the fields.

6.29 Displaying Start Process Page in External Web Applications

You can display Start Process page in external web applications, outside Interstage BPM Console. To do this, you use the HTTP POST method (with required parameters) and call the REST URL from the external application.

Note: In Interstage BPM, if SSO authentication is enabled, this functionality is not supported. It may cause an error.

Note: If you want another user to access just a single process definition, you can create a URL using the process definition id that is displayed in the **Process Definition List**. You can use `http://<hostname>:<port>/console/<tenant_name>/<application_id>/d<planID>/startPD.page` to create a URL and send it to another user. This URL is called REST URL.

While using the REST URL to display the Start Process page in external web applications, the following items would be displayed:

- QuickForms
- Attachments
- General details (name, description, and comments for the process to be started)

You can also perform the following actions that the REST URL supports:

- Actions related to Forms
- Adding/removing attachments
- Starting the process definition

Note: The approach on how to pass parameters to this URL via POST method can be decided by the external web application owner. For example, it can be as part of an HTML form having all parameters as listed in the sample implementation.

HTTP POST Details

To display the Start Process page, you need the following information to construct the POST request:

- The REST URL of the Start Process page
- The following POST parameters:
  - `username` – the Console username for which the process has to be displayed
  - `password` – the password for the username

Note: All parameter names and values above are case sensitive.

The REST URL for Start Process page is:

`http://<hostname>:<port>/console/<tenant_name>/<application_id>/d<planID>/startPD.page`

Here,

- `hostname` is the machine name/IP on which Interstage BPM is setup
- `port` is the port for which Interstage BPM is configured
- `tenant_name` is name of the tenant
- `application_id` is name of the application that has the process instance
- `planID` is the id of the process definition that has to be started

Note: If Interstage BPM has been configured for secure access, use '`https`' instead of '`http`' in all URLs above.

Sample Implementation

A sample implementation of using the POST method is shown below. This sample displays the Start Process page inside a frame.

```
<FORM METHOD=POST
ACTION="http://<hostname>:<port>/console/<tenant_name>/<application_id>/d<%=planID%>/
 startPD.page"
 name="iimForm" id="iimForm" target="frm">
    <INPUT TYPE="hidden" NAME="password" value="<%=password%>">
    <INPUT TYPE="hidden" NAME="username" value="<%=username%>">
</FORM>

<iframe name="frm" id="frm" frameborder="0" marginwidth="0" width="100%"
  height="1000px" marginheight="0" leftmargin="0" topmargin="0">
</iframe>
```

Restrictions

- If both the external web application as well as Interstage BPM Console are using the same domain (that is, having the same IP address while accessing the URL), and login parameters are missing or not passed correctly to the REST URL, then:
  - The Start Process page will not load correctly when called inside frames.
  - No login screen will appear; the frame's parent page will reload.
- If both the external web application and Interstage BPM Console are using different domains (for example, the external application using local domain and Interstage BPM Console using a remote domain) and Interstage BPM Console is called inside the frame, then Interstage BPM Console does not operate properly. To avoid this:
  - Do not use frames in the scenario described above (different domains). If you need to use frames, either of the following settings is required for Internet Explorer:
    - Add Interstage BPM Console address to local Intranet or to **Trusted sites** in **Tools > Internet Options > Security**.
    - Add Interstage BPM Console domain to **Per Site Privacy Actions** as **Allow** in **Tools > Internet Options > Privacy > Sites**.

6.30 Displaying Workitem (Task) Details Page in External Web Applications

You can display Workitem (Task) details page in external web applications, outside Interstage BPM Console. To do this, you use the HTTP POST method (with required parameters) and call the REST URL from the external application.

Note: In Interstage BPM, if SSO authentication is enabled, this functionality is not supported. It may cause an error.

Note: If you want another user to access just a single workitem, you can create a URL using the workitem id that is displayed in the **My Tasks** list. You can use
`http://<hostname>:<port>/console/<tenant_name>/<application_id>/w<WorkItemId>/taskDetails.page`
to create a URL and send it to another user. This URL is called REST URL.

While using the REST URL to display the Workitem details page in external web applications, the following items would be displayed:

- QuickForms (if available)
- Attachments
- Make choice
- Comments

You can also perform the following actions that the REST URL supports:

- Actions related to Forms
- Adding/removing attachments
- Checking in/checking out attachments
- Adding/deleting comments
- Using Make Choice action

Note: The approach on how to pass parameters to this URL via POST method can be decided by the external web application owner. For example, it can be as part of an HTML form having all parameters as listed in the sample implementation.

HTTP POST Details

To display the Workitem details page, you need the following information to construct the POST request:

- The REST URL of the Workitem details page
- The following POST parameters:
  - `username` – the Console username for which the process has to be displayed
  - `password` – the password for the username

Note: All parameter names and values above are case sensitive.

The REST URL for Workitem details page is:

`http://<hostname>:<port>/console/<tenant_name>/<application_id>/w<WorkItemId>/taskDetails.page`

Here,

- `hostname` is the machine name/IP on which Interstage BPM is setup
- `port` is the port for which Interstage BPM is configured
- `tenant_name` is name of the tenant
- `application_id` is name of the application that has the process instance
- `WorkItemId` is the id of the workitem which has to be displayed

Note: If Interstage BPM has been configured for secure access, use '`https`' instead of '`http`' in all URLs above.

Sample Implementation

A sample implementation of using the POST method is shown below. This sample displays the Workitem details page inside a frame.

```
<FORM METHOD=POST
ACTION="http://<hostname>:<port>/console/<tenant_name>/<application_id>/w<%=WorkItemId%>/
 taskDetails.page"
 name="iimForm" id="iimForm" target="frm">
    <INPUT TYPE="hidden" NAME="password" value="<%=password%>">
    <INPUT TYPE="hidden" NAME="username" value="<%=username%>">
</FORM>

<iframe name="frm" id="frm" frameborder="0" marginwidth="0" width="100%"
  height="1000px" marginheight="0" leftmargin="0" topmargin="0">
</iframe>
```

Restrictions

- If both the external web application as well as Interstage BPM Console are using the same domain (that is, having the same IP address while accessing the URL), and login parameters are missing or not passed correctly to the REST URL, then:
  - The Workitem details page will not load correctly when called inside frames.
  - No login screen will appear; the frame's parent page will reload.
- If both the external web application and Interstage BPM Console are using different domains (for example, the external application using local domain and Interstage BPM Console using a remote domain) and Interstage BPM Console is called inside the frame, then Interstage BPM Console does not operate properly. To avoid this:
  - Do not use frames in the scenario described above (different domains). If you need to use frames, either of the following settings is required for Internet Explorer:
    - Add Interstage BPM Console address to local Intranet or to **Trusted sites** in **Tools > Internet Options > Security**.
    - Add Interstage BPM Console domain to **Per Site Privacy Actions** as **Allow** in **Tools > Internet Options > Privacy > Sites**.

7 Administration

There exist two administrative roles in Interstage Business Process Manager:

- **Super User**: an administrator whose only responsibilty is to create and manage tenants, and manage the Interstage BPM Server. A Super User cannot administrate tenant users, process definitions, process instances, or work items. A Super User uses the **Interstage BPM Tenant Management Console** for administration of tenants. For information about administrative functions performed by a Super User refer the *Interstage Business Process Manager Console Tenant Management Guide* and *Interstage BPM Administration Guide*.
- **Tenant Owner**: a tenant user in an administrative role. The Tenant Owner is created by the Super User.

In non-SaaS mode, since there is only a single default tenant, the Super User and Tenant Owner may be the same person but may need to log in as a Super User or Tenant Owner to perform the different administrative functions.

This chapter provides programming examples, using the Model API, for administrative operations performed by a Tenant Owner.

A Tenant Owner can perform the following administrative functions:

- Logging in/logging out a tenant owner
- Tenant user administration (hereafter referred to as user administration)
- Process definition administration
- Process instance administration
- Work item administration

You can find the complete programming code of the examples presented in this chapter in the `Adminstration.java` and the `SampleLocalUserManagement.java` sample files.

For implementing the examples, you need the following packages from the Model API:

- **`com.fujitsu.iflow.model.util`**

  Contains low level utility classes that are commonly used by other classes and interfaces.

- **`com.fujitsu.iflow.model.workflow`**

  Contains interfaces that manage information required by process definitions and process instances. This includes providing objects that represent nodes, arrows, forms, attachments, work items and permission levels.

For more information on the Model API in general, refer to the API Javadoc.

7.1 Logging In/Logging Out an Administrator (Tenant Owner)

For administering process definitions or tenant users, an administrator (tenant owner) must be logged in to the Interstage BPM Server. Logging in an administrator means to create a session, i.e. a `WFAdminSession` object. The session is finished when an administrator is logged out again.

To log in/log out an administrator:

1. Create a new `WFAdminSession` object.

   ```
   adminSession = WFObjectFactory.getWFAdminSession();
   ```

   Use the `WFObjectFactory` class for accessing workflow objects. `getWFAdminSession()` then creates a `WFAdminSession` object.

2. Initialize the session with the appropriate configuration file.

   You can use the sample `iFlowClient.properties` file located in `<engine directory>/client/samples/examples/classes`.

   Either use this file or write the properties into a new file, which must be located in the current runtime directory. For the location of the current runtime directory, refer to section *Location of Properties Files* on page 58.

Note: Ensure you specify the tenant name logged in for the property file used. The tenant name is specified for value of `WFObjectFactory.TENANT_NAME(TenantName)`. `TenantName=Default` is specified for login to `Default` tenant.

Note: In the `iFlowClient.properties` file, any backslashes "\" or colons ":" are escaped by backslashes. For example, a server address is specified like this:

`ibpmhost\:49950`

When loading the `iFlowClient.properties` file using the `java.util.Properties.load()` method, escape characters will automatically be taken into account. If you use another way to load the properties, make sure that you handle any escape characters correctly. For details about escape sequences that may occur in the `iFlowClient.properties` file, refer to the Java documentation of the `java.util.Properties.store()` method.

Load the configuration file `iFlowClient.properties` for the session:

```
Properties sessionProps = new Properties();
   sessionProps.load(new FileInputStream("iFlowClient.properties"));
```

Initialize the session:

```
adminSession.initForApplication(null, sessionProps);
```

3. Log in an administrator.

```
adminSession.logIn(server, adminName, password);
```

The parameter `server` is retained for compatibility purposes. When the administrator logs into Interstage BPM Server, this parameter is not evaluated. Its value should be set to a STRING but not null.

4. After the administration tasks are completed, the administrator has to log out from the `WFAdminSession`. To log out an administrator:

```
if (adminSession != null ) {
   adminSession.logOut();
   }
```

7.2 Choosing a Workflow Application

After logging in, choose a workflow application within which you want to operate, using `WFSession.chooseApplication()`

```
adminSession.chooseApplication(myApplicationID);
```

Note: Choosing an application is optional. To know about detailed behavior when an application is not chosen, refer the `ApplicationModeSecurity` parameter in the *Interstage BPM Server Administration Guide*.

7.3 User and Group Administration

Every tenant user (hereafter referred to as a user) that is to work with Interstage BPM needs a user account and must be assigned to one or more groups.

User Accounts

Depending on how Interstage BPM has been configured during setup, user accounts are managed either in Interstage BPM's local user store or in a Directory Service (remote user store).

If you are using the local user store, you can add users, delete users and reset their password using the `WFAdminSession` interface.

If you are using a Directory Service, you use the functions of the Directory Service to add and delete users.

Groups

A group is a collection of users who share a function within an organization. For example, a `Manager` group might contain the first-line managers in an organization. In Interstage BPM, groups are used to determine who is responsible for carrying out a task in a process.

Groups can be managed in Interstage BPM's local group store, in a Directory Service (remote group store), or in both systems. If you are using the local group store, you can manage it using the `WFAdminSession` interface. The `WFAdminSession` interface has operations for creating groups, adding groups to other groups, adding users to groups, and so on.

Because groups can be members of other groups, you can model your organizational hierarchy into a corresponding group hierarchy. Groups can be nested to any depth.

Note: Users and groups can be managed in different stores. For example, an organization might manage user accounts in a Directory Service and groups in the local group store.

7.3.1 Managing Local Users

Prerequisite: Interstage BPM has been configured to use its local user store. For more information, refer to the *Interstage Business Process Manager Server and Console Installation Guide*.

The `WFAdminSession` interface allows you to retrieve a list of all local users, add local users, reset their password and delete local users.

- **To retrieve a list of all local users:**

```
User[] localUser = adminSession.getLocalUsers();
```

- **To add a local user:**

```
//Check whether the user already exists
User testUser = WFObjectFactory.getUser(sampleUser);
adminSession.createUser(testUser, usrPassword);
```

The user ID and the password can contain up to 200 characters. Use only alphanumeric characters, hyphens and underscore characters ("_"). User names must not begin with an at character ("@") as this is used to identify Agents in Interstage BPM.

- **To reset the password of a local user:**

```
adminSession.resetPassword(sampleUser, usrPassword);
```

Note: If you want users to be able to change their password, use `changePassword()` from the `WFSession` interface.

- **To delete a local user:**

```
adminSession.deleteUser(userID);
```

The user will be deleted and removed from all local groups to which the user belongs.

7.3.2 Managing Local Groups

This section introduces the most important operations for managing the local group store. In the programming sample, a hierarchy of three groups is created and users are assigned to those groups. The following figure shows the organizational structure that will be created:

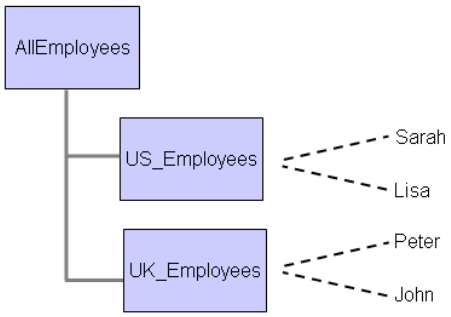

Figure 21: Sample Groups and Users

To manage local groups:

1. Create the required local groups using `createGroup()` from the `WFAdminSession` interface.

   The following sample creates an `AllEmployees`, `US_Employees` and `UK_Employees` group. Before creating a group, the sample checks whether the group already exists.

```
group = WFObjectFactory.getGroup("AllEmployees ",
        "ALL Employee group");
adminSession.createGroup(group);
group = WFObjectFactory.getGroup("US_Employees ",
        "US Employee group");
```

```
adminSession.createGroup(group);
group = WFObjectFactory.getGroup("UK_Employees ",
        "UK Employee group");
adminSession.createGroup(group);
```

The group name can contain up to 200 characters. Use only alphanumeric characters, hyphens and underscore characters ("_"). Group names must not begin with an at character ("@") as this is used to identify Agents in Interstage BPM.

2. You can retrieve all groups available in Interstage BPM using `getAllGroups()`.

```
Group[] LocalGroupList  = adminSession.getAllGroups();
```

3. Create the users you want to assign to the local groups in Interstage BPM. In this sample, the users Lisa, Sarah, Peter and John are created:

```
testUser = WFObjectFactory.getUser("UserSarah");
adminSession.createUser(testUser, "SomePassword");
testUser = WFObjectFactory.getUser("UserLisa");
adminSession.createUser(testUser, "SomePassword");
testUser = WFObjectFactory.getUser("UserPeter");
adminSession.createUser(testUser, "SomePassword");
testUser = WFObjectFactory.getUser("UserJohn");
adminSession.createUser(testUser, "SomePassword");
```

4. Assign the newly created users to the local group using `addUserToGroup()` from the `WFAdminSession` interface. In the sample, Lisa and Sarah are assigned to `US_Employees`, John and Peter are assigned to `UK_Employees`.

```
adminSession.addUserToGroup("UserSarah", "US_Employees");
adminSession.addUserToGroup("UserLisa", "US_Employees");
adminSession.addUserToGroup("UserPeter", "UK_Employees");
adminSession.addUserToGroup("UserJohn", "UK_Employees");
```

You can assign the same user to as many groups as you need.

You may assign any Interstage BPM user to any local group. The users to be assigned may be stored in Interstage BPM's local user store or in a Directory Service.

5. If you wish to create group hierarchies, use `addGroupToGroup()` to assign a child group to a parent group. In the sample, `US_Employees` and `UK_Employees` are associated with the `AllEmployees` group.

```
adminSession.addGroupToGroup("US_Employees", "AllEmployees");
adminSession.addGroupToGroup("UK_Employees", "AllEmployees");
```

When assigning a child group to a parent group, all members of the child group also become members of the parent group.

You can assign the same group to multiple groups.

Note: Do not create cyclic groups. For example, do not add `Group1` as a child group of `Group2` and `Group2` as a child group of `Group1`.

6. If a user shall no longer be a member of a group, you can remove the user from that group.

```
adminSession.removeUserFromGroup(sampleUser, roleName);
```

7.3.3 Listing Logged-In Users

To retrieve a list of all logged-in users:

- Use `getAllUserAgentInfo()` from the `WFAdminSession` interface.

```
UserInfo[] userInfoList = adminSession.getAllUserAgentInfo();
```

7.3.4 Logging Out Users

- **To log out a single user:**

  Use `logoutUser()` from the `WFAdminSession` interface.

- **To log out all users:**

  Use `logoutAllUsers()` from the `WFAdminSession` interface.

The following example shows how to log out all users listed in given `userInfoList`, except the administrator:

```
if (userInfoList != null) {
   boolean loggedOut = false;
   for (int i = 0; i < userInfoList.length; i++) {
      // Control the name of the user to get a user which is not the
      // current administrator.
      if (!userInfoList[i].getName().equals(adminName)) {
         adminSession.logoutUser(userInfoList[i].
         getClientID());

         loggedOut = true;
      }
   }
}
```

7.3.5 Resetting the User and Group Cache

In an administration session, `WFAdminSession`, an administrator can reset the cached data for groups and users from the Directory Server.

At startup, the Interstage BPM Server retrieves the groups and users from the local or remote store and stores them in a cache. In a `WFAdminSession`, an administrator can reset the cached data for groups and users.

- **To reset the group list:**

```
adminSession.resetLDAPGroupsList();
```

  `resetLDAPGroupsList()` clears the cache for groups immediately. It resets the cached memory object of the groups defined in the local or remote store. Any modifications to the group list in the local or remote store are reloaded into the memory object.

- **To reset the users list:**

```
adminSession.resetLDAPUsersList();
```

  `resetLDAPUsersList()` clears the cache for users immediately. It resets the cached memory object of the users defined in the local or remote store. Any modifications to the users list in the local or remote store are reloaded into the memory object.

Note: When using a remote store, the group cache is automatically refreshed at a regular interval. The expiration date for the LDAP group cache is set in the `LDAPGroupCacheAgeSec` parameter of the Interstage BPM Server. Refer to the *Interstage Business Process Manager Server Administration Guide* for more information.

7.4 Process Definition Administration

This section describes the following common process definition administration tasks:

- Listing process definitions
- Publishing process definitions
- Archiving process definitions
- Deleting process definitions
- Deleting archived process definitions
- Importing a process definition from an XPDL file
- Exporting a process definition to an XPDL file

Archiving and deleting of process instances and process definitions is an important part of the long term management of a running process environment for keeping the running database table clean in in good shape. Archived process instances and definitions is a step between active use and deleting. An archived object is removed from the active tables to prevent performance penalities, but is still available to applications from the slower archive tables. Process objects may alternatively be archived (exported) to XML files so the process information can be preserved outside the database. Deleting removes all record of the process objects. Archiving (exporting) and deleting can be performed at the same time.

7.4.1 Listing Process Definitions

For listing process definitions, use the `WFObjectList` interface from the `com.fujitsu.iflow.model.workflow` package. The interface retrieves a list of the existing `WFObjects` from the Interstage BPM Server.

To list process definitions:

1. Build a filter using the `Filter` class.

   Possible filter criteria for process definitions are:

   - `AllArchivedPlans`: Retrieves all archived process definitions.
   - `AllPlans`: Retrieves all process definitions.
   - `MyInactivePlans`: Retrieves all inactive process definitions.
   - `MyPlans`: Retrieves all process definitions belonging to the logged in user.

2. Use `openBatchedList()`.
3. Iterate the list of returned objects matching the filter criteria set with `openBatchedList()`.

Example

```
Plan[] planList = null;
if (filter == Filter.AllArchivedPlans
   || filter == Filter.AllPlans
   || filter == Filter.MyInactivePlans
   || filter == Filter.MyPlans) {
```

```
      WFObjectList wfObjectList =
         WFObjectFactory.getWFObjectList(adminSession);

      wfObjectList.openBatchedList(filter);

      Vector tmpList = new Vector();
      int batchSize = 50;

      Object[] elements = wfObjectList.getNextBatch(batchSize);

      Plan plan = null;

      while (elements != null && elements.length > 0) {
         for (int i = 0; i < elements.length; i++) {

            plan = (Plan) elements[i];
            // Add process definition to temporary list
            tmpList.add(plan);
         }
      // Read next list of process definitions regarding to filter
      elements = wfObjectList.getNextBatch(batchSize);
   }
   // Copy the process instance of temporary list into a
   // ProcessInstance array that will be returned
   planList = new Plan[tmpList.size()];
   planList = (Plan[]) tmpList.toArray(planList);

   return planList;
}
```

7.4.2 Publishing Process Definitions

To publish a process definition:

- Use `publishPlan()` from the `WFAdminSession` interface.

When a process definition is published, its state changes to "published".

Example

The following example shows how to publish the first process definition from a list of process definitions (`planList`) that are in private or draft state.

```
if (planList != null ) {
   boolean published = false;
   for (int i = 0; i < planList.length; i++) {
      if (planList[i].getState() == Plan.STATE_DRAFT
         || planList[i].getState() == Plan.STATE_PRIVATE) {
         adminSession.publishPlan(planList[i].getId());
      }
   published = true;
   }
}
```

7.4.3 Archiving Process Definitions

When a process definition is archived, its state changes to "archived". If you try to archive a process definition that is already archived, Interstage BPM assumes that you want a published process

definition to be archived. In this case, the process definition still exists in the regular process definition list having the state "deleted".

To archive a process definition:

- Use `archivePlan()` from the `WFAdminSession` interface.

```
adminSession.archivePlan(planList[i].getId());
```

Once you have archived a process definition, you can retrieve it again as follows:

1. Get the list of all archived process definitions for retrieving the IDs of the archived process definitions. Use the `AllArchivedPlans` filter criterion for doing so.
2. To retrieve a specific archived process definition, use `getArchivedPlan(id)` from the `WFAdminSession` interface.

```
adminSession.getArchivedPlan(planList[i].getId());
```

Note that you can only perform a subset of functions on archived process definitions, such as retrieving information on them. All functions that modify a process definition are not supported for archived process definitions (e.g. `setName`).

7.4.4 Deleting Process Definitions

An administrator can delete process definitions from all users. A user, however, can only delete a process definition of which he/she is the owner.

To delete a process definition:

- Use `deletePlan()` from the `WFAdminSession` interface.

```
adminSession.deletePlan(planList[i].getId());
```

The method selects a specified process definition from the database. The process definition must be in "draft" or "private" state. If the process definition is in any other state, its state changes to "deleted", but it will not be physically deleted from the database.

7.4.5 Deleting Archived Process Definitions

To delete an archived process definition from the database:

- Use `deleteArchivedPlan()` from the `WFAdminSession` interface.

```
adminSession.deleteArchivedPlan(archivedPlanList[i].getId());
```

7.4.6 Importing a Process Definition from an XPDL File

To import an XPDL file:

1. Create an empty process definition object with `WFObjectFactory.getPlan()`.
2. Set the current session to the process definition object with `setWFSession()`.
3. Create a process definition object from an XPDL file with `convertFromXPDL()`.

Example

For the following example to work properly, make sure that the defined import directory `DIR_PATH_IMPORT` contains some files.

```
File importDir = new File(DIR_PATH_IMPORT);

if (importDir.exists() && importDir.isDirectory()) {
   File[] fileList = importDir.listFiles();
   String filePath;
   boolean fileFound = false;
   Plan plan = null;

   for (int i = 0; i < fileList.length; i++) {
      filePath = fileList[i].getAbsolutePath();

      if (filePath.endsWith(".xml") ||
         filePath.endsWith(".xpdl")) {
         plan = WFObjectFactory.getPlan();
         plan.setWFSession(adminSession);
         plan.convertFromXPDL(new FileInputStream(fileList[i]));
         break;
      }
   }
}
```

7.4.7 Exporting a Process Definition to an XPDL File

You can convert a process definition to XPDL data using `convertToXPDL()` from the `Plan` interface.

To export a process definition to an XPDL file:

1. Take the first process definition from a given list of process definitions.

```
if (planList != null && planList.length > 0) {
   Plan plan = planList[0];
   . . .
}
```

2. Check whether the destination directory exists. If not, a new directory is created.

```
File exportDir = new File(DIR_PATH_EXPORT);
   if (!exportDir.exists() || !exportDir.isDirectory()) {
      exportDir.mkdir();
}
```

3. Generate an absolute file name for the export file.

```
String fileName = DIR_PATH_EXPORT + "/" + "PLAN_" +
plan.getName() + "_" + plan.getId() + ".xml";
```

4. Create a new file object.

```
File file = new File(fileName);
if (!file.exists()) {
   file.createNewFile();
   . . .
}
```

5. Convert the process definition to XPDL.

```
plan.convertToXPDL(new FileOutputStream(file));
```

7.5 Process Instance Administration

This section describes the following process instance administration tasks:

- Listing process instances
- Changing the ownership of a process instance
- Archiving process instances
- Suspending process instances
- Aborting process instances
- Deleting process instances
- Deleting archived process instances

7.5.1 Listing Process Instances

For listing process instances, use the `WFObjectList` interface from the `com.fujitsu.iflow.model.workflow` package. This interface retrieves a list of the existing `WFObjects` from the Interstage BPM Server.

To list process instances:

1. Build a filter using the `Filter` class.

   Possible filter criteria for process instances are:

   - `AllActiveProcesses`: Retrieves all active process instances.
   - `AllArchivedProcesses`: Retrieves all archived process instances.
   - `AllInactiveProcesses`: Retrieves all inactive process instances.
   - `AllProcesses`: Retrieves all process instances.
   - `AllProcessesInErrorState`: Retrieves all inactive process definitions.
   - `MyActiveProcesses`: Retrieves all active process instances initiated by the logged in user.
   - `MyInactiveProcesses`: Retrieves all inactive process instances initiated by the logged in user.
   - `MyProcesses`: Retrieves all process instances initiated by the logged in user.

2. Use `openBatchedList()`.
3. Iterate the list of returned objects matching the filter criteria set with `openBatchedList()`.

Refer to the `Administration.java` sample source file located in your `<engine directory>/client/samples/examples/sources` directory for the detailed sample code on how to list process instances.

7.5.2 Changing the Ownership of a Process Instance

Only an administrator and the current process instance owner can change the ownership of a process instance. Process ownership must be set at the process definition level. The process owner is determined by the setting of the attribute "Owner Role". The attribute "Owner Role" is evaluated by the Interstage BPM Server before activating a new process instance.

Use `setOwners()` from the `ProcessInstance` interface to set the owner of a running process instance. Before actually changing the ownership, this method checks whether the user or group who is to become the new owner exists; if one of the new owners does not exist, the process instance will go into error state.

In the following example, the ownership of the first process instance from a given list is assigned to the first found user from a given user list.

To change the ownership of a process instance:

1. List all existing user groups.

```
if (procInstList != null) {
   String[] owners = null;
   String group = "";
   DirectoryServices ds = WFObjectFactory.getDirectoryServices(
      adminSession, adminName, password);
   String[] groups = ds.getUserGroups();
   if (groups != null && groups.length > 0) {
      group = groups[0];
   }
   owners = ds.getUserList(group);
   if (owners != null) {
      boolean assigned = false;
      ...
   }
}
```

2. Take the first user to get the new owner for a process instance, and start the edit-mode.

```
for (int i = 0; i < procInstList.length; i++) {
  procInstList[i].startEdit();
  ...
}
```

3. Set the owner to the process instance.

```
procInstList[i].setOwners(owners);
```

4. Commit the edit-mode.

```
procInstList[i].commitEdit();
assigned = true;
```

7.5.3 Archiving Process Instances

Only process instances that are completed, aborted or in error state can be archived. Therefore, before archiving a process instance, you have to check for the process instance's current state. You can identify the state of a completed process instance using the constant `STATE_CLOSED` from the `ProcessInstance` interface. Use the constants `STATE_ERROR` or `STATE_ABORTED` to check whether a process instance is in error state or has been aborted.

To archive a process instance:

1. Check a given list of process instances for process instances that are completed, aborted or in error state. To do so, check the current state of the process instance using `getState()` from the `ProcessInstance` interface.

```
 if (procInstList[i].getState() == ProcessInstance.STATE_CLOSED ||
  procInstList[i].getState() == ProcessInstance.STATE_ABORTED ||
  procInstList[i].getState() == ProcessInstance.STATE_ERROR)
 {
   ...
}
```

2. To archive a process instance, use `archiveClosedProcess()` from the `WFAdminSession` interface.

```
adminSession.archiveClosedProcess(procInstList[i].getId());
```

Once you have archived a process instance, you can retrieve it again as follows:

1. Get the list of all archived process instances for retrieving the IDs of the archived process instances. Use the `AllArchivedProcesses` filter criterion for doing so.
2. To retrieve a specific archived process instance, use `getArchivedProcess(id)` from the `WFAdminSession` interface.

```
adminSession.getArchivedProcess(procInstList[i].getId());
```

Note that you can only perform a subset of functions on archived process instances, such as retrieving information on them. All functions that modify a process instance are not supported for archived process instances (e.g. `setName`).

7.5.4 Suspending Process Instances

When suspending a running process instance, its state changes to `STATE_SUSPENDED`. If this process instance has any work items, they are also suspended. If the process instance has any running subprocess instances, they are also suspended.

To suspend a process instance:

1. Check a given list of process instances for running process instances. To do so, check the current state of the process instance using `getState()` from the `ProcessInstance` interface.

```
 if (procInstList[i].getState() == ProcessInstance.STATE_RUNNING)
  {
   ...
}
```

2. To suspend a running process instance, use `suspend()` from the `ProcessInstance` interface.

```
procInstList[i].suspend();
```

Note: If defined on process definition level or Node level, a set of Java Actions may be executed before the process instance changes to `STATE_SUSPENDED`. This Java Action Set will become active as soon as you call the `suspend()` command. For example, a notification email is sent to the owner of the process instance, or additional log file entries are written. Refer to section *Using onAbort, onSuspend, onResume Java Action Sets* on page 110 for details on how to define an `onSuspendJavaActionSet`.

You can resume a suspended process instance using `resume()` from the `ProcessInstance` interface. The state of the process instance is then changed to `STATE_RUNNING`.

Again, if defined on process definition level or Node level, a set of Java Actions may be executed before the process instance changes to `STATE_RUNNING` again. This Java Action Set will be executed as soon as you call the `resume()` command.

7.5.5 Aborting Process Instances

When aborting a process instance, its state changes to `STATE_ABORTED`. If this process instance has any work items, they are removed. If the process instance has any running subprocess instances, they are also aborted.

To abort a process instance:

- Use `abort()` from the `ProcessInstance` interface.

  This method can only be used by an administrator or the process instance owner. Other users are not allowed to abort a process instance.

Example

In the following example, a given list of process instances is checked for running (constant `STATE_RUNNING`), suspended (constant `STATE_SUSPENDED`), and error (constant `STATE_ERROR`) process instances using `getState()` from the `ProcessInstance` interface. `getState()` returns the current state of the process instance. Then the running or suspended process instance which is found inside the list is aborted.

```
if (procInstList != null ) {
   boolean aborted = false;
   for (int i = 0; i < procInstList.length; i++) {
      if (procInstList[i].getState() ==
        ProcessInstance.STATE_RUNNING
        || procInstList[i].getState() == ProcessInstance.STATE_SUSPENDED
        || procInstList[i].getState() == ProcessInstance.STATE_ERROR)
      {
            procInstList[i].abort();
            aborted = true;
      }
   }
}
```

Note: If defined on process definition level or Node level, a set of Java Actions may be executed before the process instance changes to `STATE_ABORTED`. This Java Action Set will be executed as soon as you call the `abort()` command. For example, a notification email is sent to the owner of the process instance, or additional log file entries are written. Refer to section *Using onAbort, onSuspend, onResume Java Action Sets* on page 110 for details on how to define an `onAbortJavaActionSet`.

7.5.6 Deleting Process Instances

To delete a process instance:

- Use `deleteProcessInstance()` from the `WFAdminSession` interface.
  This method deletes a process instance from the database, using a process instance ID.

Example

In the following example the first valid process instance from a given list is deleted:

```
if (procInstList != null) {
   boolean deleted = false;
   for (int i = 0; i < procInstList.length; i++) {
      adminSession.deleteProcessInstance
      (procInstList[i].getId());
      deleted = true;
      break;
   }
}
```

7.5.7 Deleting Archived Process Instances

To delete an archived process instance from the database:

- Use `deleteArchivedProcess()` from the `WFAdminSession` interface.

```
adminSession.deleteArchivedProcess(archivedProcInstList[i].getId());
```

7.6 Work Item Administration

This section describes the following work item administration tasks:

- Reassigning work items from one user to another
- Refreshing work items

7.6.1 Reassigning Work Items to Another User

To reassign a work item to another user:

- Use `reassign()` from the `WFAdminSession` interface.

Example

The following example shows how to reassign work items from one user to another existing user.

```
if (workItemList != null && workItemList.length > 0) {
   WorkItem workItem = workItemList[0];
   UserInfo[] userList = listUsers();
   String oldAssignee = "";
   String newAssignee = "";
   if (userList != null && userList.length > 0) {
      oldAssignee = workItem.getAssignee();
      newAssignee = userList[0].getName();
      adminSession.reassign(oldAssignee, newAssignee,
      workItem.getProcessInstanceId(),
      workItem.getId(),false);
```

```
    }
}
```

7.6.2 Refreshing Work Items

If an administrator adds or deletes a user from a group in the directory, any running activities that have that group as an assignee will not be automatically updates, and the outstanding workitems will not be created or removed to match the group. In general operation, workitems are created to match the group as it is at the time that the activity is started, and changes in the group after that are purposefully ignored. In some situations, however, one would like the workitems of an activity that is already started to be altered to match the changed group members. To update the workitems, the administrator has to refresh the work items.

Refreshing the work items can take some time because often several iterations are necessary to complete the task. The time needed depends on the number of the active tasks in the database. The execution time can be reduced by using options for selecting the work items to be refreshed:

- The work items of a process instance.
- The work items of all process instances that belong to a process definition.
- The work items for a group.
- All existing work items.
- **To refresh the work items of a process instance:**

```
boolean ignoreActivityWithRoleScript = true;
String refreshResult = null;

if (procInst != null) {
   long procInstId = procInst.getId();
   refreshResult = adminSession.refreshWIforProcess(procInstId,
      ignoreActivityWithRoleScript);
}
```

- **To refresh the work items of all process instances that belong to a process definition:**

```
if (procDef != null) {
   long procDefId = procDef.getId();
   refreshResult = adminSession.refreshWIforPlan(procDefId,
      ignoreActivityWithRoleScript);
}
```

- **To refresh the work items for a group:**

```
String group = "SampleGroup";
refreshResult = adminSession.refreshWIforGroup(group,
   ignoreActivityWithRoleScript);
```

- **To refresh all work items:**

```
refreshResult = adminSession.refreshWorkItems
   (ignoreActivityWithRoleScript);
```

Note: If the `SupportGroupWorkItem` parameter of the Interstage BPM Server is set to `true`, the filter `Filter.MyActiveWorkItems` is not supported, i.e. the work items cannot be retrieved with this filter.

In this case you have to define your own filter for retrieving the work items to be refreshed. Refer to section *Listing Work Items* on page 80 for details on building a filter for work items. Refer to the *Interstage Business Process Manager Server Administration Guide* for details on the `SupportGroupWorkItem` parameter.

The result of the refreshing is formatted as an XML string, which lists the successful and the failed operations. The following gives an example of such an XML string:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <Processes>

      <success>
         <Process Name="procName" id="123"></Process>
      </success>

       <failed>
         <Process Name="procName" id="456" Status="ignored"
            message="error message" />
         <Process Name="procName" id="123">
            <Activities>
               <Activity Id="123456" Name="activityName"
                  Status="ignored" message="error message" />
            </Activities>
         </Process>
      </failed>
   </Processes>
```

The `<success>` tags contain a list of process instances for which the refreshing of the work items was successful. The `<failed>` tags contain a list of process instances for which the refreshing failed. The `<Activities>` tags contain activities of a process instance for which the refreshing failed.

8 Retrieving History Information

For each process instance, the information concerning the workflow is stored in an Interstage BPM database. In the History table of the database, each single step in a workflow of a process instance is stored.

You have the following possibilities to retrieve history information from the `History` table:

- With the Model API
- With Custom Java Action Type or Agent
- With SQL Statements

8.1 Retrieving History Information With the Model API

To retrieve history information for a process instance and node instance using the Model API:

1. Use the following code as basis:

   a) **For Process Instance:**

   `getHistory()` retrieves all the history data for the related process instance.

```
import com.fujitsu.iflow.model.workflow.ProcessInstance;
import com.fujitsu.iflow.model.util.IflowEnumeration;

int processHistoryFields[] = {
   ProcessInstance.HISTORY_ID,
   ProcessInstance.HISTORY_TIME_STAMP,
   ProcessInstance.HISTORY_EVENT_CODE,
   ProcessInstance.HISTORY_EVENT_TYPE,
   ProcessInstance.HISTORY_RESPONSIBLE,
   ProcessInstance.HISTORY_PRODUCER_TYPE,
   ProcessInstance.HISTORY_PRODUCER_ID,
   ProcessInstance.HISTORY_CONSUMER_TYPE,
   ProcessInstance.HISTORY_CONSUMER_ID,
   ProcessInstance.HISTORY_PROCESSINSTANCE_ID,
   ProcessInstance.HISTORY_ISHANDLED_CODE
};

ProcessInstance procInst =
   WFObjectFactory.getProcessInstance(procID, session);

IflowEnumeration sElements =
   procInst.getHistory(processHistoryFields);
```

   b) **For Node Instance:**

   `getHistory()` retrieves all the history data for the related node instance.

```
import com.fujitsu.iflow.model.workflow.ProcessInstance;
import com.fujitsu.iflow.model.workflow.NodeInstance;
import com.fujitsu.iflow.model.util.IflowEnumeration;

 int nodeHistoryFields[] = {
   ProcessInstance.HISTORY_ID,
   ProcessInstance.HISTORY_TIME_STAMP,
   ProcessInstance.HISTORY_EVENT_CODE,
   ProcessInstance.HISTORY_EVENT_TYPE,
   ProcessInstance.HISTORY_RESPONSIBLE,
```

```
};

ProcessInstance procInst =
   WFObjectFactory.getProcessInstance(procID, session);

NodeInstance nodeInst = procInst.getNodeInstance(nodeInstId);

IflowEnumeration sElements =
    nodeInst.getHistory(nodeHistoryFields);
```

Note: In case of Node Instance history, the `HISTORY_CONSUMER_ID` will be the same for all the history items and that will be the Node Instance Id of called node instance.

The history columns that are to be retrieved are defined by the integer array, which is passed as argument to the method.

2. Navigate through the resulting objects of type `IflowEnumeration` to access each row for the history entries.

```
while(sElements.hasMoreElements())
   {
      Hashtable htblEvent = (Hashtable) sElements.nextElement();
      // Here the code from the section "Retrieving the History
Information From a Hashtable"
   }
```

The following table shows the mapping of the constants in the `ProcessInstance` class to the columns of the `History` table.

Constants in ProcessInstance Class	Column Name
HISTORY_ID	historyId
HISTORY_TIME_STAMP	createdTime
HISTORY_EVENT_CODE	eventCode
HISTORY_EVENT_TYPE	eventType
HISTORY_PRODUCER_TYPE	producerType
HISTORY_PRODUCER_ID	producerId
HISTORY_CONSUMER_TYPE	consumerType
HISTORY_CONSUMER_ID	consumerId
HISTORY_ISHANDLED_CODE	isHandled
HISTORY_PROCESSINSTANCE_ID	processInstanceId
HISTORY_RESPONSIBLE	responsible
HISTORY_ERROR_MESSAGE	eventData
HISTORY_MODIFIED_TRACKEDUDAS	eventData

8.2 Retrieving History Information Using Custom Java Action Type or Agent

To retrieve history information for a process instance and node instance using Custom Java Action Type or Agent:

1. Create a custom JavaAction class that uses `getProcessHistory()` or `getActivityHistory()` of the `ServerEnactmentContext`.
2. Use the following code as basis:

```
import com.fujitsu.iflow.server.intf.ServerEnactmentContext;
import com.fujitsu.iflow.model.workflow.ProcessInstance;
import com.fujitsu.iflow.model.workflow.NodeInstance;
import com.fujitsu.iflow.model.util.IflowEnumeration;

public class MyClass {
 // for process instance history
 public void retrieveProcessHistory (ServerEnactmentContext sec)
   throws Exception {
  int processHistoryFields[] = {
   ProcessInstance.HISTORY_ID,
   ProcessInstance.HISTORY_TIME_STAMP,
   ProcessInstance.HISTORY_EVENT_CODE,
   ProcessInstance.HISTORY_EVENT_TYPE,
   ProcessInstance.HISTORY_RESPONSIBLE,
   ProcessInstance.HISTORY_CONSUMER_TYPE,
   ProcessInstance.HISTORY_CONSUMER_ID,
  };

  IflowEnumeration sElements =
   sec.getProcessHistory(processHistoryFields);

  while (sElements.hasMoreElements()) {
   Hashtable htblEvent = (Hashtable) sElements.nextElement();
   // Retrieving history information from a Hashtable
  }
 }

 // for node instance history
 public void retrieveActivityHistory(ServerEnactmentContext sec)
   throws Exception {
  int activityHistoryFields[] = {
   ProcessInstance.HISTORY_ID,
   ProcessInstance.HISTORY_TIME_STAMP,
   ProcessInstance.HISTORY_EVENT_CODE,
   ProcessInstance.HISTORY_EVENT_TYPE,
   ProcessInstance.HISTORY_RESPONSIBLE,
  };

  IflowEnumeration sElements =
   sec.getActivityHistory(activityHistoryFields);

  while(sElements.hasMoreElements()) {
   Hashtable htblEvent = (Hashtable) sElements.nextElement();
   // Retrieving history information from a Hashtable
  }
 }
}
```

For information about defining Java Action Types, refer *Accessing Workflow Data Using the Server Enactment Context Interface* on page 93.

8.3 Retrieving History Information From a Hashtable

With the code in section *Retrieving History Information With the Model API* on page 238, you can retrieve all history data from a process instance. The results are stored in a hashtable. The following sections show how to retrieve the values for the single constants of the `ProcessInstance` class.

8.3.1 HISTORY_ID

To retrieve the history ID of an entry, use the following code:

```
long historyId = ((Long)htblEvent.get
   (new Integer(ProcessInstance.HISTORY_ID))).longValue();
```

Each entry in the `History` table has a unique identifier.

8.3.2 HISTORY_TIME_STAMP

To retrieve the timestamp of an entry, use the following code:

```
long timestamp = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_TIME_STAMP))).longValue();
```

To convert the long value of the date into a readable form, use standard Java statements.

8.3.3 HISTORY_EVENT_CODE

To retrieve the event code of an entry, use the following code:

```
long historyEventCode = ((Long) htblEvent.get
    (new Integer(ProcessInstance.HISTORY_EVENT_CODE))).longValue();
```

The event code indicates the type of the event. The following table lists possible values and their meaning:

Event Code	Description
0	Start (Default)
1	Activate
2	Make choice
3	Accept
4	Decline
5	Reassign
6	Exit
7	Create subprocess
8	Suspend work item

Event Code	Description
9	Resume work item
10	Start work
11	Stop work
12	Voting complete
13	Process created
14	Process migrated
15	Process recall
16	Deactivate target node
17	Activate source node
25	Generate future work items

8.3.4 HISTORY_EVENT_TYPE

To retrieve the event type of an entry, use the following code:

```
String eventType = (String) htblEvent.get(
   new Integer(ProcessInstance.HISTORY_EVENT_TYPE));
```

The event type specifies the name of an activity or the name of an arrow in Interstage BPM. If, for example, an arrow has the name `Create Purchase Requisition`, the result of `HISTORY_EVENT_TYPE` is `Create Purchase Requisition`.

8.3.5 HISTORY_PRODUCER_ID

To retrieve the producer ID of an entry, use the following code:

```
long producerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PRODUCER_ID))).longValue();
```

Depending on the producer type, the producer ID specifies different types of information. Refer to section *History Table* on page 246 for more details.

8.3.6 HISTORY_PRODUCER_TYPE

To retrieve the producer type of an entry, use the following code:

```
long lProdType=((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PRODUCER_TYPE))).longValue();
```

The producer type indicates the event responsible for creating the current event. The following table lists possible values and their meaning:

Producer Type	Description
0	Arrow
3	Activity

Producer Type	Description
7	Process
15	Timer

8.3.7 HISTORY_CONSUMER_ID

To retrieve the consumer ID of an entry, use the following code:

```
long consumerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_CONSUMER_ID))).longValue();
```

Depending on the consumer type, the consumer ID specifies different types of information. Refer to section *Rules for Navigating Through the History Entries* on page 245 for details.

8.3.8 HISTORY_CONSUMER_TYPE

To retrieve the consumer type of an entry, use the following code:

```
long lConsType = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_CONSUMER_TYPE))).longValue();
```

The consumer type indicates the type of the element that the event is intended for. The following table lists possible values and their meaning:

Consumer Type	Description
3	Activity
7	Process

For more information on how to interpret the consumer type, refer to section *Rules for Navigating Through the History Entries* on page 245.

8.3.9 HISTORY_ISHANDLED_CODE

To retrieve the value `IsHandled` of an entry, use the following code:

```
long isHandledCode = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_ISHANDLED_CODE)))
   .longValue();
```

The following table lists possible values and their meaning:

isHandled Code	Description
0	Not handled
1	Handled
2	Ignored
3	Error
4	Audit

isHandled Code	Description
5	Suspended

8.3.10 HISTORY_PROCESSINSTANCE_ID

To retrieve the process instance ID of an entry, use the following code:

```
long procInstId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PROCESSINSTANCE_ID)))
   .longValue();
```

8.3.11 HISTORY_RESPONSIBLE

To retrieve the responsible for the entry in the `History` table, use the following code:

```
String responsible = (String) htblEvent.get(
   new Integer(ProcessInstance.HISTORY_RESPONSIBLE));
```

Possible values are:

- `__process`

  System user ID to indicate the activity of the workflow engine.

- `<userID>`

  Any valid user ID to indicate the user who performed an action.

8.3.12 HISTORY_ADDITIONAL_INFO

To retrieve additional history information of an entry, use the following code:

```
Hashtable additionalHistoryInfo = (Hashtable)htblEvent.get(
new Integer(ProcessInstance.HISTORY_ADDITIONAL_INFO));
```

For more information, refer *Retrieving Additional History Information* on page 121.

8.3.13 HISTORY_ERROR_MESSAGE

To retrieve the error message of an entry, use the following code:

```
String errMsg = (String) htblEvent.get(new Integer
      (ProcessInstance.HISTORY_ERROR_MESSAGE));
```

This specifies the error message when the process is in error state.

8.3.14 HISTORY_MODIFIED_TRACKEDUDAS

To retrieve the modified tracked UDAs, use the following code:

```
TrackedUDARecord[] trackedUdaRec = (TrackedUDARecord[]) (htblEvent.get(
            new Integer(ProcessInstance.HISTORY_MODIFIED_TRACKEDUDAS)));
    String name = null;
    String oldValue = null;
    String newValue = null;
    String type = null;
```

```
int operation = 0;
for (int i = 0; i <trackedUdaRec.length; i++) {
    name = trackedUdaRec[i].getName();
    oldValue = trackedUdaRec[i].getOldValue();
    newValue = trackedUdaRec[i].getNewValue();
    type = trackedUdaRec[i].getType();
    operation = trackedUdaRec[i].getUDAOperation();
    .....
}
```

This specifies the tracked UDAs being modified during the Process Instance state transition.

8.3.15 Rules for Navigating Through the History Entries

To navigate through the entries in the `History` table of a process instance, the following rules are defined.

Rule 1

To navigate through the entries of the `History` table, only entries fulfilling one of the three following conditions are evaluated:

- **Activation event**

  Applies if all of the following criteria are met:
  - `consumerType` is defined as `3` (Node)
  - `eventType` is equal to `__Activate`
- **Node**

  Applies if all of the following criteria are met:
  - `consumerType` is defined as `3` (Node)
  - `eventType` is not equal to `__Activate`
  - `producerType` is defined as `7` (Process)
- **Arrow**

  Applies if all of the following criteria are met:
  - `consumerType` is defined as `3` (Node)
  - `eventType` is not equal to `__Activate`
  - `producerType` is defined as `0` (Arrow)

Rule 2

In case there is an entry of type 1, i.e. an activation event, only the timestamp is retrieved.

Rule 3

In case there is an entry of type 2, i.e. a User Task Node, the consumer ID represents the ID of the node instance in the process instance. Using the Model API, you can retrieve the node instance as follows:

```
// CONSUMER_ID represents the node instance ID

   long consumerId = ((Long)htblEvent.get(
```

```
   new Integer(ProcessInstance.HISTORY_CONSUMER_ID))).longValue();

   NodeInstance nodeInst = procInst.getNodeInstance(consumerId);

   // Now continue to retrieve the information from the object
   // of the class NodeInstance
```

Rule 4

In case there is an entry of type 3, i.e. an arrow, the producer ID represents the ID of the arrow instance in the process instance. Using the Model API, you can retrieve the node instances of source and target nodes as follows:

```
// PRODUCER_ID represents the arrow instance ID

   long producerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PRODUCER_ID))).longValue();

   ArrowInstance arrow = procInst.getArrowInstance(producerId);

   NodeInstance sourceNodeInst = arrow.getSourceNodeInstance();
   NodeInstance targetNodeInst = arrow.getTargetNodeInstance();

   // Now continue to retrieve information from the source and
   // target objects of the arrow instance
```

8.4 History Table

In the `History` table, each step in the workflow of a process instance is defined by a separate entry, i.e. a row. A step can be defined by a system action, for example "Creation of a process instance", or by the execution of an activity which is explicitly defined in the process definition, for example "Vote for Approval".

Note: The database schema described in this section could be changed in a future version of Interstage BPM. If you develop your own application based on the database schema, you need to change the application when the database schema is changed.

Column Name	Value/Sample	Remarks
`historyId`	`888919`	Unique identifier for each row in the `History` table.

Column Name	Value/Sample	Remarks
eventCode	0=Start (default) 1=Activate 2=MakeChoice 3=Accept 4=Decline 5=Reassign 6=Exit 7=CreateSubProcess 8=SuspendWorkItem 9=ResumeWorkItem 10=StartWork 11=StopWork 12=VotingComplete 13=ProcessCreated 14=ProcessMigrated 15=ProcessRecall 16=DeactivateTargetNode 17=ActivateSourceNode 25=GenerateFutureWorkItems	Each code uniquely identifies a separate event.
eventType	"Approve" or "Submit Purchase Order:99198"	Name of the event.
eventData	NULL	Additional information, but in most cases NULL.
createdTime	"2004-11-12 13:19:12.820" or "22.12.2004 16:48"	Timestamp (string format) of the creation of this activity.
responsible	"jim" or "__process"	Name of the user or system who was responsible for creating the event. If there is no User Group assigned to the node, the system executes the node and the name "__process" is taken.

Column Name	Value/Sample	Remarks
isHandled	0=notHandled 1=handled 2=ignored 3=Error 4=audit	State of the event.
producerType	0: arrow 7: node	Element responsible for creating the current event. Either this is an arrow or a node.
producerId	99130	If producerType = 0, the producerId denotes the arrow instance ID of the current event. Using the arrow instance ID, the source and target node instances of the event can be retrieved.
consumerType	0=ArrowType 3=ActivityType 7=ProcessType	The element type that the event is intended for.
consumerId	99116	If producerType = 7 the consumerId denotes the node instance ID of the current node. If producerType = 0 the consumerId denotes the node instance ID of the target node. The same node instance can be retrieved by the arrow instance.
processInstanceId	88892	Unique identifier of a process instance.
serverName	localhost	The name of the server for which these events are generated and handled.
applicationContainerId	2	Unique identifier of the application that contains the process instance

8.5 Retrieving History Information With SQL Statements

You can retrieve history information using SQL statements on the teamflowdb database.

- **To get an overview about all stored data in the History table:**

```
SELECT * FROM HISTORY
```

- **To retrieve all rows linked to a specific process instance:**

  `SELECT * FROM HISTORY WHERE PROCESSINSTANCEID=<PROCID>`

  Replace `<PROCID>` by the requested ID (number) of the process instance.

The following figure shows as an example all the entries in the `History` table, assigned to a closed process instance.

historyId	eventCode	eventType	eventData	createdTime	responsible	isHandled	producerType	producerId	consumerType	consumerId	processInstanceId
98925	0	__ProcessCreated	NULL	22.12.2004 12:30	__process	4	7	98898	7	98898	98898
98926	0	__Start	NULL	22.12.2004 12:30	iflow	1	7	98898	7	98898	98898
98927	1	__Activate	NULL	22.12.2004 12:30	iflow	1	7	98898	3	98899	98898
98928	1	Create Purchase Requisition	NULL	22.12.2004 12:30	__process	1	0	98910	3	98902	98898
99105	0	__CommitEdit	NULL	22.12.2004 12:42	__process	4	7	98898	7	98898	98898
99106	2	Submit Purchase Requisition:9	NULL	22.12.2004 12:42	jim	1	7	98898	3	98902	98898
99107	1	Submit Purchase Requisition	NULL	22.12.2004 12:42	__process	1	0	98916	3	98908	98898
99108	1	Manager Approval Required	NULL	22.12.2004 12:42	__process	1	0	98917	3	98901	98898
99161	2	Approve:99110	NULL	22.12.2004 16:48	manager1	1	7	98898	3	98901	98898
99162	1	Approve	NULL	22.12.2004 16:48	__process	1	0	98911	3	98904	98898
99173	2	Submit Purchase Order:99163	NULL	22.12.2004 16:49	accountant1	1	7	98898	3	98904	98898
99174	1	Submit Purchase Order	NULL	22.12.2004 16:49	__process	1	0	98912	3	98905	98898
99175	6	__Close:EXIT	NULL	22.12.2004 16:49	__process	1	3	98905	7	98898	98898

Figure 22: Entries in the History Table

Appendix A: Web Services Interfaces

The topics below describe the Interstage BPM Web Services interfaces. These interfaces allow you to perform various Interstage BPM functions with Web Services sent by applications external to Interstage BPM.

Web Service Information

The Web Service requests and responses are contained in SOAP 1.1 SOAP Envelopes and are described by WSDL 1.1 WSDLs. Any exceptions (errors) are returned in SOAP Faults.

The Service Registry, Process Definition and Process Instance Web Service interfaces support several Web Service operations defined by the Wf-XML 2.0 standard. Such Web Services are marked as Wf-XMl 2.0 Web Services in their respective descriptions.

A.1 WorkItem List (Task List) Web Service Interface

This interface allows you to get a list of workitems.

Service address: `<ServerBaseURL>/_wfxml/<tenant name>/service/wilist?appId=<application ID>`

Example: `http://example.com:49950/console/_wfxml/Default/service/wilist?appId=System`

Service WSDL address: `<ServerBaseURL>/_wfxml/<tenant name>/service/wilist?appId=<application ID>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/wilist?appId=System&wsdl=true`

All webservices use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM Server property ServerBaseURL. This property is set during setup and can be viewed and modified by a Super User using the Tenant Management Console.

Note: If Interstage BPM is configured to run in Non-SaaS mode, the appId=<application ID> request parameter can be omitted to get a list of workitems from all applications.

GetWorkitemList Web Service

Retrieves a filtered list of workitems. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the request:

- **`Filters`**: Optional element. If this element is present, the filters specified as its child elements are used to get a filtered list. If this element is not present, then the list of `MyActiveWorkItems` is retrieved.
- **`BaseFilter`**: The base filter to be used for filtering. Allowed values are one of the following: `MyActiveWorkItems, MyWorkItems, AllWorkItems.`
- **`FieldFilter`**: Zero or more occurrences allowed. Each one specifies a field filter.
- **`ListField`**: Specifies a list field to filter on. Allowed values are one of the following: `LISTFIELD_PLAN_NAME, LISTFIELD_PROCESSINSTANCE_NAME, LISTFIELD_WORKITEM_NAME,`

LISTFIELD_WORKITEM_STATE, LISTFIELD_WORKITEM_ASSIGNEE, LISTFIELD_WORKITEM_CREATEDTIME, LISTFIELD_WORKITEM_DUEDATE.

- **SqlOperator**: Specifies a SQL operator for the filter. Allowed values are one of the following: SQLOP_EQUALTO, SQLOP_GREATERTHAN, SQLOP_GREATERTHANOREQUALTO, SQLOP_LESSTHAN, SQLOP_LESSTHANOREQUALTO, SQLOP_NOTEQUALTO, SQLOP_IN, SQLOP_LIKE, SQLOP_NOTLIKE.
- **DataItemFilter**: Zero or more occurrences allowed. Each one specifies a data item (UDA) filter.
- **DataItemType**: Specifies a SQL operator for the filter. Allowed values are one of the following: TYPE_BIGDECIMAL, TYPE_BOOLEAN, TYPE_DATE, TYPE_FLOAT, TYPE_INTEGER, TYPE_LONG, TYPE_STRING, TYPE_XML, or if the type of the UDA is a custom UDA, then the type of the custom UDA (for example, Address#http://www.example.com/PO1 or Address@http://www.example.com/PO1)
- **Value**: Specifies the filter value.
  - For field filters, it should have the same format as the value parameter of the Model API com.fujitsu.iflow.model.workflow.WFObjectList.addFilter(int listField, java.lang.String sqlOperator, java.lang.String value)
  - For data item filters for non-custom UDAs, it should have the same format as the value parameter of the Model API com.fujitsu.iflow.model.workflow.WFObjectList.addFilter(java.lang.String udaName, java.lang.String udaType, java.lang.String sqlOperator, java.lang.String value)
  - For data item filters for custom UDAs, it should have the same format as the value parameter of the Model API com.fujitsu.iflow.model.workflow.WFObjectList.addFilterWithXPath(java.lang.String udaName, java.lang.String xPath, java.lang.String sqlOperator, java.lang.String value)
- **XPath**: Optional element. For custom UDAs, specifies the Xpath to a leaf node of the UDA.

Description of important elements in the response:

- **Key**: The address of the workitem. See section **WorkItem (Task) Web Service Interface** for the Web Services available on this address.

HTTP GET Operation

HTTP GET on the service address is also supported. The list will be returned as an RSS feed rather than as a SOAP envelope. The list of MyActiveWorkItems is retrieved.

A.2 Process Instance List Web Services Interface

This interface allows you to get a list of process instances.

Service address: <ServerBaseURL>/_wfxml/<tenant name>/service/pilist?appId=<application ID>

Example: http://example.com:49950/console/_wfxml/Default/service/pilist?appId=System

Service WSDL address: <ServerBaseURL>/_wfxml/<tenant name>/service/pilist?appId=<application ID>&wsdl=true

Example: http://example.com:49950/console/_wfxml/Default/service/pilist?appId=System&wsdl=true

All webservices use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM server property ServerBaseURL. This property is set during setup and can be viewed and modified by a Super User using the Tenant Management Console.

Note: If Interstage BPM is configured to run in Non-SaaS mode, the appId=<application ID> request parameter can be omitted to get a list of process instances from all applications.

GetProcessInstanceList Web Service

Retrieves a filtered list of process instances. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the request:

- **`Filters`**: Optional element. If this element is present, the filters specified as its child elements are used to get a filtered list. If this element is not present, then the list of `MyActiveProcesses` is retrieved.
- **`BaseFilter`**: The base filter to be used for filtering. Allowed values are one of the following: `MyActiveProcesses, MyProcesses, AllProcesses`.
- **`FieldFilter`**: Zero or more occurrences allowed. Each one specifies a field filter.
- **`ListField`**: Specifies a list field to filter on. Allowed values are one of the following: `LISTFIELD_PLAN_NAME, LISTFIELD_PROCESSINSTANCE_NAME, LISTFIELD_PROCESSINSTANCE_CREATEDTIME, LISTFIELD_PROCESSINSTANCE_OWNER, LISTFIELD_PROCESSINSTANCE_DUEDATE`.
- **`SqlOperator`**: Specifies a SQL operator for the filter. Allowed values are one of the following: `SQLOP_EQUALTO, SQLOP_GREATERTHAN, SQLOP_GREATERTHANOREQUALTO, SQLOP_LESSTHAN, SQLOP_LESSTHANOREQUALTO, SQLOP_NOTEQUALTO, SQLOP_IN, SQLOP_LIKE, SQLOP_NOTLIKE`.
- **`DataItemFilter`**: Zero or more occurrences allowed. Each one specifies a data item (UDA) filter.
- **`DataItemType`**: Specifies a SQL operator for the filter. Allowed values are one of the following: `TYPE_BIGDECIMAL, TYPE_BOOLEAN, TYPE_DATE, TYPE_FLOAT, TYPE_INTEGER, TYPE_LONG, TYPE_STRING, TYPE_XML`, or if the type of the UDA is a custom UDA, then the type of the custom UDA (for example, `Address#http://www.example.com/PO1` or `Address@http://www.example.com/PO1`)
- **`Value`**: Specifies the filter value.
  - For field filters, it should have the same format as the value parameter of the Model API `com.fujitsu.iflow.model.workflow.WFObjectList.addFilter(int listField, java.lang.String sqlOperator, java.lang.String value)`
  - For data item filters for non-custom UDAs, it should have the same format as the value parameter of the Model API `com.fujitsu.iflow.model.workflow.WFObjectList.addFilter(java.lang.String udaName, java.lang.String udaType, java.lang.String sqlOperator, java.lang.String value)`
  - For data item filters for custom UDAs, it should have the same format as the value parameter of the Model API `com.fujitsu.iflow.model.workflow.WFObjectList.addFilterWithXPath(java.lang.String udaName, java.lang.String xPath, java.lang.String sqlOperator, java.lang.String value)`
- **`XPath`**: Optional element. For custom UDAs, specifies the Xpath to a leaf node of the UDA.

Description of important elements in the response:

- **`Key`**: The address of the process instance. See section **Process Instance Web Service Interface** for the Web Services available on this address.

HTTP GET Operation

HTTP GET on the service address is also supported. The list of `MyActiveProcesses` is retrieved.

A.3 Process Definition List Web Services Interface

This interface allows you to get a list of process definitions.

Service address: `<ServerBaseURL>/_wfxml/<tenant name>/service/pdlist?appId=<application ID>`

Example: `http://example.com:49950/console/_wfxml/Default/service/pdlist?appId=System`

Service WSDL address: `<ServerBaseURL>/_wfxml/<tenant name>/service/pdlist?appId=<application ID>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/pdlist?appId=System&wsdl=true`

All Web Services use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM Server property ServerBaseURL. This property is set during setup and can be viewed and modified by a Super User using the Tenant Management Console.

Note: If Interstage BPM is configured to run in Non-SaaS mode, the appId=<application ID> request parameter can be omitted to get a list of process definitions from all applications.

GetProcessDefinitionList Web Service

Retrieves a filtered list of process definitions. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the request:

- **`Filters`**: Optional element. If this element is present, the filters specified as its child elements are used to get a filtered list. If this element is not present, then the list of `MyPlans` is retrieved.
- **`BaseFilter`**: The base filter to be used for filtering. Allowed values are one of the following: `MyPlans, AllPlans`.
- **`FieldFilter`**: Zero or more occurrences allowed. Each one specifies a field filter.
- **`ListField`**: Specifies a list field to filter on. Allowed values are one of the following: `LISTFIELD_PLAN_NAME, LISTFIELD_PLAN_OWNER`.
- **`SqlOperator`**: Specifies a SQL operator for the filter. Allowed values are one of the following: `SQLOP_EQUALTO, SQLOP_GREATERTHAN, SQLOP_GREATERTHANOREQUALTO, SQLOP_LESSTHAN, SQLOP_LESSTHANOREQUALTO, SQLOP_NOTEQUALTO, SQLOP_IN, SQLOP_LIKE, SQLOP_NOTLIKE`.
- **`Value`**: Specifies the filter value. It should have the same format as the value parameter of the Model API `com.fujitsu.iflow.model.workflow.WFObjectList.addFilter(int listField, java.lang.String sqlOperator, java.lang.String value)`.

Description of important elements in the response:

- **`Key`**: The address of the process definition. See section **Process Definition Web Service Interface** for the Web Services available on this address.

HTTP GET operation

HTTP GET on the service address is also supported. The list of `MyPlans` is retrieved.

A.4 WorkItem (Task) Web Services Interface

This interface allows you to get the details of a workitem and update the workitem.

Service address: `<ServerBaseURL>/_wfxml/<tenant name>/service/w<workitem ID>?appId=<application ID>`

Example: `http://example.com:49950/console/_wfxml/Default/service/w28458?appId=System`

Service WSDL address: `<ServerBaseURL>/_wfxml/<tenant name>/service/w<workitem ID>?appId=<application ID>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/w28458?appId=System&wsdl=true`

All webservices use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM server property ServerBaseURL. This property can be viewed and modified by a Super User using the Tenant Management Console.

Note: The service address for each workitem is considered an opaque value. Web Service clients should not construct the service address from parts or parse it into parts. The service address is returned by the Web Service `GetWorkitemList`. It is also returned by the Model API `com.fujitsu.iflow.model.workflow.WorkItem.getWorkItemURI()`. The returned service address must be used as-is to make the Web Service requests described here.

GetProperties Web Service

Gets the properties of the workitem. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the response:

- **`Choices`**: This element contains zero or more `Choice` elements. It will have one `Choice` element for each choice available on the workitem.
- **`DataItems`**: This element contains zero or more `DataItem` elements. It will have one `DataItem` element for each data item (UDA) in the process instance to which this workitem belongs.
- **`DataItemType`**: The type of the data item.
- **`Value`**: The value of the non-custom UDAs. For custom UDAs this element is blank.
- **`ComplexDataItemValue`**: The value of custom UDAs. For non-custom UDAs, this element is absent.
- **`DefinitionKey`**, **`DefinitionId`**: The `URI` and `Id` of the process definition for this workitem. If this workitem belongs to a dynamic process instance, these elements will be blank.

UpdateWorkItem Web Service

You can use this Web Service to update the values of data items (UDAs) in the process instance to which this workitem belongs and/or make a choice on the workitem. If you want to do both, specify one `DataItemToUpdate` element for each data item that you want to update and specify the choice that you want to make as the value of the `Choice` element under the `MakeChoice` element. If you only want to update data items, don't include the `MakeChoice` element in the request. If you only want to make a choice then leave the `DataItemToUpdate` element empty. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the response:

- **`MakeChoice`**: Optional element. If the `MakeChoice` element is specified in the request, then the choice specified in the `Choice` element is made on the workitem.
- **`DataItemToUpdate`**: This element contains zero or more `DataItem` elements, one for each data item that you want to update.
- **`Value`**: Specifies the value of the data item for non-custom UDAs. For Date type data items the value should be in the Java long date format.
- **`ComplexDataItemValue`**: Specifies the value of the data item for custom UDAs.

HTTP GET Operation

HTTP GET on the service address is also supported. The result is the same as invoking the GetProperties Web Service.

A.5 Process Instance Web Services Interface

This interface allows you to get the details of a process instance and update it.

Service address: `<ServerBaseURL>/_wfxml/<tenant name>/service/p<process instance ID>?appId=<application ID>`

Note:	The form of the service address has changed in Version 11.0 compared to previous versions.

Example: `http://example.com:49950/console/_wfxml/Default/service/p2365?appId=System`

Service WSDL address: `<ServerBaseURL>/_wfxml/<tenant name>/service/p<process instance ID>?appId=<application ID>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/p2365?appId=System&wsdl=true`

All webservices use HTTP Basic Authentication.

Note:	The <ServerBaseURL> part of the addresses is the value of the Interstage BPM Server property ServerBaseURL. This property can be viewed and modified by a Super User using the Tenant Management Console.

Note:	The service address for each process instance is considered an opaque value. Web Service clients should not construct the service address from parts or parse it into parts. The service address is returned by the Web Service `GetProcessInstanceList`. It is also returned by the Model API `com.fujitsu.iflow.model.workflow.ProcessInstance.getProcessURI()`. The returned service address must be used as-is to make the Web Service requests described here.

GetDetails Web Service

Gets the details of the process instance. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the response:

- **`DefinitionKey, DefinitionId, DefinitionName`**: The URI, Id and Name of the process definition that this process instance was created from. For dynamic process instances, these elements will be blank.
- **`Attachments`**: This element contains zero or more `Attachment` elements; one for each attachment in the process instance.
- **`DataItems`**: This element contains zero or more `DataItem` elements, one for each data item (UDA) in the process instance.
- **`DataItemType`**: The type of the data item.
- **`Value`**: The value of the non-custom UDAs. For custom UDAs this element is blank.
- **`ComplexDataItemValue`**: The value of custom UDAs. For non-custom UDAs, this element is absent.

UpdateDataItems Web Service

Update the data items in the process instance. Refer to the WSDL for the exact structure of the request and response. If any DataItem specified in the request does not exist in the ProcessInstance, this web service adds that DataItem to the ProcessInstance in structural edit mode.

Description of important elements in the request:

- **`DataItemsToUpdate`**: This element contains zero or more `DataItemToUpdate` elements, one for each data item (UDA) which needs to be updated or added. If the data item already exists in the process instance, it will be updated with the specified value. If it does not exist, a structural edit will be done on the process instance, and that data item will be added with the specified name, type, and value.
- **`DataItemName`**: The ID or name of the DataItem to update. If a data item with this ID or name does not exist in the process instance, then a data item with the specified name, type and value is added to the process instance.
- **`Value`**: Specifies the value of the data item for non-custom UDAs. For Date type data items the value should be in the Java long date format.
- **`ComplexDataItemValue`**: Specifies the value of the DataItem for custom UDAs.

ChangeState Web Service

Changes the state of the process instance to the requested state. Refer to the WSDL for the exact structure of the request and response. The state will be changed only if the requested state is valid for the current state. This is a Wf-XML 2.0 Web Service.

Description of important elements in the request:

- **`State`**: Specifies the requested state. Allowed values are `open.running` (Running), `open.notrunning.suspended` (Suspended), `closed.abnormalCompleted.aborted` (Aborted). The state will be changed to Running only of the current state is Created or Suspended. Similarly, the state will be changed to Suspended only if the current state is Running. And it will be changed to Aborted only if the current state is Created, Running, or Suspended.

Subscribe Web Service

Adds an observer to the process instance. Currently, no notifications are sent to the observer. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Unsubscribe Web Service

Removes the observer if the process instance has the specified observer. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

GetProperties Web Service

Returns the properties of the process instance. This Web Service is deprecated in favor of GetDetails. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

SetProperties Web Service

Sets the properties of the process instance. This Web Service is deprecated in favor of UpdateDataItems. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

HTTP GET Operation

HTTP GET on the service address is also supported. The result is the same as invoking the GetProperties Web Service.

A.6 Process Definition Web Services Interface

This interface allows you to perform operations on a process definition.

Service address: `<ServerBaseURL>/_wfxml/<tenant name>/service/d<process definition name>/v<version>?appId=<application ID>`

Note: The form of the service address has changed in Version 11.0 compared to previous versions.

Example:
`http://example.com:49950/console/_wfxml/Default/service/dVacationRequest/v1.0?appId=System`

Service WSDL address: `<ServerBaseURL>/_wfxml/<tenant name>/service/d<process definition name>/v<version>?appId=<application ID>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/dVacationRequest/v1.0?appId=System&wsdl=true`

All webservices use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM Server property ServerBaseURL. This property can be viewed and modified by a Super User using the Tenant Management Console.

Note: The service address for each process instance is considered an opaque value. Web Service clients should not construct the service address from parts or parse it into parts. The service address is returned by the Web Service `GetProcessDefinitionList`. It is also returned by the Model API `com.fujitsu.iflow.model.workflow.Plan.getPlanURI()`. The returned service address must be used as-is to make the Web Service requests described here.

CreateInstance Web Service

Creates a process instance from this process definition. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the request:

- **`StartImmediately`**: The only value supported is `true`.
- **`ObserverKey`**: The URI of the observer. If this URI is specified, then the observer will be notified about the process instance completion. The notification will be via an ASAP Completed Web Service call.
- **`ContextData`**: This element specifies the initial values for the data items in the created process instance. The specified data items must exist in the process definition. The schema of this element is specified in the `ContextDataSchema` element of the GetProperties Web Service.

Description of important elements in the response:

- **`InstanceKey`**: The URI of the created process instance.

GetProperties Web Service

Get the properties of the process definition. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the response:

- **`Subject`**: The title of the process definition.
- **`ContextDataSchema`**: The XML schema for the `ContextData` element in the CreateInstance web service request.

GetDefinition Web Service

Returns the XPDL representation of the process definition. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the request:

- **`ProcessLanguage`**: The only supported value is `XPDL`.

Description of important elements in the response:

- **`GetDefinitionRs`**: This element contains the XPDL representation of the process definition.

SetDefinition Web Service

Updates the process definition according to the XPDL representation in the Web Service request. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the request:

- **`ProcessLanguage`**: The only supported value is `XPDL`.
- **`Definition`**: This element contains the XPDL that this process definition should be updated to.

Description of important elements in the response:

- **`SetDefinitionRs`**: This element returns the XPDL representation that was specified in the request.

ListInstances Web Service

Returns a list of process instances that were created from this process definition. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the response:

- **`ListInstancesRs`**: This element contains zero or more `Instance` elements; one for each process instance which was created from this process definition.

HTTP GET Operation

HTTP GET on the service address is also supported. The result is the same as invoking the GetProperties Web Service.

A.7 Schema Web Services Interface

Custom data type support allows you to import XML Schemas into an application, and define custom UDAs based on XSD types and/or Global Elements defined in those schemas. This interface allows you to get a list of all XML Schemas defined in an application.

Service Address:
`<ServerBaseURL>/_wfxml/<tenantName>/service/schemalist?appId=<applicationId>`

Example:
`http://example.com:49950/console/_wfxml/Default/service/schemaList?appId=System`

Service WSDL Address:
`<ServerBaseURL>/_wfxml/<tenantName>/service/schemalist?appId=<applicationId>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/schemaList?appId=System&wsdl=true`

All Web Services use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM Server property ServerBaseURL. This property is specified during setup and can be viewed and modified by a Super User using the Tenant Management Console.

GetSchemaList Web Service

Retrieves a list of schemas within an application. Refer to the WSDL for the exact structure of the request and response.

Description of important elements in the request:

- **`GetSchemaListRq`**: This element identifies this web service request as a Web Service request.

Description of important elements in the response:

- **`GetSchemaListRs`**: This element contains zero or more "SchemaInfo" elements, one for each XML schema in the application. The order of the contained SchemaInfo elements is not significant.
- **`SchemaInfo`**: This element contains one XML schema in the application. The child element of this element will be the "schema" element defined by XML schema.

HTTP GET operation

HTTP GET on the service address is also supported. The result is the same as invoking the `GetSchemaList` Web Service.

A.8 Service Registry Web Services Interface

The Service Registry is a container of process definitions. This interface allows you to get a list of process definitions in the registry, and to add process definitions to it.

Service address: `<ServerBaseURL>/_wfxml/<tenant name>/service/registry?appId=<application ID>`

Note: The form of the service address has changed in Version 11.0 compared to previous versions.

Example:
`http://example.com:49950/console/_wfxml/Default/service/registry?appId=System`

Service WSDL address: `<ServerBaseURL>/_wfxml/<tenant name>/service/registry?appId=<application ID>&wsdl=true`

Example:
`http://example.com:49950/console/_wfxml/Default/service/registry?appId=System&wsdl=true`

All webservices use HTTP Basic Authentication.

Note: The <ServerBaseURL> part of the addresses is the value of the Interstage BPM Server property ServerBaseURL. This property can be viewed and modified by a Super User using the Tenant Management Console.

Note: If Interstage BPM is configured to run in Non-SaaS mode, the appId=<application ID> request parameter can be omitted to get a list of workitems from all applications. In that case, `ListDefinitions` will return a list of process definitions from all applications, and `NewDefinition` will add a new process definition to the System application.

ListDefinitions Web Service

Returns a list of process definitions in this service registry. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the response:

- **`ListDefinitionsRs`**: This element contains zero or more `DefinitionInfo` elements, one for each process definition in this service registry.

NewDefinition Web Service

Adds a new process definition to this service registry. Refer to the WSDL for the exact structure of the request and response. This is a Wf-XML 2.0 Web Service.

Description of important elements in the request:

- **`ProcessLanguage`**: The only supported value is `XPDL`.
- **`Definition`**: This element contains the XPDL that this process definition should be updated to.

Description of important elements in the response:

- **`NewDefinitionRs`**: This element returns the XPDL representation that was specified in the request.

GetProperties Web Service

Returns the properties of this service registry. Refer to the WSDL for the exact structure of the request and response.

HTTP GET Operation

HTTP GET on the service address is also supported. The result is the same as invoking the GetProperties Web Service.

Appendix B: Using the Interstage Business Process Manager Samples

The samples in this manual demonstrate various ways to program Interstage BPM with the Model API. After setting up Interstage BPM Server, you find the following subdirectories in your `<engine directory>` directory (typically `C:\Fujitsu\InterstageBPM` on Windows and `/opt/FJSVibpm` on UNIX):

`/client/samples/examples/sources`

`/client/samples/examples/classes`

`/client/samples/examples/bin`

The contents of each subdirectory is described in detail below.

Note: The sample codes described in this guide are used just to explain how to use Model APIs. If required, you will need to add the required checks on the values of the parameters passed to the Model APIs.

Note: If you are using the sample codes in your application, the password parameter is printed in clear text. If you do not want the password parameter to be printed, remove the code for printing the password, from the sample code.

Note: The first parameter for the API `wfSession.logIn` is ServerName. This value is required only for backward compatibility and does not cause any functional impact. Therefore, dummy string 'Flow' is added to ServerName/HostName in the sample codes.

For example:

```
String server = sessionProps.getProperty("HostName") + "Flow";
session.logIn(server, userName, password);
```

Here, the HostName is loaded from `iFlowClient.properties`. If HostName property is not defined in `iFlowClient.properties`, 'nullFlow' string will be used.

In this scenario, if an error occurs, it is not caused by the 'nullFlow' string defined for the HostName property.

Note: To use sample programs in SaaS mode, set the appropriate value for the `TenantName` property in `iFlowClient.properties` available in the `<engine directory>/client/` directory.

B.1 /client/samples/examples/sources

This directory contains the Java source code of the examples used throughout this manual. To see how the examples work, read the samples' source code found in the `/client/samples/examples/sources` subdirectory of your `<engine directory>`. In the following sections you find a list of the available sample sources and information on where to find additional explanations in this manual.

B.1.1 Samples Related to Process Modeling and Execution

SimplePlan.java

This sample creates a simple Interstage BPM process definition.

Additional information can be found in section

Designing a Simple Process Definition on page 63

ComplexPlan.java

This sample creates a complex Interstage BPM process definition and shows, for example, how to add User Defined Attributes and Voting User Task Nodes, use Java Actions, etc.

Additional information can be found in sections

Designing a Complex Process Definition on page 68

Using Java Actions on page 91

Advanced Filtering & Sorting API on page 111

Special User Defined Attribute Properties on page 122

Using Extended Attributes on page 134

Transaction Control on page 141

ProcessExecution.java

This sample shows how to start a new process instance and execute work items.

Additional information can be found in sections

Working with Process Instances on page 77

Transaction Control on page 141

Note: This sample contains an `INSTALL_ROOT` constant containing the path to your `<engine directory>`. By default, the path value is set to `C:\Fujitsu\InterstageBPM`. You need to adjust this value in case you copied engine directory in a different location or you use UNIX.

SampleJavaActions.java

This sample contains methods which are used for Java Actions in process definitions like the one created with the `ComplexPlan.java`, for example, adding an attachment and sending an email.

Additional information can be found in section *Using Java Actions* on page 91.

BatchDetailsRetrieval.java

This sample shows how to work with the bulk processing capabilities provided by the `WFDetailsList` interface.

Additional information can be found in section *Retrieving Information about Multiple Objects at a Time* on page 117.

TriggerTimerSample.java

This sample demonstrates the use of triggers, timers, and business calendars. It creates a process definition that is started by a trigger.

Additional information can be found in sections
Using Triggers on page 142
Using Timers on page 166

EventActivityNodeSample.java

This sample shows the typical steps involved in working with Message Receive Nodes.
Additional information can be found in section *Using Message Receive Nodes* on page 146.

RemoteSubProcess.java

This sample shows how to define and use a remote subprocess. It creates a process definition containing a Remote Sub-Process Node and another process definition which is used as remote subprocess.
Additional information can be found in section *Modeling Remote Subprocesses* on page 178.

AgentSimulator.java and AgentSimulator.xpdl

`AgentSimulator.xpdl` is a process definition that models a simple bank loan approval process. `AgentSimulator.java` is an example of an Agent Class, which simulates a loan decision maker. For more information, refer to the comments in the `AgentSimulator.java` source file and to section *Using Agents* on page 157.

HTTPAgentExample

The `HTTPAgentExample` subdirectory contains a pre-configured HTTP Example for implementing an HTTP Agent. The example agent demonstrates the complete functionality of an HTTP Agent. For detailed information, refer to the `HTTPAgentExampleInstructions.txt` file and to section *Using HTTP Agents* on page 166.

B.1.2 Samples Related to System Administration

Administration.java

This sample demonstrates various administrative activities, such as listing users, logging out users, resetting data from a user directory, publishing and archiving process definitions, etc.

Note: This sample works with existing process definitions and process instances. Make sure that all process definitions are valid and can be executed. In addition, be aware that this sample starts new instances and changes the status of existing instances.

Additional information can be found in chapter *Administration* on page 221.

SampleLocalUserManagement.java

This sample demonstrates local user and group administration, such as creating local users and groups, assigning users to groups, associating groups with other groups, etc.
Additional information can be found in section *User and Group Administration* on page 223.

B.1.3 Samples Related to Decision Tables

DecisionTableAPI.java

This sample demonstrates various activities related to Decision Tables management, such as creating a decision table, creating condition criteria, result locators, decisions, validating and evaluating a decision table, saving a decision table, and loading a decision table file. Additional information can be found in section *Decision Tables* on page 189.

B.2 /client/samples/examples/classes

The `/client/samples/examples/classes` subdirectory contains the compiled class files for the samples.

B.3 /client/samples/examples/bin

The `/client/samples/examples/bin` subdirectory contains two batch files:

- `StartSamples.bat`: Starts the execution of a sample file. Use the `StartSamples.sh` file on UNIX.

  You need to edit this batch file and set the directory where your application-server specific `jar` files are stored for the `APP_SERVER_JARS` variable BEFORE you run it.

  In addition, make sure that the `CLASSPATH` variable points to the correct directory where the following files are located:

  - `iFlow.jar`
  - Any application-server specific `jar` files
  - JDK

  Refer to the `StartSamples.bat` or `StartSamples.sh` file for a sample path for Weblogic and JBoss.

  When you execute the batch file, you need to specify the sample file name (see below), the name of a user with administrator rights and his/her password.

Note: To run samples, if you are using your own script instead of the `StartSamples` script, ensure you add all required jar files (as mentioned above) to your script. In addition, set the JAVA VM option `-DCONFIGDIR` to `<engine directory>/client/samples/configuration`. For example:

```
"%JAVA_HOME%\bin\java" -DCONFIGDIR=<engine
directory>/client/samples/configuration <sampleName> <userId>
<password> <applicationId>
```

- `BuildSamples.bat`: Builds the class files for the Java source files. Use the `BuildSamples.sh` file on UNIX.

  Again, make sure that the `APP_SERVER_JARS` and the `CLASSPATH` variables point to the correct directory.

Appendix C: Customizing Forms

Note: The information in this topic is applicable only for forms with the `*.qf` file extension. For information about customizing `*.jsp` forms, refer the Interstage BPM *QuickForm UI Widgets Reference* guide.

A QuickForm is a standard HTML page that enables the display of process data in Interstage BPM Clients and provides the ability to change that data. The Interstage BPM engine does not constrain you to any particular form technology. This means that if you have a favorite form support technology, you should be able to attach those forms to an activity, and your customization should be able to provide a client that displays the data through that form. In a QuickForm, you can use HTML5, CSS stylesheets, JavaScript, and even JavaScript UI frameworks like jQuery, Angular JS, GWT and YUI.

QuickForms are generated by Interstage BPM Studio and by the Process Designer that is part of the Interstage BPM Console. For information on generating QuickForms using these tools, refer to the *Interstage Business Process Manager Studio User's Guide*.

This appendix describes how to customize the generated QuickForms. QuickForms can be edited with any HTML editor because they are standard HTML files. You simply need to understand a little bit about the components that are placed into the QuickForm and the effect that they have on the final produced web page. This appendix does not cover how to use specific Web-authoring tools.

C.1 QuickForm Command Overview

When you look at a generated form, you will see that certain components that appear on the form are represented by form tags. These tags have been generated within the HTML at the points where UDA values are expected to appear. These component tags start with two open brace characters and are concluded with two close brace characters. The double-braces tell the Interstage BPM Console that what lies inside is a directive telling what value to place there. Brace characters are not special characters in HTML, they do not require any special handling, so your HTML editor should treat them as normal text that you can type and edit.

The following is a brief overview of the components that appear in a QuickForm; all of them are optional:

- **`{{WorkItemHeader}}`**: This token will generate the top of the form to look the same as the rest of the Console pages, including the navigation tabs and other buttons that appear. Use this when you want the form to look the same as the Console. Without this, you have free capability to make the page look any way you want to.
- **`{{ProcessPanel}}`**: The process panel displays details about the current state of the work item including who it is assigned to, who started the process, when it was assigned, when it is due to be completed, what the choices are, and buttons to accept, decline, or reassign the work item.
- **`{{AttachmentPanel}}`**: This panel lists the documents currently attached to the process, and provides buttons for adding, removing or editing them.
- **`{{control type="Style" uda=udaIdentifier}}`**: This is the main way to put the value of a UDA into the resulting web page. The value will be properly encoded for HTML, which means that the ampersands and angle brackets will be properly replaced with the corresponding HTML entities. This is used for displaying a UDA value, or for passing as an attribute of an HTML input form element.

  When creating the form using the Interstage BPM Studio or Console, you can choose one of the following styles for each UDA - depending on its data type:

Style	BigDecimal, Float, Integer, Long	Boolean	Date	String	Comment
Default	x	x	x	x	The HTML form element that will be generated depends on the data type of the selected UDA. For example, for a UDA of type Date, the Default form element will be a Calendar control. For a UDA of type STRING, the default will be a text area.
Hidden	x	x	x	x	Generates a hidden form element in the web page. This is useful if you are including some JavaScript in the page that will read from the hidden element, and update to the hidden element for posting back to the server.
Text Field	x	-	-	x	Generates a text input field. By default, this field has a width of 10 characters allowing for the entry of a maximum of 10 characters. You can change this default setting.
Read Only	x	x	x	x	Generates a read-only field. By default, this field has a width of 10 characters. You can change this default setting.
Radio Button	x	x	-	x	Generates a radio button for the UDA value. By default, there is one radio button for the selected UDA with a default a value. You can add additional value names and assign default values.
Combo Box	x	x	-	x	Generates a combo box for either choosing or entering a UDA value. By default, there is one entry available for the selected UDA with a default a value. You can add additional value names and assign default values. These can then later be chosen from a list of entries.
Calendar	-	-	x	-	Generates a calendar control for the web page. The default date selected on the caledar is the current date.
Password	-	-	-	x	Generated a text input field in for entering passwords, i.e. the input of a user cannot be read, dots will be displayed for each entered character. By default, this field has a width of 10 characters allowing for the entry of a maximum of 10 characters. You can change this default setting.

Style	BigDecimal, Float, Integer, Long	Boolean	Date	String	Comment
Text Area	-	-	-	x	Generates an area for text input. By default, this text area consists of four columns and two rows. You can change this default setting.
Checkbox	-	-	-	x	Generates a check box for the UDA value. By default, there is one check box for the selected UDA with a default a value. You can add additional value names and assign default values.
List	-	-	-	x	Generates a list for displaying and selecting UDA values. By default, there is one entry for the selected UDA with a default a value. You can add additional value names and assign default values. In addition, multiple selection from the list is enable, and the list will consist of two rows by default. You can change this setting as well.

The following is an example of how you would put the value of the UDA with the ID 'PurchaseAmount' as Text Field into a form:

```
{{control type="TEXT" uda="PurchaseAmount" value="" maxchar="" maxwidth=""}}
```

- **`{{DefaultUDAsPanel}}`**: Rarely used, this token will generate form input elements for every UDA in the process automatically at run time. You might use this if you had a form that was to be used for many different processes with different schema. This is unusual because normally you want to use the form purposefully to control which variables the user sees.

In general you can then lay out your HTML page in any way you wish. You may wish to change the appearance of the form so that there are multiple columns, add a border, add color, etc. You may include graphics at any place in the page. JavaScript may be used to provide interactions, including data validation. Java applets may also be used within the page. When the QuickForm is being used, it has full control of every aspect of the page.

C.2 QuickForm Token Command Parameter Details

`{{ProcessPanel params}}` This generates a hidden tag to carry the work item id, and another hidden tag to carry the process instance sequence id. It also generates a number of sections, and each section is controlled by the parameter. The parameter is a comma-delimited list of identifiers, each identifier turns on a single process panel section. For example, if you only want the due date and the choices to appear in the panel, then you might specify:

`{{ProcessPanel due,choices}}`

The following is a list of the identifiers that may be used, and their effect:

`activity` – specifies that the top of the panel should be included which displays the name of the activity.

`from` – displays the second line of the panel, the “From” line, which displays the owner of the process instance.

`to` – displays the assignee of the work item.

`date` – displays the date that the work item was created and first offered to the assignee.

`process` – displays the name and description of the process instance, including a link to the process instance display page.

`due` – displays the due date for the work item.

`desc` – displays the detailed description of the activity to be performed.

`form` – if the activity has at least one associated form, it will display the list of all forms as hyperlinks that would allow the user to view the activity using the specific form.

`choices` – displays the choices that the user has at this point.

`commands` – displays the commands section of the process panel which includes the start/stop timer, the Save button, the Accept button, the Decline button, and the Reassign button.

`{{Field udaIdentifier}}` The parameter specified is the identifier of a UDA that carries the value that is to be put here. The data is encoded to be proper for putting into HTML. This token can be used to put the value directly into a page for display to the user. It might easily be placed into a table cell, or any other such HTML construct.

It also may be used to set the initial value of a form input element. If you have a requirement for an input box of a particular size, you can specify the <INPUT> tag and use this token to generate the value attribute of the tag. For example, you might use the following in order to have a 10 character edit field for a UDA variable with the identifier 'ShoeSize':

```
<INPUT NAME=”uda_ShoeSize” value=”{{Field ShoeSize}}” maxchar="10" maxwidth="12">
```

The Console will replace the QuickForm token with the value for shoe size. Similarly, if you want to use a <TEXTAREA> tag in order to have a larger editing area for a UDA with the identifier 'Description' you might use something like:

```
<TEXTAREA NAME=”uda_Description”>{{Field Description}}</TEXTAREA>
```

Another technique to consider is composing a URL from a UDA variable. Imagine that you have an existing application that can display detailed information about an item for sale. Given the item number held in a UDA variable with the identifier 'ItemNo', you might include into the form a link like this:

```
<a href=”http://mysite/itemdetail.jsp?item={{Field ItemNo}}”>details</a>
```

Or if the UDA holds the entire URL then it could be directly placed into the `href` attribute of the <a> tag. Clearly, the field token is the most versatile QuickForm token for including values into the form.

If you use the **Hidden** style for the identifier of a UDA variable, this token generates the entire hidden <INPUT> tag so that UDA values will be available to JavaScript running in the form. The value must be carried in a hidden form element if the value is to be sent back to the server. JavaScript can get the value out of the form element, and can update the form element. This is useful when the data that is stored in the UDA is not directly in the form that is convenient for display. In this case, a small JavaScript could run, read the value, convert to a displayable form, and place that form in a visible form element. Then, upon any change, a different JavaScript will be invoked to convert the data from the visible representation back to the hidden field. A similar technique can be used in JavaScript (and Java) to use the value as a key to look up information elsewhere and display the result of the lookup on the page. For example, if a database holds detailed information on an item for sale, that database might be accessed directly by the form, at the time that the form is displayed, in order to include the latest information about the item directly on the form.

C.3 QuickForm Example: Vacation Request

An example process called `Vacation Request` is included with the Interstage BPM Console that demonstrates the kind of things that might be done with QuickForms. The `Vacation Request` sample files are part of the Template Library.

1. If you have not done so already, import the `TemplateLibrary.bar` application using the **Install Application** function of the Interstage BPM Console.
2. Search for the `Vacation Request` sample files and open the `EDF_Employee.htm` form in a text editor.
3. Search for the form and input tags and note how they correspond to the instructions above.

Appendix D: Supported JavaScript Functions

Interstage BPM provides a set of JavaScript functions that you can use with Java Actions, triggers, Flexible Exclusive Gateway Nodes, and Sequential Loop Nodes.

This appendix explains which functions you can use with these elements. Also, it provides a description of some of these functions.

Apart from functions listed in this appendix, you can use the JavaScript functionality that is defined in the ECMA Standard. For information about the ECMA Standard, refer to the document `ecma-262.pdf` included with the product media.

Note: The size that a method used in a JavaScript may have is limited by the Java Virtual Machine (JVM). Currently, the method byte code size is limited to 65535 bytes (64 KBytes). If you use a method with a larger size, the JVM will throw an error and you have to reduce the size of the method before executing the JavaScript again.

D.1 General JavaScript Functions

You can use the JavaScript functions explained in this section with Java Actions, triggers, Flexible Exclusive Gateway Nodes, and Sequential Loop Nodes.

Note: Since numeric values are treated as values of type Number, you can use only the following range of integers in JavaScript:

- Minimum: -9007199254740992
- Maximum : 9007199254740992

Note that the following minimum and maximum values of type Long are not supported:

- Minimum: Packages.java.lang.Long.MIN_VALUE
- Maximum: Packages.java.lang.Long.MAX_VALUE

new Packages.java.util.Date()

For a description, refer to the Javadoc that comes with the JDK.

Date DateAdd(Date or Number date, Number offset, String field)

Returns a JavaScript Date object containing the date and time that is the result of adding the offset to the date. `field` determines the unit of time measure for the offset. Valid `field` values are:

Field Value	Description
`"ss"`	Seconds
`"mi"`	Minutes
`"hh"`	Hours
`"dd"`	Days

Example:

```
var now = Packages.java.util.Date();
uda.Date = DateAdd (now, 1, "dd");
```

Note: The example works correctly only if the User Defined Attribute (UDA) `Date` is of type DATE.

Say `now` has the following value:

```
Tue Jul 01 2006 14:02:59 GMT-0800 (PST)
```

Then, `tomorrow ( DateAdd ( now, 1, "dd" ) )` has the following value:

```
Wed Jul 02 2006 14:02:59 GMT-0800 (PST)
```

If `date` is of type Number, its value is interpreted as milliseconds since January 1, 1970.

Boolean DateCompare(Date or Number date1, String operator, Date or Number date2)

Compares two Date values and returns `true` or `false` as the result of the comparison. Valid operators are:

Operator	Description
">"	Greater than
"<"	Less than
">="	Greater than or equal to
"<="	Less than or equal to
"=="	Equal to
"!="	Not equal to

Example:

```
var now = Packages.java.util.Date();
var tomorrow = DateAdd (now, 1, "dd");
if (DateCompare (now,"<", tomorrow)); // Executes because DataCompare
evaluates to true.
```

If `date1` and `date2` are of type Number, their values are interpreted as milliseconds since January 1, 1970.

Number DateDiff(Date or Number date1, Date or Number date2, String field)

Subtracts `date2` from `date1`. Returns the difference between these date/times in days, hours, minutes or seconds depending on the `field` value. Valid `field` values are:

Field Value	Description
"ss"	Seconds
"mi"	Minutes
"hh"	Hours
"dd"	Days

Example:

```
var now = Packages.java.util.Date();
var tomorrow = DateAdd (now, 1, "dd");
var diff = DateDiff (tomorrow, now, "dd");
//diff has a value of 1.
```

If `date1` and `date2` are of type Number, their values are interpreted as milliseconds since January 1, 1970. Example:

```
var date = DateDiff (20000000,10000000,"ss");
```

BigDecimal DecimalAdd(BigDecimal value1, BigDecimal value2)

Returns a JavaScript BigDecimal object that is the sum of the parameters specified. The result inherits the precision from the parameter with the most significant figures.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

Example:

```
var x = 39;
var z = DecimalAdd ("3.1416",x);
```

If `z` is a BigDecimal, its value is `42.1416`.

Boolean DecimalCompare(BigDecimal value1, String operator, BigDecimal value2)

Compares two BigDecimal values and returns `true` or `false` as the result of the comparison.

Valid operators are:

Operator	Description
">"	Greater than
"<"	Less than
">="	Greater than or equal to
"<="	Less than or equal to
"=="	Equal to
"!="	Not equal to

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

Example:

```
var smallDecimal = "1.11";
var largeDecimal = "22.22";
if (DecimalCompare (smallDecimal,"<", largeDecimal)); Executes because
DecimalCompare evaluates to true.
```

BigDecimal DecimalDivide(BigDecimal value1, BigDecimal value2, Number scale)

Divides `value1` by `value2` and returns the result of the division. `scale` determines the number of significant digits for rounding. Any Number can be used as a `scale` value. Default value is 2.

The rounding mode is always to round half up; it rounds towards the "nearest neighbor" unless both neighbors are equidistant. In that case, it rounds up.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

BigDecimal DecimalMultiply(BigDecimal value1, BigDecimal value2, Number scale)

Multiplies `value1` by `value2` and returns the result of the multiplication. `scale` determines the number of significant digits for rounding. Any number can be used for `scale`. Default value is 2.

The rounding mode is always to round half up; it rounds towards the "nearest neighbor" unless both neighbors are equidistant. In that case, it rounds up.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

BigDecimal DecimalSubtract(BigDecimal value1, BigDecimal value2)

Subtracts `value2` from `value1`. Returns the difference between these values as a BigDecimal.

boolean toBoolean(String or Number value)

Converts `value` to a JavaScript Boolean. The value can be either a String containing `"true"` or `"false"`, or a Number containing zero (for true) or non-zero (for false). Returns `true` or `false` depending on the parameter passed. If `value` cannot be converted into a Boolean, `false` is returned.

BigDecimal toDecimal(BigDecimal value, Number scale)

Converts `value` to a BigDecimal object. Returns the result of the conversion to the number of significant figures specified with `scale`. Any Number can be used for `scale`. Default value is 2.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

Packages.java.lang.Float.parseFloat

For a description, refer to the Javadoc that comes with the JDK.

Packages.java.lang.Integer.parseInt

For a description, refer to the Javadoc that comes with the JDK.

Packages.java.lang.String.valueOf

For a description, refer to the Javadoc that comes with the JDK.

D.2 JavaScript Functions Supported with Java Actions

With Java Actions, you can use the functions explained in section *General JavaScript Functions* on page 270 and the functions listed below.

The functions listed below use the Server Enactment Context API `com.fujitsu.iflow.server.intf.ServerEnactmentContext` to provide access to workflow information. For a description, refer to the *API Javadoc*.

```
void sec.addAttachment(String attachmentName, String attachmentPath)
void sec.addProcessXMLAttributeSubstructure(String udaName, String xPath, String value)
void sec.addProcessXMLAttributeSubstructureByIdentifier(String identifier, String xPath, String value)
void sec.deleteAttachment(String attachmentName)
void sec.deleteProcessXMLAttributeSubStructure(String udaName, String xPath)
void sec.deleteProcessXMLAttributeSubStructureByIdentifier(String identifier, String xPath)
void sec.escalateActivity(String assignees)
String sec.getActivityActor(String activityName)
Array sec.getActivityAssignees()
String sec.getActivityName()
String sec.getActor()
Array sec.getAllAttachmentNames()
Array sec.getAllAttributeNames()
String sec.getAttachment(String attachmentName)
Number sec.getCurrentActivityId()
Number sec.getCurrentProcessId()
Array sec.getGroupMembers(String groupName)
String sec.getProcessAttribute(String attName)
String sec.getProcessAttributeByIdentifier(String identifier)
String sec.getProcessAttributeStringType(String udaName)
String sec.getProcessDefinitionId()
String sec.getProcessDefinitionName()
String sec.getProcessDescription()
String sec.getProcessInitiator()
String sec.getProcessName()
Array sec.getProcessOwners()
Number sec.getProcessPriority()
Number sec.getActivityPriority()
String sec.getProcessTitle()
String sec.getProcessXMLAttributeElementValue(String udaName, String xPath)
String sec.joinString(Array)
```

```
Array sec.resolveRelationship(String relationship, String sourceValue)
void sec.sendEmail(String to, String from, String cc, String bcc, String subject,
String body, String mimeType)
void sec.setActivityAssignees(Array assignees)
void sec.setOwners(Array users)
void sec.setProcessAttribute(String name, String value)
void sec.setProcessAttributeByIdentifier(String identifier, String value)
void sec.setProcessDescription(String description)
void sec.setProcessName(String name)
void sec.setProcessOwners(Array users)
void sec.setProcessPriority(Number priority)
void sec.setActivityPriority(Number priority)
void sec.setProcessTitle(String title)
void sec.setProcessXMLAttributeElementValue(String udaName, String xPath, String
value)
void sec.setProcessXMLAttributeElementValueByIdentifier(String identifier, String
xPath, String value)
void sec.setProcessXMLAttributeSubstructure(String udaName, String xPath, String
value)
void sec.setProcessXMLAttributeSubstructureByIdentifier(String identifier, String
xPath, String value)
void sec.validateProcessXMLAttributeValue(String udaName)
void sec.validateProcessXMLAttributeValueByIdentifier(String identifier)
Array sec.splitString(String commaSeparatedList)
```

Manipulating User Defined Attributes (UDAs) with JavaScript

You can use UDAs that have been added to the process definition in your JavaScripts. Use the following syntax:

```
uda.<udaIdentifier>.
```

Note that you must use the identifier and not the name of the UDA, as multibyte characters are not allowed for variable names in JavaScript.

The following example creates a variable and initializes it to the value of a UDA:

```
var someVariable = uda.Price;
```

The following example shows how to assign the value of a variable to a UDA:

```
var lastName = "Jones";
uda.udaIdentifier = lastName;
```

Note: UDAs having the identifier `get` or `set` or `Function` must not be accessed using the syntax `uda.<udaIdentifier>`. The system behaviour is undefined in that case.

As a workaround, do one of the following:

- Use `uda.get("<udaName>").`
- Assign a different identifier to UDAs to be used in JavaScript, e.g. `uda.myget` where `myget` is the identifier of a UDA having the name `get`.

The methods `uda.get` and `uda.set` allow you to access UDAs by their names:

- `uda.get` returns the value of the specified UDA:

```
var value = uda.get("<udaName>");
```

- `uda.set` sets the value of the UDA to the specified value:

```
uda.set("<udaName>", "<udaValue>");
uda.set("<udaName>", variable name);
```

When assigning a JavaScript return value to a UDA, make sure that their data type matches. Otherwise, the assignment fails.

Note: If the assignment of a value to a target UDA fails due to conversion or any other kind of errors, error details are logged in `Trace.log`. The target UDA is not updated and holds its earlier value.

Interstage BPM maps UDA data types to the following Java data types:

- UDAs of type BIGDECIMAL are mapped to `Packages.java.math.BigDecimal` objects.
- UDAs of type DATE are mapped to `Packages.java.util.Date` objects.

For details about the Java objects, refer to the Javadoc that comes with the JDK.

Manipulating Custom UDAs with JavaScript

Custom UDAs can be manipulated using the JavaScript dot notation. Consider the following custom UDA with Id `MyPurchase`.

```
<customerName>Helmut Baer</customerName>
<dueDate>2010-12-12</dueDate>
<shipTo>
   <street number="315">Elm Street</street>
   <city>Oakhurst</city>
   <state>AZ</state>
   <zip>87654</zip>
</shipTo>
<billTo>
   <street number="315">Elm Street</street>
   <city>Oakhurst</city>
   <state>AZ</state>
   <zip>87654</zip>
</billTo>
<items>
   <item>
      <partNo>192D334</partNo>
      <quantity>15</quantity>
      <price>100.00</price>
   </item>
```

```
    <item>
       <partNo>292B334</partNo>
       <quantity>1</quantity>
       <price>800.00</price>
    </item>
</items>
<totalCost>1550.00</totalCost>
```

You can perform the following operations:

- Retrieve a certain element's value. Example:

```
var expectedPart = uda.myPurchase.items.item[1].partNo;
```

  will assign `192D334` to `expectedPart`.

- Set a certain value to an element. Example:

```
uda.myPurchase.items.item[1].partNo = "123DX345";
```

  will assign `123DX345` to the first item.

- Retrieve an attribute value of a certain element. Example:

```
var streetNumber = uda.myPurchase.billTo.street.getValue(\"@number\");
```

  will assign `315` to the `streetNumber` variable. The attribute should be the last element in the path expression.

- Set an attribute value of a certain element. Example:

```
uda.myPurchase.billTo.street.setValue(\"@number\", \"415\");
```

  will assign the value `415` to the `number` attribute of the `billTo` element.

- Assign one custom UDA element's value to another element of the same (or different) custom UDA. Example:

```
uda.myPurchase.items.item[1].partNo =
uda.myPurchase.items.item[2].partNo;
```

- Copy a custom UDA to another custom UDA as a part. For example, consider a custom UDA named `myAddress` with the following structure:

```
<street number="315">xyz Street</street>
<city>Oakhurst</city>
<state>AZ</state>
<zip>87666</zip>
```

  The following code will copy the `myAddress` custom UDA to the `billTo` part of the `myPurchase` custom UDA.

```
var myAddressObj = uda.myAddress;
uda.myPurchase.billTo = myAddressObj;
```

- Assign the value of a non-custom UDA (such as type STRING) to an element of a custom UDA. Example:

```
uda.myPurchase.items.item[2].price = uda.itemPrice;
```

where `itemPrice` is a UDA of type STRING.

- Assign the value of a custom UDA element to a non-custom UDA (such as type STRING). Example:

```
uda.itemPrice = uda.myPurchase.items.item[1].price;
```

where `itemPrice` is a UDA of type STRING.

Note: All the above operations are supported for multi-byte-character element names as well. However, you need to use `XData` methods instead of directly using the multi-byte element name in the JavaScript expression. For example, consider the following expression that assigns the value of a non-custom UDA to the custom UDA element `price`.

```
uda.myPurchase.items.item[2].price = uda.itemPrice;
```

Instead of `price`, if the element name contained multi-byte characters (such as characters in Japanese), the value assignment needs to be done as follows:

```
var itemObj = uda.myPurchase.items.item[2];
var simplePriceObj = uda.itemPrice;
itemObj.setValue(\"<Element name in Japanese>\", simplePriceObj);
```

Note: Do not use short names or any namespace information in any path expressions.

D.3 JavaScript Functions Supported with Triggers

With triggers, you use JavaScript expressions to specify control conditions. Control conditions narrow down when the trigger fires.

You can use the functions explained in section *General JavaScript Functions* on page 270 and the function explained below.

String eventData.getXMLData(String xpath)

Returns the text value of the XML element specified in the `xpath` expression.

Example:

Consider the following XML fragment:

```
<Customer>
   <Data>
      <Name>John</Name>
   </Data>
</Customer>
```

The following statement assigns the text value `"John"` of the `<Name>` XML element to the `name` variable:

```
var name = eventData.getXMLData ( "/Customer/Data/Name/text()");
```

Appendix E: Enabling SSO Authentication Using the OpenID Provider Application

The SSO authentication process allows the user to log in once and gain access to multiple (SSO enabled) applications without having to login again in each application during that session.

The **OpenID Provider application** is built using the OpenID Authentication 2.0 protocol to authenticate the users present in Microsoft Active Directory servers and Oracle Directory Server Enterprise Edition. It is the central system where user authentication will be done for all web based applications.

You need to configure this application for authentication of users in a particular directory server.

To enable SSO authentication in other external applications, you can use OpenID Provider application by using OpenID Authentication 2.0 protocol.

- OpenID Provider Endpoint URL:

  `http[s]://<hostname>:<port>/<provider context root>/id/`

- OpenID format:

  `http[s]://<hostname>:<port>/<provider context root>/id/<User ID registered in Directory Server>`

If you set up this application using the Interstage BPM Setup Script, the value of `<provider context root>` is `openid`.

Note: All applications\tenants using the SSO authentication should use either IP address or hostname in the Provider Endpoint URL and OpenID format. If one application/tenant uses IP address and other uses hostname then error page will be displayed while accessing these applications. You should confirm with OpenID Provider application administrator, whether you have to use IP address or hostname in SSO enabled applications/tenants.

The external application will communicate to OpenID Provider application for sending the request for user authentication. If the user is pre-authenticated, then it will respond with the authenticated UserID. If not, then OpenID Provider login screen appears.

In case, OpenID Provider application is configured to use NTLM authentication process, then user will be automatically logged in to Interstage BPM Console on Windows using Windows login Credentials. Here, no login page will be displayed to user as automatic login happens.

For more information about setup of this application, refer to the *Interstage BPM Server and Console Installation Guide*.

Appendix F: Troubleshooting

F.1 Log File Information

For details about Log Files, refer the **Configuring Log Files** topic in the *Interstage BPM Server Administration Guide*.

F.2 Resolving Specific Error Situations

F.2.1 Interstage BPM Server Fails to Start

Check the `Error.log` file in the `<engine directory>/server/instance/default/logs` directory.

Look for	What to do
`DbService : setConnection: Connection to database server failed. Is the database server running and reachable through the network? {ORA-01089: immediate shutdown in progress - no operations are permitted.`	Check if the database is running. Also check that you can access the database from the machine where the Interstage BPM Server is setup in case the database is running on a different machine. You can use `telnet <Database Server Hostname> <Port>` from the server host machine to check that the connection to the database host/port can be established.

`LdapBroker : getContext: Could not create the directory services. {[LDAP: error code 49 - Invalid Credentials]}`
`LdapBroker : getGroupMembersByDN: Could not retrieve the user groups. {Could not create the directory services. {[LDAP: error code 49 - Invalid Credentials]}}`
`LdapBroker : Could not retrieve the user groups. {Could not create the directory services. {[LDAP: error code 49 - Invalid Credentials]}}`
	Ensure that the user name/password as specified in the `LDAPAccessUserID / LDAPAccessUserPassword` parameters of the Interstage BPM Server are correct and you can login to your Directory Server using the above user name/password.

`getContext: Could not create the directory services.`
`LdapBroker : getGroupMembersByDN: Could not retrieve the user groups. {Could not create the directory services.`
`LdapBroker : Could not retrieve the user groups. {Could not create the directory services.`
	Ensure that the LDAP Server is running on the port as specified in the `LDAPServer` parameter of the Interstage BPM Server. You can use `telnet ldapServerHostName port` from the server host machine to check that the connection to the host/port can be established.

F.2.2 Error in Trace.log

Check the `Trace.log` in the `<engine directory>/server/instance/default/logs` directory.

Look for	What to do
`getGroupMembersByDN: Could not retrieve the user groups. {[LDAP: error code 32 - No Such Object]}`	Possible cause of this error: A user has been deleted from the Directory Server (LDAP Server) but the reference of it is still there in one of the groups.

F.2.3 Timeout During JavaScript Execution

When executing large JavaScripts, the value for transaction timeout may be insufficient for the following application server:

- WebLogic Application Server

Due to this setting, script execution may fail with a "transaction timeout".

Your application server administrator can increase the transaction timeout depending on your usage requirements, for example, to 200 seconds. This setting can be changed in the following location:

- WebLogic Application Server: `<jta>/<timeout-seconds>`tag in`<MW_HOME>/user_projects/domains/<Domain Name>/config/config.xml`file.

  For Example: `<jta><timeout-seconds>200</timeout-seconds></jta>`

F.2.4 Failure in Writing to an Oracle Database

When the updating of an Oracle database table fails, for example, when you try to archive a process instance, check the Oracle alert log file located in the `<Oracle Installation Dir>/admin/<DB instance name>/bdump` directory, for example:

```
C:\ProgramFiles\Oracle\admin\orcl\bdump\alert_orcl.log
```

The following error may be observed:

```
{Database add/create request failed. {ORA-08103: object no longer exists}}
```

This failure may be due to the fact that the Datafile size reached the file size limit on the hard disk of the database server.

The system administrator of the database server needs to increase the file size on the database server hard disk.

F.2.5 Access Permissions for Generic Java Action Execution

Generic Java Actions that try to copy or move files to remote machines in the network, may fail if the proper permissions are not setup on the remote machine. To resolve such problems ensure that the proper access permissions are setup on the remote machine.

F.3 Errors during Setup of the Interstage BPM Server

If you have trouble executing the setup script, you can check the cause of error by viewing `deployment.log`. The deployment log is created at the following location:

On Windows: `<engine directory>\server\deployment\logs\deployment.log`

On Unix: `<engine directory>/server/deployment/logs/deployment.log`

The following tables explain the possible causes of errors during server startup and the appropriate action(s) to take:

Setup of IBPM fails when trying to setup a new build

I	Cause	The build directory of the existing installed build was deleted, without un-installing the build or un-installation failed.
	Action (For Windows)	Manually delete the earlier build's registry-entry, as follows: 1. Go to **Start > Run**, type `regedit`, click **OK**. 2. In the **Registry Editor** screen, go to below path. • 32bit OS: **HKEY_LOCAL_MACHINE > SOFTWARE > Fujitsu > Install > Interstage BPM Server** • 64bit OS: **HKEY_LOCAL_MACHINE > Wow6432Node > SOFTWARE > Fujitsu > Install > Interstage BPM Server** 3. Delete the **Interstage BPM Server** registry entry under **Install** key. 4. Go to below path. • 32bit OS: **HKEY_LOCAL_MACHINE > SOFTWARE > Microsoft > Windows > CurrentVersion > Fujitsu > Interstage BPM Server** • 64bit OS: **HKEY_LOCAL_MACHINE > Wow6432Node > SOFTWARE > Microsoft > Windows > CurrentVersion > Fujitsu > Interstage BPM Server** 5. Delete the **Interstage BPM Server** registry entry under **Fujitsu** key. This will ensure complete deletion of the registry entry for Windows platform. 6. Go to below path. • 32bit OS: **HKEY_LOCAL_MACHINE > SOFTWARE > Microsoft > Windows > CurrentVersion > Uninstall > DFC70E37-68E7-469C-A253-9ED7BB3BCD23** • 64bit OS: **HKEY_LOCAL_MACHINE > Wow6432Node > SOFTWARE > Microsoft > Windows > CurrentVersion > Uninstall > DFC70E37-68E7-469C-A253-9ED7BB3BCD23** 7. Delete the **Interstage Business Process Manager xx.x** registry entry under **Uninstall** key. This will ensure complete deletion of the registry entry for Windows platform.
	Action (For Solaris)	Manually delete the Solaris package information, as follows: 1. Open the Command Prompt window and run the command `pkginfo -l FJSVibpm` from any location of the Solaris machine, to check if the Solaris package information still exists. 2. If the Solaris package information is displayed, then delete the package information using the following steps: a. Create a file named **ibpm.uninst** under the **/tmp** location. b. Run the command `pkgrm FJSVibpm` to delete the Solaris package. c. Run the command `pkginfo -l FJSVibpm` once more, to ensure that the Solaris package has been deleted successfully. If no information is displayed, then it confirms that the Solaris package has been completely deleted from the Solaris platform.

	Action (For Linux)	Manually delete the RPM package information, as follows: 1. Open the Command Prompt window and run the command `rpm -qi FJSVibpm` from any location of the Linux machine, to check if the RPM package still exists. 2. If the RPM package information is displayed, then delete the package information using the command `rpm -e FJSVibpm`. 3. Run the command `rpm -qi FJSVibpm` once more to ensure that the RPM package has been deleted successfully. If no information is displayed, then it confirms that the RPM package has been completely deleted from the Linux platform.

An error occurred during Interstage BPM database creation/update

I	Cause	One of the following values provided during setup were wrong: • Database Administrator user name • Database Administrator password
	Action	Restore the database from the backup of the database made before setting up Interstage BPM. Update the `setup.config` file and run the setup script again.
II	Cause	One of the following values provided during setup were wrong: • Host name of the database server • Database SID (database instance name) • Database port
	Action	Restore the database from the backup of the database made before setting up Interstage BPM. Update the `setup.config` file and run the setup script again.
III	Cause	Database server is not running.
	Action	Start the database server. Update the `setup.config` file and run the setup script again.

Errors caused by JDBC Connection

I	Cause	The SQLRecoverableException occurs. This is caused by Oacle Bug 6485149.
	Action	Please contact Oracle Technical Support team for further instructions.

Errors caused by JDK Version

I	Cause	The version of JDK specified during setup does not match the version required for this application server.

	Action	You need to update `JAVA_HOME` variable of system environment and run the setup script again. Please update `JAVA_HOME` with `setIBPMJava.bat/setIBPMJava.sh` that exists in `<engine directory>/server/deployment/bin` when the error happens when the script of Interstage BPM is executed after completing the setup.

Problems related to LDAP Server Access

I	Cause	Even if LDAP server is up, the error message `Unable to connect to specified directory server` appears. This happens when Super user or Tenant Administrator is not registered in LDAP server.
	Action	Please register the Super user and Tenant Administrator in the LDAP server before executing the setup script.

Errors Pertaining to Active Directory

I	Cause	The Active Directory Server is remote and Active Directory is not running.
	Action	Start the Active Directory Server, and then run the setup script again.
II	Cause	Interstage BPM cannot connect to the Active Directory Server because one of the following values provided during setup was wrong: • Active Directory Key • Active Directory Organizational Unit
	Action	Update the `setup.config` file and run the setup script again.

Errors Pertaining to Oracle Directory Server Enterprise Edition

I	Cause	Interstage BPM cannot connect to the LDAP Server because one of the following values provided during setup was wrong: • LDAP Key • LDAP Organizational Unit
	Action	Update the `setup.config` file and run the setup script again.

Errors Pertaining to Un-setup

For WebLogic Application Server:

I	Cause	Interstage BPM Un-setup failed.

	Action	If un-setup fails, follow these steps to remove the resources: 1. Login to WebLogic Admin Console and navigate to **Domain Structure** > **<Domain Name>**. 2. Check if the following resources exists and if exists then delete them. • Navigate to **Deployments** and delete `InterstageBPMServer` and `InterstageBPMConsole` applications, if exists. • Navigate to **Environment**> **Servers** and delete `IBPMServer`, if exists. • Navigate to **Services** > **Messaging**> **JMS Modules** and delete `InterstageBPM-JMSSystemResource.IBPMServer`, if exists. • Navigate to **Services** > **Messaging**> **JMS Servers** and delete `InterstageBPMJMSServer.IBPMServer`, if exists. • Navigate to **Services** > **Data Sources** and delete `iflow.iFlowDS.IBPMServer`, if exists. • Navigate to **Services** > **XML Registries** and delete `ibpm_crimson_registry`, `ibpm_sun_xerces_registry`, and `ibpm_xerces_registry` if exists. 3. Delete the following directories manually from the file system: • `InterstageBPMServer` directory located at `<Weblogic Home>/Middleware/user_projects/domains/<Domain Name>/servers/AdminServer/upload/` directory. • `IBPMServer` directory at `<Weblogic Home>/Middleware/user_projects/domains/<Domain Name>/servers/` directory.

F.4 Errors Related to Interstage BPM Database Creation/Update

Errors Pertaining to the Database

I	Cause	The Database Server is not running.
	Action	Start the Database Server, and then start the Interstage BPM Server.

Errors Pertaining to a Hostname Change

I	Cause	You changed the hostname of the computer where Interstage BPM Server has been setup. As the hostname occurs in the names and values of various configuration parameters of the Interstage BPM Server, the server cannot access its configuration settings.

	Action	In the `IBPMProperties` table of the Interstage BPM database, make the following changes: • In the `PROPERTYKEY` column, update any parameter names that have the hostname in the suffix. These parameters have the format `<PARAMETER_NAME>.<HOSTNAME>` or `<PARAMETER_NAME>.<HOSTNAME>.<SERVERNAME>`. • In the `PROPERTYVALUE` column, update any parameter values containing the hostname. To update the `IBPMProperties` table, use the appropriate database commands or a database client software.

F.5 Contacting Your Local Fujitsu Support Organization

If you are unable to troubleshoot your problem:

1. Replicate the actions that caused the error.
2. Contact your local Fujitsu Support organization and provide the following information:

General Information

- Operating System
- Directory Service (type and version)
- Database server (type and version)
- JDK version
- Application server (type and version)
- Interstage BPM edition, version and build number
- Major problem area
- Priority of the issue
- Environment in which the problem occurs

Configuration Information

- The configuration file that you exported from the Interstage BPM Server

Log Files

- All log files from `<engine directory>/server/instance/default/logs`
- Setup log file from `<engine directory>/server/deployment/logs/deployment.log`
- When using WebLogic: All log files from `<MW_HOME>/user_projects/domain/<Your Domain>/servers/AdminServer/logs`

OS System Logs

- The Windows event log that you obtain using the Windows Event Viewer
- Linux system logs stored in `/var/log/messages`
- Solaris system logs stored in `/var/adm/messages`

Problem Description

- Description of the steps you performed before the problem occurred
- Frequency with which the problem occurs

Problem Details

- The application program and its source code that caused the error

- The XPDL file of the process definition that caused the error
- Information about Java Actions, Timers and Agents defined in the process definition
- Screenshot of the process instance history if the process instance goes into error state
- Stack trace if any exception is displayed

  You can obtain the stack trace by clicking **Details** on the error page displayed in the Interstage BPM Console.
- Screenshot of the exception wherever it is displayed
- Screenshot of the process instance (graphical view) if the process instance goes into error state or into an unexpected state
- Calendar files (`*.cal`) if timers are used
- The `agentsConfig.xml` file if agents are used

Glossary

ACID properties	A transaction is a set of actions that obeys the four so-called ACID properties: atomic, consistent, isolated, and durable.
Activity	The description of a task, logical step, or work to be performed in a process. An activity is represented by a work item.
Activity Time	The time it takes to perform a particular activity.
Agents	Components that asynchronously access systems external to Interstage BPM.
Annotation	An addition to a process definition allowing for adding explanatory comments to the process definition.
API	Application Programming Interface. The interfaces and classes that programmers may use in their own customized applications to access the server.
Application Variable	Dynamic variables that can be defined at application project level to share them across all the processes within a specific application project. This saves effort for creating separate variables for each process.
ASAP	Asynchronous Service Access Protocol. ASAP is a communication protocol based on SOAP and is used to start, manage, and monitor long running services.
Arrow	A connector between nodes. Arrows guide the process flow from one node to another.
Assignee	The person(s) assigned to perform an activity.
Attachment	A document file generated by any application, which has been associated with a process.
BPR	Business Process Reengineering. The field of study which concentrates on how work may be redefined in terms of processes.
Business Calendar	A calendar that specifies working days and times.
Business Process	See *Process*.
Business Rule Task Node	A node that represents calling Business Rule Engine.
Call Activity Node	A node that represents a complex task. The details of that task are defined in another process definition.
Chained-Process Node	A node representing a complex task that can be accomplished independently from the tasks defined in the parent process definition.
Compensation Action	A second level Java Action that can be defined as compensation for a first level Java Action, e.g. for cleaning up the system and ensuring a consistent state of external systems not participating in a transaction.
Database Action	A Java Action allowing for the interaction with external database.
DB Node	A node that accesses an external database using JDBC.

Directory Service (DS)	Repository for the entire network's authentication and configuration data. Provides access to services, file servers, databases, and other applications. User and application access to the repository is controlled.
Document Management System (DMS)	The system integrated with Interstage BPM which is used to store attachments, forms, etc. The DMS Adapter is the communication link between the DMS and Interstage BPM.
Due Date	Specifies when an activity is due to be completed once it has become active. A due date also specifies what will happen when it is reached and the activity has not been completed.
EJB	Enterprise JavaBeans.
Email Node	A node that sends out predefined emails.
End Node	A node that identifies the end of a process branch and completes the process. A process definition has at least one End Node.
Error Action	A Java Action that is used to handle specific errors on process definition level, on remote subprocess level, and on first level Java Action.
Flexible Exclusive Gateway Node	A Simple Exclusive Gateway Node where the condition is specified as a JavaScript expression.
Form	An HTML or XML file which may be associated with an activity, process instance, or process definition. Forms can be created using Interstage BPM; their appearance can be modified using any XML or HTML editing tool.
Framework Adapter	The Framework Adapter, also called DD Adapter, connects the Directory Service and the Document Management System Adapters. The "DD" is short for "document" and "directory". It handles authentication of the user and manages a consistent user authentication to the Directory Services and Document Management System.
Future Work Item	Work Items that the users may be assigned in future so that the tasks can be planned in advance.
Groups	Collections of users. Groups can be defined in Interstage BPM's local group store, in a Directory Service or in both systems.
Groupware	A type of software, which facilitates collaboration.
GUI	Graphical User Interface.
Initiator	The person who starts a new process instance.
Integration Action	A Java Action allowing for accessing external functions from within a process definition.
Interstage BPM Console	A web-based user interface which allows a user to create process instances, process definitions and access and respond to work items. It is also used by Interstage BPM Super Users to administrate Interstage BPM.
Interstage BPM Form	A type of form native to Interstage BPM
Iterator Node	A node that generates multiple node instances upon specifying the iterator count.

Java Action	A part of a process definition that tells how to call a Java method. Every Java Action is an instance of a particular Java Action Type.
Java Action Set	A collection of Java Actions which are all executed at a particular time within a process definition
Java Action Type	This specifies what exact operation that a particular Java Action will call. A Java Action Type corresponds to a particular static Java method. You can extend the server by adding new custom Java Action Types, and that means that you are adding your custom Java methods to be called by the server, usually as part of an application.
LDAP	Lightweight Directory Access Protocol.
Message Receive Node	A node that is designed to wait for accepting an external message. It's the same action as a Receive Task Node.
Node	A graphical representation of a step in a process. Interstage BPM supports different node types, e.g. User Task Nodes, Parallel Join Gateway Notes, Call Activity Nodes, Simple Exclusive Gateway Nodes.
No-Operation Action	A built-in Java Action that specifies no operation.
Notification Action	A Java Action allowing for notifying users on events related to process execution. Users can be notified by email, for example, that a process or a single activity has been started.
OnAbort Action Set	Java Actions in this set will be executed before a process instance is aborted.
OnResume Action Set	Java Actions in this set will be executed before a process instance is resumed.
OnSuspend Action Set	Java Actions in this set will executed before a process instance is suspended.
Owner	See *Process Definition Owner* and *Process Instance Owner*.
Parallel Join Gateway Node	A node that synchronizes flow from multiple branches in a process.
Parallel Split Gateway Node	A node that splits process flow into multiple parallel branches.
Participant	A person involved in a process.
Process	A sequence of steps that are performed to reach a business goal. Processes are modeled in process definitions.
Process Definition	The representation of a business process in a form that supports automated manipulation. A process definition defines the behavior and properties of the process instances created from it including the flow of control within the process.
Process Definition Owner	The person who created (or last edited) the process definition.
Process Initiator	See *Initiator*.

Process Instance	Represents a single enactment of a specific process definition. The structure of a process instance is exactly the same as the structure of the process definition on which it is based.
Process Instance Due Date	Represents the due date for a process instance.
Process Instance Owner	By default, the owner of a process instance is the owner of the process definition from which the process instance was created.
Process Participant	See *Participant*.
Project	A container for process definitions, forms, simulation scenarios, attachments, etc. On file system level, a project corresponds to a folder.
QuickForm	A structured, field-based HTML file created using the Interstage BPM Studio.
Receive Task Node	A node that is designed to wait for accepting an external message. It's the same action as a Message Receive Node.
Remote Sub-Process Node	A node representing a subprocess that runs on a remote workflow server.
Role	A relationship between a user or group and an object or context. This term is problematic since it has been used inconsistently in the past in the marketplace. A User Task node has an "Assignee". A group called "Window Washers" may be designated as the "Assignee" of the node. A user "Fred" may be in the group "Window Washers". This allows Fred to play the Assignee role (to be the assignee) when the node is active. The "Assignee Role" shows the relationship between Fred and the User Task node. Fred may also be a Process Owner for the process. "Process Owner" is another role, and users can play more than one role at a time. Remember that a role is a relationship between a person and a thing. For example, one person may drive a car, and "driver" is a role that person is playing with respect to the car. The things you find in a directory server are "groups". For example, a directory might have a group called "Drivers" and that is a group of people who might be allowed to drive, or who have the skill to drive, but it usually does not tell you who is currently driving. It is very rare for a directory to actually specify what person is currently playing a role.
Rule	Method used to determine choices on activities for which the rules are defined.
SaaS Mode	The 'Software as a Service' mode of Interstage BPM; deploying and using Interstage BPM in this mode allows you to create multiple tenants, and lease out Interstage BPM to these tenant organizations, who will use it as a service.
Script Task Node	A node that represents calling a script.
Server	In the Interstage BPM context, the component of the workflow management system installed on the computer to provides the run-time environment for a process.
Server Action	A Java Action allowing for interacting with the Interstage BPM Server.

Service Task Node	A node that represents using a service (an automated application).
Simple Exclusive Gateway Node	A node that directs the process flow along one of several branches, depending on specified criteria. Also known as an XOR Gateway.
Simulation Scenario	A defined setup for simulating the execution of a process definition on your local computer.
SOAP	Simple Object Access Protocol. SOAP is a standard communication protocol that allows one application to send an XML message to another application. It is used, for example, to access Web Services.
SQL	Structured Query Language.
Start Node	A node that identifies the beginning of a process. Every process definition has one and only one Start Node.
SWAP	Simple Workflow Access Protocol. SWAP passes XML messages over HTTP between workflow servers.
Swimlane	Visual grouping of activities.
Task	Same as an activity; a step in the process that requires a human response and usually a decision to be made.
Timer	Expires after a specified interval or at a specified time and date. Timers trigger certain actions when they expire.
Timer Node	A node that suspends process execution for a certain amount of time.
User Defined Attribute (UDA)	Data that process participants need to access, modify, or add, such as customer data, order numbers etc. User Defined Attributes are specified in the process definition at design time, and their values can be manipulated at run time.
User Profile	User-specific configuration information. This includes information such as whether a users wishes to receive email notifications, email address, and default directory, etc.
User Task Node	A graphical representation of a node.
Voting Rule	Rule defined on a Voting User Task Node to determine the outcome of the vote.
Voting User Task Node	A node that allows users to work on an activity in collaboration with one another.
Web Service Node	A node that retrieves data from a Web Service and makes it available for further processing.
Workflow	The sequence of activities within a business process.
Workflow Application	A bundle of artifacts such as process definitions, forms, simulation scenarios, attachments, etc. that are packaged together to form a process solution. Interstage BPM allows for the creation of Workflow Application projects having a predefined structure. Such applications can be setup on an Interstage BPM server in a single action.
Workflow Server	Same as a BPM Server: A server that provides the run-time environment for process instances.

Work Item	An assignment of a particular task to a particular user. Appears in a worklist.
Worklist	A list of workitems saying what tasks are currently assigned to which users.
WSDL	Web Services Description Language. WSDL is an XML-based language that describes the Web Services an organization offers. It also describes how to access the Web Services.
XML Action	A Java Action that performs specific operations on UDAs of type XML, for example adding XML substructures, setting text or attribute values in XML, or extracting UDA values from XML data.
XPath	XML Path Language. XPath is a language for finding information in an XML document. It is used to navigate through elements and attributes.
XPDL	XML Process Definition Language

Index

W

X