

Oracle9iAS Personalization

Programmer's Guide

Release 2 (v9.0.2)

May 2002

Part No. A95245-02

ORACLE[®]

Copyright © 2001, 2002 Oracle Corporation. All rights reserved.

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent and other intellectual and industrial property laws. Reverse engineering, disassembly or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle9i is a trademark or registered trademark of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments	ix
Preface.....	xi
Intended Audience	xi
Documentation Accessibility	xi
Structure.....	xii
Where to Find More Information	xii
Conventions.....	xiii
1 Oracle9iAS Personalization Programming	
OP API Structure.....	1-1
Executing OP Programs	1-1
Javadoc for the OP APIs	1-2
Part I Recommendation Engine API	
2 REAPI Overview	
REAPI Prerequisites.....	2-1
REAPI Definitions and Concepts	2-2
REAPI End Users (Customers and Visitors).....	2-2
Web Applications and Sessions.....	2-2
REAPI Sessionful Web Applications	2-3
REAPI Sessionless Web Applications.....	2-3
REAPI Data Collection.....	2-3

REAPI Recommendations	2-4
REAPI Hot Picks	2-4
Before Using REAPI	2-4
REAPI Demo Program	2-5
Creating REProxyRT Objects	2-5
Starting an REAPI Session	2-6
Creating Instances of REAPI Supporting Classes	2-6
Collecting Data for REAPI Recommendations	2-7
OP Data Caching	2-7
Getting REAPI Recommendations	2-8
How REAPI Creates Recommendations	2-8
Scoring for Visitors:	2-8
Scoring for Customers:	2-9
Making REAPI Recommendations	2-9
Closing an REAPI Session	2-9
Removing REProxyRT Objects	2-9

3 REAPI Supporting Classes

Ratings in OP	3-1
Location of REAPI Classes	3-1
REAPI CategoryMembership Interface	3-3
REAPI DataSource Interface	3-3
REAPI Filtering Interface	3-4
REAPI InterestDimension Interface	3-5
REAPI PersonalizationIndex Interface	3-5
REAPI ProfileDataBalance Interface	3-6
REAPI ProfileUsage Interface	3-7
REAPI RecommendationAttribute Interface	3-7
REAPI Sorting Interface	3-8
REAPI User Interface	3-8
Other Supporting REAPI Classes	3-9
ContentItem Class	3-9
DataItem Class	3-10
FilteringSettings Class	3-10
IdentificationData Class	3-11

Item Class.....	3-12
ItemDetailData Class.....	3-12
Recommendation Class	3-12
RecommendationContent Class	3-13
RecommendationList Class.....	3-13
TuningSettings Class.....	3-13

4 Using REAPI

Recommendation Proxy Classes	4-1
Location of RE Proxy Classes	4-2
RE Proxy Creation and Management	4-2
RE Data Collection	4-2
REProxyManager Class	4-3
Proxy Methods	4-3
RE Proxy Session Management	4-3
RE Proxy Data Collection and Management	4-3
Re Proxy Customer Registration	4-4
RE Proxy Recommendations.....	4-4
Ratings in OP	4-5
Meaning of Returned Value for Recommendations.....	4-5
Rules and Recommendations	4-6
RE Proxy Method Usage Notes	4-6
Session Creation.....	4-6
Data Collection.....	4-6
Add Items	4-6
Remove Items.....	4-7
Proxy Creation	4-7
Cache Size.....	4-7
Interval	4-8
Cross Sell Methods	4-8
Proxy Destruction.....	4-9

5 REAPI Examples and Usage

REAPI Demo.....	5-1
REAPI Basic Usage	5-1

Create an REProxy Object.....	5-2
Use the Proxy.....	5-2
Destroy the Proxy	5-3
Sessionful Web Application Outline	5-3
Sessionless Web Application Outline	5-4
REProxyManager Interaction with JVM	5-5
Standalone Java Applications	5-5
Java Server-Side Modules.....	5-5
Using Multiple Instances of REProxy.....	5-6
Initialization Fail Safe.....	5-6
Uninterrupted REAPI Service.....	5-7
Load Balancing.....	5-8
Extracting Individual Recommendations	5-8
Handling Multiple Currencies	5-8
Recommendation Engine Usage	5-9
Using Demographic Data	5-10
Handling Time-Based Items	5-11

Part II Recommendation Engine Batch API

6 RE Batch API Overview

RE Batch API Prerequisites.....	6-1
RE Batch API Definitions and Concepts	6-2
RE Batch API End Users (Customers)	6-2
RE Batch API Recommendations	6-2
Using RE Batch API.....	6-2
Setting Up the RE Batch API Environment.....	6-2
Customer Profile Data	6-3
Deploy a Package to an RE.....	6-3
Sample RE Batch API Usage	6-3
Creating an REBatchProxy Object.....	6-4
Creating Instances of RE Batch API Objects	6-4
Converting Data for RE Batch API.....	6-4
Managing Customer Profiles for RE Batch API	6-4
Getting RE API Batch Recommendations	6-4

Ratings in OP	6-5
Creating Recommendations.....	6-5
Scoring:	6-5
Making RE Batch Recommendations	6-5
Removing the REBatchProxy Object.....	6-5

7 RE Batch API Supporting Classes

Ratings in OP	7-1
Location of RE Batch API Classes	7-2
EnumType Interfaces for RE Batch API	7-2
CategoryMembership Interface.....	7-3
DataSource Interface	7-4
InterestDimension Interface.....	7-4
PersonalizationIndex Interface	7-5
ProfileDataBalance Interface.....	7-6
ProfileUsage Interface.....	7-6
Sorting Interface.....	7-7
Other RE Batch API Supporting Classes	7-7
DataItem Class	7-8
FilteringSettings Class.....	7-8
Item Class.....	7-9
Location Class	7-10
TuningSettings Class.....	7-10

8 Using the Recommendation Engine Batch Proxy

REProxy Batch Overview	8-1
Location of REProxyBatch Classes	8-1
REProxyBatch Creation and Management	8-2
Customer Profile Management	8-2
REProxyBatch Recommendations.....	8-2
Ratings in OP	8-2
Meaning of Returned Value for Recommendations.....	8-2
Cross Sell Method Usage Notes	8-3
Recommendation Method Usage Notes	8-3
REProxyBatch Rules and Recommendations	8-4

9 REProxyBatch API Examples and Usage

REProxyBatch API Basic Usage	9-1
Code Sample: Recommend Top	9-2
Code Sample: Recommend Cross Sell	9-2
Recommendation Engine Usage	9-2
Handling Multiple Currencies	9-3
Using Demographic Data	9-4
Handling Time-Based Items	9-4

A REAPI Sample Program

B REProxyBatch Sample Program

RE Batch Sample Program Overview	B-1
RE Batch Sample Program Output.....	B-1
Executing the RE Batch Sample Program	B-1
RE Batch Sample Program Code	B-3
batchtest.txt.....	B-3
REBatchTest.java.....	B-5

Index

Send Us Your Comments

Oracle9iAS Personalization Programmer's Guide, Release 2 (v9.0.2)

Part No. A95245-02

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: 781-238-9893 Attn: Oracle9iAS Personalization Documentation
- Postal service:
Oracle Corporation
Oracle9iAS Personalization Documentation
10 Van de Graaff Drive
Burlington, Massachusetts 01803
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 Oracle9iAS 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 Oracle9iAS Personalization.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. 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 Corporation 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.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

This manual contains the following chapters and appendixes:

Chapter 1	Describes the OP APIs.
Chapter 2	Introduces REAPI.
Chapter 3	Describes the REAPI supporting classes.
Chapter 4	Describes the REAPI methods used to manage sessions, manage data, and request recommendations.
Chapter 5	Contains examples of how to perform common tasks with the REAPI.
Chapter 6	Introduces RE Batch API.
Chapter 7	Describes the RE Batch API supporting classes.
Chapter 8	Describes the RE Batch API methods used to request recommendations.
Chapter 9	Contains examples of how to perform common tasks with the RE Batch API.
Appendix A	Contains a complete example of REAPI use.
Appendix B	Contains a complete example of RE Batch API use.

Where to Find More Information

The documentation set for Oracle9iAS Personalization at the current release consists of the following documents:

- *Oracle9i Application Server Installation Guide*, Release 9.0.2 (the appropriate version for your operating system).
- *Oracle9iAS Release Notes*, Release 9.0.2.
- *Oracle9iAS Personalization Administrator's Guide*, Release 9.0.2.

- *Oracle9iAS Personalization User's Guide*, Release 9.0.2.
- *Oracle9iAS Personalization Programmer's Guide*, Release 9.0.2. Describes how to use OP's two APIs (this document).

Related Manuals

For more information about the database underlying OP, see

- *Oracle9i Administrator's Guide*

Documentation Formats

Documentation for Oracle9iAS Personalization is provided in PDF and HTML formats.

To view the PDF files, you will need

- Adobe Acrobat Reader 3.0 or later, which you can download from <http://www.adobe.com>.

To view the HTML files, you will need

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

Conventions

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

The SQL interface to Oracle9i is referred to as SQL. This interface is the Oracle9i 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.
.	
.	

Convention	Meaning
...	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.
<i>italic text</i>	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.

Oracle9iAS Personalization Programming

Oracle9iAS Personalization provides two Java application program interfaces (APIs):

- Recommendation Engine API (REAPI)
- Recommendation Engine Batch API (RE Batch API)

REAPI 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. REAPI is described in [Part I](#) of this manual.

RE Batch API enables a web application written in Java to request Oracle9iAS Personalization-style recommendations in bulk mode. The recommendations are written to table. RE Batch API does not return results in real time. REAPI Batch is described in [Part II](#) of this manual.

OP API Structure

The two OP APIs have the same components:

- Supporting classes, used to set constraints for the mining operations
- The proxy classes, used to obtain recommendations

Executing OP Programs

Before you can execute a program using either OP API, you must deploy and build an OP package, as described in the *Oracle9iAS Personalization User's Guide*.

Javadoc for the OP APIs

Detailed descriptions of the OP APIs are not included in this manual. The API calls are documented by Javadoc; see the OP section of the *Oracle9i Application Server Documentation Library*.

Part I

Recommendation Engine API

[Part I](#) describes the OP (Oracle9iAS Personalization) REAPI (Recommendation Engine Application Programming Interface). The REAPI permits a Web application to collect targeted data and to return recommendations during a session.

This part contains the following chapter:

- [Chapter 2, "REAPI Overview"](#)
- [Chapter 3, "REAPI Supporting Classes"](#)
- [Chapter 4, "Using REAPI"](#)
- [Chapter 5, "REAPI Examples and Usage"](#)

For a complete example of REAPI usage, see [Appendix A](#).

For detailed descriptions of the REAPI classes, see the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*.

REAPI Overview

The OP (Oracle9iAS 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 a complete example of REAPI use.

OP includes a demo program that helps you learn how the API methods work.

REAPI classes and methods are described in detail in the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*.

Note: REAPI and REAPI Demo are installed on the system where Oracle9iAS 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 Pick 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 *Oracle9i Application Server Installation Guide* and the *Oracle9iAS Personalization Administrator's Guide*. For information about how to create and deploy packages, see *Oracle9iAS Personalization User's Guide* and the online help for the OP administrative UI.

REAPI 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.

REAPI 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.

Web Applications and Sessions

Some Web applications are *stateful*, that is, they maintains the state of the client activities during a certain time period; other Web applications are stateless. Most Web applications that support eCommerce are stateful or *sessionful*. A client session often starts with a login and ends with either an explicit logout or when the session times out. Oracle Personalization maintains its own session for data mining purpose regardless of whether the application is stateful or stateless. If the application is stateful, the session that OP maintains can be perfectly mapped as the application's session. (For an eCommerce application, the recommendation made to the user is based on the user activities.) If the application does not maintain user session, OP then tracks the user session on itself. In this case, the OP session starts when a particular user ID appears in any REAPI method call the first time, and the session ends when the session times out, that is, when the user ID remains inactive for a preset time period.

In summary, the Web application that calls REAPI can be either of the following:

- sessionful (stateful), 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.

REAPI Sessionful Web Applications

If the Web application is sessionful, OP will map its session to the application session. To create a sessionful application, use 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.

REAPI Sessionless Web Applications

If the Web application is sessionless, the recommendation engine (RE) will maintain OP sessions by itself. An OP session will be created when the first REAPI method (either data collection or recommendation request) issued for a given `customerId`. The RE will track user activity until the session is timed out, that is, until the given `customerId` is inactive for a specified period.

The Web application uses the `createSessionless()` method of the class `IdentificationData` to create user identification for the session.

REAPI Data Collection

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 mining table repository (MTR) database. 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 user activity during the current session as well as historical data.

REAPI 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

REAPI 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.

Before Using REAPI

Before you can use REAPI, 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.

Oracle9iAS Personalization User's Guide 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 Oracle9i 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 [Chapter 5](#).

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 [Chapter 5](#).

REAPI Demo Program

OP includes 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 Oracle9iAS 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 the *Oracle9iAS Personalization User's Guide*. There are also some examples of how to perform typical tasks using REAPI in [Chapter 5](#) of this manual and a complete example using all REAPI functionality in [Appendix A](#).

Creating REProxyRT Objects

Before any recommendation or data collection requests can be processed using REAPI methods, at least one REProxyRT object that connects to designated RE must be created.

In a Web application environment, it is better to create all required proxies during the initialization stage. This is a safe approach, because the application does not have to process any recommendation requests after the application is initialized successfully.

If it is not possible to create all required proxies during initialization, the proxies may be created when the very first recommendation request is being processed. In this case, the application code must handle race conditions properly when numerous requests come up simultaneously. If numerous recommendation requests

came up before the proxy exists, only one of the requests will create the proxy object because creating a proxy is a synchronous process. Since it may take a few hundred milliseconds to create a proxy object, many requests may be held up during the time. Thus, a racing situation may occur. Although REAPI is multi-thread safe, such a racing situation will not cause any problem for REProxyRT. However, it may cause exceptions for the application.

See [Chapter 5](#) for more information about proxies.

Starting an REAPI 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 [Chapter 5](#).

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 REAPI Supporting Classes

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 3](#), to create these instances. It is always necessary, for example, to create an `IdentificationData` object. The following classes are frequently used in REAPI signatures:

- `IdentificationData`
- `FilteringSettings`
- `TuningSettings`
- `Item`
- `DataItem`

- Recommendation
- Recommendation Content

For examples, see [Chapter 5](#) and the complete example in [Appendix A](#).

Collecting Data for REAPI Recommendations

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.

Use the following methods to collect and manage data during the current session:

```
addItem();  
addItem();  
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.

OP Data Caching

When one of the OP data collection methods (`addItem()` or `addItem()`) is called, the user profile data is first saved in a buffer (the Data Collection Cache) on the Application Server. The data collection cache is created as part of the initialization of an `REProxy` object. The size of the data buffer is custom-configurable and is specified by the input parameter `cacheSize` of the method `REProxyManager.createProxy()`. The data saved in the buffer is periodically saved (archived) in the database. The interval of archive is set by another the input parameter `interval` of the same method. The data collection cache consists of two identical buffers; when one buffer is being archived, the other is used for saving the incoming data. Thus the data collection operation runs without interruption.

Getting REAPI 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.

How REAPI Creates 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) 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 REAPI 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 an REAPI 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 REProxyRT Objects

If you wish to destroy proxies programmatically you can use one of the following methods:

- `destroyProxy()`, which destroys one name proxy
- `destroyAllProxies()`, which destroys all existing proxies.

Both methods forcefully remove proxies regardless of their active status. See detailed discussion in [Chapter 5](#) for different usage models.

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 4](#). 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

This chapter does not contain detailed descriptions of any of the classes. For detailed information, see the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*.

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 REAPI Classes

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

- `DataItem`
- `Enum`

- FilteringSettings
- Item
- ItemList
- TuningSettings
- RecommendationContent (one class in `oracle.dmt.op.re.reapi.rt`)

To use the Enum interfaces, you must include the following statement in your Java program:

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

REAPI 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

REAPI 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 Class](#) 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
```

REAPI 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 Class](#), 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 4](#).

`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;
```

REAPI 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;
```

REAPI 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 Class](#) and [TuningSettings Class](#) 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;
```

REAPI 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;
```

REAPI 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;
```

REAPI 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 Class](#) 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;
```

REAPI 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 Class](#) and [RecommendationContent Class](#) 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;
```

REAPI 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 Class](#) and [RecommendationContent Class](#) later in this chapter.

`Sorting` takes on the following values:

- `Enum.Sorting.NONE`
- `Enum.Sorting.DESCEDING`
- `Enum.Sorting.ASCENDING`

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

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

REAPI 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`:

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

Other Supporting REAPI Classes

The other supporting classes are

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

These classes are described briefly in this document. For detailed descriptions, see the Javadoc for OP.

ContentItem Class

This class 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 Class](#), described later in this chapter, is any array of items of type `ContentItem`; the

description of [RecommendationContent Class](#) explains how sorting order works when multiple orders are specified.

This class contains the following methods:

- n `getContentAttribute()`
- n `getSorting()`

Dataltem Class

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

There are two kinds of methods provided with this class:

- n A constructor for `DataItem`
- n Methods that return attribute values:
 - n `getDataSource()`
 - n `getValue()`

FilteringSettings Class

This class is used to specify the items to include or exclude when generating recommendations.

Release 2 of OP supports category filtering only.

There are three kinds of methods provided with this class:

- n A constructor for `FilteringSettings`
- n Methods that set the attributes values:
 - n `setItemFiltering(int taxonomyID)`
 - n `setItemFiltering(int taxonomyID, long[] categoryList)`
 - n `setItemExclusion(int taxonomyID, long[] categoryList)`
 - n `setItemSubTreeFiltering(int taxonomyID, long[] categoryList)`
 - n `setCategoryExclusion(int taxonomyID, long[] categoryList)`

- n `setCategorySubTreeFiltering(int taxonomyID, long[] categoryList)`
 - n `setCategoryLevelFiltering(int taxonomyID, long[] categoryList)`
 - n `setCategoryFiltering(int taxonomyID)`
 - n `setCategoryFiltering(int taxonomyID, long[] categoryList)`
- n **Methods that return attribute values:**
 - n `getTaxonomyID()`
 - n `getCategoryFiltering ()`
 - n `getCategoryList()`
 - n `getCategoryMembership()`

Not all filtering settings can be used with all methods. In particular, the following filtering setting cannot be used with the cross-sell methods:

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

IdentificationData Class

Identifies the user and/or the session.

There are two kinds of methods provided with this class:

- n **Methods that create IdentificationData instances**
 - n `createSessionful(String appSessionID, UserType userType)`
 - n `createSessionless(String appSessionID, UserType userType)`
- n **Methods that return attribute values:**
 - n `getUserID()`
 - n `getAppSessionID()`
 - n `getUserType()`

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 Class

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.

There are three kinds of methods provided with this class:

- A constructor that creates an `Item` instance
- Methods that return attribute values
 - `getType()`
 - `getID()`
- A method that is a debugging aid

ItemDetailData Class

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 Class](#) later in this chapter for more details.

There are three methods:

- `getAttribute()`
- `getValue()`
- `toString()`

Recommendation Class

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

There are two methods:

- `getAttributes()`
- `toString()`

RecommendationContent Class

This class specifies the type of information that a recommendation request should return.

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

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 Class

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

The methods in this class 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.

TuningSettings Class

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

There are three kinds of methods provided with this class:

- A constructor that creates a `TuningSettings` instance
- Methods that set attribute values
- Methods that return attribute values

The following methods set attribute values:

- `setDataSourceType()`
- `setInterestDimension()`
- `setPersonalizationIndex()`
- `setProfileDataBalance()`
- `setProfileUsage()`

The following methods return attribute values:

- `getDataSourceType()`
- `getInterestDimension()`
- `getPersonalizationIndex()`
- `getProfileDataBalance()`
- `getProfileUsage()`

Using REAPI

This chapter provides an overview of the methods that are used to manage the recommendation engine proxy, to collect data, and to obtain recommendations, followed by usage notes for some of the methods. The supporting classes for these methods are described in [Chapter 3](#).

For detailed descriptions of these methods, see the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*.

For examples of how to use these classes and methods, see [Chapter 5](#) and the complete example in [Appendix A](#).

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

All methods described in this chapter are public.

Recommendation Proxy Classes

The real time 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 RE Proxy 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 these classes reside on the system where Oracle9iAS is installed.

RE Proxy Creation and Management

`REProxyManager` handles a pool of `REProxyRT` instances. Using multiple `REProxyRT` instances within a Web server, such as Oracle9i Application Server, 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 individual `REProxyRT` objects.

The following methods manage `REProxyRT` objects:

```
n createProxy
n getProxy
n destroyAllProxies
n destroyProxy
```

For examples of how to use the proxy manager, see [Chapter 5](#) and the complete example in [Appendix A](#).

Proxy Methods

All the recommendation requests are processed through class `REProxyRT`. Obtain a `REProxyRT` object using `createProxy` or `getProxy` before you perform any recommendation tasks, such as handling sessions for a sessionful application, collecting customer profile data, and getting recommendations.

RE Proxy Session Management

The following methods manage sessions:

```
n createCustomerSession
n createVisitorSession
n closeSession
```

RE Proxy 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:

```
n addItem
```

```
n  addItem
n  removeItem
n  removeItems
```

Re Proxy Customer Registration

The following method permits you to change a visitor to a customer (registered user):

```
n  setVisitorToCustomer
```

This method can be used in both sessionful or sessionless applications.

RE Proxy Recommendations

The following methods obtain and manage recommendations:

```
n  rateItem
n  rateItems
n  recommendTopItems
n  recommendBottomItems
n  recommendFromHotPicks
n  recommendCrossSellForItem
n  recommendCrossSellForItems
n  crossSellForItemFromHotPicks
n  crossSellForItemsFromHotPicks
n  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.

The methods that return recommendations do not necessarily return a list of items. If you set `FilteringSettings.CategoryMembership` to one of the values

```
n  Enum.CategoryMembership.EXCLUDE_CATEGORIES
n  Enum.CategoryMembership.INCLUDE_CATEGORIES
```

- `Enum.CategoryMembership.SUBTREE_CATEGORIES`
- `Enum.CategoryMembership.ALL_CATEGORIES`

then the recommendation methods (such as `recommendTopItems`, etc.) return categories.

Categories are components of a taxonomy. Taxonomies are defined in the following tables in the mining table repository (MTR):

- `MTR_TAXONOMY`
- `MTR_TAXONOMY_CATEGORY`
- `MTR_TAXONOMY_CATEGORY_ITEM`
- `MTR_CATERGORY`

An appropriate taxonomy is crucial to the design of an OP application. For information about how to create taxonomies, see *Oracle9iAS Personalization Administrator's Guide*.

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 item 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.

RE Proxy Method Usage Notes

For detailed descriptions of these methods, see the OP Javadoc included in the OP section of the *Oracle9i Application Server Documentation Library*. This section provides an overview of the methods and how to use them.

Session Creation

For both `createCustomerSession` and `createVisitorSession`, the calling Web application must provide session IDs that are unique among currently active sessions. If either 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 administrator's guide.) OP has no way to tell whether `customerID` and `appSessionID` are valid values; it is the responsibility of the calling Web application to verify that these values are valid.

Data Collection

To collect data, use `addItem` or `addItems`. Use `removeItem` or `removeItems` to remove data from the local cache.

Add Items

For both `addItem` and `addItems`, 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 administrator's 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.

Data collected in the RE is automatically written to the MTR.

Remove Items

`removeItem` and `removeItems` remove items that have *not* been written to the MTR (permanent storage). Once data is written to the MTR, you cannot use these methods to remove the data.

Proxy Creation

In `createProxy`, you must specify a cache size and an interval. This section describes how to determine these values.

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

A good way to configure cache size and interval is the following:

1. Set cache size to approximately 3027 kilobytes.
2. Set interval according to the estimated data collection rate.
3. Test.
4. Adjust the archive interval.

Cache Size

The cache size is the size of the cache used by the recommendation engine, in kilobytes.

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.)

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.

Cross Sell Methods

The comments in this section apply to `crossSellForItemFromHotPicks`, `crossSellForItemsFromHotPicks`, `recommendCrossSellForItem`, and `recommendCrossSellForItems`.

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 these methods:

- `setCategoryLevelFiltering`
- `setCategorySubtreeFiltering`
- `setCategoryExclusion`

- `setCategoryFiltering(int)`
- `setCategoryFiltering(int, long[])`

Proxy Destruction

Destroy proxy objects with extreme caution. `REProxyRT` objects are shared by many clients; therefore, destruction of a proxy may interrupt recommendation services. The proxy destruction methods must be used very carefully. For Web applications, `REProxyRT` objects should be treated as part of the server services; they should not be unless it is absolutely necessary. Like other server components, these objects only need to be destroyed when the server is shut down or taken offline for maintenance purposes.

You can either destroy a specific proxy in the pool, using `destroyProxy`, or all proxies in the pool, using `destroyAllProxies`.

REAPI Examples and Usage

This chapter 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 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 Oracle9iAS 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 the *Oracle9iAS Personalization User's Guide*.

REAPI Basic Usage

The REProxY methods described in [Chapter 4](#) 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 object when it is no longer needed by any program that is using it.

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 chapter.

The following code fragment creates an object named `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,
                                       cacheSize,
                                       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 chapter for details.

Destroy the Proxy

Destroy proxy objects with extreme caution. REProxyRT objects are shared by many clients; therefore, destruction of a proxy may interrupt recommendation services. The proxy destruction methods must be used very carefully. For Web applications, REProxyRT objects should be treated as part of the server services; they should not be unless it is absolutely necessary. Like other server components, these objects only need to be destroyed when the server is shut down or taken offline for maintenance purposes.

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 chapter. You need to know the user name and password for the RE schema. If the proxy already exists, call `getProxy`.

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. */
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();
```

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 chapter. You need to know the user name and password for the RE schema. If the proxy already exists, call `getProxy`.
2. Get identification data.

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

3. Call REAPI methods: for example,

```

/*Set input parameters.*/
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.

REProxyManager Interaction with JVM

`REProxyManager` is a singleton implementation, that is, only one instance of the `REProxyManager` class is created in a given JVM instance and the class is automatically loaded in the JVM instance. This behavior has implications about the way your program behaves. The behavior is different depending on whether your application is a standalone Java program or a Java server-side module. The same principle may apply but different usage models for proxy management should be considered

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.

Java Server-Side Modules

If REAPI is called from Java server-side modules, such as servlets or Java Server Pages (JSPs), the `REProxyManager` class is loaded on the Oracle9i Application Server where the modules reside.

The Web application that owns and uses the Java modules often starts when the server boots up, and does not close until the server shuts down. In this circumstance, you may create the proxies during the initiation of the Web application or as soon as the first RE request is being processed, but never have to worry about destroying the proxy. As long as the Web application is up and running, the proxy will be used to serve ongoing recommendation requests.

Creation of a proxy is time consuming (a few hundred milliseconds on a Sun E450 server). It is therefore more efficient to never destroy a proxy until the server shuts

down, for example, when the system administrator needs to bring the Web application down for maintenance purposes.

If you choose to micro-manage proxies, that is, remove unused proxy objects, you may do so by calling the destroy methods. However, be careful with destroying methods, because both destroy methods will remove proxies forcefully, that is, they do not check to see whether any other process is using the proxy.

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 the way you might use two REs 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 the way 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 a different RE, RE2. To accomplish this, first create two different proxies, RE1 and RE2, and then call `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

Oracle9iAS 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 chapter 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 administrator's 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 Oracle9iAS/OC4J instance and that Oracle9iAS/OC4J instance is configured to use one RE at all times. The application must route users to "their" Oracle9iAS/OC4J instance and configure Oracle9iAS/OC4J instances 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 an `Oracle9iAS/OC4J.properties` file, or by being explicitly coded in the Web applications, or by some other means.
2. Allow any Oracle9iAS/OC4J instance 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 numerical 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 an ID different 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.

Part II

Recommendation Engine Batch API

[Part II](#) describes the OP (Oracle9iAS Personalization) RE Batch API (Recommendation Engine Batch Application Programming Interface) enables a web application written in Java to request Oracle9iAS Personalization-style recommendations in bulk mode.

This part contains the following chapter:

- [Chapter 6, "RE Batch API Overview"](#)
- [Chapter 7, "RE Batch API Supporting Classes"](#)
- [Chapter 8, "Using the Recommendation Engine Batch Proxy"](#)
- [Chapter 9, "REProxyBatch API Examples and Usage"](#)

For a complete example of RE Batch API usage, see [Appendix B](#)

For detailed descriptions of the RE Batch API classes and methods, see the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*. Note that many of the batch methods and classes are REAPI methods and classes.

RE Batch API Overview

The OP (Oracle9iAS Personalization) RE Batch API (Recommendation Engine Batch Application Programming Interface) enables an application written in Java to request Oracle9iAS Personalization-style recommendations in bulk mode.

RE Batch API 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.

[Chapter 9](#) contains examples of how to perform common tasks using RE Batch API.

[Appendix B](#) contains a complete example of RE Batch API usage.

RE Batch API classes and methods are described in detail in the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*

Note: RE Batch API is installed on the system where Oracle9iAS is installed.

RE Batch API Prerequisites

Before you can use RE Batch API methods, OP must be installed and the appropriate tables must be created and populated. Your database tables must be converted to the OP schemas. It is important that the OP MTR is populated with customer profiles. You should also create tables or views containing the customer IDs for which you want recommendations.

If you are using one or more taxonomies, they must be properly specified.

At least one OP package must have been built and deployed. Use the OP administrative interface to do this. For an example of how to create and deploy a package, see *Oracle9iAS Personalization User's Guide*.

Note: Do not deploy a package while an RE Batch call is in progress; do not start an RE Batch call while a deployment is in progress. Either of these activities causes an exception.

RE Batch API Definitions and Concepts

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

RE Batch API End Users (Customers)

End users (users of a Web site that uses OP for recommendations) 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. The RE Batch API makes recommendations for customers only.

RE Batch API Recommendations

Recommendations are based on historical data, which is stored in the database and retrieved when the customer profiles are loaded.

Using RE Batch API

Before you execute an RE Batch program, you must

- Set up the OP environment (create an RE, and create and deploy an OP package)
- Create the tables used by the RE Batch methods

Setting Up the RE Batch API Environment

Before you can execute RE Batch API methods, the following must be true:

- Properly formatted customer profile data must be available in the Mining Table Repository (MTR)

- A recommendation engine (RE) farm containing at least one recommendation engine must exist.
- A package must have been successfully built and then deployed in the recommendation engine farm.

The OP administrator's guide and the online help for the OP administrative GUI explain how to perform these steps.

Customer Profile Data

Customer profile data resides in the MTR.

Deploy a Package to an RE

You cannot get recommendations until there is an existing deployed package, which is created using the OP administrative interface. You must build a package before you deploy it. You cannot build a package until there is some data available; data is 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. We recommend that the REs used for bulk recommendations not be used for any other purpose. For a discussion of the design considerations, see "[Recommendation Engine Usage](#)" in [Chapter 9](#).

Note: If you try to deploy a package an RE while a batch program is running, the deployment will fail.

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 items that are recommended have prices in several currencies (for example, items are sold in Japan and India), see "[Handling Multiple Currencies](#)" in [Chapter 9](#).

Sample RE Batch API Usage

OP includes a sample Java program that illustrates the use of many of the RE Batch API methods; the program is in [Appendix B](#). There are also some examples of how to perform typical tasks in [Chapter 9](#).

Creating an REBatchProxy Object

Before you can use any of the RE Batch API methods, you must create at least one `REBatchProxy` object; see [Chapter 9](#) for details. The object establishes a JDBC connection to a specified database and schema. The connection exists until it is explicitly destroyed.

Creating Instances of RE Batch API Objects

To use the API, you must create instances of the objects used by the API method signatures. Use the RE Batch API supporting classes, described in [Chapter 8](#), to create these instances. It is always necessary, for example to create filtering settings and tuning sessions. For examples, see [Chapter 9](#).

Converting Data for RE Batch API

OP generates recommendations based on data describing past user behavior.

User data stored in an Oracle table must be transformed and stored in the Mining Table Repository (MTR) before it can be used to generate recommendations.

Managing Customer Profiles for RE Batch API

OP stores customer profiles in the Mining Table Repository (MTR). The profiles to be used must be loaded into an RE before any recommendation requests are made. The following methods manage load and unload customer profiles from an RE:

```
n loadCustomerProfiles()  
n purgeCustomerProfiles()
```

Before you load a set of customer profiles, you must create a table or a view containing a list of the customer IDs that identify the profiles that you wish to load, that is, a list of the customer IDs for which you want a recommendation.

Getting RE API Batch Recommendations

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

```
n crossSellForItem()  
n rateItem()  
n recommendTopItems()
```

These methods are used for getting recommendations for customers (registered users).

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.

Creating Recommendations

OP uses rule tables stored in the RE to calculate the recommendations requested by the methods listed above. The specific rule table used depends upon the RE Batch API method used. In general, the antecedents of the rules are matched against the historical 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. The recommendations are written to a database table.

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 recent rules.

Scoring: For scoring, all available historical data is used.

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 RE Batch Recommendations

RE Batch API methods that make recommendations write the recommendations to a database table. The schema used for the output depends on the method used. You can extract the recommendations in many ways, for example, with an appropriate SQL query, and then decide which recommendations to pass to the user.

Removing the REBatchProxy Object

Before you exit the application, you should destroy any proxy objects that you no longer need.

RE Batch API Supporting Classes

This chapter describes the supporting classes for the `REProxyBatch` class. These classes are used to create instances of the objects used by the methods described in [Chapter 8](#). You may be able to create one instance of many of these classes and use that one instance as an argument for several calls.

Note: Except for `Location`, these supporting classes are the same as the ones that are used by REAPI. (Not all REAPI classes are used by the RE Batch API.)

Before you issue any of the recommendation methods described in [Chapter 8](#), you must generate appropriate "[FilteringSettings Class](#)", "[TuningSettings Class](#)", and "[Location Class](#)" instances.

All methods described in this chapter are public.

This chapter does not contain detailed descriptions of any of the classes. For detailed information, see the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*.

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 RE Batch API Classes

The following frequently used classes are in the `oracle.dmt.re.base` subdirectory:

- `DataItem`
- `Enum`
- `FilteringSettings`
- `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 for RE Batch API

Many of the RE Batch API methods reference attributes that can take on a finite number of values. The interface `Enum` is used to implement the base class for these "enumerations."

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`
- `InterestDimension`
- `PersonalizationIndex`
- `ProfileDataBalance`

- ProfileUsage
- Sorting

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 category should be excluded from the category list. For details, see "[FilteringSettings Class](#)" 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. The method [DataItem Class](#), 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 method in the Javadoc included with OP.

`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;
```

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 would like to purchase the items. `RATING` indicates that the user likes the items. For more information, see the description of "[TuningSettings Class](#)" 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`:

```
InterestDimensionType 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 "[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 "[TuningSettings Class](#)" later in this chapter.

ProfileDataBalance takes on the following values:

- Enum.ProfileDataBalance.HISTORY

Note: The only value of profile data balance that makes sense for bulk recommendations is Enum.ProfileDataBalance.HISTORY. You must specify this value. (There is no current session data available.)

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

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

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 Class](#)" 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;
```

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 discussion of the [DataItem Class](#) later in this chapter.

`Sorting` takes on the following values:

- `Enum.Sorting.NONE`
- `Enum.Sorting.DESENDING`
- `Enum.Sorting.ASCENDING`

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

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

Other RE Batch API Supporting Classes

The other supporting classes are

- `DataItem`

- FilteringSettings
- Location
- TuningSettings

DataItem Class

This class is a subclass of class `Item`. It encapsulates data about an item.

There are two kinds of methods provided with this class:

- A constructor that creates a `DataItem` instance
- Methods that return attribute values

The following methods return attribute values:

- `getDataSource()`
- `getValue()`

FilteringSettings Class

This class is used to specify the items to include or exclude when generating recommendations.

Release 2 of OP supports category filtering only.

There are three kinds of methods provided with this class:

- A constructor for `FilteringSettings`
- Methods that set the attributes values:
 - `setItemFiltering(int taxonomyID)`
 - `setItemFiltering(int taxonomyID, long[] categoryList)`
 - `setItemExclusion(int taxonomyID, long[] categoryList)`
 - `setItemSubTreeFiltering(int taxonomyID, long[] categoryList)`
 - `setCategoryExclusion(int taxonomyID, long[] categoryList)`
 - `setCategorySubTreeFiltering(int taxonomyID, long[] categoryList)`

- `setCategoryLevelFiltering(int taxonomyID, long[] categoryList)`
- `setCategoryFiltering(int taxonomyID)`
- `setCategoryFiltering(int taxonomyID, long[] categoryList)`
- **Methods that return attribute values:**
 - `getTaxonomyID()`
 - `getCategoryFiltering ()`
 - `getCategoryList()`
 - `getCategoryMembership()`

Not all filtering settings can be used with all methods. In particular, the following filtering setting cannot be used with the cross-sell methods :

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

Item Class

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.

There are two kinds of methods provided with this class:

- A constructor that creates an `Item` instance
- Methods that return attribute values
 - `getType()`
 - `getID()`

Location Class

This class specifies the location of the input table or the location of the table containing the results of an `REProxyBatch` method. The schema for the table depends on the call made. See the descriptions of the individual methods in the Javadoc for details.

There are three kinds of methods provided with this class:

- A constructor that creates a `Location` instance
- Methods that return attribute values
 - `getDatabaseURL()`
 - `getDatabaseAlias()`
 - `getSchemaName()`
 - `getTableName()`
 - `getUserName()`
 - `getPassword()`

TuningSettings Class

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

There are three 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 methods set attribute values:

- `setDataSourceType()`
- `setInterestDimension()`
- `setPersonalizationIndex()`
- `setProfileDataBalance()`
- `setProfileUsage()`

The following methods return attribute values:

- `getDataSourceType()`
- `getInterestDimension()`
- `getPersonalizationIndex()`
- `getProfileDataBalance()`
- `getProfileUsage()`

Using the Recommendation Engine Batch Proxy

This chapter consists of an overview of the class and methods that are used to manage the customer profiles and to obtain recommendations. The supporting classes for these methods are described in [Chapter 7](#).

For detailed descriptions of these methods, see the Javadoc in the OP section of the *Oracle9i Application Server Documentation Library*.

All methods described in this chapter are public.

REProxy Batch Overview

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

- Proxy creation and management, including customer profile management (load and purge customer profiles)
- Recommendation methods (obtain recommendations)

For examples of how to use these classes and methods, see [Chapter 9](#).

Location of REProxyBatch Classes

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

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

These classes are installed on the system where Oracle9iAS is installed.

REProxyBatch Creation and Management

The `REProxyBatch.java` class establishes the JDBC connection to the RE schema where the methods execute. The connection continues to exist until the connection is explicitly destroyed with the `destroy()` method. The class also includes customer profile management methods.

Customer Profile Management

You must load customer profiles from the MTR to the RE before you can request recommendations; after you are done, you should purge the loaded profiles from the RE. The methods are

- n `LoadCustomerProfiles();`
- n `PurgeCustomerProfiles();`

REProxyBatch Recommendations

The following methods obtain recommendations:

- n `crossSellForItems`
- n `rateItem`
- n `recommendTopItems`

Communicating the returned recommendations to the end user is the responsibility of the application. The recommendations are written to an output table; the schema of the output table depends on the method called. For details, see the description of each method.

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:

- n For `InterestDimension.RATING`, the expected rating for the item is returned.

- For `InterestDimension.PURCHASING` or `InterestDimension.NAVIGATION`, a scaled probability is returned. The most probable item is assigned a value of 1 and other items are assigned values less than 1 that are proportional to how probable the items are compared to the most probable item.

Cross Sell Method Usage Notes

The comments in this section apply to `recommendCrossSellForItems`.

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[])`

Recommendation Method Usage Notes

`recommendTopItems` does not necessarily return a list of items. If you set `FilteringSettings.CategoryMembership` to one of the values

- `Enum.CategoryMembership.EXCLUDE_CATEGORIES`
- `Enum.CategoryMembership.INCLUDE_CATEGORIES`
- `Enum.CategoryMembership.SUBTREE_CATEGORIES`
- `Enum.CategoryMembership.ALL_CATEGORIES`

then `recommendTopItems` returns a list of categories.

Categories are components of a taxonomy. Taxonomies are defined in the following tables in the mining table repository (MTR):

- `MTR_TAXONOMY`
- `MTR_TAXONOMY_CATEGORY`
- `MTR_TAXONOMY_CATEGORY_ITEM`

▪ MTR_CATEGORY

An appropriate taxonomy is crucial to the design of an OP application. For information about how to create taxonomies, see *Oracle9iAS Personalization Administrator's Guide*, Release 9.0.2.

REProxyBatch 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 when the package is deployed. The specific rule table used depends upon the RE Batch API call made. In general, the antecedents of the rules are matched against the data in cache (historical data only for RE Batch) and the probabilities of the various consequents are computed. These items are then ordered by probability, and `numberOfItems` (an API argument) items are returned.

REProxyBatch API Examples and Usage

This chapter provides examples of REProxyBatch API use. In some instances, we provide coding skeletons; in others, we describe an approach for solving certain kinds of problems using OP.

REProxyBatch API Basic Usage

The REBatchProxy methods described in [Chapter 8](#) permit to write Java programs that generate recommendations.

Note: The RE Batch API classes are installed on the system where Oracle9iAS is installed. The tables that they use are installed on a different system (the system where Oracle9i is installed.) The following steps must be performed on the correct system.

To use REProxyBatch API calls, you must perform the following steps:

1. Create and deploy a package to the RE that you will use for recommendations.
2. Create an instance of REBatchProxy.
3. Create any required tables. (Alternatively, you can create the tables using SQL before you execute the program.)
4. Load customer profiles.
5. Execute the desired recommendation methods.
6. Purge the customer profiles that you loaded in step 4.
7. Destroy the database connection that you created in step 2.

You will now have a table containing the recommendations that you requested. You can use SQL to examine the table.

Code Sample: Recommend Top

The following code sample illustrates obtaining a recommendation:

```
// Create an instance of REProxyBatch
// Create customer table
// Load customer profiles
// Execute recommend_top
// Purge customer profiles loaded above
// Destroy the database connection held by REProxyBatch
```

Code Sample: Recommend Cross Sell

The following code sample illustrates obtaining cross-sell recommendations:

```
// Create an instance of REProxyBatch
// Create Items table
// Execute cross sell for items
// Destroy the database connection held by REProxyBatch
```

Recommendation Engine Usage

REBatchProxy requires at least one recommendation engine (RE) in at least one recommendation engine farm.

We recommend that the REs used for bulk recommendations not be used for any other purpose.

Note: If you try to deploy a package on an RE while a batch program is running, the deployment will fail.

In general, you may 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.

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 administrator's guide for more information.

If your application has more than one RE available for use, it must determine which one to use. You can load different sets of customer profiles into different REs, generate appropriate recommendations, and then merge the recommendation tables, if desired.

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 the way 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.

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.

REAPI Sample Program

This appendix contains `ProxyTest.java`, a sample Java program that illustrates using REAPI.

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 Oracle9iAS is installed.

Note: REAPI is installed on the system where Oracle9iAS is installed. It is simplest to run this program on that system.

```
// Copyright (c) 2001, 2002 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 recommendations 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();
}
}

}
```

REProxyBatch Sample Program

The sample program for RE Proxy Batch consists of a Java program and a property file. The sample program, property file, and the tables required to run it are installed when you install OP.

RE Batch Sample Program Overview

The Java program `REBatchTest.java` and the property file `batchtest.txt` are in the `TBS` directory on the system where you have installed OP.

`REBatchTest.java REProxyBatch` allows you to execute a subset of recommendation functions in bulk. (`REProxyRT` scores one user/item at a time.) `REProxyBatch` reads a list of items/customers to be scored from an input table and writes the result to a new output table. This program reads its input from the property file `batchtest.ini`.

RE Batch Sample Program Output

The input item details (for `rateItem` and `crossSellForItem`) are derived from the OP demo data. But in OP, the model built on the same data is not guaranteed to produce the same rules each time that it is run. Therefore, it is possible that the item being rated cannot be rated with the current set of rules. The output tables will either be empty (zero rows) or will contain fewer than expected records (if only some of the items are valid cross-sell candidates etc.).

Executing the RE Batch Sample Program

Follow these steps to execute the sample program:

1. Install OP.

2. The code and data for the sample program is installed into the following directories when you install OP:
 - The following code is installed in `${ORACLE_HOME}/dmt/reapi/batch/`
 - `batchtest.txt`
 - `README.txt`
 - `REBatchTest.java`
 - The following items associated with the data used by the sample program are installed in `${ORACLE_HOME}/dmt/reapi/batch/sampleData`
 - `create_batch_demo_input_tables.sql`
 - `customer_list_in.ctl`
 - `customer_list_in.txt`
 - `item_list_in.ctl`
 - `item_list_in.txt`
 - `load_batch_demo_data.sh`
3. Run the shell script `load_batch_demo_data.sh` to load the following tables:
 - `customer_list_in` — Used for `loadCustomerProfile`. (The output of `loadCustomerProfile` is used by `recommendTopItems` and `rateItem`.)
 - `item_list_in` — Used by `crossSellForItem`.
4. Compile the sample code. Your `CLASSPATH` variable should include the following zip/jar files:
 - `${ORACLE_HOME}/dmt/opreapi-batch.jar`
 - `${ORACLE_HOME}/dmt/oputil.jar`It also needs to include JDBC related jar/zip files:
 - `${ORACLE_HOME}/jdbc/lib/classes12.zip`
5. Change the property file to point to the appropriate entities. The comments in the property file and the file `README.txt` describes the exact changes that must be made.
6. Run `REBatchTest`, with the property file name as an input parameter.

RE Batch Sample Program Code

This section contains the code for the sample program and its property file.

batchtest.txt

The properties file for the sample program follows. Note that you must replace RE details and input/output table details to reflect your installation.

```
###
### Input file for REProxyBatch sample program
### Before Running, you will need to replace the following dummy strings with actual information:
### 1. RE* details ( Url,Username,Password) to point to the RE.
### 2. Input and Output (Result) table details for each of the calls.

#A unique name for proxy
ProxyName=REB_1

#Recommendation Engine details
REUrl=jdbc:oracle:thin:@myDBUrl
REUsername=REUser
REPassword=REPassword

#Input customer table location
Input.Url=jdbc:oracle:thin:@myDBUrl
Input.Alias=myDBAlias
Input.Schema=User1
Input.Table=customer_list_in
Input.Username=User1
Input.Password=Password1

#Customer profile table
# This table is created in RE by loadCustomerProfile. Once created
# it is used for recommendTopItems and rateItem
CustProfile=MY_CUSTOMER_PROFILE

#
# Details for recommendTopItems
#
# Number of items to be recommended per customer
TopN.NumberOfItems=10
#TuningSettings details
#valid DataSourceTypes are ALL, DEMOGRAPHIC, PURCHASING, RATING, NAVIGATION
TopN.DataSourceType=ALL
#valid InterestDimension: PURCHASING, RATING, NAVIGATION
```

```
TopN.InterestDimension=PURCHASING
#valid PersonalizationIndex: LOW, MEDIUM, HIGH
TopN.PersonalizationIndex=MEDIUM
## ProfileDataBalance needs to be specified as part of the TuningSettings object
## but its value is not used by REProxyBatch
#valid ProfileDataBalance: HISTORY, CURRENT, BALANCED
TopN.ProfileDataBalance=HISTORY
#valid ProfileUsage: INCLUDE, EXCLUDE
TopN.ProfileUsage=INCLUDE
# FilteringSettings details
TopN.Taxonomy=1
#Category list is a series of numbers separated by "-"
TopN.CategoryList=1-2-3-4-5
#Valid CategoryMembership: ExcludeItems, ExcludeCategories, IncludeItems, IncludeCategories,
#    level, SubTreeItems, SubTreeCategories, AllItems, AllCategories
TopN.CategoryMember=AllItems
# Result table details
TopNResult.Url=jdbc:oracle:thin:@myDBUrl
TopNResult.Alias=myDBAlias
TopNResult.Schema=User2
TopNResult.Table=TopN_RESULTS
TopNResult.Username=User2
TopNResult.Password=Password2

#
# Details for rateItem
#
#TuningSettings details
RateI.ItemID=417
RateI.ItemType=MOVIE
RateI.DataSourceType=RATING
RateI.InterestDimension=RATING
RateI.PersonalizationIndex=LOW
## ProfileDataBalance needs to be specified as part of the TuningSettings object
## but its value is not used by REProxyBatch
RateI.ProfileDataBalance=HISTORY
RateI.ProfileUsage=INCLUDE
RateI.Taxonomy=1
# Result table details
RateIResult.Url=jdbc:oracle:thin:@myDBUrl
RateIResult.Alias=myDBAlias
RateIResult.Schema=User3
RateIResult.Table=RATEITEM_RESULTS
RateIResult.Username=User3
RateIResult.Password=Password3
```



```

#
# Details for crossSellForItem
#
#Input items table details
ItemTable.Url=jdbc:oracle:thin:@myDBUrl
ItemTable.Alias=myDBAlias
ItemTable.Schema=User4
ItemTable.Table=item_list_in
ItemTable.Username=User4
ItemTable.Password=User4
# Number of items to be recommended per input item
XSell.NumberOfItems=10
#TuningSettings details
XSell.DataSourceType=NAVIGATION
XSell.InterestDimension=NAVIGATION
XSell.PersonalizationIndex=HIGH
## ProfileDataBalance needs to be specified as part of the TuningSettings object
## but its value is not used by REProxyBatch
XSell.ProfileDataBalance=HISTORY
XSell.ProfileUsage=EXCLUDE
#FilteringSettings details
XSell.Taxonomy=1
XSell.CategoryList=1-3-5-7-9
XSell.CategoryMember=AllItems
# Result table details
XSellResult.Url=jdbc:oracle:thin:@myDBUrl
XSellResult.Alias=myDBAlias
XSellResult.Schema=User4
XSellResult.Table=XSELL_RESULTS
XSellResult.Username=User5
XSellResult.Password=Password5

```

REBatchTest.java

The sample program follows. Note that you must replace RE details and input/output table details to reflect your installation.

```

###
### Input file for REProxyBatch sample program
### Before Running, you will need to replace the following dummy strings with actual information:
### 1. RE* details ( Url,Username,Password) to point to the RE.
### 2. Input and Output (Result) table details for each of the calls.

#A unique name for proxy

```

```
ProxyName=REB_1

#Recommendation Engine details
REUrl=jdbc:oracle:thin:@myDBUrl
REUsername=REUser
REPassword=REPassword

#Input customer table location
Input.Url=jdbc:oracle:thin:@myDBUrl
Input.Alias=myDBAlias
Input.Schema=User1
Input.Table=customer_list_in
Input.Username=User1
Input.Password=Password1

#Customer profile table
# This table is created in RE by loadCustomerProfile. Once created
# it is used for recommendTopItems and rateItem
CustProfile=MY_CUSTOMER_PROFILE

#
# Details for recommendTopItems
#
# Number of items to be recommended per customer
TopN.NumberOfItems=10
#TuningSettings details
#valid DataSourceTypes are ALL, DEMOGRAPHIC, PURCHASING, RATING, NAVIGATION
TopN.DataSourceType=ALL
#valid InterestDimension: PURCHASING, RATING, NAVIGATION
TopN.InterestDimension=PURCHASING
#valid PersonalizationIndex: LOW, MEDIUM, HIGH
TopN.PersonalizationIndex=MEDIUM
## ProfileDataBalance needs to be specified as part of the TuningSettings object
## but its value is not used by REProxyBatch
#valid ProfileDataBalance: HISTORY, CURRENT, BALANCED
TopN.ProfileDataBalance=HISTORY
#valid ProfileUsage: INCLUDE, EXCLUDE
TopN.ProfileUsage=INCLUDE
# FilteringSettings details
TopN.Taxonomy=1
#Category list is a series of numbers separated by "-"
TopN.CategoryList=1-2-3-4-5
#Valid CategoryMembership: ExcludeItems, ExcludeCategories, IncludeItems, IncludeCategories,
# level, SubTreeItems, SubTreeCategories, AllItems, AllCategories
TopN.CategoryMember=AllItems
```

```
# Result table details
TopNResult.Url=jdbc:oracle:thin:@myDBUrl
TopNResult.Alias=myDBAlias
TopNResult.Schema=User2
TopNResult.Table=TopN_RESULTS
TopNResult.Username=User2
TopNResult.Password=Password2

#
# Details for rateItem
#
#TuningSettings details
RateI.ItemID=417
RateI.ItemType=MOVIE
RateI.DataSourceType=RATING
RateI.InterestDimension=RATING
RateI.PersonalizationIndex=LOW
## ProfileDataBalance needs to be specified as part of the TuningSettings object
## but its value is not used by REProxyBatch
RateI.ProfileDataBalance=HISTORY
RateI.ProfileUsage=INCLUDE
RateI.Taxonomy=1
# Result table details
RateIResult.Url=jdbc:oracle:thin:@myDBUrl
RateIResult.Alias=myDBAlias
RateIResult.Schema=User3
RateIResult.Table=RATEITEM_RESULTS
RateIResult.Username=User3
RateIResult.Password=Password3

#
# Details for crossSellForItem
#
#Input items table details
ItemTable.Url=jdbc:oracle:thin:@myDBUrl
ItemTable.Alias=myDBAlias
ItemTable.Schema=User4
ItemTable.Table=item_list_in
ItemTable.Username=User4
ItemTable.Password=User4
# Number of items to be recommended per input item
XSell.NumberOfItems=10
#TuningSettings details
XSell.DataSourceType=NAVIGATION
XSell.InterestDimension=NAVIGATION
```

```
XSell.PersonalizationIndex=HIGH
## ProfileDataBalance needs to be specified as part of the TuningSettings object
## but its value is not used by REProxyBatch
XSell.ProfileDataBalance=HISTORY
XSell.ProfileUsage=EXCLUDE
#FilteringSettings details
XSell.Taxonomy=1
XSell.CategoryList=1-3-5-7-9
XSell.CategoryMember=AllItems
# Result table details
XSellResult.Url=jdbc:oracle:thin:@myDBUrl
XSellResult.Alias=myDBAlias
XSellResult.Schema=User4
XSellResult.Table=XSELL_RESULTS
XSellResult.Username=User5
XSellResult.Password=Password5
```

Index

A

accessibility
 of documentation, xi
API structure, 1-1

B

batch
 examples, 9-1
 use, 9-1
batch proxy, 6-1
batch recommendations, 8-4

C

cache
 data, 4-2
caching
 data, 2-7
CategoryMembership, 7-3
CategoryMembership interface, 3-3
class
 Contentitem, 3-9
 DataItem, 3-10, 7-8
 FilteringSettings, 3-10, 7-8
 IdentificationData, 3-11
 Item, 3-12, 7-9
 ItemDetailData, 3-12
 Location, 7-10
 Recommendation, 3-12
 RecommendationList, 3-13
 REProxyManager, 4-3
 TuningSettings, 3-13, 7-10

classes
 REAPI, 4-1
 REProxyBatch, 8-1
code sample
 REProxyBatch, 9-2
ContentItem class, 3-9
conventions
 table of, xiii
cross sell, 4-8
 usage notes, 8-3
currencies
 handling, 5-8, 9-3
customer profile management
 REProxyBatch, 8-2
customer registration, 4-4
customers, 2-2, 8-2
 RE Batch API, 6-2, 6-4

D

data
 RE Batch API, 6-4
data caching, 2-7, 4-2
data collection, 2-3, 4-2, 4-3, 4-6
data management, 4-3
DataItem class, 3-10, 7-8
DataSource interface, 3-3, 7-4
demographic data
 using, 9-4
destroy a proxy, 5-3
document accessibility, xi

E

end user

REAPI, 2-2

Enum interface, 3-2, 7-2

EnumType interfaces

RE Batch, 7-2

REAPI, 3-2

examples

REAPI, 5-1

executing OP programs, 1-1

executing REAPI programs, 2-1, 2-4

F

filtering, 3-10, 7-8

Filtering interface, 3-4

FilteringSettings class, 3-10, 7-8

H

handling currencies, 5-8, 9-3

handling time-based items, 5-11, 9-4

hot picks

REAPI, 2-4

I

IdentificationData class, 3-11

individual recommendations, 5-8

initialization

RE Proxy, 5-6

InterestDimension interface, 3-5, 7-4

interface

CategoryMembership, 3-3, 7-3

DataSource, 3-3, 7-4

Filtering, 3-4

InterestDimension, 3-5, 7-4

PersonalizationIndex, 3-5, 7-5

ProfileDataBalance, 3-6, 7-6

ProfileUsage, 3-7, 7-6

RecommendationAttribute, 3-7

Sorting, 3-8, 7-7

User, 3-8

Item class, 3-12, 7-9

ItemDetailData class, 3-12

J

java applications

standalone, 5-5

Java server-side modules, 5-5

Javadoc, 1-2

JVM

proxy interaction, 5-5

L

load balancing

REAPI, 5-8

Location class, 7-10

location of REAPI classes, 3-1

M

multiple currencies, 5-8, 9-3

multiple instances

REProxy, 5-6

O

OP data caching, 2-7

OP programs

how to execute, 1-1

OP ratings, 3-1

P

PersonalizationIndex interface, 3-5, 7-5

preface, xi

conventions table, xiii

prerequisites

RE Batch API, 6-1

ProfileDataBalance

interface, 7-6

ProfileDataBalance interface, 3-6

ProfileUsage interface, 3-7, 7-6

proxy, 8-1

creation, 4-2, 8-2

destroying, 5-3

initialization, 5-6

management, 4-2, 8-2

using, 5-2

- proxy creation, 4-7, 8-1
- proxy destruction, 4-9
- proxy management, 8-1
- proxy objects
 - REAPI, 2-5
 - removing, 2-9
- ProxyBatch, 9-1

R

- ratings, 8-2
 - OP, 3-1
- RE API Batch
 - getting recommendations, 6-4
- RE Bacth API
 - data, 6-4
- RE Batch API, 6-1
 - customer profiles, 6-4
 - customers, 6-2
 - environment, 6-2
 - example, 6-3
 - recommendations, 6-2, 6-4
 - using, 6-2
- RE Batch API prerequisites, 6-1
- RE Batch proxy, 8-1
- RE Batch recommendations, 6-5
- RE data collection, 4-2
- RE Proxy, 4-2
 - usage notes, 4-6
- RE usage, 5-9
 - REProxyBatch, 9-2
- REAP recommendations, 2-8
- REAPI
 - proxy objects, 2-5
 - using, 4-1
- REAPI basic usage, 5-1
- REAPI classes
 - location, 3-1
- REAPI classes and methods, 4-1
- REAPI data collection, 2-3
- REAPI demo, 5-1
- REAPI end users, 2-2
- REAPI EnumType interfaces, 3-2
- REAPI examples, 5-1
- REAPI hot picks, 2-4

- REAPI Overview, 2-1
- REAPI prerequisites, 2-1, 2-4
- REAPI recommendations, 2-4, 2-8
 - making, 2-9
 - obtaining, 2-8
- REAPI sample program, A-1
- REAPI session
 - closing, 2-9
 - starting, 2-6
- REAPI sessions, 2-2
- REAPI supporting classes, 2-6, 3-1
- REAPI usage, 5-1
- REBatchProxy
 - creating, 6-4
- REBatchProxy object
 - removing, 6-5
- recommendations
 - RE Batch API, 6-5
- Recommendation class, 3-12
- recommendation engine usage, 5-9
 - REProxyBatch, 9-2
- RecommendationAttribute interface, 3-7
- RecommendationList class, 3-13
- recommendations, 3-12, 4-6
 - creating, 2-8, 6-5
 - RE Batch API, 6-2, 6-5
 - REAPI, 2-4, 4-4
 - REProxyBatch, 8-2, 8-4
 - returned value, 8-2
 - usage notes, 8-3
- REProxy
 - multiple instances, 5-6
- REProxy object
 - creating, 5-2
 - using, 5-2
- REProxyBatch
 - code sample, 9-2
 - demographic data, 9-4
 - examples, 9-1
 - recommendations, 8-4
 - rules, 8-4
 - usage, 9-1
- REProxyBatch classes
 - location, 8-1
- REProxyBatch overview, 8-1

- REProxyBatch sample program, B-1
- REProxyManager
 - JVM interaction, 5-5
- REProxyManager class, 4-3
- REProxyRT
 - location of classes, 4-2
- REProxyRT objects, 2-5
 - removing, 2-9
- rules, 4-6
 - REProxyBatch, 8-4

S

- sample program
 - REAPI, A-1
 - REProxyBatch, B-1
- scoring
 - customers, 2-9
 - RE Batch API, 6-5
 - visitors, 2-8
- server-side modules, 5-5
- session
 - closing, 2-9
- session creation, 4-6
- session management, 4-3
- sessionful, 2-2, 2-3, 5-3
- sessionful web application, 5-3
- sessionless, 2-2, 2-3, 5-4
- sessionless web application, 5-4
- sessions, 2-2
 - management, 4-3
- Sorting interface, 3-8, 7-7
- standalone java applications, 5-5
- stateful, 2-2
- stateless, 2-2
- supporting classes
 - RE Batch, 7-1
 - REAPI, 3-1

T

- time-based items, 5-11, 9-4
- tuning, 3-13
- TuningSettings class, 3-13, 7-10

U

- uninterrupted REAPI service, 5-7
- usage
 - REAPI, 5-1
- usage notes
 - RE Proxy, 4-6
- User interface, 3-8
- using a proxy, 5-2
- using RE Batch API, 6-2

V

- visitors, 2-2