Sun Java™ System

Content Delivery Server 5.0 Customization Guide

2005Q4

Sun Microsystems, Inc.
www.sun.com

Part No.: 819-3213-10

Submit comments about this document at: http://docs.sun.com/app/docs/form/comments

Contents

Preface xxi

1. Introduction 1–1

1.1 Event Service API 1–1

1.2 Billing API 1–2

1.3 Content Management API 1–2

1.4 Content Validation API 1–3

1.5 User Profile API 1–3

1.6 WAP Gateway API 1–3

1.7 Messaging API 1–4

1.8 Confirm Service API 1–4

1.9 Subscriber API 1–4

1.10 API by Feature 1–5

2. Event Service API 2–1

2.1 SQL*Net Client Application 2–3

2.1.1 Event Tables 2–3

2.1.1.1 CDS_EVENT Table 2–4

2.1.1.2 CDS_EVENT_TYPE Table 2–5

2.1.1.3 CDS_EVENT_GROUP Table 2–5

2.1.1.4 EVENT_SOURCE_TYPE_ID Table 2–6

2.1.2 Reporting Tools 2–6

2.2 JMS Client Application 2–7

2.3 Events and Event Data 2–7

2.4 Using the Event Service API 2–12

2.4.1 Developing an SQL*Net Client Application 2–12

2.4.2 Developing a JMS Client Application 2–12

2.5 Sample Implementation of `MessageListener` 2–13

3. Billing API 3–1

3.1 General Process Flow 3–2

3.1.1 Content Listing 3–2

3.1.2 Transaction Initiation 3–3

3.1.3 Subscriber Purchase 3–4

3.1.4 Download Confirmation 3–5

3.1.5 Subscription Verification 3–6

3.1.6 Error Handling 3–7

3.2 `BillingManager` Interface 3–7

3.2.1 `authorize()` 3–8

3.2.2 `checkSubscription()` 3–9

3.2.3 `confirm()` 3–9

3.2.4 `getBillingInfo()` 3–9

3.2.5 `getBillingInfos()` 3–10

3.2.6 `getLog()` 3–11

3.2.7 `refund()` 3–11

3.2.8 `reverse()` 3–11

3.2.9 `subscribe()` 3–11

3.2.10 `unsubscribe()` 3–12

3.3 Using the Billing API 3–12

3.4 Sample Billing Adapter 3–13

4. Content Management API 4–1

4.1 General Process Flow 4–2

4.1.1 Obtaining a List of Content 4–2

4.1.2 Obtaining Content Details 4–3

4.1.3 Downloading Content 4–3

4.2 `ContentManager` Interface 4–4

4.2.1 `getContentInfo()` 4–5

4.2.2 `getContentInfos()` 4–5

4.2.3 `getContentDescriptor()` 4–5

4.2.4 `getContentBinary()` 4–6

4.3 Using the Content Management API 4–6

4.4 Sample Content Management Adapter 4–7

5. Content Validation API 5–1

5.1 General Process Flow 5–1

5.2 `ValidationAdapter` Class 5–2

5.2.1 `execute()` 5–2

5.2.2 `returns()` 5–3

5.3 `ValidationContent` Class 5–3

5.4 Using the Content Validation API 5–3

5.5 Sample Content Validation Adapter 5–5

6. User Profile API 6–1

6.1 `UserManager` Class 6–1

6.1.1 `doAccountExists()` 6–2

6.1.2 `doAddUser()` 6–2

6.1.3 `doDisableUser()` 6–2

6.1.4 `doEnableUser()` 6–2

6.1.5 doFormatMobileId() 6–2
6.1.6 doFormatLoginId() 6–2
6.1.7 doGetAllLikeInOrder() 6–3
6.1.8 doGetAllUsers() 6–3
6.1.9 doGetAllUsersContainingFirstName() 6–3
6.1.10 doGetAllUsersContainingId() 6–3
6.1.11 doGetAllUsersContainingLastName() 6–3
6.1.12 doGetAllUsersContainingName() 6–4
6.1.13 doGetAnonymousUser() 6–4
6.1.14 doGetFieldName() 6–4
6.1.15 doGetUser() 6–4
6.1.16 doGetUserByMobileId() 6–4
6.1.17 doGetUserByUniqueDeviceId() 6–4
6.1.18 doGetUserInstance() 6–5
6.1.19 doIsActive() 6–5
6.1.20 doIsAuthenticated() 6–5
6.1.21 doRemoveUser() 6–5
6.1.22 doUpdateUser() 6–5
6.2 User Interface 6–6
6.2.1 getActivateDate() 6–6
6.2.2 getAttribute() 6–6
6.2.3 getAttribute() 6–6
6.2.4 getAttributes() 6–6
6.2.5 getCity() 6–6
6.2.6 getCountry() 6–7
6.2.7 getCreateDate() 6–7
6.2.8 getDeActivateDate() 6–7
6.2.9 getEmail() 6–7

6.2.10 getFirstName() 6–7

6.2.11 getGender() 6–7

6.2.12 getLastLogin() 6–8

6.2.13 getLastName() 6–8

6.2.14 getLoginId() 6–8

6.2.15 getMiddleName() 6–8

6.2.16 getMobileId() 6–8

6.2.17 getPassword() 6–8

6.2.18 getPhone() 6–9

6.2.19 getPostalCode() 6–9

6.2.20 getSalutation() 6–9

6.2.21 getState() 6–9

6.2.22 getStreet1() 6–9

6.2.23 getStreet2() 6–9

6.2.24 getUniqueDeviceId() 6–10

6.2.25 hasLoggedIn() 6–10

6.2.26 isConfirmed() 6–10

6.2.27 isEnabled() 6–10

6.2.28 isPrepay() 6–10

6.2.29 setActivateDate() 6–10

6.2.30 setAttribute() 6–11

6.2.31 setAttributes() 6–11

6.2.32 setCity() 6–11

6.2.33 setCountry() 6–11

6.2.34 setCreateDate() 6–11

6.2.35 setDeActivateDate() 6–11

6.2.36 setEmail() 6–12

6.2.37 setFirstName() 6–12

6.2.38 `setGender()` 6–12

6.2.39 `setHasLoggedIn()` 6–12

6.2.40 `setIsEnabled()` 6–12

6.2.41 `setIsPrepay()` 6–12

6.2.42 `setLastName()` 6–13

6.2.43 `setLoginId()` 6–13

6.2.44 `setMiddleName()` 6–13

6.2.45 `setMobileId()` 6–13

6.2.46 `setPassword()` 6–13

6.2.47 `setPhone()` 6–13

6.2.48 `setPostalCode()` 6–14

6.2.49 `setSalutation()` 6–14

6.2.50 `setState()` 6–14

6.2.51 `setStreet1()` 6–14

6.2.52 `setStreet2()` 6–14

6.2.53 `setUniqueDeviceId()` 6–14

6.2.54 `updateLastLogin()` 6–15

6.3 `UserDeviceManager` Interface 6–15

6.3.1 `getUniqueDeviceID()` 6–15

6.4 Using the User Profile API 6–15

6.5 Sample Implementation of the User Manager API 6–16

6.5.1 Support Files 6–16

6.5.2 `SampleUserImpl.java` 6–20

6.5.3 `SampleUserManagerImpl.java` 6–27

7. WAP Gateway API 7–1

7.1 `WAPGatewayAdapter` Class 7–2

7.1.1 `doHandle()` 7–2

7.1.2 `getMSISDN()` 7–2

7.1.3 getUniqueId() 7–2

7.2 Using the WAP Gateway API 7–3

7.3 Sample WAP Gateway Adapter 7–3

8. Messaging API 8–1

8.1 PushMsgSender Interface 8–2

8.1.1 pushMessage() 8–2

8.2 PushMsgListener Interface 8–2

8.2.1 connect() 8–2

8.2.2 initialize() 8–3

8.2.3 listen() 8–3

8.2.4 sendKeepAliveMsg() 8–3

8.3 PushMessage Class 8–3

8.3.1 addUserAgent() 8–3

8.3.2 getAllUserAgents() 8–4

8.3.3 getAttribute() 8–4

8.3.4 getContentBinary() 8–4

8.3.5 getContentType() 8–4

8.3.6 getDestinationAddress() 8–4

8.3.7 getJMSMessageId() 8–4

8.3.8 getKeyword() 8–5

8.3.9 getMessageId() 8–5

8.3.10 getMessageText() 8–5

8.3.11 getMimeType() 8–5

8.3.12 getPushCategory() 8–5

8.3.13 getPushDomain() 8–5

8.3.14 getPushType() 8–6

8.3.15 getShortCode() 8–6

8.3.16 getSubscriberId() 8–6

8.3.17 getUniqueDeviceId() 8–6

8.3.18 getVendingContentId() 8–6

8.3.19 setAllUserAgents() 8–6

8.3.20 setAttribute() 8–7

8.3.21 setContentBinary() 8–7

8.3.22 setContentType() 8–7

8.3.23 setDestinationAddress() 8–7

8.3.24 setJMSMessageId() 8–7

8.3.25 setKeyword() 8–7

8.3.26 setMessageId() 8–8

8.3.27 setMessageText() 8–8

8.3.28 setMimeType() 8–8

8.3.29 setPushCategory() 8–8

8.3.30 setPushDomain() 8–8

8.3.31 setShortCode() 8–8

8.3.32 setSubscriberId() 8–9

8.3.33 setUniqueDeviceId() 8–9

8.3.34 setVendingContentId() 8–9

8.3.35 toString() 8–9

8.4 SMSMessage Class 8–9

8.5 WapPushMessage Class 8–10

8.6 SMTPMessage Class 8–10

8.7 ContentSlide Class 8–10

8.7.1 getContentData() 8–10

8.7.2 getContentId() 8–10

8.7.3 getContentMimeType() 8–11

8.7.4 setContentData() 8–11

8.7.5 setContentId() 8–11

8.7.6 setContentMimeType() 8–11

8.8 MMSSlide Class 8–11

8.8.1 getAudioContent() 8–11

8.8.2 getImageContent() 8–12

8.8.3 getTextContent() 8–12

8.8.4 getVideoContent() 8–12

8.8.5 setAudioContent() 8–12

8.8.6 setImageContent() 8–12

8.8.7 setTextContent() 8–12

8.8.8 setVideoContent() 8–13

8.9 MMSPushMessage Class 8–13

8.9.1 addMMSSlide() 8–13

8.9.2 addRecipient() 8–13

8.9.3 getAllMMSSlides() 8–13

8.9.4 getAllRecipients() 8–13

8.9.5 getDeliveryReportRequired() 8–14

8.9.6 getFromAddress() 8–14

8.9.7 getMessageClass() 8–14

8.9.8 getMessagePriority() 8–14

8.9.9 getReadReportRequired() 8–14

8.9.10 getSenderVisibility() 8–14

8.9.11 getSMILPresentation() 8–15

8.9.12 setDeliveryReportRequired() 8–15

8.9.13 setFromAddress() 8–15

8.9.14 setMessageClass() 8–15

8.9.15 setMessagePriority() 8–15

8.9.16 setReadReportRequired() 8–15

8.9.17 setSenderVisibility() 8–16

8.9.18 `setSMILPresentation()` 8–16

8.10 `MMSSender` Interface 8–16

8.10.1 `sendMMS()` 8–16

8.11 `PushResponse` Class 8–17

8.11.1 `getMessageId()` 8–18

8.11.2 `getResponseDescription()` 8–18

8.11.3 `getResponseStatus()` 8–18

8.12 `PushConstants` Class 8–18

8.13 Using the Messaging API 8–18

9. Confirm Service API 9–1

9.1 General Process Flow 9–1

9.2 `ConfirmServiceAdapter` Class 9–2

9.2.1 `connect()` 9–2

9.2.2 `listen()` 9–2

9.2.3 `messageReceived()` 9–3

9.3 Using the Confirm Service API 9–3

10. Subscriber API 10–1

10.1 General Process Flow 10–2

10.2 Using the Subscriber API 10–3

10.2.1 Managing Transactions 10–4

10.2.2 Example of Creating an `IApiContext` Object 10–4

10.2.3 Example of Creating a Service 10–5

10.3 XML-RPC Implementation 10–6

10.3.1 Accessing the Content Delivery Server 10–6

10.3.2 Using XML-RPC Handlers for the Subscriber API 10–7

10.3.2.1 Guidelines for Calls to XML-RPC Methods 10–7

10.3.2.2 `AuthenticationHandler` 10–9

10.3.2.3 `CategoryHandler` 10–10

10.3.2.4 `ContentHandler` 10–11

10.3.2.5 `DownloadHandler` 10–15

10.3.2.6 `GiftingHandler` 10–17

10.3.2.7 `MessageHandler` 10–18

10.3.2.8 `SystemHandler` 10–20

10.3.2.9 `UserHandler` 10–21

10.3.2.10 Parameters for the Methods 10–25

10.3.3 Examples of Using Handlers 10–42

10.3.3.1 Example of Creating an `ApiContext` Object 10–43

10.3.3.2 Example of Creating a Handler and Purchasing Content 10–44

Index Index–1

Figures

FIGURE 2-1	Event Service Overview 2–2
FIGURE 3-1	Process Flow for Content Listing 3–2
FIGURE 3-2	Process Flow for Transaction Initiation 3–3
FIGURE 3-3	Process Flow for Subscriber Purchase 3–4
FIGURE 3-4	Process Flow for Download Confirmation 3–5
FIGURE 3-5	Process Flow for Subscription Verification 3–6
FIGURE 7-1	WAP Gateway Adapter Architecture 7–1
FIGURE 8-1	Architecture of the Messaging API 8–1
FIGURE 8-2	Process Flow for Sending an MMS Message 8–17

Tables

TABLE 1-1 Mapping of Feature to Content Delivery Server API 1–5

TABLE 2-1 CDS_EVENT Table 2–4

TABLE 2-2 CDS_EVENT_TYPE Table 2–5

TABLE 2-3 CDS_EVENT_GROUP Table 2–5

TABLE 2-4 EVENT_SOURCE_TYPE_ID Table 2–6

TABLE 2-5 Events 2–7

TABLE 2-6 Event Data 2–9

TABLE 2-7 Files Required for Execution 2–13

TABLE 9-1 `ConfirmResponse` Parameters 9–2

TABLE 10-1 Methods for `AuthenticationHandler` 10–9

TABLE 10-2 Methods for `CategoryHandler` 10–10

TABLE 10-3 Methods for `ContentHandler` 10–12

TABLE 10-4 Methods for `DownloadHandler` 10–16

TABLE 10-5 Methods for `GiftingHandler` 10–17

TABLE 10-6 Methods for `MessageHandler` 10–19

TABLE 10-7 Methods for `SystemHandler` 10–20

TABLE 10-8 Methods for `UserHandler` 10–22

TABLE 10-9 Method Parameters 10–25

Code Samples

CODE EXAMPLE 2-1	Sample `MessageListener` Implementation 2-13
CODE EXAMPLE 3-1	Sample `BillingManager` Implementation 3-13
CODE EXAMPLE 4-1	Sample `ContentManager` Implementation 4-7
CODE EXAMPLE 5-1	Sample `ValidationAdapter` Implementation 5-5
CODE EXAMPLE 6-1	Sample Property File 6-16
CODE EXAMPLE 6-2	Sample External Proxy 6-16
CODE EXAMPLE 6-3	Sample `User` Implementation 6-20
CODE EXAMPLE 6-4	Example Using the `UserManager` Class 6-27
CODE EXAMPLE 7-1	Example Using the `WAPGatewayAdapter` Class 7-3
CODE EXAMPLE 8-1	Sample `pushsenderfactory.xml` File 8-19
CODE EXAMPLE 8-2	Sample `pushlistenerfactory.xml` File 8-19
CODE EXAMPLE 10-1	Create an `IApiContext` Object Using Java Classes 10-4
CODE EXAMPLE 10-2	Create a Service 10-5
CODE EXAMPLE 10-3	Create an `APIContext` Object Using XML-RPC 10-43
CODE EXAMPLE 10-4	Create a Handler 10-44

Preface

The *Sun Java™ System Content Delivery Server Customization Guide* describes each of the Content Delivery Server application programming interfaces (APIs.) These APIs are used for integrating the Content Delivery Server with an existing infrastructure.

Before You Read This Document

This guide is for programmers who are responsible for writing adapters based on the APIs and system administrators who are responsible for integrating the Sun Java System Content Delivery Server with their existing infrastructure. It assumes some knowledge of the Java programming language and networking, database, and web technologies.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document, and does not endorse and is not responsible or liable for any content, advertising, products, or other materials available through such sites.

How this Document is Organized

- Chapter 1 provides an overview of the Content Delivery Server APIs.
- Chapter 2 describes the Event Service API, which provides asynchronous reporting of billable events.
- Chapter 3 describes the Billing API, which provides an interface between the Content Delivery Server and your billing system.

- Chapter 4 describes the Content Management API, which is used to manage the interface between the Content Delivery Server and your content management system.
- Chapter 5 describes the Content Validation API, which is used to validate and protect content that is submitted to the Content Delivery Server.
- Chapter 6 describes the User Profile API, which is used to add, delete, retrieve, update, enable, and disable users in the system.
- Chapter 7 describes the WAP Gateway API, which retrieves the MSISDN, device profile, and other attributes from the HTTP header.
- Chapter 8 describes the Messaging API, which provides a mechanism for carriers or application vendors to integrate their own Wireless Access Protocol (WAP), Short Message Service (SMS), and Multimedia Messaging Service (MMS) push implementations.
- Chapter 9 enables the Content Delivery Server to handle confirmation messages sent from a Multimedia Messaging Service Center (MMSC).
- Chapter 10 provides access to data maintained by the Content Delivery Server.

Typographic Conventions

Typeface[a]	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Replace command-line variables with real names or values.	To delete a file, type `rm` *filename*.

a The settings on your browser might differ from these settings.

Related Documentation

The Sun Java System Content Delivery Server manuals are available as Portable Document Format (PDF) and Hypertext Markup Language (HTML) files. These files are available in the `Documentation` subdirectory of the directory where the Content Delivery Server is installed as well as online at `http://docs.sun.com`.

The following table summarizes the books included in the Content Delivery Server documentation set.

Book Title	Description	Part Number
Sun Java System Content Delivery Server Administration Guide	Describes how to manage content, devices, and access to the Content Delivery Server.	819-3209-10
Sun Java System Content Delivery Server Branding and Localization Guide	Describes how to customize the Subscriber Portal and Developer Portal components of the Content Delivery Server for the look and feel of your enterprise. This guide also describes how to localize the Content Delivery Server interfaces.	819-3210-10
Sun Java System Content Delivery Server Capacity Planning Guide	Provides guidelines for determining what hardware and software is needed to efficiently run the Content Delivery Server.	819-3211-10
Sun Java System Content Delivery Server Content Developer Guide	Describes how to submit content to the Content Delivery Server.	819-3212-10
Sun Java System Content Delivery Server Error Messages	Describes error messages that are generated by the Content Delivery Server and suggests actions to take to resolve problems reported.	819-3214-10
Sun Java System Content Delivery Server Installation Guide	Provides information about installing and configuring the Content Delivery Server.	819-3215-10
Sun Java System Content Delivery Server Integration Guide	Describes adapters for integrating the Content Delivery Server with existing systems such as billing, user data, WAP gateway, and push delivery. It also describes the framework for creating device-specific versions of the Subscriber Portal.	819-3216-10
Sun Java System Content Delivery Server Migration Guide	Describes how to migrate from the previous version of the Content Delivery Server to the current version.	819-3217-10
Sun Java System Content Delivery Server System Management Guide	Provides information on running and maintaining the Content Delivery Server.	819-3218-10

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions.

To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the document title and part number.

CHAPTER 1

Introduction

To deploy a commercial service, operators must integrate the Sun Java System Content Delivery Server with their existing infrastructure. Billing systems, user management systems, and reporting systems each have integration requirements, and involve levels of complexity for the operator. The Sun Java System Content Delivery Server external APIs help with this system integration.

This section introduces the following APIs:

- Event Service API
- Billing API
- Content Management API
- Content Validation API
- User Profile API
- WAP Gateway API
- Messaging API
- Confirm Service API
- Subscriber API

Section 1.10, "API by Feature" on page 1-5 identifies by feature the API that is used to integrate the Content Delivery Server with your existing systems.

1.1 Event Service API

The Event Service API provides asynchronous reporting of billable events and enables external systems to extract or be notified of these events. The Event Service API consumes all of the messages published by the manager components. For each message, the context and details of the message are extracted, inserted into the database, and broadcast for reporting and billing.

The transfer of billing data can occur in both directions, and data transfer can happen in real time or in batch mode. For example, the Content Delivery Server can push data to the operator's billing system as the usage data is recorded or use a batch pull to query the server periodically. The usage data exchanged between components is in XML format and includes pricing information that can be captured for billing integration. The Content Delivery Server can interact with any billing system through this API.

1.2 Billing API

The Billing API supports prepaid and synchronous billing models. For billing systems that support the collection of fees prior to content being purchased, the Billing API is used to verify that subscribers have enough funds in their account before allowing subscribers to download content. For billing systems that require real-time billing, the Billing API is used to charge a subscriber's account at the time content is purchased.

The Billing API also enables an operator to charge a different fee for content than what is specified in the Vending Manager. For example, if the operator wants to provide business subscribers with a special discount, the Billing API is used to retrieve the special price from the billing system and show that price to the customer.

1.3 Content Management API

The Content Management API enables content to be instrumented at the time the content is downloaded to a subscriber's device. This API is used when the content needs to be instrumented with either user-specific or billing-specific data immediately before being delivered to the subscriber. The Content Management API can also be used to change the attributes of the content, such as MIME type or content type.

1.4 Content Validation API

The Content Validation API processes content that is submitted to the Content Delivery Server. Use this API to create the content validation adapters that you need to automatically validate submitted content. Content validation adapters are used by the submission verifier workflow that is executed for content submitted to the Content Delivery Server.

1.5 User Profile API

The User Profile API provides an interface into existing subscriber databases. It integrates with data sources for subscribers, application developers, administrators, and devices. Operators do not need to create new, separate databases to work with the Content Delivery Server. If operators need to combine or merge multiple database entries, the User Profile API helps integrate this data without impacting other services requesting information. It also helps with legacy information integration.

Through the User Profile API, the Content Delivery Server provides a common service layer for all components to access the database. The abstraction of data access frees the Content Delivery Server from being tied to a particular database. This service provides scalable data access to the underlying data by caching frequently requested data. Combined with security, data access can be controlled across different users or administrators.

1.6 WAP Gateway API

The WAP Gateway API handles implementation differences in mobile identity number (MSISDN) authentication and header transmission. Using the MSISDN for authentication and session management provides benefit for the subscriber and operator. The Content Delivery Server supports single sign-on, so if the subscriber is accessing the system through a cell phone, the authentication is performed through the network and the subscriber's user name and password are not needed. The Content Delivery Server can be configured to communicate concurrently with multiple gateway types.

The Content Delivery Server works with a standard HTTP web server to handle presentation logic. It serves both HTML content for the subscriber web site and Administrator Console and WML for device-side access through a WAP 1.*x* compatible browser. The Content Delivery Server supports secure sockets layer (SSL) connections for secure communication.

1.7 Messaging API

The Messaging API provides a mechanism for carriers or application vendors to integrate their own WAP or Short Message Service (SMS) push implementations by providing an adapter. The Content Delivery Server also provides default WAP and SMS push implementations that can be used in most cases.

1.8 Confirm Service API

The Confirm Service API enables the Content Delivery Server to handle confirmation messages sent from a Multimedia Messaging Service Center (MMSC). Confirmation messages are generally sent after content is downloaded to a device. This API is used to create the connection to the MMSC and to monitor the messages that the MMSC sends.

1.9 Subscriber API

The Subscriber API provides access to data maintained by the Content Delivery Server. Use this API to get the data needed to create a client application for providing subscribers with access to content managed by the Content Delivery Server. The Subscriber API can be accessed directly by local Java technology-based applications (Java applications) or from remote applications or applications written in a programming language other than Java by using XML-RPC.

1.10 API by Feature

The following table identifies which API to use to integrate some of the Content Delivery Server features with your existing systems. For the features that you want to support in your installation of the Content Delivery Server, you must implement the API specified.

TABLE 1-1 Mapping of Feature to Content Delivery Server API

Feature	System	API
Buy for a friend	Simple Mail Transfer Protocol (SMTP), Short Message Service Center (SMSC), Push Proxy Gateway (PPG), and Multimedia Message Service Center (MMSC)	Messaging API
Tell a friend	SMTP, SMSC, PPG, and MMSC	Messaging API
Event-driven campaign[1]	SMTP, SMSC, PPG, and MMSC	Messaging API
Outbound campaign[1]	SMTP, SMSC, PPG, and MMSC	Messaging API
Confirm push message	SMTP, SMSC, PPG, and MMSC (service must return a confirmation message)	Confirm Service API
External user database	Lightweight Directory Access Protocol (LDAP)	User Profile API
Billing - prepaid	Prepaid billing system	Billing API
Billing - postpaid	Postpaid billing system	Event Service API
Integration with external digital rights management (DRM)	External DRM engine	Content Validation API, Content Management API
Single sign-on and device login	WAP gateway (must pass MSIDN or unique ID)	WAP Gateway API
Portal integration	Web portal	Subscriber API

[1]Promotional campaigns do not require integration with external systems

CHAPTER 2

Event Service API

This chapter describes the Sun Java System Content Delivery Server Event Service API. The Event Service API consists of the following external interfaces:

- A database schema for querying the event data directly from SQL*Net client applications
- The Java Message Service (JMS) topics available for subscription by JMS client applications

The Event Service propagates messages that come through the Event Queue to any interested event listener that has subscribed to the topics published by the Event Service. The Event Service also stores the event data in the Content Delivery Server database. You can use either the SQL*Net client or the JMS client application approach. Both approaches have access to the same information in near real time.

Use the Event Service API to write an application that listens for specific events and takes action as needed by your enterprise. For example, this API can be used to write a custom billing adapter that processes purchase events and charges subscribers after content is purchased.

FIGURE 2-1 is a simplified representation of the general systems and components that interact with the Event Service, which illustrates how information is passed through the various system interactions.

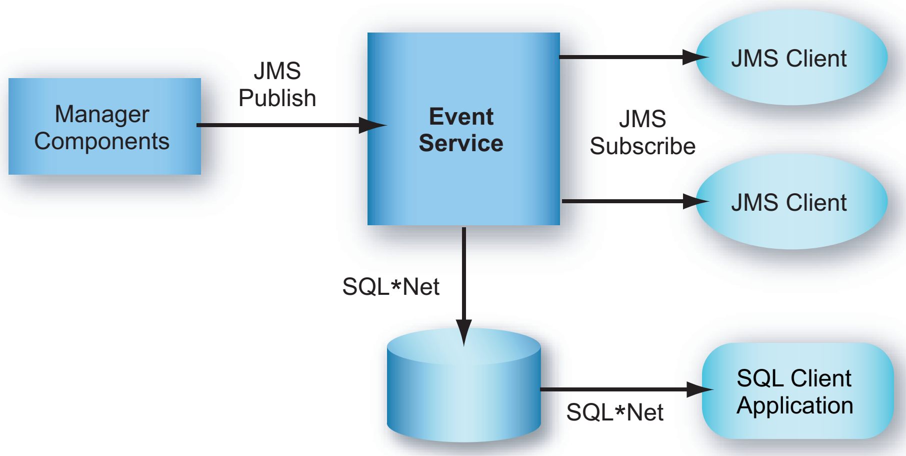

FIGURE 2-1 Event Service Overview

The Content Delivery Server Catalog Manager and Vending Manager components publish appropriate messages to a JMS queue. These messages are retrieved and processed by the Event Service. Any component within the Content Delivery Server can publish events to the Event Service. Because event publishing is an asynchronous operation, the publishing component continues processing as soon as the message is sent.

The Event Service runs as a separate process within the Content Delivery Server environment. The Event Service performs the following tasks:

- Consumes all of the messages published by the various server components
- Extracts the context and details of the message and inserts the information into the database
- Sends an acknowledgement to the JMS queue upon successful processing of the message
- If an error occurs, puts the message into an error queue

  The message can then be processed and possibly resubmitted to the Event Queue. The message is still acknowledged as successfully processed by the Event Service.

The current implementation of the Event Service API uses JMS Point-to-Point (PTP) messaging domain, where various publishers publish to a single queue and an instance of the Event Service JMS client application processes the messages.

After the events are successfully processed by the Event Service, they are placed into the database. When the Event Service is deployed with a Vending Manager, events are placed in the Vending Manager database schema. When the Event Service is deployed with only a Catalog Manager, the events are placed in the Catalog Manager database schema. The data model used to store these messages is described in Section 2.1.1, "Event Tables" on page 2-3.

2.1 SQL*Net Client Application

Data resides in a standard relational database. A set of views enables developers to write applications in any programming language that can query an Oracle database. For example, these applications can be written in the Java programming language, C++, or Visual Basic.

2.1.1 Event Tables

This section describes the tables that hold the event data after the event is processed by the Event Service. As an integrator, you have access to the data in these tables.

- If the Event Service is deployed with a Vending Manager, connect to the Oracle database using the user *prefix*_`vs_app`. *prefix* is the value specified for the Prefix element under the Vending element in the database configuration file used to create the database. Use the password that was specified for the Password element under the Vending element.
- If the Event Service is deployed with only a Catalog Manager, connect to the Oracle database using the user *prefix*_`ps_app`. *prefix* is the value specified for the Prefix element under the Catalog element in the database configuration file used to create the database. Use the password that was specified for the Password element under the Catalog element.

See the *Sun Java System Content Delivery Server Installation Guide* for information on the database configuration file.

2.1.1.1 CDS_EVENT Table

The CDS_EVENT table holds a record for each event processed successfully by the Event Service. This table is actively updated as events are processed.

TABLE 2-1 CDS_EVENT Table

Column Name	Data Type	Description
`CDS_EVENT_ID`	`NUMBER(18)`	unique, system-generated number used as the record ID. This field is the table's primary key.
`CDS_EVENT_DATE`	`DATE`	Timestamp indicating when the event message was generated.
`CDS_EVENT_TYPE_ID`	`NUMBER(18)`	Foreign key to the CDS_EVENT_TYPE table.
`EVENT_SOURCE_ID`	`NUMBER(18)`	Foreign key to the EVENT_SOURCE_TYPE table.
`SVR_INSTANCE_ID`	`NUMBER(18)`	System data.
`SVR_SESSION_ID`	`VARCHAR2(128)`	System data.
`SUB_SYSTEM_ID`	`VARCHAR2(80)`	The MSISDN associated with the session and event. If the MSISDN is not known, this column is null.
`CDS_USER_ID`	`NUMBER(18)`	The Content Delivery Server User ID associated with the session and event. If a user is not logged in, this column might be null.
`VENDOR_ID`	`NUMBER(18)`	Foreign key to the Vendor table.
`CONTENT_ID`	`NUMBER(18)`	Foreign key to the Content table.
`RAW_EVENT_MESSAGE`	`CLOB`	Raw event message XML.
`CREATE_DATE`	`DATE`	System data.
`MOD_DATE`	`DATE`	System data.
`LOCK_VERSION`	`NUMBER(1)`	System data.

2.1.1.2 CDS_EVENT_TYPE Table

The CDS_EVENT_TYPE table is a static table that contains Event Type definitions. These definitions are listed in Section 2.3, "Events and Event Data" on page 2-7.

TABLE 2-2 CDS_EVENT_TYPE Table

Column Name	Data Type	Description
`CDS_EVENT_TYPE_ID`	`NUMBER(18)`	Unique, system-generated number used as the record ID. This field is the table's primary key.
`CDS_EVENT_GROUP_ID`	`NUMBER(18)`	Foreign key to the `EVENT_GROUP` table. Used to group Event Types.
`CDS_EVENT_TYPE_NAME`	`VARCHAR2(80)`	Human-readable name.
`DESCRIPTION`	`VARCHAR2(80)`	Description of the particular Event Type.
`LONG_DESCRIPTION`	`VARCHAR2(255)`	Long description of the Event Type, if needed.
`IS_ACTIVE`	`NUMBER(1)`	Flag that indicates if this is an active event type.
`CREATE_DATE`	`DATE`	System data.
`MOD_DATE`	`DATE`	System data.
`LOCK_VERSION`	`NUMBER(1)`	System data.

2.1.1.3 CDS_EVENT_GROUP Table

The CDS_EVENT_GROUP table is a static table that maintains the definitions of all of the event groups in the system. All events belong to the event group called `cds_group`.

TABLE 2-3 CDS_EVENT_GROUP Table

Column Name	Data Type	Description
`CDS_EVENT_GROUP_ID`	`NUMBER(18)`	Unique, system-generated number used as the record ID. This field is the table's primary key.
`CDS_EVENT_GROUP_NAME`	`VARCHAR2(255)`	Human-readable name.
`DESCRIPTION`	`VARCHAR2(1024)`	Description of the particular Event Group.

TABLE 2-3 CDS_EVENT_GROUP Table *(Continued)*

Column Name	Data Type	Description
LONG_DESCRIPTION	VARCHAR2(2048)	Long description of the Event Group, if needed.
PARENT_GROUP_ID	NUMBER(18)	Enables group hierarchies.
IS_ACTIVE	NUMBER(1)	Flag that indicates if this is an active event group.
CREATE_DATE	DATE	System data.
MOD_DATE	DATE	System data.
LOCK_VERSION	NUMBER(1)	System data.

2.1.1.4 EVENT_SOURCE_TYPE_ID Table

The EVENT_SOURCE_TYPE_ID table is a static table that contains Event Source Type definitions.

TABLE 2-4 EVENT_SOURCE_TYPE_ID Table

Column Name	Data Type	Description
EVENT_SOURCE_TYPE_ID	NUMBER(18)	Unique, system-generated number used as the record ID. This field is the table's primary key.
EVENT_SOURCE_TITLE	VARCHAR2(255)	Human-readable name.
CREATE_DATE	DATE	System data.
MOD_DATE	DATE	System data.
LOCK_VERSION	NUMBER(1)	System data.

2.1.2 Reporting Tools

You can use any database reporting tool that connects to an Oracle database to build various types of reports from this data such as Crystal Reports. See Appendix B, "Reports," in the *Sun Java System Content Delivery Server Installation Guide* for additional information on report data.

2.2 JMS Client Application

In addition to the SQL-based interface to the event data, you can implement a JMS client application that interfaces directly with the Event Service through a JMS topic. While more difficult to implement, this approach provides greater flexibility for using the events generated by the Content Delivery Server.

To successfully use this API to integrate with the Event Service, you need to be familiar with writing JMS client applications and understand the Publish/Subscribe Messaging domain described in the JMS specification for the Java 2 Platform, Enterprise Edition (J2EE^{TM} platform). Any number of JMS client applications can subscribe to the messages published by the Event Service by using the Publish/Subscribe messaging model.

2.3 Events and Event Data

This section provides information on the events and the event data provided by the Event Service. Both the SQL*Net and JMS client applications use this information to filter and process the events. Your client application can handle each type of event as needed by your enterprise. For example, premium SMS billing can be performed when the `sms_content_push_sent` event is received.

The following table describes the events generated by the Event Service.

TABLE 2-5 Events

Event	**Description**
`content_changed`	A Catalog Manager administrator made a change to the content.
`content_purchased`	A subscriber purchased an item of content or downloaded a free item.
`content_refunded`	A refund was issued for an item of content.
`download_deleted`	Downloaded content was deleted from a device.
`download_error`	The device indicated that an error occurred when content was downloaded.
`download_initiated`	A subscriber started downloading content.
`download_install_notified`	A device confirmed that the download was successful.

TABLE 2-5 Events *(Continued)*

Event	Description
`external_content_updated`	Externally hosted content was updated.
`gift_cancelled`	A gift subscription was cancelled.
`gift_download_confirm`	A gift was downloaded by the recipient.
`gift_download_deleted`	A gift that was downloaded by the recipient was deleted.
`gift_download_error`	An error occurred when a gift was downloaded.
`gift_download_initiated`	The recipient of a gift started downloading it.
`gift_expired`	The gift expired.
`gift_purchased`	An item of content was purchased as a gift.
`gift_refunded`	A refund was issued for a gift purchased by the subscriber.
`gift_subscription_purchased`	A subscription to an item of content was purchased as a gift.
`gift_usage_purchased`	A number of uses of an item of content was purchased as a gift.
`mms_push_sent`	An MMS message was sent.
`pricing_changed`	The pricing of one or more items of content was changed using the Category Price Edit feature.
`sms_content_push_sent`	Binary content was sent in an SMS message. Use this event to trigger premium SMS billing.
`sms_push_sent`	An SMS message was sent.
`sms_received`	An SMS message was received.
`smtp_push_sent`	An SMTP message was sent.
`status_changed_to_deleted`	The status of an item of content was changed to Deleted.
`status_changed_to_denied`	The status of an item of content was changed to Denied.
`status_changed_to_new`	The status of an item of content was changed to New.
`status_changed_to_pending`	The status of an item of content was changed to Pending.
`status_changed_to_published`	The status of an item of content was changed to Published.
`status_changed_to_testing`	The status of an item of content was changed to Testing.

TABLE 2-5 Events *(Continued)*

Event	Description
submission_failed	Submitted content was rejected by the Content Delivery Server.
submission_successful	Submitted content was accepted by the Content Delivery Server.
subscriber_registered	A subscriber successfully registered.
subscription_cancelled	A subscription for an item of content was cancelled.
subscription_purchased	A subscription for an item of content was purchased.
usage_purchased	A number of uses were purchased for an item of content.
validation_passed	Submitted content was successfully processed by the submission verifier workflow.
validation_failed	Submitted content failed a step in the submission verifier workflow.
wap_push_sent	A WAP message was sent.

The following table lists information that can be included with an event. Each event contains only parameters that are relevant to that event.

TABLE 2-6 Event Data

Parameter	Data Type	Description
billing-ticket	String	The billing ticket for this transaction.
campaign_coupon	String	The coupon code for a campaign.
campaign_id	String	The unique identifier for the campaign.
catalog-res-id	String	The unique identifier for the content edition.
content_binary_mimetype	String	The MIME type of the content.
content_class_id	String	The unique identifier for the content item.
content_description	String	The long description of the content.
content_drm_type_id	String	The string that identifies the DRM method used to protect the content.
content_short_description	String	The short description of the content.
content-id	String	The unique identifier for the content that was purchased. This value is the same as catalog-res-id.
content_name	String	The name of the content.
current-status	String	The current status of this transaction.

TABLE 2-6 Event Data *(Continued)*

Parameter	Data Type	Description
`date`	Date	The date on which the transaction occurred.
`destination-address`	String	The address to which content is sent, for example, the MSISDN of the subscriber who requested content.
`developer-content-id`	String	The unique identifier used by the developer to identify the content.
`developer-id`	String	The unique identifier for the developer of the content.
`developer_name`	String	The name of the developer who submitted the content.
`download-confirm`	Boolean	The flag that indicates if a confirmation is required after a successful download.
`download-count`	Integer	The number of times the content can be downloaded for the price paid.
`download-current-count`	Integer	The number of times the subscriber downloaded this content, including this time.
`download-expiration`	Boolean	The flag that indicates if the download period is expired.
`download-period`	Integer	The time period during which the content can be downloaded without additional charge to the subscriber.
`download-price`	Float	The price of the content purchased.
`download-purchase`	Boolean	The flag that indicates this is a purchase request.
`download-recurring`	Boolean	The flag that indicates if the subscriber is charged for each download.
`event-log`	String	The name of the event log.
`event-msg`	String	The message issued with the event.
`event-source-type-id`	Integer	The number that identifies the source of the event.
`event-type`	Integer	The numeric representation of the event that occurred.
`event-type-id`	String	The type of event that occurred.
`external_content_id`	String	The tag used by the billing system to identify content.
`external_group_id`	String	The tag used by the billing system to identify the group to which the content belongs.
`external-request-text`	String	The text of the request from the subscriber, for example, the MO push request content.
`gift_message`	String	The message included with the gift.
`gifted_current_downloads`	Integer	The number of times the recipient downloaded this gift, including this time.

TABLE 2-6 Event Data *(Continued)*

Parameter	Data Type	Description
`gifted_current_subscriptions`	Integer	The number of subscription periods used by the recipient, including this period.
`gift_download_date`	Date	The date that the gift was first downloaded by the recipient.
`gift_expiration_date`	Date	The date by which the gift must be claimed by the recipient.
`gift_purchase_date`	Date	The date the gift was purchased by the giver.
`gifted_downloads`	Integer	The number of downloads included in the gift.
`gifted_subscriptions`	Integer	The number of subscription periods included in the gift.
`is_on_device`	Boolean	The flag that indicates if the content is already on the device.
`is-prepay`	Boolean	The flag that indicates if the subscriber prepaid for the content.
`locale`	String	The subscriber's locale.
`msisdn`	String	The MSISDN for the subscriber device.
`push-msgtext`	String	The message sent to the subscriber's device or email.
`recipient_locale_code`	String	The locale of the intended recipient of the content.
`recipient_login_id`	String	The login ID of the intended recipient of the content.
`recipient_mobile_id`	String	The mobile ID of the intended recipient of the content.
`recipient_unique_device_id`	String	The unique device ID of the intended recipient.
`server-id`	String	The unique identifier for the Vending Manager.
`session-id`	String	The string that identifies the subscriber's session.
`source-address`	String	The address of the external entity from which the message was received, for example, the MSISDN of the SMSC.
`subscription-expiration`	Date	The date that the subscription period ends.
`subscription-frequency`	String	How often the subscription price is charged.
`subscription-recurring`	Boolean	The flag that indicates if the subscriber is automatically charged for the next period when the current subscription period ends.
`subscription-price`	Float	The price of the subscription period.
`timestamp`	Timestamp	The time at which the transaction occurred.
`unique-device-id`	String	The unique identifier for the device used.

TABLE 2-6 Event Data *(Continued)*

Parameter	Data Type	Description
`usage-count`	Integer	The number of uses allowed for the price specified for `usage-price`.
`usage-price`	Float	The price charged for the number of uses specified for `usage-count`.
`user-id`	String	The unique identifier for the user who initiated the transaction.
`username`	String	The login name for the subscriber.
`vending-res-id`	String	The tag by which the Vending Manager identifies the content.

2.4 Using the Event Service API

To use the Event Service API, you need either an SQL*Net client application or a JMS client application. This section provides information on developing the type of client that you need for your system.

2.4.1 Developing an SQL*Net Client Application

The application must obtain an SQL*Net connection to the database to directly query an Oracle database. The specifics of how that is done depends on your database server and your client application's local environment.

See Section 2.1.1, "Event Tables" on page 2-3 for information on connecting to the database.

2.4.2 Developing a JMS Client Application

To compile your JMS client, include in your classpath the Java Archive (JAR) file for JMS. This file is located at /*cds-home*/`deployment`/*deployment-name*/`lib/jms.jar` where *cds-home* is the directory in which you installed the Content Delivery Server and *deployment-name* is the name that you gave the deployment.

To execute your JMS client, follow these steps:

1. **Include in your classpath the JAR files for the components in the following table.**

TABLE 2-7 Files Required for Execution

JAR File	Location[1]
JMS	*cds-home*`/deployment/`*deployment-name*`/lib/jms.jar`
File System Context (required only if you are using Sun Java System Application Server)	*cds-home*`/deployment/`*deployment-name*`/lib/cdslib/fscontext.jar`

[1]*cds-home* is the directory in which you installed the Content Delivery Server. *deployment-name* is the name that you gave the deployment.

2. **Specify the following options for the execute command:**
   - `-Dcds.home=`*cds-home*
   - `-Dcds.config.file=CDS.properties`
   - `-Dcds.config.dir=`*cds-home*`/deployment/`*deployment-name*`/conf`

2.5 Sample Implementation of `MessageListener`

The following code example is an implementation of the `MessageListener` class. This sample shows what to do to receive billing events from the CDS billing topic.

CODE EXAMPLE 2-1 Sample `MessageListener` Implementation

```
package com.sun.content.server.eventservice.subscriber.internal;

import java.util.Properties;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;

/**
 * Attach to the CDS billing topic to receive billing events
 */
```

CODE EXAMPLE 2-1 Sample `MessageListener` Implementation *(Continued)*

```
public class CDSBillingSubscriber
        implements ExceptionListener, MessageListener
{
    private static final String kUSAGE = "CDSBillingSubscriber {JNDI Factory}
{JNDI URL}";
    private static final String kTOPIC_CONNECTION_FACTORY_NAME =
"cds.jms.TopicConnectionFactory";
    private static final String kTOPIC_NAME = "cds.messaging.billingTopic";

    private TopicConnection                     fConnection;
    private TopicSession                        fSession;
    private boolean                             fDone = false;

    public static void main(String[] args)
    {
        if (args.length != 2)
        {
            System.out.println(kUSAGE);
            System.exit(-1);
        }

        try
        {
         CDSBillingSubscriber   billingSubscriber = new CDSBillingSubscriber();
            billingSubscriber.initJMS(args[0], args[1]);

            while (!billingSubscriber.fDone)
            {
                synchronized (billingSubscriber)
                {
                    System.out.println("Waiting...");
                    billingSubscriber.wait(1000 * 10);
                }
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.exit(-1);
        }
    }


    /**
     * Initialize the JMS topic subscriber
     *
     * @param jndiFactory       the JNDI context factory
```

CODE EXAMPLE 2-1 Sample `MessageListener` Implementation *(Continued)*

```
     * @param jndiProviderUrl   the JNDI connection URL
     *
     * @throws Exception
     */
    private void initJMS(String jndiFactory, String jndiProviderUrl)
        throws Exception
    {
        //  Initialize the context
        Properties  props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY, jndiFactory);
        props.put(Context.PROVIDER_URL, jndiProviderUrl);
        InitialContext  context = new InitialContext(props);

        //  Get the topic connection factory and topic
        TopicConnectionFactory  topicConnectionFactory =
                (TopicConnectionFactory
)context.lookup(kTOPIC_CONNECTION_FACTORY_NAME);
        Topic   topic = (Topic )context.lookup(kTOPIC_NAME);

        //  Initialize the topic connection
        fConnection = topicConnectionFactory.createTopicConnection();
        fConnection.setExceptionListener(this);

        //  Get a session and subscriber
        fSession = fConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
        TopicSubscriber subscriber = fSession.createSubscriber(topic);
        subscriber.setMessageListener(this);

        fConnection.start();
    }


    /**
     * Listen for messages asynchronously. Simply print them.
     *
     * @param message
     */

    public void onMessage(Message message)
    {
        try
        {
            // simply prints the message
            TextMessage txtMsg = (TextMessage )message;
            System.out.println(txtMsg.getText());
        }
```

CODE EXAMPLE 2-1 Sample `MessageListener` Implementation *(Continued)*

```
        catch (JMSException e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Listen for exceptions and stop waiting
     *
     * @param jmse
     */
    public void onException(JMSException jmse)
    {
        jmse.printStackTrace();
        fDone = true;
        this.notifyAll();
    }
}
```

CHAPTER 3

Billing API

The Sun Java System Content Delivery Server Billing API provides an interface between the Content Delivery Server and your billing system. Use the Billing API to create a customized billing adapter that supports prepaid billing or synchronous billing.

The Billing API consists of the following elements:

- `BillingManager` - Interface that you implement to integrate the Content Delivery Server with your billing system. The `BillingManager` interface handles the authorization of a purchase, confirmation of download, subscribe and unsubscribe transactions, reversal of charges for a failed download, and a refund for purchased content.
- `BillingInfo` - Class that contains the information for a billing transaction, such as the price and pricing model for the content, subscriber identification, content identification, developer identification, and billing status. This class also includes process information, such as whether authorization is required before subscribers can download an item and whether the billing system is to be notified when the content is successfully downloaded. The `BillingInfo` object provides communication between the billing system and the Content Delivery Server.
- `BillingException` - Exception that is thrown by the Billing API.
- `BillingConstants` - Class that defines the constants used by the Billing API.

For additional information on these classes, see the HTML output of the Javadoc tool at `$CDS_HOME/javadoc/cdsapi/index.html`.

3.1 General Process Flow

The Billing API handles the communication between the Content Delivery Server and the external billing system. The details of the billing transaction are maintained in a `BillingInfo` object. The billing adapter, which is your implementation of the `BillingManager` interface, determines how the transaction is processed.

This section describes the following flows of information between the Content Delivery Server and the external billing system:

- Content Listing
- Transaction Initiation
- Subscriber Purchase
- Download Confirmation
- Subscription Verification
- Error Handling

For details on the `BillingManager` methods mentioned in this section, see Section 3.2, "`BillingManager` Interface" on page 3-7.

3.1.1 Content Listing

The list process is initiated when a subscriber requests a list of available content or a Vending Manager administrator requests a list of stocked content. FIGURE 3-1 shows this process.

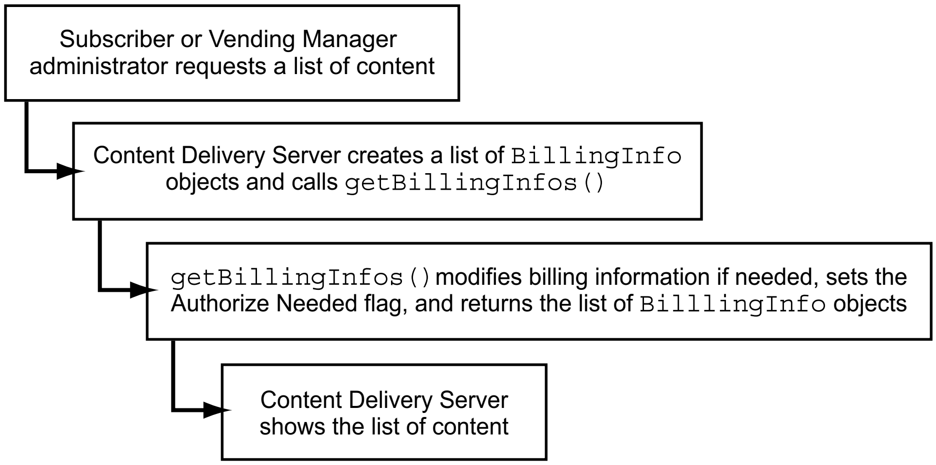
Subscriber or Vending Manager administrator requests a list of content
Content Delivery Server creates a list of BillingInfo objects and calls getBillingInfos()
getBillingInfos() modifies billing information if needed, sets the Authorize Needed flag, and returns the list of BillingInfo objects
Content Delivery Server shows the list of content

FIGURE 3-1 Process Flow for Content Listing

The following items provide additional detail on the process for listing content:

1. When a subscriber or Vending Manager administrator requests a list of content, the Content Delivery Server creates an initial `BillingInfo` object for each object in the list and calls the billing adapter's `getBillingInfos()` method.

2. Your implementation of `getBillingInfos()` can modify the purchase price or other details of each transaction, if needed. You can also specify if authorization is needed at the time of purchase and if you want a confirmation message when the content is successfully downloaded. These details are set in the `BillingInfo` objects returned to the Content Delivery Server. See Section 3.2.5, "`getBillingInfos()`" on page 3-10 for additional information.

3. The Content Delivery Server uses each returned `BillingInfo` object to show the pricing information in the list displayed.

3.1.2 Transaction Initiation

A billing transaction is initiated when a subscriber clicks View Details for an item of content. FIGURE 3-2 shows this process.

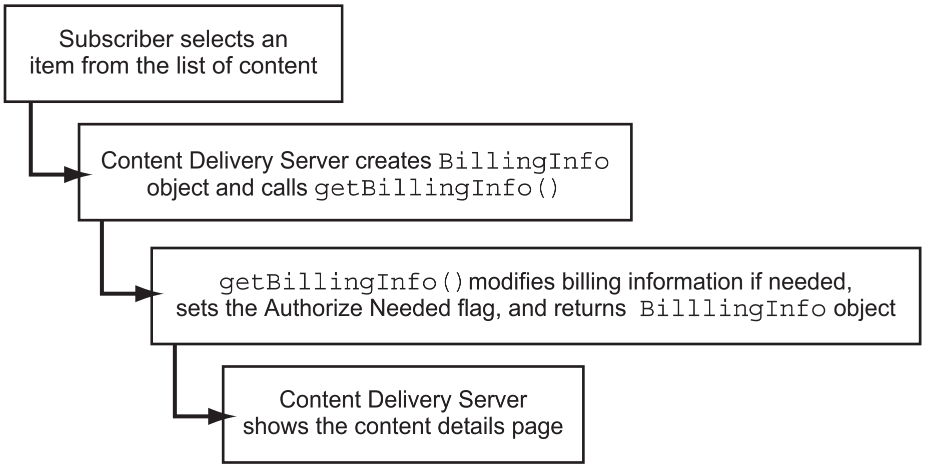
Subscriber selects an item from the list of content
Content Delivery Server creates BillingInfo object and calls getBillingInfo()
getBillingInfo() modifies billing information if needed, sets the Authorize Needed flag, and returns BilllingInfo object
Content Delivery Server shows the content details page

FIGURE 3-2 Process Flow for Transaction Initiation

The following items provide additional detail on the process for initiating transactions:

1. When a subscriber requests the details for an item, the Content Delivery Server creates the initial `BillingInfo` object for the subscriber's selection based on information in the Content Delivery Server database.

2. The Content Delivery Server then calls the billing adapter's getBillingInfo() method and passes the BillingInfo object to the method.

3. Your implementation of getBillingInfo() can modify the purchase price or other details of the transaction, if needed. You can also specify if authorization is needed at the time of purchase and if you want a confirmation message when the content is successfully downloaded. These details are set in the BillingInfo object returned to the Content Delivery Server. See Section 3.2.4, "getBillingInfo()" on page 3-9 for additional information.

4. The Content Delivery Server uses the returned BillingInfo object to show the content details and pricing information for the selected item to the subscriber.

3.1.3 Subscriber Purchase

The purchase process is initiated when the subscriber selects an item and clicks Purchase. The BillingInfo object that was created when the transaction was initiated is used for the purchase. FIGURE 3-3 shows this process.

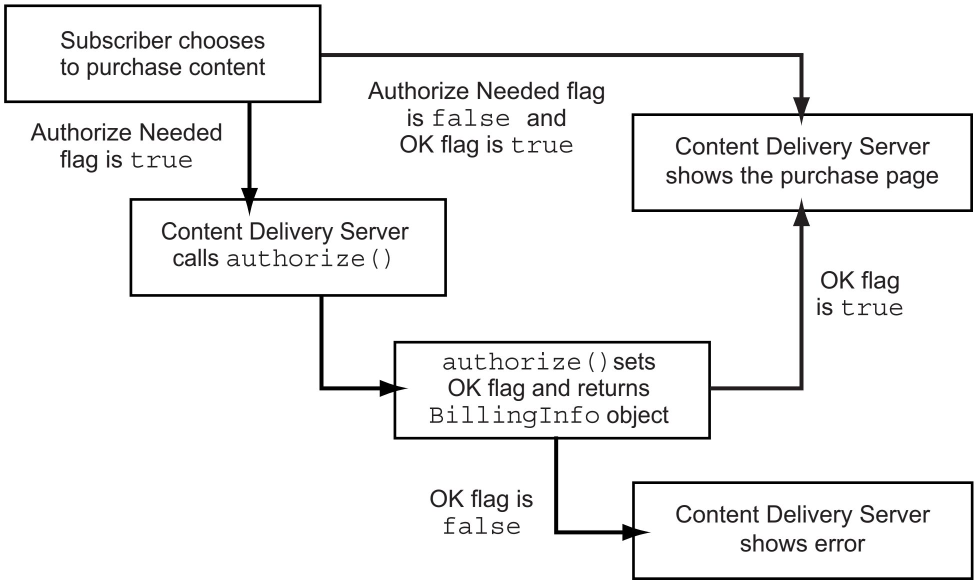
Subscriber chooses to purchase content
Authorize Needed flag is false and OK flag is true
Content Delivery Server shows the purchase page
Authorize Needed flag is true
Content Delivery Server calls authorize()
OK flag is true
authorize() sets OK flag and returns BillingInfo object
OK flag is false
Content Delivery Server shows error

FIGURE 3-3 Process Flow for Subscriber Purchase

The following items provide additional detail on the process for a subscriber purchase:

1. When the subscriber chooses to purchase the content, the Content Delivery Server checks the `BillingInfo` object returned by `getBillingInfo()` to see if authorization is required.
2. If authorization is required, the Content Delivery Server calls the billing adapter's `authorize()` method and passes the `BillingInfo` object. This method is expected to determine if the subscriber is allowed to purchase the content and set the OK flag in the `BillingInfo` object. That object is then returned to the Content Delivery Server. See Section 3.2.1, "`authorize()`" on page 3-8 for additional information.
3. If the subscriber is authorized to purchase the content, or no authorization is needed, the purchase page is presented. If the subscriber is not authorized to purchase the content, an error message is issued.

When a subscriber purchases content, the `BillingInfo` object associated with the transaction is stored in the Content Delivery Server database. For any further actions that require the billing information, the Content Delivery Server retrieves the object from the database using the subscriber ID and content ID as the key.

3.1.4 Download Confirmation

The confirmation process is initiated when the device completes the download and notifies the Content Delivery Server. FIGURE 3-4 shows this process. If the download is successful and the billing system does not require notification, no action is taken.

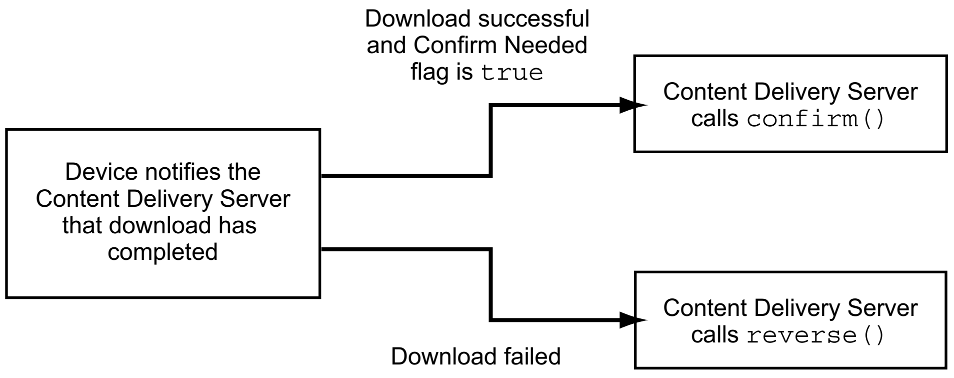
Download successful and Confirm Needed flag is true
Content Delivery Server calls confirm()
Device notifies the Content Delivery Server that download has completed
Content Delivery Server calls reverse()
Download failed

FIGURE 3-4 Process Flow for Download Confirmation

The following items provide additional detail on the process for confirming a download:

1. When the Content Delivery Server receives confirmation from the device that a download was successful, the Content Delivery Server checks the billing information to see if the billing system wants to be notified of the confirmation.

2. If confirmation is required, the Content Delivery Server calls the billing adapter's `confirm()` method. See Section 3.2.3, "`confirm()`" on page 3-9 for additional information.

3. If the device returns an error instead of a successful confirmation, the Content Delivery Server calls the billing adapter's `reverse()` method and passes the `BillingInfo` object for this transaction. Code your implementation of `reverse()` to ensure that the subscriber is not charged for the content that was not downloaded. See Section 3.2.8, "`reverse()`" on page 3-11 for additional information.

3.1.5 Subscription Verification

The subscription verification process is initiated when a subscriber accesses content after the subscription period has ended. FIGURE 3-5 shows this process.

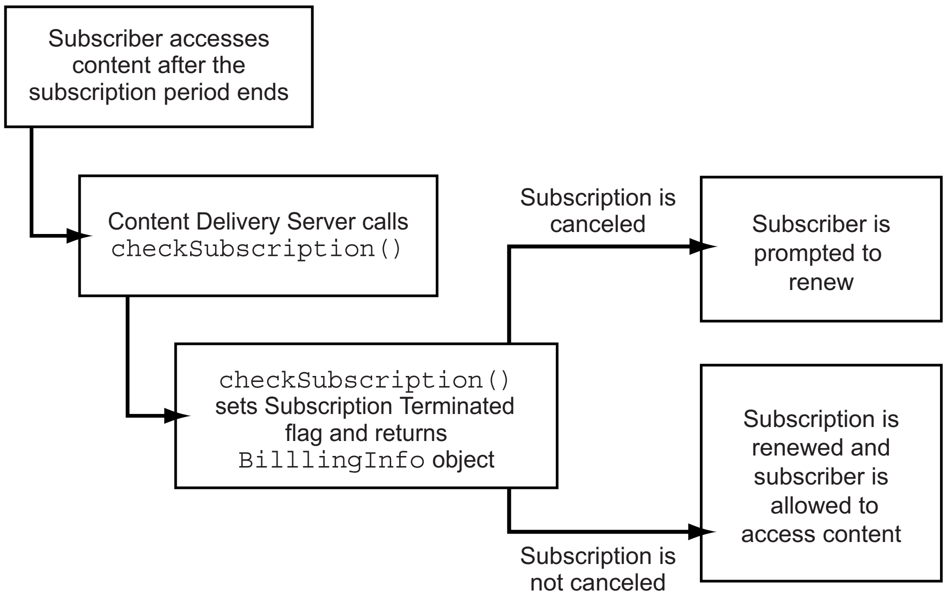
Subscriber accesses content after the subscription period ends
Content Delivery Server calls checkSubscription()
checkSubscription() sets Subscription Terminated flag and returns BillingInfo object
Subscription is canceled
Subscriber is prompted to renew
Subscription is not canceled
Subscription is renewed and subscriber is allowed to access content

FIGURE 3-5 Process Flow for Subscription Verification

The following items provide additional detail on the process for verifying a subscription:

1. When a subscriber accesses content after the subscription period has ended, the Content Delivery Server calls `checkSubscription()` to see if the subscription was canceled outside of Content Delivery Server.
2. Code your implementation of `checkSubscription()` to set the Subscription Terminated flag in the `BillingInfo` object that is returned to the Content Delivery Server. See Section 3.2.2, "`checkSubscription()`" on page 3-9 for additional information.
3. If the Subscription Terminated flag in the `BillingInfo` object is `true`, or if the Content Delivery Server subscription status is `canceled`, the subscriber is prompted to renew the subscription. If the flag is `false` and the status is not `canceled`, the subscription is automatically renewed and the content is run.

3.1.6 Error Handling

When the billing system detects an error, a generic error message is displayed to the subscriber. You can customize this message to display additional information by setting the reply message in the `BillingInfo` object returned by your implementation of the `BillingManager` interface. Whenever an error is detected by your billing adapter, call `setOK()` to set the `OK` flag in the `BillingInfo` object to false, and call `setReplyMessage()` to set the message displayed to the subscriber.

3.2 `BillingManager` Interface

The `BillingManager` interface processes the billing transaction based on the details of the transaction contained in the `BillingInfo` object. Implement `BillingManager` to create a billing adapter for your system. This billing adapter is the interface between the Content Delivery Server and your billing system.

The `BillingManager` interface is in the `com.sun.content.server.billing` package.

3.2.1 `authorize()`

```
abstract BillingInfo authorize(BillingInfo inBillingInfo,
    boolean[] inNeedToAuthorizeBillingModel)
```

If the external billing system requires that transactions be authorized, this method is called by the Content Delivery Server when the subscriber clicks the Purchase button for an item. To indicate that authorization is required, the Authorize Needed flag in the `BillingInfo` object must be set by `getBillingInfo()`.

Use this method to determine if the subscriber is authorized to purchase the content requested. If your billing system supports the prepaid billing model, use this method to verify that there are enough funds in the subscriber's account to purchase the content.

The parameter `inNeedToAuthorizeBillingModel` provides an array of flags that are indexed by the following pricing model constants, which are defined in the `BillingConstants` class:

- `DOWNLOAD` - Charge by download
- `SUBSCRIPTION` - Charge by subscription
- `TRIAL` - Offer a free trial
- `USAGE` - Charge by number of uses
- `LIMITED_TIME` - Charge for a specific interval of time

Each flag indicates if that pricing model is considered when the transaction is authorized. For example, assume you are given the following parameters:

- Pricing model is download
- Subscriber is charged for the first download only
- The current billing transaction is for a second download by the subscriber

The flag indexed by `DOWNLOAD` is set to `false`, indicating that there is no download charge for this transaction. Therefore, the download charge does not need to be considered when authorizing the transaction. Use or ignore these flags based on the needs of your system.

If the subscriber is authorized to purchase the content, set the OK flag in the `BillingInfo` object to `true` by calling `setIsOK()` before returning the `BillingInfo` object to the Content Delivery Server. If the subscriber is not authorized to purchase the content, set the OK flag to `false`.

If the Confirm Needed flag in the `BillingInfo` object was not set by `getBillingInfo()`, set the flag by calling `setConfirmNeeded()`. To notify the billing system when the content is successfully downloaded to the device, set the flag to `true`. If you do not want the billing system notified, set the flag to `false`.

3.2.2 `checkSubscription()`

```
abstract BillingInfo checkSubscription(BillingInfo inBillingInfo)
    throws BillingException
```

This method is called by the Content Delivery Server when a subscriber attempts to use content after the subscription period ends. Use this method to notify the Content Delivery Server that a subscription was terminated outside of the Content Delivery Server.

Code your implementation to set the Subscription Terminated flag by calling the `setSubscriptionTerminated()` method for the `BillingInfo` object. To indicate that the subscription was terminated, set the flag to `true`. The subscriber is then prompted to renew the subscription. To indicate that the subscription is still valid, set the flag to `false`. The subscription is then automatically renewed for another period.

3.2.3 `confirm()`

```
abstract BillingInfo confirm(BillingInfo inBillingInfo)
```

If the external billing system requires notification of successful downloads, this method is called by the Content Delivery Server when confirmation of the download is received from the subscriber's device. To indicate that confirmation is required, the Confirm Needed flag in the `BillingInfo` object must be set by either `getBillingInfo()` or `authorize()`.

Use this method to perform the actions needed when the download of content is confirmed. For example, you might want to deduct funds from a subscriber's account only after confirmation of a successful download is received.

3.2.4 `getBillingInfo()`

```
abstract BillingInfo getBillingInfo(BillingInfo inBillingInfo)
```

This method is called by the Content Delivery Server when a subscriber requests the details for an item or purchases an item. The Content Delivery Server creates a `BillingInfo` object that contains the billing information and passes it to this method.

Use this method to modify the billing information as needed and return the modified `BillingInfo` object to the Content Delivery Server. For example, to provide a discount to selected subscribers, change the price specified by the Content Delivery Server.

Code your implementation to set the Authorize Needed flag by calling the `setAuthorizeNeeded()` method for the `BillingInfo` object. To have the billing system verify that a subscriber is authorized to purchase the selected content, set the flag to `true`. If you do not want to preauthorize the subscriber, set the flag to `false`.

If you set the Authorize Needed flag to `false`, set the Confirm Needed flag and the OK flag in this method. To notify the billing system when the content is successfully downloaded to the device, set the Confirm Needed flag to `true` by calling `setConfirmNeeded()`. If you do not want the billing system notified, set the flag to `false`.

To enable the subscriber to purchase content when the Authorize Needed flag is `false`, set the OK flag to `true` by calling `setOK()`. If you set the OK flag to `false` and the Authorize Needed flag is also `false`, the subscriber is not allowed to download the selected content. If the Authorize Needed flag is set to `true`, set the OK flag in `authorize()`.

3.2.5 `getBillingInfos()`

```
abstract BillingInfo[] getBillingInfos(BillingInfo[] inBillingInfos)
```

This method is called by the Content Delivery Server when a subscriber requests the list of content available, or the Vending Manager administrator requests the list of stocked content. The Content Delivery Server creates a list of `BillingInfo` objects that contains the billing information and passes it to this method.

Use this method to modify the billing information as needed and return the modified list of `BillingInfo` objects to the Content Delivery Server. For example, to provide a discount, change the price specified by the Content Delivery Server.

Code your implementation to set the Authorize Needed flag by calling the `setAuthorizeNeeded()` method for each `BillingInfo` object in the list. To have the billing system verify that a subscriber is authorized to purchase the selected content, set the flag to `true`. If you do not want to preauthorize the subscriber, set the flag to `false`.

If you set the Authorize Needed flag to `false`, set the Confirm Needed flag and the OK flag in this method. To have the billing system notified when the content is successfully downloaded to the device, set the Confirm Needed flag to `true` by calling `setConfirmNeeded()`. If you do not want the billing system notified, set the flag to `false`.

To enable the subscriber to purchase content when the Authorize Needed flag is `false`, set the OK flag to `true` by calling `setOK()`. If you set the OK flag to `false` and the Authorize Needed flag is also `false`, the subscriber is not allowed to download the selected content. If the Authorize Needed flag is set to `true`, set the OK flag in `authorize()`.

3.2.6 getLog()

```
protected static com.sun.content.server.log.LogCategory getLog()
```

This method returns a `LogCatetory` object that is used to log error and warning messages generated by the Billing API. Use this method to get the log file to use to record errors and other information when processing billing transactions from the Content Delivery Server.

3.2.7 refund()

```
abstract void refund(BillingInfo inBillingInfo)
```

This method is called by the Content Delivery Server when a customer care agent uses the Vending Manager administration console to refund a subscriber's purchase. The Content Delivery Server creates a `BillingInfo` object from the billing information stored in the database for the original billing transaction and passes the object to this method.

Use this method to perform the actions needed to credit a subscriber's account.

3.2.8 reverse()

```
abstract void reverse(BillingInfo inBillingInfo)
```

This method is called by the Content Delivery Server when the download of content to a subscriber's device fails. The Content Delivery Server creates a `BillingInfo` object from the billing information stored in the database for the original billing transaction and passes the object to this method.

Use this method to cancel the billing transaction so the subscriber is not charged for the content.

3.2.9 subscribe()

```
abstract void subscribe(BillingInfo inBillingInfo)
```

This method is called by the Content Delivery Server when a subscriber starts a subscription for content. The Content Delivery Server creates a `BillingInfo` object from the billing information stored in the database for the original billing transaction and passes the object to this method.

Use this method to initiate a subscription for a subscriber. If the subscription is recurring, this method is called only once when the subscription starts. The billing system must remember to charge the subscriber each time the subscription period ends. If the subscription is non-recurring, this method is called each time the subscriber renews the subscription.

3.2.10 `unsubscribe()`

```
abstract void unsubscribe(BillingInfo inBillingInfo)
```

This method is called by the Content Delivery Server when a subscriber cancels a recurring subscription. The Content Delivery Server creates a `BillingInfo` object from the billing information stored in the database for the original billing transaction and passes the object to this method.

Use this method to stop the automatic charge when the subscription period ends.

3.3 Using the Billing API

The classes for the Billing API are available in the `cdsapi.jar` file, which is found in the `$CDS_HOME/deployment/`*deployment-name*`/lib/cdslib` directory. The `cdsapi.jar` file must be in your classpath when you compile your adapter.

To make your adapter available to the Content Delivery Server, follow these steps:

1. **Create a JAR file for your adapter.**
2. **Place the JAR file in the** `$CDS_HOME/deployment/`*deployment-name*`/lib/external` **directory.**
3. **Open the** `security.config` **file in the** `$CDS_HOME/deployment/`*deployment-name*`/conf` **directory.**
4. **Set the** `module.security.billingmanager` **property to the class name of your implementation of the** `BillingManager` **class, for example:**

   ```
   module.security.billingmanager=
   com.sun.content.server.billing.external.MyBillingManager
   ```
5. **Save your changes.**
6. **Restart the Content Delivery Server to make it aware of the new JAR file.**

3.4 Sample Billing Adapter

CODE EXAMPLE 3-1 shows the default implementation of the `BillingManager`.

CODE EXAMPLE 3-1 Sample `BillingManager` Implementation

```
package com.sun.content.server.billing.external;

import com.sun.content.server.billing.BillingException;
import com.sun.content.server.billing.BillingInfo;
import com.sun.content.server.billing.BillingManager;
import com.sun.content.server.log.BillingManagerKeys;
import com.sun.content.server.log.LogCategory;

/**
 * This is a sample implementation of the Billing API.
 */
public class CDSBillingManager implements BillingManager
{
  // These flags can be used to simulate responses from the billing
  //integration.
  public static final int SUCCESS = 0;
  public static final int EXCEPTION = 1;
  public static final int BILLING_EXCEPTION = 2;
  public static final int UNAUTHORIZED = 3;
  public static final int NULL = 4;

  // by default everything will pass through fine
  // but you can change these at runtime.
  public static int AUTHORIZE_RESPONSE = SUCCESS;
  public static int GET_BILLING_INFO_RESPONSE = SUCCESS;
  public static int GET_BILLING_INFOS_RESPONSE = SUCCESS;
  public static int CONFIRM_RESPONSE = SUCCESS;
  public static int DELETE_RESPONSE = SUCCESS;
  public static int REFUND_RESPONSE = SUCCESS;
  public static int REVERSE_RESPONSE = SUCCESS;
  public static int SUBSCRIBE_RESPONSE = SUCCESS;
  public static int UNSUBSCRIBE_RESPONSE = SUCCESS;
  public static int CHECK_SUBSCRIPTION_RESPONSE = SUCCESS;

  /**
   * This is used to log debug, warning, and error messages to the
   * logging system.
   */
  private static final LogCategory sLog =
    LogCategory.getLog("BillingManager");
```

CODE EXAMPLE 3-1 Sample `BillingManager` Implementation *(Continued)*

```
/**
 * see BillingManager#getBillingInfo(BillingInfo)
 */
public BillingInfo getBillingInfo(BillingInfo inBillingInfo)
       throws BillingException
{
  if (GET_BILLING_INFO_RESPONSE == NULL)
    return null;

  if (GET_BILLING_INFO_RESPONSE == EXCEPTION)
    throw new NullPointerException("Developer Null Pointer");

  if (GET_BILLING_INFO_RESPONSE == BILLING_EXCEPTION)
    throw new BillingException("Developer Billing Exception");

  // Set IsAuthorizeNeeded flag
  inBillingInfo.setAuthorizeNeeded(true);
  return inBillingInfo;
}

/**
 * see BillingManager#getBillingInfos(BillingInfo[])
 */
public BillingInfo[]
  getBillingInfos(BillingInfo[] inBillingInfos)
  throws BillingException
{
    for (int index = 0; index < inBillingInfos.length; index++)
  {
    // Set IsAuthorizeNeeded flag
    inBillingInfos[index].setAuthorizeNeeded(true);
  }

  if (GET_BILLING_INFOS_RESPONSE == NULL)
    return null;

  if (GET_BILLING_INFOS_RESPONSE == EXCEPTION)
    throw new NullPointerException("Testing Null Pointer");

  if (GET_BILLING_INFOS_RESPONSE == BILLING_EXCEPTION)
    throw new BillingException("Testing Billing Exception");

  return inBillingInfos;
}

/**
```

CODE EXAMPLE 3-1 Sample `BillingManager` Implementation *(Continued)*

```
     * see.BillingManager#authorize(BillingInfo, boolean[])
     */
    public BillingInfo authorize(BillingInfo inBillingInfo,
           boolean[] inNeedToAuthorizeBillingModel)
           throws BillingException
    {
      if (AUTHORIZE_RESPONSE == NULL)
        return null;

      if (AUTHORIZE_RESPONSE == EXCEPTION)
        throw new NullPointerException("Testing Null Pointer");

      if (AUTHORIZE_RESPONSE == BILLING_EXCEPTION)
       throw new BillingException("Testing Billing Exception");

      if (AUTHORIZE_RESPONSE == UNAUTHORIZED)
      {
        inBillingInfo.setOk(false);
        inBillingInfo.setReplyMessage("You are not authorized");
        return inBillingInfo;
      }

  // Set IsOk and IsConfirmNeeded flags
      inBillingInfo.setConfirmNeeded(true);
      inBillingInfo.setOk(true);

      return inBillingInfo;
    }

    /**
     * seeBillingManager#confirm(BillingInfo)
     */
    public BillingInfo confirm(BillingInfo inBillingInfo)
        throws BillingException
    {
      if (CONFIRM_RESPONSE == EXCEPTION)
        throw new NullPointerException("Developer Null Pointer");

      if (CONFIRM_RESPONSE == BILLING_EXCEPTION)
        throw new BillingException("Developer Billing Exception");
    }

    /**
     * see BillingManager#reverse(BillingInfo)
     */
    public void reverse(BillingInfo inBillingInfo)
        throws BillingException
```

CODE EXAMPLE 3-1 Sample `BillingManager` Implementation *(Continued)*

```
   {
     if (REVERSE_RESPONSE == EXCEPTION)
       throw new NullPointerException("Developer Null Pointer");

     if (REVERSE_RESPONSE == BILLING_EXCEPTION)
       throw new BillingException("Developer Billing Exception");
   }

   /**
    * see BillingManager#refund(BillingInfo)
    */
   public void refund(BillingInfo inBillingInfo)
       throws BillingException
   {
     if (REFUND_RESPONSE == EXCEPTION)
       throw new NullPointerException("Developer Null Pointer");

     if (REFUND_RESPONSE == BILLING_EXCEPTION)
       throw new BillingException("Developer Billing Exception");
   }

   /**
    * seeBillingManager#subscribe(BillingInfo)
    */
   public void subscribe(BillingInfo inBillingInfo)
       throws BillingException
   {
     if (SUBSCRIBE_RESPONSE == EXCEPTION)
       throw new NullPointerException("Developer Null Pointer");

     if (SUBSCRIBE_RESPONSE == BILLING_EXCEPTION)
        throw new BillingException("Developer Billing Exception");
   }

   /**
    * see BillingManager#unsubscribe(BillingInfo)
    */
   public void unsubscribe(BillingInfo inBillingInfo)
       throws BillingException
   {
     if (UNSUBSCRIBE_RESPONSE == EXCEPTION)
       throw new NullPointerException("Developer Null Pointer");

     if (UNSUBSCRIBE_RESPONSE == BILLING_EXCEPTION)
       throw new BillingException("Developer Billing Exception");
   }
```

CODE EXAMPLE 3-1 Sample `BillingManager` Implementation *(Continued)*

```
    /**
     * see BillingManager#checkSubscription(BillingInfo)
     */
    public BillingInfo checkSubscription(BillingInfo inBillingInfo)
        throws BillingException
    {
      if (CHECK_SUBSCRIPTION_RESPONSE == NULL)
        return null;

      if (CHECK_SUBSCRIPTION_RESPONSE == EXCEPTION)
        throw new NullPointerException("Developer Null Pointer");

      if (CHECK_SUBSCRIPTION_RESPONSE == BILLING_EXCEPTION)
        throw new BillingException("Developer Billing Exception");

      inBillingInfo.setSubscriptionTerminated(false);

      return inBillingInfo;
    }

    /**
     * see BillingManager#contentDelete(BillingInfo)
     */
    public void contentDelete(BillingInfo inBillingInfo)
        throws BillingException
    {
      if (DELETE_RESPONSE == EXCEPTION)
       throw new NullPointerException("Developer Null Pointer");

      if (DELETE_RESPONSE == BILLING_EXCEPTION)
       throw new BillingException("Developer Billing Exception");
    }
  }
```

CHAPTER **4**

Content Management API

The Sun Java System Content Delivery Server Content Management API provides an interface between the Content Delivery Server and your content management system. Use this API to write a content management adapter that instruments the content binary or alters the content information at the time that content is sent to a subscriber's device. For example, you can create a content management adapter that adds code to handle Digital Rights Management (DRM).

Note – If your content management adapter needs to operate on the original version of the content that was submitted by the developer, the submission verifier workflow must not perform any instrumentation or modification at the time of submission. See the *Sun Java System Content Delivery Server Integration Guide* for information on creating workflows. See the *Sun Java System Content Delivery Server Installation Guide* for information on configuring the submission verifier workflows provided with Content Delivery Server.

The time needed by the content management adapter to process the calls received can delay the delivery of the content to the subscriber. As much as possible, limit the use of the Content Manager API to operations that do not add a lot of overhead.

The Content Management API consists of the following elements:

- `ContentManager` - Interface that you implement to integrate the Content Delivery Server with your content management system. The `ContentManager` interface provides several methods for accessing the information about the content.
- `ContentInfo` - Class that contains the information for a content object, such as the content descriptor and binary, the MIME type of the content descriptor and binary, the content size and type (for example, MIDlet, ringtone, and so on) and the transaction details, such as one-step or two-step download.
- `ContentException` - Exception that is thrown by the Content Management API. Use this class to report errors in processing the information about content or in a purchase transaction.

- `ContentConstants` - Class that defines the key values used by the `ContentInfo` object to access content metadata.

For additional information on these classes, see the HTML output of the Javadoc tool at `$CDS_HOME/javadoc/cdsapi/index.html`.

4.1 General Process Flow

The Content Management API handles the communication between the Content Delivery Server and the external content management system or DRM server. The details of the content and transaction are maintained in a `ContentInfo` object. Use this API to access and manipulate the information about content at the time it is delivered to subscribers.

This section describes the flow of information between the device, the Content Delivery Server, and the content management adapter. The following topics are discussed:

- Obtaining a List of Content
- Obtaining Content Details
- Downloading Content

4.1.1 Obtaining a List of Content

A request for a list of content is initiated when the subscriber clicks a content heading under Categories in the Subscriber Portal. The following actions occur to fulfill the request:

1. The subscriber's device sends a request to Content Delivery Server for the information about the content items in the selected category.

2. The Content Delivery Server creates the initial `ContentInfo` objects for the items in the list, based on information in the Content Delivery Server database. This information is shown to subscribers or used by the Content Delivery Server.

3. Content Delivery Server calls `getContentInfos` and passes `ContentInfo` objects for the items in the list.

   Your implementation of `getContentInfos` can alter the content information as needed for your system. Modify only information that is shown to subscribers in the list of content at this time.

4. Content Delivery Server returns the list of content items for that category and additional information to the subscriber's device.

5. The Subscriber Portal displays the list in the Results section of the page.

4.1.2 Obtaining Content Details

A request for detailed information about a specific item of content is initiated when the subscriber clicks either the name of a content item or More Details in the Results page of the Subscriber Portal. The following items describe the transaction steps for obtaining content details:

1. The subscriber clicks the name of a particular content item or More Details in the Results page of the Subscriber Portal.
2. The subscriber's device sends a request to the Content Delivery Server for information on the selected content item.
3. The Content Delivery Server calls `getContentInfo` for a single item. Your implementation of the method can return detailed information about the content, such as its binary and descriptor MIME type, its content size and type (ringtone or MIDlet, for example), and the transaction details.
4. The Content Delivery Server passes the detailed information about the content item to the subscriber's device.

4.1.3 Downloading Content

The process of obtaining content is initiated when subscribers choose to purchase content from their device. Typically, the content is instrumented before it is delivered to the subscriber. The instrumentation can be based on a number of factors, such as the subscriber's phone type (CDMA or GSM), pricing model (subscription or usage), content type, or other characteristics. The following items describe the transaction steps for downloading content:

1. The subscriber chooses the desired content item in the Subscriber Portal and clicks Download Now. The subscriber must be accessing the Subscriber Portal from a device, not a PC.
2. For two-step downloads, the device sends a request to the Content Delivery Server to download the content descriptor.
3. Content Delivery Server generates the initial `ContentInfo` object for the requested content item.
4. Content Delivery Server retrieves the content binary, the descriptor, and their MIME types from the Content Delivery Server database, and passes them to the content management system.

5. Content Delivery Server calls `getContentDescriptor` to process the content details and make the appropriate updates to the descriptor file.
6. The content management system returns the updated version of the descriptor file that contains the values that are delivered to the subscriber.
7. Content Delivery Server passes the updated descriptor file to the subscriber's device.
8. The device sends a request to the Content Delivery Server for the content binary. One of the ways this occurs is when a subscriber clicks a URL displayed on the device.
9. Content Delivery Server populates the `ContentInfo` object with the content binary and its MIME type, and passes the object to the content management adapter.
10. Content Delivery Server calls `getContentBinary` to create an updated content binary file.
11. The content management adapter instruments the content and passes the updated content binary file and binary MIME type back to the Content Delivery Server.

Note – Capability matching, which determines the devices on which the content runs, is done at the time the content is published. Changes made by the content management adapter must not alter the content in such a way that the device is no longer capable of running the content. Changes that might affect the ability of the device to run the content include increasing the size of the content or changing the MIME type.

---

12. Content Delivery Server passes the updated content to the subscriber's device.

4.2 `ContentManager` Interface

The `ContentManager` interface provides the interface between the Content Delivery Server and your content management system or DRM server. It provides methods that you can implement to modify the content binary, the descriptor, and other information, before they are delivered to the subscriber.

The methods in this class take a `ContentInfo` object and a `BillingInfo` object as parameters. The `ContentInfo` object contains the content binary, the content descriptor file, and information such as the content binary and descriptor MIME type, the content size and type, and the number of steps required to download the content.

The `BillingInfo` object contains transaction details, such as the pricing model. This object also contains subscriber information. For more information on billing and `BillingInfo` objects, see Chapter 3.

The `ContentManager` interface is in the `com.sun.content.server.content` package.

4.2.1 `getContentInfo()`

```
abstract ContentInfo getContentInfo(ContentInfo inContentInfo,
BillingInfo inBillingInfo) throws ContentException
```

This method is called by the Content Delivery Server when the subscriber requests information on a single item of content. The `ContentInfo` object contains information about the initial content item, such as the content binary and descriptor MIME type, estimated content size and type (ringtone or MIDlet, for example), and how many steps are needed to download the content.

4.2.2 `getContentInfos()`

```
abstract ContentInfo[] getContentInfos(ContentInfo[]inContentInfos,
BillingInfo[] inBillingInfos) throws ContentException
```

This method is called by the Content Delivery Server when the subscriber requests information on a list of content items in a category. Code your implementation to alter only the information that is shown in the list of content.

4.2.3 `getContentDescriptor()`

```
abstract ContentInfo getContentDescriptor(ContentInfo inContentInfo,
BillingInfo inBillingInfo) throws ContentException
```

This method is called by the Content Delivery Server when the subscriber initiates a request to download the content descriptor to a device. The Content Delivery Server retrieves the content descriptor file. The content descriptor and binary are passed to the content management system. The content management system can update the content descriptor information, including the size, and return it to the Content Delivery Server for delivery to the subscriber.

4.2.4 `getContentBinary()`

`abstract ContentInfo getContentBinary(ContentInfo inContentInfo, BillingInfo inBillingInfo) throws ContentException`

This method is called by the Content Delivery Server when the subscriber initiates a request to download content binary to a device. The binary might be passed to the content management adapter when `getContentBinary` is called. The instrumented content binary to be delivered to the subscriber must be returned to the Content Delivery Server.

4.3 Using the Content Management API

The classes for the Content Management API are available in `cdsapi.jar`, which is found in the `$CDS_HOME/deployment/`*deployment-name*`/lib/cdslib` directory. The `cdsapi.jar` file and the `foundation.jar` file must be in your classpath when you compile your adapter.

To make your adapter available to the Content Delivery Server, follow these steps:

1. **Create a JAR file for your adapter.**
2. **Place the JAR file in the** `$CDS_HOME/deployment/`*deployment-name*`/lib/external` **directory.**
3. **Open the** `security.config` **file in the** `$CDS_HOME/deployment/`*deployment-name*`/conf` **directory.**
4. **Add a property named** `module.security.contentmanager` **after the existing property with the same name.**

   The existing `module.security.contentmanager` property points to the implementation used by the DRM agents provided with the Content Delivery Server and must appear first.
5. **Set the property that you added to the fully qualified package and class name of your implementation of the** `ContentManager` **interface.**

   The following code shows sample settings for the `module.security.contentmanager` property:

```
module.security.contentmanager=
com.sun.content.server.content.external.SunContentManager
module.security.contentmanager=myapps.adapters.ContentManagerImpl
```

6. **Set the** `module.security.contentmanager.enabled` **property to** `true` **to indicate that an adapter is available for the Content Delivery Server to call.**

7. **Save your changes.**

8. **Restart the Content Delivery Server to make it aware of the new JAR file.**

4.4 Sample Content Management Adapter

CODE EXAMPLE 4-1 shows a sample implementation of the `ContentManager` interface.

CODE EXAMPLE 4-1 Sample `ContentManager` Implementation

```
import com.sun.content.server.content.*;
import com.sun.content.server.billing.BillingInfo;

public class ContentManagerImpl implements ContentManager
{
  public ContentInfo getContentInfo(
    ContentInfo inContentInfo,
    BillingInfo inBillingInfo)
  throws ContentException
  {
    // Update the information that is shown to the user
    return inContentInfo;
  }

  public ContentInfo[] getContentInfos(
    ContentInfo[] inContentInfos,
    BillingInfo[] inBillingInfos)
  throws ContentException
  {
    // Iterate through each ContentInfo object and update the
    // information that is shown to the user when a list of
    // content is shown.
    return inContentInfos;
  }

  public ContentInfo getContentDescriptor(
    ContentInfo inContentInfo,
    BillingInfo inBillingInfo)
  throws ContentException
  {
    // Update content download descriptor
```

CODE EXAMPLE 4-1 Sample `ContentManager` Implementation *(Continued)*

```
      return inContentInfo;
    }

    public ContentInfo getContentBinary(
      ContentInfo inContentInfo,
      BillingInfo inBillingInfo)
    throws ContentException
    {
      // Update content binary, binary MIME type
      return inContentInfo;
    }
  }
```

CHAPTER **5**

Content Validation API

This chapter describes the Content Validation API of the Sun Java System Content Delivery Server. Use this API to create the content validation adapters that you need to validate and protect content that is submitted. Content validation adapters are used by the submission verifier workflow that is executed for content submitted to the Content Delivery Server.

The Content Validation API consists of the following classes:

- `ValidationAdapter` - Abstract class that you extend to create your own content validation adapter. Implement your adapter to validate or modify the submitted content as needed.
- `ValidationContent` - Abstract class that you extend to create your own `ValidationContent` object. A `ValidationContent` object contains the information that identifies the content that was submitted.

For additional information on these classes, see the HTML output of the Javadoc tool at `$CDS_HOME/javadoc/validation/index.html`.

5.1 General Process Flow

Each item of content that is submitted to the Content Delivery Server is processed by a workflow defined in the `$CDS_HOME/deployment/`*deployment-name*`/conf/SubmissionVerifierWorkflows.xml` file. A workflow consists of a sequence of steps. In each step, the content validation adapter named in that step performs some type of processing on the content binary. For example, one step might provide obfuscation and another step might add code for digital rights management (DRM). See the *Sun Java System Content Delivery Server Integration Guide* for information on creating content validation workflows.

For each step in the workflow, the Content Delivery Server calls the content validation adapter that is specified for that step and passes it a `ValidationContent` object and a `Properties` object. The `ValidationContent` object contains the metadata and content binary for the content submitted. The `Properties` object contains the arguments defined in the workflow for that step.

For the first step in the workflow, the Content Delivery Server creates an initial `ValidationContent` object. For each subsequent step, the `ValidationContent` object that was returned by the previous step is passed to the content validation adapter for the next step.

5.2 `ValidationAdapter` Class

The `ValidationAdapter` class processes the metadata and content binary and performs any transformation that is needed. This section describes the methods that you need to implement.

5.2.1 `execute()`

```
public abstract ValidationContent execute(ValidationContent content,
java.util.Properties properties) throws java.lang.Exception
```

This method is called by the Content Delivery Server when a step in the workflow is executed. The arguments for this method ar the `ValidationContent` object from the previous step and the arguments specified in the workflow step. You must know what type of `ValidationContent` object is created in the previous step in the workflow. For example, if your adapter is used in the first step in the workflow, it must be prepared to receive an `InitialValidationContent` object from the Content Delivery Server.

In your implementation of this method, you must handle the properties that are passed and create a `ValidationContent` object to return. Process the metadata and content binary as needed. For example, to obfuscate the code, call the obfuscator that you want to use to transform the content binary and return the new binary in the `ValidationContent` object. You must know what type of `ValidationContent` object is expected in the next step in the workflow and generate that type of object. For example, if the next step in the workflow expects to receive a custom `ValidationObject`, this method must generate that custom `ValidationObject`.

If your adapter requires information that might change or that you do not want to hard code, create a property file for these values. The adapter can then access the information from the file through the `Properties` object that is passed as a parameter. For example, if your adapter obfuscates the code, you might need a property that identifies the location of the obfuscator on the system on which the adapter is running. The property file that you create must be placed in the `$CDS_HOME/deployment/`*deployment-name*`/conf` directory. You must also set the property file name for your adapter in the `$CDS_HOME/deployment/`*deployment-name*`/conf/SubmissionVerifierAdapters.xml` file as described in Section 5.4, "Using the Content Validation API" on page 5-3.

5.2.2 `returns()`

```
public static java.lang.Class returns(java.lang.Class inputType)
throws java.lang.Exception
```

This method is called by the Content Delivery Server to verify that the adapter can handle the type of `ValidationContent` object to be passed. This method must return the same type of object that the `execute` method returns. For example, if the `execute` method returns a custom `ValidationContent` object, the `returns` method must return the same type of custom `ValidationContent` object.

See Section 5.5, "Sample Content Validation Adapter" on page 5-5 for a sample implementation of this method.

5.3 `ValidationContent` Class

The `ValidationContent` class is an abstract class that you can extend to create a customized validation adapter. This class contains the metadata and binary for the content. If an adapter in a following step in the workflow needs additional information for the content, add that information to the class that you extend from this class.

5.4 Using the Content Validation API

The classes for the Content Validation API are in the package `com.sun.content.server.validation.adapter`. This package is included in the `validation.jar` file in one of the following locations:

- For Sun Java System Application Server, in `$CDS_HOME/deployment/`*deployment-name*`/sun/domains/`*server-domain*`/`*server-name*`/applications/j2ee-modules/CDSDeveloperPortal/WEB-INF/lib`.

- For WebLogic Server, in $CDS_HOME/deployment/*deployment-name*/weblogic/domains/*server-domain*/applications/developer/WEB-INF/lib.

deployment-name is the name specified when the Developer Portal was deployed, *server-domain* is the value specified in the configuration file for the app.server.domain property, and *server-name* is the value specified in the configuration file for the app.server.name property.

The validation.jar file must be in your classpath when you compile your adapter.

To make your adapter available to the Content Delivery Server, follow these steps:

1. **Create a JAR file for your adapter.**

2. **Place the JAR file in the** $CDS_HOME/deployment/*deployment-name*/lib/external **directory.**

3. **Open the** SubmissionVerifierAdapters.xml **file in the** $CDS_HOME/deployment/*deployment-name*/conf **directory.**

4. **Add a statement for the adapter that you created.**

   For example, if you created an adapter named MyValidationAdapter that requires a property file named validation.properties, add the following statement to the file:

```
<adapter id="MyValidationAdapter" name="sample.package.MyValidationAdapter"
propertyfile="validation.properties"/>
```

5. **Save your changes.**

6. **Open the** SubmissionVerifierWorkflows.xml **file in the** $CDS_HOME/deployment/*deployment-name*/conf **directory.**

7. **Add a step to the appropriate workflow to execute the adapter that you created.**

   For the value of the adapter attribute for the step element, specify the value provided for the id attribute of the adapter element that you added to the SubmissionVerifierAdapters.xml file in Step 4. See the *Sun Java System Content Delivery Server Integration Guide* for information on creating a workflow.

8. **Save your changes.**

9. **Restart the Content Delivery Server to make it aware of the new JAR file.**

5.5 Sample Content Validation Adapter

The following code example is a sample of how the `ValidationAdapter` class can be extended to implement your own validation adapter.

CODE EXAMPLE 5-1 Sample `ValidationAdapter` Implementation

```
import com.sun.content.server.validation.adapter.*;
import java.io.FileOutputStream;
import java.util.Properties;

public class ExportToFileValidationAdapter
extends ValidationAdapter
{
  public ValidationContent execute(
    ValidationContent content, Properties properties)
  throws Exception
  {
    // Export if the filename is specified
    String outFilename =
      properties.getProperty("ExportToFile.FileName");
    if (outFilename != null)
    {
      FileOutputStream fileOutStream = null;
      try
      {
        // get the first byte[] in the map
        // ignore the rest for this sample
        byte[] bytes = (byte[])
          content.getMimeBytesMap().values().iterator().next();

        // Write the byte[] to the output file.
        fileOutStream = new FileOutputStream(outFilename);
        fileOutStream.write(bytes);
        fileOutStream.flush();
        fileOutStream.close();
        fileOutStream = null;
      }
      finally
      {
        if (fileOutStream != null)
        {
          fileOutStream.flush();
          fileOutStream.close();
        }
      }
```

CODE EXAMPLE 5-1 Sample `ValidationAdapter` Implementation *(Continued)*

```
    }

    content.setStatus(ValidationContent.VALID);

    return content;
  }

  public static Class returns(Class inputType) throws Exception
  {
    if (!ValidationContent.class.isAssignableFrom(inputType))
      throw new Exception("Wrong input type to adapter.");
    return ValidationContent.class;
  }
}
```

CHAPTER **6**

User Profile API

This chapter describes the Sun Java System Content Delivery Server User Management API. Use this API to create a subscriber adapter to add, delete, retrieve, update, enable, and disable users in the system.

The User Profile API consists of the following classes:

- `UserManager` - Abstract class that you extend to create your own subscriber adapter. This class controls the creation and status of an individual user account.
- `User` - Interface that you implement to manage the attributes specific for each user.
- `UserDeviceManager` **interface** - Interface that you implement to provide information about the device that the subscriber is using. This inteface can be used to protect against the subscriber paying once and downloading the content to several devices.

For information on classes or methods not described in this section, see the HTML output of the Javadoc tool for the User Profile API at `$CDS_HOME/javadoc/cdsapi/index.html`.

6.1 `UserManager` Class

The `UserManager` class defines methods to create, delete, or access information for a user.

6.1.1 doAccountExists()

```
abstract boolean doAccountExists(String uId)
```

Returns `true` if a user account already exists in the persistent storage for a given user ID.

6.1.2 doAddUser()

```
protected abstract boolean doAddUser(User u)
```

Creates a user.

6.1.3 doDisableUser()

```
protected abstract void doDisableUser(String uId)
```

Disables a user's account. Used to lock the user account.

6.1.4 doEnableUser()

```
protected abstract void doEnableUser(String uId)
```

Enables a user's account.

6.1.5 doFormatMobileId()

```
protected abstract String doFormatMobileId(String mobileId);
```

Formats the mobile ID to match the requirements of the external database. For example, use this method to remove special characters for a database that cannot handle hyphens within the ID.

6.1.6 doFormatLoginId()

```
protected abstract String doFormatLoginId(String loginId);
```

Formats the login ID to match the requirements of the external database. For example, use this method to convert the ID to lowercase for a database that requires lowercase IDs.

6.1.7 doGetAllLikeInOrder()

```
protected abstract void doAllLikeInOrder(String[] columns, String[]
values, String[] orders, boolean isDescending, int pageNum, int
recPerPage, String role)
```

Returns an iterator of requested objects in the order specified.

6.1.8 doGetAllUsers()

```
protected abstract Iterator doGetAllUsers(String role)
```

Returns an iterator of all the users in permanent storage based on user role.

6.1.9 doGetAllUsersContainingFirstName()

```
protected abstract Iterator doGetAllUsersContainingFirstName(String
nameLike, String role)
```

Returns an iterator of all users containing the specified substring in the first name based on user role.

6.1.10 doGetAllUsersContainingId()

```
protected abstract Iterator doGetAllUsersContainingId(String id,
String role)
```

Returns an iterator of all users containing the specified substring in the user ID based on user role.

6.1.11 doGetAllUsersContainingLastName()

```
protected abstract Iterator doGetAllUsersContainingLastName(String
lastName, String role)
```

Returns an iterator of all users containing the specified substring in the last name based on user role.

6.1.12 doGetAllUsersContainingName()

```
protected abstract Iterator doGetAllUsersContainingName(String name,
String role)
```

Returns an iterator of all users containing the specified substring in either the first name or the last name based on user role.

6.1.13 doGetAnonymousUser()

```
protected abstract User doGetAnonymousUser()
```

Returns a guest user for anonymous access to the system.

6.1.14 doGetFieldName()

```
protected abstract String doGetFieldName(int fieldConstant, String
role)
```

Gets the name of the field in the external database that corresponds to a field constant based on the user role.

6.1.15 doGetUser()

```
protected abstract User doGetUser(String uId)
```

Returns the user object associated with a given user ID.

6.1.16 doGetUserByMobileId()

```
protected abstract User doGetUserByMobileId(String inMobileId)
```

Returns the user object associated with a given user mobile ID.

6.1.17 doGetUserByUniqueDeviceId()

```
protected abstract User doGetUserByUniqueDeviceId(String
inUniqueDeviceId)
```

Returns the user object associated with a given user unique device ID.

6.1.18 `doGetUserInstance()`

```
protected abstract User doGetUserInstance()
```

Returns the instance of user class implementation being passed in the `addUser` method.

6.1.19 `doIsActive()`

```
protected abstract boolean doIsActive(String uId)
```

Returns `true` if the user account associated with the `userId` is active. Returns `false` if the account is not active.

6.1.20 `doIsAuthenticated()`

```
protected abstract boolean doIsAuthenticated(String userId, String
inPassword)
```

Returns `true` if the user ID and password provided are allowed access.

6.1.21 `doRemoveUser()`

```
protected abstract boolean doRemoveUser(String uId)
```

Removes the user from persistent storage. Returns `true` if successful.

6.1.22 `doUpdateUser()`

```
protected abstract boolean doUpdateUser(User u)
```

Updates an existing user's account.

6.2 `User` Interface

The `User` interface defines common methods to get, set, or delete user attributes.

6.2.1 `getActivateDate()`

```
public Date getActivateDate()
```

Returns the date of the user's account is activated.

6.2.2 `getAttribute()`

```
public Object getAttribute ( String name )
```

Returns the value of an attribute associated with a particular user.

6.2.3 `getAttribute()`

```
public Object getAttribute ( String name, Object defaultValue )
```

Returns the value of a specific attribute associated with a particular user or the default value if the attribute is not found.

6.2.4 `getAttributes()`

```
public Hashtable getAttributes()
```

Returns a hash table containing all the attributes associated with a particular user.

6.2.5 `getCity()`

```
public String getCity()
```

Returns the city name.

6.2.6 `getCountry()`

```
public String getCountry()
```

Returns the country name.

6.2.7 `getCreateDate()`

```
public java.util.Date getCreateDate()
```

Returns the date the user's account is created.

6.2.8 `getDeActivateDate()`

```
public Date getDeActivateDate()
```

Returns the date the user's account is activated.

6.2.9 `getEmail()`

```
public String getEmail()
```

Returns user's email ID.

6.2.10 `getFirstName()`

```
public String getFirstName()
```

Returns the user's first name.

6.2.11 `getGender()`

```
public String getGender()
```

Returns the user's gender.

6.2.12 `getLastLogin()`

```
public java.util.Date getLastLogin()
```

Returns the time of the user's last login.

6.2.13 `getLastName()`

```
public String getLastName()
```

Returns the user's last name.

6.2.14 `getLoginId()`

```
public String getLoginId()
```

Returns the user's unique login ID.

6.2.15 `getMiddleName()`

```
public String getMiddleName()
```

Returns the user's middle name.

6.2.16 `getMobileId()`

```
public String getMobileId()
```

Returns the user's mobile ID.

6.2.17 `getPassword()`

```
public String getPassword()
```

Returns the user's unique password.

6.2.18 `getPhone()`

`public String getPhone()`

Returns the user's contact phone number.

6.2.19 `getPostalCode()`

`public String getPostalCode()`

Returns the postal code of the user's account.

6.2.20 `getSalutation()`

`public String getSalutation()`

Returns user's salutation.

6.2.21 `getState()`

`public String getState()`

Returns the state name.

6.2.22 `getStreet1()`

`public String getStreet1()`

Returns the first line of the user's street address.

6.2.23 `getStreet2()`

`public String getStreet2()`

Returns the second line of the user's street address.

6.2.24 `getUniqueDeviceId()`

```
public String getUniqueDeviceId()
```

Returns the user's unique device ID.

6.2.25 `hasLoggedIn()`

```
public boolean hasLoggedIn()
```

Returns `true` if the user is currently logged in to the system.

6.2.26 `isConfirmed()`

```
public boolean isConfirmed()
```

Returns `true` if the user's account is confirmed or verified.

6.2.27 `isEnabled()`

```
public boolean isEnabled()
```

Returns `true` if the user account is currently enabled and `false` if disabled.

6.2.28 `isPrepay()`

```
public boolean isPrepay()
```

Returns `true` if the user account is prepaid.

6.2.29 `setActivateDate()`

```
public void setActivateDate(Date aDate)
```

Sets the date the user's account is activated.

6.2.30 `setAttribute()`

```
public void setAttribute ( String name, Object value )
```

Sets the attribute associated with a particular user.

6.2.31 `setAttributes()`

```
public void setAttributes(Hashtable stuff)
```

Sets the list of attributes associated with a particular user.

6.2.32 `setCity()`

```
public void setCity(String city)
```

Sets the city name.

6.2.33 `setCountry()`

```
public void setCountry(String country)
```

Sets the country name.

6.2.34 `setCreateDate()`

```
public void setCreateDate(java.util.Date date)
```

Sets the date the user's account is created.

6.2.35 `setDeActivateDate()`

```
public void setDeActivateDate(Date daDate)
```

Sets the date the user's account is inactivated.

6.2.36 `setEmail()`

```
public void setEmail(String address)
```

Sets the user’s email ID.

6.2.37 `setFirstName()`

```
public void setFirstName(String firstName)
```

Sets the user’s first name.

6.2.38 `setGender()`

```
public void setGender(String gender)
```

Sets the user’s gender.

6.2.39 `setHasLoggedIn()`

```
public void setHasLoggedIn(boolean value)
```

Sets the flag to indicate the user has logged into the system.

6.2.40 `setIsEnabled()`

```
public void setIsEnabled(boolean value)
```

Sets the account status to enabled.

6.2.41 `setIsPrepay()`

```
public void setIsPrepay(boolean value)
```

Sets the account to prepaid.

6.2.42 `setLastName()`

```
public void setLastName(String lastName)
```

Sets the user’s last name.

6.2.43 `setLoginId()`

```
public void setLoginId(String loginName)
```

Sets the user’s unique login ID.

6.2.44 `setMiddleName()`

```
public void setMiddleName(String name)
```

Sets the user’s middle name.

6.2.45 `setMobileId()`

```
public void setMobileId(String mobileId)
```

Sets the user’s mobile ID.

6.2.46 `setPassword()`

```
public void setPassword(String pass)
```

Sets the user’s password.

6.2.47 `setPhone()`

```
public void setPhone(String phone)
```

Sets the user’s contact phone number.

6.2.48 `setPostalCode()`

```
public void setPostalCode(String zip)
```

Sets the postal code for the user’s address.

6.2.49 `setSalutation()`

```
public void setSalutation(String salutation)
```

Sets the salutation for the user (for example, Mr. or Mrs.)

6.2.50 `setState()`

```
public void setState(String state)
```

Sets the state for the user’s address.

6.2.51 `setStreet1()`

```
public void setStreet1(String st1)
```

Sets the first line of the user’s street address.

6.2.52 `setStreet2()`

```
public void setStreet2(String st2)
```

Sets the second line of the user’s street address.

6.2.53 `setUniqueDeviceId()`

```
public void setUniqueDeviceId(String uniqueId)
```

Sets the user’s unique device ID.

6.2.54 `updateLastLogin()`

```
public void updateLastLogin() throws Exception
```

Updates the timestamp with the time of the last login.

6.3 `UserDeviceManager` Interface

The `UserDeviceManager` interface defines a method for accessing the unique ID for a device.

6.3.1 `getUniqueDeviceID()`

```
public String getUniqueDeviceID(String inMSISDN) throws
UserDeviceException
```

Gets the unique device ID such as the Electronic Serial Number (ESN) given the MSISDN for the device.

6.4 Using the User Profile API

The classes for the User Profile API are available in `cdsapi.jar`, which is found in the `$CDS_HOME/deployment/`*deployment-name*`/lib/cdslib` directory. The `cdsapi.jar` file must be in your classpath when you compile your adapter.

To make your adapter available to the Content Delivery Server, follow these steps:

1. **Create a JAR file for your adapter.**
2. **Place the JAR file in the** `$CDS_HOME/deployment/`*deployment-name*`/lib/external` **directory.**
3. **Open the** `security.config` **file in the** `$CDS_HOME/deployment/`*deployment-name*`/conf` **directory.**
4. **Set the** `module.security.subscriber.usermanager` **property to the class name of your implementation of the** `User` **interface, for example:**

```
module.security.subscriber.usermanager=
com.sun.content.server.server.security.user.SubscriberImpl
```

5. **Save your changes.**
6. **Restart the Content Delivery Server to make it aware of the new JAR file.**

6.5 Sample Implementation of the User Manager API

The following example is the class definition of the `MyUserMgr`. It is an example of how you can inherit the `UserManager` class to manage your user profile system and integrate it into the Content Delivery Server User Profile framework.The following sample classes provide an example of how you can use the User Profile API to manage your user profile system and integrate it into the Content Delivery Server User Profile framework.

6.5.1 Support Files

The sample implementation of the User Profile API makes use of properties and a supporting class. The following code shows the properties used.

CODE EXAMPLE 6-1 Sample Property File

```
# This is a sample of typical Operator Integration properties file
# for external user manager client/server connection settings.

# SAMPLE OPERATOR CONFIGURATIONS
#
externalserveraddress=localhost
externalserverport=7779
cdsclientlogin=cds14sun
cdsclientpasswd=cds4sun1
```

The `SampleExternalProxy` class is used as the interface between the Content Delivery Server and the external database.

CODE EXAMPLE 6-2 Sample External Proxy

```
package com.sun.content.server.operator.security.adaptor;

import java.util.*;
import com.sun.content.server.service.security.*;
import com.sun.content.server.service.security.util.*;
```

CODE EXAMPLE 6-2 Sample External Proxy *(Continued)*

```
// import external required packages to connect to the directory service

public class SampleExternalProxy
{
     // Create a Client facace for serach
    private ExternalClientObject externalUserMgr;

    // Create an instance of External Directorory server Client Proxy
    public SampleExternalProxy() throws Exception {

    // User the external package to instanciate the Proxu
    // You need to use the client/server setting in the
    // operatorproxy.properties
    // The following assume you implemented a class OperatorProxyProperties
    // to read the configuration values.

    String hostname = OperatorProxyProperties.EXTERNAL_SERVER_ADDRESS;
    String port = OperatorProxyProperties.EXTERNAL_SERVER_PORT;
    String clientLoginId = OperatorProxyProperties.CLIENT_LOGIN;
        String clientPassword = OperatorProxyProperties.CLIENT_PASSWRD;

        // instan
        try {
       externalUserMgr = new ExternalClientObject(hostname, port,
             clientLoginId, clientPassword);
        } catch (Exception expt) {
        // process the exception
         throw expt;
        }
    }

    public boolean searchUser(String userName) {

        boolean found = false;
    // Use the external User manager and search function for the given string
    // name
    // Typically the call will look like
    //  found = externalUserMgr.searchUser(userName);
    // You may need to catch potential exception and display the appropriate
    // message
        return found;
    }

    public SampleUserImpl createUserFromExternal(String loginId)
            throw Exception {

    System.out.println("DEBUG: SampleExternal Proxy--createUserFromExternal ");
```

CODE EXAMPLE 6-2 Sample External Proxy *(Continued)*

```
    System.out.println(" Creating a SampleUserImpl from External
Directory...");
    System.out.println(" Reference Login Id = "+loginId);

        try {

       // Assuming the external client has been created and connection is
        // established
       // The following call typically search for the UserName and return an
        // External UserProfile
       aUserProfileData aUPD = externalUserMgr.getUserProfileData(loginId);
       String password = aUPD.getCredential();

            // Create basic User information (firstname,
            // middlename, lastname,
            // address, etc.)
            String firstName = aUPD.getFirstName();
            String lastName = aUPD.getLastName();
            String middleName = aUPD.getMiddleInitial();
            String gender = aUPD.getGender();
            String salutation = aUPD.getOccupation();
            String street1 = aUPD.getStreet();
            String street2 = aUPD.getStreetNumber();
            String city = aUPD.getCity();
            String state = aUPD.getState();
            String postalcode = aUPD.getZipCode();
            String country = aUPD.getCountry();
            String phone = aUPD.getFixedPhone();

            // Creating email information
            String email = aUPD.getMailAddress();

            // Creating msisdn in this case Unique Device Id ()
            String uniqueDeviceId = aUPD.getMsisdn();

            // Creating Status data: User Enabled/Desabled and User is
            // Prepay/Non-Prepay are valued 1/0

            boolean enabled = false;
            if (aUPD.getStatus().equals("1"))
                enabled = true;

            boolean prepay = false;
            if (aUPD.getPrepayType().equals("1"))
                prepay = true;

            // Create activation and deactivation dates if provided
```

CODE EXAMPLE 6-2 Sample External Proxy *(Continued)*

```
            Date activatedate = new Date();
            Date deactivatedate = null;

            // Create and return a Sample User Implementation
            return new SampleUserImpl(
                loginId,
                password,
                firstName,
                lastName,
                middleName,
                gender,
                street1,
                street2,
                city,
                state,
                postalcode,
                country,
                email,
                phone,
                activatedate,
                deactivatedate,
                salutation,
                enabled,
                uniqueDeviceId,
                prepay);

        } catch (Exception ex) {
            // procecss exception
            throw ex;
        }

        return null;
    }
}
```

6.5.2 SampleUserImpl.java

The following code is a sample implementation of the User interface. This interface includes the fields that the Content Delivery Server uses for user profiles. If you have additional fields for your specific implementation, add the methods required to get and set those values.

CODE EXAMPLE 6-3 Sample User Implementation

```
package com.sun.content.server.operator.security.adaptor;
import com.sun.content.server.service.security.User;
import java.util.Date;
import java.util.Hashtable;

public class SampleUserImpl implements User
{

    private Hashtable fInfo;
    public SampleUserImpl() {

        fInfo = new Hashtable();

        setLoginId("guest");
        setFirstName("guest");
        setLastName("guest");
        setPassword ("guest");
        setCreateDate(new Date());
        setActivateDate(new Date());
        fInfo.put("enabled", String.valueOf(true));
        setMiddleName("guest");
        setEmail("guest@email.com");
        setUniqueDeviceId("1231231233");
    }

    public SampleUserImpl(String uid, String pwd, String fname, String lname,
       String mname, String gender, String street1, String street2, String city,
       String state, String postalcode, String country, String email,
       String phone, Date actdate,Date deactdate,String salutation, ,
       boolean enabled, String uniqueDeviceId, boolean isprepay) {

        fInfo = new Hashtable();

        setLoginId(uid);
        setPassword(pwd);
        setFirstName(fname);
        setLastName(lname);
        setMiddleName(mname);
        setGender(gender);
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
        setStreet1(street1);
        setStreet2(street2);
        setCity(city);
        setState(state);
        setPostalCode(postalcode);
        setCountry(country);
        setEmail(email);
        setPhone(phone);
        setFixedPhone(phone);

        // It is safe to check the activation date and use current date if null
        if ( actdate == null ) {
            setCreateDate(new Date());
            setActivateDate(new Date());
        }
        else {
            setCreateDate(actdate);
            setActivateDate(actdate);
        }

        setDeActivateDate(deactdate);
        setSalutation(salutation);
        setIsEnabled(enabled);
        setUniqueDeviceId(uniqueDeviceId);
        setIsPrepay(isprepay);
    }

    public SampleUserImpl(User inUser) {

        fInfo = new Hashtable();

        setLoginId(inUser.getLoginId());
        setFirstName(inUser.getFirstName());
        setLastName(inUser.getLastName());
        setMiddleName(inUser.getMiddleName());
        setGender(inUser.getGender());
        setStreet1(inUser.getStreet1());
        setStreet2(inUser.getStreet2());
        setCity(inUser.getCity());
        setState(inUser.getState());
        setPostalCode(inUser.getPostalCode());
        setCountry(inUser.getCountry());
        setEmail(inUser.getEmail());
        setPhone(inUser.getPhone());
        setSalutation(inUser.getSalutation());
        setIsEnabled(inUser.isEnabled());
        setPassword(inUser.getPassword());
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
        setUniqueDeviceId(inUser.getUniqueDeviceId());
    }

    // It is sometimes useful to have the corresponding external user data as a
    // hashtable
    public Hashtable getExternalUserData()
    {
    // Create a hash table
    Hashtable externalData = new Hashtable();
    // Get all the external data from fInfo and update the external data
    // externalData = parserUser(fInfo);
        return externalData;
    }

    public Date getLastLogin() {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
       throw new java.lang.UnsupportedOperationException("Method
getLastLogin()
not yet implemented.");
    }

    public Object getAttribute(String param1) {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
      throw new java.lang.UnsupportedOperationException("Method getAttribute()
not yet implemented.");
    }

    public Object getAttribute(String param1, Object parm2) {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
      throw new java.lang.UnsupportedOperationException("Method getAttribute()
not yet implemented.");
    }

    public Hashtable getAttributes() {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
      throw new java.lang.UnsupportedOperationException("Method
getAttributes()
not yet implemented.");
    }

    public void setHasLoggedIn(boolean param1) {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
       throw new java.lang.UnsupportedOperationException("Method setAttribute()
not yet implemented.");
    }
    public boolean hasLoggedIn() {
/**@todo: Implement this com.sun.content.server.service.security.User method*/
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
      throw new java.lang.UnsupportedOperationException("Method isConfirmed()
not yet implemented.");
    }

    public void setAttribute(String param1, Object parm2) {
/**@todo: Implement this com.sun.content.server.service.security.User method*/
      throw new java.lang.UnsupportedOperationException("Method setAttribute()
not yet implemented.");
    }

    public void setAttributes(Hashtable param1) {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
      throw new java.lang.UnsupportedOperationException("Method
setAttributes()
not yet implemented.");
    }

    public String getLoginId() {
        return (String)fInfo.get("loginId");
    }
    public void setLoginId(String param1) {
        if (param1 != null)
        fInfo.put("loginId", param1);
    }

    public String getFirstName() {
        return (String)fInfo.get("firstName");
    }
    public void setFirstName(String param1) {
        if (param1 != null)
        fInfo.put("firstName", param1);
    }

    public String getLastName() {
        return (String)fInfo.get("lastName");
    }
    public void setLastName(String param1) {
        if (param1 != null)
            fInfo.put("lastName", param1);
    }

    public Date getCreateDate() {
            return (Date)fInfo.get("createDate");
    }
    public void setCreateDate(Date param1) {
        if (param1 != null)
            fInfo.put("createDate", param1);
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
    }

    public String getEmail() {
            return (String)fInfo.get("email");
    }
    public void setEmail(String param1) {
        if (param1 != null)
            fInfo.put("email", param1);
    }

    public boolean isConfirmed() {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
        throw new java.lang.UnsupportedOperationException("Method
isConfirmed()
not yet implemented.");
    }

    public void updateLastLogin() throws java.lang.Exception {
 /**@todo: Implement this com.sun.content.server.service.security.User method*/
    throw new java.lang.UnsupportedOperationException("Method
updateLastLogin()
not yet implemented.");
    }

    public String getMiddleName() {
        return (String)fInfo.get("middleName");
    }
    public void setMiddleName(String param1) {
        if (param1 != null)
            fInfo.put("middleName", param1);
    }

    public String getGender() {
        return (String)fInfo.get("gender");
    }
    public void setGender(String param1) {
        if (param1 != null)
            fInfo.put("gender", param1);
    }

    public String getStreet1() {
        return (String)fInfo.get("street1");
    }
    public void setStreet1(String param1) {
        if (param1 != null)
            fInfo.put("street1", param1);
    }
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
    public String getStreet2() {
        return (String)fInfo.get("street2");
    }
    public void setStreet2(String param1) {
        if (param1 != null)
            fInfo.put("street2", param1);
    }

    public String getPostalCode() {
        return (String)fInfo.get("postalCode");
    }
    public void setPostalCode(String param1) {
        if (param1 != null)
            fInfo.put("postalCode", param1);
    }

    public String getCity() {
        return (String)fInfo.get("city");
    }
    public void setCity(String param1) {
    if (param1 != null)
        fInfo.put("city", param1);
    }

    public String getState() {
        return (String)fInfo.get("state");
    }
    public void setState(String param1) {
        if (param1 != null)
            fInfo.put("state", param1);
    }

    public String getCountry() {
            return (String)fInfo.get("country");
    }
    public void setCountry(String param1) {
    if (param1 != null)
        fInfo.put("country", param1);
    }

    public String getPhone() {
        return (String)fInfo.get("phone");
    }
    public void setPhone(String param1) {
        if (param1 != null)
            fInfo.put("phone", param1);
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
    }

    public Date getActivateDate() {
        return (Date)fInfo.get("activateDate");
    }
    public void setActivateDate(Date param1) {
        if (param1 != null)
            fInfo.put("activateDate", param1);
    }

    public Date getDeActivateDate() {
        return (Date)fInfo.get("deactivateDate");
    }
    public void setDeActivateDate(Date param1) {
        if (param1 != null)
            fInfo.put("deactivateDate", param1);
    }

    public String getSalutation() {
        return (String)fInfo.get("salutation");
    }
    public void setSalutation(String param1) {
    if (param1 != null)
        fInfo.put("salutation", param1);
    }

    public boolean isEnabled() {
        return Boolean.valueOf((String)fInfo.get("enabled")).booleanValue();
    }
    public void setIsEnabled(boolean param1) {
        fInfo.put("enabled", String.valueOf(param1));
    }

    public String getPassword() {
        return (String)fInfo.get("password");
    }
    public void setPassword(String param1) {
        if (param1 != null)
            fInfo.put("password", param1);
    }

   public String getUniqueDeviceId(){
            return (String)fInfo.get("UniqueDeviceId");
   }
   public void setUniqueDeviceId(String uniqueId){
            if (uniqueId != null)
                fInfo.put("UniqueDeviceId", uniqueId);
```

CODE EXAMPLE 6-3 Sample `User` Implementation *(Continued)*

```
    }

    public String getMobileId(){
            return (String)fInfo.get("MobileId");
    }
    public void setMobileId(String mobileId) {
            if (mobileId != null)
                fInfo.put("MobileId", mobileId);
    }

    public boolean isPrepay() {
    Boolean isprepay = (Boolean)fInfo.get("isPrepay");
    return isprepay.booleanValue();
    }
    public void setIsPrepay(boolean param1) {
    fInfo.put("isPrepay", new Boolean(param1));
    }

}
```

6.5.3 `SampleUserManagerImpl.java`

The following code is a sample extension of the `UserManager` class.

CODE EXAMPLE 6-4 Example Using the `UserManager` Class

```
package com.sun.content.server.operator.security.adaptor;

import java.util.*;

//import Content Delivery Server libraries
import com.sun.content.server.service.security.*;
import com.sun.content.server.service.security.util.*;

//import here operator required packages
// .......

public class SampleUserManagerImpl extends UserManager{

    private SampleExternalProxy proxy;

    public SampleUserManagerImpl() throws UserProfileResourceException {

    try {
       init();
```

CODE EXAMPLE 6-4 Example Using the `UserManager` Class

```
    } catch (Exception ex) {
           throw new
com.sun.content.server.service.security.util.UserProfileResourceException(
"Failed to instantiate SampleUserManagerImpl ", ex);
    }
    }

    // This method will create an External Directory Server Proxy and
    // initiatlize this User Manager
    private void init() throws UserProfileResourceException
    {
    System.out.println("Initializing External UserManager .... ");

    try {
       proxy = new SampleExternalProxy();
    } catch (Exception ex) {
       System.out.println("Fatal Error " + ex.toString());
       throw new
com.sun.content.server.service.security.util.UserProfileResourceException(
ex.toString());
    }
    }

    protected boolean doIsAuthenticated(String inUserId, String inPassword)
        throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doIsAuthenticated --- ");

    boolean isauthenticated = false;

    try {

       User aUser = proxy.createUserFromExternal(inUserId);
       isauthenticated = ( (aUser.getLoginId()==inUserId) &&
(aUser.getPassword()==inPassword));
       if (isauthenticated)
        System.out.println("SampleUserManagerImpl.doIsAuthenticated: - User
"+inUserId+" is successfully authenticated.");
       else
        System.out.println("SampleUserManagerImpl.doIsAuthenticated: - User
"+inUserId+" : unable to authenticate (wrong login and password)");
    } catch (Exception ex) {
       isauthenticated = false;
       System.out.println("SampleUserManagerImpl.doIsAuthenticated: - User
"+inUserId+" can not authenticate (user not found)");
```

CODE EXAMPLE 6-4 Example Using the `UserManager` Class

```
       throw new
com.sun.content.server.service.security.util.UserProfileResourceException(
ex.toString());
    }
    return isauthenticated;
    }

    protected boolean doAccountExists(String userId) throws
UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doAccountExists --- ");

    // Use the External Proxy search method to check if the account exist
    return proxy.searchUser(userId);
    }

    protected boolean doAddUser(User user) throws UserProfileResourceException{

    boolean updated = false;

    //This is not an allowed operation. If this is not the case add the code to
    // implement it
    System.out.println("SampleUserManagerImpl.doAddUser - This operation is not
implemented !");

    return updated;
    }

    protected void doDisableUser(String userId) throws
UserProfileResourceException {

    //This is not an allowed operation. If this is not the case add the code to
    // implement it
    System.out.println("SampleUserManagerImpl.doDisableUser - This operation is
not implemented !");

    }

    protected void doEnableUser(String userId) throws
UserProfileResourceException {

    //This is not an allowed operation. If this is not the case add the code to
    // implement it
    System.out.println("SampleUserManagerImpl.doEnableUser - This operation is
not implemented !");

    }
```

CODE EXAMPLE 6-4 Example Using the `UserManager` Class

```
    protected User doGetAnonymousUser() throws UserProfileResourceException {
    return new SampleUserImpl();
    }

  protected User doGetUser(String userId) throws UserProfileResourceException {

    User aUser = null;
    try {
       aUser = proxy.createUserFromExternal(userId);
       if (aUser != null)
        System.out.println("SampleUserManagerImpl.doGetUserByUniqueDeviceId: -
User with "+userId+" is found.");
       else
        System.out.println("SampleUserManagerImpl.doGetUserByUniqueDeviceId: -
User with "+userId+" not found");
    } catch (Exception ex) {
       System.out.println("SampleUserManagerImpl.doGetUserByUniqueDeviceId: -
User "+userId+" not found)");
       throw new
com.sun.content.server.service.security.util.UserProfileResourceException(ex);
    }
    return aUser;
    }

    /* This method can be implemented the same way as doGetUser. Instead of
     * using the createUserFromExternal(loginId) you can extend the
     * SampleExternalProxy to implement a specific
     * createUserFromExternalUsingDeviceId(uniqueDeviceId).
     * For now we will assume that the string loginId can be replaced by a search
     * key ID and it will return the matching profile.
     */
    protected User doGetUserByUniqueDeviceId(String inUniqueDeviceId)
              throws UserProfileResourceException {
    return doGetUser(inUniqueDeviceId);
    }

    // Sample implementation as doGetUser
    protected User doGetUserByMobileId(String inMobileId)
              throws UserProfileResourceException {
    return doGetUser(inMobileId);
    }

    protected Iterator doGetAllUsers(String role)
              throws UserProfileResourceException {
```

CODE EXAMPLE 6-4 Example Using the `UserManager` Class

```
    System.out.println("SampleUserManagerImpl.doGetAllUsers - This operation is
not implemented !");
    ArrayList users = new ArrayList();
    return users.iterator();
    }

    protected Iterator doGetAllUsersContainingLastName(String lastName,
               String role) throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doGetAllUsersContainingLastName
- This operation is not implemented !");
    ArrayList users = new ArrayList();
    return users.iterator();
    }

    protected Iterator doGetAllUsersContainingFirstName(String firstName,
String role) throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doGetAllUsersContainingFirstName
- This operation is not implemented !");
    ArrayList users = new ArrayList();
    return users.iterator();
    }

    protected Iterator doGetAllUsersContainingName(String name, String role)
        throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doGetAllUsersContainingName -
This operation is not implemented !");
    ArrayList users = new ArrayList();
    return users.iterator();
    }

    protected Iterator doGetAllUsersContainingId(String userId, String role)
        throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doGetAllUsersContainingId - This
operation is not implemented !");
    ArrayList users = new ArrayList();
    return users.iterator();
    }

    protected Iterator doGetAllLikeInOrder(String[] columns, String[] values,
              String[] orders, boolean isDescending, int pageNum, int recPerPage,
              String role)
         throws UserProfileResourceException {
```

CODE EXAMPLE 6-4 Example Using the `UserManager` Class

```
    System.out.println("SampleUserManagerImpl.doGetAllLikeInOrder - This
operation is not implemented !");
    ArrayList users = new ArrayList();
    return users.iterator();
    }

    protected String doGetFieldName(int fieldContant, String role)
       throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doGetFieldName - This operation
is not implemented !");
    return null;
    }

    protected boolean doIsActive(String userId) throws
UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doIsActive - This operation is
not implemented !");
    return false;
    }

    protected boolean doRemoveUser(String userId)
              throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doRemoveUser - This operation is
not implemented !");
    return false;
    }

    protected boolean doUpdateUser(User user)
              throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doUpdateUser - This operation is
not implemented !");
    return false;
    }

    protected User doGetUserInstance() throws UserProfileResourceException {

    System.out.println("SampleUserManagerImpl.doGetUserInstance - This
operation is not implemented !");
    return null;
    }

}
```

CHAPTER 7

WAP Gateway API

This chapter describes the Sun Java System Content Delivery Server WAP Gateway API. This API retrieves the MSISDN, device profile, and other attributes from the HTTP header. Authentication based on user and password is not required with WAP gateway integration.

FIGURE 7-1 is a simplified representation of the general system components that interact with this API and the access point for it. It also includes additional components that do not interact with this API, but are necessary for an overall understanding of the architecture.

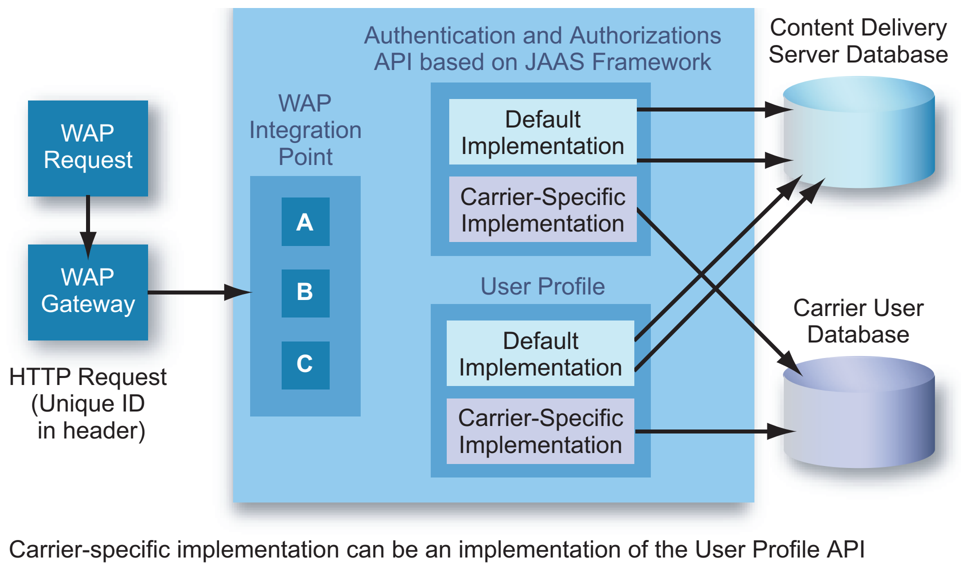
Authentication and Authorizations API based on JAAS Framework
Content Delivery Server Database
WAP Request
WAP Integration Point
Default Implementation
Carrier-Specific Implementation
A
B
C
WAP Gateway
User Profile
Carrier User Database
Default Implementation
Carrier-Specific Implementation
HTTP Request (Unique ID in header)
Carrier-specific implementation can be an implementation of the User Profile API

FIGURE 7-1 WAP Gateway Adapter Architecture

The WAP Gateway adapter parses the HTTP header from a specific WAP Gateway and obtains the MSISDN.

For information on classes or methods not described in this section, see the HTML output of the Javadoc tool for the WAP Gateway API at `$CDS_HOME/javadoc/cdsapi/index.html`.

7.1 `WAPGatewayAdapter` Class

The public abstract `WAPGatewayAdapter` class defines the methods to get the MSISDN and unique device ID from the HTTP header. It also defines a method to check if this method is supported or implemented.

For information on methods not described in this section, see the HTML output of the Javadoc tool for the WAP Gateway API at `$CDS_HOME/javadoc/cdsapi/index.html`.

7.1.1 `doHandle()`

```
public abstract boolean doHandle(String method)
throws WAPGatewayException
```

Returns `true` if `method` is implemented, and returns `false` if `method` is not implemented. This method is used to determine if the other methods for this class are implemented.

7.1.2 `getMSISDN()`

```
public abstract String getMSISDN(HttpServletRequest request)
throws WAPGatewayException
```

Returns the MSISDN as a string after parsing the HTTP header.

7.1.3 `getUniqueId()`

```
public abstract String getUniqueId(HttpServletRequest request) throws
WAPGatewayException
```

Returns the unique device ID as a string after parsing the HTTP header.

7.2 Using the WAP Gateway API

The Content Delivery Server provides an API implementation for the following WAP Gateways.

- Nokia Activ Server 2.0.1
- Nokia Artus WAP Gateway
- Openwave WAP Gateway

A WAP gateway must be configured to forward the MSISDN or the unique device ID to the Content Delivery Server. `WAPGatewayManager` and `WAPGatewayAdapter` classes are available in `cdsapi.jar`, which is in the `$CDS_HOME/deployment/`*deployment-name*`/lib/cdslib` directory. To register a new class that extends the `WAPGatewayAdapter` class for any other WAP gateway, add the class file name to the `wapgateway.config` file inside the `$CDS_HOME/deployment/`*deployment-name*`/conf` directory. The following statement is an example of how to register an adapter for Nokia Activ Server 2.0.1 with the Content Delivery Server.

```
module.gateway.id=
com.sun.content.server.service.gateway.nokia.NokiaActivServerWAPGateway
```

Place your adapter in the `$CDS_HOME/deployment/`*deployment-name*`/lib/external` directory so that the Content Delivery Server can find the file during execution.

7.3 Sample WAP Gateway Adapter

The following code example shows the pseudo code for an adapter for Nokia Activ Server that extends `WAPGatewayAdapter` class.

CODE EXAMPLE 7-1 Example Using the `WAPGatewayAdapter` Class

```
package com.sun.content.server.service.gateway.sample;

import com.sun.content.server.service.gateway.WAPGatewayAdapter;
import com.sun.content.server.service.gateway.WAPGatewayException;
import javax.servlet.http.HttpServletRequest;

public class SampleWAPGateway extends WAPGatewayAdapter
{
```

CODE EXAMPLE 7-1 Example Using the `WAPGatewayAdapter` Class *(Continued)*

```
  /* Method to check if the passed method is implemented in this
   * class or not. */
  public boolean doHandle(String method) throws WAPGatewayException
  {
    if (method.equals("getMSISDN"))
      return true;
    return false;
  }

  /* Gets the MSISDN from the header and returns as a string. */
  public String getMSISDN(HttpServletRequest request)
  {
    return request.getHeader("<key to retrieve>");
  }

  /* This method is not implemented. */
  public String getUniqueId(HttpServletRequest req)
throws WAPGatewayException
  {
    throw new WAPGatewayException("This method is not implemented");
  }
}
```

CHAPTER 8

Messaging API

This chapter describes the Sun Java System Content Delivery Server Messaging API. The Messaging API provides a mechanism for carriers or application vendors to integrate their WAP, SMS, and MMS push implementations with the Content Delivery Server by creating a push adapter. The Messaging API is used to create both push sender adapters and push listener adapters.

The following figure illustrates the high-level architecture of the Messaging API and the different components that use this API.

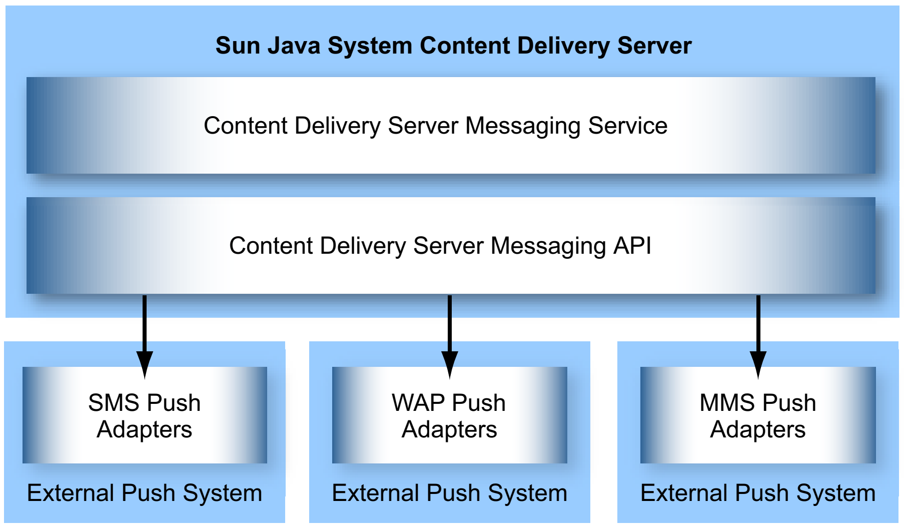
Sun Java System Content Delivery Server
Content Delivery Server Messaging Service
Content Delivery Server Messaging API
SMS Push Adapters
External Push System
WAP Push Adapters
External Push System
MMS Push Adapters
External Push System

FIGURE 8-1 Architecture of the Messaging API

An external push system can interact or receive push messages from the Content Delivery Server.

For information on classes or methods not described in this section, see the HTML output of the Javadoc tool for the Messaging API at `$CDS_HOME/javadoc/cdsapi/index.html`.

8.1 `PushMsgSender` Interface

The `PushMsgSender` interface declares the methods needed for a push message sender implementation. This interface is called by the Content Delivery Server Messaging Service when the Content Delivery Server sends a push message.

8.1.1 `pushMessage()`

```
public PushResponse pushMessage(PushMessage msg, int retryNum)
throws PushMessageFailException;
```

Sends the message. This method is called when the Content Delivery Server sends a push message.

8.2 `PushMsgListener` Interface

The `PushMsgListener` interface declares the methods needed for a push message listener implementation.

8.2.1 `connect()`

```
public boolean connect()
```

Returns `true` if the Content Delivery Server is connected to the SMSC, otherwise, returns `false`. This method is called by Content Delivery Server Messaging Service when the SMSC sends a push message.

8.2.2 `initialize()`

```
public void initialize(String pushActionType) throws
InitializationFailedException
```

Initializes SMSC parameters before making a connection to the SMSC. This is the first method called by the Content Delivery Server Messaging Service.

8.2.3 `listen()`

```
public void listen()
```

Listens for the message coming from the device. This method is called by Content Delivery Server Messaging Service after `PushMsgListener` is connected to the SMSC.

8.2.4 `sendKeepAliveMsg()`

```
public void sendKeepAliveMsg()
```

Sends signal to keep message alive. This method is called by the Content Delivery Server Messaging Service if the SMSC requires that a “Keep Alive” signal be sent to keep the connection active.

8.3 `PushMessage` Class

`PushMessage` is the base class for all of the different types of push messages generated by the Content Delivery Server. The `MMSPushMessage` class used for MMS messages extends `PushMessage`.

All of the `set` methods are used by the Content Delivery Server during message construction. All of the `get` methods can be used by a sender’s implementation of a push message.

8.3.1 `addUserAgent()`

```
public void addUserAgent(String ua)
```

Sets the user agent for the receiver’s device.

8.3.2 `getAllUserAgents()`

`public ArrayList getAllUserAgents()`

Returns all user agents for the receiver's device.

8.3.3 `getAttribute()`

`public Object getAttribute(String attributeName)`

Returns any message attribute.

8.3.4 `getContentBinary()`

`public Object getContentBinary()`

Gets the binary content included in the message.

8.3.5 `getContentType()`

`public String getContentType()`

Gets the content type of a message.

8.3.6 `getDestinationAddress()`

`public String getDestinationAddress()`

Returns the destination address, which is the phone number or email ID of the receiver.

8.3.7 `getJMSMessageId()`

This method is deprecated. Use `getMessageId()`.

8.3.8 getKeyword()

```
public String getKeyword()
```

Returns the keyword that is associated with the requested content.

8.3.9 getMessageId()

```
public String getMessageId()
```

Returns the ID of the message.

8.3.10 getMessageText()

```
public String getMessageText()
```

Returns the message text.

8.3.11 getMimeType()

```
public String getMimeType()
```

Gets the MIME type.

8.3.12 getPushCategory()

```
public long getPushCategory()
```

Returns the push message category.

8.3.13 getPushDomain()

```
public long getPushDomain()
```

Returns the domain of the push message.

8.3.14 getPushType()

```
public String getPushType()
```

Returns the message type.

8.3.15 getShortCode()

```
public String getShortCode()
```

Returns the short code to which the message is sent.

8.3.16 getSubscriberId()

```
public long getSubscriberId()
```

Returns the subscriber ID.

8.3.17 getUniqueDeviceId()

```
public String getUniqueDeviceId()
```

Returns the unique device ID.

8.3.18 getVendingContentId()

```
public long getVendingContentId()
```

Returns the vending content ID.

8.3.19 setAllUserAgents()

```
public void setAllUserAgents(ArrayList list)
```

Sets multiple user agents for the receiver's device.

8.3.20 `setAttribute()`

```
public void setAttribute(String attributeName, Object attributeVal)
```

Sets any message attribute.

8.3.21 `setContentBinary()`

```
public void setContentBinary(byte[] content)
```

Sets the content binary included in the message.

8.3.22 `setContentType()`

```
public void setContentType(String contentType)
```

Sets the content type of the message.

8.3.23 `setDestinationAddress()`

```
public void setDestinationAddress(String receiverId)
```

Sets the destination address of the receiver of the push message. The destination address can be either a phone number or email ID.

8.3.24 `setJMSMessageId()`

This method is deprecated. Use `setMessageId()`.

8.3.25 `setKeyword()`

```
public void setKeyword(String inputKeyword)
```

Sets the keyword that is associated with the requested content.

8.3.26 `setMessageId()`

```
public void setMessageId(String msgId)
```

Sets the ID of the message.

8.3.27 `setMessageText()`

```
public void setMessageText(String text)
```

Sets the message text.

8.3.28 `setMimeType()`

```
public void setMimeType(String mType)
```

Sets the MIME type.

8.3.29 `setPushCategory()`

```
public void setPushCategory(long category)
```

Sets the push message category.

8.3.30 `setPushDomain()`

```
public void setPushDomain(long domain)
```

Sets the domain of the push message.

8.3.31 `setShortCode()`

```
public void setShortCode(String inputShortCode)
```

Sets the short code to which the message is sent.

8.3.32 `setSubscriberId()`

```
public void setSubscriberId(long subId)
```

Sets the subscriber ID.

8.3.33 `setUniqueDeviceId()`

```
public void setUniqueDeviceId(String uniqueId)
```

Sets the unique ID of the device.

8.3.34 `setVendingContentId()`

```
public void setVendingContentId(long contentId)
```

Sets the vending content ID.

8.3.35 `toString()`

```
public String toString()
```

Displays the `PushMessage` object's data to the log. This method is mainly for debugging purposes.

8.4 `SMSMessage` Class

This class is deprecated. The following methods are now part of `PushMessage` class:

- `getContentType`
- `getMIMEType`
- `setContentType`
- `setMIMEType`

To get and set the content name or download URL, use the `getAttribute()` and `setAttribute()` methods in `PushMessage` with the appropriate constant defined in `PushConstants`. See the output of the Javadoc tool for these classes at `$CDS_HOME/javadoc/cdsapi/index.html` for more information.

8.5 WapPushMessage Class

This class is deprecated. To get and set the download URL, use the getAttribute() and setAttribute() methods in PushMessage with the appropriate constant defined in PushConstants. See the output of the Javadoc tool for these classes at $CDS_HOME/javadoc/cdsapi/index.html for more information.

8.6 SMTPMessage Class

This class is deprecated. To get and set the subject or message's From address, use the getAttribute() and setAttribute() methods in PushMessage with the appropriate constant defined in PushConstants. See the output of the Javadoc tool for these classes at $CDS_HOME/javadoc/cdsapi/index.html for more information.

8.7 ContentSlide Class

This class stores the binary data from the MMS push message. The binary data can have a MIME type and a unique ID associated with it. All of the set methods are used by the server during message construction. All the get methods can be used by an implementation of a push message.

8.7.1 getContentData()

```
public byte[] getContentData()
```

Returns the MMS push message's binary data.

8.7.2 getContentId()

```
public String getContentId()
```

Returns the unique ID associated with an MMS push message's binary data.

8.7.3 `getContentMimeType()`

```
public String getContentMimeType()
```

Returns the MIME type of an MMS push message’s binary data.

8.7.4 `setContentData()`

```
public void setContentData(byte[] contentData)
```

Sets the MMS push message’s binary data.

8.7.5 `setContentId()`

```
public void setContentId(String contentId)
```

Sets the unique ID associated with an MMS push message’s binary data.

8.7.6 `setContentMimeType()`

```
public void setContentMimeType(String contentMimeType)
```

Sets the MIME type of an MMS push message’s binary data.

8.8 `MMSSlide` Class

This class is a wrapper for `ContentSlide` objects and is used to construct MMS push messages. All of the `set` methods are used by the Content Delivery Server during message construction. All the `get` methods can be used by an implementation of a push message.

8.8.1 `getAudioContent()`

```
public ContentSlide getAudioContent()
```

Returns the audio content of an `MMSSlide` object.

8.8.2 getImageContent()

```
public ContentSlide getImageContent()
```

Returns the image content of an `MMSSlide` object.

8.8.3 getTextContent()

```
public ContentSlide getTextContent()
```

Returns the text content of an `MMSSlide` object.

8.8.4 getVideoContent()

```
public ContentSlide getVideoContent()
```

Returns the video content of an `MMSSlide` object.

8.8.5 setAudioContent()

```
public void setAudioContent(String contentId, String contentMimeType,
byte[] contentData)
```

Sets the audio content of an `MMSSlide` object.

8.8.6 setImageContent()

```
public void setImageContent(String contentId, String contentMimeType,
byte[] contentData)
```

Sets the image content of an `MMSSlide` object.

8.8.7 setTextContent()

```
public void setTextContent(String contentId, String contentMimeType,
byte[] contentData)
```

Sets the text content of an `MMSSlide` object.

8.8.8 `setVideoContent()`

`public void setVideoContent(String contentId, String contentMimeType, byte[] contentData)`

Sets the video content of an `MMSSlide` object.

8.9 `MMSPushMessage` Class

The `MMSPushMessage` class extends `PushMessage` and represents an MMS push message. All of the `set` methods are used by the Content Delivery Server during message construction. All of the `get` methods can be used by an implementation of a push message. This class contains the From address, the To addresses, MMSC-related data, the user agent, and any Synchronized Multimedia Integration Language (SMIL) data, if available. It also encapsulates the `MMSSlide` object.

8.9.1 `addMMSSlide()`

`public void addMMSSlide(MMSSlide mmsSlide)`

Sets the `MMSSlide` object.

8.9.2 `addRecipient()`

`public void addRecipient(String to)`

Sets the receiver's phone number or email ID.

8.9.3 `getAllMMSSlides()`

`public ArrayList getAllMMSSlides()`

Returns all of the `MMSSlide` objects associated with this `MMSPushMessage`.

8.9.4 `getAllRecipients()`

`public ArrayList getAllRecipients()`

Returns the phone number or email ID of all of the recipients of the message.

8.9.5 `getDeliveryReportRequired()`

```
public boolean getDeliveryReportRequired()
```

Returns the value for `DeliveryReportRequired` attribute.

8.9.6 `getFromAddress()`

```
public String getFromAddress()
```

Returns the sender's phone number or email ID.

8.9.7 `getMessageClass()`

```
public String getMessageClass()
```

Returns the value for `MessageClass` attribute.

8.9.8 `getMessagePriority()`

```
public String getMessagePriority()
```

Returns the value for `MessagePriority` attribute.

8.9.9 `getReadReportRequired()`

```
public boolean getReadReportRequired()
```

Returns the value for `ReadReportRequired` attribute.

8.9.10 `getSenderVisibility()`

```
public String getSenderVisibility()
```

Returns the value for `SenderVisibility` attribute.

8.9.11 getSMILPresentation()

```
public byte[] getSMILPresentation()
```

Returns the SMIL data.

8.9.12 setDeliveryReportRequired()

```
public void setDeliveryReportRequired(boolean
deliveryReportRequired)
```

Sets the value for `DeliveryReportRequired` attribute.

8.9.13 setFromAddress()

```
public void setFromAddress(String from)
```

Sets the sender's phone number or email ID.

8.9.14 setMessageClass()

```
public void setMessageClass(String messageClass)
```

Sets the value for `MessageClass` attribute.

8.9.15 setMessagePriority()

```
public void setMessagePriority(String messagePriority)
```

Sets the value for `MessagePriority` attribute.

8.9.16 setReadReportRequired()

```
public void setReadReportRequired(boolean readReportRequired)
```

Sets the value for the `ReadReportRequired` attribute.

8.9.17 `setSenderVisibility()`

```
public void setSenderVisibility(String senderVisibility)
```

Sets the value for `SenderVisibility` attribute.

8.9.18 `setSMILPresentation()`

```
public void setSMILPresentation(byte[] smil)
```

Sets the SMIL data.

8.10 `MMSSender` Interface

The `MMSSender` interface declares the method that sends an MMS message. This interface must be implemented for the vendor-specific MMSC during integration if you want to support MMS messages.

8.10.1 `sendMMS()`

```
sendMMS(com.sun.content.server.server.messaging.message.MMSPushMess
age message)
```

Encapsulates the functionality for sending an MMS message. This method is called by the Content Delivery Server Messaging Service when it receives an `MMSPushMessage` from the Content Delivery Server.

The Content Delivery Server sends an `MMSPushMessage` object to the Content Delivery Server Messaging Service. The Messaging Service uses the value of the `mms.senderclass` property in the `MsgService.properties` file to identify the fully qualified name of the class that provides the vendor-specific implementation of `MMSSender.sendMMS`. The code within the implementation of `sendMMS` transforms the `MSSPushMessage` object into a vendor-specific version of the MMS message object and sends it to the vendor-specific MMSC for processing. The following figure shows this process.

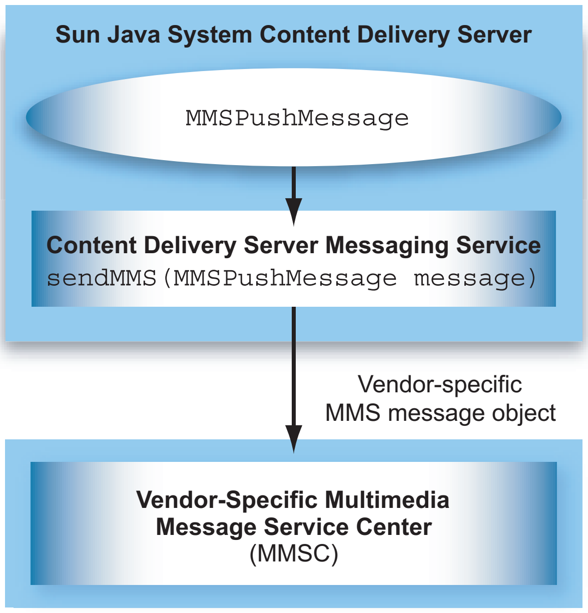
Sun Java System Content Delivery Server
MMSPushMessage
Content Delivery Server Messaging Service
sendMMS(MMSPushMessage message)
Vendor-specific MMS message object
Vendor-Specific Multimedia Message Service Center (MMSC)

FIGURE 8-2 Process Flow for Sending an MMS Message

To implement a vendor-specific version of `MMSSender.sendMMS`, include the following items:

1. Connect to the vendor-specific MMSC.
2. Implement `MMSSender.sendMMS` to do the following tasks:
   - Transform the `MMSPushMessage` object that is passed to the Content Delivery Server Messaging Service into a vendor-specific MMS message object.
   - Send the new message object to the vendor-specific MMSC.

8.11 `PushResponse` Class

The `PushResponse` class is the base class for all of the different types of push responses generated by the external push services. All of the `set` methods are used by an implementation of a push message sender. All of the `get` methods can be used by the Content Delivery Server to log the push message in the database.

8.11.1 `getMessageId()`

```
public String getMessageId()
```

Returns the message ID.

8.11.2 `getResponseDescription()`

```
public String getResponseDescription()
```

Returns the response description.

8.11.3 `getResponseStatus()`

```
public String getResponseStatus()
```

Returns the response status.

8.12 `PushConstants` Class

The `PushConstants` class contains all of the constants that the push services can support. A specific push service implementation compares these constants with the values received from the `PushMessage` object. See the output of the Javadoc tool at `$CDS_HOME/javadoc/cdsapi/index.html` for information on the constants defined.

8.13 Using the Messaging API

A push sender adapter must implement the `PushMsgSender` interface. During deployment, the implementation class needs to be registered with the Content Delivery Server. An XML file is used for registration. This XML file is located in the `$CDS_HOME/deployment/`*deployment-name*`/conf` directory and is named `pushsenderfactory.xml`.

The following example shows the structure of this file.

CODE EXAMPLE 8-1 Sample `pushsenderfactory.xml` File

```
<?xml version="1.0" encoding='UTF-8' ?>
<PushSenderConfig nodeid="0">
  <pushmsgsenderset nodeid ="1">
    <pushmsgsender nodeid ="2" class =
"com.sun.content.server.server.msgserver.push.HTTPSMSPushMsgSender" protocol="sms"/>
    <pushmsgsender nodeid ="3" class =
"com.sun.content.server.server.msgserver.push.WAPPushMsgSender" protocol="wap"/>
    <pushmsgsender nodeid ="4" class =
"com.sun.content.server.server.msgserver.push.SMTPushMsgSender" protocol="smtp"/>
    <pushmsgsender3 class =
"com.sun.content.server.server.msgserver.push.MMSPushMsgSender" protocol="mms"/>
</pushmsgsenderset>
</PushSenderConfig>
```

Four adapters are registered in this file. Specify the fully qualified class name and the protocol that the adapter supports. For example, if you have an adapter for SMS push, the protocol is `sms`. Make sure the adapter class and dependent classes are set in the classpath.

If you are using the default implementation of a push sender adapter for MMS, `MMSPushMsgSender`, you must implement the `MMSSender` interface. Set the `mms.senderclass` property in the `MsgServices.properties` file to the fully qualified name of your class. This class is in the `$CDS_HOME/deployment/`*deployment-name*`/conf` directory.

A push listener adapter implements the `PushMsgListener` interface. The implementation class needs to be registered with the Content Delivery Server. An XML file is used for registration. This XML file is located in the `$CDS_HOME/deployment/`*deployment-name*`/conf` directory and is named `pushlistenerfactory.xml`.

The following example shows the structure of this file.

CODE EXAMPLE 8-2 Sample `pushlistenerfactory.xml` File

```
<pushmsglistenerset>
    <pushmsglistener0 class=
"com.sun.content.server.server.msgserver.protocol.cimd2.CIMD2PushMsgListener"
protocol="sms"/>
</pushmsglistenerset>
```

CHAPTER **9**

Confirm Service API

The Sun Java System Content Delivery Server Confirm Service API enables the Content Delivery Server to handle confirmation messages sent from a Multimedia Messaging Service Center (MMSC). Confirmation messages are generally sent after content is downloaded to a device.

The Confirm Service API consists of the following classes:

- `ConfirmServiceAdapter` - Abstract class that you extend to connect to the MMSC and listen for messages.
- `ConfirmResponse` - Class that contains the confirmation information received.
- `ConfirmServiceException` - Exception that is thrown by the Confirm Service API.

For additional information on these classes, see the HTML output of the Javadoc tool at `$CDS_HOME/javadoc/cdsapi/index.html`.

9.1 General Process Flow

The Content Delivery Server can send content in a multimedia message to devices that support the MMS standard. When a device receives content in an MMS message, a confirmation message is returned through the MMSC. The confirm service adapter that you write using the Confirm Service API sets up the connection between the Content Delivery Server and the MMSC, and handles the confirmation messages from the MMSC.

9.2 `ConfirmServiceAdapter` Class

The `ConfirmServiceAdapter` class establishes the connection with the MMSC, listens for confirmation messages, and passes the messages received to the Content Delivery Server. Extend `ConfirmServiceAdapter` to create a confirm service adapter for your system.

The `ConfirmServiceAdapter` class is in the `com.sun.content.server.confirmservice` package.

9.2.1 `connect()`

```
public abstract boolean connect() throws ConnectionFailedException
```

Use this method to connect to the Content Delivery Server to the MMSC that you are using.

9.2.2 `listen()`

```
public abstract void listen() throws ConfirmServiceException
```

Use this method to listen for confirmation messages from the MMSC. When a confirmation message is received, use the information in the message to create a `ConfirmResponse` object and call the `messageReceived()` method.

The `ConfirmResponse` object requires the information shown in the following table.

TABLE 9-1 `ConfirmResponse` Parameters

Parameter	**Description**
`pushType`	The type of message received. The value must be one of the types defined in the `PushConstants` class (see Section 8.12, "`PushConstants` Class" on page 8-18.)
`messageID`	The ID assigned by the MMSC to identify the message.
`responseStatus`	The status of the response.
`responseDescription`	The description of the response.
`responseObject`	Not used for MMS messages. Pass null.

9.2.3 `messageReceived()`

```
protected void messageReceived(ConfirmResponse confirmResponse)
throws ConfirmServiceException
```

Use this method to send the information received in the confirmation message to the Content Delivery Server. Call this message from your implementation of the `listen()` method.

9.3 Using the Confirm Service API

Modify the `ConfirmListener.properties` file in the `$CDS_HOME/deployment/`*deployment-name*`/conf` directory to add another instance of the following property:

```
confirmservice.class.id
```

Set this property to the class name of your implementation of the `ConfirmServiceAdapter` class, for example:

```
confirmservice.class.id=
com.sun.content.server.server.confirm.mms.MMSConfirmService
```

The classes for the Confirm Service API are available in `cdsapi.jar`, which is found in the `$CDS_HOME/deployment/`*deployment-name*`/lib/cdslib` directory.

The `cdsapi.jar` file must be in your classpath when you compile your adapter.

To make your adapter available to the Content Delivery Server, follow these steps:

1. **Create a JAR file for your adapter.**
2. **Place the JAR file in the** `$CDS_HOME/deployment/`*deployment-name*`/lib/external` **directory.**
3. **Open the** `ConfirmListener.properties` **file in the** `$CDS_HOME/deployment/`*deployment-name*`/conf` **directory.**
4. **Add another instance of the** `confirmservice.class.id` **property.**

   Set this property to the class name of your implementation of the `ConfirmServiceAdapter` class, for example:

   ```
   confirmservice.class.id=
   com.sun.content.server.server.confirm.mms.MMSConfirmService
   ```
5. **Save your changes.**
6. **Restart the Content Delivery Server to make it aware of the new JAR file.**

CHAPTER **10**

Subscriber API

The Subscriber API provides access to data maintained by the Content Delivery Server. Use this API to get the data needed to create your own client application for providing subscribers with access to content managed by the Content Delivery Server.

Clients that are written in the Java programming language and located on a server with the Subscriber Portal component of the Content Delivery Server can call the Subscriber API directly. Clients written in a language other than Java, or clients that are located on a server that does not contain the Subscriber Portal must access the Subscriber API through the XML-RPC (remote procedure call) implementation.

The Subscriber API includes the following classes and interfaces that your client might use:

- `ApiContextFactory` - Class that creates an `IApiContext` object that contains the characteristics of the subscriber, such as locale, device model, and mobile ID.
- `ApiServiceFactory` - Class that creates a service for a specific subscriber. Services provide access to a collection of related information. For example, the Content Service provides access to the list of previous purchases made by a subscriber, the details of individual items, and a subscriber's bookmarked contents.
- `ApiUtil` - Class that provides utilities such as initiating a transaction or checking the version of the database.
- `IApiContext` - Interface that provides information about the current subscriber.
- `ICategoryService` - Interface that provides access to the category tree and a subscriber's category list.
- `IContentService` - Interface that provides access to content for browsing, searching, retrieving, and purchasing.
- `IDownloadService` - Interface that provides access to content descriptors.
- `IGiftingService` - Interface that enables a subscriber to send gifts or messages about an item to another subscriber.

- `IMessageService` - Interface that enables messages to be sent to this subscriber or another subscriber. Messages can be in email, MMS, SMS, or WAP push formats.
- `ISystemService` - Interface that provides access to system-level content such as locales, content types, and device models.
- `IUserService` - Interface that provides access to information on subscribers and enables new subscriber accounts to be created.
- `CDSException` - Exception thrown by the Subscriber API if an error occurs.

For additional information on these and other classes, see the HTML output of the Javadoc tool at `$CDS_HOME/javadoc/subscriberapi/index.html`.

10.1 General Process Flow

This section describes general tasks that you might want your client application to perform. The class names used in this section refer to the classes that make up the Subscriber API. See Section 10.3, "XML-RPC Implementation" on page 10-6 for the equivalent handler if you are accessing the Content Delivery Server data using XML-RPC.

In general, the client application that you create for your subscriber interface includes actions such as those described in the following list.

- Create an `IApiContext` object by calling `ApiContextFactory`.

  The `IApiContext` object describes either a specific subscriber or an anonymous subscriber. This object is used by all services to retrieve data specific to the subscriber described by the object. Typically, you would create the `IApiContext` object once per user session and store it in the `HttpSession` object. See Section 10.2.2, "Example of Creating an `IApiContext` Object" on page 10-4 for sample code.

- Create the services that you need to obtain the information that you want to use by calling the `ApiServiceFactory`.

  The services that you create depend on the tasks that you want to perform. For example, to enable a subscriber to purchase a gift for another subscriber, create an `IGiftingService` object. To provide a subscriber with the list of content already purchased, create an `IContentService` object. You must have an `IApiContext` object to create a service. Each service created provides data specific to the subscriber described by the `IApiContext` object. See Section 10.2.3, "Example of Creating a Service" on page 10-5 for sample code.

- Retrieve the information that you want to use by calling the methods for the services that you created.

  For example, if you created an `IContentService`, call `getCampaigns()` to get the list of campaigns available to the subscriber or `getPurchases()` to get the list of items that the subscriber has already purchased.

10.2 Using the Subscriber API

The classes for the Subscriber API are in the package `com.sun.content.server.subscriberapi`. This package is included in the `subscriberportal.jar` file in one of the following locations:

- For Sun Java System Application Server, in `$CDS_HOME/deployment/`*deployment-name*`/sun/domains/`*server-domain*`/`*server-name*`/applications/j2ee-modules/CDSSubscriberPortal/WEB-INF/lib`.
- For WebLogic Server, in `$CDS_HOME/deployment/`*deployment-name*`/weblogic/domains/`*server-domain*`/applications/subscriber/WEB-INF/lib`.

deployment-name is the name specified when the Catalog Manager was deployed, *server-domain* is the value specified in the deployment configuration file for the `app.server.domain` property and *server-name* is the value specified in the deployment configuration file for the `app.server.name` property.

If your client application is a Java application, create your client using the Subscriber API classes in the `subscriberportal.jar` file. This JAR file must be in your classpath when you compile your application.

To execute, place the JAR file that contains your client in the same `$CDS_HOME/deployment/`*deployment-name*`/.../lib` directory that contains the `subscriberportal.jar` file. Your client must run within the same web application structure as the Subscriber Portal provided with the Content Delivery Server. Stand-alone Java applications are not supported.

If your client is not a Java application or will not be located on the same server as the Subscriber Portal, see Section 10.3, "XML-RPC Implementation" on page 10-6 for information on accessing the Subscriber API through XML-RPC.

10.2.1 Managing Transactions

A transaction occurs each time you create an instance of a service and call its methods to perform a task. Your application must manage the transactions with the Content Delivery Server as described in the following steps:

1. **Before a service is invoked, call** `ApiUtil.initTransaction()` **to indicate the start of a transaction.**
2. **If the transaction completes successfully, call** `ApiUtil.commitTransaction()` **to commit the work done. If an error occurs during the transaction, call** `ApiUtil.rollbackTransaction()` **to terminate the work and restore the data to its previous state.**
3. **The transaction resources must be released in a** `finally` **block that calls** `ApiUtil.disposeTransaction()`.

   See the code examples in Section 10.2.2, "Example of Creating an `IApiContext` Object" on page 10-4 and Section 10.2.3, "Example of Creating a Service" on page 10-5 for sample implementations.

10.2.2 Example of Creating an `IApiContext` Object

The following code excerpt shows how to create an `IApiContext` object.

CODE EXAMPLE 10-1 Create an `IApiContext` Object Using Java Classes

```
...
try
{
  // Open a Transaction
  ApiUtil.initTransaction();

  // Create a map of credentials (from user input)
  Properties credentials = new Properties();
  credentials.put(ApiContextFactory.CREDENTIAL_USERNAME, username);
  credentials.put(ApiContextFactory.CREDENTIAL_PASSWORD, password);

  // Attempt to authenticate using the credentials
  IApiContext apiContext = ApiContextFactory.createApiContext(credentials);

  // Save the IApiContext in the HttpSession
  session.setAttribute("API_CONTEXT", apiContext);

  // Commit the Transaction
  ApiUtil.commitTransaction();
}
```

CODE EXAMPLE 10-1 Create an `IApiContext` Object Using Java Classes *(Continued)*

```
catch (CDSException e)
{
  // Rollback the Transaction
  ApiUtil.rollbackTransaction();

  // Evaluate the exception's error code
  if(e.getErrorCode().equals(CDSException.CDS_EX_SUBSCRIBER_DISABLED)
  {
    // handle disabled user
    ...
  }
  else
  {
    // handle API Exception
    ...
  }
}
finally
{
  // clean up Transaction
  ApiUtil.disposeTransaction();
}
...
```

10.2.3 Example of Creating a Service

The following code excerpt shows how to create a Content Service and use that service to purchase content.

CODE EXAMPLE 10-2 Create a Service

```
...
try
{
  // Open a Transaction
  ApiUtil.initTransaction();

  // Retrieve the IApiContext from the HttpSession
  IApiContext apiContext = (IApiContext) session.getAttribute("API_CONTEXT");

  // Get a reference to a Content Service
  IContentService cs = ApiServiceFactory.getContentService(apiContext);

  // Attempt to purchase a content item as part of a campaign
  cs.purchaseContent(contentId, campaignId, true);
```

CODE EXAMPLE 10-2 Create a Service *(Continued)*

```
  // Commit the Transaction
  ApiUtil.commitTransaction();
}
catch (CDSException e)
{
  // Rollback the Transaction
  ApiUtil.rollbackTransaction();

  // Handle API Exception
  ...
}
finally
{
  // Clean up Transaction
  ApiUtil.disposeTransaction();
}
...
```

10.3 XML-RPC Implementation

If your client is not a Java application or is not running on the server with Content Delivery Server, your client must communicate with the Content Delivery Server using XML-RPC. XML-RPC enables your client to make remote procedure calls using HTTP for the transport and XML for data encoding. You can use XML-RPC with many different programming languages by using bindings available on the Internet. All of the functionality of the Subscriber API is available through XML-RPC.

Note – A tutorial on XML-RPC is beyond the scope of this document. Information on writing applications that use XML-RPC is available from various web sites on the Internet.

10.3.1 Accessing the Content Delivery Server

To obtain data from the Content Delivery Server, your client must be able to communicate with the Content Delivery Server. Work with your network administrator to ensure that the client can contact the Content Delivery Server and that any required proxy or firewall is configured to allow this access.

In addition, the Content Delivery Server must recognize that your client is authorized to make requests for data. The `subscriberApi.xml-rpc.trustedHosts` property in the `$CDS_HOME/deployment/`*deployment-name*`/conf/SubscriberPortal.properties` file contains the list of hosts from which requests are accepted.

Set the `subscriberApi.xml-rpc.trustedHosts` property to the host name or IP address of the host on which your client is located, whether it is on the same host as the Content Delivery Server or on a different host. To accept requests from any host, leave the value blank. To accept requests from more than one host, separate the host names or IP addresses with a comma, for example:

```
subscriberApi.xml-rpc.trustedHosts=127.0.0.1,localhost
```

10.3.2 Using XML-RPC Handlers for the Subscriber API

The general flow of your application (see Section 10.1, "General Process Flow" on page 10-2) is the same whether you access the Subscriber API directly or through XML-RPC. Handlers in the XML-RPC implementation perform the same functions as the Subscriber API services. Each handler and its corresponding service have equivalent methods with equivalent parameters.

The following topics are presented in this section:

- Guidelines for Calls to XML-RPC Methods
- `AuthenticationHandler`
- `CategoryHandler`
- `ContentHandler`
- `DownloadHandler`
- `GiftingHandler`
- `MessageHandler`
- `SystemHandler`
- `UserHandler`
- Parameters for the Methods

10.3.2.1 Guidelines for Calls to XML-RPC Methods

Use the following guidelines to code your calls to the handlers. The sample code is written in the Java programming language.

- Place the parameters for the method that you are calling in a hash table. Place the hash table in a vector.

```
...
// Set up the input parameters
Vector parameters = new Vector();
Hashtable ht = new Hashtable();
ht.put("username", "user1");
ht.put("password", "cryptic1");
parameters.addElement(ht);
...
```

- To call a method, pass the name of the method and the vector that you created for the hash table that contains the parameters. A hash table is returned. The method call must include the name of the handler, for example: `AuthenticationHandler.getApiContext`.

```
...
// Send the request to Content Delivery Server
Hashtable response =
(Hashtable) client.execute("AuthenticationHandler.getApiContext",
parameters);
...
```

- Verify that the method executed successfully by checking the response code that is included in the hash table that is returned. If an error occurred, a response message is also included in the hash table.

```
...
// Evaluate the response
String errorCode = (String)response.get("response_code");
if (!errorCode.equals("1"))
{
  // Handle Error
  System.out.println((String)response.get("response_message"));
}
```

- If the method executes successfully, extract the values that are returned in the hash table. If a method does not return any values, the hash table contains only the response code.

```
...
  // Authentication successful
  Hashtable apiContext = (Hashtable)response.get("apiContext");
  Integer subscriberId = (Integer)apiContext.get("subscriberId");
  String username = (String)apiContext.get("username");
  String localeCode = (String)apiContext.get("localeCode");
  String mobileId = (String)apiContext.get("mobileId");
  Integer modelId = (Integer)apiContext.get("modelId");
  ...
```

The following sections describe the handlers that you can use. Examples are provided in Section 10.3.3, "Examples of Using Handlers" on page 10-42.

10.3.2.2 AuthenticationHandler

AuthenticationHandler is equivalent to the APIContextFactory class. This handler creates the object that contains the characteristics of the subscriber. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

AuthenticationHandler.getApiContext

The following table describes the AuthenticationHandler methods.

TABLE 10-1 Methods for AuthenticationHandler

Method Name	**Description**	**Parameters**	**Returns**[1]
getAnonymousApiContext	Create an APIContext object for an anonymous subscriber.	localeCode, modelId	apiContext
getApiContext	Authenticate the subscriber based on the information provided and create an APIContext object that contains the information for that subscriber.	One of the following items: • username and password • mobileId • uniqueId • subscriberId • requestHeaders	apiContext

[1] In addition to the objects listed, all methods return a response_code and, if an error occurs, a response_message.

10.3.2.3 CategoryHandler

CategoryHandler is equivalent to the ICategoryService interface. This handler provides access to the category tree and a subscriber's category list. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
CategoryHandler.getCategory
```

The following table describes the CategoryHandler methods.

TABLE 10-2 Methods for CategoryHandler

Method Name	Description	Parameters	Returns[1]
getCategory	Get the specified category for the current subscriber.	apiContext, categoryId, includeContentCount	category
getCategoryBranch	Get an entire branch of the category tree beginning with the category specified in categoryId. To get the entire tree, specify the root category.	apiContext, categoryId, includeContentCount	category
getNotEmptySubCategories	For the category specified in categoryId, get the list of subcategories that contain the type of content listed in contentTypeIdList with a status listed in statusList. If contentTypeIdList is not provided, all content types are considered. If statusList is not provided, all statuses are considered.	apiContext, categoryId, contentTypeIdList (optional), statusList (optional), includeContentCount	categoryList
getSubCategories	For the category specified in categoryId, get the list of all subcategories.	apiContext, categoryId, includeContentCount	categoryList
getRootCategory	Get the root category for the current subscriber.	apiContext, includeContentCount	category
hideCategory[2]	Hide the categories specified in categoryIds that the subscriber has chosen to not view and return the updated list.	apiContext, categoryIds, categoryList	categoryList

TABLE 10-2 Methods for `CategoryHandler` *(Continued)*

Method Name	Description	Parameters	Returns[1]
`moveCategoryDown`[2]	Move each category specified in `categoryIds` down one position below the next active category and return the updated list.	`apiContext`, `categoryIds`, `categoryList`	`categoryList`
`moveCategoryUp`[2]	Move each category specified in `categoryIds` up one position above the next active category and return the updated list.	`apiContext`, `categoryIds`, `categoryList`	`categoryList`
`showCategory`[2]	Show the categories specified in `categoryIds` that the subscriber has selected and return the updated list.	`apiContext`, `categoryIds`, `categoryList`	`categoryList`
`updateCategories`	Update the information in the database for the categories in `categoryList`.	`apiContext`, `categoryList`	None

[1] In addition to the objects listed, all methods return a `response_code` and, if an error occurs, a `response_message`.

[2] This method changes the category list held in memory. To make the changes permanent, you must call `updateCategories`.

10.3.2.4 `ContentHandler`

`ContentHandler` is equivalent to the `IContentService` interface. This handler provides access to the content for browsing, searching, retrieving, and purchasing. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
ContentHandler.browseContent
```

The following table describes the ContentHandler methods.

TABLE 10-3 Methods for ContentHandler

Method Name	Description	Parameters	Returns[1]
addBookmark	Add an item of content to the subscriber's list of bookmarked content.	apiContext, contentId	None
browseContent	Get the number of items specified in numberToReturn in the category specified in categoryId that are of the types specified in contentTypeIdList. If contentTypeIdList is not provided, all content types are included.	apiContext, categoryId, contentTypeIdList (optional), startIndex, numberToReturn	contentList, totalSize
cancelSubscription	Cancel the subscriber's subscription to an item of content.	apiContext, contentId	None
clearBookmarks	Clear a subscriber's list of bookmarked content.	apiContext	None
deleteBookmark	Delete an item of content from the subscriber's list of bookmarked content.	apiContext, contentId	None
getAnonymousCampaignForCoupon	Get the information for a campaign that is associated with the coupon specified. Call this method if the subscriber is anonymous.	apiContext, couponCode	campaign
getBookmarks	Get the list of items the subscriber has bookmarked.	apiContext	contentList
getBundledItems	Get the items of content in a bundle.	apiContext, contentId, criteria	contentList, totalSize

TABLE 10-3 Methods for `ContentHandler` *(Continued)*

Method Name	Description	Parameters	Returns[1]
`getBundledItems`	Get the items of content in a bundle. **Note:** This method is deprecated. Use the new version.	`apiContext`, `contentId`	`contentList`, `totalSize`
`getCampaign`	Get the information for a campaign.	`apiContext`, `campaignId`	`campaign`
`getCampaignForCoupon`	Get the information for a campaign that is associated with the coupon specified. Call this method if the subscriber is known.	`apiContext`, `couponCode`	`campaign`
`getCampaignItems`	Get the list of content associated with the campaign.	`apiContext`, `campaignId`, `startIndex`, `numberToReturn`	`contentList`, `totalSize`
`getCampaigns`	Get the list of campaigns that are available to the subscriber.	`apiContext`	`campaignList`
`getContentByClassId`	Get the content ID for a specific edition of the content identified by `contentId`.	`apiContext`, `contentId`	`contentId`
`getContentByKeyword`	Get the content ID for a specific edition of the content identified by `contentKeyword`.	`apiContext`, `contentKeyword`	`contentId`
`getContentDetails`	Get the information about an item of content.	`apiContext`, `contentId`, `criteria`	`content`
`getContentDetails`	Get the information about an item of content. **Note:** This method is deprecated. Use the new version.	`apiContext`, `contentId`, `campaignId` (optional), `bundleId` (specify only if the content is part of a bundle), `isSkipTrial`, `filter`	`content`

TABLE 10-3 Methods for `ContentHandler` *(Continued)*

Method Name	Description	Parameters	Returns[1]
`getContentDetailsCriteria`	Get an empty criteria object, which can be used for methods that accept criteria as a parameter.	`apiContext`	`criteria`
`getContentDetailsList`	Get the information for each item of content specified in `contentIdList`.	`apiContext`, `contentIdList`, `criteria`	`contentList`
`getContentSummary`	Get summary information for an item of content.	`apiContext`, `contentId`	`contentSummary`
`getDeliveryType`	Get the type of delivery used to deliver this content.	`apiContext`, `contentId`	`deliveryType`
`getPurchasedBundles`	Get the list of bundles purchased by the subscriber that contain the item of content specified in `contentId`.	`apiContext`, `contentId`	`contentList`, `totalSize`
`getPurchases`	Get the list of content that the subscriber has purchased.	`apiContext`	`purchaseList`
`getSupportedModels`	Get the list of models on which the content can run.	`apiContext`, `contentId`	`modelList`
`getTicket`	Get the purchase ticket for the content identified by `contentId`.	`apiContext`, `contentId`	`codedTicket`
`hasPurchases`	Determine if the subscriber purchased any content.	`apiContext`	`hasPurchases`
`isBookmarked`	Determine if the subscriber bookmarked the content.	`apiContext`, `contentId`	`isBookmarked`
`isContentInCampaign`	Determine if the content specified in `contentId` is in the campaign specified in `campaignId`.	`apiContext`, `contentId`, `campaignId`	`isContentInCampaign`

TABLE 10-3 Methods for ContentHandler *(Continued)*

Method Name	Description	Parameters	Returns[1]
isMMSCapable	Determine if the content can be pushed to the device using MMS.	apiContext, contentId	isMMSCapable
isSMSCapable	Determine if the content can be pushed to the device using SMS.	apiContext, contentId	isSMSCapable
purchaseContent	Purchase content that is part of a campaign. Not available for anonymous subscribers.	apiContext, contentId, campaignId (optional), isSkipTrial	None
requestContent	Send content to a subscriber. Content must first be purchased.	apiContext, contentId, requestParams, maxNumberToSend	wasDelivered
searchContent	Get the list of content that matches the search criteria. The category tree is searched beginning with the category specified in categoryId. To search all content, specify the root category (see the getRootCategory method in CategoryHandler.)	apiContext, categoryId, contentTypeIdList (optional), keyword, startIndex, numberToReturn	contentList, totalSize

[1] In addition to the objects listed, all methods return a response_code and, if an error occurs, a response_message.

10.3.2.5 DownloadHandler

DownloadHandler is equivalent to the IDownloadService interface. This handler provides access to the descriptors needed to download content. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
DownloadHandler.downloadConfirm
```

The following table describes the `DownloadHandler` methods.

TABLE 10-4 Methods for `DownloadHandler`

Method Name	Description	Parameters	Returns[1]
`downloadConfirm`	Confirm if the content was downloaded to the device.	`apiContext`, `codedTicket`, `isOneStepConfirm`, `status`	None
`downloadContent`	Download the binary file for the item of content specified by `contentId`.	`apiContext`, `contentId`	`contentLength`, `contentType`, `descriptorData`
`downloadContentDescriptor`	Create a content descriptor file and return it to the caller.	`apiContext`, `codedTicket`	`contentLength`, `contentType`, `descriptorData`
`downloadDelete`	Delete content from a device.	`apiContext`, `codedTicket`, `isOneStepConfirm`, `status`	None
`downloadJAD`	Create a Java Application Descriptor (JAD) file and return it to the caller.	`apiContext`, `codedTicket`	`contentLength`, `contentType`, `descriptorData`
`downloadJAM`	Create an application descriptor file for an iAppli application and return it to the caller.	`apiContext`, `codedTicket`	`contentLength`, `contentType`, `descriptorData`
`pushMMSContent`	Push content to the device using MMS.	`apiContext`, `contentId`	None
`pushMMSContentByTicket`	Push content identified by `codedTicket` to the device using MMS.	`apiContext`, `codedTicket`	None
`pushSMSContentBinary`	Push external content to the device using SMS.	`apiContext`, `mobileId`, `contentBinary`, `mimeType`, `contentType`, `smsParams`	None

TABLE 10-4 Methods for DownloadHandler *(Continued)*

Method Name	Description	Parameters	Returns[1]
pushSMSContentByTicket	Push content identified by codedTicket to the device using SMS.	apiContext, codedTicket, smsParams	None
sendInstall	Push content installation file to a subscriber's device.	apiContext, contentId, bundleId	None

[1] In addition to the objects listed, all methods return a response_code and, if an error occurs, a response_message.

10.3.2.6 GiftingHandler

GiftingHandler is equivalent to the IGiftingService interface. This handler enables gifts or notifications about content to be sent to another subscriber. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
GiftingHandler.createGifting
```

The following table describes the GiftingHandler methods.

TABLE 10-5 Methods for GiftingHandler

Method Name	Description	Parameters	Returns[1]
cancelGifting	Cancel a gift subscription.	apiContext, giftId	None
checkAndExpireGifting	Determine if the gift is expired.	apiContext, gifting	isGiftExpired
getGiftingById	Get the information about the gift specified by giftId.	apiContext, giftId, filter, bundledContentId	gifting
getGiftingByTicket	Get the information about the gift specified by giftTicket.	apiContext, giftTicket, filter, bundledContentId	gifting
getGiftingsByGifter	Get the information for all of the gifts given by the subscriber.	apiContext, filter	giftingList
getGiftingsByRecipient	Get the information for all of the gifts received by the subscriber.	apiContext, filter	giftingList

TABLE 10-5 Methods for GiftingHandler *(Continued)*

Method Name	Description	Parameters	Returns[1]
giftContent	Purchase an item of content as a gift for another subscriber.	apiContext, contentId, campaignId (optional), recipientApiContext, message, giftedDownloads, giftedSubscriptions	giftId
isGiftingUsed	Determine if the recipient claimed all of the uses provided by the gift.	apiContext, gifting	isGiftingUsed
messageContent	Send a message about an item of content to another subscriber.	apiContext, contentId, recipientApiContext, message	giftId

[1] In addition to the objects listed, all methods return a response_code and, if an error occurs, a response_message.

10.3.2.7 MessageHandler

MessageHandler is equivalent to the IMessageService interface. This handler enables email, SMS, WAP push, and MMS messages to be sent to the current subscriber or to another subscriber. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
MessageHandler.sendMessageToSelf
```

The following table describes the MessageHandler methods.

TABLE 10-6 Methods for MessageHandler

Method Name	Description	Parameters	Returns[1]
sendMessageToMobileId	Send a message to the number specified by the mobile ID provided.	apiContext, subject, message, url, mobileId, contentId, messageCategory	None
sendMessageToSelf	Send a message to the current subscriber.	apiContext, messageType, subject, message, url, contentId, messageCategory	None
sendMessageToSubscriberId	Send a message to the subscriber identified by the subscriber ID provided.	apiContext, messageType, subject, message, url, subscriberId, contentId, messageCategory	None
sendMessageToUsername	Send a message to the subscriber identified by the user name provided.	apiContext, messageType, subject, message, url, username, contentId, messageCategory	None
sendMMSContent	Send content to the current subscriber using MMS push.	apiContext, mobileId, contentBinary, name, contentType, mimeType, subject, message, modelId, contentId, messageCategory	None

[1] In addition to the objects listed, all methods return a response_code and, if an error occurs, a response_message.

10.3.2.8 SystemHandler

SystemHandler is equivalent to the ISystemService interface. This handler provides access to system-level data such as content types, locales, and models. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
SystemHandler.getContentTypes
```

The following table describes the SystemHandler methods.

TABLE 10-7 Methods for SystemHandler

Method Name	Description	Parameters	Returns[1]
createTicket	Create a ticket for a subscriber to use to retrieve an item of content.	apiContext, contentId	ticket
getContentTypes	Get a list of all of the content types that are defined in the Content Delivery Server.	apiContext	contentTypeList
getCountries	Get a list of all of the countries that are included in the Content Delivery Server.	apiContext	countryList
getCountry	Get the information for a single country.	apiContext, countryCode	country
getDefaultLocale	Get the default locale for the system.	apiContext	locale
getDefaultModel	Get the default device model.	apiContext	modelId
getLocale	Get the information for a single locale.	apiContext, localeCode	locale
getLocales	Get a list of all of the locales included in the Content Delivery Server.	apiContext	localeList
getManufacturers	Get the names of all of the device manufacturers included in the Content Delivery Server.	apiContext	manufacturerList
getModel	Get the information for a device.	apiContext, modelId	model
getModelId	Get the internal ID for the device associated with the user agent specified in modelId.	apiContext, userAgent	modelId

TABLE 10-7 Methods for `SystemHandler` *(Continued)*

Method Name	Description	Parameters	Returns[1]
`getModels`	Get a list of all of the devices supported by the Content Delivery Server.	`apiContext`	`modelList`
`getModelsForManufacturer`	Get the models in the Content Delivery Server for a given manufacturer. Only models that are not quarantined are returned.	`apiContext`, `manufacturer`	`modelList`
`isPushEnabled`	Determine if the subscriber's device is push enabled.	`apiContext`, `modelId`	`isPushEnabled`
`isTicketValid`	Determine if the ticket for an item of content can be used by the subscriber.	`apiContext`, `ticket`, `contentId`	`isTicketValid`
`sendEvent`	Send the system event specified in `eventType`. **Note:** This method is deprecated. The functionality is no longer required.	`apiContext`, `subscriberId`, `mobileId`, `contentId` (optional), `eventType`	None
`sendEventWithParameters`	Send a system event with a set of event parameters.	`apiContext`, `requestParams`	None

[1] In addition to the objects listed, all methods return a `response_code` and, if an error occurs, a `response_message`.

10.3.2.9 `UserHandler`

`UserHandler` is equivalent to the `IUserService` interface. This handler provides access to information on subscribers and enables new subscriber accounts to be created. The guidelines for calling a method in a handler are described in Section 10.3.2.1, "Guidelines for Calls to XML-RPC Methods" on page 10-7. To call a method, concatenate the name of the handler with the name of the method, for example:

```
UserHandler.getSubscriberId
```

The following table describes the UserHandler methods.

TABLE 10-8 Methods for UserHandler

Method Name	Description	Parameters	Returns[1]
disableSubscriberBySubscriberId	Disable the account for the subscriber identified by the subscriber ID.	apiContext, subscriberId	None
disableSubscriberByUsername	Disable the account for the subscriber identified by the user name.	apiContext, username	None
getSubscriberId	Get the subscriber ID for the subscriber identified by the user name.	apiContext, username	subscriberId
getUserPreferences	Get the preferences set by the subscriber.	apiContext	preferenceList
getUserProperties	Get the information for the current subscriber.	apiContext	propertyList
getUserPropertiesBySubscriberId	Get the information for the subscriber identified by the subscriber ID.	apiContext, subscriberId	propertyList
getUserPropertiesByUsername	Get the information for the subscriber identified by the user name.	apiContext, username	propertyList

TABLE 10-8 Methods for `UserHandler` *(Continued)*

Method Name	Description	Parameters	Returns[1]
`provision`	Populate a subscriber account in the Content Delivery Server with information from an external subscriber database.	`apiContext`, `uniqueId`, `modelId`, `mobileId`, `localeCode`	`subscriberId`
`resetPasswordBySubscriberId`	Set the password for the subscriber identified by the subscriber ID to a value generated by the system.	`apiContext`, `subscriberId`, `passwordRequiresReset`	`password`
`resetPasswordByUsername`	Set the password for the subscriber identified by the user name to a value generated by the system.	`apiContext`, `username`, `passwordRequiresReset`	`password`
`setLocaleCode`	Change the subscriber's locale code.	`apiContext`, `localeCode`	None
`setModelId`	Change the model ID to match the new device that the subscriber is using.	`apiContext`, `modelId`	None
`setPassword`	Change the subscriber's password.	`apiContext`, `password`	None

TABLE 10-8 Methods for `UserHandler` *(Continued)*

Method Name	Description	Parameters	Returns[1]
`setUserPreferences`	Set the preferences selected by a subscriber. These preferences manage the information that is shown to the user.	`apiContext`, `preferenceList`	None
`setUserProperties`	Set the information for a subscriber.	`apiContext`, `propertyList`	None
`signup`	Create a subscriber account in the Content Delivery Server and in any external subscriber database.	`apiContext`, `username`, `password`, `modelId`, `mobileId`, `uniqueId`, `localeCode`	`subscriberId`
`signupWithPropertiesAndPreferences`	Create a subscriber account in the Content Delivery Server and in any external subscriber database and set the information and preferences for that subscriber.	`apiContext`, `username`, `password`, `modelId`, `mobileId`, `uniqueId`, `localeCode`, `propertyList`, `preferenceList`	`subscriberId`

[1] In addition to the objects listed, all methods return a `response_code` and, if an error occurs, a `response_message`.

10.3.2.10 Parameters for the Methods

The following table describes the parameters for the methods. Container objects such as `Hashtable` and `Vector` contain elements that are also described in the table.

TABLE 10-9 Method Parameters

Parameter	Type	Description
`addressLine1`	`String`	First line of a subscriber's address.
`addressLine2`	`String`	Second line of a subscriber's address.
`apiContext`	`Hashtable`	Container for the information about a subscriber. This container includes the following items: • `subscriberId` • `username` • `localeCode` • `mobileId` • `uniqueId` • `modelId` • `isAnonymous` • `isProvisioned` • `isRegistered` • `passwordRequiresReset` • `isCategoryCustomized` • `roleId` • `propertyMap` **Note**: For an anonymous subscriber, only `subscriberId`, `localeCode`, and `modelId` are included.
`bundledContentId`	`Integer`	Internal ID that was assigned by the Content Delivery Server to an individual item in a bundle.
`bundleId`	`Integer`	Internal ID that was assigned by the Content Delivery Server to the bundle with which you are working.
`campaign`	`Hashtable`	Container for the information about a campaign. This container includes the following items: • `campaignId` • `name` • `description` • `campaignSubject` • `campaignMessage` • `campaignDiscount` • `campaignExpiration`

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
campaignDiscount	Double	Percentage by which the items in the campaign are discounted.
campaignExpiration	Date	Date that the campaign expires. If the campaign is expired, null is returned.
campaignId	Integer	Internal ID that was assigned by the Content Delivery Server to the campaign with which you are working.
campaignList	Vector, elements of type campaign	List of campaigns available to the subscriber.
campaignMessage	String	Promotional message included in the notifications sent to subscribers to announce the campaign.
campaignSubject	String	Subject added to email notifications sent to subscribers to announce the campaign.
category	Hashtable	Container for information about a category. This container includes the following items: • categoryId • name • description • parentCategoryId • isLeafNode • isActive • displayOrder • contentCount • subCategoryList (only included if returned by the getCategoryBranch method of CategoryHandler)
categoryId	Integer	Internal ID that was assigned by the Content Delivery Server to the category with which you are working.
categoryIds	Vector, elements of type Integer	List of category IDs. Categories are identified by the internal ID that was assigned by the Content Delivery Server.
categoryList	Vector, elements of type category	List of categories.
city	String	City for the subscriber's address.
codedTicket	String	String that identifies the purchase ticket for the download request.
contactPhone	String	Subscriber's phone number.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`content`	`Hashtable`	Container for the information about an item of content. This container includes the following items: • `contentId` • `name` • `shortDescription` • `longDescription` • `submitDate` • `sizeInKB` • `version` • `contentType` • `mimeType` • `numberOfDownloads` • `smallIconUrl` • `largeIconUrl` • `userGuideUrl` • `previewUrl` • `screenShot1Url` • `screenShot2Url` • `developerName` • `developerUrl` • `downloadTimes` • `networks` • `downloadUrl` • `isBookmarked` • `isPurchaseRequiredBeforeDownload` • `isUnsubscribeAvailable` • `isValidOnCurrentModel` • `pricingDetails` • `isActive`
`contentBinary`	String	Binary format of content.
`contentCount`	`Integer`	Number of items of content in the category.
`contentId`	`Integer`	Internal ID that was assigned by the Content Delivery Server to the item of content with which you are working.
`contentIdList`	`Vector`, elements of type `contentId`	List of content IDs.
`contentKeyword`	`String`	Keyword used to identify content, typically used for content delivered directly to a device using SMS.
`contentLength`	`Integer`	Size of the content.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
contentList	Vector, elements of type contentSummary	List of items of content.
contentSummary	Hashtable	Container for summary information about an item of content. This container includes the following items: • contentId • name • shortDescription • submitDate
contentType	String	Type of content with which you are working. This type must be one of the content types defined in the contentTypeList.
contentTypeIdList	Vector, elements of type Integer	List of content type IDs. Content types are identified by the internal ID that was assigned by the Content Delivery Server.
contentTypeList	Vector, elements of type Hashtable	Information about each content type defined in the Content Delivery Server. Each element contains the following items: • id • name
country	Hashtable	Container for information about a country. This container includes the following items: • countryCode • name
countryCode	String	Two-character ISO code that represents the subscriber's country, for example, US.
countryList	Vector, elements of type country	List of countries.
couponCode	String	String that identifies the coupon that the subscriber is using to purchase content.
criteria	Hashtable	Container for information about an item of content. This container includes the following items: • filter • campaignId (optional) • bundleId (only if the content is part of a bundle) • isSkipTrial • licenseType

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`deliveryType`	`String`	Type of delivery used to deliver content. The following values are valid: • `ems` - Enhanced Messaging System • `nsm` - Nokia Smart Messaging • `one_step_wap` - One-step WAP • `two_step_wap` - Two-step WAP
`description`	`String`	Description of the object. Depending on the method called, this is the description of the category, the campaign, the device, or the locale.
`descriptorData`	`String`	Content descriptor binary code. When returned by the `downloadContent` method of `DownloadHandler`, this is the binary preview file.
`developerName`	`String`	Name of the developer who submitted content.
`developerUrl`	`String`	URL for the developer who submitted the content.
`devicePhone`	`String`	Phone number of subscriber's device.
`displayOrder`	`Integer`	Position of the category in the list of categories. 1 indicates the top of the list.
`downloadCount`	`Integer`	Number of downloads allowed per purchase.
`downloadPeriod`	`Integer`	Number of days of use allowed per purchase.
`downloadPrice`	`Double`	Price charged to download the content.
`downloadTimes`	`Vector`, elements of type `String`	Estimated time that it takes to download the content. Each element corresponds to an element in the `networks` object to indicate the estimated time that the download takes over the corresponding type of network.
`downloadUrl`	`String`	URL from which the content is downloaded.
`emailAddress`	`String`	Subscriber's email address.
`eventType`	`String`	Type of event with which you are working. Use `sms_request_for_content` to indicate that the event is a request for information about an item of content.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
filter	Hashtable	Container for Boolean flags that indicate the type of information to return. Information of the type associated with each flag is returned only if the flag is included in the hash table and is set to `true`. The following flags are valid when working with `ContentHandler`: • `filterDetailsDownload` - Download information • `filterDetailsDownloadCount` - Download count • `filterDetailsIsBookmarked` - Information on bookmarks • `filterDetailsResourceURLs` - Resource URLs • `filterDetailsPricingAndPurchase` - Pricing and purchase information • `filterDetailsPricingAndGifting` - Pricing information for gifts • `filterDetailsIsValidOnCurrentModel` - Information on whether or not the content executes on the device The following flags are valid when working with `GiftHandler`: • `filterGiftsContent` - Information about the content • `filterGiftsDownload` - Download information
firstName	String	Subscriber's first name.
gender	String	Subscriber's gender. Valid values and what each value indicates are described in the following list: • `M` - Subscriber is male. • `F` - Subscriber is female.
giftCost	Double	Price of the gift.
giftedDownloads	Integer	Number of downloads that are paid for by the gift.
giftedSubscriptions	Integer	Number of subscriptions that are paid for by the gift.
gifterId	Integer	Subscriber ID for the subscriber who purchased the gift.
gifterMobileId	String	Mobile ID for the subscriber who purchased the gift.
giftExpirationDate	Date	Date that the gift expires.
giftId	Integer	Internal ID that is assigned by the Content Delivery Server to the gift with which you are working.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`gifting`	`Hashtable`	Container for information about a gift. This container contains the following items: • `giftId` • `giftStatus` • `giftIsOnDevice` • `contentId` • `giftTicket` • `gifterId` • `gifterMobileId` • `giftRecipientId` • `giftRecipientMobileId` • `giftRecipientModelId` • `message` • `giftedDownloads` • `giftedSubscriptions` • `giftCost` • `giftPurchaseDate` • `giftExpirationDate` • `pricingDetails`
`giftIsOnDevice`	`Boolean`	Flag that indicates if the content provided by the gift exists on the target device. `True` indicates that the content is on the device. `False` indicates that the content is not on the device.
`giftingList`	`Vector`, elements of type `gifting`	List of gifts.
`giftPurchaseDate`	`Date`	Date that the gift was purchased.
`giftRecipientId`	`Integer`	Subscriber ID for the subscriber who is the recipient of a gift.
`giftRecipientMobileId`	`String`	Mobile ID for the subscriber who is the recipient of a gift.
`giftRecipientModelId`	`Integer`	Model ID of the device for the subscriber who is the recipient of a gift

TABLE 10-9 Method Parameters *(Continued)*

Parameter	**Type**	**Description**
`giftStatus`	Integer	Status of a gift. Valid values and what each value indicates are described in the following list: • `8` - Gift is purchased • `9` - Recipient has started downloading the gift • `10` - Gift was successfully downloaded • `11` - Gift is expired • `12` - Gift is cancelled • `13` - Gift is refunded
`giftTicket`	`String`	Internal object used to validate that the subscriber can access the gift.
`hasPurchases`	`Boolean`	Flag that indicates if a subscriber has purchased content. `True` indicates that the subscriber has purchased content. `False` indicates that the subscriber has not purchased content.
`id`	`Integer`	Internal ID that was assigned by the Content Delivery Server to the locale, or the content type, that you are working with.
`includeContentCount`	Boolean	Flag that indicates if the number of items in a category is calculated. `True` indicates that the number is calculated. `False` indicates that the number is be calculated.
`isActive`	`Boolean`	When included in the `category` object, this flag indicates if the category is shown to the subscriber. `True` indicates that the category is shown. `False` indicates that the category is not shown. When included in the `content` object, this flag indicates if the content is active. `True` indicates that the content is active. `False` indicates that the content is inactive.
`isAnonymous`	`Boolean`	Flag that indicates if the subscriber is anonymous. `True` indicates that the subscriber is anonymous. `False` indicates that the subscriber is known.
`isBookmarked`	`Boolean`	Flag that indicates if the subscriber has bookmarked the content. `True` indicates that content is bookmarked. `False` indicates that the content is not bookmarked.
`isCategoryCustomized`	Boolean	Flag that indicates if the subscriber has customized the categories that are shown. `True` indicates that the categories are customized. `False` indicates that the categories are not customized.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
isContentInCampaign	Boolean	Flag that indicates if the item of content is included in a campaign. True indicates that the item is in the campaign. False indicates that the item is not in the campaign.
isDefault	Boolean	Flag that indicates if the device is the default device. True indicates that the device is the default device. False indicates that the device is not the default device.
isDownloaded	Boolean	Flag that indicates if content has been downloaded. True indicates that an item of content has been downloaded. False indicates that an item of content has not been downloaded.
isDownloadRecurring	Boolean	Flag that indicates if the subscriber is automatically charged for additional downloads after the number of purchased downloads is exceeded. True indicates that renewal is automatic. False indicates that the subscriber must manually purchase additional downloads.
isFree	Boolean	Flag that indicates if the content is free. True indicates that the content is free. False indicates that the content must be purchased.
isGiftExpired	Boolean	Flag that indicates if the gift is expired. True indicates that the gift is expired and can no longer be claimed. False indicates that the gift is not expired.
isGiftingUsed	Boolean	Flag that indicates if all of the uses purchased for the gift were used by the recipient. True indicates that all of the uses that were purchased are used. False indicates that not all of the uses that were purchased are used.
isLeafNode	Boolean	Flag that indicates if the category has any subcategories. True indicates that the category does not have subcategories. False indicates that the category has subcategories.
isMMSCapable	Boolean	Flag that indicates if the content can be sent to the device using MMS. True indicates that MMS can be used. False indicates that MMS cannot be used.
isOnDevice	Boolean	Flag that indicates if an item of content is on a subscriber's device. True indicates that the item is on the device. False indicates that the item is not on the device.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`isOneStepConfirm`	`Boolean`	Flag that indicates if one-step or two-step download is used. `True` indicates that it is a one-step download. `False` indicates that it is a two-step download.
`isProvisioned`	`Boolean`	Flag that indicates if an entry for the subscriber exists in the Content Delivery Server database. `True` indicates that an entry does exist. `False` indicates that the subscriber is anonymous.
`isPurchaseRequiredBeforeDownload`	`Boolean`	Flag that indicates if the content must be purchased before being downloaded. `True` indicates that the content must be purchased first. `False` indicates that content can be downloaded.
`isPushEnabled`	`Boolean`	Flag that indicates if the device is push enabled. `True` indicates that the device is push enabled. `False` indicates that the device is not push enabled.
`isRegistered`	`Boolean`	Flag that indicates if the subscriber is registered in an external subscriber database. `True` indicates that the subscriber is registered. `False` indicates that the subscriber is not registered.
`isSkipTrial`	`Boolean`	Flag that indicates if the subscriber chooses to skip the trial usage. `True` indicates that the subscriber chooses to skip the trial usage so the content can be purchased immediately. `False` indicates that the subscriber chooses not to skip the trial usage.
`isSMSCapable`	`Boolean`	Flag that indicates if the content can be sent to the device using SMS. `True` indicates that SMS can be used. `False` indicates that SMS cannot be used.
`isSubscriptionExpired`	`Boolean`	Flag that indicates if the subscription has expired. `True` indicates that the subscription is expired. `False` indicates that the subscription is not expired.
`isSubscriptionRecurring`	`Boolean`	Flag that indicates if the subscription is automatically renewed. `True` indicates that renewal is automatic. `False` indicates that the subscriber must manually renew the subscription.
`isTicketValid`	`Boolean`	Flag that indicates if the subscriber can use the ticket to get an item of content. `True` indicates that the subscriber can use the ticket. `False` indicates that the subscriber cannot use the ticket.
`isTrialAvailable`	`Boolean`	Flag that indicates if the content can be used on a trial basis. `True` indicates that a trial is available. `False` indicates that no trial is available.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`isUpdateAvailable`	`Boolean`	Flag that indicates an update is available for the content. `True` indicates that an update is available. `False` indicates that an update is not available.
`isUnsubscribeAvailable`	`Boolean`	Flag that indicates if a subscription for the content can be canceled. `True` indicates that the subscription can be canceled. `False` indicates that there is no subscription or that the subscription cannot be canceled.
`isUsageConsumed`	`Boolean`	Flag that indicates if a subscriber has used the content for the number of uses purchased. `True` indicates that all purchased uses are used. `False` indicates that not all purchased uses are used.
`isValidOnCurrentModel`	`Boolean`	Flag that indicates if the content can be run on the model that the subscriber is using. `True` indicates that the content does run on the model. `False` indicates that the content does not run on the model.
`keyword`	`String`	String to match when searching for content.
`languageCode`	`String`	Two-character ISO code that represents the subscriber's language, for example, `en`.
`largeIconUrl`	`String`	URL that points to the large icon for the content.
`lastName`	`String`	Subscriber's last name.
`licenseType`	`Integer`	Type of license associated with the content. Valid values and what each value indicates are described in the following list: • `0` - Content was purchased. • `1` - Content was sent as a gift. • `2` - Content was received as a gift.
`listEnd`	`String`	Together with `listStart`, specifies the range of items to return. The alphabetized list ends with items that match the string specified. The string is case sensitive. Specify null to end at the end of the complete list.
`listStart`	`String`	Together with `listEnd`, specifies the range of items to return. The alphabetized list begins with items that match the string specified. The string is case sensitive. Specify null to start at the beginning of the complete list.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`locale`	`Hashtable`	Container for locale information. This container contains the following items: • `id` • `countryCode` • `languageCode` • `localeCode` • `description`
`localeCode`	`String`	String that represents the subscriber's locale, for example, `en_US`.
`localeList`	`Vector`, elements of type `locale`	List of locales.
`longDescription`	`String`	Long description from the information about the content
`manufacturer`	`String`	Name of the manufacturer of a device.
`manufacturerList`	`Vector`, elements of type `manufacturer`	List of manufacturers, sorted alphabetically.
`maxNumberToSend`	`Integer`	Maximum number of items to deliver in a call. If more items exist than the number specified, no content is sent. To deliver all items, use `-1`.
`message`	`String`	Message to be sent to a subscriber.
`messageCategory`	`Integer`	Category of message to be sent. Categories one through seven are sent to the subscriber, category 9 is sent to the Catalog Manager administrator. Valid values and what each value indicates are described in the following list: • `1` - Message contains a URL that points to the details about an item of content. • `2` - Message is a mobile originated message that contains a URL that points to the details about an item of content. • `3` - Message contains the content binary. • `4` - Message contains a gift and includes a URL that points to the details about an item of content. • `5` - Message contains a notification and includes a URL that points to the details about an item of content. • `6` - Message contains a password reminder. • `7` - Message contains information about a campaign. • `9` - New device was added to the Content Delivery Server.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
messageType	Integer	Type of message to send. Valid values and what each value indicates are described in the following list: • 1 - Message is sent to the subscriber's device. • 2 - Message is sent to the subscriber's email.
middleName	String	Subscriber's middle name.
mimeType	String	MIME type of the content.
mobileId	String	Phone number for the subscriber.
model	Hashtable	Container for device information. This container includes the following items: • modelId • name • description • modelNumber • manufacturer • userAgentPattern • isDefault
modelId	Integer	Internal ID that was assigned by the Content Delivery Server to the device that you are working with.
modelList	Vector, elements of type model	List of devices.
modelNumber	String	Model number associated with a device.
name	String	Name of the object. Depending on the method, this is the name of the category, the country, the campaign, the device, or the content.
networks	Vector, elements of type String	List of network types that are known by the Content Delivery Server. Each element corresponds to an element in the downloadTimes Vector to indicate the estimated time that the download takes for the corresponding network.
notifyPromotions	Boolean	Indicates if the subscriber wants to be notified about promotions. True indicates that the subscriber wants to be notified. False indicates that the subscriber does not want to be notified.
numberOfDownloads	Integer	Total number of times that the content was downloaded by all subscribers.
numberToReturn	Integer	Number of items to return in a list. To return all items, specify -1.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
parentCategoryId	Integer	Internal ID that was assigned by the Content Delivery Server to the category that is the parent of the category that you are working with.
password	String	Unencrypted password for the subscriber. Encryption is performed by the Content Delivery Server.
passwordRequiresReset	Boolean	Flag that indicates if the subscriber's password must be reset when the subscriber logs in. `True` indicates that the password must be reset. `False` indicates that the password does not need to be reset.
postalCode	String	Postal code or ZIP code for a subscriber's address.
preferenceList	Hashtable	Container for a subscriber's preferences. This container includes items of type `notifyPromotions`. To delete a preference, set the value to the empty string ('').
previewUrl	String	URL that points to the preview file for the content.
pricingDetails	Hashtable	Container for pricing information for an item of content. This container includes the following items: • `campaignDiscount` • `downloadPrice` • `downloadCount` • `downloadPeriod` • `isDownloadRecurring` • `subscriptionPrice` • `subscriptionFrequency` • `isSubscriptionRecurring` • `trialCount` • `usagePrice` • `usageCount` • `isFree` • `isTrialAvailable`

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
propertyList	Vector, elements are of type Hashtable	Information about each subscriber. Each element contains the following items: • salutation • firstName • middleName • lastName • gender • emailAddress • addressLine1 • addressLine2 • city • stateProvince • postalCode • countryCode • contactPhone • devicePhone
propertyMap	Hashtable	Set of name-value pairs used to configure system behavior. These values are internal values used by the Content Delivery Server.
purchaseDate	Date	Date the subscriber purchased the item.
purchaseList	Vector, elements are of type Hashtable	Information about each item purchased by the subscriber. Each element contains the following items: • contentId • name • purchaseDate • subscriptionExpiration • pricingDetails • isValidOnCurrentModel • isSubscriptionExpired • isUsageConsumed • isUpdateAvailable • isDownloaded • isOnDevice • codedTicket

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`recipientApiContext`	`Hashtable`	Container for the information about the recipient of a gift. This container includes the following items: • `subscriberId` • `username` • `localeCode` • `mobileId` • `uniqueId` • `modelId`
`requestHeaders`	`Hashtable`	Container for the HTTP headers associated with an HTTP request.
`requestParams`	`Hashtable`	Container for key-value pairs that provide information about an event. The following keys must be included: • `request_data` - Data included in the request, such as the unparsed SMS request for content or for a campaign • `request_source` - Source of the request, such as the short code to which it is sent • `request_type` - String that identifies the type of request, such as `portal` or `mo_push` or other value recognized by your system Additional keys that your system recognizes can also be included.
`response_code`	`String`	Code that indicates if the method executed successfully. `1` indicates successful completion. `-1` indicates an error occurred.
`response_message`	`String`	Message returned by the method.
`roleId`	`Integer`	Role assigned to the subscriber. Valid values and what each value indicates are described in the following list: • `0` - Subscriber has access only to content with a status of `Testing`. • `1` - Subscriber has standard privileges.
`salutation`	`String`	Courtesy title, such as Mr.
`screenShot1Url`	`String`	URL that points to the first screen shot for the content.
`screenShot2Url`	`String`	URL that points to the second screen shot for the content.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`searchFilter`	`Hashtable`	Container for the criteria used to filter the results of a content search. This container includes the following items: • `categoryId` • `developerName` • `keyword` The container must include at least one type of criterion.
`shortDescription`	`String`	Short description from the information about the content.
`sizeInKB`	`String`	Size of content.
`smallIconUrl`	`String`	URL that points to the small icon for the content.
`smsParams`	`Hashtable`	Container for parameters required to push binary content to a device. The entries in the hash table are pairs of strings that identify the name and value of each parameter needed.
`startIndex`	`Integer`	The position in a list of items at which to begin processing.
`stateProvince`	`String`	State or province for the subscriber's address.
`status`	`String`	Confirm status string returned by the device. For possible values returned by MIDP applications, see `http://www.jcp.org/jsr/detail/118.jsp`.
`statusList`	`Vector`, elements of type `Integer`	List of content statuses. Valid values and what each value indicates are described in the following list: • `1` - Content is active. Content is stocked and available to the subscriber. • `2` - Content is inactive. Content is stocked, but not available to the subscriber. • `3` - Content is unavailable. Content is no longer available from the Catalog Manager. • `4` - Content is being tested and is available only to subscribers who are assigned the role of tester.
`subCategoryList`	`Vector`, elements of type `category`	List of categories under a specified node.
`subject`	`String`	Subject of the message to be sent to a subscriber.
`submitDate`	`Date`	Date the content was submitted.
`subscriberId`	`Integer`	Internal ID that was assigned by the Content Delivery Server to the subscriber account.

TABLE 10-9 Method Parameters *(Continued)*

Parameter	Type	Description
`subscriptionExpiration`	`Date`	Date the subscriber's subscription expires.
`subscriptionFrequency`	`String`	Period of time covered by a subscription. Valid values are `daily`, `weekly`, `monthly`, and `yearly`.
`subscriptionPrice`	`Double`	Price charged for a subscription for the content.
`ticket`	`String`	Internal object used to validate that the subscriber can access the content requested.
`totalSize`	`Integer`	Number of items found.
`trialCount`	`Integer`	Number of times the content can be used for free before the subscriber is prompted to purchase.
`uniqueId`	`String`	Unique ID for the subscriber.
`url`	`String`	URL to include in a message sent to a subscriber.
`usageCount`	`Integer`	Number of uses allowed per purchase.
`usagePrice`	`Double`	Price charged per use or number of uses.
`userAgent`	`String`	User agent for a device. This string is the exact string that was returned in the HTTP header.
`userAgentPattern`	`String`	User agent for a device. This string is a regular expression, which is a pattern that can match various text strings.
`userGuideUrl`	`String`	URL that points to the User Guide for the content.
`username`	`String`	User name for the subscriber.
`version`	`String`	The version of the content.
`wasDelivered`	`Boolean`	Flag that indicates that the content was sent to the device using SMS. `True` indicates that SMS was used. `False` indicates that SMS was not used.

10.3.3 Examples of Using Handlers

This section presents two examples of using the XML-RPC implementation of the Subscriber API.

10.3.3.1 Example of Creating an `ApiContext` Object

The following code excerpt shows how to create an `ApiContext Object`. This sample uses the bindings for the Java programming language.

CODE EXAMPLE 10-3 Create an `APIContext` Object Using XML-RPC

```
...
// Get a reference to the XmlRpcClient
String url = "http://host1:8080/subscriber/xml_rpc.do";
XmlRpcClientLite client = new XmlRpcClientLite(url);

// Set up the input parameters
Vector parameters = new Vector();
Hashtable ht = new Hashtable();
ht.put("username", "user1");
ht.put("password", "cryptic1");
parameters.addElement(ht);

// Send the request to Content Delivery Server
Hashtable response =
(Hashtable) client.execute("AuthenticationHandler.getApiContext", parameters);

// Evaluate the response
String errorCode = (String)response.get("response_code");
if (!errorCode.equals("1"))
{
  // Handle Error
  System.out.println((String)response.get("response_message"));
  ...
}
else
{
  // Authentication successful
  Hashtable apiContext = (Hashtable)response.get("apiContext");
  Integer subscriberId = (Integer)apiContext.get("subscriberId");
  String username = (String)apiContext.get("username");
  String localeCode = (String)apiContext.get("localeCode");
  String mobileId = (String)apiContext.get("mobileId");
  Integer modelId = (Integer)apiContext.get("modelId");
  ...

  // Save the ApiContext Hashtable in the Session
  session.setAttribute("API_CONTEXT", apiContext);
  ...
}

    ...
```

10.3.3.2 Example of Creating a Handler and Purchasing Content

The following code excerpt shows how to create a handler and use that handler to purchase content. This sample uses the bindings for the Java programming language.

CODE EXAMPLE 10-4 Create a Handler

```
...
// Get a reference to the XmlRpcClient
String url = "http://[[cds server host]]:[[cds server
port]]/subscriber/xml_rpc.do";
XmlRpcClientLite client = new XmlRpcClientLite(url);

// Retrieve the ApiContext Hashtable from the HttpSession
Hashtable apiContext = (Hashtable) session.getAttribute("API_CONTEXT");

// Set up the input parameters
Vector parameters = new Vector();
Hashtable ht = new Hashtable();
ht.put("apiContext", apiContext);
ht.put("contentId", new Integer(1001));
ht.put("campaignId", new Integer(1000));
ht.put("isSkipTrial", Boolean.TRUE);
parameters.addElement(ht);

// Send the request to Content Delivery Server
Hashtable response =
(Hashtable) client.execute("ContentHandler.purchaseContent", parameters);

// Evaluate the response
String errorCode = (String)response.get("response_code");
if (!errorCode.equals("1"))
{
  // Handle Error
  System.out.println((String)response.get("response_message"));
  ...
}
else
{
  // Purchase successful
  ...
}
...
```

Index

A

addBookmark, 10-12
addMMSSlide, 8-13
addRecipient, 8-13
addUserAgent, 8-3
ApiContextFactory class, 10-1
APIs
  Billing API, 1-2
  Confirm Service, 1-4
  Content Management, 1-2
  Content Validation, 1-3
  Event Service, 1-1, 2-1
  Messaging, 1-4
  Subscriber, 1-4
  User Profile, 1-3
  WAP Gateway, 1-3
ApiServiceFactory class, 10-1
ApiUtil class, 10-1
AuthenticationHandler, 10-9
authorize, 3-8

B

billing adapter, 3-1
Billing API, 1-2, 3-1
  content listing process flow, 3-2
  download confirmation process flow, 3-5
  error handling, 3-7
  general process flow, 3-2
  subscriber purchase process flow, 3-4
  subscription verification process flow, 3-6
  transaction initiation process flow, 3-3
  using, 3-12
BillingConstants, 3-1
BillingException, 3-1
BillingInfo, 3-1
BillingManager, 3-1, 3-7
browseContent, 10-12

C

cancelGifting, 10-17
cancelSubscription, 10-12
CategoryHandler, 10-10
CDS_EVENT table, 2-4
CDS_EVENT_GROUP table, 2-5
CDS_EVENT_TYPE table, 2-5
CDSExceptionCDSException class, 10-2
checkAndExpireGifting, 10-17
checkSubscription, 3-9
classes
  ApiContextFactory, 10-1
  ApiServiceFactory, 10-1
  ApiUtil, 10-1
  BillingConstants, 3-1
  BillingException, 3-1
  BillingInfo, 3-1
  BillingManager, 3-1
  CDSException, 10-2
  ConfirmServiceAdapter, 9-2
  ContentSlide, 8-10
  MMSPushMessage, 8-13
  MMSSlide, 8-11
  PushConstants, 8-18
  PushMessage, 8-3
  PushResponse, 8-17

  SMSMessage, 8-9
  SMTPMessage, 8-10
  UserManager class, 6-1
  WAPGatewayAdapter, 7-2
  WapPushMessage, 8-10
clearBookmarks, 10-12
client application
  JMS, 2-7
  SQL*Net, 2-3
confirm, 3-9
Confirm Service API, 1-4, 9-1
  general process flow, 9-1
  using, 9-3
ConfirmServiceAdapter class, 9-2
connect, 8-2, 9-2
Content Management API, 1-2
  downloading content, 4-3
  general process flow, 4-2
  getting a list of content, 4-2
  getting content details, 4-3
  using, 4-6
Content Validation API, 1-3, 5-1
ContentHandler, 10-11
ContentManager, 4-4
ContentSlide class, 8-10
conventions, typographical, xxii
createTicket, 10-20

D

deleteBookmark, 10-12
disableSubscriberBySubscriberId, 10-22
disableSubscriberByUsername, 10-22
doAccountExists, 6-2
doAddUser, 6-2
documentation, related, xxiii
doDisableUser, 6-2
doEnableUser, 6-2
doFormatLogind, 6-2
doFormatMobileId, 6-2
doGetAllLikeInOrder, 6-3
doGetAllUsers, 6-3
doGetAllUsersContainingFirstName, 6-3
doGetAllUsersContainingId, 6-3
doGetAllUsersContainingLastName, 6-3
doGetAllUsersContainingName, 6-4
doGetAnonymousUser, 6-4
doGetFieldName, 6-4
doGetUser, 6-4
doGetUserByMobileId, 6-4
doGetUserByUniqueDeviceId, 6-4
doGetUserInstance, 6-5
doHandle, 7-2
doIsActive, 6-5
doIsAuthenticated, 6-5
doRemoveUser, 6-5
doUpdateUser, 6-5
downloadConfirm, 10-16
downloadContent, 10-16
downloadContentDescriptor, 10-16
downloadDelete, 10-16
DownloadHandler, 10-15
downloadJAD, 10-16
downloadJAM, 10-16

E

event data, 2-7
Event Service API
  example, 2-13
  introduction, 1-1
  overview, 2-1
  overview diagram, 2-2
  using, 2-12
event tables, 2-3
EVENT_SOURCE_TYPE_ID table, 2-6
events, 2-7
execute, 5-2

G

getActivateDate, 6-6
getAllMMSSlides, 8-13
getAllRecipients, 8-13
getAllUserAgents, 8-4
getAnonymousApiContext, 10-9
getAnonymousCampaignForCoupon, 10-12
getApiContext, 10-9
getAttribute, 6-6, 8-4
getAttributes, 6-6
getAudioContent, 8-11
getBillingInfo, 3-9

getBillingInfos, 3-10
getBookmarks, 10-12
getBundledItems, 10-12
getCampaign, 10-13
getCampaignForCoupon, 10-13
getCampaignItems, 10-13
getCampaigns, 10-13
getCategory, 10-10
getCategoryBranch, 10-10
getCity, 6-6
getContentBinary, 4-6, 8-4
getContentByClassId, 10-13
getContentByKeyword, 10-13
getContentData, 8-10
getContentDescriptor, 4-5
getContentDetails, 10-13
getContentDetailsCriteria, 10-14
getContentDetailsList, 10-14
getContentId, 8-10
getContentInfo, 4-5
getContentInfos, 4-5
getContentMimeType, 8-11
getContentSummary, 10-14
getContentType, 8-4
getContentTypes, 10-20
getCountries, 10-20
getCountry, 6-7, 10-20
getCreateDate, 6-7
getDeActivateDate, 6-7
getDefaultLocale, 10-20
getDefaultModel, 10-20
getDeliveryReportRequired, 8-14
getDeliveryType, 10-14
getDestinationAddress, 8-4
getEmail, 6-7
getFirstName, 6-7
getFromAddress, 8-14
getGender, 6-7
getGiftingById, 10-17
getGiftingByTicket, 10-17
getGiftingsByGifter, 10-17
getGiftingsByRecipient, 10-17
getImageContent, 8-12
getJMSMessageId, 8-4
getKeyword, 8-5, 8-7
getLastLogin, 6-8
getLastName, 6-8
getLocale, 10-20
getLocales, 10-20
getLog, 3-11
getLoginId, 6-8
getManufacturers, 10-20
getMessageClass, 8-14
getMessageId, 8-5, 8-18
getMessagePriority, 8-14
getMessageText, 8-5
getMiddleName, 6-8
getMimeType, 8-5
getMobileId, 6-8
getModel, 10-20
getModelId, 10-20
getModels, 10-21
getModelsForManufacturer, 10-21
getMSISDN, 7-2
getNotEmtpySubCategories, 10-10
getPassword, 6-8
getPhone, 6-9
getPostalCode, 6-9
getPurchasedBundles, 10-14
getPurchases, 10-14
getPushCategory, 8-5
getPushDomain, 8-5
getPushType, 8-6
getReadReportRequired, 8-14
getResponseDescription, 8-18
getResponseStatus, 8-18
getRootCategory, 10-10
getSalutation, 6-9
getSenderVisibility, 8-14
getShortCode, 8-6
getSMILPresentation, 8-15
getState, 6-9
getStreet1, 6-9
getStreet2, 6-9
getSubCategories, 10-10
getSubscriberId, 8-6, 10-22

getSupportedModels, 10-14
getTextContent, 8-12
getTicket, 10-14
getUniqueDeviceID, 6-15
getUniqueDeviceId, 6-10, 8-6
getUniqueId, 7-2
getUserPreferences, 10-22
getUserProperties, 10-22
getUserPropertiesBySubscriberId, 10-22
getUserPropertiesByUsername, 10-22
getVendingContentId, 8-6
getVideoContent, 8-12
giftContent, 10-18
GiftingHandler, 10-17
guidelines, XML-RPC method calls, 10-7

H
hasLoggedIn, 6-10
hasPurchases, 10-14
hideCategory, 10-10
HTTPSMSPushMsgSender class, 8-19

I
IApiContext interface, 10-1
ICategoryService interface, 10-1
IContentService interface, 10-1
IDownloadService interface, 10-1
IGiftingService interface, 10-1
IMessageService interface, 10-2
initialize, 8-3
interfaces
  IApiContext, 10-1
  ICategoryService, 10-1
  IContentService, 10-1
  IDownloadService, 10-1
  IGiftingService, 10-1
  IMessageService, 10-2
  ISystemService, 10-2
  IUserService, 10-2
  MMSSender, 8-16
  PushMsgListener, 8-2
  PushMsgSender, 8-2
  User, 6-6
  UserDeviceManager, 6-15
introduction, 1-1

isBookmarked, 10-14
isConfirmed, 6-10
isContentInCampaign, 10-14
isEnabled, 6-10
isGiftingUsed, 10-18
isMMSCapable, 10-15
isPrepay, 6-10
isPushEnabled, 10-21
isSMSCapable, 10-15
isTicketValid, 10-21
ISystemService interface, 10-2
IUserService interface, 10-2

J
JMS client application, 2-7

L
listen, 8-3, 9-2

M
messageContent, 10-18
MessageHandler, 10-18
messageReceived, 9-3
Messaging API, 1-4
Messaging API, using, 8-18
methods
  authorize, 3-8
  checkSubscription, 3-9
  confirm, 3-9
  doAccountExists, 6-2
  doAddUser, 6-2
  doDisableUser, 6-2
  doEnableUser, 6-2
  doFormatLogind, 6-2
  doFormatMobileId, 6-2
  doGetAllLikeInOrder, 6-3
  doGetAllUsers, 6-3
  doGetAllUsersContainingFirstName, 6-3
  doGetAllUsersContainingId, 6-3
  doGetAllUsersContainingLastName, 6-3
  doGetAllUsersContainingName, 6-4
  doGetAnonymousUser, 6-4
  doGetFieldName, 6-4
  doGetUser, 6-4
  doGetUserByMobileId, 6-4
  doGetUserByUniqueDeviceId, 6-4

doGetUserInstance, 6-5
doIsActive, 6-5
doIsAuthenticated, 6-5
doRemoveUser, 6-5
doUpdateUser, 6-5
execute, 5-2
getActivateDate, 6-6
getAttribute, 6-6
getAttributes, 6-6
getBillingInfo, 3-9
getBillingInfos, 3-10
getCity, 6-6
getCountry, 6-7
getCreateDate, 6-7
getDeActivateDate, 6-7
getEmail, 6-7
getFirstName, 6-7
getGender, 6-7
getLastLogin, 6-8
getLastName, 6-8
getLog, 3-11
getLoginId, 6-8
getMiddleName, 6-8
getMobileId, 6-8
getPassword, 6-8
getPhone, 6-9
getPostalCode, 6-9
getSalutation, 6-9
getState, 6-9
getStreet1, 6-9
getStreet2, 6-9
getUniqueDeviceID, 6-15
getUniqueDeviceId, 6-10
hasLoggedIn, 6-10
isConfirmed, 6-10
isEnabled, 6-10
isPrepay, 6-10
refund, 3-11
returns, 5-3
reverse, 3-11
setActiveDate, 6-10
setAttribute, 6-11
setAttributes, 6-11
setCity, 6-11
setCountry, 6-11
setCreateDate, 6-11
setDeActivateDate, 6-11
setEmail, 6-12
setFirstName, 6-12
setGender, 6-12
setHasLoggedIn, 6-12
setIsEnabled, 6-12
setIsPrepay, 6-12
setLastName, 6-13
setLoginId, 6-13
setMiddleName, 6-13
setMobileId, 6-13
setPassword, 6-13
setPhone, 6-13
setPostalCode, 6-14
setSalutation, 6-14
setState, 6-14
setStreet1, 6-14
setStreet2, 6-14
setUniqueDeviceId, 6-14
subscribe, 3-11
unsubscribe, 3-12
updateLastLogin, 6-15
MMSPushMessage class, 8-13
MMSPushMsgSender class, 8-19
MMSSender interface, 8-16
MMSSlide class, 8-11
module.security.billingmanager property, 3-12
module.security.contentmanager property, 4-6
module.security.subscriber.usermanager property, 6-15
moveCategoryDown, 10-11
moveCategoryUp, 10-11

O

organization of this guide, xxi

P

provision, 10-23
purchaseContent, 10-15
PushConstants class, 8-18
pushMessage, 8-2
PushMessage class, 8-3
pushMMSContent, 10-16
pushMMSContentByTicket, 10-16
PushMsgListener interface, 8-2
PushMsgSender interface, 8-2
PushResponse class, 8-17

pushSMSContentBinary, 10-16
pushSMSContentByTicket, 10-17

R
refund, 3-11
related documentation, xxiii
reporting tools, 2-6
requestContent, 10-15
resetPasswordBySubscriberId, 10-23
resetPasswordByUsername, 10-23
returns, 5-3
reverse, 3-11

S
searchContent, 10-15
sendEvent, 10-21
sendEventWithParameters, 10-21
sendInstall, 10-17
sendKeepAliveMsg, 8-3
sendMessageToMobileId, 10-19
sendMessageToSelf, 10-19
sendMessageToSubscriberId, 10-19
sendMessageToUsername, 10-19
sendMMS, 8-16
sendMMSContent, 10-19
setActiveDate, 6-10
setAllUserAgents, 8-6
setAttribute, 6-11, 8-7
setAttributes, 6-11
setAudioContent, 8-12
setCity, 6-11
setContentBinary, 8-7
setContentData, 8-11
setContentId, 8-11
setContentMimeType, 8-11
setContentType, 8-7
setCountry, 6-11
setCreateDate, 6-11
setDeActivateDate, 6-11
setDeliveryReportRequired, 8-15
setDestinationAddress, 8-7
setEmail, 6-12
setFirstName, 6-12
setFromAddress, 8-15
setGender, 6-12
setHasLoggedIn, 6-12
setImageContent, 8-12
setIsEnabled, 6-12
setIsPrepay, 6-12
setJMSMessageId, 8-7
setLastName, 6-13
setLocaleCode, 10-23
setLoginId, 6-13
setMessageClass, 8-15
setMessageId, 8-8
setMessagePriority, 8-15
setMessageText, 8-8
setMiddleName, 6-13
setMimeType, 8-8
setMobileId, 6-13
setModelId, 10-23
setPassword, 6-13, 10-23
setPhone, 6-13
setPostalCode, 6-14
setPushCategory, 8-8
setPushDomain, 8-8
setReadReportRequired, 8-15
setSalutation, 6-14
setSenderVisibility, 8-16
setShortCode, 8-8
setSMILPresentation, 8-16
setState, 6-14
setStreet1, 6-14
setStreet2, 6-14
setSubscriberId, 8-9
setTextContent, 8-12
setUniqueDeviceId, 6-14, 8-9
setUserPreferences, 10-24
setUserProperties, 10-24
setVendingContentId, 8-9
setVideoContent, 8-13
showCategory, 10-11
signup, 10-24
signupWithPropertiesAndPreferences, 10-24
SMSMessage class, 8-9

SMTPMessage class, 8-10
SMTPushMsgSender class, 8-19
SQL*Net, 2-3, 2-12
SubmissionVerifierWorkflows.xml, 5-1
subscribe, 3-11
Subscriber API, 1-4, 10-1
  general process flow, 10-2
  using, 10-3
  XML-RPC
    guidelines, 10-7
    method parameters, 10-25
    setting up access, 10-6
    using the handlers, 10-7
SystemHandler, 10-20

T
toString, 8-9
typographical conventions, xxii

U
unsubscribe, 3-12
updateCategories, 10-11
updateLastLogin, 6-15
User interface, 6-6
User Profile api, 1-3
UserDeviceManager interface, 6-15
UserHandler, 10-21
UserManager class, 6-1

V
validation.jar, 5-3
ValidationAdapter, 5-2
ValidationContent, 5-3

W
WAP Gateway API, 1-3
WAP Gateway API, using, 7-3
WAPGatewayAdapter class, 7-2
WapPushMessage class, 8-10
WAPPushMsgSender class, 8-19

X
XML-RPC, *See* Subscriber API