Developers Guide
Self Service for Communications

V5.0.1
Date Published: 4.20.05

Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404
Copyright © 2005 Siebel Systems, Inc.
All rights reserved.
Printed in the United States of America

No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior agreement and written permission of Siebel Systems, Inc.

Siebel, the Siebel logo, UAN, Universal Application Network, Siebel CRM OnDemand, TrickleSync, Universal Agent, and other Siebel names referenced herein are trademarks of Siebel Systems, Inc., and may be registered in certain jurisdictions.

Other product names, designations, logos, and symbols may be trademarks or registered trademarks of their respective owners.

PRODUCT MODULES AND OPTIONS. This guide contains descriptions of modules that are optional and for which you may not have purchased a license. Siebel's Sample Database also includes data related to these optional modules. As a result, your software implementation may differ from descriptions in this guide. To find out more about the modules your organization has purchased, see your corporate purchasing agent or your Siebel sales representative.

U.S. GOVERNMENT RESTRICTED RIGHTS. Programs, Ancillary Programs and Documentation, delivered subject to the Department of Defense Federal Acquisition Regulation Supplement, are "commercial computer software" as set forth in DFARS 227.7202, Commercial Computer Software and Commercial Computer Software Documentation, and as such, any use, duplication and disclosure of the Programs, Ancillary Programs and Documentation shall be subject to the restrictions contained in the applicable Siebel license agreement. All other use, duplication and disclosure of the Programs, Ancillary Programs and Documentation by the U.S. Government shall be subject to the applicable Siebel license agreement and the restrictions contained in subsection (c) of FAR 52.227-19, Commercial Computer Software - Restricted Rights (June 1987), or FAR 52.227-14, Rights in Data—General, including Alternate III (June 1987), as applicable. Contractor/licensor is Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404.

Proprietary Information

Siebel Systems, Inc. considers information included in this documentation and in Siebel eBusiness Applications Online Help to be Confidential Information. Your access to and use of this Confidential Information are subject to the terms and conditions of: (1) the applicable Siebel Systems software license agreement, which has been executed and with which you agree to comply; and (2) the proprietary and restricted rights notices included in this documentation.

Contents

1 Preface

About Self Service Manager for Communications 7

2 Developing with and Extending Self Service Manager

How to Use This Guide 13

Core Functionality 14

Extending Connectivity 14

Accessing Self Service Manager Data 15

Interacting with Self Service Manager Transactions 15

Extending Self Service Manager with Custom Transactions 16

3 Understanding Self Service Manager

Data Loading and Synchronization 17

Bulk Load Features 17

Bulk Load Process 18

Data Load Configuration 19

Self Service Manager Bulk Loading Configuration & Runtime Requirements 22

Bulk Loading Command Syntax 23

Error Handling 24

Bulk Loading Quick Validation 25

Data Formats 25

Search & Update Functionality 29

Self Service Manager Search & Update APIs 29

Self Service Manager Search Sequence Of Events 30

Obtaining Context 31

Obtaining an instance of a `TSMDataManager` 33
Finding a Service Agreement 33
Managing Rate Plan Elements 34
Common Exceptions 35
Purging out of date data 35
Service Agreement Locking 36
Self Service Manager Transaction API 37
Self Service Manager Sequence Of Events 38
Self Service Manager Transaction APIs 39
Obtaining an instance of a `TSMServiceManager` 40
Obtaining an instance of an `IServiceAgreement` 40
Obtaining and populating specific transactions 41
Generate Voice Mail Password Transaction 41
Changing Voice Mail Password Transaction 42
Change Phone Number Transaction 43
Port Phone Number Transaction 44
Change Digital Subscriber Number Transaction 45
Activate/Deactivate/Suspend/Resume Service Transactions 46
Add Delete Feature Transaction 47
Change Rate Plan Request 49
Find Subscriber Profile/Change Subscriber Profile Transactions 51
Obtain Available area codes/exchanges/phone number Transactions 52
Bulk Transaction Handling 53
Transaction Response Handling 54
Common Exceptions 55
Enterprise Systems Connectivity 55
Concepts 55
Defining Destinations 56

Understanding Response Codes: 57

Adding messages to a response request 59

Defining a `Destination` in `sm.xma.xml` 61

Defining Transactions 62

Transaction Lifecycle Events 69

Custom Request and Response Persistence 74

Supporting Asynchronous requests 76

Integrating Transactions 78

Response Processors 81

4 Glossary of Terms

Index

1 Preface

About Self Service Manager for Communications

Siebel's Self-Service for Communications includes every application that communications service providers need to enable a complete online customer-self service experience at their website. The suite includes software applications for:

- e-Billing and Payment
- Service and Order Management
- Point-of-Sale
- Reporting and Analytics
- Rate Plan Advice

Siebel's Self-Service applications for the telecommunications industry combine Siebel's unrivaled Customer Self-Service and e-Billing software suite with its extensive industry domain expertise. The packaged, out-of-the-box applications are tailored to solve communications service providers' distinct business problems and to meet communications industry-specific process requirements.

Siebel's Self-Service for Communications includes:

Communications Billing Manager

Communication Billing Manager is a complete e-billing application for communications service providers that gives business and consumer customers valuable and convenient access to their communications bills along with the ability to easily make online payments.

Communications Self-Service Manager

Communications Self-Service Manager enables customers of communications service providers to manage every aspect of their service relationship online. From a single convenient interface, customers can easily activate and manage subscriptions, change rate plans and features, and modify subscriber profile settings. Business customers are able to complete these activities for individual employees, as well as company departments and divisions, across their entire organization.

Communication Analytics Manager

Communication Analytics Manager is a reporting solution for business customers that empowers both individual employees and business managers to analyze and understand their communications costs and usage by investigating and identifying trends and patterns across multiple views of their own unique organization.

Rate Plan Advisor

Rate Plan Advisor is a web-based application that recommends the ideal rate plan for wireless subscribers in real-time. Individual consumers as well as large businesses can analyze their actual historical voice/mobile/data usage, find the best-fit rate plans, and compare the features offered by those plans. With its intuitive wizard user interface, Rate Plan Advisor quickly guides end-customers or customer service representatives through the entire analysis process. In addition, a service provider’s customer care and marketing groups can also use Rate Plan Advisor to identify pre-churn subscribers, simulate new rate plans, and run predictive analytics.

About This Guide

The Siebel Software Developers Kit allows developers to write custom code against Siebel applications. This guide is intended for Siebel system integrator partners, senior developers with a Siebel client company, and Siebel Professional Services representatives.

The Self Service Manager SDK assumes you have an in-depth understanding of and practical experience with:

- Billing Manager and Self Service Manager system architecture, installation, deployment, application design, and administration
- Java 2 Enterprise Edition (J2EE), Enterprise JavaBeans (EJBs), servlets, and JSPs
- Apache Struts, Tiles and Log4J and Velocity
- Packaging and deploying J2EE applications for WebLogic or WebSphere
- Directory services including the Java Naming Directory Interface (JNDI) and the Lightweight Directory Access Protocol (LDAP)
- HTML and XML, web server administration, and web browsers
- The Spring Framework, see www.springframe.org for more information
- The hibernate persistence framework, see www.hibernate.org for more information

This guide also assumes you have:

- Read the Self Service Manager product documentation and are familiar with Self Service Manager functionality
- Read the reviewed and have ready the javadoc packaged with the Self Service Manager SDK
- Successfully installed Self Service Manager in a J2EE development environment
- Knowledge of how to develop J2EE web applications using JSP, Struts, Tiles and XML

Related Documentation

This guide is part of the Self Service for Communications documentation set including Self Service Manager and Billing Manager. For more information about implementing your Self Service Manager application, see one of the following guides:

Print Document	Description
Self Service Manager Architecture Guide	Overview of the Self Service Manager Architecture and Data Model
Billing Manager and Self Service Manager Installation Guides	How to install Billing Manager and Self Service Manager in a distributed environment.
Billing Manager Presentation Design Guide	How to use Composer to define the rules for mapping data to templates for viewing statements.
Billing Manager Administration Guide	How to set up and run a live Billing Manager application in a J2EE environment.
Billing Manager Data Definition Guide	How to use DefTool to define the rules for data extraction in a DDF file.
Billing Manager Developers Guide	How to extend, develop and otherwise work with the Billing Manager product

Obtaining Siebel Software and Documentation

You can download Siebel software and documentation directly from Customer Central at https://support.edocs.com. After you log in, click on the Downloads button on the left. When the next page appears, you will see a table displaying all of the available downloads. To search for specific items, select the Version and/or Category and click the Search Downloads button. If you download software, an email from Siebel Technical Support will automatically be sent to you (the registered owner) with your license key information.

If you received an Siebel product installation CD, load it on your system and navigate from its root directory to the folder where the software installer resides for your operating system. You can run the installer from that location, or you can copy it to your file system and run it from there. The product documentation included with your CD is in the Documentation folder located in the root directory. The license key information for the products on the CD is included with the package materials shipped with the CD.

If You Need Help

Technical Support is available to customers who have an active maintenance and support contract with Siebel. Technical Support engineers can help you install, configure, and maintain your Siebel application.

This guide contains general troubleshooting guidelines intended to empower you to resolve problems on your own. If you are still unable to identify and correct an issue, contact Technical Support for assistance.

Information to provide

Before contacting Siebel Technical Support, try resolving the problem yourself using the information provided in this guide. If you cannot resolve the issue on your own, be sure to gather the following information and have it handy when you contact technical support. This will enable your Siebel support engineer to more quickly assess your problem and get you back up and running more quickly.

Please be prepared to provide Technical Support the following information:

Contact information:

- Your name and role in your organization.
- Your company's name
- Your phone number and best times to call you
- Your e-mail address

Product and platform:

- In which Siebel product did the problem occur?
- What version of the product do you have?
- What is your operating system version? RDBMS? Other platform information?

Specific details about your problem:

- Did your system crash or hang?

- What system activity was taking place when the problem occurred?
- Did the system generate a screen error message? If so, please send us that message. (Type the error text or press the Print Screen button and paste the screen into your email.)
- Did the system write information to a log? If so, please send us that file. For more information, see the *Troubleshooting Guide*.
- How did the system respond to the error?
- What steps have you taken to attempt to resolve the problem?
- What other information would we need to have (supporting data files, steps we'd need to take) to replicate the problem or error?

Problem severity:

- Clearly communicate the impact of the case (Severity I, II, III, IV) as well as the Priority (Urgent, High, Medium, Low, No Rush).
- Specify whether the problem occurred in a production or test environment.

Contacting Siebel Technical Support

You can contact Technical Support online, by email, or by telephone.

Siebel provides global Technical Support services from the following Support Centers:

US Support Center
Natick, MA
Mon-Fri 8:30am – 8:00pm US EST
Telephone: 508-652-8400

Europe Support Center
London, United Kingdom
Mon-Fri 9:00am – 5:00 GMT
Telephone: +44 20 8956 2673

Asia Pac Rim Support Center
Melbourne, Australia
Mon-Fri 9:00am – 5:00pm AU
Telephone: +61 3 9909 7301

Customer Central
https://support.edocs.com

Email Support
mailto:support@edocs.com

Escalation process

Siebel managerial escalation ensures that critical problems are properly managed through resolution including aligning proper resources and providing notification and frequent status reports to the client.

Siebel escalation process has two tiers:

1. **Technical Escalation -** Siebel technical escalation chain ensures access to the right technical resources to determine the best course of action.

2. **Managerial Escalation** - All severity 1 cases are immediately brought to the attention of the Technical Support Manager, who can align the necessary resources for resolution. Our escalation process ensures that critical problems are properly managed to resolution, and that clients as well as Siebel executive management receive notification and frequent status reports.

By separating their tasks, the technical resources remain 100% focused on resolving the problem while the Support Manager handles communication and status.

To escalate your case, ask the Technical Support Engineer to:

1. Raise the severity level classification

2. Put you in contact with the Technical Support Escalation Manager

3. Request that the Director of Technical Support arrange a conference call with the Vice President of Services

4. Contact VP of Services directly if you are still in need of more immediate assistance.

2 Developing with and Extending Self Service Manager

How to Use This Guide

Developing and deploying Self Service Manager applications requires a core set of functionality be in place. Common tasks a developer must perform are customarily referred to in this guide as *developer use cases*. For each developer use case there may be one or more activities that need to be performed in order to complete the development effort. The remainder of this section details the most common developer use cases and the sections that describe the underlying technology involved to implement each.

Self Service Manager Development falls into several areas:

- Core functionality — required by all Self Service Manager deployments, populating the Self Service Manager data store, writing a connector etc.
- Extending Connectivity — adding hooks and custom processing for single or multiple back end systems
- Accessing Self Service Manager Data — interacting with the Self Service Manager database
- Interacting with Self Service Manager transactions — interacting with existing transactions
- Extending Self Service Manager with custom transactions — adding your own transactions
- Extending the Self Service Manager Data model — adding and accessing customer data within the Self Service Manager data model

Core Functionality

Every Self Service Manager deployment requires a core set of Self Service Manager functionality be created and deployed. Chief of these activities is populating the Self Service Manager database and configuring a Self Service Manager Destination.

In Order To:	Read These Sections	To Perform These Activities
Manage Self Service Manager Data	• Self Service Manager Architecture – data model • Bulk Load Process • Data Load Configuration • Validation	• Understand the core data model • Understand the data loading process • Configure your environment • Validate loaded data
Understand Transactions	• Self Service Manager Architecture – Arch. Overview • Self Service Manager Transactions – sequence of events • Self Service Manager Transactions – specific transactions	• Understand Self Service Manager Core Components • Understand OSS interaction from the front end perspective • Understand transaction specifics
Connect to an existing OSS	• Self Service Manager Architecture – Arch. Overview • Enterprise Sys. Conn. – Concepts • Enterprise Sys. Conn. – Defining Destinations	• Understand Self Service Manager Core Components • Understand ESS Component details • Connecting Self Service Manager to an external OSS

Extending Connectivity

Custom processing can be integrated into Self Service Manager for the purpose of providing specific processing for a given OSS return status. Such processing can be applied to a single OSS or to a set of OSS's.

In order to:	Read These Sections	To Perform These Activities
Add a custom response handler	• Self Service Manager Architecture – Arch. Overview • Enterprise Sys. Conn. – Concepts • Enterprise Sys. Conn. – Response Handlers	• Understand the purpose of response handlers • Understand how response handlers are called • Write and configure a custom response handlers

In order to:	Read These Sections	To Perform These Activities
Perform Custom Persistence Management	• Self Service Manager Architecture – data model • Enterprise Sys. Conn. – Concepts • Enterprise Sys. Conn. – Persistence Management • Search & Update – Entire section	• Understand the core data model • Understand when and why persistence management is important • Write and configure persistence managers • Cascade changes from an OSS back to the Self Service Manager database

Accessing Self Service Manager Data

Most applications will need to interact with the Self Service Manager database in order to search for and update existing data.

In order to:	Read These Sections	To Perform These Activities
Interact with the Self Service Manager Database	• Self Service Manager Architecture – data model • Search and Update – Entire section	• Understand the core data model • Store changes back into the Self Service Manager database Purge the Self Service Manager Database of out of data information

Interacting with Self Service Manager Transactions

Most applications will interact existing Self Service Manager transactions, either for the purpose of writing GUI implementations or otherwise interacting with the Self Service Manager Service manager to perform or interact with transaction processing.

In order to:	Read These Sections	To Perform These Activities
Interact with Core Self Service Manager Transactions	• Self Service Manager Architecture – Arch. Overview • Self Service Manager Transaction API – Seq. of events • Self Service Manager Transactions API – *Specific transactions* • Self Service Manager Transaction API – Bulk Transaction Handling • Self Service Manager Transaction API – Exception Processing • Enterprise Sys. Conn. – Concepts	• Understand the general transaction concepts • Understand transaction processing • Understand and interact with a given transaction • Handle bulk transaction responses • Handle Exceptions • Understand how transaction is handled by OSS

Extending Self Service Manager with Custom Transactions

Custom applications can add there own specific transactions which enhance Self Service Manager to provide support for transactions outside the scope of those prepackaged with Self Service Manager.

In order to:	Read These Sections	To Perform These Activities
Extend the Self Service Manager Transaction set with custom transactions	• Self Service Manager Architecture – Arch. Overview • Self Service Manager Transaction API – Seq. of events • Enterprise Sys. Conn. – Concepts • Enterprise Sys. Conn. – Defining Transactions • Enterprise Sys. Conn. – Integrating Transactions • Enterprise Sys. Conn. – Defining Destinations	• Understand the general transaction concepts • Understand transaction processing • Understand how transaction is handled by OSS • Defining new transactions • Installing and configuring new transactions • Add support at the OSS level for new transactions

3 Understanding Self Service Manager

Data Loading and Synchronization

Self Service Manager is not the system of record with respect to Service Agreements and their underlying data and as such the Self Service Manager database must be loaded and regularly synchronized.
Both service agreements and their constituent parts are loaded in bulk and then regularly synchronized via custom jobs. Key processes with respect to data management are:

- Bulk Data Loading — initial population of Self Service Manager database with rate plans, rate plan features and service agreements instances
- Synchronization — updates to rate plans, rate plan groups, and rate plan features as well as addition of new service agreements.

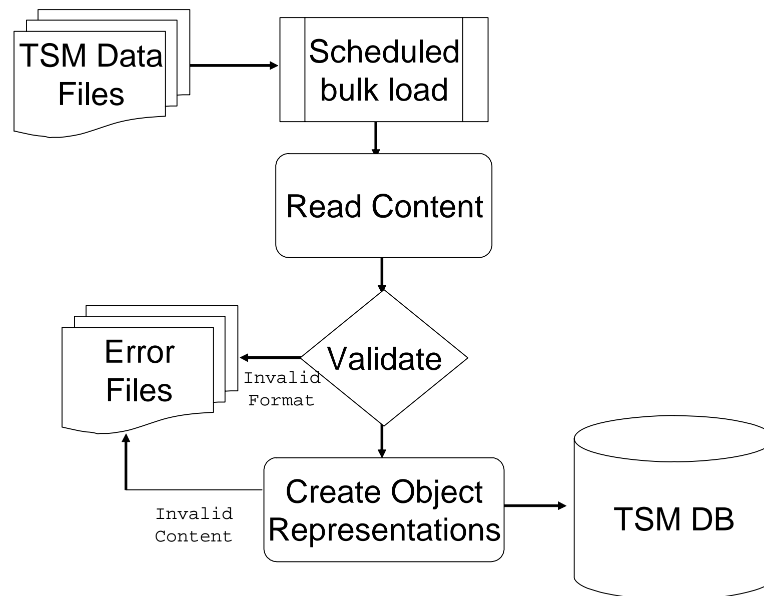
Bulk Load Features

The bulk load process is responsible for initially loading and updates to the Self Service Manager data cache with the customer's service data. Key features of bulk loading are:

- XML-based file format representing each data element, supporting create, update and inactivate functionality.
- Multi-threaded for parallel loading.
- XML-based error log.
- Statistic tracking.
- Configurable — abort threshold, range of records to process., email Notification of abort/Success, JDBC batch size.
- Configurable Data Transfer Object classes for mapping XML to Java
- Start from last abort support
- Database pre & post-load script support

Bulk Load Process

The bulk load process is driven by a set of XML based configuration classes as well as a set of processing classes. The XML configuration defines the set of files to process. Processing classes, implemented based on SAX, are responsible for handling each of the underlying data elements.



Process flow:

0. Data files are copied to a directory specified by the bulk load process.
1. Scheduled bulk load begins.
2. Content is reviewed and validated for correct format against DTD, error files are produced for any validation errors.
3. For each element in the file being loaded:
    a. Perform the required operation, for example create appropriate object representation, log invalid content to error files as appropriate.
    b. Complete the operation against the Self Service Manager database.

Data Load Configuration

The Self Service Manager data loading process is configured via `tsmproperties.xml` found in the Self Service Manager installations `config` directory (for example, %ETL_HOME%\config\tsmproperties.xml).

Self Service Manager Configuration files are validated against the DTD below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT tsm-configuration (file-config, files-to-
process)>
<!ELEMENT file-config EMPTY>
<!ATTLIST file-config
   data-dir CDATA #REQUIRED
   error-dir CDATA #REQUIRED
   mapping-dir CDATA #REQUIRED>

<!ELEMENT files-to-process (file+)>

<!ELEMENT file EMPTY>
<!ATTLIST file
   type CDATA #REQUIRED
   process-file CDATA #REQUIRED
   input-name CDATA #REQUIRED
   object-map CDATA #REQUIRED
   node-name CDATA #REQUIRED
   error-name CDATA #REQUIRED
   record-start CDATA #REQUIRED
   record-end CDATA #REQUIRED
   abort-total CDATA #REQUIRED
   abort-process CDATA #IMPLIED
   verbose CDATA #REQUIRED
   jdbc-batch-size CDATA #REQUIRED
   commit CDATA #REQUIRED
   parser-class CDATA #REQUIRED
   run-as-user CDATA #REQUIRED
   run-as-user-pw CDATA #REQUIRED
   threads CDATA #IMPLIED
   node-read-size CDATA #IMPLIED
   node-message-frequency CDATA #IMPLIED>
```

`file-config` Attributes

Attribute	Meaning
data-dir	Data file input directory. Will append `input-name` to this specification at runtime.
error-dir	Error file output directory. Will append `error-name` to this specification at runtime.
Mapping-dir	Mapping file input directory. Will append `object-map` to this specification at runtime.

`file` Attributes

Attribute	Meaning
`type`	Identifies the type of file to process. ETL supports **`device-type`**, **`rate-plan, rate-plan-group`** and **`service-agreement`**. Note that Service Agreement groups must be added last as they may require or be based on elements from the device type or rate plan groups.
`input-name`	File to be processed, prepended with `data-dir`
`object-map`	Identifies the file that contains the object mapping information that is used to map data from the input file to its corresponding DTO object, prepended with `mapping-dir`.
`node-name`	Identifies the name of the node that contains the data to be imported. This is used by XPATH to return the list of nodes
`output-name`	The name of the output file if the data is to be transformed to a pipe-delimited file
`error-name`	The name of the file to store errors that occurred during the ETL process, prepended with `error-dir`.
`record-start`	The starting record (node) in the data file. If this is left blank, ETL starts from the beginning of the file.
`record-end`	The ending record (node) to stop processing. If this is left blank, ETL runs until it reaches the end of the file.
`abort-total`	Indicates the number of errors that have occurred before the process is aborted.
`abort-process`	Indicates if the entire ETL process is to be aborted if the errors that occurred are greater than the value specified by abort-total
`threads`	Indicates the number of threads used to process the import file. This is only applicable for the service agreement import. Each thread creates a separate session.
`verbose`	Indicates if messages are displayed during processing
`jdbc-batch-size`	Not currently used
`commit`	If the value is set to true, data is persisted through the Self Service Manager object model.
`parser-class`	Indicates the class to use to perform the parsing of the data file.
`run-as-user`	A valid Self Service Manager user that will be used while importing data.
`run-as-user-wd`	Password for `run-as-user`.

Example TSMProperties.xml

```
<files-to-process>
    <file-config
        data-dir="c:\tsm\data\"
        error-dir="c:\tsm\data\"
        mapping-dir="c:\tsm\data\"/>
    <file type="device-type"
        input-name="big-telco-device-data.xml"
        object-map="tsm-device-type-mapping.xml"
        node-name="//tsm-device-type//DeviceType"
        error-name=" device-loading-error.xml"
        record-start='1" record-end="100"
        abort-total="0" abort-process="true"
        verbose="false"
        jdbc-batch-size="100"
        commit="true"
        parser-class=

"com.edocs.tsm.etl.process.TSMProcessDeviceTypes"
        run-as-user"import-admin"
        run-as-use-pw="importadmin"
    />
</files-to-process>
```

Self Service Manager bulk data loading has a predefined loading order which is important since the data has dependencies. Load order is listed below:

1. Device Types
2. Rate Plan Groups
3. Rate Plans
4. Service Agreements

Self Service Manager Data Transfer Objects

Self Service Manager Data Transfer Objects (DTO) are used to map data from the XML data files to Self Service Manager Java objects. Transfer objects are used to manipulate the data and assign it to the Self Service Manager persistence API or output it to a different format. The following DTO classes are provided:

Class	Description
com.edocs.application.tsm.etl.dto.TSMAgreementDTO	This class represents data contained in the service agreement import XML file.
com.edocs.application.tsm.etl.dto.TSMDeviceTypeDTO	This class represents data contained in the Telco Device Types import XML file.

Class	Description
`com.edocs.application.tsm.etl.TSMRatePlanGroupDTO`	This class represents data contained in the Telco Rate Plans import XML file.
`com.edocs.application.tsm.etl.dto.TSMRatePlanDTO`	This class represents data contained in the Rate Plan import file.
`com.edocs.application.tsm.etl.TSMRatePlanFeatureDTO`	This class represents a Rate Plan Feature.
`com.edocs.application.tsm.etl.` `TSMRatePlanFeatureInstanceDTO`	This class represents a Rate Plan Feature associated with a Rate Plan.
`com.edoc.application.tsm.etl.` `TSMServiceAgreementAttributesDTO`	This class represents data associated with a Service Agreement attribute.

Self Service Manager Bulk Loading Configuration & Runtime Requirements

The Self Service Manager Bulk loader requires the following be installed and configured:

- JDK 1.4 or higher. If multiple versions of the JDK are installed, 1.4 should be first in the PATH.
- Ant 1.6 or higher: The default bulk load process uses ANT via buildrun.xml. See Siebel Professional Services if you DO NOT wish to use ANT.
- Billing Manager 4.5.1 or higher
- Self Service Manager 5.0.0 or higher

Note that it is assumed that while the Self Service Manager bulk loader can be run separate from Self Service Manager, that Billing Manager and Self Service Manager were installed and configured normally with the Billing Manager/Self Service Manager install process.

Confirm that the TSMDataSource Property element in *tsm-xma.xml* matches your hibernate settings as described in the Billing Manager and Self Service Manager installation guides. For reference the default setting is shown below:

tsm.xma.xml snippet

```
<bean id="TSMDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <property name="driverClassName">
        <value>oracle.jdbc.driver.OracleDriver</value>
    </property>
    <property name="url">

<value>jdbc:oracle:thin:@localhost:1521:edx0</value>
    </property>
    <property name="username">
        <value>edx_dba</value>
    </property>
    <property name="password">
        <value>edx</value>
        </property>
</bean>
```

Bulk Loading Command Syntax

The Self Service Manager Bulk Loading process is implemented as a Java program that is run via ANT. Once ANT has been installed correctly and the ant command is in your path, the Self Service Manager Bulk load process can be run using the following syntax:

```
ant -f buildrun.xml [target]
```

Where target is any one of:

Target	Description
`Usage`	Displays all available targets. `ant -f buildrun.xml usage`
`etl-run`	Bulk load per the TSMProperties file. No previously data is removed. `ant -f buildrun.xml etl-run`
`etl-run-truncate`	Runs the ETL process and TRUNCATES ALL device types, rate plan groups, rate plans and service agreements. This ANT target executes the SQL file `ddl/trunc.sql.` `ant -f buildrun.xml etl-runtruncate`
`etl-config`	Displays the configuration that will be used when bulk load, no load is performed. `ant -f buildrun.xml etl-config`
`etl-ivp`	Runs an configuration check to make sure that bulk has been installed correctly using the parameters specified by the `-DinitFile` option. `ant -f buildrun.xml etl-ivp`
`etl-stats`	The ETL process is run but the data is not committed. This parameter should be used in conjunction with the `com.edocs.application.tsm.etl.runQueue` logging parameter.

Error Handling

Logging Errors

The bulk loader logs all errors as they are encountered. Two specific logs are maintained: The **`Node File`** contains all data nodes that were not processed successfully. A file is created for each import (Device Type, Rate Plan Group, Rate Plans and Service Agreements). The name of the file is specified within the Self Service Manager properties file. The entry *error-file* must be a valid filename.

The **`Log File`** supports application level logging. The **`log4j.xml`** property files file located in the **`config`** directory controls logging levels and output. All errors that are encountered are simultaneously written to the specified output. See http://logging.apache.org for detailed information about how to configure log4j logging.

System Errors

All system errors are written to the log file. System errors may include the items listed below:

- Invalid input file names
- Invalid output file names
- IO Errors such as disk full

Database Errors

The bulk loading process will write the data node causing the problem to the Node file and a stack trace of the error to the system log.

Data Errors

Bulk loading supports data validation in two passes. The first pass is to validate that the contents of the data files conform to their assigned DTD. This should be done before the import process is run. See the section on Validating Data for information on how to run this data. The second pass occurs during the process of loading data. Bulk loading will validate the data constraints listed below.

- All Rate Plans belong to a valid Rate Plan Category.
- All Service Agreements have a valid Rate Plan.
- All Service Agreements have a valid Device Type.
- All Service Agreements can be assigned to a valid communications customer.

Aborting the Bulk Loading Process

Bulk Loading is aborted under the conditions specified in the Self Service Manager properties file. Counters are maintained of all errors for each data file. The error counter is reset for each file. If the error count exceeds the **`abort-total`** specified for the data file, that load process aborts. If the **`abort-process`** setting is set to **`true`**, the entire ETL process is aborted.

Data validation in a two-pass process. During the first pass, data *format* is validated against the appropriate DTD. During the 2nd pass data, *values* are validated for correctness.

Bulk Loading Quick Validation

The Self Service Manager Bulk Loading process results can be validating by running on of the queries listed in the table below:

Validation Queries

Validates	**Query**
Device Types	`SELECT COUNT(*) FROM EDX_TSM_DEVICE_TYPE;` The returned value should match the number of Device Type elements in the input data.
Rate Plan Group	`SELECT COUNT(*) FROM EDX_TSM_RATEPLAN_GROUP;` The returned value should match the number of Rate Plan Group elements in the input data.

Rate Plans	`SELECT COUNT(*) FROM EDX_TSM_RATEPLAN;`
	The returned value should match the number of rate plan elements in the input data.
	`SELECT COUNT(*) FROM EDX_TSM_RP_FEATURE;`
	The returned value should match the number of RatePlanFeature elements in the input data.

Service Agreements	`SELECT COUNT(*) FROM EDX_TSM_SERVICE_AGRMNT;` The returned value should match the number of ServiceAgreement elements in the input data

Data Formats

Data is loaded into the Self Service Manager database via a well-defined set of DTDs. These DTDs define the content for the element to be loaded and are used to validate the content is well-formed.

Device Types

```
<!ELEMENT tsm-device-type ( DeviceType* ) >
<!ELEMENT DeviceType (Description) >
<!ATTLIST DeviceType
    Code CDATA #REQUIRED
    action CDATA #REQUIRED
        type (create|change|delete|inactivate) "create"
>
<!ELEMENT Description (#PCDATA) >
```

Device Type Example

```
<?xml version="1.0"?>
<!DOCTYPE tsm-device-type SYSTEM
          "../config/castor/dtd/tsm-device-type.dtd">
<tsm-device-type>
    <DeviceType Code="MOTOROLA80" action="create">
        <Description>Motorola 80</Description>
    </DeviceType>
    <DeviceType Code="NOKIAI730" action=" update">
        <Description> Nokia i730 -with
Slider</Description>
    </DeviceType>
    <DeviceType Code=" NOKIA3120" action="inactivate">
        <Description>Nokia 3120</Description>
    </DeviceType>
</tsm-device-type>
```

Rate Plans

```
<!ELEMENT tsm-rate-plan ( RatePlan* ) >
<!ELEMENT RatePlan (Description, Price, RatePlanGroup,
RatePlanFeature*) >
<!ATTLIST RatePlan
    Name CDATA #REQUIRED
    action CDATA #REQUIRED
    type (create|change|delete|inactivate) "create">
<!ELEMENT Description (#PCDATA) >
<!ELEMENT Price (#PCDATA) >
<!ELEMENT RatePlanGroup (#PCDATA)>
<!ELEMENT RatePlanFeature (FeaturePrice,
                           FeatureDescription,
                           Unit, Included)>
<!ATTLIST RatePlanFeature FeatureName CDATA #REQUIRED >
<!ELEMENT FeaturePrice (#PCDATA)>
<!ELEMENT FeatureDescription (#PCDATA) >
<!ELEMENT Unit (#PCDATA)>
<!ELEMENT Included (#PCDATA)>
```

Rate Plans Groups

```
<!ELEMENT tsm-rate-plan-group ( RatePlanGroup* ) >
<!ELEMENT RatePlanGroup (Description) >
<!ATTLIST RatePlanGroup
    Name CDATA  #REQUIRED
    action CDATA #REQUIRED
    type (create|change|delete|inactivate) "create">
<!ELEMENT Description (#PCDATA) >
```

Service Agreement Attributes

```
<!ELEMENT tsm-agreements ( Agreement* ) >
<!ELEMENT Agreement (Name, Description, SAStatus,
                     PPUStreet1,
                     PPUStreet2, PPUCity, PPUState,
                     PPUZipCode1, PPUZipCode2,
                     ESN,
                     Subscriber,
                     DeviceTypeKeyCode,
                     RatePlanKeyCode) >
<!ATTLIST Agreement LegacyId CDATA #REQUIRED >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Description (#PCDATA) >
<!ELEMENT SAStatus (#PCDATA)>
<!ELEMENT PPUStreet1 (#PCDATA) >
<!ELEMENT PPUStreet2 (#PCDATA) >
<!ELEMENT PPUCity (#PCDATA) >
<!ELEMENT PPUState (#PCDATA) >
<!ELEMENT PPUZipCode1 (#PCDATA) >
<!ELEMENT PPUZipCode2 (#PCDATA) >
<!ELEMENT ESN (#PCDATA) >
<!ELEMENT Subscriber (#PCDATA) >
<!ELEMENT DeviceTypeKeyCode (#PCDATA) >
<!ELEMENT RatePlanKeyCode (#PCDATA) >
```

Service Agreements

```
<!ELEMENT tsm-agreements (Company*) >
<!ELEMENT Company (BillingAccount*) >
<!ATTLIST Company Name ID #REQUIRED >
<!ELEMENT BillingAccount (Agreement*) >
<!ATTLIST BillingAccount AccountNumber ID #REQUIRED >
<!ELEMENT Agreement (Name,
                     Description,
                     PPUStreet1, PPUStreet2,
                     PPUCity, PPUState,
                     PPUZipCode1, PPUZipCode2,
                     DSN,
                     SubscriberFirstName,
                     SubscriberLastName,
                     DeviceTypeKeyCode, RatePlanKeyCode,
                     Status,
                     RatePlanFeatureInstance*) >
<!ATTLIST Agreement
    LegacyId CDATA #REQUIRED
    action CDATA #REQUIRED
        type (create|change|delete|inactivate) "create">
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Description (#PCDATA) >
<!ELEMENT PPUStreet1 (#PCDATA) >
<!ELEMENT PPUStreet2 (#PCDATA) >
<!ELEMENT PPUCity (#PCDATA) >
<!ELEMENT PPUState (#PCDATA) >
<!ELEMENT PPUZipCode1 (#PCDATA) >
<!ELEMENT PPUZipCode2 (#PCDATA) >
<!ELEMENT DSN (#PCDATA) >
<!ELEMENT SubscriberFirstName (#PCDATA) >
<!ELEMENT SubscriberLastName (#PCDATA) >
<!ELEMENT DeviceTypeKeyCode (#PCDATA) >
<!ELEMENT RatePlanKeyCode (#PCDATA) >
<!ELEMENT Status (#PCDATA)  >

<!ELEMENT RatePlanFeatureInstance (RatePlanFeatureName,

RatePlanFeatureDescription,
                                   FeaturePrice,
                                   Unit,
                                   Included,
                                   Activated) >
<!ELEMENT RatePlanFeatureName (#PCDATA) >
<!ELEMENT RatePlanFeatureDescription (#PCDATA)>
<!ELEMENT FeaturePrice (#PCDATA)>
<!ELEMENT Unit (#PCDATA)>
<!ELEMENT Included (#PCDATA)>
<!ELEMENT Activated (#PCDATA)>
```

Search & Update Functionality

The Self Service Manager Search & Update APIs are designed to support the ability to query for instances of Service Agreements, Rate plans, Rate Plan Groups, Rate Plans by Group etc within the local Self Service Manager database. Additionally the API also supports the ability to update various elements within the Self Service Manager database. *Note that searches and updates are handled and stored LOCALLY and not via the system of record. Changes to the system or record are made via Self Service Manager Transactions described later on in this document.*

Three specific search types are provided:

- Search for a service agreement, rate plan or other element based on a fixed search key identifier. Examples of such a searches are: `TSMDataManager.getServiceAgreement(String phoneNumber)` `TSMDataManager.getDeviceType(String deviceTypeName)`.
- Search for a service agreement within a specific hierarchy. An example of such a search is: `TSMDataManager.findServiceAgreementsByRatePlanGroup(…)`.
- Search for a service agreement within a specific hierarchy using a partial match such as “ends with”. An example of such a search is: `TSMDataManager.findServiceAgreementsBySubscriberName(.…)`

For a complete listing of all arguments, methods and exceptions for the Self Service Manager Search and Update APIs see the Javadoc for the package *`com.edocs.domain.telco.tsm.*`*.

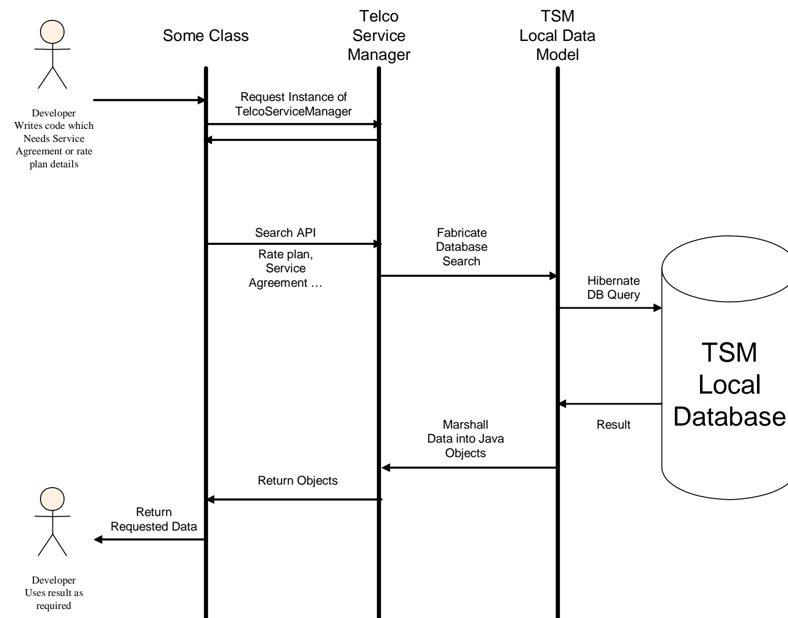
Self Service Manager Search & Update APIs

Developers interact with the Self Service Manager data model via a series of interfaces and implementation classes. Key Self Service Manager Search API classes and interfaces are:

- `com.edocs.domain.telco.tsm.ITSMDataManager & TSMDataManager`– Interface and implementation class for communications-domain-specific searching and updating of the Self Service Manager database.
- `com.edocs.domain.telco.api.tsm.IServiceAgreement` – APIsInterface to Service agreement. Contains all methods for interacting with an instance of a service agreement such as `getRatePlanFeatures`, etc.
- `com.edocs.domain.telco.api.tsm.IDeviceType` – Interface to device types such a Nokia8080 etc.

- `com.edocs.domain.telco.api.tsm.IRatePlanGroup` – Interface to grouped collections of rate plans.
- `com.edocs.domain.telco.api.tsm.IRatePlan` – Interface to definitions of individual rate plans. Note that service agreements start out as copies of the features of rate plans. `IRatePlan` instances are generic rate plan definitions.
- `com.edocs.domain.telco.api.tsm.IRatePlanFeature` – Interface to the individual features of a rate plan.
- `com.edocs.domain.telco.api.tsm.IRatePlanFeatureInstance` – Interface to feature instances within a service agreement. Feature Instances differ from features in the same way java classes differ from java class instances. Features are definitions, feature instances are specific features associated with a rate plan.

Self Service Manager Search Sequence Of Events



1. Obtain context information
2. Obtain an instance of `TSMDataManager`
3. Perform search using appropriate API

Obtaining Context

Search and update function in terms of user context. Context elements typically include the current user identify, user hierarchy position, lists of subscriber phone numbers and other information. Contextual data is typically obtained in web applications within `EdocsAction` extended classes that access the Billing Manager Customer Account Management(CAM) API and the `SessionUtils` APIs.

Context is initialized after a successful authentication and resides in session, application and request scope as appropriate, maintained by the Servlet container. See the Billing Manager javadoc for a complete list of the Session Utils APIs.

Various Self Service Manager APIs require an `IUser` instance. `IUser` instances can be obtained as shown below:

Obtaining `IUser` instances in an action class

```
import com.edocs.common.api.bsl.ISession;
import com.edocs.common.bsl.umf.api.IUser;
import org.apache.struts.action.*;
import javax.servlet.http.*;

public class getPhoneNumberExample extends EdocsAction {
    public ActionForward doAction(
                                  ActionMapping
actionMapping,
                                  ActionForm actionForm,
                                  HttpServletRequest
req,
                                  HttpServletResponse
res)
                                  throws RIException {
      ISession session
=SessionUtils.getUserSession(req);
      IUser user = session.getUser();
      String uid = user.getUID();
      // Other code appropriate to the action
  }
}
```

`ICustomer` instances represent the currently subscribed user. `ICustomer` can be used to obtain other specifics about subscriber such as the set of associated phone numbers.

`IUser` – `ICustomer` relationship

There is a one-to-one relationship between `IUser` and `ICustomer`. This relationship represents the logical joining of a “user name” and the underlying subscriber that username represents. In B2C applications `IUser` is normally assigned by a customer action which requires specifying a phone number, contract information or other billing data used to identify the customer. In B2B applications the connection between `IUser` and `ICustomer` might be made by a CSR but they act identically at run time.

Obtaining `ICustomer` instances in an action class

Add the following import to any action class and the code snippet below to obtain an `ICustomer` instance and from it the list of associated phone numbers.

```
import com.edocs.application.tbm.cam.api.*;


ICustomer subscriber = CAMClassFactory.getCustomer(req);
// Get Current Account
ICustomerAccount account =
subscriber.getCurrentAccount();
// Get all accounts and select
String[] allAccountNumbers =
subscriber.getAccountNumbers();
ICustomerAccount specificAccount =

subscriber.getAccount(allAccountNumbers[0]);
String phoneNumber = account.getAccountNumber();
```

`IHierarchyNode` instances represent the currently subscribed users position in a corporate hierarchy and is required for search- and update-related methods.

Obtaining `IHierarchyNode` instances in an action class

The Business Services Layer, or BSL, provides a variety of functions to the developer, one of which is access to hierarchy information. BSL can be used as shown below to obtain the current hierarchy node for use in subsequent calls.

```
import com.edocs.common.api.bsl.*;
import com.edocs.common.hierarchy.api.*;
import com.edocs.common.hierarchy.api.core.*;

IUser user = session.getUser();


IBusinessServices bsl = new BusinessServices();
IHierarchyNode[] hierarchy =
bsl.findHierarchyNodesForUser(user.getIdentityID());
if ((hierarchy == null) || (hierarchy.length == 0)) {
    // unexpected error
}
```

Most often the Self Service Manager Database Search and Update API calls require the root of a hierarchy which is provided as `hierarchy[0]`.

Obtaining an instance of a `TSMDataManager`

The Telco Services Data Manager interface contains a factory method for returning instances of classes implementing `ITSMDataManager` and interacting with the underlying Self Service Manager database. . `ITSMDataManager` instances can be obtained as shown below:

```
import com.edocs.domain.telco.tsm.*;
. . .
HttpServletRequest req;
ISession s =SessionUtils.getUserSession(req);
ITSMDataManager tdm =
    TSMDataManager.getTSMDataManager(s);
```

Finding a Service Agreement

Service agreements can be obtained either via a phone number or via selection criteria. Phone number, subscriber name, rate plan, etc. are all supported searches. Search examples for the most common use cases as shown below.

Find Service Agreement By Phone Number

```
ICustomer subscriber = CAMClassFactory.getCustomer(req);
ICustomerAccount account =
subscriber.getCurrentAccount();
String phoneNmbr = account.getAccountNumber();
IServiceAgreement sa = tsm
getServiceAgreement(phoneNmbr);
```

Find Service Agreement By Rate Plan Group

```
IUser user = // as shown previously
ITSMDataManager tdm = // as previously shown
IHierarchyNode[] hierarchy = // as previously shown
ServiceAgreementStatus status =
ServiceAgreementStatus.ACTIVE;
IServiceAgreement[] sas =
tdm.findServiceAgreementsByRatePlan(user.getIdentityID()
,
                                    hierarchy[0],
                                    "SomeRatePlan,
                                     status);
```

Several search methods may search using “starts with”, “ends with” type syntax. Search types are:

- `SearchType.EXACT` – return only exact matches for the provided search string
- `SearchType.CONTAINS` – return any match which contains the provided search string
- `SearchType.STARTS_WITH` – return any match which starts with the provided search string

Find Service Agreements By Subscriber Name

"Find by subscriber name" is an example of a search that can select service agreements based on search types.

```
IUser user = // as shown previously
ITSMDataManager tdm = // as previously show
IHierarchyNode[] hierarchy = // as previously shown
IServiceAgreement[] sas =
     tdm.findServiceAgreementsBySubscriberName(
                           user.getIdentityID(),
                           hierarchy[0],
                           "lastnameprefix",
                            SearchType.STARTS_WITH);
```

Managing Rate Plan Elements

Devices, Rate Plans, Rate Plan Groups, and Rate Plan features can be managed via the Self Service Manager Search and Update API. Typical uses of these API's are:

- Obtain an instance of a specific element such as `IDeviceType`.
- Obtain all instances of a specific element.
- Create a new instance of an element[1].
- Update existing element values.
- Set an element inactive

Return All `IDeviceType` objects.

```
ITSMDataManager tdm = // as previously show
IDeviceType[] dt = tdm.getAllDeviceTypes()
```

[1] Note that newly created elements are ONLY maintained within the edocs TSM database and NOT propagated back to the system of record.

Return A specific `IDeviceType`.

```
ITSMDataManager tdm = // as previously show
IDeviceType dt = tdm.getDeviceType("Nokia8080");
```

Create A new `IDeviceType` objects.

```
IDeviceType dt = new DeviceType("Kyocera SE47",
                                "Kyocera slide phone");
ITSMDataManager tdm = // as previously show
IDeviceType dt = tdm.createDeviceType (dt);
```

Update an existing `IDeviceType`.

```
ITSMDataManager tdm = // as previously show
IDeviceType dt = tdm.getDeviceType("Nokia8080");
dt.setDescription("some update");
tdm.updateDeviceType(dt);
```

Deactivate an existing `IDeviceType`.

```
ITSMDataManager tdm = // as previously show
IDeviceType dt =.getDeviceType("Nokia8080");
tdm.deactivateDevice(dt);
```

Common Exceptions

The Self Service Manager methods may throw exceptions from the package: `com.edocs.domain.telco.api.tsm.exception.*`.

Exceptions fall into three specific areas:

- `InvalidArgumentException` – Programmer error or a particular argument is invalid, for example a call to obtain the current hierarchy element failed or an attempt was made to purge a data model element which is still referenced by other elements.
- `DataStoreException` – Self Service Manager cannot contact the underlying database. Normally a configuration error.
- `CommunicationException` – Self Service Manager cannot contact the underlying ESS layer, not thrown by search and update methods.

Purging out of date data

Over time devices, rate plans, rate plan groups and service agreements become inactive. The `TSMDataManager` provides specific functions for purging the database of such stale data.

Purge `IServiceAgreement` objects.

Purge a specific Service Agreement.

Example: Purge inactive service agreements

```
IUser user = // as shown previously
ITSMDataManager tdm = // as previously shown
IHierarchyNode[] hierarchy = // as previously shown
ServiceAgreementStatus status =
ServiceAgreementStatus.INACTIVE;
IServiceAgreement[] sas =
tdm.findServiceAgreementsByRatePlan(user.getIdentityID()
,
                                    hierarchy[0],
                                    "SomeRatePlan,
                                     status);
for (int i = 0; i< sas.length;i++)
    purgeServiceAgreement(sas[i]);
```

Purge `IDeviceType` objects.

Purge a specific Device Type. Throws `InvalidArgumentException` if referenced and the device type is not purged.

```
void ITSMDataManager purgeDeviceType(IDeviceType dt);
```

Purge `IRatePlan` objects.

Purge a specific Rate Plan. Throws `InvalidArgumentException` if referenced by a service agreement and the Rate Plan is not purged.

```
void  purgeRatePlan(IRatePlan rp);
```

Purge `IRatePlanGroup` objects.

Purge a specific Rate Plan Group. Throws `InvalidArgumentException` if referenced and the rate plan group is referenced by a rate plan and is not purged.

```
void purgeRatePlanGroup(IRatePlanGFroup rpg);
```

Service Agreement Locking

Under normal circumstances Service Agreements should not be modified while a transaction is pending. However the Self Service Manager Database manager is not tied directly to the Self Service Manager Transaction subsystem, but rather they are loosely coupled independent sub systems. Service agreements can be marked part of a order, with associated id, while a Self Service Manager Transaction is in progress. The `ITSMDatabaseManager` interface provides

```
void setServiceAgreementStatus(IServiceAgreement,
                               Long,
                               ServiceAgreementStatus
status);
```

and

```
Transactionsvoid resetOrderInProgress(IServiceAgreement
sa);
```

Either of which might throw:

`InvalidArgumentException` – if invalid arguments are passed

or

`DataStoreException` – if a database operation fails.

Self Service Manager uses these API's to lock and unlock Service Agreements while transactions are in progress.

Self Service Manager Transaction API

The Self Service Manager Transaction APIs are designed to support the ability to interact with an OSS/BSS to manage various elements associated with a subscribers rate plan. Core to managing features and services is the concept of a *Self Service Manager transaction*. Self Service Manager transactions encapsulate taking an agreement or set of agreements and managing current feature set for those agreements. Transactions could be as simple as changing a single feature such as voice mail password, or as complex as completely re-writing an service agreement feature set.

The Self Service Manager Transaction API manages Service Agreement state by creating sets of Self Service Manager Transactions. Self Service Manager Transactions currently fall into the following categories:

- Add/Delete Features – Add or delete a specific feature from a service agreement.
- Change Voice Mail Password – Change the voice mail password associated with a given device.
- Change Mobile Phone Number – Change the mobile phone number associated with a given device.
- Port Phone Number – insert a pre-existing phone number from another provider.
- Change Device Serial Number – Change the serial number associated with a device.
- Change Rate Plan – Select a different rate plan.

- Suspend or Resume service – Suspend or resume service for a given device.
- Change Subscriber Profile – Update the subscriber information for a given service agreement.

The Self Service Manager Service Manager supports two types of transactions, *Transient* and *Persisted*. Both types of transactions can interact with the underlying OSS layer.

Transient transactions are typically “helper functions”, and do not participate in the transaction lifecycle – and don’t require an audit trail.

Persisted transactions both interact with the underlying OSS, an manage and track the lifecycle of the transaction. Persistent transactions save information into the local Self Service Manager database about transaction state, request and result data etc.

Any of the Persistent transactions can be *Synchronous* or *Asynchronous*.

Synchronous transactions will wait for a response for the server and block for the lifetime of the request.

Asynchronous transactions send a request to the Legacy System and expect to complete some time “later”. Asynchronous transactions return a acknowledgment which puts the Service Request into a “Pending State”, waiting for an asynchronous reply to be received from the legacy system.

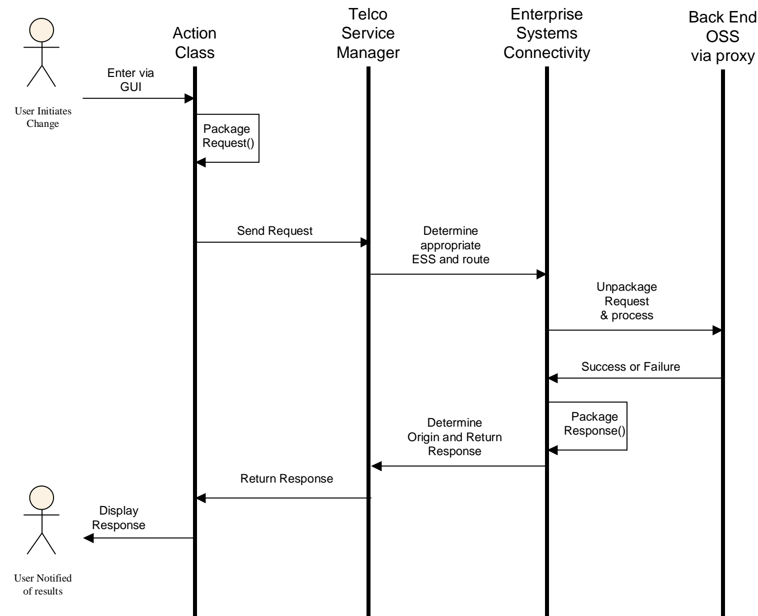
For a complete listing of all arguments, methods and exceptions for the Self Service Manager Transactions APIs see the Javadoc for the packages:

- `com.edocs.common.tsm.sm.core`.* - the Self Service Manager transactions defined specific to the Self Service Manager implementation.
- `com.edocs.common.sm.api`.*. – general Service Manager controlling lifecycle management etc .

Self Service Manager Sequence Of Events

The state of a service agreement contents is changed via a Self Service Manager Transaction Request. Transaction Requests, representing a request to change a given feature or value, are passed to the Self Service Manager. Self Service Manager then routes the request to the registered proxy/connector for fulfillment. The proxy completes or rejects the request and then in real-time returns a Transaction *Response*. Transaction Responses encapsulate the result of the request and are in turn returned to the calling method via Self Service Manager.

Action Sequence Diagram - Synchronous



Self Service Manager Transaction APIs

Developers interact with back-end OSS systems via a set of Self Service Manager Transaction APIs. Are core set of transactions is provided with Self Service Manager along with the ability to add custom Self Service Manager Transactions.

Key Self Service Manager Transaction API classes and interfaces are:

- `com.edocs.common.tsm.sm.core.ITSMServiceManager` & `TSMServiceManager` – Interface and implementation class for interacting with the underlying service manager to create and manage transactions pertaining to the Self Service Manager.
  - `com.edocs.common.sm.api.ISMServiceRequest` – generic service request. Contains all the details of a specific update or change. Rarely used directly, derived classes provide specific instances of service requests.

    The `ISMServiceRequest` has the following major sub-components Case – Hook for the lifecycle management object model.
  - `ISMAction` – Defines the “type” of transaction (with ramifications on response handling)

    - `ISMRequest` - Wrapper for the request data associated with the "Action"
    - `ISMResponse` – Wrapper for the response data associated with the "Action"
    - `ISMCredentials` – Wrapper for the credentials for the specific system.
- `com.edocs.common.sm.api.ISMServiceResponse` – The result of issuing a Self Service Manager transaction request. Returned by `TSMServiceManager.processRequest().com.edocs.common.sm.api.ISMCredentials`,
- `com.edocs.common.tsm.sm.core.TSMCredentials` – Class encapsulation of user instance data required for transaction support. `TSMCredentials` wraps the credentials for the Self Service Manager user in an SM- required format.

  Running a transactions require several specific steps

1. Search – find an instance of a `IServiceAgreement`. See prior sections for details of searching.
2. Obtain and prepare an instance of `ITSMServiceManager`
3. Create and populate a set of request data
4. Execute the transaction against the service manager
5. Examine the resulting response

Obtaining an instance of a `TSMServiceManager`

The `TSMServiceManager` interface contains a factory method for returning instances of classes implementing `ITSMServiceManager`. . `ITSMServiceManager` instances can be obtained as shown below:

```
import com.edocs.domain.telco.tsm.*;
. . .
ITSMServiceManager tsm =
    TSMServiceManager.getTSMServiceManager();
```

Obtaining an instance of an `IServiceAgreement`

Previous sections have detailed the process of obtaining an instance of an `IServiceAgreement`. However since almost all service request transactions depend on service agreement a simple example is provided below:

```
00 ICustomer subscriber =
01    CAMClassFactory.getCustomer(request);
02 ICustomerAccount account =
subscriber.getCurrentAccount();
03 String phoneNmbr = account.getAccountNumber();
04 ITSMDataManager tdm =
05    TSMDataManager.getTSMDataManager(s);
06 IServiceAgreement sa = tdm
getServiceAgreement(phoneNmbr);
```

Obtaining and populating specific transactions

The Self Service Manager Service Manager is an extension of the underlying service manager and adds functionality specific to supporting communications-domain-specific transactions. Transaction instances are obtained via method calls against a `TSMServiceManager` instance and then populated as required by the underlying transaction. In general the process entails obtaining a transaction request object, populating that object with appropriate content, and then submitting the request to the `TSMServiceManager` instance. The `TSMServiceManager` then processes the request and returns a response object that contains both the result of the transaction, and response data returned by the subsystem that handled the request.

Generate Voice Mail Password Transaction

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.`**`GenPasswdRsp`** – Response returned by generate password requests
- `com.edocs.common.tsm.sm.core.ITSMServiceManager.newChangeVmPassGenerateRequest` – Create a service request instance using a provided service agreement representing a generate password request.

Example

```
00 String pNumber = …;
01 ITSMDataManager tdm =
02    TSMDataManager.getTSMDataManager(s);
03 IServiceAgreement sa = tdm
getServiceAgreement(phoneNmbr);
04 ISession s = SessionUtils.getUserSession(request);
05 ISMCredentials cred = new
TSMCredentials(s.getUser());
06  ITSMServiceManager tsm =
07
TSMServiceManager.getTSMServiceManager();
09  ISMServiceRequest srx =
10              tsm.newChangeVmPassGenerateRequest(sa);
11 try {
12     ISMResponse rsp = tsm.processServiceRequest(cred,
srx);
13     if (rsp.isSuccessful()) {
14         GenPasswdRsp data = (GenPasswdRsp)
rsp.getData();
15         String sPass = data.getGeneratedPassword();
16     }
17 catch (SMServiceNotSupportedException e) {. . .}
18 catch (SMException e) { . . . }
19 catch (SMAuthorizationException e) {. . .}
20 catch (SMInvalidArgumentException e) {. . .}
```

Code Examination

Line 09 & 19: Use the provided method to create a service request, Line 11: Execute the transaction. Lines 13 and 14, interrogate the response and extract the generated password.

Changing Voice Mail Password Transaction

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.` **`newChangeVoicemailPasswordRequest`** –Create a service request instance representing a change password request.

Example

```
00 ISession s = SessionUtils.getUserSession(request);
01 ISMCredentials cred = new
TSMCredentials(s.getUser());
02
03 ITSMServiceManager tsm =
04    TSMServiceManager.getTSMServiceManager();
05 String pNumber = …;
06 String newPwd = …;
07 ISMServiceRequest srx =
08    tsm.newChangeVoicemailPasswordRequest(pNumber,
newPwd);
09
10 ISMResponse rsp = null;
11 try {
12      rsp = tsm.processServiceRequest(cred, srx);
13     if( !rsp.isSuccessful()) { error code }
14 } catch (. . .)
```

Code Examination

Line 01: Create a credentials instance that will be used by the service manager to validate if the current user has rights to perform a specific operation.

Lines 03 & 04: Using the factory method obtain an instance of a `TSMServerManager`.

Lines 07 & 08: Using the Self Service Manager instance create a transaction request. Note that each transaction request is specific to the actual transaction being executed. In the case of a Change Voicemail Password request we provide the specific phone number and new password when the request is created. In other cases additional code may be required to populate the underlying requests context.

Line 12: Execute the request, returning an instance of a service manager response object. Service manager response objects contain a variety of information about the transaction request, including its transaction id and success or failure status.

Line 13: Did the transaction complete successfully?

Change Phone Number Transaction

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.newChangePhoneNumberRequest`–Create a service request instance representing a change phone number request.

Example

```
00 ICustomer subscriber =
01    CAMClassFactory.getCustomer(request);
02 ICustomerAccount account =
subscriber.getCurrentAccount();
03 String phoneNmbr = account.getAccountNumber();
04 ITSMDataManager tdm =
05    TSMDataManager.getTSMDataManager(s);
06 IServiceAgreement sa = tdm
getServiceAgreement(phoneNmbr);
07
08 ISession s = SessionUtils.getUserSession(request);
09 ISMCredentials cred = new
TSMCredentials(s.getUser());
10 ITSMServiceManager tsm =
11     TSMServiceManager.getTSMServiceManager();
12 String newNumber = …; // some phone #
13 ISMServiceRequest srx =
14    tsm. newChangePhoneNumberRequest (sa, newNumber);
15 ISMResponse rsp = null;
16 try {
17      rsp = tsm.processServiceRequest(cred, srx);
18     if( !rsp.isSuccessful()) { error code }
14 } catch (. . .)
```

Code Examination

Lines 01-11: Obtain a service agreement, credentials and `TSMServiceManager` instance. Lines 13 & 14, obtain a service request representing the change phone number request. Lines 17 & 18, execute the request and examine the result.

Note that under normal circumstances a successful change to a service agreement element such as phone number should be propagated back to the database via the `TSMDataManager` database interfaces.

Port Phone Number Transaction

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.newChangePortNumberRequest`–Create a service request instance representing a port phone number request.

Example

```
00 // Obtain service agreement normally
01 IServiceAgreement sa = . . .
02
03 ISession s = SessionUtils.getUserSession(request);
04 ISMCredentials cred = new
TSMCredentials(s.getUser());
05 ITSMServiceManager tsm =
06     TSMServiceManager.getTSMServiceManager();
07 String phoneNumber = …; // phone number to port
08 ISMServiceRequest srx =
09    tsm. newPortPhoneNumberRequest(sa, phoneNumber);
10 ISMResponse rsp = null;
11 try {
12      rsp = tsm.processServiceRequest(cred, srx);
13     if( !rsp.isSuccessful()) { error code }
14 } catch (. . .)
```

Code Examination

Lines 00-02: Obtain a service agreement, credentials and `TSMServiceManager` instance.   Lines 08 & 09, obtain a service request representing the port # request, note that we provide the phone # being ported.. Lines 12 & 13, execute the request and examine the result.

Note that under normal circumstances a successful change to a service agreement element such as port phone # should be propagated back to the database via the TSMDataManager database interfaces.

Change Digital Subscriber Number Transaction

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.newChgDSNRequest`–Create a service request instance representing a change DSN request.

Example

```
00 // Obtain service agreement normally
01 IServiceAgreement sa = . . .
02
03 ISession s = SessionUtils.getUserSession(request);
04 ISMCredentials cred = new
TSMCredentials(s.getUser());
05 ITSMServiceManager tsm =
06     TSMServiceManager.getTSMServiceManager();
07 String newDSN = …; // some dsn likely obtained from a
form
08 ISMServiceRequest srx =
09    tsm. newChgDSNRequest(sa, newDSN);
10 ISMResponse rsp = null;
11 try {
12      rsp = tsm.processServiceRequest(cred, srx);
13     if( !rsp.isSuccessful()) { error code }
14 } catch (. . .)
```

Code Examination

Lines 00-02: Obtain a service agreement, credentials and `TSMServiceManager` instance. Lines 08 & 09, obtain a service request representing the change DSN request. Lines 12 & 13, execute the request and examine the result.

Note that under normal circumstances a successful change to a service agreement element such as DSN should be propagated back to the database via the `TSMDataManager` database interfaces.

Activate/Deactivate/Suspend/Resume Service Transactions

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.newActivateServiceRequest`– Create a service request instance representing an activate request.
- `com.edocs.common.tsm.sm.core.newDeactivateServiceRequest`– Create a service request instance representing a deactivate request.
- `com.edocs.common.tsm.sm.core.newSuspendServiceRequest`– Create a service request instance representing a suspend request.
- `com.edocs.common.tsm.sm.core.newResumeServiceRequest`– Create a service request instance representing a resume request.

Example

```
00 // Obtain service agreement normally
01 IServiceAgreement sa = . . .
02
03 ISession s = SessionUtils.getUserSession(request);
04 ISMCredentials cred = new
TSMCredentials(s.getUser());
05 ITSMServiceManager tsm =
06     TSMServiceManager.getTSMServiceManager();
08 ISMServiceRequest srx =
09    tsm.newActivateServiceRequest(sa);

08 ISMServiceRequest srx =
09    tsm.newDeactivateServiceRequest(sa);

08 ISMServiceRequest srx =
09    tsm.newSuspendRequest(sa);

08 ISMServiceRequest srx =
09    tsm.newResumeRequest(sa);
10 ISMResponse rsp = null;
11 try {
12      rsp = tsm.processServiceRequest(cred, srx);
13     if( !rsp.isSuccessful()) { error code }
14 } catch (. . .)
```

Code Examination

Lines 00-02: Obtain a service agreement, credentials and `TSMServiceManager` instance. Lines 08 & 09, obtain a service request representing the request. Each suspend/resume/activate/deactivate action works in the exact same way. Lines 12 & 13, execute the request and examine the result.

Note that under normal circumstances a successful change to a service agreement element such as DSN should be propagated back to the database via the `TSMDataManager` database interfaces.

Add Delete Feature Transaction

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.` **`newAddDeleteFeatureRequest`** – Create a service request instance representing a “change feature request”.

Example

```
00 ICustomer subscriber =
01    CAMClassFactory.getCustomer(request);
02 ICustomerAccount account =
subscriber.getCurrentAccount();
03 String phoneNmbr = account.getAccountNumber();
04 ITSMDataManager tdm =
05    TSMDataManager.getTSMDataManager(s);
06 IServiceAgreement sa = tdm
getServiceAgreement(phoneNmbr);
07
08 // All the features in the rate plan
09 Collection  ratePlanFeatures=sa.getRpFeatures();
10
11 // All the current features
12 Collection optionalFeatures = new Vector();
13 Iterator iter=rtPlanCol.iterator();
14 while(iter.hasNext()){
15   IRatePlanFeatureInstance ratePlanFeature=
16      (IRatePlanFeatureInstance)iter.next();
17   if(!ratePlanFeature.isIncluded() ||
18     !ratePlanFeature.isActivated()){
19       ratePlanFeature.setActivated(true);
20       optionalFeatures.add(ratePlanFeature);
21    }
22 }
23 ISession s = SessionUtils.getUserSession(request);
24 ISMCredentials cred = new
TSMCredentials(s.getUser());
25 ITSMServiceManager tsm =
26     TSMServiceManager.getTSMServiceManager();
27 ISMServiceRequest tx =
tsm.newAddDeleteFeatureRequest(sa,
28
optionalFeatures );
29 try {
30     ISMResponse rsp = tsm.processServiceRequest(cred,
srx);
31    if( !rsp.isSuccessful()) { error code }
32 } catch (. . .)
```

Code Examination

The bulk of the example code for the add delete features transaction actually has very little to do with the transaction itself, but deals primarily with obtaining a service agreement and examining its features to determine which are included and active.

Lines 06 & 09: Obtain the service agreement associated with a given phone number (Line 06) from the Self Service Manager database and from the service agreement obtain the current feature set (Line 09).

Lines 12-20: From the retrieved set of rate plan elements, create a new set of rate plan elements activating all listed features.

Lines 27-28: Create a new service request as an `AddDeleteFeatures` request for the provided service agreement, adding the set of optional features.

Line 30: Send the service request to the `TSMServiceManager` for processing by the underlying OSS.

Change Rate Plan Request

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.`**`newChangeRatePlatRequest`** – Create a service request instance representing a "change rate plan request".
- `com.edocs.domain.telco.util.`**`NameSelectionPair`**– Instance, normally used as an array, of a feature name, active flag for turning on and off optional features in a rate plan.

Example

```
00 ICustomer subscriber =
01    CAMClassFactory.getCustomer(request);
02 ICustomerAccount account =
subscriber.getCurrentAccount();
03 String phoneNmbr = account.getAccountNumber();
04 ITSMDataManager tdm =
05    TSMDataManager.getTSMDataManager(s);
06 IServiceAgreement sa = tdm
getServiceAgreement(phoneNmbr);
07
08 IRatePlan rp=tdm.getRatePlan("SomeRatePlan");
09 int oCount=rp.getOptionalFeatures().length;
10 NameSelectionPair[] oFeatures=
11                    new NameSelectionPair[oCount];
12 Iterator i = rp.getFeatures().iterator();
13 for (int which =0;i.hasNext(); which++) {
14   IRatePlanFeature rpf =(IRatePlanFeature)i.next();
15   if(!rpf.isIncluded().booleanValue()){
16   oFeatures[which] =
17          new NameSelectionPair (rpf.getName(),
18                                 new Boolean(true));
19 }
20 ISession s = SessionUtils.getUserSession(request);
21 ISMCredentials cred =
            new TSMCredentials(s.getUser());
22 ITSMServiceManager tsm =
23     TSMServiceManager.getTSMServiceManager();
24 ISMServiceRequest tx =
25        tsm.newChangeRatePlanRequest(sa,rp,oFeatures);
26 try {
30     ISMResponse rsp = tsm.processServiceRequest(cred,
31                                                 srx);
32    if( !rsp.isSuccessful()) { error code }
33 } catch (. . .)
```

Code Examination

The bulk of the example code for the change rate plan s transaction actually has very little to do with the transaction itself, but deals primarily with obtaining a service agreement and creating a set of optional features which should be enabled for the given plan instance. .

Lines 08 -19: Obtain an instance of a named rate plan(08), and from that rate plan obtain the set of optional features. From the set of optional features, create a populated `NameSelectionPair` containing each active feature. In the example shown all features are active (line 18). The named selection pair array is then used on line 24-25 when the change rate plan service request is created. Line 30 then executes the request.

Find Subscriber Profile/Change Subscriber Profile Transactions

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.` **`newChangeSubscriberProfileRequest`** –Create a service request instance representing a “change subscriber profile request”.
- `com.edocs.common.tsm.sm.core.`**`ISubscriberProfile`**– Interface definition for subscriber profiles.

Common Code

```
00 ICustomer subscriber =
01    CAMClassFactory.getCustomer(request);
02 ICustomerAccount account =
subscriber.getCurrentAccount();
03 String phoneNmbr = account.getAccountNumber();
04 ITSMDataManager tdm =
05    TSMDataManager.getTSMDataManager(s);
06 IServiceAgreement sa = tdm
getServiceAgreement(phoneNmbr);
07
08 ISession s = SessionUtils.getUserSession(request);
09 ISMCredentials cred = new
TSMCredentials(s.getUser());
10 ITSMServiceManager tsm =
11     TSMServiceManager.getTSMServiceManager();
```

Find a Subscriber Profile

```
12
13 ISubscriberProfile sp =
tsm.findSubscriberProfile(cred,sa);
14 if (sp == null) { error, no profile found }
```

Modify Subscriber Profile

```
15 sp.setFirstName(“Chad”);
16 sp.setLastName(“Bigglesworth”);
17
18 ISMServiceRequest tx =
19               tsm.newChangeSubscriberProfileRequest(sa,
20                                                    prof
);
21 try {
22     ISMResponse rsp = tsm.processServiceRequest(cred,
srx);
23    if( !rsp.isSuccessful()) { error code }
324 } catch (. . .)
```

Code Examination

The bulk of the example code for the find/change subscriber profile has to do with obtaining the initial service agreement. Lines 01-06 obtain an instance of a service agreement. Lines 08-11 obtain a set of user credentials and an instance of a `TSMServiceManager`. Lines 13 and 14 use the service agreement and credentials to return the associated profile or `null` if no profile is found.

Lines 15 and 16 make some change to the profile, see the `ISubscriberProfile` javadoc for a complete listing of all subscriber profile mutator methods.

Lines 18 and 19 obtain a service request representing a change profile and line 22 executes the request.

Obtain Available area codes/exchanges/phone number Transactions

Key classes, interfaces and methods:

- `com.edocs.common.tsm.sm.core.ITSMServiceManager.` **`getAvailAreaCodes`** – Return a list of available area codes for a given zip code.
- `com.edocs.common.tsm.sm.core.ITSMServiceManager.` **`getAvailExchanges`** – Return a list of available exchanges for a given area code
- `com.edocs.common.tsm.sm.core.ITSMServiceManager.` **`getAvailPhoneNumbers`** – Return a list of available phone numbers for a given area code and exchange

Common Code

```
01 ISession s = SessionUtils.getUserSession(request);
02 ISMCredentials cred = new
TSMCredentials(s.getUser());
03  ITSMServiceManager tsm =
04
TSMServiceManager.getTSMServiceManager();
```

Obtain Area Codes

```
05  String[]ac = tsm.getAvailAreaCodes(cred,”01760”);
06  if (!ac || ac.length == 0) { error }
```

Obtain Available Exchanges

```
07  String[]ex = tsm.getAvailExchanges(cred,ac[n]);
08  if (!ex || ex.length == 0) { error }
```

Obtain Available Phone Numbers

```
09  String[]pn = tsm.getAvailExchanges(cred,ac[n],
ex[m],5);
10  if (!pn || pn.length == 0) { error }
```

Code Examination

Line 01-04: Obtain credentials and an instance of a `TSMServiceManager`.
Lines 05 and 06, for a given zip code, return the current list of area codes.
Line 07 and 08, for a given area code, return the list of available exchanges.
Lines 09-10, for a given area code, exchange in area code, return specified count of phone numbers.

Bulk Transaction Handling

Bulk transactions are individual service requests compiled together and then submitted as a group. The process for working with bulk transactions is similar to the process for handling individual transaction. The example below shows the common steps for working with a bulk transactions.

```
00 String[] phones = ..
01 String[] pwds = ...
02 ISMServiceRequest bsr = sm.newBulkRequest();
03 for ( int i = 0; i< phones.length; i++) {
04    ISMServiceRequest sr=
05
sm.newChangeVoicemailPasswordRequest(phones[i],
06
pwds[i]);
07     bsr.addBulkRequestElement(sr);
08 }
09 ISMResponse rsp = sm.processServiceRequest(cred,
srx);
10
11
```

Code Examination

Line 02 – Create a bulk transaction request, as opposed to a individual transaction request. Lines 04-06 - For each component of the bulk transaction, create an individual request. Line 07 – Add the individual element to the bulk service request. Line 09 – Process the result normally.

Bulk Transaction Unrolling

Bulk transactions are conceptually a set of individual transactions grouped together for ease of management. From the perspective of transaction processing, each bulk transaction in "unrolled" into individual transactions. Each individual transaction is then passed to the underlying transport for processing. All other processing, for example asynchronous response handling, is then managed as if the transaction as individual rather then a member of a bulk transaction.

Bulk Requests supported in Self Service Manager 5.0.0

Request	Description & Notes
Change Voice Mail Password	No special processing required. Obtain a service request and register with the `ISMServiceRequest` instance normally.
Generate Voice Mail Password	No special processing required. Obtain a service request and register with the `ISMServiceRequest` instance normally.
Add Delete Feature Request	No special processing required. Obtain a service request and register with the `ISMServiceRequest` instance normally.

Transaction Response Handling

The Self Service Manager reports transaction status via `ISMResponse` objects. `ISMResponse.isSuccessful()` method returns `true` or `false` indicating if a transaction was successful. In addition `ISMResponse.getResponseCode()` returns a string representation of a response code. Possible return values are:

- `ISMResponse.RSP_CODE_ACK` – The transaction is in a pending state.
- `ISMResponse.RSP_CODE_SUCCESS` – Transaction completed successfully.
- `ISMResponse.RSP_CODE_FAILURE` – Transaction failed.
- `ISMResponse.RSP_CODE_SYSUNAVAIL` – Transaction failed, underlying OSS not available (Reserved).
- `ISMResponse.RSP_CODE_NEED_CUSTOMER_INFO` – Transaction failed, in sufficient customer data(Reserved).

In the event of a transaction failure use `Collection ISMResponse.getErrorMsgs()` & `Collection ISMResponse.getInformationalMsgs()` to determine the exact nature of the problem.

Common Exceptions

The Self Service Manager transaction methods may throw exceptions from the package: `com.edocs.common.sm.api.exception.*`.

Exceptions fall into these specific areas:

- `SMException` – Generic base exception thrown by the Transaction APIs
- `SMServiceNotSupportedException` – No destination found for named transaction.
- `SMAuthorizationException` – User does not have the rights to perform requested operation.
- `SMInvalidArgumentException` – In sufficient or invalid arguments.

For a complete listing of all arguments, methods and exceptions for the Self Service Manager Search and Update APIs see the Javadoc for the package *`com.edocs.common.sm.api.exception.*`*.

Enterprise Systems Connectivity

Concepts

The Enterprise Systems Connectivity layer of Self Service Manager provides the required infrastructure to connect Self Service Manager with external service providers. Conceptually Self Service Manager managers are composed of a number of component parts whose relationships are between defined in *`sm.xma.xml`*:

- Transactions also know as *Actions* – define the set of operations that can be executed by applications. Actions are mapped to a specific *Connector* by a *Service Manager*. Self Service Manager comes with a core set of actions and can be extended with developer written custom actions.
- Connectors – are a combination of inbound and outbound *transforms* as well as the *transport* class that performs the work of connecting to a specific back end OSS. Connectors represent a set of services managed by a specific OSS.
- Transports – implement the required functionality to interconnect Self Service Manager with a back end OSS and handle a well-defined set of actions.
- Transforms – implement the required functionality to convert a message from one format to another. *Outbound* transforms convert messages before they are passed to the underlying Connector. *Inbound* transforms responses from the underlying OSS into a format appropriate for use by Self Service Manager.

- Service Managers – define the interrelationship between actions and the connectors that handle them. Each Service Manager has an associated set of *Response Handlers* that manage the processing of different return result codes.
- Response Handlers – are mapped to specific Connector responses and can perform custom processing as required by the underlying action. A default set of no-op response handlers is provided by Self Service Manager.

Defining Destinations

The combination of transport, inbound, and outbound translations are referred to as a *Destination* and defined within `sm.xma.xml`. Out-of-the-box Self Service Manager has defined support for a single destination that handles core transactions as dummy code. Demonstration Transport, Inbound and outbound classes are provided in the package `com.edocs.common.tsm.sm.core.demo`. At a minimum client must develop a transport supporting the core transaction set.

Example Transport

Transports must implement `com.edocs.common.eai.api.core.IETransport` and implementing public `IEAIResponse sendMessage(…)` as shown below. A typical implementation of a send message includes a switch statement for each action handled by the transport class. Actions, defined later in this document, are identified by a unique integer that is the basis for the switch statement. *You MUST implement a custom replacement for the demo transport.*

```
import java.io.Serializable;
import com.edocs.common.eai.api.core.*;
import com.edocs.common.eai.api.exception.*;
import com.edocs.common.tsm.sm.core.ChangeVMPasswordRsp;

public class DEMOTransport extends IETransport {

// mimics the definitions in sm.xma.xml
    private static final int ACTION_TYPE_CHGVMPASS =
100;

    public String getName() {
        return "Demo Transport";
    }

    public IEAIResponse sendMessage(Long requestId,
                                    IAction action,
                                    Serializable data)
           throws ServerLookupException,
TransportException {
        IEAIResponse rsp = null;
        Serializable rspData = null;
        switch (action.getMessageType()) {
            case ACTION_TYPE_CHGVMPASS : {
                ChangeVMPasswordRsp payload =
                        new ChangeVMPasswordRsp();
                EAIResponse rsp = new
EAIResponse(requestId

payload);
                rsp.setConfirmationNumber(
                            System.currentTimeMillis());

rsp.setResponseCode(rsp.RSP_CODE_SUCCESS);
                break;
                }
        } // end switch
        return rsp;
    }
}
```

Understanding Response Codes:

For a given request, the underlying transport must return one of a set of well defined response codes: The current response code set and their meanings are:

Supported Response Codes

Response Code	Meaning
`IEAResponse.RSP_CODE_SUCCESS`	Call to the back end OSS/BSS succeeded.
`IEAResponse.RSP_CODE_FAILURE`	Call to the back end OSS/BSS failed. Add one or more error messages.
`IEAResponse.RSP_CODE_SYSUNAVAIL`	Call to the back end OSS/BSS failed due to connectivity issues. Will be retried. Add one or more error messages.
`IEAResponse.RSP_CODE_ACK`	Call will be completed in the background. Transport must start a thread to handle the response which calls the registered called back handler when completed.

`sendMessage` syntax

```
public IEAIResponse sendMessage(Long requestId,
                                IAction action,
                                Serializable data)
```

Where:
`long requestID` – unique transaction identifier provided by the ESC Subsystem.
`IAction action` – the action be performed. See the Javadoc for a definition of the `IAction` interface. `IAction.getMessageType()` returns the action identifier for the current action.
`Serializable data` – the developer defined payload or contents of the action.

Returns:
`IEAIResponse` – Normally a derived class implementing or extending `IEAIResponse` defining response content particular to the action being processed.

Adding messages to a response request

Response instances can transport to the originating caller a set of messages. Messages can be either errors or informational and are added using code similar to that shown below.

```
EAIResponse rsp = . . . ;

// Obtain the message container
IEIResponseMessages msgs = rsp.getResponseMessages();
// Add error or informational messages
msgs.newErrorMsg("Something bad happened", serializable
data);
// insert the updated message collection back into the
to be
// returned response
rsp.setResponseMessages(msgs);
```

The ultimate recipient of the response could then obtain the send messages using code similar to:

```
IEIResponseMessages msgs = rsp.getResponseMessages();
if (msgs.hadErrorMsgs() ) {
   Iterator it = msgs.getErrorMsgs().iterator();
   while(it.hasNext) {
         IEIResponseMessage msg = it.next():
         String msgString = msg.getMessageCode();
         Serializable[]data = msg.getMessageData();
}
```

See the Self Service Manager javadoc for a complete list of all classes and methods pertaining to response messages.

Example Transform

Transforms can be either *inbound* or *outbound* and provide a mechanism for processing a message payload before it is sent or after it has been processed, , generally to convert the data from the Siebel object model to the Customer's object model. Outbound transforms are called before the transport send message method is invoked and can modify or transform data as required. Inbound transforms are called after the transform send message method is invoked and transform responses as required.

Both inbound and outbound transforms are optional.

Transforms must extend `com.edocs.common.eai.api.core.`IEITranslator and implement public `Serializable translate()` as shown below. A typical implementation of `translate` includes a switch statement for each action handled by the translator class.

```
import java.io.Serializable;
import com.edocs.common.eai.api.core.IEIAction;
import com.edocs.common.eai.api.core.IEITranslator;
import
com.edocs.common.eai.api.exception.TranslationException;
import com.edocs.common.tsm.sm.core.ChangeVMPasswordRsp;

public class InboundXForm implements IEITranslator {
    // mimics the definitions in sm.xma.xml
    private static final int ACTION_TYPE_CHGVMPASS =
100;
    public Serializable translate(IEIAction action,
                                  Serializable msg)

                         throws TranslationException {
        Serializable rspData = null;
        switch (action.getMessageType()) {
            case ACTION_TYPE_CHGVMPASS : {
               ChangeVMPasswordRsp rsp =
                              (ChangeVMPasswordRsp) msg;
               rspData = rsp;
               break;
            }
        }
        return rspData;
     }

    public Serializable translate(IEIAction action,
                                  Serializable[] msgs)
                                  throws
TranslationException
    { return null; }
}
```

Example `translate` syntax

```
public Serializable translate(IAction action,
                              Serializable msg)
```

Where:

`IAction action` – the action be performed. See the Javadoc for a definition of the `IAction` interface. `IAction.getMessageType`() returns the action identifier for the current action.

`Serializable msg` – the developer defined payload or contents of the action.

Returns:

`Serializabe` – Transformed content.

Defining a `Destination` in `sm.xma.xml`

Self Service Manager uses the inversion of control concepts of the Spring framework to externalize the class dependencies representing the core Self Service Manager systems in terms of bean references. Each bean reference defines the set of properties and other beans that a given object uses or references. For more information on spring and the spring framework see www.springframework.org.

The Spring Framework defines relationships in terms of java beans which can be used a component parts of other beans. At a minimum spring defines a parent (uses) child (is used by) relation \ship as:

```
<bean id="child.bean"
          class="com.myco.mypackage.child"/>
<bean id="parent" class="com.myco.mypackage.parent">
    <property name="child">
        <ref bean="child.bean"/>
    </property>
</bean>
```

Spring defines an expected set of bean methods that in turn are used to maintain the class relationships external to the classes themselves.

Destinations are defined in `sm.xma.xml` with three bean elements representing the inbound, outbound and transport components. The Destination bean, defined in terms of `com.edocs.common.eai.core.Destination`, then specifies these three beans as properties.

Properties of `com.edocs.common.eai.core.Destination:`

Property	Value
`inboundTranslator`	Bean ID of a bean implementing `com.edocs.common.eai.api.core.IEITranslator` Can be omitted.
`outboundTranslator`	Bean ID of a bean implementing `com.edocs.common.eai.api.core.IEITranslator.` Can be omitted.
`transport`	Bean ID of a bean implementing `com.edocs.common.eai.api.core.ITransport`
`timeout`	Timeout value in seconds for `sendMessage` & `translate` method calls.

`Destination` Bean Example

```
<beans>
. . .
    <bean id="DemoTSMOutboundTrn.Java"

class="com.edocs.common.tsm.sm.core.demo.OutboundXForm"/>
     <bean id="DemoTSMInboundTrn.Java"

class="com.edocs.common.tsm.sm.core.demo.InboundXForm"/>
    <bean id="DemoTSMTransport"
      class="com.edocs.common.tsm.sm.core.Demo.Transport"/>
    <bean id="DemoTSMDestination"
          class="com.edocs.common.eai.core.Destination">
        <property name="outboundTranslator">
            <ref bean="DemoTSMOutboundTrn.Java"/>
        </property>
        <property name="inboundTranslator">
            <ref bean="DemoTSMInboundTrn.Java"/>
        </property>
        <property name="transport">
            <ref bean="DemoTSMTransport"/>
        </property>
         <property name="timeout">
            <value>2</value>
        </property>
    </bean>

. . .
```

Defining Transactions

Transactions, or *actions* are they are called within the Self Service Manager framework; define the operations which client applications can perform on Service Agreements via an underlying OSS. At run time, an action is executed by a client application, which in turn is routed via the Self Service Manager subsystems to the underlying connector that performs the required work. The connector then returns back an appropriate response.

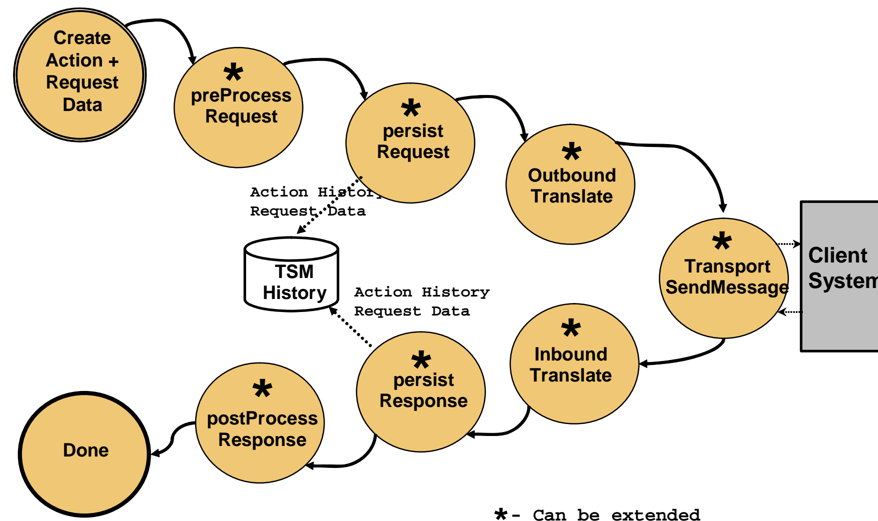
Actions are composed of:

- A set of request/response objects defining the data into and returned from the underlying OSS when the action is performed.
- An extension class of `TSMServiceManager` that implements the code to fabricate `ISMServiceRequest` instances specific to the new requests.
- A snippet of XML defining the action and associated the specified request and response classes with that action
- DBMS statements to register the action in the Self Service Manager Database tables.

- An optional set of response handlers.

Actions can be either Synchronous or Asynchronous and go through a defined set of lifecycle states. The following diagrams detail the lifecycle states of an action.

Synchronous Action Lifecycle



The lifecycle of a Self Service Manager transaction is designed to be extendable and robust. Synchronous lifecycle states are:

1. Create Action – The client application creates a specific action and its associated request data and passes the action to the underlying Self Service Manager Transaction subsystem.

2. Persist Request – The specifics of the action request are persisted to the Self Service Manager history database for both audit and user review purposes. Persist request is an extendable service.

3. Pre-Process Request – The outbound request is then pre-processed. Default pre-processing is a noop however, pre-processing is extendable as required.

4. Outbound Translate – The outbound request can be transformed or translated to another format required buy the underlying connector as required. Default translate is a noop.

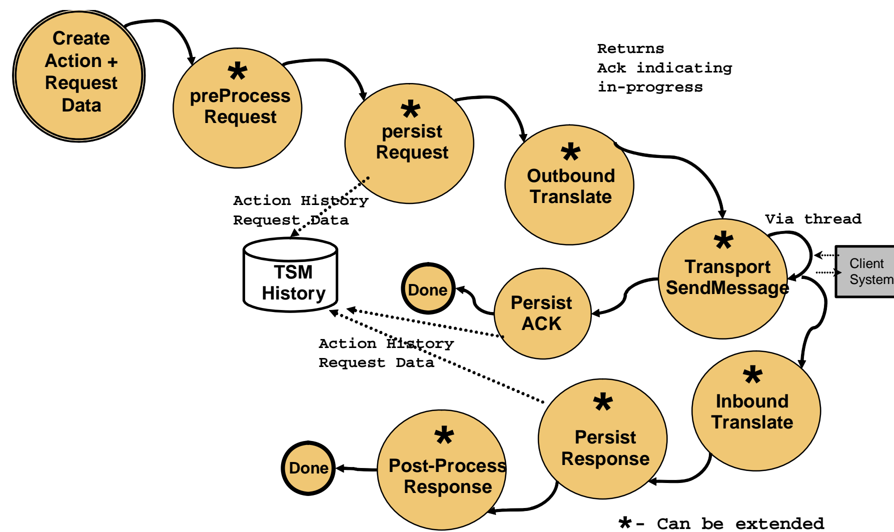
5. Connector `SendMessage` – The connector is then invoked to send the message to the back end client system which performs the requested action and returns a response in “real time”. Note that regardless of success or failure the result of the request is sent back down stream.

6. Inbound Translate – The Inbound response can be transformed or translated to another format required buy the underlying connector as required. Default translate is a noop.

7. Persist Response – The specifics of the results of the action are persisted to the Self Service Manager history database for both audit and user review purposes. Persist response is an extendable service.

8. Post Process Response – The inbound result can then be post-processed as required. Default post-processing is a no-op however, post-processing is extendable as required.

Asynchronous Action Lifecycle



The Asynchronous lifecycle states are shown below: Note the create action to Outbound Translate states are identical to the synchronous case.

1. Connector `SendMessage` – Starts a developer written thread to process the send message action. Returns an asynchronous acknowledgement to indicate the action is in progress.

2. Persist ACK – The returned acknowledgement is persisted to the Self Service Manager history database as action pending.

3. Inbound Translate – The Inbound response can be transformed or translated to another format required buy the underlying connector as required. Default translate is a noop.

4. Persist Response – The specifics of the results of the action are persisted to the Self Service Manager history database for both audit and user review purposes. Persist response is an extendable service.

5. Post Process Response – The inbound result can then be post-processed as required. Default post-processing is a no-op however, post-processing is extendable as required.

General Transaction Characteristics

Transactions are defined within Self Service Manager in two parts, within an XML configuration file and within the Self Service Manager database. Defined transaction request data is exposed to Self Service Manager via a Custom Request class derived from `TelNoReq`. Custom response classes, derived from `Serializable`, provide the returned result. And custom methods, added by extending, `TSMServiceManager`, provide the factories required to tie the areas together.

Actions are defined within Self Service Manager via entries in `sm.xma.xml`. Each action has one attribute and 3 required properties. Actions may also add a map of optional response handlers.

Properties of `com.edocs.common.sm.core.SMAction:`

Property	Value
`id`	Any unique name starting with Action. The id can be anything but is normally the name of the action less spaces. For example, Set Special name might be `setSpecialName`.
`class`	Bean class: must be or extend `com.edocs.application.tsm.sm.core.TSMAction.`
`messageType`	A unique integer identifier, values less then 1000 are reserved for Seibel actions.
`externalName`	External name matching the id.
`transient`	Action is transient if true. Transient operations are NOT persisted to the Self Service Manager database and cannot be tracked for the purposes of history. Default `false`.

Defining a `Action` in `sm.xma.xml`

Self Service Manager Transactions are named combinations of actions, action input data and action response data. To define a new action:

1. Select an unused message value and name for your action and registration
2. Create request and response classes,
3. Define the action and associate with it its underlying request and response classes.

The example below, defined in `sm.xma.xml`, defines an action called Set Special Name. Request and response objects referenced are coded below.

```
<beans>
. . .
    <bean id="ReqData.setSpecialName"
          class=" com.myco.demo.setSpecialNameReq"/>
    <bean id="RspData.setSpecialName "
          class=" com.myco.demo setSpecialNameRsp"/>
    <bean id="Action.setSpecialName"
         class="com.edocs.common.sm.core.TSMAction" >
        <property name="messageType">
                       <value>1000</value></property>
        <property name="externalName">
             <value>setSpecialName</value></property>
    </bean>
. . .
```

Defining a `Action` in `Self Service Manager Database`

Actions must also be registered in the Self Service Manager database. The snippet of SQL below adds the required records to the Billing Manager database tables in support of a new action 1000, a unique integer not previously used for any action.

```
insert into CMSCASE_TYPE_REF
          (case_type_key, 'descrip', is_active,
key_code)
          values
          (1000, 'Set Special Name', 1, setSpecialName);
```

If multiple languages are supported, messages can be entered into the database as shown below. Each element differing by locale and translated text:

```
insert into AMSI18N_TRN_REF
       (document_type_key, document_key, locale,
field_name,

translated_text)
       values
      (15003, 1000, 'es_US', 'Description',
                                'spanish: Set Special
Name');
```

Sample Request

Request objects are the data associated with a Self Service Manager transaction. All Self Service Manager Transactions are combinations of an underlying action and the data that action works on. The first step in developing a custom Self Service Manager transaction is creating the request class.

In the following example we create an action that adds a “special name” to a service agreement. The special name is no more then a string passed to the underlying Destination. Since all requests pertaining to Self Service Manager are by definition communications requests, and each must extend `TelNoReq` and include any special data required by the underlying OSS connector to process the request.

```
package com.myco.demo;
import java.io.Serializable;
import com.edocs.application.tsm.sm.data.*;
public class setSpecialNameReq extends TelNoReq
                               implements Serializable {
    String specialName= "";
    public void setSpecialName(String specialName) {
        this.specialName = specialName;
    }
    public setSpecialNameReq (IServiceAgreement sa,
                               String specialName) {
      this.specialName = specialName;
      this.setServiceAgreement(sa);
    }
}
```

Sample Response

Response objects are simple Java classes that are generated by the underlying OCC connector and can return special status or other content to the calling action. A sample response is shown below:

```
package com.myco.demo;
import java.io.Serializable;
public class setSpecialNameRsp implements Serializable {
    public setSpecialNameRsp () { }
}
```

Response objects can be combined with OSS Connectors to provide a query/response mechanism that does not actually change a service agreement but rather returns an answer to a question. Such a transaction is normally configured as transient and takes a query style payload rather then the contents of an update. The connector associated with the transaction then takes the question, performs the query, and returns the response.

Sample `TSMServiceManager` extension classes

`TSMServiceManager` extended classes are the glue that tie actions to request data. `ISMServiceRequest` objects package actions together with their required data. TSMServiceManager is, amongst other things, a service request factory. New service requests are created by extending this class and adding additional methods supporting custom transactions. The example below shows an example of such a class which defines a new method, `newSetSpecialNameRequest`, which returns an instance of `ISMServiceRequest`.

```
00 package com.myco.demo;
01 import org.springframework.context.ApplicationContext;
02 import com.edocs.common.sm.api.*;
03 import com.edocs.common.sm.api.exception.*;
04 import com.edocs.common.tsm.sm.core.*;
05 public class CustomServiceManager extends
TSMServiceManager{
06    public ISMServiceRequest newSetSpecialNameRequest(
07
IServiceAgreement sa,
08                                    String sn) {
09        ApplicationContext ctx = this.getTheContext();
10        ISMServiceRequest srx = null;
11        IEIAction action = (ISMAction)
12
ctx.getBean("Action.setSpecialName")
13         setSpecialNameReq data =
14                     new setSpecialNameReq(sa,sn);
15        srx = newServiceRequest(action, data);
16        return srx;
17    }
18 }
```

Code Examination

Self Service Manager transaction code centers around `ISMServiceRequest` instances. `ISMServiceRequest` instances contain both the data and the action to perform on that data. The following paragraphs detail the development of sample code shown above.

All custom Self Service Manager classes must extend `TSMServiceManager` (line 05). Service Requests are based on methods returning instances of classes implementing `ISMServiceRequest`(line 06). Service Actions are returned via the Spring framework using a context, line 09, and a getBean operation on that context (line 10-11). An instance of the request data(lines 12-13) is then passed to the `newServiceRequest()` inherited method to create the required `ISMServiceRequest` instance.

The new transaction can then be used as if it was part of the out-of-the-box Self Service Manager transaction set as shown below:

```
ISMServiceRequest tx = tsm.
newAddSpecialNameRequest(sa,"name" );
try {
    ISMResponse rsp = tsm.processServiceRequest(cred, srx);
    if( !rsp.isSuccessful()) { error code }
    }
 . . .
```

Transaction Lifecycle Events

Self Service Manager supports the ability to integrate into the lifecycle of Self Service Manager Actions.
As previously shown, actions have four lifecycle events which can be intercepted.

Lifecycle processing falls into two categories:

- Pre and post process – interjecting custom processing before an action is processed and/or after an action returns from processing
- Persistence Management – custom persistence processing replacing or augmenting existing persistence management.

An example of action pre/post processing of actions might be performing custom success code such as storing a returned password or contents back into the Self Service Manager local database.

An example of action persistence management might be adding custom persistence to an action in support of storing additional log or processing information into a custom database, persisting requests to an XML data for later processing and reporting or otherwise storing custom information.

Pre and post processing:

The `com.edocs.common.sm.api.ISMAction` interface defines three methods, shown below, which can be implemented by the developer to perform custom pre and post processing.

```
public interface ISMAction extends IEIAction {
    public void preProcessRequest(ISMServiceRequest srx)
         throws SMPersistException, SMException;
    public void postProcessResponseSuccess(
                              ISMServiceManager sm,
                              ISMServiceRequest srx)
        throws SMPersistException, SMException;
    public void postProcessResponse(
                              ISMServiceManager sm,
                              ISMServiceRequest srx)
        throws SMPersistException, SMException;
. . .
}
```

Per the synchronous and asynchronous lifecycle diagrams the `preProcessRequest` method is invoked by the SM subsystem immediately after the action is submitted for processing and BEFORE the action is persisted. In this way changes can be made to the data and those changes persisted to backing store along with the remainder of the request. Post processing occurs immediately following the request being persisted as it normally does not involve changing any content or results but rather making changes to the Self Service Manager database.

Post processing is supported via two methods `postProcessResponse` and `postProcessResponseSuccess`, the latter only invoked when actions return a result of `IEAResponse.RSP_CODE_SUCCESS`.

Note that the `TSMAction` extends `SMAction` which implements `ISMAction` and is the base class for all Self Service Manager actions as it adds a host of functionality particular to the communications domain. `TSMAction` should the starting point of an action class used in the context of Self Service Manager.

Adding lifecycle management support

Lifecycle management support is added to actions by implementing the one or more of the Process Request/Response methods shown above. An example action, which interacts with the Telco Data Manager to persist a successful result back to the local database, is shown below.

```
00 package com.myco.demo;
01
02 import com.edocs.application.tsm.sm.core.*;
03 import com.edocs.common.sm.api.*;
04 import com.edocs.common.sm.api.exception.*;
05 import com.edocs.domain.telco.api.tsm.*;
06 import com.edocs.domain.telco.api.tsm.exception.*;
07 import org.apache.commons.logging.*;
08
09 public class setSpecialNameAction extends TSMAction {
10   public static final int ACTION_TYPE_SETSPECIALNAME
11       = 1000;
12   public static final String ACTION_NAME =
13       "setSpecialName";
14   private static log =
15       LogFactory.getLog(setSpecialNameAction.class);
16 public void
17 postProcessResponseSuccess(ISMServiceManager sm,
18                         ISMServiceRequest srx)
19           throws SMPersistException, SMException {
20      try {
21          ISMRequest req = srx.getTheRequest();
22
23          // Obtain an instance of the
24          // Telco Data Manager
25          // Assumming we need to persist some data
26          // back to the db
27          ITSMDataManager tdm =
28                     getTelcoDataManager(sm, srx);
29
30          // Get the original request
31          setSpecialNamReq data =
32              (setSpecialNamReq) req.getData();
33
34
35          // use the TSMAction method
36          // loadServiceAgreement to get the
37          // associated service agreement
38          // assuming we need the service agreement to
39          // persist the data
40          IServiceAgreement sa =
41                loadServiceAgreement(sm, srx, data)
42
43          // extract phone number from the
44          // request data assuming we use the phone
45          //  number for a key somewhere!
45          String newPhoneNumber =
46              data.getNewPhoneNumber();
47
48            // call the data manager to perform
49            // some sort of data operation
50            // such as update the service agreement
51            // with a new phone number etc
52
```

```
53           // tdm.doSomethingtoDB(sa,...)
54           if (log.isDebugEnabled()) {
55               log.debug("Successfully updated db");
56          }
57      } catch (InvalidArgumentException e) {
58        throw new SMException();
59      } catch (DataStoreException e) {
60         throw new SMPersistException();
61      } catch (CommunicationException e) {
62         throw new SMException();
63      }
64  }
65 }
```

Code Examination

Lines 02-06 detail the required imports. Line 08 imports the apache log package uses in lines 13-14 and 54-55. Lines 10-13 define constants matching those in `sm.xma.xml` and defining the actions identify. Lines 17 to 19 define the overloaded post process method.

The remaining code is particular to the actions post processing. The convenience method `getTelcoDataManager` can be used to return a `ITSMDataManager` instance in the event code needs to update the data base with returned a returned result, a common occurrence. Likewise the code uses the `loadServerAgreement` method to obtain the associated service agreement. See the javadocs for `TSMAction` for a complete list of the `TSMAction` inherited methods. Line 53 represents some method to submit the resulting change back to the Self Service Manager database. Use the appropriate method, *rather then the dummy shown,* to persist the result back to the database.

Updated Action Definition

The action definition, found in `sm.xma.xml`, would then be updated as shown below to reflect the extended action class.

```
<beans>
. . .
    <bean id="ReqData.setSpecialName"
          class=" com.myco.demo.setSpecialNameReq"/>
    <bean id="RspData.setSpecialName "
          class=" com.myco.demo setSpecialNameRsp"/>
    <bean id="Action.setSpecialName"
         class="com.myco.demo.setSpecialNameAction " >
        <property name="messageType">
                       <value>1000</value></property>
        <property name="externalName">
              <value>setSpecialName</value></property>
    </bean>
. . .
```

Persistence Management

The `com.edocs.common.sm.api.ISMAction` interface defines two methods, shown below, which can be implemented by the developer to perform custom persistence management.

```
public interface ISMAction extends IEIAction {
   public void persistNewRequest(ISMServiceManager sm,
                                 ISMServiceRequest srx)
        throws SMPersistException;

   public void persistNewResponse(ISMServiceManager sm,
                                  ISMServiceRequest srx)
        throws SMPersistException;
. . .
}
```

Per the synchronous and asynchronous lifecycle diagrams the `persistNewRequest` method is invoked by the SM subsystem after the request has bee preprocessed but before it is translated. Likewise, the `persistNewResponse` method is called immediately following any outbound transformation, but before post processing.

persistNewRequest and persistNewResponse are defined almost identically and typically contain similar processing with the exception of the focus of the processing (before request being processed versus after completion). Two specific use cases are normally managed by these methods, extend the existing persistence, or replace the existing persistence. The example below shows how one might extend the persistence mechanism to a secondary storage mechanism, while maintaining the existing persistence mechanism.

Adding persistence management support

Persistence management support is added to actions by overriding the one or both of the persist methods shown above. An example action, which uses the underlying persistence engine is shown below. *Note that only the additional method is shown not the entire class.*

```
00 package com.myco.demo;
. . .
01 public void persistNewRequest(ISMServiceManager sm,
02                              ISMServiceRequest srx)
03           throws SMPersistException {
04          try {
05              super.persistNewRequest(sm,srx)
06          } catch (SMPersistException sme) {
07              throw sme;
08          }
09
10          // Obtain the original request
11          ISMRequest req = srx.getTheRequest();
12
13          // Cast as appropriate
14          setSpecialNamReq data =
15              (setSpecialNamReq) req.getData();
15          // Do custom persistence
17          // save statistics to a db etc.
18   }
. . .
```

Code Examination

Note that this example is only partial and would normally be part of a larger `TSMAction` extended class. . Line 05 uses the parent method to perform any out of the box persistence, catching and throwing any unexpected exceptions. Lines 11 onward are devoted to extracting the resulting request data and performing any custom actions required.

Custom Request and Response Persistence

Action data is persisted to backing store as XML via Castor. See www.castor.org for a complete description of Castor. Castor provides marshall/unmarshall support for converting Java objects into and out of XML via mappings which may be automatically generated or specified via xml based mapping files.

For many objects Castor mappings are generated automatically and mapped into and out of XML without issue. Occasionally, given very complex object nesting, castor may generate an invalid mapping resulting in run time errors. In such cases castor mappings can be defined externally via `sm.castor.xml`, a portion of which is shown below

Example sm.castor.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
    <class name="com.edocs.common.sm.core.SMResponse"
           auto-complete="false">
        <description>Default mapping for class

com.edocs.common.sm.core.SMResponse</description>
        <map-to xml="SMResponse"/>
        <field name="responseMessages"
               type=

"com.edocs.common.eai.api.core.IEIResponseMessages"
            required="false" direct="false"
transient="false">
            <bind-xml name="response-messages"
                      node="element"
                      reference="false"/>
        </field>
        <field name="confirmationId"
               type="string"
               required="false"
               direct="false" transient="false">
            <bind-xml name="confirmation-id"
                      node="element" reference="false"/>
        </field>
. . .
    </class>
    <class
name="com.edocs.common.eai.core.EIResponseMessage"
           auto-complete="false">
. . .
    </class>
. . .
</mapping>
```

Custom castor mappings can be specified by adding new <class> elements to the default castor mapping file.

Assuming the earlier `setSpecialNameReq` java object we could define a custom castor mapping as:

Custom Castor Mapping for `setSpecialNameReq`

```
<class name="com.myco.demo.setSpecialNameReq"
    auto-complete="false">
    <description>Default mapping for class
        com.myco.demo. setSpecialNameReq </description>
    <map-to xml="setspecialname-req"/>
    <field name="specialName" type="string"
           required="false" direct="false"
           transient="false">
        <bind-xml name="name" node="element"
                  reference="false"/>
    </field>

</class>
```

The mapping would then be added to `sm.castor.xml` and objects matching the class type would be marshaled and unmarshalled from xml using the providing mapping. The benefits of castor go beyond the simple example above and mappings can be defined in terms of complex object hierarchies including arrays, collects, sets etc. See the castor documentation for a complete description of castor mappings.

Supporting Asynchronous requests

The ESC subsystem assumes that transport `sendMessage` returning either `IEAResponse.RSP_CODE_SUCCESS` or `IEAResponse.RSP_CODE_FAILURE` have completed. Similarly `sendMessage` calls returning `IEAResponse.RSP_CODE_ACK` are marked as pending and are considered in process, running asynchronously either in a background thread or in some other fashion`. Such requests follow the lifecycle shown in the asynchronous transaction diagram by creating a thread to process the result and then call back into the SM layer via a callback mechanism to signal request complete.

As a result asynchronous requests must be handled in separate threads or via external applications.

Automatic versus programmatic asynchronous behaviors

Actions are normally synchronous in nature. Under normal circumstances an action starts, does its work and completes returning success or failure. However actions can sometimes take long periods of time to complete. Service Manager supports such actions in two ways, each of which places the burden of completing the action on a different set of shoulders. First, actions can be marked asynchronous by the Service Manager itself. Service Manager watches each transactions and determines and if any action has "timed out". Actions which have timed out are ack'd asynchronously automatically. Timed out actions are not stopped or in any way changed. In fact they continue to execute normally with the expectation that they will complete but for some reason are running late. The Service Manager, when the action completes, runs the remainder of the actions lifecycle normally.

Actions which return `RSP_CODE_ACK` are considered programmatically or explicitly acknowledged. Such actions will never complete on their own, although Service Manager will eventually delete such actions if they remain inactive long enough. Once an action is marked inactive an external mechanism must be provided for capturing and completely the actions processing.

In support of asynchronous messaging the `ISMServiceManager` interface, and implementation, provides the following method.

receiveAsynchResponse method

```
public interface ISMServiceManager {
. . .
    public void receiveAsynchResponse(
                Long requestId,
                String statusCode,
                Serializable data)

. . .
}
```

The receive asynchronous response method is used by

1. Obtaining the originating request id, The originating id must match the id provided via the sendMessage call originally staring the request.
2. Obtaining an appropriate status code, a status of `IEIResponse.RSP_CODE_SUCCESS`; or `IEIResponse.RSP_CODE_FAILURE` from the back end service.
3. Returning any optional `serialable` data expected by action.

The example below shows the core steps to completing an asynchronous request and would normally be inserted into a web service, a stand alone web application or other utility application which can be called by the OSS completing the action. Key to completing a transaction is the ability to obtain the originating ID and interface with the SM Service Manager. The **bolded** lines below illustrate how to create the result codes and well as call back into the service manager.

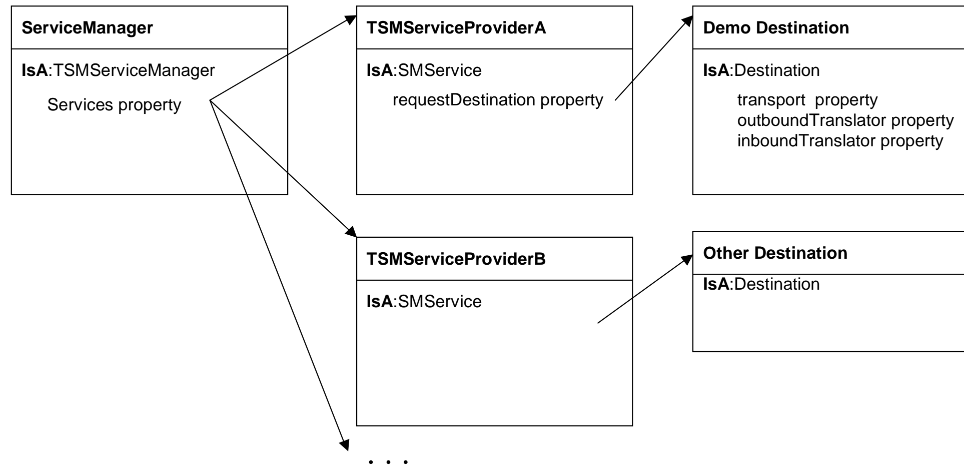
Completing Asynchronous transactions explicitly

```
import com.edocs.common.sm.api.*;
import com.edocs.common.eai.core.EIResponse;
import com.edocs.common.eai.api.core.*;
import com.edocs.common.eai.api.exception.*;
import com.edocs.application.tsm.sm.core.*;
import com.edocs.application.tsm.sm.data.*;
import com.edocs.domain.telco.api.tsm.*;
Long originalID;
// captured from sendMessage(Long requestId,...)
int actionID = -1; // from sendMessage(…,IEIAction
action,...)
Serializable payload = null;
String rspCode = IEIResponse.RSP_CODE_SUCCESS;
switch (actionID) {
    case TSMAction.ACTION_TYPE_PORT_NUMBER:
        PortPhoneNumberRsp rspData = new
PortPhoneNumberRsp();
        // populate from implementation
        payload = rspData
        break
}
ISMServiceManager sm =

TSMServiceManager.getTSMServiceManager();
sm.receiveAynchResponse(originalID, rspCode, payload);
. . .
```

Integrating Transactions

Self Service Manager Transactions are connected to the underlying connector that handles them via services definitions in the `ServiceManager` bean. Each such connection is defined in `sm.xma.xml` as shown in the relationship diagram below:



Service Manager beans have a **`services`** property that defines the mapping between an action request and the underlying `SMServices` instance bean that provides the service. `SMServices` instances in turn define a set of core operational properties and a `Destination` bean which implements the transport required to provide the service.

A portion of the `ServicesManager` bean definition is shown below.

ServiceManager Bean Example

```
<bean id="ServiceManager"

class="com.edocs.common.tsm.sm.core.TSMServiceManager">
    . . .
    <property name="services">
        <map>
           <entry key="chgVmPass">
              <ref bean="TSMService"/>
           </entry>
           . . .
           <entry key="bulkRequest">
              <ref bean="TSMService"/>
           </entry>
       </map>
    </property>
    <property name="responseHandlers" >
       <map>
           <entry key="created">
                <ref bean="LifecycleHandler.created"/>
            </entry>
            <entry key="success">
                <ref bean="LifecycleHandler.success"/>
            </entry>
          . . .
       </map>
    </property>
</bean>
```

Properties of com.edocs.common.tsm.sm.core.TSMServiceManager:

Property/Attribute	Value
id	Always ServiceManager.
class	Always com.edocs.common.tsm.sm.core.TSMServiceManager.
services	Map of service. Each entry key represents an action and the associated ref bean the SMService bean which provides it.
responseHandlers	Map of response handlers. See Response Processors

Each supported service has an entry in the services map that defines the name of the service provider and the name of the action the provider supports.

`SMService` Bean Example

```
<bean id="TSMService"
      class="com.edocs.common.sm.core.SMService">
    <property name = "requestDestination">
        <ref bean=" DemoTSMDestination"/>
    </property>
    <property name="requestMode">
         <value>sync</value>
    </property>
    <property name="authorizationCheck">
        <value>false</value>
    </property>
    <property name="timeout">
        <value>2</value>
    </property>
</bean>
```

Properties of `com.edocs.common.tsm.sm.core.ServiceManager:`

Property/Attribute	**Value**
`Id`	Bean identifier, referenced in **`ServiceManager`** bean definition.
`Class`	Always `com.edocs.common.sm.core.SMService`.
`RequestDestination`	ID of bean to be used as the destination of the actions sent to this service. See Defining Destinations
`RequestMode`	Future use, must always be synchronized
`authorizationCheck`	Future use, must always be false
`Timeout`	Timeout value for messages to this service, in minutes.

Response Processors

Both Actions and Service Managers can define processing that is executed when a connector returns a specific response; such processing is referred to as a *Response Processor* or *Handler*. Response handlers are special processors called when a given result is returned. For example a special response handler might be defined for a given service to handle system unavailable errors. Default response handlers are defined to handle the specific responses list below:

Default Response Handlers

Response Handler Codes	When called
Success	Called when an action returns a status of success
Ack	Future use only. Called when a response handler returns a status of ack
Failure	Called when an action returns failure
systemUnavailable	Called when an action returns system unavailable
bulkElementSuccess	Called when an action returns a status of success for an element of the bulk operations
bulkElementFailure	Called when an action returns a status of failure for an element of the bulk operations

Developers overload the provided response handlers and providing replacements that perform specialized processing when a given response is processed. Such handlers can be associated with a SMService instance or with specific actions.

Response handlers are developed by extending the class SMResponseHandler and implementing processEIResponse(…) as shown below. Response handlers have complete access to both the calling Service Manager instance and the action that generated the response.

Note that after any custom code the super.processEIResponce() method must be called as shown to complete the lifecycle management functions of the SM system.

Custom Response Handler

```
package com.myco.demo;
import com.edocs.common.sm.api.ISMServiceManager;
import com.edocs.common.sm.api.ISMServiceRequest;
import com.edocs.common.sm.api.exception.SMException;
import com.edocs.common.sm.core.*;
public class CustomResponseHandler
                extends SMResponseHandler {
    public void processEIResponse(ISMServiceManager sm,
                                  ISMServiceRequest srx)
                                       throws
SMException {
        System.out.println(
                  "CustomResponseHandler called for
action " +
                   srx.getAction().getExternalName());
        super.processEIResponse(sm, srx);
    }
}
```

Configuring Custom Response Handlers

There are two steps in configuring response handlers:

1. Define a bean representing the response handler
2. Reference the newly defined handler in either a `TSMServiceManager` or an `SMAction` instance.

Defining a bean reference for a response handler involves adding a snippet of XML as shown below.

```
<bean id="CustomResponse.systemUnavailable"
      class=" com.myco.demo.CustomResponseHander" >
    <property name="theStatus">
         <ref bean="Status.runnable" />
     </property>
     <property name="theQueue">
         <ref bean="Queue.runnable"/>
      </property>
      <property name="historyTypeKey">
           <value>1000</value>
      </property>
</bean>
```

Properties of `SMResponseHandler` derived classes:

Property/Attribute	Value
`id`	Bean identifier, by convention the name of the response handler followed by ".”(dot) followed by the status being handled.
`class`	Fully qualified class name of a class extending `com.edocs.common.sm.core.SMResponseHandler`
`resolutionMsg`	Text string representing the message to be stored in the history table with respect to this response handler.
`historyTypeKey`	A unique integer identifier used to identify the action in the history list, values less then 1000 are reserved for history.
`theQueue`	Reserved. Optional. Bean id of a queue where this message should be placed. Possible values are: `Queue.runnable` – retry operation. `Queue.needsManualIntervention` – action cannot be completed without manual intervention. `Queue.ack` – action is part of a bulk asynchronous action. Future use only.
`theStatus`	Bean id of an existing status message matching the status being handled. For example: `status.runnable`, **`status.success`**, `status.failure` etc.

Defined response handlers are associated with actions by setting up a map of status to response handlers using the optional `SMAction` property `responseHandlers`. Or by replacing one of the default response handlers defined for `TSMServiceManager` beans. An example of specifying a custom response handler for an action is shown below.

```
<bean id="Action.chgVmPass"
      class="com.edocs.common.sm.core.SMAction" >
. . .
    <property name="responseHandlers" >
        <map>
            <entry key="success">
                <ref bean="CustomResponse.success"/>
            </entry>
            <entry key="systemUnavailable">
                <ref
bean="CustomResponse.systemUnavailable"/>
            </entry>

         </map>
    </property>
</bean>
```

Note that a single response handler class can be configured to handle multiple by adding code in the `processEIResponse(ISMServiceManager sm, ISMServiceRequest srx)` method similar to:

```
ISMResponse resp = srx.getTheResponse();
String respCode = resp.getResponseCode();
if (respCode.equals(ISMResponse.RSP_CODE_SUCCESS)) . . .
```

And then creating multiple instances of the response handler, one for each status such as:

```
<bean id="CustomResponse.systemUnavailable"
      class=" com.myco.demo.CustomResponseHander" >
. . .
</bean>
<bean id="CustomResponse.success"
      class=" com.myco.demo.CustomResponseHander" >
. . .
</bean>

. . .
```

4 Glossary of Terms

Bulk **–** The concept of performing the same action on multiple service agreements or accounts in one transaction flow.

CTN **–** Cellular Telephone Number: See MTN and MIN.

DSN **–** Device Serial Number: The unique identification number embedded in a wireless phone by the manufacturer. Each time a call is placed, the DSN is automatically transmitted to the base station so the wireless carrier's mobile switching office can check the call's validity. The DSN cannot be altered in the field. The DSN differs from the mobile identification number, which is the wireless carrier's identifier for a phone in the network. MINs and DSNs can be electronically checked to help prevent fraud.

ETL**–** Process of extracting data from some source, such as a text file, translating that data into a readable format, and loading the data into an edocs database..

Final Request **–** A final request occurs upon the submission of the transaction to the appropriate backend system for processing, all information needed to process the transaction is included in this request.

Interim Request **–** An interim request is used to retrieve additional information from the external backend systems that is required to complete a transaction.

LNP **–** Local Number Portability: The ability of subscribers to switch local or wireless carriers and still retain the same phone number, as they can now with long-distance carriers.

MIN **–**Mobile Identification Number: Uniquely identifies a mobile unit within a wireless carrier's network. The MIN often can be dialed from other wireless or wireline networks. The MIN is meant to be changeable, since the phone could change hands or a customer to another city. The number differs from the ESN which is the unit number assigned by a phone manufacturer. (also see MTN).

MTN**–** Mobile Telephone Number: See MIN.

OSS Operational Support System, a generic term for a suite of programs that enable an enterprise to monitor, analyze and manage a network system. OSS has since been applied to the business world in general to mean a system that supports an organization’s network operations.

SIM – Subscriber Identity Module card - a small printed circuit board that must be inserted in any GSM-based mobile phone when signing on as a subscriber. It contains subscriber details, security information and memory for a personal directory of numbers. The card can be a small plug-in type or sized as a credit-card but has the same functionality. The SIM card also stores data that identifies the caller to the network service provider.

TNI – Telephone Number Inventory is a term used to identify the system of record for all telephone number management.

Self Service Manager–Self Service Manager is an Siebel application that provides Self-Service and Order Management functionality within the Self Service for Communications Application Suite.

Index

B

Billing Manager
- Account Number, 32
- CAM, 31
- Customer Account Management, 31
- Obtaining ICustomer, 32
- SessionUtils, 31

Bulk, 85

Bulk Load
- Error Handling, 24
- ETLMain, 23
- Run time requirements, 22
- Self Service Manager Data Transfer Objects, 21
- tsmproperties.xml, 19
- Validation, 25

Bulk Loading
- Device Type Example, 26
- DeviceType, 26
- RatePlan, 26
- RatePlanGroup, 27
- Service Agreement, 28
- Service Agreement Attributes, 27

C

Cellular Telephone Number, 85

CTN, 85

D

Digital Serial Number, 85

DSN, 85

E

ESC
- Action, 55
- Action Example Bean, 65
- Connector, 55
- Custom Response Handler Example, 82
- Default Response Handlers, 82
- Defining Actions, 62
- Defining Destination, 61
- Defining Transactions, 62
- Destination, 56
- Destination Example Bean, 62
- Destination Properties, 61
- Response Handlers, 56
- Service Managers, 56
- ServiceManager Example Bean, 80
- ServiceManager Properties, 81
- SMAction Example Bean, 84
- SMAction Properties, 65
- SMResponseHandler Properties, 83
- SMService Example Bean, 81
- Transaction Characteristics, 65
- Transform, 55
- Transform Example, 59

Transport, 55
Transport Example, 56
TSMServiceManager Properties, 80
ETL, 85
Extract Translate Load, 85

F

Final Request, 85

H

Help
technical support, 10
Hierarchy
Obtaining IHierarchyNode, 32

I

Interim Request, 85

L

LNP, 85
Local Number Portability, 85

M

MIN, 85
Mobile Identification Number, 85
Mobile Telephone Number, 85
MTN, 85

O

Operational Support System, 85
OSS, 85

S

Self Service for Communications, 86
Self Service Manager, 86
Self Service Manager Transaction
Self Service Manager Service Manager, 41
SIM, 86
Spring Framework, 61
SSM APIs
DeviceType, 29
IDeviceType, 29
IRatePlan, 30
IRatePlanFeatureInstance, 30
IRatePlanGroup, 30
IServiceAgreement, 29
ITSMDataManager, 29
RatePlan, 30
RatePlanFeatureInstance, 30
ServiceAgreement, 29
TSMDataManager, 29
SSM APIsr
RatePlanGroup, 30
SSM Purge
purgeDeviceType, 36
purgeRatePlan, 36
purgeRatePlanGroup, 36
purgeServiceAgreement, 36
SSM Search
createDeviceType, 35
getAllDeviceTypes, 34
getDeviceType, 35
Obtaining IServiceAgreement, 33
Obtaining ITSMDataManager, 33
SSM Update
deactivateDevice, 35
updateDeviceType, 35

Subscriber Identity Module card, 86

T

Telephone Number Inventory, 86
TNI, 86
Transactions
  Activate Service Transaction, 46
  Add Delete Feature Tx, 47
  Bulk Transaction Handling, 53
  Categories, 37
  Change Digital Subscriber Transaction, 45
  Change Phone Number Transaction, 43
  Change Rate Plan Tx, 49
  Change Subscriber ProfileTx, 51
  Change Voice Mail Password Tx, 42
  Deactivate Service Transaction, 46
  Find Subscriber Profile Tx, 51
  Generate Voicemail Password Tx, 41
  Obtain available area codes Tx, 52
  Obtain available exchanges Tx, 52
  Obtain available phone number Tx, 52
  Obtaining ITSMServicesManager, 40
  Port Phone Number Transaction, 44
  resetOrderInProgress, 37
  Response Handling, 54
  setServiceAgreementStatus, 37
  SMAuthorizationException, 55
  SMException, 55
  SMInvalidArgumentException, 55
  SMServiceNotSupportedException, 55
  Transaction, 55
  Transaction Response, 38
  Transient vs *Persisted* Transactions, 38