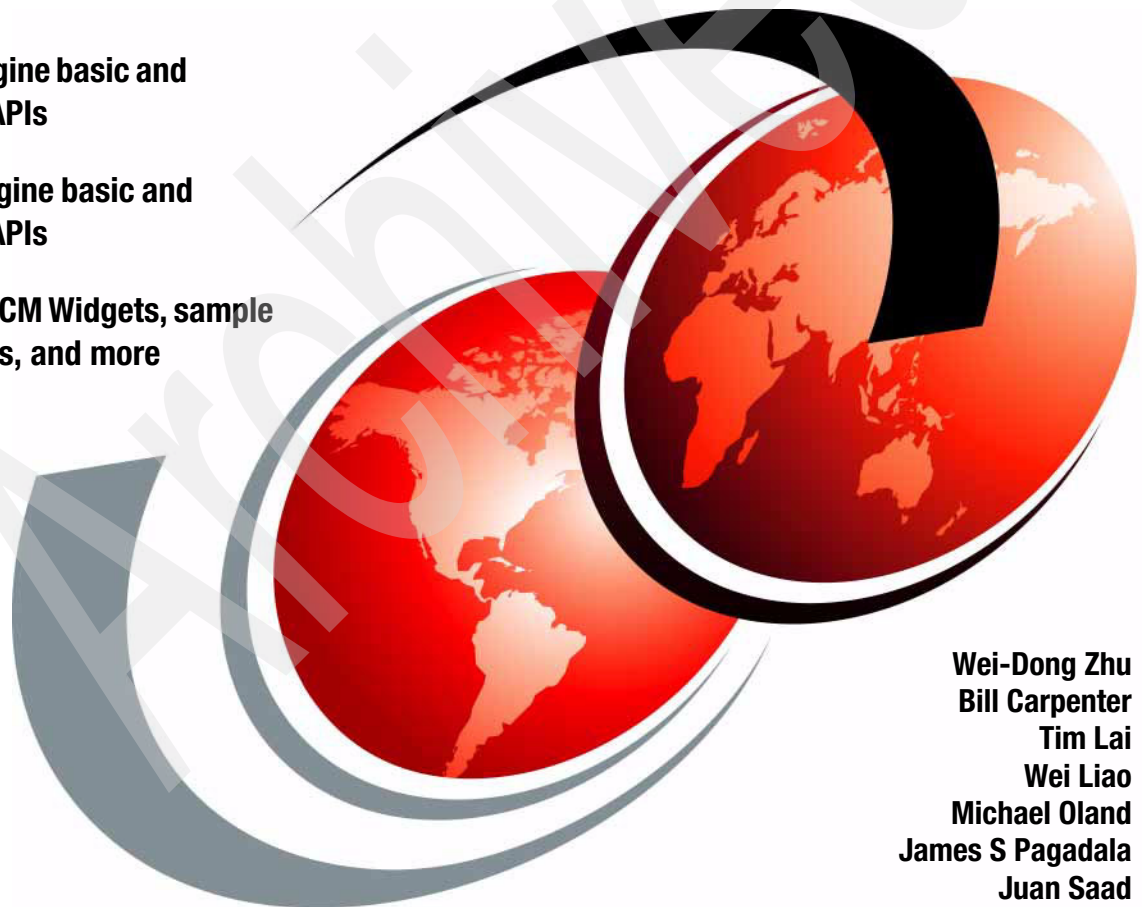# Developing Applications with IBM FileNet P8 APIs

- **Content Engine basic and advanced APIs**

- **Process Engine basic and advanced APIs**

- **REST API, ECM Widgets, sample applications, and more**

Wei-Dong Zhu
Bill Carpenter
Tim Lai
Wei Liao
Michael Oland
James S Pagadala
Juan Saad

IBM

Redbooks

IBM International Technical Support Organization

Developing Applications with IBM FileNet P8 APIs

December 2009

Note: Before using this information and the product it supports, read the information in "Notices" on page xiii.

First Edition (December 2009)

This edition applies to Version 4, Release 5, of IBM FileNet Content Manager (product number 5724-R81) and Version 4, Release 5, of IBM FileNet Business Process Manager (product number 5724-R76)

Contents

Figures

Notices

This information was developed for products and services offered in the U.S.A.

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

FileNet®	Redbooks®	WebSphere®
IBM®	Redpaper™	
Rational®	Redbooks (logo) ®	

The following terms are trademarks of other companies:

FileNet, and the FileNet logo are registered trademarks of FileNet Corporation in the United States, other countries or both.

JBoss, and the Shadowman logo are trademarks or registered trademarks of Red Hat, Inc. in the U.S. and other countries.

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication can help you develop content and process management applications with IBM FileNet® APIs. The IBM FileNet P8 suite of products contains a set of robust APIs that range from core platform APIs to supporting application APIs. This book focuses specifically on Content Engine and Process Engine APIs.

The two core components of IBM FileNet P8 Platform are Content Engine and Process Engine. Content Engine handles content storage, indexing, and retrieval. In addition, it offers services for enabling active content through events and subscriptions, allowing code to be executed on an event-driven basis. Process Engine handles all workflow and process routing. The Content Engine and Process Engine APIs enable you to develop applications that can perform the Content Engine and Process Engine functions.

In this book, we show you how to set up programming environments for Java™, .NET, REST API, and ECM Widget development. We provide a class overview, a description of how to make connections to the engines, and examples of exception handling.

Content Engine API topics that we discuss include creating, retrieving, updating, and deleting objects; querying and viewing documents; and batching and batch execution. We also explore more complex topics, including permissions and authorization, versioning, relationships, annotations, workflow subscriptions and event actions, metadata discovery, and dynamic security inheritance.

Process Engine API topics that we discuss include launching a workflow, searching for and processing work items, and working with process status. The more complex topics we cover include, Component Integrator application space, role, workbasket, resource navigation in Process Engine REST API, ECM Widgets, and building a custom Get Next In-basket widget.

To help you better understand programming with IBM FileNet APIs, we provide a sample application implemented for a fictional company. We include the data model, security model, workflows, and various applications developed for the sample. You can download them for your reference.

In this book, we also teach you how to enable and interpret the various logging options for the IBM FileNet P8 Platform APIs. We point you to the technical troubleshooting articles and provide important data-gathering points when troubleshooting IBM FileNet programming issues.

This book is intended for IBM FileNet P8 application developers. We recommend using this book in conjunction with the online ECM help.

The team who wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Wei-Dong Zhu (Jackie) is an Enterprise Content Management Project Leader with the International Technical Support Organization in Los Angeles, California. She has more than 10 years of software development experience in accounting, image workflow processing, and digital media distribution. Jackie holds a Master of Science degree in Computer Science from the University of the Southern California. Jackie joined IBM in 1996. She is a Certified Solution Designer for IBM Content Manager and has managed and lead the production of many enterprise content management books.

Bill Carpenter is an Enterprise Content Management Architect with the IBM Software Group in Seattle, Washington. Bill has 10 years of experience in the IBM enterprise content management business, as a Developer, Development Manager, and Architect. He is co-author of the book *IBM FileNet Content Manager Implementation Best Practices and Recommendations*, SG24-7547. He has extensive experience in building large software systems for Fortune 50 companies and has served as the CTO of an Internet startup. He has been a contributor to several open source projects. Bill holds degrees in Mathematics and Computer Science from Rensselaer Polytechnic Institute in Troy, New York.

Tim Lai is a Team Lead for the IBM FileNet Software Development Support Group in Costa Mesa, California. He has 12 years of experience in programming, consulting, and supporting ECM, E-Commerce, ERP, CRM, and Data Warehouse solutions worldwide. Tim holds a Bachelors of Science degree in Industrial and Systems Engineering from the University of Southern California. He joined IBM FileNet in 2003.

Wei Liao is a Enterprise Content Management Software Developer in the IBM China Development Lab. He has four years of IT industry experience in programming, J2EE development, Windows® .NET development, Web development and enterprise content management. He holds a Master of Science degree in Computer Science from the Beijing University of Technology. His core competency is Web 2.0 and Enterprise Mashup. He is proficient in building rich Internet applications, based on Web standards.

Michael Oland is an IT Advisory Specialist with IBM Enterprise Content Management Lab Services in Washington, D.C. He has 10 years experience in enterprise content management, which includes Web content delivery and management, document and information management, and business process management. For the last six years, he has focused primarily on content migration, and developing tools for content migration. He holds a degree in Broadcasting from the University of Tennessee, Knoxville.

James S Pagadala is a J2EE Enterprise Application Consultant from IBM India. He has more than seven years of experience in developing J2EE applications. His primary focus is supply chain management. He has managed and lead many J2EE applications in this domain. James has a Master of Computer Applications degree from the Karnataka Regional Engineering College, Surathkal, India. He joined IBM in 2001.

Juan Saad is a Senior Architect for Enterprise Content Management solutions with an IBM Business Partner in Melbourne, Australia. Juan has nine years of experience with organizations, designing, implementing, and maintaining Business Process Management and Content Management solutions based on the IBM FileNet P8 Platform. Juan has been involved in all phases of a system development life cycle, both functionally and technically, including client interface, business process analysis and modeling, application design and development, and system testing and research methods. He is a Certified Administrator and Developer for several IBM FileNet P8 products such as Business Process Manager, Content Manager, Records Manager, Email Manager, and Forms Manager.

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

► Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

► Send your comments in an e-mail to:

redbooks@us.ibm.com

► Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

1

Platform and API overview

Before beginning development for the IBM FileNet P8 Platform, it is important to review the platform itself, the core engines that comprise a IBM FileNet P8 deployment, the application programming interface (API) available for each engine, and how they communicate with each other.

This chapter discusses the following topics:

► Platform overview

► IBM FileNet API overview

► Communication between applications

1

1.1 Platform overview

The core IBM FileNet P8 Platform consists of two primary engines, Content Engine (CE) and Process Engine (PE). On top of these engines are built all user interfacing applications, such as the Application Engine (AE), Workplace XT, and Enterprise Manager. As an additional layer on top of the base, optional components are available for license, including IBM FileNet Records Manager. Figure 1-1 shows the relationship between the core engines and data storage.
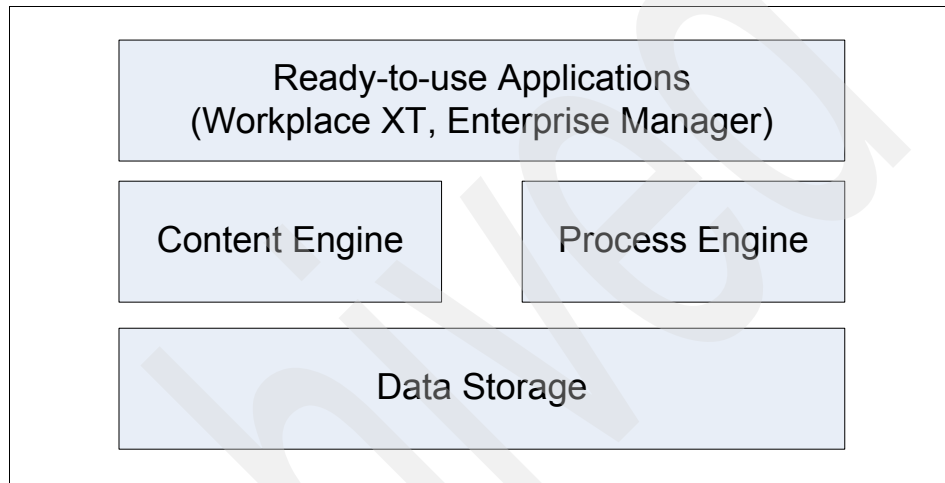
Figure 1-1   Core engines and data storage

The CE handles content storage, indexing, and retrieval. In addition, the CE offers services for enabling active content through events and subscriptions, allowing code to be executed on an event-driven basis. The PE handles all workflow and process routing. The applications that are ready for use include Workplace XT and IBM FileNet Enterprise Manager.

In addition to the core engines, a number of add-on products offer additional functionality and are built on the core engines. Some of these applications include IBM FileNet Records Manager and IBM FileNet Capture. See Figure 1-2 on page 3.
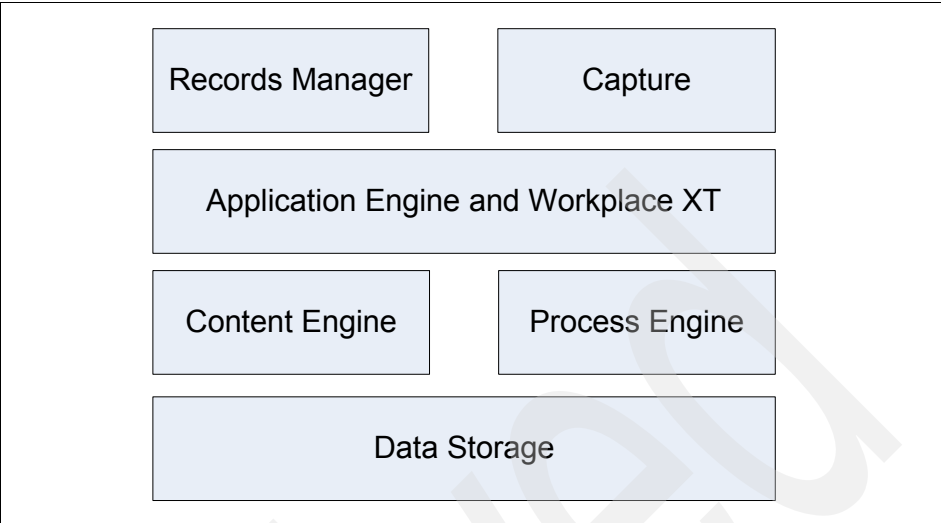
Figure 1-2   Core engines with additional applications

1.2 IBM FileNet API overview

The IBM FileNet suite of products contains a set of robust application programming interfaces (APIs). These APIs range from core platform APIs to supporting application APIs.

Core platform APIs

Within the core products, both primary engines, the CE and the PE, offer a series of APIs that are available for different languages and uses. Table 1-1 lists the primary APIs and what they are intended to be used for.

Table 1-1   Core P8 Platform APIs by use

Language or use	Content Engine	Process Engine
Java	Java API	Java API
.NET	.NET API	-
Web Services	CE Web Services	PE Web Services
Other	-	REST
P8 3.x Compatibility	COM Compatibility Layer	-
-	Java Compatibility Layer	-

Supporting application APIs

Along with the core platform APIs are other add-on applications, which often have APIs that are specific to that application. Several of these APIs are:

► IBM FileNet Records Manager Java API
► eForms JavaScript API
► eForms Java API
► Capture COM API
► Remote Capture Services .NET API
► Image Services Resource Adapter API

1.2.1 Content Engine APIs

To provide services for creating, querying, and working with content, the CE offers two primary and feature-compatible APIs: the Java and .NET APIs. In addition, the CE offers two compatibility APIs for applications that were written with the CE 3.x versions.

Content Engine Java API

IBM FileNet Content Manager provides a full-featured CE Java API. Any feature that is available in the CE server is available to Java programmers. These features include routine operations, such as retrieving and updating Document objects, and specialized operations, such as adding a custom class or property to an object store's metadata definitions.

This API can be configured to communicate with the CE server through the EJB or Web Service transports. The Java API set complies with the Enterprise JavaBeans 2.0 specification for the EJB transports that are used internally.

The Java API uses Java Authentication and Authorization Service (JAAS) for authentication purposes. Fine-grained authorization is implemented within the CE server and does not use JAAS.

Content Engine .NET API

IBM FileNet Content Manager provides a full-featured .NET API, which you can use to write programs in any .NET-compatible language. With a couple of exceptions, any feature that is available in the CE server is available to .NET programmers. The exceptions are mainly custom code that must be executed within the server, for example, EventActions. Because the CE server is a J2EE application, internally executed custom code is limited to Java-compatible technologies.

One significant feature available only with the .NET API is the use of Kerberos to perform authentication with Microsoft® Windows integrated login. This is only

possible when both the client application and the CE server are running on Microsoft Windows.

The .NET API communicates with the CE server over the Web Services transport.

Content Engine Web Services

Modern, loosely coupled frameworks, such as service-oriented architecture, favor Web services protocols for connecting components. IBM FileNet Content Manager provides Content Engine Web Services (CEWS) for accessing nearly all features available in the CE server.

Typically, if you, as a programmer, want to use a Web services interface, you obtain the interface description in the form of a Web Services Description Language (WSDL) file. You run the WSDL file through a toolkit to generate programming language objects for interacting with the Web services interface. You then usually build up a library of utilities to provide abstraction layers, caching, security controls, and other conveniences. The Java and .NET APIs provided by IBM FileNet Content Manager are already equivalent to that, and both APIs can use Web services as a transport. Consequently, there is not as much motivation to use CEWS directly.

There are still occasions where the direct use of CEWS might be useful:

▶ You have an application already using CEWS, and no plans exist for immediately porting it to the Java or .NET API.

▶ You are building an application component as part of a framework or technology stack in which the use of Web services is the model for communicating with external systems and the use of the CE Java or .NET API is not a reasonable possibility.

For these occasions, the direct use of CEWS is a good choice and is fully supported.

In theory, any current Web services toolkit can use the Content Engine WSDL to generate the interfaces that you use for your application. In practice, however, toolkits are still individualistic in their handling of various WSDL features; writing a WSDL for a complex service that is usable by a wide cross-section of Web services toolkits is difficult. Check the latest hardware and software support documentation, IBM FileNet P8 Platform 4.5.x hardware and software requirements, and only use a supported toolkit.

Your toolkit generates programming language stubs and other artifacts so that you can include CEWS calls in your program logic. The details of those artifacts vary from toolkit to toolkit, but you will surely see representations of CE objects,

properties, and update operations. The IBM FileNet Content Manager product documentation describes how these pieces interrelate, but we recommend you also read the developer documentation for the Java or .NET API to get an additional understanding of how the things mentioned in the WSDL were intended to be used.

A common question for CEWS developers is what the XML actually looks like for various requests. The official specification, of course, is the WSDL, and any XML generated by your client must comply with that. The simplest way to obtain samples of actual XML is to use either the .NET or Java API with CEWS transport and use a network tracing tool to capture the data from the interactions. You will see both the requests and responses. Because of the rich variety of possible interactions in CEWS, IBM support does not have a set of ready-made XML samples.

> **Note:** As of release 4.5.0, CE supports three Web services endpoints. The difference is in their handling of content attachments. The SOAP endpoint (FNCEWS40SOAP) uses inline content and carries significant performance costs. It should therefore be avoided. Support for the Direct Internet Message Encapsulation (DIME) endpoint (FNCEWS40DIME) is documented as deprecated and will eventually be removed. Therefore, all new code should be written for the Message Transmission Optimization Mechanism (MTOM) endpoint (FNCEWS40MTOM), and existing DIME code should be migrated to MTOM as soon as possible.

Content Engine Java Compatibility Layer API

IBM FileNet Content Manager 3.x provided a set of Java APIs which is now called the Java Compatibility Layer in P8 4.x. This Java compatibility layer allows almost all custom applications that are written in P8 3.x to continue functioning in P8 4.x with a few configuration changes.

All new development should be done with the Content Manager 4.x Java APIs because new features are incorporated only into the 4.x APIs.

The Java Compatibility Layer API is only mentioned in this book in terms of how to configure the environment. The book does not discuss Java Compatibility Layer programming.

Content Engine COM Compatibility Layer API

IBM FileNet Content Manager 3.x also provided a COM API set. This COM API set is now supported as a compatibility layer in P8 4.x, allowing many COM-based custom applications that are written in P8 3.x to continue functioning in P8 4.x with a few configuration changes.

Any new Microsoft environment development should be done by using the IBM FileNet Content Manager 4.x .NET API because all new features will be incorporated into the 4.x APIs.

The COM Compatibility Layer API is only mentioned in this Redbooks publication in terms of how to configure the environment. There will be no discussion on COM Compatibility Layer programming.

1.2.2 Process Engine APIs

To provide services for creating and managing workflows, the PE offers three APIs: the Java API, Process Engine Web Services, and the REST API.

Process Engine Java API

IBM FileNet Business Process Manager contains a full featured Java API set that allows for various interactions with the PE server. This API set may be familiar to a lot of IBM FileNet experienced programmers from the eProcess versions through IBM FileNet P8 4.x. This API set provides access to define, administer, and control runtime workflow processing. It can also be used to build custom Step Processors and Work Performers.

The PE Java API uses the CE Java API for authentication. Authentication-related calls can be done with EJB or WSI transports, though the use of the Web services transport is the normal configuration.

Process Engine Web Services

Process Engine Web Services (PEWS) provides a WSDL-based interface to communicate with PE. PEWS is a functional subset of the PE Java API.

Because there is no native PE .NET API, PEWS is ideal for development with .NET.

Similar to the CEWS, PEWS is also compliant with Web Services Interoperability Organization (WS-I) Basic Profile 1.0 and WS-Security.

Process Engine REST Services API

A new feature introduced in IBM FileNet P8 4.5 is the PE REST Service.

The PE REST Service provides access to the PE by following the Representational State Transfer (REST) architecture style. PE objects are exposed as REST resources which are accessible by standard HTTP methods. This API is ideal for Ajax-based Web applications that need to talk to PE.

1.2.3 Records Manager Java API

The IBM FileNet Records Manager Java API set allows for various interactions with IBM FileNet Records Manager including declaring, classifying, storing, and disposing of records. This API set is currently an extension of the IBM FileNet Content Manager Java 3.5 API set.

This API set is not covered in this book.

1.2.4 The eForms APIs

To provide services for managing electronic forms, eForms offers two APIs: the eForms JavaScript API and the eForms Java API.

The eForms JavaScript API

The eForms JavaScript API enables programmers to customize the client side processing of forms in a user's Web browser. Examples include:

► Retrieving and populating form data
► Attaching JavaScript event handlers
► Performing client side validation of user inputted data

This API set is not covered in this book.

The eForms Java API

The eForms Java API provides a low level access to the eForms objects and methods. This API performs the server side data processing. In practice, for most implementations, the eForms JavaScript API is the most useful in customizing eForms solutions.

This API set is not covered in this book.

1.2.5 Capture APIs

To provide services for acquiring images into IBM FileNet content repositories, IBM FileNet Capture offers two APIs: the Capture COM API and the Remote Capture Services .NET API.

Capture COM API

The Capture COM API is an existing IBM FileNet API set that allows for image acquisition into IBM FileNet content repositories including CE and Image Services. This API set provides Capture repository objects and ActiveX controls

for image acquisition (for example, scanning and document upload), batching, advanced document recognition (ADR), and others.

This API set is not covered in this book.

Remote Capture Services .NET API

Remote Capture Services (RCS) provides a .NET framework for developers to build custom .NET Web applications. The RCS toolkit includes Web interface controls, the RCS framework, as well as Capture repository objects.

This API set is not covered in this book.

1.2.6 Image Services Resource Adapter

The Image Services Resource Adapter (ISRA) is an existing IBM FileNet API that was developed for interaction with IBM FileNet Image Services. It is a Java based API that is J2EE Connector Architecture (JCA) compliant. This API set is not frequently used as interactions with Image Services can be accomplished through Content Federation Services for Image Services (CFS-IS). However, for systems that do not use Content Federation Services, ISRA can be used to communicate with Image Services.

This API set is not covered in this book.

1.3 Communication between applications

When approaching development for the IBM FileNet P8 Platform, an important aspect to remember is how the various components integrate and communicate with each other, because this helps with debugging communications issues in custom applications. Figure 1-3 on page 10 shows the communications paths between CE, PE, Application Engine, Workplace XT, and Enterprise Manager.
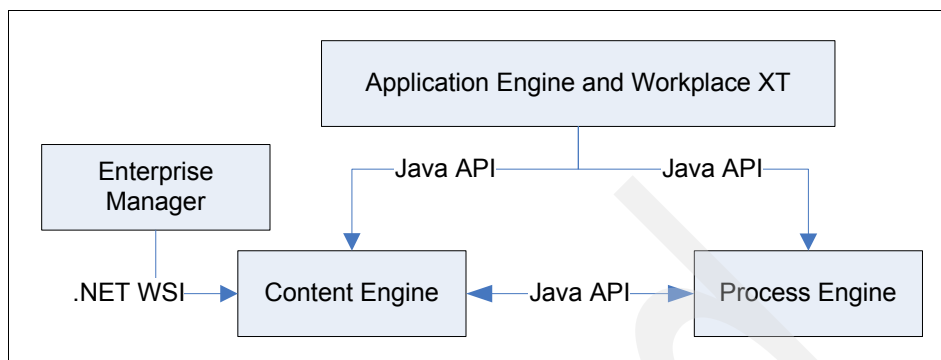
Figure 1-3   Application communication between core engines and available applications

For custom applications, the communications path used depends on both the API used and which transport is selected for use.

Figure 1-4 shows the typical communication path for .NET-based applications.
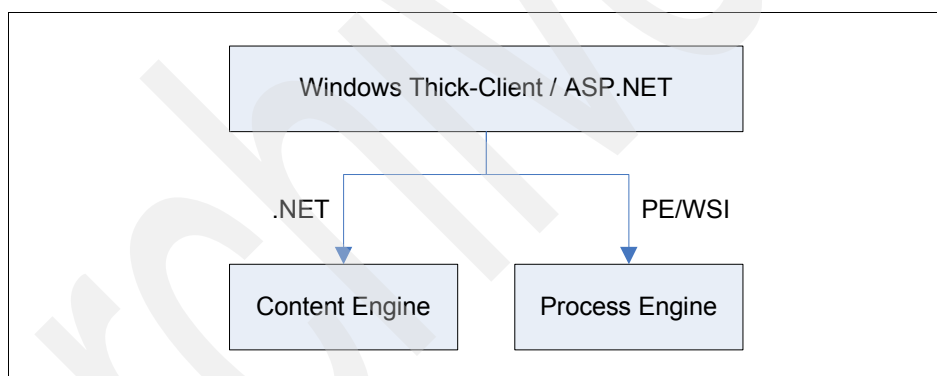
Figure 1-4   .NET application communication

Figure 1-5 on page 11 shows the typical communication path for Java-based applications. Although a J2EE application can use Web services transport to communicate with the CE, doing so is unusual. After an application is inside an application server, the EJB protocol has advantages in performance and ease of integration.
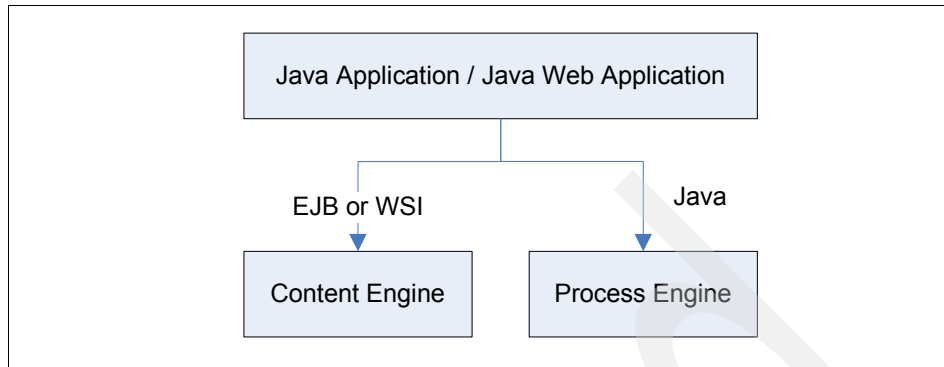
Figure 1-5   Typical Java application communication

In the remaining chapters of this book, we describe how to set up development environments and provide detailed discussions about the CE and PE APIs with a sample use case application.

2

Setting up development environments

In this chapter, we discuss setting up Java development environments for Content Engine (CE) and Process Engine (PE). We discuss the basic requirements for setting up programming with IBM FileNet APIs as well as providing instructions, where appropriate, for the sample application included with this book.

This chapter discusses the following topics:

► Content Engine Java development setup

► Process Engine Java development setup

► .NET environment setup for CE and PE

► PE REST API sample code development setup

► ECM Widgets development setup

2.1 Content Engine Java development setup

This section specifies setting up Java development environment to work with Content Engine Java API. We include the required libraries, transport protocols, thick client versus thin client requirements, and sample CE Java API application setup using Eclipse.

2.1.1 Required libraries

The two primary Java archive (JAR) files that are required for CE 4.0 development are:

- ► `Jace.jar`
- ► `log4j-<version>.jar`

    Where `<version>` is the supported version that is installed with CE.

These two `.jar` files can be obtained through installing the CE Client installer. By default, the CE Client is installed to `c:\Program Files\FileNet\CEClient` and the two `.jar` files are in the `lib` subdirectory. See Figure 2-1.
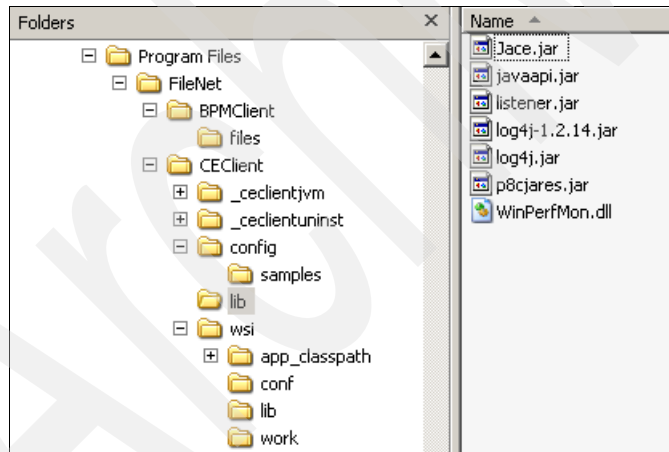
Figure 2-1   Content Engine Client libraries installation directory

2.1.2 Transport protocols

Communication with CE can occur through the EJB transport or Content Engine Web Services (CEWS) transport. The underlying communication protocols are handled internally by CE APIs so the application code remains the same regardless of transport selected. Chapter 3, "Introduction to Content Engine API

programming" on page 39 discusses more details of the transport protocols and the Uniform Resource Identifiers (URIs) to use for each.

2.1.3 Thick client versus thin client requirements

The term thick client as used in this book refers to stand-alone Java applications that are running outside of a J2EE application server such as a scheduled nightly batch job that uploads documents to CE. Thin client as used in this book refers to Java applications deployed and running in a J2EE application server such as a JSP-based Web application. The requirements for thick client versus thin client vary slightly, so we discuss each in detail.

Thick client EJB transport requirements

For the EJB transport, `.jar` files that are specific to the application server are required in the thick client environment. The required `.jar` files can change with each application server version. For the application servers supported in the current CE 4.5.0 release, Table 2-1 lists the required `.jar` files.

Table 2-1   Thick client EJB transport library requirements

Application server	Required .jar files
WebSphere®	Install the WebSphere Application Client version and patch that matches the CE application server. Use the WebSphere JRE. During runtime, use the `java.ext.dirs` JVM parameter mentioned in the next section
WebLogic	`wlclient.jar` or `weblogic.jar`
JBOSS	`jbossall-client.jar`

Figure 2-2 on page 16 shows the WebSphere 6.1 Application Client installed directories.
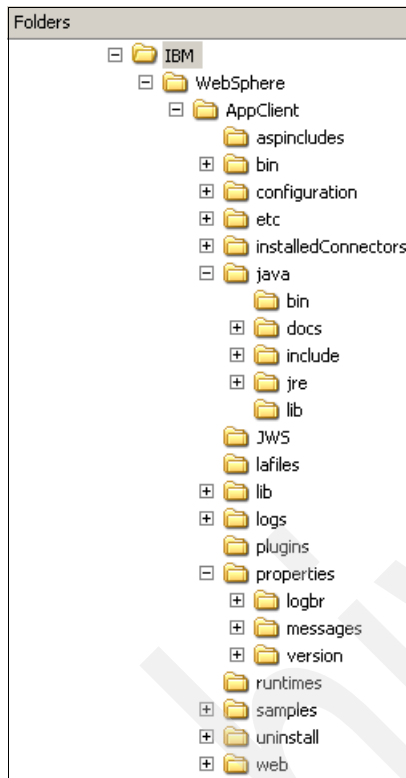
Figure 2-2   WebSphere 6.1 Application Client installation directory

In addition to the application server `.jar` files, the Java virtual machine (JVM) parameters listed in this section are required for thick client applications:

► For EJB transport with WebSphere (Figure 2-3 on page 17):

```
-Dcom.ibm.CORBA.ConfigURL=<WebSphereHome>\properties\sas.client.prop
s
-Djava.ext.dirs=<WebSphereHome>\java\jre\lib\ext;<WebSphereHome>\lib
;<WebSphereHome>\plugins
-Djava.security.auth.login.config=c:\Program
Files\FileNet\CEClient\config\samples\jaas.conf.WebSphere
```

Where, *<WebSphereHome>* refers to the WebSphere Application Client home directory, for example, `c:\Program Files\IBM\WebSphere\AppClient`.
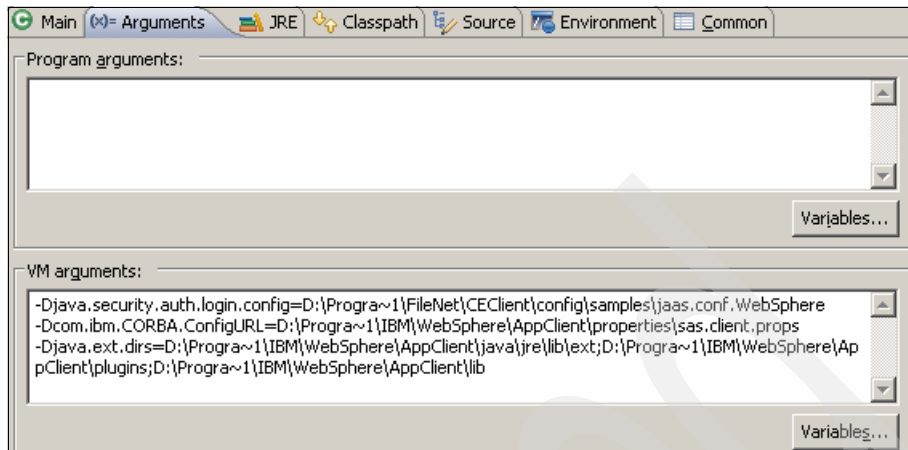
Figure 2-3   Eclipse runtime configuration for WebSphere EJB transport JVM parameters

> **Note:** The CE server host name you use in your connection URI and the
> sas.client.props must be resolvable through DNS from your client machine.
> For WebSphere, the host name and port specified in your client application
> must match the host name and port number specified in the
> BOOTSTRAP_ADDRESS parameter in the WebSphere Administration
> Console.

In `sas.client.props`, set the following properties (see Figure 2-4):

```
com.ibm.CORBA.securityServerHost=ceServerName
com.ibm.CORBA.securityServerPort=2809
com.ibm.CORBA.loginSource=none
```



```
com.ibm.CORBA.authenticationTarget=BasicAuth
com.ibm.CORBA.authenticationRetryEnabled=true
com.ibm.CORBA.authenticationRetryCount=3
com.ibm.CORBA.validateBasicAuth=true
com.ibm.CORBA.securityServerHost=hqdemo1
com.ibm.CORBA.securityServerPort=2809
com.ibm.CORBA.loginTimeout=300
com.ibm.CORBA.loginSource=none
```

Figure 2-4   Sample WebSphere sas.client.props entries

► For EJB transport with Weblogic:

```
-Djava.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
-Djava.naming.provider.url=t3://ceServer:7001
-Djava.security.auth.login.config=c:\Program
Files\FileNet\CEClient\config\samples\jaas.conf.WebLogic
```

► For EJB transport with JBoss:

```
-Djava.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
-Djava.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
-Djava.naming.provider.url=jnp://ceServer:1099
-Djava.security.auth.login.config=c:\Program
Files\FileNet\CEClient\config\samples\jaas.conf.JBoss
```

Note that the port numbers assume default application server ports. Change them as needed for your specific environments.

Thick client CEWS transport requirements

The CEWS transport does not require the client application and CE server to be on the same application server libraries. The CEWS transport is independent of the CE application server so the following instructions can be followed on any CEWS transport client.

> **Note:** For the CEWS transport in a thick client environment, the requirements are slightly different depending on which version of CE you are using. From CE 4.0 to 4.5.0, CE Java API utilized Systinet WASPJ libraries for communication with the CE server. Starting with CE 4.5.1, Systinet libraries are no longer used. We describe both configurations.

For CE 4.0 to 4.5.0, the Systinet libraries and configuration files are installed with the CE Client installation (default installation directory is `c:\Program Files\FileNet\CEClient\wsi`). Table 2-2 lists the required `.jar` files.

Table 2-2   Thick client CEWS transport library requirements for CE 4.0 to 4.5.0

Application server	Required .jar files
Applies to all application servers	► wasp.jar This file can be obtained from the `c:\Program Files\FileNet\CEClient\wsi\lib` directory. The other `.jar` files in that directory should *not* be placed on your classpath.

Figure 2-5 on page 19 shows the CEWS installation directory where `wasp.jar` can be obtained.
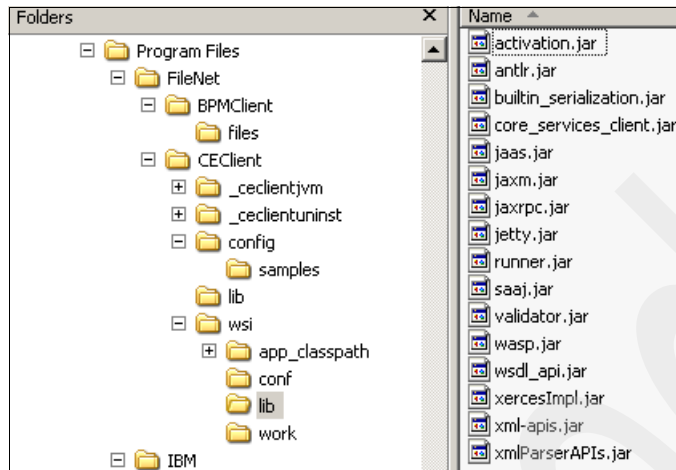
Figure 2-5   CE 4.5.0 CEWS transport installation directory

Starting with CE 4.5.1, Systinet libraries are no longer used. Instead, three new IBM libraries are required for CEWS transport as listed in Table 2-3.

Table 2-3   Thick client CEWS transport library requirements for CE 4.5.1 and beyond

Application server	Required .jar files
Applies to all application servers	► stax-api.jar ► xlxpScanner.jar ► xlxpScannerUtils.jar

As mentioned earlier, the CEWS transport requirements are the same for any application server. For the CEWS transport in CE 4.0 to 4.5.0, two JVM parameters are required (Figure 2-6 on page 20):

► `-Dwasp.location=c:\Program Files\FileNet\CEClient\wsi`

► `-Djava.security.auth.login.config=c:\Program Files\FileNet\CEClient\config\samples\jaas.conf.WSI`
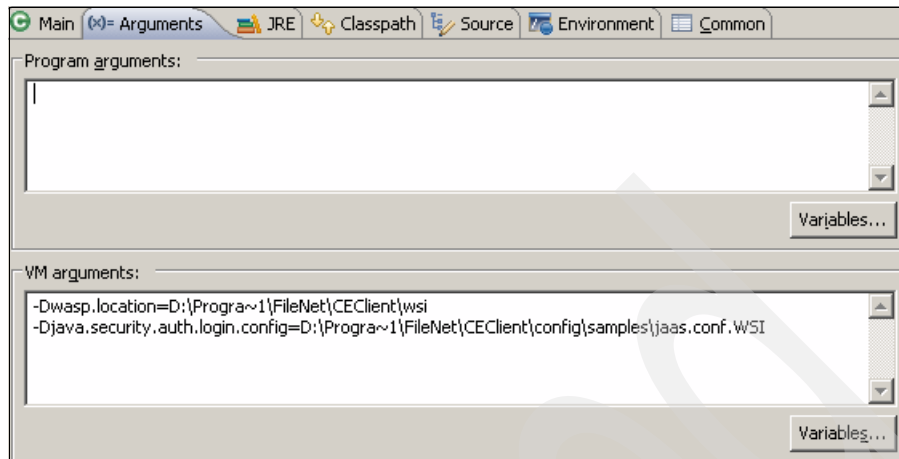
Figure 2-6   Eclipse runtime configuration for CEWS transport JVM parameters

For the CEWS transport in CE 4.5.1 and beyond, the only JVM parameter required is the `java.security.auth.login.config` file, thereby eliminating the requirement to define the wasp.location parameter.

Thin client EJB transport requirements

Thin client J2EE Web applications using the EJB transport have slightly different requirements than thick client stand-alone applications because much of the application server libraries and JVM variables are already defined or available within the application server container. Thin client applications do not require explicit references to application server libraries such as the WebSphere application client, `weblogic.jar`, or `jbossall-client.jar`.

The only JVM parameter required for thin client EJB transport applications is:

```
-Djava.security.auth.login.config=c:\Program
Files\FileNet\CEClient\config\samples\jaas.conf.AppServer
```

Replace *AppServer* with your application server such as WebSphere, WebLogic, or JBoss.

The requirements to setup a thin client Web application in a J2EE application server using EJB transport would be similar to the instructions for deploying WorkplaceXT. WorkplaceXT is a Java application that communicates over the EJB transport. The main point is to set up the trust relationships between the client application server and the CE application server. Refer to the P8 installation guide (under the section for Deploying Application Engine) for the steps to deploy and configure WorkplaceXT. It describes setting up the trust relationships, the required JVM parameter, classloader settings, and others.

One way to troubleshoot connectivity issues is to deploy WorkplaceXT on the client application server. If WorkplaceXT can connect to CE successfully, you can then focus on configuring your custom application in the same fashion.

Thin client CEWS transport requirements

> **Note:** You should use EJB transport with thin clients when possible. Especially in releases prior to 4.5.1, there is a strong possibility of having classloader conflicts with application server classes. Resolving classloader issues in J2EE applications can be time-consuming and frustrating.

Similar to thick clients, thin clients that are using the CEWS transport do not have a requirement for homogeneous application server versions and patches. So you can have a Weblogic thin client application communicate using the CE Java API over the CEWS transport to a CE server running on WebSphere.

From CE 4.0 to 4.5.0, a thin client application that communicates over CEWS transport must copy all eight `.jar` files mentioned in Table 2-4 on page 22 into the custom application WAR file's `WEB-INF\lib` directory. Example steps for a WebSphere thin client application are as follows:

1. Create and deploy a custom EAR file in which the `WAR\WEB-INF\lib` folder contains the eight .jar files in Table 2-4 on page 22.

2. In the WebSphere administration console, set the EAR and WAR classloader mode to **parent last**. (WebSphere 6.0 refers to this as *parent last*; WebSphere 6.1 refers to it as *application first*).

3. Install the CE client libraries with the Client installer as mentioned previously.

4. Add the following JVM arguments by using the WebSphere administration console and selecting **server** → **Process Definition** → **Java Virtual Machine** → **Generic JVM arguments**:

```
-Dwasp.location=c:\Program Files\FileNet\CEClient\wsi
-Djava.security.auth.login.config=c:\Program
Files\FileNet\CEClient\config\samples\jaas.conf.WSI
```

Table 2-4   Thin client CEWS transport library requirements for CE 4.0 to 4.5.0

Application server	Required .jar files
Applies to all application servers	► wasp.jar ► jaxrpc.jar ► jetty.jar ► runner.jar ► saaj.jar ► wsdl_api.jar ► activation.jar ► builtin_serialization.jar These eight `.jar` files can be obtained from the `c:\Program Files\FileNet\CEClient\wsi\lib` directory. See Figure 2-5 on page 19.

For CE 4.5.1 and later, there are no requirement differences between a thick client stand-alone application and a thin client application. The three `.jar` files that are still required for a thin client application using CEWS transport are listed in Table 2-5.

Table 2-5   Thin client CEWS transport library requirements for CE 4.5.1 and beyond

Application server	Required .jar files
Applies to all application servers	► stax-api.jar ► xlxpScanner.jar ► xlxpScannerUtils.jar

With the 4.5.1+ implementations, the only required JVM parameter is:

```
-Djava.security.auth.login.config=c:\Program
Files\FileNet\CEClient\config\samples\jaas.conf.WSI
```

2.1.4 Sample Content Engine Java API application setup in Eclipse

This book provides a list of CE Java API code samples. In addition to this book, you can find a number of other programming samples on the IBM Web site:

http://www.ibm.com/support/docview.wss?rs=3278&uid=swg27010422

We use one of the samples found there to demonstrate setting up a Java development environment on Eclipse. At this Web address, scroll to the "FileNet P8 Platform Developer Samples" section and select **FileNet Content Engine Java API Demo Sample Code**.

Download the `Demo_Java.zip` package. Use the following steps to configure and run this demonstration application in Eclipse:

1. Extract the `Demo_Java.zip` to a folder on your development environment. We refer to this folder where `Demo_Java.zip` is extracted to as *<DEMO_HOME>*.

2. Create a Java project in Eclipse. For the demo application, we use the project name `ContentEngineDemo`.

3. Add all the required CE `.jar` files as listed in 2.1.1, "Required libraries" on page 14. This demonstration is an example of a thick client application. Figure 2-7 shows the required `.jar` files for CE.
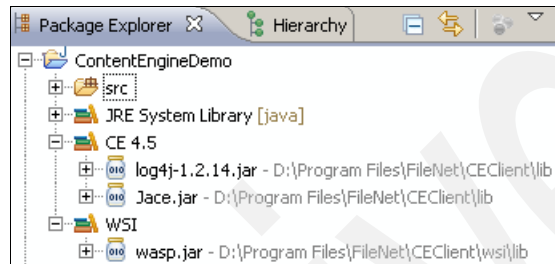
Figure 2-7   Required .jar files for CE and CEWS transport

4. For the demonstration (Demo) application, we use the CEWS transport. Add the `.jar` file listed in 2.1.3, "Thick client versus thin client requirements" on page 15. Figure 2-7 shows the required `.jar` file for thick client CEWS transport.

5. Create a package called `cesample` under the `src` folder in Eclipse. See Figure 2-8.
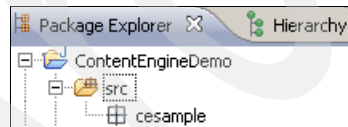
Figure 2-8   ContentEngineDemo create cesample package

6. Import all the classes from the folder *<DEMO_HOME>*`/src/cesample` as shown in Figure 2-9 on page 24.
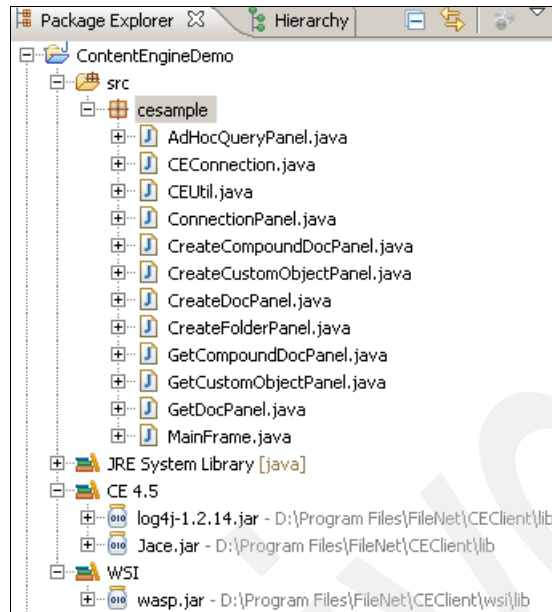
Figure 2-9   Imported demo Java classes

7. Create a Java application runtime profile in Eclipse for the `MainFrame.java` class. See Figure 2-10.
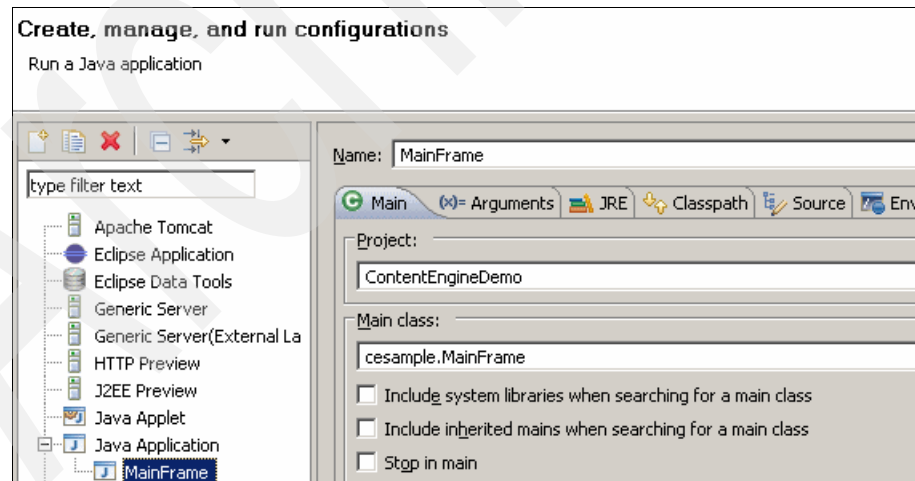
Figure 2-10   Java Application runtime profile for MainFrame.java

8. Add all the required JVM parameters as listed in 2.1.3, "Thick client versus thin client requirements" on page 15. See Figure 2-11 on page 25.
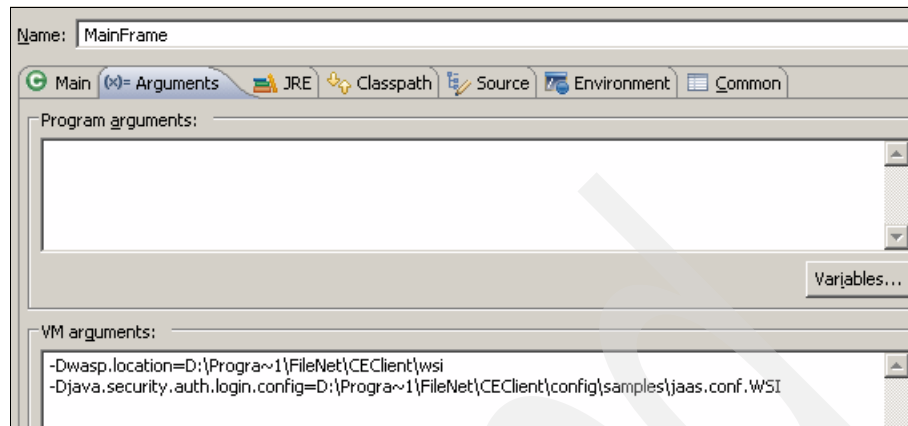
Figure 2-11   Runtime JVM parameters for MainFrame.java

9. Run the Java class `MainFrame.java` as a Java Application from Eclipse. See Figure 2-12.
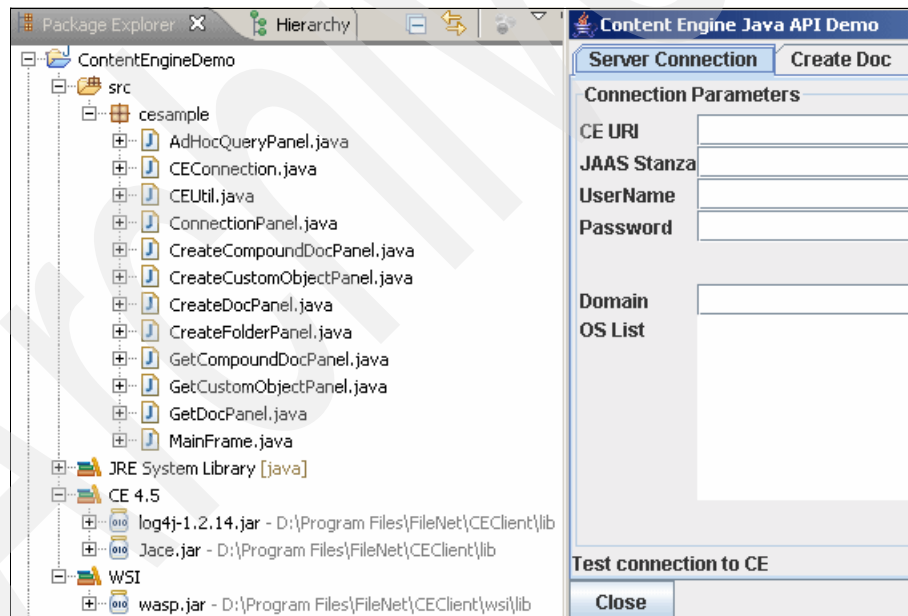
Figure 2-12   Demo application main page

10. Enter the following input values to test the CE connectivity. For this example, we use the CEWS transport to connect to CE:

   – CE URI as:

     ```
     http://<server-name>:<port-number>/wsi/FNCEWS40MTOM
     ```

   – JAAS Stanza as FileNetP8WSI

   – Valid user name

   – Password of user

11. After entering the input values, click **Test Connection**. The demo application connects to CE and retrieves the domain name along with a list of existing object stores. Figure 2-13 shows a successful connection and lists the COLL, and E2E object stores.
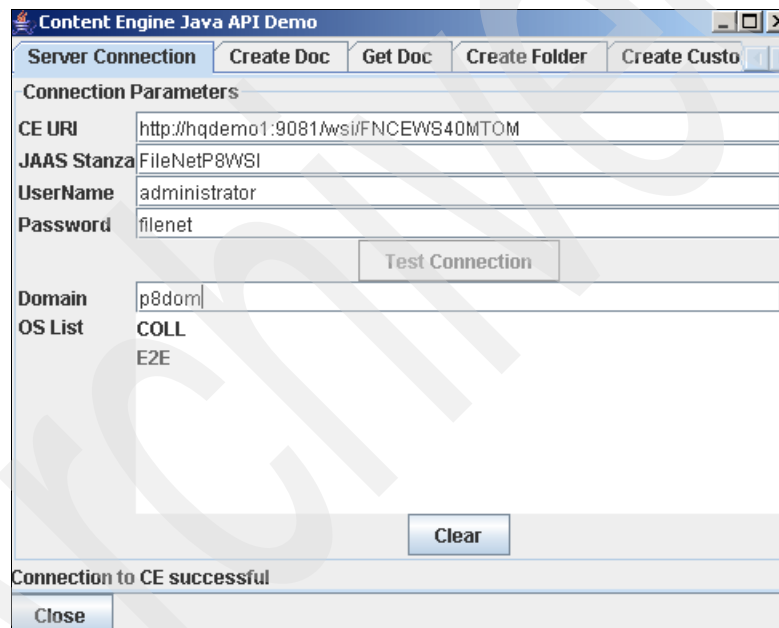
Figure 2-13   Demo application: Test Server Connection

12. You can now click the other tabs of the Demo application such as Create Doc, Get Doc, and Create Folder, to test each feature.

A `Readme.doc` file accompanies the Demo application. Refer to it for further information about using the Demo application.

2.2 Process Engine Java development setup

PE relies on CE for authentication and directory service access operations. This section assumes a development environment is properly configured for CE and the selected transport protocol.

2.2.1 Required libraries

The PE API consists of the following libraries:

► pe.jar

► pe3pt.jar

► peResources.jar

These three files can be obtained by running the PE Client Installer. By default, the PE client is installed to `c:\Program Files\FileNet\BPMClient`. The `.jar` files are located in the `files` subdirectory, as shown in Figure 2-14.
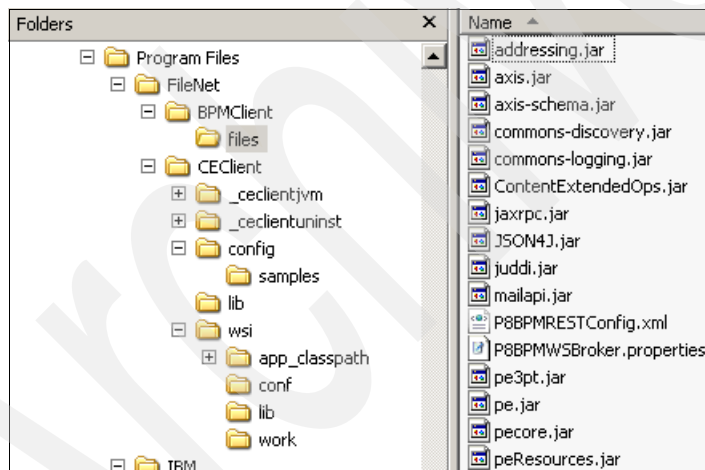
Figure 2-14   Process Engine Client libraries installation directory

In Eclipse, a PE development environment looks similar to Figure 2-15 on page 28.
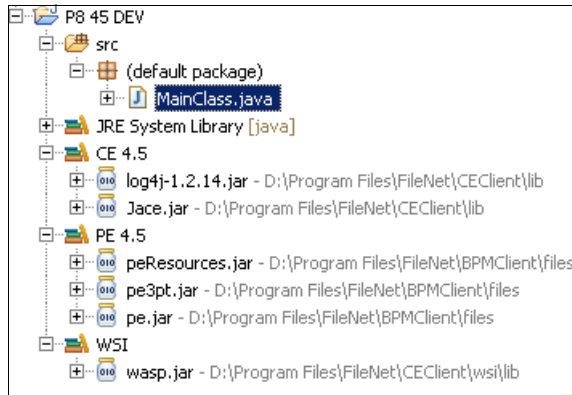
Figure 2-15   Eclipse jar files for Process Engine development

Java virtual machine parameters

Ensure the Java virtual machine (JVM) parameters required to set up CE for Java development are already configured. Add the following parameter to set up PE for Java development (and see Figure 2-16):

```
-Dfilenet.pe.bootstrap.ceuri=[Content Engine URI]
```

Add the following parameter to use EJB transport with WebSphere:

```
-Dfilenet.pe.bootstrap.ceuri=iiop://ceserver:2809/FileNet/Engine
```

Add the following parameter to use CEWS transport with WebSphere:

```
-Dfilenet.pe.bootstrap.ceuri=http://ceserver:9080/wsi/FNCEWS40MTOM/
```
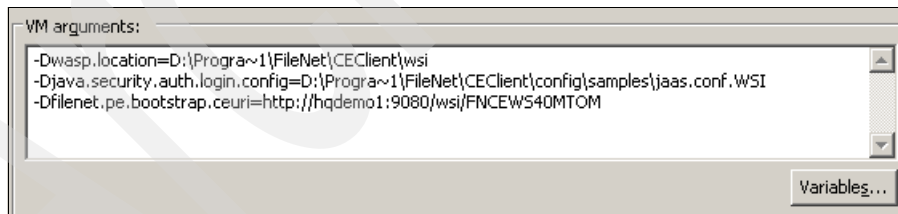

Figure 2-16   Process Engine JVM parameters

When the PE Java development project has the `WcmApiConfig.properties` file in a directory as part of the classpath, there is no need to add the `filenet.pe.bootstrap.ceuri` parameter. This setting is taken from the `WcmApiConfig.properties` values (see Example 2-1 on page 29 for using with EJB transport with WebSphere and Example 2-2 on page 29 for using with CEWS transport with WebSphere).

Example 2-1 WcmApiConfig.properties using EJB transport with WebSphere

```
RemoteServerUrl = cemp:iiop://ceserver:2809/FileNet/Engine
RemoteServerUploadUrl = cemp:iiop://ceserver:2809/FileNet/Engine
RemoteServerDownloadUrl = cemp:iiop://ceserver:2809/FileNet/Engine
...
```

Example 2-2 WcmApiConfig.properties using CEWS transport with WebSphere

```
RemoteServerUrl = cemp:http://ceserver:9080/wsi/FNCEWS40MTOM/
RemoteServerUploadUrl = cemp:http://ceserver:9080/wsi/FNCEWS40MTOM/
RemoteServerDownloadUrl = cemp:http://ceserver:9080/wsi/FNCEWS40MTOM/
...
```

2.3 .NET environment setup for CE and PE

In addition to Java, IBM FileNet P8 offers connectivity and API sets for the
Microsoft .NET 2.0 environment. Configuring a client workstation or server to use
the P8 APIs typically involves installing the prerequisites and adding references
to the P8 API library DLL or Web Services Description Language (WSDL) file. In
this section, we provide examples of setting up a Visual Studio.NET 2005
development environment for:

► CE .NET API

► Content Engine Web Services API

► Process Engine Web Services API

2.3.1 Prerequisites

For a .NET environment, the two primary prerequisites that apply to both the CE
and PE are:

► Microsoft .NET Runtime 2.0

► Microsoft Web Services Enhancements 3.0

2.3.2 Running the sample application supplied by IBM

After the prerequisites have been installed, you can run the sample applications
supplied by IBM. If they come compiled, you run them simply by double-clicking
on the supplied executable; however, if they are source-only, then you must
compile them using Microsoft Visual Studio.NET.

2.3.3 Configuring VisualStudio.NET 2005

To build an application in VisualStudio using the P8 API, create the solution and then reference the appropriate .NET assemblies, as follows:

1. Install the CE .NET API (if not already installed).

2. Start Visual Studio 2005.

3. Create the solution.

4. Add a reference to the Microsoft.Web.Services3 assembly by selecting
   Project → **Add Reference** → **.NET** → **Select Microsoft.Web.Services3**.

Content Engine .NET API

CE 4.x provides a native .NET API. To use the CE .NET API, install the CE .NET API library using one of the following ways:

► Run the IBM FileNet CE install CD and selecting install the .NET API.

► Copy the FileNet.Api.dll to the target system.

► Bundle the above FileNet.Api.dll in with a custom application being deployed.

To use the CE .NET API in Visual Studio.NET 2005, add a reference to the FileNet.Api.dll by selecting **Project** → **Add Reference** → **File** → **Find FileNet.Api.dll**, as shown in Figure 2-17 on page 31.
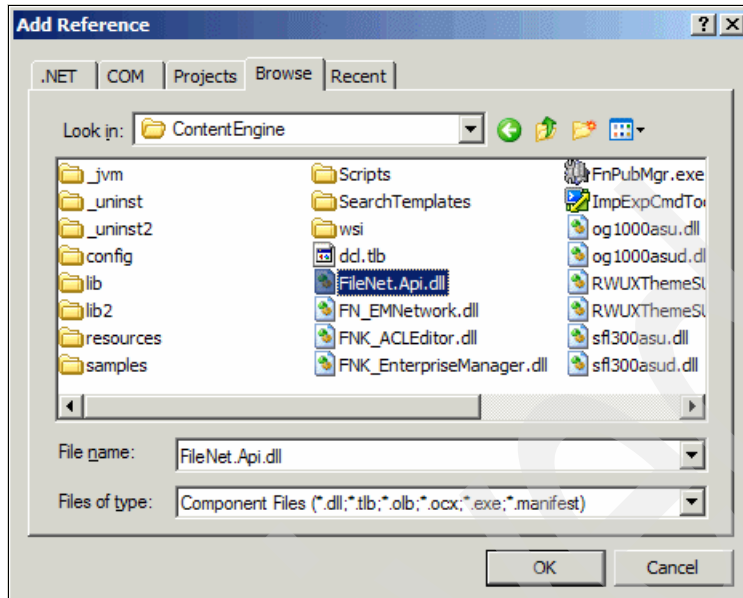
Figure 2-17   Adding a reference to the CE API DLL

Content Engine Web Services

Content Engine Web Services (CEWS) can be consumed by various client environments including Java and .NET. In this example, we show how to add CEWS in Visual Studio.NET 2005.

In Visual Studio .NET 2005, add a Web Reference to the Content Engine WSDL, as follows (see Figure 2-18 on page 32):

1. Select **Project** → **Add Web Reference**.

2. Enter: `http://<server-name>:<port-number>/wsi/FNCEWS40MTOM`

3. Give the reference a name (the samples use CEWSI).

4. Click **Add Reference**.
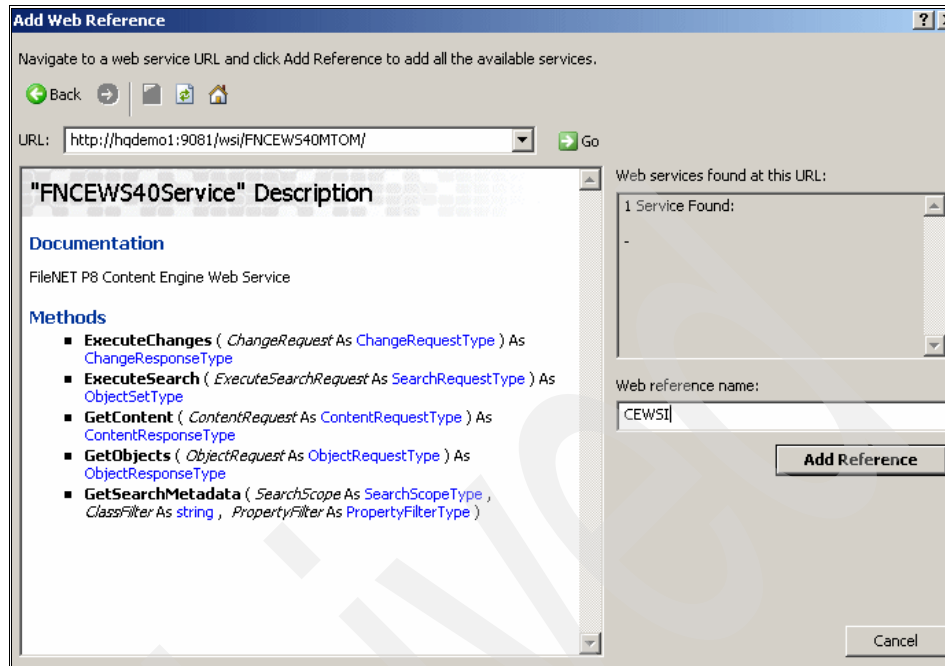
Figure 2-18   Adding a Content Engine Web Service .NET Web Reference

Process Engine Web Services API

Similar to CEWS, Process Engine Web Services (PEWS) can also be consumed by various client environments including Java and .NET. In this example, we show how to add PEWS in Visual Studio.NET 2005.

In Visual Studio.NET 2005, add a Web Reference to the Process Engine WSDL as follows (see Figure 2-19 on page 33):

1. Select **Project** → **Add Web Reference.**

2. Enter: `http://<server-name>:<port-number>/wsi/ProcessEngineWS`

3. Give the reference a name (the samples use peWS).

4. Click **Add Reference**.
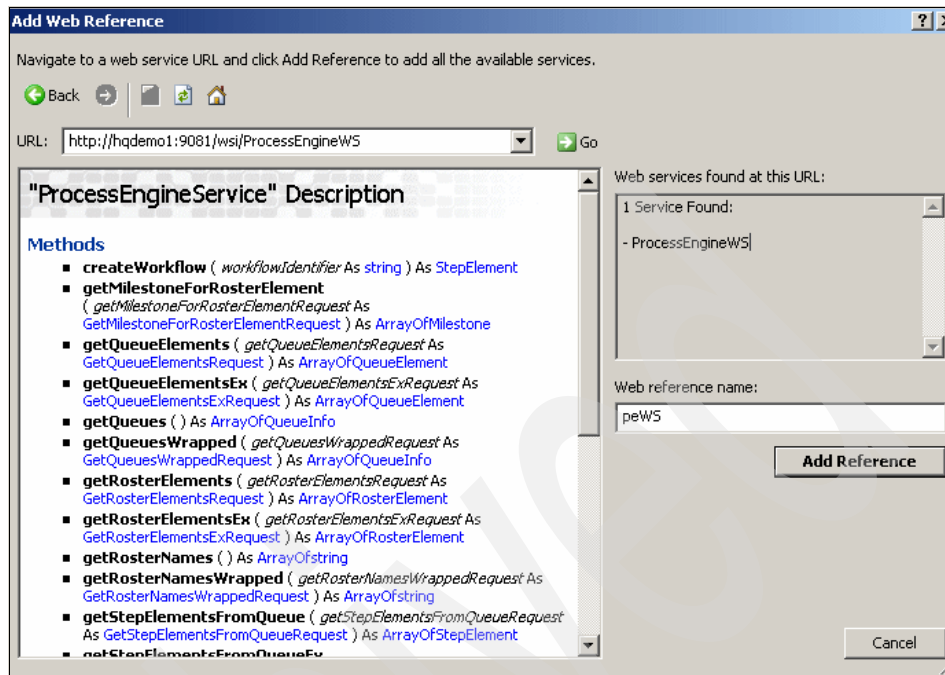
Figure 2-19   Adding a Process Engine Web Service .NET Web Reference

2.4 PE REST API sample code development setup

To demonstrate the PE REST Service API, we include a sample application with this book. To run the PE REST sample application, you need to create a Web application in Eclipse. To create and deploy the application follow these steps:

1. Download Dojo library version 1.0.2 from the following Dojo toolkit Web site and save it to your local hard disk:

   `http://download.dojotoolkit.org/release-1.0.2/dojo-release-1.0.2.zip`

2. In Eclipse, select **File** → **New** → **Other** to open the wizard dialog. Select **Dynamic Web Project**. See Figure 2-20 on page 34.
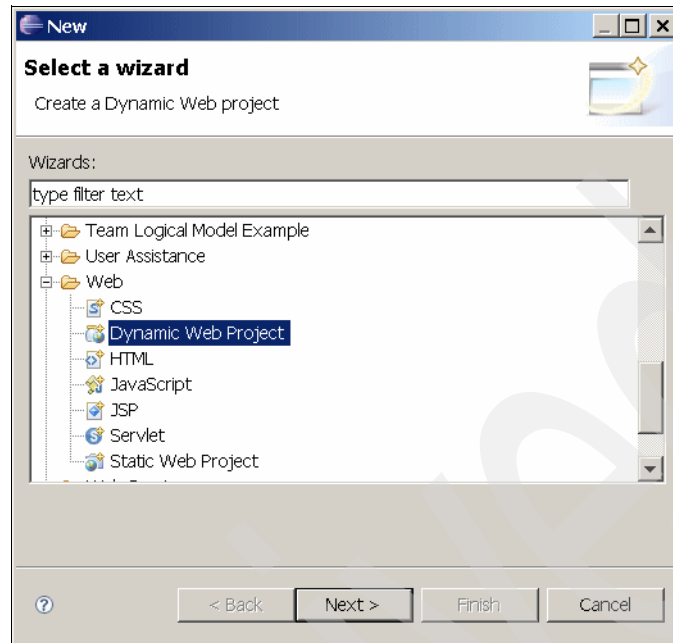
Figure 2-20   Create New Web Application

3. In the next dialog, specify the project name **PERESTSample**. Then, click **Finish** to create the project. See Figure 2-21 on page 35.
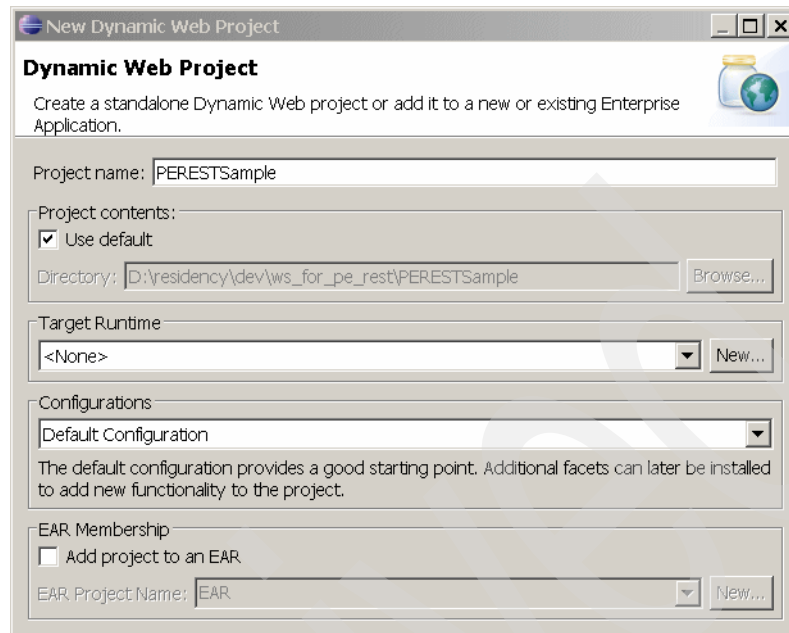
Figure 2-21   Specify the Project Name

4. Extract the Dojo library `.zip` file to a temporary folder and copy the Dojo library into project folder `dojoroot`. After this step, the project structure looks like Figure 2-22.
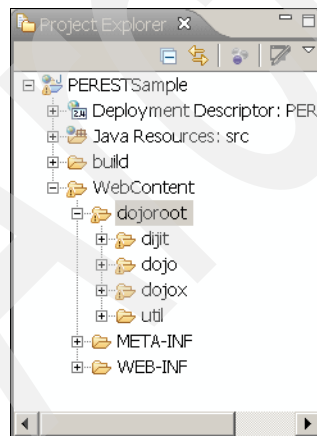
Figure 2-22   Code structure after you copy Dojo library

5. Create a test page to run the sample code for PE REST API under folder samples. See Figure 2-23 on page 36.
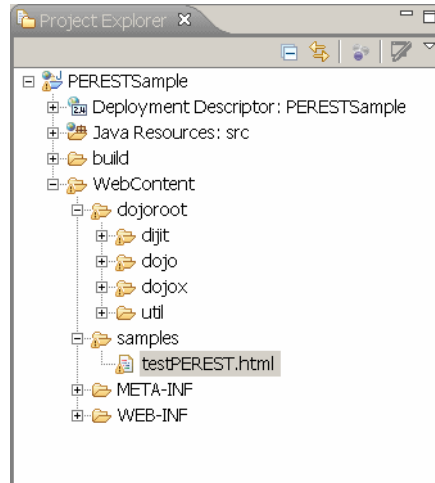
Figure 2-23   Test page for PE REST API

6. In the `testPEREST.html` test page, copy and paste the code skeleton shown in Example 2-3.

Example 2-3   Code skeleton to test PE REST API

```
<html>
   <head>
      <title>Test PE REST API</title>
      <script type="text/javascript"
         src="../dojoroot/dojo/dojo.js"
         djConfig="isDebug:true"></script>
      <script type="text/javascript">
         dojo.addOnLoad(function(){
            // copy the sample code here
         })
      </script>
   </head>
   <body>
   </body>
</html>
```

7. Copy the sample code of PE REST API into the body of the anonymous function called by `dojo.addOnLoad`. Example 2-4 on page 37 shows a code example.

Example 2-4   A code sample to run PE REST API

```
<html>
    <head>
        <title>Test PE REST API</title>
        <script type="text/javascript"
            src="../dojoroot/dojo/dojo.js"
            djConfig="isDebug:true"></script>
        <script type="text/javascript">
            dojo.addOnLoad(function(){
                // Ensure to be authenticated with the Application Server
container
                var baseURL =
"http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
                var applicationSpaceName = "GenericApproval";
                // Construct the URI for roles
                var url = baseURL + "appspaces/" + applicationSpaceName +
"/myroles";

                // Use GET method to retrieve the roles
                dojo.xhrGet({
                        url: url,
                        handleAs: "json-comment-optional",
                        load: function(data) {
                            // Callback to handle the data
                            for(var roleName in data) {
                                var role = data[roleName];
                                console.log("AuthoredName:" + roleName);
                                console.log("Display Name:" + role.name);
                                console.log("URI:" + role.URI);
                            }
                        },
                        error: function(data) {
                            // The error callback
                            console.dir(data);
                        }
                    }
                );
            })
        </script>
    </head>
    <body>
    </body>
</html>
```

8. After you create the project and sample code, deploy this project to WebSphere to test it. In Eclipse, select **File** → **Export** to open the export wizard. In the pop-up dialog, select **WAR File**, and save it to a folder. See Figure 2-24 on page 38.
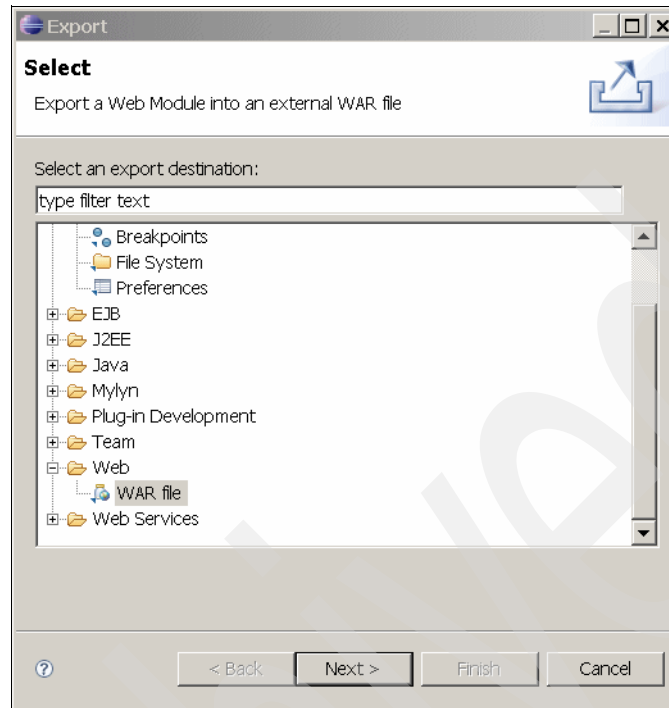
Figure 2-24   Export the project as WAR file

9. Deploy the WAR file to WebSphere. The context root for the deployed Web application should be `PERESTSample`.

10. Access the test page to run the sample code for PE REST API. The Web address of the test page is:

    `http://<server-name>:<port-number>/PERESTSample/testPEREST.html`

> **Note:** Before you test the sample code, log in to WorkplaceXT first to ensure you can be authenticated with the application server container.

2.5 ECM Widgets development setup

ECM Widgets are explained in detail in Chapter 6, "Advanced Process Engine API programming" on page 167. It also covers the development environment requirements.

3

Introduction to Content Engine API programming

This chapter provides an introduction to developing with the Content Engine (CE) APIs. It covers the basic principles and concepts to create, find, and work with CE objects. Most of the code snippets we show in this chapter are from the sample applications that we created for this book. For more details about the sample applications, refer to Chapter 7, "Sample applications for Fictional Auto Rental Company A" on page 223.

This chapter discusses the following topics:

► Content Engine API class overview

► Making the initial connection

► Exception handling

► Creating, retrieving, updating, and deleting objects

► getInstance() versus fetchInstance()

► Querying

► Viewing documents

► Batching and batch execution

3.1 Content Engine API class overview

The CE API is divided into logical groups of related items that use packages in Java and namespaces in .NET. For ease of use, the class structure generally mirrors the classes in CE metadata (for example, documents are accessed by using the `Document` and `DocumentSet` classes, folders with `Folder` and `FolderSet` classes, and so on). The API also offers a set of `Factory` classes that exist to create new objects and fetch existing objects from the CE. In this book, we describe the two primary CE APIs:

▶ CE Java API

▶ CE .NET API

The Java and .NET APIs are similar, varying only in stylistic naming conventions. Classes and interfaces in packages that are directly under `com.filenet.api` (for Java) and namespace `FileNet.Api` (for .NET) are *exposed*, meaning they are available for your use. Unless specifically documented otherwise, classes in any other package or namespace are not exposed; they are strictly internal and not supported for external use.

For CE development, the solution platform and the business use cases dictate what API to use. For example, if you plan to implement a Windows-based thick client, using the .NET API might be appropriate. However, for event action development or in a pure Java shop, the Java API would be appropriate.

In this book, we include code snippets that show how to use the CE APIs for many use cases that are related to CE development. Because both the Java and .NET APIs are feature-compatible, most code snippets in this chapter are shown in both C# and Java languages. Of course, for the .NET API, any .NET-compatible language binding can be used for development.

3.1.1 Content Engine API class model

> **Note:** Most of the CE API types are interfaces, but a few are classes. To avoid continuously repeating that we are talking about classes or interfaces, we refer to them all simply as classes.

The CE API has over a hundred classes. They are arranged into a number of packages or namespaces, organized by functional areas, which include:

► Core

Core (`com.filenet.api.core` and `FileNet.Api.Core`) provides classes related to the core business objects and other classes that are used in most applications. Examples include:

– Java: `EntireNetwork, Document, Folder, CustomObject, Factory, UpdatingBatch, Connection`

– .NET: `EntireNetwork, IDocument, IFolder, ICustomObject, Factory, UpdatingBatch, Connection`

► Meta

Meta (`com.filenet.api.meta` and `FileNet.Api.Meta`) provides classes for holding immutable metadata for CE classes and properties. Examples include:

– Java: `ClassDescription, PropertyDescription, PropertyDescriptionDateTime`

– .NET: `ClassDescription, IPropertyDescription, IPropertyDescriptionDateTime`

► Admin

Admin (`com.filenet.api.admin` and `FileNet.Api.Admin`) provides classes used in the administration of a CE, including classes for updating metadata objects. Examples include:

– Java: `ClassDefinition, PropertyDefinition, PropertyDefinitionDateTime, DirectoryConfiguration, PEConnectionPoint, ServerInstance, TableDefinition`

– .NET: `ClassDefinition, IPropertyDefinition, IPropertyDefinitionDateTime, IDirectoryConfiguration, IPEConnectionPoint, IServerInstance, ITableDefinition`

► Security

Security (`com.filenet.api.security` and `FileNet.Api.Security`) provides classes related to authentication, authorization, and user-specific and group-specific data. Examples include:

– Java: `User, Group, AccessPermission, MarkingSet`

– .NET: `IUser, IGroup, IAccessPermission, IMarkingSet`

- ► Query

  Query (`com.filenet.api.query` and `FileNet.Api.Query`) provides classes related to constructing and performing CE searches. Examples include:
  - Java: `SearchScope`, `RepositoryRow`
  - .NET: `SearchScope`, `IRepositoryRow`

- ► Collection

  Collection (`com.filenet.api.collection` and `FileNet.Api.Collection`) provides type-safe classes that are related to collections of objects. Examples include:
  - Java: `FolderSet`, `ContentElementList`
  - .NET: `IFolderSet`, `IContentElementList`

- ► Events

  Events (`com.filenet.api.events`, not available in .NET) provides classes representing events triggered on CE objects, as well as classes related to handling those events raised within the CE. Because all events are run inside the CE context, there is no equivalent .NET package. Examples include:
  - Java: `FileEvent`, `UnfileEvent`, `EventAction`, `InstanceSubscription`

- ► Property

  Property (`com.filenet.api.property` and `FileNet.Api.Property`) provides classes related to CE properties and property values. Examples include:
  - Java: `Properties`, `PropertyDateTime`, `PropertyDateTimeList`, `PropertyFilter`
  - .NET: `IProperties`, `IPropertyDateTime`, `IPropertyDateTimeList`, `IPropertyFilter`

- ► Constants

  Constants (`com.filenet.api.constants` and `FileNet.Api.Constants`) provides classes defining collections of related, type-safe constant values. Examples include:
  - Java and .NET: `AccessRight`, `Cardinality`, `DatabaseType`, `PropertyNames`, `ReservationType`

Figure 3-1 on page 43 shows the relationship between key classes that are used for creating, storing, searching, and retrieving objects within the CE.
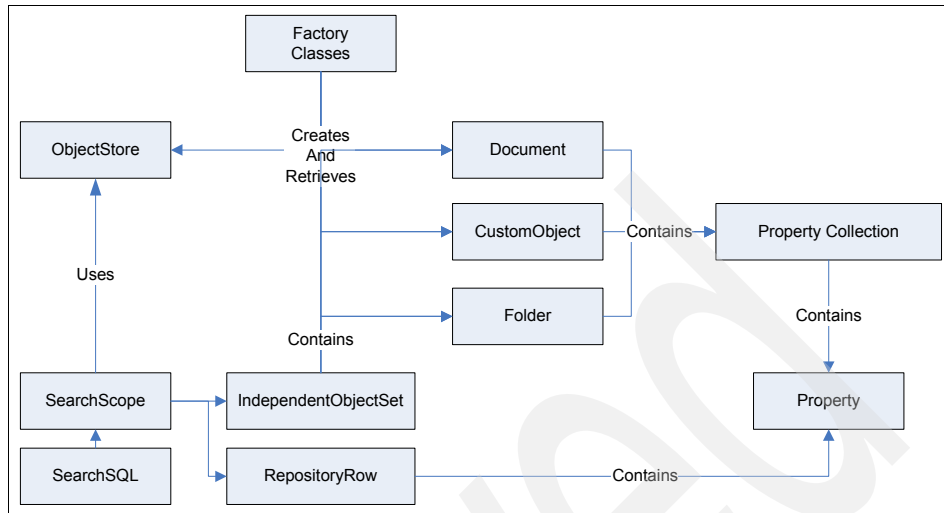
Figure 3-1   Class relationships and general object flow

3.2 Making the initial connection

Before working with the CE, an initial connection must be made. Making this
initial connection to the CE typically involves the following steps:

1. Authenticate or prepare the `UserContext`.
2. Get a connection object.
3. Get a domain object.
4. Get an object store object.

After the connection has been made, assuming the application user is properly
authenticated, the core objects (Connection, Domain, and ObjectStore) will be
available for use.

Connection

A *connection* object tells the API how to connect to the CE server, primarily
through the Uniform Resource Identifier (URI). The API then deduces both the
method for connecting and the location of the CE from the URI. A connection
object does not contain any user identification information. A connection is a
lightweight API object that does not maintain any server state or tie up any server
resources except when a server call is actually in progress.

Domain

A *domain* is an object holding IBM FileNet P8 resources at or above the object store level. These resources include:

- ► Object stores
- ► Fixed content devices
- ► Marking sets

ObjectStore

An *object store* is a collection of classes, objects, and metadata, and represents a logical division of data within a CE domain. Other objects can be retrieved directly from the object store or can be used with the Factory methods to create and retrieve objects such as:

- ► Documents
- ► Custom objects
- ► Folders
- ► Class and property definitions

3.2.1 User authentication

When making an initial connection to the CE, the user must be authenticated to access resources and to retrieve objects. Java and .NET have slightly different requirements for user authentication.

Authentication in Java

For a Java application, the three primary methods for authenticating a user are:

- ► Container managed authentication

- ► A custom JAAS LoginModule that authenticates directly with the authentication provider

- ► The UserContext object with its pushSubject method

For container-managed and LoginModule-based authentication, work with your application server provider or internal administration and development teams for the appropriate code.

Although the ideal situation for authentication is either container-managed or LoginModule-based authentication, the Java API does provide a convenience method for authentication through the UserContext object. When doing a UserContext-based login, you create a Java Authentication and Authorization Service (JAAS) subject and push it onto the UserContext stack. After all CE-related operations are complete, you should pop the Subject back off the

stack. See Example 3-1 for how to create the subject, push it onto the UserContext, and then pop it off the stack.

Example 3-1   UserContext authentication in Java

```
Subject subject = UserContext.createSubject(connection, "username",
      "password", null);

UserContext.get().pushSubject(subject);
try
{
   // do something
}
finally
{
   UserContext.get().popSubject();
}
```

User authentication in .NET

For .NET-based applications, authentication is always handled through the UserContext object. The UserContext requires a valid SecurityToken be passed to either the SetProcessSecurityToken or the SetThreadSecurityToken static method. In a .NET application, this security token can be either a UsernameToken provided by Microsoft Web Services Enhancements or a KerberosToken for Kerberos-based single sign on. Example 3-2 shows basic UsernameToken authentication.

Example 3-2   UsernameToken authentication in .NET

```
UsernameToken token = new UsernameToken("username", "password",
                            PasswordOption.SendPlainText);

UserContext.SetProcessSecurityToken(token);
```

3.2.2 Java

Java offers two data transport layers: the native Enterprise JavaBeans (EJB) transport offered by the application server or the CE Web Services (CEWS) transport. The transport used is indicated by the CE server URI passed into the Connection object. Table 3-1 on page 46 shows examples of available CE connection URIs.

> **Note:** For the Web services transport, the three endpoints are: FNCEWS40MTOM, FNCEWS40DIME, and FNCEWS40SOAP. The primary endpoint is MTOM. The other two endpoints, DIME and SOAP, exist for compatibility purposes only. All examples in this book use MTOM as the endpoint.
>
> For historical reasons, CEWS transport is sometimes referred to as Web Services Interface (WSI) transport or Web services transport. These are just naming differences and all refer to exactly the same thing. The CEWS transport is exactly the same set of Web services as would be used by someone calling CEWS directly (assuming that the persons uses a FNCEWS40* endpoint).

Table 3-1   Content Engine connection URIs

Protocol	URI
Web Services	`http://server:port/wsi/FNCEWS40MTOM/`
EJB/WebSphere	`iiop://server:port/FileNet/Engine`
EJB/Weblogic	`t3://server:port/FileNet/Engine`
EJB/JBoss	`jnp://server:port/FileNet/Engine`

Getting an object store in Java consists of the following steps (see Example 3-3 on page 47 and Example 3-4 on page 47):

1. Get the connection object from `Factory.Connection`.

2. Create the Java Authentication and Authorization Service (JAAS) subject either by using `UserContext.createSubject` and pushing that subject into the `UserContext`, or by any way that creates a valid JAAS subject.

3. Get the domain object by using `Factory.Domain.getInstance()`.

4. Fetch the object store by using `Factory.ObjectStore.fetchInstance()`.

> **Note:** The main differences between CEWS and EJB when connecting are the URI and the JAAS stanza, as shown in **bold** in Example 3-3 and Example 3-4 on page 47.

Example 3-3   Getting the initial connection with Web services transport in Java

```
// Set the constants
// Note: use /wsi/FNCEWS40MTOM/ for CEWS transport
String uri = "http://ceserver:port/wsi/FNCEWS40MTOM/";
String username = "username";
String password = "password";

// Get the connection
Connection conn = Factory.Connection.getConnection(uri);

// Get the user context
UserContext uc = UserContext.get();

// Build the subject using the FileNetP8WSI stanza
// Use the FileNetP8WSI stanza for CEWS
uc.pushSubject(
    UserContext.createSubject(conn,username,password,"FileNetP8WSI" )
);

try
{
    // Get the default domain
    Domain domain = Factory.Domain.getInstance(conn, null);

    // Get an object store
    ObjectStore os = Factory.ObjectStore.fetchInstance(domain,
        "ObjectStoreName", null);
}
finally
{
    uc.popSubject();
}
```

Example 3-4   Getting the initial connection with EJB transport in Java

```
// Set the constants
// Use /FileNet/Engine for EJB
String uri = "http://ceserver:port/FileNet/Engine";
String username = "username";
String password = "password";

// Get the connection
Connection conn = Factory.Connection.getConnection(uri);
```

```
// Get the user context
UserContext uc = UserContext.get();

// Build the subject using the FileNetP8 stanza
// Use FileNetP8 for the EJB transport (also the default)
uc.pushSubject(
   UserContext.createSubject(conn,username,password,"FileNetP8" )
);

try
{
   // Get the default domain
   Domain domain = Factory.Domain.getInstance(conn, null);

   // Get an object store
   ObjectStore os = Factory.ObjectStore.fetchInstance(domain,
      "ObjectStoreName", null);
}
finally
{
   // Pop the subject off the UserContext stack
   uc.popSubject();
}
```

Note: There are other ways to handle authentication. All the API requires is a valid JAAS `Subject`. To use alternate authentication, work with your application server provider.

3.2.3 .NET

Unlike the Java API, the .NET API uses only the Web services transport for communicating with the CE. Making the initial connection to the CE and retrieving an object store involves the following steps:

1. Create the `UsernameToken` for authentication.
2. Pass the token to the `UserContext`.
3. Get the domain using `Factory.Domain.GetInstance()`.
4. Fetch the object store `Factory.ObjectStore.FetchInstance()`.

Example 3-5 on page 49 is adapted from:

`ITSO.AutoRental.DataAccess.ContentEngineConnector.Connect()`

It shows the process of getting the initial connection, retrieving the domain, and retrieving an object store in C#.

Example 3-5   Getting the initial connection in C#

```
public void Connect()
{
   string uri = "http://ceserver:port/wsi/FNCEWS40MTOM/"
   string username = "username";
   string password = "password";

   // Create a security token using the username and password
   UsernameToken token = new UsernameToken(username, password,
                         PasswordOption.SendPlainText);
   // Associate this UserContext with the whole process
   UserContext.SetProcessSecurityToken(token);

   // Now get the connection
   IConnection conn = Factory.Connection.GetConnection(uri);

   // Get the default domain
   _domain = Factory.Domain.GetInstance(conn, null);

   // Do the initial connection
   _os = Factory.ObjectStore.FetchInstance(_domain, objectStore, null);
}
```

3.3 Exception handling

When the CE signals an error, it throws a *runtime exception*. Within the API, there is only one type of exception: EngineRuntimeException. Instead of multiple exception types for the various error conditions available, the EngineRuntimeException carries with it an exception code that indicates the actual failure. Calling EngineRuntimeException.getExceptionCode() will return the exception code for the specific exception condition.

After an exception has been caught, it is up to the developer to determine if the application can recover from the exception or if the exception should be logged or displayed to the user to indicate that something failed.

A complete list of the exception codes is available in the JavaDoc and .NET DNA online reference as properties of ExceptionCode.

Example 3-6 and Example 3-7 on page 51 show an attempt to fetch an object and, in case of failure, attempt to create the object. This code has been adapted from the following method in the sample application Java code:

```
RentalActivityHandler.getRentalNumber()
```

Example 3-6 Exception handling with Java

```java
while (confirmationId == null)
{
   try
   {
      Integer temp;

      if( createObject )
         // Calls the create custom object code
         temp = createRentalNumber(dom,os);
      else
         // Fetches the custom object
         temp = getRentalNumber(os);

      confirmationId = temp;
   }
   catch( EngineRuntimeException ex)
   {
      ExceptionCode e = ex.getExceptionCode();

      if( e == ExceptionCode.E_OBJECT_NOT_FOUND )
      {
         // We can't find the requested object, so we should create it
         createObject = true;
         continue;
      }
      else if ( e == ExceptionCode.E_NOT_UNIQUE )
      {
         // We tried creating an object and it's already there, use it
         createObject = false;
         continue;
      }

      // We hit an error we can't handle, so rethrow the exception
      throw ex;
   }
}
```

Example 3-7   Exception handling in C#

```csharp
int confirmationId = -1;
bool createObject = false;

while( confirmationId == -1 )
{
   try
   {
      int temp;

      if( createObject )
         temp = CreateRentalNumber(dom,os);
      else
         temp = GetRentalNumber(os);

      confirmationId = temp;
   }
   catch( EngineRuntimeException ex)
   {
      ExceptionCode e = ex.GetExceptionCode();

      if( e == ExceptionCode.E_OBJECT_NOT_FOUND )
      {
         // We can't find the requested object, so we should create it
         createObject = true;
         continue;
      }
      else if ( e == ExceptionCode.E_NOT_UNIQUE )
      {
         // We tried creating an object and it's already there, use it
         createObject = false;
         continue;
      }

      // We've hit an error we can't handle, so rethrow the exception
      throw ex;
   }
}
```

3.4 Creating, retrieving, updating, and deleting objects

Within the CE, there are a number of common objects that an application will have to use. These include:

► Documents
► Folders
► Custom Objects

For these objects, there are a number of common needs in accessing these objects:

► Creating objects
► Working with properties
► Retrieving objects
► Deleting objects
► Retrieving content
► Working with property filters
► Working with versions (described in 4.2, "Versioning" on page 90.)

3.4.1 Pending actions

Within the CE API, all changes (creations, updates, and deletions) are treated as local changes only until the `.save()` (Java) or `.Save()` (.NET) method has been called. Instead of immediately saving the changes, the API creates a *pending action* describing the change. Changes to objects can be committed individually or as a batch. See 3.8, "Batching and batch execution" on page 81 for details about batch execution.

3.4.2 Creating objects

To create any object with the CE API:

1. Use the `Factory` convenience methods to create the object locally.

2. Apply any metadata locally to the newly created object. See 3.4.3, "Working with properties" on page 53 for details.

3. For objects that can have versions, optionally handle check-in.

4. Commit the object to the CE, creating the object in the CE.

Example 3-8 on page 53 and Example 3-9 on page 53 are adapted from the `ITSONewVehicleActivity.java` program and show how to create a `CustomObject` using both Java and C#.

Note: When creating objects, specifying the object ID for the object being created is possible by setting the value of the ID property. In addition, the ID class offers a method to create an IBM FileNet P8 object ID. Unless you have a specific reason for doing so, you should not explicitly specify the object ID. An ID will be created by the API or by the CE server.

Example 3-8   Creating objects in Java

```
CustomObject activity = Factory.CustomObject.createInstance(os,
"ITSOIdleActivity");

// Property values are set here

// Save the new IdleActivity
activity.save(RefreshMode.NO_REFRESH);
```

Example 3-9   Creating objects in C#

```
ICustomObject activity = Factory.CustomObject.CreateInstance(os,
"ITSOIdleActivity");

// Property values are set here

// Save the new IdleActivity
activity.Save(RefreshMode.NO_REFRESH);
```

3.4.3 Working with properties

Each object in the CE has *metadata*. This metadata can range from the date that the object was created to the title of a document to a binary large object (BLOB) of data. Metadata is stored and accessed using the *properties* of the given object.

The CE provides the following metadata types for use:

► Text
► 32-bit Integer
► 64-bit Floating Point
► Binary
► Boolean
► Date/Time
► GUID
► Object

Properties can also can be either *single-valued* or *multi-valued,* meaning the property can store either a single value or one or more values.

Object properties

Object-valued properties (OVP) work a bit differently than the other property types. For more information, see 4.3.1, "Object-valued properties" on page 94.

Properties collections

All CE objects have properties, or metadata, that describe that object. When an object has been retrieved with the API, these properties are stored locally in a *properties collection*. This collection holds property values for existing objects or stores new or updated values to be committed to the CE.

The primary property collection classes are `Properties` for Java and `IProperties` for .NET and are typically accessed with the `.getProperties()` method in Java or the `.Properties` property in .NET. Although the properties collection can be iterated or enumerated, a more common approach is to access a given property directly through the `.get(name)` method or `.Item[name]` property, which returns a `Property` object that represents that property and its value (or values).

Dynamically determining property type requires a cascade of `instanceof` in Java or `is` in .NET.

> **Note:** For clarity, the following code examples omit the save method call, so, as written, the examples would not commit any property changes.

Setting single-valued property value

To set the value of a single-valued property in Java:

1. Create it locally or retrieve the object.

2. Get the properties collection.

3. Add the value by using the `.putValue()` method.

Example 3-10 on page 55, adapted from the `ITSONewVehicleActivity.java` program, shows how to retrieve the properties of a document and to update the value of the ITSOStartDate property with the current date and time.

Example 3-10   Setting a single-valued property in Java

```
// Get the properties collection
Properties props = activity.getProperties();

// Get the start date
Date start = new Date(System.currentTimeMillis());

// Set the properties
props.putValue("ITSOStartDate", start);
```

The .NET API works slightly different from the Java API.

To set a single value property in .NET:

1. Create locally or retrieve the object.

2. Get the properties collection.

3. Set the value of the property by using the `properties["propertyname"]` convention.

> **Note:** For clarity, the following code examples omit the save method call, so, as written, the examples would not commit any property changes.

Example 3-11 shows how to retrieve the properties of a document and to update the value of the ITSOStartDate property with the current date and time.

Example 3-11   Getting and setting properties in C#

```
// Get the properties collection for a Document
IProperties properties = d.Properties;
Property["ITSOStartDate"] = DateTime.Now;
```

Setting multi-valued property values

To set a multi-valued property in Java:

1. Retrieve or create the object.

2. Set the value of the property to be a list using the `.putValue()` method.

> **Note:** Setting the value of a multi-valued object-valued property is not possible. For details, see 4.3.1, "Object-valued properties" on page 94.

Example 3-12 on page 56 shows how to retrieve the properties of a document, create a list of integer properties with two values, and to set the local value of the property "MutliValuedIntProperty" to be those values.

Example 3-12   Setting a multi-valued property in Java

```
// Get the properties collection for a Document
Properties properties = document.getProperties();

// Create a multi-value integer list
Integer32List list = Factory.Integer32List.createList();
list.add(100);
list.add(212);

// Add that to the properties for the document
properties.putValue("MutliValuedIntProperty",list);
```

To set a multi-valued property in .NET:

1. Create or retrieve the object.

2. Get the properties collection.

3. Create the value list.

4. Add the values.

5. Set the value of the property to be the list using the properties["name"] convention.

See Example 3-13.

Example 3-13   Setting multi-valued property in C#

```
IProperties properties = d.Properties;
// Create a multi-value integer list and add 2 values
IInteger32List list = Factory.Integer32List.CreateList();
list.Add(100);
list.Add(212);

property["MultiValuedIntProperty"] = list;
```

Retrieving property values

To retrieve property values in Java:

1. Retrieve the object.

2. Get the properties collection.

3. Get the value (or values) by using either of the following methods:

   ```
   .get<type>Value()
   .get<type>ValueList()
   ```

Example 3-14 shows, using Java, how to retrieve the value of a single-valued string property, `StringProperty`, and a multi-valued ID property, `RelatedIds`.

Example 3-14   Retrieving property values from a CE object using Java

```java
// Get the properties collection from a custom object
Properties properties = o.getProperties();

// Get a string value and print to the console
System.out.println( properties.getStringValue("StringProperty") );

// Get a multi-value ID property
IdList ids = properties.getIdListValue("RelatedIds");

// Iterate through the list of values, printing them to the console
for( Iterator i = ids.iterator(); i.hasNext(); )
{
    // Get the value and cast to an Id property type
    Id property = (Id)i.next();
    // Print the value to the console
    System.out.println(property.toString());
}
```

To retrieve property values in C#:

1. Retrieve the object.

2. Get the properties collection.

3. Get the value(s) using .Get<*type*>Value() or .Get<*type*>ValueList() (for example, `GetStringValue` or `GetInt32ListValue()`)

Example 3-15 shows how to retrieve the value of a single-valued string property, StringProperty, and a multi-valued ID property, `RelatedIds`.

Example 3-15   Retrieving property values from a CE object using C#

```csharp
// Get the properties for a custom object
Properties properties = o.Properties;

// Write out the string property value
Console.WriteLine("Property value: " +
    properties.GetStringValue("StringProperty"));

// Get a multi-valued ID property
IIdList ids = properties.GetIdListValue("RelatedIds");

// Loop through the properties
```

```
for( IEnumerator i = ids.GetEnumerator(); i.MoveNext(); )
{
   // Cast the current property to an Id property
   Id property = (Id)i.Current;
   // Print out the value to the console
   Console.WriteLine(property.ToString());
}
```

Properties in local cache

The CE APIs give you explicit control over whether or not any particular property is in the local cache. For example, when retrieving an object from the CE, having the result exclude some of the properties is possible. For details about how, see 3.4.7, "Working with property filters" on page 66. If a `PropertyFilter` has been applied during the `fetchInstance`, `refresh`, or `fetchProperties` calls, not all properties of an object will be available. Trying to get any of the missing properties will result in an `EngineRuntimeException` exception.

3.4.4 Retrieving objects

Objects from CE are typically retrieved by using the `Factory` methods `getInstance()` or `fetchInstance()`, depending on how you will use the instantiated object and whether the object must be used immediately. For the differences between the two, see 3.5, "getInstance() versus fetchInstance()" on page 70.

There are methods for both retrieval techniques using either an *object ID* or a *path* as a document retrieval identifier. In addition, the API can be directed whether to retrieve all properties or a subset of the properties for the requested object. See 3.4.7, "Working with property filters" on page 66 for the details about using property filters.

Example 3-16 on page 59 and Example 3-17 on page 59 show how to retrieve documents, folders, and custom objects both by ID and by path.

> **Note:** The following examples all do an immediate fetch with no property filter for demonstration purposes. This approach allows for immediate usage of the objects. However, it may might be the best for performance. Review 3.5, "getInstance() versus fetchInstance()" on page 70 to understand when to get and when to fetch.

Example 3-16   Fetching objects in Java

```
// Retrieve document by path
Document d1 = Factory.Document.fetchInstance(os,"/path/to/doc",null);
// Retrieve document by ID
Document d2 =
   Factory.Document.fetchInstance(os,
      new Id("{C6E0130B-F679-4244-9269-CCA073AE31C9}"),null);

// Retrieve folder by path
Folder f1 = Factory.Folder.fetchInstance(os,"/path/to/folder",null);
// Retrieve folder by ID
Folder f2 =
   Factory.Folder.fetchInstance(os,
      new Id("{C6E0130B-F679-4244-9269-CCA073AE31C9}"),null);

// Retrieve custom object by path
CustomObject o1 =
      Factory.CustomObject.fetchInstance(os,"/path/to/object", null);
// Retrieve custom object by ID
CustomObject o2 =
   Factory.CustomObject.fetchInstance(os,
      new Id("{C6E0130B-F679-4244-9269-CCA073AE31C9}"),null);
```

Example 3-17   Fetching objects in C#

```
// Retrieve by path
IDocument d1 = Factory.Document.FetchInstance(os,"/path/to/doc",null);
// Retrieve by ID
IDocument d2 =
   Factory.Document.FetchInstance(os,
      new Id("{C6E0130B-F679-4244-9269-CCA073AE31C9}"),null);

// Retrieve by path
IFolder f1 = Factory.Folder.FetchInstance(os, "/path/to/folder", null);
// Retrieve by ID
IFolder f2 =
   Factory.Folder.FetchInstance(os,
      new Id("{C6E0130B-F679-4244-9269-CCA073AE31C9}"),null);

// Retrieve by path
ICustomObject o1 =
      Factory.CustomObject.FetchInstance(os, "/path/to/object", null);
// Retrieve by ID
ICustomObject o2 =
   Factory.CustomObject.FetchInstance(os,
      new Id("{C6E0130B-F679-4244-9269-CCA073AE31C9}"),null);
```

3.4.5 Deleting objects

An object can be deleted from the CE by using the `.delete()` method (in Java) and the `.Delete()` method (in .NET). Deleting an object removes both the metadata and associated content (in the case of documents and annotations).

> **Note:** Versionable objects have special behaviors related to deletion. For more information, see 4.2, "Versioning" on page 90.

To delete an object from the CE:

1. Be sure that the object is instantiated first (with either `getInstance` or `fetchInstance`).
2. Call the `delete` method to add the delete operation to the object's PendingActions.
3. Call the `save` method to perform the delete.

Example 3-18 and Example 3-19 on page 61 show how to instantiate and delete documents, folders, and custom objects.

> **Note:** In the following examples, all objects are created locally by using a `getInstance` call instead of `fetching`. This approach cuts the number of round-trips in half, improving performance, and reducing network utilization.

Example 3-18   Deleting objects in Java

```
// Get the object
Document doc = Factory.Document.getInstance(os,"Document","/Doc");
// Mark the document for deletion locally
doc.delete();
// Perform the delete
doc.save(Refresh.NO_REFRESH);

// Get the folder for deletion
Folder folder = Factory.Folder.getInstance(os,"Folder","/Folder");
// Prepare the folder for deletion
f.delete();
// Commit the delete
f.save(RefreshMode.NO_REFRESH);

// Get the object
CustomObject o =
    Factory.CustomObject.getInstance(os,"CustomObject","/Object");
```

```
// Prepare the object for deletion
o.delete();
// Commit the deletion
o.save(RefreshMode.NO_REFRESH);
```

Example 3-19   Deleting objects in C#

```
// Get the object
IDocument doc = Factory.Document.GetInstance(os,"Document","/Doc");
// Mark the document for deletion locally
doc.Delete();
// Perform the delete
doc.Save(RefreshMode.NO_REFRESH);
// Get the folder for deletion
IFolder folder = Factory.Folder.GetInstance(os,"Folder","/Folder");
// Prepare the folder for deletion
f.Delete();
// Commit the delete
f.Save(RefreshMode.NO_REFRESH);
// Get the object
ICustomObject o =
Factory.CustomObject.GetInstance(os,"CustomObject","/Object");
// Prepare the object for deletion
o.Delete();
// Commit the deletion
o.Save(RefreshMode.NO_REFRESH);
```

3.4.6 Retrieving content

When working with objects, it will likely be necessary to access the content
directly. Retrieving the content allows the application to use or save the binary (or
text) content of the object for processing elsewhere.

> **Note:** See 3.7, "Viewing documents" on page 78 for more information about
> simply viewing the object if that behavior is necessary.

The CE allows a single object to have multiple *content elements*. This approach
allows for operations, such as storing a multi-page scanned document as a
series of individual files, one for each page, instead of having to store a single file
with multiple pages. From a performance and network usage standpoint, this can
reduce the load on the network by allowing only the desired page to be
transferred instead of having to transfer the complete multi-page file just to
access a single page.

Finally, the CE stores a few metadata items for each content element attached, stored as a `ContentTransfer`. This metadata includes:

► Content size: Numerical size in bytes for the content element

► Content type: Content type of the element in MIME format

► Retrieval name: Optional property that indicates the original file name of the element

► Element sequence number: Ordered index for the content element indicating which element it is

See "Using ContentElement objects" on page 63 for details about using ContentElement objects.

Retrieving content using Document object

To retrieve content from an object:

1. Retrieve the object.

2. Get the `InputStream` (Java) or `Stream` (.NET) for the desired content element by using `.accessContentStream()` (Java) or `.AccessContentStream()` (.NET).

3. Use the `InputStream.read()` (Java) or `Stream.Read()` (.NET) methods to retrieve the binary content.

Example 3-20 and Example 3-21 on page 63, adapted from `ContentEngineConnector.cs` from the .NET Kiosk application, show how to retrieve a document and save its content to a file. In this case, we know we are dealing with a short, line-oriented text file. In the more general case, you would want to treat the content as an arbitrary stream of bytes.

Example 3-20   Directly accessing a document's content in Java

```
InputStream stream = document.accessContentStream(0);

BufferedReader reader =
                    new BufferedReader(new InputStreamReader(stream));

String text = "", line;

while ((line = reader.readLine()) != null)
   text += line;

reader.close();
```

Example 3-21   Directly accessing a document's content in C#

```
Stream stream = document.AccessContentStream(0);

TextReader reader = new StreamReader(stream);

string text = "", line;

while ((line = reader.ReadLine()) != null)
   text += line;

reader.Close();
```

Using ContentElement objects

To retrieve the content collection and associated `ContentElement`s:

1. Retrieve the object.
2. Call the `get_ContentElements()` method (Java) or `.ContentElements` property (.NET).
3. Access the appropriate content element using the `.get()` method (Java) or `.Items[]` property (.NET).
4. Optionally, iterate through the content elements by using the `.iterator()` (Java) or `.GetEnumeration()` (.NET) methods.

Example 3-22 and Example 3-23 on page 65 show how to retrieve a document, loop through the content elements, and save each content element to the file system as `"element#"`.

Example 3-22   Working with content elements in Java

```
public static void main(String[] args)
{
   // Authentication and connection goes here (left out for space)
   // Retrieve the document
   Document d = Factory.Document.fetchInstance(os,"/document",null);

   // Get the content elements
   ContentElementList elements = d.get_ContentElements();

   // Grab the first content element
   ContentTransfer element = (ContentTransfer)elements.get(0);

   String filename = element.get_RetrievalName();
   InputStream stream = element.accessContentStream();
```

```java
      Double size = writeContent(stream,filename);
      Double expected = element.get_ContentSize()

      if( size != expected )
         System.err.println("Invalid content size retrieved");

      d = Factory.Document.fetchInstance("/document2",null);
      elements = d.get_ContentElements();

      for( Iterator i = elements.iterator(); i.hasNext(); )
      {
         element = (ContentTransfer)i.next();
         writeContent(
            element.accessContentStream(),
            "element" + element.get_ElementSequenceNumber()
         );
      }
   }

   public static Double writeContent(InputStream s, String filename)
      throws IOException
   {
      // Open a buffered output stream to write the content to
      // a file named the same as the retrieval name
      BufferedOutputStream writer = new BufferedOutputStream(
         new FileOutputStream(filename) );

      Double size = new Double(0);
      int bufferSize;
      byte[] buffer = new byte[1024];

      // Loop through the content and write it to the system
      while( ( bufferSize = s.read(buffer) ) != -1 )
      {
         size += bufferSize;
         writer.write(buffer, 0, bufferSize);
      }

      writer.close();
      s.close();

      return size;
   }
```

Example 3-23   Working with content elements in C#

```csharp
static void Main(string[] args)
{
   // Authenticate and connect (left out for space)

   // Retrieve the document
   IDocument d = Factory.Document.FetchInstance(os,"/document", null);

   // Get the content elements
   IContentElementList elements = d.ContentElements;

   // Grab the first content element
   IContentTransfer element = (IContentTransfer)elements[0];

   // Grab the retrieval name
   string filename = element.RetrievalName;

   // Get the content stream
   Stream stream = element.AccessContentStream();

   // Write the content and compare the size expected
   double size = writeContent(stream,filename);

   if( size != element.ContentSize )
      Console.Error.WriteLine("Invalid content size retrieved");

   // Fetch second document
   d = Factory.Document.FetchInstance(os,"/document2",null);

   // Grab the content elements
   elements = d.ContentElements;

   // Iterate through each element and write the content
   for( IEnumerator i = elements.GetEnumerator(); i.MoveNext(); )
   {
      element = (IContentTransfer)i.Current;
      writeContent(
         element.AccessContentStream(),
         "element" + element.ElementSequenceNumber
      );
   }
}

// Convenience method that writes the content to the given file
```

```
public static double writeContent(Stream stream, String filename)
{
   byte[] buffer = new byte[4096];
   int bufferSize;
   double size = 0;

   // Open a binary writer for the stream
   BinaryWriter writer = new BinaryWriter(
      File.Open(filename, FileMode.Create));

   // Write the content
   while ((bufferSize = stream.Read(buffer, 0, buffer.Length)) != 0)
   {
      size += bufferSize;
      writer.Write(buffer, 0, bufferSize);
   }

   // Close the streams
   writer.Close();
   stream.Close();
   return size;
}
```

3.4.7 Working with property filters

Property filters are optional parameters to a number of methods that fetch
objects or properties from the CE. They allow highly granular control of the
objects or properties being returned.

Property filters can be used two ways to help boost performance:

► Limiting the number of properties to be retrieved from the CE, reducing the
  volume of data transmitted.

► Including more properties through recursion, reducing the total number of
  requests to the CE. Recursion implicitly follows chains of object-valued
  properties.

Limiting requested properties

When fetching an object from the CE, it is possible to have the API retrieve all
properties for the request object. However, if the application only needs a few
properties to complete a procedure, fetching all properties wastes time and
network resources. Instead, limiting the data returned to only the applicable
properties will make the call and application more efficient.

To limit data returned using property filters:

1. Directly fetch the properties using the `.fetchProperties()` (Java) or `.FetchProperties()` (.NET) methods.

2. Create a `PropertyFilter` object.

3. Set the include and exclude properties.

4. Set the recursion to determine how far to follow object properties.

5. Pass the object to a `.fetchInstance()` (Java) or `.FetchInstance()` (.NET) call.

> **Note:** Trying to use a property that has not been retrieved will result in an exception. If you do not use property filters, the API generally manages the retrieval of additional properties. This process avoids the exceptions but can cost extra round-trips to the server. When you use a property filter, you get explicit control over the presence or absence of all properties.

Example 3-24 and Example 3-25 on page 68 show how to use property filters to control which properties are retrieved when fetching a document and fetching a document's properties.

Example 3-24   Limiting properties returned using property filters in Java

```
// Get the document
Document d = Factory.Document.getInstance(os, "Document", "/Doc1");

// Fetch properties DocumentTitle and DateCreated
d.fetchProperties(new String[] { "DocumentTitle", "DateCreated" });

// Write out the Document Title
System.out.println(d.getProperties().getStringValue("DocumentTitle"));

// Write out the Creator (throws EngineRuntimeException)
//System.out.println(d.getProperties().getStringValue("Creator"));

// Same result using PropertyFilter
PropertyFilter filter = new PropertyFilter();

filter.addIncludeProperty(
        new FilterElement(null, null, null,
            "DocumentTitle DateCreated"));

// Calling fetch instance instead of getInstance
d = Factory.Document.fetchInstance(os, "/Doc2", filter);
```

Example 3-25   Limiting properties returned using property filters in C#

```csharp
// Retrieve the document
IDocument d = Factory.Document.GetInstance(os, "Document", "/Doc1");

// Fetch Document Title and Date Created
d.FetchProperties(new string[] { "DocumentTitle", "DateCreated" });

// Write out the DocumentTitle (works)
Console.WriteLine(d.Properties.GetStringValue("DocumentTitle"));

// Write out the Creator (throws EngineRuntimeException)
//Console.WriteLine(d.Properties.GetStringValue("Creator"));

// Same result using PropertyFilter
PropertyFilter filter = new PropertyFilter();

filter.AddIncludeProperty(
    new FilterElement(null, null, null,
    "DocumentTitle DateCreated"));

// Calling fetch instance instead of get instance
d = Factory.Document.FetchInstance(os, "/Doc2", filter);
```

Using property filters to eliminate round trips

Property filters can also be used to return more data than normal. This approach is especially helpful when retrieving an object that has other objects as the values of properties. Instead of retrieving the object itself in one call and then the object values in multiple additional calls, adding a property filter to do recursion can cause all the objects to be returned in the initial call.

For example, an application might need to retrieve not only a document, but also information about all annotations on that document. If a document has two annotations, normally this call would take three round-trips to the CE: one to fetch the document and potentially one to fetch each annotation. With an appropriately configured property filter, the retrieval can happen in a single round-trip.

Example 3-26 on page 69 and Example 3-27 on page 69 both demonstrate using property filters, and are adapted from `GetRentalRecord` in `ContentEngineConnector.cs` in the sample .NET application.

Example 3-26   Property filter in Java

```
SearchSQL sql = new SearchSQL(
   "select " +
      "ITSOCustomer, ITSOStartDate, ITSOEndDate, ITSOChargesNet," +
      "ITSOChargesTotal, ITSORentalAgreement ITSOVehicle," +
      "ITSODailyRate, ITSOConfirmationId "
   "from " +
      "ITSORentalActivity " +
   "where " +
      "ITSOConfirmationId = " + identifier);

// Search the instantiated object store
SearchScope scope = new SearchScope(os);

// Create a property filter object
PropertyFilter filters = new PropertyFilter();

filters.addIncludeProperty(2, null, null,
   "ContentElements Containees ITSOVehicleId ITSOVehicleYear " +
   "ITSOVehicleMake ITSOVehicleColor ITSOCustomerName");

IndependentObjectSet set = scope.fetchObjects(sql, 1, filters, false);
```

Example 3-27   Property filters in .NET

```
SearchSQL sql = new SearchSQL(
   "select " +
      "ITSOCustomer, ITSOStartDate, ITSOEndDate, ITSOChargesNet," +
      "ITSOChargesTotal, ITSORentalAgreement ITSOVehicle," +
      "ITSODailyRate, ITSOConfirmationId " +
   "from " +
      "ITSORentalActivity " +
   "where " +
      "ITSOConfirmationId = " + identifier);

SearchScope scope = new SearchScope(_os);
PropertyFilter filters = new PropertyFilter();

filters.AddIncludeProperty(2, null, null,
   "ContentElements Containees ITSOVehicleId ITSOVehicleYear " +
   "ITSOVehicleMake ITSOVehicleColor ITSOCustomerName");

IIndependentObjectSet set = scope.FetchObjects(sql, 1, filters, false);
```

Without property filters, the operation in the examples would require four round-trip calls to the CE, as follows:

- ► Execute the search and retrieve the ITSORentalActivity custom object.
- ► Retrieve the ITSOVehicle folder properties.
- ► Retrieve the ITSORentalAgreement properties.
- ► Retrieve the ITSOCustomer properties.

Using the code examples, the total number of round-trips to the CE is reduced from four to one. Figure 3-2 shows the objects and properties that are returned from the code in the examples.

ITSORentalActivity	
ITSOCustomer	{4B608AFB-E6EA-4e76-A2CB-71543EAE5789}
ITSOStartDate	2009-11-11 09:00:00
ITSOEndDate	2009-12-12 09:00:00
ITSOChargesNet	100000
ITSOChargesTotal	120000
ITSORentalAgreement	{35F89870-64A4-4e75-A71B-596E7215473F}
ITSOVehicle	{C4D27FCE-13A7-405c-8BDD-7EF09FB67251}
ITSODailyRate	25000
ITSOConfirmationID	1009792649

ITSOCustomer	
Id	{4B608AFB-E6EA-4e76-A2CB-71543EAE5789}
ITSOUserId	<not retrieved>
ITSOCustomerName	John Q. Renter

ITSORentalAgreement	
Id	{35F89870-64A4-4e75-A71B-596E7215473F}
ContentElements	Agreement.txt
DocumentTitle	<not retrieved>

ITSOVehicle	
Id	{C4D27FCE-13A7-405c-8BDD-7EF09FB67251}
ITSOVehicleID	1F786HH76DCC1
ITSOVehicleYear	1975
ITSOVehicleMake	Dacia
ITSOVehicleModel	1310
ITSOVehicleColor	Blue
Containees	{E407D1FC-EF11-48bc-A977-55DF8A9A9B60}
FolderName	<not retrieved>

Figure 3-2   Objects and properties returned

3.5 getInstance() versus fetchInstance()

The API offers two methods for instantiating CE objects: `getInstance()` and `fetchInstance()`.

getInstance() and GetInstance()

A `getInstance()` (Java) or `GetInstance()` (.NET) call takes an object store, class, and retrieval identifier and creates locally a representation of the requested object, whether or not it actually exists. After the call, the object is immediately available locally to use, however no properties or content is available until one of the following methods is called:

- ► `.refresh()` or `.Refresh()`
- ► `.fetchProperties()` or `.FetchProperties()`

A `getInstance()` call is useful for batching, or for calls where the properties are not necessary, just the action upon the object (such as deleting an object). Because there is no initial round trip to the server to fetch the object, this is also called *fetchless instantiation*.

fetchInstance() and FetchInstance()

A `fetchInstance()` (Java) or `FetchInstance()` (.NET) call results in an immediate round-trip to the CE, retrieving any requested properties or all object properties, depending on the value of the property filter. The result is a local object that contains properties that can be immediately queried and modified as necessary. The downside is that the call to the CE is immediate, guaranteeing the network round-trip.

Deciding which to use

The advantages and disadvantages to get and fetch methods are:

▶ You get a performance improvement by using fetchless instantiation, but the corresponding cost is that your application's error handling might have to deal with missing objects at a later stage. In practice, that error-handling cost is not too severe.

▶ Another cost is the loss of *first writer wins* protection in the CE. Fetchless instantiation explicitly ignores any other updates to the same object. With a fetched object, the update sequence number is checked for consistency by the CE before any changes are committed for that object.

▶ If you actually need to get the values of any properties of an object, at least one round trip to the CE server will occur, in any case. This approach erases the performance advantage of fetchless instantiation.

3.6 Querying

When developing for the CE, one of the primary ways of finding documents is to do a search, based on data from a user. Workplace and WorkplaceXT both offer search templates and stored searches that a user can execute.

When you program, doing the search directly in code by using the SearchScope and SearchSQL classes is necessary. Searching the CE requires the following steps:

1. Create the SQL statement.
2. Set the SearchScope.
3. Execute the search.
4. Work with the results.

3.6.1 SearchSQL

The CE offers a language for executing queries within single or multiple object stores. The SearchSQL class offers helper methods to assist you in constructing a SQL statement, or the SQL statement can be constructed independently and passed to a SearchSQL instance as a string.

SQL statements must follow the CE SQL syntax, which generally conforms to SQL-92, with extensions for FileNet-specific constructs. The complete SQL syntax guide is available in the IBM FileNet P8 Platform documentation by selecting **ECM Help** → **Developer Help** → **Content Engine Development** → **Reference** → **SQL Syntax Descriptions**.

Building a query with SearchSQL

The CE API has a SearchSQL object with a number of convenience methods for creating the search SQL. In addition, the SearchSQL object can be instantiated with an actual SQL statement. Example 3-28 and Example 3-29 show an example of building a query with SearchSQL.

Example 3-28   Building a query with SearchSQL in Java

```
SearchSQL sqlObject = new SearchSQL("SELECT DocumentTitle, Id
FROM Document WHERE Creator = 'jsmith'");
```

Example 3-29   Building a query with SearchSQL in C#

```
SearchSQL sqlObject = new SearchSQL("SELECT DocumentTitle, Id
FROM Document WHERE Creator = 'jsmith'");
```

> **Note:** The helper methods on SearchSQL and a SearchSQL object that are instantiated with a SQL statement are mutually-exclusive. The SQL passed in (on creation) cannot be updated with the convenience methods. Experienced developers typically find that supplying the SQL as a string is the simpler approach.

Using the IBM FileNet Enterprise Manager to build SQL

When you initially build the SQL, use the Enterprise Manager's Query Builder tool to construct your query or to quickly validate that it works as intended. After the query has been constructed, it can be copied to the code and passed to the SearchSQL object.

To create SQL by using the Enterprise Manager:

1. Select the target Object Store.

2. Right-click the Search Results node.

3. Select new search.

4. Build the search in the Content Engine Query Builder.

5. Select **View → SQL View**.

Figure 3-3 shows how to use the Enterprise Manager's Content Engine Query Builder to create the search SQL for use in a SearchSQL object.
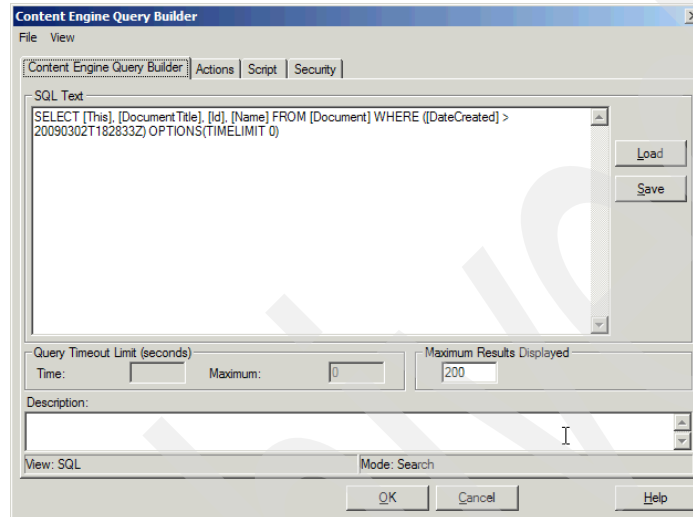
Figure 3-3   Compiled SQL statement from Content Engine Query Builder

Executing the query and working with the results

After a search has been executed, the SearchScope returns an `IndependentObjectSet` or `RepositoryRowSet` of the returned items, depending on whether you fetch objects or rows, respectively. See Example 3-30 and Example 3-31 on page 74.

Example 3-30   Search example in Java

```
// Construct the sql statement
SearchSQL sql = new SearchSQL(
   "select ISTOStartDate, ITSOEndDate, ITSOVehicle " +
   "from ITSOIdleActivity " +
   "where " +
   "ITSOVehicle = OBJECT('{D5DC8C04-2625-496f-A280-D791AFE87A73}') " +
   "AND ITSOStartDate < 20090801T000000Z OR " +
   "ITSOEndDate > 20090701T000000Z" );

// Search the object store with the sql and get the returned items
SearchScope scope = new SearchScope(os);
```

```
IndependentObjectSet set = scope.fetchObjects(sql, null, null, false);

// Loop through the returned results
for( Iterator i = set.iterator(); i.hasNext(); )
{
   // Get the document
   CustomObject obj = (CustomObject)i.next();

   // Code to work with object goes here
}
```

Example 3-31   Search example in C#

```
// Construct the sql statement
SearchSQL sql = new SearchSQL(
   "select ITSOStartDate, ITSOEndDate, ITSOVehicle " +
   "from ITSOIdleActivity " +
   "where " +
   "ITSOVehicle = OBJECT('{D5DC8C04-2625-496f-A280-D791AFE87A73}') " +
   "AND ITSOStartDate < 20090801T000000Z OR " +
   "ITSOEndDate > 20090701T000000Z" );

// Search the object store with the sql and get the returned items
SearchScope scope = new SearchScope(os);

IIndependentObjectSet set = scope.FetchObjects(sql, null, null, false);
IEnumerator e = set.GetEnumerator();

while(e.MoveNext())
{
   // Get the current object
   ICustomObject obj = (ICustomObject)e.Current;

   // Code to work with object goes here
}
```

3.6.2 Search scope

When you execute a search, the scope of the search must be defined. This scope can be either a single object store or the search scope can be expanded to include multiple object stores. Methods on the search scope define the type of objects to be returned.

The SearchScope methods execute the SQL statement on one or more object stores to find objects (IndependentObject instances), database rows (RepositoryRow instances), or metadata (ClassDescription instances). If the query includes a JOIN operator, you must fetch rows instead of objects.

Cross object store searching

When doing a cross-object store search (also called a *merged scope* search), the search results are merged based on the merge mode. The merge mode can be either:

► INTERSECTION: The search results will contain the classes occurring in all repositories searched.

► UNION: The search results will contain the classes occurring in any repository searched.

When the merge mode is UNION and a class or property is not found in any repository, the following occurs:

► For classes, an INNER JOIN returns no rows, and an OUTER JOIN returns nulls. JOIN types are specified in the JoinOperator class

► For properties, the property value is null in a selection list or WHERE clause, and is omitted from an ORDER BY clause.

To search across multiple object stores:

1. Create an array of object stores to search.

2. Create the search scope with the array of object stores to search and the merge mode.

3. Create the SQL.

4. Execute the search.

5. Work with the results.

See Example 3-32 and Example 3-33 on page 76.

Example 3-32   Setting the scope to be multiple object stores in Java

```
// Create the object store array
ObjectStore[] osArray = new ObjectStore[]{os1,os2};

// Create the search scope
SearchScope objStores = new SearchScope(osArray,
MergeMode.INTERSECTION);
```

Example 3-33   Setting the scope to be multiple object stores in C#

```
// Create the object store array
IObjectStore[] osArray = new IObjectStore[] { os1, os2 };
// Create the search scope
SearchScope objStores = new SearchScope(osArray,
MergeMode.INTERSECTION);
```

3.6.3 Content searches

Full-text searches (also known as *content-based retrieval* (CBR)) can be used to search for words or phrases that are part of object content or that occur in string properties of these objects. For the content in an object or its string properties to be searched, you must enable CBR for the object class and any of its string properties to be included in a content search.

A content search is indicated by either a CONTAINS or FREETEXT operator in the SQL statement contained in SearchSQL. The CONTAINS and FREETEXT operators have somewhat different operand formats and provide somewhat different search characteristics. Although the CONTAINS operator can search content in all properties, or in a single property, the FREETEXT operator can search content only in all properties. Note that attempting to specify a property name for FREETEXT will generate an exception. The CONTAINS operator is generally preferable to the FREETEXT operator.

For more information about the CONTAINS and FREETEXT operators, go to **ECM Help → Developer Help → Content Engine Development → Reference → SQL Syntax Descriptions → Full-Text Queries**.

> **Note:** Full-text queries can have unexpected performance impact. You should experiment with CBR queries on realistic data sets to ensure that you have taken performance into account.

3.6.4 Paging support

When working with a large result set, returning too many items at once can cause memory usage problems. The API allows for breaking the result set into groups of items called *pages*. In this way, an application can request and work with a smaller subset of items at a time. You indicate that you want paged results by making your query continuable.

Example 3-34 on page 77 and Example 3-35 on page 77 show how to page through a result in Java and .NET.

Example 3-34   Paging in Java

```java
// A query that will return a large result set
SearchSQL sql = new SearchSQL("select [DocumentTitle] from document");

SearchScope scope = new SearchScope(os);

// Set the paging to be 50 items per page and enable continuation
IndependentObjectSet s = scope.fetchObjects(sql, 50, null, true);

// Get the page iterator
PageIterator p = s.pageIterator();

// Loop through each page
while(p.nextPage())
{
    // Loop through each item in the page
    for(Object obj : p.getCurrentPage())
    {
        // Get the document object and write Document Title
        Document doc = (Document)obj;
        System.out.println(
                doc.getProperties().getStringValue("DocumentTitle"));
    }
}
```

Example 3-35   Paging in C#

```csharp
SearchSQL sql = new SearchSQL("select [DocumentTitle] from document");

SearchScope scope = new SearchScope(os);

IIndependentObjectSet s = scope.FetchObjects(sql, 50, null, true);

IPageEnumerator p = s.GetPageEnumerator();

while(p.NextPage())
{
    Console.WriteLine(p.ElementCount);

    foreach(object obj in p.CurrentPage )
    {
        IDocument doc = (IDocument)obj;
        Console.WriteLine(
                doc.Properties.GetStringValue("DocumentTitle"));
    }
}
```

3.7 Viewing documents

The CE can store a wide variety of document file types, including images. For image file types that users need to view, applications can choose to retrieve the image content and display it in their preferred viewer application, or they can use WorkplaceXT's Java viewer. The supported image file types for WorkplaceXT's Java Viewer can be found in **ECM Help** → **User Help** → **Actions, preferences, and tools** → **Work with Image Viewer**.

For applications that choose to use the WorkplaceXT Java viewer applet, the supported way to instantiate the viewer is to obtain a user token and call the getContent servlet from WorkplaceXT. Using the Java viewer applet directly with a separate application is not supported.

Example 3-36 and Example 3-37 on page 79 show the code samples for calling the WorkplaceXT Java viewer applet.

Example 3-36   Document viewing in Java and JSP

```
<%@ page import="java.util.*, java.io.*, java.net.*"%>
<html>
<body>

<%

        String baseP8URL = "http://aeServer:9080/WorkplaceXT/";
        String user = "user";
        String password = "password";
        String objectStore = "ObjectStoreName";
        String docID = "{C9712786-4B17-4512-9E13-9F4154B35FC2}";


        // Call WorkplaceXT's setCredentials servlet to obtain user token
        URL url = new URL(baseP8URL +
                    "setCredentials?op=getUserToken&userId="
                    + user + "&password=" + password + "&verify=true");


        HttpURLConnection p8con =
            (HttpURLConnection) url.openConnection();

        p8con.setRequestMethod("POST");
        p8con.connect();
        InputStream in = p8con.getInputStream();
        int c = -1;
        String tempUserToken = "";
        while ( (c=in.read()) >= 0) tempUserToken +=
                new Character((char)c).toString();
```

```
            String userToken = URLEncoder.encode(tempUserToken, "UTF-8");

            // Build URL to getContent servlet
            String contentUrl = baseP8URL + "getContent?objectStoreName=" +
                        objectStore + "&id=" + docID +
                        "&objectType=document&ut=" + userToken +
                        "&impersonate=true";
%>

<script language="javascript">
  window.top.location="<%=contentUrl%>"
</script>


</body>
</html>
```

Example 3-37   Document viewing in C# and ASP.NET

```
using System.Net;
using System.IO;

public partial class _Default : System.Web.UI.Page
{
    public string WorkplaceRootUrl = "";
    public string ContentUrl = "";
    public string Username = "";

    private void Page_Load(object sender, System.EventArgs e)
    {

        // set constant values
        // replace these values with values applicable to your site
        WorkplaceRootUrl = "http://aeServer:9080/WorkplaceXT/";
        Username = "user";
        string pwd = "password";
        string objectStoreName = "ObjectStoreName";
        string Id = "{C9712786-4B17-4512-9E13-9F4154B35FC2}";


      // Call WorkplaceXT's setCredentials servlet to obtain user token
        string UserToken = getCEUserToken(WorkplaceRootUrl, Username,  pwd);

        // create URLs for the JavaViewer
          ContentUrl = WorkplaceRootUrl + "getContent?objectStoreName=" +
objectStoreName + "&id=" + Id + "&objectType=document&ut=" + UserToken +
"&impersonate=true";

}
```

```
            private string getCEUserToken(string baseURL, string uid, string pwd)
        {
            string UserToken = "";

            // make the request and get the response
            WebRequest request = WebRequest.Create(baseURL +
"setCredentials?op=getUserToken&userId=" + uid + "&password=" + pwd +
"&verify=true");
              request.Method = "POST";
            WebResponse response = request.GetResponse();

            // read the response from the stream into a byte[]
            Stream stream = response.GetResponseStream();
            byte[] token = new byte[response.ContentLength];
            stream.Read(token, 0, (int)response.ContentLength);
            response.Close();

            // and convert the bytes in the array into a string.
            foreach (byte chr in token)
                UserToken += System.Convert.ToChar(chr);

            // return the encoded string
            return Server.UrlEncode(UserToken);
        }
}


// Below is the aspx page source

<html>
<body>

        <script language="javascript">
                window.top.location="<%=ContentUrl%>"
        </script>
</body>
</html>
```

3.7.1 User tokens

The code examples in Example 3-36 on page 78 and Example 3-37 on page 79
for viewing documents use user tokens obtained from WorkplaceXT. The method
used in those examples is an sample of how a custom application that does not
have access to IBM FileNet API libraries can obtain a user token through an
HTTP URL call by calling the WorkplaceXT setCredentials servlet.

User tokens can also be obtained in other ways and are described in detail in **ECM Help → Developer Help → Workplace Development → Workplace Customization Guide → User Tokens**.

3.8 Batching and batch execution

To make more efficient use of network resources and to allow for grouping the related operations into a single work unit, the CE API offers batching capabilities. A *batch* accumulates and packages multiple operations (method calls) on objects. The batch is then executed in a single operation. Whether a batch is a transactional operation depends on its type:

► Batch updates type

This type of batch operation creates, updates, or deletes persisted objects, and is executed transactionally. `IndependentlyPersistableObject` references are accumulated, and an instance of the `UpdatingBatch` class is executed as a single transaction (the `updateBatch` method). The batch execution does not return a value: all of the pending commits succeed or all fail. For the failure case the transaction is rolled back and an exception is thrown.

► Batch retrieval type

This type of batch operation retrieves independent objects, and is not executed transactionally. `IndependentObject` references are accumulated, and each included object then is either refreshed (retrieved) or gets its own exception. As for single-operation object saves and retrievals, any changes to the retrieved objects are done in place, so the existing `IndependentObject` references continue to be valid and reflect the changes.

> **Note:** A batch can significantly improve performance. Use batching when application logic lends itself to executing a series of operations that can be completed (or be in progress) independently, without reliance on either the state of another object included in the batch, or the result of another operation in the batch.

Refresh versus no refresh

When committing operations and batches to the CE, the operations have an option to do a refresh or no refresh on completion. Setting this to `RefreshMode.REFRESH` causes all local properties to be reloaded from the CE, providing the most recent values for a given object. `Refresh.NO_REFRESH` indicates to the API that no additional information, other than success and failure, should be returned from the call.

Selecting the right mode can affect performance of operations or the work unit as a whole. For example, when committing a group of documents to the CE, you might only want to know whether the commit failed or is successful. So doing a refresh in this situation is not necessary, and would cause the entire operation to be slower.

However, if the code has to work with the committed object, doing a refresh on commit operation can save a round-trip to the CE to fetch any updated properties (such as the ID of a newly created object).

Batch updates

You can perform bulk updating of objects in a batch by using `UpdatingBatch`.

To update objects in batch:

1. Create the `UpdatingBatch`.

2. Retrieve the objects to update.

3. Apply the changes to the local objects (but do not `.save()` the changes).

4. Add the objects to the batch.

5. Update the batch.

Example 3-38 and Example 3-39 on page 83, adapted from `RentalActivityHandler.insertVehicleActivity`, show how to update an existing object and create a second object in a single batch operation.

Example 3-38   Creating a batch of update actions in Java

```
// Get the IdleActivity record (from a previous search)
CustomObject previous = (CustomObject)set.iterator().next();

// Create the next custom object
CustomObject next =
    Factory.CustomObject.createInstance(os,"ITSOIdleActivity");

// Set the properties
Properties props = next.getProperties();
props.putValue("ITSOVehicle", vehicle);
props.putValue("ITSOStartDate", end);

// Update the previous object
previous.getProperties().putValue("ITSOEndDate", start);

// Create the batch
UpdatingBatch batch = UpdatingBatch.createUpdatingBatchInstance(domain,
```

```
                        RefreshMode.REFRESH);

// Add the objects
batch.add(previous, null);
batch.add(next, null);

// Execute the batch
batch.updateBatch();
```

Example 3-39   Creating a batch of update actions in C#

```
// Get the IdleActivity record (from a previous search)
ICustomObject previous = (ICustomObject)getPrevious();

// Create the next custom object
ICustomObject next =
    Factory.CustomObject.CreateInstance(_os, "ITSOIdleActivity");

// Create the batch
UpdatingBatch batch = UpdatingBatch.CreateUpdatingBatchInstance(domain,
                                RefreshMode.REFRESH);

// Add the objects
batch.Add(previous, null);
batch.Add(next, null);

// Execute the batch
batch.UpdateBatch();
```

Batch retrieval

In addition to bulk creating and updating objects, batches can also be used to retrieve objects in bulk. Using `RetrievingBatch` can help limit the number of round-trips to the CE, increasing performance, and reducing network traffic.

To retrieve items in a batch:

1. Create the `RetrievingBatch`.

2. Instantiate the objects to be retrieved.

3. Add them to the batch.

4. Retrieve the batch.

5. Work with the returned item(s).

Example 3-40 and Example 3-41 show how to retrieve two documents in a batch using RetrievingBatch.

Example 3-40   Batch retrieval in Java

```
RetrievingBatch rb =
      RetrievingBatch.createRetrievingBatchInstance(domain);

Folder vehicle =
      Factory.Folder.getInstance(os, "ITSOVehicle", vehiclePath);

PropertyFilter filter = new PropertyFilter();
filter.addIncludeProperty(0, null, null,
      PropertyNames.ID + " " +
      PropertyNames.NAME + " " +
      "ITSODailyRate ITSOFranchiseCode" );

rb.add(vehicle, filter);

Document doc =
   Factory.Document.getInstance(os, "ITSODocument", path);

filter = new PropertyFilter();
filter.addIncludeProperty(0, null, null,
      PropertyNames.ID + " ITSOFranchiseCode");

rb.add(doc, filter);

rb.retrieveBatch();
```

Example 3-41   Batch retrieval in C#

```
RetrievingBatch rb =
      RetrievingBatch.CreateRetrievingBatchInstance(domain);

IFolder vehicle =
      Factory.Folder.GetInstance(os, "ITSOVehicle", vehiclePath);

PropertyFilter filter = new PropertyFilter();
filter.AddIncludeProperty(0, null, null,
      PropertyNames.ID + " " +
      PropertyNames.NAME + " " +
      "ITSODailyRate ITSOFranchiseCode");

rb.Add(vehicle, filter);
```

```
IDocument doc =
   Factory.Document.GetInstance(_os, "ITSODocument", path);

filter = new PropertyFilter();
filter.AddIncludeProperty(0, null, null,
      PropertyNames.ID + " ITSOFranchiseCode");

rb.Add(doc, filter);

rb.RetrieveBatch();
```

4

Advanced Content Engine API programming

This chapter continues the discussion of Content Engine (CE) development topics. Although we refer to these topics as *advanced*, they are actually a continuation of the concepts we presented in Chapter 3, "Introduction to Content Engine API programming" on page 39.

Most of the code snippets we show in this chapter are from the sample applications that we created for this book. For more details about the sample applications, refer to Chapter 7, "Sample applications for Fictional Auto Rental Company A" on page 223.

This chapter discusses the following topics:

► Permissions and authorization
► Versioning
► Relationships
► Annotations
► Subscriptions and event actions
► Workflow subscriptions and workflow event actions
► Metadata discovery
► Dynamic security inheritance

4.1 Permissions and authorization

The CE contains features for fine-grained control of access to objects. This control includes both discretionary and mandatory access control. The most fundamental of these features is the *access control list* (ACL). It is represented in the APIs as the class `AccessPermissionList`. An ACL is a list of *access control entries* (ACEs), each of which describes access for a particular user or group.

An ACE is represented in the APIs as the class `AccessPermission`, which is the value type for the Permissions property accessor methods. Users and groups must be defined in the configured directory used by the CE. The CE actually stores the unique identifier for the user or group and translates it dynamically to the GranteeName property. If that translation fails (because the user or group has been deleted from the directory), the unique identifier is presented as the GranteeName, and the GranteeType is set to `SecurityPrincipalType.UNKNOWN`.

The access being granted or denied is given by the `AccessMask` property. It is an integer-valued property representing a simple bit mask of AccessRight values. Bit-wise arithmetic operations can be used to manipulate the values. The APIs also contain a series of constants of type AccessLevel. They just represent combinations of AccessRight values that are useful for some scenarios. They are not additional types of access controls.

Example 4-1 and Example 4-2 on page 89 show an update to the ACL for an object to remove `AccessRight.DELETE permission` for each non-inherited ACE. The net result is that those grantees will not be allowed to delete the object. (This does not affect the owner, and it does not affect a grantee with delete rights through inheritance.)

Example 4-1   AccessPermissionList manipulation in Java

```
/**
 * Removes the AccessRight.DELETE right from all direct ACEs for an
 * object. Caller must have fetched the Permissions property.
 * Return true if we actually changed anything.  Caller must call save.
 */
public boolean removeDeletePerm(IndependentlyPersistableObject ipo)
{
    boolean madeChanges = false;
    Properties props = ipo.getProperties();
    if (!props.isPropertyPresent(PropertyNames.PERMISSIONS))
    {
        return madeChanges;
    }
    AccessPermissionList acl =
```

```
            (AccessPermissionList)props.getDependentObjectListValue(
                        PropertyNames.PERMISSIONS);
    for (Iterator it = acl.iterator(); it.hasNext();)
    {
        AccessPermission ace = (AccessPermission)it.next();
        PermissionSource acePS = ace.get_PermissionSource();
        // Skip any inherited ACEs
        if (acePS == PermissionSource.SOURCE_PARENT) continue;
        int rights = ace.get_AccessMask().intValue();
        if ((rights & AccessRight.DELETE_AS_INT)  !=  0)
        {
            madeChanges = true;
            // Remove the DELETE bit.
            rights &= ~AccessRight.DELETE_AS_INT;
            ace.set_AccessMask(rights);
        }
    }
    return madeChanges;
}
```

Example 4-2   AccessPermissionList manipulation in C#

```
/**
 * Removes the AccessRight.DELETE right from all direct ACEs for an
 * object. Caller must have fetched the Permissions property.
 * Return true if we actually changed anything.  Caller must call save.
 */
public Boolean RemoveDeletePerm(IIndependentlyPersistableObject ipo)
{
    Boolean madeChanges = false;
    IProperties props = ipo.Properties;
    if (!props.IsPropertyPresent(PropertyNames.PERMISSIONS))
    {
        return madeChanges;
    }
    IAccessPermissionList acl =
        (IAccessPermissionList)props[PropertyNames.PERMISSIONS];
    foreach (IAccessPermission ace in acl)
    {
        PermissionSource acePS = ace.PermissionSource;
        // Skip any inherited ACEs
        if (acePS == PermissionSource.SOURCE_PARENT) continue;
        int rights = (int)ace.AccessMask;
        if ((rights & (int)AccessRight.DELETE) != 0)
        {
```

```
                madeChanges = true;
                // Remove the DELETE bit.
                rights &= ~((int)AccessRight.DELETE);
                ace.AccessMask = rights;
            }
        }
        return madeChanges;
    }
```

4.2 Versioning

One of the fundamentals of traditional document management, the forerunner of modern enterprise content management, is that documents change over time. Rather than simply replacing the content as a whole, IBM FileNet Content Manager allows you to make a completely new version of the document, thereby preserving the older versions of the content.

A Document object in CE represents a single version of what you might think of as a document (for example, a spreadsheet, a text file, or a photograph). The collection of all versions of a given document is represented in CE by a VersionSeries object. Each version in a VersionSeries object has its own security, life cycle, and other characteristics as an independently persistable object. The latest version of a document in a VersionSeries is called the *current version* and is pointed to by the CurrentVersion property. There is a two-level version numbering scheme (major and minor versions). The latest major version in a VersionSeries is called the *released version* and is pointed to by the ReleasedVersion property. A checked-in document can also be *promoted* to a major version or *demoted* to a minor version.

Figure 4-1 on page 91 illustrates a VersionSeries object with multiple independent objects. Version 1.0 and 1.1 are superseded. Version 2.0 is the current official release. Version 2.1 is a checked-out version of 2.0 with a reservation state.
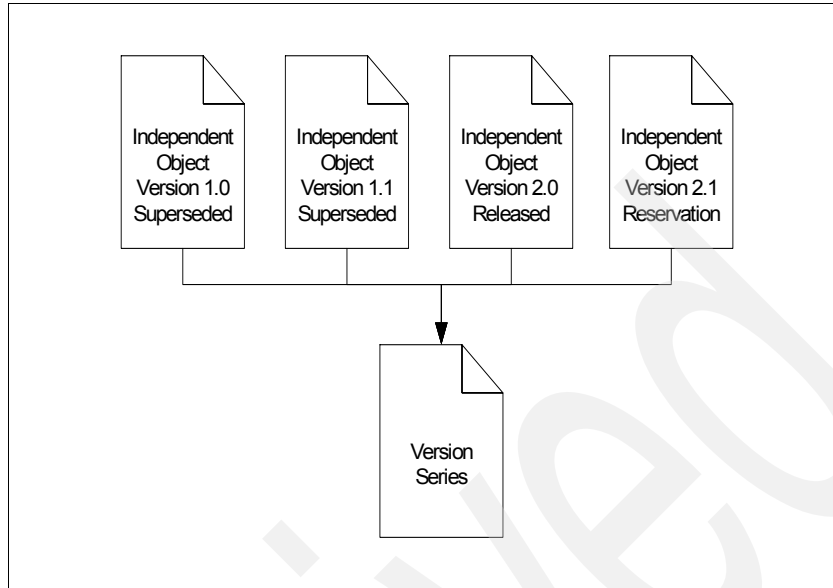
Figure 4-1   VersionSeries object with multiple IndependentObjects

To create a new version of a document, perform a check-out and check-in cycle:

1. Check out the current version of the document. This creates a new
    reservation document version. A reservation is not a different type of object. It
    is merely a new document version marked as being in the reservation state.
    The immediately previous version is in the *reserved* state.

    Properties may be automatically copied from the current version to the
    reserved version, depending on the value of the PropertyDescription boolean
    CopyToReservation metaproperty.

2. Make content changes and possibly property changes.

3. Check in the reserved document version.

    If you change your mind and want to cancel the check-out step, you can call
    the `cancelCheckout` helper method or simply delete the reserved document
    version (these are exactly the same operation).

Content can only be updated on a document object that is in the reserved state.
You can update content as many times as you want, with new content completely
replacing the old content for that reserved version. After you perform the check-in
step, the content for that version becomes immutable. (Properties for document
versions can be changed at any time. Locking out property changes for a
particular document version is possible by calling the freeze method.)

Within a given version series, individual independent versions can be deleted and removed from the version series by instantiating a specific document object and calling its delete method. Deleting the entire version series requires instantiating the VersionSeries object and calling its delete method. The instantiation of the VersionSeries can be done by using a `Factory.VersionSeries` method or by getting the value of the VersionSeries property from any of the documents in the version series.

Example 4-3 and Example 4-4 on page 93 show how to retrieve a document, check it out, add content, and check the document back in as major version.

Example 4-3   Check-out and check-in cycle in Java

```
// Get the document (saving a round-trip that a fetch would require)
Document doc = Factory.Document.getInstance(os,"Document","/Doc1");

// Checkout the document and save
doc.checkout(ReservationType.EXCLUSIVE, null, null, null);
doc.save(RefreshMode.REFRESH);

// Get the reservation object
Document res = (Document)doc.get_Reservation();

// Update the properties
res.getProperties().putValue("DocumentTitle","NextVersion");

// Prepare the content for attaching
// Create the element list
ContentElementList list = Factory.ContentElement.createList();

// Create a content transfer element by attaching a simple text file
ContentTransfer element = Factory.ContentTransfer.createInstance();

// Set the MIME type
element.set_ContentType("text/plain");

// Set the retrieval name
element.set_RetrievalName("file.txt");

// Set the content source
element.setCaptureSource(new FileInputStream(new File("file.txt")));

// Add the item to the list
list.add(element);

// Add the content transfer list to the document
res.set_ContentElements(list);

// Set the PendingAction to be check-in as a major version without
```

```
// automatic content classification
res.checkin(AutoClassify.DO_NOT_AUTO_CLASSIFY,
CheckinType.MAJOR_VERSION);

// Save the document to the repository
res.save(RefreshMode.NO_REFRESH);
```

Example 4-4   Check-out and check-in cycle in C#

```csharp
// Get the document (saving a round-trip that a fetch would require)
IDocument doc = Factory.Document.GetInstance(os,"Document","/Doc1");

// Checkout the document and save
doc.Checkout(ReservationType.EXCLUSIVE, null, null, null);
doc.Save(RefreshMode.REFRESH);

// Get the reservation object
IDocument res = (IDocument)doc.Reservation;

// Update the properties
IProperties properties = res.Properties;

// Set the value
properties["DocumentTitle"] = "NewVersion";

// Create the content element list
IContentTransferList l = Factory.ContentTransfer.CreateList();

// Create the content element
IContentTransfer element = Factory.ContentTransfer.CreateInstance();
element.RetrievalName = "file.txt";
element.ContentType = "text/plain";
element.SetCaptureSource(
System.IO.File.Open("file.txt", FileMode.Open));
l.Add(element);

// Add the elements to the object
res.ContentElements = l;

// Set the check-in type
res.Checkin(AutoClassify.DO_NOT_AUTO_CLASSIFY, CheckinType.MAJOR_VERSION);

// Commit the changes to the CE
res.Save(RefreshMode.NO_REFRESH);
```

4.3 Relationships

The CE APIs and the CE server itself have many features which emulate features of modern object-oriented programming languages. One of those features is the ability to create explicit relationships between objects. In most cases, the server perform referential integrity checks based on those relationships.

4.3.1 Object-valued properties

The fundamental mechanism for relationships between objects is the object-valued property (OVP). In a programming language, the analogy would be to a reference or pointer to another object. Just as in a programming language, an OVP has a required type. The server enforces the restriction that only objects of the correct type (including possibly a subclass of the required type) are assigned to an OVP. Although defining an OVP (or any other property) programmatically is unusual, actually populating and using OVPs is very common.

OVPs can be either single-valued or multi-valued (MVOVP). Except for a few system OVPs, specifically lists of dependent objects, MVOVPs are always read-only. This concept can be confusing, but it becomes very clear when you understand just a little about the implementation. MVOVPs are not persisted in object instances in the repository. Instead, the metadata notes that some single-valued OVPs point to the object holding the MVOVP. When the value of the MVOVP is needed, the server dynamically generates a query to find all objects that are pointing to this object and returns the results of that query.

For example, the system class `Annotation` has an OVP named `AnnotatedObject`. Classes that can have annotations (folders, documents, and custom objects) have an MVOVP named `Annotations`. Because the value of `Annotations` reflects the values of another property on a different set of objects, it is called a *reflective property*. Combinations of the two such properties (for example, `AnnotatedObject` and `Annotations`) are sometimes called *association properties*. When you create a new annotation and populate its `AnnotatedObject` property, the corresponding `Annotations` property appears to be instantaneously updated.

Example 4-5 on page 95 and Example 4-6 on page 95 show the assignment of OVPs. It is mechanically similar to any other property value assignment.

Example 4-5   OVP manipulation in Java, adapted from ITSONewVehicleActivity.java

```
/* Folder vehicle = ... */

// Create the ITSOIdleActivity object
CustomObject activity =
    Factory.CustomObject.createInstance(os, "ITSOIdleActivity");

// Get the properties collection
Properties props = activity.getProperties();

// Set the properties
props.putValue("ITSOFranchiseCode",
    vehicle.getProperties().getStringValue("ITSOFranchiseCode"));
props.putObjectValue("ITSOVehicle", vehicle);

// Save the new IdleActivity
activity.save(RefreshMode.NO_REFRESH);
```

Example 4-6   OVP manipulation in C#

```
/* IFolder vehicle = ... */

// Create the ITSOIdleActivity object
ICustomObject activity =
    Factory.CustomObject.CreateInstance(os, "ITSOIdleActivity");

// Get the properties collection
IProperties props = activity.Properties;

// Set the properties
props["ITSOFranchiseCode"] =
    vehicle.getProperties().getStringValue("ITSOFranchiseCode");
props["ITSOVehicle"] = vehicle;

// Save the new IdleActivity
activity.Save(RefreshMode.NO_REFRESH);
```

4.3.2 Filing in a folder

Perhaps the most well-known feature involving OVPs is the containment
relationship whereby objects are filed into a folder. Besides the folder and the
object being filed, there is an intermediate relationship object.

It is either of the following objects:

- `ReferentialContainmentRelationship` (RCR) object
- `DynamicReferentialContainmentRelationship` (DRCR) object

Both RCR and DRCR are extensible. The relationship object, being an intermediary, enables a folder to contain many objects and an object to be contained in many folders. In fact, a given object may be contained more than once in the same folder.

The RCR (and DRCR) `Head` property is reflected by containee's `Containers` property. The RCR `Tail` property is reflected by the folder `Containees` property. To understand the nomenclature of head and tail, imagine arrows pointing from the folder to the various containees. The RCR properties give you the tails and heads of those arrows. Imagine the RCR object as a single marker of sorts in the middle of each arrow.

The APIs provide convenience methods for filing and unfiling objects. They are convenience methods because they do not represent fundamental operations and are instead built upon creation or deletion of RCR or DRCR objects. Newcomers to the APIs generally find the convenience methods easier to understand, but developers experienced in the APIs generally find direct creation and manipulation of the RCR and DRCR objects much clearer. The difference in terms of lines of code is very small.

Example 4-7 and Example 4-8 on page 97 show a sample of filing, adapted from `RentalActivityHandler.java` in the sample application. Because it happens to be filing a CustomObject, the sample uses RCR. For filing a document object, you want to use DRCR unless you have a very specific use case for using RCR.

Example 4-7   Filing a document to a folder in Java

```
Document d = Factory.Document.getInstance(os, "Document", "/Document");
Folder f = Factory.Folder.getInstance(os, "Folder", "/Folder");

ReferentialContainmentRelationship rcr =
    Factory.ReferentialContainmentRelationship.createInstance(os,
    ClassNames.DYNAMIC_REFERENTIAL_CONTAINMENT_RELATIONSHIP);

rcr.set_Head(d);
rcr.set_Tail(f);

rcr.save(RefreshMode.NO_REFRESH);
```

Example 4-8   Filing a document in a folder in C#

```
IDocument d = Factory.Document.GetInstance(os, "Document",
"/Document");
IFolder f = Factory.Folder.GetInstance(os, "Folder", "/Folder");

IReferentialContainmentRelationship rcr =
    Factory.ReferentialContainmentRelationship.CreateInstance(os,
    ClassNames.DYNAMIC_REFERENTIAL_CONTAINMENT_RELATIONSHIP);

rcr.Head = d;
rcr.Tail = f;

rcr.Save(RefreshMode.NO_REFRESH);
```

4.3.3 Compound documents

Certain documents can be thought of as being comprised of component parts. For example, a Web page consists of a text part containing the HTML, perhaps additional text parts for the cascading style sheets and JavaScript libraries. There are likely additional parts representing graphical images. Taken together, these can be thought of as a type of compound document.

The *component document* parts typically form some sort of hierarchy for a given complete document. Component parts may be reused in several related or unrelated documents, depending on the application use case. All of that taken together prompts the need for creating many-to-many relationships among an arbitrary collection of document components. There are many ways to model that in CE, but the feature provided specifically for it is the ComponentRelatonship class.

In many ways, the ComponentRelationship class resembles the other relationship classes described in 4.3.2, "Filing in a folder" on page 95. In compound document terminology, the relationships are between component parents and children. The ComponentRelationship class has several properties describing or allowing you to manipulate various aspects of the parent and child relationships. For example, unlike folder containment, which is unordered, compound document relationships can be explicitly ordered.

Example 4-9 on page 98 and Example 4-10 on page 98 show simple manipulation of compound documents.

Example 4-9   Compound documents in Java

```java
/**
 * Count the total number of children and parent documents
 */
public int childrenAndParents(Document doc)
{
    System.out.println("document: " + doc);
    int childCount = 0;
    // Treat this doc as a parent and count its children
    if (doc.get_CompoundDocumentState() ==
            CompoundDocumentState.COMPOUND_DOCUMENT)
    {
        // Could get_ChildRelationships instead.
        DocumentSet children = doc.get_ChildDocuments();
        for (Iterator it = children.iterator(); it.hasNext();)
        {
            Document child = (Document)it.next();
            System.out.println("child: " + child);
            ++childCount;
        }
    }
    int parentCount = 0;
    // Any document can be a child, so always count parents.
    // Could get_ParentRelationships instead.
    DocumentSet parents = doc.get_ParentDocuments();
    for (Iterator it = parents.iterator(); it.hasNext();)
    {
        Document parent = (Document)it.next();
        System.out.println("parent: " + parent);
        ++parentCount;
    }
    return childCount + parentCount;
}
```

Example 4-10   Compound documents in C#

```csharp
/// <summary>
/// Count the total number of children and parent documents
/// </summary>
public int ChildrenAndParents(Document doc)
{
    Console.WriteLine("document: " + doc);
    int childCount = 0;
    // Treat this doc as a parent and count its children
```

```
        if (doc.CompoundDocumentState ==
            CompoundDocumentState.COMPOUND_DOCUMENT)
        {
            // Could use ChildRelationships instead.
            IDocumentSet children = doc.ChildDocuments;
            foreach (IDocument child in children)
            {
                Console.WriteLine("child: " + child);
                ++childCount;
            }
        }
        int parentCount = 0;
        // Any document can be a child, so always count parents.
        // Could use ParentRelationships instead.
        IDocumentSet parents = doc.ParentDocuments;
        foreach (IDocument parent in parents)
        {
            Console.WriteLine("parent: " + parent);
            ++parentCount;
        }
        return childCount + parentCount;
}
```

4.4 Annotations

An *annotation*, as its name implies, is some kind of additional information
associated with an object. For example, you can think of using a pen to physically
mark up a photograph or to add review comments to a document. Although one
typically thinks of an annotation as being graphically overlaid on the annotated
object, there is no requirement for that to be so. The non-versionable Annotation
class may contain content elements. It can also be made into a subclass, so any
desired custom properties may be added to it. The specific subclass type can be
used to distinguish among different annotation types on an object.

The CE annotation feature provides relationships and convenient default security
and other behaviors for annotation objects. Documents, folders, and custom
objects may have associated annotations. For Documents, the Annotation
property AnnotatedContentElement provides the opportunity to affiliate an
annotation with a specific content element. When you do that, using the invariant
content element ElementSequenceNumber to make that association is better than
using the volatile position number of the content element within a Document's
content elements collection.

An annotation can be for any purpose that an application developer feels is appropriate. The Application Engine's Java viewer applet has its own specific implementation of annotations, but the formats of those annotations are subject to change at any time without notice. Annotations may also be carried over from Image Services with Content Federation Services for Image Services (CFS-IS).

Example 4-11 and Example 4-12 on page 101 illustrate the creation of a custom annotation and associating it with a specific Document content element. The custom properties are adapted from the sample application.

Example 4-11   Custom annotation in Java

```
public Annotation createThumbnail(Document gallery, int eltNum)
{
        ContentElementList galleryElements =
gallery.get_ContentElements();
        ContentElement contentElement =
(ContentElement)galleryElements.get(eltNum);
        if (!(contentElement instanceof ContentTransfer))
        {
            throw new RuntimeException("Cannot make thumbnail for
content reference");
        }
        ContentTransfer gCt = (ContentTransfer)contentElement;
        String type = gCt.get_ContentType();
        String rName = gCt.get_RetrievalName();
        Integer esn = gCt.get_ElementSequenceNumber();

        Properties gProps = gallery.getProperties();
        String franchiseCode =
gProps.getStringValue("ITSOFranchiseCode");

        // Instantiate the new thumbnail Annotation
        ObjectStore os = gallery.getObjectStore();
        Annotation thumbnail =
            Factory.Annotation.createInstance(os, "ITSOThumbnail");

        Properties tProps = thumbnail.getProperties();
        tProps.putValue("ITSOFranchiseCode", franchiseCode);
        thumbnail.set_AnnotatedObject(gallery);
        // Better to use immutable ESN than element position
        thumbnail.set_DescriptiveText("thumbnail image [" + esn + "]");
        thumbnail.set_AnnotatedContentElement(esn);

        ContentTransfer tCt = Factory.ContentTransfer.createInstance();
        tCt.set_ContentType(type);
```

```
            if (rName != null  &&  rName.length() > 0)
            {
                tCt.set_RetrievalName("t_" + rName);
            }

            // The computation of the thumbnail is done elsewhere
            InputStream photoStream = gCt.accessContentStream();
            tCt.setCaptureSource(computeThumbnailGraphics(type,
photoStream));

            ContentElementList thumbElements =
Factory.ContentElement.createList();
            thumbElements.add(tCt);
            thumbnail.set_ContentElements(thumbElements);

            // Return the new Annotation.  Caller must save.
            return thumbnail;
}
```

Example 4-12   Custom annotation in C#

```
public IAnnotation CreateThumbnail(IDocument gallery, int eltNum)
{
        IContentElementList galleryElements = gallery.ContentElements;
        IContentElement contentElement =
(IContentElement)galleryElements[eltNum];
        if (!(contentElement is IContentTransfer))
        {
            throw new Exception("Cannot make thumbnail for content
reference");
        }
        IContentTransfer gCt = (IContentTransfer)contentElement;
        string type = gCt.ContentType;
        string rName = gCt.RetrievalName;
        int? esn = gCt.ElementSequenceNumber;

        IProperties gProps = gallery.Properties;
        string franchiseCode =
gProps.GetStringValue("ITSOFranchiseCode");

        // Instantiate the new thumbnail Annotation
        IObjectStore os = gallery.GetObjectStore();
        IAnnotation thumbnail =
            Factory.Annotation.CreateInstance(os, "ITSOThumbnail");
```

```
        IProperties tProps = thumbnail.Properties;
        tProps["ITSOFranchiseCode"] = franchiseCode;
        thumbnail.AnnotatedObject = gallery;
        // Better to use immutable ESN than element position
        thumbnail.DescriptiveText = "thumbnail image [" + esn + "]";
        thumbnail.AnnotatedContentElement = esn;

        IContentTransfer tCt =
Factory.ContentTransfer.CreateInstance();
        tCt.ContentType = type;
        if (rName != null  &&  rName.Length > 0)
        {
            tCt.RetrievalName = "t_" + rName;
        }

        // The computation of the thumbnail is done elsewhere
        Stream photoStream = gCt.AccessContentStream();
        tCt.SetCaptureSource(ComputeThumbnailGraphics(type,
photoStream));

        IContentElementList thumbElements =
Factory.ContentElement.CreateList();
        thumbElements.Add(tCt);
        thumbnail.ContentElements = thumbElements;

        // Return the new Annotation.  Caller must save.
        return thumbnail;
}
```

4.5 Subscriptions and event actions

A key enabler of *active content* is the CE event subscription subsystem. Event
subscriptions may be created for most types of update request arriving at the CE.
These subscriptions are often set up manually through Enterprise Manager, but
they can also be established programmatically. The subscriptions establish the
triggers leading to the execution of user-provided code running inside the CE
server. Subscriptions may be established on specific instances of objects,
through the class InstanceSubscription, or for an entire class (and, optionally,
any subclasses) of objects with the class ClassSubscription.

> **Note:** The event subscription subsystem is strongly related to the audit subsystem. That approach is understandable when you consider that the same activities are interesting both for auditing and for event notifications. The audit subsystem can also be enabled for content retrieval but the event subsystem cannot. We do not cover the audit subsystem in this book. Refer to the product documentation for further information.

Event subscriptions are divided into *synchronous* and *asynchronous* types. The subscription object sets the type to either synchronous or asynchronous. Informally, the associated EventAction is commonly referred to as a synchronous or asynchronous event handler.

Note the following information about synchronous and asynchronous:

► A synchronous event handler is executed as part of the path to persisting a change. It is within the same transaction scope. In other words, the change has not actually happened yet. If the event handler throws an exception, the entire transaction is aborted. In this way, the synchronous event action can veto a change. An important restriction on synchronous event handlers is that they are not allowed to update the source object, either directly or by acting on a copy.

► An asynchronous event handler is executed after a change has been committed. The delay between the triggering change and the running of the asynchronous event handler is typically small, but guaranteed maximum delay. An asynchronous event handler has its own transaction scope. Although it cannot veto the change that has already happened, the asynchronous event handler can update the source object. It can also do just about anything else that the developer wants it to do if it fits within the transaction scope and timeout and can be done in the context of a J2EE application server container.

Both types of event handlers allow the insertion of user code in a common location in the CE server. The synchronous case can be used to approve or veto updates to keep them consistent with an application policy. The asynchronous case can be used to make changes to keep state consistent, again according to an application policy. Either type of event handler can be used to perform notifications or to initiate actions in external systems.

Unlike other forms of application code, an event handler has the nature of a callback and is called by the CE server code. It receives a copy of the source object along with sufficient information to understand the nature of the triggering update activity. Because event handlers execute in the context of the server, normal access control checks do not apply. Another consequence of executing in

server context, however, is that event handlers must be written in Java. Interactions with the CE server are through the Java API.

Example 4-13 shows an example of an event handler. It is adapted from `ITSONewVehicleActivity.java` in the sample application.

Example 4-13   Event handler in Java

```
public class ITSONewVehicleActivity implements EventActionHandler
{
    public void onEvent(ObjectChangeEvent event, Id subId)
    {
        try
        {
            // Get the object store and the new vehicle folder
            ObjectStore os = event.getObjectStore();
            Folder vehicle = (Folder) event.get_SourceObject();

            // Create the ITSOIdleActivity object
            CustomObject activity =
                    Factory.CustomObject.createInstance(os,
"ITSOIdleActivity");

            // Get the properties collection
            Properties props = activity.getProperties();

            // Get the start date
            Date start = new Date(System.currentTimeMillis());

            // Calculate an end date of 20 years from now
            GregorianCalendar end = new GregorianCalendar();
            end.setTime(start);
            end.add(Calendar.YEAR, 20);

            // Set the properties
            props.putValue("ITSOFranchiseCode",
vehicle.getProperties().getStringValue("ITSOFranchiseCode"));
            props.putObjectValue("ITSOVehicle", vehicle);
            props.putValue("ITSOStartDate", start);
            props.putValue("ITSOEndDate", end.getTime());

            // Save the new IdleActivity
            activity.save(RefreshMode.NO_REFRESH);
        }
        catch (RuntimeException r)
        {
```

```
            throw r;
        }
        catch (Throwable t)
        {
            // Catch any other exception and wrap it
            throw new RuntimeException("Unexpected exception", t);
        }
    }
}
```

4.6 Workflow subscriptions and workflow event actions

A special case of the event subscription subsystem is represented by classes that are related to the launching of workflows. `InstanceWorkflowSubscription` and `ClassWorkflowSubscription` are subclasses of the more general subscription classes. They have properties that associate the necessary artifacts (the workflow definition, property mapping, and so on) needed for a successful workflow launch. An interface to the PE is integrated with the CE, and setting up the automated launching of workflows that are triggered by events on objects is very simple to do.

A workflow subscription takes a `WorkflowEventAction` to handle events instead of the more general `EventAction` class. A requirement is to use a `WorkflowEventAction` with a workflow subscription.

Because the programming model for WorkflowEventAction is exactly the same as for EventAction, refer to Example 4-13 on page 104 for a code sample.

4.7 Metadata discovery

Nearly everything about CE content classes and properties is discoverable at run time. You can discover data types, localizable display names, default values, and many more attributes. In some sense, this feature is analogous to the reflection features that are available in Java and C#.

Every class in an ObjectStore is defined in a ClassDefinition. Every property of a class is defined by a PropertyDefinition. A ClassDefinition has a list of PropertyDefinitions. Although the definition objects are fully read-write capable, updating these objects programmatically is unusual. Alternatively, consulting the metadata attributes to control application logic is quite common. To facilitate that,

there is a read-only and slightly more compact representation of the metadata embodied in ClassDescription and PropertyDescription objects.

> **Note:** Newcomers to CE often find the similar terminology of *definition* and *description* confusing. Think of it this way: The definition objects are the place where you can actively define various attributes for a class or property. The description objects merely describe what has been defined (and so are read-only).

The high-level steps for finding a piece of metadata for a particular property is:

1. Find the applicable ClassDescription object.

2. Iterate over the PropertyDescriptions collection looking for the PropertyDescription object of interest.

3. Read attributes from that PropertyDescription object.

Consulting the metadata is common, and changes to it are so rare that the APIs implement a client-side metadata cache (CMC). The CMC transparently intercepts some metadata fetches and satisfies them with locally cached copies. Because the class and property description objects are read-only, there is no harm in sharing them. CMC automatically keeps track of the differences in requester locale and organizes the data by ObjectStore.

Although most of the CMC behavior is automatic and transparent, there is also an explicit way of fetching things from CMC. Many developers find this clearer than the implicit behavior. To use the explicit behavior, use a Factory method to instantiate a `MetadataCache` object. From there, the use of `MetadataCache` methods is strightforward.

Example 4-14 and Example 4-15 on page 107 show metadata discovery. If you want to find all of the custom classes and properties in the sample application, you can call this method with a value of `itso` for both prefix parameters.

Example 4-14   Metadata discovery in Java

```
public void cmcDemo(ObjectStore os, String classPrefix, String propPrefix)
{
    classPrefix = classPrefix.toLowerCase();
    propPrefix  = propPrefix.toLowerCase();
    MetadataCache cmc = Factory.MetadataCache.getDefaultInstance();

    // Use a query to get a list of class names from the ObjectStore.
    // Can't query on ClassDescription, so query on ClassDefinition
    // (the class names are identical by design).
    SearchScope ss = new SearchScope(os);
    String sqlString = "SELECT " + PropertyNames.SYMBOLIC_NAME
```

```
                      + " FROM " + ClassNames.CLASS_DEFINITION;
        SearchSQL sql = new SearchSQL(sqlString);
        RepositoryRowSet rrs = ss.fetchRows(sql, null, null, true);

        // Iterate over the results.  For any class with a name that
        // starts with the classPrefix, pull a ClassDescription out of
        // CMC and look for PropertyDescriptions whose names start with
        // the propPrefix.
        for (Iterator it = rrs.iterator(); it.hasNext();)
        {
            RepositoryRow row = (RepositoryRow)it.next();
            Properties props = row.getProperties();
            String className =
                props.getStringValue(PropertyNames.SYMBOLIC_NAME);
            if (className.toLowerCase().startsWith(classPrefix))
            {
                ClassDescription cd =
                    cmc.getClassDescription(os, className);
                PropertyDescriptionList pds =
                    cd.get_PropertyDescriptions();
                for (Iterator itp = pds.iterator(); itp.hasNext();)
                {
                    PropertyDescription pd = (PropertyDescription)itp.next();
                    String propName = pd.get_SymbolicName();
                    if (propName.toLowerCase().startsWith(propPrefix))
                    {
                        TypeID type = pd.get_DataType();
                        // Print the results.
                        System.out.println(className + "." +
                                         propName + " (" + type + ")");
                    }
                }
            }
        }
    }
}
```

Example 4-15   Metadata discovery in C#

```
public void cmcDemo(IObjectStore os, string classPrefix, string propPrefix)
{
    classPrefix = classPrefix.ToLower();
    propPrefix  = propPrefix.ToLower();
    IMetadataCache cmc = Factory.MetadataCache.GetDefaultInstance();

    // Use a query to get a list of class names from the ObjectStore.
    // Can't query on ClassDescription, so query on ClassDefinition
    // (the class names are identical by design).
    SearchScope ss = new SearchScope(os);
    String sqlString = "SELECT " + PropertyNames.SYMBOLIC_NAME
```

```
                      + " FROM " + ClassNames.CLASS_DEFINITION;
SearchSQL sql = new SearchSQL(sqlString);
IRepositoryRowSet rrs = ss.FetchRows(sql, null, null, true);

// Iterate over the results.  For any class with a name that
// starts with the classPrefix, pull a ClassDescription out of
// CMC and look for PropertyDescriptions whose names start with
// the propPrefix.
foreach (IRepositoryRow row in rrs)
{
    IProperties props = row.Properties;
    string className =
        props.GetStringValue(PropertyNames.SYMBOLIC_NAME);
    if (className.ToLower().StartsWith(classPrefix))
    {
        IClassDescription cd =
            cmc.GetClassDescription(os, className);
        IPropertyDescriptionList pds =
            cd.PropertyDescriptions;
        foreach (IPropertyDescription pd in pds)
        {
            string propName = pd.SymbolicName;
            if (propName.ToLower().StartsWith(propPrefix))
            {
                TypeID type = pd.DataType;
                // Print the results.
                Console.WriteLine(className + "." +
                                  propName + " (" + type + ")");
            }
        }
    }
}
}
```

4.8 Dynamic security inheritance

Dynamic security inheritance, introduced in IBM FileNet Content Manager 4.0.1, is not specifically an API feature. It is an important server feature that can greatly simplify security-related programming. The idea of security inheritance is that some or all of the access control entries (ACEs) from one object are automatically inherited by another object or objects. Dynamic security inheritance means that the inheritance of the ACEs is done as needed, when access checks for the inheriting object are being calculated. This is not in any way a copying of ACEs from one object to another. It is true inheritance. The CE server goes to a great deal of trouble to make the process extremely efficient.

Some traditional inheritance features in the server are implemented in terms of dynamic security inheritance. Perhaps the most well-known of these is the inheritance of folder security by the folder's containees. It is also possible for developers to define custom security inheritance relationships. The implementation mechanism is to define a custom object-valued property (OVP) and designate that property as pointing to a security proxy object. Specifically, you set the value of the corresponding `PropertyDefinitionObject` `SecurityProxyType` property to `SecurityProxyType.FULL`. On the security proxy object itself, you mark the `InheritableDepth` property of each ACE according to how you would like the inheritance to proceed (none, immediate children, or infinite).

Dynamic security inheritance can act transitively. That is, object A can inherit security from object B, which in turn can inherit security from object C. There is no built-in limitation for how long such chains can be. Directly applied ACEs have precedence over inherited ACEs in the access check calculations. In practical terms, this means that a user or group that is denied access through inheritance can still be granted access by a directly applied ACE.

An interesting use of dynamic security inheritance is to augment the enterprise directory group structure. This can be especially handy in situations where the natural organizational groups typically found in a directory do not represent security access groups very well. It can also be helpful in situations where security access groups or memberships in those groups changes frequently, and the site does not want to make such frequent changes to the directory. In some ways, security proxy objects can be used to implement an authorization scheme that resembles a limited form of role-based access control (RBAC) that is often sufficient to meet site needs.

Rather than providing a code sample for this feature (which is typically an administrative set-up and the application of appropriate default values for the security proxy property on new object instances), we instead provide a description of how it can work with the sample application described elsewhere in this book. Understanding the sample application in detail is not necessary in order to follow this description.

A franchise location consists of maintenance employees, front-office employees, and supervisors responsible for managing both maintenance and front-office operations. Maintenance records (`ITSOMaintenanceActivity` objects in the sample application data model) may be accessed by maintenance employees and supervisory employees, but only within that franchise location. Those objects (and all objects in the sample application data model) have an OVP called `ITSOSecurityProxy`. There is also a class of objects called `ITSORole` that is intended to act as security proxy objects, among other things; see the sample application description for more details.

Although directly adding all maintenance employees and supervisors of a franchise location to every object where they need access is possible, it then becomes necessary to update the ACL on many objects any time a maintenance employee or supervisor is hired, resigns, changes status, and so on. This is both an operational and performance burden, and the chances for data errors are high. We can solve it more efficiently and elegantly with a chain of proxy objects:

► Create an ITSORole object to represent the concept of *employees who should have access to maintenance records at this franchise location*. It should have no directly applied ACEs. We refer to this object as MS.

► Create an ITSORole object to represent *supervisors at this franchise location*. It has direct ACEs that are granting access to all the supervisors at that location. We refer to this object as S.

► Create an ITSORole object to represent *maintenance employees at this franchise location*. It has direct ACEs granting access to all of the maintenance employees at that location. We refer to this object as M.

► Maintenance records at this franchise location are created with inheritance from object MS. This is done by populating the ITSOSecurityProxy property with a pointer to MS.

► Object MS inherits security from both object M and object S. This is done by populating the ITSOSecurityProxy property with a pointer to object M and the ITSOSecurityProxy2 with a pointer to object S.

With this arrangement, access for supervisors (object S) and maintenance employees (object M) flows to object MS. From there, the access for the combined groups flows to the maintenance record object.

There are many ways to organize this sort of custom inheritance. If access control from groups is being combined, as in this example, you need a class with more than one security proxy property. That is why we used the intermediate object MS in this example; we only needed the second security proxy property on the ITSORole object instead of all the other objects in the data model.

Although modeling any kind of aggregation with intermediate objects with just two security proxy properties should be possible, defining more might be more convenient for you. A good practice is to define as few security proxy properties as you think you need and then add more later if necessary.

5

Introduction to Process Engine API programming

In this chapter, we introduce the concepts and elements of the IBM FileNet Business Process Manager API. We also introduce the usage of workflow-related operations to develop process applications using the available Process Engine (PE) APIs. Most of the code snippets we show in this chapter are from the sample applications that we created for this book. For more details about the sample applications, refer to Chapter 7, "Sample applications for Fictional Auto Rental Company A" on page 223.

This chapter discusses the following topics:

► Process Engine API overview
► Establishing a Process Engine session
► Handling API exceptions
► Launching a workflow
► Search work items
► Process work items
► Work with process status

111

5.1 Process Engine API overview

PE provides a number of API packages available for process application development. Among this set of APIs are packages that are most commonly used for PE programming, which at the same time are the base for other sets of PE APIs. In this book, we present the following PE APIs:

► Process Engine Java API

   This interface includes all the functionality exposed by the PE. It is the API base for both the PEWS and REST API interfaces used for complex PE development.

► Process Engine Web Services

   Use this API for .NET development, which uses wrapper classes generated from the Web Service Description Language (WSDL) of the Process Engine Web Services (PEWS). This API provides functionality for most of the PE use cases presented in this chapter.

► Process Engine REST API

   The Representational State Transfer (REST) service is included in a servlet deployed in the WorkplaceXT application. This service is a style of accessing PE resources over HTTP using query string format for both request and response data. The REST API resources use the JavaScript Object Notation (JSON) MIME type as a lightweight data-interchange format.

   For further details about the JSON response schemas for the REST resources, refer to the following P8 documentation path: **ECM Help → Developer Help → Process Engine Development → Developer's Guide → Process Engine REST Service → Process Engine REST Service Reference**.

PE also provides a Process Orchestration API, which enables the coordination of events within a workflow definition based on Web services. This API is implemented through the Web services adaptor as part of Component Integrator. For further details about the Component Integrator Web services adaptor and how to implement it, refer to "Chapter 8: Implementing Component integrator and Web services" in *Introducing IBM FileNet Business Process Manager*, SG24-7509.

For PE development, the solution platform and the process use cases dictate which API to use. For example, if you plan to implement a complex process application by using the J2EE platform, select the PE Java API for this scenario.

In this book, we include code snippets showing how to implement the PE APIs for each of the use cases related to PE development. Because the PE Java API

exposes all PE functionality, this API is used for all use cases that are presented; however, for certain cases, we also show how those cases can be implemented using the PEWS API for .NET clients or the PE REST API, depending on the use case and whether the API includes the required functionality.

5.1.1 Functional groups

There are over 100 classes in the PE Java API. Although all these classes are arranged in the single `filenet.vw.api` package, there are three functional groups within this API.

The following list describes the three functional groups that comprise the PE Java API package:

- Administration and configuration API

  Use this API to access and modify system-wide administration information or tasks, such as initializing and emptying regions or removing databases, accessing configuration information for an isolated region, and accessing information for a user and groups.

- Workflow definition API

  This API exposes the workflow definition classes to create, delete, and access a workflow and its sub-objects such as step and field objects.

- Runtime API

  This API handles database and work objects that are used to log in to the PE and create a process session, retrieve work objects, query queues, retrieve step processor information, and perform other workflow-related operations.

Figure 5-1 presents the stack of primary objects (classes) in the PE Runtime API, which is used to perform the most common actions for process application development. This chapter focuses on use cases related to the PE Runtime API.
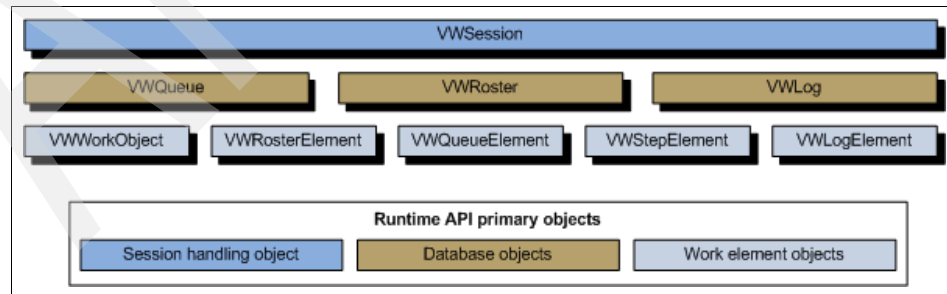
Figure 5-1   Primary Runtime API objects

5.1.2 Available API functionality

To better understand the functionality available across the PE APIs presented in this book, Table 5-1 lists available operations for each API.

Table 5-1   Process Engine APIs and available functionality

Process Engine API	Available functionality
Java API	This API exposes full functionality included in all PE API functional groups.
PEWS API	This API exposes limited functionality available in the PE Runtime API. The operations that are available by using this API are: ► Launch a workflow ► Get queue elements ► Get queue names ► Get roster elements ► Get roster names ► Get step elements from a roster ► Get step elements from a queue ► Get work class names ► Retrieve step element ► Reassign step element ► Unlock step element ► Update step element ► Get milestones
REST API	This API exposes limited functionality available in the PE Runtime API. The operations that are available by using this API are: ► Launch a workflow ► Get queue elements ► Get queue elements count ► Get application space roles ► Get role definition ► Retrieve step element ► Update step element ► Get queue workbasket definition ► Get queue workbasket attributes ► Get workbasket column definitions ► Get workbasket filter definitions ► Get workbasket filter attributes ► Get work classes collection ► Get work class definition

5.1.3 Naming conventions

The PE Java API includes naming conventions for objects and methods to assist process application developers in keeping the usage of PE programming more understandable by making the APIs easier to read.

Table 5-2 describes the naming conventions used by PE API objects.

Table 5-2   Process Engine API objects naming convention

Object naming convention	Description	Examples
VW + *ObjectName*	Convention used by API base objects	VWRoster, VWQueue
VWObjectName + Element	Convention used by API objects to access a base object element or instance	VWRosterElement, VWQueueElement
VWObjectName + Definition	Convention used by API objects to retrieve and administer base objects	VWRosterDefinition, VWQueueDefinition
VWObjectName + Type	Convention used by API objects to retrieve a base object's type	VWFetchType, VWAttachmentType

PE Java API object methods also use naming conventions to perform actions over certain PE objects.

Table 5-3 on page 116 describes the naming convention used by PE API object methods.

Table 5-3   Process Engine API object methods naming convention

Object method	Description	Examples
fetchMethod()	Retrieves an object from a PE isolated region executing a database operation.	`VWRoster.fetchCount()` method gets the number of work item in the roster.
getMethod()	Retrieves a parameter from the PE object. These methods usually do not require additional database operations because the parameters are retrieved from cache.	`VWRoster.getName()` method returns the translated name.
createMethod()	Instantiates a VW object, retrieving it from the PE isolated region.	`VWQueue.createQuery()` method performs a filtered fetch of queue items and returns a VWQueueQuery object.
setMethod()	Sets a parameter value for a PE object.	`VWDataField.setValue()` method sets the value of the data contained in an editable field.
isMethod()	Tests presence of an object state.	`VWDataField.isArray()` method determines whether the data field is an array.
doMethod()	Processes work on the PE isolated region, usually with a void return type.	`VWWorkObject.doDispatch()` method saves changes made in the work object, unlocks it, and moves the current step to the next workflow step.
convertMethod()	Retrieves an object parameter corresponding to another one.	`VWSession.convertIdToUserName()` method converts the user ID assigned by PE to a user name.

5.1.4 Core classes

The following API classes are the most commonly used according to their typical purpose:

► PE session class
► PE error handling class
► Create and process work classes
► Search work classes
► Retrieve work status classes

The *PE session class* consists of the `VWSession object` used to establish a session and log on to a PE server. Several PE operations can be performed, such as querying rosters and queues, retrieving a list of rosters or queues, administering the system, and establishing an audit trail.

The *PE error handling class* consists of the `VWException` object, which handles all workflow-related exceptions.

The *create and process work classes* are used to create a workflow processes, get and assign parameter values to work items, and process work items across the workflow life cycle. This group includes the following classes:

► VWWorkObject
► VWStepElement
► VWParameter
► VWFieldType
► VWDataField

The *search work classes* are used to find work items either on a PE roster, work queue, or event log. This group includes the following classes:

► VWRoster
► VWRosterQuery
► VWRosterElement
► VWQueue
► VWQueueQuery
► VWQueueElement
► VWLog
► VWLogQuery
► VWLogElement

The *retrieve work status classes* are used to fetch step history data for workflow process associated with a workflow map and information related to milestone events. This group includes the following classes:

► VWProcess
► VWWorkflowDefinition
► VWWorkflowHistory
► VWStepHistory
► VWStepOcurrenceHistory
► VWStepWorkObjectHistory
► VWWorkflowMilestones
► VWMilestoneElement

5.1.5 Functional relationship

Figure 5-2 illustrates important functional relationships and calling sequences among the core PE Runtime API classes. The figure includes the core classes and methods that are the focus of this section.
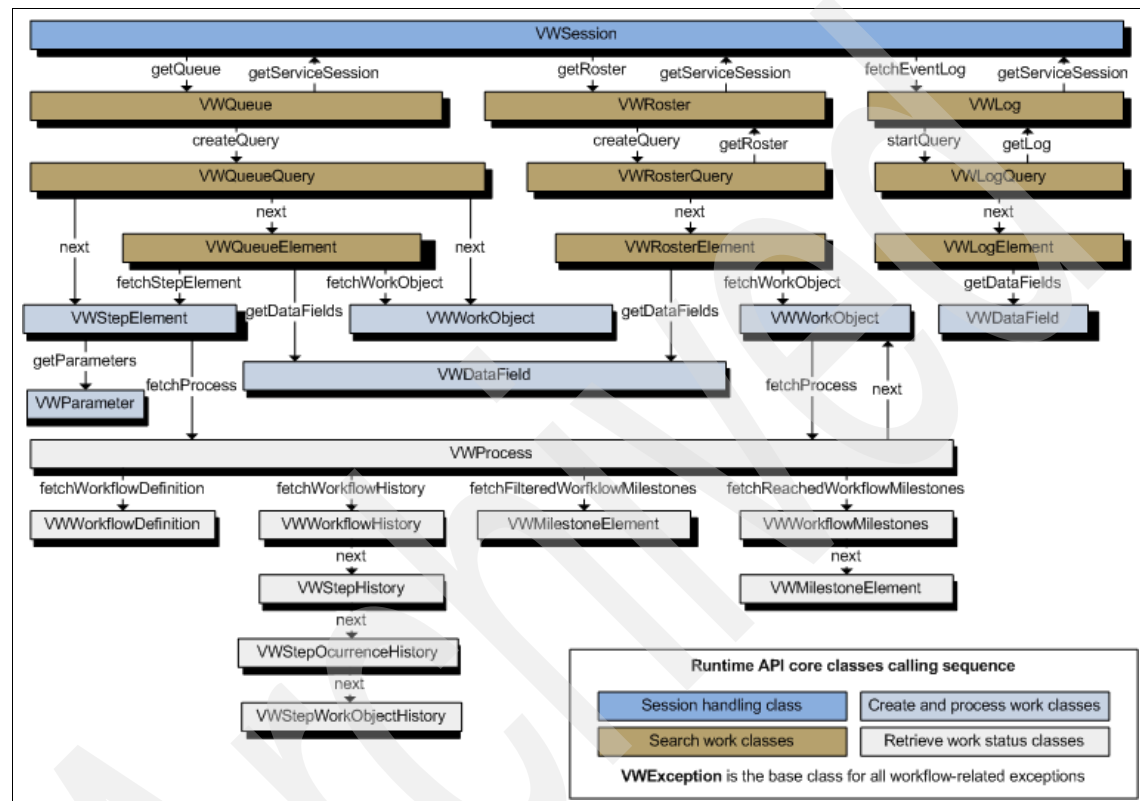

Figure 5-2   Runtime API core classes calling sequences

In the next section, we put in practice these API classes for process use cases.

5.2 Establishing a Process Engine session

To establish or log on to a PE session, the following elements are required:

- ► A connection point must be specified.
- ► The Content Engine (CE) host must be accessible.
- ► The CE host Uniform Resource Identifier (URI) must be set.

Note: The CE URI is not needed for Process Engine Web Service (PEWS) connections. The PEWS is deployed as part of the CE enterprise application.

Connection point

A connection point name identifies a specific isolated region in the workflow database connecting the PE API to an associated isolated region. A defined connection point consists of a PE server, communications port, and isolated region number.

For Process Engine Web Service (PEWS) clients, the PE connection point must be specified in the SOAP header request using the *router* SOAP header element. If no router value is specified, the default value is used. The default connection point for the Process Engine Web Service is *PEWSConnectionPoint*.

PE API usage tips: Performance improvements can be achieved by minimizing the number of Remote Procedure Calls (RPCs) to the PE. Methods that fetch data or cause certain action to occur typically cause RPCs to the PE server, whereas methods that get and set data are local to the object. Similarly, minimize logging on and logging off the PE.

5.2.1 Java API scenario

The `VWSession` object establishes a session and logs onto the PE. This object allows the caller to query rosters and queues, retrieve lists of roster and queue names, and administer the system.

PE relies on CE for authentication and directory service access operations. For further details about the available transport protocols, see the CE transport protocols from Chapter 3, "Introduction to Content Engine API programming" on page 39.

The PE Java API offers two data transport layers for connecting to CE, the native Enterprise JavaBeans (EJB) transport that is offered by the application server or the Content Engine Web Services (CEWS) transport.

Setting the Content Engine URI

The CE URI can be specified in any of the following ways:

- ► Call `VWSession.setBootstrapCEURI` to specify the URI as a String.

- ► Call `VWSession.setBootstrapConfiguration` to specify the URI as an InputStream.

- ► Specify the value of the system property `filenet.pe.bootstrap.ceuri.`

- ► Specify the value of the *RemoteServerUrl* property in a file named `WcmApiConfig.properties` located in your application's class path.

Table 5-4 shows the possible CE URIs to be used depending on the selected connection transport and application server where CE is deployed.

Table 5-4   Content Engine connection URIs

Protocol	URI
Web services	http://<server>:<port>/wsi/FNCEWS40MTOM/
EJB/WebSphere	iiop://<server>:<port>/FileNet/Engine
EJB/Weblogic	t3://<server>:<port>/FileNet/Engine
EJB/Jboss	jnp://<server>:<port>/FileNet/Engine

Note: You might sometimes see CE connection URIs with a prefix of `cemp:`. This prefix is for historical purposes only and is ignored by the PE APIs. When you are specifying URIs directly, there is no reason to use that prefix.

Authentication

This section assumes the authentication mechanism between CE and the directory service is properly configured.

Example 5-1 on page 121 shows how to create the `VWSession object`, set the CE URI, and log on to the PE passing the required parameters.

Example 5-1   Getting the initial connection using Java API

```
// User Information
String userName = "Administrator";
String password = "filenet";

// Connection Point
String connectionPoint = "CEPoint";

// Create a Process Engine Session Object
VWSession myPESession = new VWSession();

// Set Bootstrap Content Engine URI
myPESession.setBootstrapCEURI("iiop://ceserver:2809/FileNet/Engine");

// Log onto the Process Engine Server
myPESession.logon(userName, password, connectionPoint);
```

Session logoff

The session's logoff method ends the session with the PE and frees all resources.

Although the system calls this method internally, an explicit call makes finalization more immediate and certain, as shown in Example 5-2. Otherwise, finalization is uncertain even when the Java virtual machine (JVM) shuts down.

Example 5-2   Log off from Process Engine using Java API

```
// Log off from the Process Engine
myPESession.logoff();
```

5.2.2 PEWS API scenario

Unlike the PE Java API, the Process Engine Web Service (PEWS) API implements only the Web services transport for communicating with the CE. In fact, that connectivity is implicit because the PEWS server is collocated in the same J2EE application as the CEWS server.

The PEWS URL must be specified by the client in order to lookup for the PEWS Web Service Description Language (WSDL) available through the CE.

The URL for the Process Engine Web Service is as follows:

```
http://content_engine_host:port/wsi/ProcessEngineWS
```

The SOAP header request must include the correct elements and values, such as account credentials. These credentials are verified when functional operations are attempted. Example 5-3 shows how to initialize the PEWS object and set the correct parameter values including the request SOAP context.

Example 5-3   Creating a PEWS RequestSoapContext header using a C# client

```
// User Information
string userName = "Administrator";
string password = "filenet";

// Connection Point Value
string connectionPoint = "CEPoint";

// URL for the PEWS
string wsUrl = "http://ceserver:9080/wsi/ProcessEngineWS";

// PE Web Service Policy
private static Policy pePolicy = new Policy(new PolicyAssertion[]
                    {new PEAssertion(),new PESecurityAssertion()});

// Create the WS Service Port object
peWS.ProcessEngineServiceWse peWSServicePort = new
peWS.ProcessEngineServiceWse();

// Set PE Web Service Policy
peWSServicePort.SetPolicy(pePolicy);

// Set connection point name and wsURL parameters
peWSServicePort.RequestSoapContext["router"]=connectionPoint;
peWSServicePort.Url = wsUrl;

// Create User Context for the RequestSoapContext
SecurityToken tok = new UsernameToken(userName, password,
    PasswordOption.SendPlainText);
UserContext.SetProcessSecurityToken(tok);
```

In the example, the C# client application uses authentication helper classes to initiate a Process Engine Web Service Policy object and a UserContext object, which are required to interject the WS-Security and other SOAP headers for authentication purposes.

You can refer to these helper classes that are included with the pick-up and drop-off kiosk application as part of the provided sample application in this book.

See Chapter 7, "Sample applications for Fictional Auto Rental Company A" on page 223.

5.2.3 REST API scenario

For the REST API, there is no explicit PE logon action, however, the REST API resource requests require the calling client to be authenticated with the application server container. The HTTP Basic authentication is supported as the default authentication method. The HTTP Basic authentication passes credentials in the clear, so you should secure the connection with Transport Layer Security and Secure Sockets Layer (TLS/SSL) or other means.

> **Note:** The LDAP registry referenced must be the same registry used by the CE.

The client's authorization for accessing REST operations on a specified resource is determined by the permissions that are assigned to the resource.

5.3 Handling API exceptions

Exception handling is the way that applications detect and recover from exceptional conditions. Exceptional conditions are any unexpected occurrences that are not accounted for in a system's normal operation.

Exception handling techniques can be separated into two main categories:

- ► Expected exception handling
- ► Unexpected exception handling

In certain cases, expected exception handling is capable of doing *forward* error recovery, but both expected and unexpected exception handling methods can perform *backward* error recovery.

Forward error recovery can mask any exceptional occurrences and continue normal operation. Backward error recovery must halt normal system execution and attempt to return to a previous normal state to continue execution and retry the operation.

PE has two models to handle these exception categories for process applications, reflecting the C++ implementation of a server module and the Java implementation of the server and remaining modules. In this section, we describe how to handle PE Java exceptions.

5.3.1 VWException object

VWException is the base class for all workflow exceptions and is the only exception thrown by PE Java API method calls.

Table 5-5 shows the `VWException` main methods to get the String properties related to a PE Java exception.

Table 5-5   Principal String properties of the VWException object

Method	Description
VWException.getKey()	Gets the exception key, which is used in looking up a resource in the VWExceptions file. This method shows what interface produced the exception
VWException.getMessage()	Contains the associated string for the key from the exceptions resource file. Customized messages can be obtained from the exceptions resource file
VWException.getCauseClassName()	Returns the class name for the cause of the PE exception.
VWException.getCauseDescription()	Returns the description for the cause of the PE exception.

5.3.2 Steps to handle an exception

The following steps show how to handle a PE exception by using the Java API:

1. Include a try-catch block.
2. Call a method that can throw a `VWException` object.
3. Catch the exception, and identify and handle the error.

Example 5-4 shows how to handle an expected exception during PE logon. In this particular example, an empty password value is specified; therefore, an exception is thrown.

Example 5-4   Handle Process Engine exceptions using the Java API

```
// User Information
String userName = "Administrator";
String password = "";

// Connection Point
String connectionPoint = "CEPoint";
```

```
// Create a Process Engine Session Object
VWSession myPESession = new VWSession();

// Set Bootstrap Content Engine URI
myPESession.setBootstrapCEURI("iiop://ceserver:2809/FileNet/Engine");

// Start try-catch block to handle exception when log onto PE
try {
    myPESession.logon(userName, password, connectionPoint);
}
catch (VWException vwe) {
    System.out.println("\nVWException Key: " + vwe.getKey() + "\n");
    System.out.println("VWException Cause Class Name: "
    + vwe.getCauseClassName() + "\n");
    System.out.println("VWException CauseDescription: "
    + vwe.getCauseDescription()+ "\n");
    System.out.println("VWException Message: " + vwe.getMessage());
    // Perform error recovery
    ...
}
```

5.4 Launching a workflow

IBM FileNet Business Process Manager combines processes with active content management. This combination facilitates a variety of process management scenarios. Certain scenarios might require launching workflow processes automatically within applications.

The following steps retrieve and launch a workflow process using the PE Java and Web services APIs:

1. Create a VWSession object and log on to PE.

2. Retrieve the workflow definition (work class) names from the VWSession object.

3. Create the workflow process for the selected work class.

4. Get and set workflow parameters for the launch step (getting and setting parameters is described in 5.6, "Process work items" on page 140).

5. Dispatch the workflow launch step.

Steps to retrieve and launch a workflow using the PE REST API are described in 5.4.1, "REST API scenario" on page 127.

Example 5-5 and Example 5-6 show how to retrieve and create a workflow using both the Java API and a PEWS client.

Example 5-5   Create a workflow process using the Java API

```
// Create session object and log onto Process Engine
...

// Workflow name to launch
String workflowName = "myWorkflow";

// Retrieve transfered work classes
String[] workClassNames = myPESession.fetchWorkClassNames(true);
   for (int i=0;i<workClassNames.length;i++)
      System.out.println(workClassNames[i] );

// Launch Workflow
VWStepElement stepElement = myPESession.createWorkflow(workflowName);

// Get and Set Workflow parameters for the Launch Step
...

// Dispatch Worflow Launch Step
stepElement.doDispatch();
```

Example 5-6   Create a workflow process using PEWS

```
// Create the PEWS RequestSoapContext header
...

// Workflow name to launch
string workflowName = "myWorkflow";

// Get the available workclasses
ArrayList arrayWorkClasses = new ArrayList();
string[] workClassNames = peWSServicePort.getWorkClassNames();
foreach (string workClassName in workClassNames)
{
...
}

private peWS.StepElement stepElement;
stepElement = peWSServicePort.createWorkflow(workflowName);

// Get and Set Workflow parameters for the Launch Step
...
```

```
// Dispatch Worflow Launch Step
peWS.UpdateStepRequest updStepRequest = new peWS.UpdateStepRequest();
peWS.UpdateFlagEnum updFlagEnum  = peWS.UpdateFlagEnum.UPDATE_DISPATCH;
updStepRequest.stepElement = stepElement;
updStepRequest.updateFlag = updFlagEnum;
peWSServicePort.updateStep(updStepRequest);
```

5.4.1 REST API scenario

The following steps launch a workflow process by using the PE REST API:

1. Send HTTP GET using the `workclasses` resource to retrieve work class information.

2. Get and update work class launch step fields.

3. Send HTTP POST using the `workclasses` resource to create the workflow.

Note: For this operation, the Post Once Exactly (POE) parameter value must be set to 1 (POE:1) in both request headers.

Example 5-7 shows how to create a workflow process with a Dojo toolkit client.

Example 5-7   Create a workflow process with a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
// Set request URI
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
// Set REST API resource and wokflow name
var workflowName = "myWorkflow";
var url = baseURL +  "workclasses/" + workflowName;

var newWorkItemURL = "";
var defaultWorkObjectValue;

// Use HTTP GET Method to retrieve work class information
dojo.xhrGet({
        url: url,
        // Handle the return result as JSON object
        handleAs: "json-comment-optional",
        // Set the POE (POST Once Exactly) parameter in header to true
        headers: {"POE": "1"},
        // The response callback
        load: function(data, ioArgs) {
            // Get the resource URL for the new work object to create
            newWorkItemURL = ioArgs.xhr.getResponseHeader('POE-Links');
```

```
                // Get the work object's default values
                defaultWorkObjectValue = data;
        },
        error: function(data) {
            // The error callback
            console.dir(data);
        },
        sync: true
    }
);

// Set URL for POST request
url = baseURL + newWorkItemURL;
var workObject = defaultWorkObjectValue;

// Update work class lauch step fields
workObject.dataFields["CompanyName"].value = "IBM";

// Set the attachment value
workObject.attachments["Documents"].value =
[{"vsId":"{997E0AAC-8068-4990-AC08-D748E3063162}",
"title":"MyDocument1",
"type":3,
"libraryName":"objectstore",
"libraryType":3,
"version":"{D48EE0A9-61CA-4DAD-8B61-961CE362434B}",
"desc":"web.xml"}];

// Use HTTP POST Method to create the new work object
dojo.rawXhrPost({
        // the response callback
        url: url,
        handleAs: "json-comment-optional",
        headers: {"Content-Type": "application/json", "POE": "1"},
        _raw: true,
        // Post back the modified work object
        postData: dojo.toJson(workObject),
        load: function(data, ioArgs) {
            console.log(dojo.toJson(data));
        },
        error: function(data) {
            // the error callback
            console.log(data);
        },
        sync: true
    }
);
```

5.5 Search work items

All work searching operations are performed on PE database objects. The primary database objects are:

▶ VWRoster

A workflow roster stores the current location of and other information about all workflows. Workflow rosters provide the PE with an efficient way to locate specific workflows and work items.

▶ VWQueue

A queue contains each individual work item in a table that is waiting to be processed. Each entry in the roster has a corresponding entry in a queue. The API can retrieve and modify work items in work queues and user queues.

▶ VWLog

Event logs contain a record of specific system- or workflow-related events for each isolated region. This type of logging is useful for tracking workflow activity. Each event that is logged is part of an event category and has an associated event number.

These PE objects are stored as tables in the database. In addition, a set of views are created to enable direct database access from a custom application. In this section, we present how to search work by using the APIs (however, direct read-only access to the database views is supported).

When the PE database is accessed directly, it must only be for viewing information in a read-only manner. Any modification of table data without using the provided APIs or applications is not supported.

Work search operations that use the PE REST API are based on Workbasket properties. For further details about how to search work items by using the REST API, see 6.2.3, "Retrieve workbasket" on page 192. Also see 6.2.4, "Query work items from workBasket" on page 194.

To optimize PE queries:

- ► Set the appropriate query flags to return only user fields from roster, queue, and log queries. For example, to avoid retrieving system fields and helper data from the work items returned, set the query flags for the `queryFlags` parameter of `createQuery()`:
  - – QUERY_GET_NO_SYSTEM_FIELDS
  - – QUERY_GET_NO_TRANSLATED_SYSTEM_FIELDS

  Note that helper methods on elements are not functional without translated system fields.

- ► Create indexes and limit the number of indexes to achieve more efficient queries with less data transmitted. Indexes must be unique.

- ► Use filters to get ranges of elements for more efficient queries with less data transmitted. Use filters to limit elements retrieved to a specified range or to set a maximum number of objects to be retrieved for each fetch from the PE server (default is 50 items).

Create query parameters

To successfully return results from any of the PE database objects, the fetch criteria must be correctly specified by the input query parameters. Table 5-6 describes the parameters included in the `createQuery` method to query a PE database object by using the APIs.

Table 5-6   Parameters to build a query by using the APIs

Parameter	Description
indexName	Specify the name of the database search index for the queue.
minValues	Specify an array of objects containing the minimum values with which to compare the index fields.
maxValues	Specify an array of objects containing the maximum values with which to compare the index fields.
queryFlags	Specify a value for the search options
filter	Specify a SQL WHERE clause to be used as a filter; has placeholder variables that are replaced by values in substitutionVars parameter.
substitutionVars	Specify an array of objects containing the values to substitute for the placeholder variables specified in filter.
fetchType	Specify a value associated with the element type to be retrieved using the object returned by the query object.

> **Note:** Use `VWWorkObjectNumber` class as a substitution value type for queries when using `F_WobNum` or `F_WorkFlowNumber`. The type is substituted into the filter string specified. This class takes care of formatting the work object or workflow number so the developer does not have to.

Query flags

These flags represent an integer value that specifies the search options. Setting this parameter refines the search for the work items that are specified by the search index. Table 5-7 shows the available query flags for work search.

Table 5-7   Process Engine query flag options for work search

Query flag	Integer value	Description
QUERY_GET_NO_SYSTEM_FIELDS	1024	Returns objects without system fields.
QUERY_GET_NO_TRANSLATED_SYSTEM_FIELDS	2048	Returns objects without translated fields.
QUERY_MAX_VALUES_INCLUSIVE	64	Specifies the inclusion of maximum search values.
QUERY_MIN_VALUES_INCLUSIVE	32	Specifies the inclusion of minimum search values.
QUERY_NO_OPTIONS	0	Specifies no search option.
QUERY_READ_UNWRITABLE	4	Specifies the return of read-only work items.
QUERY_RESOLVE_NAMES	8192	Supports conversion of user names to IDs for Process Engine Web services to perform queries using user ID fields within an index or filter.

5.5.1 Query a roster

The following steps retrieve work items from a workflow roster by using the PE Java and Web services APIs:

1. Create a `VWSession` object and log on to PE.
2. Retrieve the roster to search and the roster count.
3. Set query parameters.
4. Perform the query.
5. Process the results.

To optimize roster queries:

- ► Expose only infrequently updated fields. Exposed data fields on rosters are more costly for updates.

- ► Set an optimal buffer size on the roster-related queries. For example, consider increasing the buffer size (default is 50, maximum is 200) to minimize round-trips to the PE server.

- ► Query roster elements instead of work objects. Query work objects through the roster causes the application to access database records twice instead of once.

Example 5-8 and Example 5-9 on page 133 show how to retrieve work items from a workflow roster by using both the Java API and a PEWS client.

Example 5-8   Query workflow roster using the Java API

```
// Create session object and log onto Process Engine
...

// Set Roster Name
String rosterName= "DefaultRoster";

// Retrieve Roster Object and Roster count
VWRoster roster = myPESession.getRoster(rosterName);
System.out.println("Workflow Count: " + roster.fetchCount());

// Set Query Parameters
int queryFlags=VWRoster.QUERY_NO_OPTIONS;
String queryFilter="F_WobNum=:A";
String wobNum = "5BDD567B70453C48A5388AC56ABC69E4";
// VWWorkObjectNumber class takes care of the value format
// used in place of F_WobNum and F_WorkFlowNumber
Object[] substitutionVars = {new VWWorkObjectNumber(wobNum)};
int fetchType = VWFetchType.FETCH_TYPE_ROSTER_ELEMENT;

// Perform Query
VWRosterQuery query = roster.createQuery
(null,null,null,queryFlags,queryFilter,substitutionVars,queryType);

// Process Results
while(query.hasNext()) {
    VWRosterElement rosterItem = (VWRosterElement) query.next();

    System.out.println("WF  Number: "+rosterItem.getWorkflowNumber());
```

```
    System.out.println("WOB Number:
    "+rosterItem.getWorkObjectNumber());
    System.out.println("F_StartTime: " +
    rosterItem.getFieldValue("F_StartTime"));
    System.out.println("F_Subject: " +
    rosterItem.getFieldValue("F_Subject"));
}
```

Example 5-9   Query workflow roster using PEWS

```
// Create the PEWS RequestSoapContext header
...

// Set Roster Name
string rosterName= "DefaultRoster";

// Set Query Parameters
peWS.GetRosterElementsRequest query = new peWS.GetRosterElementsRequest();
query.queryFlags = new peWS.QueryFlagEnum?[]
{peWS.QueryFlagEnum.QUERY_READ_LOCKED, peWS.QueryFlagEnum.QUERY_LITE_ELEMENTS};
query.nToReturn = 100;
query.rosterName = rosterName;

// Perform Query
peWS.RosterElement[] elements = peWSServicePort.getRosterElements(query);

// Process Results
foreach (peWS.RosterElement element in elements)
{
    peWS.Field[] fields = element.Fields;

    foreach(peWS.Field field in fields)
    {
        // Only process the field if there is a value
        if (field.Values.Length > 0)
        {
            // Process field values
            //field.Name;
            //field.Values[0].Item.GetType();
            //field.Values[0].Item;
        }
    }
}
```

5.5.2 Workflow queues

A workflow queue contains each individual work item in a table that is waiting to be processed. Each entry in the roster has a corresponding entry in a queue. The three workflow queue categories are:

► User queues

   Each user has an inbox that holds work items assigned to that user. A user can also have a queue of tracker items.

► Work queues

   A work queue holds work items that can be completed by one of a number of users, rather than by a specific participant, or work items that can be completed by an automated process.

► System queues

   System queues (Conductor, Delay, and InstructionSheetInterpreter) hold work items that are undergoing system processes or are waiting.

> **Note:** The following cases describe the differences between the `Inbox` and the `Inbox(0)` user queues:
>
> ► Use `Inbox` queue to retrieve the inbox work items of the user that is currently logged on.
>
> ► Use `Inbox(0)` queue to retrieve inbox work items of all users.
>
> ► Use `Inbox(0)` queue and `F_BoundUser` system field in the query filter to retrieve inbox work items of a specific user (or users).

Query queues

The following steps retrieve work items from a workflow queue using the PE Java and Web services APIs:

1. Create a `VWSession` object and log on to PE.
2. Retrieve the queue to search.
3. Get the queue depth.
4. Set the query parameters.
5. Perform the query.
6. Process the results returned by the query.

The same steps are included to retrieve work items from a workflow queue for all queue categories described above.

Refer to Table 5-6 on page 130 for a detailed description of the required parameters that are included in the `createQuery` method to query a queue by using the PE APIs.

To optimize queue queries:

► Use the `QUERY_LOCK_OBJECTS` flag. This avoids the necessity of an additional RPC to check the lock status of an object (or to lock an object). Using the flag also reduces lock contention.

► Set an optimal buffer size on the query. For example, consider increasing the buffer size (default is 50, maximum is 200) to minimize round-trips to the PE server.

► Partition, if possible, to use more than one queue.

Example 5-10 and Example 5-11 on page 136 show how to retrieve work items from a workflow queue using both the Java API and a PEWS client.

Example 5-10   Query workflow queue using the Java API

```
// Create session object and log onto Process Engine
...

// Queue Name
String queueName = "Inbox";

// Retrieve the Queue to be searched and Queue depth
VWQueue queue = myPESession.getQueue(queueName);
System.out.println("Queue Depth: " + queue.fetchCount());

// Set Query Parameters
String wobNum = "5BDD567B70453C48A5388AC56ABC69E4";
VWWorkObjectNumber wob = new VWWorkObjectNumber(wobNum);
VWWorkObjectNumber[] queryMin = new VWWorkObjectNumber[1];
VWWorkObjectNumber[] queryMax = new VWWorkObjectNumber[1];

String queryIndex = "F_WobNum";
queryMin[0] = wob;
queryMax[0] = wob;
int queryFlags = VWQueue.QUERY_MIN_VALUES_INCLUSIVE +
VWQueue.QUERY_MAX_VALUES_INCLUSIVE;
int fetchType = VWFetchType.FETCH_TYPE_QUEUE_ELEMENT;

// Perform Query
VWQueueQuery queueQuery = queue.createQuery(queryIndex,queryMin,
queryMax,queryFlags,null,null,fetchType);

// Process Results
while(queueElement.hasNext()) {
```

```
            VWQueueElement queueElement = (VWQueueElement) queueQuery.next();

            System.out.println("Username: "+
            queueElement.getFieldValue("F_BoundUser").toString());
            System.out.println("WOB Number: "+
            queueElement.getWorkObjectNumber());
            System.out.println("F_Subject: " +
            queueElement.getFieldValue("F_Subject"));
}
```

Example 5-11   Query workflow queue using PEWS

```
// Create the PEWS RequestSoapContext header
...

// Set Queue Name
string queueName= "Inbox";

// Set Query Parameters
peWS.GetQueueElementsRequest query = new peWS.GetQueueElementsRequest();
query.queryFlags = new peWS.QueryFlagEnum?[1] {
peWS.QueryFlagEnum.QUERY_READ_LOCKED };
query.nToReturn = 100;
query.queueName = queueName;

// Perform Query
peWS.QueueElement[] queueelements = peWSServicePort.getQueueElements(query);

// Process Results
foreach (peWS.QueueElement queueelement in queueelements)
{
    peWS.Field[] fields = queueelement.Fields;

    foreach(peWS.Field field in fields)
    {

        // Only process the field if there is a value
        if (field.Values.Length > 0)
        {
            // Process field values
            //field.Name;
            //field.Values[0].Item.GetType();
            //field.Values[0].Item;
        }
    }
}
```

5.5.3 Query event log

The Process Configuration Console automatically creates an event log (that is called `DefaultEventLog`) for each isolated region including a set of event-logging options. The logging options determine whether PE logs a message when certain events occur within the region. Each event-logging option represents an event category; if enabled, the occurrence of any event within that category generates a log message.

> **Note:** A custom (user-defined) message can also be logged if a `Log` system function is executed within a workflow.

Event-logging categories

Each system event that is logged is part of an event category, and has an associated event number. Table 5-8 shows the primary event-logging categories used for workflow event tracking. For further details about all event-logging categories available, refer to the following P8 documentation path: **ECM Help → User Help → Integrating workflow → Workflow overview → Process Engine Reference → Events and statistics → Event log categories.**

Table 5-8   Primary event logging categories

Event log category	Event logged	Event number	Description
Creation	VW_WOChildCreationMsg	130	Records the creation of a child work item.
	VW_WOParentCreationMsg	140	Records the creation of a parent work item.
Termination	VW_WOChildTerminationMsg	150	Records the termination of a child work item.
	VW_WOParentTerminationMsg	160	Records the termination of a parent work item.
	VW_WFTermination	165	Records the completion of all work items in a workflow.
Administration message	VW_WOForcedToSkipInstructionMsg	180	Records when a work item is forced to skip an instruction.
	VW_WOForcedToTerminateMsg	190	Records when a work item is forced to terminate.
	VW_WOForcedToDeleteMsg	200	Records when a work item is forced to delete.

Event log category	Event logged	Event number	Description
Begin operation	VW_WPBeginServiceMsg	350	Records when a step processor or user locks a work item.
	VW_WPWorkObjectQueuedMsg	352	Records when a work item is queued.
End operation	VW_WPEndServiceNormalMsg	360	Records when a work item is updated and dispatched to the next queue.
	VW_WPWOBSaveWithLockMsg	365	Records when a work item is saved while retaining the existing lock.
	VW_WPEndServiceAbnormalMsg	370	Records when work item processing ends abnormally.
	VW_WPEndServiceReleaseMsg	380	Records when a work item is updated and unlocked in the same queue.
	VW_WPEndServiceReleaseDelegateMsg	382	Records when a work item is delegated to another user.
	VW_WPEndServiceReleaseReassignMsg	384	Records when a work item is reassigned to another user.
	VW_WPEndServiceReleaseReturnMsg	386	Records when a work item is returned to a user following delegation.
	VW_WPEndServiceAbortMsg	390	Records when a user cancels, stops, or ends an operation.
Empty step / System step	VVW_WOEmptyStepMsg	500	Records when a step is executed without specifying a queue.
	VW_WOCompleteSysStepMsg	510	Records when a compound step is completed.

The following steps retrieve work items from an event log using the PE Java API:

1. Create a `VWSession` object and log on to PE.
2. Retrieve the event log to search.
3. Set the query parameters.

4. Perform the query and count the elements returned.
5. Process the results.

Refer to Table 5-6 on page 130 for a detailed description of the required parameters included in the `createQuery` method to query a queue using the PE APIs.

To optimize event log queries:

▶ Partition to use different logs for different workflows.

▶ Disable any logging options that you do not need. Note that although this improves performance, it also affects the Process Tracker and Process Analyzer.

▶ Manage log records. Use the `vwlog` tool to maintain logs. Use operating system scheduling tools to periodically remove unused log records.

▶ Save log information in text files.

Example 5-12 shows how to retrieve work items from an event log by using the Java API.

Example 5-12   Query event log by using the Java API

```
// Create session object and log onto Process Engine
...
// Set Event Log Name
String eventLogName = "DefaultEventLog";

// Retrieve the Event Log to be searched
VWLog log = myPESession.fetchEventLog(eventLogName);

// Set Query Parameters
String queryFilter = "F_EventType = 240"; // user logged in event type
int queryFlags = VWLog.QUERY_NO_OPTIONS;

// Perform Query
VWLogQuery query = log.startQuery(null,null,null,queryFlags,queryFilter,null);

// Process Results
VWLogElement logElement = (VWLogElement) query.next();

int eventType = logElement.getEventType();
String stringEventType = VWLoggingOptionType.getLocalizedString(eventType);
String[] fieldNames = logElement.getFieldNames();
String FieldValue = logElement.getFieldValue(fieldNames[0]).toString();
```

5.6 Process work items

Process applications enable users to manage information and access resources that are associated with a workflow. Developing a process application involves, among other things, accessing process data and resources, and interfacing to the PE to perform tasks that are associated with a workflow step. These tasks are usually performed by either a step processor or a work performer application.

► Step processor

  A step processor is an application that provides the information and resources for a participant to complete a step in a workflow. When a participant opens a work item at run time, the step processor displays the necessary instructions, attachments, field values, response options, and other resources that the participant needs to complete the work.

► Work performer

  A work performer is an application that performs an operation or set of operations that are associated with a workflow. Typically, work performers are designed without a user interface and are used to perform automatic workflow operations, such as those that are associated with a specific step in a workflow definition.

These process applications must perform fundamental operations, which include:

► Retrieve step element.
► Get step element parameters.
► Set step element parameter values.
► Complete the work item.

5.6.1 Retrieve step element

To process a work item in a process application, the appropriate step element must be retrieved from the queue.

The following steps are required to retrieve a step element from a workflow queue by using the PE Java and Web services APIs:

1. Create a `VWSession` object and log on to PE.
2. Get the queue containing the work item.
3. Set the query parameters to get step elements.
4. Perform the queue query.
5. Get the desired step element from the queue.

For further details about steps to retrieve a step element using the PE REST API, see "Retrieve step element: REST API scenario" on page 142.

Example 5-13 and Example 5-14 show how to retrieve a step element from a user queue by using both the Java API and a PEWS client.

Example 5-13   Retrieve a step element from the Inbox by using the Java API

```
// Create session object and log onto Process Engine
...
// Queue Name
String queueName = "Inbox";

// Retrieve the Queue
VWQueue queue = myPESession.getQueue(queueName);

// Set Query Parameters
String wobNum = "5BDD567B70453C48A5388AC56ABC69E4";
VWWorkObjectNumber wob = new VWWorkObjectNumber(wobNum);
VWWorkObjectNumber[] queryMin = new VWWorkObjectNumber[1];
VWWorkObjectNumber[] queryMax = new VWWorkObjectNumber[1];

String queryIndex = "F_WobNum";
queryMin[0] = wob;
queryMax[0] = wob;

// Query Flags and Type to retrieve Step Elements
int queryFlags = VWQueue.QUERY_READ_LOCKED;
int queryType = VWFetchType.FETCH_TYPE_STEP_ELEMENT;
VWQueueQuery queueQuery = queue.createQuery
(queryIndex,queryMin,queryMax,queryFlags,null,null,queryType);

// Get an individual Step Element
VWStepElement stepElement = (VWStepElement) queueQuery.next();
```

Example 5-14   Retrieve a step element from the Inbox using PEWS

```
// Create the PEWS RequestSoapContext header
...
// Set Queue Name
string queueName= "Inbox";

// Set Query Parameters
string wobNum= "5BDD567B70453C48A5388AC56ABC69E4";
peWS.RetrieveStepRequest request = new peWS.RetrieveStepRequest();
request.bLock = true;
request.bOverrideLock = true;
request.queueName = queueName;
request.wobNum = wobNum;
```

```
peWS.GetStepElementsFromQueueRequest qry = new
peWS.GetStepElementsFromQueueRequest();
request.queryFlags = new peWS.QueryFlagEnum?
[]{peWS.QueryFlagEnum.QUERY_NO_OPTIONS};

// Perform Query and Retrieve Step Element
peWS.StepElement = peWSServicePort.retrieveStep(request);
```

Retrieve step element: REST API scenario

The following steps retrieve a step element by using the PE REST API:

1. Send HTTP GET by using the stepelements resource to retrieve step element information, specifying queue name and work object number.

2. Process the response results.

Example 5-15 shows how to retrieve a step element from the Inbox with Dojo toolkit client.

Example 5-15 Retrieve a step element from the Inbox with Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
// Set request URI
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";

// Set REST API resource and work item number
var queueName = "Inbox";
var wobNumber = "5BDD567B70453C48A5388AC56ABC69E4";
var url = baseURL +  "queues/" +
queueName + "/stepelements/" + wobNumber;

var stepElement;
// Use HTTP GET method to retrieve the step element by specifying
// the queue name and work object number
dojo.xhrGet({
        url: url,
        // Handle the result as JSON object
        handleAs: "json-comment-optional",
        // the callback to handle the response
        load: function(data) {
            stepElement = data;
            // Process the attachments
            for (var attachmentName in stepElement.attachments) {
                // ....
                console.log(attachmentName);
            }
```

```
            // Process the data fields
            for (var dataFieldName in stepElement.dataFields) {
                // ....
                console.log(dataFieldName);
            }
            // Process the system properties
            for (var systemPropName in stepElement.systemProperties) {
                // ....
                console.log(systemPropName);
            }

            // Process step processor
            //var stepProcessor = stepElement.stepProcessor;
        },
        error: function(data) {
            // the error callback
            console.dir(data);
        }
    }
);
```

5.6.2 Get step element parameters

To process a work item from a process application, current information included
in the step element must be retrieved so that it can be displayed to the participant
and to perform business rules that are based on the field properties.

Table 5-9 on page 144 presents the required primary field flags necessary to
properly handle step element and work item fields.

Table 5-9   Primary process field flags

Field flag	Values	Description
Field type	Integer (FIELD_TYPE_INT) String (FIELD_TYPE_STRING) Boolean (FIELD_TYPE_BOOLEAN) Float (FIELD_TYPE_FLOAT) Time (FIELD_TYPE_TIME) Attachment (FIELD_TYPE_ATTACHMENT) Participant (FIELD_TYPE_PARTICIPANT) All types (ALL_FIELD_TYPES)	An integer value that specifies the parameter type or types
Field creator	User (FIELD_USER_DEFINED) System (FIELD_SYSTEM_DEFINED) User and system (FIELD_USER_AND_SYSTEM_DEFINED)	An integer that specifies one of the field creator types
Field mode	Read only (MODE_TYPE_IN) Write only (MODE_TYPE_OUT) Read and write (MODE_TYPE_IN_OUT)	An integer value that indicates the mode of the step parameter

The following steps get step element parameters by using the PE Java and Web services APIs:

1. Retrieve the user- and system-defined `VWParameters`.
2. Get the name, data type, read and write mode, and value for each parameter.
3. Retrieve the step element responses.

For further details about included steps to retrieve step element parameters using the PE REST API, see "Get step element parameters: REST API scenario" on page 146.

Example 5-16 and Example 5-17 on page 145 show how to retrieve step element parameters and responses using both the Java API and a PEWS client.

Example 5-16   Get step element parameters and responses using the Java API

```
// Create session object and log onto Process Engine
...
// Retrieve Step Element
...

// Specify Field Type and Field Creator flags to retrieve
// user- and system-defined parameters
VWParameter[] parameters = stepElement.
getParameters(VWFieldType.ALL_FIELD_TYPES,
VWStepElement.FIELD_USER_AND_SYSTEM_DEFINED);
```

```
// Get the name, type, mode, and value for each parameter
for (int i = 0; i < parameters.length; i++ ) {
   // Get Parameter Name
   String parameterName = parameters[i].getName();
   // Get Parameter Type (Field Type Flag)
   String parameterType = VWFieldType.getLocalizedString
   (parameters[i].getFieldType());
   // Get Parameter Mode (Field Mode Flag)
   String parameterMode = VWModeType.getLocalizedString
   (parameters[i].getMode());
   // Get Parameter Value
   String parameterValue =parameters[i].getStringValue();
}

// Retrieve Step Element Responses
String[] stepResponses = stepElement.getStepResponses();
if(stepResponses != null) {
   int len = stepResponses.length;
   for(int j = 0; j < len; j++ ) {
      // Process Response
      System.out.println(stepResponses[j]);
   }
}
```

Example 5-17   Get step element parameters and responses using PEWS

```
// Create the PEWS RequestSoapContext header
...
// Retrieve Step Element
...

// Retrieve Step Element Parameters
peWS.Parameter[] parameters = stepElement.Parameters;

// Get the name, type, mode, and value for each parameter
foreach(peWS.Parameter param in parameters)
{
   // Get Parameter Name
   string paramName = param.Name;
   // Get Parameter Type (peWS.FieldTypeEnum)
   string paramType = param.Type;
   // Get Parameter Mode (peWS.ModeTypeEnum)
   string paramMode = param.Mode;
```

```
    // Get Parameter Value
    if( param.Values.Length > 0)
    {
        foreach( peWS.Value stepvalue in param.Values)
        {
            string paramValue = stepvalue.Item;
        }
    }
}

// Retrieve Step Element Responses
peWS.ArrayOfResponse stepResponses = stepElement.Responses;
if (stepResponses!=null)
{
    foreach (string response in stepResponses.Response)
    {
        // Process Response
        System.Windows.Forms.
        MessageBox.Show("Step Element Response: "+response);
    }
}
```

Get step element parameters: REST API scenario

The following steps retrieve step element parameters by using the PE REST API:

1. Send HTTP GET using the `stepelements resource` to retrieve the step element specifying queue name and work object number.

2. Process response results.

Example 5-18 shows how to get step element parameters and responses with a Dojo toolkit client.

Example 5-18   Get step element parameters and responses with a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
// Set request URI
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";

// Set REST API resource and work item number
var queueName = "Inbox";
var wobNumber = "5BDD567B70453C48A5388AC56ABC69E4";
var url = baseURL +  "queues/" +
queueName + "/stepelements/" + wobNumber;

// Helper funtion to print parameter properties
```

```
var printParameter = function(parameter) {
    // get the attachment name
    var name = parameter.name;
    // get the mode
    var mode = parameter.mode;
    // get the type
    var type = parameter.type;
    // get the value
    var value = parameter.value;

    console.log("Attachment " + name + ": ");
    console.log(" - Mode: " + mode);
    console.log(" - Type: " + type);
    console.log(" - Value: " + value);
};

// Use HTTP GET method to retrieve the step element by specifying
// the queue name and work object number
dojo.xhrGet({
        url: url,
        handleAs: "json-comment-optional",
        load: function(data) {
            // the callback to handle the response
            stepElement = data;
            var attachments = stepElement.attachments;
            var dataFields = stepElement.dataFields;
            var stepProcessor = stepElement.stepProcessor;
            var systemProperties = stepElement.systemProperties;

            // print the attachment parameters
            for (var attachmentName in attachments) {
                var attachment = attachments[attachmentName];
                printParameter(attachment);
            }

            // print the dataField parameters
            for (var dataFieldName in dataFields) {
                var dataField = dataFields[dataFieldName];
                printParameter(dataField);
            }

            // print the systemField parameters
            for (var systemPropName in systemProperties) {
                var systemPropValue = systemProperties[systemPropName];
                console.log("systemProperty (" + systemPropName + "):"
```

```
                              + systemPropValue);
                 }

                 // print the step responses
                 var responses = systemProperties.responses;
                 for (var i = 0; i < responses.length; i++) {
                     console.log("response (" + i + "):" + responses[i]);
                 }

            },
            error: function(data) {
                // the error callback
                console.dir(data);
            }
        }
    }
);
```

5.6.3 Set step element parameter values

To process work from a process application, step elements have to be locked in order to set parameter values, and either save or complete the work.

Besides the common field types that comprise a workflow process, two workflow parameters require additional handling in order to set their values programmatically: workflow attachments and participants. In this section, we show how to get and set values for the common workflow field types, and for workflow attachments and participants:

► Workflow attachments

This parameter specifies the workflow attachments that a participant can use to complete a step in the workflow. These attachments can be objects such as documents, document arrays, folders, stored searches, URLs, or files that are located on a shared file system.

► Workflow participants

This parameter specifies the participants who can process work in the workflow. Participants can be a user, users, a group, or groups. When participants are assigned to a particular step in the workflow, and there are active work items in the step, the work items appear in the inbox of the participants.

A workflow attachment can be a document, folder, or other type of objects. Table 5-10 on page 149 lists the workflow attachment types.

Table 5-10 Workflow attachment types

Attachment type	Integer value	Description
ATTACHMENT_TYPE_CUSTOM_OBJECT	6	Indicates the attachment type is a custom object
ATTACHMENT_TYPE_DOCUMENT	3	Indicates the attachment is the document type
ATTACHMENT_TYPE_FOLDER	2	Indicates the attachment is the folder type
ATTACHMENT_TYPE_LIBRARY	1	Indicates the attachment type is library or Object Store
ATTACHMENT_TYPE_STORED_SEARCH	4	Indicates the attachment is the stored search type
ATTACHMENT_TYPE_UNDEFINED	0	Indicates the attachment type is not known
ATTACHMENT_TYPE_URL	5	Indicates the attachment type is a UNC or URL

Note: The `VWParticipant` object represents the name of a user or user group in any of the following ways:

► Short (logon) name
► Distinguished name (DN)
► Display name (configurable)

For a detailed description of the primary field flags required to properly handle step element and work item fields, see the list in Table 5-9 on page 144.

The following steps are required to update step element parameter values using the PE Java and Web services APIs:

1. Lock the step element.
2. Check whether the parameter is read only.
3. Switch through each data type for editable parameters.
4. For each data type, check whether the parameter is single or an array.
5. Set the value for the parameter.
6. Set the value for the system-defined response parameter.

For further details about included steps to update step element parameter values by using the PE REST API, see the "Set step element parameter values: REST API scenario" on page 154.

Example 5-19 and Example 5-20 on page 152 show how to perform the steps required to set step elements parameter values, using both the Java API and a PEWS client.

Example 5-19   Set step element parameters and responses by using the Java API

```
// Create session object and log onto Process Engine
...
// Retrieve Step Element
...
// Retrieve Step Element Parameters
...
// Lock the Step Element
stepElement.doLock(true);

// Process Step Element Parameters
for (int i = 0; i < parameters.length; i++ ) {

    // Check parameter mode
    boolean readOnly =
    (parameters[i].getMode() == VWModeType.MODE_TYPE_IN);
    // If the parameter is editable, switch through each data type
    if (!readOnly) {

        // For each data type,
        // check whether the parameter is single or an array
        // and set the parameter value(s)
        switch (parameters[i].getFieldType()) {

            case VWFieldType.FIELD_TYPE_STRING:
                if (parameters[i].isArray()) {
                    String[] arrParamValues =
                    new String[] {"value_1", "value_2", "value_3"};
                    stepElement.setParameterValue
                    (parameters[i].getName(),arrParamValues,true);
                } else {
                    String paramValue = "value_1";
                    stepElement.setParameterValue
                    (parameters[i].getName(),paramValue,true);
                }
                break;
```

```java
          case VWFieldType.FIELD_TYPE_ATTACHMENT:
             if (!parameters[i].isArray()) {
                // Get the value for the VWAttachment
                VWAttachment attachment =
                (VWAttachment) parameters[i].getValue();
                // Set the attachment name
                attachment.setAttachmentName("Document Title");
                // Set the attachment description
                attachment.setAttachmentDescription
                ("A document added programmatically");
                // Set the type of object (Document)
                attachment.setType
                (VWAttachmentType.ATTACHMENT_TYPE_DOCUMENT);
                // Set the library type and name (CE Object Store)
                attachment.setLibraryType
                (VWLibraryType.LIBRARY_TYPE_CONTENT_ENGINE);
                attachment.setLibraryName("ObjectStoreName");
                // Set the document ID and version
                attachment.setId
                ("{BBE5AD7F-2449-4DC3-AA38-012A65EC4286}");
                attachment.setVersion
                ("{BBE5AD7F-2449-4DC3-AA38-012A65EC4286}");
                // Set the parameter value
                stepElement.setParameterValue
                (parameters[i].getName(),attachment,true);
             }
             break;

          case VWFieldType.FIELD_TYPE_PARTICIPANT:
             // Instantiate a new VWParticipant array
             VWParticipant[] participant = new VWParticipant[1];
             // Set the participant name using username value
             String participantUserName = "Administrator";
             participant[0].setParticipantName(participantUserName);
             // Set the parameter value
             stepElement.setParameterValue
             (parameters[i].getName(),participant,true);
             break;

       default:
          // Do not take action for other data types
          break;

    }
}
```

```
    }

    // Set the value for the system-defined Response parameter
    if (stepElement.getStepResponses() != null) {
        String responseValue = "Ok";
        stepElement.setSelectedResponse(responseValue);
    }
```

Example 5-20   Set step element parameters and responses by using PEWS

```
// Create the PEWS RequestSoapContext header
...
// Retrieve and Lock Step Element
// request.bLock = true;
// request.bOverrideLock = true;
...
// Retrieve Step Element Parameters

// Process Step Element Parameters
foreach(peWS.Parameter param in parameters)
{
    // If the parameter is editable, switch through each data type
    bool readOnly = param.Mode == peWS.ModeTypeEnum.MODE_TYPE_IN;
    if (!readOnly)
    {

        switch(param.Type)
        {
            case peWS.FieldTypeEnum.FIELD_TYPE_STRING:
                // Set the parameter value
                string paramValue = "value_1";
                peWS.Value stepvalue = new peWS.Value();
                peWS.Value[] values = {paramValue};
                param.Values = values;
                param.Modified = true;
                break;

            case peWS.FieldTypeEnum.FIELD_TYPE_ATTACHMENT:
                // Get the value for the attachment object
                peWS.Attachment peAttachment =
                (peWS.Attachment)stepvalue.Item;
                // Set the attachment name
                peAttachment.Name = "Document Title";
                // Set the attachment description
                peAttachment.Description =
```

```
                    "A document added programmatically";
                    // Set the library type and name (CE Object Store)
                    peAttachment.LibraryType =
                    peWS.LibraryTypeEnum.LIBRARY_TYPE_CONTENT_ENGINE;
                    peAttachment.Library = "ObjectStoreName";
                    // Set the type of object (Document)
                    peAttachment.Type =
                    peWS.AttachmentTypeEnum.ATTACHMENT_TYPE_DOCUMENT;
                    // Set the document ID and version
                    peAttachment.Id = "{BBE5AD7F-2449-4DC3-AA38-012A65EC4286}";
                    peAttachment.Version =
                    "{BBE5AD7F-2449-4DC3-AA38-012A65EC4286}";
                    // Set the parameter value
                    peWS.Value stepvalue = new peWS.Value();
                    peWS.Value[] values = {peAttachment};
                    param.Values = values;
                    param.Modified = true;
                    break;

            case peWS.FieldTypeEnum.FIELD_TYPE_PARTICIPANT:
                    // Set the participant name using username value
                    string participantUserName = "Administrator";
                    // Set the parameter value
                    peWS.Value stepvalue = new peWS.Value();
                    peWS.Value[] values = {participantUserName};
                    param.Values = values;
                    param.Modified = true;
                    break;

            default:
                    // Do not take action for other data types
                    break;
        }
    }
}

// Set the value for the system-defined Response parameter
peWS.ArrayOfResponse stepResponses = stepElement.Responses;
if (stepResponses!=null)
{
    string responseValue = "Ok";
    stepResponses.Selected = responseValue;
}
```

Set step element parameter values: REST API scenario

The following steps are required to update step element parameter values using the PE REST API:

1. Send HTTP GET using the `stepelements resource` to retrieve the step element and `eTag` parameter.

2. Send HTTP PUT using the `stepelements` resource to lock the step element.

3. Set step element parameter values and response.

Example 5-21 shows how to set step element parameters and responses with a Dojo toolkit client.

Example 5-21   Set step element parameters and responses with a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
// Set request URI
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";

// Set REST API resource and work item number
var queueName = "Inbox";
var wobNumber = "5BDD567B70453C48A5388AC56ABC69E4";
var url = baseURL +  "queues/" +
queueName + "/stepelements/" + wobNumber;

var eTag;
var stepElement;

// Use the HTTP GET method to retrieve the step element and eTag
dojo.xhrGet({
        url: url,
        handleAs: "json-comment-optional",
        // The callback to handle the response
        load: function(data, ioArgs) {
            // Get the step element
            stepElement = data;
            // Get the eTag
            eTag = ioArgs.xhr.getResponseHeader("ETag");
        },
        // The error callback
        error: function(data) {
            console.dir(data);
        },
        sync: true

// Use the HTTP PUT method to lock the step element
```

```
var urlForLock = url + "?action=lock&responseContent=1";
dojo.rawXhrPut({
        url:  urlForLock,
        handleAs: "text",
        // Set the value of "If-Match" to the value of eTag
         // that you got in the 1st step
        headers: {"Content-Type":"application/json","If-Match":eTag},
        _raw: true,
        // The callback to handle the response
        load: function(data, ioArgs) {
            console.log(dojo.toJson(data));
            console.log("Lock successfully");
        },
        // The error callback
        error: function(data) {
            console.dir(data);
        },
        sync: true
    }
);

// Set the step element parameter values
for (var dataFieldName in stepElement.dataFields) {
    var dataField = stepElement.dataFields[dataFieldName];
    if (dataField.type === 2) {
        dataField.value = "Value_1";
        dataField.modified = true;
        console.log("Set the value for data field: " + dataFieldName);
    }
}
for (var attachmentName in stepElement.attachments) {
    console.log("Set the value for attachment: " + attachmentName);
    var attachment = stepElement.attachments[attachmentName];
    if (attachment.isArray) {
        attachment.value =
        [{"vsId":"{997E0AAC-8068-4990-AC08-D748E3063162}",
        "title":"Document Title",
        "type":3,
        "libraryName":"objectstore",
        "libraryType":3,
        "version":"{D48EE0A9-61CA-4DAD-8B61-961CE362434B}",
        "desc":"A document added programmatically"}];
    } else {
        attachment.value =
        {"vsId":"{997E0AAC-8068-4990-AC08-D748E3063162}",
```

```
               "title":"Document Title",
               "type":3,
               "libraryName":"objectstore",
               "libraryType":3,
               "version":"{D48EE0A9-61CA-4DAD-8B61-961CE362434B}",
               "desc":"A document added programmatically"};
         }
         attachment.modified = true;
      }

      // set workflow group participant
      for (var workflowGroupName in stepElement.workflowGroups) {
         console.log("Set the value for workflow group: " +
workflowGroupName);
         var workflowGroup = stepElement.workflowGroups[workflowGroupName];
         if (workflowGroup.isArray) {
            workflowGroup.value = [{"Administrator"}];
         } else {
            workflowGroup.value = {"Administrator"};
         }
         workflowGroup.modified = true;
      }

      // Set the selected Response
      stepElement.systemProperties.selectedResponse = "Complete";
```

5.6.4 Complete work items

To complete work, a step processor or work performer application must be able
to save or cancel changes that are made to the work item, and be able to
advance the work item to the next step in the workflow.

The following steps are included in the examples in order to complete work items
using the PE Java and Web services APIs:

- ► Save changes to the work item without advancing it in the workflow.

- ► Cancel changes to the work item without advancing it in the workflow.

- ► Save changes to the work item and advance it in the workflow.

For further details about included steps to complete work items by using the PE
REST API, see the "Complete work items: REST API scenario" on page 160.

Example 5-22 on page 157 and Example 5-23 on page 158 show the possible
actions to complete work items by using both the Java API and a PEWS client.

Example 5-22   Complete a work item using the Java API

```
// Create session object and log onto Process Engine
...
// Retrieve Step Element and Step Parameters
...
// Set Step Element Parameters
...
// Action Types
final int ACTION_TYPE_REASIGN = 1;
final int ACTION_TYPE_RETURN = 2;
final int ACTION_TYPE_ABORT = 3;
final int ACTION_TYPE_SAVE= 4;
final int ACTION_TYPE_DISPATCH = 5;

// Complete Step Element

// Action to perform on the Step Element
int actionToPerform = 5; // Dispatch

switch (actionToPerform) {

   case ACTION_TYPE_REASIGN:
      // Determine whether a step element
      // can be reassigned and reassign it
      if (stepElement.getCanReassign()) {
         String participantName = "Administrator";
         stepElement.doReassign(participantName,true,null);
      }
      break;

   case ACTION_TYPE_RETURN:
      // Determine whether a step element can be returned to the
      // queue from which the user delegated or reassigned it and
      // return it
      if (stepElement.getCanReturnToSource())
         stepElement.doReturnToSource();
      break;

   case ACTION_TYPE_ABORT:
      // Cancel the changes to the work item
      // without advancing it in the workflow
      stepElement.doAbort();
      break;
```

```
        case ACTION_TYPE_SAVE:
            // Save the changes to the work item
            // and unlock it without advancing it in the workflow
            stepElement.doSave(true);
            break;

        case ACTION_TYPE_DISPATCH:
            // Save the changes to the work item
            // and advance it in the workflow
            stepElement.doDispatch();
            break;
}
```

Note: `Abort actions on step elements` are not available for the Process Engine Web Services API. Instead, the `unlockStep` operation is available for step elements (to cancel changes on a work item and unlock it).

Example 5-23   Complete a work item using PEWS

```
// Create the PEWS RequestSoapContext header
...
// Retrieve Step Element and Step Parameters
...
// Set Step Element Parameters
...
// Action Types
sealed int ACTION_TYPE_REASIGN = 1;
sealed int ACTION_TYPE_RETURN = 2;
sealed int ACTION_TYPE_SAVE= 3;
sealed int ACTION_TYPE_DISPATCH = 4;


// Complete Step Element

// Action to perform on the Step Element
int actionToPerform = 4; // Dispatch

switch (actionToPerform)
{
    case ACTION_TYPE_REASIGN:
        if (stepElement.CanReassign)
        {
            string participantName = "Administrator";
            peWS.ReassignStepRequest reassignRqt =
            new peWS.ReassignStepRequest ();
            reassignRqt.delegate = true;
            reassignRqt.newUserName = participantName;
            reassignRqt.stepElement = stepElement;
```

```
            peWSServicePort.reassignStepRequest(reassignRqt);
        }
        break;

    case ACTION_TYPE_RETURN:
        // Determine whether a step element can be returned to the
        // queue from which the user delegated or reassigned it and
        // return it
        if (stepElement.CanReturnToSource)
        {
            peWS.UpdateStepRequest updStepRequest =
            new peWS.UpdateStepRequest();
            peWS.UpdateFlagEnum updFlagEnum =
            peWS.UpdateFlagEnum.UPDATE_SAVE_RETURN;
            updStepRequest.stepElement = stepElement;
            updStepRequest.updateFlag = updFlagEnum;
            peWSServicePort.updateStep(updStepRequest);
        }
        break;

    case ACTION_TYPE_SAVE:
        // Save the changes to the work item
        // and unlock it without advancing it in the workflow
        peWS.UpdateStepRequest updStepRequest =
        new peWS.UpdateStepRequest();
        peWS.UpdateFlagEnum updFlagEnum =
        peWS.UpdateFlagEnum.UPDATE_SAVE_UNLOCK;
        updStepRequest.stepElement = stepElement;
        updStepRequest.updateFlag = updFlagEnum;
        peWSServicePort.updateStep(updStepRequest);
        break;

    case ACTION_TYPE_DISPATCH:
        // Save the changes to the work item
        // and advance it in the workflow
        peWS.UpdateStepRequest updStepRequest =
        new peWS.UpdateStepRequest();
        peWS.UpdateFlagEnum updFlagEnum  =
        peWS.UpdateFlagEnum.UPDATE_DISPATCH;
        updStepRequest.stepElement = stepElement;
        updStepRequest.updateFlag = updFlagEnum;
        peWSServicePort.updateStep(updStepRequest);
    break;
}
```

Complete work items: REST API scenario

To complete work items by using the PE REST API, follow these steps:

1. Send HTTP GET using the `stepelements` resource to retrieve the step element and `eTag` parameter.

2. Send HTTP PUT using the `stepelements` resource to lock the step element.

3. Set step element parameter values and response.

4. Send HTTP PUT using the `stepelements` resource and the `action` parameter.

Table 5-11 presents the possible values that the `action` parameter can contain to perform an action on a step element. This parameter is required for all HTTP PUT requests by using the `stepelements` resource.

Table 5-11   Action parameter values

Action value	Description
lock	Locks the work item.
overrideLock	Locks the work item. If the work item is already locked, the current lock will be overridden.
save	Updates the work item and leaves it locked. In the request content, sends the updated version of the representation that is returned in the GET response.
saveAndUnlock	Updates the work item and unlocks it. In the request content, sends the updated version of the representation that returned in the GET response.
abort	Unlocks the work item without making any updates.
dispatch	Updates the work item, unlocks it, and advances the work item in the workflow. In the request content, sends the updated version of the representation returned in the GET response.

Example 5-24 shows how to complete a work item with a Dojo toolkit client.

Example 5-24   Complete a work item with a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
// Set request URI
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";

// Set REST API resource and work item number
var queueName = "Inbox";
var wobNumber = "5BDD567B70453C48A5388AC56ABC69E4";
```

```
var url = baseURL +  "queues/" +
queueName + "/stepelements/" + wobNumber;

// Use the HTTP GET method to retrieve the step element and eTag
...

// Use the HTTP PUT method to lock the step element
...

// Set step element parameter values and response
...

// Complete the step element by HTTP PUT method
var urlForDispatch = url + "?action=dispatch&responseContent=0";
dojo.rawXhrPut({
        url:  urlForDispatch,
        handleAs: "text",
        // Set the value of "If-Match" to the value of eTag
         //retrieved from the initial request
        headers: {"Content-Type":"application/json","If-Match":eTag},
        _raw: true,
        // Set the data you want to put back
        putData: dojo.toJson(stepElement),
        // The callback to handle the response
        load: function(data, ioArgs) {
            console.log("Complete the step element successfully");
        },
        // The error callback
        error: function(data) {
            console.dir(data);
        },
        sync: true
    }
);
```

5.7 Work with process status

Use process status elements to keep track of a running workflow. Status elements can be retrieved and monitored. Elements might be milestones reached within a running workflow, workflow steps completed, which participants are completing these steps, and other valuable information about the workflow. The main components of process status indicators are workflow history and milestones elements:

► Workflow milestones

To track the progress of a workflow, key points (milestones) are included within a workflow definition. On each workflow map, a milestone can be placed either before or after a step. Each of these milestones is related to a message that is specified by the workflow designer. The message is written to a log file when the running workflow reaches the milestone.

► Workflow history

To track workflow activity, workflow-related historical information is extracted from the event logs, which contain specific records related to a workflow event for each PE-isolated region. This historical information is useful to track workflow-related activities, such as how long workflows take to complete and which workflows are currently running.

5.7.1 Retrieve process history

To retrieve the information about the workflow history by using the PE Java API:

1. Retrieve the work item object.
2. Get `VWProcess` object from the work object.
3. Get the workflow definitions from the `VWProcess`.
4. Get the maps for each workflow definition.
5. Get the workflow history information for each map.
6. Get the step history objects for each workflow history.
7. Get the step occurrence history objects for each step history object.
8. Get the step work object information for each step occurrence.
9. Get the participant information for each work object.

Table 5-12 on page 163 shows the information that can be retrieved from the PE objects to track the workflow-related historical information.

Table 5-12   Process history information

Process Engine object	Information to retrieve
VWProcess	Status, history, child, and split work object information
VWProcessHistory	Launch step comments, launch date, workflow originator, launch step response
VWStepHistory	Step occurrence completion date, date participant received work, step status
VWStepOcurrenceHistory	Step name
VWStepWorkObjectHistory	Whether this process was completed
VWParticipantHistory	Comments, responses, dates completed and received, exposed log fields, activity status, name of participant who completed, delegated, reassigned, returned, or terminated step

Example 5-25 shows how to retrieve information about the workflow history by using the Java API

Example 5-25   Retrieve workflow history information using the Java API

```
// Create session object and log onto Process Engine
...
// Get the specific work item
...
// Get VWProcess object from work object
VWProcess process = stepElement.fetchProcess();

// Get workflow definitions from the VWProcess
VWWorkflowDefinition workflowDefinition =
process.fetchWorkflowDefinition(false);

// Get maps for each workflow definition
VWMapDefinition[] workflowMaps = workflowDefinition.getMaps();

// Iterate through each map in the workflow Definition
for (int i = 0; i < workflowMaps.length; i++) {

    // Get map ID and map name for each map definition
    int mapID = workflowMaps[i].getMapId();
    String mapName = workflowMaps[i].getName();

    // Get workflow history information for each map
    VWWorkflowHistory workflowHistory =
    process.fetchWorkflowHistory(mapID);
    String workflowOriginator = workflowHistory.getOriginator();
```

```
                // Iterate through each item in the Workflow History
                while (workflowHistory.hasNext()) {

                    // Get step history objects for each workflow history
                    VWStepHistory stepHistory = workflowHistory.next();

                    // Iterate through each item in the Step History
                    while (stepHistory.hasNext()) {

                        // Get step occurrence history
                        // objects for each step history object
                        VWStepOccurrenceHistory stepOccurenceHistory =
                        stepHistory.next();
                        Date stepOcurrenceDateReceived =
                        stepOccurenceHistory.getDateReceived();
                        Date stepOcurrenceDateCompleted =
                        stepOccurenceHistory.getCompletionDate();

                        while (stepOccurenceHistory.hasNext()) {

                            // Get step work object information
                            // for each step occurrence
                            VWStepWorkObjectHistory stepWorkObjectHistory =
                            stepOccurenceHistory.next();
                            stepWorkObjectHistory.resetFetch();

                            // Get participant information for each work object
                            while (stepWorkObjectHistory.hasNext()) {
                                VWParticipantHistory participantHistory =
                                stepWorkObjectHistory.next();
                                Date participantDateReceived =
                                participantHistory.getDateReceived();
                                String participantComments =
                                participantHistory.getComments();
                                String participantUser =
                                participantHistory.getUserName();
                                String participantName =
                                participantHistory.getParticipantName();
                            } // while stepWorkObjectHistory
                        } // while stepOccurenceHistory
                    }  // while stepHistory
                } // while workflowHistory
            } // for workflow maps
```

5.7.2 Retrieve process milestones

The following steps are included to retrieve the workflow milestones history information using the PE Java and Web services APIs:

1. Retrieve the work item object.
2. Get VWProcess object from the work object.
3. Get all milestones definition from the VWProcess object.
4. Get reached milestones from the VWProcess object.
5. Get milestone element information for each milestone.
6. Handle information for each milestone element.

Each milestone that is defined within a workflow is associated with a level (1 - 99). The workflow-reached milestones are based on the milestone level that is specified. For example, if the milestone level is set to 5, reached-milestones of levels 1 - 5 are retrieved.

Example 5-26 and Example 5-27 on page 166 show how to perform the steps required to retrieve the workflow milestones history information, using both the Java API and a PEWS client.

Example 5-26   Get milestones of a workflow using the Java API

```
// Create session object and log onto Process Engine
...
// Get the specific work item
...
// Get VWProcess object from work object
...
// Get all milestones definition from the workflow process
VWMilestoneDefinition[] milestonesDefinition =
process.getMilestoneDefinitions();
for (int i = 0; i < milestonesDefinition.length; i++) {
    // Milestone element information
    VWMilestoneDefinition milestoneDefinition = milestonesDefinition[i];
    String milestoneName = milestoneDefinition.getName();
    String milestoneMessage = milestoneDefinition.getMessage();
    int milestoneLevel = milestoneDefinition.getLevel();
}
// Get reached milestones from the VWProcess object
int level = 99;
VWWorkflowMilestones milestones =
process.fetchReachedWorkflowMilestones(level);

while (milestones.hasNext()) {
    // Milestone element information
```

```
        VWMilestoneElement milestoneElement = milestones.next();
        String milestoneName = milestoneElement.getName();
        String milestoneMessage = milestoneElement.getMessage();
        Date milestoneTimeLogged = milestoneElement.getTimestamp();
}
```

Example 5-27   Get milestones of a workflow using PEWS

```
// Create the PEWS RequestSoapContext header
...
// Get milestones work object from the roster
string rosterName = "DefaultRoster";
string wobnum = "5BDD567B70453C48A5388AC56ABC69E4";

peWS.GetMilestoneForRosterElementRequest query = new
peWS.GetMilestoneForRosterElementRequest ();
query.milestoneLevel = 99;
query.rosterName = rosterName;
query.queryValue = wobnum;
query.queryEnum = peWS.MilestoneQueryEnum.MILESTONE_QUERY_WOBNUMBER;

peWS.Milestone [] milestones =
peWSServicePort.getMilestoneForRosterElement(query);

if (milestones!=null)
{
    foreach (peWS.Milestone m in milestones)
    {
      // Milestone element information
      string map = m.Map;
      string milestoneMessage = m.Message;
      string milestoneName = m.Name;
      int milestoneId = m.Id;
      int milestoneLevel = m.Level;
      int stepId = m.StepId;
      DateTime milestoneDateReached = m.Reached;
    }
}
```

6

Advanced Process Engine API programming

In this chapter, we introduce advanced topics for the IBM FileNet Business Process Manager API programming. Although we refer to these topics as *advanced*, they are actually a continuation of the concepts we presented in Chapter 5, "Introduction to Process Engine API programming" on page 111.

Most of the code snippets we show in this chapter are from the sample applications that we created for this book. For more details about the sample applications, refer to Chapter 7, "Sample applications for Fictional Auto Rental Company A" on page 223.

This chapter discusses the following topics:

► Component Integrator
► Application space, role, and workbasket
► Resource navigation in Process Engine REST API
► ECM Widgets overview
► Building a custom Get Next In-basket widget

6.1 Component Integrator

Component Integrator (CI) is part of the IBM FileNet P8 platform, and that connects the PE with an external entity called a component. A component can be either a Java object or Java Messaging System (JMS) queue, making them available in a workflow.

CI provides a technology platform that enables organizations to integrate and coordinate their business processes. This is typically called Enterprise Information System (EIS) integration. The use of CI provides enterprise solution capabilities, such as:

- ► Extend business functionality easily without full application development.
- ► Automate work processing.
- ► Perform external functions from within a workflow.
- ► Use existing Java business objects and components.
- ► Integrate with a Java Message Service (JMS).

For further details about CI general topics, including how to implement the Web Services adaptor, refer to "Chapter 8: Implementing Component Integrator and Web services" from *Introducing IBM FileNet Business Process Manager*, SG24-7509.

In this section, we present the available actions provided by the CE_Operations component, and how to implement a custom Java adaptor.

6.1.1 CE_Operations component

CE_Operations is a built-in component that are used to integrate the Content Engine (CE) and Process Engine (PE), facilitating interaction with documents and other objects in an object store from within a workflow. This component is provided as part of a typical Application Engine (AE) or WorkplaceXT installation (although it does not run within the J2EE application as part of AE or WorkplaceXT).

The following required steps implement CE_Operations in a workflow definition:

1. Determine the symbolic name and data type of the object class property using IBM FileNet Enterprise Manager.

2. Define the appropriate attachments and data fields in the workflow definition. These data fields hold the values retrieved from the object property in the component step.

3. Add the component step on the workflow map, select the CE_Operations component queue, and define the appropriate parameters for each operation.

Available actions

Table 6-1 describes available actions that are provided by the CE_Operations component.

Table 6-1   Available CE_Operations

Action	Description
File objects	File one or more objects (documents and custom objects) to a specified folder.
Unfile objects	Unfile one or more objects (documents and custom objects) from a specified folder.
Publish document	Publish a specified document by using a publishing template.
Get object property values	Get single or multiple property values from a supplied document, custom object, or folder. The available methods to get property values support integer, double, date, Boolean, string, and object types.
Set object property values	Set single or multiple property values of a supplied document, custom object, or folder. The available methods to set property values support integer, double, date, Boolean, string, and object types.
Change object class	Change the class of a supplied document, folder, or custom object.
Delete object	Delete one or more objects (documents, folders, and custom objects).
Document versioning	Check out a specified document from the repository and check in a specified document as a new major version.
Copy object	Copy one or more objects (documents and custom objects) within the same repository.
Create object	Create a new object (documents, folders, and custom objects) instance using a specified class and property values. For document objects, MIME type and content parameters must be specified
Move object	Move one or more objects (documents and custom objects) from their existing folder to a specified folder.
Get object	Get an object (custom object, folder, or document) from a specified repository using a specified path.

Action	Description
Apply security template	Apply a security template using a specified template to one or more objects (documents, folders, and custom objects).
Search object	Execute a specified search template or stored search, returning the first object or an array of objects matching the search criteria.
Send mail	Available methods to send mail supporting the following scenarios: ► Text (plain) with no attachment ► With documents attached ► Using a template with no attachment ► Using a template with documents attached ► With the URLs of the attached documents appended to the end of the body of the e-mail ► Using a template with the URLs of the attached documents appended to the end of the body of the e-mail

Note: All `sendMail` methods require that e-mail notification be properly configured by using the Process Task Manager. For further details about how to configure e-mail notification, refer to the following P8 documentation path: **ECM Help** → **System Administrator** → **Enterprise-wide Administration** → **Process Task Manager** → **Process Engine** → **Configure Process Engine** → **Email notification**.

6.1.2 Implementing a custom Java component

Java adapters handle PE calls to Java objects, which are represented to the PE as operations on queues (work items) where each operation is performed by a method of the Java class. The Java adapter executes the interface to the Java component, then automatically waits for a response from this component, updates the work item, and dispatches this work item to the next workflow step.

The following steps describe the development process to implement a custom component:

1. Identify the business need for the component.
2. Define component operations.
3. Implement the custom component.
4. Configure component queue(s) to service the custom adaptor operations.
5. Iteratively test and debug the custom component.
6. Deploy custom component to production environment.

> **Note:** For multithread-safe custom components, concurrent threads may be increased to enable parallel execution of the component on a multiprocessor system, improving the component execution performance. The number of threads for the component can be increased according the number of processors on the system.

The custom Java classes serving as a gateway for the component must comply with the following conditions:

- ► A constructor that has no parameters.
- ► Only public methods are available for the component operations.
- ► Public methods exposed as component operations can contain only the following parameter types (single and array):
  - String
  - Date
  - Integer
  - Float
  - Double
  - Boolean
  - VWAttachment
  - VWParticipant

Java component association

The Java component operations are defined as a public method in a Java class (gateway) compiled to a Java archive (JAR) file. The JAR file must be deployed to a network directory that the AE can connect to; however we recommend using a directory location on the AE server.

The Java adaptor associates a Java component queue with a Java archive file and Login Context as shown in Figure 6-1.
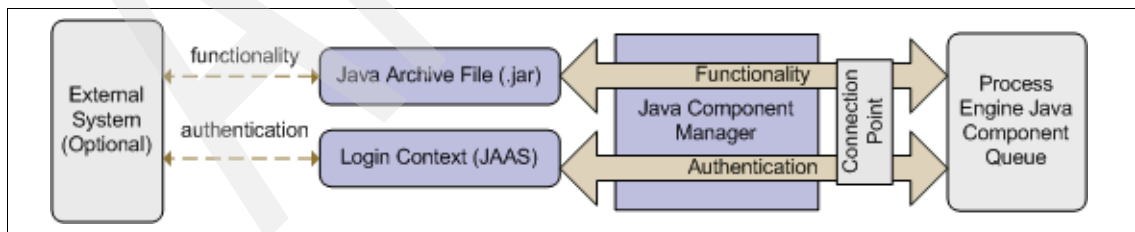
Figure 6-1   Java component association

ITSO_Operations custom component

ITSO_Operations is a custom Java component that is implemented to perform certain business operations, which are used by this book's sample applications. This custom adaptor also includes operations not used in the sample applications but implemented to show how to perform certain actions, such as custom PE API calls.

In this section, we use the ITSO_Operations component package to describe certain elements that are included, in order to implement a custom Java adaptor. These elements are:

- ► The Java business classes
- ► The Java gateway class
- ► The custom Java Authentication and Authorization Service (JAAS) class and configuration file

Implementing the Java business classes

The business entities for ITSO_Operations are implemented as Data Access Object (DAO) classes that are consumed by the gateway class. The two business entities that are implemented for this custom Java adaptor are:

- ► ITSOVehicleDAO: Business entity for vehicle-related operations
- ► ITSOCustomerDAO: Business entity for customer-related operations

ITSO_Operations business entities perform custom CE operations (not available in CE_Operations) and PE operations that are required for sample application. Certain actions, such as the verification of a customer's credit card number (`verifyCustomerCredit`) are artificial, created only as placeholders with the intention to show how certain business operations can be implemented to perform calls to external systems (for example, custom database).

Example 6-1 and Example 6-2 on page 174 show specific DAO methods that are implemented for both vehicle and customer business entities.

Example 6-1   Customer business entity

```
/**
 * DAO class used as business entity for customer-related operations
 */
public class ITSOCustomerDAO {

    // Retrieve the content from a ITSOCustomer
    // document and parse specific meta-data
    public String[] getCustomerDetailsFromOS(String objectStoreName,
        String customerID, String[] selectedFields) throws Exception {
        String[] customerDetails = new String[] {};
```

```
            ITSOUtil itsoUtil = new ITSOUtil();
            SearchScope search = new
            SearchScope(itsoUtil.getCEConnection().fetchOS(objectStoreName));
            SearchSQL sql = new SearchSQL();
            sql.setSelectList("*");
            sql.setFromClauseInitialValue("ITSOCustomer", "d", true);
            sql.setWhereClause("ITSOCustomerUserId='"+customerID+"'
               AND NOT (IsClass(d, CodeModule))");
            DocumentSet documents = (DocumentSet)search.fetchObjects
               (sql,Integer.getInteger("1"),null, Boolean.valueOf(true));
            Iterator itDoc = documents.iterator();

            if (itDoc.hasNext()) {
                Document doc = (Document) itDoc.next();
                ContentElementList contents = doc.get_ContentElements();
                ContentElement content;
                Iterator itContent = contents.iterator();

                if (itContent.hasNext()) {
                    content = (ContentElement)itContent.next();
                    customerDetails = parseCustomerDetailsContent
                        ((ContentTransfer)content, selectedFields);
                }
            }
            return customerDetails;
        }

        // Get external database connection from the
        // custom LoginModule and perform call
        public Boolean verifyCustomerCredit(String creditCardNumber,
            Integer requestedAmount) throws Exception {
                return Boolean.TRUE;
        }

        // Additional Customer DAO Methods
        ...
}
```

Example 6-2   Vehicle business entity

```
/**
 * DAO class used as business entity for vehicle-related operations
 */
public class ITSOVehicleDAO {

   public String[] getVehicleMaintananceQueueDetails(String vehicleID,
      String[] selectedFields) throws Exception {
      String[] vehicleDetails = new String[] {};

      String maintenanceQueue = "ITSO_Maintenance";
      ITSOUtil itsoUtil = new ITSOUtil();
      VWQueue queue = itsoUtil.getPESession().
         getQueue(maintenanceQueue);

      String queryFilter="ITSOVehicleId=:A";
      String[] substitutionVars = {new String(vehicleID)};
      int queryFlags = VWQueue.QUERY_READ_LOCKED;
      int fetchType = VWFetchType.FETCH_TYPE_WORKOBJECT;

      VWQueueQuery queueQuery = queue.createQuery(null,null,null,
         queryFlags,queryFilter,substitutionVars,fetchType);

      if (queueQuery.hasNext()) {
         vehicleDetails = new String[selectedFields.length];
         VWWorkObject workItem = (VWWorkObject) queueQuery.next();

         for (int i=0;i<selectedFields.length;i++)
            vehicleDetails[i] = workItem.getFieldValue
               (selectedFields[i]).toString();
      }
      return vehicleDetails;
   }

   // Additional Vehicle DAO Methods
   ...
}
```

Implementing the Java gateway class

The Java gateway class of a custom component is used to expose the public
business methods that are used by the component queue operations.

The ITSO_Operations component implements the Java gateway class
ITSOOperations to expose the business methods (public methods) and define the
component queue operations. The gateway class has a constructor without any
parameters and the public methods only contain allowable component operation
parameter types.

The gateway class consumes the business entities (ITSOVehicleDAO and
ITSOConsumerDAO) to perform all business-related logic. See Example 6-3.

Example 6-3 Operations gateway class: ITSOOperations.java

```
/**
 * Gateway class for ITSO_Operations.
 */
public class ITSOOperations {

    public ITSOOperations() {
    }

    // Validate the customer's credit from external database system
    public Boolean verifyCustomerCredit(String creditCardNumber,
        Integer requestedAmount) throws Exception {
        return new ITSOCustomerDAO().
            verifyCustomerCredit(creditCardNumber, requestedAmount);
    }

    // Get customer details from ITSOCustomer
    // document content using ad-hoc query
    public String[] getCustomerDetailsFromOS(String objectStoreName,
        String customerID, String[] selectedFields) throws Exception {
        return new ITSOCustomerDAO().
            getCustomerDetailsFromOS(objectStoreName,
            customerID, selectedFields);
    }

    // Get vehicle maintenance details from the
    // Process Engine maintenance queue
    public String[] getVehicleMaintDetails(String vehicleID,
        String[] selectedFields) throws Exception {
        return new ITSOVehicleDAO().
            getVehicleMaintananceQueueDetails(vehicleID, selectedFields);
    }

    // Additional Business Methods
    ...
}
```

Implementing the custom login context

A login context or configuration context in a custom component includes the login modules that provide the required authentication for all Java components. For example, a Java component must authenticate with a database for its operations, therefore a custom login module that is handling the database authentication must be defined for the login context of the component. These login contexts can be shared between many Java components.

The login context for Component Manager is defined within a Java Authentication and Authorization Service (JAAS) configuration file. A JAAS configuration file is provided (`taskman.login.config`) including the default login contexts provided by IBM FileNet, as shown in Example 6-4.

Example 6-4   Default login contexts provided in the taskman.login.config file

```
CELogin
{
    filenet.vw.server.VWLoginModule required routerurl="";
    com.filenet.wcm.toolkit.server.operations.util.CELoginModule
    required credTag=Clear;
    filenet.contentops.ceoperations.util.CELoginModule
    required credTag=Clear;
};

PEWSLogin
{
    com.filenet.wcm.toolkit.server.operations.util.CELoginModule
    required credTag=Clear;
};

FileNetP8
{
    com.filenet.api.util.WSILoginModule required debug=false;
};
```

The `FileNetP8` stanza must be present for any JAAS configuration file containing the appropriate login module to connect to CE, depending on the CE transport protocol. By default, when running outside of an application server, the Content Engine Web Services (CEWS) transport protocol is recommended (`com.filenet.api.util.WSILoginModule`) because it is typically easier to configure. EJB transport can have performance advantages.

The `CELogin` stanza is used by the CE_Operations component. This login context includes the following login module that provides authentication to CE:

`com.filenet.wcm.toolkit.server.operations.util.CELoginModule`

This login module is used in conjunction with the CE 3.x Java Compatibility Layer; therefore, a `com.filenet.wcm.api.Session object` is initiated using this module.

At the minimum, a login context that is used for custom Java components must contain the `filenet.vw.server.VWLoginModule` module, which authenticates to a PE Isolated Region. This approach means that the provided user information should exist for both the directory server used by the IBM FileNet P8 platform and the custom component.

Because ITSO_Operations performs custom CE actions that are not available within CE_Operations, the custom `ITSOLogin` login context is implemented for the component.

The following steps describe how the custom login context is implemented for the ITSO_Operations component:

1. Create `ITSOPrincipal` class.
2. Create `ITSOLoginModule` class.
3. Add `ITSOLogin` login context to `taskman.login.config` file.
4. Create `ITSOUtil` class to retrieve the `ITSOPrincipal` object.
5. Deploy ITSO_Operations component using `ITSOLogin` context.

`ITSOPrincipal` implements the `java.security.Principal` interface, which is used by `ITSOLoginModule` class. See Example 6-5.

Example 6-5   ITSO_Operations principal class: ITSOPrincipal.java

```
/**
 * Principal class for ITSO_Operations
 */
public class ITSOPrincipal implements Principal, Serializable {

    private String m_username;
    private ITSOCEConnection m_ceconnection;

    public ITSOPrincipal( ITSOCEConnection m_ceconnection,
        String m_username )
    {
        if( m_username == null )
            throw new NullPointerException( "illegal null input" );

            this.m_ceconnection = m_ceconnection;
            this.m_username = m_username;
    }

    public ITSOCEConnection getCEConnection()
```

```
      {
         return this.m_ceconnection;
      }

      public boolean equals( Object o )
      {
         if( o == null )
             return false;

         if( this == o )
             return true;

         if( !(o instanceof ITSOPrincipal) )
             return false;
         ITSOPrincipal that = (ITSOPrincipal) o;

         if( this.getName().equals( that.getName() ) )
             return true;
         return false;
      }

      // Additional ITSOPrincipal methods
      ...
}
```

ITSOLoginModule implements the javax.security.auth.spi.LoginModule
interface including the custom CE authentication and passing the connection
object to ITSOPrincipal for further handling. See Example 6-6.

Example 6-6   ITSO_Operations login module class: ITSOLoginModule.java

```
/**
 * Custom Login Module class for ITSO_Operations.
 */
public class ITSOLoginModule implements LoginModule {

// ITSO Principal
   private ITSOPrincipal m_principal;

   public void initialize(Subject subject,
     CallbackHandler callbackHandler, Map sharedState, Map options)
   {
         this.m_subject        = subject;
         this.m_callbackHandler = callbackHandler;;
         this.m_options         = options;
```

```
            this.m_sharedState    = sharedState;
        }

        public boolean login() throws LoginException
        {
            if (m_callbackHandler == null) {
                throw new LoginException("Error: no CallbackHandler
                    available to get authentication information");
            }

          String password = null;

          m_username = (String)m_sharedState.get
             ("javax.security.auth.login.name");
          password   = (String)m_sharedState.get
             ("javax.security.auth.login.password");

          if (m_username == null) {
             Callback[] callbacks = null;
             callbacks = new Callback[3];
             callbacks[0] = new TextOutputCallback(TextOutputCallback.
               INFORMATION, "Custom Authentication");
             callbacks[1] = new NameCallback("user name:");
             callbacks[2] = new PasswordCallback("password:", false);

             try {
               m_callbackHandler.handle(callbacks);
               m_username = ((NameCallback)callbacks[1]).getName();

               char[] tmpPassword = ((PasswordCallback)callbacks[2]).
                  getPassword();

               if (tmpPassword != null)
                  password = new String(tmpPassword);
               else
                  password = null;

               ((PasswordCallback)callbacks[2]).clearPassword();

               m_sharedState.put("javax.security.auth.login.name",
                  m_username);
               m_sharedState.put("javax.security.auth.login.password",
                  password);

             }
```

```
            catch (IOException ioe)
            {
                throw new LoginException(ioe.toString());
            }
            catch (UnsupportedCallbackException uce)
            {
                throw new LoginException("Error: no CallbackHandler
                    available to get authentication information");
            }
        }

        try {
            /**
             * Perform custom CE authentication
             */
            String ceURI = System.getProperty
                ("filenet.pe.bootstrap.ceuri");
            m_ceconnection = new ITSOCEConnection(m_username, password,
                "FileNetP8WSI", ceURI);

        }
        catch (Exception e)
        {
            m_ceconnection = null;
            throw new LoginException(e.toString());
        }

        m_validSession = true;
        return (m_validSession);

    }

    public boolean commit() throws LoginException {
        if(m_ceconnection == null)
            return false;
        m_principal = new ITSOPrincipal(m_ceconnection, m_username);
        if(!m_subject.getPrivateCredentials().contains(m_principal))
            m_subject.getPrivateCredentials().add(m_principal);
        return true;
    }

    // Additional ITSOLoginModule methods
    ...
}
```

ITSOLogin is the custom login context appended to `taskman.login.config` file
that is used by the ITSO_Operations component. See Example 6-7.

Example 6-7   JAAS custom login context implemented for ITSO_Operations

```
ITSOLogin
{
   filenet.vw.server.VWLoginModule required;
   itso.autorental.components.authentication.ITSOLoginModule
   required debug=true;
};
```

The `ITSOUtil` class is a custom class that is implemented to retrieve the
`ITSOPrincipal`, including the custom CE authenticated object. A method to
retrieve the PE session object (`VWSession`) is also implemented for custom
PE-related operations. See Example 6-8.

Example 6-8   ITSO_Operations class to retrieve ITSOPrincipal: ITSOUtil.java

```
/**
 * Utility class for ITSO_Operations.
 */
public class ITSOUtil {

    private VWSession pesession = null;

    public VWSession getPESession() throws Exception {
      if(this.pesession == null || !(this.pesession.isLoggedOn())) {
          Subject subject =
              Subject.getSubject(AccessController.getContext());
          Set creds = subject.getPrivateCredentials
              (Class.forName("filenet.vw.api.VWSession"));
          Iterator i = null;
          if (creds != null)
              i = creds.iterator();
          if (i != null)
              this.pesession = (VWSession) i.next();
      }
      return this.pesession;
    }

      public ITSOCEConnection getCEConnection() throws Exception {
          Subject subject Subject.getSubject
              (AccessController.getContext());
          Set set = null;
          try {
```

```
            set = subject.getPrivateCredentials(Class.forName
("itso.autorental.components.authentication.ITSOPrincipal"));
        }
        catch(Exception exception) { }
        if(set == null)
            throw new Exception("Failed to find principal");
        Iterator iterator = set.iterator();
        if(iterator == null || !iterator.hasNext())
            throw new Exception("Failed to find principal");
        Principal principal = (Principal)iterator.next();
        if(!(principal instanceof ITSOPrincipal))
            throw new Exception("Failed to find principal");
        else
            return ((ITSOPrincipal)principal).getCEConnection();
    }
}
```

Configuring ITSO_Operations using Eclipse

Usually, the component classes and resources must be packaged into a Java archive (JAR) file to be referenced by the Process Configuration Console applet when the component queue is created. This manual step can be time-consuming and error-prone.

Prior to configuring and testing the component queue by using Eclipse, see Chapter 2, "Setting up development environments" on page 13 to set up an Eclipse project for P8 development.

The programmatic creation of the component queue is possible with the tool `filenet.vw.integrator.base.PEComponentQueueHelper`, which is part of the libraries that are required by PE. Table 6-2 lists the parameters for the tool.

Table 6-2   Parameters for filenet.vw.integrator.base.PEComponentQueueHelper

Parameter	Description
/PEuser PEUser	PE user name
/PEpw PEPw	PE password
/PErouter PEConnectionPointName	Connection point name
/queue componentQueueName	Name of the component queue to be created
/JAAS JAASContext	JAAS login context name
/user JAASUsername	JAAS user name

Parameter	Description
/pw JAASPassword	JAAS password
/class componentClassName	Name of the component queue gateway class
/methods name1,name2,...	(Optional) Names of the methods of the classes to be imported as the component queue operations. If not specified, all public methods of the class are imported as operations for the queue

To use `filenet.vw.integrator.base.PEComponentQueueHelper` tool in Eclipse, perform the following steps (see Figure 6-2):

1. Create a run configuration for the tool.

2. Enter `PEComponentQueueHelper` for name and enter `filenet.vw.integrator.base.PEComponentQueueHelper` for the main class.
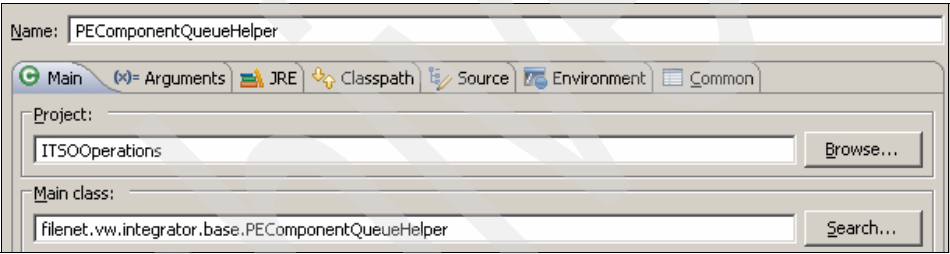
Figure 6-2   Run configuration for PEComponentQueueHelper

3. On the Arguments tab, set the following values by using the information in Table 6-3. Replace the parameter values with that of your environment. For an example, see Figure 6-3 on page 184.

Table 6-3   Parameters for PEComponentQueueHelper

Program arguments	`/PEuser administrator /PEpw filenet /PErouter CEPoint /queue ITSO_Operations /user administrator /pw filenet /JAAS ITSOLogin /class itso.autorental.components.ITSOOperations`
VM arguments	`-Djava.security.auth.login.config=${workspace_loc}/P8Libraries/CE_API/config/jaas.conf.WSI -Dwasp.location=${workspace_loc}/P8Libraries/CE_API/wsi -Dfilenet.pe.bootstrap.ceuri=http://CEServer:CEPort/wsi/FNCEWS40DIME/`
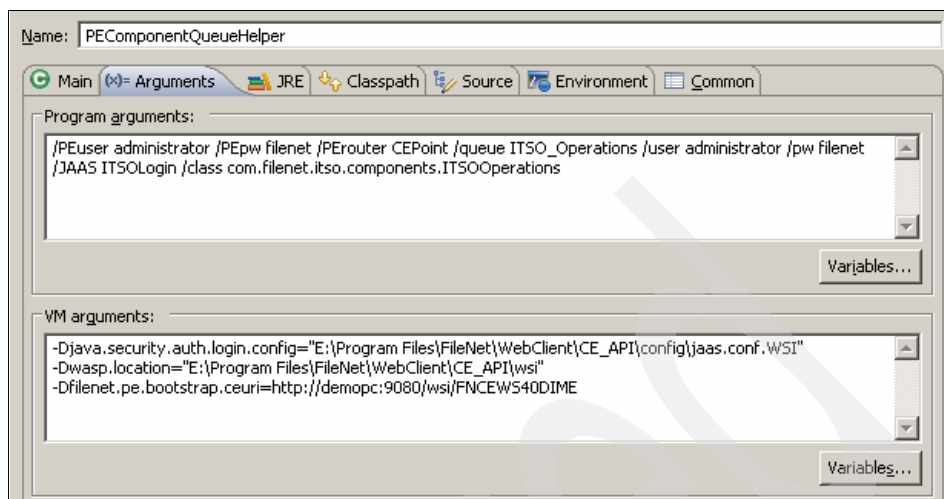
Figure 6-3   Example for PEComponentQueueHelper parameters in Eclipse

4. Run `PEComponentQueueHelper` and use the Process Configuration Console to verify that the component queue is created with the specified component name (ITSO_Operations), JAAS information, and operations.

Testing and debugging the component using Eclipse

Component Manager runs as an independent Java application on the AE. It is a PE API application polling component queues for work. It can be set up to just process specific queues instead of all component queues. At initialization, Component Manager instantiates an object of the class for the custom component and sets up the necessary JAAS framework that fits the component needs for accessing its own resources, such as a Java Database Connectivity (JDBC) connection.

When a work object arrives in the queue, the Component Manager invokes the corresponding method of the class on the class object. Multi-threaded custom components run in multiple threads, where each thread has its own class object. Debugging a custom component really means debugging Component Manager.

An instance of Component Manager can be run in Eclipse with the command line and using `filenet.vw.integrator.base.VWComponentManager` class, which is part of the required PE libraries. Table 6-4 on page 185 lists Component Manager parameters.

Table 6-4   Component Manager parameters description

Parameter	Description
/named	Specifies the component manager name. Parameter is optional.
/routerURL=PEConnectionPointName	Specifies the connection point name to access the PE.
/username=PEUsername	Specifies PE user name.
/password=PEPassword	Specifies PE password.
/registryPort=32771	Specifies the RMI registry port to register the Component Manager instance. The default value is 32771. This parameter allows the AE Process Task Manager to manage the instance
/eventPort=32773	The Component Manager can be set up to receive notification from the PE server when work arrives at any component queues. This approach avoids excessive polling. However, only one instance can be set up to receive notification. The default value is 32773

/registryName=<Component Manager RMI registry Name>	Specifies the Component Manager RMI server name; it must be unique within the same RMI registry. When AE Process Task Manager starts, by default, the parameter is:
	`FileNet.VW.VWComponentManager.<connectionPointName>.<queueName>`
	For example:
	`FileNet.VW.VWComponentManager.hqruby_600.CE_Operations`

/queues=queueName	Specifies all the queues (using a value of *), a specific queue, or a list of queues separated by comma.
/unbind	When present, this stops the current instance that has the RMI registry object with the specified /registryName

To set up Eclipse to run Component Manager, which allows for an easy way to debug the component, perform the following steps:

1. Create a run configuration for Component Manager.

2. Enter `Component Manager START` for name and enter `filenet.vw.integrator.base.VWComponentManager` for the main class. See Figure 6-4 on page 186.
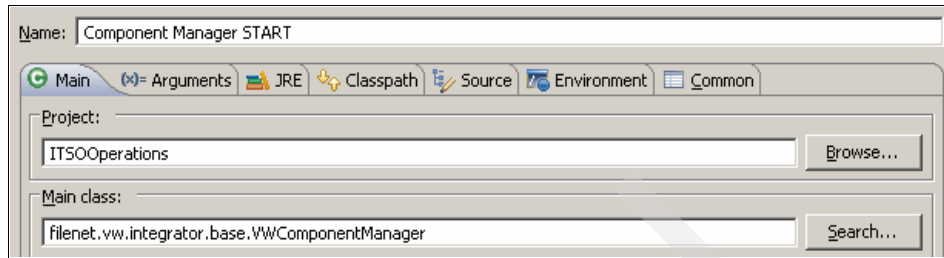
Figure 6-4   Run configuration for Component Manager START

3. On the Arguments tab, set the values by using the information in Table 6-5. Replace the parameter values with that of your environment. For an example, see Figure 6-5.

Table 6-5   Parameter for Component Manager START

Program arguments	/named /routerURL CEPoint /userName administrator /password filenet /registryPort 32771 /eventPort 32773 /registryName FileNet.VW.VWComponentManager.ITSO_Operations /queues=ITSO_Operations
VM arguments	-Djava.security.auth.login.config=${workspace_loc}/config/taskman.login.config -Dwasp.location=${workspace_loc}/P8Libraries/CE_API/wsi -Dfilenet.pe.bootstrap.ceuri=http://CEServer:CEPort/wsi/FNCEWS40DIME/
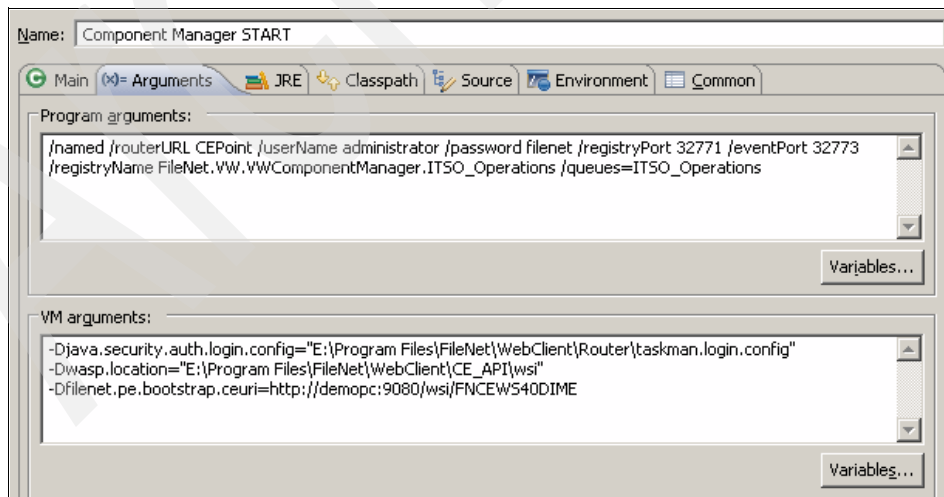
Figure 6-5   Example for Component Manager START parameters in Eclipse

4. Before running the Component Manager, ensure that all other Component Manager instances for the same PE connection point have been stopped. Otherwise, work in the queue is picked up and processed by those instances.

Create a run configuration to stop Component Manager by duplicating the `Component Manager START` run configuration, and name it `Component Manager STOP`. Add `/unbind` to the program arguments as shown in Figure 6-6.
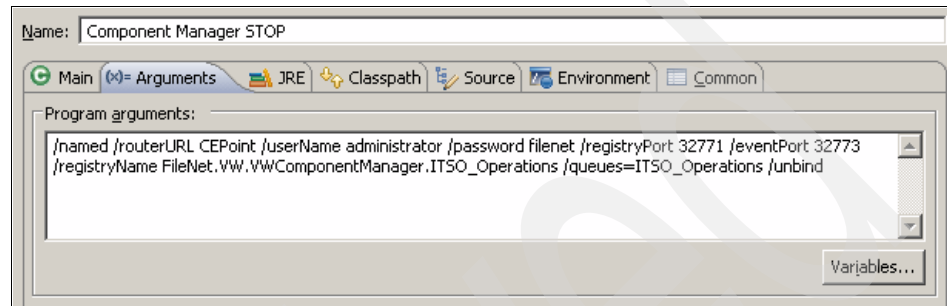
Figure 6-6   Example for Component Manager STOP parameters in Eclipse

Run the `Component Manager STOP` configuration to stop the running Component Manager instance.

5. Run `Component Manager START` in debug mode to start debugging and testing the component (ITSO_Operations) operations. Make sure to add a breakpoint to the intended component section (or sections) to be debugged.

6.2 Application space, role, and workbasket

In Process Engine (PE) 4.5.0, several concepts were introduced to the IBM FileNet P8 platform. You can leverage them to build a custom application that uses a workflow.

An *application space* is a way of organizing the resources for a custom application. Use either Process Designer or Process Configuration Console to specify an application space to contain the roles of users who will perform the activities in the application.

A *role* includes the access to workbaskets appropriate for the role. You can define one or more workbaskets for each role.

A *workbasket*, also known as an In-basket, filters the work items based on your query specification defined in PE. Its definition also includes the workflow fields that will be displayed, and the custom filters that filter the work items at run time

In this section, we present information for using PE Java API and REST API to access the application space, role, and workbasket.

6.2.1 Retrieve role list

After user logon to PE, a list of roles will be associated to the current user.

The following steps are included to retrieve a role list using the PE Java API:

1. Create a `VWSession` object and log on to PE.
2. Retrieve roles from the `VWSession` object by providing the application space name.

Using Java API

Example 6-9 shows getting role listing by using PE Java API.

Example 6-9   Get role list by using PE Java API

```
// Create session object and log onto Process Engine
...
String appSpaceName = "GenericApproval";
// Fetch the roles from session using the application space name
VWRole roles[] = session.fetchMyRoles(appSpaceName);
for(int i = 0; i < roles.length; i++) {
   System.out.println("Role Name:" + roles[i].getName());
   System.out.println("AuthoredName:" + roles[i].getAuthoredName());
}
```

PE REST API scenario

The following steps are required to retrieve role list for the current user:

1. Send HTTP GET using the `myroles` resource to retrieve the role list.
2. Process the information of each role.

Example 6-10 shows how to retrieve role list with a Dojo toolkit client.

Example 6-10   Retrieve role list with a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
var applicationSpaceName = "GenericApproval";
// Construct the URI for roles
var url = baseURL + "appspaces/" + applicationSpaceName + "/myroles";

// Use GET method to retrieve the roles
```

```
dojo.xhrGet({
        url: url,
        handleAs: "json-comment-optional",
        load: function(data) {
            // Callback to handle the data
            for(var roleName in data) {
                var role = data[roleName];
                console.log("AuthoredName:" + roleName);
                console.log("Display Name:" + role.name);
                console.log("URI:" + role.URI);
            }
        },
        error: function(data) {
            // The error callback
            console.dir(data);
        }
    }
);
```

6.2.2 Retrieve role description and attributes

Using PE Java API, you can directly retrieve a role object by specifying the
application space name and role name, then you will be able to process the
workbasket that belongs to the current role.

The following steps are included to retrieve role object:

1. Create a `VWSession object` and log on to PE.

2. Fetch VWRole object from the VWSession object by providing the role name
   and application space name.

3. Process each workbasket.

4. Process the role attributes.

Using Java API
Example 6-11 on page 190 shows how to retrieve role description using Java
API.

Example 6-11   Retrieve role description by using Java API

```java
// Create session object and log onto Process Engine
...
VWRole role;
String roleName = "Approver";
String appSpaceName = "GenericApproval";

role = session.fetchMyRole(roleName, appSpaceName);
System.out.println("name:" + role.getName());
System.out.println("authoredName:" + role.getAuthoredName());
System.out.println("description:" + role.getDescription());
System.out.println("homePage:" + role.getHomePageURL());

filenet.vw.api.VWRole.WorkBasketReference wbs[] =
role.getWorkBasketReferences();

// Process the workbasketsif(wbs != null) {
    for(int i = 0; i < wbs.length; i++) {
        System.out.println("Workbasket name:" +
wbs[i].getWorkBasketName());
        System.out.println("Queue name:" + wbs[i].getWorkBasketName());
        System.out.println("Workbasket authoredName:" +
wbs[i].getWorkBasketAuthoredName());
    }
}

VWAttributeInfo info = role.getAttributeInfo();
String names[] = info.getAttributeNames();

for(int i = 0; i < names.length; i++) {
    String fieldName = names[i];
    Integer filedType = info.getFieldType(fieldName);
    Boolean isArray = new Boolean(info.isFieldArray(fieldName));
    Object object = info.getFieldValue(fieldName);
    // Print out the role attribute value
}
```

REST API scenario

The following steps are required to retrieve a role object:

1. Send HTTP GET using the role resource to retrieve the role.
2. Process the information for the role.

Example 6-12 shows how to retrieve role with a Dojo toolkit client.

Example 6-12   Retrieve role with a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
var applicationSpaceName = "GenericApprovalGenericApproval";
var roleName = "Approver";

// Construct the URI for role
var url = baseURL + "appspaces/" + applicationSpaceName + "/roles/" +
roleName;

// Use GET method to retrieve the roles
dojo.xhrGet({
        url: url,
        handleAs: "json-comment-optional",
        load: function(data) {
             var role = data;
             console.log("HomePage:" + role.homePage);
             console.log("Description: " + role.description);
             console.log("Display Name: " + role.name);
             console.log("AuthoredName: " + role.authoredName);
             console.log("Attributes: " + role.attributes);
             var workbaskets = role.workbaskets;
            // Process the workbaskets
             for (var workbasketName in workbaskets) {
                 var workbasket = workbaskets[workbasketName];
                 console.log("Workbasket Display Name:" +
workbasket.name);
                 console.log("Workbasket URI:" + workbasket.URI);
                 console.log("Workbasket AuthoredName:" +
workbasketName);
             }
        },
        error: function(data) {
            // The error callback
            console.dir(data);
        }
    }
);
```

6.2.3 Retrieve workbasket

Using PE Java API you can retrieve a workbasket object from a Queue object.

The following steps are included to retrieve workbasket object:

1. Create a `VWSession object` and log on to PE.
2. Fetch VWQueue object from the VWSession object by providing the Queue name.
3. Fetch `VWWorkBasket` object from `VWQueue` object by providing the workbasket name.

Using Java API

Example 6-13 shows how to retrieve workbasket using Java API.

Example 6-13   Retrieve workbasket using Java API

```
// Create session object and log onto Process Engine
...
String queueName = "GenericApprovalQueue";
String workbasketName = "Approve";
VWWorkBasket workbasket;

VWQueue queue = session.getQueue(queueName);
workbasket = queue.fetchWorkBasket(workbasketName);

System.out.println("Workbasket name:" + workbasket.getName());
System.out.println("Workbasket authoredName:" +
workbasket.getAuthoredName());
System.out.println("Workbasket description:" +
workbasket.getDescription());

// retrieve the columns
filenet.vw.api.VWWorkBasket.Column columns[] = workbasket.getColumns();
if(columns != null) {
    for(int i = 0; i < columns.length; i++) {
        String name = columns[i].getName();
        String authoredName = columns[i].getAuthoredName();
        String prompt = columns[i].getPrompt();
        int type = columns[i].getType();
        boolean isSortable = columns[i].isSortable();
    }
}
// Retrieve the filers
filenet.vw.api.VWWorkBasket.Filter filters[] = workbasket.getFilters();
```

```
if(filters != null) {
    for(int i = 0; i < filters.length; i++) {
        String filterName = filters[i].getName();
        String authoredFilterName = filters[i].getAuthoredName();
        String description = filters[i].getDescription();
        String prompt = filters[i].getPrompt();
        int type = filters[i].getType();
        int operator = filters[i].getOperator();
    }
}
```

REST API scenario

The following steps are required to retrieve Workbasket

1. Send HTTP GET using the Workbasket resource.
2. Process the information of the Workbasket.

Example 6-14 shows how to retrieve workbasket using REST API.

Example 6-14   Retrieve workbasket using a Dojo toolkit client

```
// Ensure to be authenticated with the Application Server container
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
var queueName = "GenericApprovalQueue";
var workbasketName = "Approve";
var url = baseURL + "queues/" + queueName + "/workbaskets/" +
workbasketName;

// Use GET method to retrieve
dojo.xhrGet({
        url: url,
        handleAs: "json-comment-optional",
        load: function(data) {
            var workbasket = data;
            var columns = workbasket.columns;
            var filters = workbasket.filters;
            var description = workbasket.description;
            var displayName = workbasket.name;
            var authoredName = workbasket.authoredName;

            // Print the columns
            for (var columnName in columns) {
                var column = columns[columnName];
                console.log("Column DisplayName:" + column.name);
                console.log("Sortable:" + column.sortable);
```

```
                        console.log("Type:" + column.type);
                        console.log("Prompt:" + column.prompt);
                        console.log("Ordinal:" + column.ordinal);
                    }

                    // Print the filters
                    for (var filterName in filters) {
                        var filter = filters[filterName];
                        console.log("Filter DisplayName:" + filter.name);
                        console.log("Type:" + filter.type);
                        console.log("Description:" + filter.description);
                        console.log("Prompt:" + filter.prompt);
                        console.log("Operator:" + filter.operator);
                        console.log("Attributes URL:" + filter.attributes);
                    }
                },
                sync: true,
                error: function(data) {
                    // The error callback
                    console.dir(data);
                }
            }
        );
```

6.2.4 Query work items from workBasket

Using PE Java API, you can search work items from a Workbasket object. The
work items can be divided into pages by specifying the `pageSize` query
parameter.

The following steps are included to retrieve workbasket object:

1. Create a `VWSession` object and log on to PE.

2. Fetch VWQueue object from the VWSession object by providing the Queue
   name.

3. Fetch `VWWorkBasket` object from `VWQueue` object by providing the Workbasket
   name.

4. Use `fetchNextBatch` method and specify the query condition (such as
   pageSize, filter) to query first page of the work items.

5. Use `fetchNextBatch` method and the `lastRecord` object from the returned
   QueryResult object to query the next and following pages.

Using Java API

Example 6-15 shows how to query work items from workbasket using Java API.

Example 6-15   Query work Items from workbasket using Java API

```
// Create session object and log onto Process Engine
...
String queueName = "GenericApproval";
String workbasketName = "Index";
VWWorkBasket workbasket;
// the filter condition
String[] filterNames = {"CompanyName"};
String[] subsVars = {"XYZ"};

// Get the Queue
VWQueue queue = session.getQueue(queueName);
// Get the Workbasket
workbasket = queue.fetchWorkBasket(workbasketName);
int pageSize = 4;

System.out.println("Workbasket name:" + workbasket.getName());
System.out.println("Workbasket authoredName:"
        + workbasket.getAuthoredName());
System.out.println("Workbasket description:"
        + workbasket.getDescription());

// Query the first page
filenet.vw.api.VWWorkBasket.QueryResults results = workbasket
        .fetchNextBatch(
                VWQueue.QUERY_READ_LOCKED,  // queryFlags
                pageSize,              // buffer_size
                null,                  // sort column
                filterNames,           // filter names
                subsVars,              // subsitution vars
                null,                  // last record
                3                      // fetch type, 3 means Queue
element
                );

Object queueElements[] = null;
String lastRecord = null;

int pageNumber = 1;

while (results != null) {
```

```
        // save the last record to a local variable
        // it will be used to query the next page
        lastRecord = results.getLastRecord();
        queueElements = results.getWorkObjects();

        System.out.println("Process page: " + pageNumber);

        for (int i = 0; i < queueElements.length; i++) {
            VWQueueElement qe = (VWQueueElement) queueElements[i];
            System.out.println("WobNumber:" + qe.getWorkObjectNumber());
            System.out.println("QueueName:" + qe.getQueueName());
            System.out.println("StepName:" + qe.getStepName());
            filenet.vw.api.VWWorkBasket.Column columns[] =
workbasket.getColumns();
            if (columns != null) {
                for (int j = 0; j < columns.length; j++) {
                    String name = columns[j].getAuthoredName();
                    System.out.println("name:" + name);
                    Object value = qe.getDataField(name).getValue();
                    // Print out the value
                    System.out.println("Value:" + value.toString());
                }
            }
        }

        results = workbasket.fetchNextBatch(
                VWQueue.QUERY_READ_LOCKED,  // queryFlags
                pageSize,                   // buffer_size
                null,                       // sort column
                filterNames,                // filter names
                subsVars,                   // subsitution vars
                lastRecord,                 // last record
                3                           // fetch type, 3 means Queue
element
                );
        System.out.println("Next page");
        pageNumber++;
}
```

REST API scenario

The following steps are required to retrieve queue elements from workbasket:

1. Send HTTP POST using the queue elements resource to get the first page of queue element.

2. Process the returned results for the first page.

3. From the returned results for the first page, get the lastRecord object and save it to a local variable.

4. Use the saved lastRecord object to construct a queryString object and put it on the POST body and send the request to get the next page.

5. Process the returned result of the next page.

6. Continue to send POST requests until the returned result is empty.

Example 6-16 shows how to query queue elements from workbasket using REST API.

Example 6-16   Query queue elements from workbasket using Dojo toolkit

```
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
var queueName = "GenericApproval";
var workbasketName = "Index";
var pageSize = 4;

var url = baseURL + "queues/" + queueName + "/workbaskets/" + workbasketName +
"/queueelements";

// Helper method to print the queue elements
var printQueueElements = function(queueElements) {
    dojo.forEach(
        queueElements,
        function(queueElement) {
            console.log("workObjectNumber:" + queueElement.workObjectNumber);
            console.log("queueName:" + queueElement.queueName);
            console.log("lockedById:" + queueElement.lockedById);
            console.log("lockedBy:" + queueElement.lockedBy);
            console.log("ETag:" + queueElement.ETag);
            console.log("stepElementURL:" + queueElement.stepElement);
            for (var columnName in queueElement.columns) {
                var columnValue = queueElement.columns[columnName];
                console.log("columnName:" + columnName);
                console.log("columnValue:" + columnValue);
            }
        },
        this);
};
// Construct the query parameters
var queryData = "pageSize=" + pageSize + "&queryFlags=1";
```

```
var lastRecord;
var queueElements;
var pageNumber = 1;
// Query the first page
dojo.rawXhrPost({
        url: url,
        handleAs: "json-comment-optional",
        load: function(data) {
            lastRecord = data.lastRecord;
            queueElements = data.queueElements;
            printQueueElements(queueElements);
            pageNumber++;
        },
         _raw: true,
        sync: true,
        postData: queryData,
        error: function(data) {
            // The error callback
            console.dir(data);
        }
    }
);

// Query the next page
while(queueElements && queueElements.length > 0) {
    // Construct the query parameters
    // the lastRecord will be used to fetch the next batch result
    queryData = "pageSize=" + pageSize + "&queryFlags=1&lastRecord=" +
lastRecord;
    dojo.rawXhrPost({
            url: url,
            handleAs: "json-comment-optional",
            load: function(data) {
                lastRecord = data.lastRecord;
                queueElements = data.queueElements;
                printQueueElements(queueElements);
                pageNumber++;
            },
             _raw: true,
            sync: true,
            postData: queryData,
            error: function(data) {
                // The error callback
                console.dir(data);
            }
        }
    );
}
```

6.3 Resource navigation in Process Engine REST API

When navigating between REST resources, a best practice is to follow the URIs found in the response instead of manually constructing the URIs. Generally, the client application should have no prior knowledge beyond the initial URI of the REST API.

Resource navigation example

In Example 6-17, we start from Myroles resource, which returns a list of role resource URIs. We choose the first role resource URI and use it to retrieve the detailed information. The returned role information contains the URIs of the attached workbasket resource. Similarly, we follow the first workbasket resource URI to retrieve the detailed workbasket information.

Example 6-17   Resource navigation example using REST API

```
var baseURL = "http://localhost:9080/WorkplaceXT/P8BPMREST/p8/bpm/v1/";
var applicationSpaceName = "GenericApproval";

// Construct the URI for the initial myroles resource
var url = baseURL + "appspaces/" + applicationSpaceName + "/myroles";

var roleURI = null;
var workbasketURI = null;

// Use GET method to retrieve myroles,
// Save the first role's resource URI to local variable
dojo.xhrGet({
        url: url,
        handleAs: "json-comment-optional",
        load: function(data) {
            // Callback to handle the data
            for(var roleName in data) {
                var role = data[roleName];
                console.log("AuthoredName:" + roleName);
                console.log("Display Name:" + role.name);
                console.log("URI:" + role.URI);
                // construct the complete role URI
                roleURI = baseURL + role.URI;
                break;
            }
        },
        sync: true,
        error: function(data) {
```

```
                // The error callback
                console.dir(data);
            }
        }
    );

    if (roleURI === null) {
        console.log("roleURI is empty.");
        return;
    }

    // Use the saved URI and GET method to retrieve role resource
    // Save the URI of the first Workbasket to a local variable
    dojo.xhrGet({
            url: roleURI,
            handleAs: "json-comment-optional",
            load: function(data) {
                var role = data;
                console.log("HomePage:" + role.homePage);
                console.log("Description: " + role.description);
                console.log("Display Name: " + role.name);
                console.log("AuthoredName: " + role.authoredName);
                console.log("Attributes: " + role.attributes);
                var workbaskets = role.workbaskets;
                for (var workbasketName in workbaskets) {
                    var workbasket = workbaskets[workbasketName];
                    console.log("Workbasket Display Name:" +
    workbasket.name);
                    console.log("Workbasket URI:" + workbasket.URI);
                    console.log("Workbasket AuthoredName:" +
    workbasketName);
                    // construct the complete workbasket URI
                    workbasketURI = baseURL + workbasket.URI;
                    break;
                }
            },
            sync: true,
            error: function(data) {
                // The error callback
                console.dir(data);
            }
        }
    );

    if (workbasketURI === null) {
```

```
                console.log("workbasketURI is empty.");
                return;
            }

            // Use GET method to retrieve the information of the workbasket
            dojo.xhrGet({
                    url: workbasketURI,
                    handleAs: "json-comment-optional",
                    load: function(data) {
                        var workbasket = data;
                        var columns = workbasket.columns;
                        var filters = workbasket.filters;
                        var description = workbasket.description;
                        var displayName = workbasket.name;
                        var authoredName = workbasket.authoredName;

                        // print out the columns and filters
                        // ...
                    },
                    sync: true,
                    error: function(data) {
                        // The error callback
                        console.dir(data);
                    }
                }
            );
```

6.4 ECM Widgets overview

ECM Widgets is a mashup-based framework aiming to build business process
management (BPM) solutions rapidly and efficiently.

ECM Widgets is ideal for both the business and IT user. It provides a set of UI
components, allowing the business user to quickly build applications just by drag
and drop, without programming. It also allows the IT user to create custom
widgets that can be integrated with the existing ECM Widgets. In this book, we
focus on the second scenario.

In this section, we first give you an introduction to ECM Widgets. After that, we
show how to create a custom widget and integrate it with other ECM Widgets.

6.4.1 ECM Widgets concepts

Before working with ECM Widget, we first introduce the concepts of widget, WebSphere Business Space, In-basket, events, and pages. In addition, we introduce types of widgets, including In-basket widget, Step Completion widget, Attachment widget, Viewer widget, and Work Data widget.

Widget

ECM Widgets is a collection of widgets to assemble user interfaces for IBM enterprise content management and business process management applications. In the context of ECM Widgets, a widget means a user interface component that follows the iWidget specification. For information about the iWidget specification, see the IBM Mashup Center wiki:

http://www.lotus.com/ldd/mashupswiki.nsf/dx/widget-programming-guide

WebSphere Business Space

WebSphere Business Space is a mashup container that hosts the ECM Widgets and other widgets. It enables the designer to drag and drop widgets onto a mashup page, lay out the widgets, set the configuration of the widgets, and wire the widgets together to define communications between the widgets through events.

In-basket

In the context of ECM Widgets, In-basket has the same meaning as the workbasket that is defined in PE.

Events

All widgets that follow the iWidget specification communicate with each other through an event mechanism. The two types of events are: published event and handled event. For example, when a user clicks a button in Widget A, an event is published to another Widget B. To enable Widget B to subscribe to events sent from Widget A, you must wire Widget A with Widget B. This process is called *wiring widgets*. WebSphere Business Space has a graphical user interface (GUI) that allows you to wire the widgets.

Pages

The three types of mashup pages for ECM Widgets are: My Work page, Step Processor Page, and Single Page application.

A *My Work* page is a top-level page that displays one or more In-baskets for selecting work items. Figure 6-7 on page 203 shows an example of My Work page.
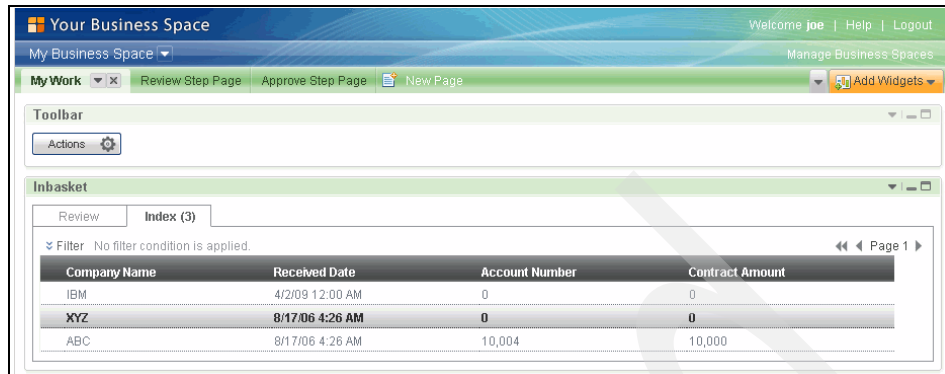
Figure 6-7 My Work page

A *Step Processor Page* is used to process a work item selected from an In-basket. A step processor page exists for each step in a workflow. Figure 6-8 shows an example.
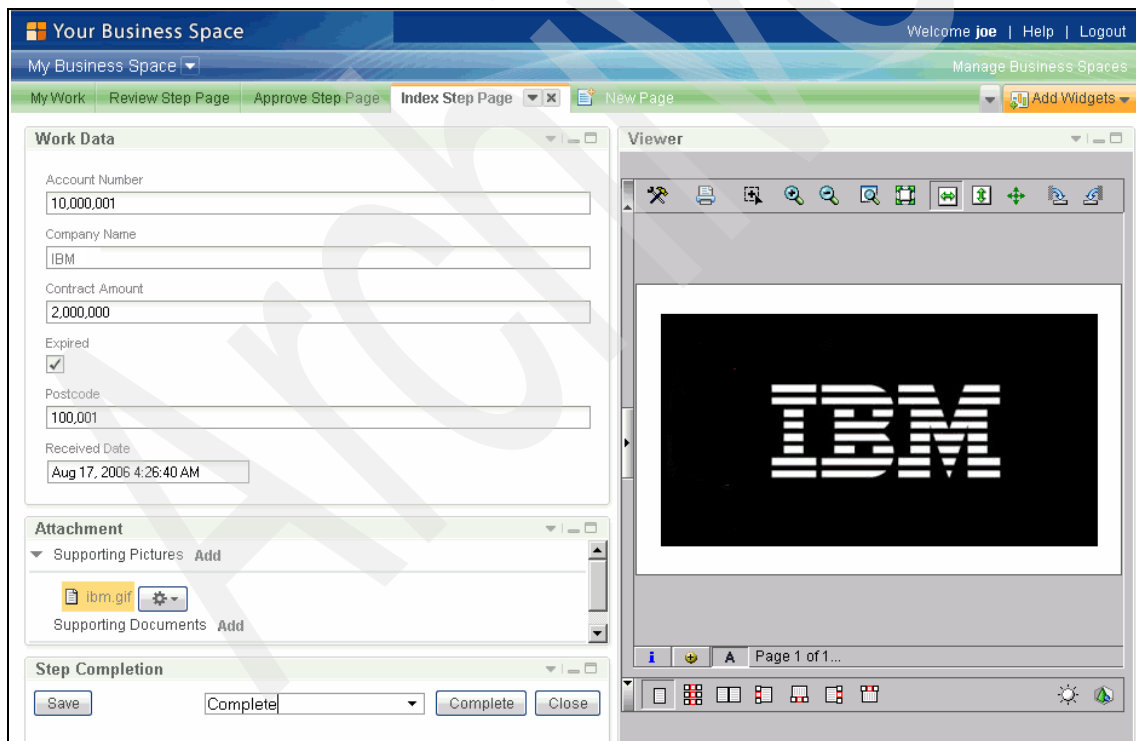
Figure 6-8 Step Processor Page

ECM Widgets also support a *Single Page* application. This means that all widgets are put into one mashup page, and no page-switching exists when a user opens a work item. Figure 6-9 shows an example.
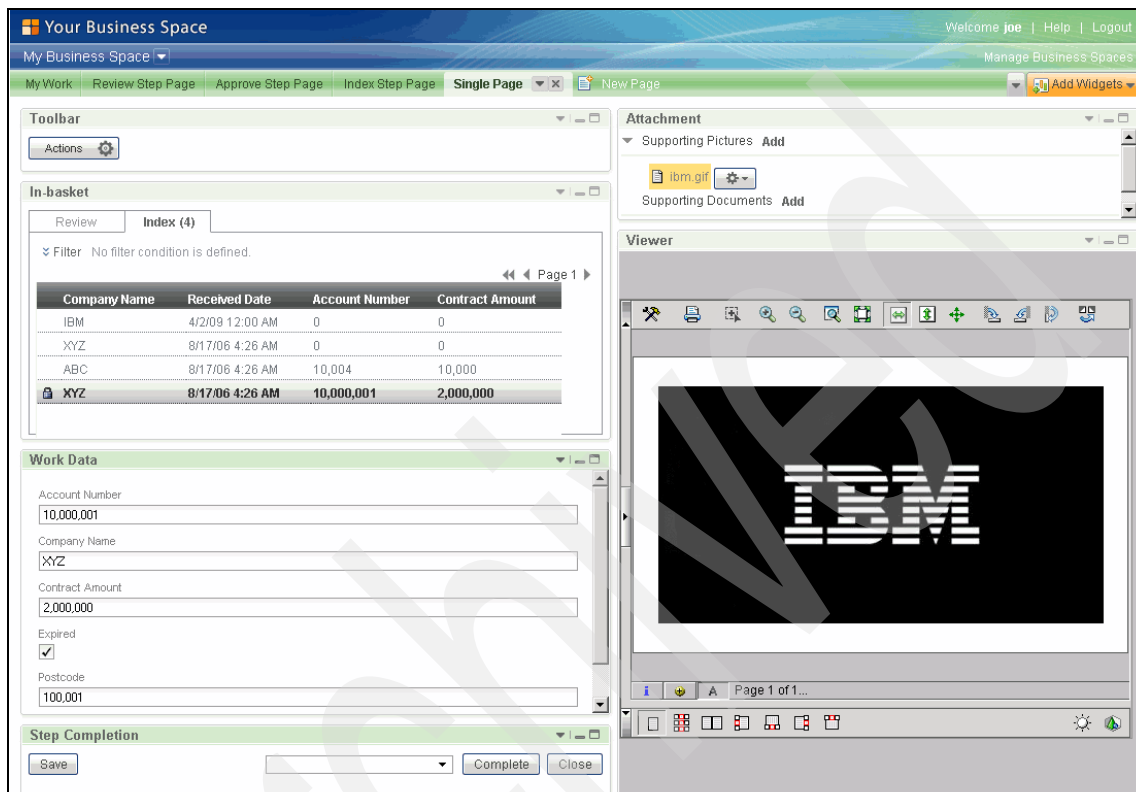
Figure 6-9   Single Page application

In-basket widget

The In-basket widget displays a list of In-baskets that belong to the current role. When you open an In-basket, it displays a list of work items according to the configuration defined in PE. When you double click a work item from the In-basket widget, it triggers an event as a notification that the user wants to open the work item. The In-basket widget can be placed on the My Work page or a Single Page application.

Step Completion widget

The Step Completion widget displays the workflow response list, executes responses, and saves or closes the work item. When this widget receives a "Work Item ID" payload, it uses this payload to retrieve a work item from PE and

broadcast the work item to other widgets, such as Attachment and Work Data widget.

Attachment widget

The Attachment widget displays attachment lists that are grouped by workflow fields. It provides document-related functions: checkout, checkin, cancel checkout, view, properties, download, and remove attachment.

Viewer widget

The Viewer widget wraps the WorkplaceXT Image Viewer applet for displaying document attachments.

Work Data widget

The Work Data widget displays customer-defined fields that are exposed on the step.

6.4.2 ECM Widgets system architecture

ECM Widgets is a framework that is based on mashup technology. Components in this framework follow the iWidget specification to communicate with each other. In ECM Widgets 4.5.1, we host the ECM Widgets in the WebSphere Business Space, which is a mashup container that provides a runtime to run the widgets, and also provides a GUI-based builder to assemble widgets to build applications.

At run time, ECM Widgets reside on the browser side. All widgets are written purely in JavaScript. ECM Widgets talk to the CE and PE through the REST interface to launch workflows, process work items, and search for documents within CE. They also leverage WorkplaceXT to accomplish several CE-related functions, such as checkout and checkin documents, view document properties.

Figure 6-10 on page 206 shows the system architecture of ECM Widgets.
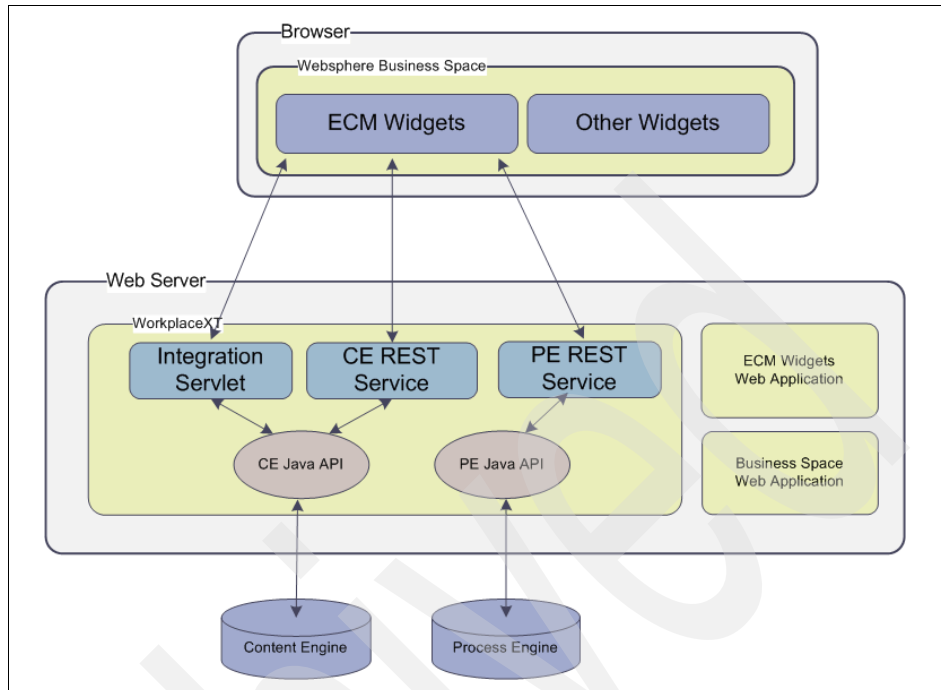
Figure 6-10   ECM Widgets system architecture

6.5 Building a custom Get Next In-basket widget

ECM Widgets enables the IT user to build new custom widgets and integrate them with the existing widgets. In this section, we present instructions for building a custom *Get Next* In-basket widget. As its name implies, this widget automatically opens the next work item from an In-basket when the current work item is completed. It also allows the user to open the next work item directly by clicking a **Get Next** button. Figure 6-11 on page 207 shows a mashup page that contains the Get Next widget and integrates it with ECM Widgets.

> **Note:** Before you build a custom widget, you must have ECM Widgets 4.5.1 installed in your IBM FileNet P8 system.
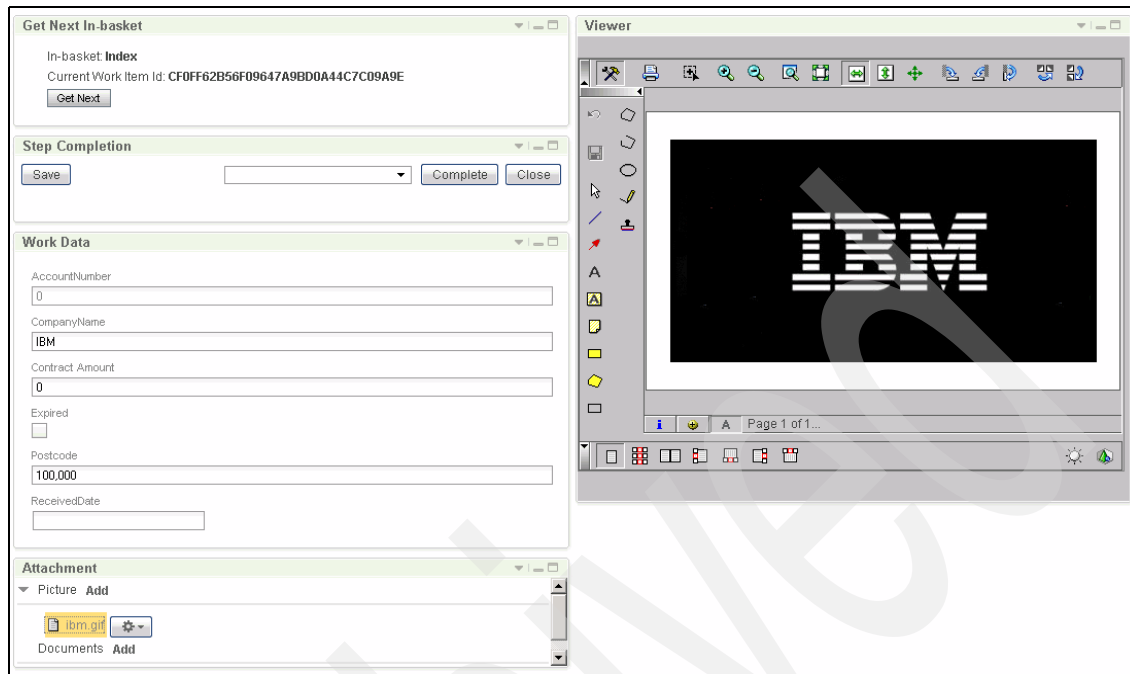
Figure 6-11   Get Next In-basket Widget working with the other ECM Widgets

6.5.1 Use case for the Get Next widget

When a user opens the mashup page that contains the Get Next In-basket widget, the widget retrieves the next queue element resource through PE REST service. Each queue element contains the identifier for a work item. This identifier is used to generate a work item ID payload that is sent to the Step Completion widget. In turn, the Step Completion widget uses the work item ID to fetch a complete work item object and broadcast it to the Work Data and Attachment widgets. Then, the Work Data and Attachment widgets consumes the work item payload and renders it.

After the user finishes modifying the work item and completes it by clicking the **Complete** button in the Step Complete widget, a Close Step Processor event is sent back to the Get Next widget. That drives Get Next widget to get the next queue element and start working on the next work item.

Another scenario also exists that lets the user open the next work item directly by clicking the **Get Next** button. When the button is clicked, the Get Next widget fetches the queue element, constructs the work item ID, AND publishes this

payload to the Step Completion widget. In turn, the Step Completion widget, Work Data widget, and Attachment widget will work on the next work item.

Figure 6-12 illustrates the use case sequence diagram.
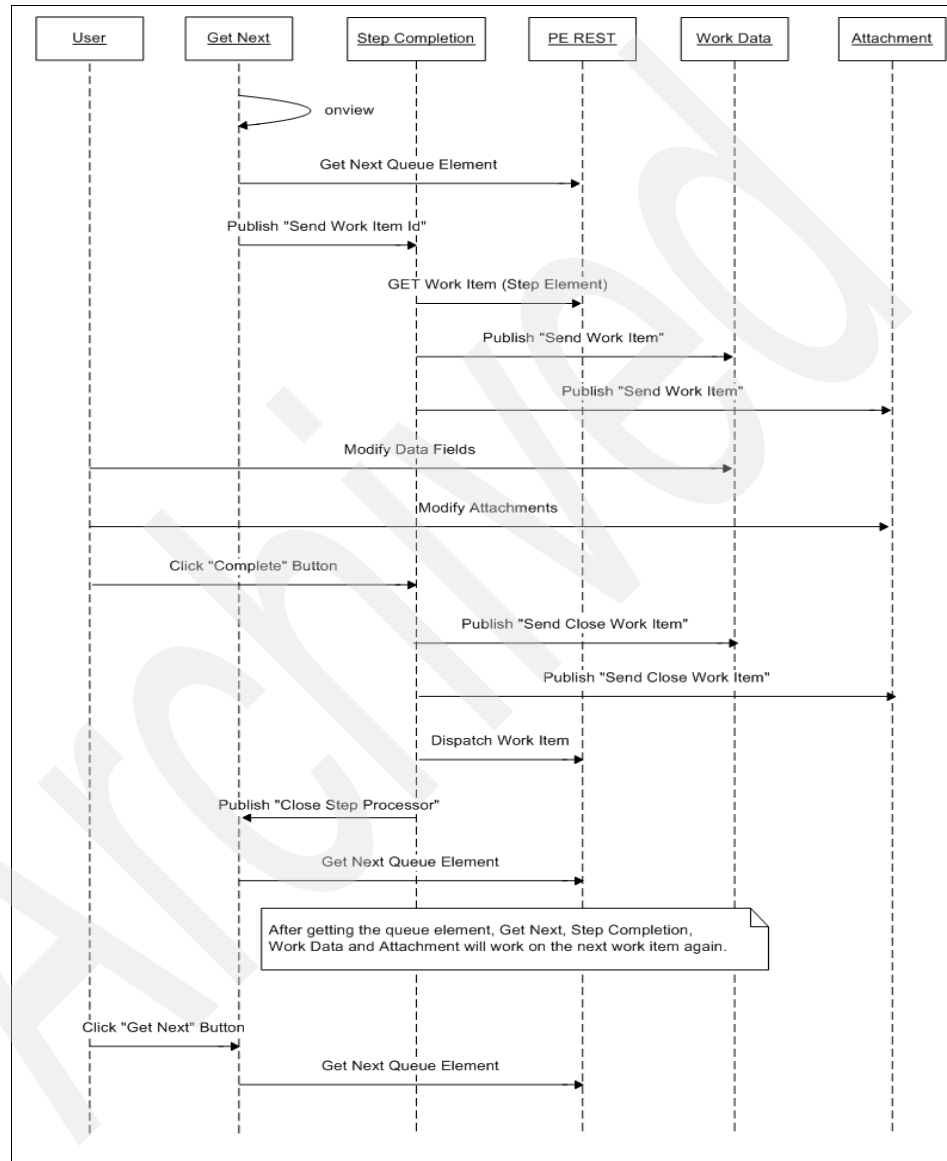
Figure 6-12   Use case for the Get Next widget

6.5.2 Setup development environment

To build this sample widget, create a Web application in Eclipse, as follows:

1. In Eclipse, select **File** → **New** to open the wizard dialog, then select the **Dynamic Web Project**. See Figure 6-13
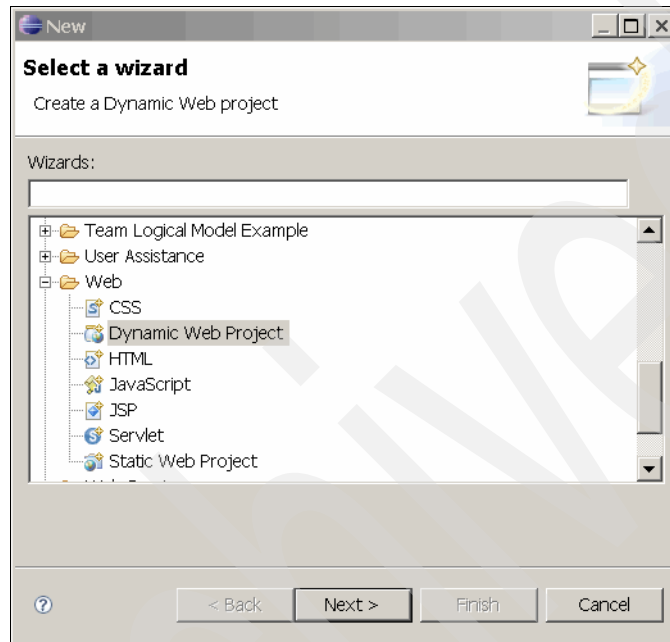
Figure 6-13   Create a new Web application

2. In the next dialog, specify the project name `SampleECMWidgets` (Figure 6-14 on page 210) and then click **Finish** to create the project.

In the created project, you will be able to edit the source code for this sample widget.
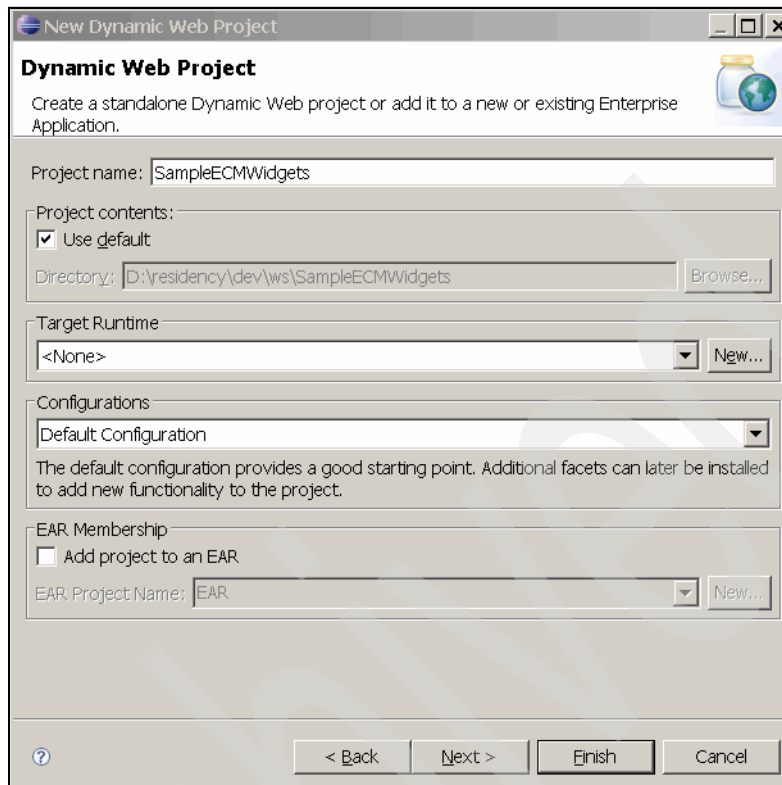
Figure 6-14   Specify the project name

6.5.3 Code structure of the Get Next widget

The Get Next widget is based on Dojo toolkit 1.0.2, which is bundled with WebSphere Business Space Web application. The widget code structure follows the normal Dojo package naming convention, which is presented in Figure 6-15 on page 211.
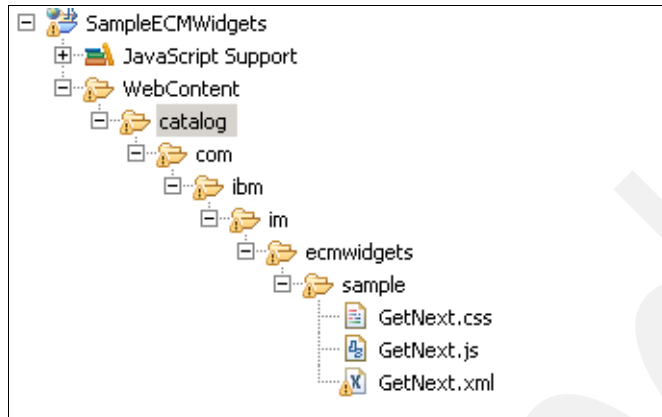
Figure 6-15   Code Structure of Get Next Widget

As shown in Figure 6-15, the code for the Get Next widget is divided into three files:

► `GetNext.xml`

   This file is the XML definition for the widget.

► `GetNext.js`

   This file contains the JavaScript logic to control the behavior of the widget.

► `GetNext.css`

   This file contains the look and feel of the widget. You need to add your cascading style sheet (CSS) definitions into this file.

6.5.4 Defining the Get Next widget

According to the iWidget specification, a widget can be defined in XML style. In `GetNext.xml` (as shown in Example 6-18 on page 212), we define various perspectives of this widget, which are:

► iScope object

   This object is defined by the iScope attribute of `iw:iwidget` element, which points to the `com.ibm.im.ecmwidgets.sample.GetNext` Dojo class.

► Widget attributes

   In the `iw:itemSet` element, there are several `iw:item` elements that contain attribute declarations. Table 6-6 on page 212 lists these attributes.

Table 6-6   Widget attributes

Attribute name	Description
bpmServiceBaseUrl	The base URL for PE REST Service. To simplify the sample code, we do not provide a GUI to configure this value. If you want to change it, modify the default value stored in `GetNext.xml`.
applicationSpaceName	The application space name for retrieving role list. To simplify the sample code, we do not provide a GUI to configure this value. If you want to change it, modify the default value stored in `GetNext.xml`.
inbasketName	The In-basket name used to retrieve queue elements. You can configure this value in the configuration panel of the Get Next widget.
queueElementsUrl	The URL of the queue elements resource. After you configure the role name and In-basket name in configuration panel, the URL of the queue elements resource for the selected In-basket will be saved into mashup page.

- ► `"Send Work Item Id"` event

  This is a published event. The three scenarios that trigger this event are:

  – The Get Next widget loads the first queue element.
  – A user clicks the **Get Next** button.
  – The Get Next widget receives a `"Receive Close Step Processor"` event.

- ► `"Receive Close Step Processor"` event

  This is a handled event. If you wire the Step Completion widget with the Get Next widget using this event, when a user clicks the **Complete** button in the Step Completion widget, an event will be published from the Step Completion widget to the Get Next widget. As defined in the onEvent attribute of this event, the event handler is `"handleReceiveCloseStepProcessor"` and which is a JavaScript function declared in the `GetNext.js` file.

Example 6-18   GetNext.xml

```
<iw:iwidget name="GetNext"
xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget"
iScope="com.ibm.im.ecmwidgets.sample.GetNext"
allowInstanceContent="true" supportedModes="view edit" mode="view">
    <iw:resource uri="GetNext.js"/>
    <iw:resource uri="GetNext.css"/>
    <iw:itemSet id="attributes">
```

```
            <iw:item id="bpmServiceBaseUrl"
value="/WorkplaceXT/P8BPMREST/p8/bpm/v1"/>
            <iw:item id="applicationSpaceName" value="GenericApproval"/>
            <iw:item id="inbasketName" value=""/>
          <iw:item id="queueElementsUrl" value=""/>
      </iw:itemSet>
      <iw:payloadDef name="WorkItemId">
          ....
      </iw:payloadDef>
      <iw:event id="Send Work Item Id" published="true"
eventDescName="SendWorkItemIdDescription"/>
      <iw:eventDescription id="SendWorkItemIdDescription"
payloadType="WorkItemId" description="" lang="en"/>
      <iw:payloadDef name="WorkItem">
          ...
      </iw:payloadDef>
      <iw:payloadDef name="CloseStepProcessor"></iw:payloadDef>
      <iw:event id="Receive Close Step Processor" handled="true"
onEvent="handleReceiveCloseStepProcessor"
eventDescName="ReceiveCloseStepProcessorDescription"/>
      <iw:eventDescription id="ReceiveCloseStepProcessorDescription"
payloadType="CloseStepProcessor" description="" lang="en"/>
      <iw:content mode="view">
          <![CDATA[
      <div id="_IWID_viewModeContent" class="ecmwdgt
ecmwdgtViewModeContent ecmwdgtGetNext"></div>
        ]]>
      </iw:content>
      <iw:content mode="edit">
          <![CDATA[
      <div id="_IWID_editModeContent" class="ecmwdgt
ecmwdgtEditModeContent ecmwdgtGetNext"></div>
        ]]>
      </iw:content>
</iw:iwidget>
```

6.5.5 Code skeleton for GetNext.js

The `GetNext.js` file is the iScope Dojo class that controls the behavior of the Get
Next widget. Example 6-19 on page 214 defines its basic structure in an
object-oriented way. Read the comments for each function to understand its
purpose.

Example 6-19   Code skeleton for Get Next widget: GetNext.js

```
dojo.provide("com.ibm.im.ecmwidgets.sample.GetNext");

dojo.declare("com.ibm.im.ecmwidgets.sample.GetNext", null, {

    onLoad: function(){
        // Will be invoked when mashup container loads this widget.
    },

    onview: function(){
        // Will be invoked when widget shows the view mode pane.
        this._renderViewModeContentNode();
    },

    onedit: function() {
        // Will be invoded when widget shows the edit mode pane.
        this._renderEditModeContentNode();
    },

    handleReceiveCloseStepProcessor: function() {
        // Handle the incoming "Close Step Processor" event.
        this._getNextQueueElement();
    },

    _getNextQueueElement: function() {
        // Get the next queue element and publish the
        // "Send Work Item Id" event.
    },

    _renderEditModeContentNode: function() {
        // Render the edit mode content pane.
    },

    _renderViewModeContentNode: function() {
        // Render the view mode content pane.
    },

    _request: function(method, xhrArgs) {
        // This is the helper method that request Ajax request to PE REST
        // service.
    }
});
```

6.5.6 Rendering the Get Next widget user interface

The Get Next widget supports the view and edit modes. When a widget switches to a certain mode, the corresponding handler function is invoked. According to the iWidget specification 1.0, for the view mode, the `onview()` function is called; for the edit mode, the `onedit()` function is called.

In this sample widget, the `onview` function calls `_renderViewModeContentNode` method to render the view mode GUI. Similarly, the `onedit` function calls `_renderEditModeContentNode` method to render the edit mode GUI. For the implementation of these two render methods, check the sample applications in this book.

Figure 6-16 and Figure 6-17 show the Get Next widget in view and edit modes.
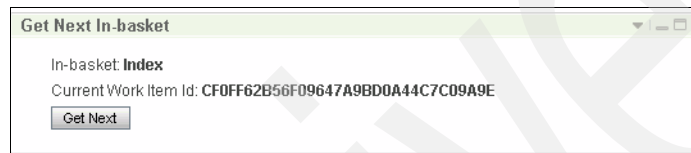
Figure 6-16   The Get Next widget in the view mode

Figure 6-17   The Get Next widget in the edit mode

6.5.7 Invoking PE REST service to fetch queue element

The `_request` function is a helper function that performs the request to PE REST service. It returns a Deferred Dojo object that allows you to register `callback` and `errback` functions. See Example 6-20.

Example 6-20   _request function

```
_request: function(method, xhrArgs) {
   if (!xhrArgs.url) {
```

```
            console.error("No target xhr url");
            var deferred = new dojo.Deferred();
            deferred.errback();
            return deferred;
        }
        xhrArgs.url = this.bpmServiceBaseUrl + "/" + xhrArgs.url;
        return dojo[method](xhrArgs);
    }
```

The _getNextQueueElement function is called when the Get Next widget wants to fetch the next queue element. This method constructs the URL for the queue element resource and registers a callback function _getNextQueueElementCallback to handle the return result. It then initiates the request by calling the _request function. See Example 6-21.

Example 6-21 _getNextQueueElement method

```
    _getNextQueueElement: function() {
        console.debug("[" + this.declaredClass + "." +
arguments.callee.nom + "]");
        var postData = "pageSize=1&queryFlags=1";
        if (this.lastRecord) {
            postData += "&lastRecord=" +
encodeURIComponent(this.lastRecord);
        }
        this._request("rawXhrPost", {
            url: this.queueElementsUrl,
            handleAs: "json",
            headers: {
                "Content-Type": "application/x-www-form-urlencoded"
            },
            postData: postData
        }).addCallback(this, "_getNextQueueElementCallback");
    },
```

The _getNextQueueElementCallback function handles the result coming from the PE REST service, getting the next queue element and publishing the data through the "Send Work Item Id" event. As a result, the target widget will be invoked to handle the incoming work item ID payload. See Example 6-22 on page 217.

Example 6-22   _getNextQueueElementCallback

```
_getNextQueueElementCallback: function(result) {
    this.lastRecord = result.lastRecord;
    var queueElements = result.queueElements;
    if (!queueElements) {
        console.info("No more work items.");
        return;
    }
    var queueElement = queueElements[0];
    var lockedById = queueElement.lockedById;
    if (lockedById !== 0 && lockedById !== this.currentUserId) {
        console.info("Work item is locked by another user: " +
lockedById);
        return this._getNextQueueElement();
    }
    var workItemId = {
        queueName: queueElement.queueName,
        workObjectNumber: queueElement.workObjectNumber
    };
    this.iContext.iEvents.fireEvent("Send Work Item Id",
"WorkItemId", workItemId);
    this.workItemIdNode.innerHTML = queueElement.workObjectNumber;
},
```

6.5.8 Invoking PE REST service to fetch role and In-basket list

The steps to fetch role and In-basket list are similar to the steps for queue element. For the detailed information, see the source code in the sample widget application. Also, you can find the sample code to fetch role in Example 6-12 on page 191 and sample code for In-basket in Example 6-13 on page 192.

6.5.9 Deploying the widget

After you finish coding, deploy the custom widget by following these steps:

1. Export the project as a WAR file from Eclipse, as follows:

   a. Select **File** → **Export**.

   b. In the Export dialog (Figure 6-18 on page 218), select **WAR file**, and save it to a location.
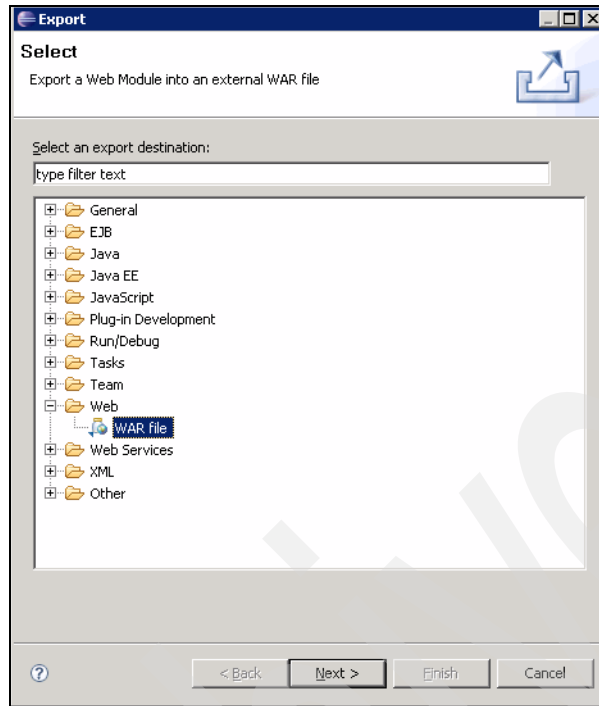
Figure 6-18   Export sample Web application

2. Deploy the WAR file to WebSphere. The context root for the deployed Web application should be `SampleECMWidgets`.

3. Stop the Web application of WebSphere Business Space.

4. Copy the widget registration file of the Get Next widget to the Business Space's registry data directory. You can find the sample registration file in the sample widget application that you download from IBM Redbooks Web server. Refer to Appendix A, "Additional material" on page 315 to locate the information.

5. Start the Web application of Business Space. If you deploy the Get Next widget successfully, the widget appears in the widget toolbox of Business Space (Figure 6-19 on page 219) and you can drag and drop the Get Next widget into a mashup page.
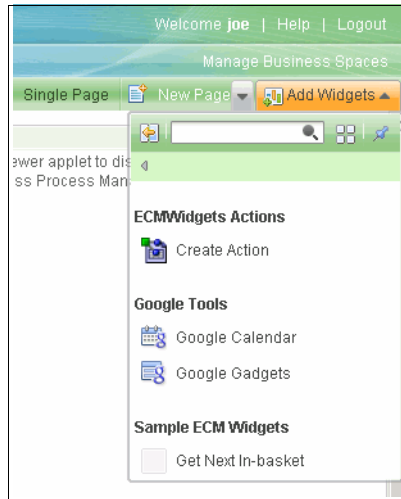
Figure 6-19   The Get Next widget in Business Space toolbox

6.5.10 Building the solution

As an example, to build a solution with the Get Next widget, follow these steps:

1. Open a space in WebSphere Business Space. Create a new mashup page, and name it `Get Next`. See Figure 6-20.
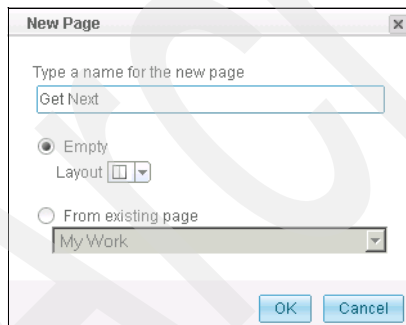
Figure 6-20   Create mashup page

2. Drag and drop the Get Next In-basket, Step Completion, Work Data, Attachment, and Viewer widgets to the page. See Figure 6-21 on page 220.
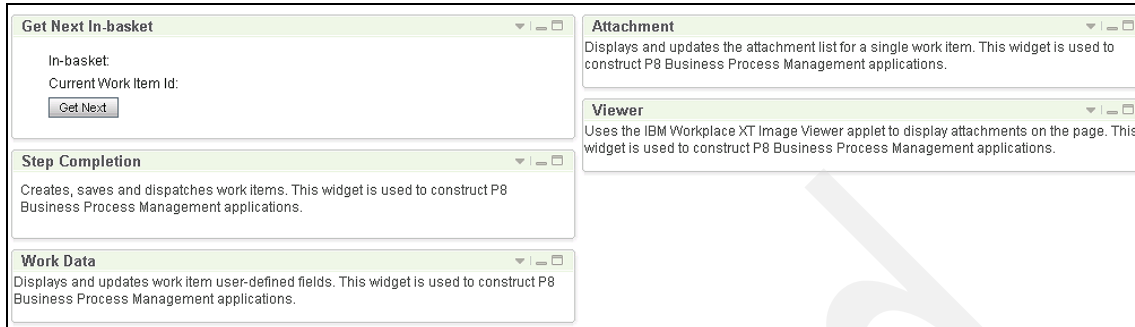
Figure 6-21   Place the widgets in the mashup page

3. Wire the Get Next In-basket widget with the Step Completion widget by the "Send Work Item Id" published event and "Receive Work Item Id" handled event. See Figure 6-22 for the widget wiring.
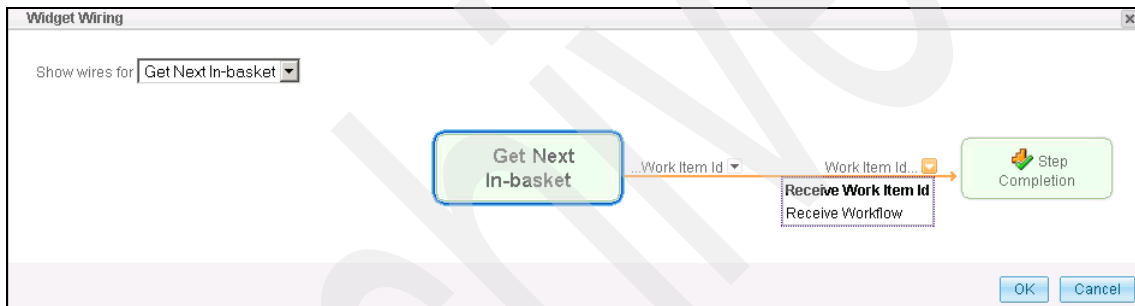
Figure 6-22   Wire Get Next and Step Completion by "Send Work Item Id" and "Receive Work Item Id" events

4. Wire the Step Completion widget with the Get Next widget by the "Send Close Step Processor" published event and "Receive Close Step Processor" received event. See Figure 6-23 for the widget wiring.
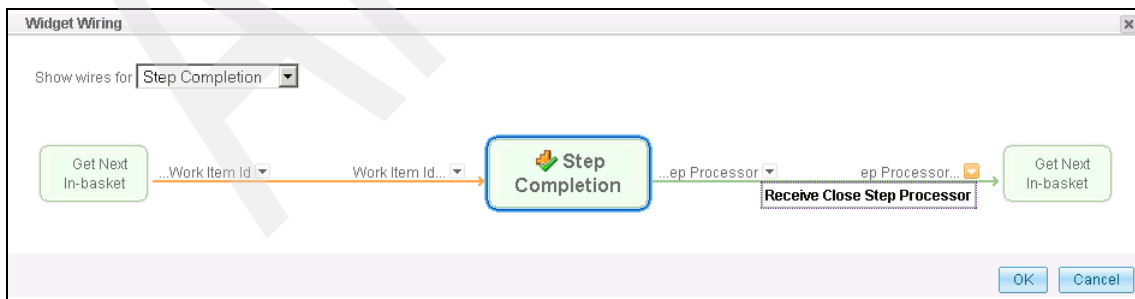
Figure 6-23   Wire Step Completion and Get Next by "Send Work Item Id" and "Receive Work Item Id" events

5. Open the configuration panel of the Get Next widget, select a role name and an In-basket name. See Figure 6-24. Click **Save**.

As a result, the Get Next widget starts to run and the other widgets display the content of current work item.

Figure 6-24   Configure the Get Next widget

7

Sample applications for Fictional Auto Rental Company A

To help you better understand programming with IBM FileNet APIs, in this chapter, we describe the sample applications that are implemented for Fictional Auto Rental Company A.

This chapter discusses the following topics:

► Introduction to sample applications

► Business use cases

► User view of the sample applications

► Data model

► Security model

► Workflows

► Internal architecture of sample applications

► Deployment instructions for sample applications

7.1 Introduction to sample applications

To tie together many of the development concepts that are discussed in other chapters, we present a set of sample applications. These sample applications include:

► Reservation Web application

► Kiosk application

► Agent handheld application (not implemented)

► Fleet Status Manager Web application

► Billing Report application

DISCLAIMER OF WARRANTIES: The accompanying code is sample code, created by IBM Corporation. This sample code is not part of any standard or IBM product and is provided to you solely for the purpose of assisting you in the development of your applications. The code is provided *as is*, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample code, even if they have been advised of the possibility of such damages.

The sample applications are available for download, in both source and binary form, from the IBM Redbooks publications Web site as a companion to this book. You can download them and then deploy them in your environment with only a small amount of configuration and setup. For downloading instructions, see Appendix A, "Additional material" on page 315. For deployment instructions, see 7.8, "Deployment instructions for sample applications" on page 276.

Our aim in designing these samples was to illustrate various programming and architectural techniques without trying to add them artificially. The sample applications are complete for what they are designed to do. You can easily add more features to make them a complete business solution. Although we have tested the applications to make sure that they work correctly for our own narrative purposes, and although we make the entire sample available to download for examination and study, we do not guarantee that it is free of defects.

The company, business model elements, and business use cases for the sample applications are all completely fictional. We believe them to be realistic in limited circumstances, but we make no claims for applicability or fitness for any real circumstances.

The sample code has hard-coded English language strings, U.S. conventions for date display, and so on. We are firm believers in globalization and localization of

software, but we developed the sample without it because we believe it is somewhat easier to follow as sample code. If we were to develop this as a production application, we would definitely follow best practices for globalization and localization. Likewise, and for similar reasons, the sample applications do not adhere to best practices for accessibility. That would also be necessary and desirable for production applications.

7.2 Business use cases

Fictional Auto Rental Company A ("the company") is a large, well-financed franchiser of vehicle rental operations. The company differentiates itself from the competition by having each local franchise offer a unique and varied mix of vehicles for rental to the public. As with any business, and especially for a large franchiser, the company looks for ways to keep costs low by exploiting automation. Many aspects of running the business are automated. To keep internal costs reasonable, franchisees use services from a high-end data center that is operated by the company headquarters. There are smaller computers at franchise locations, but servers that are running J2EE application servers are shared among many franchises.

The primary interaction for customers to reserve vehicles is through a Web site. In contrast to mainstream rental companies, a customer of the company selects a specific vehicle. The customer can see a photo of the actual vehicle, read comments from other customers, and even see limited facts about the vehicle maintenance history.

Using that same Web site, customers can register and maintain a user profile of personal factual information (for example, address and phone number) and preferences (for example, default class of vehicle). The unique nature of the company inventory of vehicles means that some rentals are done when a particular vehicle is available, so customers can sign up for alerts for specific vehicle availability and other events.

Back-office applications keep track of vehicle scheduling, expected pick-ups and drop-offs, rental history, and maintenance. Maintenance can be routine (for example, an oil change, typically handled by in-house staff) or exceptional (for example, repair of body damage, typically handled by an outside service contractor).

Employees of the company want to be able to check on the availability and historical records for a specific vehicle, but they also want dashboard functions, which will tell them quickly how many vehicles are in each of various status categories (for example, out for rental, in maintenance, idle, and overdue for return).

7.3 User view of the sample applications

In this section, we discuss the applications as seen from the point of view of the users. For our sample applications, the users are employees and customers of the company and its franchisees.

7.3.1 User view: Reservation Web application

For a description of the internal architecture of this application, see 7.7.1, "Architecture: Reservation Web application" on page 268.

For a description of the deployment instructions for this application, see 7.8.4, "Deployment: Reservation Web application" on page 292.
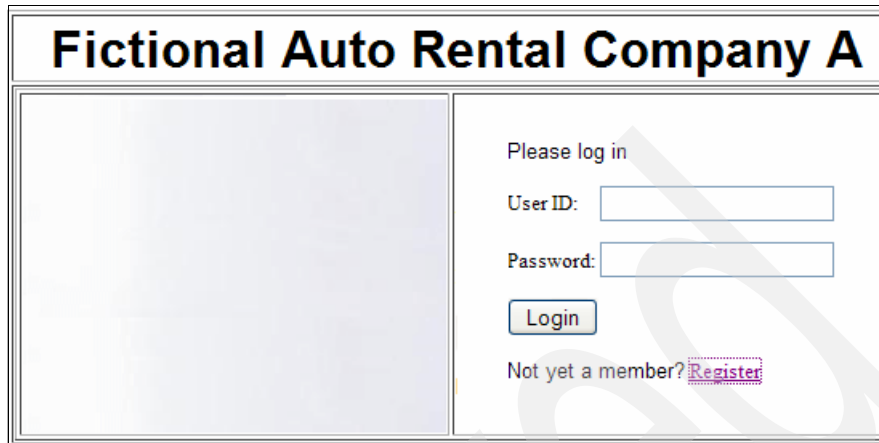
Vehicle reservations are made primarily through a Web site. The site has some relatively static information (for example, information about the company and information about services available). Web site content is also available, is dynamic in nature, and is driven by ECM systems.

The Reservation Web application provides the following functions. For simplicity, not all of these features are implemented in the sample code.

► Customers can browse the available inventory of vehicles and view information related to any particular vehicle.

► A customer can create an account with user ID and password. After creating an account, the customer can enter information in a user profile.

► A customer can rent a vehicle that is available for a specific date and location.

   To rent a vehicle, the customer must have an account, and must also supply certain information (for example, address, telephone number, credit card information). The information defaults to information from the user profile and gives an option to save any updates back to the profile.

► The customer is given the option of permanently storing or not storing the credit card information that is used for a given rental transaction.

► The customer can reset the password and have the new password sent through e-mail.

► With each confirmed vehicle reservation, a confirmation number is provided for the customer for rental pickup.

Several screen captures of the Reservation Web application are shown in the remaining figures in this section.
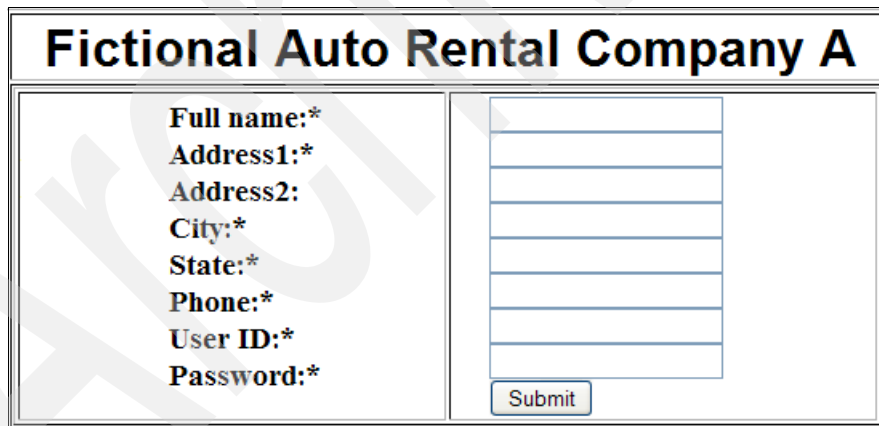
Figure 7-1 shows the login panel.

Figure 7-1   Reservation Web application login panel

If not a member, the user must register in the system. Figure 7-2 shows the
registration panel. The user must enter his or her information including a new
user ID and password that are required when reserving a vehicle from this
application.

Figure 7-2   Reservation Web application registration panel

Figure 7-3 shows the main reservation data panel. This is where the user can select where and when to pick up the automobile.

Figure 7-3 Reservation Web application main data panel

After the user selects where and when to rent a vehicle, the system displays a list of available vehicles to be rented during this period of time in that location, as shown in Figure 7-4. The user selects a vehicle of his or her choice. Each vehicle has a photo associated with it, a brief description, and a daily rate for rental.

Figure 7-4 Reservation Web application vehicle selection panel

After selecting the rental vehicle, the user enters the information of the credit card that is used for the reservation. Figure 7-5 shows the credit card information collection panel.
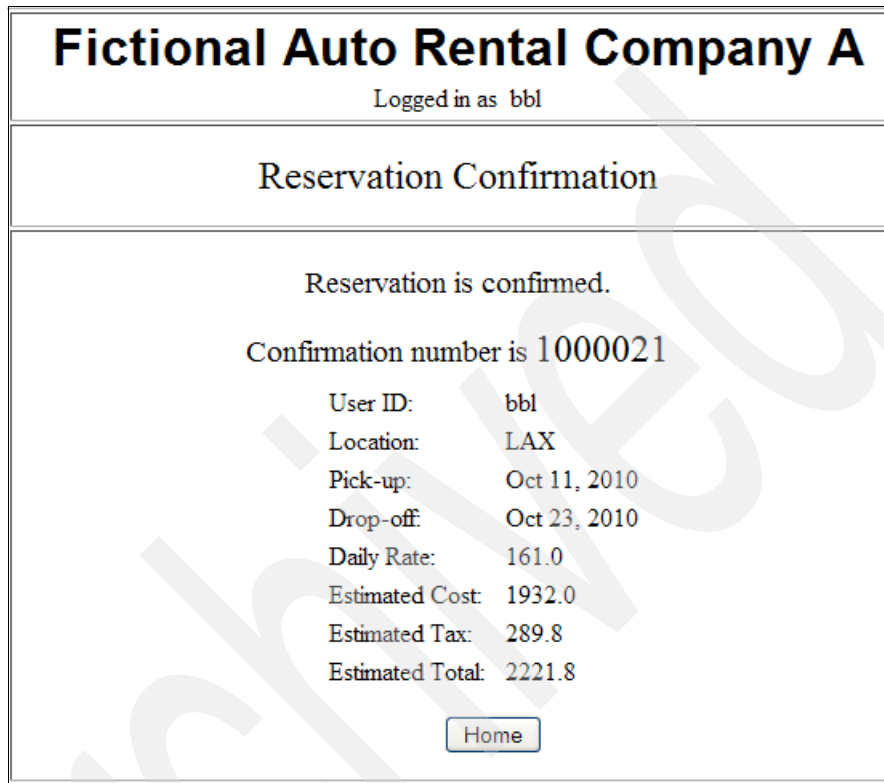
Figure 7-5   Reservation Web application credit card information collection panel

When the reservation is complete, the system displays a reservation confirmation panel. Figure 7-6 shows the confirmation panel for a successful rental reservation.

Figure 7-6   Reservation Web application rental reservation confirmation panel

7.3.2 User view: Kiosk application

For a description of the internal architecture of this application, see 7.7.2, "Architecture: Kiosk application" on page 270.
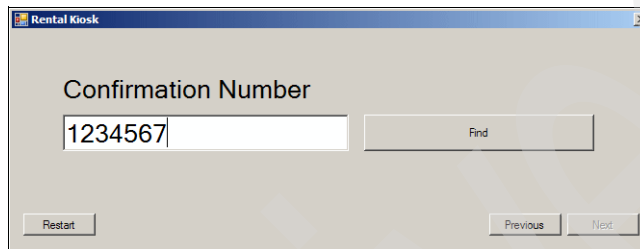
For a description of the deployment instructions for this application, see 7.8.5, "Deployment: Kiosk application" on page 295.

The company uses a PC-based Kiosk application for a variety of functions. Many of these functions depend on hardware, which you probably do not have, so we simulate these actions with other user interface elements.

Upon approaching the kiosk, the user (who is the customer) is presented with options for either picking up or dropping off a car.

At the rental counter when picking up a car, the customer enters the rental car confirmation number (provided in the confirmation panel as shown in Figure 7-6). The Kiosk application matches the confirmation number to the reservation and (in reality) physically delivers the keys and rental paperwork to the customer. In our samples however, the Kiosk application simulates delivery of the keys and paperwork by supplying two codes. One code identifies the vehicle being rented, and the other code represents this specific rental transaction.
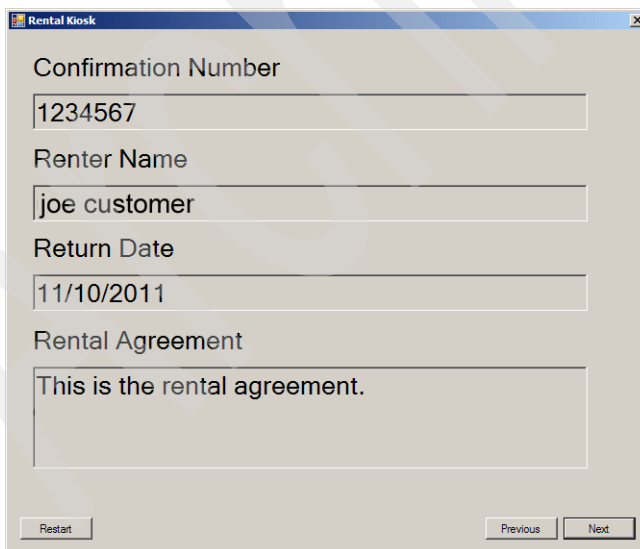
Figure 7-7 shows the interface for Rental Kiosk application where the user enters the rental car confirmation number.

Figure 7-7   Entering confirmation number

Figure 7-8 shows the rental information based on the confirmation number entered.

Figure 7-8   Rental information dialog

When a vehicle returns to the rental lot, a sensor scans a vehicle bar code from a window sticker. We simulate this by having the user enter the code using the keyboard. See Figure 7-9.
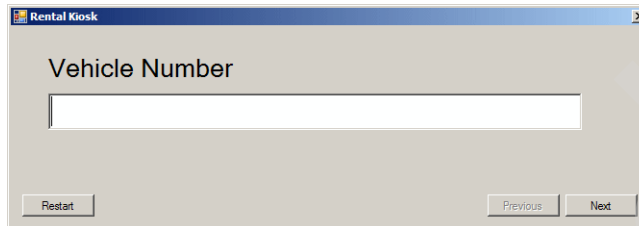
Figure 7-9   Vehicle return dialog

7.3.3 User view: Agent handheld application

When a vehicle is returned from rental, an employee of the company accepts the vehicle from the customer. Agents use a custom-designed handheld device. The device communicates with the back-office systems through a radio link.

The agent handheld application provides the following functions:

► Agents scan the bar code from the window sticker of the vehicle.

► On-board diagnostics (OBD) are read from the vehicle OBD II port and stored with the vehicle's maintenance records. The diagnostics might indicate the need for a certain kind of maintenance.

► If it is not available from the on-board diagnostics data (which varies from vehicle to vehicle), the vehicle mileage and fuel level are manually recorded by the agent.

► The agent also records any exceptional conditions, such as body damage, or missing parts or accessories.

In the sample application, we do not show the agent handheld application.

7.3.4 User view: Fleet Status Manager Web application

For a description of the internal architecture of this application, see 7.7.3, "Architecture: Fleet Status Manager Web application" on page 272.

For a description of the deployment instructions for this application, see 7.8.6, "Deployment: Fleet Status Manager Web application" on page 296.

Most minor maintenance is done by an in-house maintenance shop collocated with the rental office. The company keeps meticulous maintenance records by

using IBM FileNet P8 system. As part of the sample, we do not provide an implementation of the Kiosk application used by the vehicle technicians to record low-level maintenance details (for example, parts used and breakdown of labor), but we do provide a Web application used by the maintenance supervisor to accept, schedule, and release vehicles. The same Web application is used by rental office staff because maintenance is just one aspect of tracking vehicle availability.

The Fleet Status Manager Web application provides the following functions:

► A fleet status dashboard mode provides a summary of the status of all vehicles in the inventory (for example, out for rental, idle, in maintenance).

► A vehicle may be put into maintenance status by authorized employees. This can be well before the vehicle is actually moved to the maintenance area. The effect is that the vehicle is unavailable for future rentals during this period.

► The Kiosk application is used to scan the window sticker bar code as the vehicle is moved physically into or out of the maintenance area. We do not show that aspect in the sample application.

► An authorized maintenance employee may enter or update a maintenance substatus for the vehicle (for example, waiting for parts, working, sent to external service vendor). Most substatus changes require an estimated completion time for that status.

► An authorized maintenance employee may take a vehicle out of maintenance status, typically to return it to service.

► As a special case of the maintenance application, newly-acquired vehicles are added to the inventory and scheduled for an initial maintenance examination. Likewise, when a vehicle is being sold or disposed of, it gets a final maintenance inspection as it is removed from the active inventory.

Several screen captures of this Fleet Status Manager Web Application are shown in the remaining figures in this section.

Figure 7-10 shows the Fleet Status Dashboard page, which lists vehicles in inventory and their status. Supervisor users have access to this page through security roles.

Id	Year	Make	Model	Type	Status	Activity S
ABI4SQ0ZBPIXB469U8ST	2007	Tesla	Testarossa	Ex	ITSOIdleActivity	Tue Apr 07 15:2
QJATUA8IZAHGD0EF0LXA	2008	Audi	Spyder	Ex	ITSOIdleActivity	Tue Apr 07 15:2
Z9HTL73JPA7ZWDVG8N1P	2008	Alfa Romeo	Spyder	Ex	ITSOIdleActivity	Tue Apr 07 15:2
MWK55MSJJSS6KK1ZE4XA	2006	Tesla	Countach	Ex	ITSOIdleActivity	Tue Apr 07 15:2
08BCFPMQXFGCMD5LYQRR	2000	Ferrari	Spyder	Ex	ITSOIdleActivity	Tue Apr 07 15:2
YX9IA8UTLHQMR94DU92K	2003	Tesla	Testarossa	Ex	ITSOIdleActivity	Tue Apr 07 15:2
YG6J2OTY4Z8FE2VUDID0	2006	Alfa Romeo	Model S	Ex	ITSOIdleActivity	Tue Apr 07 15:2
9WI8WZR7MP9Y98KAPW3X	2004	Alfa Romeo	Model S	Ex	ITSOIdleActivity	Tue Apr 07 15:2
7NSNQJ8VBV1MRX6IEFVE	2002	Ferrari	Testarossa	Ex	ITSOIdleActivity	Tue Apr 07 15:2
8XE8FCMVT5CSOBNJPYKM	2000	Tesla	Countach	Ex	ITSOIdleActivity	Tue Apr 07 15:2
ITSOVEHICLE7	1978	Honda	CVCC	OL	ITSOIdleActivity	Thu Apr 09 15:3
ITSOVEHICLE0	1978	Honda	CVCC	OL	ITSOIdleActivity	Thu Apr 09 15:3
ITSOVEHICLE1	1978	Honda	CVCC	OL	ITSOIdleActivity	Fri Apr 17 09:27
ITSOVEHICLE2	1978	Honda	CVCC	OL	ITSOIdleActivity	Fri Apr 17 14:11
ITSOVEHICLE3	1978	Honda	CVCC	OL	ITSOIdleActivity	Fri Apr 17 14:13
ITSOVEHICLE4	1978	Honda	CVCC	OL	ITSOIdleActivity	Fri Apr 17 14:14
ITSOVEHICLE5	1978	Honda	CVCC	OL	ITSOIdleActivity	Fri Apr 17 14:35
ITSOVEHICLE6	1978	Honda	CVCC	OL	ITSOIdleActivity	Fri Apr 17 14:35
ITSOVEHICLE8	1978	Honda	CVCC	OL	ITSOMaintenanceActivity	Fri Apr 10 15:19
ITSOVEHICLE9	1978	Honda	CVCC	OL	ITSOMaintenanceActivity	Fri Apr 10 15:19

Figure 7-10   Fleet Status Dashboard

When the supervisor selects one of the vehicles from the inventory, the vehicle information appears, as shown in Figure 7-11. The supervisor can then send the vehicle to maintenance.

Figure 7-11   Fleet Manager page to send vehicle for maintenance

When a maintenance worker logs into this application, the worker is routed to a page that lists the vehicles in the maintenance group queue as well as those in that specific worker's inbox queue. These vehicles are represented by Process Engine (PE) work items. Figure 7-12 shows a typical page that a maintenance worker sees after logging in.

Figure 7-12   Maintenance queue page

The maintenance worker can select a vehicle from the Unassigned Work Items list, and route the item to his or her own work queue. See Figure 7-13.

VIN:	ITSOVEHICLE8
Year:	1978
Make:	Honda
Model:	CVCC
Type:	OL
Status:	Maintenance
Wob Num:	8B92B1F7D2300140BD99EE068CD91E28
Maintenance Code:	service
Maintenance Comment:	5000 miles service

Route work to my Queue

Figure 7-13   Maintenance worker page: route work item to worker's work queue

After the vehicle is in the maintenance worker's work queue, the maintenance worker can complete the work item or send it back to the group queue. See Figure 7-14.

Car in my work queue

VIN:	ITSOVEHICLE7
Year:	1978
Make:	Honda
Model:	CVCC
Type:	OL
Status:	Maintenance
Wob Num:	CBA25C5103600E4489E828F2B626AE68
Maintenance Code:	brakes
Maintenance Comment:	change front brakes

Send back to Group Queue

Complete Work

Description of Work Performed:

Figure 7-14   Maintenance worker page: process work item

Figure 7-15 shows the maintenance queues page when logging in as a different maintenance worker. The purpose of this picture is to show that the group queue work items are available to all users who have access to this queue, although the user's inbox only shows that user's assigned work items.

Figure 7-15   Maintenance queue page for a different user

All of these work items can be tracked by using the Process Administrator applet in WorkplaceXT. Figure 7-16 shows the maintenance work items in the roster. The F_BoundUser column shows which user's inbox that the work item is assigned to.

Figure 7-16   Process Administrator showing work items in roster

7.3.5 User view: Billing Report application

For a description of the internal architecture of this application, see 7.7.4, "Architecture: Billing Report application" on page 274.

For a description of the deployment instructions for this application, see 7.8.7, "Deployment: Billing Report application" on page 299.

Although we refer to it as the Billing Report application, it is really a collection of reports run periodically for the purpose of running the company's business. These can be run on demand, but they are typically run daily, during off hours, and cover the previous 24-hour period.

The Billing Report application provides the following functions:

- ► Summary and details of all vehicle hours by status (for example, out for rental, idle, and in maintenance).

- ► Summary and details of all payments made (credit card charges) for completed rentals.

- ► Summary and details of all maintenance expenses (parts, labor, third party).

- ► Exception report of all unusual events (for example, vehicle not ready when promised to a customer, vehicle overdue for return, vehicle entering non-routine maintenance).

In the sample application, we only show the summary and details report for vehicles hours by status.

Several screen captures of this Billing report application are shown in the remaining figures in this section.

Figure 7-17 shows the login dialog for this application. The user must enter an IBM FileNet P8 user ID and password to log in.
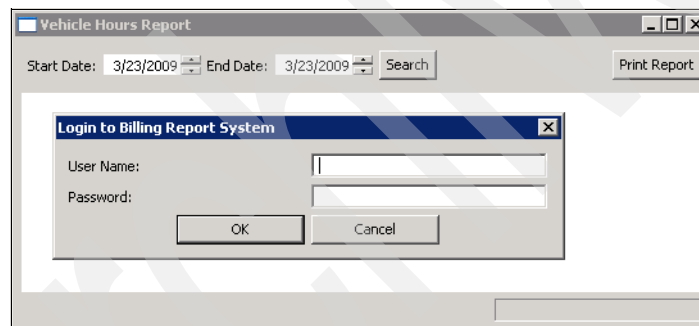
Figure 7-17 Login dialog for Billing Report application

If the user enters wrong information, the Billing Report application raises an error message, as shown in Figure 7-18.
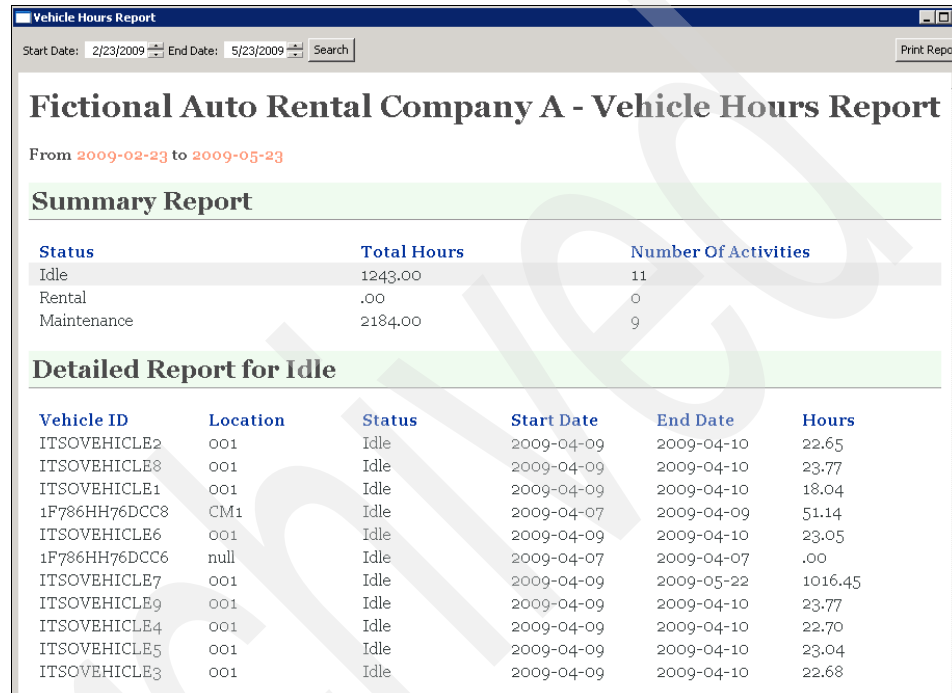
Figure 7-18 Login failed

After login, the user provides the start date and end date to generate the report. Figure 7-19 shows the generated report displayed in the Billing Report

application. The vehicle hours report contains two sections: Summary Report and Detailed Report.

The Summary Report lists the total hours and total numbers of activities for the various status types of the vehicle. By clicking a row in Summary Report, the Detailed Report is refreshed and lists the corresponding detailed information.

Figure 7-19   Main window for Billing Report application

For this sample application, we did not implement the Print Report function.

7.4 Data model

Many types of data are involved in running the sample applications. We concentrate here only on the data that resides within the IBM FileNet P8 back-end systems and used by the sample applications. These consist of custom classes and properties defined within the Content Engine (CE) metadata.

Figure 7-20, Figure 7-21, and Figure 7-22 provide a Unified Modeling Language (UML) overview of the classes. To keep the figures uncluttered, relationships from individual UML attributes (corresponding to CE properties) are not shown.

They can be inferred from the attribute data types in the diagrams and also from the explanatory material in the tables in this chapter. The data file for the UML diagrams (created with IBM Rational® Software Architect 7.5) is included in the downloadable material that accompanies this book, in file:

`sg247743-sample/CEartifacts/FictionalAutoRentalCompanyADataModel.emx`

In the diagrams and in the explanations that follow, we use the class and property symbolic names. These are the unique, case-insensitive, non-localizable names that are used by applications. Classes and properties also have localizable display names. A common general pattern for display names is to create the English version by removing the prefix from the symbolic name and inserting spaces between words in the symbolic name. For example, a symbolic name of `ITSOVehicleActivity` could have a display name of `Vehicle Activity`. In the data import files for the sample applications, we use display names that are the same as the symbolic names.

Figure 7-20 shows the model for ITSODocument class and its subclasses.
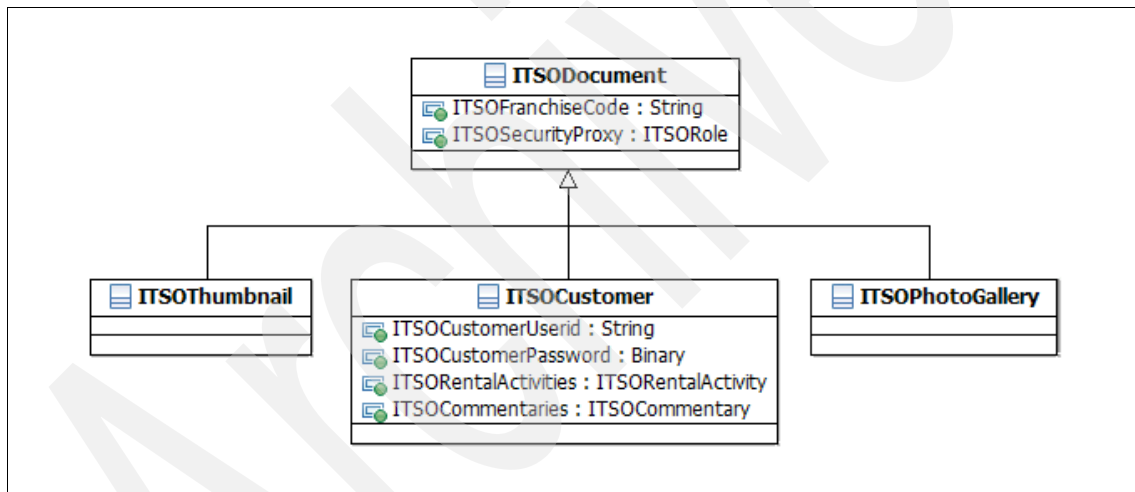
Figure 7-20   Model for ITSODocument and subclasses

Figure 7-21 shows the models for ITSOFolder, ITSOAnnotation, and their subclasses.
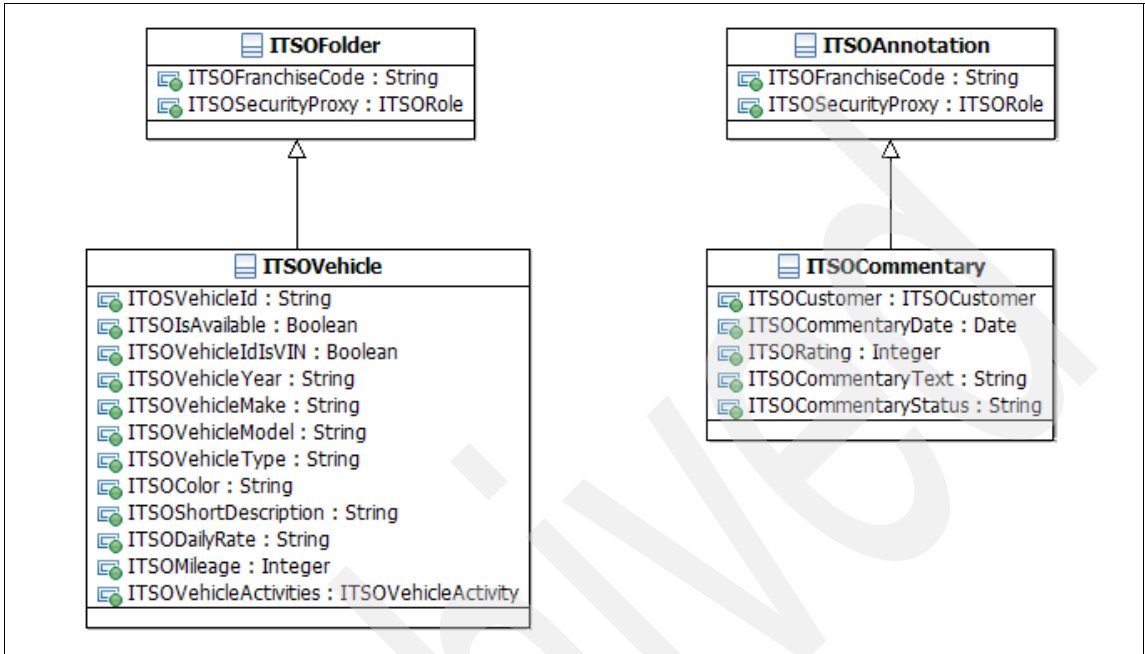
Figure 7-21   Models for ITSOFolder, ITSOAnnotation, and subclasses

Figure 7-22 shows the model for ITSOCustomObject and its subclasses.
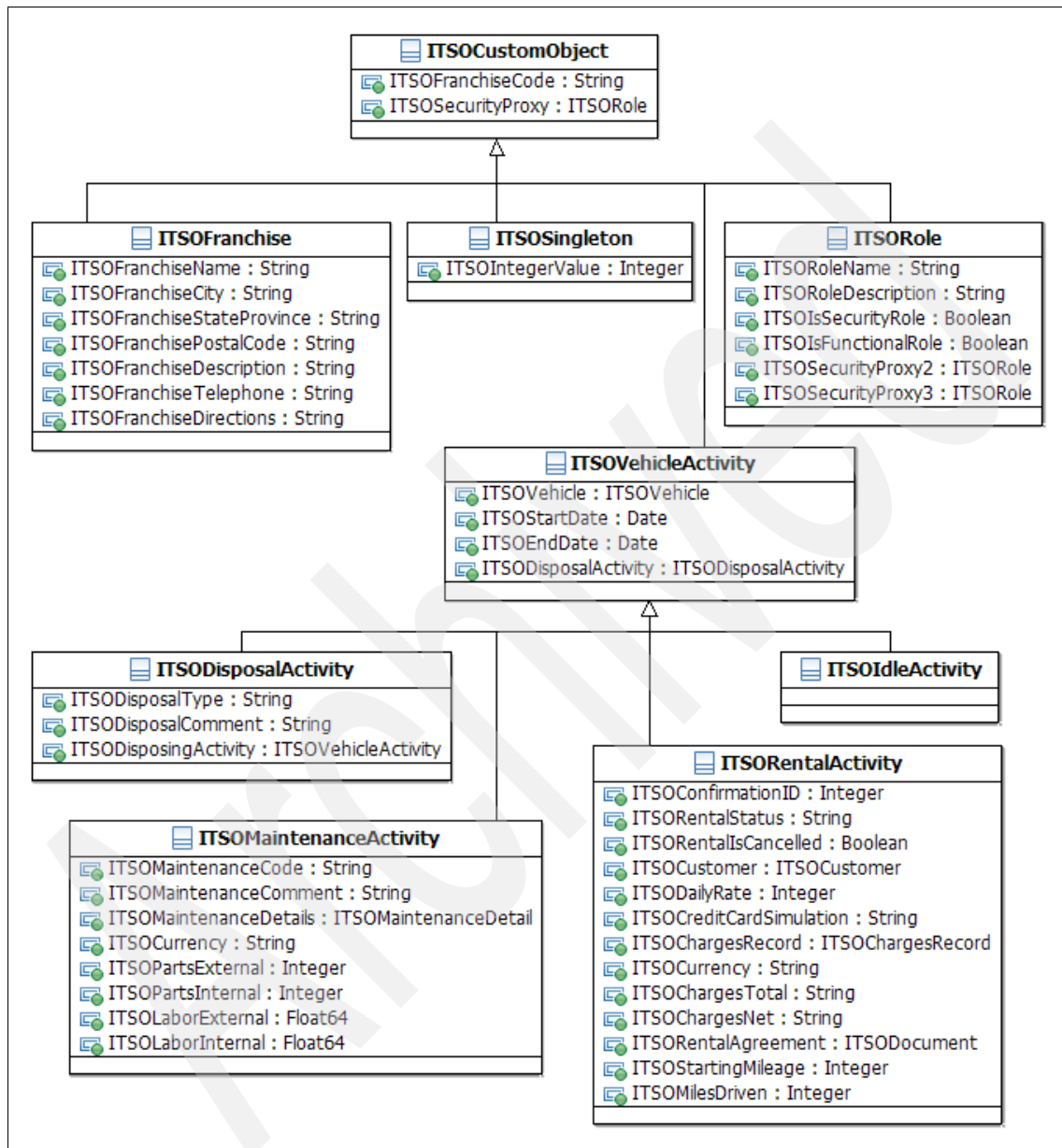
Figure 7-22   Model for ITSOCustomObject and subclasses

7.4.1 Base classes

As an organizational technique, we first define a series of base classes. There is a custom base class for each of the types of objects used elsewhere in the data model. Aside from showing our intent explicitly in the object hierarchy, these base classes also serve as a point where we can apply any common custom properties and perform broad queries. If we did that instead with the system base classes, we might affect or include classes that are not part of our application.

The custom base classes include:

► ITSODocument is a subclass of system class Document.

► ITSOFolder is a subclass of system class Folder.

► ITSOCustomObject is a subclass of system class CustomObject.

► ITSOAnnotation is a subclass of system class Annotation.

Properties common to all custom base classes are listed in Table 7-1 on page 244.

> **Note:** In the data model tables, the following notational conventions apply:
>
> ► String(n) means a string-valued property with a maximum length of n. Binary(n) means a binary value of at most n bytes.
>
> ► Long String means a string-valued property with its UsesLongColumn metaproperty set to true.
>
> ► OVP(xyz) means an object-valued property with a required type of xyz.
>
> ► MVOVP(xyz) means a multi-valued object-valued property with a required type of xyz.
>
> ► Date-valued properties are always date and time combinations with a resolution accurate to whatever the underlying database supports.

Table 7-1   ITSODocument, ITSOFolder, ITSOCustomObject, ITSOAnnotation

Class: ITSODocument, ITSOFolder, ITSOCustomObject, ITSOAnnotation		
Property	**Type**	**Comment**
ITSOFranchiseCode	String(10)	Value required. Franchise identity code is always normalized to lowercase. A pseudo-code of `itsoautohq` is for the company headquarters.
ITSOSecurityProxy	OVP (ITSORole)	Most objects have a single pointer to an ITSORole object for security inheritance purposes. Even objects which are publicly visible have a value here for simplified administration.

Folders must exist at a single place in a strictly hierarchical namespace. All folders of any kind for these applications are under the top-level folder `/ITSOAutoRental/`. A franchise has a franchise identity code, and folders specific to a particular franchise are placed under a subfolder named after the franchise code. For example, a franchise with identity code `ought2` would have all its folders under `/ITSOAutoRental/ought2/`.

7.4.2 ITSOVechicle

This class acts as a collector of information about many things that are related to a specific physical vehicle. Examples of these assorted things include scanned images of legal documents related to ownership and registration, photographs, and accident reports.

Because it acts as a collecting point for many different types of things, it is modelled as a folder object. Various documents of unpredictable types may be filed into a folder for a specific vehicle. A large franchise may have hundreds of vehicles in inventory, so browsing folders as a way of searching for vehicles is not very practical. We instead expect the applications to identify a particular vehicle through a search and then use the corresponding folder as a paradigm for navigating to related documents.

Each vehicle has a unique identity code. That identity code is used as the name of the ITSOVehicle folder. To facilitate occasional browsing navigation, the ITSOVehicle folders are filed into subfolders that are named for the character in the ninth position of the identity code (all of which are filed under a vehicles subfolder). If no ninth character exists, the last non-space character is used. If that character is a letter, it is converted to lowercase. For example, a vehicle with

an identity code of `JT4TN12D8V0030519` could be represented by a folder with the following path:

`/ITSOAutoRental/ought2/vehicles/8/JT4TN12D8V0030519`

Allowing at least 17 characters for the identity code allows for the direct use of the industry standard Vehicle Identification Number (VIN). Position 9 of the VIN is a check digit and thus well-distributed; it has eleven possible values (digits 0 - 9, letter X). Because of the unique inventories of the company franchisees, they might have older vehicles without a VIN, or with a VIN in a different format. Much of the provenance information for a vehicle (for example, year, make, and model of manufacture) can be extracted by decoding a standard VIN, but it must be explicitly provided for other types of identity codes. For purposes of the sample application, we assume that all vehicles have a VIN and the `ITSOVehicleId` property always matches the `ITSOVehicleVIN` property. In other words, `ITSOVehicleIdIsVIN` is always true for the sample application.

By convention, there are documents filed into an ITSOVehicle folder with specific containment names to indicate specific purposes. See, for example, 7.4.3, "ITSOPhotoGallery" on page 247. Note the following information:

► Not shown elsewhere in this book is an ITSODocument with a containment name of `obd`. Its content is an XML document holding OBD II diagnostics data that is collected from the vehicle. Each time OBD II data is collected, a new version of the document is created so that old data sets are preserved.

► Not shown elsewhere in this book is an ITSODocument with a containment name of `description`. It contains the full description of the vehicle. The actual descriptive narrative is contained in the first content element, and it must be in a format understood by the rendering application. For our samples, we use text or HTML descriptions.

Rental charges for a vehicle are abstracted to a separate ITSOChargesSchedule object, which can be shared by many vehicles. For our sample application, we do not use or further describe the ITSOChargesSchedule (vehicle rental charges are supplied by hard-coded application logic). After it is created, ITSOChargesSchedule objects are not modified because they are still pointed to by ITSORentalActivity objects from the past. A DeletionAction metaproperty value of `DeletionAction.PREVENT` on the MVOVP that is pointing from the ITSOChargesSchedule to ITSOVehicle objects can be used to ensure referential integrity for attempted deletions.

Data

Table 7-2 shows the properties of ITSOVehicle.

Table 7-2   ITSOVehicle

Class: ITSOVehicle (subclass of ITSOFolder)		
Property	**Type**	**Comment**
ITSOVehicleId	String(25)	Value required. Standard VIN is 17 characters.
ITSOIsAvailable	Boolean	As a performance measure, this field can be manually set to false when a vehicle is in maintenance for an unusually long time. It is also set to false when the vehicle is disposed of.
ITSOVehicleIdIsVIN	Boolean	Value required. VIN decoding can be used if this value is true.
ITSOVehicleYear	String(4)	Manufacturer's model year
ITSOVehicleMake	String(10)	Manufacturer name
ITSOVehicleModel	String(10)	Manufacturer's model name
ITSOVehicleType	String(1)	Same codes as used in a VIN
ITSOColor	String(15)	Intended for searching. More detailed and nuanced color descriptions can be provided in the vehicle description.
ITSOShortDescription	Long String(100)	Value required. Short description that is suitable for display in search summaries, and other things. This property avoids an immediate fetch of the full description document that is contained in the folder.
ITSODailyRate	String(10)	Daily rate
ITSOMileage	Integer	Total mileage of car
ITSOVehicleActivities	MVOVP (ITSOVehicleActivity)	Reflective of ITSOVehicleActivity property ITSOVehicle.

Security

ITSOVehicle security is defined as follows:

- ► ITSOVehicle can be read by customers and all employees.
- ► ITSOVehicle can be created, updated, and deleted by franchise supervisory employees.
- ► The OBD II document can be created, read, and updated by any franchise employee. An update for the OBD II document is a checkout and checkin cycle.
- ► The OBD II document versions can be deleted by franchise supervisory employees.
- ► ITSOChargesSchedule can be read by customers and all employees.
- ► ITSOChargesSchedule can be created, updated, and deleted by franchise supervisory employees.

7.4.3 ITSOPhotoGallery

A document filed in an ITSOVehicle folder with a containment name of `photogallery` contains photographs of the vehicle. All photographs are in either JPEG, GIF, or PNG format. The primary photograph is in the first content element. Additional photographs are in other content elements.

Data

The ITSOPhotoGallery is a subclass of ITSODocument and has no specific subclass custom properties.

Security

An ITSOPhotoGallery completely inherits its security from the containing ITSOVehicle, and is implemented through the normal dynamic security inheritance of the SecurityFolder property.

7.4.4 ITSOThumbnail

By convention, each content element in an ITSOPhotoGallery document has an ITSOThumbnail annotation containing a smaller version of the image. The thumbnail image is contained in the first and only content element of the annotation and is also in JPEG, GIF, or PNG format.

Data

The ITSOThumbnail is a subclass of ITSOAnnotation and has no specific subclass custom properties.

Security

An ITSOThumbnail completely inherits its security from the annotated ITSOPhotoGallery, and is implemented through the normal dynamic security inheritance of the AnnotatedObject property.

7.4.5 ITSOVehicleActivity

A vehicle is always in some kind of activity state; the default state is idle.

All activity records indicate a timespan, expressed as start and end date properties. In some cases, those dates are adjusted over the lifetime of the activity record. For example, the start and end dates are estimates for a reservation and are adjusted to actual rental period dates when the vehicle is picked up and returned. We can look both backward and forward in time to see when a vehicle was in a state or when it is expected to be in a state. For example, when a vehicle is reserved for rental by a customer, an activity record with a future timespan is created.

Any period of time that is not covered by an activity record is considered idle time, and the vehicle is available for rental for that period (although business rules enforce a buffer period around some types of activities to recognize that the vehicle might not become available exactly when expected).

The major states for vehicles are reflected in subclasses of ITSOVehicleActivity (see 7.4.7, "ITSORentalActivity" on page 250; 7.4.8, "ITSOMaintenanceActivity" on page 251; and 7.4.9, "ITSODisposalActivity" on page 253). Several other minor activities exist, but we do not show them here (for example, when a vehicle has been returned and is being washed and prepared for the next rental).

Data

Table 7-3 shows the properties of ITSOVehicleActivity.

Table 7-3   ITSOVehicleActivity

Class: ITSOVehicleActivity (subclass of ITSOCustomObject)		
Property	**Type**	**Comment**
ITSOVehicle	OVP (ITSOVehicle)	-
ITSOStartDate	Date	-
ITSOEndDate	Date	-
ITSODisposalActivity	MVOVP (ITSODisposalActivity)	If this activity led directly to a disposal, then this property points to the disposal activity object. Although for referential integrity purposes this is a reflective property corresponding to the ITSODispoalActivity property ITSODisposingActivity, we expect it to have either no value or a single value.

Security

ITSOVehicleActivity's security is defined as follows:

► All ITSOVehicleActivity records can be read by customers and all employees.

► Customers must read all records for the purpose of doing reservation activity; however, application logic restricts display to only ITSORentalActivity records which are affiliated with that customer through the ITSOCustomer property.

► Further access is described for each ITSOVehicleActivity subclass.

7.4.6 ITSOSingleton

ITSOSingleton is used to maintain the last confirmation ID that is used for a car rental. See Table 7-4.

Table 7-4   ITSOSingleton

Class: ITSOSingleton (subclass of ITSOCustomObject)		
Property	**Type**	**Comment**
ITSOIntegerValue	Integer	Used to generate the unique ITSOConfirmationID for rental reservations.

7.4.7 ITSORentalActivity

This type of activity record is used both for reservations, in-progress rentals, and completed rentals.

Payment details, including credit card number and itemized charges, are recorded in an ITSOChargesRecord. For simplicity in our sample application, we do not use or further describe the ITSOChargesRecord. Instead, we directly record an artificial credit card-like number in the ITSOCreditCardSimulation property.

Data

Table 7-5 shows the properties of ITSORentalActivity.

Table 7-5   ITSORentalActivity

Class: ITSORentalActivity (subclass of ITSOVehicleActivity)		
Property	**Type**	**Comment**
ITSOConfirmationID	Integer	Fetch the ITSOSingleton object and get the ITSOIntegerValue and increment it by 1 for generating ITSOConfirmationID. Save the ITSOSingleton object with incremented value.
ITSORentalStatus	String(10)	Value required. It can be one of the following values: `reserved`, `active`, `historical`
ITSORentalIsCancelled	Boolean	Is used for historical and auditing purposes. Cancelled reservations do not participate in vehicle activity.
ITSOCustomer	OVP (ITSOCustomer)	-
ITSODailyRate	Integer	Daily rate
ITSOCreditCardSimulation	String(12)	Artificial credit card-like payment indicator
ITSOChargesRecord	OVP (ITSOChargesRecord)	Pointer to a separate payment record containing credit card and other details

Class: ITSORentalActivity (subclass of ITSOVehicleActivity)		
Property	**Type**	**Comment**
ITSOCurrency	String(3)	The currency type for money amounts. Values are usd, cad, eu, and so on.
ITSOChargesTotal	String	Total estimated or actual rental charges rolled up from the ITSOChargesRecord
ITSOChargesNet	String	Net estimated or actual rental charges (after direct costs like taxes and fees are deducted) that are rolled up from the ITSOChargesRecord
ITSORentalAgreement	OVP (ITSODocument)	Is used to demonstrate versioning
ITSOStartingMileage	Integer	Mileage at the beginning of rental
ITSOMilesDriven	Integer	Miles driven during rental period.

Security

ITSORentalActivity security is defined as follows:

► ITSORentalActivity objects may be created, read, and updated by any customer, but application logic restricts read access to objects associated with that customer. All create and update activities are mediated by an application.

► ITSORentalActivity objects may be created, read, updated, and deleted by any employee of the franchise to which the vehicle belongs.

7.4.8 ITSOMaintenanceActivity

This type of activity records a single maintenance event, generally meaning a single contiguous trip to the maintenance area. There are additional ITSOMaintenanceDetail objects showing time spent waiting, sent to outside service contractors, and so on. The details are aggregated to a single ITSOMaintenanceActivity record. The sample application does not show the ITSOMaintenanceDetail objects.

Data

Table 7-6 shows the properties of ITSOMaintenanceActivity.

Table 7-6   ITSOMaintenanceActivity

Class: ITSOMaintenanceActivity (subclass of ITSOVehicleActivity)		
Property	**Type**	**Comment**
ITSOMaintenanceCode	String (10)	Required value. One of a set of fixed values including `oilchange`, `tirerepair`, `bodydamage`, and so on. Unusual maintenance can be described in: `ITSOMaintenanceComment`
ITSOMaintenanceComment	Long String(5000)	Manually entered text describing the nature of the maintenance
ITSOMaintenanceDetails	MVOVP (ITSOMaintenanceDetail)	Reflective from ITSOMaintenanceDetail property ITSOMaintenanceActivity
ITSOCurrency	String(3)	Currency type for money amounts. Values are `usd`, `cad`, `eu`, and so on.
ITSOPartsExternal	Integer	Cost of parts charged by an outside service contractor. Money amounts are multiplied by 100 to preserve accuracy.
ITSOPartsInternal	Integer	Cost of parts from in-house sources. Money amounts are multiplied by 100 to preserve accuracy.
ITSOLaborExternal	Float	Labor hours from outside service contractors
ITSOLaborInternal	Float	Labor hours from in-house technicians

Security

ITSOMaintenanceActivity security is defined as follows:

► ITSOMaintenanceActivity objects may be read by any employee of the franchise to which the vehicle belongs.

► ITSOMaintenanceDetail objects may be read by any maintenance employee or supervisory employee of the franchise to which the vehicle belongs.

► ITSOMaintenanceActivity objects may be created, updated, or deleted by any supervisory employee of the franchise to which the vehicle belongs. There is typically at least one supervisory employee in the maintenance department.

► ITSOMaintenanceDetail objects may be created, updated, or deleted by any maintenance employee of the franchise to which the vehicle belongs.

7.4.9 ITSODisposalActivity

When a vehicle is sold or removed permanently from inventory, the vehicle records are kept online for a period of time. To prevent an unavailable vehicle from being scheduled for any real activities, it is given an ITSODisposalActivity record with an end date in the distant future. The sample application does not show the process of disposing of vehicles.

Data

Table 7-7 shows the properties of ITSODisposalActivity.

Table 7-7   ITSODisposalActivity

Class: ITSODisposalActivity (subclass of ITSOVehicleActivity)		
Property	**Type**	**Comment**
ITSODisposalType	String(10)	Value required. One of the values `sold`, `scrapped`, `stolen`.
ITSODisposalComment	Long String(5000)	Manually entered text describing the nature of the disposal.
ITSODisposingActivity	OVP (ITSOVehicleActivity)	An optional pointer to a rental or maintenance activity that led to the disposal.

Security

ITSODisposalActivity is defined as follows:

- ► ITSODisposalActivity objects may be read by any employee of the franchise to which the vehicle belongs.
- ► ITSODisposalActivity objects may be created/updated/deleted by any supervisory employee of the franchise to which the vehicle belongs.

7.4.10 ITSOIdleActivity

When a vehicle has no explicit activity of some other type, it is said to be idle. It is possible to deduce the idle periods by noting the gaps between other types of activities. However, it is difficult to perform a query for that gap, so the repository uses explicit ITSOIdleActivity objects to keep track of the gaps. To find vehicles available for rental, the application queries for ITSOIdleActivity records for the applicable time frame.

Data

The ITSOIdleActivity is a subclass of ITSOVehicleActivity and has no specific subclass custom properties.

Security

ITSOIdleActivity security is defined as follows:

- ► ITSOIdleActivity objects may be read by any customer or employee.
- ► Because ITSOIdleActivity objects are managed completely by event action handlers (which run in system context), they are read-only to all customers and employees.

7.4.11 ITSOCustomer

Employees have entries in the enterprise directory, but customers do not. Customer information is kept in the repository. Customer information can be coarsely divided into two types: customer factual information (for example, address) and customer preference information. Information that is not typically needed for routine customer searches is kept as XML as the document content. A new document version is created when the information is revised so that prior data sets can be kept.

Data

Table 7-8 shows the properties of ITSOCustomer.

Table 7-8   ITSOCustomer

Class: ITSOCustomer (subclass of ITSODocument)		
Property	**Type**	**Comment**
ITSOCustomerUserid	String (20)	Used by customer for login identity
ITSOCustomerPassword	Binary (20)	SHA-1 160-bit hashed password value. Used for authentication.
ITSORentalActivities	MVOVP (ITSORentalActivity)	Reflective from ITSORentalActivity property ITSOCustomer
ITSOCommentaries	MVOVP (ITSOCommentary)	Reflective from ITSOCommentary property ITSOCustomer

Security

ITSOCustomer security is defined as follows:

► ITSOCustomer objects may be created, read, updated, or deleted by customers or any employee at headquarters or at a franchise location.

► Application logic restricts a customer to viewing or acting on the single applicable ITSOCustomer object.

7.4.12 ITSOCommentary

As an aspect of social networking, the company allows customers to make comments and provide ratings of specific vehicles. These comments are visible to customers as they browse the vehicle inventory. A comment is always tied to a particular ITSOCustomer, but it is not a requirement that a customer has rented that specific vehicle in the past. For example, customers might like to make comments that are related only to the type or model of vehicle. ITSOCommentary is implemented as a custom annotation class annotating the ITSOVehicle objects.

Although the company does not censor negative comments, all comments added go through an editorial moderation process to screen for obscenities, spam, legal risks, and so on. Moderators may also make minor formatting, grammar, and spelling corrections.

Data

Table 7-9 shows the properties of ITSOCommentary.

Table 7-9   ITSOCommentary

Class: ITSOCommentary (subclass of ITSOAnnotation)		
Property	**Type**	**Comment**
ITSOCustomer	OVP (ITSOCustomer)	-
ITSOCommentaryDate	Date	The moderation process is prone to disrupt the DateCreated and DateLastModified properties, so this explicit date reflects when the commentary was originally created by the customer.
ITSORating	Integer	Values are restricted to 0 - 10, where 0 means not rated. A conventional five-star scale with the possibility of half-stars can be created by dividing this value in half.
ITSOCommentaryText	Long String(5000)	Value is assumed to be HTML text.
ITSOCommentaryStatus	String(5)	Moderation life cycle status of a particular comment. Values are raw (as received from the author), edit (being edited by a moderator), and ready (available for viewing by customers)

Security

ITSOCommentary security is defined as follows:

► When a commentary is initially created, it has an explicit access control entry of AccessType.DENY for all access rights for the customers. As part of the moderator approval process, that access control entry is removed. Thus, normal access control mechanisms will render unmoderated commentaries invisible to customers.

► ITSOCommentary objects are visible to customers (after the moderation process) and any headquarters or franchise employee.

► ITSOCommentary objects may be created, updated, or deleted by a customer or any non-maintenance employee of the franchise to which the vehicle belongs. Application logic restricts customers to updating or deleting comments that they created.

7.4.13 ITSOFranchise

ITSOFranchise objects provide details for Fictional Auto Rental Company A franchise locations. The inherited ITSOFranchiseCode is used by many other classes to narrow those objects to a particular franchise location.

> **Note:** The sample application assumes one location per city. In reality, one city could have multiple locations.

Data

Table 7-10 shows the properties of ITSOFranchise.

Table 7-10   ITSOFranchise

Class: ITSOFranchise (subclass of ITSOCustomObject)	
Property	**Type**
ITSOFranchiseName	String(20)
ITSOFranchiseCity	String(20)
ITSOFranchiseStateProvince	String(3)
ITSOFranchisePostalCode	String(10)
ITSOFranchiseDescription	String(30)
ITSOFranchiseTelephone	String(12)
ITSOFranchiseDirections	String(30)

7.4.14 ITSORole

ITSORole objects identify functional types of users. The sample application considers the following distinct types of users, each represented by a distinct ITSORole object with the indicated ITSORoleName:

- ► Customer (`customer`)
- ► Headquarters employee (`hq`)
- ► Franchise supervisory employee (`supervisor`)
- ► Franchise front-office (customer-facing) employee (`frontoffice`)
- ► Franchise maintenance employee (`maintenance`)

An important role is customer, which is the role for the single account that is used by the back-end systems for all customer interactions. The corresponding ITSORole object has an ITSOFranchiseCode of `itsoautohq`, as does the ITSORole object for headquarters employees.

Objects that represent roles serve two distinct purposes in the sample application. It is not a requirement that the same set of ITSORole objects be used for both purposes. Because a large amount of overlap exists, we do not specialize it further into subclasses. Instead, we distinguish the type of individual ITSORole object through custom properties and business rules, which use those properties.

A franchise supervisory employee has access to everything to which either a franchise front-office employee or a franchise maintenance employee has access. We model this with ITSORole objects by having both the `frontoffice` and `maintenance` ITSORole objects for a given franchise inheriting security (through the ITSOSecurityProxy property) from the `supervisor` ITSORole object for that same franchise.

Roles for application customization

Roles are used to control the visibility of various features in the user interfaces of application components. This is not a form of security access control; it is merely user experience policy guidance to the application logic. Someone could write a different application and completely bypass this role-based activity.

The specific mechanism is that if a user can see a particular ITSORole object (that is, AccessRight.READ), then that user is allowed to see the related functions in the user interface. All maintenance workers in a particular franchise would be listed in the ACL for the ITSORole object with that ITSOFranchiseCode and an ITOSRoleName of `maintenance`.

Thus, the ACL on the ITSORole object is used to indirectly control the application. For example, screens that can logically only be used by maintenance employees can be hidden from users who do not have access to at least one

`maintenance` ITSORole object. To make its decision, the application issues a query of the form:

```
SELECT TOP 1 Id from ITSORole WHERE ITSORoleName = 'maintenance'
```

If at least one result is returned from that query, the application knows that the user has access to one or more `maintenance` ITSORole objects. In this particular example, where we are controlling the display of application features visible only to maintenance employees, matching on the ITSOFranchiseCode is not necessary. Security constraints on the maintenance-related objects ensure that users see only objects that are associated with a particular franchise.

Roles for dynamic security inheritance

Roles are used to control access to other types of objects. The specific mechanism that is used is dynamic security inheritance. Dynamic security inheritance is a feature whereby an object can inherit some or all of its security access control from one or more proxy objects. In our data model, the ITSORole objects are the security proxy objects. The use of the ITSORole objects for this purpose is described in more detail in the 6.2, "Application space, role, and workbasket" on page 187.

Data

Table 7-11 on page 260 shows the properties defined for ITSORole.

Table 7-11   ITSORole

Class: ITSORole (subclass of ITSOCustomObject)		
Property	**Type**	**Comment**
ITSORoleName	String(20)	Short but meaningful name for this ITSORole object, as described in the text. Uniqueness is enforced by convention when combined with the ITSOFranchiseCode for this object.
ITSORoleDescription	Long String(250)	-
ITSOIsSecurityRole	Boolean	Value is true if this ITSORole is used for security access control.
ITSOIsFunctionalRole	Boolean	Value is true if this ITSORole is used for controlling user interface elements.
ITSOSecurityProxy2	OVP (ITSORole)	Additional security proxy property. Used when combining two roles together.
ITSOSecurityProxy3	OVP (ITSORole)	See ITSOSecurityProxy2 property.

Security

The security of the ITSORole objects (of both types) is dictated by how each individual ITSORole object is used. Unlike most of the other objects in our data model, no users (other than the system administrators) inherently have access to the ITSORole objects.

7.5 Security model

For the purposes of our sample application, we divide users into the distinct populations given in 7.4.14, "ITSORole" on page 258. In a real application, much greater granularity would likely exist, and we would also have to account for users being members of more than one functional group. In addition to user populations with corresponding ITSORole objects, there are also headquarters system administrators who fill various functional roles, which we do not address in our sample application.

Note: For simplicity in the sample application, we have made the customer pseudo-identity slightly more powerful than we would like, and we enforce additional controls at the application level. For example, to be able to make reservations, the customer must have security access to all ITSOIdleActivity records, but our policy is that they can see only their own ITSORentalActivity objects and no other ITSOActivity objects. Furthermore, in some parts of the sample application, we have to supply explicit credentials to authenticate as the customer pseudo-identity.

Both of these situations are undesirable as security best practices. A more secure (but, unfortunately, too complex to show in our sample application) implementation would pass all customer access through an EJB method with a configured RunAs role. That is, each EJB method is configured (in the deployment descriptor) to run as the customer pseudo-identity when further J2EE calls are made. This has the advantage of consolidating the logic which enforces additional restrictions on access to objects. We could probably even configure the customer pseudo-identity in the enterprise directory to prohibit logins.

EJB RunAs roles is a standard J2EE feature. Actually configuring it has aspects that are specific to application server and is beyond the scope of this book. Nonetheless, it is an important security implementation pattern, and you should consult your application server documentation to learn more about it.

Customers do not have records in the the company enterprise directory, whereas all headquarters and franchise employees do. A single customer pseudo-identity is present in the enterprise directory and is used for all access across all franchises. Customer account information is stored as objects in the Content Manager repository.

The enterprise directory uses a group structure to organize employees, but it is static and relatively flat. There is a top-level directory group for headquarters and a top-level directory group for each franchise. Within those top-level groups, the enterprise directory does not use further subgroups.

Access control security for objects is implemented according to these principals:

► Customers can see only vehicle information that is intended to be part of Web pages. They cannot see other vehicle information or other business objects.

► Customers can see all vehicles from all franchises. The application logic allows them to limit or expand their vehicle searches.

► An employee in one franchise location cannot view the business objects of another franchise location.

- Different groups of employees may have different types of access to different business objects for the franchise.

- Appropriate employees of the company headquarters have access to selected business objects for any franchise.

- Back-end system administrators have access to all objects in the system. All system administrators are part of the company headquarters.

Specific mechanisms that are used for security access include direct access control entries (Aces) on objects, standard security inheritance mechanisms (for example, object Aces inherited by an annotation object), and a collection of ITSORole objects acting as security proxies through dynamic security inheritance.

The dynamic security inheritance embodied in the ITSORole objects is transitive, so an ITSORole object can itself inherit ACEs from another ITSORole object. Cascading combinations of inheritance can be used to control access to objects. The advantages to this are:

- ITSORole objects can stand in for a more granular group structure than is available from the enterprise directory.

- When an employee moves from one role to another (for example, a franchise line employee becoming a franchise supervisory employee), the security for any number of objects may be instantly updated by changing the ACLs on applicable ITSORole objects.

7.6 Workflows

In this section, we describe the workflows implemented for the the company management solution. For a complete business solution, additional elements, such as new workflows, can be identified and implemented across the applications that comprise this solution; however, for the sample application, we provide the following workflow definitions:

- Vehicle reservation workflow
- Vehicle maintenance workflow

These workflow definitions use a specific isolated region configuration, including a custom roster, event log, and queues. A custom component queue (ITSO_Operations) is also included as part of this configuration to perform certain custom calls within the workflows.

7.6.1 Isolated region configuration

An isolated region is a logical subdivision of the workflow database that contains the queues for the work items, event logs, rosters, and other configuration information. It also contains all transferred workflows and running work items.

Table 7-12 shows the isolated region configuration implemented for the company management application.

Table 7-12   Isolated region configuration used by the sample application

Isolated region element	Name	Fields exposed
Rosters	ITSORoster	-
Event logs	ITSOEventLog	-
Work queues	ITSO_Reservation	ITSOConfirmationId(30) ITSOVehicleStatus(10)
	ITSO_Maintenance	ITSOVehicleId(30)
	ITSO_Malfunction	ITSOConfirmationId(30) ITSOVehicleStatus(10) ITSOVehicleId(30)
Component queues	CE_Operations	-
	ITSO_Operations	-

7.6.2 Component queues

Two Java adaptors are used by the sample application workflows. The following component queues are implemented to service these Java adaptors:

▶ CE_Operations

Built-in component used to integrate the CE and PE facilitating the management of documents and other objects in an object store from within a workflow. This component is provided, by default. Most of the component operations used by the sample application are included in CE_Operations.

Note: The `sendMail` operation from CE_Operations is commonly used within the sample application workflows to send plain text e-mail notifications. In order for these notifications to work, e-mail settings must be properly configured using the Process Task Manager.

- ITSO_Operations

  Implemented to service a custom Java component to perform certain business operations used by the sample application workflows. This component is also provided as a sample to show how to implement a custom Java component by using Component Integrator.

  Table 7-13 describes the operations included in this custom Java adaptor.

Table 7-13   ITSO_Operations component queue operations

Operation name	Description
verifyCustomerCredit	Performs customer credit card validation by calling a third-party system. The operation simulates the call to the external system, always returning a true value.
getCustomerDetailsFromOS	Retrieves and parses the ITSOCustomer document content to extract certain metadata values and assign them to the workflow data fields. This operation demonstrates how to perform a custom call to CE.
sendVehicleToMaintenance	Uses certain business rules related to a vehicle maintenance activity and launches the vehicle maintenance workflow, passing all required objects. This operation demonstrates how to perform a custom call to PE.

7.6.3 Vehicle reservation workflow

This workflow interacts with both the Reservation Web application and the Kiosk application. It is used to manage the vehicle reservation life cycle, from the moment a vehicle reservation is requested by a customer until the moment the vehicle is returned to the rental agency.

Figure 7-23 shows the main diagram that is used by this workflow definition.
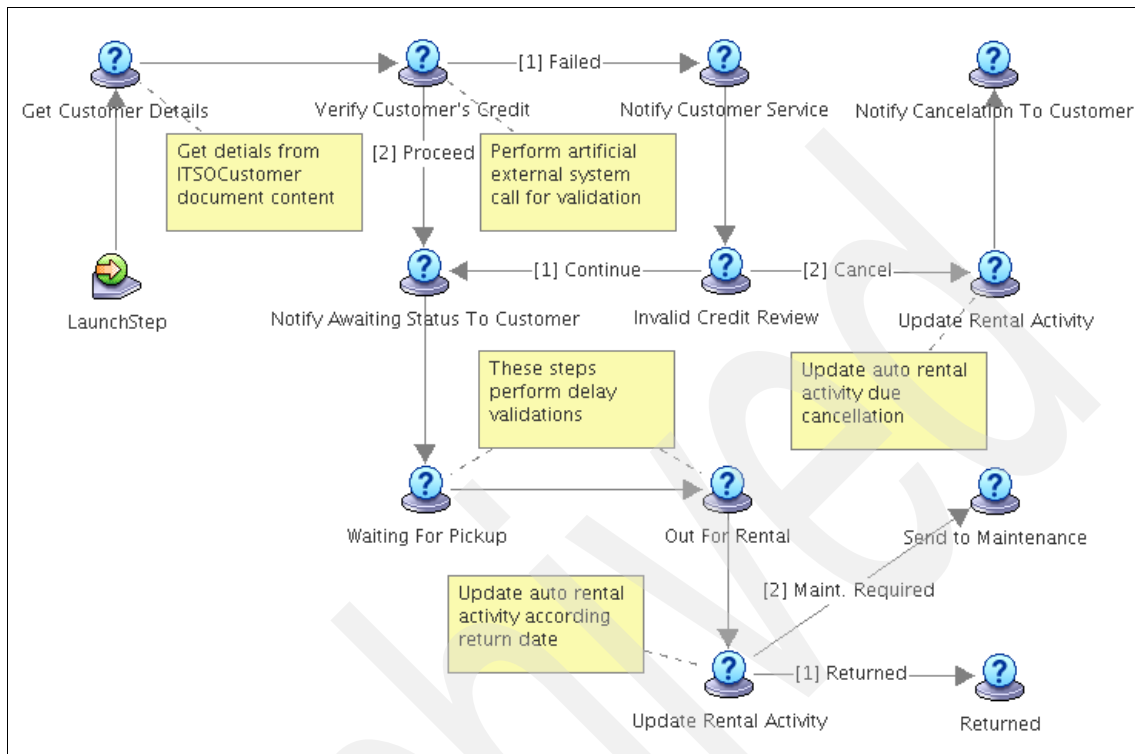
Figure 7-23   ITSO vehicle reservation main workflow diagram

Although workflows can be launched by using workflow subscription events, the ITSO vehicle reservation workflow is launched programmatically by the Reservation Web application in order to pass the required objects as attachment references and avoid unnecessary calls within the workflow. The Kiosk application interacts with this workflow when a customer picks up and returns a vehicle.

The vehicle reservation workflow definition is comprised of system, component, and work steps. All system and component steps are executed without human intervention to either perform a task within the workflow or call to an external system.

Table 7-14 describes the work steps defined in this workflow.

Table 7-14   ITSO vehicle reservation work steps

Work step	Work queue	Description
Invalid credit review	ITSO_Reservation	When the customer's credit validation is unsuccessful, the vehicle item is routed to this step for review. At this stage, the reservation can be either cancelled or approved.
Waiting for pickup	ITSO_Reservation	The vehicle reservation is approved and waiting to be picked up and processed from the kiosk Web application.
Out for rental	ITSO_Reservation	The vehicle is rented and waiting to be returned by the customer and processed from the kiosk Web application.
Review	ITSO_Malfunction	The malfunction step is to handle unexpected workflow exceptions.

7.6.4 Vehicle maintenance workflow

This workflow is used to manage the vehicle maintenance life cycle from the Fleet Status Manager Web application.

Figure 7-24 shows the main diagram that is used by this workflow definition.
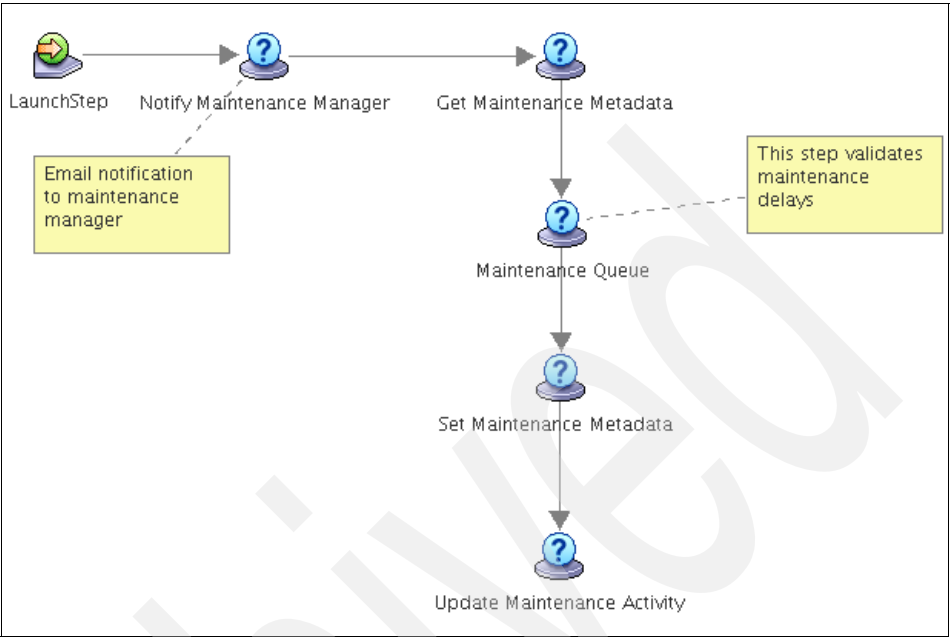
Figure 7-24   ITSO vehicle maintenance main workflow diagram

The vehicle maintenance workflow can be launched either from the vehicle reservation workflow when the vehicle is returned and must be sent to maintenance, or from the Fleet Status Manager Web application when a vehicle is manually sent to maintenance. The Fleet Status Manager Web application is also used to process all vehicles sent to maintenance.

Table 7-15 describes the work step defined in the vehicle maintenance workflow definition.

Table 7-15   ITSO vehicle maintenance work steps

Work step	Work queue	Description
Maintenance queue	ITSO_Maintenance	The vehicle is sent to maintenance and waits in the queue until it gets assigned and processed.
Review	ITSO_Malfunction	Malfunction step to handle unexpected workflow exceptions.

7.7 Internal architecture of sample applications

In this section, we describe the technical architecture of various application components. By technical architecture, we mean a high-level view of the implementation technologies used and how the application components interact.

7.7.1 Architecture: Reservation Web application

For a description of the user view of this application, see 7.3.1, "User view: Reservation Web application" on page 226.

For a description of the deployment instructions for this application, see 7.8.4, "Deployment: Reservation Web application" on page 292.

The Reservation Web application is a J2EE application that is written by using Java Server Faces (JSF). It communicates with the CE (using the CE Java API) and the PE (using the PE Java API). The overall program flow is illustrated in Figure 7-25.
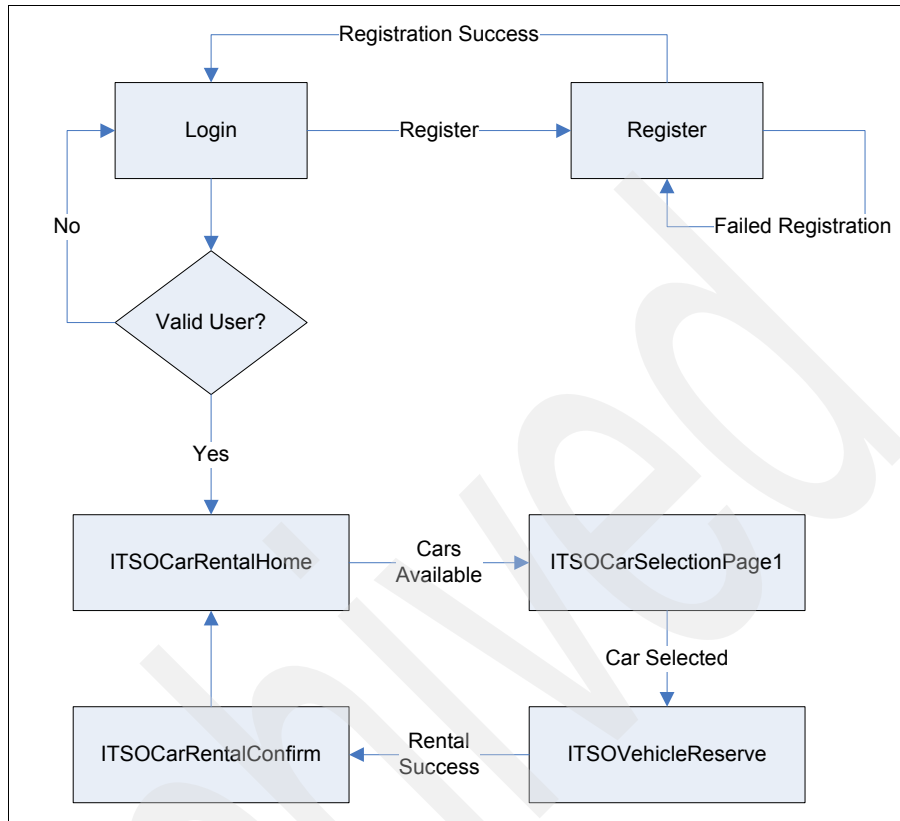
Figure 7-25   Program flow for Reservation Web application

This application consists of the following Java classes:

- ► `com.itso.MetadataConstants`: The `MetadataConstants` class contains CE and PE property and class names.

- ► `itso.autorental.ITSOUtil`: This class contains various convenience methods. The intent of these methods is mainly to isolate various common interactions with the CE and PE so that it is not spread throughout the other classes.

- ► `itso.autorental.process.ITSOAutoRentalWorkflow`: This class contains methods related to interaction with the PE API.

- ► `com.itso.bean` package: This package consists of JavaBean classes for data objects.

- ▶ `itso.autorental.RentalActivityHandler`: Convenience methods including creating and deleting various vehicle activities, as follows:
  - `createMaintenanceActivity()`: This method takes a vehicle ID and franchise code and creates a maintenance activity.
  - `createRentalActivity()`: This method creates a rental activity for a confirmed reservation.
  - `deleteVehicleActivity()`: This method removes a vehicle activity (for example, maintenance activity) given the activity ID.
- ▶ The `pagecode` package contains the JSF backing classes for each JSP.

7.7.2 Architecture: Kiosk application

For a description of the user view of this application, see 7.3.2, "User view: Kiosk application" on page 230.

For a description of the deployment instructions for this application, see 7.8.5, "Deployment: Kiosk application" on page 295.

The Kiosk application is a Microsoft .NET 2.0 application that is written in C# and communicates directly with the CE (using the CE .NET API) and the PE (using the Process Engine Web Services API). See Figure 7-26 for the application flow diagram.
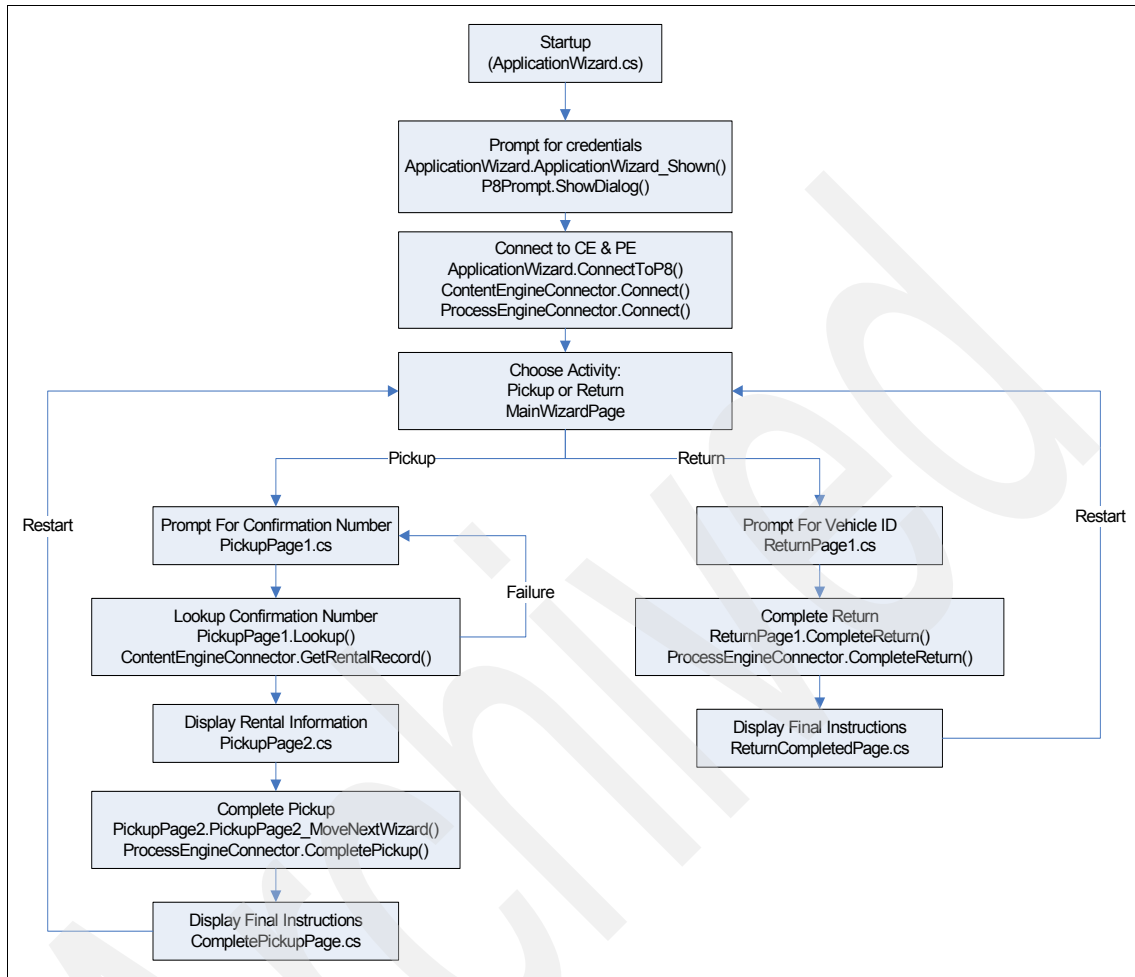
Figure 7-26   Program flow for Kiosk application

The application is responsible for:

▶ Pickup

– Prompting the customer for the rental confirmation ID
– Finding appropriate rental and displaying rental information to customer
– Completing the rental by completing the workflow step

▶ Return

– Prompting the customer for the vehicle ID
– Completing the rental workflow step

Content Engine connectivity

The class `ContentEngineConnector` provides the primary connection to the CE and offers two primary public methods:

► Connect

The `Connect` method performs the initial connection and authorization to the CE by taking the user name, password, and connection string and retrieving the configured object store.

► GetRentalRecord

The `GetRentalRecord` method fetches a rental record and all associated information from the CE and returns a `RentalRecord` object. The `RentalRecord` object contains the vehicle information (including image) and general information about the rental.

Process Engine connectivity

The class `ProcessEngineConnector` provides the primary connection to the PE and offers three primary public methods:

► Connect

The `Connect` method performs the initial connection and authorization to the PE by taking the user name, password, connection point and connection string and executing a `getQueues()` call to confirm authentication.

► CompletePickup

The `CompletePickup` method takes the confirmation number, finds the appropriate workflow, and completes that step, moving the workflow on to the next step.

► CompleteReturn

The `CompleteReturn` method takes the confirmation number, finds the appropriate workflow, and completes that step, moving the workflow on to the next step.

7.7.3 Architecture: Fleet Status Manager Web application

For a description of the user view of this application, see 7.3.4, "User view: Fleet Status Manager Web application" on page 232.

For a description of the deployment instructions for this application, see 7.8.6, "Deployment: Fleet Status Manager Web application" on page 296.

The Fleet Status Manager application is a J2EE application written using Java Server Faces (JSF). It communicates with the CE (using the CE Java API) and the PE (using the PE Java API). Overall program flow is illustrated in Figure 7-27.
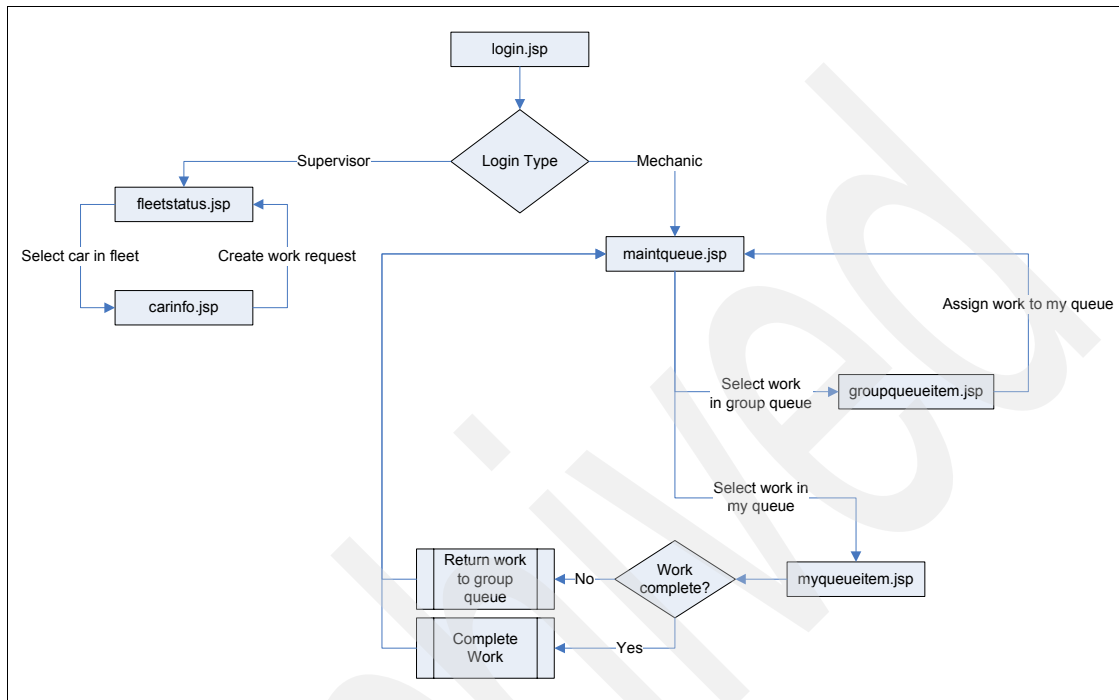
Figure 7-27   Program flow for Fleet Status Manager application

This application consists of the following Java classes:

▶ `itso.autorental.MetadataConstants`: The `Constants` class contains CE and PE property and class names.

▶ `itso.autorental.ITSOUtil`: This class contains various convenience methods, including:

— `loginToFileNet()`: This method establishes connections to CE and PE.

— `loadProperties()`: This method loads configuration data from a properties file.

— `getFleetStatus()`: This method takes a franchise code and object store and retrieves all cars and their statuses within that franchise. This method returns an array which is used for populating the JSF HtmlDataTable.

— `getQueueWorkItems()`: This method retrieves all maintenance activities in a particular franchise code and group queue. This method also returns an array which is used for populating the JSF HtmlDataTable.

- – `assignWorkItemToMyQueue()`: This method takes a queue name, wob number, and a user name. It assigns the work item to this user's queue.

  – `reassignWorkItemToGroupQueue()`: This method takes a wob number and reassigns the work item back to the group queue.

  – `completeWorkItem()`: This method takes a wob number and completes the workflow.

► `itso.autorental.ITSOVehicle`: This class represents an Vehicle object for this application.

► `itso.autorental.UserRole`: This class represents a user role for the logged in user. User can be a maintenance worker or a supervisor.

► `itso.autorental.RentalActivityHandler`: This class has convenience methods, including creating and deleting various vehicle activities:

  – `createMaintenanceActivity()`: This method takes a vehicle id and franchise code and creates a maintenance activity.

  – `deleteVehicleActivity()`: This method removes a vehicle activity (for example, maintenance activity) given the activity ID.

► The `itso.autorental.fleetstatusmanager.beans` package contains the JSF base classes for each JSP.

7.7.4 Architecture: Billing Report application

For a description of the user view of this application, see 7.3.5, "User view: Billing Report application" on page 237.

For a description of the deployment instructions for this application, see 7.8.7, "Deployment: Billing Report application" on page 299.

The Billing Report application is a stand-alone Java application based on the Standard Widget Toolkit (SWT). It uses real P8 credentials to log in to P8 system.

When a user clicks the **Search** button to execute a query, the system calls the CE API to search the ITSOVehicleActivity objects based on the search criteria provided by the user. The application processes the returned search result to calculate the total hours and total numbers for different vehicle activity types. After the calculation process finishes, the calculated result is converted into a JavaScript file. The application loads an HTML file to render this JavaScript file and displays the rendered result in an embedded browser window.

Content Engine connectivity

The classes `CEHelper` and `VehicleHoursReportGenerator` provide the primary connection to the CE and offers several primary methods:

► `CEHelper.login()`: This method performs the initial connection and authorization to the CE by taking the user name, password, and connection string.

► `CEHelper.validateConnection()`: This method validates the connection between the client application and CE. If the connection is not valid, it will throw an exception.

► `CEHelper.getObjectStore()`: This method retrieves the object store according to the object store name.

► `VehicleHoursReportGenerator.queryVehicleHoursSummary()`: This method fetches ITSOVehicleActivity from the CE and calculates the total hours and total numbers for different vehicle activity types.

Figure 7-28 shows the complete application flow diagram.
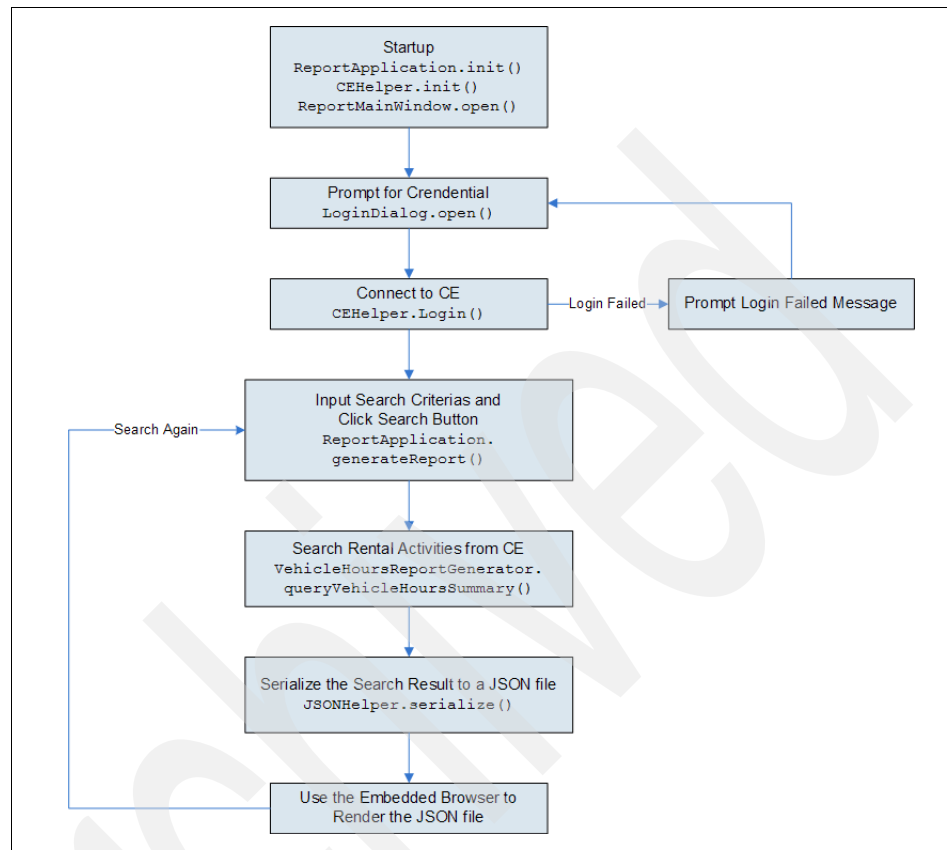
Figure 7-28   Program flow for Billing Report application

7.8 Deployment instructions for sample applications

In this section, we describe how to download, install, and configure the artifacts that comprise the the company application. The installation and configuration procedures can vary depending on the destination environment. Certain procedures that are described in this section assume that the deployment is performed on a Microsoft Windows-based P8 platform, but should easily be able to adjust for other platforms. Most steps are Java-oriented and platform-independent. In most cases, we use forward-slash (/) notation for paths because it is generally interpreted properly on both Windows and non-Windows platforms. In some cases, the application server is assumed to be WebSphere

Application Server 6.1. Because the steps are relatively simple, adjusting for WebLogic or JBoss should be straightforward.

The samples are provided in source form, and we also provide prebuilt binaries enabling you to more easily run them before setting up your development environment. The instructions here are focused mainly on how to run the prebuilt binaries, but, where applicable, we offer guidance for using the sources to re-create those binaries.

> **Note:** Installing and running these sample applications in a production environment is not advisable. Part of the setup procedure involves introducing changes to the CE and PE servers, and for that reason we strongly recommend using a development environment or other non-critical environment.

Getting the sample applications up and running requires the following steps:

1. Download and unpack the samples.

    The IBM Redbooks publications Web site for this book contains a download link to a file named sg247743-sample.zip. (Refer to Appendix A, "Additional material" on page 315 for locating the information.) Download that file and extract it in a convenient location on your local system. Depending on the tools you use locally for some of the steps, you might have to ensure that no spaces exist in the path leading to that location. The top-level directory is called sg247743-sample/, and all of our paths in these instructions are relative to that top-level directory. The overall structure is shown graphically in 7.8.1, "Application package structure" on page 278.

2. Prepare your CE and PE.

    The sample applications work in coordination with an established environment for the CE (for example, custom classes and properties) and PE (for example, workflow definitions). The downloaded sample applications include artifacts that are used to establish those environments. Specific instructions are provided in 7.8.2, "Content Engine artifacts" on page 279 and 7.8.3, "Process Engine artifacts" on page 285.

3. Add P8 and third-party modules.

    These modules will be added to the environment in locations under sg247743-sample/. Where necessary, specific instructions are provided in sections for specific components. We assume that you have a working P8 installation available locally; P8 components can be copied from that environment. You might also find convenience in running the P8 client installer to obtain a local copy of P8 components. In any case, you should use P8 client-side components that are compatible with the P8 CE and PE servers that you will be using.

4. Modify configuration files.

The sample applications use configuration files for URIs pointing to the CE and PE, among other things, so that you can use the binary files in the samples. Specific instructions are provided in sections for specific components where necessary.

5. Deploy each sample component.

Because we use a variety of technologies to demonstrate various things, there are also a variety of steps needed to install and deploy components of the sample applications.

7.8.1 Application package structure

Figure 7-29 depicts the overall directory structure of the downloadable sample package, and Table 7-16 on page 279 briefly describes several of those specific directories. Additional information is given, where applicable, in the sections for specific sample applications.
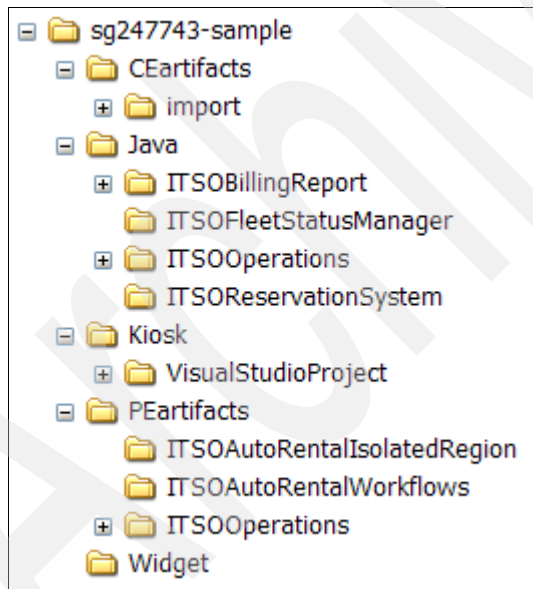
Figure 7-29   Directory structure of sg247743-sample

Table 7-16 describes the high-level directory structure for the downloaded sample applications.

Table 7-16   The sample application directory structure

Directory	Description
CEartifacts/	Information related to the metadata used in the CE. Directory includes a set of files to be imported into the CE to establish custom classes, custom properties, and some instance data.
PEartifacts/	Process definitions and other configuration items for the PE. Also includes source files and precompiled binaries for the ITSOOperations connector for Component Integrator.
Java/	Source files and precompiled binaries for most of the Java-based components of the sample applications. For ITSOReservationSystem and ITSOFleetStatusManager directories, the source files are included in the EAR or WAR files, along with the compiled class files. For the other applications, the sources are in an obvious subdirectory.
Kiosk/	Source files and precompiled binaries for the Windows-only Kiosk application.
Widget/	Prebuilt WAR file for the sample widget. Because the widget is implemented in JavaScript, the WAR file also contains the source files.

7.8.2 Content Engine artifacts

You should have a working P8 system before trying to work with the sample applications. Even so, certain metadata (class and property definitions) and instance data pieces should be added to the CE environment, specifically in support of the sample applications. The sample applications operate with a single ObjectStore of your choosing. Although it is not a requirement, you might want to create an empty ObjectStore just for the purposes of experimenting with the sample applications.

> **Note:** Although problems can sometimes occur when importing data into an ObjectStore at a different CE release than the ObjectStore from which it was exported, the constructs that we use in the data files are straightforward. You are unlikely to have any problem if you are using any IBM FileNet Content Manager 4.5.x or IBM FileNet Business Process Manager 4.5.x release.

The metadata customization for the sample applications consists of custom subclasses. There are no changes to standard system classes. The custom

classes and properties are described in 7.4, "Data model" on page 239. We also provide sample instance data for various CE classes that are used by the sample applications. Importing this sample data is not strictly necessary, but you would otherwise have to manually create certain data if you want your exploration of the sample applications to be meaningful. Although the structure of our sample data correctly matches the data model description, it is only sparsely populated. Other than in structure, the data is not intended to be particularly realistic. Even after you have imported the sample instance data, you might want to create additional instance data of your own for the purposes of experimenting with the sample applications.

You can add this custom metadata and sample instance data to your ObjectStore through the standard import procedure, as follows:

1. Start the IBM FileNet Enterprise Manager application and log on to the P8 domain.

2. Navigate to the ObjectStore you wish to use for the sample applications, right-click on it, and select **All Tasks** → **Import All**. See Figure 7-30.
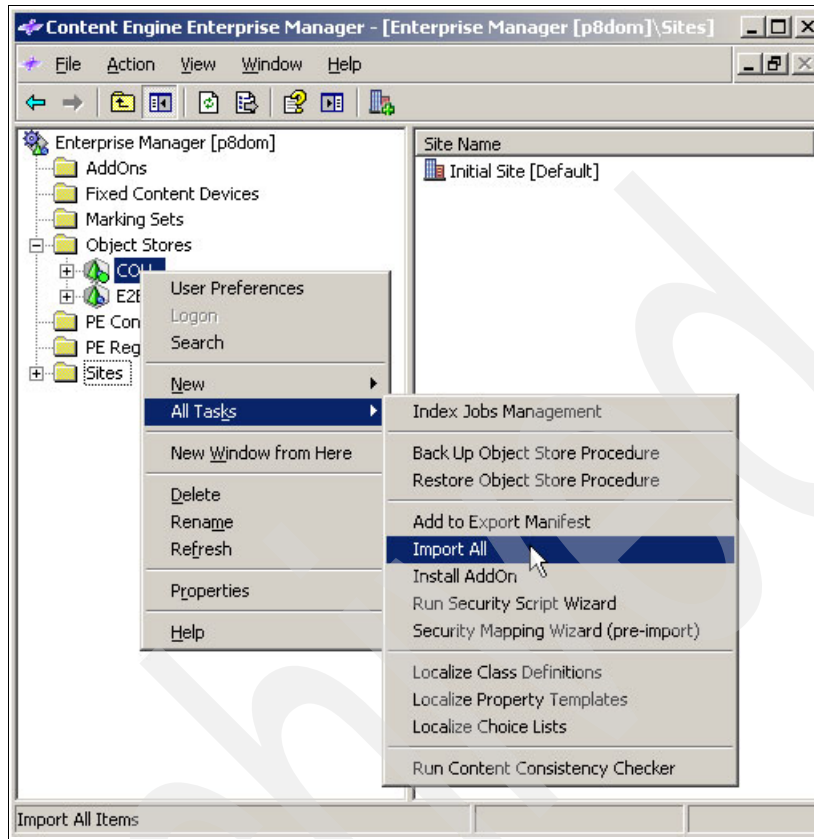
Figure 7-30   CE import task

3. Enterprise Manager presents an import helper panel. On the Import Options tab (see Figure 7-31), enter the following information:

   a. In the Import Manifest File field, browse to and select the file:

      `sg247743-sample/CEartifacts/import/farcasample_CEExport_Manifest.xml`

   b. Leave the External Content Path field empty.

   c. For Storage Location for Imported Objects, select **Default Database Storage Policy** from the drop-down menu.

   d. For Standard Options, check only **Import Object ID** and **Ignore duplicate error messages during import**. Deselect all other boxes. The reason for ignoring duplicates is that a few things in the import data might (or might not) already be present in your ObjectStore. It is also helpful if you have to re-run the import process.
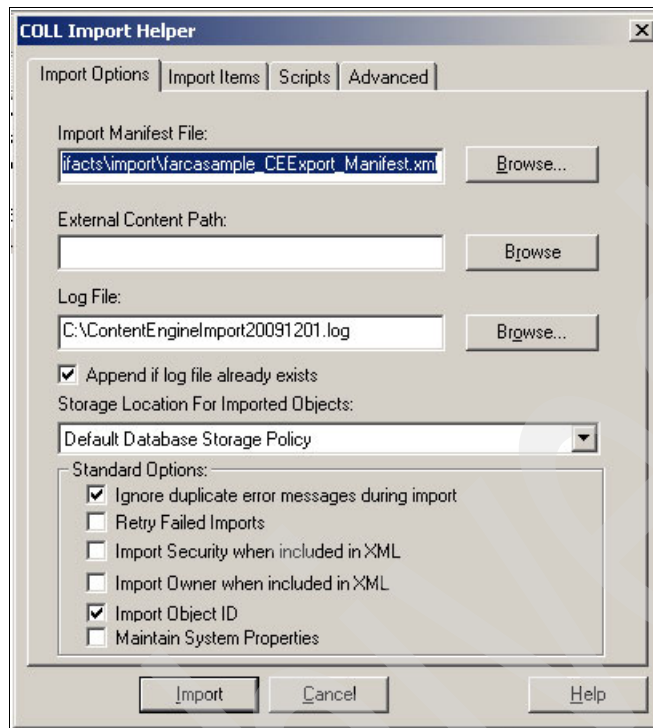
Figure 7-31　CE import helper panel Import Options

4. On the Import Items tab (in Figure 7-32)**,** check only **Property Templates** and **Class Definitions**. Deselect all other boxes. This step causes only metadata objects to be imported. Instance data will be imported in a later step.
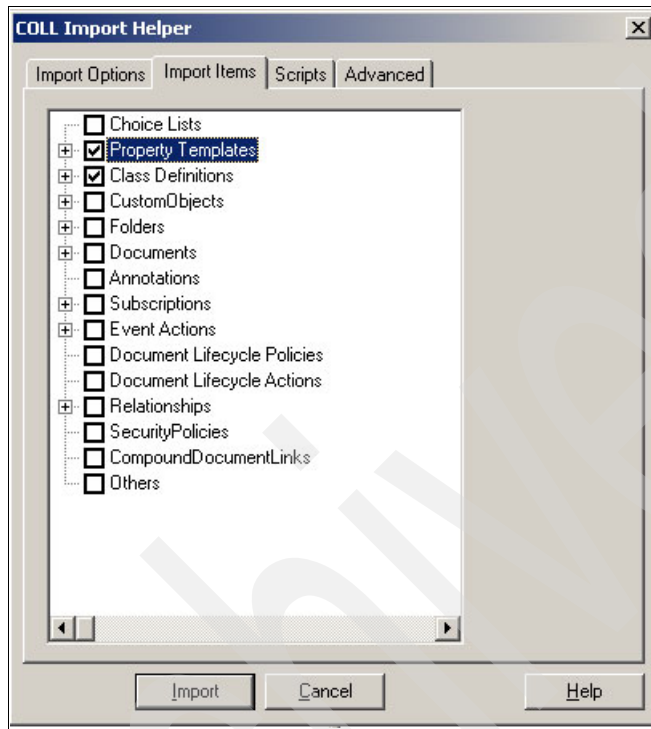
Figure 7-32   CE import helper panel Import Items for metadata

5. On the Advanced tab, make sure all available checkboxes are checked.

6. Click the **Import** button. The import of metadata should proceed without errors or warnings. If problems occur, consult Enterprise Manager documentation and resolve those problems before going further.

7. Repeat these steps, but on the Import Items tab, deselect only **Property Templates** and **Class Definitions** check boxes. Check all other boxes. See Figure 7-33. This step causes instance data to be imported. Depending on your specific version of Enterprise Manager, you might have to repeat the import of instance data to resolve all dependencies. Warnings about duplicated items can be ignored, but errors for missing items must be resolved by repeating the import. You are looking for a summary that says the import was successful, not merely partially successful.
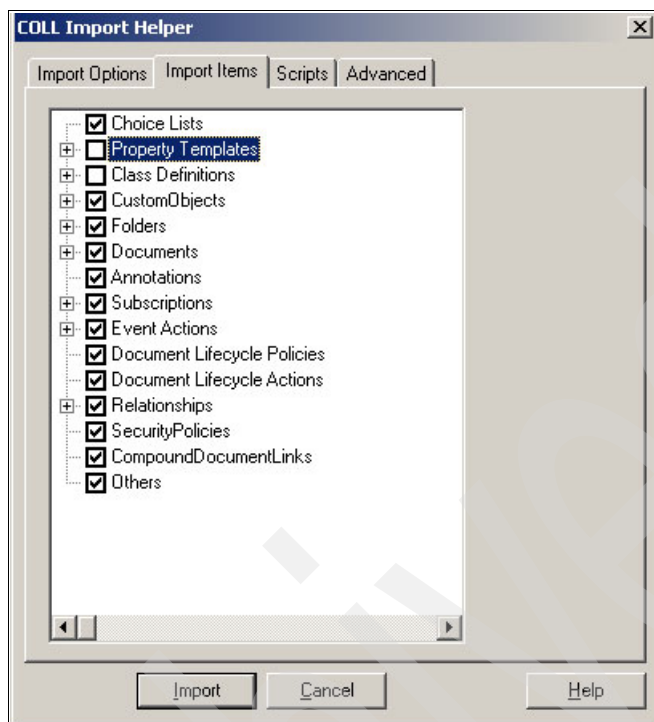
Figure 7-33   CE import helper panel Import Items for instance data

After the instance import, you should see a structure like Figure 7-34 under the /ITSOAutoRental/ folder in your ObjectStore.

Figure 7-34 ITSOAutoRental and subfolders

7.8.3 Process Engine artifacts

The following steps describe how to deploy and configure the PE artifacts required for the sample applications:

1. Ensure that the PE notification settings are properly configured as follows:

   a. Log onto the Application Server.

   b. Open Process Task Manager.

c. Stop all component manager instances; see Figure 7-35.
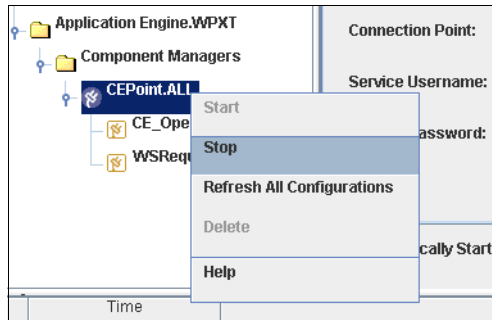
Figure 7-35   Stop all component manager instances

d. Ensure that the PE notification settings are properly configured in the Process Task Manager (Figure 7-36). These notification settings are required in order to send e-mail by using the CE_Operations component.
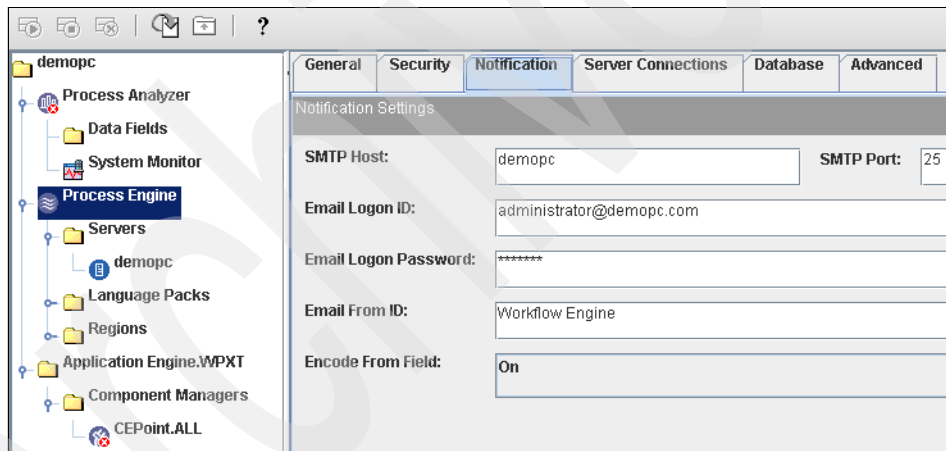
Figure 7-36   Ensure correct notification settings on Process Task Manager

2. Import the `ITSOAutoRentalIsolatedRegion.xml` file as follows:

a. Log onto the Workplace or WorkplaceXT application using an administrator account.

b. Open the Process Configuration Console. See Figure 7-37.

a. Open a connection on the connection point where the new configuration will be imported.

b. Right-click on the selected connection point node and select the **Import from XML file** option.

c. Browse to and select the following file to import:

```
sg247743-sample/PEartifacts/ITSOAutoRentalIsolatedRegion/ITSOAuto
RentalIsolatedRegion.xml
```

d. Select **Merge** as the import type.

e. Click the **Import** button and select **Yes** from the warning message.
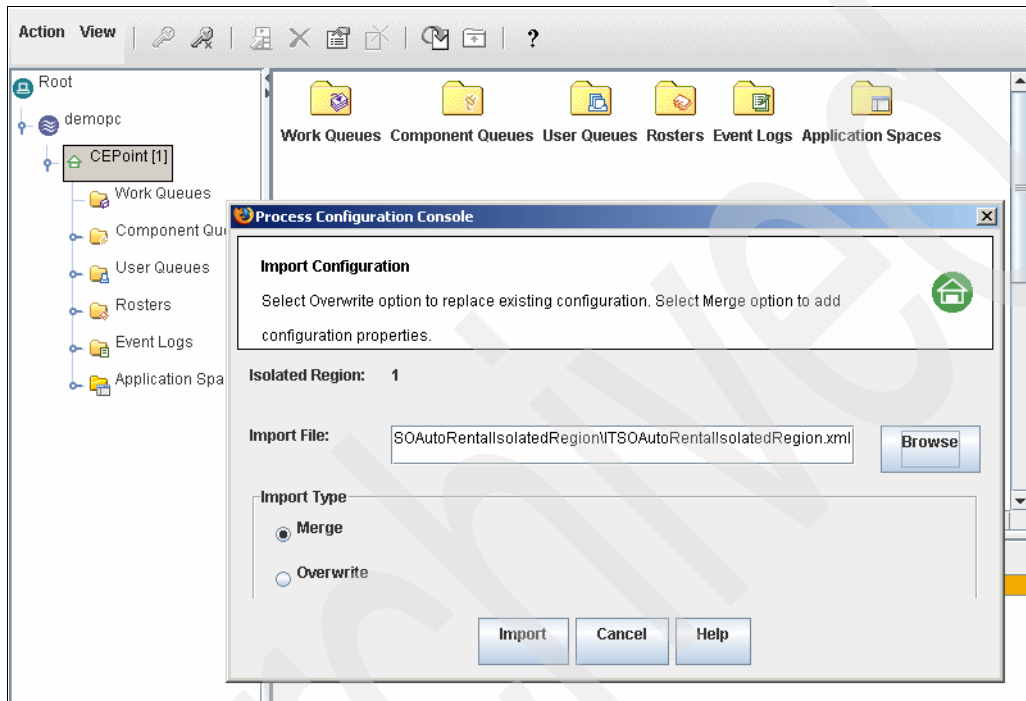
Figure 7-37   Import isolated region configuration

3. Set component queues adaptor security (Figure 7-38) as follows:

a. Select the **ITSO_Operations** component queue from the Component Queues node.

b. Open the component properties and navigate to the Adaptor tab.

c. Enter a valid user name and password for the JAAS Credentials section and click **OK**.

d. Repeat steps a - c for the **CE_Operations** component adaptor.

e. Commit your changes.

Figure 7-38   Set JAAS credentials on component queues

4. Add the workflow definitions (`.pep` files) to the object store by using the Workplace or WorkplaceXT application as follows:

   a. Create a folder where the workflow definitions are to be added. For example, `ITSOAutoRentalWorkflows`.

   b. Add the following workflow definition in the created folder, specifying the document class as **Workflow Definition**:

      `sg247743-sample/PEartifacts/ITSOAutoRentalWorkflows/ITSO Vehicle Reservation Workflow.pep`

   c. Right-click on the added workflow definition and select the **Transfer Workflow** action.

   d. Leave the default prompted name to transfer the workflow definition and click the **Transfer** button.

   e. Repeat steps a to d for the `ITSO Vehicle Maintenance Workflow.pep` workflow definition.

5. Copy the contents of the `sg247743-sample/PEartifacts/ITSOOperations/` directory to the Application Engine server and configure general settings as follows:

   a. Log onto the Application Server and create a custom filesystem folder where the ITSOOperations PE artifacts are to be copied. For example, `ITSOAutoRental`.

   b. Copy `ITSOOperations.jar` and `log4j.xml` files from the sample to the created folder.

   c. Open the `ITSOOperations.jar` file with any compression program (for example, WinZip or the Java `jar` command).

   d. Locate and open the `ITSOAutoRental.properties` file with a text editor, and enter the correct property values for your environment.

   e. Save and update changes performed to `ITSOAutoRental.properties` (see Example 7-1).

Example 7-1   ITSOAutoRental.properties sample

```
##########the company General Settings##########
# Object Store Name
ITSOObjectStore=DemoOS

##########the company Email Settings###########
# Sender Email Address
ITSOEmailFrom=administrator@example.com
# Customer Service Email Address
ITSOCustomerServiceEmail=customer_service@example.com
# Vehicle Maintenance Manager Email Address
ITSOMaintenanceMgrEmail=maintenance_manager@example.com

##############ITSO Malfunction Settings########################
# Debug malfunction flag (default: false)
ITSOMalfunctionDebug=false
# Retry malfunction flag (default: false)
ITSOMalfunctionRetry=false
```

6. Configure ITSOOperations custom logging.

   The `log4j.xml` file provided with the ITSOOperations package is preconfigured on debug level for the specified packages and creates a daily log file at the following location, by default:

   `C:/itso_autorental_ops.log`

Logging parameters may be changed within the `log4j.xml` file according to your environment (Example 7-2). If you are not running on Windows, you should at least change the daily log file name.

Example 7-2   Provided log4j.xml file for the ITSO_Operations component

```
..
<!--  ITSO_Operations Trace Dated Log File -->
    <appender name="ITSOOpsTraceDatedLogFile"
class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File" value="c:/itso_autorental_ops.log"/>
        <param name="DatePattern" value=".dd-MM-yyyy"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %p [%t] %c (%M:%L) -
%m%n"/>
        </layout>
    </appender>
...
    <!--  ITSO_Operations package logging -->
    <logger name="itso.autorental.components" additivity="false">
        <level value="debug"/>
        <appender-ref ref="ITSOOpsTraceDatedLogFile"/>
        <appender-ref ref="console"/>
    </logger>
...
```

7. Add `ITSOLogin` login context to the `taskman.login.config` file:

   a. ITSO_Operations component uses a custom login context named `ITSOLogin`. The following sample file is provided, to show you how this custom login is implemented:

   `sg247743-sample/PEartifacts/ITSOOperations/taskman.login.config.sample`

   b. Navigate to the `taskman.login.config` file that is used by the Component Manager. For example, the file might be in this location:

   `E:\Program Files\FileNet\WebClient\Router\taskman.login.config`

   c. Append the `ITSOLogin` login context, as shown in Example 7-3.

Example 7-3   ITSOLogin login context

```
ITSOLogin
{
    filenet.vw.server.VWLoginModule required;
    itso.autorental.components.authentication.ITSOLoginModule required
        debug=true;
};
```

8. Add ITSO_Operations required library and configure the JRE parameters:

    a. Open the Process Task Manager and select the **Required Libraries** tab on the Component Manager node. See Figure 7-39.

    b. Add the following file to the list of Component Manager required libraries:

       sg247743-sample/PEartifacts/ITSOOperations/ITSOOperations.jar

    c. Click **Apply**.
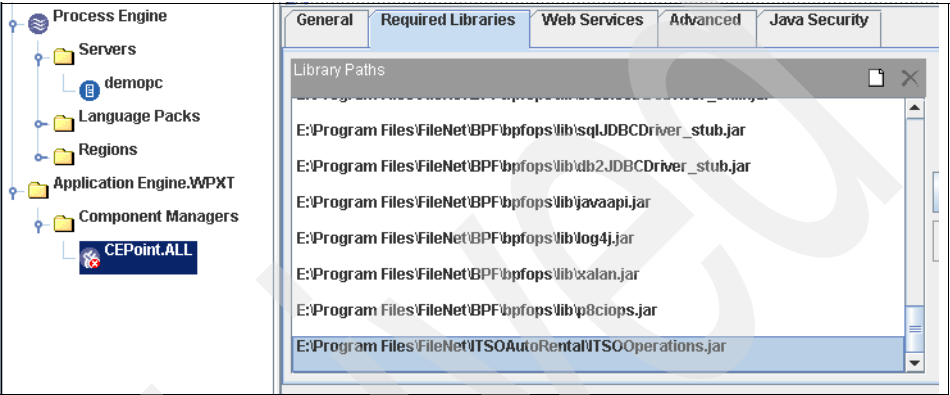
Figure 7-39   ITSOOperations.jar required library

    d. Select the **Advanced** tab (Figure 7-40) on the Component Manager node and add the following required JRE parameter, which is the location of the ITSO_Operations log4j.xml file:

       itsolog4j.configuration

       For example:

       -Ditsolog4j.configuration=E:/Program
       Files/FileNet/ITSOAutoRental/log4j.xml

    e. Click **Apply**.
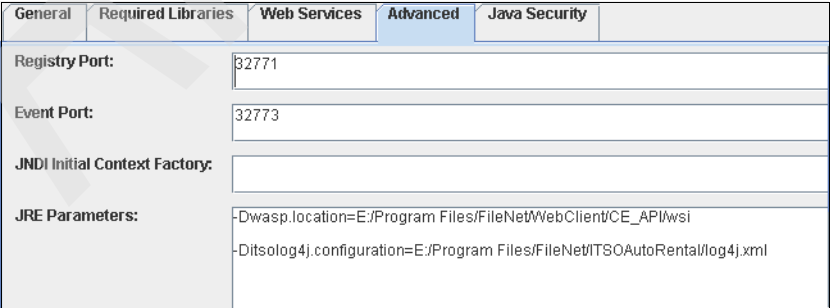
Figure 7-40   ITSO_Operations JRE parameters sample

9. Start the Component Manager and verify that all component queues have started, with no errors. See Figure 7-41.
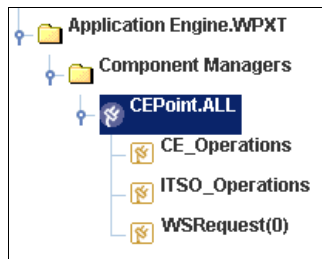
Figure 7-41   Component Manager started with no errors

7.8.4 Deployment: Reservation Web application

For a description of the user view of this application, see 7.3.1, "User view: Reservation Web application" on page 226.

For a description of the internal architecture of this application, see 7.7.1, "Architecture: Reservation Web application" on page 268.

In this section, we give details of preparing and deploying the Reservation Web application. These instructions are very similar to those in 7.8.6, "Deployment: Fleet Status Manager Web application" on page 296.

EAR file preparation

Preparation steps include adding P8 libraries to the EAR file and making other configuration changes. Because JavaServer Faces (JSF) is directly supported by most application servers, adding JSF-related JAR files to the EAR or WAR might not be necessary. Consult your application server documentation for applicable JSF configuration for a Web application.

The preparation steps are as follows:

1. Locate the EAR file:

   `sg247743-sample/Java/ITSOReservationSystem/ITSOCarRentalApp.ear`

2. Unpack the EAR file using a tool that understands ZIP format (for example, WinZip or the Java `jar` command). For the purposes of this explanation, we assume you unpacked it to the `ITSOCarRentalApp.ear` directory.

3. Navigate to the `ITSOFleetStatusManager.ear` directory and locate the following WAR file:

   `ITSOCarRentalApp.ear/ITSOCarRentalApp.war`

4. Unpack the WAR file by using a tool that understands ZIP format (for example, WinZip or the Java `jar` command). For the purposes of this explanation, we assume you unpacked it to the `ITSOCarRentalApp.war` directory.

5. Navigate to the `ITSOCarRentalApp.war/WEB-INF/classes/` directory.

6. Edit the `SampleCarRental.properties` file and set each parameter appropriately for your FileNet environment. Because this application runs inside a J2EE application server, the simplest approach is to configure the CE connection URI to use EJB transport.

7. Navigate to the `ITSOCarRentalApp.war/WEB-INF/lib` directory. You might have to create the `lib` subdirectory.

8. Add the CE and PE API JAR files from your P8 environment, as described in Chapter 2, "Setting up development environments" on page 13. Because you are using CE EJB transport, the CE JAR files are limited to:

   – `Jace.jar`
   – `log4j.jar`

   The PE JAR files are:

   – `pe.jar`
   – `pe3pt.jar`
   – `peResources.jar`

9. Re-create `ITSOCarRentalApp.war`, including the changes to the properties file and the addition of the API JAR files.

10. Re-create `ITSOCarRentalApp.ear`, including the changes to the `ITSOCarRentalApp.war file.`

11. Continue with deployment, as described in the "Deployment" section. In those deployment instructions, we assume that you replaced the original `ITSOCarRentalApp.ear` file with the modified version in the original location.

Deployment

Details of deploying a Web application vary from application server to application server, and details can be different for different releases of the same application server brand. To simplify the explanation, we give the steps to deploy the Reservation Web application in WebSphere Application Server 6.1, as follows:

1. Make sure that you have performed the preparation steps as described in the "EAR file preparation" on page 292.

2. Log on to the WebSphere administrative console

3. Install the `ITSOCarRentalApp.ear` file as follows:

   a. Select **Applications** → **Install New Application**.

   b. Click **Browse** and navigate to the file:

      `sg247743-sample/Java/ITSOReservationSystem/ITSOCarRentalApp.ear`

   c. Click **Next**. Keep all defaults for each dialog page by clicking **Next**, and click **Finish** on the last dialog page. Save all your changes.

4. Configure the class loading and updating information:

   a. Select **Applications** → **Enterprise Applications**.

   b. Select the **ITSOCarRentalApp** link.

   c. Select **Class loading and update detection**.

   d. For the Class loader order field, select **Classes loaded with application class loader first**.

   e. Accept the default value for the Polling interval for updated files field.

   f. Click **OK**. Click **Save**.

5. Configure class loader order information:

   a. Select **Applications** → **Enterprise Applications**.

   b. Select the **ITSOCarRentalApp** link.

   c. Click **Manage Modules**.

   d. Select the **ITSOCarRentalWeb** link.

   e. For the Class loader order field, select **Classes loaded with application class loader first**.

   f. Click **OK**. Click **Save**.

6. If you are using CE EJB transport, no extra JVM parameters are required. If you are using CEWS transport with a CE release before 4.5.1, set up the JVM parameters:

   a. Go to the WebSphere administrative console.

   b. Select **Servers** → **Application servers**.

   c. Click on your server (for example, `server1`).

   d. Select **Java and Process Management** → **Process Definition** → **Java Virtual Machine**.

   e. On the Generic JVM Arguments tab, enter the JVM parameters that are applicable to CEWS transport, as described in Chapter 2, "Setting up development environments" on page 13.

   f. Click **OK**. Click **Save** to save your changes.

7. Start your application in WebSphere:

   a. Select **Applications** → **Enterprise Applications**.

   b. Select the **ITSOCarRentalApp** check box.

   c. Click **Start**. Your application should start properly.

8. Launch your application from a browser:

   a. Open a Web browser.

   b. Navigate to:

      `http://<YourAppServer:port>/ITSOCarRentalWeb/Login.faces`

   c. Log in using a customer account. The first time, you have to follow the registration link to create a customer account.

7.8.5 Deployment: Kiosk application

For a description of the user view of this application, see 7.3.2, "User view: Kiosk application" on page 230.

For a description of the internal architecture of this application, see 7.7.2, "Architecture: Kiosk application" on page 270.

Because it is implemented as a .NET application, the Kiosk application only runs on Windows. To deploy and run the Kiosk application:

1. Ensure the Microsoft .NET 2.0 framework is installed.

2. Ensure the Microsoft Web Services Enhancements 3.0 is installed.

3. Ensure the IBM FileNet P8 .NET client is installed or copy `FileNet.Api.dll` from a Windows CE client environment (for example, where your IBM FileNet Enterprise Manager is installed) to the same directory as the Kiosk application.

4. Change to the directory of the Kiosk application:

   `sg247743-sample/Kiosk/`

5. Run `ITSO.Application.exe`, either from the command line or by double-clicking it from Windows Explorer.

6. On the initial screen, enter the user name, password, CE connection URI (for example, `http://<CEServer:port>/wsi/FNCEWS40MTOM`), object store, and Process Engine connection point.

7. Click **Connect**.

7.8.6 Deployment: Fleet Status Manager Web application

For a description of the user view of this application, see 7.3.4, "User view: Fleet Status Manager Web application" on page 232.

For a description of the internal architecture of this application, see 7.7.3, "Architecture: Fleet Status Manager Web application" on page 272.

In this section, we give details of preparing and deploying the Fleet Status Manager Web application. These instructions are very similar to those in 7.8.4, "Deployment: Reservation Web application" on page 292.

EAR file preparation

Preparation steps include adding P8 libraries to the EAR file and making other configuration changes. Because JavaServer Faces (JSF) is directly supported by most application servers, adding JSF-related JAR files to the EAR or WAR might not be necessary. Consult your application server documentation for applicable JSF configuration for a Web application.

The preparation steps are as follows:

1. Locate the EAR file:

   `sg247743-sample/Java/ITSOFleetStatusManager/ITSOFleetStatusManagerApp.ear`

2. Unpack the EAR file using a tool that understands ZIP format (for example, WinZip or the Java `jar` command). For the purposes of this explanation, we assume you unpacked it to the following directory:

   `ITSOFleetStatusManagerApp.ear`

3. Navigate to the `ITSOFleetStatusManager.ear` directory and locate the following WAR file:

   `ITSOFleetStatusManager.ear/ITSOFleetStatusManager.war`

4. Unpack the WAR file by using a tool that understands ZIP format (for example, WinZip or the Java `jar` command). For the purposes of this explanation, we assume you unpacked it to the following directory:

   `ITSOFleetStatusManagerApp.war`

5. Navigate to the `ITSOFleetStatusManager.war/WEB-INF/classes/` directory.

6. Edit the `SampleCarRental.properties` file and set each parameter appropriately for your FileNet environment. Because this application runs inside a J2EE application server, the simplest approach is to configure the CE connection URI to use EJB transport.

7. Navigate to `ITSOFleetStatusManager.war/WEB-INF/lib` directory. You might have to create the `lib` subdirectory.

8. Add the CE and PE API JAR files from your P8 environment, as described in Chapter 2, "Setting up development environments" on page 13.

   Because you are using CE EJB transport, the CE JAR files are limited to:

   – `Jace.jar`
   – `log4j.jar`

   The PE JAR file are:

   – `pe.jar`
   – `pe3pt.jar`
   – `peResources.jar`

9. Re-create the `ITSOFleetStatusManagerApp.war` file, including the changes to the properties file and the addition of the API JAR files.

10. Re-create the `ITSOFleetStatusManagerApp.ear` file, including the changes to `ITSOFleetStatusManagerApp.war` file.

11. Continue with deployment, as described in the section Deployment. In those deployment instructions, we assume that you replaced the original `ITSOFleetStatusManagerApp.ear` with the modified version in the original location.

Deployment

Details of deploying a Web application vary from application server to application server, and details can be different for different releases of the same application server brand. To simplify the explanation, we give the steps to deploy the FleetStatusManager Web application in WebSphere Application Server 6.1.

1. Make sure that you have performed the preparation steps as described in the section "EAR file preparation" on page 296.

2. Log on to the WebSphere administrative console

3. Install the `ITSOFleetStatusManager.ear` file:

   a. Select **Applications** → **Install New Application**.

   b. Click **Browse** and navigate to the file:

      `sg247743-sample/Java/ITSOFleetStatusManager/ITSOFleetStatusManager.ear`

   c. Click **Next**. Keep all defaults for each dialog page by clicking **Next**, and click **Finish** on the last dialog page. Save all your changes.

4. Configure the class loading and updating information:

   a. Select **Applications** → **Enterprise Applications**.

   b. Select the **ITSOFleetStatusManagerApp** link.

   c. Select **Class loading and update detection**.

d. For the Class loader order field, select **Classes loaded with application class loader first**.

e. Accept the default value for Polling interval for updated files field. Click **OK**. Click **Save**.

5. Configure class loader order information:

   a. Select **Applications → Enterprise Applications**.

   b. Select the **ITSOFleetStatusManagerApp** link.

   c. Click **Manage Modules**.

   d. Select the **ITSOFleetStatusManager** link.

   e. For the Class loader order field, select **Classes loaded with application class loader first**.

   f. Click **OK**. Click **Save**.

6. If you are using CE EJB transport, no extra JVM parameters are required. If you are using CEWS transport with a CE release before 4.5.1, set up the JVM parameters:

   a. Go to the WebSphere administrative console.

   b. Select **Server → Application servers**.

   c. Click on your server (for example, `server1`).

   d. Select **Java and Process Management → Process Definition → Java Virtual Machine**.

   e. On the Generic JVM Arguments tab, enter the JVM parameters that are applicable to CEWS transport, as described in Chapter 2, "Setting up development environments" on page 13.

   f. Click **OK**. Click **Save** to save your changes.

7. Start your application in WebSphere:

   a. Select **Applications → Enterprise Applications**.

   b. Select the **ITSOFleetStatusManager** check box.

   c. Click **Start**. Your application starts.

8. Launch your application from a browser:

   a. Open a Web browser.

   b. Navigate to:

      `http://<YourAppServer:port>/ITSOFleetStatusManager/login.faces`

   c. Log in by using a valid LDAP account.

7.8.7 Deployment: Billing Report application

For a description of the user view of this application, see 7.3.5, "User view: Billing Report application" on page 237.

For a description of the internal architecture of this application, see 7.7.4, "Architecture: Billing Report application" on page 274.

The Billing Report is a thick desktop Java application with a graphical user interface (GUI). It can be run from the command line (which we describe briefly); it can also be run from within the Eclipse IDE, which we describe in detail. Eclipse is a popular integrated development environment for Java and other application types. It is available from:

http://www.eclipse.org

These deployment instructions were developed by using Eclipse 3.5, but any recent Eclipse release is similar.

Both the command line and the Eclipse IDE methods share a common directory structure, and for both methods, you will need a JAR file from Eclipse Standard Widget Toolkit (SWT). Download the SWT package from the following location, and select the version for your operating system:

http://www.eclipse.org/swt/

Although the Billing Report application was developed with SWT 3.4, it is not especially sensitive to the SWT version, so any recent version should work. Save the `.zip` file in a convenient location in your environment.

Preparing the application environment

The first steps for deploying the Billing Report application are to add P8 and third-party components to the environment, as follows:

1. Change directory to:

   `sg247743-sample/Java/ITSOBillingReport/BillingReportApplication/deploy/`

2. Extract the `swt.jar` file (from the downloaded SWT ZIP archive) into the `lib/` subdirectory.

3. Copy CE client JAR files to the `lib/` subdirectory. JAR files include `Jace.jar`, `log4j.jar`, and other transport-dependent JAR files. For our sample setup, we use CEWS transport, assuming a P8 4.5.1 environment. Therefore, copy the applicable JAR files into the `lib/` subdirectory. JAR files include `stax-api.jar`, `xlxpScanner.jar`, and `xlxpScannerUtils.jar` file. If you want to use EJB transport or use CEWS transport with a pre-4.5.1 environment, consult general setup instructions for those environments in Chapter 2, "Setting up development environments" on page 13.

Running from the command line

To run the application from the command line:

1. If you are not already there, change directory to:

    `sg247743-sample/Java/ITSOBillingReport/BillingReportApplication/deploy/`

2. Edit file `billing_wsi.bat` file. Ensure that the paths and environment variable values are correct for your environment. The script assumes that your current directory is the `deploy/` directory and that dependent JAR files are located in the `lib/` subdirectory. At a minimum, you should supply appropriate values for an ObjectStore name and the CE connection URI.

3. Run the `billing_wsi.bat` file. Further details of the running application are given in 7.3.5, "User view: Billing Report application" on page 237.

Importing into Eclipse

As a preparatory step for deploying the Billing Report application, first import the Billing Report application as a project into Eclipse.

1. Run Eclipse. If you have multiple workspaces, select the workspace where you want to place the Billing Report application.

2. Select **File → Import**.

3. In the pop-up dialog, select **Existing Projects into Workspace**.

4. In the next window, select **Select root directory**, and browse to the following directory:

    `sg247743-sample/Java/ITSOBillingReport/BillingReportApplication/`

    If you were to visit that directory outside of eclipse, you would see that it contains files `.project` and `.classpath` to provide project and dependency data to Eclipse.

5. Make sure that the check box **Copy project into workspace** is *not* checked.

6. Click **Finish** to import the application. See Figure 7-42.
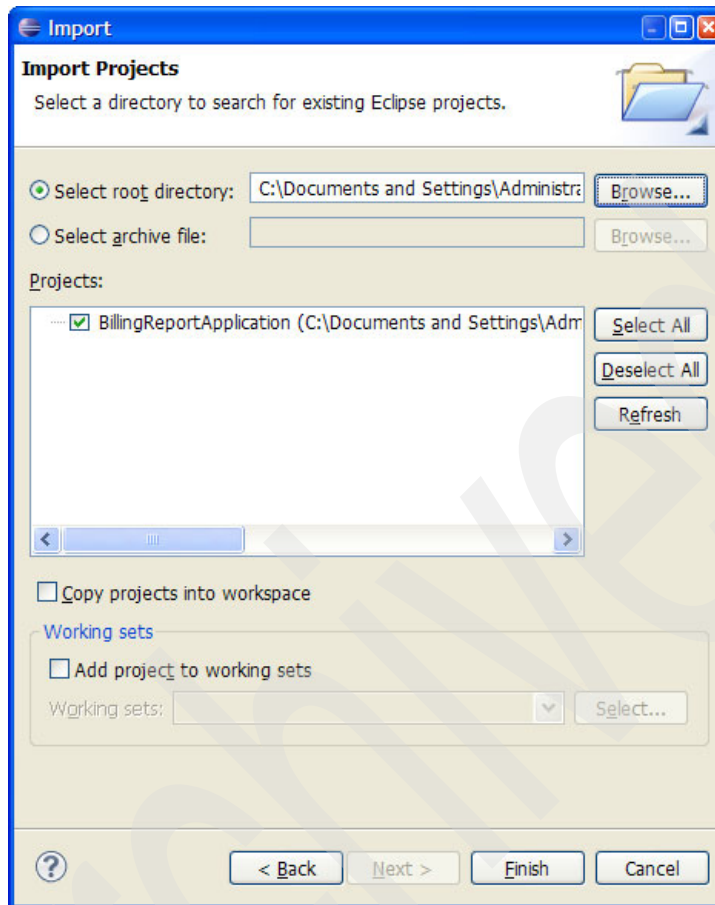
Figure 7-42   Importing the BillingReportApplication project into Eclipse

If you have chosen to follow our example and prepared the `deploy/lib/` subdirectory with CEWS transport JARs for a P8 4.5.1 environment, all dependencies in the Eclipse project will probably be resolved automatically. Otherwise, you will have to make adjustments to the Eclipse build path for the project. We do not describe that here.

Your view of the BillingReportApplication project in the Eclipse project explorer should look like Figure 7-43. Depending on your exact Eclipse version and your personal preference settings, the view could be somewhat different.
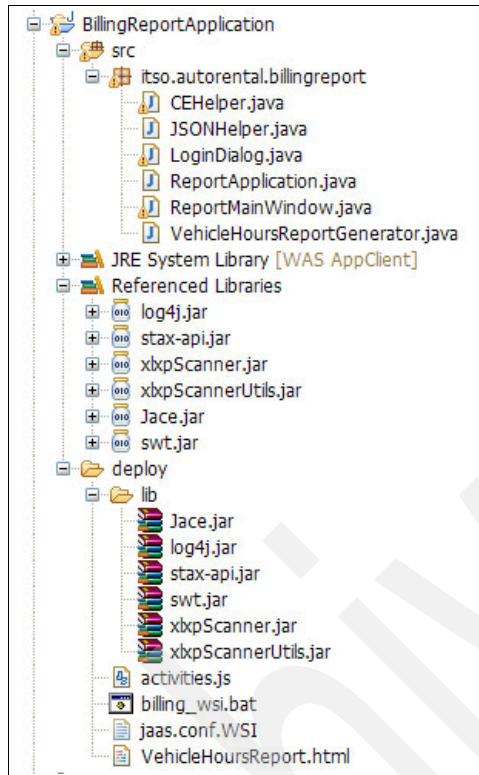
Figure 7-43   BillingReportApplication structure in Eclipse

The BillingReportApplication project in Eclipse is set up to find source files in the
src/ directory and compile them into the deploy/classes/ directory (which does
not ordinarily show up in the Eclipse project explorer view). Dependent JAR files
are found in the deploy/lib/ subdirectory. From the project explorer view, you
can examine the source files at your leisure.

Running from Eclipse

To run the application from Eclipse:

1. Create a new run configuration in Eclipse. There are a few ways to do this, but
   perhaps the simplest way is to right-click on **ReportApplication.java** and
   select **Run as** → **Java application**. See Figure 7-44. The application either
   fails immediately or the GUI presents you with a login window. If the latter
   happens, exit the application. Even if the application does not launch
   correctly, it still creates a configuration that has many correct settings that you
   can use later.

Figure 7-44   Run As Java Application

2. From the Eclipse menu bar, select **Run** → **Run Configurations**.

3. The ReportApplication project should already be selected. If it is not, navigate to it. On the Main tab, the following Main class should already be filled in:

   `itso.autorental.billingreport.ReportApplication`

   If it is not filled in, enter that value.

4. Click the **Arguments** tab and provide the VM arguments setting, as shown in Figure 7-45. Be careful to put these in the **VM arguments** box and not the **Program arguments** box. Notice that these arguments are very similar to those used in the BAT file for the command line procedure in "Running from the command line" on page 300. They include (with values that must be adjusted for your environment):

   ```
   -Ddefault.objectstore.name=MyObjectStore
   -Djaas.stanza=FileNetP8WSI
   -Dce.uri=http://server.example.com:9080/wsi/FNCEWS40MTOM
   -Djava.security.auth.login.config="jaas.conf.WSI"
   ```

5. In the Working directory subpanel, select **Other**, click the **Workspace** button, and navigate to the `BillingReportApplication/deploy/` directory. This is the same working directory we used when running from the command line.
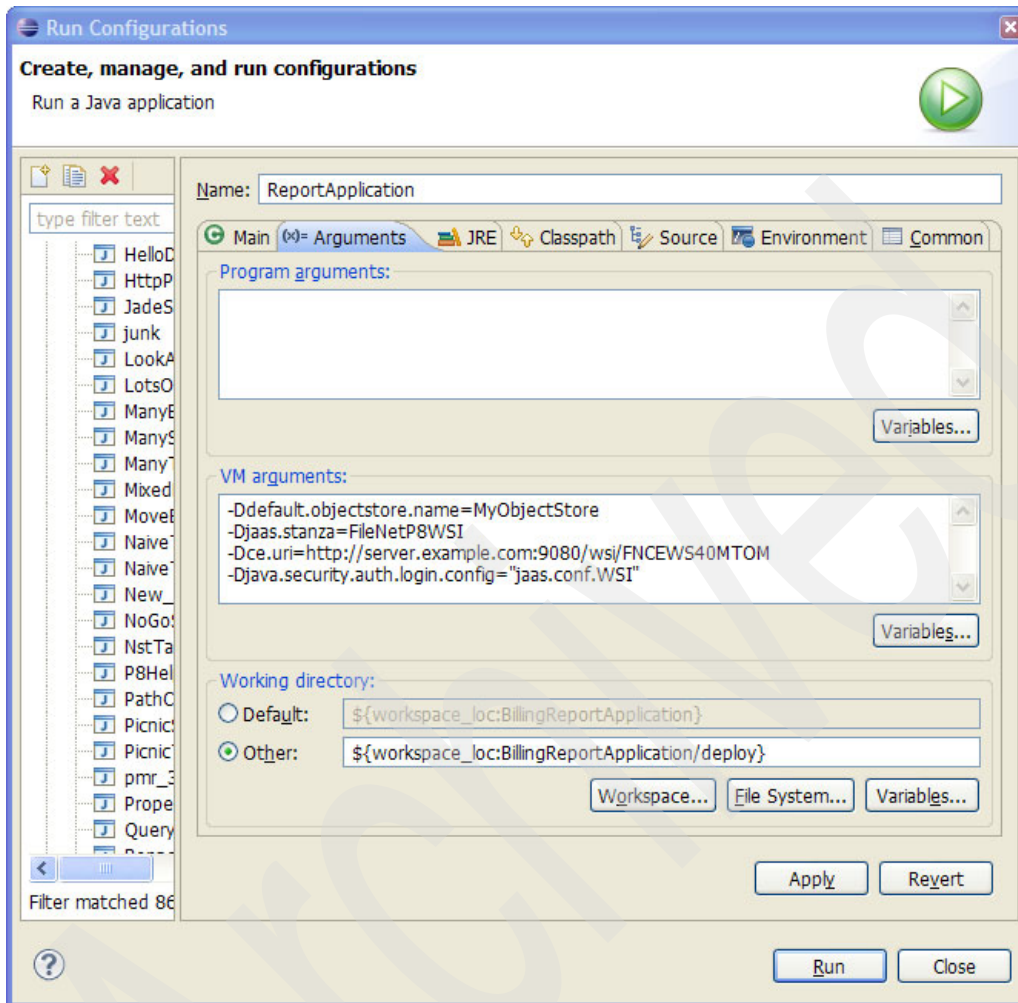
Figure 7-45   Billing application JVM requirements

6. Click **Apply** and click **Close**. You now have the correct run configuration for the Billing Report application.

7. Back in the Eclipse project, right-click the **ReportApplication.java** and select **Run as → Java application** to run the Billing Report application. You should now see the main window of the Billing Report application. Further details of the running application are given in 7.3.5, "User view: Billing Report application" on page 237.

8

Logging and troubleshooting

In this chapter, we discuss how to enable and interpret the various logging options for the IBM FileNet P8 Platform application programming interfaces (APIs). We also direct you to several technical troubleshooting articles and provide important data gathering points when troubleshooting IBM FileNet programming issues.

This chapter discusses logging only from the perspective of the client-side API components. The topic of server-side logs that are generated by Content Engine (CE) and Process Engine (PE) is a larger topic for overall troubleshooting and is not within the scope of this book.

This chapter discusses the following topics:

► How to enable logging for the various IBM FileNet APIs

► How to read the IBM FileNet API log files

► Pointers to technical articles for troubleshooting

► Data to gather when reporting IBM FileNet API issues

8.1 Logging

In this section, we describe how to enable logging for various FileNet APIs, including CE and PE Java API, CE .NET API, and Content Engine Web Services (CEWS). We also present several examples of interpreting the client-side API log files.

8.1.1 CE Java API

The CE Java API uses Apache log4j for a configurable set of logging options. You can obtain details of log4j configuration and options on Apache's log4j Web site:

http://logging.apache.org/log4j

Setting up the CE Java API log4j environment

You can enable logging in the CE Java API in more than one way.

One way is mentioned in FileNet ECM online help documentation. The documentation mentions packaging a `log4j.properties` file into a custom JAR file and deploying it to the application's classpath. Here are those steps:

1. Make a copy of the following file and copy it to the client application server:
   `FileNet\ContentEngine\config\samples\log4j.properties.client`

2. Create an empty JAR file with an easily identifiable name such as `debugCElog4j.jar`. Making this new JAR enables you to perform a simple process later to disable CE Java API logging by simply removing this `debugCElog4j.jar` file.

3. Remove the `.client` extension from the `log4j.properties.client` file. Edit this `log4j.properties` file, and add it to the root directory of the `debugCElog4j.jar` file. We discuss the specifics of modifying the `log4j.properties` file in "Modifying the CE log4j.properties file" on page 307.

4. Copy the `debugCElog4j.jar` file to either the application's `WEB-INF\lib` folder or a specified location in the JVM CLASSPATH parameter.

5. Restart the application. Executing the CE Java API code generates two files:
   ```
   p8_api_trace.log
   p8_api_error.log.
   ```

Another way to enable log4j logging is to configure a JVM argument that specifies the location of the `log4j.properties` file (on MicroSoft Windows):

`-Dlog4j.configuration=file:c:\<YourPath>\log4j.properties`

The benefit of this JVM argument method is the ability to change log4j log levels dynamically by simply modifying the log4j.properties file and does not require restarting the JVM.

Modifying the CE log4j.properties file

The CE log4j.properties file contains subcomponents of the tracing API that can be configured. If you are interested only in enabling full CE Java API tracing, follow these steps:

1. Uncomment the following line to enable the client API error logger:

   `#log4j.logger.filenet_error.api = warn`

2. Change the following line from off to debug:

   ```
   #log4j.logger.filenet_tracing = off, FileNetTraceRollingAppender
   log4j.logger.filenet_tracing = debug, FileNetTraceRollingAppender
   ```

3. Uncomment the following line for filenet_trace.api logging:

   `log4j.logger.filenet_tracing.api = debug`

More options for modifying the log4j.properties file can be found by selecting:

ECM_help → Developer Help → Content Engine Development → Java and .NET Developer's Guide → Trace Logging → Working with the Apache log4j Configuration File → Sample log4j Configuration Files

As mentioned in the previous section, you can disable tracing dynamically if you use the JVM argument method, which removes the necessity to restart the JVM.

Log file size and rollover

By default, the various log4j appenders in the log4j.properties file have the following file size and file count settings:

```
MaxFileSize=100MB
MaxBackupIndex=1
```

These settings specify that only one log file will be generated and overwritten when the log file size exceeds 100 MB. It might be necessary to increase the MaxFileSize and MaxBackupIndex values so that logs are not being overwritten, resulting in lost data. We recommend that you increase the MaxBackupIndex when tracing data over a lengthy amount of time.

Reading the p8_api_trace.log file

Reading the p8_api_trace.log file takes some experience and familiarity before fully understanding these logs. We provide several tips to get you started.

CE Java API version

Example 8-1 displays the build number for the client CE Java API library. This build number can be cross referenced against the P8 Patch Dependency spreadsheet found on the IBM FileNet P8 support site.

Example 8-1   BuildVersion output in CE Java API trace file

```
DEBUG [WebContainer : 1] - Configuration for:BuildVersion
value:dap440.247 applied
```

This `BuildVersion` is useful to determine the exact CE API patch level you are using, and which helps when you are trying to identify a particular patch level that you must be on for a particular bug fix.

This `BuildVersion` is also useful in determining whether your client CE APIs are of the same version as your CE server. CE server also outputs the build number when logging is enabled. You also see a message similar to Example 8-2 in your CE server's `p8_server_trace.log` file if the client and server's build versions do not match. The local version is always the version (from the point of view of the component) that is producing the log line.

Example 8-2   Mismatch build version message in CE Server trace file

```
2008-04-24T20:56:49.314Z B129CD3B API  - The transport build version
does not match the local build version: transport=dap000.441,
local=dap435.029
```

Thread Identifiers

Because client applications are likely to be multithreaded, multi-user environments, the CE Java API logs can become inundated with data across various threads and users. To follow the log entries for a particular thread, look at the thread ID, such as thread ID 3 in Example 8-3. The thread ID, which appears in all log lines, is the part inside [square brackets]. Formats of thread IDs vary from one application server to another.

Example 8-3   Thread identifier

```
DEBUG [WebContainer : 3] - UserContext.set() invoked in its creator
Thread <885535944> for UserContext <null>.
```

Request / Response

The request and data that are sent to the CE server, as well as the resulting response, can be useful when troubleshooting issues. Example 8-4 on page 309 shows an example of an outgoing `executeSearchRequest` call and Example 8-5 on page 309 shows the corresponding result. The format of the logged data is

logically similar to, but not the same as, the corresponding CEWS data on the wire. When similar data is logged on the CE server, the hard line breaks are replaced with the character sequence \n so that logging stays on a single line.

Example 8-4   executeSearchRequest call

```
2009-03-25 14:08:16,078 DEBUG [WebContainer : 3] -
<executeSearchRequest
clientCaller="com.filenet.ae.bean.datasource.CeQuery.executeQuery(CeQue
ry.java:191)" maxElements="200" continueable="true" rid="1">
  <searchMode type="SearchMode" value="1" rid="2"/>
  <searchScope type="ObjectStoreScope" rid="3">
    <objectStore type="GlobalIdentity" classIdentity="ObjectStore"
identity="{5B4ADE2D-2FCC-4D8E-AFFC-B406A7123235}" rid="4"/>
  </searchScope>
  <search type="RepositorySearch" searchSQL="SELECT [VersionStatus],
[IsReserved], [Id], [MajorVersionNumber], [DateLastModified],
[DocumentTitle], [MinorVersionNumber], [ContentSize],
[CompoundDocumentState], [ClassDescription], [IsCurrentVersion],
[VersionSeries], [LastModifier], [MimeType] FROM Document WHERE THIS
INFOLDER '{4DC50869-7E4E-4EFF-9CEA-0D413B0E48EB}' ORDER BY
DocumentTitle" rid="5">
    <propertyFilter type="PropertyFilter" rid="6">
<includePropertyList size="0" rid="7"/>
      <includeTypeList size="0" rid="8"/>
      <excludePropertyList size="0" rid="9"/>
    </propertyFilter>
  </search>
</executeSearchRequest>
```

Example 8-5   Corresponding executeSearchResponse

```
2009-03-25 14:08:16,078 DEBUG [WebContainer : 3] -
<executeSearchResponse elapsed="0" rid="1">
  <value type="RepositoryRowCollection" stale="false" rid="2">
    <list size="2" rid="3">
      <entry type="RepositoryRow" rid="4">
        <properties type="RepositoryRowProperties" rid="5">
          <property type="PropertyInteger32" name="VersionStatus"
dirty="false" access="1" rid="6">
            <value>1</value>
          </property>
          <property type="PropertyBoolean" name="IsReserved"
dirty="false" access="1" rid="7">
            <value>false</value>
          </property>
```

```
              <property type="PropertyId" name="Id" dirty="false"
access="1" rid="8">
                  <value
type="Id">{C9712786-4B17-4512-9E13-9F4154B35FC2}</value>
              </property>
              <property type="PropertyInteger32" name="MajorVersionNumber"
dirty="false" access="1" rid="9">
                  <value>1</value>
              </property>
              <property type="PropertyDateTime" name="DateLastModified"
dirty="false" access="1" rid="10">
                  <value>2009-03-19T20:16:31.406Z</value>
              </property>

  <!-- ... portions omitted ... -->
  <paging type="DefaultPaging" rid="44">
        <connection type="Connection"
uri="iiop://hqdemo1:2810/FileNet/Engine"
asURI="iiop://hqdemo1:2810/FileNet/Engine" rid="45">
          <parameters type="ConfigurationParameters" rid="46">
            <values size="0" rid="47"/>
          </parameters>
        </connection>
      </paging>
    </value>
  </executeSearchResponse>
```

8.1.2 CE .NET API

As of the current P8 4.5 release, the CE .NET API does not have client-side
logging. You can enable CE Server logging in FileNet Enterprise Manager to
trace the requests from the client. Because the .NET API always uses CEWS
transport, you can also use network tracing to capture the actual requests and
responses.

8.1.3 Content Engine Web Services

CEWS clients are stub classes that are generated from the server WSDLs. As
such, these stub classes do not contain client-side logging unless it is provided
by the client Web services toolkit that you are using. You can enable CE Server
logging in FileNet Enterprise Manager to trace the requests from the client. You
can also use network tracing to capture the actual requests and responses.

8.1.4 PE Java API

The PE Java API also uses log4j for a configurable set of logging options.

Setting up the PE Java API log4j environment

To enable the PE Java API, follow these steps:

1. Make a copy of the following file and copy it to the client application server's JRE `lib` folder:

   `\fnsw_loc\sd\fnlog4j.properties.sample`

2. Remove the .sample extension from the `fnlog4j.properties.sample` file. Edit this `fnlog4j.properties` file.

3. Modify in the `fnlog4j.properties` file, which is discussed in the next section ("Modifying the fnlog4j.properties file").

Modifying the fnlog4j.properties file

The PE `fnlog4j.properties` file contains various subcomponents that can be configured. If you are interested in enabling full PE Java API client tracing, set the loggers to DEBUG mode in the PE `fnlog4j.properties` file:

```
log4j.debug=true
log4j.logger.filenet.vw=DEBUG, CON, TXT
log4j.logger.filenet.pe=DEBUG, CON, TXT
```

You can disable full PE Java API tracing dynamically by changing the following loggers back to INFO mode in the file:

```
log4j.logger.filenet.vw=INFO, CON, TXT
log4j.logger.filenet.pe=INFO, CON, TXT
```

8.1.5 Process Engine Web Services

Process Engine Web Services (PEWS) clients are stub classes that are generated from the server WSDLs. As such, these stub classes do not contain any client side logging unless it is provided by the client Web services toolkit you are using. You can enable the PEWS listener on the CE and PE server by setting the following logger:

```
log4j.logger.filenet.ws=DEBUG,CON,TXT
```

8.1.6 PE REST Service

You can enable the REST Service trace logging on the Application Engine and WorkplaceXT server by setting the loggers as shown in Example 8-6.

Example 8-6   Enable PE REST Service tracing

```
# for tracking REST accesses-----------------------------------------
log4j.appender.REST=org.apache.log4j.DailyRollingFileAppender
log4j.appender.REST.layout=org.apache.log4j.PatternLayout
log4j.appender.REST.layout.ConversionPattern=%d{yyyy/MM/dd HH:mm:ss} [%t] %m%n
log4j.appender.REST.File=c:\\RESTLog.txt
log4j.appender.REST.Append=true

# ---------- filenet.pe.rest.* loggers---------
log4j.logger.filenet.pe.rest=ERROR,TXT
#------------------------------------------------
log4j.logger.filenet.pe.rest.servlet=ERROR
log4j.logger.filenet.pe.rest.request=ERROR
log4j.logger.filenet.pe.rest.response=ERROR
log4j.logger.filenet.pe.rest.handler=ERROR
log4j.logger.filenet.pe.rest.utils=ERROR
```

8.2 Troubleshooting

In this section, we describe troubleshooting techniques for CE and PE. We also discuss the data that you must gather so that the IBM support team can help you with troubleshooting.

8.2.1 log4j debugging

If client-side API logging is enabled using the steps in 8.1, "Logging" on page 306, but no logs are generated, these additional steps can help further isolate the issue.

When no logging is configured for the CE Java API, a warning message is written to the `System.err` file. See Example 8-7.

Example 8-7   Message when log4j is not configured

```
log4j:WARN The log4j system is not properly configured!
log4j:WARN All ERROR messages will be sent to the system console until proper
configuration has been detected
```

If you do not see these lines, most likely log4j is configured but the `log4j.properties` file is being picked up in a different location. This can occur when a custom application uses its own packaging of log4j.

To determine that information, you can enable log4j's debug mode by setting the following JVM parameter:

`-Dlog4j.debug=true`

This setting generates output in `System.out` similar to Example 8-8.

Example 8-8   log4j debug output

```
log4j: Trying to find [log4j.xml] using context classloader
sun.misc.Launcher$AppClassLoader@a39137.
log4j: Trying to find [log4j.xml] using sun.misc.Launcher$AppClassLoader@a39137
class loader.
log4j: Trying to find [log4j.xml] using ClassLoader.getSystemResource().
log4j: Trying to find [log4j.properties] using context classloader
sun.misc.Launcher$AppClassLoader@a39137.
log4j: Using URL [file:/C:/classpath/log4j.properties] for automatic log4j
configuration.
log4j: Reading configuration from URL file:/C:/whatever/log4j.properties ...
```

8.2.2 Content Engine troubleshooting techniques

The following technical articles provide tips about programming with FileNet P8 CE APIs and troubleshooting techniques:

► *Writing great code with the IBM FileNet P8 APIs, Part 1: Hello, Document! Getting started with your first FileNet P8 program*

  http://www.ibm.com/developerworks/data/library/techarticle/dm-0810carpenter

► *Writing great code with the IBM FileNet P8 APIs, Part 2: Spying on your handiwork Techniques for seeing what you're putting on the wire*

  http://www.ibm.com/developerworks/data/library/techarticle/dm-0902carpenter

8.2.3 Process Engine troubleshooting techniques

The following article discusses various techniques to troubleshoot PE to CE connectivity issues:

http://www.ibm.com/support/docview.wss?uid=swg21328045

8.2.4 Data must be gathered for troubleshooting

When you report FileNet programming issues to IBM FileNet support, obtain the necessary information as described in the following technical articles:

► Content Engine 4.5

http://www.ibm.com/support/docview.wss?uid=swg21308231

► Process Engine 4.5

http://www.ibm.com/support/docview.wss?uid=swg21327304

A

Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

`ftp://www.redbooks.ibm.com/redbooks/`SG247743

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** (and then **Access additional materials**) and open the directory that corresponds with the IBM Redbooks form number, SG247743.

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name | Description
SG247743-sample.zip | Code for the sample application used in the book (the file is compressed in .zip file format)

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 10 GB minimum
Operating System: Microsoft Windows XP
Processor: Intel® Core Duo processor
Memory: 1 GB minimum

How to use the Web material

Create a subdirectory (folder) on your workstation, and extract the contents of the Web material .zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see "How to get Redbooks" on page 318. Note that some of the documents referenced here may be available in softcopy only.

► *IBM FileNet Content Manager Implementation Best Practices and Recommendations,* SG24-7547

► *Introducing IBM FileNet Business Process Manager*, SG24-7509

► *IBM FileNet P8 Platform and Architecture,* SG24-7667

► *Understanding IBM FileNet Records Manager,* SG24-7623

► *IBM Content Manager OnDemand Web Enablement Kit Java APIs: The Basics and Beyond,* SG24-7646

Online resources

These Web sites are also relevant as further information sources:

► IBM FileNet P8 Platform main information page

http://www.ibm.com/software/data/content-management/filenet-p8-platform

► IBM FileNet P8 Platform product documentation

http://www.ibm.com/support/docview.wss?rs=3247&uid=swg27010422

The above URL includes links to all expansion IBM FileNet P8 products.

► IBM FileNet Content Manager

http://www.ibm.com/software/data/content-management/filenet-content-manager

► IBM FileNet Business Process Manager

http://www.ibm.com/software/data/content-management/filenet-business-process-manager

► IBM InfoSphere Enterprise Records

http://www.ibm.com/software/data/content-management/filenet-records-manager/

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

.NET
    environment setup   29
.NET API   3–4, 40, 48, 55, 270, 306, 310

A

access   88
access control entry   88, 108, 257
access control list (ACL)   88, 258
access permission
    manipulation in C#   89
    manipulation in Java   88
accessing Document content
    in C#   63
    in Java   62
AccessPermission   88
AccessRight.DELETE
    permission   88–89
active content   102
Admin package
    CE API   41
administration API
    API
        administration related   113
advanced document recognition (ADR)   9
annotation   99–100
annotations property   94
API   1, 3–4, 13, 39–41, 80, 88, 94, 96, 111–114
    .NET   3–4, 40, 48, 55, 270
    .NET API   306, 310
    CE .NET API, installing   30
    CE API, patch level,   308
    CE Java   268, 273
    CE Java API   306–308
    CE Java API code   306
    CE Java API log   308
    CE Java API log4j environment   306
    CE Java API trace file   308
    COM Compatibility Layer   6–7
    configuration   113
    eForms   8
    exception   123
    Java   3–4, 6–7, 40, 44, 48, 104, 112–114, 124,
        126, 132, 188–189, 192
    Java Compatibility Layer, Content Engine (CE)
        6
    Java, Content Engine   14
    Java, IBM InfoSphere Enterprise Records   8
    Java, Process Engine   7, 114, 121, 124
    log file, IBM FileNet   305
    PE API   269, 293, 297
    PE API classes, functional relationship   118
    PE API, core classes   116
    PE error handling class   117
    PE REST API   112–113, 140, 144, 150
    PE REST API, scenario   188
    PE REST API, service   207, 215
    PE retrieve work status classes   117
    PE search work classes   117
    PEWS API   114
    Process Engine (PE)   114–115, 119
    REST API   7, 112, 114, 123, 188, 190, 193
    REST Services, Process Engine   7
    workflow definition related   113
API set   4, 6–7
Application Engine (AE)   2, 9, 100, 168, 312
application environment   299
application programming interface
    See API
application server   10, 44–45, 48, 119–120, 123,
127, 142, 146, 176, 188, 191, 276, 285, 289, 306,
308, 311
    container   154, 160, 193
    documentation   261, 292, 296
application space   187
architecture
    ECM Widgets   205
    sample applications   268
association property   94
Attachment widget   202, 205, 208
authentication
    in .NET   45
    in Java   44
    PE   120
authorization   88
    accessing REST operations   123

B

batch
    refresh versus no refresh   81
    retrieval   81
    retrieving items in batch   83
    transactional operation   81
    updates   81
    updating objects in batch   82
Billing Report application   224, 237–239
    architecture   274
    user view   237
binary   279
BOOTSTRAP_ADDRESS   17
building SQL
    using IBM FileNet Enterprise Manager   72

C

cache, metadata   106
callback   103, 127–128, 142–143, 179, 189, 191, 194
cascading style sheet (CSS)   97, 211
CE   2–5, 13, 39–40, 72, 87, 119, 121, 168, 176, 239, 263–264, 272, 275, 305
    API class model   40
    gather information for troubleshooting   314
    Java API   4, 7, 14
    Java Compatibility Layer API   6
    Query Builder   73
    sample Java application setup   22
    URI   120, 125
    Web service   5
    WSDL   5
CE .NET API
    installing   30
CE API
    class model   40
    Core package   41
    Events package   42
    patch level   308
    type   40
CE Java API   268, 273, 306–308
    code   306
    log   308
    log4j environment   306
    trace file   308
CE_Operations   176, 263
CEWS   31, 306
CEWS transport   6, 14, 21, 23, 45–46, 294,

298–299, 310
    requirements   18–19
CFS-IS   9, 100
check in
    document   90
    document in C#   93
    document in Java   92
check out
    document in C#   93
    document in Java   92
class model
    CE API   40
client
    thick versus thin   15
Collection package
    CE API   42
collection properties   54
COM Compatibility Layer
    API   6–7
    programming   7
component
    association   171
    document   97
    operation   171
    package   172
Component Manager   176, 184–186, 286, 290–291
    configuration   185
    parameters   185
component queue   168, 170–171, 182, 184, 263, 287–288
    configuring and testing   182
    programmatic creation   182
ComponentRelationship   97
compound document
    manipulation   97
configurable set   306, 311
configuration API   113
connection   43
    initial connection   43
    initial connection in C#   49
    initial connection with EJB transport in Java   47
    initial connection with Web services transport in Java   47
    making initial connection   44
connection point   119
connectivity
    troubleshooting   21
containment relationship   95
CONTAINS   76

local cache property 58
log file
   API 305
   size and rollover 307
   thread identifier 308
log4j environment 306
log4j.jar 14
log4j.properties 306–307, 313
login 176

M

maintenance employee 109–110, 233, 253, 257
major version 90, 92
manipulation of compound document 97
mashup container 202
mashup page 202, 204, 206
   widget 218
merged scope search 75
Meta package
   CE API 41
metadata 53
   cache 106
   discover at runtime 105
   object 41, 283
minor version 90
multi-page file 61
multi-page scanned document 61
multi-valued
   property 54
   property, setting 55
   property, setting in C# 56
   property, setting in Java 56
multi-valued object-valued property
   See MVOVP
MVOVP 94, 243, 245–246
My Work Page 202

N

Next widget 207, 210–211

O

object
   creating object 52
   creating object in C# 53
   creating object in Java 53
   deleting in C# 61
   deleting in Java 60

   deleting object 60
   fetching, in C# 59
   fetching, in Java 59
   retrieving 42, 58
object store 4, 43–44, 46, 72, 74, 104, 149, 151, 153, 168, 263, 272–273, 275, 288–289
   search cross object stores 75
ObjectChangeEvent event (ONEVENT) 104
object-valued property (OVP) 54, 94, 109, 243–244, 249
optimizing event log query 139
optimizing queue query 135
optimizing, queue 135
ORDER BY, merged scope search 75

P

p8_api_trace.log 307
paging support, result set 76
PE 13, 105, 111–113, 126, 132, 135, 167–168, 170, 187, 235, 270, 272, 305, 311
   authentication 120
   connection point 295
   exception handling 123–124
   gather information for troubleshooting 314
   Java API 7, 114, 121, 124
   Java development project 28
   Java development setup 27
   maintenance queue 175
   object 163
   REST API 112, 199
   REST Services API 7
   server 121
   session 119, 121, 125, 139
   troubleshooting 313
   Web service 7
   Web Service Policy object 122
   Web services 112
PE API 114–115, 119, 121, 124, 269, 293, 297
   application polling component queue 184
   core classes 116
   object 119
   object method 115
   other sets 112
   usage tip 119
PE API classes
   functional relationships 118
PE error handling class 117
PE query, optimizing 130

R

S

Developing Applications with IBM FileNet P8 APIs

Developing Applications with IBM FileNet P8 APIs

Content Engine basic and advanced APIs

Process Engine basic and advanced APIs

REST API, ECM Widgets, sample applications, and more

This IBM Redbooks publication can help you develop content and process management applications with IBM FileNet APIs. The IBM FileNet P8 suite of products contain a set of robust APIs that range from core platform APIs to supporting application APIs. This book focuses specifically on Content Engine and Process Engine APIs.

Content Engine API topics that we discuss include creating, retrieving, updating, and deleting objects; querying and viewing documents; and batching and batch execution. We also explore more complex topics, including permissions and authorization, versioning, relationships, annotations, workflow subscriptions and event actions, metadata discovery, and dynamic security inheritance.

Process Engine API topics that we discuss include launching a workflow, searching for and processing work items, and working with process status. The more complex topics we cover include, Component Integrator application space, role, workbasket, resource navigation in Process Engine REST API, ECM Widgets, and building a custom Get Next In-basket widget.

To help you better understand programming with IBM FileNet APIs, we provide a sample application implemented for a fictional company. We include the data model, security model, workflows, and various applications developed for the sample. You can download them for your reference.

This book is intended for IBM FileNet P8 application developers. We recommend using this book in conjunction with the online ECM help.