Oracle® Application Server Personalization

Programmer's Guide

10*g* Release 2 (10.1.2)

B14051-01

July 2005

ORACLE®

Oracle Application Server Personalization Programmer’s Guide 10*g* Release 2 (10.1.2)

B14051-01

Contents

Preface

This manual describes how a Java programmer can use Oracle Application Server Personalization (OracleAS Personalization) 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 OracleAS 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. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers 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 does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Where to Find More Information

Documentation for OracleAS Personalization at the current release consists of the following documents:

- *Oracle Application Server Release Notes*, 10*g* Release 2 (10.1.2), which contains a chapter for each component of Oracle Application Server. The chapter for the OracleAS Personalization component contains platform-specific information, a bug report, and information about any late-breaking changes.
- *Oracle Application Server Personalization User's Guide, release* 10*g* Release 2 (10.1.2).
- *Oracle Application Server Personalization Administrator's Guide,* release 10*g* Release 2 (10.1.2).
- *Oracle Application Server Personalization Programmer's Guide,* release 10*g* Release 2 (10.1.2) (this document). A programmer's manual for programming the recommendation engines in real time and for obtaining bulk recommendations.
- The API classes and methods are also described in Javadoc (Oracle Application Server Personalization API Reference), updated for the current release.

Related Manuals

OracleAS Personalization documentation is a component of the Oracle Application Server 10*g* Release 2 (10.1.2) Documentation Library. See especially:

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

Documentation Formats

Documentation for OracleAS 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

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Recommendation Engine API

Part I describes the OracleAS 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 OracleAS Personalization section of the Oracle Application Server 10*g* Release 2 (10.1.2) Documentation Library.

1 OracleAS Personalization Programming

Oracle Application Server Personalization (OracleAS 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 OracleAS Personalization 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 OracleAS Personalization-style recommendations in bulk mode. The recommendations are written to a table. RE Batch API does not return results in real time. REAPI Batch is described in Part II of this manual.

1.1 OracleAS Personalization API Structure

The two OracleAS Personalization APIs have the same components:

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

1.2 Executing OracleAS Personalization Programs

Before you can execute a program using either OracleAS Personalization API, you must deploy and build an OracleAS Personalization package, as described in the Oracle Application Server Personalization User's Guide.

1.3 Javadoc for the OracleAS Personalization APIs

Detailed descriptions of the OracleAS Personalization APIs are not included in this manual. The API calls are documented by Javadoc; see the Oracle Application Server Personalization API Reference.

2 REAPI Overview

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

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

Appendix A contains a complete example of REAPI use.

OracleAS Personalization 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 OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library.

Note:: REAPI and REAPI Demo are installed on the system where Oracle Application Server is installed.

2.1 REAPI Prerequisites

Before you can use REAPI methods, OracleAS Personalization 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 OracleAS Personalization package before you request any recommendations. Use the OracleAS Personalization Administrative UI to do this.

For detailed information about how to install OracleAS Personalization, see the Oracle Application Server Installation Guide and the Oracle Application Server Personalization Administrator's Guide. For information about how to create and deploy packages, see Oracle Application Server Personalization User's Guide and the online help for the OracleAS Personalization Administrative UI.

2.2 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.

2.2.1 REAPI End Users (Customers and Visitors)

End users (users of a Web site that uses OracleAS Personalization 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.

2.2.2 Web Applications and Sessions

Some Web applications are *stateful*, that is, they maintain 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. OracleAS 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 OracleAS Personalization 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 sessions, OracleAS Personalization then tracks each user session itself. In this case, the OracleAS Personalization 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

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

During the OracleAS Personalization session, the Web application can collect data or request recommendations or both.

2.2.3 REAPI Sessionful Web Applications

If the Web application is sessionful, OracleAS Personalization 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.

2.2.4 REAPI Sessionless Web Applications

If the Web application is sessionless, the recommendation engine (RE) will maintain OracleAS Personalization sessions by itself. An OracleAS Personalization 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.

2.2.5 REAPI Data Collection

OracleAS Personalization 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 OracleAS Personalization 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. OracleAS Personalization 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 Oracle Application Server 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.

2.2.6 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

2.2.7 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 OracleAS Personalization administrator in the Mining Table Repository (MTR); each hot picks group is identified by a (long) integer.

2.3 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.

Oracle Application Server Personalization User's Guide and the online help for the OracleAS Personalization Administrative UI explain how to perform these steps.

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

You can then either collect data or get recommendations. You cannot get recommendations until there is an existing deployed package, which is created using the OracleAS Personalization 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 OracleAS Personalization 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 Section 5.9, "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 a user can afford to buy. The MTR_CUSTOMER table in the MTR schema (see Table 4-8 in the *Oracle Application Server Personalization Administrator's Guide*) provides several fields for customer income. These and other fields in the MTR_CUSTOMER table can be used as criteria for making recommendations.

If the users of the Web site live in several countries (for example, the Web site sells items in Japan and India), see Section 5.8, "Handling Multiple Currencies" in Chapter 5.

2.3.1 REAPI Demo Program

OracleAS Personalization 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 OracleAS Personalization is correctly installed.

After you have installed OracleAS Personalization, 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 Oracle Application Server is installed. The REAPI test site is displayed.

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

For information about how run the demo, see the Oracle Application Server 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.

2.3.2 Creating REProxyRT Objects

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

In a Web application environment, it is better to create all required proxy objects during the initialization stage to avoid proxy startup overhead when making the first recommendation request. See Chapter 5 for more information about proxies.

2.3.3 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 OracleAS Personalization will start an internal session for a given user when the Web application makes the first REAPI call.)

OracleAS Personalization 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 OracleAS Personalization assigns the same session ID to both users. OracleAS Personalization treats them as a single user. After the OracleAS Personalization session times out, OracleAS Personalization 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.

2.3.4 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.

2.3.5 Collecting Data for REAPI Recommendations

OracleAS Personalization generates recommendations based on data describing past or current user behavior or both.

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();

addItems();

removeItem();

removeItems();
```

These methods add information to or remove information from the OracleAS Personalization Recommendation Engine (RE) and its cache for a specified OracleAS

Personalization internal session ID. The session ID is stored in the `IdentificationData` passed in the REAPI method.

OracleAS Personalization Data Caching

When one of the OracleAS Personalization data collection methods (`addItem()` or `addItems()`) 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. An overloaded `createProxy` method allows the archive interval to be set. 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.

2.3.6 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

OracleAS Personalization 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 invoked. 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, current visitor session data is used in conjunction with the historical information collected for previous visitors and customers. 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. (OracleAS

Personalization 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 of customers can also be used for scoring.

The OracleAS Personalization Mining Table Repository (MTR) contains historical rating, transactional data, and navigational data stored in both detailed and aggregated formats for the customer. 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.

2.3.7 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.

2.3.8 Closing an REAPI Session

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

In a sessionless Web application, the OracleAS Personalization 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 Oracle Application Server Personalization Administrator's Guide for information about configuration parameters.

2.3.9 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. Any active sessions will eventually timeout based on the setting in the RE_CONFIGURATION table in the RE schema.

See detailed discussion in Chapter 5 for different usage models.

3

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 OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library.

3.1 Ratings in OracleAS Personalization

Ratings in OracleAS Personalization 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. OracleAS Personalization algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

3.2 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;
```

3.3 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`

3.3.1 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 Section 3.4.3, "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
```

3.3.2 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 what information is used to influence recommendations. A Web application can choose what data to take into account when making recommendations using this interface. For example, `Enum.DataSource.DEMOGRAPHIC` specifies that demographic data should be considered in the production of recommendations. The class Section 3.4.2, "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;
```

Note: For cross-sell recommendations, navigation data can only predict navigation behavior and purchasing data can only predict purchasing behaviour. Only navigation and purchasing data source types are supported for cross-sell recommendations.

3.3.3 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 Section 3.4.3, "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;
```

3.3.4 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` specifies the meaning of the recommendation values, or, in other words, what you are interested in predicting. See more details at Section 4.2.6.2, "Meaning of Returned Value for Recommendations". Compare this to `DataSource`, which specifies the type of data used to influence recommendations.

`InterestDimension` takes on the following values:

- `Enum.InterestDimension.NAVIGATION`

  Predict how likely a visitor or customer is likely to click on items.
- `Enum.InterestDimension.PURCHASING`

  Predict how likely a visitor or customer will purchase items.
- `Enum.InterestDimension.RATING`

  Predict how a visitor or customer will rate an item.

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

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

See Also: Section 3.4.9, "RecommendationList Class" and Section 3.4.10, "TuningSettings Class" in this chapter.

3.3.5 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 Section 3.4.10, "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;
```

3.3.6 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 Section 3.4.10, "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;
```

3.3.7 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 Section 3.4.10, "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;
```

3.3.8 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 Section 3.4.1, "ContentItem Class" and Section 3.4.8, "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;
```

3.3.9 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 Section 3.4.1, "ContentItem Class" and Section 3.4.8, "RecommendationContent Class" later in this chapter.

`Sorting` takes on the following values:

- `Enum.Sorting.NONE`
- `Enum.Sorting.DESCENDING`

- `Enum.Sorting.ASCENDING`

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

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

3.3.10 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 Section 3.4.4, "IdentificationData Class" later in this chapter.

`UserType` takes on the following values:

- `Enum.User.CUSTOMER`
- `Enum.User.VISITOR`

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

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

3.4 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 OracleAS Personalization.

3.4.1 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. Section 3.4.8, "RecommendationContent

Class", described later in this chapter, is any array of items of type `ContentItem`; the description of Section 3.4.8, "RecommendationContent Class" explains how sorting order works when multiple orders are specified.

This class contains the following methods:

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

3.4.2 DataItem 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 `addItems()`.

There are two kinds of methods provided with this class:

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

3.4.3 FilteringSettings Class

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

OracleAS Personalization supports category filtering only.

There are three kinds of methods provided with this class:

- A constructor for `FilteringSettings`
- Methods that set the attribute 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 settings cannot be used with cross-sell methods:

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

3.4.4 IdentificationData Class

Identifies the user, the session, or both.

There are two kinds of methods provided with this class:

- Methods that create `IdentificationData` instances
  - `createSessionful(String appSessionID, UserType userType)`
  - `createSessionless(String appSessionID, UserType userType)`
- Methods that return attribute values:
  - `getUserID()`
  - `getAppSessionID()`
  - `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, OracleAS Personalization 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 have the given ID.

3.4.5 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 `ID`s must be unique within a given `type`, but different `type`s can have the same `ID`s.

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

3.4.6 ItemDetailData Class

This class is created internally by OracleAS Personalization 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 Section 3.4.7, "Recommendation Class" later in this chapter for more details.

There are three methods:

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

3.4.7 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()`

3.4.8 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, and so forth.

3.4.9 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.

3.4.10 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()`

4 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 OracleAS Personalization section of the Oracle Application Server 10*g* 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.

All methods described in this chapter are public.

4.1 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)

4.2 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 Oracle Application Server is installed.

4.2.1 RE Proxy Creation and Management

`REProxyManager` handles a pool of `REProxyRT` instances. Using multiple `REProxyRT` instances within a Web server, such as Oracle 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.

4.2.1.1 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.

4.2.1.2 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:

- `createProxy`
- `getProxy`
- `destroyAllProxies`
- `destroyProxy`

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

4.2.2 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.

4.2.3 RE Proxy Session Management

The following methods manage sessions:

- `createCustomerSession`
- `createVisitorSession`
- `closeSession`

4.2.4 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:

- `addItem`
- `addItems`
- `removeItem`
- `removeItems`

4.2.5 Re Proxy Customer Registration

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

- `setVisitorToCustomer`

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

4.2.6 RE Proxy Recommendations

The following methods obtain and manage recommendations:

- `rateItem`
- `rateItems`
- `recommendTopItems`
- `recommendBottomItems`
- `recommendFromHotPicks`
- `recommendCrossSellForItem`
- `recommendCrossSellForItems`
- `crossSellForItemFromHotPicks`
- `crossSellForItemsFromHotPicks`
- `selectFromHotPicks`

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

The methods that return recommendations do 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 the recommendation methods (such as `recommendTopItems`) 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 OracleAS Personalization application. For information about how to create taxonomies, see Oracle Application Server Personalization Administrator's Guide.

4.2.6.1 Ratings in OracleAS Personalization

Ratings in OracleAS Personalization 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. OracleAS Personalization algorithms use these assumptions, so it is important that ratings are in ascending order of goodness. Note that sorting in ascending or descending order is possible on recommendations containing ratings.

4.2.6.2 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.

4.3 Rules and Recommendations

OracleAS Personalization uses rule tables stored in the RE to generate the recommendations requested by the recommendation methods. The rule tables are created in the MOR when a package is built and deployed to the RE. 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.

4.4 RE Proxy Method Usage Notes

For detailed descriptions of these methods, see the OracleAS Personalization Javadoc included in the OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library. This section provides an overview of the methods and how to use them.

4.4.1 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 OracleAS Personalization. (For more information about data synchronization, see the administrator's guide.) OracleAS Personalization 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.

4.4.2 Data Collection

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

4.4.2.1 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 OracleAS Personalization 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 RE database.

Data collected in the RE is automatically written to the MTR, if the RE is configured to do so.

4.4.2.2 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. In the applied scenario where a customer returns an item after purchasing it, this item will have to be removed from the MTR or source sales database. Similarly, if a customer abandons the purchase process before fully completing the process, the items selected for purchase will have to be removed from the MTR.

4.4.3 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. (Example provided in Section 4.4.3.1, "Cache Size".)
3. Test.
4. Adjust the archive interval.

4.4.3.1 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 performance increasingly worsens above that size. 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 3.4 MB (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 (assuming a safety factor of 1.5 to ensure that no data is dropped): 100 * 340 * 1.5 * 20, which is approximately 2 MB.

4.4.3.2 Interval

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

1. Data collection volume and speed: The more frequent the data collection and the larger the volume of data collected, the shorter the 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.

4.4.4 Cross-Sell Methods

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

For cross-sell recommendations, interest dimension must be the same as that of the data source type. Data source type must be either navigation or purchasing. No other types are supported.

The following filtering settings *cannot* be used with these methods:

- `setCategoryLevelFiltering`
- `setCategorySubtreeFiltering`

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

4.4.5 Proxy Destruction

Destroy proxy objects with caution. `REProxyRT` objects are shared by many clients; therefore, destruction of a proxy may interrupt recommendation services. For Web applications, `REProxyRT` objects should be treated as part of the server services; they should not be destroyed unless it is absolutely necessary. Note that there is one `REProxyRT` object per JVM. 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`.

5 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 OracleAS Personalization.

5.1 REAPI Demo

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

After you have installed OracleAS Personalization, 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 Oracle Application Server is installed. The REAPI test site is displayed.

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

For information about how to install and run the demo, see the Oracle Application Server Personalization User's Guide.

5.2 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.

5.2.1 Create an REProxy Object

This section illustrates basic `REProxy` usage; for more information about `REProxy` and other ways to use it, see Section 5.5, "REProxyManager Interaction with JVM" and Section 5.6, "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,
                                     cacheSixe,
                                     interval);
  ...
} catch (Exception e) {
    // exception handling here
}
```

5.2.2 Use the Proxy

After you've created an `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 Section 5.3, "Sessionful Web Application Outline" or Section 5.4, "Sessionless Web Application Outline" later in this chapter for details.

5.3 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 Section 5.2.1, "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();
   ```

5.4 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 Section 5.2.1, "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. See the example in the similar step in the previous section (Section 5.3, "Sessionful Web Application Outline").
4. Make other REAPI calls as required.

5.5 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

5.5.1 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.

5.5.2 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 Oracle 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.

5.6 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

5.6.1 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;
}
```

5.6.2 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;
   }
}
```

5.6.3 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;
   }
```

5.7 Extracting Individual Recommendations

To extract individual recommendations, use the `getAttributes` method of the `Recommendation` class rather than attempt to extract the individual recommendations from the array. The following code shows this usage:

```
protected void printRecList(RecommendationList recList, String ID)
 {
   if (recList.getNumberOfRecommendations() <= 0 )
   {
     System.out.println("No Recommendation Returned");
   }
   else {
     System.out.println("The Following " + recList.getNumberOfRecommendations() +
                  " Recommendations were returned for user " + ID);
     System.out.println("ItemID\t\tItemType\t\tPrediction");
     StringBuffer id = new StringBuffer(30);
     StringBuffer predctn = new StringBuffer(30);
     StringBuffer typ =  new StringBuffer(30);

     for ( int i = 0; i < recList.getNumberOfRecommendations(); i++ )
     {
       Recommendation reck = recList.getRecommendation(i);
       ItemDetailData[] idd = reck.getAttributes();
       for (int j = 0; j < idd.length; j++)
       {
         if (idd[j].getAtribute() == Enum.RecommendationAttribute.ID){
             id.replace(0,idd[j].getValue().length(),idd[j].getValue());}
         if (idd[j].getAtribute() == Enum.RecommendationAttribute.PREDICTION){
             predctn.replace(0,idd[j].getValue().length(),idd[j].getValue());}
         if (idd[j].getAtribute() == Enum.RecommendationAttribute.TYPE){
             typ.replace(0,idd[j].getValue().length(),idd[j].getValue());}
       }
     System.out.println(id + "\t\t" + typ + "\t\t" + predctn);
     }
   }
 }
```

5.8 Handling Multiple Currencies

OracleAS Personalization stores currency data in the demographic table (for example, someone’s income) as numbers; that is, OracleAS Personalization 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 OracleAS Personalization before OracleAS Personalization 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 OracleAS Personalization before OracleAS Personalization creates recommendations.

5.9 Recommendation Engine Usage

OracleAS 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 Section 5.6.3, "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 Oracle Application Server Containers for J2EE (OC4J) instance and that OC4J instance is configured to use one RE at all times. The application must route users to "their" OC4J instance and configure 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 `OC4J.properties` file, or by being explicitly coded in the Web applications, or by some other means.

2. Allow any 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 failover 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 solution 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.

5.10 Using Demographic Data

Demographic data that is used to produce recommendations is populated into the MTR_CUSTOMER table. This table includes several predefined fields and 50 fields for generic attributes. The first 25 of these attributes are of type VARCHAR, and the second 25 attributes are of type NUMBER. These generic attributes can be populated or mapped to any table column. The generic attributes are treated as categorical or numerical data only. If a mapped column is a DATE type, it should be converted to a NUMBER type using the TO_NUMBER function.

To improve model performance, OracleAS Personalization MTR data should be binned to reduce attribute cardinality, that is, the number of distinct values in a column. This is particularly necessary for numeric columns where there can be many values along a continuous range. The binning parameters are specified in the MTR_BIN_BOUNDARY table.

As a categorical binning example, consider the case where the two-letter United States state abbreviation is added as one of the generic attributes. The entire United States can be divided into four main regions: western states, mid-western states, southern states, and eastern states. These regions can correspond to bin numbers of 1, 2, 3, and 4, respectively. Hence, bin 1 contains CA, OR, WA, AZ, and other western states. Bin 3 contains TX, MS, LA, and the other southern states. Bin 2 contains the mid-western states, and bin 4 contains the eastern states.

As a numerical binning example, consider the attribute "age" where the values may range from 1 to 100. We may define bin 1 as ages 1 through 25, bin 2 as ages 26 through 50, and bin 3 as ages 51 through 100.

> **See Also:** The OracleAS Personalization schemas chapter in the *Oracle Application Server Personalization Administrator's Guide* for MTR_BIN_BOUNDARY details.

5.11 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", for example. 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 OracleAS Personalization RE Batch API (Recommendation Engine Batch Application Programming Interface) enables a web application written in Java to request OracleAS 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 OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library. Note that many of the batch methods and classes are REAPI methods and classes.

.

6 RE Batch API Overview

The OracleAS Personalization RE Batch API (Recommendation Engine Batch Application Programming Interface) enables an application written in Java to request OracleAS 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 OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library

Note: RE Batch API is installed on the system where Oracle Application Server is installed.

6.1 RE Batch API Prerequisites

Before you can use RE Batch API methods, OracleAS Personalization must be installed and the appropriate tables must be created and populated. Your database tables must be converted to the OracleAS Personalization schemas. It is important that the OracleAS Personalization 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 OracleAS Personalization package must have been built and deployed. Use the OracleAS Personalization administrative interface to do this. For an example of how to create and deploy a package, see Oracle Application Server 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.

6.2 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.

6.2.1 RE Batch API End Users (Customers)

End users (users of a Web site that uses OracleAS Personalization 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.

6.2.2 RE Batch API Recommendations

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

6.3 Using RE Batch API

Before you execute an RE Batch program, you must

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

6.3.1 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 Oracle Application Server Personalization Administrator's Guide and the online help for the OracleAS Personalization Administrative GUI explain how to perform these steps.

6.3.1.1 Customer Profile Data

Customer profile data resides in the MTR.

6.3.1.2 Deploy a Package to an RE

You cannot get recommendations until there is an existing deployed package, which is created using the OracleAS Personalization 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 OracleAS Personalization 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 Section 9.2, "Recommendation Engine Usage" in Chapter 9.

Note: If you try to deploy a package to 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 Section 5.8, "Handling Multiple Currencies".

6.3.2 Sample RE Batch API Usage

OracleAS Personalization 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.

6.3.3 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.

6.3.4 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.

6.3.5 Converting Data for RE Batch API

OracleAS Personalization 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.

6.3.6 Managing Customer Profiles for RE Batch API

OracleAS Personalization 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:

- `loadCustomerProfiles()`
- `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.

6.3.7 Getting RE API Batch Recommendations

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

- `crossSellForItem()`
- `rateItem()`
- `recommendTopItems()`

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

6.3.7.1 Ratings in OracleAS Personalization

Ratings in OracleAS Personalization 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. OracleAS Personalization algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

6.3.7.2 Creating Recommendations

OracleAS Personalization 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 OracleAS Personalization 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.

6.3.8 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.

6.3.9 Removing the REBatchProxy Object

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

7 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 Section 7.4.2, "FilteringSettings Class", Section 7.4.5, "TuningSettings Class", and Section 7.4.4, "Location Class" instances.

All methods described in this chapter are public.

This chapter does not contained detailed descriptions of any of the classes. For detailed information, see the Javadoc in the OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library.

The supporting classes are divided into two categories:

- `EnumType` interfaces
- Other supporting classes

7.1 Ratings in OracleAS Personalization

Ratings in OracleAS Personalization 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. OracleAS Personalization algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

7.2 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;
```

7.3 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`

7.3.1 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 Section 7.4.2, "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;
```

7.3.2 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 OracleAS Personalization performs certain operations. For example, `Enum.DataSource.DEMOGRAPHIC` specifies that demographic data. The method Section 7.4.1, "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 OracleAS Personalization.

`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;
```

7.3.3 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 Section 7.4.5, "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;
```

7.3.4 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 Section 7.4.5, "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;
```

7.3.5 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 Section 7.4.5, "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;
```

7.3.6 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 Section 7.4.5, "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;
```

7.3.7 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 Section 7.4.1, "DataItem Class" later in this chapter.

`Sorting` takes on the following values:

- `Enum.Sorting.NONE`
- `Enum.Sorting.DESCENDING`

- `Enum.Sorting.ASCENDING`

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

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

7.4 Other RE Batch API Supporting Classes

The other supporting classes are

- `DataItem`
- `FilteringSettings`
- `Location`
- `TuningSettings`

7.4.1 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()`

7.4.2 FilteringSettings Class

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

Release 2 of OracleAS Personalization 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 will all methods. In particular, the following filtering setting cannot be used with the cross-sell methods:

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

7.4.3 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 `ID`s must be unique within a given `type`, but different `type`s can have the same `ID`s.

There are two kinds of methods provided with this class:

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

7.4.4 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
  - `get DatabaseURL()`
  - `getDatabaseAlias()`
  - `getSchemaName()`
  - `getTableName()`
  - `getUserName()`
  - `getPassword()`

7.4.5 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()`

8 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 OracleAS Personalization section of the Oracle Application Server 10*g* Documentation Library.

All methods described in this chapter are public.

8.1 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.

8.2 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 Oracle Application Server is installed.

8.2.1 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.

8.2.1.1 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

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

8.2.2 REProxyBatch Recommendations

The following methods obtain recommendations:

- `crossSellForItems`
- `rateItem`
- `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.

8.2.2.1 Ratings in OracleAS Personalization

Ratings in OracleAS Personalization 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. OracleAS Personalization algorithms use these assumptions, so it is important that ratings are in ascending order of goodness.

8.2.2.2 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`, 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.

8.2.2.3 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[])`

8.2.2.4 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 OracleAS Personalization application. For information about how to create taxonomies, see Oracle Application Server Personalization Administrator's Guide.

8.3 REProxyBatch Rules and Recommendations

OracleAS Personalization 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.

9 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 OracleAS Personalization.

9.1 REProxyBatch API Basic Usage

The REBatchProxy methods described in Chapter 8 permit you to write Java programs that generate recommendations.

Note: The RE Batch API classes are installed on the system where Oracle Application Server is installed. The tables that they use are installed on a different system (the system where Oracle9*i* 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.

Refer to Appendix B, "REProxyBatch Sample Program" for code samples on how to obtain recommend top and cross-sell recommendations.

9.2 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 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 them merge the recommendation tables, if desired.

See Also:

- Section 5.8, "Handling Multiple Currencies"
- Section 5.10, "Using Demographic Data"
- Section 5.11, "Handling Time-Based Items"

A
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 Oracle Application Server is installed.

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

```
// Copyright (c) 2001, 2005 Oracle Corp
////////////////////////////////////////////////////////////////////////
//
// Test program exercises the functionality of
// REAPI.
//
// Step 1 creates a unique session ID
// Step 2 creates a proxy instance
// Step 3 creates a session
// Step 4 creates settings data (IdentificationData, TuningSettings,
//        FilteringSettings, hotPick list, item list)
// Step 5 gets recomendations and ratings
// Step 6 closes session
// Step 7 destroys the proxy and flushes data cache
////////////////////////////////////////////////////////////////////////

import oracle.dmt.op.re.base.DataItem;
import oracle.dmt.op.re.base.Enum;
import oracle.dmt.op.re.base.FilteringSettings;
import oracle.dmt.op.re.base.TuningSettings;
import oracle.dmt.op.re.reapi.rt.IdentificationData;
import oracle.dmt.op.re.reapi.rt.REProxyManager;
import oracle.dmt.op.re.reapi.rt.REProxyRT;
import oracle.dmt.op.re.reapi.rt.RecommendationContent;
import oracle.dmt.op.re.reapi.rt.RecommendationList;
import oracle.dmt.op.re.reexception.BadDBConnectionException;
import oracle.dmt.op.re.reexception.ErrorExecutingRE;
import oracle.dmt.op.re.reexception.InvalidIDException;

/**
 * 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)
  {
    long lStart;
    long lTotal = 0;
    String sProxyName = "REP1";
    String sdbURL = "jdbc:oracle:thin:@host:1521:sid"; // sdbURL must be correct
                                                       // for your installation
    String sUser = "RE";
    String sPass = "REPW";

    int cSize = 3000;   // Kbytes
    int interval = 10000;   // in millisec
    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.LOW,
        Enum.ProfileDataBalance.BALANCED,
    Enum.ProfileUsage.INCLUDE);

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

  long[] m_catList = {1,2,3,4,5,83,48,18,93,83};

  int taxonomy=1;
  FilteringSettings filters =
              new FilteringSettings(taxonomy);
  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",
        449,
        Enum.DataSource.NAVIGATION,
        "1");
 items[1] = new DataItem(
        "MOVIE",
        282,
        Enum.DataSource.PURCHASING,
        "1");
 items[2] = new DataItem(
        "MOVIE",
        122,
        Enum.DataSource.NAVIGATION,
        "1");
 items[3] = new DataItem(
        "MOVIE",
        199,
        Enum.DataSource.NAVIGATION,
        "1");

// Step 5: Get recomendations and ratings and print them out.
// Note use of toString.
try{
System.out.println("\n################ 1 recommendBottomItems" +
                                        "################");
 //Call recommendBottonItems
 RecommendationList es1 = proxy.recommendBottomItems(
    idData,
    10,
    tunings,
    filters,
    recContent);
System.out.println("Number of Recommendations: " +
                    es1.getNumberOfRecommendations());
System.out.println(es1.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################ 2 rateItems ################");
//Call rateItems
```

```
 RecommendationList es2 = proxy.rateItems(
     idData,
     items,
     1,
     tunings,
     recContent);
 System.out.println("Number of Recommendations: " +
                     es2.getNumberOfRecommendations());
 System.out.println(es2.toString());
 } catch(ErrorExecutingRE e) {
     e.printStackTrace();
 }

 try{
 System.out.println("\n################ 3 selectFromHotPicks " +
                                           "################");
 //call selectFromHotPicks
 RecommendationList es3 = proxy.selectFromHotPicks(
     idData,
     6,
     hotPickGroups,
     tunings,
     filters,
     recContent);
 System.out.println("");
 System.out.println("Number of Recommendations: " +
                     es3.getNumberOfRecommendations());
 System.out.println(es3.toString());
 } catch(ErrorExecutingRE e) {
     e.printStackTrace();
 }

 try{
 System.out.println("\n################ 4 crossSellForItemFromHotPicks " +
                                                     "################");
 //Call crossSellForItemFromHotPicks
 RecommendationList es4 = proxy.crossSellForItemFromHotPicks(
     idData,
     items[3],
     10,
     hotPickGroups,
     tunings,
     filters,
     recContent);
 System.out.println("");
 System.out.println("Number of Recommendations: " +
                     es4.getNumberOfRecommendations());
 System.out.println(es4.toString());
 } catch(ErrorExecutingRE e) {
     e.printStackTrace();
 }

 try{
 System.out.println("\n################ 5 recommendCrossSellForItem " +
                                                  "################");
//Call recommendCrossSellForItem
 RecommendationList es5 = proxy.recommendCrossSellForItem(
     idData,
     items[0],
     10,
    tunings,
    filters,
    recContent);
System.out.println("");
System.out.println("Number of Recommendations: " +
                    es5.getNumberOfRecommendations());
System.out.println(es5.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################ 6 recommendCrossSellForItems " +
                                                  "################");
 RecommendationList  es6 = proxy.recommendCrossSellForItems(
    idData,
    items,
    10,
    tunings,
    filters,
    recContent);
System.out.println("");
System.out.println("Number of Recommendations: " +
                    es6.getNumberOfRecommendations());
System.out.println(es6.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################ 7 crossSellForItemsFromHotPicks " +
                                                     "################");
RecommendationList es7 = proxy.crossSellForItemsFromHotPicks(
    idData,
    items,
    10,
    hotPickGroups,
    tunings,
    filters,
    recContent);
System.out.println("");
System.out.println("Number of Recommendations: " +
                    es7.getNumberOfRecommendations());
System.out.println(es7.toString());
} catch(ErrorExecutingRE e) {
    e.printStackTrace();
}

try{
System.out.println("\n################ 8 rateItem ################");
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################ 9 recommendFromHotPicks " +
                                                "################");
        RecommendationList es10 = proxy.recommendFromHotPicks(
        idData,
        10,
        hotPickGroups,
        tunings,
        filters,
        recContent);
        System.out.println("");
        System.out.println("Number of Recommendations: " +
                           es10.getNumberOfRecommendations());
        System.out.println(es10.toString());
        } catch(ErrorExecutingRE e) {
            e.printStackTrace();
        }

        try{
        System.out.println("\n################ 10 recommendTopItems " +
                                             "################");
        RecommendationList es11 = proxy.recommendTopItems(
        idData,
        10,
        tunings,
        filters,
        recContent);
        System.out.println("");
        System.out.println("Number of Recommendations: " +
                            es11.getNumberOfRecommendations());
        System.out.println(es11.toString());
        } catch(ErrorExecutingRE e) {
            e.printStackTrace();
        }

        } catch(BadDBConnectionException bdbe) {
            bdbe.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();
        }
      }
    }
```

B

REProxyBatch Sample Program

The sample program for `REProxyBatch` 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 OracleAS Personalization.

B.1 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 OracleAS Personalization.

`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`.

B.1.1 RE Batch Sample Program Output

The input item details (for `rateItem` and `crossSellForItem`) are derived from the OracleAS Personalization demo data. But in OracleAS Personalization, 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 (for example, if only some of the items are valid cross-sell candidates.).

B.2 Executing the RE Batch Sample Program

Follow these steps to execute the sample program:

1. Install OracleAS Personalization.
2. The code and data for the sample program is installed into the following directories when you install OracleAS Personalization:
   - 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.

B.3 RE Batch Sample Program Code

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

B.3.1 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=myDBAlas
XSellResult.Schema=User4
XSellResult.Table=XSELL_RESULTS
XSellResult.Username=User5
XSellResult.Password=Password5
```

B.3.2 REBatchTest.java

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

```
// Copyright (c) 2001, 2002, Oracle Corporation.  All rights reserved.

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.sql.SQLException;

import java.util.Enumeration;
import java.util.Properties;
import java.util.StringTokenizer;

import oracle.dmt.op.re.base.Enum;
import oracle.dmt.op.re.base.FilteringSettings;
import oracle.dmt.op.re.base.Item;
import oracle.dmt.op.re.base.TuningSettings;
import oracle.dmt.op.re.reapi.batch.Location;
import oracle.dmt.op.re.reapi.batch.REProxyBatch;
import oracle.dmt.op.re.reexception.BadDBConnectionException;
import oracle.dmt.op.re.reexception.InvalidIDException;
import oracle.dmt.op.re.reexception.NullParameterException;
import oracle.dmt.op.re.reexception.StringTooLargeException;
```

```
/**
 * Class REBatchTest
 * This class demonstrates the use of REProxyBatch API included with Oracle
 * Personalization. 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 a property file supplied as input. The tags used in the input file are
 * listed here as static constants in CAPITAL LETTERS.
 *
 * Before you execute the program, you must edit the property file batchtest.ini
 * to reflect your environment.
 *
 * @author Oracle Corporation
 */
public class REBatchTest
{

  static private Properties m_Props;

  static private boolean m_Verbose;
  static private final String s_Title = "REBatchTest";
  static final boolean debug = true;

  // These tags appear in the input property file.
  static String TOPN = "TopN";
  static String XSELL = "XSell";
  static String RITEM = "RateI";
  static String DSTYPE = "DataSourceType";
  static String INTDIMENSION = "InterestDimension";
  static String PDBALANCE = "ProfileDataBalance";
  static String PUSAGE = "ProfileUsage";
  static String PINDEX = "PersonalizationIndex";
  static String NUMOFITEMS = "NumberOfItems";
  static String PROXYNAME = "ProxyName";
  static String REURL = "REUrl";
  static String REUSERNAME = "REUsername";
  static String REPASSWORD = "REPassword";
  static String URL = "Url";
  static String ALIAS = "Alias";
  static String USNAME = "Username";
  static String PSWORD = "Password";
  static String SCHEMA = "Schema";
  static String TABLE = "Table";
  static String INPUT = "Input";
  static String CUSTPROFILE = "CustProfile";
  static String ITEMID = "ItemID";
  static String ITEMTYPE = "ItemType";
  static String RESULT = "Result";
  static String TAXONOMY = "Taxonomy";
  static String ITEMTAB = "ItemTable";
  static String CATEGORYS = "CategoryList";
  static String CATNODE = "CategoryMember";
  static String DOT = ".";

  // These variables are initialized from the property file.
  static private String m_proxyName; // Name of the REProxyBatch instance
  // Recommendation Engine Details
```

```
static private String m_reUrl;
static private String m_reUserName;
static private String m_rePassword;

// Location of the customer table; the customer table must be created before
// this program can be run.
// Refer to the REProxyBatch document for the expected schema.
static private Location m_inCustTable;
// Location of the items table; the itemstable must be created before this
// program can be run.
// Refer to the REProxyBatch document for the expected schema.
static private Location m_inItemTable;

// Name of the customer profile table created in RE after loadCustomerProfile
// is invoked.
static private String m_custProfile;

/**
* main Entry point
* @param args
**/
public static void main(String[] args)
{
  long lStart;
  long lTotal = 0;
  REProxyBatch proxy;
  // Name of the property file should be passed as an input.
  if (args.length < 1) {
    System.out.println("Usage: REBatchTest FullpathToPropfile");
    System.exit(-1);
  }
  // Initialize class variables using the property file.
  try {
    new REBatchTest(args[0]);
  } catch (Exception ioe) {
    System.out.println("Error in main(): " + ioe.getMessage());
    ioe.printStackTrace();
    System.exit(-1);
  }
  // Start a timer.
  lStart = System.currentTimeMillis();
  try {
    // Initialize a REProxyBatch object
    proxy = new REProxyBatch(m_proxyName,
                             m_reUrl,
                             m_reUserName,
                             m_rePassword);

    // Invoke loadCustomerProfiles.
    System.out.println("Loading customer profile.");
    proxy.loadCustomerProfiles(
      m_inCustTable, // Name of the input customer IDs table
      m_custProfile); // Name of the profile table from loadCustomerProfiles
    System.out.println("Completed loading customer profile, in the table: "
                       + m_custProfile + " in current RE schema.");

    // Initialize inputs needed for recommendTopItems
    // Number of items to recommend (per customer)
    int nRec = getNumOfItems(TOPN);
    // Initialize TuningSettings object from the property file
```

```
TuningSettings tuningT = getTuning(TOPN);
// Initialize FilteringSettings object from the property file
FilteringSettings filterT = getFilter(TOPN);
// Location of the result table
Location resLocT = getLoc(addStr(TOPN, RESULT));
System.out.println("Recommending top items. Customer profile table name:"
                    + m_custProfile );

// Invoke recommendTopItems
proxy.recommendTopItems(
  m_custProfile, // Customer profile table name, from loadCustomerProfiles
  nRec, // Number of recommendations per customer.
  tuningT, // TunningSettings to be used for recommendations
  filterT, // FilteringSettings to be used for recommendations
  resLocT); // Location of the result table
System.out.println("Completed recommendTopItems.");
System.out.println("Result table details: Schema: " +
                    resLocT.getSchemaName() + ", Table Name: "
                    + resLocT.getTableName() + ", Database: "
                    + resLocT.getDBAlias());

// Initialize inputs needed for rateItem
// Read the item to be rated. This single item is rated for all the
// customers in the customer profiles table.
Item item = getItem(RITEM);
// Taxonomy to be used for rating
int nTaxID = Integer.parseInt(m_Props.getProperty(addStr(RITEM, DOT,
                              TAXONOMY)));
// Initialize TuningSettings object from the property file
TuningSettings tuningR = getTuning(RITEM);
// Location of the result table,
Location resLocR = getLoc(addStr(RITEM, RESULT));
System.out.println("Rating item.");
System.out.println("Rating item: " + item.getID() + "," + item.getType() +
                    ". Customer profile table name:" + m_custProfile );
// Invoke rateItem
proxy.rateItem(
  m_custProfile, // Customer profile table name, from loadCustomerProfiles
  item, // Item to be rated
  nTaxID, // Taxonomy to be used for rating
  tuningR, // TunningSettings to be used for rating
  resLocR); // Location of the result table
System.out.println("Completed rateItem.");
System.out.println("Result table details: Schema: " +
                    resLocR.getSchemaName() + ", Table Name: " +
                    resLocR.getTableName() + ", Database: " +
                    resLocR.getDBAlias());

// Invoke purgeCustomerProfiles
// Cross sell is an item based recommendation call. Hence it does not need
// the loaded customer profile table.
System.out.println("Purging customer profile: " + m_custProfile +
                  " from current RE.");
proxy.purgeCustomerProfiles(m_custProfile);

// crossSellForItem
// Initialize inputs needed for rateItem
// Number of cross sell items to be recommended (per input item)
nRec = getNumOfItems(XSELL);
// Initialize TuningSettings object from the property file
```

```
    TuningSettings tuningX = getTuning(XSELL);
    // Initialize FilteringSettings object from the property file
    FilteringSettings filterX = getFilter(XSELL);
    // Location of the result table
    Location resLocX = getLoc(addStr(XSELL, RESULT));
    System.out.println("Recommending cross sell for item. Input table " +
                       "details: Schema: " + m_inItemTable.getSchemaName() +
                       ", Table Name: " + m_inItemTable.getTableName() +
                       ", Database: " + m_inItemTable.getDBAlias());

    // Invoke crossSellForItem
    proxy.crossSellForItem(
      m_inItemTable, // Name of the input items table created by the end user
      nRec, // Number of cross sell items to be recommended per input item
      tuningX, // Tunning settings to be used for recommendations
      filterX, // Filtering settings to be used for recommendations
      resLocX); // Location of the result object
    System.out.println("Completed crossSellForItem.");
    System.out.println("Result table details: Schema: " +
                         resLocX.getSchemaName() + ", Table Name: " +
                         resLocX.getTableName() + ", Database: " +
                         resLocX.getDBAlias());

    // Destroy the proxy object (to free up the database connections).
    System.out.println("Completed REProxyBatch operations. " +
                       "Destroying the proxy.");
    proxy.destroy();
    proxy = null;
  } catch (NullParameterException npe) {
    System.out.println(npe.getMessage());
    npe.printStackTrace();
  } catch (BadDBConnectionException bde) {
    System.out.println(bde.getMessage());
    bde.printStackTrace();
  } catch (SQLException se) {
    System.out.println(se.getMessage());
    se.printStackTrace();
  } catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
  }
}

/**
 * Constructor
 * Read the property file.
 */
public REBatchTest(String sProp)
throws FileNotFoundException, IOException, StringTooLargeException
{
  getConfig(sProp);
}

/**
** All the remaining code in this file deals with reading different values
** from the property file and initializing class variables.
**/

/*
* A helper function returning appended string
```

```
*/
static private String addStr(String s1, String s2) {
  StringBuffer sb = new StringBuffer(s1);
  sb.append(s2);
  return (sb.toString());
}
/*
* A helper function returning appended string
*/
static private String addStr(String s1, String s2, String s3) {
  StringBuffer sb = new StringBuffer(s1);
  sb.append(s2).append(s3);
  return (sb.toString());
}
/*
** Read recommendation engine details from the property file.
**
*/
static private void getConfig(String propFile)
throws FileNotFoundException, SecurityException, IOException,
      StringTooLargeException {
  FileInputStream is = new FileInputStream(propFile);
  String RE = "RE";

  try {
    m_Props = new Properties();
    m_Props.load(is);
    // RE achema access info
    m_proxyName = m_Props.getProperty(PROXYNAME);
    // Recommendation Engine database details
    m_reUrl = m_Props.getProperty(addStr(RE, URL));
    m_reUserName = m_Props.getProperty(addStr(RE, USNAME));
    m_rePassword = m_Props.getProperty(addStr(RE, PSWORD));

    // Input customer table
    m_inCustTable = getLoc(INPUT);
    // Input items table
    m_inItemTable = getLoc(ITEMTAB);
    // Customer profile table name
    m_custProfile = m_Props.getProperty(CUSTPROFILE);
  } catch (SecurityException se) {
    System.err.println("Can’t read the property file: " + propFile + "!");
    m_Props = null;
  }
}
/*
** Read number of items from the property file.
**
*/
static private int getNumOfItems(String sPrix) {
  return (Integer.parseInt(m_Props.getProperty(addStr(sPrix, DOT,
                                                   NUMOFITEMS))));
}
/*
** Read location object from the property file.
**
*/
static private Location getLoc(String sPrix)throws StringTooLargeException{
  String sUrl = null;
  String sAlias = null;
  String sTable = null;
  String sSchema = null;
  String sPsword = null;
  String sUsname = null;
  Enumeration propNames = m_Props.propertyNames();
  boolean bFound = false;

  while (propNames.hasMoreElements()) {
    String name = (String)propNames.nextElement();
    if (name.startsWith(sPrix)) {
      // Url
      if (name.endsWith(URL)) {
        sUrl = m_Props.getProperty(name);
        bFound = true;
      }
      // Alias
      if (name.endsWith(ALIAS))
        sAlias = m_Props.getProperty(name);
      // Schema
      if (name.endsWith(SCHEMA))
        sSchema = m_Props.getProperty(name);
      // Table
      if (name.endsWith(TABLE))
        sTable = m_Props.getProperty(name);
      // Username
      if (name.endsWith(USNAME))
        sUsname = m_Props.getProperty(name);
      // Password
      if (name.endsWith(PSWORD))
        sPsword = m_Props.getProperty(name);
    }
  }
  return (bFound ? new Location(sAlias, sUrl, sSchema, sTable, sUsname,
                              sPsword) : null);
}
/*
** Read item details from the property file.
**
*/
static private Item getItem(String sPrix) throws StringTooLargeException,
                                                  InvalidIDException,
                                                  NullParameterException{
  String sType = null;
  long lID = 0;
  Enumeration propNames = m_Props.propertyNames();
  boolean bFound = false;

  while (propNames.hasMoreElements()) {
    String name = (String)propNames.nextElement();
    if (name.startsWith(sPrix)) {
      // ID
      if (name.endsWith(ITEMID)) {
        lID = Long.parseLong(m_Props.getProperty(name));
        bFound = true;
      }
      // Type
      if (name.endsWith(ITEMTYPE))
        sType = m_Props.getProperty(name);
    }
  }
  return (bFound ? new Item(sType, lID) : null);
}
/*
** Read tuning settings from the property file.
**
*/
static private TuningSettings getTuning(String sPrix) throws
                                                       NullParameterException {
  Enumeration propNames = m_Props.propertyNames();
  Enum.DataSourceType dsType = null;
  Enum.InterestDimensionType iDimension = null;
  Enum.PersonalizationIndexType pIndex = null;
  Enum.ProfileDataBalanceType pdBalance = null;
  Enum.ProfileUsageType pUsage = null;

  while (propNames.hasMoreElements()) {
    String name = (String)propNames.nextElement();
    if (name.startsWith(sPrix)) {
      // datasource type
      if (name.endsWith(DSTYPE)) {
        String sDsType = m_Props.getProperty(name);
        dsType = Enum.DataSourceType.getType(sDsType);
      }
      // interest dimension
      if (name.endsWith(INTDIMENSION)) {
        String sIntDim = m_Props.getProperty(name);
        iDimension = Enum.InterestDimensionType.getType(sIntDim);
      }
      // personalization index
      if (name.endsWith(PINDEX)) {
        String spIndex = m_Props.getProperty(name);
        pIndex = Enum.PersonalizationIndexType.getType(spIndex);
      }
      // profiledata balance
      if (name.endsWith(PDBALANCE)) {
        String sPdBalans = m_Props.getProperty(name);
        pdBalance = Enum.ProfileDataBalanceType.getType(sPdBalans);
      }
      // profile usage
      if (name.endsWith(PUSAGE)) {
        String spUse = m_Props.getProperty(name);
        pUsage = Enum.ProfileUsageType.getType(spUse);
      }
    }
  }
  return (new TuningSettings(dsType, iDimension, pIndex, pdBalance, pUsage));
}
/*
** Read list of categories from the property file.
**
*/
static private long [] getCatList(String str) {
  StringTokenizer stz = new StringTokenizer(str, "-");
  int nE = stz.countTokens();
  if (nE < 1)
    return null;

  long [] lAry = new long [nE];
  int n = 0;
```

```
    while (stz.hasMoreTokens()) {
      String tmp = stz.nextToken();
      tmp.trim();
      lAry[n++] = Long.parseLong(tmp);
    }
    return lAry;
  }
  /*
  ** Read filtering settings from the property file.
  **
  */
  static private FilteringSettings getFilter(String sPrix)
  throws Exception {
    boolean bFound = false;
    Enumeration propNames = m_Props.propertyNames();
    int nTaxoID = 0;
    long [] catList = null;
    Enum.CategoryMembershipType cmType = null;

    while (propNames.hasMoreElements()) {
      String name = (String)propNames.nextElement();
      if (name.startsWith(sPrix)) {
        // Taxonomy
        if (name.endsWith(TAXONOMY))
          nTaxoID = Integer.parseInt(m_Props.getProperty(name));
        if (name.endsWith(CATEGORYS)) {
          String sCList = m_Props.getProperty(name);
          catList = getCatList(sCList);
        }
        if (name.endsWith(CATNODE)) {
          bFound = true;
          String sCmType = m_Props.getProperty(name);
          cmType = Enum.CategoryMembershipType.getType(sCmType);
        }
      }
    }
    return bFound ? FilteringSettings.setFilteringSettings(nTaxoID, catList,
                                                           cmType) : null;
  }

  protected static String getFunctionName(String trace) {
   // now go through the string to find what you want
   trace = trace.substring(trace.indexOf('\n')).trim();
   trace = trace.substring(trace.indexOf(' ') + 1);
   //trace = trace.substring(0, trace.indexOf(')'));
   return (trace);
  }

}
```

Index