Oracle9*i*AS Personalization

Recommendation Engine API Programmer's Guide

Release 9.0.1

July 2001

Part No. A87536-01

ORACLE

Oracle9*i*AS Personalization Recommendation Engine API Programmer's Guide, Release 9.0.1

Part No. A87536-01

Contents

2 REAPI Supporting Classes

3 Using the Recommendation Engine Proxy

A REAPI Examples and Usage

Send Us Your Comments

Oracle9*i*AS Personalization Recommendation Engine API Programmer's Guide, Release 9.0.1

Part No. A87536-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- DARWINDOC@us.oracle.com
- FAX: 781-684-7738. Attn: Oracle9*i*AS Personalization Documentation
- Postal service:
  Oracle Corporation
  Oracle9*i*AS Personalization Documentation
  200 Fifth Avenue
  Waltham, Massachusetts 02451
  U.S.A.

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This manual describes how a Java programmer can use Oracle9*i*AS Personalization (OP) Recommendation Engine API (REAPI) to collect data and obtain recommendations in real time.

Intended Audience

This manual is intended for Java programmers who create and maintain Web sites that use Oracle9*i*AS Personalization.

Structure

This manual contains the following chapters and appendixes:

Chapter 1	Introduces REAPI.
Chapter 2	Describes the REAPI supporting classes.
Chapter 3	Describes the methods used to manage sessions, manage data, and request recommendations.
Appendix A	Contains examples of how to perform common tasks with the REAPI.
Appendix B	Contains a complete example of REAPI use.

Where to Find More Information

The documentation set for Oracle9*i*AS Personalization at the current release consists of the following documents:

- README.htm, on the Oracle9*i*AS Personalization CD; this file contains platform-specific installation instructions.
- *Oracle9*i*AS Personalization Release Notes*, Release 9.0.1.
- *Oracle9*i*AS Personalization Administrator's Guide*, Release 9.0.1 (includes installation instructions that are the same across all platforms).
- *Getting Started with Oracle9*i*AS Personalization*, Release 9.0.1.
- *Oracle9*i*AS Personalization Recommendation Engine API Programmer's Guide*, Release 9.0.1. A programmer's manual for programming the recommendation engines in real time (this document).
- *Oracle9*i*AS Personalization Recommendation Engine Batch API Programmer's Guide*, Release 9.0.1. A programmer's manual for obtaining bulk recommendations.

Related Manuals

For more information about the database underlying OP, see:

- *Oracle9*i *Administrator's Guide*
- *Oracle9*i *Application Server Installation Guide* (the appropriate version for your operating system).

Requirements

OP documentation is distributed on the same CD that OP is distributed on. Documentation is provided in PDF and HTML formats.

After OP is installed, the OP documentation can be read by opening the following URL using either Netscape or Internet Explorer:

```
http://<server-name>/opDoc/
```

where `<server-name>` is name of the system where OP is installed.

You can read or print the documentation directly from the CD or from your browser.

To view the PDF files, you will need

- Adobe Acrobat Reader 3.0 or later, which you can download from `www.adobe.com`.

To view the HTML files, you will need

- Netscape 4.x or later, or
- Internet Explorer 4.x or later

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Conventions

In this manual, Windows refers to the Windows95, Windows98, and the Windows NT operating systems.

The SQL interface to Oracle 9*i* is referred to as SQL. This interface is the Oracle 9*i* implementation of the SQL standard ANSI X3.135-1992, ISO 9075:1992, commonly referred to as the ANSI/ISO SQL standard or SQL92.

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The table below shows the conventions used in this manual and their meanings.

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
syntax text	Text in syntax font is used to describe the syntax of Java code.
italic text or *syntax*	Text or syntax in italics specify user-supplied names or data.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.

1
Introduction

The OP (Oracle9i*AS Personalization*) REAPI (Recommendation Engine Application Programming Interface) enables a Web application written in Java to collect and preprocess data used to build OP models and to request recommendations. The recommendations are returned in real time.

OP also includes the Recommendation Engine Batch API, which returns bulk recommendations.

REAPI was designed to be extensible, to minimize the number of API functions, to be uniform, and to keep the number of arguments to a minimum.

Appendix A contains examples of how to perform common tasks using REAPI.

Appendix B contains a complete example of REAPI use.

OP includes a demo program that helps you learn how the API methods work. OP also contains javadoc associated with the `jar` files.

Note: REAPI and REAPI Demo are installed on the system where Oracle9*i*AS is installed.

REAPI Prerequisites

Before you can use REAPI methods, OP must be installed and the appropriate tables must be created and populated. If you plan to use existing data, the data must be converted to use the appropriate schema. If you plan to use Hot Picks, you must specify Hot Picks groups, as well as Hot Picks. If you are using one or more taxonomies, they must be properly specified.

If you plan to request recommendations, you must build and deploy an OP package before you request any recommendations. Use the OP Administrative UI to do this.

For detailed information about how to install OP, see the *Oracle9iAS Personalization Administrator's Guide*. For information about how to create and deploy packages, see *Getting Started with Oracle9iAS Personalization* and the online help for the Administrative UI.

Definitions and Concepts

This section describes the collections of methods that make up the REAPI and concepts and terms used in the description of the API.

End Users (Customers and Visitors)

End users (users of a Web site that uses OP for personalization services) are divided into two groups: customers and visitors. A *customer* is a registered user, who can be identified by a unique customer ID assigned by the Web application. A *visitor* is an unregistered user; a visitor is usually assigned a visitor ID by the Web application. A visitor can become a customer by completing registration. End users are specified using the IdentificationData class.

Sessions

The Web application that calls REAPI can be either

- *sessionful*, that is, it creates a session for each user visit to the Web site
- *sessionless* (stateless), that is, it does not create such a session

OP is always sessionful; it creates a session even if the Web application does not.

During the OP session, the Web application can collect data and/or request recommendations.

Sessionful Web Applications

If the Web application is sessionful, it can explicitly create an OP session using one of the following methods:

- createCustomerSession to create a session for a customer (registered user)
- createVisitorSession to create a session for a visitors (a user who isn't registered)

The Web application then uses the createSessionful() method of the class IdentificationData to create identification data used during the session.

Sessionless Web Applications

If the Web application is sessionless, the recommendation engine (RE) automatically creates an OP session for the first REAPI method (either data collection or recommendation request) issued for a given customerId.

The Web application then uses the createSessionless() method of the class IdentificationData to create identification data used during the session.

Collecting Data

OP supports collecting several kinds of data: demographic data, purchasing, rating, and navigation data. The Web application decides what kind of data to collect.

> **Note:** Ratings in OP are in "ascending order of goodness", that is, the higher the rating, the more the user prefers the item. Low rated items are items that the user does not prefer. OP algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

Data for both visitors and customers can be either persisted (stored in the database) or not. Data is collected in an RE and is persisted in the MTR. A configuration parameter specifies whether or not to persist data. For more information about what data is persisted and when, see the discussion of data synchronization in the *Oracle9iAS Personalization Administrator's Guide*.

Data collection makes it possible to generate recommendations based on activity during the current session as well as historical data.

Making Recommendations

For both visitors and customers, recommendations are based on two kinds of data:

- Historical data, which is stored in the database and retrieved at the beginning of the current session
- Data collected during the current session

Hot Picks

On some e-commerce sites, vendors promote certain products called “hot picks”; the hot picks might, for example, be this week’s specials. The hot pick items are grouped into *Hot Pick Groups*. The hot pick items and groups are specified by the OP administrator in the Mining Table Repository (MTR); each hot picks group is identified by a (long) integer.

Using REAPI

Before you can use OP, the following must be true:

- A recommendation engine farm containing at least one recommendation engine must exist.
- A package must have been successfully deployed in the recommendation engine farm.

Getting Started with Oracle9iAS Personalization and the online help for the OP Administrative UI explain how to perform these steps.

Some REAPI methods collect data in the Recommendation Engine (RE), which resides in Oracle9*i* database; others retrieve recommendations.

You can then either collect data or get recommendations. You cannot get recommendations until there is an existing deployed package, which is created using the OP Administrative UI. You cannot create a package until there is some data available; this data can either be collected using the REAPI or converted from existing data collected by your Web application and stored in an Oracle database.

When you design an OP application, you must decide if there should be more than one RE and, if there are several REs, how to use them. For a discussion of the design considerations, see "Recommendation Engine Usage" in Appendix A.

Recommendations may want to take income level (salary) into consideration; for example, you may want to recommend items that the user can afford to buy. If the users of the Web site live in several countries (for example, the Web site sells items in Japan and India), see "Handling Multiple Currencies" in Appendix A.

Sample REAPI Usage

OP includes a sample program (REAPI Demo) that illustrates the use of many of the REAPI methods. This sample program can be used to learn about REAPI calls and can also be used to verify that OP is correctly installed.

After you have installed OP, start REAPI Demo by opening the following URL in Netscape or Internet Explorer:

```
http://server/redemo/
```

where *server* is the name of the system where Oracle9*i*AS is installed. The REAPI test site is displayed.

To view the source code for the OP REAPI Demo, click "View Source Code."

For information about how run the demo, see *Getting Started with Oracle9iAS Personalization*. There are also some examples of how to perform typical tasks using REAPI in Appendix A of this manual and a complete example using all REAPI functionality in Appendix B.

Creating an REProxyRT Object

Before you can use any of the REAPI methods, you must create at least one REPRoxyRT object; see Appendix A for details.

Starting a Session

If the Web application is sessionful, it must start a session. The Web application must take care to specify a unique session ID for each application session. For an example of how to do this, see Appendix A.

If the Web application is sessionless, it does not have to start a session. (In this case OP will start an internal session for a given user when the Web application makes the first REAPI call.)

OP starts a session for each user, as defined by the user ID provided by the Web application. If two people are using a site at the same time and they both use the same user ID (and the application does not distinguish between different sessions), then OP assigns the same session ID to both users. OP treats them as a single user. After the OP session times out, OP assigns a new session ID when the user logs in again.

Sessionful and sessionless applications get recommendations on behalf of a user. User IDs must be unique.

Creating Instances of Objects

To use the REAPI, you must create instances of the objects used by the API method signatures. Use the REAPI supporting classes, described in Chapter 2, to create

these instances. It is always necessary, for example, to create an IdentificationData object. For examples, see Appendix A and the complete example in Appendix B.

Providing Data

OP generates recommendations based on data describing past and/or current user behavior.

If the Web application has user data stored in an Oracle table, the data must be transformed and stored in the Mining Table Repository (MTR) before it can be used to generate recommendations.

A Web application can also collect data during the current session. This data can be used to make recommendations during the current session and it can be stored to make recommendations in future sessions.

The following methods manage data that you collect during the current session:

```
addItem();
addItems();
removeItem();
removeItems();
```

These methods add information to or remove information from the OP Recommendation Engine (RE) and its cache for a specified OP internal session ID. The session ID is stored in the IdentificationData passed in the REAPI method.

Getting Recommendations

To get a recommendation, the Web application calls one of the following recommendation methods:

- crossSellForItemFromHotPicks()
- crossSellForItemsFromHotPicks()
- rateItem()
- rateItems()
- recommendBottomItems()
- recommendCrossSellForItem()
- recommendCrossSellForItems()
- recommendFromHotPicks()

- recommendTopItems()
- selectFromHotPicks()

These methods are used to get recommendations for either visitors or customers.

Creating Recommendations

OP uses rule tables stored in the RE cache to calculate the recommendations requested by the methods listed above. The specific rule table used depends upon the REAPI method made. In general, the antecedents of the rules are matched against the data in cache (both historical and current session data) and the probabilities of the various consequents are computed. These items are then ordered by probability, and numberOfItems (an API argument) items are returned.

If there is enough memory in the RE database, the RE caches all rules associated with a particular package deployed from the MTR to the RE, not just the most recently used rules.

Scoring for Visitors: For visitors, only current session data is used. Usually only navigational data (click stream) data is persisted for visitors, but if the Web application persists other kinds of data for visitors, that data will also be used for model building. (OP builds a model when it creates a package.) The scoring of these different methods uses only the data stored in the RE cache by addItem() methods.

Scoring for Customers: For customers, the scoring is the same as for visitors. For customers, historical data can also be used for scoring.

The OP Mining Table Repository (MTR) contains historical rating, transactional data, and navigational data stored in both detailed and aggregated formats. The MTR also contains demographic data. When scoring for customers, the RE retrieves the demographic data and the aggregated version of the other data source types.

Making Recommendations

REAPI methods that make recommendations return the recommendations to the Web application. The Web application then decides which recommendations to pass to the user.

Closing a Session

A sessionful Web application should use `closeSession()` to close the OP session. If there is no explicit `closeSession()` method, OP automatically closes the session when it times out.

In a sessionless Web application, the OP session closes when it times out.

For either sessionless or sessionful Web applications, the time-out period is specified as a configuration parameter.

See the *Oracle9iAS Personalization Administrator's Guide* for information about configuration parameters.

Removing the REProxyRT Object

Before you exit the application, you should destroy any proxy objects that you created. To do this, use one of the following methods in the class REProxyManager: destroyProxy or destroyAllProxies.

2

REAPI Supporting Classes

This chapter describes the supporting classes for the REProxy class. These classes are used to create instances of the objects used by the methods described in Chapter 3. You may be able to create one instance of many of these classes and use that one instance as an argument for several calls.

All methods described in this chapter are public.

The supporting classes are divided into two categories:

- EnumType interfaces
- Other supporting classes

Ratings in OP

Ratings in OP are in "ascending order of goodness", that is, the higher the rating, the more the user prefers the item. Low rated items are items that the user does not prefer. OP algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

Location of Classes

The following classes are in the oracle.dmt.op.re.base package:

- DataItem
- Enum
- FilteringSettings
- Item

- ItemList
- TuningSettings

For example, to use the Enum interfaces, you must include the following statement in your Java program:

```
import oracle.dmt.op.re.base.Enum;
```

EnumType Interfaces

Many of the REAPI methods reference attributes that can take on a finite number of values. The interface Enum is used to implement the base class for these enumerated constants.

The Enum interface has a nested EnumType class with the following general methods:

```
int getId();

String toString();

String getName();

boolean isEqual(EnumType);
```

The following interfaces extend EnumType:

- CategoryMembership
- DataSource
- Filtering
- InterestDimension
- PersonalizationIndex
- ProfileDataBalance
- ProfileUsage
- RecommendationAttribute
- Sorting
- User

CategoryMembership Interface

CategoryMembershipType is implemented as:

- CategoryMembershipType (a class that extends EnumType)
- CategoryMembership (an interface)

The class CategoryMembership has the following methods:

```
CategoryMemberShipType getType(String name);

CategoryMemberShipType getType(int);
```

CategoryMembership specifies how categories in a list of categories should be applied for filtering. For example, Enum.CategoryMembership.EXCLUDE_ITEMS specifies that items from the categories in the category list should be excluded from the recommendations list. For details, see FilteringSettings later in this chapter.

CategoryMembership takes on the following values:

- Enum.CategoryMembership.EXCLUDE_ITEMS
- Enum.CategoryMembership.INCLUDE_ITEMS
- Enum.CategoryMembership.EXCLUDE_CATEGORIES
- Enum.CategoryMembership.INCLUDE_CATEGORIES
- Enum.CategoryMembership.LEVEL
- Enum.CategoryMembership.SUBTREE_ITEMS
- Enum.CategoryMembership.SUBTREE_CATEGORIES
- Enum.CategoryMembership.ALL_ITEMS
- Enum.CategoryMembership.ALL_CATEGORIES

The following statement assigns Enum.CategoryMembership.LEVEL to the variable myEnum:

```
CategoryMembershipType myEnum = Enum.CategoryMembership.LEVEL;
```

DataSource Interface

DataSource is implemented as:

- DataSourceType (a class that extends EnumType)
- DataSource (an interface)

The class DataSourceType has the following methods:

```
DataSourceType getType(String name);
```

DataSourceType getType(int);

DataSource specifies the type of data that is used when OP performs certain operations. For example, Enum.DataSource.DEMOGRAPHIC specifies that demographic data should be used. The class DataItem, described later in this chapter, uses DataSource. Note that a given method may not support all values of DataSource. For details, see the description of the methods in Chapter 3.

DataSource takes on the following values:

- Enum.DataSource.DEMOGRAPHIC
- Enum.DataSource.PURCHASING
- Enum.DataSource.RATING
- Enum.DataSource.NAVIGATION
- Enum.DataSource.ALL

The following statement assigns Enum.DataSource.ALL to the variable myEnum:

```
DataSourceType myEnum = Enum.DataSource.ALL;
```

Filtering Interface

Filtering is implemented as:

- FilteringType (a class that extends EnumType)
- Filtering (an interface)

The class FilteringType has the following methods:

FilteringType getType(String name);

FilteringType getType(int);

Filtering is used to turn filtering on or off. See the description of the FilteringSettings class, later in this chapter for more information.

Filtering takes on the following values:

- Enum.Filtering.ON
- Enum.Filtering.OFF

The following statement assigns Enum.Filtering.OFF to the variable myEnum:

```
FilteringType myEnum = Enum.Filtering.OFF;
```

InterestDimension Interface

InterestDimension is implemented as:

- InterestDimensionType (a class that extends EnumType)
- InterestDimension (an interface)

The class InterestDimensionType has the following methods:

```
InterestDimensionType getType(String name);

InterestDimensionType getType(int);
```

InterestDimension indicates the type of interest that the user of the Web site has in a given item. NAVIGATION indicates that the user is interested in the items. PURCHASING indicates that the user purchased an item. RATING indicates that the user likes the items. For more information, see the description of the RecommendationList and TuningSettings classes later in this chapter.

InterestDimension takes on the following values:

- Enum.InterestDimension.NAVIGATION
- Enum.InterestDimension.PURCHASING
- Enum.InterestDimension.RATING

The following statement assigns Enum.InterestDimension.PURCHASING to the variable myEnum:

```
InterestDimension myEnum = Enum.InterestDimension.PURCHASING;
```

PersonalizationIndex Interface

PersonalizationIndex is implemented as:

- PersonalizationIndexType (a class that extends EnumType)
- PersonalizationIndex (an interface)

The class PersonalizationIndexType has the following methods:

```
PersonalizationIndexType getType(String name);

PersonalizationIndexType getType(int);
```

PersonalizationIndex specifies how "unusual" the recommendations returned will be. For example, LOW specifies not unusual. For more information, see the description of the TuningSettings class later in this chapter.

PersonalizationIndex takes on the following values:

- Enum.PersonalizationIndex.LOW
- Enum.PersonalizationIndex.MEDIUM
- Enum.PersonalizationIndex.HIGH

The following statement assigns Enum.PersonalizationIndex.LOW to the variable myEnum:

```
PersonalizationIndexType myEnum = Enum.PersonalizationIndex.LOW;
```

ProfileDataBalance Interface

ProfileDataBalance is implemented as:

- ProfileDataBalanceType (a class that extends EnumType)
- ProfileDataBalance (an interface)

The class ProfileDataBalanceType has the following methods:

ProfileDataBalanceType getType(String name);

ProfileDataBalanceType getType(int);

ProfileDataBalance specifies whether to take data from the current session or from history or to balance data between data from the current session and history when making recommendations. For more information, see the description of the TuningSettings class later in this chapter.

ProfileDataBalance takes on the following values:

- Enum.ProfileDataBalance.HISTORY
- Enum.ProfileDataBalance.BALANCED
- Enum.ProfileDataBalance.CURRENT

The following statement assigns Enum.ProfileDataBalance.BALANCED to the variable myEnum:

```
ProfileDataBalanceType myEnum = Enum.ProfileDataBalance.BALANCED;
```

ProfileUsage Interface

ProfileUsage is implemented as:

- ProfileUsageType (a class that extends EnumType)
- ProfileUsage (an interface)

The class ProfileUsageType has the following methods:

```
ProfileUsageType getType(String name);

ProfileUsageType getType(int);
```

ProfileUsage specifies whether the recommendation list can include or exclude items in a customer's profile. For more information, see the description of TuningSettings later in this chapter.

ProfileUsage takes on the following values:

- Enum.ProfileUsage.INCLUDE
- Enum.ProfileUsage.EXCLUDE

The following statement assigns Enum.ProfileUsage.INCLUDE to the variable myEnum:

```
ProfileUsageType myEnum = Enum.ProfileUsage.INCLUDE;
```

RecommendationAttribute Interface

RecommendationAttribute is implemented as:

- RecommendationAttributeType (a class that extends EnumType)
- RecommendationAttribute (an interface)

The class RecommendationAttributeType has the following methods:

```
RecommendationAttributeType getType(String name);

RecommendationAttributeType getType(int);
```

RecommendationAttribute indicates the attribute to be included in the returned content; possible choices are type, ID, and prediction. For more information, see the descriptions of the ContentItem and RecommendationContent classes later in this chapter.

RecommendationAttribute takes on the following values:

- Enum.RecommendationAttribute.TYPE
- Enum.RecommendationAttribute.ID
- Enum.RecommendationAttribute.PREDICTION

The following statement assigns Enum.RecommendationAttribute.URL to the variable myEnum:

```
RecommendationAttributeType myEnum = Enum.RecommendationAttribute.TYPE;
```

Sorting Interface

Sorting is implemented as:

- SortingType (a class that extends EnumType)
- Sorting (an interface)

The class SortingType has the following methods:

```
SortingType getType(String name);

SortingType getType(int);
```

Sorting indicates whether sorting is done (none implies no sorting), and, if sorting is done, how it is done (ascending or descending). For more information, see the discussions of the ContentItem and RecommendationContent classes later in this chapter.

Sorting takes on the following values:

- Enum.Sorting.NONE
- Enum.Sorting.DESCENDING
- Enum.Sorting.ASCENDING

The following statement assigns Enum.Sorting.NONE to the variable myEnum:

```
SortingType myEnum = Enum.Sorting.NONE;
```

User Interface

User is implemented as:

- UserType (a class that extends EnumType)
- User (an interface)

The class UserType has the following methods:

```
UserType getType(String name);

UserType getType(int);
```

UserType is either customer, a registered user of the calling Web site, or visitor, an unregistered user. For more information see the description of the IdentificationData class later in this chapter.

UserType takes on the following values:

- Enum.User.CUSTOMER

- Enum.User.VISITOR

The following statement assigns Enum.User.CUSTOMER to the variable myEnum:

```
UserTypeType myEnum = Enum.User.CUSTOMER;
```

Other Supporting Classes

The other supporting classes are

- ContentItem
- DataItem
- FilteringSettings
- IdentificationData
- Item
- ItemDetailData
- Recommendation
- RecommendationContent
- RecommendationList
- TuningSettings

ContentItem

Encapsulates the information that should be included in the object returned by a recommendation request. It describes the attributes to be included in the recommendation list returned by a call as well as specifying whether the list should be sorted according to one of the attributes. RecommendationContent, described later in this chapter, is any array of items of type ContentItem; the description of RecommendationContent explains how sorting order works when multiple orders are specified.

Attributes

No public attributes.

Methods

getContentAttribute();

Returns the attribute to be included in the returned content. Attributes can be any of the following:

- Enum.RecommendationAttribute.TYPE
- Enum.RecommendationAttribute.ID
- Enum.RecommendationAttribute.PREDICTION

getSorting()

Returns the sorting option for the content item.The following sorting options are supported:

- Enum.Sorting.NONE — result is not sorted according to this attribute.
- Enum.Sorting.DESCENDING — result is sorted in descending order according to this attribute
- Enum.Sorting.ASCENDING — result is sorted in ascending order according to this attribute

DataItem

A subclass of class Item. This class encapsulates data about an item. This object is used as an argument in the data collection methods addItem() and addItems().

Attributes

No public attributes.

Methods

There are two kinds of methods provided with this class:

- A constructor for DataItem
- Methods that return attribute values

```
//Constructor
DataItem(String type, long ID, DataSourceType dataSource, String value);
```

Create a DataItem instance for a given item. An item is uniquely identified by its type and ID. For a description of an attribute, see the description of the method that returns that attribute value.

getDataSource():

Returns the value of type DataSourceType indicating the kind of data associated with the item; the values supported are

- Enum.DataSource.DEMOGRAPHIC
- Enum.DataSource.PURCHASING
- Enum.DataSource.RATING
- Enum.DataSource.NAVIGATION

getValue()

Returns the value of type String associated with the item.

The following constructor creates a data item. For a description of an attribute, see the description of the corresponding method that returns the attribute value.

Usage Notes

dataSource cannot be Enum.DataSource.ALL.

FilteringSettings

Specifies the items to include or exclude when generating recommendations.

Release 1 of OP supports category filtering only.

Attributes

No public attributes.

Methods

There are three kinds of methods provided with this class:

- A constructor for FilteringSettings
- Methods that set the attributes values
- Methods that return attribute values

```
//Constructor
FilteringSettings (int taxonomyID);
```

Creates an instance with CategoryFiltering set to Enum.Filtering.OFF, CategoryMembership set to Enum.CategoryMembership.ALL_ITEMS, and taxonomyID set to the provided integer.

The following methods set the attributes of a FilteringSettings instance. First you must use the default constructor to create an instance. For a description of an attribute, see the description of the method that returns that attribute value.

```
setItemFiltering(int taxonomyID);
```

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.ALL_ITEMS, categoryList set to null, and taxonomyID set to the provided integer. Use this method to recommend items from *all* leaves in the taxonomy with the specified ID.

```
setItemFiltering(int taxonomyID, long[] categoryList);
```

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.INCLUDE_ITEMS, taxonomyID set to the provided integer, and categoryList set to the provided array. Use this method to recommend items that belong to the categories in the categoryList argument and the taxonomy with the specified ID.

```
setItemExclusion(int taxonomyID, long[] categoryList]);
```

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.EXCLUDE_ITEMS, taxonomyID set to the provided integer,

and categoryList set to the provided array. Use this method to recommend those items in the taxonomy with the specified ID that do *not* belong to the categories in the categoryList argument.

setItemSubTreeFiltering(int taxonomyID, long[] categoryList]);

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.SUBTREE_ITEMS, taxonomyID set to the provided integer, and categoryList set to the provided array. Use this method to recommend those items in the taxonomy with the specified ID that belong to the subtrees of the categories in the categoryList argument.

setCategoryExclusion(int taxonomyID, long[] categoryList]);

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.EXCLUDE_CATEGORIES, categoryList set to the provided array, and taxonomyID set to the provided integer. Use this method to recommend those categories in the taxonomy with the specified ID that are *not* in the category list.

setCategorySubTreeFiltering(int taxonomyID, long[] categoryList]);

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.SUBTREE_CATEGORIES, taxonomyID set to the provided integer, and categoryList set to the provided array. Use this method to recommend categories in the taxonomy with the specified ID that belong to the subtrees of the categories in the categoryList argument.

setCategoryLevelFiltering(int taxonomyID, long[] categoryList]);

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.LEVEL, taxonomyID set to the provided integer, and categoryList set to the provided array. Use this method to recommend categories in the taxonomy with the specified ID that belong to the same levels as the categories in the categoryList argument.

setCategoryFiltering(int taxonomyID);

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to Enum.CategoryMembership.ALL_CATEGORIES, and taxonomyID set to the provided integer. Use this method to recommend categories from all categories in the taxonomy with the specified ID.

setCategoryFiltering(int taxonomyID, long[] categoryList);

Creates an instance with CategoryFiltering set to Enum.Filtering.ON, CategoryMembership set to INCLUDE _CATEGORIES, taxonomyID set to the provided integer, and categoryList

set to the argument provided. Use this method to recommend categories in the taxonomy with the specified ID that belong to the subtrees of the categories in the categoryList argument.

The following methods return the FilteringSettings attributes:

getTaxonomyID : int

Returns the integer that identifies the taxonomy to which the categories belong.

getCategoryFiltering : FilteringType

Returns the value of Category Filtering. The options are

- Enum.Filtering.ON — perform category filtering
- Enum.Filtering.OFF— do not perform category filtering

getCategoryList : long[]

Returns the array of 64-bit integers that indicates the list of categories to be used as a filter for recommending categories or items.

getCategoryMembership : CategoryMembershipType

Returns a value that specifies how the categories in the category list should be applied for filtering. Options are

- Enum.CategoryMembership.EXCLUDE_ITEMS — exclude items that belong to categories in the category list
- Enum.CategoryMembership.EXCLUDE_CATEGORIES — exclude categories in the category list
- Enum.CategoryMembership.INCLUDE_ITEMS — include items that belong to categories in the category list
- Enum.CategoryMembership.INCLUDE_CATEGORIES — include categories in the category list
- Enum.CategoryMembership.LEVEL — recommend categories from the same level as the category in the list
- Enum.CategoryMembership.SUBTREE_ITEMS — recommend items that belong to levels in the category list below those of the categories in the category list
- Enum.CategoryMembership.SUBTREE_CATEGORIES — recommend categories that belong to the subtrees in the category list below the categories in the category list.
- Enum.CategoryMembership.ALL_ITEMS — include all items in the specified taxonomy

- Enum.CategoryMembership.ALL_CATEGORIES — include all categories in the specified taxonomy

Usage Notes

Not all filtering settings can be used will all methods. In particular, the following filtering setting cannot be used with the cross-sell methods (recommendCrossSellForItem, recommendCrossSellForItems, crossSellForItemFromHotPicks, and crossSellForItemsFromHotPicks):

- setCategoryLevelFiltering
- setCategorySubtreeFiltering
- setCategoryExclusion
- setCategoryFiltering(int)
- setCategoryFiltering(int, long[])

IdentificationData

Identifies the user and / or the session.

Attributes

No public attributes.

Methods

There are two kinds of methods provided with this class:

- Methods that create IdentificationData instances
- Methods that return attribute values

The following methods create IdentificationData instances. For a description of an attribute, see the description of the method that returns that attribute value.

createSessionful(String appSessionID, UserType userType)

Create an IdentificationData instance for use in REAPI calls from a sessionful Web application.

createSessionless(String appSessionID, UserType userType)

Create an IdentificationData instance for use in REAPI calls from a sessionless application.

The following methods get attributes:

getUserID()

Returns the identifier of type String that the calling Web application assigns to the user

getAppSessionID() :

Returns the identifier of type String that the calling Web application assigns to the session

getUserType()

Returns a value of type UserType, either Enum.UserType.CUSTOMER or Enum.UserType.VISITOR. This attribute is used to identify the user as a customer (registered user) or as a visitor (unregistered user)

Usage Notes

The calling Web application should assign a userID to all users, both customers (registered users) and visitors. IDs for customers must be unique. If IDs for visitors are not unique, OP will not be able to make recommendations that are specific to a given visitor; instead the same recommendations would be made for all visitors who had the given ID.

Item

This class is used to represent items that can be recommended and for which data can be collected. An item is uniquely represented by the combination of type and ID. Item IDs must be unique within a given type, but different types can have the same IDs.

Attributes

No public attributes.

Methods

There are three kinds of methods provided with this class:

- A constructor that creates an Item instance
- Methods that return attribute values
- A method that is a debugging aid

```
//Constructor
Item(String type,  long ID);
```

Creates an item instance with the specified type and ID. For descriptions of type and ID, see the descriptions of the methods that return them.

The following methods return attributes:

getType()

Returns a value of type String representing the group to which an item belongs. For example, a Web site might have two types of items: products and banner ads. Individual products will have unique IDs, as will individual banner ads.

getID (): long

Returns the unique identifier (type long) for an item within a given type.

Usage Note

Different items in a given type must have different IDs.

ItemDetailData

This class is created internally by OP as part of the result of recommendation request. The calling Web application will have to examine the attributes to determine what attributes and values they contain. See the description of Recommendation later in this chapter for more details.

Attributes

No public attributes.

Methods

getAttribute() : RecommendationAttributeType

Returns the recommendation attribute. Attributes can be any of the following:

- Enum.RecommendationAttribute.TYPE
- Enum.RecommendationAttribute.ID
- Enum.RecommendationAttribute.PREDICTION

getValue()

Returns the value (type String) of the item attribute.

toString()

Converts to String, so that it can be printed.

Recommendation

This class encapsulates information about a single recommended item. The information about the item is stored in the attributes array.

Attribute

No public attributes.

Methods

getAttributes()

Returns an array of type ItemDetailData[] for the recommendation. Each instance stores information about a different item attribute. Item attributes are itemID, item type, and prediction value. For release 1.0 of OP, the set of item attributes is fixed and is precisely the set of attributes listed.

toString()

Converts to String, so that it can be printed.

Usage Note

You should use this method to access information about recommended items and not try to access the array element directly.

RecommendationContent

Specifies the type of information that a recommendation request should return.

Attributes

No public attributes.

Methods

There are two kinds of methods provided with this class:

- Two constructors that create RecommendationContent instances depending on how sorting is to be done
- A method that returns the content items

The following constructors create recommendation content instances depending on how sorting is to be done:

```
//Constructor - default sorting
RecommendationContent(SortingType sorting);
```

This constructor creates a default RecommendationContent instance with default sorting. The default instance has the following three entries:

- ContentItem[0].ContentAttribute is Enum.RecommendationAttribute.TYPE and ContentItem[0].sorting is Enum.Sorting.NONE.
- ContentItem[1].ContentAttribute is Enum.RecommendationAttribute.ID and ContentItem[1].sorting is Enum.Sorting.NONE.
- ContentItem[2].ContentAttribute is Enum.RecommendationAttribute.PREDICTION and ContentItem[2].sorting is the provided sorting type.

```
//Constructor - specified sorting
RecommendationContent(SortingType sortingForType, SortingType sortingForID, SortingType
sortingforPrediction);
```

This constructor creates a default RecommendationContent instance with sorting as specified in the parameters. The default instance has the following three entries:

- ContentItem[0].ContentAttribute is Enum.RecommendationAttribute.TYPE and ContentItem[0].sorting is sortingForType.
- ContentItem[1].ContentAttribute is Enum.RecommendationAttribute.ID and ContentItem[1].sorting is sortingForID

- ContentItem[2].ContentAttribute is Enum.RecommendationAttribute.PREDICTION and ContentItem[2].sorting is sortingForPrediction.

The following method returns the content items

getContentItems : ContentItem[]

Returns the array of type ContentItem[]. Each entry specifies the type of information to be included in the result object of a recommendation request call. The information includes the attribute name and how or if it should be sorted. See "Usage Notes" (below) for information about sorting multiple array instances.

Usage Notes

If multiple instances of the array are to be sorted, the sorting order follows the array index order. That is, the result is sorted according to the attribute in the first array entry marked to be sorted, followed by the attribute in the second entry marked to be sorted, etc.

RecommendationList

A collection of recommendations for a specific InterestDimension. RecommendationList is the class returned by all REAPI methods that return recommendations.

Attributes

No public attributes.

Methods

The methods described below permit the calling Web application to determine the interest dimension type, to determine the actual number of recommendations returned, and to get the individual recommendations. See the description of the Recommendation class for more information.

getInterestDimension()

Returns the interest dimension of type InterestDimension, which specifies the interest dimension that the items were ranked against. Possible values are

- Enum.InterestDimesion.NAVIGATION — items were ranked as those that the customer was interested in
- Enum.InterestDimesion.PURCHASING — items were ranked as ones that the customer purchased
- Enum.InterestDimesion.RATING — items were ranked as ones that the customer liked

getNumberOfRecommendations()

The actual number (type int) of recommendations returned.

getRecommendation(int index)

Returns the recommendation instance (type Recommendation) at position index in the Recommendation array. If index is out of range, the method returns an exception.

toString()

Converts to String, so that it can be printed.

TuningSettings

Specifies settings to be applied when computing a recommendation. An instance of this class is passed to all recommendation requests.

Attributes

No public attributes.

Methods

There are two kinds of methods provided with this class:

A constructor that creates an TuningSettings instance

Methods that set attribute values

Methods that return attribute values

The following constructor creates a TuningSettings instance:

```
//Constructor
TuningSettings(DataSourceType dataSource,
    InterestDimensionType   interestDimension,
    PersonalizationIndexType  personalizationIndex,
    ProfileDataBalanceType   profileDataBalance,
   ProfileUsageType  profileUsage);
```

Create a tuning settings instance for use in recommendation requests. For descriptions of the attributes, see the descriptions of the methods that set attribute values.

The following methods set attribute values:

setDataSourceType();

Sets the type of data to consider when computing recommendations. The options are

- Enum.DataSource.NAVIGATIONAL — use navigational (click stream) data only.
- Enum.DataSource.PURCHASING — use purchasing data only.
- Enum.DataSource.RATING — use rating data only.
- Enum.DataSource.DEMOGRAPHIC — use demographic data only.

- Enum.DataSource.ALL — use all data (navigational, purchasing, rating, and demographics).

setInterestDimension();

Sets the interest dimension that items should be ranked against. The options are

- Enum.InterestDimension.RATING — rank items according to the expected rating that the customer would assign to them. This interest dimension is useful in applications that collect rating data on items (that is, explicit preference or interest).
- Enum.InterestDimension.PURCHASING — rank items according to the likelihood that the customer will buy them. This interest dimension is useful in situations where selling items is the main goal.
- Enum.InterestDimension.NAVIGATION — rank items according to the likelihood that the customer would be interested in the items. This interest dimension is useful in situations where the intent is measured indirectly using navigational data (implicit preferences or interest). For example, how interested a customer would be in an article about a given subject.

setPersonalizationIndex();

Sets the attribute that specifies how "unusual" the recommendations returned will be. The options are

- Enum.PersonalizationIndex.LOW — recommend obvious items for a given customer profile; "Best Sellers" will appear more frequently among the recommendations in this case
- Enum.PersonalizationIndex.MEDIUM — recommend a balanced mixture of obvious and less obvious items for a given customer profile
- Enum.PersonalizationIndex.HIGH — recommend less obvious items for a given customer profile

setProfileDataBalance();

Sets the attribute that indicates whether to take data from current session or from history when making recommendations. The options are

- Enum.ProfileDataBalance.HISTORY — use historical data only for creating recommendation
- Enum.ProfileDataBalance.BALANCED — use a balanced mixture of historical data and current session data for creating recommendations

- Enum.ProfileDataBalance.CURRENT — use current session data only for creating recommendation

setProfileUsage();

Sets the attribute that specifies if the recommendation list can include items in a customer's profile. Options are

- Enum.ProfileUsage.INCLUDE — include items in the profile
- Enum.ProfileUsage.EXCLUDE — do not include items in the profile

There are times when it is appropriate to exclude items in the profile; for example, you might not want to recommend that a customer buy books that he has already purchased.

The following methods return attribute values:

getDataSourceType();

Returns the type of data to consider when computing recommendations. The options are

- Enum.DataSource.NAVIGATIONAL — use navigational (click stream) data only.
- Enum.DataSource.PURCHASING — use purchasing data only.
- Enum.DataSource.RATING — use rating data only.
- Enum.DataSource.DEMOGRAPHIC — use demographic data only.
- Enum.DataSource.ALL — use all data (navigational, purchasing, rating, and demographics).

getInterestDimension();

Returns the interest dimension that items should be ranked against. The options are

- Enum.InterestDimension.RATING — rank items according to the expected rating that the customer would assign to them. This interest dimension is useful in applications that collect rating data on items (that is, explicit preference or interest).
- Enum.InterestDimension.PURCHASING — rank items according to the likelihood that the customer will buy them. This interest dimension is useful in situations where selling items is the main goal.
- Enum.InterestDimension.NAVIGATION — rank items according to the likelihood that the customer would be interested in the items. This interest dimension is useful in situations where the intent is measured indirectly using navigational data

(implicit preferences or interest). For example, how interested a customer would be in an article about a given subject.

getPersonalizationIndex();

Returns the attribute that specifies how "unusual" the recommendations returned will be. The options are

- Enum.PersonalizationIndex.LOW — recommend obvious items for a given customer profile; "Best Sellers" will appear more frequently among the recommendations in this case
- Enum.PersonalizationIndex.MEDIUM — recommend a balanced mixture of obvious and less obvious items for a given customer profile
- Enum.PersonalizationIndex.HIGH — recommend less obvious items for a given customer profile

getProfileDataBalance();

Returns the attribute that indicates whether to take data from current session or from history when making recommendations. The options are

- Enum.ProfileDataBalance.HISTORY — use historical data only for creating recommendation
- Enum.ProfileDataBalance.BALANCED — use a balanced mixture of historical data and current session data for creating recommendations
- Enum.ProfileDataBalance.CURRENT — use current session data only for creating recommendation

getProfileUsage();

Returns the attribute that specifies if the recommendation list can include items in a customer's profile. Options are

- Enum.ProfileUsage.INCLUDE — include items in the profile
- Enum.ProfileUsage.EXCLUDE — do not include items in the profile

There are times when it is appropriate to exclude items in the profile; for example, you might not want to recommend that a customer buy books that he has already purchased.

3

Using the Recommendation Engine Proxy

This chapter consists of an overview of the methods that are used to manage the recommendation engine proxy, to collect data, and to obtain recommendations, followed by the individual methods listed in alphabetical order. The supporting classes for these methods are described in Chapter 2.

For examples of how to uses these classes and methods, see Appendix A and the complete example in Appendix B.

All of these methods return results in real time. Usually they return recommendations for a single user.

All methods described in this chapter are public.

Overview

The recommendation proxy (REProxyRT) methods can be divided according to function, as follows:

- Proxy creation and management (the proxy manager and related methods)
- Session management (create and close)
- Data collection (collect, preprocess, and store data in recommendation engine (RE) tables)
- Recommendations (obtain various types of recommendations)

Location of Classes

To use the REProxyRT (and its exceptions), you must include the following statements in your Java program:

```
import oracle.dmt.op.re.reapi.rt.*;
import oracle.dmt.op.re.reexception.*;
```

All of these classes reside on the system where Oracle9*i*AS is installed.

Proxy Creation and Management

REProxyManager handles a pool of REProxyRT instances. Using multiple REProxyRT instances within a JServ provides the following benefits:

- Fault tolerance (if one instance fails, there is another to use)
- Load distribution (the load can be spread among all proxy instances)
- Domain-dependent recommendations (each proxy instance is associated with a specific RE)

Multiple proxy instances can result in the following issues:

- Collected data may be lost when an instance of the proxy fails and the application shifts to another instance.
- A given customer must be connected to the same RE for all transactions during a session.

The REProxyManager class also includes a caching mechanism that supports data collection in the recommendation engine.

RE Data Collection

The REProxyRT class includes the DataCollection Cache, which supports data collection in the RE. Every time you create an REProxyRT object, the cache is built as a subcomponent of the proxy object. When data is collected using the REAPI calls addItem() and addItems(), the data is stored in the cache (in the memory) and is periodically flushed to RE schema. This "batch save" improves RE performance. The cache is created when a new REProxyRT object is created. The refresh rate is defined by an input parameter to REProxyManager.createProxy().

Currently, only item and user ID data in the classes DataItem and IdentificationData are cached, and they are cached as current session data.

REProxyManager Class

REProxyManager is a singleton implementation, that is, only one instance of the REProxyManager class is created in a particular JVM instance, and the class is loaded automatically.

The REProxyManager class is used to create and manage the instances of REProxyRT. REProxyManager has only static public methods. REProxyManager does not have a public constructor and hence cannot be created by the user. REProxyManager maintains an REProxyRT pool and uses proxy names to reference to individual REProxyRT objects.

The following methods manage REProxyRT objects:

- destroyAllProxies
- destroyProxy
- createProxy
- getProxy

For examples of how to use the proxy manager, see Appendix A and the complete example in Appendix B.

Session Management

The following methods manage sessions:

- createCustomerSession
- createVisitorSession
- closeSession
- setVisitorToCustomer

Data Collection and Management

The following methods collect, preprocess, and store data in RE tables. The collected data can be persisted by setting appropriate configuration parameters:

- addItem
- addItems
- removeItem
- removeItems

Recommendations

The following methods obtain and manage recommendations:

- rateItem
- rateItems

- recommendTopItems
- recommendBottomItems
- recommendFromHotPicks
- recommendCrossSellForItem
- recommendCrossSellForItems
- crossSellForItemFromHotPicks
- crossSellForItemsFromHotPicks
- selectFromHotPicks

Communicating the returned recommendations to the end user is the responsibility of the calling Web application. The calling Web application must also decide which recommendations to pass to the user. For example, the Web application may want to check that an item is in stock before recommending the item.

Ratings in OP

Ratings in OP are in "ascending order of goodness", that is, the higher the rating, the more the user prefers the item. Low rated items are items that the user does not prefer. OP algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

Meaning of Returned Value for Recommendations

The meaning of the value returned for recommendation instances where ItemDetailData.attribute is equal to Enum.RecommendationAttribute.PREDICTION depends on the value of interestDimension as follows:

- For InterestDimension.RATING, the expected rating for the item is returned.
- For InterestDimension.PURCHASING or InterestDimension.NAVIGATION, the ranking is returned. The most probable item is assigned a value of 1 and other items are assigned integer values representing their rank according to how probable the items is.

Rules and Recommendations

OP uses rule tables stored in the RE to generate the recommendations requested by the recommendation methods. The rule tables are created when a package is built and stored in the RE, that is, when the package is deployed. The specific rule table used depends upon the REAPI call made. In general, the antecedents of the rules are matched against the data in cache (both historical and current session data) and

the probabilities of the various consequents are computed. These items are then ordered by probability, and numberOfItems (an API argument) items are returned.

Method Details

The rest of this chapter contains detailed descriptions of the methods, which are listed in alphabetical order.

addItem

A data collection method that adds the specified item to the profile of the user or the session specified by idData. item specifies the data source type, item type, item ID, and value of the item.

Syntax

addItem(IdentificationData idData, DataItem item)

Arguments

idData
Identifies a user or a session. item is added to this user's profile or this session.

item
Item, of type DataItem, to be added to the specified user's profile.

Exception

BufferIsFullException

Usage Notes

Items are cached locally at the REAPI and synchronously written to the RE; the frequency of the writes is specified as a configuration parameter when OP is installed. It is important that the data synchronization interval is frequent enough to support the Web applications' requirements. For more information about data synchronization, see the administration guide.

When an application needs to add several items at a time, it can either use several addItem calls or one addItems call. When using addItems, the application must maintain the details of the items to be added until the call is made; in other words, the application needs to keep the state. It may be simpler to issue several addItem calls.

addItem and addItems are asynchronous, so the calling application does not need to wait until either call saves the data to the database.

addItems

A data collection method that adds one or more items to the profile of the user or the session specified by idData. item[i] specifies the data source type, item type, item ID, and value of the i^{th} item. The items are stored in an array.

Syntax

addItems(IdentificationData idData, DataItem[] items)

Arguments

idData
Identifies a user or a session. The items specified by items are added to this user's profile or this session.

items
The items, an array of type DataItem, to be added to the specified user's profile or the specified session.

Exceptions

BufferIsFullException

Usage Notes

Items are cached locally first and synchronously written to the RE; the frequency of the writes is specified as a configuration parameter when OP is installed. It is important that the data synchronization interval is frequent enough to support the Web applications' requirements. For more information about data synchronization, see the administration guide.

When an application needs to add several items at a time, it can either use several addItem calls on one addItems call. When using addItems, the application must maintain the details of the items to be added until the call is made; in other words, the application needs to keep the state. It may be simpler to issue several addItem calls.

addItem and addItems are asynchronous, so the calling application does not need to wait until either call saves the data to the database.

closeSession

Closes a session and marks the session as closed. Session data corresponding to closed sessions is either purged or archived by a data synchronization process that runs periodically. Archiving or purging and the frequency of synchronization are specified as configuration parameters of the recommendation engine schema (RE schema). For more information about data synchronization, see the administration guide.

If the calling Web application does not call this method explicitly, the session times out after a specified time interval. The time-out interval is specified as a configuration parameter of the RE schema.

The application can also configure the RE to not time out any sessions. In this case, the calling Web application must close all sessions explicitly using this method. If sessions are not closed (and do not time out), the RE tables may fill up rapidly. When the tables are large, recommendation performance is poor; furthermore, disk space may be exhausted.

See the administration guide for information about setting configuration parameters.

Syntax

```
closeSession(IdentificationData idData);
```

Arguments

idData
Identifies the session to close.

Exceptions

REProxyInitException is raised if the session corresponding to the idData does not exist.

BadDBConnectionException

ConnectionPoolIsFullException

InvalidIDException

ErrorExecutingRE

Usage Notes

This method is used by sessionful Web applications only.

createCustomerSession

Creates a new session for the specified customer (registered user). This method is used by sessionful applications that keep track of session state for existing customers.

OP maintains the mapping between appSessionID and customerID. In subsequent calls, the calling application uses appSessionID to identify the customer session.

Syntax

createCustomerSession(String customerID, String appSessionID);

Arguments

customerID
Type String, assigned by the calling Web application

appSessionID
Type String, session ID assigned by the calling Web application or null if unknown

Exceptions

Throws InvalidIDException if customerID or appSessionID is null.

BadDBConnectionException

ErrorExecutingRE

REProxyInitException

StringTooLargeException

ConnectionPoolIsFullException

Example

```
createCustomerSession("100", "Session1");
```

Usage Notes

This method is used by sessionful Web applications only.

The calling Web application must provide session IDs that are unique among currently active sessions. If this method is invoked with a session ID that is currently active at the RE, an exception is thrown. However, a session ID can be

reused as long as that session ID is not already active at the RE. appSessionID is synchronized to the MTR by OP. (For more information about data synchronization, see the administration guide.) If the calling application requires session IDs to be unique, its the responsibility of the calling Web application to ensure that appSessionID values are also unique.

OP has no way to tell if customerID and appSessionID are valid values; it is the responsibility of the calling Web application to verify that these values are valid.

createProxy

Creates an object of type REProxyRT. A proxy connects to a specific RE database and logs in to the database.

The proxy object includes a DataCollection Cache, which caches data collected by addItem() and addItems() methods; collected data is periodically written to the recommendation engine (RE) at the interval specified in the interval argument.

Syntax

```
REProxyManager.createProxy(String  proxyName,
                String  dbURL,
                String  userName,
                 String passWord,
                 int cacheSize,
            int interval);
```

Arguments

proxyName
Type String, the unique name of the object being created

dbURL
Type String, the JDBC database URL, in the format *jdbcdriver:@hostname:portnumber:SID*, where *jdbcdriver* is either jdbc:oracle:thin or jdbc:oracle:oci8; hostname is the name of the database host machine (as specified by the DBA); portnumber is the port number of the database instance (see your DBA for this); SID is the service ID of the database instance (see your DBA for this). For example, "jdbc:oralce:thin:@dbserver1.mycompany.com:1521:MYDB1"

userName
Type String, the database login name

passWord
Type String, the database password

Note: dbURL, userName, and passWord combine to specify the RE schema that you plan to use. In other words, you must specify the user name and password for the RE schema.

cacheSize
Integer specifying the cache size used by the recommendation engine, in kilobytes; see "Usage Notes" for information about how to determine an appropriate cache size.

interval
Integer, specifying the cache archive interval, in milliseconds; if interval is set to 10000, data is archived every 10 seconds. See "Usage Notes" for information about how to determine an appropriate interval.

Return Value

An REProxyRT object named proxyName or null, if the call fails

Exceptions

REProxyInitException

StringTooLargeException

Usage Notes

Cache Size: There are several factors to consider when determining the cache size.

1. System resources: Since cache takes memory space, you must make sure that you have enough memory to do what you want
2. Archive interval: the longer the interval, the larger the cache size
3. Maximum VArray size: The PL/SQL procedure that performs the archive uses VArrays and the maximum size is currently set at 5000. The archive can handle more than 5000 items, but the performance is much worse. Therefore, it is not recommended to have the cache buffer larger than 5000. Each data item stored in the cache takes up about 340 bytes; so the maximum VArray size translates to 3.3 MBytes (the actual cache buffer size is half of that since the cache has two buffers)
4. Data collection rate, the most important factor: If the data collection rate is no more than 100 items per second and the archive interval is 20 seconds, then a reasonable

cache size is 100 * 340 * 1.5 * 20, which is approximately 2000 Kilobytes. (This calculation assumes a safety factor of 1.5 to ensure that no data is dropped.)

It takes experimentation to determine an optimum interval coupled with an appropriate cache size.

Interval: The interval determines how often the collected data is archived (flushed from the memory to RE schema). There are several factors to consider when determining the setting:

1. Data collection volume and speed: The more frequent the data collection and the larger the volume of data collected, the shorter interval should be
2. Cache size: The smaller the cache, the shorter the interval.
3. Use of current session data: if you want to use the current session data to improve the recommendation accuracy, the data should not be held in the cache for too long. If the volume and speed of the data collection is not a problem, an interval of 10 - 30 seconds may be fine.

It takes experimentation to determine an optimum interval coupled with an appropriate cache size.

Example

See Appendix A for examples of use.

createVisitorSession

Creates a new session for a visitor. A visitor is an end user unknown to this site or application (that is, a visitor is not a registered user of the site). The calling Web application may assign specific temporary IDs to these unregistered users in order to track them. This method is used by sessionful applications that keep track of session state for visitors. It is not required for sessionless Web applications.

OP maintains the mapping between appSessionID and visitorID. In subsequent calls, the calling application uses appSessionID to identify the visitor session.

Syntax

createVisitorSession(String visitorID, String appSessionID)

Arguments

visitorID
Type String, the ID assigned by the calling Web application

appSessionID
Type String, the session ID assigned by the calling Web application

Exceptions

Throws an InvalidIDException if visitorID or appSessionID is null.

BadDBConnectionException

ErrorExecutingRE

REProxyInitException

StringTooLargeException

ConnectionPoolsIsFullException

Example

```
createVisitorSession("101", "Session2");
```

Usage Notes

OP has no way to tell if visitorID and appSessionID are valid values; it is the responsibility of the calling Web application to verify that these values are valid.

The calling Web application must provide session IDs that are unique among currently active sessions. If this method is invoked with a session ID that is currently active at the RE, an exception is thrown. However, a session ID can be reused as long as that session ID is not already active at the RE. appSessionID is synchronized to the MTR by OP. (For more information about data synchronization, see the administration guide.) If the calling application requires session IDs to be unique, it should check that they are unique at the MTR.

crossSellForItemFromHotPicks

Returns cross-sell recommendations for a specified item from the items in the specified hot picks groups. It answers questions such as: Which N items from the specified hot picks groups are the user most likely to buy, assuming that the user buys the specified item?

Syntax

```
crossSellForItemFromHotPicks (IdentificationData idData,
    Item: item,
    int numOfItems,
    long[] hotPickGroups,
    TuningSettings tunings,
    FilteringSettings  filters,
    RecommendationContent recContent);
```

Arguments

item
The item for which cross-sell recommendations are required

numberOfItems
Type int, the number of items from the hot picks group to be returned. This number represents the maximum number of items to be returned; the actual number returned may be less.

hotPickGroups
Type long[], an array specifying the hot pick group IDs which identify the hot pick groups used to make recommendations

tuningSettings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use one of the following public methods to set these values: setItemFiltering(),

setItemExclusion(), or setItemSubTreeFiltering(). For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The list of cross-sell recommendations (type RecommendationList).

Interpreting the recommendations depends on the value of personalization index:

- For PersonalizationIndex.LOW, the return value is the rule confidence, which is equal P(antecedent, consequent) / P(antecendent)
- For PersonalizationIndex.MEDIUM, the return value is the weighted lift, computed as (confidence) / (0.5 +0.5*Prob(consquent))
- For PersonalizationIndex.HIGH, the return value is the lift, computed as (confidence) / Prob(consequent)

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

BadDBConnectionException

ClassNotFoundException

ArrayTooLargeException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

Usage Notes

Interest dimension must be the same as that of the data source type of the input item.

Data source type must be either navigational or purchasing. No other types are supported.

The following filtering setting *cannot* be used with this method:

- setCategoryLevelFiltering
- setCategorySubtreeFiltering
- setCategoryExclusion
- setCategoryFiltering(int)
- setCategoryFiltering(int, long[])

crossSellForItemsFromHotPicks

Returns cross-sell recommendations for specified items from the items in a specified hot picks group. It answers questions such as: Which are the N items from the specified hot picks group is the user is likely to buy, assuming that the user buys at least one of the specified items?

Syntax

```
crossSellForItemsFromHotPicks (IdentificationData idData,
    Item:[] items,
    int numOfItems,
    long[] hotPickGroups,
    TuningSettings tunings,
    FilteringSettings  filters,
    RecommendationContent recContent);
```

Arguments

items
The items for which cross-sell recommendations are required

numberOfItems
Type int, the number of items from the hot picks group to be returned. This number represents the maximum number of items to be returned; the actual number returned may be less.

hotPickGroups
Type long[], an array specifying the hot pick groups from which to make recommendations

tunings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use one of the following public methods to set these values: setItemFiltering(),

setItemExclusion(), or setItemSubTreeFiltering(). For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The list of cross-sell recommendations (type RecommendationList).

Interpreting the recommendations depends on the value of personalization index:

- For PersonalizationIndex.LOW, the return value is the rule confidence, which is equal P(antecedent, consequent) / P(antecedent)
- For PersonalizationIndex.MEDIUM, the return value is the weighted lift, computed as (confidence) / (0.5 +0.5*Prob(consquent))
- For PersonalizationIndex.HIGH, the return value is the lift, computed as (confidence) / Prob(consequent)

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

ArrayTooLargeException

ConnectionPoolIsFullException

InvalidIDException

BadDBConnectionException

ClassNotFoundException

ErrorExecutingRE

ReProxyInitException

Usage Notes

Interest dimension must be the same as that of the data source type of the input item.

Data source type must be either navigational or purchasing. No other types are supported.

The following filtering setting *cannot* be used with this method:

- setCategoryLevelFiltering
- setCategorySubtreeFiltering
- setCategoryExclusion
- setCategoryFiltering(int)
- setCategoryFiltering(int, long[])

destroyAllProxies

Destroy all REProxyRT objects in the pool of REProxyRT objects.

Syntax

```
REProxyManager.destroyAllProxies();
```

Arguments

None.

Return Value

None.

Exceptions

None.

Example

See Appendix A and the complete example in Appendix B for examples of use.

Usage Note

This method destroys all proxies in the pool. If you want to destroy one or more specific proxies, use destroyProxy().

destroyProxy

Destroy the specified REProxyRT object in the pool of REProxyRT objects.

Syntax

REProxyManager.destroyProxy(String proxyName);

Arguments

proxyName
Type String, the name of the REProxyRT object to be destroyed from the pool of REProxyRT objects

Return Value

None.

Exceptions

None.

Usage Note

If you want to destroy all proxies in the pool, use destroyAllProxy().

getProxy

Get an instance of the specified REProxyRT object.

Syntax

REProxyManager.getProxy(String proxyName);

Arguments

proxyName
Type String, the name of the REProxyRT object instance

Return Value

An instance of the named REProxyRT object or null if the specified object does not exist

Exceptions

REProxyInitException

Example

See Appendix A and the complete example in Appendix B for examples of use.

rateItem

Returns the degree of interest, computed along an interest dimension, for an item. The degree of interest could indicate, for example, the likelihood that customer X will buy item Y.

Syntax

```
rateItem(IdentificationData idData,
    Item  item,
    int taxonomyID,
    TuningSettings tunings,
    RecommendationContent recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session

item
Type Item, identifies the item to be rated

tunings
Specifies settings to be applied when computing a recommendation; for details, see "TuningSettings" in Chapter 2

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The degree of interest (type Float).

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

rateItems

Returns the degrees of interest, computed along an interest dimension, for an array of items. The i^{th} degree of interest could indicate, for example, the likelihood that customer X will buy the i^{th} item.

Syntax

```
rateItems(IdentificationData idData,
    Item[]  items
    int taxonomyID,
    TuningSettings tunings,
    RecommendationContent recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session

items
The items to be rated, an array of Item

tunings

Specifies settings to be applied when computing a recommendation; for details, see "TuningSettings" in Chapter 2

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The degrees of interest, an array of values of type Float.

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

ClassNotFoundException

recommendBottomItems

Returns the degree of interest and other relevant item information for the numberOfItems items with the lowest degree of interest along the specified interest dimension. It answers questions such as: Which are the N items that person X is least likely to buy or like?

Syntax

```
recommendBottomItems(IdentificationData idData,
    int numberOfItems,
    TuningSettings: tunings,
    FilteringSettings: filters,
    RecommendationContent recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session.

numberOfItems
Type int, the number of cross-sell recommendations to be returned. This number represents the maximum number of recommendations to be returned; the actual number returned may be less.

tunings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use any of the "set" methods to set these values. For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

List of recommended items (type RecommendationList).

See "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

ClassNotFoundException

recommendCrossSellForItem

Returns the degree of interest and other relevant item information for the specified number of items most strongly associated with the target item in the argument list. It answers questions such as: Which are the N items a person is most likely to buy or be interested in, given that he bought or is interested in item Y?

Syntax

```
recommendCrossSellForItem (IdentifactionData idData,
    Item  item,
    int numberOfItems,,
    TuningSettings  tunings,
    FilteringSettings  filters,
    RecommendationContent  recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session

item
The item for which cross-sell recommendations are required

numberOfItems
Type int, the number of cross-sell recommendations to be returned. This number represents the maximum number of recommendations to be returned; the actual number returned may be less.

tunings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use one of the following public methods to set these values: setItemFiltering(), setItemExclusion(), or setItemSubTreeFiltering(). For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The list of cross-sell recommendations (type RecommendationList).

Interpreting the recommendations depends on the value of personalization index:

- For PersonalizationIndex.LOW, the return value is the rule confidence, which is equal P(antecedent, consequent) / P(antecedent)
- For PersonalizationIndex.MEDIUM, the return value is the weighted lift, computed as (confidence) / (0.5 +0.5*Prob(consquent))
- For PersonalizationIndex.HIGH, the return value is the lift, computed as (confidence) / Prob(consequent)

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Usage Notes

Interest dimension must be that same as that of the data source type.

Data source type must be either navigational or purchasing. No other types are supported.

The following filtering setting *cannot* be used with this method:

- setCategoryLevelFiltering
- setCategorySubtreeFiltering
- setCategoryExclusion
- setCategoryFiltering(int)
- setCategoryFiltering(int, long[])

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

ClassNotFoundException

recommendCrossSellForItems

Returns the degree of interest and other relevant item information for the N items most strongly associated with the specified items. It answers questions such as: Which are the N items a person is most likely to buy or be interested in, given that he bought or is interested in items Y_1, Y_2,..., Y_n?

Syntax

```
recommendCrossSellForItems (IdentifactionData idData,
    Item [] items,
    int numberOfItems,,
    TuningSettings  tunings,
    FilteringSettings  filters,
    RecommendationContent  recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session

items
The array of items for which cross-sell recommendations are required

numberOfItems
The number of cross-sell recommendations to be returned. This number represents the maximum number of recommendations to be returned; the actual number returned may be less

tunings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use one of the following public methods to set these values: setItemFiltering(), setItemExclusion(), or setItemSubTreeFiltering(). For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The list of cross-sell recommendations (type RecommendationList).

Interpreting the recommendations depends on the value of personalization index:

- For PersonalizationIndex.LOW, the return value is the rule confidence, which is equal P(antecedent, consequent) / P(antecedent)
- For PersonalizationIndex.MEDIUM, the return value is the weighted lift, computed as (confidence) / (0.5 +0.5*Prob(consquent))
- For PersonalizationIndex.HIGH, the return value is the lift, computed as (confidence) / Prob(consequent)

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ArrayTooLargeException

ReProxyInitException

ClassNotFoundException

Usage Notes

Interest dimension must be that same as that of the data source type.

Data source type must be either navigational or purchasing. No other types are supported.

The following filtering setting *cannot* be used with this method:

- setCategoryLevelFiltering

- setCategorySubtreeFiltering
- setCategoryExclusion
- setCategoryFiltering(int)
- setCategoryFiltering(int, long[])

recommendFromHotPicks

Returns the degree of interest and other relevant item information for the N items with the highest degree of interest along the specified interest dimension. Items are selected from the hot picks table. It answers questions such as: Which are the N items from a particular set of Hot Picks group is the specified user most likely to buy or like?

Syntax

```
recommendFromHotPicks(IdentificationData idData,
    int numberOfItems,
    long[} hotPickGroups,
    TuningSettings: tunings,
    FilteringSettings  filters,
    RecommendationContent recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session

numberOfItems
Type int, the number of items to be returned. This number represents the maximum number of items to be returned; the actual number returned may be less

hotPickGroups
Type long[], an array of hot pick group ID specifying the hot pick groups from which to make recommendations

tunings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use one of the following public methods to set these values: setItemFiltering(),

setItemExclusion(), or setItemSubTreeFiltering(). For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies the type of information that a recommendation request should return; for details, see "RecommendationContent" in Chapter 2

Return Value

The list of recommended items (type RecommendationList).

Interpreting the recommendations depends on the value of personalization index:

- For PersonalizationIndex.LOW, the return value is the rule confidence, which is equal P(antecedent, consequent) / P(antecedent)
- For PersonalizationIndex.MEDIUM, the return value is the weighted lift, computed as (confidence) / (0.5 +0.5*Prob(consquent))
- For PersonalizationIndex.HIGH, the return value is the lift, computed as (confidence) / Prob(consequent)

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

ClassNotFoundException

recommendTopItems

Returns the degree of interest and other relevant item information for the numberOfItems items with the highest degree of interest along the specified interest dimension. It answers questions such as: Which are the N items that person X is most likely to buy / like?

Syntax

```
recommendTopItems(IdentificationData, idData
    int numberOfItems,
    TuningSettings: tunings,
    FilteringSettings: filters,
    RecommendationContent recContent);
```

Arguments

idData
Type IdentificationData, identifies user or session

numberOfItems
Type int, the number of items to be returned. This number represents the maximum number of items to be returned; the actual number returned may be less

tunings
Specifies tuning parameters for recommendation, including data source type, personalization index, profile data balance, and interest dimension. Note that interest dimension must be the same as the data source type of the input item. For more information, see "TuningSettings" in Chapter 2.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use any of the "set" methods to specify the values. For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies the type of information that a recommendation request should return; for details, see RecommendationContent in Chapter 2

Return Value

The list of recommended items (type RecommendationList).

Interpreting the recommendations depends on the value of personalization index:

- For PersonalizationIndex.LOW, the return value is the rule confidence, which is equal P(antecedent, consequent) / P(antecedent)
- For PersonalizationIndex.MEDIUM, the return value is the weighted lift, computed as (confidence) / (0.5 +0.5*Prob(consquent))
- For PersonalizationIndex.HIGH, the return value is the lift, computed as (confidence) / Prob(consequent)

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

ClassNotFoundException

StringTooLargeException

removeItem

Removes the item from the specified session or from the profile of the specified user. This method removes an item that has not been written to the MTR (permanent storage). Data is written to the MTR after the session is closed or times out.

Syntax

removeItem(IdentificationData idData, DataItem item);

Arguments

idData
Identifies a user or a session; the specified item is removed from this user's profile or from this session

item
The item to be removed, of type DataItem

Exceptions

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

Usage Notes

This method cannot be used to remove items from a user profile that has been written to permanent storage.

If a specified item is not in the user's profile, OP does nothing.

removeItems

Removes the items from the specified session or from the profile of the specified user. This method removes items that have not been written to the MTR (permanent storage). Data is written to the MTR after the session is closed or times out.

Syntax

```
removeItems(IdentificationData idData, DataItem[] items);
```

Arguments

idData
Identifies a user or session; the items specified by items are removed from this session or from this user's profile

items
The items to be removed, an array of type DataItem.

Exceptions

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

Usage Notes

This method cannot be used to remove items from a user profile that has been written to permanent storage.

If a specified item is not in the user's profile, OP does nothing.

Example

See Appendix A and the complete example in Appendix B for examples of use.

selectFromHotPicks

Selects the specified number of items from a specified set of hot picks group.

Syntax

```
selectFromHotPicks (IdentificationData idData,
    int numberOfItems ,
    long[] hotPickGroups,
    TuningSettings  tunings,
    FilteringSettings  filters,
    RecommendationContent  recContent);
```

Arguments

idData
Identifies a user or session; the items specified by items are removed from this session or from this user's profile

numberOfItems
Type int, the number of items to be returned. This number represents the maximum number of items to be returned; the actual number returned may be less

hotPickGroups
Type long[], an array specifying the hot pick groups from which to make recommendations

tunings
For release 1 of OP, an empty object must be passed here.

filters
Specifies filtering parameters for the recommendations, including taxonomy ID, category filtering, category list (optional) and category membership. You can use of You can use any the "set" methods to set these values. For more information, see "FilteringSettings" in Chapter 2.

recContent
Specifies recommendation attributes and sorting; for details, see "RecommendationContent" in Chapter 2.

Return Value

The list of selected items (type RecommendationList).

For more information, see "Meaning of Returned Value for Recommendations", earlier in this chapter.

Exceptions

NullParameterException

InvalidIDException

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

ClassNotFoundException

ArrayTooLargeException

setVisitorToCustomer

Changes a visitor to a customer. This method is called when a visitor (unregistered user) registers during a session. This call associates the user ID with the customer ID generated by the Web application when the visitor becomes a registered customer. OP creates a new internal session for the customer. The session data for the visitor is transferred to the customer session. The historical data for the customer is also loaded to the customer session RE cache from permanent storage (MTR).

Syntax

setVisitorToCustomer (IdentificationData idData, String customerID);

Arguments

idData
Identifies the visitor or the session

customerID
The string that the calling Web application uses to identify the registered user; this is the ID created when the visitor registers with the Web application

Exceptions

BadDBConnectionException

ConnectionPoolIsFullException

ErrorExecutingRE

ReProxyInitException

Usage Notes

idData and customerID cannot be null.

A

REAPI Examples and Usage

This appendix provides examples of REAPI use. In some instances, we provide code snippets; in others, we describe an approach for performing certain kinds of tasks using OP.

REAPI Demo

OP includes REAPI Demo, a sample program that illustrates the use of many of the REAPI methods. This sample program can be used to learn about REAPI calls and can also used to verify that OP is correctly installed.

After you have installed OP, start REAPI Demo by opening the following URL in Netscape or Internet Explorer:

`http://server/redemo/`

where *server* is the name of the system where Oracle9*i*AS is installed. The REAPI test site is displayed.

To view the source code for the OP REAPI Demo, click "View Source Code."

For information about how to install and run the demo, see *Getting Started with Oracle9iAS Personalization*.

REAPI Basic Usage

The REProxy methods described in Chapter 3 permit you to instrument your Web site. To use REAPI calls, you must perform the following steps:

1. Get an REProxy object.

2. Use the proxy instance as required in REAPI calls. The outline that your program should follow depends on whether your Web application is sessionful or sessionless.
3. Destroy the proxy instance when you are done with it. You may also want to destroy all proxy objects.

Create an REProxy Object

This section illustrates basic REProxy usage; for more information about REProxy and other ways to use it see, "REProxyManager Interaction with JVM" and "Using Multiple Instances of REProxy", later in this appendix.

The following code fragment creates an object name `proxy`: You use this object to perform REAPI calls. Note that you must specify the username and password for the RE schema.

```
final String proxyName = "RE1";
final String dbURL = "jdbc.oracle.thin:@DBServer.myshop.com:1521:DB1";
final String user = "myself";
final String passWd = "secret";
final int cacheSize = 2048;        // 2 mbytes
final int interval = 10000;    // 10 seconds
REProxy proxy;
...

try {
  proxy = REProxyManager.createProxy(proxyName,
                                     dbURL,
                                     user,
                                     passWd,
                                     cacheSixe,
                                     interval);
  ...
} catch (Exception e) {
    // exception handling here
}
```

Use the Proxy

After you've created a REProxy object and gotten an instance of it, you use the proxy to specify REAPI calls, as, for example,

```
proxy.closeSession();
```

The sequence of calls depends on whether the application is sessionful or sessionless; see "Sessionful Web Application Outline" or "Sessionless Web Application Outline" later in this appendix for details.

Destroy the Proxy

When you are finished with the proxy instance, you should destroy it. Failure to destroy proxy instances can result in excess usage of system resources and even loss of data.

```
// after it's all done
REProxyManager.destroyProxy(proxyName);
```

Destroy All Proxy Objects

To destroy all proxy objects from the pool, use `destroyAllProxies()`.

Sessionful Web Application Outline

The following outlines the required steps in the required order for a sessionful Web application (an application that starts a session for each customer).

1. Create an REProxy object as described in "Create an REProxy Object", earlier in this appendix. You need to know the user name and password for the RE schema.
2. Create a customer session or a visitor session

   ```
   proxy.createCustomerSession(userID, appSessionID); //customer session

   proxy.createVisitorSession(userID, appSessionID); //visitor session
   ```

3. Get identification data.

   ```
   idData = IdentificationData.createSessionful(appSessionID);
   ```

4. Call REAPI methods: for example,

   ```
   /*Set input parameters.; see Chapter 2 for details. */
   int nRec=10;
   TuningSettings tune = new TuningSettings(Enum.DataSourec.NAVIGATION,
                          Enum.InterestDimension.NAVIGATION,
                          Enum.PersonalizationIndex.HIGH,
                          Enum.ProfileDataBalance.BALANCED,
   ```

```
                        Enum.ProfileUsage.EXCLUDE);
long [] catList = {1, 2, 3, 4};
FilteringSettings filters = new FilteringSettings();
filters.setItemFiltering(1, catList);
RecommendationContent rContent = new RecommendationContent (
                                    Enum.Sorting.ASCENDING);
/*Get a recommendation. */
try {
     RecommendationList rList = proxy.recommendTopItems(idData,
                                 nRec, tune, filters, rContent);
/* Parse the results and pass recommendations to the user*/
```

5. Make other REAPI calls as required.
6. Close the session.

   ```
   proxy.closeSession();
   ```

7. Destroy the proxy.

   ```
   REProxyManager.destroyProxy(proxy); //destroy proxy when done
   ```

Sessionless Web Application Outline

The following outlines the required steps in the required order for a sessionless Web application (an application that does not start a session for each customer). Note that sessionless applications close when they time out.

1. Create an REProxy object as described in "Create an REProxy Object", earlier in this appendix. You need to know the user name and password for the RE schema.
2. Get identification data.

   ```
   idData = IdentificationData.createSessionless(customerID);
   ```

3. Call REAPI methods: for example,

   ```
   /*Set input parameters; see Chapter 2 for details.*/
   int nRec=10;
   TuningSettings tune = new TuningSettings(Enum.DataSourec.NAVIGATION,
                        Enum.InterestDimension.NAVIGATION,
                        Enum.PersonalizationIndex.HIGH,
                        Enum.ProfileDataBalance.BALANCED,
                        Enum.ProfileUsage.EXCLUDE);
   long [] catList = {1, 2, 3, 4};
   ```

```
FilteringSettings filters = new FilteringSettings();
filters.setItemFiltering(1, catList);
RecommendationContent rContent = new RecommendationContent (
                                        Enum.Sortinh.ASCENDING);
/*Get a recommendation. */
try {
     RecommendationList rList = proxy.recommendTopItems(idData,
                                   nRec, tune, filters, rContent);
/* Parse the results and pass recommendations to the user*/
```

4. Make other REAPI calls as required.
5. Destroy the proxy.

```
REProxyManager.destroyProxy(proxy); //destroy proxy when done
```

REProxyManager Interaction with JVM

REProxyManager is a singleton implementation, that is, only one instance of the REProxyManager class is created in a give JVM instance and the class is automatically loaded in the JVM instance. This behavior has implications about how your program behaves. The behavior is different depending on whether your application is a standalone Java program or if it is a servelet.

Standalone Java Applications

Suppose that you create a standalone Java application using REAPI calls that you execute from the command line with a command such as

```
java myapplication.class
```

Such an application has the following characteristics:

- It runs in a separate JVM instance.
- The REProxyManager instance is automatically loaded into the JVM instance.
- After the application finishes executing, the JVM instance goes away.

If you do not destroy the proxy before the program exits, the REProxy objects remain in memory; they cannot be accessed because the JVM instance that created them no longer exists.

To avoid memory leaks, you must destroy the proxy before the program ends.

Servelets

If REAPI calls are used in a servelet, REProxyManager continues to exist as long as the JServ instance continues to exist; REProxy objects are not removed. To conserve resources, you should destroy proxy objects when you are done using them.

Using Multiple Instances of REProxy

REProxyManager manages a pool of one or more proxies. This section illustrates several ways to use multiple proxies:

- Initialization fail safe
- Ensuring that REAPI server is not interrupted
- Load balancing

Initialization Fail Safe

The following code fragment illustrates how you might use two RE to prevent utilization failure. This code assumes that the schema for normal recommendation service is named "RE"; if "RE" fails, you will use a backup RE schema, named "RE_BACKUP".

```
REProxy initProxy(...)
{
  REProxy proxy;

  // initialization
  try {
    if ((proxy = REProxyManager.getProxy("RE")) == null)
      proxy = REProxyManager.createProxy("RE",
                                         dbURL,
                                         username,
                                         passWd,
                                         cacheSize,
                                         interval);
  } catch (REProxyInitException rie) {
    proxy = REProxyManager.createProxy("RE_BACKUP",
                                         dbURL1,
                                         username1,
                                         passWd1,
                                         cacheSize,
                                         interval);
  }
```

```
    return proxy;
  }
```

Uninterrupted REAPI Service

The following code fragment illustrates how to guarantee that the recommendation service does not fail when the regular RE server fails. The code implements the class `NeverFail` for this purpose.

```
 class NeverFail() {
    REProxy re1;
    REProxy re2;

    void initProxies() {
      try {
        if ((re1 = REProxyManager.getProxy("RE1")) == null)
         String dbURL1="jdbc:oracle:thin:@db1.mycorp.com:1521:orc1";
         re1 = REProxyManager.createProxy("RE1",
                                           dbURL1,
                                          "user1",
                                          "pw1",
                                          2048,
                                          10000);
        if ((re2 = REProxyManager.getProxy("RE2")) == null)
          String dbURL2="jdbc:oracle:thin:@db2.mycorp.com:1521:orc2";
            re2 = REProxyManager.createProxy("RE2",
                                          dbURL2,
                                          "user2",
                                          "pw2",
                                          2048,
                                          10000);
      } catch (REProxyInitException rie) {
        // exception handling
      }
    }

    RecommendationList getRecommendation() {
      RecommendationList rList;

      // initialize input
      ....
      try {
        rList = re1.recommendTopItems(...);
      } catch (Exception e) {
```

```
      rList = re2.recommendTopItems(...);
      return rList;
    }
    return rList;
  }
}
```

Load Balancing

The following code fragment illustrates a simple way to do load balancing so that not all customers are handled by the same RE. This example assumes that customers with odd IDs are processed using RE1 and those with even IDs are processed using RE2, a different RE. To accomplish this, first create two different proxies `re1` and `re2`, and then code `getRecommendation()` as follows:

```
RecommendationList getRecommendation() {
    RecommendationList rList;

    // initialize input
    ....
    try {
      if ((idData.getUserID() % 2) == 1)
        rList = re1.recommendTopItems(...);
      else
        rList = re2.recommendTopItems(...);
    } catch (Exception e) {
      // exception handling
      ......
    }
    return rList;
  }
```

Extracting Individual Recommendations

Use the getAttributes method of the "Recommendation" class rather than attempting to extract the individual recommendations from the array.

Handling Multiple Currencies

OP stores currency data in the demographic table (for example, someone's income) as numbers; that is, OP does not store any kind of label. Both ten dollars (US) and ten pounds sterling (UK) are stored as "10".

There are several ways to ensure that currency data is interpreted correctly; the solution that you pick for your application depends on how your application uses currency data.

- Include a country code in customer demographics.

  This solution allows the country to be taken into account, but it does not closely associate the value with the country.

- Convert all currencies to a common currency such as Euros or United States dollars.

  This solution permits you to compare individual currency values in a meaningful way (10 pounds sterling is more than $10 US) but does not permit you preserve the difference between data such as a salary of $30,000 US in the US, versus the same $30,000 US salary in Brazil. You need such information if, for example, you want to recommend items to highly remunerated individuals in both the US and Brazil; the salary in US dollars of highly remunerated individuals will vary considerably from country to country.

  This approach requires that you preprocess the data outside of OP before OP creates recommendations.

- Bin currency values according to the mean to get relative values that can be compared across countries.

  This solution would permit you, for example, to determine the highly remunerated individuals for a given country, but it requires that you determine and maintain the bin boundaries appropriately.

  This approach requires that you preprocess the data outside of OP before OP creates recommendations.

Recommendation Engine Usage

Oracle9*i*AS Personalization requires at least one recommendation engine (RE) in at least one recommendation engine farm. In general, you will want to use more than one RE to get satisfactory recommendation performance. Most applications will use multiple REs on different machines and subsequently different database instances.

See "Load Balancing" earlier in this appendix for an example of how you might code one of these solutions.

Typically, for a given application, these REs will belong to the same RE farm. If a physical system has multiple processors, and the processors can be leveraged effectively by the database, the number of REs required for a given number of users can be reduced, perhaps even to one. See the administration guide for more information.

If your application has more than one RE available for use, it must determine which one to use. Here are three possible solutions:

1. A given user of the Web site (either a visitor or a customer) is always handled by the same JServ instance and that JServ instance is configured to use one RE at all times. The application must route users to "their" JServs and configure JServs to contact specific REs. The REProxy class takes configuration arguments to specify which RE to connect to. The application must determine how to get these configuration arguments, either from a jserv.properties file, or by being explicitly coded in the Web applications, or by some other means.
2. Allow any JServ to handle any customer. This requires that a customer be "hashed" to a specific RE. It is important that the same customer be routed to the same RE, at least within the session, since data is cached for the user's session in the RE.
3. Provide a fail-over mechanism in the application to allow a different RE to be contacted in the event the primary RE for a given customer cannot be contacted. This can be applied in addition to either solutions 1 or 2 above. In this case, the application specifies the primary RE and the backup RE (or the multiple backup REs) and controls the logic to switch between REs. The same user session may not always be routed to the same RE; however, the ability to get some kind of recommendation will be maintained. Note that it may not be necessary to implement such a solution, especially in a reasonably robust environment.

Using Demographic Data

The schema of the MTR_CUSTOMER table consists of 50 generic attributes that can be mapped to any column in the site database. In order to support all different data types, all attributes are of type VARCHAR. Therefore, the mapped columns should be converted to strings. In this release of OP, these mapped columns are treated as categorical or numeric only. If any of the mapped columns is a DATE attribute, it should be converted to a number using the TO_NUMBER function. The converted

values can then be binned just like any other attribute by specifying the bin boundaries.

There is binning for demographic data. The attributes that are binned can be of type boolean. In OP, the bin numbers are represented internally as integers, but the actual values are passed back to the calling applications. That is, the Web application passes in the actual values and gets back actual values.

Handling Time-Based Items

For certain items, such as airline tickets, the price depends on when the item is purchased. For example, an airline ticket for a Boston to London flight has one price if it purchased 6 months before the date of the flight and a different price if it is purchased two days before the date of the flight.

If the Web application assigns the same item ID to all tickets for the same trip, regardless of when they are purchased, then the items should have different item types, such as "6-month advance", "2-day advance", etc. Alternatively, the application could define taxonomies on the items and get recommendations on the categories.

If the application assigns different item IDs to the same flight purchased at different times (so that a ticket purchased 6 months before the flight has a different ID from a ticket for the same flight purchased 2 days before the flight), all tickets can have the same item type. In this case recommending item IDs may not be appropriate; therefore, the application should define a taxonomy and request recommendations on the categories.

B

Sample Program

This appendix contains `ProxyTest.java`, a sample Java program that illustrates using REAPI. You can find the source code for this example in TBS

Before you can execute this program, an appropriate model must be built and deployed to an RE. If no data is returned, it may indicate that the model is not sufficient for the data. The code is installed in `${ORACLE_HOME}/dmt/reapi/rt/` on the system where Oracle9*i*AS is installed.

> **Note:** REAPI is installed on the system where Oracle9*i*AS is installed. It is simplest to run this program on that system.

```
// Copyright (c) 2001 Oracle Corp

////////////////////////////////////////////////////////////////////
//
// Test program exercises the functionality of
// REAPI.
//
// Step 1 creates a unique session ID
// Step 2 creates a proxy instance
// Step 3 creates a session
// Step 4 creates settings data (IdentificationData, TuningSettings,
//        FilteringSettings, hotPick list, item list)
// Step 5 gets recomendations and ratings
// Step 6 closes session
// Step 7 destroys the proxy and flushes data cache
////////////////////////////////////////////////////////////////////

import java.math.BigDecimal;
import java.lang.Long;
```

```
import java.sql.*;
import java.io.IOException;
import java.io.StringWriter;
import java.io.PrintWriter;
import oracle.jdeveloper.cm.CMException;
import oracle.dmt.op.re.reapi.rt.*;
import oracle.dmt.op.re.reapi.batch.*;
import oracle.dmt.op.re.reexception.*;
import oracle.dmt.op.re.base.*;
import oracle.dmt.oputil.exceptions.MessageLogException;
import oracle.dmt.oputil.exceptions.StringTooLongException;

/**
 * Class ProxyTest
 * <P>
 * @author Oracle Corporation
 */
public class ProxyTest
{
  static boolean bVerbose;
  static final String SESSIONEXISTS = "";
  /**
   * Constructor
   */
  public ProxyTest()
  {
  }

  /**
   * main
   * @param args
   */
public static void main(String[] args) throws ClassNotFoundException
  {
    long lStart;
    long lTotal = 0;
    String sProxyName = "REP1";
   String sdbURL = "jdbc:oracle:thin:@server-name:1521:darw900"; // sdbURL must
be correct for your installation
    String sUser = "RE01";
    String sPass = "REPW";

    int cSize = 3000;   // Kbytes
    int interval = 10000;   // in millisec
    new ProxyTest();
```

```
    REProxyRT proxy;
    // Step 1: Create a unique Session ID.
    String custID = "1";
    java.util.Date tmp = new java.util.Date();
    Long tmpInt = new Long(tmp.getTime());
    String sessionID = tmpInt.toString();


    String trace = null;

    lStart = System.currentTimeMillis();
    try
    {
      // Step 2:  Get a proxy instance.
      if ((proxy = REProxyManager.getProxy(sProxyName)) == null)
        proxy = REProxyManager.createProxy(sProxyName, sdbURL, sUser, sPass,
cSize, interval);

      // Step 3: create OP session
      proxy.createCustomerSession(custID, sessionID);

      // Step 4:  create settings data
      IdentificationData idData =
            IdentificationData.createSessionful(
                sessionID,
                Enum.User.CUSTOMER);

      TuningSettings tunings = new TuningSettings(
        Enum.DataSource.NAVIGATION,
        Enum.InterestDimension.NAVIGATION,
        Enum.PersonalizationIndex.HIGH,
        Enum.ProfileDataBalance.BALANCED,
        Enum.ProfileUsage.EXCLUDE);

      long[] hotPickGroups = {1,2,3,4,5,6,7,10,11};

      long[] m_catList = {1,2,3,4,5};

      FilteringSettings filters =
        new FilteringSettings(1);
        int taxonomy=1;
      filters.setItemFiltering( taxonomy, m_catList);
      RecommendationContent recContent = new
RecommendationContent(Enum.Sorting.ASCENDING);
```

```
  try{

 //Create list of items for testing
  DataItem[] items = new DataItem[4];
  items[0] = new DataItem(
        "MOVIE",
        72,
        Enum.DataSource.RATING,
        "1.5");
 items[1] = new DataItem(
        "MOVIE",
        287,
        Enum.DataSource.RATING,
        "1.5");
 items[2] = new DataItem(
        "MOVIE",
        122,
        Enum.DataSource.RATING,
        "1.5");
 items[3] = new DataItem(
        "MOVIE",
        1300,
        Enum.DataSource.RATING,
        "1.5");
int count =  1;

// Step 5: Get recomendations and ratings and print them out.
// Note use of toString.
try{
System.out.println("\n################# " + count++ + " #################");
 //Call recommendBottonItems
 RecommendationList es1 = proxy.recommendBottomItems(
    idData,
    10,
    tunings,
    filters,
    recContent);
System.out.println("");
es1.toString();
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################# " + count++ + " #################");
```

```
//Call rateItems
RecommendationList es2 = proxy.rateItems(
    idData,
    items,
    1,
    tunings,
    recContent);
System.out.println("");
System.out.println(es2.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################# " + count++ + " #################");
//call selectFromHotPicks
RecommendationList es3 = proxy.selectFromHotPicks(
    idData,
    10,
    hotPickGroups,
    tunings,
    filters,
    recContent);
    System.out.println("");
System.out.println("");
System.out.println(es3.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}


try{
System.out.println("\n################# " + count++ + " #################");
//Call crossSellForItemFromHotPicks
RecommendationList es4 = proxy.crossSellForItemFromHotPicks(
    idData,
    items[0],
    10,
    hotPickGroups,
    tunings,
    filters,
    recContent);
System.out.println("");
System.out.println(es4.toString());
} catch(ErrorExecutingRE e) {
```

```
    e.printStackTrace();
}

try{
System.out.println("\n################ " + count++ + " ################");
//Call recommendCrossSellForItem
RecommendationList es5 = proxy.recommendCrossSellForItem(
    idData,
    items[0],
    10,
    tunings,
    filters,
    recContent);
System.out.println("");
System.out.println(es5.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################ " + count++ + " ################");
 RecommendationList  es6 = proxy.recommendCrossSellForItems(
    idData,
    items,
    10,
    tunings,
    filters,
    recContent);
System.out.println("");
System.out.println(es6.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################ " + count++ + " ################");
RecommendationList es7 = proxy.crossSellForItemsFromHotPicks(
    idData,
    items,
    10,
    hotPickGroups,
    tunings,
    filters,
    recContent);
System.out.println("");
```

```
System.out.println(es7.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}


try{
System.out.println("\n################# " + count++ + " #################");
float es9 = proxy.rateItem(
idData,
items[2],
1,
tunings,
recContent
);
System.out.println("");
System.out.println("Result for recomend item:  " + es9);
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################# " + count++ + " #################");
RecommendationList es10 = proxy.recommendFromHotPicks(
idData,
10,
hotPickGroups,
tunings,
filters,
recContent);
System.out.println("");
System.out.println(es10.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################# " + count++ + " #################");
RecommendationList es11 = proxy.recommendTopItems(
idData,
10,
tunings,
filters,
recContent);
System.out.println("");
```

```
    System.out.println(es11.toString());
    } catch(ErrorExecutingRE e) {
        e.printStackTrace();
    }

    } catch(BadDBConnectionException bdbe) {
        bdbe.printStackTrace();
    }catch (ClassNotFoundException exc) {
        exc.printStackTrace();
    }


    // Step 6: Close session
    proxy.closeSession(idData);

    // Step 7: Call destroy proxy (will flush data cache)
    REProxyManager.destroyProxy(sProxyName);

    } catch (ErrorExecutingRE se) {
      System.err.println(se);
    } catch (InvalidIDException iie) {
        System.err.println(iie);
    } catch(BadDBConnectionException bdbe) {
        bdbe.printStackTrace();
    } catch (Exception e) {
      System.err.println(e);
      e.printStackTrace();
    }
  }


}
```