Oracle® Database Mobile Server

Developer's Guide

Release 11.3.0.1

E39324-02

April 2014

ORACLE®

Oracle Database Mobile Server Developer's Guide Release 11.3.0.1

E39324-02

Copyright © 2013, 2014, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

3 Managing Synchronization on the Mobile Client

4 Using Mobile Database Workbench to Create Publications

Preface

This preface introduces you to the *Oracle Database Mobile Server Developer's Guide*, discussing the intended audience, documentation accessibility, and structure of this document.

Audience

This manual is intended for application developers as the primary audience and for database administrators who are interested in application development as the secondary audience.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

Use the following manuals as reference when performing administration tasks for either a WebLogic or Glassfish server:

- *Oracle® Fusion Middleware documentation for Oracle WebLogic Server*
- *Oracle® GlassFish Server 3.1 documentation*

Conventions

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.

Convention	Meaning
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
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.
`italic monospace`	Italic monospace type indicates a variable in a code example that you must replace. For example: `Driver=`*`install_dir`*`/lib/libtten.sl` Replace *`install_dir`* with the path of your TimesTen installation directory.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.

1

Overview for Designing Mobile Applications

The following sections provide an introduction to Oracle Database Mobile Server and an overview of the application development process:

1.1 Introduction

Oracle Database Mobile Server facilitates the development, deployment, and management of mobile database applications for a large number of mobile users. A mobile application is an application that can run on mobile devices without requiring constant connectivity to the server. The application requires a small, local database on the mobile device, whose content is a subset of data that is stored in the enterprise data server. This database can be either a Berkeley DB or SQLite database. Modifications made to the client database by the application are reconciled with the back-end server data through data synchronization.

The mobile client in the Oracle Database Mobile Server is a preconfigured component to facilitate running a mobile application. It contains synchronization and software components to manage the device.

Once the application has been developed, it has to be deployed. Deployment sets up the server so that end users can easily install and use the applications. All mobile applications are deployed first to the mobile server after which the applications are downloaded to the appropriate mobile client.

Deployment consists of five major steps:

1. Design the server system to achieve the required level of performance, scalability, security, availability, and connectivity. Oracle Database Mobile Server provides tools such as the Consperf utility to tune the performance of data synchronization. It also provides benchmark data that can be used for capacity planning for scalability. Security measures such as authentication, authorization, and encryption are supported using the appropriate standards. Availability and scalability are also supported by means of load balancing, caching, and the transparent switch-over technologies of the application server and the Oracle database server.

2. Publish the application to the server. This refers to installing all components for an application on the mobile server. The Packaging Wizard tool can be used to publish applications to the mobile server.
3. Provision the applications to the mobile users. This phase includes determining user accesses to applications with a specified subset of data. The Mobile Manager to create users, grant privileges to execute applications, and define the data subsets for them, among others. You can also use the Java API to provision applications.
4. Test for functionality and performance in a real deployment environment. A mobile application system is a complex system involving many mobile device client technologies (operating systems, and form factors), connectivity options (LAN, Wireless LAN, cellular, and wireless data), and server configuration options. Nothing can substitute for testing and performance tuning of the system before it is rolled out. Particular attention should be paid to tuning the performance of the data subsetting queries, as it is the most frequent cause of performance problems.
5. Determine the method for initial installation of applications on mobile devices (application delivery). Initial installation involves installing the mobile client and user applications. The volume of data required to install applications on a mobile device for the first time could be quite high, necessitating the use of either a high-speed reliable connection between the mobile device and the server, or using a technique known as offline instantiation. In offline instantiation, everything needed to install an application on a mobile device is put on a CD or an external storage device and physically mailed to the user. The user uses this media to install the application on the device by means of a desktop machine. Oracle Database Mobile Server provides a tool for offline instantiation.

After deployment, both the application and the data schema may change because of enhancements or defect resolution. The mobile server takes care of managing application updates and data schema evolution. However, the administrator must republish the application and the schema. The mobile server automatically updates the mobile clients that have an older version of the application or the data.

Oracle Database Mobile Server installation provides you with an option to install the mobile server or the Mobile Development Kit. For application development, you need to install the Mobile Development Kit on your development machine. The installation of the mobile server requires an Oracle database in which the mobile repository is created.

1.2 Oracle Database Mobile Server Application Model and Architecture

In the application model, each application defines its data requirements using a publication. A publication is similar to a database schema and it contains one or more publication items. A publication item is like a parameterized view definition and defines a subset of data, using a SQL query with bind variables in it. These bind variables are called *subscription parameters* or *template variables*.

A subscription defines the relationship between a user and a publication. This is analogous to a newspaper or magazine subscription. Accordingly, once you subscribe to a particular publication, you begin to receive information associated with that publication. With a newspaper you receive the daily paper or the Sunday paper, or both. With Oracle Database Mobile Server, the user receives snapshots, and, depending on the subscription parameter values, those snapshots are partitioned with data tailored for the user.

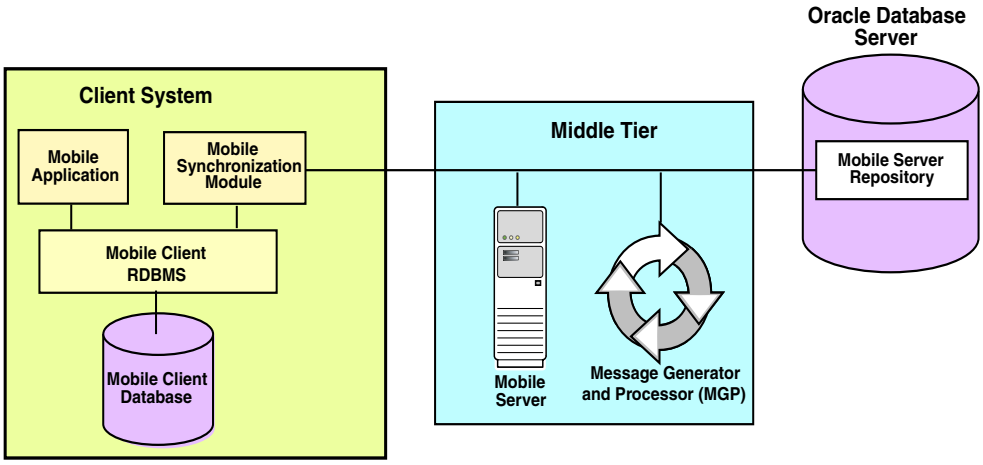
When a user synchronizes the mobile client for the first time, the mobile client creates the client database on the client machine for each subscription that is provisioned to the user. The client database could be a Berkeley DB or SQLite database, which is set in the publication. The mobile client then creates a snapshot in this database for each publication item contained in the subscription, and populates it with data retrieved from the server database by running the SQL query (with all the variables bound) associated with the publication item. Once installed, Oracle Database Mobile Server is transparent to the end user; it requires minimal tuning or administration.

As the user accesses and uses the application, changes made to the client database are captured by the snapshots. At a certain time when the connection to the mobile server is available, the user may synchronize the changes with the mobile server. Synchronization may be initiated by the user using the Mobile Sync application (mSync) directly or by programmatically calling the Mobile Sync API from the application. The Mobile Sync application communicates with the mobile server and uploads the changes made in the client machine. It then downloads the changes for the client that are already prepared by the mobile server.

A background process called the Message Generator and Processor (MGP), which runs in the same tier as the mobile server, periodically collects all the uploaded changes from many mobile users and then applies them to the server database. Next, MGP prepares changes that need to be sent to each mobile user. This step is essential because the next time the mobile user synchronizes with the mobile server, these changes can be downloaded to the client and applied to the client database.

Figure 1–1 illustrates the architecture of Oracle Database Mobile Server applications.

Figure 1–1 Oracle Database Mobile Server Architecture



The following sections describe the separate components of Oracle Database Mobile Server:

- Section 1.2.1, "Mobile Client Database"
- Section 1.2.2, "Mobile Sync"
- Section 1.2.3, "Mobile Server"
- Section 1.2.4, "Message Generator and Processor (MGP)"
- Section 1.2.5, "Mobile Server Repository"
- Section 1.2.6, "Device Manager"

1.2.1 Mobile Client Database

The mobile client uses a client database, which can be either a Berkeley DB or SQLite database, to store the relational data in one or more data files on the file system on the client. You can install either Berkeley DB or SQLite on any device.

> **Note:** All details about both client database options and how to install them are described in detail in Chapter 1, "Mobile Client Overview" in the *Oracle Database Mobile Server Mobile Client Guide*.

The installation requirements for both mobile client databases are listed in the following sections:

- Section 1.2.1.1, "Berkeley DB"
- Section 1.2.1.2, "SQLite"

1.2.1.1 Berkeley DB

If you choose to use Berkeley DB as the client database, then install the Berkeley DB Mobile Client. When you install the Berkeley DB Mobile Client, the following components get installed: a Berkeley DB client database used to store application data; a Sync Engine for managing synchronization between the client database and the server repository; a DMAgent for managing the client device administrative actions.

1.2.1.2 SQLite

If you choose to use Sqlite as the client database, then install the SQLite Mobile Client. When you install the SQLite Mobile Client, the following components get installed: a SQlite client database used to store application data; a Sync Engine for managing synchronization between the client database and the server repository; a DMAgent for managing the client device administrative actions. All details about SQLite are documented on the SQLite Web site at `http://www.sqlite.org/`.

1.2.2 Mobile Sync

Mobile Sync (msync) is a small footprint application that resides on the mobile device. Mobile Sync enables you to synchronize data between handheld devices, desktop and laptop computers and Oracle databases. Mobile Sync authenticates locally, collects changes from the mobile client database and sends them to the server, where the user is authenticated before the changes are uploaded.

Use the `msync` executable for Mobile Sync.

Mobile Sync synchronizes the snapshots in Oracle Database Mobile Server with the data in corresponding Oracle data server. These snapshots are created by the mobile server for each user from the publication items associated with a mobile application. The mobile server also coordinates the synchronization process.

The Mobile Sync application communicates with the mobile server using any of the supported protocols, such as HTTP or HTTPS. When called by the mobile user, the Mobile Sync application collects the user information and authenticates the user with the mobile server. It collects the changes made from the snapshot change logs and uploads these changes to the mobile server. It then downloads the changes for the user from the mobile server and applies them to the mobile server.

In addition to this basic function, the Mobile Sync application can also encrypt, decrypt, and compress transmitted data.

When you install the Mobile Development Kit, the Mobile Sync application is also installed on your development machine. The mobile server also installs the Mobile Sync on the client machine as part of application installation.

Unlike base tables and views, snapshots cannot be created in Oracle Database Mobile Server by using SQL statements. They can only be created by the mobile server based on subscriptions which are derived from publication items associated with an application.

1.2.3 Mobile Server

The installation of the mobile server requires an Oracle database to be running. You can use an existing test database as well. The mobile server stores its metadata in this database.

The mobile server provides the interface between the mobile infrastructure and the enterprise database. Most administration tasks are accomplished through the mobile server Web application—the Mobile Manager.

The mobile server provides the following features.

- application publishing
- application provisioning
- application installation and update
- data synchronization

The Mobile Manager application provides the capability to manage users, devices, publications and applications. This utility can provide the following:

- Monitors and manages synchronization between the client data store and the enterprise data store.
- Sends administrative commands to the mobile clients. These commands capture data and logs from the client or instruct the client to carry out necessary tasks. For example, the Mobile Manager could send a command to a client to perform synchronization or to remove the entire client data store, if a device may have been compromised.

> **Note:** You can accomplish the same tasks as the Mobile Manager with the Application Programming Interfaces (APIs).

As with any Web server tier, the mobile server may be configured within a farm for improved performance within the mobile infrastructure. This enables the use of a load balancer, such as the balancer included with Oracle WebLogic, or with one provided by a 3rd party vendor. The mobile server is designed to be fully integrated with WebLogic to take advantage of the features within WebLogic.

> **Note:** As the mobile server is a Web-based environment, it is important to design for a proper security environment as for any Web server.

The mobile server has two major modules called the Resource Manager and the Consolidator Manager. The Resource Manager is responsible for application publishing, application provisioning, and application installation. The Consolidator Manager is responsible for data and application synchronization.

Application publishing refers to uploading your application to the mobile server so that it can be provisioned to the mobile users. Once you have finished developing your application, you can publish it to the mobile server.

Application provisioning is concerned with creating subscriptions for users and assigning application execution privilege to them. Application provisioning can also be done in one of two ways.

- Using the administration tool called the Mobile Manager, you can create users and groups, create subscriptions for users by assigning values to subscription parameters, and give users or groups privileges to use the application.
- Using the Resource Manager API, you can programmatically perform the above tasks.

End users install mobile applications in two steps.

1. As the mobile user, browse the setup page on the mobile server and choose the setup program for the platform you want to use.
2. Run the Mobile Sync (mSync) command on your mobile device, which prompts for the mobile user name and password. The Mobile Sync application communicates with the Consolidator Manager module of the mobile server and downloads the applications and the data provisioning for the user.

After the installation of the applications and data, you can start using the application. Periodically, use msync or a custom command to synchronize your local database with the server database. This synchronization updates all application data that have changed.

1.2.4 Message Generator and Processor (MGP)

The Consolidator Manager module of the mobile server uploads the changes from the client database to the server, and it downloads the relevant server changes to the client. But it does not reconcile the changes. The reconciliation of changes and the resolution of any conflicts arising from the changes are handled by MGP. MGP runs as a background process which can be controlled to start its cycle at certain intervals.

Note: The mobile infrastructure may allow for multiple mobile servers to be configured within a farm. However, there may only be one MGP application utilized for the entire farm.

Each cycle of MGP consists of two phases: Apply and Compose.

The Apply Phase

In the apply phase, MGP collects the changes that were uploaded by the users since the last apply phase and applies them to the server database. For each user that has uploaded his changes, the MGP applies the changes for each subscription in a single transaction. If the transaction fails, MGP logs the reason in the log file and stores the changes in the error file.

The Compose Phase

When the apply phase is finished, MGP goes into the compose phase, where it starts preparing the changes that need to be downloaded for each client.

Applying Changes to the Server Database

Because of the asynchronous nature of data synchronization, the mobile user may sometimes get an unexpected result. A typical case is when the user updates a record that is also updated by someone else on the server. After a round of synchronization, the user may not get the server changes.

This happens because the user's changes have not been reconciled with the server database changes yet. In the next cycle of MGP, the changes are reconciled with the server database, and any conflicts arising from the reconciliation are resolved. Then a new record is prepared for downloading the changes to the client. When the user synchronizes again (the second time), the user gets the record that reflects the server changes. If there is a conflict between the server changes and the client changes, the user gets the record that reflects either the server changes or the client changes, depending on how the conflict resolution policy is defined.

1.2.5 Mobile Server Repository

The mobile server repository contains all the application data as well as all information needed to run the mobile server. The repository contains the repository schema under which all the data mapping and internal tables utilized to maintain data synchronization exist. This schema also stores the application, application tables and its data published for use with a mobile client.

The information is normally stored in the same database where the application data resides. The only exception to this is in cases where the application data resides in a remote instance and there is a synonym defined in the mobile server to this remote instance.

The repository contains some internal tables that the mobile server uses to perform its functions. You may query these tables to gain more details about the current state of the environment; however, most of the information needed from these tables is already accessible from the Mobile Manager. You should never alter any of the internal tables and their contents unless explicitly directed to by Oracle Support Services or Oracle Development.

Administration, backup, and recovery of the repository are no different then for any other Oracle database requiring standard Database Administrator (DBA) skills

Changes to the repository should only be made using the Mobile Manager or the Resource Manager API.

1.2.6 Device Manager

The Device Manager manages client devices. On install of the mobile client, the Device Manager registers a device with the mobile server. The Device Manager invokes the update executable after synchronization completes to determine if any mobile application updates are available, then downloads and installs these application updates to a mobile client. You can request—through the Mobile Manager—that certain commands are invoked on the client. The Device Manager executes these commands. The Device Manager is responsible for most administrative actions between the mobile server and the mobile client.

1.3 Creating the Publish-Subscribe Model for Mobile Users

To enable users to access their data, you need to first define the data in the snapshot. Then, subscribe the appropriate users to access only their data. On the client device, data is stored in a special type of relational table, called a snapshot table. A snapshot

table behaves exactly the same as a regular relational table, but also includes functionality to track changes made to the table.

The publication item, which is executed against the server database, can determine the record set that is downloaded to the snapshot table. The result set of the query defines the structure (columns) of the snapshot table on the client device as well as its contents.

A collection of publication items is a publication, which corresponds to a single database on a client device. All snapshot tables that are based on publication items part of a single publication are stored in the same client database.

Oracle Database Mobile Server operates within a publish-subscribe model. We use the example of the magazine as an effective way to explain the publish-subscribe model. A magazine is created with specific data that would be of interest to readers, such as sports, hunting, automobiles, and so on. Readers request a subscription for the specific magazine they feel would be in their interest to read. Once this subscription is created only the magazines to which the reader has been subscribed are sent to the reader.

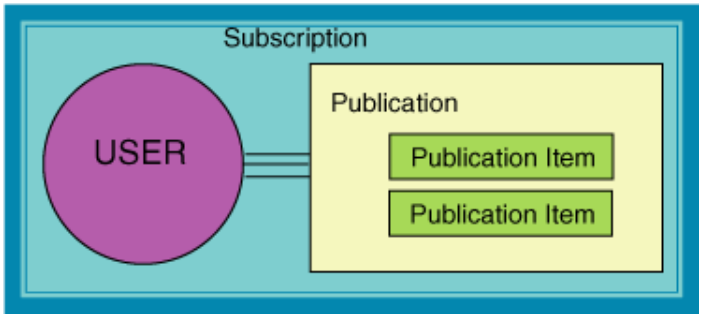
For Oracle Database Mobile Server, the publication is the magazine, the publication items are the specific articles of data and the subscription is the granting of access to the publication for specific users. In the Oracle Database Mobile Server application model, each application defines its data requirements using a publication. Data subsets, known as publications items, are created and added to a publication. Application files are also uploaded to the same publication. Once these publications are deployed to the mobile server, any user may be granted a subscription to the publication.

Technically, a publication is like a database schema and it contains one or more publication items. A publication item is like a parameterized view definition and defines a subset of data, using a SQL query with bind variables in it. These bind variables are called *subscription parameters* or *template variables*.

As shown in Figure 1–2, a subscription defines the relationship between a user and a publication. Once you subscribe to a particular publication, you begin to receive information associated with that publication. With a newspaper you receive the daily paper or the Sunday paper, or both. With Oracle Database Mobile Server you receive snapshots, and, depending on your subscription parameter values, those snapshots are partitioned with data tailored for you.

Subscription parameter values can be set by the administrator in order to tailor the snapshot data for each user.

Figure 1–2 Subscription Defines Relationship Between User and Publication



The subscription is the definition of how to retrieve data from the back-end database; the snapshot is the actual data that conforms to the definition within the subscription and which belongs to the user.

This process really forms a simple development cycle for mobile applications, as follows:

1. Create the publication and its publication items that contains the data subset for a particular application.
2. Grant users a subscription to a publication. This forms the specific dataset that is used on a mobile client.
3. Develop and test the mobile application to work with the specific data set.
4. Deploy the application to the mobile server and install it on the client.

Two of the more common questions and sources of confusion that comes up are what has to be done first:

1. Do you create the publication first or the publication items?

   It does not matter. You can create either the publication or the publication item first. Consider an article for a magazine. That article may have been written by a freelance author. The article exists before it belongs to any publication. The author submits this to two or three magazine publishers since it is relevant to the content they advertise. Two decide it is appropriate for the publication they are distributing currently while one does not include it since the content is not quite what their readers want.

2. Do you have to create a separate publication item for each publication?

   No, you can have one or more publication items in a publication.

> **Note:** You can create publications with the Mobile Database Workbench or the Java APIs.

The following sections describe other pertinent information for publication items:

- Section 1.3.1, "Defining the Weight and Conflict Resolution for Publication Items"
- Section 1.3.2, "Behavior and Requirements for Primary Keys, Foreign Keys and Not Null Fields in Publication Items"

1.3.1 Defining the Weight and Conflict Resolution for Publication Items

The following important aspects of the publication item should be taken into account when you are designing your application:

- Weight—The publication item weight is used to control the order in processing publication items, which avoids conflicts. Changes made on the client are processed according to weight in order to prevent conflicts, such as foreign key violations. The weight determines what tables are applied to the enterprise database first. For example, the `scott.emp` table has a foreign key constraint to the `scott.dept` table. If a new department number is added to the `dept` table and a new record utilizing the new department number were added to the `emp` table, then the transaction would be placed in the error queue if the new record utilizing the new department in the `emp` table was applied to the repository before the new department in the `dept` table was applied. To prevent the violation of the foreign key constraint on the enterprise server, you set the `dept` snapshot to a weight of 1 and the `emp` snapshot to a weight of 2, which applies all updates to the `dept` table before any updates to the `emp` table as the lower weight is always processed first.

- Conflict Resolution—In the same scenario, what if someone already updated the enterprise server with the new department number? This causes a conflict when the client attempts to synchronize with the new department that utilizes the same number. To handle this, conflict resolution may be set to either "client wins" or "server wins". If set to "server wins", then the setting on the server takes precedence to the setting on the client. The client transaction is sent to the error queue. However, if "client wins" is set, then the new department number from the client overrides the setting on the server.

1.3.2 Behavior and Requirements for Primary Keys, Foreign Keys and Not Null Fields in Publication Items

Only primary keys and not NULL fields are replicated down to the client. Publication items require a primary key field or, as in the case of a view, primary key hints.

If a foreign key needs to be applied to the client, then the script for the foreign key needs to be added to the publication, so that it is executed when the client synchronizes for the first time. You can set the script for the foreign key within either the MDW scripts section or the API.

Constraints are not the only type of script that may be executed on the client. The script could execute any valid SQL DDL statement on the client.

1.4 Mobile Development Kit (MDK)

Before you develop an application using Oracle Database Mobile Server, you should install the Mobile Development Kit (MDK) on the machine on which you intend to develop your application. For instructions on how to install the Mobile Development Kit, see Section 4.3, "Installing Oracle Database Mobile Server" in the *Oracle Database Mobile Server Installation Guide*.

The Mobile Development Kit includes the following components.

- Berkeley DB and SQLite Mobile Client binaries.
- Mobile Database Workbench (MDW)—A development tool for creating a publication.
- Packaging Wizard—A tool to publish applications to the mobile server.
- Mobile Sync—A transactional synchronization engine that includes the executable (`msync.exe`) and the Java wrapper for it.

Using any C, C++, or Java development tool in conjunction with the Mobile Development Kit for Windows, you can develop your mobile applications for Windows against Oracle Database Mobile Server, and then publish the applications to the mobile server by using the Packaging Wizard. See Section 4.3, "Installing Oracle Database Mobile Server" in the *Oracle Database Mobile Server Installation Guide* for instructions on how to install the mobile server.

Once you have published the applications to the mobile server, you can use the Mobile Manager to provision the applications to the mobile users. Provisioning involves specifying the values of the subscription parameters used for subsetting the data needed by the application for a particular user. A user to whom an application has been provisioned can then log in to the mobile server and request it to set up everything the user needs to run the applications on the user's device.

1.4.1 Using the Mobile Database Workbench

The Mobile Database Workbench (MDW) is a tool that enables you to iteratively create and test publications—testing each object as you add it to a publication. Publications are stored within a project, which can be saved and restored from your file system, so that you can continue to add and modify any of the contained objects within it.

All work is created within a project, which can be saved to the file system and retrieved for further modifications later. Once you create the project, start creating the publication items, sequences, and scripts that are to be associated with the publication. You can create the publication and associated objects in any order, but you always associate an existing object with the publication. Thus, it saves time to start with creating the objects first and associating it with the publication afterwards.

For detailed information on how to use MDW, see Chapter 4, "Using Mobile Database Workbench to Create Publications".

1.4.2 Using the Packaging Wizard

The Packaging Wizard is a graphical tool that enables you to perform the following tasks.

1. Create a new mobile application.
2. Edit an existing mobile application.
3. Publish an application to the mobile server.

When you create a new mobile application, you must define its components and files. In some cases, you may want to edit the definition of existing mobile application components. For example, if you develop a new version of your application, you can use the Packaging Wizard to update your application definition. The Packaging Wizard also enables you to package application components in a JAR file which can be published using the Mobile Manager. The Packaging Wizard also enables you to create SQL scripts which can be used to execute any SQL statements in the Oracle database.

For detailed information on how to use the Packaging Wizard, see Chapter 5, "Using the Packaging Wizard".

1.5 Mobile Application Design

Before you start to design your mobile application, it is important to read the following sections to understand the differences between an enterprise application and the mobile application as well as the choices you have in designing your application:

- Section 1.5.1, "Steps for Designing Your Mobile Application"
- Section 1.5.2, "Application Programming Interfaces"
- Section 1.5.3, "Application Deployment into the Mobile Environment"

1.5.1 Steps for Designing Your Mobile Application

With a proper design, you can avoid the most common causes for mobile project failure, not meeting the needs of the business, poor performance, or issues occurring within a production environment. Proper design of the mobile system includes the infrastructure and the mobile application. Without proper design, a mobile architecture could end up costing more then it saves.

The following assumption is one of the most common misconceptions for taking an enterprise application and incorporating it into a mobile component:

The mobile application is a scaled down version of the enterprise application.

By taking an existing enterprise application, you may intend to provide the same functionality in remote or disconnected locations. Since the enterprise application has already undergone thorough requirements gathering, design, development, testing, and successful implementation, you may assume that it automatically works seamlessly as a mobile application and so do not test it in this environment. This assumption may lead to project failure.

For example, take an enterprise form-based client/server application. You have a client connecting through a middle-tier connecting to a database on the back-end. Taking this to a mobile infrastructure, such as with Oracle Database Mobile Server, adds a completely new tier that did not exist within the original infrastructure. The mobile server tier introduces new concerns, such as the following:

- Security—There is now a system in the infrastructure that potentially gives any outsider access to the entire organization if proper security configuration and implementation is not performed.
- Bandwidth—The mobile server may become a bottleneck for all remote locations without the implementation of a farm.
- Scalability—The applications are performing synchronization of hundreds to millions of records, which is not the same as providing static Web pages to a large number of users. A system that is fine for serving static Web pages may not be capable of servicing hundreds of users performing synchronization.

A complete redesign of the system specifications may be in order.

You may also need to re-evaluate the original design of the enterprise application. The following lists a few design considerations for the mobile application:

- Memory—An application designed, tested, and implemented on a multiprocessor system with several gigabytes of memory does not perform the same on a mobile device with only a single processor and maybe 512 megabytes of memory.
- Resource Limitations—Several years ago, limitations of available resources made the usage of data types an extreme concern. The storage space saved by using a small integer over an integer was crucial due to limited memory available. With advances in memory and system resources, this has not been a concern to most modern developers. Now, the mobile infrastructure brings resource limitations back to the list of chief concerns for the design and development of mobile applications. One of the most significant of these limitations is the bandwidth available for the mobile client. If a mobile client is only able to synchronize over a cell phone network, you may not wish to bring a million records down to a client that only needs a few thousand records. This decision impacts the synchronization performance, as well as the costs associated with the synchronization. If the mobile client was only utilized to collect data, then you can create a data collection queue for synchronization and avoid the whole download phase of synchronization.
- Use of Indexes—You use indexes for avoiding full-table scans. So, if you use the same data subset originally designed for a Windows machine down on a client device and do not use an index, then the performance may be adversely effected. Oracle Database Mobile Server uses two types of scans for queries: full table scans and index based scans.

Thus, we recommend the following steps:

- Section 1.5.1.1, "Design for Synchronization"
- Section 1.5.1.2, "Design for Sequences"

1.5.1.1 Design for Synchronization

If you are using the mobile option, synchronization holds the mobile infrastructure together.

Analyze all of the data needed by the mobile user, as follows:

- Most snapshots are created where the data can be modified on either the client or the server, where the modifications are propagated to the other side through synchronization.
- If any snapshots require only the ability to read the data—that is, all modifications to the data are made on the server-level and not by the user—then create read-only snapshots.
- If all or a majority of the users use the same read-only snapshots, then create a cached user that shares the read-only data across multiple clients.

Analyze the type of synchronization that is appropriate for the user's needs, as described below:

- For optimal performance, use fast refresh for all publications, if appropriate.
- Only design publication items for a complete refresh if the following is true:
  - If it is absolutely critical for all changes to be processed and applied to the data store immediately.
  - If it is critical that any enterprise updates are immediately brought down to the client.

> **Note:** The complete refresh is the most resource intensive method and should only be utilized after full consideration of the performance hit is analyzed. Only time critical publication items should be specified for a complete refresh synchronization type.

- If a mobile user is only performing data collection and it is not necessary for server updates to be brought down to the user, then implement a push-only synchronization model for those publication items. For more information, see Section 2.12.2, "Creating Data Collection Queues for Uploading Client Collected Data".
- When a mobile user only requires specific table to be updated or synchronized, perform a selective synchronization methodology limiting the synchronization process to specific tables or specific publications.

1.5.1.2 Design for Sequences

Sequences guarantee uniqueness of a value, such as a primary key. Design how the sequences are generated within the mobile infrastructure. For example, if the enterprise database generates a sequence number and the mobile client generates the same sequence number a conflict with the data occurs and causes an error.

Native sequences may be formed specifically for the mobile clients. These sequences would never populate on the enterprise database itself, so there is no risk of a conflict occurring. This works well when data updates only occur from the mobile clients and input to the database does not come from any other source. However, it is often

necessary to have the sequences generated by both the database and the clients. To accomplish this, sequences must be designed so the database uses a range separate from the range used by the clients. For example, you could define the sequences where the database uses all odd numbers and the clients uses all even numbers.

You must design sequences for mobile clients, so that each client uses a unique range of values without any two clients using the same range. For this you specify the sequence range for each client, such as sequences 1 through 1000 for client A and sequences 1001 through 2000 for client B. Using these ranges for the sequence numbers prevents each client from using the same sequence number as used by another client.

For full details on sequences, see Section 4.6, "Create a Sequence".

1.5.2 Application Programming Interfaces

When you are developing your application, you may decide that you want to control more aspects of Oracle Database Mobile Server within your application—rather than relying on user interaction. In this case, you can use the Oracle Database Mobile Server Application Programming Interfaces (APIs). Almost any task performed by the tools and utilities included with Oracle Database Mobile Server may also be accomplished with the APIs. Some of the more advanced functionality within the product is only available through the use of the APIs. Except for the synchronization APIs which are provided for most languages utilized for application development, most of the APIs are Java interfaces that must be developed with the Java programming language.

The most common APIs utilized and their uses are as follows:

- Synchronization APIs: These APIs provide all of the basic synchronization functionality that is found within the mSync utility. The advantages of using these APIs are that the synchronization process can be fully integrated within the actual mobile application. The APIs also provide the ability for a push-only synchronization, which allows mobile clients to only upload data skipping the downloading of new data or applications. The push-only model is useful when bandwidth is limited and when the client just collects data—that is, it is not necessary for a remote client to have updated data from the enterprise.
- Consolidator APIs: The Consolidator APIs provide administrative functionality for creating users, setting the user properties, working with applications, and so on. You can automate common administration tasks and speed up some of the administration tasks required, such as the creation of a large amount of users. The only limitation is that application and user settings are not displayed in the Mobile Manager Web administration tool as these APIs directly access the repository.
- Mobile Resource Manager APIs: The Mobile Resource Manager APIs also provides administration functionality for users and applications; however, these API's actually update the Mobile Manager administration tool as well as the repository. This utility may be used to create users, set user access, set the user template variables, and many other tasks.
- Device Manager APIs: The Device Manager APIs provide the ability customize the management of devices. These APIs may be used to gather information on devices, send commands to devices, register devices, and so on.

1.5.3 Application Deployment into the Mobile Environment

Deployment of applications includes setting up the server system so that end users can easily install and use the applications. Mobile applications are deployed to the mobile server.

Deployment consists of the following steps:

1. Create the publication with the Mobile Database Workbench (MDW). See Section 1.4.1, "Using the Mobile Database Workbench" for more information.
2. Publishing the application to the server includes installing all the components for an application on the mobile server with the Packaging Wizard tool. See Section 1.4.2, "Using the Packaging Wizard" for details.
3. Provisioning the applications to the mobile users through the Mobile Manager, which is a GUI interface for the mobile server. This phase includes determining user access to applications with a specified subset of data. The Mobile Manager can create users, grant privileges to execute applications, and define the data subsets for them, among others. You can also use the Java API to provision applications.
4. Testing for functionality and performance in a real deployment environment. A mobile application system is a complex system involving the following:
   - Multiple mobile device client technologies—such as, operating systems, form factors, and so on.
   - Multiple connectivity options—such as, LAN, Wireless LAN, cellular, wireless data, and other technologies.
   - Multiple server configuration options.

   When testing, pay particular attention to tuning the performance of the data subsetting queries, as it is the most frequent cause of performance problems.
5. Determining the method of initial installation of applications on mobile devices (application delivery). Initial installation involves installing the mobile client, the application code, and the initial client database. The volume of data required to install applications on a mobile device for the first time could be quite high, necessitating the use of either a high-speed reliable connection between the mobile device and the server, or using a technique known as offline instantiation. In offline instantiation, everything needed to install an application on a mobile device is put on a CD or any storage media device and physically given to the user. The user then uses this media to install the application on the device by means of a desktop machine. Oracle Database Mobile Server provides a tool for offline instantiation.

After deployment, both the application and the data schema may change because of enhancements or defect resolution. The mobile server manages application updates and data schema evolution. The only requirement is that the administrator must republish the application and the schema. The mobile server automatically updates the mobile clients that have older version of the application or the data.

2

Synchronization

The mobile client database contains a subset of data stored in the Oracle database. This subset is stored in snapshots in the mobile client database. Unlike a base table, a snapshot keeps track of changes made to it in a change log. Users can make changes in the mobile client database and can synchronize these with the Oracle database.

The following sections describe synchronization functions between the mobile clients and an Oracle database using the mobile server. This chapter discusses how you can programmatically initiate the synchronization both from the client or the server side.

2.1 How Oracle Database Mobile Server Synchronizes

In Oracle Database Mobile Server, the synchronization is used for multiple clients—rather than a single user. In order to accommodate a large number of concurrent users, the application tables on the back-end database cannot be locked by a single user. Thus, the synchronization process involves using queues to manage the information between the mobile clients and the application tables in the database.

Oracle Database Mobile Server uses a synchronization model that maintains data integrity between the mobile server and the mobile client. In addition, the synchronization is asynchronous and that as a result, change propagation is not immediate. The benefit, however, is that the clients do not stay connected for long while the changes are being applied.

You can specify if the synchronization occurs automatically or by manual request. For more details, see Section 2.1.3, "Deciding on Automatic or Manual Synchronization".

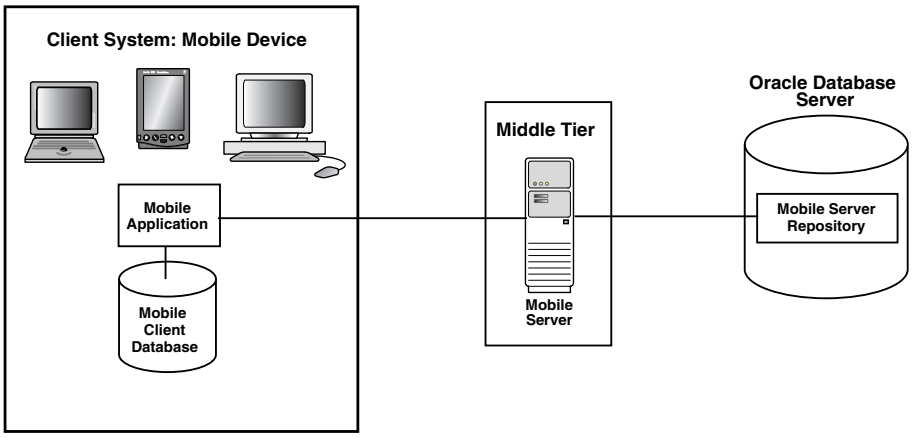
A simplified view of synchronization is as follows:

- On the client—The mobile application communicates through the Sync Server with the mobile server and uploads the changes made in the client machine. It then downloads the changes for the client that are already prepared by the mobile server.
- On the mobile server—A background process called the Message Generator and Processor (MGP), which runs in the same tier as the mobile server, periodically collects all the uploaded changes from many mobile users and then applies them to the server database. Next, MGP prepares changes that need to be sent to each mobile user. This step is essential because the next time the mobile user synchronizes with the mobile server, these changes can be downloaded to the client and applied to the client database.

Figure 2–1 illustrates the architecture for Oracle Database Mobile Server applications.

> **Note:** This section describes how the synchronization is performed across several components and enterprise tiers to complete successfully. For more details on each component, see Section 1.2, "Oracle Database Mobile Server Application Model and Architecture".

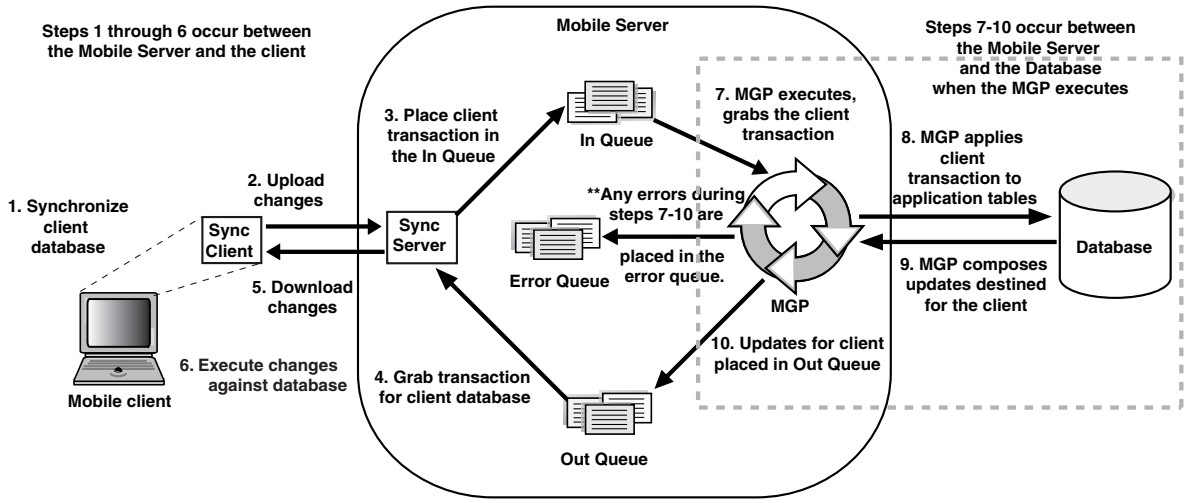
Figure 2–1 Oracle Database Mobile Server Architecture



The mobile server replicates data between the mobile clients with their client databases and the application tables, which are stored on a back-end Oracle database.

Thus, the more detailed description of how synchronization is performed within the separate components of Oracle Database Mobile Server is demonstrated by Figure 2–2.

Figure 2–2 Data Synchronization Architecture



1. A synchronization is initiated on the mobile client either by the user or from automatic synchronization.
2. The mobile client software gathers all of the client changes into a transaction and the Sync Client uploads the transaction to the Sync Server on the mobile server.
3. Sync Server places the transaction into the In-Queue.

> **Note:** When packaging your application, you can specify if the transaction is to be applied at the same time as the synchronization. If you set this option, then the transaction is immediately applied to the application tables. However, this may not be scaleable and you should only do this if the application of the transaction immediately is important and you have enough resources to handle the load.

4. Sync Server gathers all transactions destined for the mobile client from the Out-Queue.
5. Sync Client downloads all changes for client database.
6. The mobile client applies all changes for client database. If this is the first synchronization, the client database is created.
7. All transactions uploaded by all mobile clients are gathered by the MGP out of the In-Queue. The MGP executes independently and periodically based upon an interval specified in the Job Scheduler in the mobile server.
8. The MGP executes the apply phase by applying all transactions for the mobile clients to their respective application tables to the back-end Oracle database. The MGP commits after processing each publication. If any conflicts occur during this phase, most are resolved by the MGP or by the conflict resolution rules. If the conflict cannot be resolved, the transaction is moved into the Error Queue. See Section 1.3.1, "Defining the Weight and Conflict Resolution for Publication Items" for more information.

> **Note:** The behavior of the apply/compose phase can be modified. See Section 5.1.1, "Defining Behavior of Apply/Compose Phase for Synchronization" in the *Oracle Database Mobile Server Administration and Deployment Guide* for more information.

9. MGP executes the compose phase by gathering the client data into outgoing transactions for mobile clients.
10. MGP places the composed data for mobile clients into the Out-Queue, where the Sync Server downloads these updates to the client on the next client synchronization.

Overall, synchronization involves two parties: the mobile client using the Sync Client/Server to upload and download changes and the MGP process interacting with the queues and the application tables to apply and compose transactions. These are displayed separately in the Data Synchronization section of the Mobile Manager.

The following sections describe synchronization activity:

- Section 2.1.1, "Mobile Client Database Created on First Synchronization"
- Section 2.1.2, "Using Multiple Databases for Application Data"
- Section 2.1.3, "Deciding on Automatic or Manual Synchronization"
- Section 2.1.4, "Deciding on Synchronization Refresh Option"
- Section 2.1.5, "Synchronizing to a File With File-Based Sync"
- Section 2.1.6, "How Downloaded Data is Processed on the Mobile Client"
- Section 2.1.7, "How Updates Are Propagated to the Back-End Database"

2.1.1 Mobile Client Database Created on First Synchronization

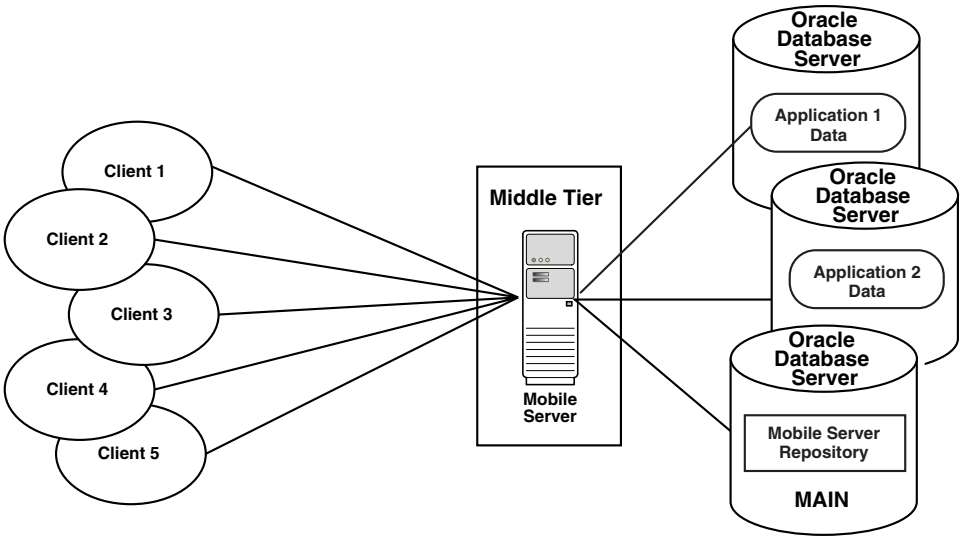
When a user synchronizes a mobile client for the first time, the mobile client creates a database on the client machine for each subscription that is provisioned to the user. The mobile client then creates a snapshot in this database for each publication item contained in the subscription, and populates it with data retrieved from the server database by running the SQL query (with all the variables bound) associated with the publication item. Once installed, Oracle Database Mobile Server is transparent to the end user; it requires minimal tuning or administration.

As the user accesses and uses the application, changes made to the data in the client database are captured by the snapshots. When the connection to the mobile server is available, the changes can be synchronized with the mobile server.

2.1.2 Using Multiple Databases for Application Data

By default, the repository metadata and the application data are stored on the same database. However, if for performance or other reasons, you may store application data on a separate database other than the main database where the repository exists. In this manner, the repository exists on the main database and the data for one or more applications may exist on the main database or another database of your choosing.

Figure 2–3 Separating Application Data from Repository



You can register one or more databases to host the application data. Once registered, you can specify during publication creation where to host the application data. Synchronization is executed on a per publication basis rotating through the databases.

2.1.3 Deciding on Automatic or Manual Synchronization

In the past, all that was available was manual synchronization. That is, a client manually requests a synchronization either through an application program executing an API or by a user manually pushing the Sync button.

Manual Synchronization may be initiated, as follows:

- The user initiates the Mobile Sync (mSync) application directly.
- The application programmatically invokes the Mobile Sync API.

Automatic Synchronization can be configured to automatically occur under specific circumstances and conditions. When these conditions are met, Oracle Database Mobile Server automatically performs the synchronization for you without locking your database, so you can continue to work while the synchronization happens in the background. This way, synchronization can happen seamlessly without the client's knowledge.

Note: Automatic synchronization is enabled on per-publication basis. A publication can be enabled for automatic synchronization. The publications that are not enabled for automatic synchronization can only be synchronized manually. A publication contains one or more publication items that can be marked as either manual or automatic.

Currently, you can enable automatic synchronization for a publication by marking one or more of its publication items automatic. Once you do that, all publication items within that publication are synchronized automatically. A publication without automatic publication items is enabled only for manual synchronization.

For example, you may choose to enable automatic synchronization for the following scenarios:

- If you have a user who changes data on their handheld device, but does not sync as often as you would prefer.
- If you have multiple users who all sync at the same time and overload your system.

These are just a few examples of how automatic synchronization can make managing your data easier, be more timely, and occur at the moment you need it to be uploaded.

Synchronization is closely tied to how you define the snapshot for your application. See Section 1.3, "Creating the Publish-Subscribe Model for Mobile Users" for a description of a snapshot and its components. One of the components is a publication item. If you want automatic synchronization, you define it at the publication item level.

Note: When a manual synchronization is requested by the client, ALL publications are synchronized at that time - including those defined as manual and automatic synchronization.

The differences between the two types of synchronization are as follows:

Table 2–1 Difference Between Automatic and Manual Synchronization

	Manual Synchronization	Automatic Synchronization
Initiation	After the snapshot is set up, you can initiate either by the user initiating mSync or by an application invoking one of the synchronization APIs.	All of the set up for automatic synchronization is configured. Once configured, it happens automatically, so there is no synchronization API. Configuration for automatic synchronization can be defined when you create the publication item, publication or the platform. For more information, see Section 2.2.1, "Enable Automatic Synchronization at the Publication Level"
Controlling synchronization	Synchronization occurs exactly when the user/application requests it.	Synchronization occurs without the user being aware of it occurring. You can start, stop, pause, resume and query the status of automatic synchronization using Sync Agent Control APIs. For more information see, Section 3.2, "Manage Automatic Synchronization on the Mobile Client"
Objects synchronized	All	The following objects are not synchronized by an automatic synchronization: sequences, DDL scripts, indexes and automatic synchronization rules and conditions.

Manual synchronization supports selective sync option where the user can choose which publications need to be synchronized instead of synchronizing all publications on the client. See Section 3.1.1.1.4, "Selective Synchronization".

Automatic synchronization also uses selective sync, but this is done automatically without user's interference. Automatic synchronization is driven by rules: events and conditions, which determine if, when and which publications need to be synchronized. See Section 2.2.2, "Define the Rules Under Which the Automatic Synchronization Starts" for details on events and conditions.

2.1.3.1 Synchronization Priorities

Sometimes, some data records may need to be sent from client to server and from server to client in more expedient manner than the rest of the data. In other words, some data may have higher priority and needs to be synchronized first without having to wait for the rest of the data. Synchronization allows to define the priority of data for every record. Two priorities are supported: high and normal. Usually most of the data is of normal priority and only some data is of high priority. Data priority can be set on per-record basis. See Section 3.5.1.2, "Data Priority Handling" of the *Mobile Client Guide* on how to set record priorities on the client. Both manual and automatic synchronization support priority setting: high priority sync synchronizes only records of high priority and normal priority sync synchronizes all records including high and normal priorities. Section 3.1, "Invoke Manual Synchronization on the Mobile Client" indicates how sync priority can be set in all types of manual synchronization that the APIs supported.

Automatic synchronization manages sync priority using different rules for high and normal priority data: high priority rules and normal priority rules. Usually high priority rules are defined such that they allow high priority data to be synchronized quicker. For example, there may be a network condition rule that restricts normal priority synchronization to a time where high network bandwidth is available. But same kind of rule for high priority synchronization may allow synchronization with any network bandwidth. For more information see, Section 2.2.2, "Define the Rules Under Which the Automatic Synchronization Starts".

2.1.4 Deciding on Synchronization Refresh Option

How or when data changes are applied to either the mobile server or the mobile client depends upon the synchronization refresh option at the publication item level. Synchronization refresh options may ease the cost burden for resources, such as wireless connectivity, bandwidth and network availability, personnel loss of time during the synchronization process, and so on.

Oracle Database Mobile Server employs synchronization refresh options that may be utilized to synchronize data between the Oracle enterprise database and the mobile client. With the following Oracle Database Mobile Server refresh options, you can maintain data accuracy and integrity between the Oracle database and mobile client:

- Section 2.1.4.1, "Fast Refresh"
- Section 2.1.4.2, "Complete Refresh"
- Section 2.1.4.3, "Queue-Based Refresh"
- Section 2.1.4.4, "Forced Refresh"

2.1.4.1 Fast Refresh

The most common method of synchronization is a fast refresh publication item where changes are uploaded and downloaded by the client. Meanwhile, the MGP periodically collects changes uploaded by all clients and applies them to the back-end Oracle database tables. Then, the MGP composes new data, ready to be downloaded to each client during the next synchronization, based on pre-defined subscriptions.

2.1.4.2 Complete Refresh

During a complete refresh, all data for a publication is downloaded to the client. For example, during the first synchronization session, all data on the client is refreshed from the Oracle database. This form of synchronization takes longer because all rows that qualify for a subscription are transferred to the client device, regardless of existing client data.

The complete refresh model is resource intensive as all aspects of synchronization are performed. This model should only be utilized for snapshots/publication items where it is an absolute requirement.

2.1.4.3 Queue-Based Refresh

The developer creates their own queues to handle the synchronization data transfer. There is no synchronization logic created with a queue-based refresh; instead, the synchronization logic is implemented solely by the developer. A queue-based publication item is ideally suited for scenarios that require synchronization to behave in a different manner than normally executed. For instance, data collection on the client; all data is collected on the client and pushed to the server.

With data collection, there is no need to worry about conflict detection, client state information, or server-side updates. Therefore, there is no need to add the additional overhead normally associated with a fast refresh or complete refresh publication item.

2.1.4.4 Forced Refresh

This is not a refresh option; however, we discuss it here because it is often mistaken for a refresh option—specifically, it is often confused with the complete refresh option. The Forced Refresh is a one-time execution request made from within Mobile Manager, the GUI interface for the mobile server. The forced refresh option may result in a loss of critical data on the client.

The forced refresh option is an emergency only synchronization option. This option is used when a client is corrupt or malfunctioning, so that you decide to replace the mobile client data with a fresh copy of data from the enterprise data store with the forced refresh. When this option is selected, any data transactions that have been made on the client are lost.

When a forced refresh is initiated all data on the client is removed. The client then brings down an accurate copy of the client data from the enterprise database to start fresh with exactly what is currently stored in the enterprise data store.

2.1.5 Synchronizing to a File With File-Based Sync

There are times when you do not have network access to the mobile server, but there is a way you can use removable media to transport a file between the mobile server and the client. In this instance, you may want to use File-Based Sync, which saves all transactions in an encrypted file either for the upload from the client for the mobile server or the download from the mobile server for the client.

Once saved within the encrypted file, the file is manually transported and copied onto the desired recipient—whether mobile client or mobile server. This file is uploaded and the normal synchronization steps are performed. The only difference is that the interim transmission of the data is through a file copied to the correct machine—rather than transmitted over a network.

For full details on file-based synchronization, see Section 5.10, "Synchronizing to a File with File-Based Sync" in the *Oracle Database Mobile Server Administration and*

Deployment Guide. To enable and perform file-based synchronization through the APIs, see Chapter 3, "Managing Synchronization on the Mobile Client".

2.1.6 How Downloaded Data is Processed on the Mobile Client

The client processes the downloaded data. By default, the steps taken to process the received data on the client is as follows:

1. Process each publication item
2. Commit
3. Process each DDL statement
4. Commit

> **Note:** The acknowledgment is sent only in the subsequent synchronization.

In addition, the configuration could effect how the data is processed on the client. Low memory may cause a commit before all of the publication items are processed. If the client is on a WIN32 device and available memory is running low, then an auto commit is performed. However, if the client is on a Windows Mobile device and if memory is getting low, the synchronization throws an error and exits. In this situation, the commit is not performed.

2.1.7 How Updates Are Propagated to the Back-End Database

The synchronization process applies client operations to the tables in the back-end database, as follows:

1. The operations for each publication item are processed according to table weight. The publication creator assigns the table weight to publication items within a specific publication. This value can be an integer between 1 and 1023. For example, a publication can have more than one publication item of weight "2" which would have `INSERT` operations performed after those for any publication item of a lower weight within the same publication. You define the order weight for tables when you add a publication item to the publication. See Section 2.4.1.7.2, "Using Table Weight" for more information.
2. Within each publication item being processed, the SQL operations are processed as follows:
   a. Client `INSERT` operations are executed first, from lowest to highest table weight order.
   b. Client `DELETE` operations are executed next, from highest to lowest table weight order.
   c. Client `UPDATE` operations are executed last, from highest to lowest table weight order.

For details and an example of exactly how the weights and SQL operations are processed, see Section 2.4.1.7.2, "Using Table Weight".

> **Note:** This order of executing operations can cause constraint violations. See Section 2.9, "Synchronizing With Database Constraints" for more information.

In addition, the order in which SQL statements are executed against the client database is not the same as how synchronization propagates these modifications. Instead, synchronization captures the end result of all SQL modifications as follows:

1. Insert an employee record 4 with name of Joe Judson.
2. Update employee record 4 with address.
3. Update employee record 4 with salary.
4. Update employee record 4 with office number
5. Update employee record 4 with work email address.

When synchronization occurs, all modifications are captured and only a single insert is performed on the back-end database. The insert contains the primary key, name, address, salary, office number and email address. Even though the data was created with multiple updates, the Sync Server only takes the final result and makes a single insert.

2.1.8 Oracle Database Mobile Server (DMS) Encryption

DMS combines RSA asymmetric encryption with the 128-bit Advanced Encryption Standard (AES). RSA public/private key encryption is used to transport user credentials and session key info, while the data payload itself is encrypted with AES. This allows for both fast and secure mobile data exchange. The communication between the client and the server is initiated by the client with the following message format:

[RSA_HEADER(usr/pwd/session_key)][ENCRYPTED_PAYLOAD]

In the formula above, RSA_HEADER contains a mobile client's username, password, as well as a session key. While the username and password are provided by the client, each session key is randomly generated. The key is calculated using cryptographically-safe random number generator where the choice of the generator is OS-dependent. Combined username, password and session key are encrypted using the server's public key.

ENCRYPTED_PAYLOAD is encrypted with the 128-bit AES algorithm in Cipher-Block Chaining (CBC) mode using session_key included in the RSA_ HEADER. The server, upon receiving a client's encrypted request, decrypts the RSA_ HEADER with the server's private key, authenticates the client based on the included credentials, and finally decrypts the ENCRYPTED_PAYLOAD using the included session_key.

The server's response to the client is then encrypted with the same session_key.

Note: Both RSA and AES encryptions are FIPS 140-2 approved.

Note: The user can select either SSL or AES for data encryption.

2.2 Enabling Automatic Synchronization

Automatic synchronization occurs in the background, so that the user does not have to perform a synchronization; thus, the client appears continually connected to the back-end database without user interaction. All modifications to each record are saved in a log within the client database. When you requested synchronization manually, Oracle Database Mobile Server locked the database while processing your request.

However, with automatic synchronization, it could be occurring while you are performing other tasks to the client database.

When synchronization occurs, all of the modified records stored in the log are uploaded to the server. In addition, any modified records from the server are downloaded into the client database. This occurs in the same manner as manual synchronization. The only difference is when the synchronization is executed and how the modified records are stored.

The following are details about automatic synchronization:

Table 2–2 Automatic Synchronization

Steps for Automatic Synchronization	See the Following for Details
The developer enables the publication item to use automatic synchronization.	Section 2.2.1, "Enable Automatic Synchronization at the Publication Level"
The client can disable and enable automatic synchronization with the Sync Control API.	Section 3.2, "Manage Automatic Synchronization on the Mobile Client"
You can configure under what rules the automatic synchronization occurs.	Section 2.2.2, "Define the Rules Under Which the Automatic Synchronization Starts"
The server can notify the client of data waiting for download.	Section 2.12.3, "Selecting How and When to Notify Clients of Composed Data"
The client application can request status of the outcome of an automatic synchronization.	Section 2.2.4, "Retrieve Status for Automatic Synchronization Events"

The following sections detail how you can configure for automatic synchronization:

- Section 2.2.1, "Enable Automatic Synchronization at the Publication Level"
- Section 2.2.2, "Define the Rules Under Which the Automatic Synchronization Starts"
- Section 2.2.3, "Enable the Server to Notify the Client to Initiate a Synchronization to Download Data"
- Section 2.2.4, "Retrieve Status for Automatic Synchronization Events"

2.2.1 Enable Automatic Synchronization at the Publication Level

Automatic synchronization can be enabled at publication level. Within a publication, you can have one or more publication items. If automatic synchronization is enabled for one of the publication items, by default all the publication items in the same publication would be enabled with automatic synchronization.

Do not define a publication where some of the publication items are automatic synchronization enabled but the others are not. See Section 4.4, "Create a Publication Item" for details of how to enable synchronization in a publication item using MDW or Section 2.4.1.3, "Create Publication Items"using the API.

To manage automatic synchronization, see Section 3.2, "Manage Automatic Synchronization on the Mobile Client".

2.2.2 Define the Rules Under Which the Automatic Synchronization Starts

You can configure under what circumstances a synchronization should occur and then Oracle Database Mobile Server performs the synchronization for you automatically. The circumstances under which an automatic synchronization occurs is defined within the synchronization rules, which includes the following:

- Events—An event is variable, as follows:
  - Data events: For example, you can specify that a synchronization occurs when there are a certain number of modified records in the client database.
  - System events: For example, you can specify that if the battery drops below a predefined minimum, you want to synchronize before the battery is depleted.
- Conditions—A condition is an aspect of the client that needs to be present for a synchronization to occur. This includes conditions such as battery life or network availability.

The relationship between events and conditions when evaluating if an automatic synchronization occurs is as follows:

```
when EVENT and if (CONDITIONS), then SYNC
```

So, if an event occurs, the conditions are evaluated. If the conditions are valid, then the synchronization occurs; if the conditions are not met, then the synchronization is queued until the conditions are valid.

For example, if the event for new data inserted and the condition specified is that the network must be available, then a synchronization occurs when the network is available and there is new data.

You can define the rules for automatic synchronization within certain parts of the normal snapshot setup and platform configuration, as follows:

- Publication level: Within the publication, you specify the rules under which the synchronization occurs for all publication items in that publication.
- Platform level: Some of the rules are specific to the platform of the client, such as battery life, network bandwidth, and so on. These rules apply to all enabled publication items that exist on this particular platform, such as Windows Mobile.

If after defining these rules and publishing the application, you want to modify the rules, you can do so through MDW. However, you must perform a manual synchronization. The manual synchronization restarts the automatic Sync Agent, which then uses the new rules. The new settings are NOT applied during automatic synchronization.

The following sections detail all of the rules you can configure for automatic synchronization:

- Section 2.2.2.1, "Default vs Custom Rules"
- Section 2.2.2.2, "Configure Publication-Level Automatic Synchronization Rules"
- Section 2.2.2.3, "Configure Platform-Level Automatic Synchronization Rules"

2.2.2.1 Default vs Custom Rules

Default rules are used to bootstrap the syncagent if user-defined rules are not present. For each database, there is a default database level event which triggers sync after any database commit, where 1 or more records were modified. Additionally, there is a default platform level condition that allows sync only if network is detected on the client. User-defined rules override these default rules as follows:

1. The database level event above are overridden (and removed) if a publication level event is defined for the same database.
2. All default database level events are removed if a platform level event is defined.

3. The default platform level network condition is overridden (and removed) if another network condition is defined.

> **Note:** The rules, events and conditions, at publication level and at platform level, are defined separately for high priority sync and normal priority sync. High priority rules apply only to high priority records and only trigger/allow/forbid high priority sync.

Two default database level events are created for each database: for high and normal priorities. This results in high priority sync being triggered after commit (if high priority record(s) are modified) and normal sync being triggered after commit (if normal record(s) are modified). Likewise, 2 default platform level network conditions are created as well that apply to high and normal priority syncs correspondingly.

> **Note:** The user-defined rules override default rules, as described above, separately for each of the priorities. For example, if you define a high priority platform level event, it overrides all high priority default database events, but not normal priority default database events. So in this case sync is still triggered on commit if normal priority records are modified.

2.2.2.2 Configure Publication-Level Automatic Synchronization Rules

Within the publication, you specify the rules under which the synchronization occurs for all publication items in that publication. These rules are defined when you create the publication either using MDW or programmatically with the APIs. To create this through MDW, see Section 4.5, "Define the Rules Under Which the Automatic Synchronization Starts" ; to add publication-level automatic synchronization rules with the API, see Section 2.4.1.4, "Define Publication-Level Automatic Synchronization Rules".

When you are creating the publication, you can define events that causes an automatic synchronization. Although these are defined at the publication level, they enable only the publication items within this publication that has automatic synchronization enabled.

Table 2–3 describes the publication level events for automatic synchronization. The lowest value that can be provided is 1.

Table 2–3 Automatic Events for the Publication

Events	Description
Client commit	For mobile client only. Upon commit to the client database, the mobile client detects the total number of record changes in the transaction log. If the number of modifications is equal to or greater than your pre-defined number, automatic synchronization occurs. This rule is on by default and set to start an automatic synchronization if only one record is changed. You must modify this rule if you do not want the automatic synchronization to occur after every commit.
Server MGP compose	If after the MGP compose cycle, the number of modified records for a user is equal to or greater than your pre-defined number, then an automatic synchronization occurs. Thus, if there are a certain number of records contained in an Out Queue destined for a client on the server, these modifications are synchronized to the client.

Note: If you want to modify the publication-level automatic synchronization rules after you publish the appliation, you can do so through the Mobile Manager, as follows:

1. Click **Data Synchronization**.
2. Click **Repository**.
3. Click **Publications**.
4. Select the publication and click **Automatic Synchronization Rules**.

2.2.2.3 Configure Platform-Level Automatic Synchronization Rules

Some of the rules are specific to the platform of the client, such as battery life, network bandwidth, and so on. These rules apply to all enabled publication items that exist on this particular platform, such as Windows Mobile. You configure these rules through Mobile Manager or MDW. This section describes Mobile Manager.

The platform-level synchronization rules apply to a selected client platform and all publications that exist on that platform. You can specify both platform events and conditions using the Mobile Manager.

To assign platform-level automatic synchronization rules, perform the following in Mobile Manager:

1. Click **Data Synchronization**.
2. Click **Platform Settings**, which brings up a page with the list of all the platforms that support automatic synchronization.
3. Click on the desired platform.
4. You can modify the following for each platform:
   - Event Rules—See Section 2.2.2.3.1, "Event Rules for Platforms".
   - Conditions—See Section 2.2.2.3.2, "Condition Rules for Platforms".
   - Network settings—See Section 2.2.2.3.4, "Network Configuration for the Client Platform".

2.2.2.3.1 Event Rules for Platforms

Table 2–4 shows the platform events for automatic synchronization.

Table 2–4 Automatic Event Rules for the Client Platform

Event	Description
Network bandwidth	If the mobile client detects that it is connected to a network with a pre-defined minimum bandwidth, then automatic synchronization occurs. Refer to Section 2.2.2.3.3, "Network Speed of SyncAgent"
Battery life	If the battery life drops below a pre-defined minimum, then synchronization is automatically triggered.
AC Power	As soon as AC power is detected, then synchronization is automatically triggered.

Table 2–4 (Cont.) Automatic Event Rules for the Client Platform

Event	Description
Time	Synchronize at a specific time or time interval. You can configure an automatic synchronization to occur at a specific time each day or as an interval. ■ Select **Specify Time** if you want to automatically synchronize at a specific hour, such as 8:00 AM, everyday. ■ Select **Specify Time Interval** if you want to synchronize at a specific interval. For example, if you want to synchronize every hour, then specify how long to wait in-between synchronization attempts.

2.2.2.3.2 Condition Rules for Platforms Table 2–5 shows the platform conditions for automatic synchronization.

Table 2–5 Automatic Condition Rules for Client Platform

Condition	Description
Battery level	Specify the minimum battery level required in order for an automatic synchronization to start. The battery level is specified as a percentage.
Network conditions	Network quality can be specified using several properties. This condition enables you to specify a minimum value for the following network properties: ■ Minimum network bandwidth, which is measured in bits per second. ■ Maximum ping delay, which is measured in milliseconds. ■ Data priority, which is either high or regular. You can specify the priority of your data in the table row. For example, you can define a rule where all high priority data is automatically synchronized at a specified network bandwidth. The ping delay is optional. If not specified, the ping is not calculated. Refer to Section 2.2.2.3.3, "Network Speed of SyncAgent"

2.2.2.3.3 Network Speed of SyncAgent For some platforms, there are APIs to determine network type and optionally subtype but for some platforms like the Windows CE (Windows Mobile) and Android, there is no API to query the exact network speed. For these platforms, syncagent has a hardcoded set of values for network speed based on the network type/subtype to determine the network type/subtype and choose the hardcoded value based on that.

There are 2 ways network speed is used:

- It is reported in syncagent status (see BGAgentStatus) in bits per second (bps).
- It is used by syncagent to evaluate network rules. The network rules are created by the user and set up on the server. They include network speed as a threshold parameter (for example, sync is allowed only if network speed >= specified value). Since for the aforementioned platforms, the network speed cannot be exactly determined, the user needs to decide on how to create a network rule based on network type/subtype. The user can choose the network speed value based on the network type/subtype to allow sync, for example.

For example, the user wants to allow only sync on Windows Mobile device if the network is UMTS or faster, the user can set network speed threshold in the rule to be

2000000. Sync would then only be allowed on networks such as UMTS and 1xRTT, BLUETOOTH, HSPDI, WIFI which are considered faster than UMTS.

For some other platforms like PJ client SE and OJEC syncagent currently has no network management, so network rules are ignored.

2.2.2.3.4 Network Configuration for the Client Platform You can set proxy information for your network provider, if required for accessing the internet.

> **Note:** If you are not using a proxy, then you do not need to define proxy information on this page.

You could have two types of networks, as follows:

- Always-on: Define the proxy and port number. Click **Apply** when finished.
- Dial-up:
  - Click **Add Dial-up Network** to add a a new entry for dial-up configuration.
  - To edit an existing configuration, select the name of the existing configuration.
  - To delete an existing configuration, select the checkbox next to the desired configuration and click **Delete**.

If the platform has an always-on network, then this network is always tried first for the connection. If this network is not available, then the dial-up networks are tried in the order specified. You can rearrange the order of the dial-up networks by selecting one of the networks and clicking the up or down button. For dial-up, Oracle Database Mobile Server can automatically establish the network connection before initiating the synchronization.

2.2.3 Enable the Server to Notify the Client to Initiate a Synchronization to Download Data

If you have designed the compose yourself—that is, you do not use the MGP—then, you can notify the client if any data exists on the server that can be downloaded to the client through enqueue notification APIs. You can also use these APIs to manage the automatic synchronization schedule for your clients.

For more information on enqueue notification APIs, see Section 2.12.3, "Selecting How and When to Notify Clients of Composed Data".

2.2.4 Retrieve Status for Automatic Synchronization Events

You can develop your client application retrieve status for the Sync Agent and automatic synchronization events or to be notified of the stage for automatic synchronization. For full details, see Section 3.2.1, "OSE APIs for Managing Automatic Synchronization" and Section 3.2.3, "OCAPI Notification APIs for the Automatic Synchronization Cycle Status".

2.3 What is The Process for Setting Up a User For Synchronization?

Before you perform synchronization, you must do the following:

- Create the publication.
- Create the user and grant the user access to the publication.

- Package publication with an application and publish to the mobile server. This is an optional step.

This is referred to as the publish and subscribe model, which can be implemented in one of the following two ways:

- Declaratively, using MDW to create the publication and the Packaging Wizard to package and publish the applications. This is the recommended method. See Section 2.3.1, "Creating a Snapshot Definition Declaratively" for details.
- Programmatically, using the Resource Manager and the Consolidator Manager APIs to invoke certain advanced features or customize an implementation. This technique is recommended for advanced users requiring specialized functionality. See Section 2.3.2, "Creating the Snapshot Definition Programmatically" for details.

Once created and subscribed, the user can be synchronized, as follows:

- Using manual synchronization where the user initiates it from the device or programmatically from within an application. This chapter discusses how to start the synchronization programmatically in Section 3.1, "Invoke Manual Synchronization on the Mobile Client".
- Using automatic synchronization which is enabled within the publication item itself or the platform configuration. For more informationon automatic synchronization, see Section 2.2, "Enabling Automatic Synchronization".

On the back-end of the synchronization process, you have the option to customize how the apply and compose phase are executed. See Section 2.6, "Customize the Compose Phase Using MyCompose".

2.3.1 Creating a Snapshot Definition Declaratively

Use the Mobile Database Workbench (MDW), a GUI based tool of Oracle Database Mobile Server—described fully in Chapter 4, "Using Mobile Database Workbench to Create Publications"—to create snapshots declaratively. The convenience of a graphical tool is a safer and less error prone technique for developers to create a mobile application. Before actual application programming begins, the following steps must be executed:

1. Verify that the base tables exist on the server database; if not, create the base table.
2. Use MDW to define an application and the snapshot with the necessary publicatinon and its publication items. See Chapter 4, "Using Mobile Database Workbench to Create Publications" for details.
3. Use the Packaging Wizard to publish the application to the mobile server. This creates the publication items associated with the application. See Chapter 5, "Using the Packaging Wizard" for details.
4. Use the Mobile Manager to create a subscription for a given user.
5. Install the application on the development machine.
6. If using manual synchronization, initiate synchronization for the mobile client with the mobile server to create the client-side snapshots. For the mobile client, create the client database automatically.

2.3.1.1 Manage Snapshots

The mobile server administrator can manage a snapshot, which is a full set or a subset of rows of a table or view. Create the snapshot by executing a SQL query against the base table. Snapshots are either read-only or updatable.

The following sections describe how to manage snapshots using MDW:

- Section 2.3.1.1.1, "Read-only Snapshots"
- Section 2.3.1.1.2, "Updatable Snapshots"
- Section 2.3.1.1.3, "Refresh a Snapshot"
- Section 2.3.1.1.4, "Snapshot Template Variables"

2.3.1.1.1 Read-only Snapshots Read-only snapshots are used for querying purposes only. The data is downloaded from the Oracle server to the client; no data on the client is ever uploaded to the server. Any data added on the client in a read-only snapshot can be lost, since it is never uploaded to the server. Changes made to the master table in the back-end Oracle database server are replicated to the mobile client. See Section 4.8.2, "Publication Item Tab Associates Publication Items With the Publication" for instructions on how to define the publication item as read-only.

Note: A subscription created as complete refresh and read-only is light weight; thus, to keep the subscription light weight, the primary keys are not included in the replication. If you want to include primary keys, then create them with the `createPublicationItemIndex` API.

Also, because read-only does not upload any data from the client, there are no conflicts. Thus, when specified within MDW, you can only select Custom for conflict resolution.

2.3.1.1.2 Updatable Snapshots When you define a snapshot as updatable, then the data propagated within a synchronization is bi-directional. That is, any modifications made on the client are uploaded to the server; any modifications made on the back-end Oracle server are downloaded to the client. See Section 4.8.2, "Publication Item Tab Associates Publication Items With the Publication" for instructions on how to define the publication item as updatable.

A snapshot can only be updated when all the base tables that the snapshot is based on have a primary key or virtual primary key. If the base tables do not have a primary key, a snapshot cannot be updated and becomes read-only. Table 2–6 shows each refresh method type and whether it is updatable or read-only depending on primary key or virtual primary key:

Table 2–6 Which Refresh Methods Can Be Updatable or Read-Only

	Fast	Complete	Queue-Based
Table Uses a Primary Key	Updatable or Read-Only	Updatable or Read-Only	Updatable or Read-Only
Table Uses a Virtual Primary Key	Updatable or Read-Only	Updatable or Read-Only	Updatable or Read-Only
No Primary Key or Virtual Primary Key Used	Not applicable since all Fast Refresh tables use a primary or virtual primary key.	Read-Only	Read-Only

2.3.1.1.3 Refresh a Snapshot Your snapshot definition determines whether an updatable snapshot uses the complete or fast refresh method.

- The complete refresh method recreates the snapshot every time it is refreshed. Note that when it recreates the snapshot, all of the data on the client database is erased and then the snapshot for this user on the back-end Oracle database is brought down to the client.
- The fast refresh method refreshes only the modified data within the snapshot definition on both the client and server. In general, the simpler your snapshot definition, the faster it is updated. All fast refresh methods require a primary key or a virtual primary key.

See Section 4.4, "Create a Publication Item" and Section 2.8, "Understanding Your Refresh Options"

2.3.1.1.4 Snapshot Template Variables Snapshots are application-based. In some cases, you may quantify the data that your application downloads for each user by specifying all of the returned data match a predicate. You can accomplish this by using snapshot templates.

A snapshot template is an SQL query that contains data subsetting parameters. A data subsetting parameter is a colon (`:`), followed by an identifier name, as follows:

```
:var1
```

> **Note:** If the subsetting parameter is on a CHAR column of a specified length, then you should either preset all characters to spaces before setting the value or pad for the length of the column with spaces after setting the parameter.

When the mobile client creates snapshots on the client machine, the mobile server replaces the snapshot variables with user-specific values. By specifying different values for different users, you can control the data returned by the query for each user.

You can use MDW to specify a snapshot template variable in the same way that you create a snapshot definition for any platform.

Data subsetting parameters are bind variables and so should not be enclosed in quotation marks (`'`). If you want to specify a string as the value of the data subsetting parameter, then the string contains single quotation marks. You can specify the values for the template variables within the Mobile Manager.

The following examples specify a different value for every user. By specifying a different value for every user, the administrator controls the behavior and output of the snapshot template.

```
select * from emp where deptno = :dno
```

You define this select statement in your publication item. See Section 4.4.1, "Create SQL Statement for Publication Item" for instructions. Then, modify the user in the Mobile Manager to add the value for `:dno`. Then, when the user synchronizes, the value defined for the user is replaced in the select script. See Section 4.5, "Managing Application Parameter Input (Data Subsetting)" in the *Oracle Database Mobile Server Administration and Deployment Guide* for information on how to define the value of the variable. This value can only be defined after the application is published and the user is associated with it.

Table 2–7 provides a sample set of snapshot query values specified for separate users.

Table 2–7 Snapshot Query Values for Separate Users

User	Value	Snapshot Query
John	10	`select * from emp where deptno = 10`
Jane	20	`select * from emp where deptno = 20`

```
select * from emp where ename = :ename
```

Table 2–8 provides another sample snapshot query value.

Table 2–8 Snapshot Query Value for User Names

User	Value	Snapshot Query
John	'KING'	`select * from emp where ename = 'KING'`

2.3.2 Creating the Snapshot Definition Programmatically

You can use the Resource Manager or Consolidator Manager APIs to programmatically create the publication items on the mobile server. Create publication items from views and customize code to construct snapshots.

> **Note:** The Consolidator Manager API can only create a publication. See Section 2.4, "Creating Publications Using Oracle Database Mobile Server APIs" for information on the Consolidator Manager API. Use the Resource Manager APIs to package it with an application, and publish it to the mobile server. See the `oracle.mobile.admin.MobileResourceManager` in the *Oracle Database Mobile Server JavaDoc*, which you can link to off the *`MOBILE_HOME`*`/Mobile/doc/index.htm` page.

The base tables must exist before the Consolidator Manager API can be invoked. The following steps are required to create a subscription:

- Create a publication
- Create a publication item and add it to the publication
- Create a user
- Creating a subscription for the user based on the publication

The details of how to create a publication are documented in Chapter 4, "Using Mobile Database Workbench to Create Publications". Anything that you can do with the MDW tool, you can also perform programmatically using the Consolidator Manager API. Refer to the *Oracle Database Mobile Server JavaDoc* for the syntax.

2.4 Creating Publications Using Oracle Database Mobile Server APIs

The mobile server uses a publish and subscribe model to centrally manage data distribution between Oracle database servers and mobile clients. Basic functions, such as creating publication items and publications, can be implemented easily using the Mobile Development Workspace (MDW). See Chapter 4, "Using Mobile Database Workbench to Create Publications" for more information.

These functions can also be performed using the Consolidator Manager or Resource Manager APIs by writing Java programs to customize the functions as needed. Some

of the advanced functionality can only be enabled programmatically using the Consolidator Manager or Resource Manager APIs.

The publish and subscribe model can be implemented one of two ways:

- Declaratively, using MDW to create the publication and the Packaging Wizard to package and publish the applications. This is the recommended method. This method is described fully in Chapter 4, "Using Mobile Database Workbench to Create Publications" and Chapter 5, "Using the Packaging Wizard".
- Programmatically, using the Consolidator Manager or Resource Manager APIs to invoke certain advanced features or customize an implementation. This technique is recommended for advanced users requiring specialized functionality.
  - Publications created with the Consolidator Manager API can be packaged with an application. See Section 2.4.1, "Defining a Publication With Java Consolidator Manager APIs".
  - The Resource Manager API can be used to associate a publication with an application. See the `oracle.mobile.admin.MobileResourceManager.setApplicationPublication()` in the *Oracle Database Mobile Server JavaDoc*, which is located on the *`ORACLE_HOME`*`/Mobile/doc/index.htm` page.

2.4.1 Defining a Publication With Java Consolidator Manager APIs

While we recommend that you use MDW (see Chapter 4, "Using Mobile Database Workbench to Create Publications") for creating your publications, you can also create them, including the publication items and the user, with the Consolidator Manager API. Choose this option if you are performing more advanced techniques with your publications.

The Consolidator Manager APIs can be packaged with the packaging wizard by following the steps below:

1. Start the Packaging Wizard by opening a Command Prompt and entering 'wtgpack' or by going to Start -> Programs-> Oracle Database Mobile Server 11g and clicking on the Packaging Wizard shortcut.
2. On the 'Make a Selection' screen, select 'Create a New Aplication' and click 'OK'.
3. For the application attributes, select Berkeley DB or SQLite, a Platform and Locale then click 'Next'.
4. On the application panel, enter the Application Name, Virutal Path, Description, and Local Application Directory (use the 'Browse' button to navigate to this folder). Click 'OK' .
5. Click 'Browse' for the Publication Name.
6. In the 'Connect to Database' dialog, enter the Repository Username, Password and Database URL.
7. Click 'OK' .
8. In the 'Publication Name' list, select the publication and click 'Add' .
9. Click 'Next.'
10. Use the 'Load' option to upload any files and click 'Finish' .
11. In the 'Application Definition Completed' screen, ensure 'Publish the Current Application' is selected and click 'OK' .

12. For the 'Publish the Application' screen, enter the Mobile Server URL, Mobile Manger Admin Username, Password, and Repository Directory. Click 'OK' .
13. When the Message dialog reports that the application is published successfully, click 'OK' .
14. Click 'Exit' to close the Application Definition Completed dialog and the Packaging Wizard.

After creating the database tables in the back-end database, create the Resource Manager and Consolidator Manager objects to facilitate the creation of your publication:

- The Resource Manager object enables you to create users to associate with the subscription.
- The Consolidator Manager object enables you to create the subscription.

The order of creating the elements in the publication is the same as if you were using MDW. You must create a publication first and then add the publication items and other elements to it. Once the publications are created, subscribe users to them. See the *Oracle Database Mobile Server JavaDoc* for full details on each method. See Chapter 4, "Using Mobile Database Workbench to Create Publications" for more details on the order of creating each element.

Note: The following sections use the `sample11.java` sample to demonstrate the Resource Manager and Consolidator Manager methods used to create the publication and the users for the publication. The full source code for this sample can be found in the following directories:

On UNIX: `<ORACLE_HOME>/mobile/server/demos/consolidator_api`

On Windows: `<ORACLE_HOME>\Mobile\Server\demos\consolidator_api`

- Section 2.4.1.1, "Create the Mobile Server User"
- Section 2.4.1.2, "Create Publications"
- Section 2.4.1.3, "Create Publication Items"
- Section 2.4.1.4, "Define Publication-Level Automatic Synchronization Rules"
- Section 2.4.1.5, "Data Subsetting: Defining Client Subscription Parameters for Publications"
- Section 2.4.1.6, "Create Publication Item Indexes"
- Section 2.4.1.7, "Adding Publication Items to Publications"
- Section 2.4.1.8, "Creating Client-Side Sequences for the Downloaded Snapshot"
- Section 2.4.1.9, "Subscribing Users to a Publication"
- Section 2.4.1.10, "Instantiate the Subscription"
- Section 2.4.1.11, "Bringing the Data From the Subscription Down to the Client"
- Section 2.4.1.12, "Modifying a Publication Item"
- Section 2.4.1.13, "Callback Customization for DML Operations"

- Section 2.4.1.14, "Restricting Predicate"

Note: To call the Publish and Subscribe methods, the following JAR files must be specified in your `CLASSPATH`.

- `<ORACLE_HOME>\jdbc\lib\ojdbc6.jar`
- `<ORACLE_HOME>\Mobile\classes\consolidator.jar`
- `<ORACLE_HOME>\Mobile\classes\classgen.jar`
- `<ORACLE_HOME>\Mobile\classes\servlet.jar`
- `<ORACLE_HOME>\Mobile\classes\xmlparserv2.jar`
- `<ORACLE_HOME>\Mobile\classes\jssl-1_1.jar`
- `<ORACLE_HOME>\Mobile\classes\javax-ssl-1_2.jar`
- `<ORACLE_HOME>\Mobile\classes\share.jar`
- `<ORACLE_HOME>\Mobile\classes\oracle_ice.jar`
- `<ORACLE_HOME>\Mobile\classes\phaos.jar`
- `<ORACLE_HOME>\Mobile\classes\jewt4.jar`
- `<ORACLE_HOME>\Mobile\classes\jewt4-nls.jar`
- `<ORACLE_HOME>\Mobile\classes\wtgpack.jar`
- `<ORACLE_HOME>\Mobile\classes\jzlib.jar`
- `<ORACLE_HOME>\Mobile\Sdk\bin\devmgr.jar`

2.4.1.1 Create the Mobile Server User

Use the `createUser` method of the `MobileResourceManager` object to create the user for the publication.

1. Create the `MobileResourceManager` object. A connection is opened to the mobile server. Provide the schema name, password, and JDBC URL for the database the contains the schema (the repository).
2. Create one or more users with the `createUser` method. Provide the user name, password, the user's real name, and privilege, which can be one of the one of the following: "O" for publishing an application, "U" for connecting as user, or "A" for administrator. If NULL, no privilege is assigned.

Note: Always request a drop user before you execute a create, in case this user already exists.

3. Commit the transaction, which was opened when you created the `MobileResourceManager` object, and close the connection.

```
MobileResourceManager mobileResourceManager =
    new MobileResourceManager(CONS_SCHEMA, DEFAULT_PASSWORD, JDBC_URL);
mobileResourceManager.createUser("S11U1", "manager", "S11U1", "U");
mobileResourceManager.commitTransaction();
mobileResourceManager.closeConnection();
```

> **Note:** If you do not want to create any users, you do not need to create the `MobileResourceManager` object.

2.4.1.1.1 Change Password You can change passwords for mobile server users with the `setPassword` method, which has the following syntax:

```
public static void setPassword
   (String userName,
    String newpwd) throws Throwable
```

> **Note:** Both user name and passwords are limited to a maximum of 28 characters.

Execute the `setPassword` method before you commit the transaction and release the connection. The following example changes the password for the user `MOBILE`:

```
mobileResourceManager.setPassword("MOBILE","MOBILENEW");
```

2.4.1.2 Create Publications

A subscription is an association of publications and the users who access the information gathered by the publications. Create any publication through the `ConsolidatorManager` object.

1. Create the `ConsolidatorManager` object.
2. Connect to the database using the `openConnection` method. Provide the schema name, password, and JDBC URL for the database the contains the schema.
3. Create the publication with the `createPublication` method, which creates an empty publication. An example of the `createPublication` method syntax is as follows:

   ```
   createPublication(
       java.lang.String name,
       java.lang.String db_inst,
       int client_storage_type,
       java.lang.String client_name_template,
       java.lang.String enforce_ri,
       int dev_types_flg)
   ```

   The `createPublication` method can have some of the following input parameters:

   - **name**—A character string specifying the new publication name.
   - **db_inst**—`NULL`, unless you are using a registered database for application data. If using a registered database, provide the application database name in this field.
   - **client_storage_type**—An integer specifying the client storage type for all publication items in the new publication. If you are defining a publication exclusively for a Berkeley DB or SQLite Mobile Client, specify the `Consolidator.BDB_CREATOR_ID` or `Consolidator.SQLITE_CREATOR_ID` appropriately as the storage type.

     Other values are `Consolidator.DFLT_CREATOR_ID and Consolidator.OKPI_CREATOR_ID`.

- **client_name_template**—A template for publication item instance names on client devices. This parameter contains the following predefined values:
  - `%s`—Default.
  - `DATABASE.%s`—Causes all publication items to be instantiated inside a client database with the name `DATABASE`.
  - `SFT-EE_%s`—Must be used for Satellite Forms-based applications.
- **enforce_ri**—Reserved for future use. Use `NULL` or an empty string.
- **dev_types_flg**—Specifies which device types or platforms the publication supports. The default flag is set to `Consolidator.DEV_FLG_GEN`, which includes all device platforms. If a publication is for more than one platform, use the sum of the platform flags.

  Available platforms are as follows:
  - SQLite DB: "SQLite LINUX", SQLite WCE", "SQLite WIN32", "SQLiteJava"
  - Berkeley DB: "Berkeley DB LINUX", "Berkeley DB WCE", "Berkeley DB WIN32", "SQLiteJava"

  Note: For Pure Java Client, Android and BlackBerry platforms, "SQLiteJava" should be specified.

  To retrieve the device flag for a platform, call the `getPlatformDevFlg` function. The syntax for this function is as follows:

```
int getPlatformDevFlg(java.lang.String platform)
```

  Note: Always request a drop publication before you execute a create, in case this publication already exists.

```
ConsolidatorManager consolidatorManager = new ConsolidatorManager();
consolidatorManager.openConnection(CONS_SCHEMA, DEFAULT_PASSWORD, JDBC_URL);
consolidatorManager.createPublication("T_SAMPLE11", NULL
            Consolidator.SQLITE_CREATOR_ID, "Orders.%s", NULL);
```

  Note: Special characters including spaces are supported in publication names. The publication name is case-sensitive.

After a publication is created, the client name template is fixed. If changes to the client name template are absolutely necessary, then it must be recreated by recreating the publication. For an example of how to recreate an existing publication with a different name and change the client name template, see the ReCreateTemplate.java sample. This sample demonstrates how to create a new publication and mirror the old publication's publication items, scripts, sync rules, and subscribing users in the new publication. After the new publication is created, then there will be a new client name template as well.

The full source code for this sample can be found in the following directories:

- On Windows:

```
<ORACLE_HOME>\Mobile\Server\demos\consolidator_api\recreateTemplate.bat
<ORACLE_HOME>\Mobile\Server\demos\consolidator_api\ReCreateTemplate.java
```

- On Linux:

```
<ORACLE_HOME>/mobile/server/demos/consolidator_api/recreateTemplate.sh
<ORACLE_HOME>/mobile/server/demos/consolidator_api/ReCreateTemplate.java
```

To use this sample, first run this command to compile the sample source code:

```
javac -cp .;consolidator.jar ReCreateTemplate.java.
```

Then run the recreateTemplate.bat script on Windows and ReCreateTemplate.sh script on Linux with necessary parameters. For example:

```
recreateTemplate.bat mobileadmin manager jdbc:oracle:thin:@localhost:1521:orcl MY_
PUB MY_NEW_PUB
```

2.4.1.3 Create Publication Items

An empty publication does not have anything that is helpful until a publication item is added to it. Thus, after creating the publication, it is necessary to create the publication item, which defines the snapshot of the base tables that is downloaded for your user.

> **Note:** You can create a publication using MDW. To see more details on publications and publication items, refer to Section 4.4, "Create a Publication Item".

When you create each publication item, you can specify the following:

- Column data: When you specify column data in the publication item, you should first verify what data types are supported and how others are modified when brought down to the client database.

  Also, the publication item query must select primary keys in the same order as they are defined in the base table.
- Automatic or Manual Synchronization: Whether the publication item is to be synchronized automatically or manually.
- Refresh Mode: The refresh mode of the publication item is specified during creation to be either fast, complete-refresh, or queue-based.
- Data-Subsetting Parameters: You can also establish the data-subsetting parameters when creating the publication item, which provides a finer degree of control on the data requirements for a given client.
- If you are using a registered database for application data.

> **Note:** For full details on the method parameters, see the *Oracle Database Mobile Server JavaDoc*.

Publication item names are limited to twenty-six characters and must be unique across all publications. The publication item name is case-sensitive. The following examples create a publication item named `P_SAMPLE11-M`.

> **Note:** Always drop the publication item in case an item with the same name already exists.

The following example uses the `createPublicationItem` method, which creates a manual synchronization publication item `P_SAMPLE11-M` based on the

ORD_MASTER database table with fast refresh. Use the addPublicationItem method to add this publication item to the publication.

> **Note:** For full details on the method parameters, see the *Oracle Database Mobile Server JavaDoc*.

```
consolidatorManager.createPublicationItem("P_SAMPLE11-M", "MASTER",
    "ORD_MASTER", "F", "SELECT * FROM MASTER.ORD_MASTER", NULL, NULL);
```

When you create a publication item that uses automatic synchronization through the createPublicationItem method, you can also define the following:

- Automatic Synchronization: Set the publication to use automatic synchronization by setting the isLogBased flag to true.
- Server-initiated change notifications: If you set the doChangeNtf flag to true, then the mobile server sends a notification to the client if any changes are made on the server for this publication item.
- Set what constraints are replicated to the client: If you set the setDfltColOptions flag to true, the default values and NOT NULL constraints are replicated to the client. However, if you are using a SQLite Mobile Client, then you might want to set the setDfltColOptions flag to false, as SQLite does not support the same SQL functions as Oracle. If setDfltColOptions is set to true (default) when the publication item is created, synchronization automatically uses the default clause from Oracle meta data, which is not supported by SQLite. Alternatively, you can execute the ConsolidatorManager.setPubItemColOption method to set a supported SQLite expression.
- Create a client sub-query to return unique client ids in the cl2log_rec_stmt parameter. The client sub-query correlates the primary key of the changed records in the log table with the Consolidator client id. The log table contains the changes for the table and is named clg$<tablename>.

> **Notes:**
>
> - If you are creating a fast refresh publication item on a table with a composite primary key, the snapshot query must match the primary key columns in the order that they are present in the table definition. This automatically happens during the column selection when MDW is used or when a SELECT * query is used. Note that the order of the primary key columns in the table definition may be different from those in the primary key constraint definition.
> - A subscription created as complete refresh and read only is light weight; thus, to keep the subscription light weight, the primary keys are not included in the replication. If you want to include a primary key, then you can create it with the createPublicationItemIndex API.

For example, if the publication item SQL query is as follows:

```
SELECT * FROM scott.emp a
   WHERE deptno in
       (select deptno from scott.emp b
```

```
        where b.empno = :empno )
```

Assuming that the Consolidator client id is `empno` and the snapshot table is `emp`, then the client sub-query queries for data changes in the `clg$emp` log table as follows:

```
SELECT empno as clid$$cs FROM scott.clg$emp
   UNION SELECT empno as clid$$cs FROM scott.emp
         WHERE deptno in (select deptno from scott.clg$dept)
```

The following example uses the automatic synchronization version of `createPublicationItem` method, which uses the `PubItemProps` class to define all publication item definitions, including automatic synchronization, as follows:

```
PubItemProps pi_props = new PubItemProps();
pi_props.db_inst = NULL;          // Provide registered db instance name or NULL
pi_props.owner = "MASTER";                  // owner schema
pi_props.store = STORES[i][0];              // store
pi_props.refresh_mode = "F"; //default      // uses fast refresh
pi_props.select_stmt =                  // specify select statement for snapshot
   "SELECT * FROM "+"MASTER"+"."+STORES[i][0]+ " WHERE C1 =:CLIENTID";
pi_props.cl2log_rec_stmt = "SELECT base.C1 FROM "   // client sub-query to
      + "MASTER"+"."+STORES[i][0] + " base,"        // return unique clientids
      + "MASTER"+".CLG$"+STORES[i][0] + " log"
      + "  WHERE base.ID = log.ID";
// Setting "isLogBased" to True enables automatic sync for this pub item.
pi_props.isLogBased = true;
// If doChangeNtf is true, automatic publication item sends notifications
// from server about new/modified records
pi_props.doChangeNtf = true;

cm.createPublicationItem(PUBITEMS[i], pi_props);
cm.addPublicationItem(PUB,PUBITEMS[i],NULL,NULL,"S",NULL,NULL);
```

2.4.1.3.1 Defining Publication Items for Updatable Multi-Table Views Publication items can be defined for both tables and views. When publishing updatable multi-table views, the following restrictions apply:

- The view must contain a parent table with a primary key defined.
- `INSTEAD OF` triggers must be defined for data manipulation language (DML) operations on the view. See Section 2.8, "Understanding Your Refresh Options" for more information.
- All base tables of the view must be published.

2.4.1.4 Define Publication-Level Automatic Synchronization Rules

Once the publication is created, you can create and add automatic synchronization rules that apply to all enabled publication items in this publication. Perform the following to add a rule to a publication:

1. The rule is made up of a rule name and a `String` that contains the rule definition. The rules can be created using the `Rules` classes and `RuleInfo` objects.
   a. Define the rule and convert it to a `String` using the `RuleInfo` object and the `setSyncRuleParams` method.

```
RuleInfo ri = Rules.RULE_MAX_DB_REC_ri;
ri.params.put(Rules.PARAM_NREC,"5");
String ruleText = cm.setSyncRuleParams(ri.type,ri.params);
```

There are `RuleInfo` objects for all of the main automatic synchronization rules. So, in order to specify a rule, you obtain the appropriate `RuleInfo` object from the Rules class and then define the variable. Table 2–9, " Automatic Synchronization Rule Info Objects" describes the different types of rules you can specify for triggering automatic synchronization:

Note: See the *Oracle Database Mobile Server JavaDoc* for syntax and the parameters that you need to set for each rule.

Table 2–9 Automatic Synchronization Rule Info Objects

Rule Info Object	Description
`RULE_MAX_DB_REC_ri`	For mobile clients only. An automatic synchronization is triggered if the client transaction log contains more than NREC modified records.
`RULE_NOTIFY_MAX_PUB_REC_ri`	Synchronize if the Out Queue contains more than `NREC` modified records, where you specify the `NREC` of modifed records in the server database to trigger an automatic synchronization.
`RULE_MAX_PI_REC_ri`	Client automatically synchronizes if the number of modified records for a publication item is greater than `NREC`.
`RULE_HIGH_BANDWIDTH_ri`	Synchronize when the network bandwidth is greater than `<number>` bits/second. Where `<number>` is an integer that indicates the bandwidth bits/seconds. When the bandwidth is at this value, the synchronization occurs.
`RULE_LOW_PWR_ri`	Synchronize when the battery level drops to `<number>`%, where `<number>` is a percentage. Often you may wish to synchronize before you lose battery power.
`RULE_AC_PWR_ri`	Synchronize when the AC power is detected; that is, when the device is plugged in.
`RULE_MIN_MEM_ri`	Specify the minimum battery level required in order for an automatic synchronization to start. The battery level is specified as a percentage.
`RULE_NET_PRIORITY_ri`	Network conditions can be specified using the following properties: data priority, ping delay and network bandwidth.
`RULE_MIN_PWR_ri`	If the battery life drops below a pre-defined minimum, then synchronization is automatically triggered.
`NET_CONFIG_ri`	Configure network parameters (currently only the network specific proxy configuration is supported) The configuration rule contains a vector of hashtables with a hashtable representing properties of each individual network.

Table 2–9 (Cont.) Automatic Synchronization Rule Info Objects

Rule Info Object	Description
RULE_TIME_INTERVAL_ri	Schedule sync at a given time of day with a certain frequency (interval). Specify the time (PARAM_START_TIME) for an automatic synchronization to start. The format of time is standard date string: H24:MI:SS e.g. 00:00:00 or 23:59:00 The time is GMT. If not set, the synchronization starts when the Sync Agent starts and all other conditions are satisfied Set the period (PARAM_PERIOD), in seconds, to specify the frequency of scheduled synchronization events.

b. Define a name for the rule, which should be a name not attached to any particular publication, so you can use the rule for several publications.

2. Create the rule with the `createSyncRule` method, which creates the rule with the name, the `String` containing the rule, and a boolean on whether to replace the rule if it already exists. Once completed, then this rule can be associated with any publication.

```
boolean replace = true;
cm.createSyncRule ( ruleName, ruleText, replace );
```

3. Associate the rule with the desired publication or platform using the `addSyncRule` method. This method can add any existing rule to a designated publication. To add to a publication, use the publication name as the first parameter, as follows:

```
cm.addSyncRule( PUB, ruleName );
```

To add a rule to a client platform—Win32 or Windows Mobile platform—perform the following:

```
cm.addSyncRule( Consolidator.DEFAULT_TEMPLATE_WIN32, rulename );
```

Where the platform name is a constant defined in the Consolidator class as `DEFAULT_TEMPLATE_WIN32`, `DEFAULT_TEMPLATE_WCE`, DEFAULT_TEMPLATE_LINUX or DEFAULT_TEMPLATE_JAVA.

You can also perform the following:

- Section 2.4.1.4.1, "Retrieve All Publications Associated with a Rule"
- Section 2.4.1.4.2, "Retrieve Rule Text"
- Section 2.4.1.4.3, "Check if Rule is Modified"
- Section 2.4.1.4.4, "Remove Rule"

2.4.1.4.1 Retrieve All Publications Associated with a Rule Just as you can with scripts and sequences that are associated with publications, you can retrieve all publications that are associated with a rule with the `getPublicationNames` method. The following retrieves all publications that are associated with the rule within the `ruleName` variable. The object type is defined as `Consolidator.RULES_OBJECT`.

```
String[] pubs  = cm.getPublicationNames ( ruleName , Consolidator.RULES_OBJECT);
```

2.4.1.4.2 Retrieve Rule Text You can retrieve the text of the rule using the `getSyncRule` and providing the rule name. This is useful if you are not sure what the rule is and need to discover the text before associating it with another publication.

```
String retStr = cm.getSyncRule ( ruleName );
```

2.4.1.4.3 Check if Rule is Modified You can compare the rule within the repository with a provided string to see if the rule has been modified with the `isSyncRuleModified` method. A boolean value of true is returned if the provided `ruleText` is different from what exists in the repository.

```
boolean ismod = cm.isSyncRuleModified ( ruleName, ruleText );
```

2.4.1.4.4 Remove Rule You can remove the association of a rule from a publication by using the `removeSyncRule` method. You can delete the entire rule from the repository by using the `dropSyncRule` method. If you drop the rule and it is still associated with one or more publications, the rule is automatically unassociated from these publications.

2.4.1.5 Data Subsetting: Defining Client Subscription Parameters for Publications

Data subsetting is the ability to create specific subsets of data and assign them to a parameter name that can be assigned to a subscribing user. When creating publication items, a parameterized `Select` statement can be defined. Subscription parameters must be specified at the time the publication item is created, and are used during synchronization to control the data published to a specific client.

Creating a Data Subset Example

```
consolidatorManager.createPublicationItem("CORP_DIR1",
   "DIRECTORY1", "ADDRLRL4P", "F" ,
   "SELECT LastName, FirstName, company, phone1, phone2, phone3, phone4,
    phone5, phone1id, phone2id, phone3id, displayphone, address, city, state,
    zipcode, country, title, custom1, custom2, custom3, note
    FROM directory1.addrlrl4p WHERE company = :COMPANY", NULL, NULL);
```

In this sample statement, data is being retrieved from a publication named `CORP_DIR1`, and is subset by the variable `COMPANY`.

> **Note:** Within the select statement, the parameter name for the data subset must be prefixed with a colon, for example `:COMPANY`.

When a publication uses data subsetting parameters, set the parameters for each subscription to the publication. For example, in the previous example, the parameter `COMPANY` was used as an input variable to describe what data is returned to the client. You can set the value for this parameter with the `setSubscriptionParameter` method. The following example sets the subscription parameter `COMPANY` for the client `DAVIDL` in the `CORP_DIR1` publication to `DAVECO`:

```
consolidatorManager.setSubscriptionParameter("CORP_DIR1", "DAVIDL",
       "COMPANY", "'DAVECO'");
```

> **Note:** This method should only be used on publications created using the Consolidator Manager API. To create template variables, a similar technique is possible using MDW.

2.4.1.6 Create Publication Item Indexes

The mobile server supports automatic deployment of indexes on mobile clients. The mobile server automatically replicates primary key indexes from the server database.

The Consolidator Manager API provides calls to explicitly deploy unique, regular, and primary key indexes to clients as well.

By default, the primary key index of a table is automatically replicated from the server. You can create secondary indexes on the snapshot table for a publication item. If you do not want the primary index, you must explicitly drop it from the publication item.

If you want to create and associate other indexes on any columns in your application tables in the publication item, then use the `createPublicationItemIndex` method. You can drop an index from the publication item and from the snapshot table with the `dropPublicationItemIndex` method.

The following demonstrates how to set up indexes on the name field in our publication item `P_SAMPLE11-M`:

```
consolidatorManager.createPublicationItemIndex("P_SAMPLE11M-I3",
   "P_SAMPLE11-M", "I", "NAME");
```

An index can contain more than one column. You can define an index with multiple columns, as follows:

```
consolidatorManager.createPublicationItemIndex("P_SAMPLE11D-I1", "P_SAMPLE11-D",
      "I", "KEY,NAME");
```

Note: All indexes created by this API can be viewed within the `CV$ALL_PUBLICATIONS_INDEXES` view.

2.4.1.6.1 Define Client Indexes Client-side indexes can be defined for existing publication items. There are three types of indexes that can be specified:

- P - Primary key is an index based off of the primary keys.
- U - Unique enforces the unique constraint on the indexed columns, which ensures that duplicate values do not exist in the columns being indexed.
- I - Regular does not provide the UNIQUE constraint on the indexed columns.

Note: When an index of type 'U' or 'P' is defined on a publication item, there is no check for duplicate keys on the server. If the same constraints do not exist on the base object of the publication item, synchronization may fail with a duplicate key violation. See the *Oracle Database Mobile Server API Specification* for more information.

2.4.1.7 Adding Publication Items to Publications

Once you create a publication item, you must associate it with a publication using the `addPublicationItem` method, as follows:

```
consolidatorManager.addPublicationItem("T_SAMPLE11", "P_SAMPLE11-M",
     NULL, NULL, "S", NULL, NULL);
```

See Section 2.4.1.12, "Modifying a Publication Item" for details on how to change the definition.

2.4.1.7.1 Defining Conflict Rules When adding a publication item to a publication, the user can specify winning rules to resolve synchronization conflicts in favor of either the client or the server. See Section 2.10, "Resolving Conflicts with Winning Rules" for more information.

2.4.1.7.2 Using Table Weight Table weight is an integer associated with publication items that determines in what order the transactions for all publication items within the publication are processed. For example, if three publication items exist—one that contains SQL to modify the `emp` table, one that modifies the `dept` table, and one that modifies the `mgr` table, then you can define the order in which the transactions associated with each publication item are executed. In our example, assign table weight of 1 to the publication item that contains the `dept` table, table weight of 2 to the publication item that contains the `mgr` table, and table weight of 3 to the publication item that contains the `emp` table. In doing this, you ensure that the publication item that contains the master table `dept` is always processed first, followed by the publication item that modifies the `mgr` table, and lastly by the publication item that modifies the `emp` table.

The insert, update, and delete client operations are executed in the following order:

1. Client `INSERT` operations are executed first, from lowest to highest table weight order. This ensures that the master table entries are added before the details table entries.
2. Client `DELETE` operations are executed next, from highest to lowest table weight order. Processing the delete operations ensures that the details table entries are removed before the master table entries.
3. Client `UPDATE` operations are executed last, from highest to lowest table weight order.

In our example with `dept`, `mgr`, and `emp` tables, the execution order would be as follows:

1. All insert operations for `dept` are processed.
2. All insert operations for `mgr` are processed.
3. All insert operations for `emp` are processed.
4. All delete operations for `emp` are processed.
5. All delete operations for `mgr` are processed.
6. All delete operations for `dept` are processed.
7. All update operations for `emp` are processed.
8. All update operations for `mgr` are processed.
9. All update operations for `dept` are processed.

A publication can have more than one publication item of weight 2. In this case, it does not matter which publication is executed first.

Define the order weight for publication items when you add it to the publication.

2.4.1.8 Creating Client-Side Sequences for the Downloaded Snapshot

A sequence is a database schema object that generates sequential numbers. After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. These unique integers can include primary key values. If a transaction generates a sequence number, the sequence is incremented immediately whether you commit or roll back the transaction. For full details of what a sequence is and how Oracle Database Mobile Server creates them, see Section 4.6, "Create a Sequence".

Note: Sequences are only supported in Berkeley DB Mobile Clients.

If you do not want to use MDW to create a sequence, you can use the Consolidator Manager API to manage the sequences with methods that create/drop a sequence, add/remove a sequence from a publication, modify a sequence, and advance a sequence window for each user. All of the same behavior exists for the Consolidator Manager APIs as are available through MDW.

> **Note:** The sequence name is case-sensitive.

Once you have created the sequence, you place it into the publication with the publication item to which it applies.

> **Note:** If the sequences do not work properly, check your parent publications. All parent publications must have at least one publication item. If you do not have any publication items for the parent publication, then create a dummy publication item within the parent.

See the *Oracle Database Mobile Server API Specification* for a complete listing of the APIs to define and administrate sequences.

2.4.1.9 Subscribing Users to a Publication

Subscribe the users to a publication using the `createSubscription` function. The following creates a subscription between the `S11U1` user and the `T_SAMPLE11` publication:

```
consolidatorManager.createSubscription("T_SAMPLE11", "S11U1");
```

2.4.1.10 Instantiate the Subscription

After you subscribe a user to a publication, you complete the subscription process by instantiating the subscription, which associates the user with the publication in the back-end database. The next time that the user synchronizes, the data snapshot from the publication is provided to the user.

```
consolidatorManager.instantiateSubscription("T_SAMPLE11", "S11U1");

//Close the connection.
consolidatorManager.closeConnection();
```

> **Note:** If you need to set subscription parameters for data subsetting, this must be completed before instantiating the subscription. See Section 2.4.1.5, "Data Subsetting: Defining Client Subscription Parameters for Publications" for more information.

2.4.1.11 Bringing the Data From the Subscription Down to the Client

You can perform the synchronization and bring down the data from the subscription you just created. The client executes SQL queries against the client database to retrieve any information. This subscription is not associated with any application, as it was created using the low-level Consolidator Manager APIs.

2.4.1.12 Modifying a Publication Item

You can add additional columns to existing publication items. These new columns are pushed to all subscribing clients the next time they synchronize. This is accomplished through a complete refresh of all changed publication items.

- An administrator can add multiple columns, modify the WHERE clause, add new parameters, and change data type.
- This feature is supported for all mobile client platforms.
- The client does not upload snapshot information to the server. This also means the client cannot change snapshots directly on the client database.
- Publication item upgrades are deferred during high priority synchronizations. This is necessary for low bandwidth networks, such as wireless, because all publication item upgrades require a complete refresh of changed publication items. While the high priority flag is set, high priority clients continues to receive the old publication item format.
- The server needs to support a maximum of two versions of the publication item which has been altered.

To change the definition, use one of the following:

- If the publication item is read-only, then modify the publication item either with the `reCreatePublicationItem` method or by dropping and creating the publication item with the `dropPublicationItem` and `createPublicationItem` APIs.
- If the publication item is updatable, then you can use the `alterPublicationItem` method. This method enables a smooth transition of changing any table structure on both the client and the server for updatable publications.

  If you use the `alterPublicationItem` method, you must follow it up by executing the `resetCache` method. The metadata cache should be reset every time a change is made to the publication or publication items. If you make the change though Mobile Manager, then the Mobile Manager calls the `resetCache` method. You can reset the metadata cache from the Mobile Manager or execute the `resetCache` method, part of the `ConsolidatorManager` class.

  You may use the `alterPublicationItem` method for schema evolution to add columns to an existing publication item. The WHERE clause may also be altered. If additional parameters are added to the WHERE clause, then these parameters must be set before the alter occurs. See the `setSubscriptionParams` method. However, if you are creating a fast refresh publication item on a table with a composite primary key, the snapshot query must match the primary key columns in the order that they are present in the table definition. This automatically happens during the column selection when MDW is used or when a SELECT * query is used. Note that the order of the primary key columns in the table definition may be different from those in the primary key constraint definition.

```
consolidatorManager.alterPublicationItem("P_SAMEPLE1", "select * from
        EMP");
```

> **Note:** If the select statement does not change, then the call to the `alterPublicationItem()` method has no effect.

2.4.1.13 Callback Customization for DML Operations

Once a publication item has been created, a user can use the Consolidator Manager API to specify a customized PL/SQL procedure that is stored in the mobile server repository to be called in place of all DML operations for that publication item. There can be only one DML procedure for each publication item. The procedure should be created as follows:

```
AnySchema.AnyPackage.AnyName(DML in CHAR(1), COL1 in TYPE, COL2 in TYPE,
   COLn.., PK1 in TYPE, PK2 in TYPE, PKn..)
```

> **Note:** You can use the `generateMobileDMLProcedure` to generate the procedure specification for a given publication item. This specification can be used as a starting point in creating your own custom DML handling logic in a PL/SQL procedure. See the *Oracle Database Mobile Server API Specification* for more information.

The parameters for customizing a DML operation are listed in Table 2–10:

Table 2–10 Mobile DML Operation Parameters

Parameter	Description
`DML`	DML operation for each row. Values can be "D" for DELETE, "I" for INSERT, or "U" for UPDATE.
`COL1 ... COLn`	List of columns defined in the publication item. The column names must be specified in the same order that they appear n the publication item query. If the publication item was created with "SELECT * FROM exp", the column order must be the same as they appear in the table "exp".
PK1 ... PKn	List of primary key columns. The column names must be specified in the same order that they appear in the base or parent table.

The following defines a DML procedure for publication item `exp`:

```
select A,B,C from publication_item_exp_table
```

Assuming `A` is the primary key column for `exp`, then your DML procedure would have the following signature:

```
any_schema.any_package.any_name(DML in CHAR(1), A in TYPE, B in TYPE, C
                  in TYPE,A_OLD in TYPE)
```

During runtime, this procedure is invoked with 'I', 'U', or 'D' as the DML type. For insert and delete operations, `A_OLD` is `NULL`. In the case of updates, it is set to the primary key of the row that is being updated. Once the PL/SQL procedure is defined, it can be attached to the publication item through the following API call:

```
consolidatorManager.addMobileDmlProcedure("PUB_exp","exp",
                                          "any_schema.any_package.any_name")
```

where `exp` is the publication item name and `PUB_exp` is the publication name.

Refer to the *Oracle Database Mobile Server API Specification* for more information.

2.4.1.13.1 DML Procedure Example The following piece of PL/SQL code defines an actual DML procedure for a publication item in one of the sample publications. As described below, the `ORD_MASTER` table. The query was defined as:

```
SELECT * FROM "ord_master", where ord_master has a single
             column primary key on "ID"
```

ord_master Table

```
SQL> desc ord_master
Name                                      Null?    Type
----------------------------------------- -------- -------------
ID                                        NOT NULL NUMBER(9)
DDATE                                              DATE
STATUS                                             NUMBER(9)
NAME                                               VARCHAR2(20)
DESCRIPTION                                        VARCHAR2(20)
```

Code Example

```
CREATE OR REPLACE PACKAGE SAMPLE11.ORD_UPDATE_PKG AS
PROCEDURE UPDATE_ORD_MASTER (DML IN CHAR, ID IN NUMBER, DDATE IN DATE,
STATUS IN NUMBER, NAME IN VARCHAR2, DESCRIPTION IN VARCHAR2, ID_OLD IN
NUMBER);
END ORD_UPDATE_PKG;
/
.
CREATE OR REPLACE PACKAGE BODY SAMPLE11.ORD_UPDATE_PKG AS
PROCEDURE UPDATE_ORD_MASTER (DML IN CHAR, ID IN NUMBER, DDATE IN DATE,
STATUS IN NUMBER, NAME IN VARCHAR2, DESCRIPTION IN VARCHAR2, ID_OLD IN
NUMBER) is
  begin
    if DML = 'U' then
     execute immediate 'update ord_master set id = :id, ddate = :ddate,
status = :status,
name = :name, description = '''||'from
ord_update_pkg'||''' where id = :id_old'
      using id, ddate, status, name, id_old;
    end if;
    if DML = 'I' then
 begin
      execute immediate 'insert into ord_master values(:id, :ddate,
:status, :name, '''||'from ord_update_pkg'||''')'
        using id, ddate, status, name;
 exception
  when others then NULL
 end;
    end if;
    if DML = 'D' then
     execute immediate 'delete from ord_master where id = :id'
      using id;
    end if;
  end UPDATE_ORD_MASTER;
end ORD_UPDATE_PKG;
/
```

The API call to add this DML procedure is as follows:

```
consolidatorManager.addMobileDMLProcedure("T_SAMPLE11",
          "P_SAMPLE11-M","SAMPLE11.ORD_UPDATE_PKG.UPDATE_ORD_MASTER")
```

where `T_SAMPLE11` is the publication name and `P_SAMPLE11-M` is the publication item name.

2.4.1.14 Restricting Predicate

A restricting predicate can be assigned to a publication item as it is added to a publication.The predicate is used to limit data downloaded to the client. The parameter, which is for advanced use, can be `NULL`. For using a restricting predicate, see Section 1.2.10 "Priority-Based Replication" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide*.

2.5 Client Device Database DDL Operations

The first time a client synchronizes, Oracle Database Mobile Server automatically creates the snapshot tables for the user subscriptions on the mobile client. If you would like to execute additional DDL statements on the database, add the DDL statements as part of your publication. Oracle Database Mobile Server executes these DDL statements when the user synchronizes.

This is typically used for adding constraints and check values.

For example, you can add a foreign key constraint to a publication item. In this instance, if the Oracle Database Mobile Server created snapshots S1 and S2 during the initial synchronization, where the definition of S1 and S2 are as follows:

```
S1 (C1 NUMBER PRIMARY KEY, C2 VARCHAR2(100), C3 NUMBER);
S2 (C1 NUMBER PRIMARY KEY, C2 VARCHAR2(100), C3 NUMBER);
```

If you would like to create a foreign key constraint between C3 on S2 and the primary key of S1 , then add the following DDL statement to your publication item:

> **Note:** '"ALTER TABLE...ADD CONSTRAINT" syntax is not supported in Berkeley DB and SQLite, so a trigger is used to add a foreign key constraint in this example. Refer to `http://www.sqlite.org/omitted.html` for the SQL features that SQLite does not implement.

```
CREATE TRIGGER S2_FK
BEFORE INSERT ON S2 FOR EACH ROW
BEGIN SELECT RAISE(FAIL, 'foreign key constraint S2_FK violated')
WHERE (SELECT C1 FROM S1 WHERE C1=NEW.C3) IS NULL;
END;
```

Then, Oracle Database Mobile Server executes any DDL statements after the snapshot creation or, if the snapshot has already been created, after the next synchronization.

See the *Oracle Database Mobile Server API Specification* for more information on these APIs.

2.6 Customize the Compose Phase Using MyCompose

The compose phase takes a query for one or more server-side base tables and puts the generated DML operations for the publication item into the Out Queue to be downloaded into the client. The Consolidator Manager manages all DML operations using the physical DML logs on the server-side base tables. This can be resource intensive if the DML operations are complex—for example, if there are complex data-subsetting queries being used. The tools to customize this process include an extendable `MyCompose` with compose methods which can be overridden, and additional `ConsolidatorManager` APIs to register and load the customized class.

Note: See the *Oracle Database Mobile Server API Specification* for more information on these APIs.

When you want to customize the compose phase of the synchronization process, you must perform the activities described in the following sections:

- Section 2.6.1, "Create a Class That Extends MyCompose to Perform the Compose"
- Section 2.6.2, "Implement the Extended MyCompose Methods in the User-Defined Class"
- Section 2.6.3, "Use Get Methods to Retrieve Information You Need in the User-Defined Compose Class"
- Section 2.6.4, "Register the User-Defined Class With the Publication Item"

2.6.1 Create a Class That Extends MyCompose to Perform the Compose

The `MyCompose` class is an abstract class, which serves as the super-class for creating a user-written sub-class, as follows:

```
public class ItemACompose extends oracle.lite.sync.MyCompose
{ ... }
```

All user-written classes—such as `ItemACompose`—produce publication item DML operations to be sent to a client device by interpreting the base table DML logs. The sub-class is registered with the publication item, and takes over all compose phase operations for that publication item. The sub-class can be registered with more than one publication item—if it is generic—however, internally the Composer makes each instance of the extended class unique within each publication item.

2.6.2 Implement the Extended MyCompose Methods in the User-Defined Class

The `MyCompose` class includes the following methods—`needCompose`, `doCompose`, `init`, and `destroy`—which are used to customize the compose phase. One or more of these methods can be overridden in the sub-class to customize compose phase operations. Most users customize the compose phase for a single client. In this case, only implement the `doCompose` and `needCompose` methods. The `init` and `destroy` methods are only used when a process is performed for all clients, either before or after individual client processing.

The following sections describe how to implement these methods:

- Section 2.6.2.1, "Implement the needCompose Method"
- Section 2.6.2.2, "Implement the doCompose Method"
- Section 2.6.2.3, "Implement the init Method"
- Section 2.6.2.4, "Implement the destroy Method"

Note: To retrieve information, use the methods described in Section 2.6.3, "Use Get Methods to Retrieve Information You Need in the User-Defined Compose Class".

2.6.2.1 Implement the needCompose Method

The needCompose method to identifies a client that has changes to a specific publication item that is to be downloaded. Use this method as a way to trigger the doCompose method.

```
public int needCompose(Connection conn, Connection rmt_conn, String clientid)
 throws Throwable
```

The parameters for the needCompose method are listed in Table 2–11:

Table 2–11 needCompose Parameters

Parameter	Definition
conn	Database connection to the Main mobile server repository.
rmt_conn	Database connection to the remote database for application. Set to NULL if the base tables are on the Main database where the repository exists.
clientid	Specifies the client that is being composed.

The following example examines a client base table for changes—in this case, the presence of dirty records. If there are changes, then the method returns MyCompose.YES, which triggers the doCompose method.

```
public int needCompose(Connection conn, Connection rmtConn, String clientid)
 throws Throwable{
        boolean baseDirty = false;
        String [][] baseTables = this.getBaseTables();

        for(int i = 0; i < baseTables.length; i++){
            if(this.baseTableDirty(baseTables[i][0], baseTables[i][1])){
                baseDirty = true;
                break;
            }
        }

        if(baseDirty){
            return MyCompose.YES;
        }else{
            return MyCompose.NO;
        }
    }
```

This sample uses subsidiary methods discussed in Section 2.6.3, "Use Get Methods to Retrieve Information You Need in the User-Defined Compose Class" to check if the publication item has any tables with changes that need to be sent to the client. In this example, the base tables are retrieved, then checked for changed, or dirty, records. If the result of that test is true, a value of Yes is returned, which triggers the call for the doCompose method.

2.6.2.2 Implement the doCompose Method

The doCompose method populates the DML log table for a specific publication item, which is subscribed to by a client.

```
public int doCompose(Connection conn, Connection rmt_conn,
   String clientid) throws Throwable
```

The parameters for the doCompose method are listed in Table 2–12:

Table 2–12 doCompose Parameters

Parameter	Definition
conn	Database connection to the Main mobile server repository.
rmt_conn	Database connection to the remote database for application. Set to NULL if the base tables are on the Main database where the repository exists.
clientid	Specifies the client that is being composed.

The following example contains a publication item with only one base table where a DML (Insert, Update, or Delete) operation on the base table is performed on the publication item. This method is called for each client subscribed to the publication item.

```
public int doCompose(Connection conn, Connection rmtConn, String clientid)
              throws Throwable {
        int rowCount = 0;

        Connection auxConn = rmtConn;
        if(auxConn == NULL)
            auxConn = rmtConn;

        String[][] baseTables = getBaseTables();
        String baseTableDMLLogName =
            getBaseTableDMLLogName(baseTables[0][0], baseTables[0][1]);
        String baseTablePK =
            getBaseTablePK(baseTables[0][0], baseTables[0][1]);
        String pubItemDMLTableName = getPubItemDMLTableName();
        String pubItemPK = getPubItemPK();
        String mapView = getMapView(clientid);

        Statement st = auxConn.createStatement();
        String sql = NULL;

        // insert
        sql = "INSERT INTO " + pubItemDMLTableName + " SELECT " + baseTablePK +
             ", DMLTYPE$$ FROM " + baseTableDMLLogName;

        rowCount += st.executeUpdate(sql);

        st.close();
        return rowCount;
    }
```

This code uses subsidiary methods discussed in Section 2.6.3, "Use Get Methods to Retrieve Information You Need in the User-Defined Compose Class" to create a SQL statement. The `MyCompose` method retrieves the base table, the base table primary key, the base table DML log name and the publication item DML table name using the appropriate `get` methods. You can use the table names and other information returned by these methods to create a dynamic SQL statement, which performs an insert into the publication item DML table of the contents of the base table primary key and DML operation from the base table DML log.

2.6.2.3 Implement the init Method

The `init` method provides the framework for user-created compose preparation processes. The `init` method is called once for all clients before the individual client compose phase. The default implementation has no effect.

```
public void init(Connection conn)
```

The parameter for the `init` method is described in Table 2–13:

Table 2–13 init Parameters

Parameter	Definition
`conn`	Database connection to the Main mobile server repository.

2.6.2.4 Implement the destroy Method

The `destroy` method provides the framework for compose cleanup processes. The `destroy` method is called once for all clients after to the individual client compose phase. The default implementation has no effect.

```
public void destroy(Connection conn)
```

The parameter for the `destroy` method is described in Table 2–14:

Table 2–14 destroy Parameters

Parameter	Definition
`conn`	Database connection to the Main mobile server repository.

2.6.3 Use Get Methods to Retrieve Information You Need in the User-Defined Compose Class

The following methods return information for use by primary `MyCompose` methods.

- Section 2.6.3.1, "Retrieve the Publication Name With the getPublication Method"
- Section 2.6.3.2, "Retrieve the Publication Item Name With the getPublicationItem Method"
- Section 2.6.3.3, "Retrieve the DML Table Name With the getPubItemDMLTableName Method"
- Section 2.6.3.4, "Retrieve the Primary Key With the getPubItemPK Method"
- Section 2.6.3.5, "Retrieve All Base Tables With the getBaseTables Method"
- Section 2.6.3.6, "Retrieve the Primary Key With the getBaseTablePK Method"
- Section 2.6.3.7, "Discover If Base Table Has Changed With the baseTableDirty Method"
- Section 2.6.3.8, "Retrieve the Name for DML Log Table With the getBaseTableDMLLogName Method"
- Section 2.6.3.9, "Retrieve View of the Map Table With the getMapView Method"

2.6.3.1 Retrieve the Publication Name With the getPublication Method

The `getPublication` method returns the name of the publication.

```
public String getPublication()
```

2.6.3.2 Retrieve the Publication Item Name With the getPublicationItem Method

The `getPublicationItem` method returns the publication item name.

```
public String getPublicationItem()
```

2.6.3.3 Retrieve the DML Table Name With the getPubItemDMLTableName Method

The `getPubItemDMLTableName` method returns the name of the DML table or DML table view, including schema name, which the `doCompose` or `init` methods are supposed to insert into.

```
public String getPubItemDMLTableName()
```

You can embed the returned value into dynamic SQL statements. The table or view structure is as follows:

```
<PubItem PK> DMLTYPE$$
```

The parameters for `getPubItemDMLTableName` are listed in Table 2–15:

Table 2–15 getPubItemDMLTableName View Structure Parameters

Parameter	Definition
`PubItemPK`	The value returned by `getPubItemPK()`
`DMLTYPE$$`	This can have the values 'I' for insert, 'D' for delete, or 'U' for Update.

2.6.3.4 Retrieve the Primary Key With the getPubItemPK Method

Returns the primary key for the listed publication in comma separated format in the form of `<col1>,<col2>,<col3>`.

```
public String getPubItemPK() throws Throwable
```

2.6.3.5 Retrieve All Base Tables With the getBaseTables Method

Returns all the base tables for the publication item in an array of two-string arrays. Each two-string array contains the base table schema and name. The parent table is always the first base table returned, in other words, `baseTables[0]`.

```
public string [][] getBaseTables() throws Throwable
```

2.6.3.6 Retrieve the Primary Key With the getBaseTablePK Method

Returns the primary key for the listed base table in comma separated format, in the form of `<col1>, col2>,<col3>`.

```
public String getBaseTablePK (String owner, String baseTable) throws Throwable
```

The parameters for `getBaseTablePK` are listed in Table 2–16:

Table 2–16 getBaseTablePK Parameters

Parameter	Definition
`owner`	The schema name of the base table owner.
`baseTable`	The base table name.

2.6.3.7 Discover If Base Table Has Changed With the baseTableDirty Method

Returns the a boolean value for whether or not the base table has changes to be synchronized.

```
public boolean baseTableDirty(String owner, String store)
```

The parameters for `baseTableDirty` are listed in Table 2–17:

Table 2–17 baseTableDirty Parameters

Parameter	Definition
`owner`	The schema name of the base table.
`store`	The base table name.

2.6.3.8 Retrieve the Name for DML Log Table With the getBaseTableDMLLogName Method

Returns the name for the physical DML log table or DML log table view for a base table.

```
public string getBaseTableDMLLogName(String owner, String baseTable)
```

The parameters for `getBaseTableDMLLogName` are listed in Table 2–18:

Table 2–18 getBaseTableDMLLogName Parameters

Parameter	Definition
`owner`	The schema name of the base table owner.
`baseTable`	The base table name.

You can embed the returned value into dynamic SQL statements. There may be multiple physical logs if the publication item has multiple base tables. The parent base table physical primary key corresponds to the primary key of the publication item. The structure of the log is as follows:

```
<Base Table PK> DMLTYPE$$
```

The parameters for `getBaseTableDMLLogName` view structure are listed in Table 2–19:

Table 2–19 getBaseTableDMLLogName View Structure Parameters

Parameter	Definition
`Base Table PK`	The primary key of the parent base table.
`DMLTYPE$$`	This can have the values 'I' for insert, 'D' for delete, or 'U' for Update.

2.6.3.9 Retrieve View of the Map Table With the getMapView Method

Returns a view of the map table which can be used in a dynamic SQL statement and contains a primary key list for each client device. The view can be an inline view.

```
public String getMapView() throws Throwable
```

The structure of the map table view is as follows:

```
CLID$$CS <Pub Item PK> DMLTYPE$$
```

The parameters of the map table view are listed in Table 2–20:

Table 2–20 getMapView View Structure Parameters

Parameter	Definition
`CLID$$CS`	This is the client ID column.
`Base Table PK`	The primary key columns of the publication item.

Table 2–20 (Cont.) getMapView View Structure Parameters

Parameter	Definition
DMLTYPE$$	This can have the values 'I' for insert, 'D' for delete, or 'U' for Update.

2.6.4 Register the User-Defined Class With the Publication Item

Once you have created your sub-class, it must be registered with a publication item. The Consolidator Manager API now has two methods `registerMyCompose` and `deRegisterMyCompose` to permit adding and removing the sub-class from a publication item.

- The `registerMyCompose` method registers the sub-class and loads it into the mobile server repository, including the class byte code. By loading the code into the repository, the sub-class can be used without having to be loaded at runtime.
- The `deRegisterMyCompose` method removes the sub-class from the mobile server repository.

2.7 Customize What Occurs Before and After Synchronization Phases

You can customize what happens before and after certain synchronization processes by creating one or more PL/SQL packages. The following sections detail the different options you have for customization:

- Section 2.7.1, "Customize What Occurs Before and After Every Phase of Each Synchronization"
- Section 2.7.2, "Customize What Occurs Before and After Compose/Apply Phases for a Single Publication Item"

2.7.1 Customize What Occurs Before and After Every Phase of Each Synchronization

You can customize the MGP phase of the synchronization process through a set of predefined callback methods that add functionality to be executed before or after certain phases of the synchronization process. These callback methods are defined in the `CUSTOMIZE` PL/SQL package. Note that these callback methods are called before or after the defined phase for every publication item.

> **Note:** If you want to customize certain activity for only a specific publication item, see Section 2.7.2, "Customize What Occurs Before and After Compose/Apply Phases for a Single Publication Item" for more information.

Manually create this package in the mobile server repository and any remote database that has publication items that are relevant for the customization.

The methods and their respective calling sequence are as follows:

> **Note:** Some of the procedures in the package are invoked for each client defined in your mobile server, such as the `BeforeClientCompose` and `AfterClientCompose` methods.

- Section 2.7.1.1, "NullSync"

2.7.1.1 NullSync

The `NullSync` procedure is called at the beginning of every synchronization session. It can be used to determine whether or not a particular user is uploading data.

```
procedure NullSync (clientid varchar2, isNullSync boolean);
```

2.7.1.2 BeforeProcessApply

The `BeforeProcessApply` procedure is called before the entire apply phase of the MGP process.

```
procedure BeforeProcessApply;
```

2.7.1.3 AfterProcessApply

The `AfterProcessApply` procedure is called after the entire apply phase of the MGP process.

```
procedure AfterProcessApply;
```

2.7.1.4 BeforeProcessCompose

The `BeforeProcessCompose` procedure is called before the entire compose phase of the MGP process.

```
procedure BeforeProcessCompose;
```

2.7.1.5 AfterProcessCompose

The `AfterProcessCompose` procedure is called after the entire compose phase of the MGP process.

```
procedure AfterProcessCompose;
```

2.7.1.6 BeforeProcessLogs

The `BeforeProcessLogs` procedure is called before the database log tables (CLG$) are generated for the compose phase of the MGP process. This log tables capture changes for MGP and should not be confused with the trace logs.

```
procedure BeforeProcessLogs;
```

2.7.1.7 AfterProcessLogs

The `AfterProcessLogs` procedure is called after the database log tables (CLG$) are generated for the compose phase of the MGP process. This log tables capture changes for MGP and should not be confused with the trace logs.

```
procedure AfterProcessLogs;
```

2.7.1.8 BeforeClientCompose

The `BeforeClientCompose` procedure is called before each user is composed during the compose phase of the MGP process.

```
procedure BeforeClientCompose (clientid varchar2);
```

2.7.1.9 AfterClientCompose

The `AfterClientCompose` procedure is called after each user is composed during the compose phase of the MGP process.

```
procedure AfterClientCompose (clientid varchar2);
```

2.7.1.10 BeforeSyncMapCleanup

For every publication item, Oracle Database Mobile Server maintains a map table, where the MGP inserts the DML operations to be carried out on the client database or new records to be inserted in the case of a complete refresh. At the end of the every synchronization session, the map tables are cleaned up where all old entries are deleted.

During this cleanup, if the connection properties are not ideal, then you may have performance issues. The callbacks added before and after the map cleanup operation enable you to optimize the connection properties and revert back to old connection properties after the operation is complete.

The `BeforeSyncMapCleanup` procedure is called at the beginning of the cleanup; the `AfterSyncMapCleanup` procedure is called after cleanup is finished. You can configure the connection settings can be changed in the `BeforeSyncMapCleanup` and reverted back in the `AfterSyncMapCleanup` procedure. These methods are invoked only once during the synchronization cycle.

The properties you can manage in these callback procedures are as follows:

- Any session level hints
- You can set the `OPTIMIZER_INDEX_CACHING` and `OPTIMIZER_INDEX_COST_ADJ` session parameters, as follows:
  - `ALTER SESSION SET OPTIMIZER_INDEX_CACHING=0;`
  - `ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ=100;`

Note: In the CONSOLIDATOR section of the mobile.ora file, you may want to modify the MAX_U_COUNT parameter before the synchronization starts.

The MAX_U_COUNT parameter controls the number of SQL statements that are executed together in a SQL batch statement while performing the map cleanup. The default value for the MAX_U_COUNT parameter is 256. However, if the value is 256 during the map cleanup, then a maximum of 256 SQL statements can be executed together in a batch. Modify this parameter and restart the mobile server to enable a larger batch of SQL statements to be processed during map cleanup.

2.7.1.11 AfterSyncMapCleanup

The AfterSyncMapCleanup procedure is called at the end of the map cleanup. If you set any parameters in the BeforeSyncMapCleanup callback, you can set them back to the original settings in this procedure. See Section 2.7.1.10, "BeforeSyncMapCleanup" for more information.

2.7.1.12 Example Using the Customize Package

If a developer wants to use any of the procedures listed above, perform the following:

- Manually create the CUSTOMIZE package in the mobile server schema.
- Define all of the methods with the following specification:

```
create or replace package CUSTOMIZE as
    procedure NullSync (clientid varchar2, isNullSync boolean);
    procedure BeforeProcessApply ;
    procedure AfterProcessApply ;
    procedure BeforeProcessCompose ;
    procedure AfterProcessCompose ;
    procedure BeforeProcessLogs ;
    procedure AfterProcessLogs ;
    procedure BeforeClientCompose(clientid varchar2);
    procedure AfterClientCompose(clientid varchar2);
    end CUSTOMIZE;
```

WARNING: It is the developer's responsibility to ensure that the package is defined properly and that the logic contained does not jeopardize the integrity of the synchronization process.

2.7.1.13 Error Handling For CUSTOMIZE Package

Errors are logged for the CUSTOMIZE package only if logging is enabled for the MGP component for the finest level for all event types. Thus, you should set the logging level to ALL and the type to ALL.

If any errors occur due to an invalid CUSTOMIZE package, they are logged only on the first MGP cycle after the mobile server restarts. On subsequent synchronizations, the errors are not re-written to the logs, sine the MGP does not attempt to re-execute the CUSTOMIZE package until the mobile server is restarted.

Note: One requirement is that the CUSTOMIZE package can only be executed as user mobileadmin.

To locate these errors easily within the `MGP_<x>.log` files, search for the `MGP.callBoundCallBack` method. Another option is to restart the mobile server and check the MGP log right after the next synchronization.

2.7.2 Customize What Occurs Before and After Compose/Apply Phases for a Single Publication Item

When creating publication items, the user can define a customizable PL/SQL package that MGP calls during the Apply and Compose phase of the MGP background process **for that particular publication item**. To customize the compose/apply phases for a publication item, perform the following:

1. Create the PL/SQL package with the customized before/after procedures.
2. Register this PL/SQL package with the publication item.

> **Note:** If you are using a remote database for application data, then the callbacks must be defined on the same database as the application.

Then when the publication item is being processed, MGP calls the appropriate procedures from your package.

Client data is accumulated in the In Queue before being processed by the MGP. Once processed by the MGP, data is accumulated in the Out Queue before being pulled to the client by Mobile Sync.

You can implement the following PL/SQL procedures to incorporate customized code into the MGP process. The `clientname` and `tranid` are passed to allow for customization at the user and transaction level.

- The `BeforeApply` method is invoked before the client data is applied:

  ```
  procedure BeforeApply(clientname varchar2)
  ```

- The `AfterApply` method is invoked after all client data is applied.

  ```
  procedure AfterApply(clientname varchar2)
  ```

- The `BeforeTranApply` method is invoked before the client data with `tranid` is applied.

  ```
  procedure BeforeTranApply(tranid number)
  ```

- The `AfterTranApply` method is invoked after all client data with `tranid` is applied.

  ```
  procedure AfterTranApply(tranid number)
  ```

- The `BeforeCompose` method is invoked before the Out Queue is composed.

  ```
  procedure BeforeCompose(clientname varchar2)
  ```

- The `AfterCompose` method is invoked after the Out Queue is composed.

  ```
  procedure AfterCompose(clientname varchar2)
  ```

The following is a PL/SQL example that creates a callback package and registers it when creating the `P_SAMPLE3` publication item. The `BeforeApply` procedure disables constraints before the apply phase; the `AfterApply` procedure enables these constraints. Even though you are only creating procedures for the before and after

apply phase of the MGP process, you still have to provide empty procedures for the other parts of the MGP process.

1. Create PL/SQL package declaration with callback owner/schema name of `SAMPLE3` and callback package name of `SAMP3_PKG`.
2. Create the package definition, with all MGP process procedures with callback owner.callback package name of `SAMPLE3.SAMP3_PKG`. Provide a `NULL` procedure for any procedure you do not want to modify.
3. Register the package as the callback package for the `SAMPLE3` publication item. If you are creating the publication item, provide the callback schema/owner and the callback package names as input parameters to the `createPublicationItem` method. If you want to add the callback package to an existing publication item, do the following:
   a. Retrieve the template metadata with `getTemplateItemMetaData` for the publication item.
   b. Modify the attributes that specify the callback owner/schema (`cbk_owner`) and the callback package (`cbk_name`).
   c. Register the package by executing the `setTemplateItemMetaData` method.

```
// create package declaration
  stmt.executeUpdate("CREATE OR REPLACE PACKAGE SAMPLE3.SAMP3_PKG as"
  + " procedure BeforeCompose(clientname varchar2);"
  + " procedure AfterCompose(clientname varchar2);"
  + " procedure BeforeApply(clientname varchar2);"
  + " procedure AfterApply(clientname varchar2);"
  + " procedure BeforeTranApply(tranid number);"
  + " procedure AfterTranApply(tranid number);"
  + " end;"
  );
// create package definition
  stmt.executeUpdate("CREATE OR REPLACE PACKAGE body SAMPLE3.SAMP3_PKG as"
  + " procedure BeforeTranApply(tranid number) is"
  + " begin"
    + " NULL;"
  + " end;"
  + " procedure AfterTranApply(tranid number) is"
  + " begin"
    + " NULL;"
  + " end;"
  + " procedure BeforeCompose(clientname varchar2) is"
  + " begin"
       + "   NULL;"
  + " end;"
  + " procedure AfterCompose(clientname varchar2) is"
  + " begin"
  + "   NULL;"
  + " end;"
  + " procedure BeforeApply(clientname varchar2) is"
  + "   cur integer;"
  + "   ign integer;"
  + "   begin"
  + "     cur := dbms_sql.open_cursor;"
  + "     dbms_sql.parse(cur,'SET CONSTRAINT SAMPLE3.address14_fk DEFERRED',
                          dbms_sql.native);"
  + "     ign := dbms_sql.execute(cur);"
  + "     dbms_sql.close_cursor(cur);"
  + "   end;"
```

```
  + " procedure AfterApply(clientname varchar2) is"
  + "   cur integer;"
  + "   ign integer;"
  + "   begin"
  + "     cur := dbms_sql.open_cursor;"
  + "     dbms_sql.parse(cur, 'SET CONSTRAINT SAMPLE3.address14_fk IMMEDIATE',
                          dbms_sql.native);"
  + "     ign := dbms_sql.execute(cur);"
  + "     dbms_sql.close_cursor(cur);"
  + "   end;"
  + " end;"
  );
```

Then, register the callback package with the `createPublicationItem` method call, as follows:

```
// register SAMPLE3.SAMP3_PKG as the callback for MGP processing of
// P_SAMPLE3 publication item.

cm.createPublicationItem("P_SAMPLE3","SAMPLE3","ADDRESS", "F",
    "SELECT * FROM SAMPLE3.ADDRESS", "SAMPLE3", "SAMP3_PKG");
```

In the previous code example, the following is required:

- `stmt`, which is used when creating the package definition, is an instance of `java.sql.Statement`
- `cm`, which is used when registering the callback package, is an instance of `oracle.lite.sync.ConsolidatorManager`
- The callback package must have the following procedures defined:
  - `BeforeCompose (clientname varchar2);`
  - `AfterCompose (clientname varchar2);`
  - `BeforeApply (clientname varchar2);`
  - `AfterApply (clientname varchar2);`
  - `BeforeTranApply (tranid number);`
  - `AfterTranApply (tranid number);`

2.8 Understanding Your Refresh Options

The mobile server supports several refresh options. During a fast refresh, incremental changes are synchronized. However, during a complete refresh, all data is refreshed with current data. The refresh mode is established when you create the publication item using the `createPublicationItem` API call. In order to change the refresh mode, first drop the publication item and recreate it with the appropriate mode.

The following sections describe the types of refresh for your publication item that can be used to define how to synchronize:

- Fast Refresh: The most common method of synchronization is a fast refresh publication item where changes are uploaded by the client, and changes for the client are downloaded. Meanwhile, the MGP periodically collects the changes uploaded by all clients and applies them to database tables. It then composes new data, ready to be downloaded to each client during the next synchronization, based on predefined subscriptions.

- Complete Refresh: During a complete refresh, all data for a publication is downloaded to the client. For example, during the very first synchronization session, all data on the client is refreshed from the client database. This form of synchronization takes longer because all rows that qualify for a subscription are transferred to the client device, regardless of existing client data.
- Queue-Based: The developer creates their own queues to handle the synchronization data transfer. This can be considered the most basic form of publication item, for the simple reason that there is no synchronization logic created with it. The synchronization logic is left entirely in the hands of the developer. A queue-based publication item is ideally suited for scenarios that do not require actual synchronization, but require something somewhere in between. For instance, data collection on the client. With data collection, there is no need to worry about conflict detection, client state information, or server-side updates. Therefore, there is no need to add the additional overhead normally associated with a fast refresh or complete refresh publication item.
- Forced Refresh: This is actually NOT a refresh option; however, we discuss it here in order to inform you of the consequences of performing a forced refresh. When a Forced Refresh is initiated all data on the client is removed. The client then brings down an accurate copy of the client data from the enterprise database to start fresh with exactly what is currently stored in the enterprise data store.

The following sections describe the refresh options in more detail:

- Section 2.8.1, "Fast Refresh"
- Section 2.8.2, "Complete Refresh for Views"
- Section 2.8.3, "Queue-Based Refresh"
- Section 2.8.4, "Forced Refresh"

2.8.1 Fast Refresh

Publication items are created for fast refresh by default. Under fast refresh, only incremental changes are replicated. The advantages of fast refresh are reduced overhead and increased speed when replicating data stores with large amounts of data where there are limited changes between synchronization sessions.

The mobile server performs a fast refresh of a view if the view meets the following criteria:

- Each of the view base tables must have a primary key.
- All primary keys from all base tables must be included in the view column list.
- If the item is a view, and the item predicate involves multiple tables, then all tables contained in the predicate definition must have primary keys and must have corresponding publication items.

The view requires only a unique primary key for the parent table. The primary keys of other tables may be duplicated. For each base table primary key column, you must provide the mobile server with a hint about the column name in the view. You can accomplish this by using the `primaryKeyHint` method of the Consolidator Manager object. See the *Oracle Database Mobile Server JavaDoc* for more information.

2.8.2 Complete Refresh for Views

A complete refresh is simply a complete execution of the snapshot query. When application synchronization performance is slow, tune the snapshot query. Complete

refresh is not optimized for performance. Therefore, to improve performance, use the fast refresh option. The Consperf utility analyzes only fast refresh publication items.

Publication items can be created for complete refresh using the `C` refresh mode in the `createPublicationItem` API from the Consolidator Manager API. When this mode is specified, client data is completely refreshed with current data from the server after every sync. An administrator can force a complete refresh for an entire publication through an API call. This function forces complete refresh of a publication for a given client.

See the *Oracle Database Mobile Server JavaDoc* for more information.

The following lists what can cause a complete refresh, ordered from most likely to least likely:

1. The same mobile user synching from multiple devices on the same platform, or synching from different platforms when the publications are not platform specific.
2. Republishing the application.
3. An unexpected server apply condition, such as constraint violations, unresolved conflicts, and other database exceptions.
4. Modifying the application, such as changing subsetting parameters or adding/altering publication items. This refresh only affects the publication items.
5. A force refresh requested by server administrator or a force refresh requested by the client.
6. On mobile clients, restoring an old client database.
7. Two separate applications using the same back-end store.
8. An unexpected client apply conditions, such as a moved or deleted database, database corruption, memory corruption, other general system failures.
9. Loss of transaction integrity between the server and client. The server fails post processing after completing the download and disconnects from the client.
10. Data transport corruptions.

2.8.3 Queue-Based Refresh

You can create your own queues. The mobile server uploads and downloads changes from the user. Perform customized apply/compose modifications to the back-end database with your own implementation. See the Section 2.12, "Customizing Synchronization With Your Own Queues" for more information.

2.8.4 Forced Refresh

This is actually NOT a refresh option; however, we discuss it here in order to inform you of the consequences of performing a forced refresh. Out of all the different synchronization options, the Forced Refresh synchronization architecture is probably the most misunderstood synchronization type. This option is commonly confused with the Complete Refresh synchronization. This confusion may result in tragic consequences and the loss of critical data on the client.

The Forced Refresh option is an emergency only synchronization option. This option is for when a client is so corrupt or malfunctioning so severely that the determination is made to replace the mobile client data with a fresh copy of data from the enterprise data store. When this option is selected, any data transactions that have been made on the client are lost.

When a Forced Refresh is initiated all data on the client is removed. The client then brings down an accurate copy of the client data from the enterprise database to start fresh with exactly what is currently stored in the enterprise data store.

2.9 Synchronizing With Database Constraints

When you have database constraints on your table, you must develop your application in a certain way to facilitate the synchronization of the data and keeping the database constraints.

The following sections detail each constraint and what issues you must take into account:

- Section 2.9.1, "Synchronization And Database Constraints"
- Section 2.9.2, "Primary Key is Unique"
- Section 2.9.3, "Foreign Key Constraints"
- Section 2.9.4, "Unique Key Constraint"
- Section 2.9.5, "NOT NULL Constraint"
- Section 2.9.6, "Generating Constraints on the Mobile Client"

2.9.1 Synchronization And Database Constraints

Oracle Database Mobile Server does not keep a record of the SQL operations executed against the database; instead, only the final changes are saved and synchronized to the back-end database.

For example, if you have a client with a unique key constraint, where the following is executed against the client database:

1. Record with primary key of one and unique field of ABC is deleted.
2. Record with primary key of 4 and unique field of ABC is inserted.

When this is synchronized, according the Section 2.4.1.7.2, "Using Table Weight" discussion, the insert is performed before the delete. This would add a duplicate field for ABC and cause a unique key constraint violation. In order to avoid this, you should defer all constraint checking until after all transactions are applied. See Section 2.9.3.2, "Defer Constraint Checking Until After All Transactions Are Applied".

Another example of how synchronization captures the end result of all SQL modifications is as follows:

1. Insert an employee record 4 with name of Joe Judson.
2. Update employee record 4 with address.
3. Update employee record 4 with salary.
4. Update employee record 4 with office number
5. Update employee record 4 with work email address.

When synchronization occurs, all modifications are captured and only a single insert is performed on the back-end database. The insert contains the primary key, name, address, salary, office number and email address. Even though the data was created with multiple updates, the Sync Server only takes the final result and makes a single insert.

Note: If you want these constraints to apply on the mobile client, see Section 2.9.6, "Generating Constraints on the Mobile Client".

2.9.2 Primary Key is Unique

When you have multiple clients, each updating the same table, you must have a method for guaranteeing that the primary key is unique across all clients. Oracle Database Mobile Server provides you a sequence number that you can use as the primary key, which is guaranteed to be unique across all mobile clients.

For more information on the sequence number, see Section 2.4.1.8, "Creating Client-Side Sequences for the Downloaded Snapshot".

2.9.3 Foreign Key Constraints

A foreign key exists in a details table and points to a row in the master table. Thus, before a client adds a record to the details table, the master table must first exist.

For example, two tables `EMP` and `DEPT` have referential integrity constraints and are an example of a master-detail relationship. The `DEPT` table is the master table; the `EMP` table is the details table. The `DeptNo` field (department number) in the `EMP` table is a foreign key that points to the `DeptNo` field in the `DEPT` table. The `DeptNo` value for each employee in the `EMP` table must be a valid `DeptNo` value in the `DEPT` table.

When a user adds a new employee, first the employee's department must exist in the `DEPT` table. If it does not exist, then the user first adds the department in the `DEPT` table, and then adds a new employee to this department in the `EMP` table. The transaction first updates `DEPT` and then updates the `EMP` table. However, Oracle Database Mobile Server does not store the sequence in which these operations were executed.

Oracle Database Mobile Server does not keep a record of the SQL operations executed against the database; instead, only the final changes are saved and synchronized to the back-end database. For our employee example, when the user replicates with the mobile server, the mobile server could initiate the updates the `EMP` table first. If this occurs, then it attempts to create a new record in `EMP` with an invalid foreign key value for `DeptNo`. Oracle database detects a referential integrity violation. The mobile server rolls back the transaction and places the transaction data in the mobile server error queue. In this case, the foreign key constraint violation occurred because the operations within the transaction are performed out of their original sequence.

In order to avoid this violation, you can do one of two things:

- Section 2.9.3.1, "Set Update Order for Tables With Weights"
- Section 2.9.3.2, "Defer Constraint Checking Until After All Transactions Are Applied"

2.9.3.1 Set Update Order for Tables With Weights

Set the order in which tables are updated on the back-end Oracle database with weights. To avoid integrity constraints with a master-details relationship, the master table must always be updated first in order to guarantee that it exists before any records are added to a details table. In our example, you must set the `DEPT` table with a lower weight than the `EMP` table to ensure that all records are added to the `DEPT` table first.

You define the order weight for tables when you add a publication item to the publication. For more information on weights, see Section 2.4.1.7.2, "Using Table Weight".

2.9.3.2 Defer Constraint Checking Until After All Transactions Are Applied

You can use a PL/SQL procedure avoid foreign key constraint violations based on out-of-sequence operations by using `DEFERRABLE` constraints in conjunction with the `BeforeApply` and `AfterApply` functions. `DEFERRABLE` constraints can be either `INITIALLY IMMEDIATE` or `INITIALLY DEFERRED`. The behavior of `DEFERRABLE INITIALLY IMMEDIATE` foreign key constraints is identical to regular immediate constraints. They can be applied interchangeably to applications without impacting functionality.

The mobile server calls the `BeforeApply` function before it applies client transactions to the server and calls the `AfterApply` function after it applies the transactions. Using the `BeforeApply` function, you can set constraints to `DEFFERED` to delay referential integrity checks. After the transaction is applied, call the `AfterApply` function to set constraints to `IMMEDIATE`. At this point, if a client transaction violates referential integrity, it is rolled back and moved into the error queues.

To prevent foreign key constraint violations using `DEFERRABLE` constraints:

1. Drop all foreign key constraints and then recreate them as `DEFERRABLE` constraints.
2. Bind user-defined PL/SQL procedures to publications that contain tables with referential integrity constraints.
3. The PL/SQL procedure should set constraints to `DEFERRED` in the `BeforeApply` function and `IMMEDIATE` in the `AfterApply` function as in the following example featuring a table named `SAMPLE3` and a constraint named `address.14_fk`:

```
procedure BeforeApply(clientname varchar2) is
cur integer;
begin
  cur := dbms_sql.open_cursor;
  dbms_sql.parse(cur,'SET CONSTRAINT SAMPLE3.address14_fk
                 DEFERRED', dbms_sql.native);
  dbms_sql.close_cursor(cur);
end;
procedure AfterApply(clientname varchar2) is
cur integer;
begin
  cur := dbms_sql.open_cursor;
  dbms_sql.parse(cur, 'SET CONSTRAINT SAMPLE3.address14_fk
                 IMMEDIATE', dbms_sql.native);
  dbms_sql.close_cursor(cur);
end;
```

2.9.4 Unique Key Constraint

A unique key constraint enforces uniqueness of data. However, you may have multiple clients across multiple devices updating the same table. Thus, a record may be unique on a single client, but not across all clients. Enforcing uniqueness is the customer's reponsibility and depends on the data.

How do you guarantee that the records added on separate clients are unique? You can use the sequence numbers generated on the client by Oracle Database Mobile Server.

See Section 2.4.1.8, "Creating Client-Side Sequences for the Downloaded Snapshot" for more information.

2.9.5 NOT NULL Constraint

When you have a NOT NULL constraint on the client or on the server, you must ensure that this constraint is set on both sides.

- On the server—Create a NOT NULL constraint on the back-end server table using the Oracle database commands.
- For the client—Set a column as NOT NULL by executing the `setPubItemColOption` method in the `ConsolidatorManager` API. Provide `Consolidator.NOT_NULL` as the input parameter for `nullType`. The constraint is then enforced on the table in the client database.

2.9.6 Generating Constraints on the Mobile Client

The Primary Key, Foreign Key, Not Null and Default Value constraints can be synchronized to the mobile client; the Unique constraints cannot be synchronized. For foreign key constraints, you decide if you want the foreign key on the mobile client. That is, when you create a foreign key constraint on a table on the back-end server, you may or may not want this constraint to exist on the mobile client.

- Each publication that is defined is specific to a certain usage. For example, if you have a foreign key constraint between two tables, such as department and employee, your publication may only specify that information from the employee table is downloaded. In this situation, you would not want the foreign constraint between the employee and department table to be enforced on the client.
- If you do have a master-detail relationship or other constraint relationships synchronized down to the client, then you would want to have the constraint generated on the client.

In order to generate the constraints on the mobile client, perform the following:

1. Within the process for creating or modifying an existing publication using the APIs, invoke the `assignWeights` method of the `ConsolidatorManager` object, which does the following tasks:
   a. Calculates a weight for each of the publication items included in the publication.
   b. Creates a script that, when invoked on the client, generates the constraints on the client. This script is automatically added to the publication.
2. On the mobile client, perform a synchronization for the user, which brings down the snapshot and the constraint script. The script is automatically executed on the mobile client.

Once executed on the client, all constraints on the server for this publication are also enforced on the mobile client.

2.9.6.1 The assignWeights Method

The `assignWeights` method automatically calculates weights for all publication items belonging to a publication. If a new publication item is added or if there is a change in the referential relationships, the API should be called again.

The following defines the `assignWeights` method and its parameters:

```
public void assignWeights(java.lang.String pub, boolean createScripts)
```

```
throws ConsolidatorException
```

Where:

- `pub` - Publication name
- `createScripts` - If true, creates referential constraints scripts and adds them to the publication to be propagated to subscribed clients.

2.10 Resolving Conflicts with Winning Rules

When you have a conflict, you need to determine which party wins. The following are the settings that you can choose for conflict resolution on the server:

- Client wins—When the client wins, the mobile server automatically applies client changes to the server. And if you have a record that is set for `INSERT`, yet a record already exists, the mobile server automatically modifies it to be an `UPDATE`.
- Server wins—If the server wins, the client updates are not applied to the application tables. Instead, the mobile server automatically composes changes for the client. The client updates are placed into the error queue, just in case you still want these changes to be applied to the server—even though the winning rules state that the server wins.

The mobile server uses internal versioning to detect synchronization conflicts. A separate version number is maintained for each client and server record. When the client updates are applied to the server, then the mobile server checks the version numbers of the client against the version numbers on the server. If the version does not match, then the conflict resolves according to the defined winning rules—such as client wins or server wins, as follows:

The mobile server does not automatically resolve synchronization errors. Instead, the mobile server rolls back the corresponding transactions, and moves the transaction operations into the mobile server error queue. It is up to the administrator to view the error queue and determine if the correct action occurred. If not, the administrator must correct and re-execute the transaction. If it did execute correctly, then purge the transaction from the error queue.

One type of error is a synchronization conflict, which is detected in any of the following situations:

- The client and the server update the same row.
- The client deletes the same row that the server updates.
- The client updates a row at the same time that the server deletes it when the "server wins" conflict rule is specified. This is considered a synchronization error for compatibility with Oracle database advanced replication.
- Both the client and server create rows with the same primary key values.
- Two separate clients update the same row.
- Two clients insert a row with the same primary key.
- One client deletes a row that a second client updates.

Note: In the case where two clients conflict, then the client whose data gets applied first effectively becomes the server and the other client becomes the client in resolving this conflict.

- For systems with delayed data processing, where the client data is not directly applied to the base table—for instance, in a three-tiered architecture—a situation could occur when a client inserts a row and then updates the same row, while the row has not yet been inserted into the base table. In that case, if the `DEF_APPLY` parameter in `C$ALL_CONFIG` is set to `TRUE`, an `INSERT` operation is performed, instead of the `UPDATE`. It is up to the application developer to resolve the resulting primary key conflict. If, however, `DEF_APPLY` is not set, a "`NO DATA FOUND`" exception is thrown.

All the other errors, including nullity violations and foreign key constraint violations are synchronization errors. See Section 2.9, "Synchronizing With Database Constraints" for more information.

On the server, synchronization errors and conflicts are placed into the error queue. For each publication item created, a separate and corresponding error queue is created. The purpose of this queue is to store transactions that fail due to unresolved conflicts. The administrator can attempt to resolve the conflicts, either by modifying the error queue data or that of the server, and then attempt to re-apply the transaction.

The administrator can resolve the errors, and then re-execute or purge transactions from the error queue using either of the following:

- Resolve errors and conflicts in the error queue using the Mobile Manager GUI—See Section 5.12.4.3, "Viewing Server-Side Synchronization Conflicts and Errors in the Error Queue" in the *Oracle Database Mobile Server Administration and Deployment Guide* on how to update the client transaction in the error queue and re-execute the statement using the Mobile Manager GUI.
- Resolve errors and conflicts programmatically with the Consolidator Manager API. You can access the mobile server error queue tables directly and customize the conflict rules, as described in the following sections:
  - Section 2.10.1, "Resolving Errors and Conflicts on the Mobile Server Using the Error Queue"
  - Section 2.10.2, "Customizing Synchronization Conflict Resolution Outcomes"

2.10.1 Resolving Errors and Conflicts on the Mobile Server Using the Error Queue

The error queue stores transactions that fail due to synchronization errors or unresolved conflicts. For unresolved conflicts, only the "Server Wins" conflicts are reported. If you have set your conflict rules to "Client Wins", then these are not reported. The administrator can do one of the following:

- Attempt to correct the error by modifying the error queue data or that of the server, and re-apply the transaction through the `executeTransaction` method of the Consolidator Manager object.
- If a conflict was reported and resolved to your satisfaction, then you can purge the transaction from the error queue with the `purgeTransaction` method of the Consolidator Manager object. Otherwise, you can override the default conflict resolution by modifying the error queue data and re-apply the transaction.

View the error queue through the Mobile Manager GUI, where you can see what the conflict was. You can fix the problem and reapply the data by modifying the DML operation appropriately and then re-executing. See Section 5.12.4.3, "Viewing Server-Side Synchronization Conflicts and Errors in the Error Queue"" in the *Oracle Database Mobile Server Administration and Deployment Guide* for directions.

2.10.2 Customizing Synchronization Conflict Resolution Outcomes

You can customize synchronization conflict resolution by doing the following:

1. Configure the winning rule to Client Wins.
2. Perform only ONE of the following:
   - Create and attach one or more triggers on the back-end Oracle database base tables to execute before the `INSERT`, `UPDATE`, or `DELETE` DML statements. The triggers should be created to evaluate the data and handle the conflict. Triggers are created to compare old and new row values and resolve client changes as defined by you. See the *Oracle Database* documentation for full details on how to create and attach triggers.
   - Create a custom DML procedure. See Section 2.4.1.13, "Callback Customization for DML Operations" for an example of how to create a custom DML procedure.

     You can use the `generateMobileDMLProcedure` to generate the procedure specification for a given publication item. This specification can be used as a starting point in creating your own custom DML handling logic in a PL/SQL procedure. You use the `addMobileDMLProcedure` API to attach the PL/SQL procedure to the publication item. See the *Oracle Database Mobile Server API Specification* for more information.

2.11 Using the Sync Discovery API to Retrieve Statistics

The Sync Discovery feature is used to request an estimate of the size of the download for a specific client, based on historical data. The following statistics are gathered to maintain the historical data:

- The total number of rows send for each publication item.
- The total data size for these rows.
- The compressed data size for these rows.

The following sections contain methods that can be used to gather statistics:

- Section 2.11.1, "getDownloadInfo Method"
- Section 2.11.2, "DownloadInfo Class Access Methods"
- Section 2.11.3, "PublicationSize Class"

2.11.1 getDownloadInfo Method

The `getDownloadInfo` method returns the `DownloadInfo` object. The `DownloadInfo` object contains a set of `PublicationSize` objects and access methods. The `PublicationSize` objects carry the size information of a publication item. The method `Iterator iterator()` can then be used to view each `PublicationSize` object in the `DownloadInfo` object.

```
DownloadInfo dl = consolidatorManager.getDownloadInfo("S11U1", true, true);
```

Note: See the *Oracle Database Mobile Server JavaDoc* for more information.

2.11.2 DownloadInfo Class Access Methods

The access methods provided by the `DownloadInfo` class are listed in Table 2–21:

Table 2–21 DownloadInfo Class Access Methods

Method	Definition
`iterator`	Returns an Iterator object so that the user can traverse through the all the `PublicationSize` objects that are contained inside the `DownloadInfo` object.
`getTotalSize`	Returns the size information of all `PublicationSize` objects in bytes, and by extension, the size of all publication items subscribed to by that user. If no historical information is available for those publication items, the value returned is '-1'.
`getPubSize`	Returns the size of all publication items that belong to the publication referred to by the string `pubName`. If no historical information is available for those publication items, the value returned is '-1'.
`getPubRecCount`	Returns the number of all records of all the publication items that belong to the publication referred by the string `pubName`, that be synchronization during the next synchronization.
`getPubItemSize`	Returns the size of a particular publication item referred by `pubItemName`. It follows the following rules in order. 1. If the publication item is empty, it return '0'. 2. If no historical information is available for those publication items, it return '-1'.
`getPubItemRecCount`	Returns the number of records of the publication item referred by `pubItemName` that be synced in the next synchronization.

Note: See the *Oracle Database Mobile Server JavaDoc* for more information.

2.11.3 PublicationSize Class

The access methods provided by the `PublicationSize` class are listed in Table 2–22:

Table 2–22 PublicationSize Class Access Methods

Parameter	Definition
`getPubName`	Returns the name of the publication containing the publication item.
`getPubItemName`	Returns the name of the publication item referred to by the `PublicationSize` object.
`getSize`	Returns the total size of the publication item referred to by the `PublicationSize` object.
`getNumOfRows`	Returns the number of rows of the publication item that is synchronized in the next synchronization.

Note: See the *Oracle Database Mobile Server JavaDoc* for more information.

Sample Code

```
import   java.sql.*;
import   java.util.Iterator;
import   java.util.HashSet;

import   oracle.lite.sync.ConsolidatorManager;
import   oracle.lite.sync.DownloadInfo;
import   oracle.lite.sync.PublicationSize;

public class TestGetDownloadInfo
{

   public static void main(String argv[]) throws Throwable
   {
// Open Consolidator Manager connection
      try
      {
// Create a ConsolidatorManager object
         ConsolidatorManager cm = new ConsolidatorManager ();
// Open a Consolidator Manager connection
         cm.openConnection ("MOBILEADMIN", "MANAGER",
                     "jdbc:oracle:thin:@server:1521:orcl", System.out);
// Call getDownloadInfo
         DownloadInfo dlInfo = cm.getDownloadInfo ("S11U1", true, true);
// Call iterator for the Iterator object and then we can use that to transverse
// through the set of PublicationSize objects.
         Iterator it = dlInfo.iterator ();
// A temporary holder for the PublicationSize object.
         PublicationSize ps = NULL;
// A temporary holder for the name of all the Publications in a HashSet object.
         HashSet pubNames = new HashSet ();
// A temporary holder for the name of all the Publication Items in a HashSet
// object.
         HashSet pubItemNames = new HashSet ();
// Traverse through the set.
         while (it.hasNext ())
         {
// Obtain the next PublicationSize object by calling next ().
            ps = (PublicationSize)it.next ();

// Obtain the name of the Publication this PublicationSize object is associated
// with by calling getPubName ().
            pubName = ps.getPubName ();
            System.out.println ("Publication: " + pubName);

// We save pubName for later use.
            pubNames.add (pubName);

// Obtain the Publication name of it by calling getPubName ().
            pubItemName = ps.getPubItemName ();
            System.out.println ("Publication Item Name: " + pubItemName);

// We save pubItemName for later use.
            pubItemNames.add (pubItemName);

// Obtain the size of it by calling getSize ().
            size = ps.getSize ();
            System.out.println ("Size of the Publication: " + size);

// Obtain the number of rows by calling getNumOfRows ().
```

```
            numOfRows = ps.getNumOfRows ();
            System.out.println ("Number of rows in the Publication: "
                                + numOfRows);
         }

// Obtain the size of all the Publications contained in the
// DownloadInfo objects.
         long totalSize = dlInfo.getTotalSize ();
         System.out.println ("Total size of all Publications: " + totalSize);

// A temporary holder for the Publication size.
         long pubSize = 0;

// A temporary holder for the Publication number of rows.
         long pubRecCount = 0;

// A temporary holder for the name of the Publication.
         String tmpPubName = NULL;

// Transverse through the Publication names that we saved earlier.
         it = pubNames.iterator ();
         while (it.hasNext ())
         {
// Obtain the saved name.
            tmpPubName = (String) it.next ();

// Obtain the size of the Publication.
            pubSize = dlInfo.getPubSize (tmpPubName);
            System.out.println ("Size of " + tmpPubName + ": " + pubSize);

// Obtain the number of rows of the Publication.
            pubRecCount = dlInfo.getPubRecCount (tmpPubName);
            System.out.println ("Number of rows in " + tmpPubName + ": "
                                + pubRecCount);
         }

// A temporary holder for the Publication Item size.
         long pubItemSize = 0;

// A temporary holder for the Publication Item number of rows.
         long pubItemRecCount = 0;

// A temporary holder for the name of the Publication Item.
         String tmpPubItemName = NULL;

// Traverse through the Publication Item names that we saved earlier.
         it = pubItemNames.iterator ();
         while (it.hasNext ())
         {
// Obtain the saved name.
            tmpPubItemName = (String) it.next ();

// Obtain the size of the Publication Item.
            pubItemSize = dlInfo.getPubItemSize (tmpPubItemName);
            System.out.println ("Size of " + pubItemSize + ": " + pubItemSize);

// Obtain the number of rows of the Publication Item.
            pubItemRecCount = dlInfo.getPubItemRecCount (tmpPubItemName);
            System.out.println ("Number of rows in " + tmpPubItemName + ": "
                               + pubItemRecCount);
```

```
            }
            System.out.println ();

// Close the connection
            cm.closeConnection ();
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
}
```

2.12 Customizing Synchronization With Your Own Queues

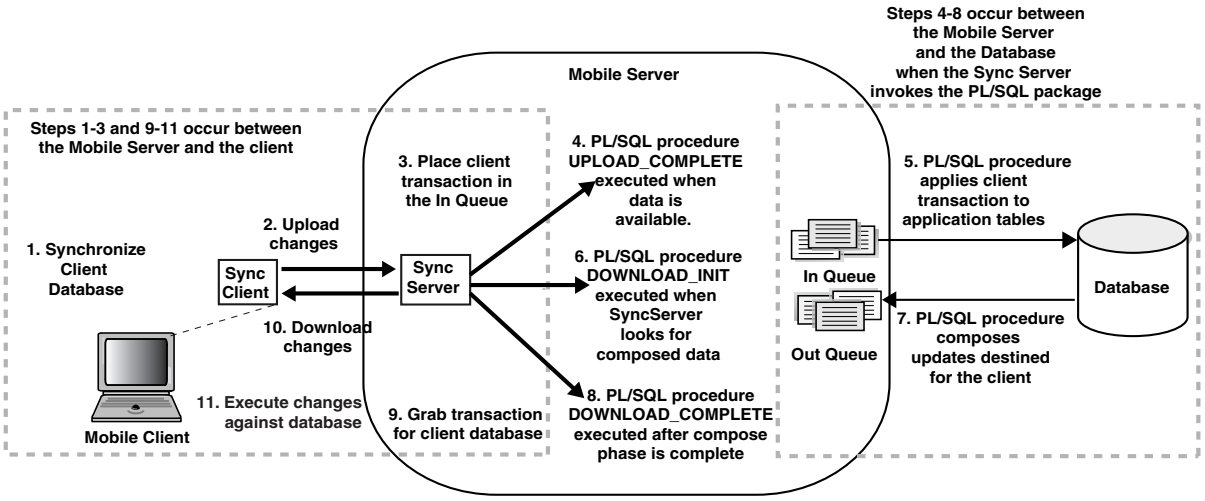
Application developers can manage the synchronization process programmatically by using queue-based publication items. By default on the server-side, the MGP manages both the In Queues and the Out Queues by gathering all updates posted to the In Queue, applying these updates to the relevant tables, and then composing all new updates created on the server that are destined for the client and posting it to the Out Queue. This is described in Section 2.1, "How Oracle Database Mobile Server Synchronizes".

However, you can bypass the MGP and provide your own solution for the apply and compose phases on the server-side for selected publication items. You may wish to bypass the MGP for the publication item if one or more of the following are true:

- If you want to facilitate synchronous data exchange, use queue-based publication items.
- If you have complex business rules for data subsetting, in how you decide what data each user receives, then use queue-based publication items. You can incorporate these business rules into generation of the client's queue data. This is especially true if the rules are dynamically evaluated during runtime.
- If your client collects large amounts of data only for upload to the server, never receives data from the server, and it does not require conflict resolution, then use the data collection queues.

Figure 2–4 shows how the Sync Server invokes the `UPLOAD_COMPLETE` PL/SQL procedure when the client upload is complete. And before it downloads all composed updates to the client, the Sync Server invokes the `DOWNLOAD_INIT` PL/SQL procedure.

Figure 2–4 Queue-Based Synchronization Architecture



To bypass the MGP, do the following:

1. Define your publication item as queue-based or data collection. Then, the MGP is not aware of the queues associated with this publication item. You can do this when creating the publication item either through MDW or Consolidator APIs.
2. If queue-based, then create a package, either PL/SQL or Java, that implements the queue interface callback methods. This includes the following callback methods:
   - `UPLOAD_COMPLETE` to process the incoming updates from the client.
   - `DOWNLOAD_INIT` to complete the compose phase.
   - `DOWNLOAD_COMPLETE` if you have any processing to perform after the compose phase.
3. Create the queues. The In Queue, `CFM$<publication_item_name>` is created by default for you. Create the Out Queue as `CTM$<publication_item_name>`.

The following sections describe the methods for customizing the server-side apply/compose phases-++:

- Section 2.12.1, "Customizing Apply/Compose Phase of Synchronization with a Queue-Based Publication Item"—You can define both the apply and compose phases using queue-based publication items.
- Section 2.12.2, "Creating Data Collection Queues for Uploading Client Collected Data"—You use the data collection queue for uploading data from the client. The queues are optimized for when a client collects data to upload to the server and never receives data from the server.
- Section 2.12.3, "Selecting How and When to Notify Clients of Composed Data"—You can notify a client that there is new data on the server ready to be downloaded to initiate a synchronization.

2.12.1 Customizing Apply/Compose Phase of Synchronization with a Queue-Based Publication Item

Note: The sample for queue-based publication items is located in `<ORACLE_HOME>/Mobile/Sdk/samples/Sync/QBasedPI`.

When you want to substitute your own logic for the apply/compose phase of the synchronization process, use a queue-based publication item. The following briefly gives an overview of how the process works internally when using a queue-based publication item:

- When data arrives from the client it is placed in the publication item In Queues. The Sync Server calls `UPLOAD_COMPLETE`, after which the data is committed. All records in the current synchronization session are given the same transaction identifier. The Queue Control Table (`C$INQ`) indicates which publication item In Queues have received new transactions with the unique transaction identifier. Thus, this table shows which queues need processing.
- If you have a queue-based publication item, you must implement the compose phase, if you have one. The MGP is unaware of queue-based publication items and so is not able to perform any action for this publication item. When you implement your own compose logic, you decide when and how the compose logic is invoked. For example, you could do the following:
  - You could have a script execute your compose logic at a certain time of the day.
  - You could schedule the compose procedure as a job in the Job Scheduler.
  - You could include the compose logic as part of the `DOWNLOAD_INIT` function, so that it executes before the client downloads.

    Note: If you decide to implement the compose phase independent of the `DOWNLOAD_INIT` function; then once the compose is finished, you may want the client to receive the data as soon as possible. In this case, invoke the `EN_QUEUE_NOTIFICATION` function to start an automatic synchronization from the client. For more information on this function, see Section 2.12.3, "Selecting How and When to Notify Clients of Composed Data".

    Before the Sync Server begins the download phase of the synchronization session, it calls `DOWNLOAD_INIT`. In this procedure, you can customize the compose or develop any pre-download logic for the client. The Sync Server finds a list of the publication items, which can be downloaded based on the client's subscription. A list of publication items and their refresh mode, ('Y' for complete refresh, 'N' for fast refresh) is inserted into a temporary table (`C$PUB_LIST_Q`). Items can be deleted or the refresh status can be modified in this table since the Sync Server refers to `C$PUB_LIST_Q` to determine the items that are downloaded to the client.

Similar to the In Queue, every record in the Out Queue should be associated with it a transaction identifier (`TRANID$$`). The Sync Server passes the `last_tran` parameter to indicate the last transaction that the client has successfully applied. New Out Queue records that have not been downloaded to the client are be marked with the value of `curr_tran` parameter. The value of `curr_tran` is always greater than that of `last_tran`, though not sequential. The Sync Server downloads records from the Out Queues when the value of `TRANID$$` is greater than `last_tran`. When the data is downloaded, the Sync Server calls `DOWNLOAD_COMPLETE`.

When you decide to use queue-based publication items, you need to do the following:

1. Create both the In and Out Queues used in the apply and compose phases.
   - You can use the default In Queue, which is named `CFM$<publication_item_name>`. Alternatively, you can create the queue of this name manually.

For example, if you wanted the In Queue to be a view, then you would create the In Queue manually.

- Create the Out Queue for the compose phase as `CTM$<publication_item_name>`.

2. Create the publication item and define it as a queue-based publication item. This can be done either through MDW or the Consolidator APIs.
3. Create the PL/SQL or Java callback methods for performing the apply and compose phases. Since the MGP has nothing to do with the queues used for these phases, when you are finished processing the data, you must manage the queues by deleting any rows that have completed the necessary processing.
4. Register the package to be used for all of the queue processing for a particular publication item.

> **Note:** Normally, you define the package on the Main database where the repository is located. However, if you are using a remote database for your application data, then the package must be defined on the remote database.

2.12.1.1 Queue Creation

If a queue-based publication item is created, it always uses a queue by the name of `CFM$<publication_item_name>`. However, if you want to customize how the In Queue is defined—for example, by defining certain rules, making it a view or designating the location of the queue—then you can create your own In Queue. The Out Queue is never defined for you, so you must create an Out Queue named `CTM$<publication_item_name>` in the mobile server repository manually using SQL.

These queues are created based upon the publication item tables. For example, the following table `ACTIVESTATEMENT` has five columns, as follows:

```
create table ACTIVESTATEMENT(
        StatementName varchar2(50) primary key,
        TestSuiteName varchar2(50),
        TestCaseName varchar2(50),
        CurrLine varchar2(4000),
        ASOrder integer) nologging;
```

The application stores its data in these five columns. When synchronization occurs, this data must be uploaded and downloaded. However, there is also meta-information necessary for facilitating the synchronization phases. Therefore, the Out Queue that you create contains the meta-information in the `CLID$$CS`, `TRANID$$` and `DMLTYPE$$` columns, as well as the columns from the `ACTIVESTATEMENT` table, as follows:

```
create table CTM$AUTOTS_PUBITEM(
CLID$$CS VARCHAR2 (30),
StatementName varchar2(50) primary key,
TestSuiteName varchar2(50),
TestCaseName varchar2(50),
CurrLine varchar2(4000),
ASOrder integer,
TRANID$$ NUMBER (10),
DMLTYPE$$ CHAR (1) CHECK (DMLTYPE$$ IN ('I','U','D'))) nologging;
```

Thus, before you can create the queues, you must already know the structure of the tables for the publication item, as well as the publication item name.

The following shows the structure and creation of the queues:

- In Queue
- Out Queue
- Queue Control Table
- Temporary Table

In Queue

All In Queues are named `CFM$<name>` where name is the publication item name. It contains the application publication item table columns, as well as the fields listed in Table 2–23:

Table 2–23 In Queue Interface Creation Parameters

Parameter	Description
`CLID$$CS`	A unique string identifying the client.
`TRANID$$`	A unique number identifying the transaction.
`SEQNO$$`	A unique number for every DML language operation per transaction in the inqueue (CFM$) only.
`DMLTYPE$$`	Checks the type of DML instruction: ■ 'I' - Insert ■ 'D' - Delete ■ 'U' - Update

The following designates the structure when creating the In Queue:

```
create table 'CFM$'+name
(
CLID$$CS   VARCHAR2 (30),
TRANID$$   NUMBER (10),
SEQNO$$    NUMBER (10),
DMLTYPE$$  CHAR (1) CHECK (DMLTYPE$$  IN ('I','U','D'),
publication item column definitions
)
```

> **Note:** You must have the parameters in the same order as shown above for the In Queue. It is different than the ordering in the Out Queue.

Out Queue

All Out Queues are named `CTM$<name>` where name is the publication item name. It contains the application publication item table columns, as well as the fields listed in Table 2–24:

Table 2–24 Out Queue Interface Creation Parameters

Parameter	Description
`CLID$$CS`	A unique string identifying the client.
`TRANID$$`	A unique number identifying the transaction.

Table 2–24 (Cont.) Out Queue Interface Creation Parameters

Parameter	Description
DMLTYPE$$	Checks the type of DML instruction: ■ 'I' - Insert ■ 'D' - Delete ■ 'U' - Update

The following designates the structure when creating the In Queue:

```
create table 'CTM$'+name
(
CLID$$CS   VARCHAR2 (30),
publication item column definitions
TRANID$$   NUMBER (10),
DMLTYPE$$  CHAR (1) CHECK (DMLTYPE$$  IN ('I','U','D'),
)
```

Note: You must have the parameters in the same order as shown above for the Out Queue. It is different than the ordering in the In Queue.

Another example of creating an Out Queue is in the FServ example, which uses the default In Queue of CFM$PI_FSERV_TASKS and creates the CTM$PI_FSERV_TASKS Out Queue for the PI_FSERV_TASKS publication item, as follows:

```
create table CTM$PI_FSERV_TASKS(
                  CLID$$CS     varchar2(30),
                  ID           number,
                  EMP_ID       number,
                  CUST_ID      number,
                  STAT_ID      number,
                  NOTES        varchar2(255)
                  TRANID$$     number(10),
                  DMLTYPE$$    char(1) check(DMLTYPE$$ in ('I','U','D')),
);
```

Note: The application publication item table for the FServ example contains columns for ID, EMP_ID, CUST_ID, STAT_ID, and NOTES.

Queue Control Table

The Sync Server automatically creates a queue control table, C$INQ, and a temporary table, C$PUB_LIST_Q. You can process the information in the queue control table in the PL/SQL or Java callout methods to determine which publication items have received new transactions.

The parameters for the control table queue are listed in Table 2–25:

Table 2–25 Queue Control Table Parameters

Parameter	Description
CLID$$CS	A unique string identifying the client.
TRANID$$	A unique number identifying the transaction.
STORE	Represents the publication item name in the queue control table.

The control table has the following structure:

```
'C$INQ'
(
CLIENTID   VARCHAR2 (30),
TRANID$$   NUMBER,
STORE      VARCHAR2 (30),
)
```

Temporary Table

The DOWNLOAD_INIT procedure uses the Temporary Table C$PUB_LIST_Q for determining what publication items to download in the compose phase.

```
'C$PUB_LIST_Q'
(
NAME   VARCHAR2 (30),
COMP_REF   CHAR(1),
CHECK(COMP_REF IN('Y','N'))
)
```

The parameters for the manually created queues are listed in Table 2–26:

Table 2–26 Queue Interface Creation Parameters

Parameter	Description
NAME	The publication item name that is to be downloaded from the repository to the Out Queue.
COMP_REF	This value is 'Y' for complete refresh.

2.12.1.2 Queue-Based PL/SQL Callouts

The PL/SQL package for the queue-based publication callouts is in a package where the UPLOAD_COMPLETE, DOWNLOAD_INIT, DOWNLOAD_COMPLETE, and POPULATE_Q_REC_COUNT procedures are defined. The signatures for both callout procedures are as follows:

```
CREATE OR REPLACE PACKAGE CONS_QPKG AS
/*
 * notifies that In Queue has a new transaction by providing the client
 * identifier and the transaction identifier.
*/
PROCEDURE UPLOAD_COMPLETE(
     CLIENTID      IN     VARCHAR2,
     TRAN_ID      IN     NUMBER     -- IN queue tranid
     );
/*
 * initializes client data for download. provides the compose phase for the
 * client. The input data for this procedure is the client id, the last
 * and current transaction markers and the priority.
*/
PROCEDURE DOWNLOAD_INIT(
     CLIENTID      IN     VARCHAR2,
     LAST_TRAN     IN     NUMBER,
     CURR_TRAN     IN     NUMBER,
     HIGH_PRTY     IN     VARCHAR2
     );
/*
 *  notifies when all the client's data is sent
*/
PROCEDURE DOWNLOAD_COMPLETE(
```

```
     CLIENTID     IN     VARCHAR2
     );

PROCEDURE POPULATE_Q_REC_COUNT(
     CLIENTID     IN     VARCHAR2
     );

END CONS_QPKG;
/
```

2.12.1.2.1 In Queue Apply Phase Processing Within the UPLOAD_COMPLETE procedure, you should develop a method of applying all changes from the client to the correct tables in the repository. The FServ example performs the following:

1. From the Master Table C$INQ, locates the rows for the designated client and transaction identifiers that have been marked for update.
2. Retrieves the application publication item data and the DMLTYPE$$ from the In Queue, based on the client and transaction identifiers.
3. Performs insert, update, or delete (determined by the value in DMLTYPE$$) for updates in the application tables in the repository.
4. After updates are complete, delete the rows in the C$INQ and the In Queue that you just processed.

```
PROCEDURE UPLOAD_COMPLETE(CLIENTID IN VARCHAR2, TRAN_ID IN NUMBER) IS
/*create cursors for execution later */
/* PI_CUR locates the rows for the client out of the master table */
CURSOR PI_CUR(C_CLIENTID VARCHAR2, C_TRAN_ID NUMBER ) IS
   SELECT STORE FROM C$INQ
       WHERE CLID$$CS = C_CLIENTID AND TRANID$$ = C_TRAN_ID FOR UPDATE;
/* TASKS_CUR retrieves the values for the client data to be updated */
/*   from the In Queue */
CURSOR TASKS_CUR(C_CLIENTID varchar2, C_TRAN_ID number ) IS
   SELECT ID, EMP_ID, STAT_ID, NOTES, DMLTYPE$$ FROM CFM$PI_FSERV_TASKS
       WHERE CLID$$CS = C_CLIENTID AND TRANID$$ = C_TRAN_ID FOR UPDATE;
/* create variables */
TASK_OBJ TASKS_CUR%ROWTYPE;
PI_OBJ PI_CUR%ROWTYPE;
INSERT_NOT_ALLOWED EXCEPTION;
DELETE_NOT_ALLOWED EXCEPTION;
UNKNOWN_DMLTYPE EXCEPTION;

BEGIN

   OPEN PI_CUR(CLIENTID, TRAN_ID);
   /* C$INQ is used to find out which publication items have received data
      from clients. The publication item name is available in the STORE column
    */
   LOOP
     FETCH PI_CUR INTO PI_OBJ;
     EXIT WHEN PI_CUR%NOTFOUND;

   /* Locate the updates for the publication item PI_FSERV_TASKS */
     IF PI_OBJ.STORE = 'PI_FSERV_TASKS' THEN
       OPEN TASKS_CUR(CLIENTID, TRAN_ID);
       LOOP
         /* Process the In Queue for PI_FSERV_TASKS */
         FETCH TASKS_CUR INTO TASK_OBJ;
         EXIT WHEN TASKS_CUR%NOTFOUND;
```

```
        /* Discover the DML command requested. For this publication, only
            updates are allowed.
        IF TASK_OBJ.DMLTYPE$$ = 'I' THEN
           RAISE INSERT_NOT_ALLOWED;
        ELSIF TASK_OBJ.DMLTYPE$$ = 'U' THEN
           FSERV_TASKS.UPDATE_TASK(TASK_OBJ.ID, TASK_OBJ.EMP_ID,
                TASK_OBJ.STAT_ID, TASK_OBJ.NOTES);
        ELSIF TASK_OBJ.DMLTYPE$$ = 'D' THEN
           RAISE DELETE_NOT_ALLOWED;
        ELSE
           RAISE UNKNOWN_DMLTYPE;
        END IF;

        /* after processing, delete the update request from the In Queue */
        DELETE FROM CFM$PI_FSERV_TASKS WHERE CURRENT OF TASKS_CUR;
      END LOOP;
      close TASKS_CUR;
    END IF;

    /* after completing all updates for the client apply phase, delete from
       master queue */
    DELETE FROM C$INQ WHERE CURRENT OF PI_CUR;
  END LOOP;
END;
```

2.12.1.2.2 Out Queue Compose Phase Processing Within the `DOWNLOAD_INIT` procedure, develop a method of composing all changes from the server that are destined for the client from the publication item tables in the repository. The FServ example performs the following:

1. From the Temporary Table `C$PUB_LIST_Q`, discover the publication items that you should download data for the user using the client id, current and last transaction.
2. Retrieves the application publication item data into the Out Queue. This example always uses complete refresh.

```
PROCEDURE DOWNLOAD_INIT( CLIENTID IN VARCHAR2,
                         LAST_TRAN IN NUMBER,
                         CURR_TRAN IN NUMBER,
                         HIGH_PRTY IN VARCHAR2 ) IS
 /*create cursor used later in procedure which retrieves the publication name
  from the temporary table to perform compose phase.*/
 CURSOR PI_CUR IS SELECT NAME from C$PUB_LIST_Q;
 /*create variables*/
 PI_NAME VARCHAR2(50);
 STATID_CLOSE NUMBER;

 BEGIN

   OPEN PI_CUR;
   /* C$PUB_LIST_Q (the temporary table) is used to find out which pub items
      have data to download to clients through the publication item Out Queue.
      The publication item name is available in the NAME column
    */
   LOOP
     FETCH PI_CUR INTO PI_NAME;
     EXIT WHEN PI_CUR%NOTFOUND;

     /* Populate the Out Queue of pub item PI_FSERV_TASKS with all
```

```
        unclosed tasks for the employee with this CLIENTID using a complete
        refresh. COMP_REF is always reset to Y since partial refresh has
        not been implemented.
      */
     /* if the PI_FSERV_TASKS publication item has data ready for the client,
        then perform a complete refresh and place all data in the Out Queue */
     IF PI_NAME = 'PI_FSERV_TASKS' THEN
        UPDATE C$PUB_LIST_Q SET COMP_REF='Y' where NAME = 'PI_FSERV_TASKS';
        SELECT ID INTO STATID_CLOSE FROM MASTER.TASK_STATUS
             WHERE DESCRIPTION='CLOSED';
        INSERT INTO CTM$PI_FSERV_TASKS(CLID$$CS, ID, EMP_ID, CUST_ID,
             STAT_ID, NOTES, TRANID$$, DMLTYPE$$)
             SELECT CLIENTID, a.ID, a.EMP_ID, a.CUST_ID, a.STAT_ID, a.NOTES,
               CURR_TRAN, 'I' FROM MASTER.TASKS a, MASTER.EMPLOYEES b
               WHERE a.STAT_ID < STATID_CLOSE AND b.CLIENTID = CLIENTID
               AND a.EMP_ID = b.ID;
     END IF;
   END LOOP;
 END;
```

If, however, you want to perform another type of refresh than a complete refresh, such as an incremental refresh, then do the following:

1. Read the value of `COMP_REF`
2. If the value is N, insert only the new data into the Out Queue.

In this situation, the `LAST_TRAN` parameter becomes useful.

2.12.1.3 Create a Publication Item as a Queue

You create the publication item as you would normally, with one change: define the publication item as queue-based. See Section 4.4, "Create a Publication Item" for directions on how to define the publication item as queue-based when using MDW.

If you are using the Consolidator APIs, then the `createQueuePublicationItem` method creates a publication item in the form of a queue. This API call registers the publication item and creates `CFM$<name>` table as an In Queue, if one does not exist.

> **Note:** See the *Oracle Database Mobile Server JavaDoc* for more information.

You must provide the Consolidator Manager with the primary key, owner and name of the base table or view in order to create a queue that can be updated or refreshed with fast-refresh. If the base table or view name has no primary key, one can be specified in the primary key columns parameter. If primary key columns parameter is `NULL`, then Consolidator Manager uses the primary key of the base table.

2.12.1.4 Register the PL/SQL Package Outside the Repository

Once you finish developing the PL/SQL package, register the package in the `MOBILEADMIN` schema with the `registerQueuePkg` method. This method registers the package separately from the mobile server repository; although it refers to the In Queues, Out Queues, queue control table and temporary table that are defined in the repository.

The following methods register or remove a procedure, or retrieve the procedure name.

- The `registerQueuePkg` method registers the string `pkg` as the current procedure. The following registers the FServ package.

> **Note:** The developer used Consolidator Manager APIs to create the subscription, so this was included in the Java application that created the subscription.

```
/* Register the queue package for this publication */
  consolidatorManager.registerQueuePkg(QPKG_NAME, PUB_FSERV);
```

- The `getQueuePkg` method returns the name of the currently registered procedure.
- The `unRegisterQueuePkg` method removes the currently registered procedure.

> **Note:** See the *Oracle Database Mobile Server JavaDoc* for more information.

2.12.2 Creating Data Collection Queues for Uploading Client Collected Data

If you have an application that collects data on a client, such as taking inventory or the amount collected on a parking meter, then you can use data collection queues to improve the performance of uploading the data collected to the server. Since the data only flows from the client to the server, then synchronous communication is the best method for uploading massive amounts of data.

> **Note:** If you are collecting data on the client, but still need updates from the server, you can use the default method for synchronization or create your own queues. See Section 2.12.1, "Customizing Apply/Compose Phase of Synchronization with a Queue-Based Publication Item" for more information.

Data collection queues can be used for the following two types of data collection:

- New records that are inserted on the client.
- Existing records that are downloaded to the client in order that the user can modify and upload these records.

An example of the second type is a supply counting application. If you want to count the number of items in stock, then you could design the application table with the columns: Item and Count. Initially, populate the Item column and synchronize the data to the device, as follows:

Table 2–27 Stock Inventory Table

Item	Count
Apples	-
Pears	-
Oranges	-

The user on the client updates each item with the inventory amount, as follows:

Table 2–28 Stock Inventory Table

Item	Count
Apples	2
Pears	3
Oranges	1

The Data Collection Queue is lightweight and simple to create. Data collection queues are the same as regular queues with the exception that they provide automatic apply of the data uploaded by the client. However, you can customize whether the data is implicitly applied or not. This queue does not require the MGP to apply the changes. It does not create objects in the application schema or map data.

Data Collection Queues are easier to implement than a Queue-Based publication item. There is no need to create a package with callback methods, as Oracle Database Mobile Server takes care of automatically uploading any new data from the client. In addition, you configure how Oracle Database Mobile Server handles if there is any data to be downloaded or if you want the data on the client to be erased when it is uploaded to the server.

When you create the Data Collection Queue, the following is performed for you:

- Automatically generates the in-queue when the publication item is created, which is named as follows: `CFM$<publication_item_name>`.
- Optionally, enables the developer to choose automatic removal of client data once captured to the server. This is specified when you create the publication item.
- Optionally, if you need an out-queue, then the developer can specify the out-queue or to have Oracle Database Mobile Server automatically generate an out-queue, which would be named as follows: `CTM$<publication_item_name>`.

Just like for regular queues, users can create their own Out Queue logic. By default, the Out Queue created is an empty view with the name of (`CTM$<`*`publication_item`*`>`). An empty view is a view that selects zero records. Therefore, by default, data collection queues do not pick up any data from the server.

You can modify how the data collection queue behaves when you create it using the `ConsolidatorManager.createDataCollectionQueue` method. The following parameters effect the behavior of your data collection queue:

- Specify an Out Queue—Out Queue creation is affected by the `isOutView` boolean input parameter. If `isOutView` is TRUE, then creates the Out Queue as an empty view; if FALSE, then creates the Out Queue as a table.
- Automatic Removal of Data on the Client—Users can customize the default behavior of data purging on the client by setting the `purgeClientAfterSync` parameter to either true or false.
  - If TRUE, then the client uploads its data changes and removes the records from the client database. At this point, the table on the client is empty. If the Out Queue on the server is empty, the client no longer has any records. If the Out Queue is not empty, the client downloads these records and the table on the client contains only these records.
  - If FALSE, then the client records remain on the device after synchronization unless the server explicitly sends the `DELETE` command, in the same manner as a normal publication item.

2.12.2.1 Creating a Data Collection Queue

When you create a data collection queue, you perform the following:

> **Note:** All `ConsolidatorManager` methods are fully documented in the *Oracle Database Mobile Server Javadoc*. This section provides context of the order in which to execute these methods.

1. Create the tables for the data that the queue updates on the back-end Oracle database.
2. Create the data collection queue and its publication item using the `ConsolidatorManager createDataCollectionQueue` method, where the input parameters are as follows:
   - `name`—A character string specifying a new publication item name.
   - `owner`—A string specifying the base schema object owner.
   - `store`—A string specifying the table name that it is based on.
   - `inq_cols`—A string specifigying columns in the order in which to replicate them. If `NULL`, then defaults to `*`, which makes the SQL statement, `select * from <table>`.
   - `pk_columns`—A string specifying the primary keys.
   - `purgeClientAfterSync`—If true, removes client data from the mobile device when uploaded to the server.
   - `isOutView`—If true, then creates Out Queue as an empty view, otherwise creates Out Queue as a table.

   The following creates the `PI_CUSTOMERS` data collection queue:

```
cm.createDataCollectionQueue( "PI_CUSTOMERS", /* Publication Item name */
        MYSCHEMA,                             /* Schema owner */
       "CUSTOMERS",                           /* store */
        NULL,                                 /* inqueue_columns
        NULL,                                 /* NULL selects all pk_columns
        true,                                 /* removes old data after sync
        true );                               /* isOutView */
```

3. Create the publication that is to be used by the data collection queue. Use the `ConsolidatorManager createPublication` method. The following creates the `PUB_CUSTOMERS` publication that is used by the `PI_CUSTOMERS` data collection queue:

```
cm.createPublication("PUB_CUSTOMERS",0, "sales.%s", NULL);
```

4. Add the publication item created within step 1 within this publication with the `ConsolidatorManager addPublicationItem` method. The following adds a publication item to the publication:

```
cm.addPublicationItem("PUB_CUSTOMERS", "PI_CUSTOMERS", NULL, NULL,
                  "S", NULL, NULL);
```

5. If you want to have data download from the server to the mobile client, create an Out Queue with a name that consists of `CTM$<publication_item_name>`. The following replaces the default Out Queue view for `CUSTOMER` with a view that selects all customers assigned to the `EMP_ID` associated with current sync session.

```
stmt.executeUpdate(
    "CREATE OR REPLACE VIEW CTM$"+pubIs[0]+" ( CLID$$CS, TRANID$$,
           DMLTYPE$$,"+" CUST_ID, CNAME, CCOMPANY, CPHONE,
            CCONTACT_DATE )"+"\n
          AS SELECT CONS_EXT.GET_CURR_CLIENT, 999999999, 'I',cust.*
        FROM CUSTOMERS cust "+"\n
        WHERE cust.CUST_ID IN (SELECT CUST_ID
        FROM CUSTOMER_ASSIGNMENT WHERE EMP_ID IN "+"\n
        (SELECT EMP_ID FROM SESSION_EMP
             WHERE SESSION_ID = DBMS_SESSION.UNIQUE_SESSION_ID))"
);
```

Note: See the Oracle Database Mobile Server samples page for the full data collection queue example from which these snippets were taken. The example demonstrates both a regular queue and a data collection queue.

2.12.3 Selecting How and When to Notify Clients of Composed Data

If you have created your own compose logic, such as in the queue-based publications, then you may want the server to notify the client that there is data to be downloaded. You can take control of starting an automatic synchronization from the server using the enqueue notification APIs.

There are other situations where you may want to control how and when clients are notified of compose data from the synchronization process. For example, if you have so many clients that to notify all of them of the data waiting for them would overload your system, you may want to control the process by notifying clients in batches.

In the normal synchronization process, when the compose phase is completed, all clients that have data in the Out Queue are notified to download the data. If, for example, you have 2000 clients, having all 2000 clients request a download at the same time could overrun your server and cause a performance issue. In this scenario, you could take control of the notification process and notify 100 clients at a time over the span of a couple of hours. This way, all of the clients receive the data in a timely fashion and your server is not overrun.

You can use the enqueue notification functionality, as follows:

- If you implement queue-based publications for the compose phase, you can notify the clients with the `EN_QUEUE_NOTIFICATION` function within the Queue-based `DOWNLOAD_INIT` function.
- If you write your own compose function, use the `enQueueNotification` method to notify the client that there is data to download.

This starts an automatic synchronization process for the intended client.

The enqueue notification APIs enable the server to tell the client that there is data to be downloaded and what type of data is waiting. Notifying the client of what type of data is waiting enables the client to evaluate whether it conforms to any automatic synchronization rules. For example, if the server has 10 records of low priority data, but the client has set the Server MGP Compose rule to only start an automatic synchronization if 20 records of low priority data exist, then the automatic synchronization is not started. So, the notification API input parameters include parameters that enable the server to describe the data that exists on the server.

A notification API is provided for you in both PL/SQL and Java, as follows:

- **Java**: the `ConsolidatorManager enQueueNotification` method

```
public long enQueueNotification(java.lang.String clientid,
                                java.lang.String publication,
                                java.lang.String pubItems,
                                int recordCount,
                                int dataSize,
                                int priority)
                          throws ConsolidatorException
```

- **PL/SQL**: the `EN_QUEUE_NOTIFICATION` function

```
FUNCTION EN_QUEUE_NOTIFICATION(
  CLIENTID        IN VARCHAR2,
  PUBLICATION     IN VARCHAR2,
  PUB_ITEMS       IN VARCHAR2,
  RECORD_COUNT    IN NUMBER,
  DATA_SIZE       IN NUMBER,
  PRIORITY        IN NUMBER)
RETURN NUMBER;
```

Where the parameters for the above are as follows:

Table 2–29 Enqueue Notification Parameters

Parameters	Description
`clientid`	Consolidator client id, which is normally the user name on the client device. This identifies the client to be notified. If the client does not have any automatic synchronization rules, this is the only required paramter for an automatic synchronization to start.
`publication`	Name of the publication for which you want notification control. This tells the client for which publication the data is destined.
`pubItems`	One or more publication items for which you want notification. Separate multiple publication items with a comma. This notifies the clients for which publication items the data applies.
`recordCount`	This notifies the client how many records exist on the server for the download.
`dataSize`	Reserved for future expansion.
`priority`	This notifies the client of the priority of the data that exists on the server. The value is 0 for high and 1 for low.

The enqueue notification API returns a unique notification ID, which can be used to query notification status in the `isNotificationSent` method, which is as follows:

- JAVA

```
public boolean isNotificationSent(long notificationId)
  throws ConsolidatorException
```

- PL/SQL

```
FUNCTION NOTIFICATION_SENT(
   NOTIFICATION_ID IN NUMBER)
RETURN BOOLEAN;
```

If the notification has been sent, a boolean value of TRUE is returned.

2.13 Synchronization Performance

There are certain optimizations you can do to increase performance. See Section 1.2 "Increasing Synchronization Performance" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for a full description.

2.14 Troubleshooting Synchronization Errors

The following section can assist you in troubleshooting any synchronization errors:

- Section 2.14.1, "Foreign Key Constraints in Updatable Publication Items"

2.14.1 Foreign Key Constraints in Updatable Publication Items

Replicating tables between Oracle database and clients in updatable mode can result in foreign key constraint violations if the tables have referential integrity constraints. When a foreign key constraint violation occurs, the server rejects the client transaction.

- Section 2.14.1.1, "Foreign Key Constraint Violation Example"
- Section 2.14.1.2, "Avoiding Constraint Violations with Table Weights"
- Section 2.14.1.3, "Avoiding Constraint Violations with BeforeApply and After Apply"

2.14.1.1 Foreign Key Constraint Violation Example

For example, two tables `EMP` and `DEPT` have referential integrity constraints. The `DeptNo` (department number) attribute in the `DEPT` table is a foreign key in the `EMP` table. The `DeptNo` value for each employee in the `EMP` table must be a valid `DeptNo` value in the `DEPT` table.

A mobile server user adds a new department to the `DEPT` table, and then adds a new employee to this department in the `EMP` table. The transaction first updates `DEPT` and then updates the `EMP` table. However, the database application does not store the sequence in which these operations were executed.

When the user replicates with the mobile server, the mobile server updates the `EMP` table first. In doing so, it attempts to create a new record in `EMP` with an invalid foreign key value for `DeptNo`. Oracle database detects a referential integrity violation. The mobile server rolls back the transaction and places the transaction data in the mobile server error queue. In this case, the foreign key constraint violation occurred because the operations within the transaction are performed out of their original sequence.

Avoid this violation by setting table weights to each of the tables in the master-detail relationship. See Section 2.14.1.2, "Avoiding Constraint Violations with Table Weights" for more information.

2.14.1.2 Avoiding Constraint Violations with Table Weights

The mobile server uses table weight to determine in which order to apply client operations to master tables. Table weight is expressed as an integer and are implemented as follows:

1. Client `INSERT` operations are executed first, from lowest to highest table weight order.
2. Client `DELETE` operations are executed next, from highest to lowest table weight order.

3. Client `UPDATE` operations are executed last, from lowest to highest table weight order.

In the example listed in Section 2.14.1.1, "Foreign Key Constraint Violation Example", a constraint violation error could be resolved by assigning `DEPT` a lower table weight than `EMP`. For example:

```
(DEPT weight=1, EMP weight=2)
```

You define the order weight for tables when you add a publication item to the publication. For more information on setting table weights in the publication item, see Section 2.4.1.7.2, "Using Table Weight".

2.14.1.3 Avoiding Constraint Violations with BeforeApply and After Apply

You can use a PL/SQL procedure to avoid foreign key constraint violations based on out-of-sequence operations by using `DEFERRABLE` constraints in conjunction with the `BeforeApply` and `AfterApply` functions. See Section 2.9.3.2, "Defer Constraint Checking Until After All Transactions Are Applied" for more information.

2.15 Register a Remote Oracle Database for Application Data

By default, the repository metadata and the application schemas are present in the same database. However, it is possible to place the application schemas in a database other than the MAIN database where the repository exists. This can be an advantage from a performance or administrative viewpoint.

Thus, you can spread your application data across multiple databases.

> **Note:** We refer to the database where the application schema resides as remote because it is separate from the MAIN database that contains the repository. It does not mean that the database is geographically remote. It can be local or remote. For performance reasons, the mobile server must have connectivity to all databases involved in the synchronization—MAIN and remote.

This section describes how to register a remote Oracle database containing application schemas, using the `ConsolidatorManager` APIs. However, it is recommended that you use the Oracle Database Mobile Server GUI tools for this task unless you have a specific need to use the API. For concepts and description of how to perform this with the Oracle Database Mobile Server GUI tools, see Section 5.8.1, "Register or Deregister an Oracle Database for Application Data" in the *Oracle Database Mobile Server Administration and Deployment Guide*.

To use an Oracle database other than the Oracle database used for the repository, perform the following:

1. Use the `apprepwizard` script to setup a remote application repository. See Section 2.15.1, "Set up a Remote Application Repository With the APPREPWIZARD Script" for details.
2. Register the Oracle database as described in Section 2.15.2, "Register or Deregister a Remote Oracle Database for Application Data".
3. When creating the publication and publication items, specify the name of the registered Oracle database that contains the application schemas. All data for a single application—that is, all publication items for the publication—must be contained in the same Oracle database.

2.15.1 Set up a Remote Application Repository With the APPREPWIZARD Script

Use the `apprepwizard` script to setup a remote application repository. This script creates and initializes an administrator schema with the same name as the adminstrator schema in the Main database. For example, if the administrator schema name in the Main database is `mobileadmin`, then the `apprepwizard` script creates a `mobileadmin` schema on the remote database.

The `apprepwizard` script is located in the *`ORACLE_HOME`*`/Mobile/Server/admin`. The usage of this script is as follows:

```
apprepwizard.bat <MAIN_Repository_Schema_Name> <MAIN_Repository_Schema_Password>
 <Application_Database_Administrator_User_Name>
 <Application_Database_Administrator_Password>
 <Application_Database_JDBC_URL> <Application_Database_Schema_Password>
 [<DB_name>]
```

Where each parameter is as follows:

- `MAIN_Repository_Schema_Name`: Provide the repository schema name, which exists on the Main database. The default is `MOBILEADMIN`.
- `MAIN_Repository_Schema_Password`: Provide the password for the repository administrator schema.
- `Application_Database_Administrator_User_Name`: Any user with administrator privileges at the application database. such as `SYSTEM`.
- `Application_Database_Administrator_Password`: Password of the administrator user for the application database.
- `Application_Database_JDBC_URL`: JDBC URL of the application database.
- `Application_Database_Schema_Password`: Password of the schema, which is created at the application database. The user name is the same as the repository schema name.
- `DB_Name`: Optionally, the user can provide a name to identify this database. This name is used in logging. By default, the log is sent to the console. If this name is provided as the last parameter, then the log is generated in the By default, the log is sent to the console. If the database name is provided as the last parameter, then the log is generated in the *`ORACLE_HOME`*`/Mobile/Server/<DB_NAME>/apprepository.log` file.

This script installs silently. Thus, If you execute this script without any arguments, nothing is performed.

2.15.2 Register or Deregister a Remote Oracle Database for Application Data

Use the following `ConsolidatorManager` APIs to register, deregister, or alter the properties of the remote Oracle database:

```
void registerDatabase(String name, Consolidator.DBProps props)
void deRegisterDatabase(String name)
void alterDatabase(String name, Consolidator.DBProps props)
```

Where:

- **Name**—An identifying name for the database where the application schema resides. Once defined, this name cannot be modified. This name must be unique across all registered database names.

- **DBProps**—A class that contains the JDBC URL, password and description, as follows:

```
public static class DBProps {
 public String jdbcUrl;
 public String adminPassword;
 public String description;
}
```

  - **JDBC URL**—The JDBC URL can be one of the following formats:
    * The URL for a single Oracle database has the following structure: `jdbc:oracle:thin:@<host>:<port>:<SID>`
    * The JDBC URL for an Oracle RAC database can have more than one address in it for multiple Oracle databases in the cluster and follows this URL structure:

```
jdbc:oracle:thin:@(DESCRIPTION=
 (ADDRESS_LIST=
   (ADDRESS=(PROTOCOL=TCP)(HOST=PRIMARY_NODE_HOSTNAME)(PORT=1521))
   (ADDRESS=(PROTOCOL=TCP)(HOST=SECONDARY_NODE_HOSTNAME)(PORT=1521))
 )
 (CONNECT_DATA=(SERVICE_NAME=DATABASE_SERVICENAME)))
```

  - **Password**—The administrator password is used to logon to the database. The administrator name is the same as what was defined for the main database.

    When defining, the password must conform to the following restrictions:
    - not case sensitive
    - cannot contain white space characters
    - maximum length of 28 characters
    - must begin with an alphabet
    - can contain only alphanumeric characters
    - cannot be an Oracle database reserved word
  - **Description**—A user-defined description to help identify this database.

Refer to the `ConsolidatorManager` in the *Oracle Database Mobile Server JavaDoc* for more details.

The following code example registers a database as `APP1`. The `registerDatabase` API stores access information for the application repository and provides a name so that publications, publication items, and MGP Jobs can be created against this repository. It does not define the administrator schema.

```
Consolidator.DBProps props = new Consolidator.DBProps();
props.jdbcUrl = "jdbc:oracle:thin:@apphost:1521:app1";
props.description="App database 1"
props.adminPassword = "secret";
consMgr.registerDatabase("APP1", props);
```

The following code example deregisters the `APP1` database.

```
consMgr.deRegisterDatabase("APP1");
```

You can retrieve the names of all of the registered databases with the `getDatabaseInstances` method, which is as follows:

```
Map getDatabaseInstances()
```

The `Map` returned by `getDatabaseInstances` method contains a keyset of the application database names and the entry for each key is a `Consolidator.DBProps` class where the `adminPassword` is always `NULL` for security purposes.

2.15.3 Create Publication, Publication Item, Hints and Virtual Primary Keys on a Remote Database

You must have already registered the remote database before defining publications, publication items, hints, and virtual primary keys that use the application data schemas and tables on the remote database. In the `ConsolidatorManager` API calls, the registered name of the remote database is required.

> **Note:** The publication and publication item names are unique irrespective of where the data resides.

All publication items within a publication must be defined on tables within the same database.

The following example illustrates the creation of a publication and a publication item against a remote database registered as `APP1`. Refer to the `ConsolidatorManager` in the *Oracle Database Mobile Server JavaDoc* for more details.

```
ConsolidatorManager consMgr = new ConsolidatorManager();
consMgr.openConnection("mobileadmin", "mobileadmin",
 "oracle:jdbc:thin:@host1:1521:master");
consMgr.createPublication( "PUB1","APP1",Consolidator.DFLT_CREATOR_ID,
 "ddb.%s", NULL);

Consolidator.PubItemProps taskPIProps = new Consolidator.PubItemProps();
taskPIProps.db_inst = "APP1"; // Remote App database name as registered
taskPIProps.owner = "APPUSER1";
taskPIProps.store = "TASKS";
taskPIProps.refresh_mode = "F";
taskPIProps.select_stmt = "select id, emp_id, cust_id, stat_id, notes
 from APPUSER1.TASKS";
taskPIProps.cbk_owner = "MOBILEADMIN";
taskPIProps.cbk_name = "TASKSPI_PKG";
consMgr.createPublicationItem( "PI_1_TASKS", taskPIPProps);

consMgr.addPublicationItem("PUB1", "PI_1_TASKS", NULL, NULL, "S", NULL, NULL);
consMgr.createSubscription( "PUB1", "USER1");
consMgr.instantiateSubscription("PUB1", "USER1");
consMgr.closeConnection();
```

Other API calls for managing data collection queues, hints, and virtual primary keys that require the remote database name are shown below. Refer to the `ConsolidatorManager` in the *Oracle Database Mobile Server JavaDoc* for more details.

- **Data Collection Queue**

```
void createDataCollectionQueue(String name, String db_inst,
   String owner, String store, String inq_cols, String pk_columns,
   boolean purgeClientAfterSync, boolean isOutView)
```

- **Hint**

```
void parentHint(String db_inst, String owner, String store, String owner_d,
    String store_d)
```

```
void dependencyHint(String db_inst, String owner, String store,
    String owner_d, String store_d)
void removeDependencyHint(String db_inst, String owner, String store,
    String owner_d, String store_d)
```

- **Virtual Primary Key**

```
public void createVirtualPKColumn(String db_inst, String owner,
    String store, String column)
public void dropVirtualPKColumns(String db_inst, String owner,
    String store)
```

The APIs used for creating a publication and publication item is the same except for the addition of the remote database name. Following is an example that provides the remote database name, APP1, in bold for creating a publication and publication item:

```
ConsolidatorManager consMgr = new ConsolidatorManager();
consMgr.openConnection("mobileadmin", "mobileadmin",
"oracle:jdbc:thin:@host1:1521:master");
consMgr.createPublication( "PUB1","APP1",Consolidator.DFLT_CREATOR_ID,
       "ddb.%s", NULL);
Consolidator.PubItemProps taskPIProps = new Consolidator.PubItemProps();
taskPIProps.db_inst = "APP1"; // Remote APP instance name as registered
taskPIProps.owner = "APPUSER1";
taskPIProps.store = "TASKS";
taskPIProps.refresh_mode = "F";
taskPIProps.select_stmt = "select id, emp_id, cust_id, stat_id, notes from
APPUSER1.TASKSî;
taskPIProps.cbk_owner = "MOBILEADMIN";
taskPIProps.cbk_name = "TASKSPI_PKG";
consMgr.createPublicationItem( "PI_1_TASKS", taskPIPProps);
consMgr.addPublicationItem("PUB1", "PI_1_TASKS", NULL, NULL, "S", NULL, NULL);
consMgr.createSubscription( "PUB1", "USER1");
consMgr.instantiateSubscription("PUB1", "USER1");
consMgr.closeConnection();
```

2.15.4 Using Callbacks on Remote Databases

The following sections describe how the synchronization callbacks, described in Section 2.7, "Customize What Occurs Before and After Synchronization Phases", must be handled for the remote database:

- Section 2.15.4.1, "Customize Callbacks on the Remote Database"
- Section 2.15.4.2, "Publication Item Level Callbacks for the MGP Apply/Compose Phases"

2.15.4.1 Customize Callbacks on the Remote Database

The Customize callbacks, as described in Section 2.7.1, "Customize What Occurs Before and After Every Phase of Each Synchronization", are created to perform defined tasks before or after any phase of synchronization.

Most of the callbacks pertain to MGP processing. Since an MGP Job executes against a database, these callbacks are invoked separately by each job against the corresponding database. Callbacks that are not related to the MGP are invoked against the MAIN database. Thus, the callback PL/SQL package must be created on the MAIN database as well as on the appropriate remote databases.

2.15.4.2 Publication Item Level Callbacks for the MGP Apply/Compose Phases

Define the MGP publication item level callbacks on the database against which the publication item is defined. Then, these can access the base tables on that database.

For full details on the MGP publication item level callbacks, see Section 2.7.2, "Customize What Occurs Before and After Compose/Apply Phases for a Single Publication Item".

2.15.4.3 Customizing the Apply/Compose Phase for a Queue-Based Publication Item on a Remote Database

When you customize the apply/compose phase for a queue-based publication item, as described in Section 2.12.1, "Customizing Apply/Compose Phase of Synchronization with a Queue-Based Publication Item", then these packages must be defined on the database where the queue-based publication item base tables exist. Thus, if the base tables exist on a remote database, then the packages must be defined on the remote database.

2.16 Create a Synonym for Remote Database Link Support For a Publication Item

Publication items can be defined for database objects existing on remote databases outside of the mobile server repository. Local private synonyms of the remote objects can be created in the Oracle database. However, we recommend that you use the remote database functionality as described in Section 2.15, "Register a Remote Oracle Database for Application Data".

If you still decide to use database links for defining publication items on remote databases, then you can execute the following SQL script located in the `<ORACLE_HOME>\Mobile\server\admin\consolidator_rmt.sql` directory on the remote schema in order to create Consolidator Manager logging objects.

The synonyms should then be published using the `createPublicationItem` method of the `ConsolidatorManager` object. If the remote object is a view that needs to be published in updatable mode and/or fast-refresh mode, the remote parent table must also be published locally. Parent hints should be provided for the synonym of the remote view similar those used for local, updatable and/or fast refreshable views.

Two additional methods have been created, `dependencyHint` and `removeDependencyHint`, to deal with non-apparent dependencies introduced by publication of remote objects.

Remote links to the Oracle database must be established before attempting remote linking procedures, please refer to the *Oracle SQL Reference* for this information.

> **Note:** The performance of synchronization from remote databases is subject to network throughput and the performance of remote query processing. Because of this, remote data synchronization is best used for simple views or tables with limited amount of data.

The following sections describe how to manage remote links:

- Section 2.16.1, "Publishing Synonyms for the Remote Object Using CreatePublicationItem"
- Section 2.16.2, "Creating or Removing a Dependency Hint"

2.16.1 Publishing Synonyms for the Remote Object Using CreatePublicationItem

The `createPublicationItem` method creates a new, stand-alone publication item as a remote database object. If the URL string is used, the remote connection is established and closed automatically. If the connection is `NULL` or cannot be established, an exception is thrown. The remote connection information is used to create logging objects on the linked database and to extract metadata.

> **Note:** See the *Oracle Database Mobile Server JavaDoc* for more information.

```
consolidatorManager.createPublicationItem(
     "jdbc:oracle:oci8:@oracle.world",
     "P_SAMPLE1",
     "SAMPLE1",
     "PAYROLL_SYN",
     "F"
     "SELECT * FROM sample1.PAYROLL_SYN"+"WHERE SALARY >:CAP", NULL, NULL);
```

> **Note:** Within the select statement, the parameter name for the data subset must be prefixed with a colon, for example `:CAP`.

2.16.2 Creating or Removing a Dependency Hint

Use the `dependencyHint` method to create a hint for a non-apparent dependency.

```
Given remote view definition
        create payroll_view as
        select p.pid, e.name
        from payroll p, emp e
        where p.emp_id = e.emp_id;

Execute locally
        create synonym v_payroll_syn for payroll_view@<remote_link_address>;
        create synonym t_emp_syn for emp@<remote_link_address>;
```

Where `<remote_link_address>` is the link established on the Oracle database. Use `dependencyHint` to indicate that the local synonym `v_payroll_syn` depends on the local synonym `t_emp_syn`:

```
consolidatorManager.dependencyHint("SAMPLE1","V_PAYROLL_SYN","SAMPLE1","T_EMP_
SYN");
```

Use the `removeDependencyHint` method to remove a hint for a non-apparent dependency.

> **Note:** See the *Oracle Database Mobile Server JavaDoc* for more information.

2.17 Parent Tables Needed for Updateable Views

For a view to be updatable, it must have a parent table. A parent table can be any one of the view base tables in which a primary key is included in the view column list and is unique in the view row set. If you want to make a view updatable, provide the

mobile server with the appropriate hint and the view parent table before you create a publication item on the view.

To make publication items based on a updatable view, use the following two mechanisms:

- Parent table hints
- `INSTEAD OF` triggers or DML procedure callouts

2.17.1 Creating a Parent Hint

Parent table hints define the parent table for a given view. Parent table hints are provided through the `parentHint` method of the Consolidator Manager object, as follows:

```
consolidatorManager.parentHint("SAMPLE3","ADDROLRL4P","SAMPLE3","ADDRESS");
```

See the *Oracle Database Mobile Server JavaDoc* for more information.

2.17.2 INSTEAD OF Triggers

`INSTEAD OF` triggers are used to execute `INSTEAD OF INSERT`, `INSTEAD OF UPDATE`, or `INSTEAD OF DELETE` commands. `INSTEAD OF` triggers also map these DML commands into operations that are performed against the view base tables. `INSTEAD OF` triggers are a function of the Oracle database. See the Oracle database documentation for details on `INSTEAD OF` triggers.

2.18 Manipulating Application Tables

If you need to manipulate the application tables to create a secondary index or a virtual primary key, you can use `ConsolidatorManager` methods to programmatically perform these tasks in your application, as described in the following sections:

- Section 2.18.1, "Creating Secondary Indexes on Client Device"
- Section 2.18.2, "Virtual Primary Key"

2.18.1 Creating Secondary Indexes on Client Device

The first time a client synchronizes, the mobile server automatically enables a mobile client to create the database objects on the client in the form of snapshots. By default, the primary key index of a table is automatically replicated from the server. You can create secondary indexes on a publication item through the Consolidator Manager APIs. See the *Oracle Database Mobile Server Javadoc* for specific API information. See Section 2.4.1.6, "Create Publication Item Indexes" for an example.

2.18.2 Virtual Primary Key

You can specify a virtual primary key for publication items where the base object does not have a primary key defined. This is useful if you want to create a fast refresh publication item on a table that does not have a primary key.

A virtual primary key must be unique and not NULL. A virtual primary key can consist of a single or multiple columns, where each column included in the virtual primary key must not NULL. If a `NULL` value is entered into any column of a virtual primary key, this results in an error. If the virtual primary key is on a single column, it

must be unique; if the virtual primary key consists of a composite of multiple columns, then the composite must be unique.

If you want to create a virtual primary key for more than one column, then the API must be called separately for each column that you wish to assign to that virtual primary key.

Use the `createVirtualPKColumn` method to create a virtual primary key column.

```
consolidatorManager.createVirtualPKColumn("SAMPLE1", "DEPT", "DEPT_ID");
```

Use the `dropVirtualPKColumns` method to drop a virtual primary key.

```
consolidatorManager.dropVirtualPKColumns("SAMPLE1", "DEPT");
```

Note: See the *Oracle Database Mobile Server JavaDoc* for more information.

2.19 Facilitating Schema Evolution

You can use schema evolution when adding or altering a column in the application tables. If you alter the schema, then the client receives a complete refresh on the modified publication item, but not for the entire publication.

Note: You should stop all synchronization events and MGP activity during a schema evolution.

The following types of schema modifications are supported:

- Adding new columns.
- Changing the type of a column. You can only modify the type of a column in accordance to the Oracle Database limitations.
- Increasing the width of a column.
- Modifying the publication item sub-query.
- Modifying the column order in the publication item select statement.
- Removing columns from the publication item SQL query.

For facilitating schema evolution, perform the following:

1. If necessary, modify the table in the back-end Oracle database.
2. If necessary, modify the publication item SQL query directly on the production repository with MDW or use the `alterPublicationItem` API. Modifying the SQL query causes the schema evolution to occur.

   A schema evolution only occurs if the SQL query is modified. Some schema modifications will directly lead to modifying the query, while others will not. The SQL query will not change for a column type or width change. In these cases, modify the SQL query by adding a space in the string. This will force schema evolution to occur properly.
3. After altering the SQL query, either use Mobile Manager to refresh the metadata cache or restart the mobile server. To refresh the metadata cache through the mobile server, select *Data Synchronization -> Administration -> Reset Metadata Cache* or execute the resetCache method of the ConsolidatorManager class.

Note: Schema evolution does not occur when clients perform high-priority sync. However, the next regular-priority foreground sync will receive the new schema along with a complete refresh of the publication item data.

2.19.1 Schema Evolution Involving a Primary Key

To perform a schema evolution that includes a modification to the primary key, recreate the publication items that use the base table where the primary key is changed.

The following steps describe how to change the primary key column and then recreate the publication item.

Note: The steps below must be followed in the order listed.

1. Remove the altered publication item from the publication using MDW or removePublicationItem API.
2. Modify the primary key definition of the base table in the back-end Oracle database.
3. Drop the altered publication item from the repository using MDW or dropPublicationItem API.
4. Create a new publication item based on the changed base table using MDW or createPublicationItem API. This should be a duplicate of the previously dropped publication item.
5. Add the publication item to the publication through MDW or addPublicationItem API.
6. Recreate the publication item and have it automatically re-added to the publicationt using recreatePublicationItem API (alternatively to steps 3-5).
7. Reset the Metadata Cache using the Mobile Manager by selecting *Data Synchronization -> Administration -> Reset Metadata Cache*.
8. Synchronize on the existing client device to bring over the new publication.
9. Verify that primary key definition on the client device is the same as that on the server, after the synchronization is complete.

The following steps describe how to remove the primary key constraint, add a new column and identify it as the primary or virtual primary key and then recreate the publication item:

Note: The steps below must be followed in the order listed.

1. Remove the publication item from the publication using MDW, and drop the publication item from the repository.
2. Modify the table in the back-end Oracle database, as described in the following steps:
   - Drop the primary key constraint:

     ```
     alter table table1 drop constraint pk_constraint;
     ```

- Add a new primary key or virtual primary key:

  ```
  alter table table1 add my_new_col number(5,0) not null;
  ```
- Populate the new primary key or virtual primary key column with proper values.
- Alter the table to create a primary key or virtual primary key constraint on the new column:

  ```
  alter table table1 add constraint pk_constraint unique (my_new_col);
  ```

3. Create a new publication item for table1 in MDW. This should be a duplicate of the previously dropped publication item, but with the new column included. When creating the publication item, verify that the my_new_col appears as the primary key.
4. Add the publication item to the publication.
5. Reset the metadata cache using the Mobile Manager by selecting *Data Synchronization -> Administration -> Reset Metadata Cache*.
6. Verify that the new primary key is in effect in the *Parent Table Primary Key* and *Base Table Primary Key* fields in the *Publication Item Detail* screen (in the Mobile Manager).
7. Synchronize on the existing client device to bring over the new publication.
8. Verify that the new column is present and that it is included in the primary key on the client device, after the synchronization is complete.

2.20 Set DBA or Operational Privileges for the Mobile Server

You can set either DBA or operational privileges for the mobile server with the following Consolidator Manager API:

```
void setMobilePrivileges( String dba_schema, String dba_pass, int type )
       throws ConsolidatorException
```

where the input parameter are as follows:

- `dba_schema`—The DBA schema name
- `dba_pass`—The DBA password
- `type`—Define the user by setting this parameter to either `Consolidator.DBA` or `Consolidator.OPER`

If you specify Consolidator.DBA, then the privileges needed are those necessary for granting DBA privileges that are required for publish/subscribe functions of the mobile server.

If you specify `Consolidator.OPER` type, then the privileges needed are those necessary for executing the mobile server without any schema modifications. The OPER is given DML and select access to publication item base objects, version, log, and error queue tables.

The mobile server privileges are modified using the `C$MOBILE_PRIVILEGES` PL/SQL package, which is created for you automatically after the first time you use the `setMobilePrivileges` procedure. After the package is created, the mobile server privileges can be administered from SQL or from this Java API.

3

Managing Synchronization on the Mobile Client

To manage manual and automatic synchronization, Native (C) APIs and Java APIs are available for Win32, Windows Mobile and Linux clients. In addition, NET APIs are also available for manual and automatic synchronization on Win32 and Windows Mobile clients. Native APIs are implemented in the following:

- The ose.dll(libose.so on linux) implements manual synchronization APIs - both OSE (11g API) and OCAPI (10. 3 and earlier API for manual synchronization for backward compatibility, implemented as a wrapper around the new API). In MDK, OSE APIs are declared in ose.h and OCAPI APIs are declared in ocapi.h (see Section 3.1.1.2, "OSE Synchronization APIs For Native Applications").
- The bgsync.dll(libbgsync.so on linux) implements automatic synchronization control APIs (new 11g APIs). In MDK, these APIs are declared in bgsync.h (see Section 3.2.1.2.1, "Overview") .
- The olSyncAgent.dll (libautosync.so on linux) implements older 10.3 automatic synchronization control APIs for backward compatibility (see Section 3.2.1.4, "OCAPI Sync Control APIs"). In MDK, these APIs are declared in olite_bgsync.h. Java APIs for these platforms are implemented in jsync.jar. In addition, jsync.jar uses JNI code implemented in msync_java.dll (libmsync_java.so on linux).

To call native APIs on Win32, Linux and Windows Mobile do the following:

- For Win32, ensure that all client dlls (both specified above and their dependencies) are in the PATH.
- For Linux, ensure all the shared libraries are found in LD_LIBRARY_PATH.
- For Windows Mobile, the client setup will put these dlls into \Windows directory on the device.

  NET APIs are implemented in Oracle.OpenSync.dll (see Section 3.2.1.3.1, "Overview"), which depends on all other client binaries, so the requirement above also applies.

Note: To call Java APIs on Win32, Windows Mobile and Linux, in addition to above requirements, make sure that msync_ java.dll(libmsync_java.so on linux) is also found in PATH (LD_ LIBRARY_PATH on linux).

Ensure that jsync.jar is in the CLASSPATH.

Note: The dlls/shared libraries in MDK are located in <MOBILE_HOME>\Mobile\Sdk\bin and the header files are located at <MOBILE_HOME>\Mobile\Sdk\include.

Note: For iOS clients:

Both SQLite and Berkeley DB clients are supported. The iOS sync APIs for SQLite client are implemented in libosync.a static library which you link with the application.

For Berkeley DB client, the static library name is libosync_bdb.a.

The iOS clients support native manual synchronization APIs (See Section 3.1.1.2, "OSE Synchronization APIs For Native Applications") as well as automatic synchronization APIs (See Section 3.2.1.2, "Native APIs for the Sync Agent and Automatic Synchronization").

Older 10.3 APIs (OCAPI) and Java APIs are not supported.

Note: Limitations of iOS client:

The iOS client runs within the user's application and can only access database files stored within the application sandbox. Also, like Android and Java SE clients (as mentioned below), syncagent only runs within the application process.

For Android and Blackberry platforms, all synchronization functionality is implemented in pure java. For Android, SQLite client library is osync_android.jar and Berkeley DB client is osync_bdb_android.jar, located in MDK under the following:

- <MOBILE_HOME>\Mobile\Sdk\android\lib
- <MOBILE_HOME>\Mobile\Sdk\android\lib\bdb.

Note: The Android Berkeley DB client also requires Berkeley DB jdbc driver (sqlite.jar) and Berkeley DB native library (liboracle-jdbc.so, located in MDK under <MOBILE_HOME>\Mobile\Sdk\android\lib\bdb\armeabi) and the Blackberry client library is in osync_rim.jar (in MDK under <MOBILE_HOME>\Mobile\Sdk\blackberry\lib.

The actual binaries installed on the device are osync_rim*.cod files under <MOBILE_HOME>\Mobile\Sdk\blackberry, but osync_rim.jar will be used for building mobile application).

Note: Automatic synchronization is not currently supported for Blackberry client.

The pure java client referred as Pure Java SE (PJ SE) runs on Win32 and Linux (and even Windows Mobile). The client library used for that is osync_se.jar (in MDK located in <ORACLE_HOME>\Mobile\Sdk\bin . Note the following:

- There are both SQLite and Berkeley DB PJ SE clients.
- The settings in ose.ini are used to configure whether SQLite or Berkeley DB client is used. See Section A.1.2 SQLite Mobile Client Parameters - SQLITE of the Mobile Client guide for more details on the parameters.
- Once these settings are set, they should not be changed.
- PJ SE client uses SQLite or Berkeley DB JDBC driver so there is a dependency on the JDBC driver jar file, JNI and other native libraries used to implement the JDBC driver.
- Ensure that both osync_se.jar and the jdbc driver are in CLASSPATH and dependent native libraries are in the PATH (LD_LIBRARY_PATH on linux).

Note: PJ SE client has the same design and limitations for automatic synchronization as PJ Android client, that is, the syncagent only runs within application process and not outside of it.

Versions of compilers and run time libraries are listed in the following table:

Table 3–1 Compiler and Run Time Library Versions

APIs Language	Compiler Version	Run Time Library Version
Java	Oracle JDK 1.6/1.7	Oracle JDK 1.6/1.7
.net	vs 2008	▪ vs 2008 and .NET Framework 2.0 (for win32) ▪ vs 2008 and .NET Compact Framework 3.5 (for windows mobile)
c and c++	vs 2008 /gcc 3.4.6	vs 2008

The following sections describe the mobile client synchronization APIs available to manage both manual and automatic synchronization programmatically within your application on the mobile client:

- Section 3.1, "Invoke Manual Synchronization on the Mobile Client"
- Section 3.2, "Manage Automatic Synchronization on the Mobile Client"

3.1 Invoke Manual Synchronization on the Mobile Client

Besides using a tool like msync, your client side application can do synchronzation programmatically. As described above, different sets of APIs are available for different platforms, but they do represent the same functionality. Manual synchronization APIs have a concept of sync session. Using the session you can provide necessary parameters, customize synchronization options, invoke synchronization and track its progress. Use the following APIs for invoking manual synchronization on your mobile clients:

- Section 3.1.1, "OSE Synchronization API for Applications on Mobile Clients"
- Section 3.1.2, "SQLite Synchronization API for .Net Clients"
- Section 3.1.3, "OCAPI Synchronization API for the Mobile Client"

3.1.1 OSE Synchronization API for Applications on Mobile Clients

OSE synchronization interfaces are available for pure Java clients, native clients and .Net clients. The following sections provide more details:

> **Note:** Use the OSE classes for all new application development for your mobile clients. These are the classes that are supported for the future.

- Section 3.1.1.1, "OSE Synchronization Java API"
- Section 3.1.1.2, "OSE Synchronization APIs For Native Applications"
- Section 3.1.1.3, "OSE .Net Synchronization API"
- Section 3.1.1.4, "OSE Synchronization JavaScript API for PhoneGap"

3.1.1.1 OSE Synchronization Java API

The following sections describe how you can use the OSE Java APIs to invoke synchronization:

> **Note:** For more details on these classes, refer to the *Oracle Database Mobile Server JavaDoc*.

- Section 3.1.1.1.1, "Overview"
- Section 3.1.1.1.2, "OSESession Class"
- Section 3.1.1.1.3, "OSEProgressListener Interface"
- Section 3.1.1.1.4, "Selective Synchronization"
- Section 3.1.1.1.5, "Custom Transport with the OSETransport Class"
- Section 3.1.1.1.6, "Sequences Emulated for SQLite Mobile Clients in Replicated Environment"
- Section 3.1.1.1.7, "OSEException Class"

3.1.1.1.1 Overview The Java interface for mobile client synchronization resides in the `oracle.opensync.ose` package.

The Java interface provides for the following functions:

- Setting (and optionally retrieving) client-side parameters such as user name, password and server URL
- Customizing syncronization with various runtime options
- Invoking sync
- Tracking sync progress

The following are the classes and interface for the Java API for mobile clients:

- `OSESession` Class
- `OSEProgressListener` Interface
- `OSETransport` Interface
- `OSEException` Class

3.1.1.1.2 OSESession Class `OSESession` enables setting synchronization parameters and options. This class exposes APIs to invoke and control synchronization by using the provided synchronization options.

In a multi-threaded environment, you cannot execute `OSESession` methods from multiple threads. Each thread should open its own session. The only exception is `cancelSync`, which can be executed by another thread.

> **Note:** Synchronization progress is reported through the `OSEProgressListener` interface, which is set by the `OSESession setProgress(OSEProgressListener)` method.

The parameters for the constructor are listed in Table 3–2.

Constructors

```
OSESession( )

OSESession( String user )

OSESession( String user, char[] pwd)
```

Table 3–2 OSESession Class Constructor

Parameter	Description
`user`	A string containing the name used for authentication by the mobile server.
`password`	A character array containing the user password.

Public Methods

The public methods and their parameters for the `OSESession` class are listed in Table 3–3:

Table 3–3 OSESession Class Public Method Parameters

Method	Description
`void cancelSync( )`	Attempts to cancel the synchronization process with a non-blocking call. If successful, throws `OSEException` with error code `OSEExceptionConstants.SYNC_CANCELED`.
`void close()`	Closes any active database connections that the session maintains. This method is called before application exits.
`void saveUser()` `String getUser()`	The `saveUser` method saves user information, such as users specific information, and the last synchronization user id. The `getUser` method retrieves current synchronization client name.
`void selectPub(String name)`	Provided the publication name, adds the publication to the list of publications to be synchronized selectively. See Section 3.1.1.1.4, "Selective Synchronization" for more information.
`void setAppRoot(String appRoot)` `String getAppRoot( )`	Sets or retrieves the current root directory, as set in the `DATA_DIRECTORY` parameter, for internal synchronization and database files for the application.

Table 3–3 (Cont.) OSESession Class Public Method Parameters

Method	Description
`boolean getBackground()` `void setBackground(boolean on)`	Sets or returns `TRUE` is a synchronization event is an automatic synchronization; `FALSE` if not.
`void setEncryptionType (int type)` `int getEncryptionType( )`	Sets or retrieves the current encryption type. Possible types can are as follows: ■ `ENC_AES` - AES encryption, which is the default. ■ `ENC_SSL` - SSL over HTTP. ■ `ENC_NONE` - No encryption.
`void setForceRefresh (boolean on)`	Set to wipe out all of the client data and replace it with server data, if true.
`boolean getForceRefresh( )`	Retrieves value of force refresh.
`void setSavePassword (boolean on)` `boolean getSavePassword()`	This is used to set and get the flag for persistently saving the user password. If true, the password is saved.
`void setNewPassword(char[] pwd)`	Allows clients to modify their password on the server. After a successful synchronization, the client's password on the server is changed to the new password.
`void setPassword(char[] pwd)`	Provide or modify the mobile client password.
`void setProgress (OSEProgressListener p)`	Set synchronization progress listener. For more details, see Section 3.1.1.1.3, "OSEProgressListener Interface".
`void setProxy(java.lang.String proxy)` `java.lang.String getProxy()`	Sets or returns the current HTTP proxy, which can be the hostname or IP address of the proxy server. NULL is returned if proxy is not used.
`void setSyncApps(boolean on)` `boolean getSyncApps()`	Sets or retrieves a flag that indicates whether the application client updates should be downloaded during the next synchronization. If set to `FALSE`, client updates are only uploaded to the server.
`setSyncDirection(int)` `getSyncDirection( )`	Sets or retrieves the current synchronization direction of data for the mobile client. You can indicate whether the client should perform normal synchronization with `DIR_SENDRECEIVE`, where data is both uploaded and downloaded. Alternatively, if you set the direction for data to be as follows: ■ `DIR_SENDRECEIVE`: Default. Sets the direction to send and receive. ■ `DIR_SEND`: Sets the direction to upload client data, but no server data is downloaded. ■ `DIR_RECEIVE`: Sets the direction to download data from the server, but no client data is uploaded. This direction setting affects only user data. All mobile server data, such as awknowledgements, will still be uploaded or downloaded as appropriate.

Table 3–3 (Cont.) OSESession Class Public Method Parameters

Method	Description
`void setSyncNewPub(boolean on)` `boolean getSyncNewPub( )`	Sets flag for enabling synchronization of new publications. By default, this is set to true and all publications are synchronized. However, if you set this to false, any new subscribed publications on the server are not downloaded to the client.
`int getSyncPriority()` `public void setSyncPriority(int prio) throws OSEException`	Sets or retrieves the synchronization priority. The default is `PRIO_DEFAULT`, which is OFF. Only high priority table or rows are synchronized when set to `PRIO_HIGH`. You can only use fast refresh with a high priority restricting predicate. If you use any other type of refresh, the high priority restricting predicate is ignored. See Section 1.2.10, "Priority-Based Replication" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for more information.
`void setTransportType(int type)` `int getTransportType( )`	Sets and retrieves the current transport type, which can be one of the following: ■ `TR_HTTP`: Default transport. ■ `TR_USER`: Custom transport.
`void setURL (java.lang.String url)` `java.lang.String getURL( )`	Sets or retrieves the HTTP URL of the mobile server.
`void setUseFiles(boolean on)` `boolean getUseFiles()`	Set flag to switch between using streaming or files to transport synchronization data. If set to true, synchronization stores uploaded and downloaded data in a file; otherwise, data is streamed. When using files, the `ose$in.bin` file contains the data received from the server. The `ose$out.bin` file contains the data sent to the server. These files are located in the *<mobileclient_root>*`\bin` directory on Win32, Windows Mobile and Linux platfoms or in the directory specified by the `DATA_DIRECTORY` on the Android or Blackberry platforms. Note: streaming requires that the underlying client transport stack implements HTTP 1.1. Thus, if a platform does not support streaming, `setUseFiles` must be congifigured as `TRUE`.
`void setUseResume(boolean on)` `boolean getUseResume()`	If `setUseResume` is set to TRUE, enables the resume feature, which attempts to resume sending and receiving data for a synchronization after a network failure. Requires that `setUseFiles` is also set to TRUE; otherwise, this method is ignored. The resume feature provides a more reliable transport for synchronizing data with minimal overhead.
`void setUserTransport(OSETransport t)`	Sets custom user-defined transport for synchronization, which you implement in the `OSETransport` interface. See Section 3.1.1.1.5, "Custom Transport with the OSETransport Class" for more details.

Table 3–3 (Cont.) OSESession Class Public Method Parameters

Method	Description
`void sync()`	Initiates a manual synchronization from within the application.
`void addProgressListener(OSEProgressListener pl)`	Add progress listener.
`void removeProgressListener(OSEProgressListener pl)`	Remove progress listener.
`boolean getEncryptDatabases()`	Get the current value of encrypt database flag.
`void setEncryptDatabases()`	Enable database encryption for new databases created during sync.
`public void shareConnection(String dbName, Object connObj`	dbName is database name. connObj is connection object and has to be a valid connection object for particular type of database as used in OSE plugin. For pure java sync client, it must be an instance of corresponding platform-specific database connection class: net.rim.device.api.database.Database for Blackberry client, android.database.sqlite.SQLiteDatabase for Android client, java.sql.Connection for SE client

Example

The following example sets the user name and password to `JOHN/john`. The mobile server URL is identified as `localhost:88`. And a synchronization is initiated with the `sync` method.

```
/* set up user name and password */
String user = "JOHN";
String pwd  = "john";

/* create OSESession with user John */
OSESession sess = new OSESession(user, pwd.toCharArray());

/* Identify Mobile Server URL */
sess.setURL("localhost:88");

/* Identify the progress listener, myProgressTracker */
sess.setProgress(myProgressTracker);

/* Initiate Sync */
sess.sync();
```

3.1.1.1.3 OSEProgressListener Interface

The `OSEProgressListener` interface enables progress updates to be trapped during synchronization.

Sync calls the progress function to report the current stage and the percent of completion of that stage. The parameters for the progress method are listed in Table 3–4:

Method

```
void progress (int stage, int val);
```

Table 3–4 OSEProgress Method Parameters

Parameter	Description
`stage`	This is set to one of the constants listed in Table 3–5.
`val`	This is the percentage of completion for specific `stage`.

The names of the constants which report the synchronization progress are listed in Table 3–5.

Table 3–5 OSEProgressListener Interface Constants

Constant Name	Progress Type
`PREPARE`	States that the synchronization engine is preparing local data to be sent to the server. This includes getting locally modified data. For streaming implementations this takes a shorter amount of time.
`SEND`	States that the synchronization engine is sending data to the network.
`RECEIVE`	States that the synchronization engine is receiving data from the server.
`PROCESS`	States that the synchronization engine is applying the newly received data from the server to the local data stores.
`IDLE`	States that the synchronization engine has completed the synchronization process.
`COMPOSE`	Not supported yet.
`APPLY`	Not supported yet.

Example

This simple class implements the `OSEProgressListener`.

```
class myProgressTracker implements OSEProgressListener

{
  public void progress
     (int state,
     int val)
    {
      System.out.println( "Status: "+state+"="+ val+"%" );
     } //progress
 }
```

3.1.1.1.4 Selective Synchronization Selective sync specifies whether a publication should be synchronized or not for the next session. Set the flag with the `selectPub` method to indicate whether the publication is to be synchronized on the next execution of the `sync` method. The default setting is `NULL` for all publications.

> **Note:** Automatic synchronization selectively synchronizes only publications that contain automatic publication items.

Table 3–6 lists the name and description of parameter for the `selectPub` method.

Table 3–6 selectPub Parameters

Name	Description
`publication_name`	The name of the publication which is being synchronized. If the value for the `publication_name` is `NULL`, it means all publications in the database, which turns off selective sync. For more information, see Section 2.4, "Creating Publications Using Oracle Database Mobile Server APIs".

3.1.1.1.5 Custom Transport with the OSETransport Class You can implement a custom user-defined transport for synchronization with the `OSETransport` interface. Implement the following methods to connect and disconnect the connection and to open the input and output streams. These methods are used for the transport when you provide your implementation in the `setUserTransport` method in the `OSESession` class.

Table 3–7 OSETransport methods

Method	Description
`void connect()`	Open transport connection.
`void disconnect()`	Closes transport connection and releases its resources.
`java.io.OutputStream openOutputStream()`	Opens output stream.
`java.io.InputStream openInputStream()`	Opens input stream.

3.1.1.1.6 Sequences Emulated for SQLite Mobile Clients in Replicated Environment SQLite only supports sequences in a replicated environment. Sequences partitioned per client are useful for generating unique, non-overlapping values to avoid data conflicts.

> **Note:** For details on using sequences in Berkeley DB, see Section 2.4.1.8, "Creating Client-Side Sequences for the Downloaded Snapshot".

In order to emulate sequence behavior on SQLite, the Sync Client replicates the partitioned sequence information from the server. The mobile client applications can generate unique column values from the partitioned sequence information, which is replicated for every SQLite database in the `C$SEQ_CLIENTS` table.

The `C$SEQ_CLIENTS` table has the following definition:

```
TABLE C$SEQ_CLIENTS(
  "NAME" VARCHAR2(30) NOT NULL, -- sequence name
  "CUR_VALUE" NUMBER(10,0),  -- current value, set to NULL on the first sync
  "MAX_VALUE" NUMBER(10,0) NOT NULL,  -- max value
  "MIN_VALUE" NUMBER(10,0) NOT NULL,  -- min value
  "PUB_NAME" VARCHAR2(30) NOT NULL,  -- publication name (for internal use)
  "DB_NAME" VARCHAR2(30),  -- database name (for internal use)
  "INCREMENT_BY" NUMBER(10,0) NOT NULL,  -- increment
  PRIMARY KEY ("NAME"))
```

To select the next sequence value, users can perform the following:

1. Select `CUR_VALUE` or `MIN_VALUE` if `CUR_VALUE` is `NULL`.
2. Add `INCREMENT_BY` to the `CUR_VALUE`.

3. Update the CUR_VALUE with the CUR_VALUE+INCREMENT_BY for the sequence.

Example 3–1 Emulating Sequences on the Mobile Client

The following Java code example demonstrates this functionality:

```
 /* Advance sequence value using sequence properties replicated from the
 * server using C$SEQ_CLIENTS table
 * @param seq sequence name
 * @return int next sequence value
 * @throws DatabaseException if sequence is not found.
 */
int advanceSequence(String seq) throws Exception
{
    int seqNextVal = 0;
    Statement statement = _db.createStatement(
        "SELECT ifnull((cur_value+increment_by),min_value)
            FROM "+SEQUENCES_TABLE+" WHERE name = ?");
    statement.prepare();
    statement.bind(1, seq);
    Cursor cursor = statement.getCursor();
    if(cursor.next())
    {
        Row row = cursor.getRow();
        seqNextVal = row.getInteger(0);
    } else {
        throw new Exception("Sequence not found: " + seq);
    }
    cursor.close();
    statement.close();
    statement = _db.createStatement(
        "UPDATE "+SEQUENCES_TABLE+" SET cur_value =
             ifnull((cur_value+increment_by),min_value) WHERE name = ?");
    statement.prepare();
    statement.bind(1, seq);
    statement.execute();
    statement.close();

    return seqNextVal;
}
```

3.1.1.1.7 OSEException Class This class signals a non-recoverable error during the synchronization process. The `OSEException` class constructs a `clear` object. The parameters for the constructor are listed in Table 3–8, " OSEException Constructor Parameter Description":

Constructors

`OSEException(int errCode)`

`OSEException(int errCode,Object arg)`

`OSEException(int errCode,Object arg, Throwable cause)`

`OSEException(int errCode, Object arg1, Object arg2)`

`OSEException(int errCode, Object arg1, Object arg2, Object arg3)`

`OSEException(int errCode, Object [] args, Throwable cause)`

Table 3–8 OSEException Constructor Parameter Description

Parameter	Description
`errorCode`	Error codes are provided within the OSEExceptionConstants class. Error codes for synchronization are provided in the OSEExceptionConstants class. Some OSEException instances are thrown from OSE APIs. Others are used as causes of the synchronization error messages. The message handler returns an error message. For a complete list of the error messages that can be thrown in OSEException, see "*Exception Error Codes and Messages*" in the Oracle Database Mobile Server Message Reference.
`arg, args, arg1, arg2, arg3`	Return variables for information within the error message.
`cause`	The cause of this throwable or NULL if the cause is nonexistent or unknown.

Public Methods

OSEException class extends BaseException.

The methods for getting error code, cause and message are listed in Table 3–9, " BaseException Class Public Methods".

Table 3–9 BaseException Class Public Methods

Method	Description
`getErrorCode()`	Returns exception error code
`getMessage()`	Returns exception message
`getCause()`	Returns exception cause
`toString()`	Returns string representation of this BaseException instance. The string contains information from the whole chain of causes if present (cause of this exception is also often instance of BaseException). For each exception in the chain, it contains error code, error message and additional diagnostic information if present.

For a complete list of the error messages that can be thrown in the `OSEException`, see "Exception Error Codes and Messages" in the *Oracle Database Mobile Server Message Reference*.

3.1.1.2 OSE Synchronization APIs For Native Applications

You can initiate and monitor synchronization from a native client application. The OSE synchronization methods for the native interface are defined in `ose.h`, which is located in `<ORACLE_HOME>\Mobile\Sdk\include`, and implemented in `ose.dll`, which is located in the `<ORACLE_HOME>\Mobile\Sdk\bin` directory.

The following sections describe how to set up and initiate synchronization:

- Section 3.1.1.2.1, "Overview of Native Synchronization API"
- Section 3.1.1.2.2, "Initializing the Environment With oseOpenSession"
- Section 3.1.1.2.3, "Setting Session Options"

3.1.1.2.1 Overview of Native Synchronization API For starting synchronization, the application should perform the following:

1. Invoke the `oseOpenSession` method to initialize the session and its environment and resources.
2. Set any session options. Invoke the `oseSaveUser` method to preserve the last user, URL, proxy and optionally the user password for future synchronization events.
3. Invoke `oseSync` method to synchronize, which returns after the synchronization completes, an error occurs, or the user interrupts the process.

   Synchronization progress is reported through the `oseProgressFunc` interface, which is set by the `oseSetProgress` method.
4. If synchronization failed, use the `oseGetLastError` method to retrieve the error message.
5. When done with the session, invoke the `oseCloseSession` method to close the session and release its resources.

3.1.1.2.2 Initializing the Environment With oseOpenSession The `oseOpenSession` method initializes the synchronization environment—which is passed to each subsequent call with the `oseSess` handle.

In a multi-threaded environment, you cannot concurrently use a session from multiple threads, even with the same user. Instead, each thread should open its own session with the `oseOpenSession` method. The only exception is `oseCancelSync`, which can be executed by another thread.

Note: Every time you invoke the `oseOpenSession` method, you must also clean up with `oseCloseSession` method. These methods should always be called in pairs. See Section 3.1.1.2.9, "Close the Synchronization Environment Using oseCloseSession" for more information.

Syntax

```
oseError oseOpenSession(const char *user, const char *pwd, oseSess *sess);
```

Table 3–10 lists the oseOpenSession parameters.

Table 3–10 oseOpenSession Parameters

Name	Description
user	User name for synchronization. If NULL, the last saved user name is provided. If the last user name was not saved, the OSE_ERR_USER_NOT_SPECIFIED error is returned.
pwd	User password. If NULL, the last saved password for this user is provided if it was previously saved. Alternatively, the password can be provided later if the OSE_OPT_PASSWORD option is specified. If, at the time of sync, the password is not provided and was not previously saved, the OSE_ERR_PWD_NOT_SPECIFIED error is returned.
sess	Pointer to a session handle into which the new session is returned. This handle cannot be NULL.

This call initializes the oseSess synchronization environment handle—which holds context information for the synchronization engine—and restores any session options that were saved with the last oseSaveUser method invocation. See Section 3.1.1.2.4, "Saving User Settings With oseSaveUser" for more information on oseSaveUser.

If successful, zero is returned; otherwise, an OSE error code is returned.

3.1.1.2.3 Setting Session Options You can set certain session options explicitly with the set session methods. Every session option, except OSE_OPT_NEW_PASSWORD, is set for the duration of the session, unless it is explicitly reset with the appropriate set session option method.

When the session is created, the initial value for each option is loaded from the following:

- The ose.ini file.
- If it is not set in the ose.ini file, the option value is loaded from the saved options in the internal OSE Meta files. The OSE Meta files save session information that was either set on the previous synchronization or set explicitly with the appropriate set session option method. New values for options set by the user, such as OSE_OPT_URL or OSE_OPT_PROXY, are saved in the internal OSE Meta files during next synchronization.
- If not set in the ose.ini file or saved as within the OSE Meta files, the default value is taken for each option.

Options are separated into boolean, numeric, and string options:

- Boolean options are those options that can only be set to OSE_TRUE or OSE_FALSE.
- Numeric options are set to an integer value.
- String options are those options that are defined with a character string.

The following sections describe the session options and the methods that can get or set the values for these options:

- Boolean and Numeric Session Options
- String Session Options

Boolean and Numeric Session Options

Use the `oseSetNumOption` and `oseGetNumOption` methods to set and get the boolean and numeric session options.

oseSetNumOption

```
oseError oseSetNumOption(oseSess sess, int opt, long val);
```

Table 3–11 lists the `oseSetNumOption` parameters.

Returns zero if the option is set successfully. An OSE error code is returned if an invalid option code or an invalid value is specified.

Table 3–11 oseSetNumOption Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`int opt`	Option code. See Table 3–13 for a list of all of the options that can bet set by this method.
`long val`	The option value to set for the session. See Table 3–13 for potential values for this option.

oseGetNumOption

```
oseError oseGetNumOption(oseSess sess, int opt, long *val);
```

Table 3–12 lists the `oseGetNumOption` parameters.

Returns zero if the option is retrieved successfully. An OSE error code is returned if an invalid option code is specified.

Table 3–12 oseGetNumOption Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`int opt`	Option code. See Table 3–13 for a list of all of the options that can bet retrieved by this method.
`long *val`	Pointer to a variable into which to return the option value. The pointer cannot be `NULL`. See Table 3–13 for potential values for this option.

Table 3–13 lists all boolean and numeric synchronization options and potential values. For all boolean options, the value can only be either `OSE_TRUE` and `OSE_FALSE`.

Table 3–13 Numeric and Boolean Session Options

Session Option	Description
OSE_OPT_SYNC_DIRECTION	Specifies the synchronization direction with one of the following numeric constants: ■ Bidirectional sync (default): OSE_SYNC_DIR_SENDRECEIVE, value 0. ■ Data is only sent, but not received, OSE_SYNC_DIR_SEND, value 1. ■ Data is only received, but not sent: OSE_SYNC_DIR_RECEIVE, value 2.
OSE_OPT_ENCRYPTION_TYPE	Specifies the encryption type, which indicates how the data is encrypted when transfered over the network. ■ AES encryption: OSE_ENC_TYPE_AES, value 0. ■ HTTPS protocol over secure sockets: OSE_ENC_TYPE_SSL, value 1. ■ No encryption: OSE_ENC_TYPE_NONE, value 2.
OSE_OPT_TRANSPORT_TYPE	Transport type designates the protocol used to transfer data to and from the mobile server. ■ Data is transfered using the HTTP protocol: OSE_TR_TYPE_HTTP, value 0. ■ Data is transfered by the custom transport provided by the application: OSE_TR_TYPE_USER, value 1. ■ Data is transfered manually using File-Based Sync): OSE_TR_TYPE_FILE, value 2.
OSE_OPT_TRANSPORT_DIRECTION	Transport direction. This is used to indicate connected or disconnected (that is, File-Based) sync. ■ Connected synchronization (default). This can be used with either bidirectional or unidirectional synchronization. If synchronization direction is set to OSE_SYNC_DIR_SEND, it specifies send-only synchronization over connected transport, where the data is sent and an acknowledgement is received. OSE_TR_DIR_SENDRECEIVE, value 0. ■ Disconnected synchronization, send-only synchronization. Data is sent without acknowledgement. This can only be used with the OSE_TR_TYPE_FILE transport type. OSE_TR_DIR_SEND, value 1. ■ Disconnected synchronization, receive-only synchronization. This can only be used with the OSE_TR_TYPE_FILE transport type. OSE_TR_DIR_RECEIVE, value 2.
OSE_OPT_SAVE_PASSWORD	Boolean option. Indicates whether the user synchronization password should be saved on the client, which means that it will not need to be explicitly provided for future sessions. The password is saved in an encrypted form.
OSE_OPT_BACKGROUND	Boolean option. If OSE_OPT_BACKGROUND is set to true, synchronization is more performant and consists solely of DML operations, with no schema updates. OSE_FALSE is the default, indicating normal synchronization.

Table 3–13 (Cont.) Numeric and Boolean Session Options

Session Option	Description
OSE_OPT_SYNC_PRIO	Synchronization priority constants: ■ High priority: OSE_PRIO_HIGH, value 0. ■ Normal priority: OSE_PRIO_DEFAULT, value 1.
OSE_OPT_SYNC_APPS	Boolean option. Indicates whether synchronization should download a list of application and client updates that can be installed later. OSE_TRUE is the default. Also, see the OSE_OPT_HAS_SOFT_UPDATES option.
OSE_OPT_SYNC_NEW_PUB	Boolean option. Indicates whether new publications can be created during synchronization. OSE_TRUE is the default.
OSE_OPT_FORCE_REFRESH	Boolean option. Indicates if the synchronization is force-refresh. Ignores client changes and reloads all client data from the mobile server. OSE_FALSE is the default.
OSE_OPT_USE_FILES	Boolean option. Indicates whether files are used to temporarily store the data either before it is sent or after it is received. If enabled, client changes are first saved into a file named oseOutFile.bin (default), then sent to the mobile server. The received data is saved to another file named oseInFile.bin (default) and then read and transferred to the database from that file. This is used for the resume transport or for protocols where the total data size to be sent needs to be known in advance, such as HTTP 1.0. OSE_FALSE is the default. The related ose.ini parameter is OSE.FILES. Also, see the OSE_OPT_RESUME_TRANSPORT option.
OSE_OPT_RESUME_TRANSPORT	Boolean option. Indicates whether the resume protocol should be used on top of the synchronization transport. To use this option, the OSE_OPT_USE_FILES option must be enabled. If not already set, OSE_OPT_USE_FILES will be set implicitly. The resume protocol is typically used for lengthy synchronization sessions over unstable network connections. It resumes sending and receiving data from the point of a network disconnect, thus avoiding the restart of synchronization from scratch. This option is used only with connected transport (OSE_TR_DIR_SENDRECEIVE). OSE_FALSE is the default. Related ose.ini parameter is OSE.RESUME.
OSE_OPT_HAS_SOFT_UPDATES	Boolean read-only option. You can only retrieve the value with the oseGetNumOption method. This option indicates whether any updates are available for the client after the last synchronization. This is used by the mSync tool to specify whether it should launch the update utility to retrieve updates. Also, see the OSE_OPT_SYNC_APPS option.

Table 3–13 (Cont.) Numeric and Boolean Session Options

Session Option	Description
OSE_OPT_ENCRYPT_DATABASES	Boolean option. Indicates whether any databases newly created during synchronization should be encrypted. The encryption key for each database is either retrieved from the synchronization keystore or generated based on the user password, if not found in the keystore. Applications can define their own keys for each database before database creation with the keystore APIs. OSE_FALSE is the default. The related ose.ini parameter is OSE.ENCRYPTDB. You can set, get, and remove the encryption key with the oseSetDBKey, oseGetDBKey, and oseRemoveDBKey methods, which are described in Section 3.1.1.2.13, "Set and Retrieve Data Encryption Keys".

String Session Options

Use the oseSetStrOption and oseGetStrOption methods to set and get the string session options. Every string option, except the OSE_OPT_APP_ROOT option, defaults to NULL if not loaded during session initialization.

oseSetStrOption

```
oseError oseSetStrOption(oseSess sess, int opt, const char *val);
```

Table 3–14 lists the oseSetStrOption parameters.

Returns zero if the option is set successfully. An OSE error code is returned if an invalid option code or an invalid value is specified.

Table 3–14 oseSetStrOption Parameters

Name	Description
oseSess sess	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns OSE_ERR_INVALID_SESS if the handle is NULL.
int opt	Option code. See Table 3–16 for a list of all of the options that can bet set by this method.
const char *val	The option value to set for the session. See Table 3–16 for potential values for this option.

oseGetStrOption and oseGetStrOptionNC

```
oseError oseGetStrOption(oseSess sess, int opt, char *val, int bufSize);
oseError oseGetStrOptionNC(oseSess sess, int opt, const char **val);
```

The difference between the two calls is that the oseGetStrOptionNC does not require you to allocate the buffer for the returned string. Instead, a pointer to a string is passed. The string is NULL-terminated, which is how the length is known.

Table 3–15 lists the oseGetStrOption parameters.

Returns zero if the option is set successfully. OSE_ERR_INVALID_STR_OPT error is returned if an invalid option code is specified. OSE_ERR_INVALID_BUFFER error is returned if the buffer provided is too small to store the value and the terminating NULL character.

Table 3–15 oseGetStrOption Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`int opt`	Option code. See Table 3–16 for a list of all of the options that can bet retrieved by this method.
`char *val`	For `oseGetStrOption`: Pointer to a buffer into which to return the option value. The pointer cannot be `NULL`.
`const char **val`	For `oseGetStrOptionNC`: Pointer to a character pointer into which return the value. The pointer cannot be `NULL`. The memory for the option value is stored within the OSE engine and is valid until next call to `oseGetStrOptionNC`.
`int bufSize`	Size of the buffer.

Table 3–16 lists all string session options and potential values.

Table 3–16 Session String Options

Option	Option Description
`OSE_OPT_URL`	Mobile server URL.
`OSE_OPT_PROXY`	HTTP proxy, if present, as `host:port` or `host`, where the port defaults to 80.
`OSE_OPT_NEW_PASSWORD`	New password provided to modify the synchronization password during the next synchronization. This option needs to be set each time the password needs to be changed for the session.
`OSE_OPT_USER_NAME`	Read-only option, used to retrieve the synchronization user name.
`OSE_OPT_PASSWORD`	Used to set the synchronization password in the current session. This overwrites the password that was originally provided to the `oseOpenSession` method or retrieved from the OSE configuration files. This is only used to set the synchronization password in the case that a `NULL` password was passed to the `oseOpenSession` method. This password is saved in the OSE Meta files if the `OSE_OPT_SAVE_PASSWORD` option was enabled.
`OSE_OPT_APP_ROOT`	Root directory for internal synchronization files. By default, it is the synchronization client installation `bin` directory.
`OSE_OPT_FILE_URL`	File URL used for a file-based sync. Specifies the path to the file, which can optionally prefixed by `file://`.

3.1.1.2.4 Saving User Settings With oseSaveUser The `oseSaveUser` method saves the last user, URL, proxy and optionally the user password for future synchronization events into an OSE Meta file. However, the password is saved only if the `OSE_OPT_SAVE_PASSWORD` option is enabled. These settings are normally saved at the end of each synchronization, if changes are detected. These settings can be used for the current session or used by the `oseOpenSession` method to initialize the environment when next invoked.

Returns zero if successful and `OSE_ERR_INTERNAL_ERROR` if an input/output error occured during saving.

Syntax

```
oseError oseSaveUser(oseSess sess);
```

Table 3–17 lists the `oseSaveUser` parameters.

Returns zero if successful. Returns the `OSE_ERR_INTERNAL_ERROR` error if an input/output error occurs when saving the settings.

Table 3–17 oseSaveUser Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.

This saves or overwrites the user settings into an OSE Meta file on the client side.

> **Note:** See Section 3.1.1.2.3, "Setting Session Options" for details on the user settings saved.

3.1.1.2.5 Start the Synchronization With the oseSync Method Starts the synchronization process synchronously.

Returns zero if synchronization is successful. Returns `OSE_ERR_INVALID_SESS` if session handle was invalid and `OSE_ERR_SYNC_CANCELED` if synchronization was canceled from another thread by the `oseCancelSync` method. Other OSE error codes are returned if the synchronization fails.

Syntax

```
oseError oseSync(oseSess sess);
```

Table 3–18 lists the `oseSync` parameters.

Table 3–18 oseSync Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.

3.1.1.2.6 Manage What Publications Are Synchronized With oseSelectPub Selective sync allows only certain publications to be synchronized. The `oseSelectPub` method selects a publication for selective sync, which specifies if it is to be synchronized on the next synchronization. Selective sync only works if you have first performed at least one synchronization for the client. Then, a selective sync for the publication occurs at the next invocation of the `oseSync` method.

The application can select publications needed by repeatedly calling the `oseSelectPub` method. To revert to the regular (non-selective) synchronization selection, invoke this method with `NULL` as the publication name.

The default setting is for all publications to be synchronized.

Syntax

```
oseError oseSelectPub(oseSess sess, const char *pub);
```

Table 3–19 lists the oseSelectPub parameters.

Returns zero if successful. Returns the OSE_ERR_PUB_NOT_FOUND error if the publication with the provided name was not found.

Table 3–19 oseSelectPub Parameters

Name	Description
oseSess sess	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns OSE_ERR_INVALID_SESS if the handle is NULL.
pub	Publication name to select for the next synchronization. NULL deselects all publications and reverts back to a non-selective synchronization.

3.1.1.2.7 See Progress of Synchronization with Progress Listening You can implement a progress callback function that is called so that you can be notified of the progress of the synchronization operations. If you create and set the progress callback method, the mobile client invokes this callback method at appropriate times while the oseSync method is executing.

The following is the declaration of the progress callback function.

```
oseError (* oseProgressFunc)(void *ctx, int stage, int val);
```

When the oseOpenSession invokes the method you declare in oseSetProgress, it provides the following information as input to your method:

- ctx—A pointer to user-defined state information for the callback. This handle is a pointer to a user-defined structure. Since the oseProgressFunc callback method is user-implemented, it should know how to process the user-defined structure. The user context structure may or may not be initialized by the application before calling the oseSetProgress method.
- stage—The stage in the synchronization process, which is one of the following values, where these values are defined in ose.h:

Table 3–20 Description of the Stage Values

Stage Value	Value	Description
OSE_SYNC_STATE_IDLE	0	No synchronization occuring.
OSE_SYNC_STATE_PREPARE	1	Preparing data into temporary files. This state only occurs if the OSE_OPT_USE_FILES option is enabled.
OSE_SYNC_STATE_SEND	2	Sending the data to the mobile server.
OSE_SYNC_STATE_RECEIVE	3	Receiving data from the mobile server.
OSE_SYNC_STATE_PROCESS	4	Processing data from temporary file. This state only occurs if the OSE_OPT_USE_FILES option is enabled.

- val—The percentage completed in the particular stage that synchronization is in from 0 to 100.

After you define the progress callback function, you can initialize it with the mobile client by executing the oseSetProgress method:

```
oseError oseSetProgress(oseSess sess, void *ctx, oseProgressFunc pf);
```

You can unregister the progress callback function by executing `oseSetProgress` method as follows:

```
oseSetProgress(oseSess, NULL, NULL);
```

Table 3–21 lists the `oseSetProgress` parameters.

Table 3–21 oseSetProgress Parameters

Name	Description
`sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`ctx`	A pointer to the session environment settings. This handle is a pointer to a user-defined structure. Since the `oseProgressFunc` callback method is user-implemented, it should know how to process the user-defined structure. The user context structure may or may not be initialized by the application before calling the `oseSetProgress` method.
`pf`	The progress callback function.

3.1.1.2.8 Cancel a synchronization event using oseCancelSync Cancels the synchronization operation from another thread and returns immediately without waiting for the synchronization operation to abort.

Syntax

```
oseError oseCancelSync(oseSess sess);
```

Table 3–22 lists the `oseCancelSync` parameters.

Returns zero, if successful. Returns `OSE_ERR_INVALID_SESS` if the `oseSess` session handle is invalid.

Table 3–22 oseCancelSync Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.

3.1.1.2.9 Close the Synchronization Environment Using oseCloseSession Clears and performs a cleanup of the synchronization environment and resources. This function must be invoked for every `oseOpenSession`, even if `oseSync` has not been executed even once.

Returns zero, if successful. Returns `OSE_ERR_INVALID_SESS` if the `oseSess` session handle is invalid.

Syntax

```
oseError oseCloseSession(oseSess sess);
```

Table 3–23 lists the `oseCloseSession` parameters.

Table 3–23 oseCloseSession Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.

3.1.1.2.10 Retrieve Synchronization Error Information with oseGetLastError Retrieves the extended synchronization error message and code from the last call. This information contains the last OSE error info as well as any internal errors that caused the error.

Syntax

```
oseError oseGetLastError(oseSess sess, const oseErrorDesc **errDesc);
```

Table 3–24 lists the `oseGetLastError` parameters.

Returns zero if successful. Returns the `OSE_ERR_INVALID_SESS` error if the session handle was invalid. Returns the `OSE_ERR_INTERNAL_ERROR` error if a system error has occured.

Table 3–24 oseGetLastError Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. This handle can be `NULL` if trying to retrieve error information from a failed `oseOpenSession` call.
`oseErrorDesc errDesc`	A pointer to an `oseErrorDesc` pointer into which the `oseErrorDesc` pointer is returned. The pointer cannot be `NULL`. The structures referenced by this pointer are only valid until the next OSE call.

The `oseErrorDesc` structure is defined as follows:

```
typedef struct _oseErrorDesc {
  oseError code;               /* error code */
  const char *type;            /* a string describing the type of error */
  const char *msg;             /* error message */
  struct _oseErrorDesc *cause; /* underlying cause, if present */
} oseErrorDesc;
```

If the error has an underlying cause, the `oseErrorDesc.cause` points to another `oseErrorDesc` structure, which in turn can have its own cause, and so on. This is useful if the OSE call returns an `OSE_ERR_INTERNAL_ERROR`, which can be further defined within another `oseErrorDesc` structure. If the cause is not present, `oseErrorDesc.cause` is `NULL`.

> **Note:** The memory for `oseErrorDesc` structure is allocated within the mobile client and should not be freed by the application.

3.1.1.2.11 Enable File-Based Synchronization through Native APIs When you want to use file-based synchronization, you must enable file-based synchronization. Once enabled, then when you initiate manual synchronization, then the synchronization file is created. See Section 5.10, "Synchronizing to a File Using File-Base Sync" in the *Oracle Database Mobile Server Administration and Deployment Guide* for more details on file-based synchronization.

To enable file-based synchronization programmatically, perform the following:

1. Ensure that any previous settings of the File-Based Sync properties are set to `NULL`.
2. Initialize the session with the `oseOpenSession` method providing the user name and password for the user that is initializing the synchronization.
3. Specify File-Based Sync by setting `OSE_OPT_TRANSPORT_TYPE` to `OSE_TR_TYPE_FILE`.
4. Specify the synchronization direction in the `OSE_OPT_TRANSPORT_DIRECTION` option as follows:
   - `OSE_TR_DIR_SEND`: Send, which creates the synchronization file.
   - `OSE_TR_DIR_RECEIVE`: Receive, which takes in a file from the mobile server.
5. Set the `OSE_OPT_FILE_URL` to the path and filename of the file.
   - If sending, the path and filename is where the mobile client saves the uploaded data for the mobile server. This file is created with the mobile client transactions destined for the mobile server.
   - If receiving, the path and filename where the data file that was received from the mobile server. This file is loaded and processed within the mobile client.

3.1.1.2.12 Share the Database Connection Provides a database connection handle from the application to use in the OSE engine, instead of the OSE engine opening its own database connection, which is the default. The connection handle is set for the duration of the session unless explicitly unset by the same call with a `NULL` connection handle value.

Applications can keep open cursors while invoking synchronization. If the connection is not shared, the OSE engine needs to create its own connection and start a new exclusive transaction; however, SQLite does not support creating exclusive transactions when another connection has open cursors.

Syntax

```
oseError oseShareConnection(oseSess sess, const char *db, void *connHdl);
```

Table 3–25 lists the `oseShareConnection` parameters.

Returns zero, if successful. Returns `OSE_ERR_PLUGIN_ERROR` if a plugin error has occured, the value of which can be retrieved with the `oseGetLastError` method.

Table 3–25 oseShareConnection Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`const char *db`	Database name for which the connection handle is provided.
`void *connHdl`	Valid database connection handle for a particular type of database that is used in the OSE plugin. For example, for SQLite and Berkeley DB, the connection handle should be of type `sqlite3 *`. A `NULL` value will unshare the connection and the OSE engine opens its own database connection.

3.1.1.2.13 Set and Retrieve Data Encryption Keys

An application can provide its own custom key to encrypt each database instead of using a key generated from the synchronization password. Subsequently, the key in the key store is used by the mobile client to open the database after it is created.

The `oseSetDBKey` method sets a database encryption key provided by the application in the OSE key store for the database. Applications need to re-execute this call when they re-encrypt the database with a different key, so that OSE engine has the current key to access the database during synchronization.

For more details, see the description for the `OSE_OPT_ENCRYPT_DATABASES` option in Table 3–13.

Syntax

```
oseError oseSetDBKey(oseSess sess, const char *db,
       const void *key, oseSize keyLen);
```

Table 3–26 lists the `oseSetDBKey` parameters.

Returns zero, if successful. Returns the `OSE_ERR_INTERNAL_ERROR` error if the internal error occured in the key store.

Table 3–26 oseSetDBKey Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`db`	Database name for which to set the key.
`key`	Buffer with the provided key.
`keyLen`	Length of the provided key.

The `oseGetDBKey` method retrieves the database encryption key from the OSE key store. For more details, see the description for the `OSE_OPT_ENCRYPT_DATABASES` option in Table 3–13.

Syntax

```
oseError oseGetDBKey(oseSess sess, const char *db, void *buf,
     oseSize bufSize, oseSize *retLen);
```

Table 3–27 lists the `oseGetDBKey` parameters.

Returns zero, if successful. Returns the `OSE_ERR_INVALID_BUFFER` error if the buffer was too small to store the key. Returns the `OSE_ERR_INTERNAL_ERROR` error if the internal error occured in the key store.

Table 3–27 oseGetDBKey Parameters

Name	Description
`oseSess sess`	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns `OSE_ERR_INVALID_SESS` if the handle is `NULL`.
`db`	Database name for which to retrieve the key.
`buf`	Buffer to store the key.
`bufSize`	Size of the buffer that stores the key.

Table 3–27 (Cont.) oseGetDBKey Parameters

Name	Description
retLen	Actual length of the key. The length will be zero if the key is not found.

The oseRemoveDBKey method removes the database encryption key from the OSE key store. For more details, see the description for the OSE_OPT_ENCRYPT_DATABASES option in Table 3–13.

Syntax

```
oseError oseRemoveDBKey(oseSess sess, const char *db);
```

Table 3–28 lists the oseRemoveDBKey parameters.

Returns zero, if successful. Returns the OSE_ERR_INTERNAL_ERROR error if the internal error occured in the key store.

Table 3–28 oseRemoveDBKey Parameters

Name	Description
oseSess sess	Session pointer to a session handle, which contains the synchronization environment across all calls for this session. Returns OSE_ERR_INVALID_SESS if the handle is NULL.
db	Database name from which the key is removed.

3.1.1.2.14 Accessing Mobile Client Configuration Parameters

Mobile Client configuration parameters are stored in ose.ini configuration file. For more information about this configuration file, see the Section A.1, "*OSE.INI File Overview*" of the "*Mobile Client Guide*".

On some platforms such as Win32, WinCE and Linux, this file can be read and modified using ascii editor. However on platforms like iOS, where all Mobile Client-related data, including ose.ini file, is stored within an application sandbox, user cannot access this file.

In addition, on iOS platform, device management is not currently supported, so ose.ini configuration parameters are not donwloaded from the mobile server.

Hence, two APIs are provided to read and modify ose.ini configuration parameters: oseSetParam and oseGetParamNC.

For more information on the supported ose.ini parameters, see Mobile Client Guide, Appendix A.

Syntax

```
OSE_API oseError oseSetParam(const char *cat, const char *name, const char *val)
```

Table 3–29 lists the oseSetParam parameters.

Set initialization parameter in ose.ini (or ose.txt) configuration file. This is a generic routine to set parameter for any sync component based on component name (category) and parameter name.

> **Note:** The new parameter value will only take effect when new OSE session is opened.

Returns zero if successful. Returns OSE_ERR_INTENRAL_ERROR error code if ose.ini file could not be modified or saved.

Table 3–29 oseSetParam Prameters

Name	Description
`cat`	Parameter category (for example, OSE, SQLITE, BGSYNC, NETWORK)
`name`	Parameter name (for example, RESUME, DATA_DIRECTORY, etc.)
`val`	Parameter value represented as string. If the given parameter already exists in ose.ini, its value will be overwritten by val.

Syntax

```
OSE_API oseError oseGetParamNC(const char *cat, const char *name, const char
**val);
```

Table 3–30 lists the `oseGetParamNC` parameters.

Get initialization parameter value from ose.ini (or ose.txt) configuration file. This is a generic routine to get parameter value for any sync component based on component name (category) and parameter name. The parameter value is retrieved without copying. The memory for the parameter value is stored within OSE engine and is valid until the next call to oseGetParamNC.

Returns zero if successful. Returns OSE_ERR_INTENRAL_ERROR error code if ose.ini file could not be read.

Table 3–30 oseGetParamNC Prameters

Name	Description
`cat`	Parameter category (for example, OSE, SQLITE, BGSYNC, NETWORK)
`name`	Parameter name (for example, RESUME, DATA_DIRECTORY, etc.)
`val`	Pointer to a character pointer into which the return value cannot be NULL. If a given parameter does not exist in ose.ini, pointer pointed to by val will be set to NULL.

Example

- The following code:

```
oseError e;
e = oseSetParam("SQLITE", "DATA_DIRECTORY", "tmp/sqlite_db");
```

will set the following in ose.ini:

```
SQLITE.DATA_DIRECTORY=tmp/sqlite_db
```

- The following code:

```
oseError e;
e = oseSetParam("OSE", "RESUME", "YES");
```

will set the following in ose.ini:

```
OSE.RESUME=YES
```

- The following code:

```
const char *v;
oseError e;
e = oseGetParamNC("SQLITE", "DATA_DIRECTORY", &v);
```

will retrieve the value of SQLITE.DATA_DIRECTORY parameter into the variable v.

3.1.1.3 OSE .Net Synchronization API

The following sections describe how you can use the OSE .Net Synchronization APIs to build your own client synchronization initiation on mobile clients:

> **Note:** OCAPI Synchronization APIs for C# are no longer supported, use the APIs described in this section instead.

3.1.1.3.1 Overview The .Net interface for mobile client synchronization resides in the `Oracle.OpenSync.OSE` namespace and is implemented in `Oracle.OpenSync.dll`.

The .Net interface provides for the following functions:

- Setting client-side user profiles containing data.
- Starting the synchronization process.
- Tracking the progress of the synchronization process.

3.1.1.3.2 Enumerations Used by OSESession Table 3–31 lists all enumerations and values that can be used by `OSESession` APIs.

Table 3–31 Enumerations

Enumeration	Members
`SyncDirection`	Specifies the synchronization direction as follows: ■ `SyncDirection.SendReceive`: Bidirectional sync (default). Value 0. ■ `SyncDirection.Send`: Data is only sent, but not received. Value 1. ■ `SyncDirection.Receive`: Data is only received, but not sent. Value 2.
`EncryptionType`	Specifies the encryption type, which indicates how the data is encrypted when transfered over the network. ■ `EncryptionType.AES`: AES encryption. Value 0. ■ `EncryptionType.SSL`: HTTPS protocol over secure sockets. Value 1. ■ `EncryptionType.None`: No encryption. Value 2.

Table 3–31 (Cont.) Enumerations

Enumeration	Members
`TransportType`	Transport type designates the protocol used to transfer data to and from the mobile server. ▪ `TransportType.Http`: Data is transfered using the HTTP protocol. Value 0. ▪ `TransportType.User`: Data is transfered by the custom transport provided by the application. Value 1. ▪ `TransportType.File`: Data is transfered manually using File-Based Sync). Value 2.
`DataPriority`	Synchronization priority: ▪ `DataPriority.High`: High priority. Value 0. ▪ `DataPriority.Normal`: Normal priority. Value 1.
`SyncProgressStage`	Synchronization stage: ▪ `SyncProgressStage.Prepare`: States that the synchronization engine is preparing local data to be sent to the server in a local file. This includes getting locally modified data. For streaming implementations this takes a shorter amount of time. Only used if `OSESession.UseFiles` is true. ▪ `SyncProgressStage.Send`: States that the synchronization engine is sending data to the network. ▪ `SyncProgressStage.Receive`: States that the synchronization engine is receiving data from the server. ▪ `SyncProgressStage.Process`: States that the synchronization engine is applying the newly received data from the server to the local data stores. Only used if `OSESession.UseFiles` is true. ▪ `SyncProgressStage.Idle`: States that the synchronization engine has completed the synchronization process.

3.1.1.3.3 OSESession Class

`OSESession` enables setting synchronization parameters and options. This class exposes APIs to invoke and control synchronization by using the provided synchronization options.

In a multi-threaded environment, you cannot execute `OSESession` methods from multiple threads. Each thread should open its own session. The only exception is `CancelSync`, which can be executed by another thread.

The parameters for the constructor are listed in Table 3–32.

Constructors

```
OSESession( )

OSESession( String user )

OSESession( String user, char[] pwd)
```

Table 3–32 OSESession Class Constructor

Parameter	Description
`user`	A string containing the name used for authentication by the mobile server.
`password`	A character array containing the user password.

Public Methods

The public methods and their parameters for the `OSESession` class are listed in Table 3–33:

Table 3–33 OSESession Class Public Method Parameters

Method	Description
`CancelSync( )`	Attempts to cancel the synchronization process with a non-blocking call.
`Close()`	Closes any active database connections that the session maintains. This method is called before application exits.
`SaveUser()`	The `SaveUser` method saves session options and user information.
`SelectPub(String name)` `UnselectPubs()`	The `SelectPub` method provides the publication name, adds the publication to the list of publications to be synchronized selectively. The `UnselectPubs` method reverts back to normal synchronization for all publications. See Section 3.1.1.3.6, "Selective Synchronization" for more information.
`event OSEProgressHandler SyncProgress`	Set synchronization progress listener with the `SyncProgress` event. For more details, see Section 3.1.1.3.5, "OSEProgressHandler Interface".
`Sync()`	Initiates a manual synchronization from within the application.
`ShareConnection(string dbName, IntPtr hdl)` `UnshareConnection(string dbName)`	Provides database connection handle from the application to use in the OSE engine, instead of engine opening its own database connection, which is default. The connection handle is set for the duration of the session unless explicitly unset by calling `UnshareConnection()`. Parameters include the database name and a connection handle, which has to be a valid native handle for the particular type of database that is used in the OSE plugin. For example, SQLite and Berkeley DB handle should be of native type `sqlite3 *`.

OSESession Class Properties

The following documents the properties for the `OSESession` class.

- Read-write properties read or write the value of a field with `get` and `set` accessors.
- Read-only properties read the value of a field with the `get` accessor.
- Write-only properties set the value of a field with the `set` accessor.

Boolean properties get or set the value of a field to either true or false.

Table 3–34 lists all properties. Table 3–31 lists all enumerations.

Table 3–34 OSESession Properties

Property	Accessors	Description
`SyncDirection`	Read-write	Gets or sets the synchronization direction on the `SyncDirection` enumeration. The default is `SyncDirection.SendReceive`.
`EncryptionType`	Read-write	Gets or sets the encryption type on the `EncryptionType` enumeration. The default for which is `EncryptionType.AES`.
`TransportType`	Read-write	Gets or sets the transport type on the `TransportType` enumeration. The default for which is `TransportType.Http`.
`SyncPriority`	Read-write	Gets or sets the synchronization priority on the `DataPriority` enumeration. The default for which is `DataPriority.Normal`.
`SavePassword`	Read-write	Boolean property. The default is `FALSE`. Indicates whether the user synchronization password should be saved on the client, which means that it will not need to be explicitly provided for future sessions. The password is saved in an encrypted form.
`Background`	Read-write	Boolean property. Indicates whether the next synchronization should be an automatic synchronization. The default is `FALSE`, indicating manual synchronization.
`SyncApps`	Read-write	Boolean property. Indicates whether synchronization should download a list of application and client updates that can be installed later. The default is `TRUE`.
`SyncNewPub`	Read-write	Boolean property. Indicates whether new publications can be created during synchronization. The default is `TRUE`.
`ForceRefresh`	Read-write	Boolean property. Indicates if the synchronization is force-refresh. Ignores client changes and reloads all client data from the mobile server. The default is `FALSE`.
`UseFiles`	Read-write	Boolean property. Indicates whether files are used to temporarily store the data either before it is sent or after it is received. If enabled, client changes are first saved into a file named `oseOutFile.bin` (default), then sent to the mobile server. The received data is saved to another file named `oseInFile.bin` (default) and then read and transferred to the database from that file. This is used for the resume transport or for protocols where the total data size to be sent needs to be known in advance, such as HTTP 1.0. The default is `FALSE`. The related `ose.ini` parameter is `OSE.FILES`.

Table 3–34 (Cont.) OSESession Properties

Property	Accessors	Description
`UseResume`	Read-write	Boolean property. Indicates whether the resume protocol should be used on top of the synchronization transport. To use this option, the `UseFiles` property must be enabled. If not already set, `UseFiles` will be set implicitly. The resume protocol is typically used for lengthy synchronization sessions over unstable network connections. It resumes sending and receiving data from the point of a network disconnect, thus avoiding the restart of synchronization from scratch. This property is used only with connected transport (`SyncDirection.SendReceive`). The default is `FALSE`. Related `ose.ini` parameter is `OSE.RESUME`.
`URL`	Read-write	Mobile server URL property.
`Proxy`	Read-write	HTTP proxy property, if present, as `host:port` or `host`, where the port defaults to 80 if not specified.
`NewPassword`	Write-only	New password provided to modify the synchronization password during the next synchronization. This option needs to be set each time the password needs to be changed for the session.
`User`	Read-only	Read-only property, used to retrieve the synchronization user name.
`Password`	Write-only	Used to set the synchronization password in the current session. This overwrites the password that was originally provided to the `oseOpenSession` method or retrieved from the OSE configuration files. This is only used to set the synchronization password in the case that a `NULL` password was passed to the `oseOpenSession` method. This password is saved in the OSE Meta files if the `SavePassword` property was enabled.
`AppRoot`	Read-write	Root directory for internal synchronization files. By default, it is the synchronization client installation `bin` directory.

Example

```
/* Create the OSESession object */
oseSess = new OSESession(user, pwd != NULL ?
   pwd.ToCharArray() : NULL);

/* Assign Session options */
if (url != NULL)
  oseSess.URL = url;
if (f)
  oseSess.UseFiles = true;
if (sp)
  oseSess.SavePassword = true;
if (bResume)
  oseSess.UseResume = true;
if (ne)
  oseSess.EncryptionType = EncryptionType.None;
else if (ssl)
  oseSess.EncryptionType = EncryptionType.SSL;
```

```
/* Save the session options */
if (!ns)
  oseSess.SaveUser();

/* Assign a progress handler named syncProgress */
oseSess.SyncProgress += new OSEProgressHandler(syncProgress);

/* Perform the synchronization */
oseSess.Sync();
```

3.1.1.3.4 OSEProgressEventArgs Properties

The following documents the properties for the `OSEProgressEventArgs` class, which are used primarily in the `OSEProgressHandler` interface.

- Read-write properties read or write the value of a field with `get` and `set` accessors.
- Read-only properties read the value of a field with the `get` accessor.
- Write-only properties set the value of a field with the `set` accessor.

Table 3–35 lists all properties.

Table 3–35 OSEProgressEventArgs Properties

Property	Accessors	Description
`Stage`	Read-only	Gets the synchronization stage from the `SyncProgressStage` enumeration. The default for which is `SyncProgressStage.Idle`.
`Value`	Read-only	Gets the percentage of completion for a specific `Stage`.

3.1.1.3.5 OSEProgressHandler Interface The `OSEProgressHandler` interface enables progress updates to be trapped during synchronization.

Sync calls the a registered progress handler to report the current stage and the percent of completion of that stage, which can be retrieve with the `Stage` and `Value` properties, as described in Table 3–34.

Syntax

```
public delegate void OSEProgressHandler(object sender,
     OSEProgressEventArgs args);
```

Table 3–36 OSEProgressHandler Parameters

Name	Description
`object sender`	Sender of the event, which is set to the `OSESession` object to which the handler is attached.
`OSEProgressEventArgs args`	Progress event arguments that include `Stage` and `Value` properties, which are described in Table 3–35.

You can implement your own progress handler by providing the same parameter declaration as in the `OSEProgressHandler` declaration. The following implements a progress handler called `syncProgress`, which takes in a sender and an `OSEProgressEventArgs` structure. It evaluates the `Stage` and the `Value` of the `OSEProgressEventArgs` structure when invoked by the Sync Agent.

```
static void syncProgress(object sender, OSEProgressEventArgs args)
       {
           if (args.Stage == SyncProgressStage.Idle)
               return;

           if (prevVal == -1)
           {
               Console.Write(args.Stage);
               prevVal = 0;
           }

           int dots = (args.Value - prevVal) / 2;
           for(int i = 0; i < dots; i++)
               Console.Write('.');

           if (args.Value == 100)
           {
               Console.WriteLine();
               prevVal = -1;
           }
           else
               prevVal = args.Value;
       }
```

Then, you set the progress handler by setting the `SyncProgress` event, as shown below:

```
oseSess.SyncProgress += new OSEProgressHandler(syncProgress);
```

3.1.1.3.6 Selective Synchronization Selective sync specifies whether a publication should be synchronized or not for the next session. Provide the name of the publication with the `SelectPub` method to indicate whether the publication is to be synchronized on the next execution of the `Sync` method. The default setting is `NULL` for all publications.

> **Note:** Automatic synchronization selectively synchronizes only publications that contain automatic publication items.

Table 3–37 lists the name and description of parameter for the `SelectPub` method.

Table 3–37 SelectPub Parameters

Name	Description
`name`	The name of the publication which is being synchronized. If the value for the `name` is `NULL`, it means all publications in the database, which turns off selective sync. You can also turn off selective sync with the `UnselectPubs` method. For more information, see Section 2.4, "Creating Publications Using Oracle Database Mobile Server APIs".

3.1.1.3.7 OSEException Class This class signals an error during the synchronization process.

The `OSEException` read-only properties are listed in Table 3–38.

Table 3–38 OSEException Properties

Parameters	Description
`ErrorCode`	Gets the exception error code. The error code can be one of the `OSEErrorCode` enumeration values, which are documented in the "OSEException Error Messages" in the *Oracle Database Mobile Server Message Reference*.
`Kind`	Gets the exception type name.
`InnerException`	Use the `InnerException` property to get underlying cause of the `OSEException`.

Constructors

```
OSEException(int errorCode)

OSEException(int errorCode, string errorMessage)
```

Table 3–39 OSEException Parameter Description

Parameter	Description
`errorCode`	The error code can be one of the `OSEErrorCode` enumeration values, which are documented in the "Exception Error Codes and Messages" in the *Oracle Database Mobile Server Message Reference*.
`errorMessage`	A readable text message that provides extra information.

For a complete list of the error messages in the `OSEException` class, see "Exception Error Codes and Messages" in the *Oracle Database Mobile Server Message Reference*.

3.1.1.4 OSE Synchronization JavaScript API for PhoneGap

The following sections describe how you can use the PhoneGap:

- Section 3.1.1.4.1, "Overview"
- Section 3.1.1.4.2, "OSESession Class"

3.1.1.4.1 Overview

PhoneGap framework allows applications written in HTML/JavaScript to run on various mobile device platforms and access certain native APIs. The mobile client synchronization PhoneGap plugin enables existing or new HTML/JavaScript applications to synchronize data with the Database Mobile Server. Among several APIs provided by PhoneGap, there is a Storage API that can be used to access the synchronized data stored in the local SQLite database. For more details, see `http://docs.phonegap.com/en/2.5.0/cordova_storage_storage.md.html#Storage`

In addition, there is an Android sqlitePlugin which allows similar SQLite specific data access. For more details, see `https://github.com/brodyspark/PhoneGap-sqlitePlugin-Android`

The OSE JavaScript interface provides for the following functions:

- Setting client-side user profiles containing data such as user name, password, and server.
- Starting the synchronization process.
- Tracking the progress of the synchronization process.

For Android clients, the underlying native PhoneGap plugin implementation creates wrappers on top of the existing Java OSE API and is modeled after it. It has the follwing two jar files:

- phonegap_sync_android.jar
- osync_android.jar

It also has a JavaScript file osync.js.

The file locations are:

- <MOBILE_HOME>\Mobile\Sdk\android\phonegap\libs\phonegap_sync_android.jar
- <MOBILE_HOME>\Mobile\Sdk\android\lib\osync_android.jar
- <MOBILE_HOME>\Mobile\Sdk\android\phonegap\assets\www\osync.js

The sync API is incapsulated in OSESession JavaScript class. The background sync API is exposed in BGSession class. For an example of a sync PhoneGap application for Android, please see the demo shipped with MDK under:

- <MOBILE_HOME>\Mobile\Sdk\samples\sync\android\phonegap

3.1.1.4.2 OSESession Class OSESession enables setting synchronization parameters and options. This class exposes JavaScript API to invoke and control synchronization by using the provided synchronization options.

Note: There should only be a single OSESession instance per PhoneGap application.

The OSESession Class Constructors are as follows:

Constructors

```
OSESession (success_cbk, error_cbk)

OSESession (success_cbk, error_cbk, user)

OSESession( success_cbk, error_cbk, user, pwd)
```

Table 3–40 lists all the parameters for the constructor.

Table 3–40 OSESession Class Constructor

Parameter	Description
`success_cbk`	A callback function that gets invoked upon successful sync completion.
`error_cbk`	An error callback function that gets invoked on any error in OSE API.
`user`	A string containing the name used for authentication by the mobile server.
`password`	A string containing the user password.

Public Methods

The public methods and their parameters for the OSESession class are listed in Table 3–41:

Table 3–41 OSESession Class Public Method Parameters

Parameter	Description
`cancelSync( )`	Attempts to cancel the synchronization process with a non-blocking call. If successful, sync calls error callback with "Sync Canceled" error message.
`close()`	Closes any active database connections that the session maintains. This method is called before application exits or before a new session needs to be created (for example, when a user id is changed).
`saveUser()` `String getUser()`	The saveUser method saves user information, such as users specific information, and the last synchronization user id. The getUser method retrieves current synchronization client name.
`selectPub(name)`	Provided the publication name, adds the publication to the list of publications to be synchronized selectively. See Section 3.1.1.1.4, "Selective Synchronization" for more information.
`setAppRoot(appRoot)` `String getAppRoot( )`	Sets or retrieves the current root directory, as set in the DATA_DIRECTORY parameter, for internal synchronization and database files for the application.
`boolean getBackground()` `setBackground(boolean on)`	Sets or returns **TRUE** if a synchronization event is an automatic synchronization; **FALSE** if not.
`setEncryptionType (int type)` `int getEncryptionType( )`	Sets or retrieves the current encryption type. Possible types can are as follows: ■ ENC_AES - AES encryption, which is the default. ■ ENC_SSL - SSL over HTTP. ■ ENC_NONE - No encryption.
`setForceRefresh (boolean on)`	Set to wipe out all of the client data and replace it with server data, if true.
`boolean getForceRefresh( )`	Retrieves value of force refresh.
`setSavePassword (boolean on)` `boolean getSavePassword()`	This is used to set and get the flag for persistently saving the user password. If true, the password is saved.
`setNewPassword(pwd)`	Allows clients to modify their password on the server. After a successful synchronization, the client's password on the server is changed to the new password.
`setPassword(pwd)`	Provide or modify the mobile client password.
`setProgress (boolean on)`	Sets whether or not native sync progress listener is displayed.
`setProxy(proxy)` `String getProxy()`	Sets or returns the current HTTP proxy, which can be the hostname or IP address of the proxy server. NULL is returned if proxy is not used.
`setSyncApps(boolean on)` `boolean getSyncApps()`	Sets or retrieves a flag that indicates whether the application client updates should be downloaded during the next synchronization. If set to **FALSE**, client updates are only uploaded to the server.

Table 3–41 (Cont.) OSESession Class Public Method Parameters

Parameter	Description
`setSyncDirection(int dir)` `getSyncDirection( )`	Sets or retrieves the current synchronization direction of data for the mobile client. You can indicate whether the client should perform normal synchronization with DIR_SENDRECEIVE, where data is both uploaded and downloaded. Alternatively, if you set the direction for data to be as follows: ■ DIR_SENDRECEIVE: Default. Sets the direction to send and receive. ■ DIR_SEND: Sets the direction to upload client data, but no server data is downloaded. ■ DIR_RECEIVE: Sets the direction to download data from the server, but no client data is uploaded. (Reserved for future use). This direction setting affects only the user data. All mobile server data, such as acknowledgements, will still be uploaded or downloaded as appropriate.
`setSyncNewPub(boolean on)` `boolean getSyncNewPub( )`	Sets or retrieves flag for enabling synchronization of new publications. By default, this is set to true and all publications are synchronized. However, if you set this to false, any new subscribed publications on the server are not downloaded to the client.
`int getSyncPriority()` `setSyncPriority(int prio)`	Sets or retrieves the synchronization priority. The default is PRIO_DEFAULT, which is OFF. Only high priority table or rows are synchronized when set to PRIO_HIGH. You can only use fast refresh with a high priority restricting predicate. If you use any other type of refresh, the high priority restricting predicate is ignored. For more information, see Section 1.2.10, "*Priority-Based Replication*" in the Oracle Database Mobile Server *Troubleshooting and Tuning Guide*.
`setURL (url)` `String getURL( )`	Sets or retrieves the HTTP URL of the mobile server.
`setUseFiles(boolean on)` `boolean getUseFiles()`	Sets or retrieves flag to switch between using streaming or files to transport synchronization data. If set to true, synchronization stores uploaded and downloaded data in a file; otherwise, data is streamed. When using files, the ose$in.bin file contains the data received from the server. The ose$out.bin file contains the data sent to the server. These files are located in the <mobileclient_root>\bin directory on Win32, Windows Mobile and Linux platfoms or in the directory specified by the DATA_DIRECTORY on the Android or Blackberry platforms. Note: streaming requires that the underlying client transport stack implements HTTP 1.1. Thus, if a platform does not support streaming, setUseFiles must be configured as TRUE.
`setUseResume(boolean on)` `boolean getUseResume()`	If setUseResume is set to **TRUE**, enables the resume feature, which attempts to resume sending and receiving data for a synchronization after a network failure. Requires that setUseFiles is also set to **TRUE**; otherwise, this method is ignored. The resume feature provides a more reliable transport for synchronizing data with minimal overhead.
`sync ()`	Initiates a manual synchronization from within the application.

Table 3–41 (Cont.) OSESession Class Public Method Parameters

Parameter	Description
`boolean getEncryptDatabases( )`	Get the current value of encrypt database flag.
`setEncryptDatabases( boolean on)`	Enable database encryption for new databases created during sync.

Example

The following example sets the user name and password to JOHN/john. The mobile server URL is identified as localhost:88. And a synchronization is initiated with the sync method.

```
//* set up user name and password */
var user = "JOHN";
var pwd = "john";
//PhoneGap error callback.  Prints error message on console
function onError(msg) {
 console.log("Encountered Error: " + msg);
};
//PhoneGap success callback.  Prints success on console
function onSuccess() {
 console.log("Success");
};

/* create OSESession with user John */
OSESession sess = new OSESession(onSuccess, onError, user, pwd);

/* Identify Mobile Server URL */
sess.setURL("localhost:88");

/* Enable progress monitor GUI */
sess.setProgress(true);

/* Initiate Sync */
sess.sync();
```

3.1.2 SQLite Synchronization API for .Net Clients

You can initiate and monitor synchronization from an .Net provider application. See the SQLite documentation for information on the .Net provider.

3.1.3 OCAPI Synchronization API for the Mobile Client

> **Note:** The following OCAPI synchronization APIs are currently supported for the mobile client, but are not the direction recommended for future applications. To develop applications for future support, migrate existing applications to use the Section 3.1.1, "OSE Synchronization API for Applications on Mobile Clients".

The following sections describe the OCAPI synchronization APIs:

- Section 3.1.3.1, "OCAPI Synchronization APIs For C or C++ Applications"
- Section 3.1.3.2, "mSync, OCAPI, and mSyncCom API"

3.1.3.1 OCAPI Synchronization APIs For C or C++ Applications

You can initiate and monitor synchronization from a C or C++ client application. The OCAPI synchronization methods for the C/C++ interface are contained in `ocapi.h` and `ocapi.dll`, which are located in the `<MOBILE_HOME>\Mobile\Sdk\include` directory and `<MOBILE_HOME>\Mobile\Sdk\Bin` directory respectively.

A C++ example is provided in the <MOBILE_HOME>`\Mobile\Sdk\Samples\sync\win32\msync` directory. You can follow the instruction in ReadMe.txt to build the executable- SimpleSync.exe.

The following sections describe how to set up and initiate synchronization:

3.1.3.1.1 Overview of C/C++ Synchronization API

For starting synchronization, the application should perform the following:

1. Create, memset, and initialize the `ocEnv` structure.
2. Invoke the `ocSessionInit()` method.
3. Set any optional fields in the `ocEnv` structure, such as user name and password. If you want to preserve all optional fields set in the `ocEnv` structure for future synchronization sessions, then execute the `ocSaveUserInfo` method.
4. Optionally, you can set proxy information with the `ocSetSyncOption` method or specify the synchronization type for each table with the `ocSetTableSyncFlag` function.
5. Invoke the `ocDoSynchronize()` method, which returns after the synchronization completes, an error occurs, or the user interrupts the process. While executing, the `ocDoSynchronize` function invokes any callback function set in the `ocEnv.fnProgress` field. The callback function must not call any blocking functions, as this process is not reentrant or threaded.
6. Once synchronization completes, then invoke the `ocSessionTerm()` method to clear the `ocEnv` data structure.

7. If synchronization failed, then use the `ocGetLastError` function to retrieve the error message.

For an example, see the `SimpleSync.cpp` sample code.

3.1.3.1.2 Initializing the Environment With ocSessionInit The `ocSessionInit` function initializes the synchronization environment—which is contained in the `ocEnv` structure or was created with `ocSaveUserInfo`. For more information, see Section 3.1.3.1.7, "Managing User Settings With ocSaveUserInfo".

> **Note:** Every time you invoke the `ocSessionInit` function, you must also clean up with `ocSessionTerm`. These functions should always be called in pairs. See Section 3.1.3.1.12, "Clear the Synchronization Environment Using ocSessionTerm" for more information.

Syntax

```
int ocSessionInit( ocEnv env );
```

Table 3–42 lists the `ocSessioninit` parameter and its description.

Table 3–42 ocSessionInit Parameters

Name	Description
`env`	An `ocEnv` class, which contains the synchronization environment.

This call initializes the `ocEnv` structure—which holds context information for the synchronization engine—and restores any user settings that were saved in the last `ocSaveUserInfo` call, such as user name and password (See Section 3.1.3.1.7, "Managing User Settings With ocSaveUserInfo"). An `ocEnv` structure is passed as the input parameter. Perform the following to prepare the `ocEnv` variable:

1. Create the `ocEnv` by allocating a variable the size of `ocEnv`.
2. Memset the `ocEnv` variable before invoking the `ocSessionInit` function. If you do not perform a `memset` on the `ocEnv` variable, then the `ocSessionInit` function will not perform correctly.
3. Set all required fields in the `ocEnv` structure before passing it to `ocSessionInit`. If you want to save the user preferences for future sessions, then invoke the `ocSaveUserInfo` method.

For a full description of `ocEnv`, see Section 3.1.3.1.4, "ocEnv Data Structure".

The following example allocates a new `ocEnv`, which is then passed into the `ocSessionInit` call.

```
env = new ocEnv;
// Reset ocenv
memset( env, 0, sizeof(ocEnv) );

// init OCAPI
ocError rc = ocSessionInit(env);
```

3.1.3.1.3 Managing the C/C++ Data Structures Two data structures—ocEnv Data Structure and ocTransportEnv Data Structure—are used for certain functions in the Mobile Sync API.

3.1.3.1.4 ocEnv Data Structure The `ocEnv` data structure holds internal memory buffers and state information. Before using this structure, the application initializes it by passing it to the `ocSessionInit` method.

Table 3–43 lists the field name, type, usage, and corresponding description of the `ocEnv` structure parameters.

- Required—If the usage is required, then you either set before calling the `ocSessionInit` function or you have saved these parameters previously with the `ocSaveUserInfo` function.
- Optional—If the usage is optional, then optionally set after calling the `ocSessionInit` function and before the `ocDoSynchronize` function.
- Read Only.

Table 3–43 ocEnv Structure Field Parameters

Field	Type	Usage	Description
`username`	`char[32]`	Required.	Name of the user to authenticate. This name is limited to 28 characters, because of other parts of the product.
`password`	`char[32]`	Required.	User password (clear text). This name is limited to 28 characters, because of other parts of the product.
`trType`	`Enum`	Required.	If set to `OC_BUILDIN_HTTP`, use HTTP built-in transport driver. This is the default. If set to `OC_USER_METHOD`, use user provided transport functions. If set to `OC_FILE_TRANSPORT`, the synchronization uses file-based sync. See Section 3.1.3.1.14, "Enable File-Based Synchronization through C or C++ APIs".
`newPassword`	`char[32]`	Optional.	If first character of this string is not NULL—in otherwords (char) 0—this string is sent to the server to change the user password; the password change is effective on the next synchronization session.
`savePassword`	`Short`	Optional.	If set to 1, the password is saved locally and is loaded the next time `ocSessionInit` is called.
`appRoot`	`char[32]`	Optional.	Directory to where the application will be copied. If first character is NULL, then it uses the default directory.
`priority`	`Short`	Optional.	0= OFF (default) 1= ON; Only high priority table or rows are synchronized when turned on. You can only use fast refresh with a high priority restricting predicate. If you use any other type of refresh, the high priority restricting predicate is ignored. See Section 1.2.10, "Priority-Based Replication" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for more information.
`secure`	`Short`	Optional.	If set to 0, then AES is used on the transport. If set to `OC_SSL_ENCRYPTION`, use SSL synchronization (SSL-enabled device only).

Table 3–43 (Cont.) ocEnv Structure Field Parameters

Field	Type	Usage	Description
syncDirection	Enum	Optional.	If set to 0 (OC_SENDRECEIVE), then synchronization is bi-directional (default). If set to OC_SENDONLY, then push changes only to the server. This stops the synchronization after the local changes are collected and sent. User must write own transport method (like floppy bases) when using this method. If set to OC_RECEIVEONLY, then send no changes and only receive update from server. This only performs the receive and allow changes function to local database stages.
exError	ocError	Read-only.	Extended error code - either OS or OKAPI error code.
transportEnv	ocTransportEnv		Transport buffer. See Section 3.1.3.1.5, "ocTransportEnv Data Structure".
progressProc	fnProgress	Optional.	If not NULL, points to the callback for progress listening. See Section 3.1.3.1.11, "See Progress of Synchronization with Progress Listening".
totalSendDataLen	Long		Reserved
totalRecieveDataLen	Long		Reserved
userContext	Void*	Optional.	Can be set to anything by the caller for context information (such as progress dialog handle, renderer object pointer, and so on.
ocContext	Void*		Reserved.
logged	Short		Reserved.
bufferSize	Long		Reserved (for Wireless/Nettech only).
pushOnly	Short	Optional.	If set to 1, then only push changes to the server.
syncApps	Short	Optional.	Set to 1 (by default), performs application deployment. If set to 0, then no applications will be received from the server.
syncNewPublications	Short	Optional.	If set to 1 (default), receives any new publication created from the server since last synchronization. If set to 0, only synchronizes existing publications (useful for slow transports like wireless).

Table 3–43 (Cont.) ocEnv Structure Field Parameters

Field	Type	Usage	Description
updateLog	Short	Optional.	Debug only. If set to 1, logs server-side insert and update row information to the publication client database.
options	Short	Optional.	Debug only. A bitset of the following flags: ■ OCAPI_OPT_SENDMETADATA Sends meta-info to the server. ■ or OCAPI_OPT_DEBUG Enables debugging messages. ■ OCAPI_OPT_DEBUG_F Saves all bytes sent and received for debugging. ■ OCAPI_OPT_NOCOMP Disables compression. ■ OCAPI_OPT_ABORT If set, OCAPI will try to abort the current synchronization session. ■ OCAPI_OPT_FULLREFRESH Forces OCAPI to purge all existing data and do a full refresh.
cancel	Short		Caller can set to 1 on next operation. ocDoSynchronize returns with -9032.

The environment structure contains fields that the caller can update to change the way Mobile Sync module works. The following example demonstrates how to set the fields within the ocEnv structure.

```
typedef struct ocEnv_s {
 // User info
char username[MAX_USERNAME];    // Mobile Sync Client id, limited to 28 characters
char password[MAX_USERNAME];    // Mobile Sync Client password for
                                // authentication during sync, limited to 28 chars
char newPassword[MAX_USERNAME]; // resetting Mobile Sync Client password
                                   // on server side if this field is not blank
short savePassword;           // if set to 1, save password
char appRoot[MAX_PATHNAME];     // dir path on client device for deploying files
short priority;              // High priority table only or not
short secure;            // if set to 1, data encrypted over the wire
enum {
OC_SENDRECEIVE = 0,     // full step of synchronize
OC_SENDONLY,     // send phase only
OC_RECEIVEONLY,     // receive phase only
OC_SENDTOFILE,     // send into local file | pdb
OC_RECEIVEFROMFILE     // receive from local file | pdb
}syncDirection;     // synchronize direction

enum {
OC_BUILDIN_HTTP = 0,     // Use build-in HTTP transport method
OC_USER_METHOD     // Use user defined transport method
}trType;           // type of transport

ocError exError;     // extra error code
```

```
ocTransportEnv transportEnv;     // transport control information

                        // GUI related function entry
progressProc fnProgress;     // callback to track progress; this is optional

                 // Values used for Progress Bar. If 0, progress bar won't show.
long totalSendDataLen; // set by Mobile Sync API informing transport total number
                      // of bytes to send; set before the first fnSend() is called
long totalReceiveDataLen;     // to be set by transport informing Mobile Sync API
                        // total number of bytes to receive;
                        // should be set at first fnReceive() call.
void* userContext;      // user defined context
void* ocContext;        // internal use only
short logged;           // internal use only
long bufferSize;        // send/receive buffer size, default is 0
short pushOnly;         // Push only flag
short syncApps;         // Application deployment flag
short cancel;           // cancel
} ocEnv;
```

3.1.3.1.5 ocTransportEnv Data Structure You can configure the HTTP URL, proxy, proxy port number and other HTTP-specific transport definitions in the `ocTrHttp` structure. This structure is an HTTP public structure defined in `octrhttp.h`.

You access the `ocTrHttp` structure from within the `ocTransportEnv` data structure, which is provided as part of the `ocEnv` data structure. The following demonstrates the fields within the `ocTransportEnv` structure:

```
typedef struct ocTransportEnv_s {
void* ocTrInfo;            // transport internal context
```

The `ocTrInfo` is a pointer that points to the HTTP parameters in the `ocTrHttp` structure. The following code example retrieves the `ocTrInfo` pointer to the HTTP parameters and then modifies the URL, proxy, and proxy port number to the input arguments:

```
ocTrHttp* http_params = (ocTrHttp*)(env->transportEnv.ocTrInfo);
// set server_name
strcpy(http_params->url, argv[3]);
// set proxy
strcpy(http_params->proxy, argv[4]);
// set proxy port
http_params->proxyPort = atoi(argv[5])
```

3.1.3.1.6 Retrieving Publication Information With ocGetPublication This function gets the publication name on the client from the application name. The user knows only the application name, which happens when the Packaging Wizard is used to package an application before publishing it. If the application needs the publication name in order to interact with the database, then this function is used to retrieve that name, given the application name.

Syntax

```
ocError ocGetPublication(ocEnv* env, const char* application_name,
 char* buf, int buf_len);
```

The parameters for the `ocGetPublication` function are listed in Table 3–44 below.

Table 3–44 ocGetPublication Parameters

Name	Description
`ocEnv* env`	Pointer to an `ocEnv` structure buffer to hold the return synchronization environment.
`const char* application_name(in)`	The name of the application.
`char* buf(out)`	The buffer where the publication name is returned.
`int buf_len(in)`	The buffer length, which must be at least 32 bytes.

Return value of 0 indicates that the function has been executed successfully. Any other value is an error code.

The following code example demonstrates how to get the publication name.

```
void sync()
{
         ocEnv env;
         int rc;

         // Clean up ocenv
         memset(&env 0, sizeof(env) );

         // init OCAPI
         rc = ocSessionInit(&env);

         strcpy(env.username, "john");
         strcpy(env.password, "john");

         // We use transportEnv as HTTP paramters
         ocTrHttp* http_params = (ocTrHttp*)(env.transportEnv.ocTrInfo);
         strcpy(http_params->url, "your_host");

         // Do not synchronize applicaton "Sample3"
         char buf[32];
         rc = ocGetPublication(&env, "Sample3", buf, sizeof(buf));
         rc = ocSetTableSyncFlag(&env, buf, NULL, 0);

         // call sync
         rc = ocDoSynchronize(&env);
         if (rc < 0)
                fprintf(stderr, "ocDoSynchronize failed with %d:%d\n",
                  rc, env.exError);
         else
                printf("Synchronization compeleted\n");

         // close OCAPI session
         rc = ocSessionTerm(&env);
         return 0;
}
```

3.1.3.1.7 Managing User Settings With ocSaveUserInfo Saves user settings for the `ocEnv` structure. These settings can be used for the current session or used by the `ocSessionInit` function to initialize the environment when next invoked.

Syntax

```
int ocSaveUserInfo( ocEnv *env );
```

Table 3–45 lists the `ocSaveUserInfo` parameter and its description.

Table 3–45 ocSaveUserInfo Parameters

Name	Description
`env`	Pointer to the synchronization environment.

This saves or overwrites the user settings into a file or database on the client side. The following information provided in the environment structure is saved:

> **Note:** See Section 3.1.3.1.4, "ocEnv Data Structure" or Section 3.1.3.1.5, "ocTransportEnv Data Structure" for more information.

- `username`
- `password`
- `savePassword`
- `newPassword`
- `priority`
- `secure`
- `pushOnly`
- `syncApps`
- `syncNewPublications`

If you use the HTTP default transport set in the `ocTransportEnv` structure, then the following is also saved:

- `url`
- `useProxy`
- `proxy`
- `proxyPort`

For more information on how to use these fields, see Section 3.1.3.1.3, "Managing the C/C++ Data Structures".

3.1.3.1.8 Manage What Tables Are Synchronized With ocSetTableSyncFlag Update the table flags for selective sync. Call this for each table to specify whether it should be synchronized (1) or not (0) for the next session. Selective sync only works if you have first performed at least one synchronization for the client. Then, set the flag so that on the next synchronization—that is, on the next invocation of the `ocDoSynchronize` method—a selective sync occurs.

The default `sync_flag` setting for `ocSetTableSyncFlag` is TRUE (1) for all the tables; that is, all tables are flagged to be synchronized. If you want to selectively synchronize specific tables, you must first disable the default setting for all tables and then enable the synchronization for only the specific tables that you want to synchronize.

Syntax

```
ocSetTableSyncFlag(ocEnv *env, const char* publication_name,
              const char* table_name, short sync_flag)
```

Table 3–46 lists the name and description of parameters for the ocSetTableSyncFlag function.

Table 3–46 ocSetTableSyncFlag Parameters

Name	Description
env	Pointer to the synchronization environment.
publication_name	The name of the publication which is being synchronized. If the value for the publication_name is NULL, it means all publications in the database. This string is the same as the client_name_template parameter of the Consolidator Manager CreatePublication method. In most cases, you will use NULL for this parameter. For more information, see Section 2.4, "Creating Publications Using Oracle Database Mobile Server APIs".
table_name	This is the name of the snapshot. It is the same as the name of the store, the third parameter of CreatePublicationItem(). For more information, see Section 2.4, "Creating Publications Using Oracle Database Mobile Server APIs".
sync_flag	If the sync_flag is set to 1, you must synchronize the publication. If the sync_flag is set to 0, then do not synchronize. The value for the sync_flag is not stored persistently. Each time before ocDoSynchronize(), you must call ocSetTableSyncFlag().

This function allows client applications to select the way specific tables are synchronized.

Set sync_flag for each table or each publication. If sync_flag = 0, the table is not synchronized.

To synchronize specific tables only, you must perform the following steps:

1. Disable the default setting, which is set to 1 (TRUE) for all the tables.

   Example:

   ```
   ocSetTableSyncFlag(&env, <publication_name>,NULL,0)
   ```

   Where <publication_name> must be replaced by the actual name of your publication, and where the value NULL is specified to mean **all** the tables for that publication without exception.

2. Enable the selective sync for specific tables.

   Example:

   ```
   ocSetTableSyncFlag(&env, <publication_name>,<table_name>,1)
   ```

3.1.3.1.9 Configure Proxy Information If you are using a firewall and need to configure proxy information, perform the following before you execute the ocDoSynchronize method:

1. Configure the proxy URL, IP address and/or port number through the ocSaveUserInfo function. See Section 3.1.3.1.7, "Managing User Settings With ocSaveUserInfo" for more information.
2. If required, configure the proxy user name and password. To configure the proxy user name and password, use the ocSetSyncOption and provide the following:

```
ocSetSyncOption( env, "HTTPUSER=<username>;HTTPPASS=<password>");
```

Note: The user name and password are limited to 28 characters.

Where the `ocSetSyncOption` syntax is as follows:

```
int ocSetSyncOption(ocEnv *env, const char *str);
```

You can set one or more name/value pairs separated by a semi-colon in the string. The previous example shows the `HTTPUSER` and `HTTPPASS` name/value pairs. You can also set the URL string as follows: `URL=www.myhost.com`.

3.1.3.1.10 Start the Synchronization With the ocDoSynchronize Method

Starts the synchronization process.

Syntax

```
int ocDoSynchronize( ocEnv *env );
```

Table 3–47 lists the name and description of the `ocDoSynchronize` parameter.

Table 3–47 ocDoSynchronize Parameters

Name	Description
`env`	Pointer to the synchronization environment.

This starts the synchronization cycle. A round trip synchronization is activated if `syncDirection` is `OC_SENDRECEIVE` (default). If `syncDirection` is `OC_SENDONLY` or `OC_RECEIVEONLY`, then the developer must implement a custom transport. If the developer wishes to upload only changes, then set `pushonly=1`. You cannot only download changes under the existing synchronization architecture.

This method returns when the synchronize completes. A return value of 0 indicates that the function has been executed successfully. If an error occurred, local errors are returned by `ocDoSynchronize`, which are defined in `ocerror.h`. For errors returned by the server, see the `ol_sync.log` error log file, which is written into the working directory of the application. Each line in the error file has the following format:

```
<type>, <code>, <date>, <message>
```

Where:

- `<type>`: The type of the message, which can either be set to `ERROR` or `SUCCESS`.
- `<code>`: Error code of the last operation of the synchronization.
- `<date>`: Date and timestamp for when the synchronization completes. This is in the format of `dd/mm/yyyy hh:mm:ss`.
- `<message>`: A readable message text.

3.1.3.1.11 See Progress of Synchronization with Progress Listening

If you create and set the progress callback function, the mobile client invokes this callback function at different times while the `ocDoSynchronize` method is executing. Create the callback function, as follows:

```
void myProgressProc ( void *env, int stage, int present);
```

When the ocDoSynchronize invokes your myProgressProc function, it provides the following information as input to your function:

- env—A pointer to the environment (ocEnv structure) for the synchronization session. This provides the function to retrieve the userContext pointer.
- stage—A number that denotes the stage in the synchronization process, which is one of the following values, where these values are defined in ocapi.h:

Table 3–48 Description of the Stage Values

Stage Value	Description
OC_PREPARE_START	Start of the prepare stage, which collects all internal data from the database and prepares to send the data to the server.
OC_PREPARING	Progress in the prepare stage.
OC_PREPARE_FINISH	Prepare stage is completed.
OC_SEND_START	Starting to send the data to the server.
OC_SENDING	Sending the data.
OC_SEND_FINISH	Completed sending the data.
OC_RECEIVE_START	Starting to receive data.
OC_RECEIVING	Receiving data from the server.
OC_RECEIVE_FINISH	Completed receiving data from the server.
OC_PROCESS_START	Starting to process received data.
OC_PROCESSING	Processing received data.
OC_PROCESS_FINISH	Completed processing. Synchronization is finished.
OC_RETRY_CALL	Resume synchronization is restarted.
OC_SYNC_FINISH	Last callback after the synchronization.

- present—The percentage completed in the particular stage that synchronization is in from 0 to 100.

If the function is a member of a class, then it must be defined as static.

After you create the callback function, set the function pointer in the ocEnv.fnProgress (Table 3–43) to the address of your callback function. Save this with the ocSaveUserInfo or ocSessionInit methods.

3.1.3.1.12 Clear the Synchronization Environment Using ocSessionTerm

Clears and performs a cleanup of the synchronization environment and buffers. This function must be invoked for every ocSessionInit, even if the ocDoSynchronize function is not performed.

Syntax

```
int ocSessionTerm( ocEnv *env );
```

Table 3–49 lists the ocSessionTerm parameter and its description.

Table 3–49 ocSessionTerm Parameters

Name	Description
env	Pointer to the environment structure returned by ocSessionInit.

De-initializes all the structures and memory created by the `ocSessionInit()` call. Users must ensure that they are always called in pairs.

3.1.3.1.13 Retrieve Synchronization Error Message with ocGetLastError Retrieves the synchronization error message and code.

Syntax

```
int ocGetLastError( ocEnv *env, char *buf, int buf_size);
```

Table 3–50 lists the `ocGetLastError` parameters.

Table 3–50 ocGet Parameters

Name	Description
`env`	Pointer to the environment structure returned by `ocSessionInit`.
`buf`	A string with the error message.
`buf_size`	The size of the error message string.

3.1.3.1.14 Enable File-Based Synchronization through C or C++ APIs When you want to use file-based synchronization, you must enable file-based synchronization. Once enabled, then when you initiate manual synchronization, then the synchronization file is created. See Section 5.10, "Synchronizing to a File Using File-Base Sync" in the *Oracle Database Mobile Server Administration and Deployment Guide* for more details on file-based synchronization.

To enable file-based synchronization programmatically with the `ocEnv` structure, perform the following:

1. Ensure that any previous settings of the File-Based Sync properties are set to `NULL`.
2. Initialize the environment with the `ocSessionInit` method.
3. Set the user name and password for the user that is initializing the synchronization.
4. Specify the synchronization direction and directory and filename for the synchronization file. The synchronization direction is either send, which creates the synchronization file, or receive, which takes in a file from the mobile server. These are configured in the `SEND_FILE_PROP` and `RECEIVE_FILE_PROP` properties with the `ocSetSyncProperty` method.
   - When you set the `SEND_FILE_PROP` property, specify the filename—including the relative or full path—where you want the mobile client to save the upload data for the mobile server. This file is created with the mobile client transactions destined for the mobile server.
   - When you set the `RECEIVE_FILE_PROP` property, specify the filename—including the relative or full path—where the data file that was received from the mobile server. This file is loaded and processed within the mobile client.

The following code example sets the direction, filename, user name and password. Notice that the `ocEnv` structure is memset to zero to ensure that if a previous direction and filename were specified, then these are invalidated for the next file-based synchronization. The `SEND_FILE_PROP` property is set with the filename and direction, which tells the Sync Client to marshall the mobile client transactions that are

to be uploaded to the mobile server into this file. If you were receiving a synchronization file from the mobile server, you would have set the `RECEIVE_FILE_PROP` property with the location and name of this file.

Finally, the `ocEnv` structure is provided to the `ocDoSynchronize` method, which performs the file-based synchronization.

```
ocEnv env;
memset(&env, 0, sizeof(ocEnv));
ocSessionInit(&env);
strcpy(env.username, "S11U1");
strcpy(env.password, "manager");
ocSetSyncProperty(&env, SEND_FILE_PROP, "C:\\temp\\send1.bin");
ocDoSynchronize(&env);
ocSessionTerm(&env);
```

3.1.3.2 mSync, OCAPI, and mSyncCom API

For more information, refer to the *Oracle Database Mobile Server API Specification*.

3.2 Manage Automatic Synchronization on the Mobile Client

The following APIs are used to manage automatic synchronization on the Mobile cllient:

- Section 3.2.1, "OSE APIs for Managing Automatic Synchronization"
- Section 3.2.2, "OCAPI APIs for Retrieving Status on Automatic Synchronization"
- Section 3.2.3, "OCAPI Notification APIs for the Automatic Synchronization Cycle Status"

3.2.1 OSE APIs for Managing Automatic Synchronization

> **Note:** Use the OSE classes for all new application development for your mobile clients. These are the classes that will be supported for the future.

Automatic synchronization is enabled by default if a publication is enabled for automated synchronization. However, you may programmatically turn on and off automatic synchronization on the mobile client using the Sync Control API.

Use the start or stop methods to start or stop the Sync Agent. The user may want to stop the Sync Agent for many reasons, such as aborting an automatic synchronization that may be running longer than desired, freeing up system resources, or de-fragmenting or backing up a client database.

> **Note:** There is also a GUI for starting, stopping the automatic synchronization process from the mobile server. See Section 5.5.2, "Start, Stop, or Get Status for Automatic Synchronization" in the *Oracle Database Mobile Server Administration and Deployment Guide* for more details.

The following are the different methods of managing automatic synchronization and the Sync Agent:

- Pause/Resume—Pause and resume all Sync Agent activities without stopping the process or freeing any resources, which would occur if you stopped or disabled the Sync Agent. Pause and resume are the most efficient method for suspending the Sync Agent and all automatic synchronization events, since it does not stop the process or free any resources.

  By default, if you are using the mSync GUI or sync API to initiate a manual synchronization, the underlying code pauses and resumes the automatic synchronization for you, as described below:

  1. Pause the automatic synchronization with the Sync Control API.
  2. Initiate the manual synchronization with the programmatic API.
  3. Resume the automatic synchronization with the Sync Control API.

  In some circumstances pausing syncagent might not be immediate if syncagent is currently running automatic synchronization task and cancellation cannot occur in the moment for various reasons. If syncagent cannot pause and you want the manual synchronization to start immediately, you can kill syncagent (for native clients). This can be done either by calling API in program or forcibly ending it with *autosync.exe* manually on Windows or with *autosync* manually on Linux.

  To kill syncagent by calling API, you can call bgControlAgent (bgSess, BG_CTRL_STOP, BG_CTRL_OPT_TERMINATE) in C, BGSession.kill() in java.

  To kill syncagent with *autosync.exe* on Windows or *autosync* on Linux, you can try to stop syncagent and it takes time, then the "Stop" button will change to "End" and you can kill it by pressing this button. Or the process *syncagent.exe* on Windows or *syncagent* on Linux can be manually terminated. After the manual synchronization is finished successfully, syncagent is started regardless whether it was running before or not. If the manual synchronization fails, syncagent is resumed - that is, it is only started if it was running before manual synchronization.

- Start/Stop—Stop the Sync Agent, which includes stopping the process and freeing all resources, until a start operation is executed or the client is restarted. If the start operation is not executed, the Sync Agent is automatically resumed when the client restarts.
- Enable/Disable—Disabling the Sync Agent stops the Sync Agent until an enable operation is executed. Even restarting the client will not re-enable the Sync Agent. Thus, in a disabled state, no automatic synchronization events will occur.

The following control APIs can be used to manage automatic synchronization:

- Section 3.2.1.1, "JAVA APIs for the Sync Agent and Automatic Synchronization"
- Section 3.2.1.2, "Native APIs for the Sync Agent and Automatic Synchronization"
- Section 3.2.1.3, "The .Net APIs for the Sync Agent and Automatic Synchronization"
- Section 3.2.1.4, "OCAPI Sync Control APIs"
- Section 3.2.1.5, "JavaScript APIs for the Sync Agent and Automatic Synchronization in PhoneGap"

3.2.1.1 JAVA APIs for the Sync Agent and Automatic Synchronization

The following sections describe how to manage automatic synchronization through the Sync Agent and how to retrieve status of both the Sync Agent and any automatic synchronization events:

> **Note:** For more details on these classes, refer to the *Oracle Database Mobile Server JavaDoc*.

3.2.1.1.1 Overview Once automatic synchronization for the mobile client is enabled, you can manage it either locally through the Sync Agent APIs or remotely through the Mobile Manager UI controls. The Sync Agent controls and manages all aspects of automatic synchronization, which occurs in the background. If a manual synchronization is started, Sync Agent stops automatic synchronization as indicated and resumes the automatic synchronization activities when the manual synchronization finishes.

The Java interface for controlling the Sync Agent and automatic synchronization resides in the `oracle.opensync.syncagent` package.

The Java interface provides for the following functions:

- Tracking the progress of the automatic synchronization process.
- Retrieve the status of the Sync Agent.
- Specify custom handlers for events that occur in automatic synchronization.

The following are the classes and interface for the Java API for controlling the Sync Agent and automatic synchronization:

- `BGSession` Class
- `BGAgentStatus` Class
- `BGSyncStatus` Class
- `BGMessageHandler` Interface

3.2.1.1.2 BGSession Class `BGSession` is the main class for controlling automatic synchronization through the Sync Agent, as follows:

- Start, stop, pause, resume, enable or disable automatic synchronization.
- Retrieve automatic synchronization status information.
- Specify message handlers for retrieving information about automatic synchronization events.

> **Note:** In a multi-threaded environment a single `BGSession` should not be used from multiple threads. Each thread should open its own session.

Constructor

```
BGSession( )
```

Public Methods

The public methods and their parameters for the `BGSession` class are listed in Table 3–51:

Table 3–51 BGSession Class Public Method Parameters

Method	Description
`void addMessageHandler (BGMessageHandler h)` `void removeMessageHandler (BGMessageHandler h)`	Adds or removes a custom message handler to the Sync Agent. See Section 3.2.1.1.5, "BGMessageHandler Interface" for more information.
`boolean agentEnabled()`	Returns `TRUE` if the Sync Agent is enabled; otherwise, `FALSE`.
`void close()`	Closes the session and release all the resources used by the session.
`void enableAgent(boolean on)`	`TRUE` enables the Sync Agent; `FALSE` disables the Sync Agent.
`BGAgentStatus getAgentStatus()`	Retrieves the current Sync Agent status. See Section 3.2.1.1.3, "BGAgentStatus Object" for more details on the status information returned.
`int getAgentStatusCode()`	Retrieves the current Sync Agent status code, which are described in Table 3–54, " Sync Agent Status Codes".
`BGSyncStatus getSyncStatus()`	Get current status of automatic synchronization managed by the Sync Agent. See Table 3–55 for more details on the status information returned.
`void pause()`	Pauses the Sync Agent. If the agent is aready paused or being paused, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be paused before returning. Use the `waitForStatus` method to wait for the Sync Agent.
`void resume()`	Resumes the Sync Agent. If the agent is already resumed or resuming, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be resumed before returning. Use the `waitForStatus` method to wait for the Sync Agent.
`void showUI()`	Starts up the Sync Agent UI.
`void start()`	Start the Sync Agent. If the agent is already running, starting, or resuming, this call is ignored. If the agent is paused, this call resumes the Sync Agent. This call is asynchronous and does not wait for the Sync Agent to be started before returning. Use the `waitForStatus` method to wait for the Sync Agent.

Table 3–51 (Cont.) BGSession Class Public Method Parameters

Method	Description
`void stop()`	Stop the Sync Agent. If the agent is already stopped or stopping, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be stopped before returning. Use the `waitForStatus` method to wait for the Sync Agent.
`void waitForStatus (int statusCode)` `boolean waitForStatus (int statusCode, long timeOut)`	Wait for the Sync Agent to reach specified status. You can also wait for a specified timeout. ▪ The parameter can be one of the following: `RUNNING`, `PAUSED` or `STOPPED`. ▪ The `timeOut` parameter is the maximum time to wait for in milliseconds. Unlimited time if no timeout provided. Returns `TRUE` if the agent has reached specified status; `FALSE` if the timeout has occurred.
`BGException getFatalError()`	If sync agent is in DEFUNCT state, retrieve the error information that caused the bad internal state. See Table 3–54, " Sync Agent Status Codes" for description of DEFUNCT.

Example

The following example demonstrates how to start the Sync Agent, retrieve status of the Sync Agent and add a message handler for the session:

```
// Create the BGSession object
 BGSession sess = new BGSession();
      try {
          //Start the Sync Agent, which enables all automatic synchronization
          //events
          sess.start();
          //Wait until the Sync Agent successfully starts
          sess.waitForStatus(BGAgentStatus.RUNNING);
          //Retrieve the status of the Sync Agent
          BGAgentStatus s = sess.getAgentStatus();
          //Print out the user that is using automatic synchronization
          System.out.println("User name: " + s.clientId);
          //Add a message handler
          sess.addMessageHandler(new myMessageHandler());
          ...
      }
      finally {
          //When finished, close the session to release all resources
          sess.close();
      }
```

3.2.1.1.3 BGAgentStatus Object

The `BGAgentStatus` object represents the current status of the Sync Agent.

Public Methods

The methods for the `BGAgentStatus` are listed in Table 3–52.

Table 3–52 BGAgentStatus Class Public Method

Method	Description
`static java.lang.String statusName(int statusCode)`	Get language-specific name of a given status code. When you provide one of the status codes shown in Table 3–54, the appropriate name is returned. Translation dependent on the device language settings

Fields

`BGAgentStatus` provides status information on the Sync Agent. Table 3–53 lists and describes the status information fields within the `BGAgentStatus` class.

Table 3–53 BGAgentStatus Class Fields

Parameters	Description
`java.lang.String appName`	The name of the application or process that is executing the Sync Agent. On some platforms, such as Android, it is possible to execute the Sync Agent within an application process.
`int batteryPower`	Remaining percentage of battery life, if relevant.
`java.lang.String clientId`	Sync user name.
`java.lang.String networkName`	Name of the network currently used for synchronization, evaluated by Sync Agent.
`int networkSpeed`	Network bandwidth in bits per second.
`int processId`	Process id of the process that is executing the Sync Agent, if relevant for a given platform.
`int statusCode`	Retrieves the status of the Sync Agent. Status codes that can be returned are detailed in Table 3–54, " Sync Agent Status Codes".

The `BGAgentStatus` object defines the Sync Agent status codes, which are as follows:

Table 3–54 Sync Agent Status Codes

Status Code	Status Name	Description
0	`STOPPED`	Sync Agent application is not running.
1	`START_PENDING`	Sync Agent is in the process of starting.
2	`RUNNING`	Sync Agent is running. Any tasks within Sync Agent such as synchronization, compose, apply, rule evaluation, network evaluation and other operations can be active.
3	`PAUSE_PENDING`	Sync Agent is in the process of being paused.
4	`PAUSED`	Sync Agent is paused. When paused, none of the tasks within Sync Agent are running. However, resources such as memory and threads, are saved in the case of a speedy resume. Pause and resume are generally faster than start and stop. When a manual synchronization is started, this pauses the Sync Agent until the manual synchronization is completed. At that point, the Sync Agent is resumed.
5	`RESUME_PENDING`	Sync Agent is in the process of resuming.
6	`STOP_PENDING`	Sync Agent is in the process of stopping.

Table 3–54 (Cont.) Sync Agent Status Codes

Status Code	Status Name	Description
7	DEFUNCT	Sync Agent encountered fatal error and is in a bad internal state. Sync Agent's environment needs to be cleaned up and restarted.

Example

The following provides an example of retrieving and processing the Sync Agent status:

```
/* retrieve the Sync Agent status */
BGAgentStatus as = bgSess.getAgentStatus();
/* Print Sync Agent status */
 System.out.println("Agent Status:          " +
    BGAgentStatus.statusName(as.statusCode));
if (as.statusCode == BGAgentStatus.STOPPED) return;
/* Identify the client id, process id and name */
System.out.println("Client ID:             " + as.clientId);
System.out.println("Process Name:          " + as.appName);
System.out.println("Process ID:            " + as.processId);
/* network name and speed */
if (as.networkSpeed > 0) {
   System.out.println("Network Name:          " + as.networkName);
   System.out.println("Network Speed:         " + as.networkSpeed + " bps");
}
else System.out.println("Network is not present");
/* battery power */
if (as.batteryPower > 0)
   System.out.println("Battery Power:         " + as.batteryPower + "%");
else
   System.out.println("Battery is not present");
```

3.2.1.1.4 BGSyncStatus Object Current status of automatic synchronization. If automatic synchronization is in progress, `startTime` will have a non-zero value and `endTime` will be zero.

Fields

`BGSyncStatus` provides status information on automatic synchronization in the fields listed in Table 3–55.

Table 3–55 BGSyncStatus Class Fields

Parameters	Description
`long endTime`	End time of the last synchronization in milliseconds since the standard base time of January 1, 1970, 00:00:00 GMT. Returns zero if the synchronzation is currently in progress or has not yet run.
`java.lang.Throwable lastError`	Exception object thrown during the last synchronization. Returns `NULL` if the last synchronization was successful or no synchronization has completed yet.
`int prio`	Priority of the current or last synchronization.
`int progressStage`	Progress stage of synchronization if it is in progress.

Table 3–55 (Cont.) BGSyncStatus Class Fields

Parameters	Description
`int progressVal`	Progress value in percentage of synchronization, if it is in progress.
`java.lang.String[] pubs`	Array of names of publications synchronized currently or during last synchronization.
`long startTime`	Start time of current or last synchronization, in milliseconds, since the standard base time of January 1, 1970, 00:00:00 GMT. Returns zero if the synchronization has not yet started or the last synchronization time is unknown.

Example

The following provides an example of retrieving and processing the synchronization status:

```
/* Retrieve the synchronization status */
BGSyncStatus ss = bgSess.getSyncStatus();

/* start time */
if (ss.startTime == 0) return;
System.out.println("Sync Started:          " + time2str(ss.startTime));
/* end time */
if (ss.endTime != 0)
  System.out.println("Sync Finished:         " + time2str(ss.endTime));
/* number of publications synchronized */
if (ss.pubs != NULL && ss.pubs.length != 0) {
  System.out.print("Publications synced:   ");
  for(int i = 0; i < ss.pubs.length; i++) {
    System.out.print(ss.pubs[i]);
    if (i == ss.pubs.length - 1)
       System.out.println();
    else
       System.out.print(", ");
 }
}
/* synchronization priority */
System.out.println("Sync Priority:         " +
    (ss.prio == OSESession.PRIO_HIGH ? "High" : "Normal"));
/* synchronization result */
System.out.print("Sync Result:           ");
if (ss.lastError == NULL)
  System.out.println("Success");
else
  System.out.println("Failure: " + ss.lastError.toString());
```

3.2.1.1.5 BGMessageHandler Interface The `BGMessageHandler` interface enables the Sync Agent and automatic synchronization message and error data to be trapped during synchronization.

Sync calls the `handleLogMessage` method with a parameter message of type `LogMessage` to report the current state and any errors for the Sync Agent or automatic synchronization. Within the `handleLogMessage` method, you can perform the appropriate action for the errors returned in the `LogMessage` structure.

For a complete description of the `LogMessage` class, see Section 3.2.1.1.6, "LogMessage Class".

Example

This example demonstrates how to implement the `BGMessageHandler`.

```
class myMessageHandler implements BGMessageHandler;

{
 public void handleLogMessage(message)
 {
  private PrintStream ps =
     new PrintStream(new FileOutputStream(FILE_NAME, false));

  ps.println("Time: " + new Date(message.time));
  ps.println("Type: " + (message.type == LogMessage.INFO ? "INFO" :
    (message.type == LogMessage.WARNING ? "WARNING" : "ERROR")));
  ps.println("Id: " + message.id);
  if (message.text != NULL)
    ps.println("Text: " + message.text);
  if (message.cause != NULL)
    ps.println("Cause: " + message.cause);
 }
}
```

3.2.1.1.6 LogMessage Class The `LogMessage` class contains error message information passed to handlers when an event occurs within the application. It exists in the `oracle.opensync.util` package.

Table 3–56 LogMessage Class

Name	Description
`java.lang.Throwable cause`	For error messages, optional cause of the error, which can be `NULL`. If the error has an underlying cause, the `cause` could potentially point to several secondary messages through iterative `java.lang.Throwable` objects. This is useful if OSE returns an internal error. If `cause` is not present, `NULL` is returned.
`static int ERROR`	Error message number.
`int id`	Application-specific message number.
`static int INFO`	Informational message number.
`static int NUM_TYPES`	
`java.lang.String source`	Name of the application that created the message.
`java.lang.String text`	Message text.
`long time`	Message creation time. Number in milliseconds since the epoch.
`int type`	Message type: `INFO`, `WARNING` or `ERROR`.
`static int WARNING`	Warning message.

Example

The following demonstrates how to print out the error information in the `LogMessage` class:

```
private void printMsg(PrintStream ps, LogMessage m)
{
```

```
 ps.println("Time: " + new Date(m.time));
 ps.println("Type: " + (m.type == LogMessage.INFO ? "INFO" :
   (m.type == LogMessage.WARNING ? "WARNING" : "ERROR")));
ps.println("Id: " + m.id);
if (m.text != NULL)
   ps.println("Text: " + m.text);
if (m.cause != NULL)
   ps.println("Cause: " + m.cause);
ps.println();
}
```

3.2.1.1.7 BGException Class This class signals a non-recoverable error during the synchronization process. The `BGException()` class constructs a `clear` object. The parameters for the constructor are listed in Table 3–57:

Constructors

```
BGException(int errCode)

BGException(int errCode, java.lang.Object arg)

BGException(int errCode, java.lang.Object[] args,
java.lang.Throwable cause)

BGException(int errCode, java.lang.Object arg1, java.lang.Object
arg2)

BGException(int errCode, java.lang.Object arg1, java.lang.Object
arg2, java.lang.Object arg3)

BGException(int errCode, java.lang.Object arg,
java.lang.Throwable cause)

BGException(int errCode, java.lang.Throwable cause)
```

Table 3–57 BGException Constructor Parameter Description

Parameter	Description
`errCode`	Error codes are provided within the `BGExceptionConstants` class. Error codes for automatic synchronization are provided in the `BGExceptionConstants` class. Some `BGException` instances are thrown from Sync Control APIs. Others are used as causes of the automatic synchronization error messages. The message handler returns an error message. For a complete list of the error messages that can be thrown in `BGException`, see "Exception Error Codes and Messages" in the *Oracle Database Mobile Server Message Reference*.
`arg, args, arg1, arg2, arg3`	Return variables for information within the error message.
`cause`	The cause of this throwable or NULL if the cause is nonexistent or unknown.

BGException class extends BaseException class. The methods for getting cause and message are listed in Table 3–9, " BaseException Class Public Methods".

For a complete list of the error messages that can be thrown in the BGException, see "Exception Error Codes and Messages" in the Oracle Database Mobile Server Message Reference.

3.2.1.2 Native APIs for the Sync Agent and Automatic Synchronization

The following sections describe how to manage automatic synchronization through the Sync Agent and how to retrieve status of both the Sync Agent and any automatic synchronization events:

3.2.1.2.1 Overview Once automatic synchronization for the mobile client is enabled, you can manage it locally through the Sync Agent APIs and remotely through the Mobile Manager UI controls. The Sync Agent controls and manages all aspects of automatic synchronization, which occurs in the background. If a manual synchronization is paused, the Sync Agent stops automatic synchronization as indicated and resumes the automatic synchronization activities when the manual synchronization finishes.

The native interface for controlling the Sync Agent and automatic synchronization are defined in the `<ORACLE_HOME>\Mobile\Sdk\include\bgsync.h` file and implemented in `<ORACLE_HOME>\Mobile\Sdk\bin\bgsync.dll`.

The native interface provides for the following functions:

- Start, stop, pause, resume, enable or disable automatic synchronization.
- Retrieve automatic synchronization status information.
- Specify message handlers for retrieving information about automatic synchronization events.

3.2.1.2.2 Initializing the Environment The `bgOpenSession` method initializes the automatic synchronization environment—which is passed to each subsequent call with the `bgSess` handle.

In a multi-threaded environment, you cannot concurrently use a session from multiple threads, even with the same user. Instead, each thread should open its own session with the `bgOpenSession` method.

> **Note:** Every time you invoke the `bgOpenSession` method, you must also clean up with `bgCloseSession` method. These methods should always be called in pairs. See Section 3.2.1.2.6, "Close the Synchronization Environment" for more information.

Syntax

```
bgError bgOpenSession(bgSess *sess);
```

Table 3–58 lists the `bgOpenSession` parameters.

Table 3–58 bgOpenSession Parameters

Name	Description
`bgSess sess`	Pointer to a session handle into which the new session is returned. Returns `BG_ERR_INVALID_SESS` if the handle is `NULL`.

This call initializes the `bgSess` automatic synchronization environment handle—which holds context information for the synchronization engine.

Returns zero if successful. Returns `BG_ERR_INTERNAL` if a system error has occured.

3.2.1.2.3 Synchronization Status You can retrieve the status of automatic synchronization events or of the Sync Agent. The following sections describe the methods for retrieving the status:

- Retrieve Sync Agent Status
- Retrieve Status of the Current Automatic Synchronization Event

Retrieve Sync Agent Status

Get the Sync Agent operational status with the `bgGetAgentStatus` method.

Syntax

```
bgError bgGetAgentStatus(bgSess sess, bgAgentStatus *s);
```

Table 3–59 lists the `bgGetAgentStatus` parameters.

Table 3–59 bgGetAgentStatus Parameters

Name	Description
`bgSess sess`	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns `BG_ERR_INVALID_SESS` if the handle is `NULL`.
`bgAgentStatus *s`	Pointer to the Sync Agent status structure in which the status is returned. This pointer cannot be `NULL`. Note that the memory for the pointer fields is maintained by the Sync Agent and should not be freed by the application. See Table 3–60 for details on the returned `bgAgentStatus` structure.

Returns zero if the agent has reached specified status. Returns `BG_ERR_INVALID_SESSION` if the session handle is invalid. Returns `BG_ERR_INTERNAL` if a system error has occured.

The bgAgentStatus Structure

The `bgAgentStatus` structure provides status information on the Sync Agent.

Syntax

```
typedef struct _bgAgentStatus {
  ose1B statusCode;
  oseBool isExternal;
  oseU2B _reserved; /**< for alignment */
  const char *clientId;
  const char *processName;
  const char *networkName;
  ose4B processId;
  ose4B networkSpeed;
```

```
  ose4B batteryPower;
} bgAgentStatus;
```

Table 3–60 lists and describes the status information fields.

Table 3–60 bgAgentStatus Fields

Parameters	Description
`statusCode`	Retrieves the status of the Sync Agent. Status codes that can be returned are detailed in Table 3–61.
`isExternal`	A boolean value that if `OSE_TRUE`, the Sync Agent was started in a separate process.
`clientId`	Sync user name.
`processName`	Name of the process within which the Sync Agent is currently running.
`networkName`	Name of the network currently used for synchronization, evaluated by Sync Agent.
`processId`	Process id of the process that is executing the Sync Agent, if relevant for a given platform.
`networkSpeed`	Network bandwidth in bits per second.
`batteryPower`	Remaining percentage of battery life, if relevant.

Table 3–61 lists the Sync Agent status codes:

Table 3–61 Sync Agent Status Codes

Code	Status Name	Description
0	`BG_STATUS_STOPPED`	Sync Agent application is not running.
1	`BG_STATUS_START_PENDING`	Sync Agent is in the process of starting.
2	`BG_STATUS_RUNNING`	Sync Agent is running. Any tasks within Sync Agent such as synchronization, compose, apply, rule evaluation, network evaluation and other operations can be active.
3	`BG_STATUS_PAUSE_PENDING`	Sync Agent is in the process of being paused.
4	`BG_STATUS_PAUSED`	Sync Agent is paused. When paused, none of the tasks within Sync Agent are running. However, resources such as memory and threads, are saved in the case of a speedy resume. Pause and resume are generally faster than start and stop. When a manual synchronization is started, this pauses the Sync Agent until the manual synchronization is completed. At that point, the Sync Agent is resumed.
5	`BG_STATUS_RESUME_PENDING`	Sync Agent is in the process of resuming.
6	`BG_STATUS_STOP_PENDING`	Sync Agent is in the process of stopping.

Retrieve Status of the Current Automatic Synchronization Event

Get the current status of the automatic synchronization event within the Sync Agent with the `bgGetSyncStatus` method.

Syntax

```
bgError bgGetSyncStatus(bgSess sess, bgSyncStatus *s);
```

Table 3–62 lists the bgGetSyncStatus parameters.

Table 3–62 bgGetSyncStatus Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.
bgSyncStatus *s	Pointer to the synchronization status structure in which the status is returned. This pointer cannot be NULL. Note that the memory for the pointer is maintained by the Sync Agent and should not be freed by the application. See Table 3–63 for details on the returned bgSyncStatus structure.

Returns zero if the agent has reached specified status. Returns BG_ERR_INVALID_SESSION if the session handle is invalid. Returns BG_ERR_INTERNAL if a system error has occured.

The bgSyncStatus Structure

The bgSyncStatus structure provides status information on a current or the last automatic synchronization event in the Sync Agent.

Syntax

```
typedef struct _bgSyncStatus {
  oseSize pubCnt;
  const char **pubs;
  osePrio prio;
  oseU1B _reserved[3]; /**< for alignment */
  ose8B startTime;
  ose8B endTime;
  oseError res;
  const char *errMsg;
  const char *stateName;
  ose2B state;
  ose2B progress;
} bgSyncStatus;
```

Table 3–63 describes the fields in bgSyncStatus.

Table 3–63 bgSyncStatus Fields

Parameters	Description
pubCnt	Number of publications synchronized.
pubs	Array of names of publications synchronized currently or during last synchronization.
prio	Priority of the current or last synchronization.
startTime	Start time of current or last synchronization, in milliseconds, since the standard base time of January 1, 1970, 00:00:00 GMT. Returns zero if the synchronization has not yet started or the last synchronization time is unknown.

Table 3–63 (Cont.) bgSyncStatus Fields

Parameters	Description
endTime	End time of the last synchronization in milliseconds since the standard base time of January 1, 1970, 00:00:00 GMT. Returns zero if the synchronzation is currently in progress or has not yet run.
res	Last synchronization error code. Returns zero if the last synchronization was successful or no synchronization has completed yet.
errMsg	Last synchronization error message or NULL if no error.
stateName	Current synchronization stage name.
state	Current synchronization stage Returns OSE_SYNC_STATE_IDLE if the synchronization is not in progress.
progress	Progress value in percentage of synchronization, if it is in progress.

3.2.1.2.4 Control the Sync Agent You can issue control commands to the Sync Agent with the bgControlAgent method. This call returns immediately and does not wait for completion of command execution. Use the bgWaitForStatus method to wait until the Sync Agent reaches a certain status.

- Issue Sync Agent Control Commands
- Wait for Specific Sync Agent Status

Issue Sync Agent Control Commands

Returns zero if successful. Returns BG_ERR_INVALID_SESSION if the session handle is invalid. Returns BG_ERR_INVALID_COMMAND if the control command code is invalid. Returns BG_ERR_CANNOT_ACCEPT_CTRL if the Sync Agent is not able to execute the provided command in its current state.

Syntax

```
bgError bgControlAgent(bgSess sess, int ctrl, int opt);
```

Table 3–64 lists the bgControlAgent parameters.

Table 3–64 bgControlAgent Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.
int ctrl	Sync Agent control command codes, which are listed in Table 3–65.
int opt	options for given command

Table 3–65 lists the Sync Agent control command codes and options.

Table 3–65 Sync Agent Control Command Codes

Name	Description	Option Value
BG_CTRL_START	Start the Sync Agent. If the agent is running, pending or resume pending, this command has no effect. If the Sync Agent is paused, this command resumes the agent.	By default, this starts the Sync Agent within a separate process named syncagent or syncagent.exe on Windows platforms. However, if you set the option to BG_CTRL_OPT_START_INTERNAL, the Sync Agent is started within the current process. This is the default for iOS clients.
BG_CTRL_STOP	Stop the Sync Agent. If the agent is stopped or stop pending, this command has no effect.	By default, stops the Sync Agent gracefully. However, if you set the option to BG_CTRL_OPT_TERMINATE, this kills the Sync Agent process, which should be used only as a last resort. This is not advisable if the Sync Agent was started with the BG_CTRL_OPT_START_INTERNAL option.
BG_CTRL_PAUSE	Pause the Sync Agent. If the agent is paused or pause pending, this command has no effect.	No options available.
BG_CTRL_RESUME	Resume the Sync Agent. If the agent is running, start pending, or resume pending, this command has no effect.	No options available.

Wait for Specific Sync Agent Status

Wait for the Sync Agent to reach specified status. You can also wait for a specified timeout.

This method is often used to wait for the Sync Agent to start. For example, the following code shows starting the Sync Agent and then waiting until the Sync Agent is up and running:

```
bgSess sess;
bgOpenSession(&sess);
bgControlAgent(sess, BG_CTRL_START, 0);
bgWaitForStatus(sess, BG_STATUS_RUNNING);
```

Syntax

```
bgError bgWaitForStatus(bgSess sess, int statusCode, long timeOut);
```

Table 3–66 lists the bgWaitForStatus parameters.

Table 3–66 bgWaitForStatus Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.
int statusCode	The status can be one of the following: BG_STATUS_STOPPED, BG_STATUS_RUNNING or BG_STATUS_PAUSED.
long timeOut	The timeOut parameter is the maximum time to wait for in milliseconds. Unlimited time is enabled if timeOut is specified as -1.

Returns zero if the agent has reached specified status. Returns BG_ERR_INVALID_SESSION if the session handle is invalid. Returns BG_ERR_WAIT_TIMEOUT if the timeout has expired. Returns BG_ERR_INVALID_WAIT_STATUS if the status code is invalid. Other errors may also be returned.

3.2.1.2.5 Setting Synchronization Parameters You can set certain session parameters explicitly with the set session methods. The new parameter value takes effect only after the Sync Agent is restarted. When the session is created, the initial value for each option is loaded from the ose.ini file.

Options are separated into boolean and numeric parameters:

- Boolean options are those options that can only be set to OSE_TRUE or OSE_FALSE.
- Numeric options are set to an integer value.

Use the bgSetNumParam and bgGetNumParam methods to set and get the boolean and numeric session options. The bgSetNumParam method sets a parameter in the ose.ini file.

bgSetNumParam

```
bgError bgSetNumParam(bgSess sess, int param, long val);
```

Table 3–67 lists the bgSetNumParam parameters.

Returns zero if the option is retrieved successfully. Returns BG_ERR_INVALID_SESSION if the session handle is invalid. Returns BG_ERR_INVALID_PARAM if an invalid parameter code is specified. Returns BG_ERR_INTERNAL if a system error has occured. Other errors may also be returned.

Table 3–67 bgSetNumParam Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.
int param	Parameter code. See Table 3–13 for a list of all of the options that can bet set by this method.
long val	The parameter value to set. See Table 3–13 for potential values for this option.

bgGetNumParam

```
bgError bgGetNumParam(bgSess sess, int param, long *val);
```

Table 3–68 lists the bgGetNumParam parameters.

Returns zero if the option is retrieved successfully. Returns BG_ERR_INVALID_SESSION if the session handle is invalid. Returns BG_ERR_INVALID_PARAM if an invalid parameter code is specified. Returns BG_ERR_INTERNAL if a system error has occured.

Table 3–68 bgGetNumParam Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.

Table 3–68 (Cont.) bgGetNumParam Parameters

Name	Description
`int param`	Param code. See Table 3–69 for a list of all of the options that can bet retrieved by this method.
`long *val`	Pointer to a variable into which to return the parameter value. The pointer cannot be `NULL`. See Table 3–69 for potential values for this option.

Table 3–69 lists all boolean and numeric synchronization options and potential values. For all boolean options, the value can only be either `OSE_TRUE` or `OSE_FALSE`.

Table 3–69 Numeric and Boolean Session Options

Session Option	Description
`BG_PARAM_DISABLE_AGENT`	Boolean parameter. If `OSE_TRUE`, specifies that the Sync Agent should be disabled. If the Sync Agent is disabled, the application and system startup process cannot start the Sync Agent. In this case, `BG_ERR_AGENT_DISABLED` error is returned from the `bgControlAgent` method. `OSE_FALSE` is the default. Refers to the `ose.ini` parameter: `BGSYNC.DISABLE`.
`BG_PARAM_MAX_LOG_FILE_COUNT`	Maximum number of log files to keep in the `bglog` directory. The log is circular, so that when the maximum number of files is reached and a new log file needs to be added, the oldest file will be removed. For Windows 32 and Linux, the default is 128; for Windows CE, the default is 32. Refers to the `ose.ini` parameter: `BGSYNC.MAX_LOG_FILES`.
`BG_PARAM_MAX_LOG_FILE_SIZE`	Maximum log file size in bytes. On Windows 32 and Linux, the default is 1 MB. On Windows CE, the default is 128 KB. Refers to the `ose.ini` parameter: `BGSYNC.MAX_LOG_FILE_SIZE`.
`BG_PARAM_NET_WAIT_TIMEOUT`	Time interval in milliseconds for the network manager to wait before evaluating the network state in absence of notifications. The network manager evaluates the network when a notification is received or the interval expires. The network is also evaluated periodically after each period of the said interval. The default is 10 minutes (600000 milliseconds). Refers to the `ose.ini` parameter: `BGSYNC.NET_WAIT_TIMEOUT`.

3.2.1.2.6 Close the Synchronization Environment Clears and performs a cleanup of the synchronization environment and resources. This function must be invoked for every `bgOpenSession`.

Returns zero if successful. Returns `BG_ERR_INVALID_SESSION` if the session handle is invalid.

Syntax

```
bgError bgCloseSession(bgSess sess);
```

Table 3–70 lists the bgCloseSession parameters.

Table 3–70 bgCloseSession Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.

3.2.1.2.7 Trap Sync Agent Messages with a Callback Function You can create a callback function that is called when Sync Agent messages are generated, which traps automatic synchronization messages and error data. You register or unregister this callback function with the bgAddMsgCallback or bgRemoveMsgCallback methods.

- Register the Callback Function
- Unregister the Callback Function

Register the Callback Function

Returns zero if successful. Returns BG_ERR_INVALID_SESSION if the session handle is invalid. Returns BG_ERR_INTERNAL if a system error has occured.

Syntax

```
bgError bgAddMsgCallback(bgSess sess, bgUserCtx ctx, bgMsgCallback cb);
```

Table 3–71 lists the bgAddMsgCallback parameters.

Table 3–71 bgAddMsgCallback Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.
bgUserCtx ctx	A user-defined structure that contains the session environment settings. Since the bgMsgCallback callback method is user-implemented, it should know how to process the user-defined structure. The user context structure must be initialized by the application before calling the bgAddMsgCallback method.
bgMsgCallback cb	The callback function handle.

The Sync Agent invokes the callback function with a message to report the current state and any errors for the Sync Agent or automatic synchronization. Within the callback method, you can perform the appropriate action for the errors returned.

The message callback function is declared as follows:

```
typedef void (* bgMsgCallback)(bgUserCtx ctx, const bgMsg *msg);
```

Table 3–72 lists the bgMsgCallback parameters.

Table 3–72 bgMsgCallback Parameters

Name	Description
`bgUserCtx ctx`	A user-defined structure that contains the session environment settings. Since the `bgMsgCallback` callback method is user-implemented, it should know how to process the user-defined structure. The user context structure must be initialized by the application before calling the `bgAddMsgCallback` method.
`bgMsg *msg`	Pointer for the message structure that reports on the current state and any errors for the Sync Agent or automatic synchronization.

The `bgMsg` structure contains error message information passed to handlers when an event occurs within the Sync Agent or automatic synchronization.

Note: The memory for `bgMsg` structure is allocated within the mobile client and should not be freed by the application.

Syntax

```
typedef struct _bgMsg {
  ose8B time;
  ose4B type;
  ose4B id;
  const char *txt;
  const bgErrorDesc *cause;
} bgMsg;
```

Table 3–73 describes the fields of `bgMsg`.

Table 3–73 bgMsg Structure

Name	Description
`time`	Message creation time. Number in milliseconds since the epoch.
`type`	Message type: `BG_MSG_TYPE_INFO`, `BG_MSG_TYPE_WARNING` or `BG_MSG_TYPE_ERROR`.
`id`	Application-specific message number.
`txt`	Message text.
`cause`	For error messages, optional cause of the error, which can be `NULL`. If not `NULL`, points to a `bgErrorDesc` structure. If the error has an underlying cause, the `cause` could potentially point to several secondary messages through a chain of `bgErrorDesc` structures through `cause`. This is useful if the Sync Agent returns an internal error. If `cause` is not present, then `cause` is set to `NULL`.

Unregister the Callback Function

Remove the message callback function from the Sync Agent, after which it will no longer be invoked.

Returns zero if successful. Returns `BG_ERR_INVALID_SESSION` if the session handle is invalid. Returns `BG_ERR_INTERNAL` if a system error has occured.

Syntax

```
bgError bgRemoveMsgCallback(bgSess sess, bgUserCtx ctx, bgMsgCallback cb);
```

Table 3–74 lists the bgRemoveMsgCallback parameters.

Table 3–74 bgRemoveMsgCallback Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. Returns BG_ERR_INVALID_SESS if the handle is NULL.
bgUserCtx ctx	A user-defined structure that contains the session environment settings.
bgMsgCallback cb	The callback function handle.

3.2.1.2.8 Retrieve Synchronization Error Message Retrieves the last error of a call to one of the automatic synchronization APIs.

Syntax

```
bgError bgGetLastError(bgSess sess, const bgErrorDesc **errDesc);
```

Table 3–75 lists the bgGetLastError parameters.

Returns zero if successful. Returns the BG_ERR_INVALID_SESS error if the session handle was invalid. Returns the BG_ERR_INTERNAL error if a system error has occured.

Table 3–75 bgGetLastError Parameters

Name	Description
bgSess sess	Session pointer to a session handle, which contains the automatic synchronization environment across all calls for this session. This handle can be NULL if trying to retrieve error information from a failed bgOpenSession call.
const bgErrorDesc **errDesc	A pointer to an bgErrorDesc pointer into which the bgErrorDesc pointer is returned. The pointer cannot be NULL. The structures referenced by this pointer are only valid until the next call.

The bgErrorDesc structure is a typedef for oseErrorDesc, which is defined as follows:

```
typedef struct _oseErrorDesc {
  oseError code;                /* error code */
  const char *type;             /* a string describing the type of error */
  const char *msg;              /* error message */
  struct _oseErrorDesc *cause;  /* underlying cause, if present */
} oseErrorDesc;
```

If the error has an underlying cause, the oseErrorDesc.cause points to another oseErrorDesc structure, which in turn can have its own cause, and so on. This is useful if the OSE call returns an OSE_ERR_INTERNAL_ERROR, which can be further defined within another oseErrorDesc structure. If the cause is not present, oseErrorDesc.cause is NULL.

Note: The memory for oseErrorDesc structure is allocated within mobile server and should not be freed by the application.

3.2.1.3 The .Net APIs for the Sync Agent and Automatic Synchronization

The following sections describe how to manage automatic synchronization through the Sync Agent and how to retrieve status of both the Sync Agent and any automatic synchronization events:

> **Note:** OCAPI C# Sync Control APIs are no longer supported, use the APIs described in this section instead.

3.2.1.3.1 Overview Once automatic synchronization for the mobile client is enabled, you can manage it locally through the Sync Agent APIs and remotely through the Mobile Manager UI controls. The Sync Agent controls and manages all aspects of automatic synchronization, which occurs in the background. If a manual synchronization is paused, the Sync Agent stops automatic synchronization as indicated and resumes the automatic synchronization activities when the manual synchronization finishes.

The .Net interface for controlling the Sync Agent and automatic synchronization resides in the `Oracle.OpenSync.SyncAgent` namespace. The .Net APIs are implemented in `Oracle.OpenSync.dll`.

The .Net interface provides for the following functions:

- Controlling the Sync Agent.
- Retrieve the status of the Sync Agent and automatic synchronization events.
- Specify custom handlers for events that occur in automatic synchronization.

The following are the classes and interface for the .Net API for controlling the Sync Agent and automatic synchronization:

- `BGStatusCode` Enumeration
- `BGSession` Class
- `BGAgentStatus` Class
- `BGSyncStatus` Class
- `BGMessageType` Enumeration
- `BGMessageHandler` Interface

3.2.1.3.2 BGStatusCode Enumeration The `BGStatusCode` enumeration specifies the Sync Agent status codes.

Syntax

```
public enum BGStatusCode
    {
        Stopped = 0,
        StartPending,
        Running,
        PausePending,
        Paused,
        ResumePending,
        StopPending
    }
```

Table 3–76 provides more information about these codes.

Table 3–76 Sync Agent Status Codes

Status Name	Value	Description
`Stopped`	0	Sync Agent application is not running.
`StartPending`	1	Sync Agent is in the process of starting.
`Running`	2	Sync Agent is running. Any tasks within Sync Agent such as synchronization, compose, apply, rule evaluation, network evaluation and other operations can be active.
`PausePending`	3	Sync Agent is in the process of being paused.
`Paused`	4	Sync Agent is paused. When paused, none of the tasks within Sync Agent are running. However, resources such as memory and threads, are saved in the case of a speedy resume. Pause and resume are generally faster than start and stop. When a manual synchronization is started, this pauses the Sync Agent until the manual synchronization is completed. At that point, the Sync Agent is resumed.
`ResumePending`	5	Sync Agent is in the process of resuming.
`StopPending`	6	Sync Agent is in the process of stopping.

3.2.1.3.3 BGSession Class `BGSession` is the main class for controlling automatic synchronization through the Sync Agent, as follows:

- Start, stop, pause, resume, enable or disable automatic synchronization.
- Retrieve automatic synchronization status information.
- Specify message handlers for retrieving information about automatic synchronization events.

> **Note:** In a multi-threaded environment a single `BGSession` should not be used from multiple threads. Each thread should open its own session.

Properties

The following documents the properties for the `BGSession` class.

- Read-write properties read or write the value of a field with `get` and `set` accessors.
- Read-only properties read the value of a field with the `get` accessor.
- Write-only properties set the value of a field with the `set` accessor.

Boolean properties get or set the value of a field to either true or false.

Table 3–77 lists all BGSession properties. Section 3.2.1.3.2, "BGStatusCode Enumeration" lists all enumerations.

Table 3–77 BGSession Properties

Property	Description
StatusCode	Gets the Sync Agent status code from the BGStatusCode enumeration.
Enabled	Boolean property that gets or sets to TRUE if the Sync Agent is enabled; otherwise, FALSE. If the Sync Agent is disabled, it is not allowed to start, which is useful when you only want manual synchronization events.

Constructor

```
BGSession( )
```

Public Methods

The public methods and their parameters for the BGSession class are listed in Table 3–78:

Table 3–78 BGSession Class Public Method Parameters

Method	Description
void Start()	Start the Sync Agent. If the agent is already running, starting, or resuming, this call is ignored. If the agent is paused, this call resumes the Sync Agent. This call is asynchronous and does not wait for the Sync Agent to be started before returning. Use the waitForStatus method to wait for the Sync Agent.
void StartInternal()	Same as Start, but starts the Sync Agent within the application process.
void Pause()	Pauses the Sync Agent. If the agent is aready paused or being paused, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be paused before returning. Use the waitForStatus method to wait for the Sync Agent.
void Resume()	Resumes the Sync Agent. If the agent is already resumed or resuming, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be resumed before returning. Use the waitForStatus method to wait for the Sync Agent.
void Stop()	Stop the Sync Agent. If the agent is already stopped or stopping, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be stopped before returning. Use the waitForStatus method to wait for the Sync Agent.
void Kill()	Forcefully terminate the Sync Agent process instead of stopping it gracefully. Use this option as a last resort if the Stop method is not working because some tasks are hanging in the Sync Agent. Not recommended if the Sync Agent was started with StartInternal.
void Close()	Closes the session and release all the resources used by the session.

Table 3–78 (Cont.) BGSession Class Public Method Parameters

Method	Description
`BGAgentStatus GetAgentStatus()`	Retrieves the current Sync Agent status. See Section 3.2.1.3.4, "BGAgentStatus Object" for more details on the status information returned.
`BGSyncStatus GetSyncStatus()`	Get current status of automatic synchronization managed by the Sync Agent. See Table 3–82 for more details on the status information returned.
`void WaitForStatus (int statusCode)`	Wait for the Sync Agent to reach specified status. You can also wait for a specified timeout.
`boolean WaitForStatus (int statusCode, long timeOut)`	■ The status can be one of the following: `RUNNING`, `PAUSED` or `STOPPED`. ■ The `timeOut` parameter is the maximum time to wait for in milliseconds. Unlimited time if no timeout provided. Returns `TRUE` if the agent has reached specified status; `FALSE` if the timeout has occurred.

Example

The following example demonstrates how to start the Sync Agent, retrieve status of the Sync Agent and add a message handler for the session:

```
// Create the BGSession object
 using(BGsession sess = new BGSession())
 {
  //Start the Sync Agent, which enables all automatic synchronization events
  sess.Start();
  //Wait until the Sync Agent successfully starts
  sess.WaitForStatus(BGAgentStatus.RUNNING);
  //Retrieve the status of the Sync Agent
  BGAgentStatus s = sess.GetAgentStatus();
  //Print out the user that is using automatic synchronization
  Console.WriteLine("User name: " + s.clientId);
  //Add a message handler
  sess.MessageReceived += new BGMessageHandler(myHandler);
  ...
  }
 //When finished, close the session to release all resources
 sess.Close();
```

3.2.1.3.4 BGAgentStatus Object

The `BGAgentStatus` object represents the current status of the Sync Agent.

Public Methods

The methods for the `BGAgentStatus` are listed in Table 3–79.

Table 3–79 BGAgentStatus Class Public Method

Method	Description
`string GetStatusName(BGStatusCode statusCode)`	Get language-specific name of a given status code. When you provide one of the status codes shown in Table 3–76, the appropriate name is returned. Translation dependent on the device language settings

Fields

BGAgentStatus provides status information on the Sync Agent. Table 3–80 lists and describes the status information fields within the BGAgentStatus class.

Table 3–80 BGAgentStatus Class Fields

Parameters	Description
string AppName	The name of the application or process that is executing the Sync Agent. On some platforms, such as Android, it is possible to execute the Sync Agent within an application process.
int BatteryPower	Remaining percentage of battery life, if relevant.
string ClientId	Sync user name.
string NetworkName	Name of the network currently used for synchronization, evaluated by Sync Agent.
int NetworkSpeed	Network bandwidth in bits per second.
int ProcessId	Process id of the process that is executing the Sync Agent, if relevant for a given platform.
BGStatusCode StatusCode	Retrieves the status of the Sync Agent. Status codes that can be returned are detailed in Table 3–76.

Example

The following provides an example of retrieving and processing the Sync Agent status:

```
BGAgentStatus ags = bgSess.GetAgentStatus();
 Console.WriteLine("Agent Status:          " +
   BGAgentStatus.GetStatusName(ags.StatusCode));
if (ags.StatusCode == BGStatusCode.Stopped)
 return;

Console.WriteLine("Client ID:             " + ags.ClientId);
Console.WriteLine("Process Name:          " + ags.AppName);
Console.WriteLine("Process ID:            " + ags.ProcessId);
if (ags.NetworkSpeed > 0) {
  Console.WriteLine("Network Name:          " + ags.NetworkName);
  Console.WriteLine("Network Speed:         " + ags.NetworkSpeed + " bps");
}
else
  Console.WriteLine("Network is not present");
if (ags.BatteryPower > 0)
  Console.WriteLine("Battery Power:         " + ags.BatteryPower + "%");
else
  Console.WriteLine("Battery is not present");
```

3.2.1.3.5 BGSyncStatus Object The BGSyncStatus object provides status information on a current or the last automatic synchronization event in the Sync Agent.

Public Methods

The methods for the BGSyncStatus are listed in Table 3–81.

Table 3–81 BGSyncStatus Class Public Method

Method	Description
`String GetSyncStageName( SyncProgressStage stage)`	Get language-specific name of a given sync status code. When you provide one of the status codes shown in Table 3–76, the appropriate name is returned. Translation dependent on the device language settings

Fields

`BGSyncStatus` provides status information on automatic synchronization in the fields listed in Table 3–82.

Table 3–82 BGSyncStatus Class Fields

Parameters	Description
`DateTime EndTime`	End time of the last synchronization. `EndTime` is set to `DateTime.MinValue` if the synchronization is currently in progress or has not yet run.
`Exception LastError`	Exception object thrown during the last synchronization. Returns `NULL` if the last synchronization was successful or no synchronization has completed yet.
`DataPriority Prio`	Priority of the current or last synchronization.
`SyncProgressStage ProgressStage`	Progress stage of synchronization if it is in progress.
`int ProgressVal`	Progress value in percentage of synchronization, if it is in progress.
`string[] Pubs`	Array of names of publications synchronized currently or during last synchronization.
`DateTime StartTime`	Start time of the last synchronization. `StartTime` is set to `DateTime.MinValue` if the synchronization has not yet run.

Properties

The following documents the properties for the `BGSyncStatus` class.

- Read-write properties read or write the value of a field with `get` and `set` accessors.
- Read-only properties read the value of a field with the `get` accessor.
- Write-only properties set the value of a field with the `set` accessor.

Boolean properties get or set the value of a field to either true or false.

Table 3–83 lists all `BGSyncStatus` properties. Section 3.2.1.3.2, "BGStatusCode Enumeration" lists all enumerations.

Table 3–83 BGSyncStatus Properties

Property	Description
`SyncOccured`	Read-only boolean property. If an automatic synchronizatoin has already occured, `SyncOccured` is `TRUE`.

Table 3–83 (Cont.) BGSyncStatus Properties

Property	Description
SyncInProgress	Read-only boolean property. If automatic synchronization is in progress, SyncInProgress is TRUE.

Example

The following provides an example of retrieving and processing the synchronization status:

```
BGSyncStatus ss = bgSess.GetSyncStatus();
 if (!ss.SyncOccured) return;

 Console.WriteLine();
 Console.WriteLine("Sync Started:          " + dt2str(ss.StartTime));

 if (!ss.SyncInProgress)
   Console.WriteLine("Sync Finished:         " + dt2str(ss.EndTime));
 if (ss.Pubs != NULL && ss.Pubs.Length != 0) {
   Console.Write("Publications synced:   ");
   for(int i = 0; i < ss.Pubs.Length; i++) {
     Console.Write(ss.Pubs[i]);
     if (i == ss.Pubs.Length - 1)
       Console.WriteLine();
     else
       Console.Write(", ");
   }
  }

 Console.WriteLine("Sync Priority:         " + ss.Prio.ToString());
 Console.Write("Sync Result:           ");
 if (ss.LastError == NULL)
   Console.WriteLine("Success");
 else
   Console.WriteLine("Failure: " + ss.LastError.ToString());
 Console.WriteLine();
```

3.2.1.3.6 BGMessageHandler Interface The BGMessageHandler interface enables the Sync Agent and automatic synchronization message and error data to be trapped during synchronization.

The message handler is defined as follows:

```
public delegate void BGMessageHandler(object sender, BGMsgEventArgs args);
```

You can implement a callback function with the same input parameters as the BGMessageHandler and specify it as your callback function. The following is an example of a user-implemented callback function called msgCallback. It processes the error message information available in the Msg parameter that is contained within the BGMsgEventArgs structure.

```
static void msgCallback(object sender, BGMsgEventArgs args)
{
  Console.WriteLine("Time:        " + dt2str(args.Msg.Time));
  Console.WriteLine("Type         " + args.Msg.Type);
  Console.WriteLine("Id           " + args.Msg.Id);
  if (args.Msg.Text != NULL)
     Console.WriteLine("Text         " + args.Msg.Text);
  if (args.Msg.Cause != NULL)
     Console.WriteLine("Cause        " + args.Msg.Cause);
```

```
  Console.WriteLine();
}
```

The following event adds or removes a custom message handler to the Sync Agent.

```
public event BGMessageHandler MessageReceived
```

With the `MessageReceived` event, you can add the message handler as follows:

```
sess.MessageReceived += new BGMessageHandler(msgCallback);
```

Sync calls the `msgCallback` method with a message to report the current state and any errors for the Sync Agent or automatic synchronization. Within the `msgCallback` method, you can perform the appropriate action for the errors returned in the `Msg` variable of type `BGMessage`, which is a public variable in the `BGMsgEventArgs` structure. In addition, `System.EventArgs` is the base class for `BGMsgEventArgs`, which contains event data.

For a complete description of the `BGMsgEventArgs` class, see Section 3.2.1.3.8, "BGMsgEventArgs Class".

3.2.1.3.7 BGMessageType Enumeration The `BGMessageType` enumeration contains the definitions for the different message types.

Syntax

```
public enum BGMessageType
   {
       Info,
       Warning,
       Error
   }
```

Table 3–84 provides more details about these message types.

Table 3–84 BGMessageType

Status Name	Value	Description
`Info`	0	Informational message.
`Warning`	1	Warning message.
`Error`	2	Error message.

3.2.1.3.8 BGMsgEventArgs Class The `BGMsgEventArgs` class contains the event arguments for the `BGMessageHandler` interface. This class encapsulates the `BGMessage` object. Both classes exist in the `Oracle.OpenSync.SyncAgent` namespace.

Syntax

```
public class BGMsgEventArgs : EventArgs
{
       public readonly BGMessage Msg;
}
```

The `BGMessage` class contains error message information passed to handlers when an event occurs within the application.

Table 3–85 BGMessage Class Public Fields

Name	Description
`DateTime Time`	Message creation time.
`BGMessageType Type`	Message type: `INFO`, `WARNING` or `ERROR`. See the Section 3.2.1.3.7, "BGMessageType Enumeration" for details.
`BGMessageId Id`	Error message number. The error code can be one of the `BGMessageId` enumeration values, which are documented in the "BGException Error Messages" in the *Oracle Database Mobile Server Message Reference*.
`string Text`	Message text.
`Exception Cause`	For error messages, optional cause of the error, which can be `NULL`.

3.2.1.3.9 BGException Class The `BGException` class signals an error during execution of .Net Sync Agent Control API. The errors can be recoverable. In addition, the `BGException` class can appear in the `Cause` field in a `BGMessage` object.

The `BGException` class inherits from `OSEException`, which is documented in Section 3.1.1.3.7, "OSEException Class".

The `BGException` read-only properties are listed in Table 3–86.

Table 3–86 BGException Properties

Parameters	Description
`ErrorCode`	Gets the exception error code. The error code can be one of the `BGMessageId` enumeration values, which are documented in the "BGException Error Messages" in the *Oracle Database Mobile Server Message Reference*.
`Kind`	Gets the exception type name.
`InnerException`	Use the `InnerException` property to get underlying cause of the `BGException`.

Constructors

```
BGException(int errCode)

BGException(int errCode, string msg)
```

The parameters for the constructor are listed in Table 3–87:

Table 3–87 BGException Constructor Parameter Description

Parameter	Description
`errCode`	The error code can be one of the `BGMessageId` enumeration values, which are documented in the "BGException Error Messages" in the *Oracle Database Mobile Server Message Reference*.
`msg`	The error message.

3.2.1.4 OCAPI Sync Control APIs

The following sections describe how to start/stop or enable/disable automatic synchronization from the OCAPI Sync Control API for the mobile client:

- Section 3.2.1.4.1, "C/C++ Sync Control APIs to Start or Enable Automatic Synchronization"

- Section 3.2.1.4.2, "Java Sync Control APIs to Start or Enable Automatic Synchronization"

Each of these APIs includes a stop method that has a timeout input parameter. You can supply one of the following values for the timeout, which is a long that specifies a time in milliseconds to wait for any current activity in the automatic synchronization to complete.

- `BG_STOP_TIMEOUT`: A value in seconds that allows the automatic synchronization process to complete before stopping the service. By default, this is set to 5 seconds.
- `BG_KILL_AGENT`: A value of -1 that makes the automatic synchronization service stop immediately, even if it is in the middle of a synchronization. If an automatic synchronization is in process, it will be terminated. NO errors or messages are returned.
- Any long value in milliseconds: If the automatic synchronization does not stop within the time designated, the method returns with an error of `BG_ERROR_TIMEOUT`. At this point, reissue the stop method to terminate the automatic synchronization immediately by supplying `BG_KILL_AGENT` or -1 as the input value.

3.2.1.4.1 C/C++ Sync Control APIs to Start or Enable Automatic Synchronization

The following sections describe the Sync Control APIs for C/C++ applications.

To start or stop the Sync Agent, use the following APIs:

```
olError olStartSyncAgent() ;
olError olStopSyncAgent(long timeout);
```

To enable or disable the Sync Agent, use the following APIs:

```
typedef struct _olSyncOpt {
olBool bDisable;
} olSyncOpt;
olError olGetSyncOptions(olSyncOpt *opt);
olError olSetSyncOptions(const olSyncOpt *opt);
```

The `olGetSyncOptions` and `olSetSyncOptions` methods take a pointer to the `olSyncOpt` structure as a parameter. The `olSyncOpt` structure contains the `bDisable` boolean, which is `true` if the Sync Agent is disabled.

To enable the Sync Agent, perform the following:

```
olSyncOpt opt.bDisable = FALSE;
olSetSyncOptions(&opt);
```

To disable the Sync Agent, perform the following:

```
olSyncOpt opt.bDisable = TRUE;
olSetSyncOptions(&opt);
```

Use `olGetSyncOptions` method to retrieve the current value of the `bDisable` boolean.

3.2.1.4.2 Java Sync Control APIs to Start or Enable Automatic Synchronization

The following `BGSyncControl` class has the following methods:

- `start`—Start automatic synchronization that was previously stopped.

- `stop`—Stop automatic synchronization. Normally, this is used to stop automatic synchronization before a manual synchronization is invoked. Then, use the `start` method to restart automatic synchronization.
- `enable`—Enables automatic synchronization that was previously disabled.
- `disable`—Disables automatic synchronization on a client. Even if the client is restarted, automatic synchronization is not enabled unless you enable synchronization.
- `isEnabled`—Returns a boolean where true states that automatic synchronization is enabled.

```
package oracle.lite.msync;
class BGSyncControl {
  public void start() throws SyncException;
  public void stop(long timeout) throws SyncException;
 void enable();
 void disable();
 bool isEnabled();
}
```

3.2.1.5 JavaScript APIs for the Sync Agent and Automatic Synchronization in PhoneGap

The following sections describe how to manage automatic synchronization through the Sync Agent and how to retrieve status of both the Sync Agent and any automatic synchronization events:

> **Note:** For more details on these classes, refer to the osync.js comments.

- Section 3.2.1.5.1, "Overview"
- Section 3.2.1.5.2, "BGSession Class"
- Section 3.2.1.5.3, "BGAgentStatus Object"
- Section 3.2.1.5.4, "BGSyncStatus Object"

3.2.1.5.1 Overview Once automatic synchronization for the mobile client is enabled, you can manage it through the Sync Agent APIs. The Sync Agent controls and manages all aspects of automatic synchronization, which occurs in the background. If a manual synchronization is started, Sync Agent stops automatic synchronization as indicated and resumes the automatic synchronization activities when the manual synchronization finishes.

The JavaScript interface provides for the following functions:

- Tracking the progress of the automatic synchronization process.
- Retrieve the status of the Sync Agent.

The following are the classes and interface for the JavaScript API for controlling the Sync Agent and automatic synchronization:

- BGSession Class
- BGAgentStatus Class
- BGSyncStatus Class

3.2.1.5.2 BGSession Class BGSession is the main class for controlling automatic synchronization through the Sync Agent, as follows:

- Start, stop, pause, resume, enable or disable automatic synchronization.
- Retrieve automatic synchronization status information.

> **Note:** There should be only a single BGSession object per your PhoneGap application.

Constructors

```
BGSession (success_cbk, error_cbk)
```

Table 3–88 BGException Constructor Parameter Description

Parameter	Description
`success_cbk`	A callback function that gets invoked upon successful API completion.
`error_cbk`	An error callback function that gets invoked on any error in BGSession API.

Public Methods

The public methods and their parameters for the BGSession class are listed in Table 3–89:

Table 3–89 BGSession Class Public Method Parameters

Method	Description
`boolean agentEnabled()`	Returns **TRUE** if the Sync Agent is enabled; otherwise, **FALSE**.
`close()`	Closes the session and release all the resources used by the session.
`enableAgent(boolean on)`	**TRUE** enables the Sync Agent; **FALSE** disables the Sync Agent.
`BGAgentStatus getAgentStatus()`	Retrieves the current Sync Agent status. See Section 3.2.1.5.3, "BGAgentStatus Object" for more details on the status information returned.
int getAgentStatusCode ()	See Table 3–92 for more details on the status information returned.
BGSyncStatus getSyncStatus()	Gets current status of automatic synchronization managed by the Sync Agent. See Table 3–93 for more details on the status information returned.
`pause()`	Pauses the Sync Agent. If the agent is aready paused or being paused, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be paused before returning. Use the waitForStatus method to wait for the Sync Agent.
`resume()`	Resumes the Sync Agent. If the agent is already resumed or resuming, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be resumed before returning. Use the waitForStatus method to wait for the Sync Agent.
`showUI()`	Starts up the Sync Agent UI.

Table 3–89 (Cont.) BGSession Class Public Method Parameters

Method	Description
`start()`	Start the Sync Agent. If the agent is already running, starting, or resuming, this call is ignored. If the agent is paused, this call resumes the Sync Agent. This call is asynchronous and does not wait for the Sync Agent to be started before returning. Use the waitForStatus method to wait for the Sync Agent.Starts up the Sync Agent UI.
`stop()`	Stop the Sync Agent. If the agent is already stopped or stopping, this call is ignored. This call is asynchronous, it does not wait for the Sync Agent to be stopped before returning. Use the waitForStatus method to wait for the Sync Agent.
`waitForStatus (int statusCode)`	Wait for the Sync Agent to reach specified status. You can also wait for a specified timeout.
`boolean waitForStatus (int statusCode, long timeOut)`	▪ The parameter can be one of the following: RUNNING, PAUSED or STOPPED. ▪ The timeOut parameter is the maximum time to wait for in milliseconds. Unlimited time if no timeout provided. Returns TRUE if the agent has reached specified status; FALSE if the timeout has occurred.
`getFatalError()`	If sync agent is in DEFUNCT state, retrieve the error information that caused the bad internal state. See "Table 3–92, " Sync Agent Status Codes" for description of DEFUNCT.

Example

The following example demonstrates how to start the Sync Agent, retrieve status of the Sync Agent and add a message handler for the session:

```
// Create the BGSession object
BGSession sess = new BGSession(onSuccess, onError);
// Create an instance of BGAgentStatus
var BGAgentStatusConst = new BGAgentStatus();
try {
 //Start the Sync Agent, which enables all automatic synchronization
 //events
 sess.start();
 //Wait until the Sync Agent successfully starts
 sess.waitForStatus(BGAgentStatusConst.RUNNING);
 //Retrieve the status of the Sync Agent
 var s = sess.getAgentStatus();
 //Print out the user that is using automatic synchronization
 console.log("User name: " + s.clientId);
 ...
 }
 finally {
 //When finished, close the session to release all resources
 sess.close();
 }
```

3.2.1.5.3 BGAgentStatus Object The BGAgentStatus object represents the current status of the Sync Agent.

Public Methods

The methods for the BGAgentStatus are listed in Table 3–90:

Table 3–90 BGAgentStatus Class Public Method

Method	Description
`String statusName(int statusCode)`	Get language-specific name of a given status code. When you provide one of the status codes shown in Table 3–92, " Sync Agent Status Codes", the appropriate name is returned. Translation is dependent on the device language settings.

Fields

BGAgentStatus provides status information on the Sync Agent. Table 3–91, " BGAgentStatus Class Fields" lists and describes the status information fields within the BGAgentStatus class.

Table 3–91 BGAgentStatus Class Fields

Parameter	Description
`appName`	The name of the application or process that is executing the Sync Agent.
`int batteryPower`	Remaining percentage of battery life, if relevant.
`clientId`	Sync user name.
`networkName`	Name of the network currently used for synchronization, evaluated by Sync Agent.
`int networkSpeed`	Network bandwidth in bits per second.
int processId	Process id of the process that is executing the Sync Agent, if relevant for a given platform.
`int statusCode`	Retrieves the status of the Sync Agent. Status codes that can be returned are detailed in Table 3–92, " Sync Agent Status Codes".

The BGAgentStatus object defines the Sync Agent status codes, which are as follows:

Table 3–92 Sync Agent Status Codes

Status Code	Status Name	Description
0	`STOPPED`	Sync Agent application is not running.
1	`START_PENDING`	Sync Agent is in the process of starting.
2	`RUNNING`	Sync Agent is running. Any tasks within Sync Agent such as synchronization, compose, apply, rule evaluation, network evaluation and other operations can be active.
3	`PAUSE_PENDING`	Sync Agent is in the process of being paused.
4	`PAUSED`	Sync Agent is paused. When paused, none of the tasks within Sync Agent are running. However, resources such as memory and threads, are saved in the case of a speedy resume. Pause and resume are generally faster than start and stop. When a manual synchronization is started, this pauses the Sync Agent until the manual synchronization is completed. At that point, the Sync Agent is resumed.
5	`RESUME_PENDING`	Sync Agent is in the process of resuming.
6	`STOP_PENDING`	Sync Agent is in the process of stopping.

Table 3–92 (Cont.) Sync Agent Status Codes

Status Code	Status Name	Description
7	DEFUNCT	Sync Agent encountered fatal error and is in a bad internal state. Sync Agent's environment needs to be cleaned up and restarted.

Example

The following provides an example of retrieving and processing the Sync Agent status:

```
/* retrieve the Sync Agent status */
var bgStatus = bgSess.getAgentStatus();

/* Print Sync Agent status */
console.log("Agent Status:          " +
 BGAgentStatus.statusName(bgStatus.statusCode));

/* If agent is stopped, return */
if (bgStatus.statusCode == BGAgentStatusConst.STOPPED)
 return;

/* Identify the client id, process id and name */
console.log("Client ID:             " + bgStatus.clientId);
console.log("Process Name:          " + bgStatus.appName);
console.log("Process ID:            " + bgStatus.processId);
/* network name and speed */
if (bgStatus .networkSpeed > 0) {
 console.log("Network Name:          " + bgStatus.networkName);
 console.log("Network Speed:         " + bgStatus.networkSpeed + " bps");
}
else
 console.log("Network is not present");
/* battery power */
if (bgStatus.batteryPower > 0)
 console.log("Battery Power:         " + bgStatus.batteryPower + "%");
else
console.log("Battery is not present");
```

3.2.1.5.4 BGSyncStatus Object The current status of automatic synchronization is if automatic synchronization is in progress, startTime will have a non-zero value and endTime will be zero.

Fields

BGSyncStatus provides status information on automatic synchronization in the fields listed in Table 3–93, " BGSyncStatus Class Fields":

Table 3–93 BGSyncStatus Class Fields

Parameters	Description
`long endTime`	End time of the last synchronization in milliseconds since the standard base time of January 1, 1970, 00:00:00 GMT. Returns zero if the synchronzation is currently in progress or has not yet run.
`lastError`	Exception message string thrown during the last synchronization. Returns NULL if the last synchronization was successful or no synchronization has completed yet.
`int prio`	Priority of the current or last synchronization.

Table 3–93 (Cont.) BGSyncStatus Class Fields

Parameters	Description
`int progressStage`	Progress stage of synchronization if it is in progress.
`int progressVal`	Progress value in percentage of synchronization, if it is in progress.
`pubs`	Comma-separated names of publications synchronized currently or during last synchronization.
`long startTime`	Start time of current or last synchronization, in milliseconds, since the standard base time of January 1, 1970, 00:00:00 GMT. Returns zero if the synchronization has not yet started or the last synchronization time is unknown.

Example

The following provides an example of retrieving and processing the synchronization status:

```
/* Retrieve the synchronization status */
var bgSyncStatus = bgSess.getSyncStatus();


/* start time */
if (bgSyncStatus.startTime == 0) return;
console.log("Sync Started:          " + time2str(bgSyncStatus.startTime));

/* end time */
if (bgSyncStatus.endTime != 0)
console.log("Sync Finished:         " + time2str(bgSyncStatus.endTime));

/* publications synchronized */
console.log("Publications:          " + bgSyncStatus.pubs);

/* synchronization priority */
console.log("Sync Priority:         " +
 (bgSyncStatus.prio == oseSession.PRIO_HIGH ? "High" : "Normal"));
```

3.2.2 OCAPI APIs for Retrieving Status on Automatic Synchronization

> **Note:** The following OCAPI APIs are currently supported for the mobile client, but are not the direction recommended for future applications. To develop applications for future support, migrate existing applications to use the OSE APIs for Managing Automatic Synchronization.

If you want to know at what stage the automatic synchronization cycle is, you can request status from the Sync Agent. In the client application, execute the get status API, which will return immediately with at what stage the automatic synchronization cycle is executing. This is different from the notification message API, which only returns when an event is completed within the synchronization cycle.

The get status API returns a structure that describes this event.

The following sections provide implementation details for each development language:

- Section 3.2.2.1, "Retrieving Status for Automatic Synchronization in Java Applications"

- Section 3.2.2.2, "Retrieving Status for Automatic Synchronization in C and C++ Applications"
- Section 3.2.2.3, "Fields of the Automatic Synchronization Status Structure"

3.2.2.1 Retrieving Status for Automatic Synchronization in Java Applications

Use the `getStatus` method in your Java client application to retrieve status on the automatic synchronization, as follows:

```
public BGSyncStatus getStatus() throws SyncException
```

This method returns the `BGSyncStatus` class with the status information on the automatic synchronization, as follows:

```
public class BGSyncStatus
{
   public String clientId;
   public short  syncState;
   public String syncStateStr;
   public short syncProgress;
   public short lastSyncError;
   public short lastSyncType;
   public Date lastSyncTime;

   public short applyState;
   public String applyStateStr;
   public short applyProgress;
   public short lastApplyError;
   public Date  lastApplyTime;

   public String networkName;
   public int networkSpeed;
   public int batteryPower;
}
```

See Section 3.2.2.3, "Fields of the Automatic Synchronization Status Structure" for a description of the input parameters in the structure.

3.2.2.2 Retrieving Status for Automatic Synchronization in C and C++ Applications

Use the `olGetSyncStatus` method in your C/C++ client application to retrieve status on the automatic synchronization, as follows:

```
olError olGetSyncStatus(olSyncStatus *s);
```

The Sync Agent returns the `olSyncStatus` class, which you provide as an input parameter, with the information on what happened, as follows:

```
typedef struct _olSyncStatus {
    char clientId[BG_MAX_USERNAME];
    ol2B syncState;
    ol2B syncProgress;
    char syncStateStr[BG_MAX_STATUS_STR];
    olError lastSyncError;
    ol2B lastSyncType;
    ol8B lastSyncTime;
    ol2B applyState;
    ol2B applyProgress;
    char applyStateStr[BG_MAX_STATUS_STR];
    olError lastApplyError;
    olU2B _reserved;
```

```
    ol8B lastApplyTime;
    char networkName[BG_MAX_STATUS_STR];
    ol4B networkSpeed;
    ol4B batteryPower;
} olSyncStatus;
```

See Section 3.2.2.3, "Fields of the Automatic Synchronization Status Structure" for a description of the input parameters in the structure.

3.2.2.3 Fields of the Automatic Synchronization Status Structure

The status structure/class have the following fields:

Table 3–94 Status Class Fields

Field	Description
`clientId`	User name
`syncState`	A numeric value that denotes the current synchronization stage, such as compose, send, or receive.
`syncStateStr`	String describing the state, as denoted in the `syncState`, for the automatic synchronization.
`syncProgress`	A percentage that indicates the current progress for the automatic synchronization.
`lastSyncError`	If an error occurred in the last synchronization, this is the error code. If no error, this value is zero.
`lastSyncType`	The priority of the data for the last synchronization. If 1, then high priority data; if 0, then regular priority data was synchronized.
`lastSyncTime`	Time of the last automatic synchronization.
`applyState`	Code that indicates the state for the apply phase.
`applyStateStr`	String describing the state for the apply phase, as denoted in the `applyState` variable.
`applyProgress`	A percentage that indicates the current progress for the apply phase.
`lastApplyError`	If an error occurred in the last apply phase, this is the error code. If no error, this value is zero.
`lastApplyTime`	Time of the last apply phase.
`networkName`	The network name assigned to this network.
`networkSpeed`	Current bandwidth of the network.
`batteryPower`	Current battery power percentage.

3.2.3 OCAPI Notification APIs for the Automatic Synchronization Cycle Status

Note: The following OCAPI APIs are currently supported for the mobile client, but are not the direction recommended for future applications. To develop applications for future support, migrate existing applications to use the OSE APIs for Managing Automatic Synchronization.

You can develop a mobile client application to be notified when an automatic synchronization cycle occurs. The application is notified from the Sync Agent when the automatic synchronization completes as well as when a critical event occurs in the client device. For example, when the device battery runs critically low, Oracle Database Mobile Server can notify the application.

In the client application, create a procedure that executes one of the following message APIs. When your application calls the get message API, it blocks until an event occurs within an automatic synchronization. It returns a structure that describes this event.

The following sections provide implementation details for each development language:

- Section 3.2.3.1, "Automatic Synchronization Notification for C/C++ Application"
- Section 3.2.3.2, "Automatic Synchronization Notification for Java Applications"
- Section 3.2.3.3, "Fields of the Automatic Synchronization Message Structure"

3.2.3.1 Automatic Synchronization Notification for C/C++ Application

Use the `olGetSyncMsg` method in your client application to receive the automatic synchronization notification when implementing for C/C++ applications. In order to block for the status, you need to perform the following:

1. Start the application messaging service with the `olStartSyncMsg` method, providing a queue handle of type `olAppMsgQ`. This message starts the messaging service and returns the queue handle in the `olAppMsgQ`.
2. Execute the `olGetSyncMsg` with the `olAppMsgQ` message handle and the defined `olSyncMsg` structure for the returned automatic synchronization information.

The following provides the method definitions:

```
typedef void *olAppMsgQ
/* start application messaging, get queue handle */
olError olStartSyncMsg(olAppMsgQ *q);
/*Provide the queue handle and block to retrieve automatic synchronization event
*/
olError olGetSyncMsg(olAppMsgQ q, olSyncMsg *m);
```

The `olGetSyncMsg` method blocks until an event occurs, then the Sync Agent returns the `olSyncMsg` class, which you provide as an input parameter, with the information on what happened, as follows:

```
typedef struct _olSyncMsg {
    ol2B type;
    ol2B id;
    char msg[BG_MAX_MSG];
} olSyncMsg;
```

See Section 3.2.3.3, "Fields of the Automatic Synchronization Message Structure" for a description of the input parameters in the structure.

The C/C++ application performs in a different manner than the Java and C# versions in that this creates a message service with its own message queue. Thus, when finished you must perform some cleanup to ensure that the message queue handle is released. Use the `olStopSyncMsg` method to stop the messaging service and release the handle. This must be performed for every message queue that is opened with the `olStartSyncMsg` method.

```
olError olStopSyncMsg(olAppMsgQ q);
```

If you want to force an existing `olGetSyncMsg` to return, use the `olCancelSyncMsg` from another thread in the application. This causes the `olGetSyncMsg` to return with the `BG_ERR_APP_MSG_CANCEL` error.

```
olError olCancelSyncMsg(olAppMsgQ q);
```

3.2.3.2 Automatic Synchronization Notification for Java Applications

Use the `getMessage` method in your client application to receive the automatic synchronization notification when implementing for Java applications, as follows:

```
public class BGSyncControl
{
 public BGSyncMsg getMessage() throws SyncException;
}
```

This method blocks until an event occurs, then the Sync Agent returns the `BGSyncMsg` class with the information on what happened, as follows:

```
public class BGSyncMsg{
    public int type;
    public int id;
    public String msg;
}
```

See Section 3.2.3.3, "Fields of the Automatic Synchronization Message Structure" for a description of the input parameters in the class.

3.2.3.3 Fields of the Automatic Synchronization Message Structure

The message structure/class has the following fields:

Table 3–95 The Sync Message Variables

Variable	Description
Event type	The event can be of three types, each of which indicate the level of severity of this notification: ■ `INFO` ■ `ERROR` ■ `WARNING`

Table 3–95 (Cont.) The Sync Message Variables

Variable	Description
Event identifier for `INFO` types:	The INFO event identifer describes what occurred, as follows: ■ `SYNC_STARTED`: The Sync Agent has started the synchronization task. ■ `SYNC_SUCCEEDED`: Data synchronization completed successfully. ■ `APPLY_STARTED`: The Sync Agent has started the apply task. ■ `APPLY_SUCCEEDED`: The apply phase completed successfully. ■ `SVR_NOTIF`: The Sync Agent has received a server notification. The message contains information about the server notification, such as publication name, number of modified records and the record priority (high priority or normal). ■ `NETWORK_CHANGED`: Device has moved into a different network ■ `AGENT_STARTED`: The Sync Agent started. ■ `AGENT_STOPPED`: The Sync Agent stopped.
Event identifier for the `WARNING` type:	The `WARNING` event identifier describes in more detail what occurred, as follows: ■ `BATTERY_LOW`: Device's battery is running low ■ `MEMORY_LOW`: Device's memory is running low
Event identifier for the `ERROR` type:	The `ERROR` event identifier describes in more detail what occurred, as follows: ■ `APPLY_FAILED`: The apply failed. In this case, 'message' contains the reason for failure. ■ `SYNC_FAILED`: Data synchronization failed. In this case, 'message' contains the reason for failure. ■ `AGENT_ERROR`: An internal error condition occurred. The message contains the actual error message. Examples would be failure to load a rule, failure to process server notification, failure to evaluate system power, and so on. In spite of this error, the Sync Agent continues to execute. Fatal errors are written to the `olSyncAgent.err` file.
Event Message	String message that expounds on the information provided by the event type and identifier.

4

Using Mobile Database Workbench to Create Publications

The following sections describe how to use the Mobile Database Workbench (MDW) to create publications. When using MDW, you first create a project and then create the other objects contained within a publication.

4.1 Use MDW to Create Publications

The Mobile Database Workbench (MDW) tool enables you to iteratively create and test publications—testing each object as you add it to a publication. Publications are stored within a project, which can be saved and restored from your file system, so that you can continue to add and modify any of the contained objects within it.

All work is created within a project, which can be saved to the file system and retrieved for further modifications later. Once you create the project, start creating the publication items, sequences, and scripts that are to be associated with the publication. You can create the publication and associated objects in any order, but you always associate an existing object with the publication. Thus, it saves time to start with creating the objects first and associating it with the publication afterwards.

To launch MDW, execute `oramdw`, which is located in *`$ORACLE_HOME`*`\Mobile\Sdk\bin`.

4.2 Create a Project

Create a new project with MDW. The project is the vehicle that contains your iterative approach to defining publications, publication items, sequences, and scripts. The project can be saved and restored from your file system, so that you can continue to modify any of the contained objects within it.

You cannot perform any action on developing your publications without first creating the project.

You must have access to the back-end database with the mobile server repository and already defined the tables and schema that you are going to be using in your publication items before entering the project wizard.

Perform the following to create the project:

1. Click **File->New->Project** to start the Project Wizard.
2. An Introductory screen appears. If you do not want this introductory screen to display each time you start a new project, check the "Skip This Page Next Time" box.
3. Define the project name. Enter the project name and location for your new project, as follows:
   - Project name: This name can be any valid Java identifier. The name cannot contain any spaces. For example, your project name may be something like `MY_NEW_PROJECT`.
   - Project location: Enter a valid location in the directory on your machine that has write permission to store the project. On a Windows machine, you could enter the location as `c:\myprojects`. To browse for a directory, click **Browse**.
   - The mobile client database type: Select the type of the mobile client, which can be either the SQLite or Berkeley DB.

   Click **Next** to move to the next step in the wizard.
4. Provide the repository access information. Because you are interacting with the repository to create and manipulate synchronization objects, including the SQL scripts for the publication items, you need access to the repository. Enter the following access information:

   User Name and Password

   Specify the mobile server repository user name and password (with administrator privilege). This is the same user name and password with which the repository was originally created. For example, the default mobile server repository user name and password is `mobileadmin/manager`.

   The administrator user name and password are used to connect to the back-end database, create the schema and assign database privileges for the mobile server. In order to perform these actions, the administrator user must have the following privileges:

   – The following privileges are required with the Admin option:

   ```
   ALTER ANY TABLE, ALTER SESSION, ALTER SYSTEM, ANALYZE ANY,
   CREATE SESSION, CREATE ANY SEQUENCE, CREATE ANY VIEW,
   CREATE ANY TRIGGER, CREATE ANY INDEX, CREATE ANY TABLE,
   CREATE ANY SYNONYM, CREATE ANY PROCEDURE, CREATE
   PROCEDURE, CREATE SEQUENCE, CREATE SYNONYM, CREATE TABLE,
   ```

```
CREATE VIEW, CREATE INDEXTYPE, DELETE ANY TABLE, DROP ANY
SEQUENCE, DROP ANY PROCEDURE, DROP ANY VIEW, DROP ANY
SYNONYM, DROP ANY TRIGGER, DROP ANY INDEX, DROP ANY TABLE,
INSERT ANY TABLE, SELECT ANY TABLE, SELECT ANY DICTIONARY,
SELECT_CATALOG_ROLE, UPDATE ANY TABLE
```

Specify the Connection

When you define the connection, you have two options:

- Simple connection definition. For a connection to a single back-end Oracle database, select the **Oracle JDBC Thin driver** and provide the host (or IP address), port and SID for the Oracle database that contains the mobile client repository.

  This creates a connect string which be as follows:

  ```
  jdbc:oracle:thin:@<host>:<port>:<SID>
  ```

- Oracle RAC connection definition. If you are connecting to a Oracle RAC configuration, then select the **Oracle JDBC Thin driver—Advanced**. Selecting this option enables the Connect String field where you can enter the tnsnames connect string that defines all databases included in the Oracle RAC configuration. For example, the following is a tnsnames connect string definition that includes two Oracle databases in the Oracle RAC configuration:

  ```
  (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=-ON)
   (ADDRESS=(PROTOCOL=TCP)(HOST=HOST1)(PORT=1555))
   (ADDRESS=(PROTOCOL=TCP)(HOST=HOST2)(PORT=1555)))
   (CONNECT_DATA=(SERVICE_NAME=ORCL.TW.ORACLE.COM)
   (FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC)(RETRIES=180)(DELAY=5))))
  ```

Click **Next** to move to the next step in the wizard. Once you click Next, the wizard verifies that the database connection information is correct. If incorrect, the wizard prompts you to re-enter the information. You can only advance if you enter the correct information where the mobile server repository is located.

5. Specify the application information, as follows:

   a. Specify the application schema user name and password, each of which are limited to 28 characters. The mobile application schema contains all database tables, views, synonyms used to build the snapshots for the application.

   > **Note:** All schema objects for an application exist in the same back-end repository, which is why the Oracle database host, port and SID are only read-only on this screen.

   b. Use the Database Instance pull-down to select the database where the application is to be deployed. In this database, the application schema be created. You can select the Main database, where the mobile client repository is stored, or any registered remote databases meant solely for application data.

      Once selected, the JDBC URL for the selected database is displayed in the Connect String field.

Click **Next** to move on to the last screen in the Project Wizard. As you click **Next**, MDW verifies that the user name and password that you entered are valid for connecting to the application schema in the back-end database. If these are not valid, you cannot advance until you supply a valid user name and password.

6. A summary page appears. Once the creation of the project is completed, this page displays all of the information about your new project.
   - Click **Back** to modify any of the information supplied.
   - Click **Finish** to complete the project creation.
   - Click **Cancel** to abort creation of this project.

At this point, you can create your publication within this project.

4.3 Use the Quick Wizard to Create Your Publication

The Quick Start Wizard enables you to create a simple publication in just a few steps. It generates the publication items within your publication by assuming that you want the default settings. In addition, the snapshot defaults to select all items within the table. For example, if the table selected is `EMP`, then the select statement defaults to `select * from emp`.

You can associate a publication item in a publication, which is then associated in an application. The publication item is the vehicle that defines the SQL to retrieve data from the database for the application users. When you execute the quick wizard, it creates a publication item for each table you wish to include in the publication. In addition, the wizard defaults the SQL statement used to define the data subset for each table as `select * from <table_name>`.

> **Note:** Since this tool is a quick wizard, it associates a single publication item for each table you include in the publication. In order to create a more complex snapshot—such as one that enables automatic synchronization, creates multiple publication items based on the same table or a more complex SQL statement—see Section 4.4, "Create a Publication Item".

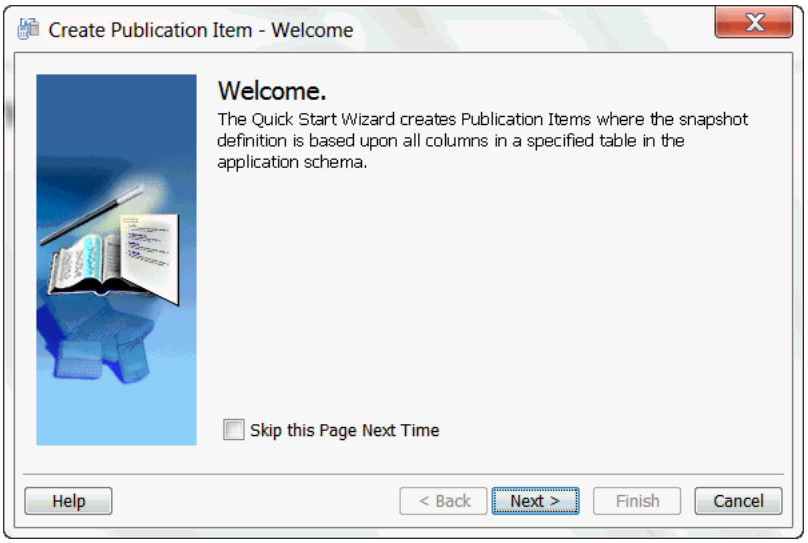
The publication item name defaults to the following: `<table_name>_PI<number>` where `<number>` is sequential between 1 and 9. For example, the first publication item created on table `EMP` would be named `EMP_PI1`. If, in a separate publication, you have already defined a publication item for `EMP_PI1`, then the next time you execute the wizard for the table `EMP`, it be named `EMP_PI2`.

After creating this publication item, this wizard enables you to test it immediately. When the wizard completes, you can always return to the main menu and modify any of the default settings or specify a more specific data subset with your own SQL statement.

For each of the screens in the wizard, click **Next** to advance to the next screen.

1. To start the quick wizard, select the **Quick Wizard** button.
2. An introductory screen appears. If you do not want this introductory screen to display each time you start a new project, check the "Skip This Page Next Time" box.

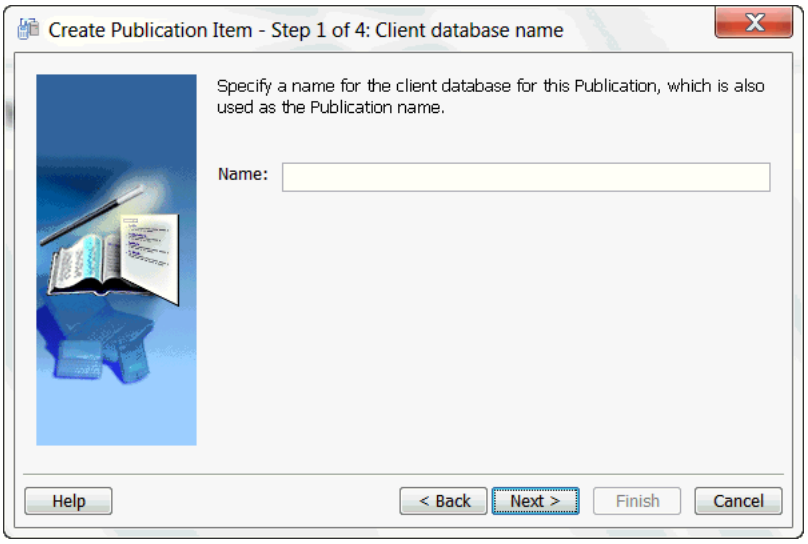
Figure 4–1 Welcome Screen for Quick Wizard to Create a Publication



3. Provide a name for the client database. This is the database that exists on the device to contain the downloaded snapshot information. The name that you choose also be used as the name of the publication.

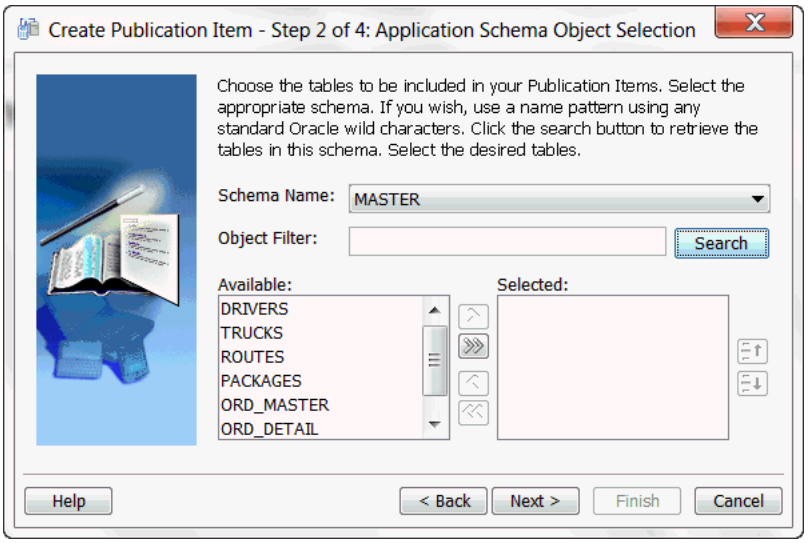
   When this publication is finished, a client database is created on your device and the first synchronization to retrieve the snapshot from the back-end Oracle database is initiated.

Figure 4–2 Define Client Database Name



4. Select the tables to be included in the publication item, as follows:

Figure 4–3 Define the Tables to Include in the Publication



- Choose the application schema to associate with this publication item: The application schema is the schema from which the publication item retrieves data. All available schemas in the database are listed in the pull-down list. You must have created the schema before starting this wizard.

  Note: If the schema you want is not in this list, cancel the wizard, create the schema in the back-end database, and then re-start this wizard.

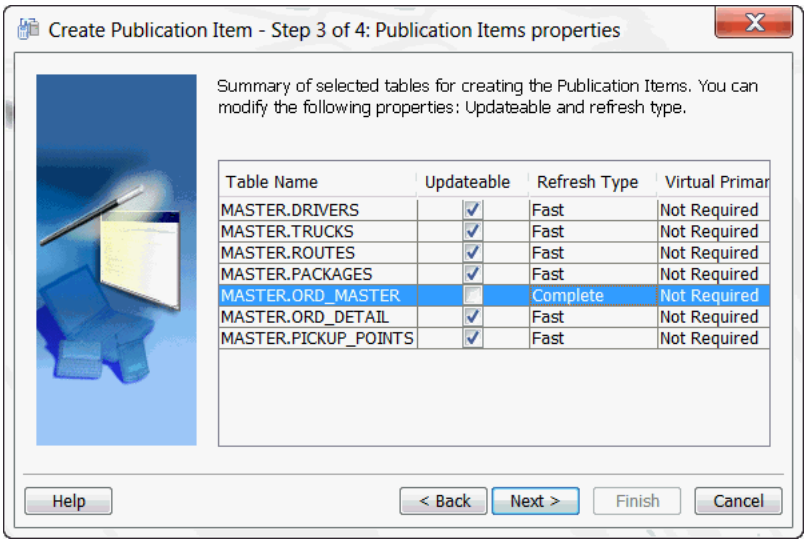
- Click **Search** to display all tables within this schema in the Available column. To search for a specific table or tables, enter the name or partial name with wild charaters in the Object Filter field and then click **Search**. You can use any of the standard Oracle wild card characters.
- Select the tables that you want in the publication item and click the arrow buttons to move one or all tables into the Selected column. You can move these tables back and forth using the arrow buttons.

  Note: If you do not see the object that you expect to see, verify that you created the table in this schema in the back-end Oracle database.

- When you are satisfied with the list of the tables in this publication item, then click **Next**.

5. Once the creation of the publication item is completed, a Summary page displays the defaults used for each table included in this publication item, as follows:

Figure 4–4 Modify the Table Properties for Synchronization



- **Table name**: Displays the schema and name of the table included in this publication item.
- **Updatable**: This is checked if the table is listed as updatable. You can toggle this item to read-only by double-clicking on the field. However, if it is unchecked, you should only enable it if the table has a virtual primary key.

  For more information on Read-Only or Updatable options, see Section 2.3.1.1, "Manage Snapshots".
- **Refresh Type**: By default, all tables use fast refresh. If the table does not have a primary key, then the table uses complete refresh. Double-click on this field to bring up a pull-down with the option to change the refresh type to either fast or complete.

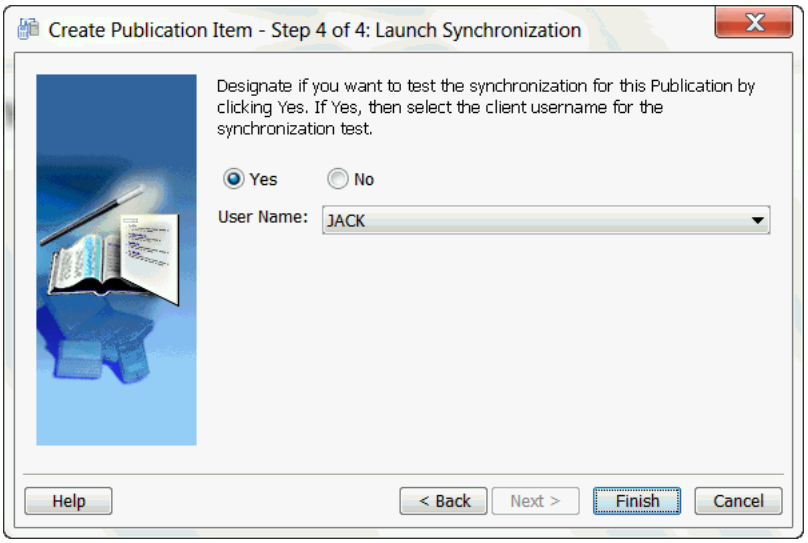
  For more information on Fast or Complete refresh types, see Section 2.8, "Understanding Your Refresh Options".
- **Virtual Primary Key**: This field displays the virtual primary key for the table. If you want to have the table be updatable or use the fast refresh type, then the table must have a virtual primary key. If the table does not have a primary key, but it does contain a field with UNIQUE constraints, then you can specify this field as the virtual primary key to be able to use fast refresh or updatable.

  Note: Any virtual primary key added must be unique and not NULL.

  To specify a column in the table as your virtual primary key, double-click on the Virtual Primary Key field to list all of the UNIQUE fields. If you select one of them to be the virtual primary key, then you can use the Updatable or fast refresh options for this table.

6. Decide if you want to test this publication.

Figure 4–5 Decide to Test the Publication



You can specify that you want to test this publication as soon as the wizard exits. By default, **Yes** is selected. This provides a test of the publication against the back-end Oracle database.

In order to perform this test, a valid client user name must be provided. From the drop-down list, select the client user name that you would like to use. You be prompted for the password during synchronization.

7. You can end the wizard by performing one of the following:
   - Click **Back** to modify any of the information supplied.
   - Click **Finish** to complete the project creation.
   - Click **Cancel** to abort creation of this project.
8. If you clicked **Yes** for testing the publication, then the Test Publication screen is brought up. Click the **Synchronize** button to start the test.

This creates a basic publication, which you can now view in the project in the MDW main screen. You can modify this publication in any way.

4.4 Create a Publication Item

The Publication Item Wizard steps you through the process of creating a publication item in the project. A publication item encapsulates a snapshot definition. It can be based on a table, view or synonym in the Master Application schema in the back-end database. If you use a synonym for a remote object to build a publication item, then you are required to provide the JDBC connection information to the remote database where the remote object resides.

After you create the publication items in the project, then you can associate multiple publication items with a publication, which is then associated with an application. See Section 4.8.2, "Publication Item Tab Associates Publication Items With the Publication" for details.

You can create a publication item through the publication item wizard by clicking **File->New->Publication Item**.

1. The publication item wizard introduction appears. If you do not want this introductory screen to display each time you start a new project, check the "Skip This Page Next Time" box.

   Click **Next** to advance to the next screen.

2. Define name, refresh type, and automatic synchronization, as follows:

   - Publication item name: This name can be any valid Java identifier. The name cannot contain any spaces. Publication item names are limited to twenty-six characters and must be unique across all publications. For example, your publication item name may be something like `MY_PUBLICATION_ITEM`.
   - Refresh type: The refresh mode of the publication item is specified during creation to be either fast or complete refresh. See the Section 2.8, "Understanding Your Refresh Options" for more information.

     From the drop-down list choose one of the following refresh types:

     – Complete: All data is refreshed with current data. Everytime you synchronize, all data in the snapshot is retrieved. This can be performance intensive.

     – Fast: This is the recommended mode. Only incremental changes are synchronized. Thus, you are not downloading the complete data snapshot each time a synchronization is requested. The advantages of fast refresh are reduced overhead and increased speed when replicating data stores with large amounts of data where there are limited changes between synchronization sessions.

     – Queue-based: You can create your own queue. The mobile server upload and download changes from the user. You perform the activity of the MGP and apply/compose the modifications to the back-end database. See the Section 2.12, "Customizing Synchronization With Your Own Queues" for more information.

     Once you create the publication item with a particular refresh type, the only way to modify the publication item to have a different refresh type is to delete is and recreate it with the desired refresh type.

   - Enable Automatic Synchronization checkbox: This defines the publication item to use Automatic Synchronization, where synchronization for this publication item occurs automatically and in the background. That is, you do not have to manually press the Sync button as it occurs automatically.

     In a publication, all publication items should either be enabled or disabled for the automatic synchronization. Mixing the two types in a single publication can potentially break the publication's transactional consistency.

Note: If, after you have published the application, you want to turn off automatic synchronization, you can only disable automatic synchronization as follows:

1. Execute the API to disable the automatic synchronization. Use the `reCreatePublicationItem` method to disable automatic synchronization. For more information on modifying a publication item using the APIs, see Section 2.4.1.12, "Modifying a Publication Item".
2. Execute a manual synchronization on the device.The manual synchronization restarts the automatic Sync Agent, which then use the new rules The new settings NOT be downloaded automatically during automatic synchronization.

Step 7 shows you how to specify—with a SQL statement—which users receive the automatic synchronization. Section 4.5, "Define the Rules Under Which the Automatic Synchronization Starts" shows you how to define automatic synchronization rules that apply to this publication item.

3. Designate the publication item object with the appropriate schema, as follows:
   - Choose the application schema to associate with this publication item: The application schema is the schema from which the publication item retrieves data. All available schemas except mobile repository user schema you defined before and system schema of Oracle database are listed in the pull-down list. You must have created the schema before creating the publication item.

     If the schema you want is not in this list, cancel this publication item, create the schema in the back-end database, and then associate the schema with the publication item.
   - Designate the object type as a table, view, or synonym.
   - To choose the table, view or synonym from within the schema that you wish to base this publication item on, click **Search** on the Object Filter. This brings up several items in the Object List. Select the object that you are interested in and click **Next**.

     Note: To search only for objects that match a condition, designate the condition in the Filter box and click **Search**. You can use the same pattern matching characters in a valid `SQL WHERE` clause. The filter is case-sensitive; use upper-case characters.

     ---

4. Choose columns to add to the publication item. When you are adding columns to the publication item, you should first verify what data types are supported and how others are modified when brought down to the client database.

   There are two tabs to enable you to structure your publication item, as follows:
   - Column Selection: Choose the columns from within the object that you use to retrieve information for the application. To choose the appropriate columns, select the column name and click the left or right arrow buttons to move between the Available and Selected windows. To move all columns, use the double arrows.

     Note: The primary key defaults to being in the Selected window, as it is required if you are using Fast Refresh and Updateable option. Since most publication items use these options, MDW places the primary key as Selected. You can move it back to the Available window.

     The publication item query must select primary keys in the same order as they are defined in the base table.

   - Structure: If you are not sure what columns you want, you can see the entire table structure by clicking this tab.

> **Note:** Oracle Database Mobile Server does not support creating publication items for a table with object type columns, even if the publication item query does not include any of the object type columns. However, it is possible to define a view which selects only columns of supported data types and then create a publication item using the view definition.

The Oracle Database Mobile server does not support creation of publication items for a table with NVARCHAR2 data type. If you want to use national character set in your table, do the following:

- create a table with column type VARCHAR2, not NVARCHAR2
- set Oracle database character set as UTF8 or AL32UTF
- use DB_CHAR_ENCODING=UTF8 in the client side.

If you have specified a fast refresh, you must provide a primary key. If you have specified a table or view that does not have a primary key, exit out of this wizard and create a virtual primary key specifying one of the columns in the table or view. If you do not create a virtual primary key before specifying this publication item, then any future synchronization of this primary key fail.

> **Note:** Any virtual primary key must be unique and not NULL.

5. Create indexes and associate them with the publication item. These indexes display in the Index Selection window. The index name, index type and the index columns are shown. If you want to add or remove an index in the publication item, use the following options:

> **Note:** You can only modify an index by removing and creating it again.

   - Add: Click **Add** to create a new index and associate it with the publication item. You need to provide the index name, type and snapshot columns for the new index.
   - Remove: Removes an existing index from the publication item by selecting the index in the Index Selection window and clicking **Remove**.

   By default, Oracle Database Mobile Server creates a primary key index for every publication item and is created on the primary key of the snapshot table. This index is named `<piName>_PK`. You cannot remove the primary key index from the publication item within this screen. If you want to remove the primary key index, use the `dropPublicationItemIndex` method. For more details, see Section 2.4.1.6, "Create Publication Item Indexes".

   If a snapshot is based on a table without a primary key, then the primary key index not be created and the Index Selection box is empty.

   When the desired indexes have been added or removed for this publication item, click **Next** to advance to the next window.

6. Modify the SQL statement for the publication item. From the columns that you selected in the previous screen, this simple SQL statement is available as a

template for you to modify. You can add any qualifiers and complexity to this base statement. To view the structure of the schema object, select the Structure tab.

- Perform Iterative Modifications

  See Section 4.4.1, "Create SQL Statement for Publication Item" for directions on how to edit and execute the SQL statement for this publication item.

- Apply/Compose Callbacks

  When creating publication items, the user can specify a customized package to be called during the Apply and Compose phase of the MGP background process. Client data is accumulated in the in queue before being processed by the MGP. Once processed by the MGP, data is accumulated in the out queue before being pulled to the client by Mobile Sync. See Section 2.7.2, "Customize What Occurs Before and After Compose/Apply Phases for a Single Publication Item" for more information on how to create the callbacks.

  Provide the schema and package names for any apply/compose callbacks.

- Dependency Hint

  Click **Add** to add a dependency hint. All existing dependency hints are shown in the window. See Section 4.4.2, "Create a Dependency Hint" for more information.

7. If you specified a view, then you may need to define parent table and primary key hints. See Section 4.4.3, "Specify Parent Table and Primary Key Hints" for directions on how to define these hints.
8. If you specified automatic synchronization for this publication item, then the Automatic Synchronization Query page is shown, which includes the following:

   - By default, all users are included in the Compose, which means that all users receive the data that is being retrieved from the server and brought down to the mobile clients. If you want to specify that only certain users that subscribe to this application receive the application data, then check this box.
   - If you clicked the checkbox, then specify which users are to receive the Compose data with a SQL query.

     For example, if you only want `MADAUSER` within the `mobileadmin` schema to receive the Compose data, then type in `select name from mobileadmin.users where name ='MADAUSER'`. Click **Run** to verify that the SQL query returns what you want; click **Next** to advance to the next page.

9. Summary page provides an overview of the publication item that you just created. To view any dependency hints, click **Advanced**. If you have used a view as the base for your publication item and you created Parent Table or Primary Key hints, click Hint to view these hints. The Parent Table and Primary Key hints are only valid on views.

   To modify any part of the publication item, click **Back** to return to the previous screens.

   Click **Finish** if you are satisfied with the publication item. Click **Cancel** to eliminate all work in creating this publication item.

4.4.1 Create SQL Statement for Publication Item

You can compose your SQL statement through iterative steps to ensure that you are creating a valid statement. You have the following options:

- Modify the query by clicking **Edit**.
- Execute the statement against the schema in the back-end database by clicking **Run**. You be notified directly if the statement fails or view the data from the schema object is retrieved for you to view.

  When you click **Run**, then the query of the publication item is validated by executing the query against the back-end database. A dialog is displayed with the returned snapshot that this publication item would generate. If there is more than one page, then click **Previous** and **Next** to move between the pages. Click **Close** to return to the publication item screen.

  If the publication item SQL statement requireds an input value, then a dialog appears for you to input the value for the SQL query. You can modify this value or the SQL query until you receive the results that you desire.
- To return the query to the original statement, click **Reset**.
- When finished, click **Next**.

4.4.2 Create a Dependency Hint

If the updates to this publication item effects another table, use the dependency hint to notify the synchronization engine to update the other table. For example, if the publication item updates the employee table, and these updates should also apply to the department table, add a dependency hint notifying the synchronization engine of the relationship with the department table.

For your dependency hint, specify whether the hint is based upon a table or synonym. Then, use the pulldown lists to select the schema and table/synonym names. Click **OK** to save the hint or **Cancel** to return to the Advanced screen.

4.4.3 Specify Parent Table and Primary Key Hints

When you use a view, you may need to specify a parent table and primary key hints. A view can be composed of one or more tables joined together. If you have specified fast refresh, then you must specify which table is the parent table and which column is the primary key.

- To create a parent table hint, select the base table from the Base Tables drop-down list and check the Parent Table Hint checkbox.

  Note: If you do not check the Parent Table Hint checkbox, then the hint is not created when you click **Next**.

- To create a primary key hint, click **Add**. Identify the primary key hint for this view, as follows:

  1. From the View Columns drop-down list, select the view column that you want to be the primary key. This column name may be an alias of the actual `table.column` name.
  2. From the Base Tables drop-down list, select the base table where the actual column exists that is to be the primary key.
  3. From the Primary Key Columns drop-down list, all primary key columns from the base table are shown, select the appropriate column for the primary key. If you have a composite primary key, iteratively add each column within the composite primary key.

> **Note:** If you do not provide accurate details in regards to the view, base table, and primary key to which the view column maps, the publication item may save, but the execution of the publication item fail. For example, if you choose a view column which does not map to the primary key column, MDW allow the action, but any execution of the publication item result in failure.

Click **OK** to accept this primary key hint.

4.5 Define the Rules Under Which the Automatic Synchronization Starts

Once you have enabled a publication item to use automatic synchronization, you must define the rules under which the automatic synchronization executes. The circumstances under which an automatic synchronization occurs is defined within the synchronization rules. There are two types of automatic synchronization rules: events and conditions. If an event is true, it starts a synchronization; however, the synchronization cannot occur unless all conditions are true, as well. This evaluates as follows:

```
when EVENT and if (CONDITIONS) then sync;
```

If an event is true, then a synchronization can start—but only if all conditions are true.

Thus, event and condition rules are as follows:

- Events—An event is variable, as follows:
  - Data events: For example, you can specify that a synchronization occurs when there are a certain number of modified records in the client database.
  - System events: For example, you can specify that if the battery drops below a predefined minimum, you want to synchronize before the battery is depleted.
- Conditions—A condition is an aspect of the client that needs to be present for a synchronization to occur. This includes conditions such as battery life or network availability.

For example, if the event for new data inserted and the condition specified is that the network must be available, then a synchronization only occurs when the network is available and there is new data.

When you define the rules for the synchronization, you can define them in two places:

- Publication level: You specify the rules under which the synchronization occurs at the publication level for all publication items in that publication.
- Platform level: Some of the rules are very specific to the platform of the client, such as battery life, network bandwidth, and so on.

> **Note:** This section describes how to do this through MDW; see Section 2.2.2, "Define the Rules Under Which the Automatic Synchronization Starts" for directions on how to perform this programmatically.

If after defining these rules and publishing the application, you want to modify the rules, you can do so through MDW. However, you must perform a manual synchronization. The manual synchronization restarts the automatic Sync Agent,

which then use the new rules The new settings NOT be downloaded automatically during automatic synchronization.

The following sections detail all of the rules you can configure for automatic synchronization:

- Section 4.5.1, "Configure Publication-Level Automatic Synchronization Rules"
- Section 4.5.2, "Configure Platform-Level Automatic Synchronization Rules"

4.5.1 Configure Publication-Level Automatic Synchronization Rules

When you are creating the publication, you can define data events that cause an automatic synchronization. Although these are defined at the publication level, they apply only to the publication items within this publication that have automatic synchronization enabled.

For full details of how to configure these data events, see Section 4.8.5, "Event Tab Configures Automatic Synchronization Rules for this Publication".

Table 4–1 describes the publication level data events that trigger automatic synchronization. The lowest value you can specify is 1.

Table 4–1 Automatic Events for the Publication

Events	Description
Client commit	Upon commit to the client database, the mobile client detects the total number of record changes in the transaction log. If the number of modifications is equal to or greater than your pre-defined number, automatic synchronization occurs.
Server MGP compose	If after the MGP compose cycle, the number of modified records for a user is equal to or greater than your pre-defined number, then an automatic synchronization occurs. Thus, if there are a certain number of records contained in an Out Queue destined for a client on the server, these modifications are synchronized to the client.

Note: If you want to modify the publication-level automatic synchronization rules after you publish the appliation, you can do so through the Mobile Manager, as follows:

1. Click **Data Synchronization**.
2. Click **Repository**.
3. Click **Publications**.
4. Select the publication and click **Automatic Synchronization Rules**.

4.5.2 Configure Platform-Level Automatic Synchronization Rules

The platform-level synchronization rules apply to a selected client platform and all publications that exist on that platform. You can specify both platform events and conditions using either MDW or the Mobile Manager. This section describes MDW; see Section 5.5.1, "Specifying Platform Rules for Automatic Synchronization" in the *Oracle Database Mobile Server Administration and Deployment Guide* for directions on how to define these rules using Mobile Manager.

To assign platform-level automatic synchronization rules, perform the following in MDW:

1. Click **Platform**.

2. Select either the Win32, WINCE, Linux or JAVA platform, which brings up a page with two tabs: Events and Conditions. These rules apply to all publications for this platform.
3. You can modify the following for each platform:
   - Event Rules—See Section 4.5.2.1, "Define System Event Rules for the Platform".
   - Conditions—See Section 4.5.2.2, "Define Automatic Synchronization Conditions for the Platform".

Note: You can only modify the network settings for the platform using Mobile Manager, see Section 5.5.1, "Specifying Platform Rules for Automatic Synchronization" in the *Oracle Database Mobile Server Administration and Deployment Guide* for more information.

4.5.2.1 Define System Event Rules for the Platform

When you choose the Event tab, select the checkbox for each event that you want to enable. If the event requires a value, enter the value you desire. This initiates the automatic synchronization the first time the event occurs. For example, if the battery runs below the percentage you specified, the automatic synchronization occurs. As the battery continues to deplete, you not trigger another synchronization.

The following system events trigger an automatic synchronization if true.

- Network Bandwidth: Synchronize when the network bandwidth is greater than `<number>` bits/second. Where `<number>` is an integer that indicates the bandwidth bits/seconds. When the bandwidth is at this value, the synchronization occurs.
- Battery Life: Synchronize when the battery level drops to `<number>`%, where `<number>` is a percentage. Often you may wish to synchronize before you lose battery power. Set this to the percentage of battery left, when you want the synchronization to automatically occur.
- AC Power: Synchronize when the AC power is detected. Select this checkbox if you want the synchronization to occur when the device is plugged in.

4.5.2.2 Define Automatic Synchronization Conditions for the Platform

When you choose the Condition tab, you can set under what conditions the automatic synchronization is allowed or disallowed, as follows:

- Battery Level: Specify the minimum battery level required in order for an automatic synchronization to start. The battery level is specified as a percentage.
- Network Availability: Network quality can be specified using several properties. For example, if you have a very low network bandwidth and a high ping delay, you may only want to synchronize your high priority data. To add network quality condition for a specified data priority, click the **Add** button, which brings up a screen where you can specify a minimum value for the following network properties:
  - Data Priority: You could have defined records in the snapshot with a data priority number. Use this condition to specify under what conditions the different data priority records are synchronized. Data priority can be either one or zero, where zero is high priority. By default, all records are entered with a value of NULL, which is the lowest priority.

Note: You can only use fast refresh with a high priority restricting predicate. If you use any other type of refresh, the high priority restricting predicate is ignored.

See Section 1.2.10, "Priority-Based Replication" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for more information.

- Minimum Network Bandwidth (bits/sec): Configure the minimum bandwidth (bits/second) in which the automatic synchronization can occur for records with this data priority.
- Maximum Ping Delay (ms): Configure the maximum ping delay (milliseconds) in which the automatic synchronization can occur for records with this data priority.
- Include Dial-up Networks?: The always-on network is used if available. However, if this network is not available, select **YES** if you want to use any of the dial-up networks for this data priority.

4.6 Create a Sequence

Note: Sequences are supported only within Berkeley DB. The only support for sequences on a SQLite Mobile Client is to emulate sequences in a replicated environment. For details, see Section 3.1.1.1.6, "Sequences Emulated for SQLite Mobile Clients in Replicated Environment".

A sequence is a database object, from which you can generate unique integers. You can use sequences to automatically generate primary key values. However, when you have multiple clients accessing a single server, you need a method to guarantee unique identifying numbers for new records from multiple clients. Oracle Database Mobile Server provides a method for unique sequence numbers.

After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. These unique integers can include primary key values. If a transaction generates a sequence number, the sequence is incremented immediately whether you commit or roll back the transaction.

For Oracle Database Mobile Server, you can add a sequence to a publication; then, the sequence is created on all subscribed clients during the initial synchronization. On each client database, the sequence can be used independently. However, since the sequences are used to generate primary key and unique key values for snapshot tables, it is important to ensure that different clients do not generate the same sequence values. If they did, then conflicts may occur when the clients synchronize their changes to the server tables.

The Sync Server guarantees uniqueness across all clients. During synchronization, the Sync Server assigns separate sequence ranges, known as a window, to each client when necessary. A client cannot increment a sequence beyond its current window. Once a client exhausts its window, the Sync Server assigns a new window on the next synchronization. All windows are unique and never reassigned.

Since the sequence windows are obtained from the Sync Server only during synchronization, there is a chance that the client could exhaust all available sequence numbers in its window in between synchronization events. To prevent this from

happening, the administrator can configure clients to obtain a new window before the current one is exhausted by setting the threshold value. A threshold is less than the window size. If the range of values left in the window is less than the threshold size, then during the next synchronization, a new window be assigned to the client. For example, you set the window to 200 and the threshold to 25. During a synchronization event, the Sync Server notices that the current sequence value is greater than or equal to 175, then it allocates the next window for the client.

The following describes how to use Oracle Database Mobile Server sequences:

- Only clients use the sequence: If you have more than a single client, you want each client to use a specific range of unique sequence numbers, so that none of the records have duplicate sequence numbers.

  Specify the start value and the window size. When you define the size of the window, you provide the Sync Server the number of assigned identifiers for each client and the range of values never overlaps with those of other clients.

- Server and clients use the same sequence: If you want the server and one or more clients to share a sequence, then the server and the client use every other number—the identifiers are generated where the server uses all even numbers and the clients use odd numbers—or vice versa, if the start value of the sequence is an even number. Specify the start value, window size and select the "Generate Server-side Sequence" option that tells Oracle Database Mobile Server to generate a server-side sequence. The increment value always defaults to 2 for this case, even if you specify another number. If you have more than one client, configure the window size to ensure that the client sequence numbers do not conflict.

4.6.1 Configuring Sequences in MDW

Within Oracle Database Mobile Server, you configure how you want sequence numbers generated in the sequence definition. In MDW, create a sequence by clicking **File->New->Sequence**. When you are creating your publication, configure the following values to instruct how the sequence is generated for all clients and the server:

- Name: This sequence name must be a valid database identifier.
- Starts With: Enter the number with which you want this sequence to start.
- Increment: Specify the increment from the starting value for the next value in the sequence.

  Note: If you have checked the **Generate server-side sequence** checkbox and set the increment value to 1, then this value is ignored and is set to 2. When you specify the server-side sequence, then both the client and the server use every other number in the sequence. Thus, you cannot increment by 1 on the client.

- Window Size: Provide the size of the window that the Sync Server assigns to the client. For each client, the Sync Server assigns the initial range of sequence values at the time of subscription. For example, if you set the window size to 100 and the Starts With value to 1, then the Sync Server assigns the client windows as follows:
  - Client A: sequence numbers 1 through 100
  - Client B: sequence numbers 101 through 200
  - Client C: sequence numbers 201 through 300

If any client exhausts their window, they are assigned another 100, which is the defined window size in our example, during the next synchronization. If you also click the **Generate Server-Side Sequence** checkbox, then the sequence numbers used by the clients are the odd numbers in their range, such as 1, 3, 5, 7 and so on.

- Threshold: When the number of identifiers left in the window is less than the threshold, a new window of sequence numbers is assigned to the client on the next synchronization. For example, if you have a window size of 200 and threshold of 25, then when the current sequence number is equal to or greater than 175, the Sync Server assigns the next window of values to the client.

  Note: If the window size is 100 and the threshold is 25, then no matter what the increment is, the next window is assigned when the sequence numbers are equal to or greater than 75. It is based on the window size, not on the number of sequence values left for the client.

- Description: A description of the sequence.
- Generate server-side sequence: If you want the client and the server-sides to share a sequence, where one side has all even numbers and the other has the odd numbers, check this box. If unchecked, then the sequence is created solely for the client.

  When you check the Generate Server-Side Sequence checkbox, then no matter what value is specified for Increment, it always be set to 2. A sequence of the supplied name is created automatically on the server in the `mobileadmin` schema to use all even numbers. Specify a 1 as the Starting Value, so that on the server side, the sequence uses even values starting with 2 and on the clients, the odd values are used. Thus, the server and client sequence values are unique.

  If there are multiple clients, then to ensure that the clients use unique numbers, set up separate windows for each client. There is no window for the server, because the server uses all even numbers in the whole range of the sequence.

  For example, the sequence number for the first client starts at 1 and increments by 2 for all of its sequence numbers. The first client still has a window size, which in this example is 100, but it starts with an odd number within that window and always increments by 2 to avoid any even numbers. Thus, client A has the window of 1 to 100, but the sequence numbers would be 1, 3, 5, and so on up to 99.

Oracle Database Mobile Server creates and maintains the sequence based on the sequence definition in the publication. Once you create a sequence in the project, you can associate it with a publication. See Section 4.8.3, "Sequence Tab Associates Existing Sequences With the Publication" for details.

See the Section 2.4.1.8, "Creating Client-Side Sequences for the Downloaded Snapshot" for more information on sequences.

4.6.2 Configuration Scenarios for Sequence Generation

When setting up a sequence, you can configure one of the following three scenarios:

- Multiple Clients: In this case, always define Start Value and the Window Size parameters. When you define the size of the window, you provide the Sync Server the number of assigned identifiers for each client and the range of values never overlaps with those of other clients. Also, set the starting value for each client.

- Server and Clients Use Same Sequence: If you want the server and one or more clients to share a sequence, select the "Generate Server-Side Sequence" checkbox. If you have multiple clients, set the Start Value and the Window Size. The checkbox tells Oracle Database Mobile Server to create a sequence on the server side, where the clients use the Start Value and the server uses Start Value +1. The identifiers are generated where the server or the client uses either all odd numbers or even numbers. The window ensures that the clients do not use the same sequence window.

4.6.3 Example of a Sequence

For this example, the sequence is defined as follows:

Table 4–2

Parameter	Definition
Name	`audiodb_seq`
Start Value	1
Increment	1
Window size	200
Threshold	25

The first client starts at 1 with an increment of 1. The full range of sequential values provided to client 1 is 200 and a new set of sequential numbers is assigned during synchronization by the Sync Server when the unused portion of the window is less than 25.

Oracle Database Mobile Server creates the sequence locally on the mobile client by executing the following SQL statement:

```
create sequence audiodb_seq start with 1 maxvalue 200 increment by 1;
```

On the second client, the Sync Server adjusts the numbers appropriately to accommodate what was created on client 1 and creates the sequence locally with the following SQL statement:

```
create sequence audiodb_seq start with 201 maxvalue 400 increment by 1;
```

During each synchronization, the Sync Server tracks the number of assigned windows to ensure that each client has a unique range. When the Sync Server assigns a new set of sequential numbers as identifiers for the client, it recreates the sequence, as follows:

```
drop sequence audiodb_seq;
create sequence audiodb_seq start with 401 maxvalue 600 increment by 1;
```

4.6.4 Example of a Client and Server Sharing a Sequence

You can define a sequence to provide unique sequence values by assigning all odd or even sequence numbers to either the client or the server. The value specified in the Start Value sets the starting value for the clients. If the server is sharing a sequence with a client, then the start value also determines the values for the server. If the starting value is odd, then the server use all even numbers; if the starting value is even, then the server uses all odd numbers.

The following example demonstrates how to set up a sequence where the odd numbers are for the client and the even numbers for the server.

Enter the following sequence definitions for the client in MDW when defining the publication:

Table 4–3

Parameter	Definition
Name	`audiodb_seq`
Start Value	1
Increment	2
Window size	200
Threshold	25
Generate server-side sequence	Check on

The sequence on the server starts at 2 and uses all even numbers; within the publication, you specified that all clients use odd numbers starting at 1.

4.7 Create and Load a Script Into The Project

You can add a script to this project. Create the script on your file system and then upload it to MDW. Before you add the script to the project, you can use MDW to test the script. See the following sections for more information:

- Section 4.7.1, "Writing SQL Scripts"
- Section 4.7.2, "Load the Script Into the Project"

4.7.1 Writing SQL Scripts

When you write and upload a SQL script to the project, each script is executed independently by Oracle Database Mobile Server in no specified order. Therefore, if you have dependencies and need the DDL statements to be executed in a certain order, include all statements in the correct order in a single script, where each DDL statement is separated by a semicolon (";").

Alternatively, you can specify the weight for the script when loading them to specify the order in which each script is executed on the client. See Section 4.7.2, "Load the Script Into the Project" for more details.

If a SQL script fails upon execution, Oracle Database Mobile Server execute it once more, in case the failure was due to a dependency of a later script. However, if you have a script with a dependency on another script, you could effect your performance while Oracle Database Mobile Server re-executes all of the scripts to resolve dependencies.

Note: If you upload scripts using one of the Consolidator APIs, you must also ensure that the order of execution for these scripts does not matter. Include all dependent DDL statements in a single script and in the order necessary for clean resolution.

4.7.2 Load the Script Into the Project

Define the script on your machine. Once defined, perform the following:

1. Bring the script into the project by clicking **File->New->Script**.

2. Provide a user-defined name to identify the script and browse for the script in your file system.
3. Specify the weight, if necessary. You can specify the weight for the script when loading them to specify the order in which each script is executed on the client. For example, when creating a master detail table on the client, you must create first the master table and then the detail table. The client does not know which script should be executed first, unless you specify a weight to let the client know the order in which to execute the scripts.
4. Click **OK** to accept the definition and **Cancel** to return to the previous screen.

Once you include a script in the project, you can associate it with a publication. See Section 4.8.4, "Script Tab Associates Existing Scripts With the Publication" for more information.

4.8 Create a Publication

Create a publication by clicking **File->New->Publication**. You can create the publication at any time. This starts the dialog for creating a publication.

There are six tabs included for configuring information about the new publication. On configures general information about the publication, one defines event rules for automatic synchronization, and the others enable you to associate different objects with the publication.

If you click **OK**, then you can associate the objects by selecting the publication name and then selecting the appropriate tab.

When you are finished creating the publication, click **File->Save** to save the publication.

- Section 4.8.1, "General Tab Configures Publication Name"
- Section 4.8.2, "Publication Item Tab Associates Publication Items With the Publication"
- Section 4.8.3, "Sequence Tab Associates Existing Sequences With the Publication"
- Section 4.8.4, "Script Tab Associates Existing Scripts With the Publication"
- Section 4.8.5, "Event Tab Configures Automatic Synchronization Rules for this Publication"

4.8.1 General Tab Configures Publication Name

The General tab provides the following information about your new publication within your project:

- Publication name: Enter a valid Java identifier for the publication name. The name cannot contain any spaces or special characters.
- Optional description: You can add a description to remind you of the content of this publication.
- Client database name: This defaults to the same name as the publication name. However, you can modify it. The purpose of this name is to specify the name of the mobile client database, which is created during the first synchronization.

4.8.2 Publication Item Tab Associates Publication Items With the Publication

Selecting the Publication Item tab from within the publication enables you to associate any existing publication item to this publication.

Manage Publication Items In This Publication

- To add an existing publication item to this publication, Click **Add**.
- To remove a publication item from this publication, select the desire publication item from the list and click **Remove**.
- To edit the details of the association for the publication item, select the desired publication item and click **Edit**.

To accept the current changes, click **OK**.

4.8.2.1 Associating a Publication Item to this Publication

To associate any publication item to this publication, the publication item must first exist. Thus, all of the information requested on this screen is about existing publication items.

Provide the following information to identify the publication item to associate to this publication:

Identify Existing Publication Item

From the Name drop-down list, select the name of the publication item.

Updatable or Read-Only Snapshot

Select if the snapshot is updatable or read-only. See Section 2.3.1.1, "Manage Snapshots" for more details.

- Read-only snapshots are used for querying purposes. Changes made to the master table are replicated to the snapshot by the mobile client.
- Updatable snapshots provide updatable copies of a master table. You can define updatable snapshots to contain a full copy of a master table or a subset of rows in the master table that satisfy a value-based selection criteria. You can make changes to the snapshot which the Mobile Sync propagates back to the master table.

  A snapshot can only be updated when all the base tables that the snapshot is based on have a primary key. If the base tables do not have a primary key, a snapshot cannot be updated and becomes read-only.

Conflict Resolution

When adding a publication item to a publication, the user can specify winning rules to resolve synchronization conflicts in favor of either the client or the server. A mobile server synchronization conflict is detected under any of the following situations:

- The same row was updated on the client and on the server.
- Both the client and server created rows with equal primary keys.
- The client deleted a row and the server updated the same row.
- The client updated a row and the server deleted the same row. This is considered a synchronization error for compatibility with Oracle database advanced replication.
- For systems with delayed data processing, where a client's data is not directly applied to the base table (for instance in a three tier architecture) a situation could

occur when first a client inserts a row and then updates the same row, while the row has not yet been inserted into the base table. In that case, if the DEF_APPLY parameter in C$ALL_CONFIG is set to TRUE, an INSERT operation is performed, instead of the UPDATE. It is up to the application developer to resolve the resulting primary key conflict. If, however, DEF_APPLY is not set, a "NO DATA FOUND" exception is thrown (see below for the synchronization error handling).

- All the other errors including nullity violations and foreign key constraint violations are synchronization errors.
- If synchronization errors are not automatically resolved, the corresponding transactions are rolled back and the transaction operations are moved into mobile server error queue in C$EQ, while the data is stored in CEQ$. The mobile server database administrators can change these transaction operations and re-execute or purge transactions from the error queue.

Choose the type of conflict resolution you want for this publication item, as follows:

- Client wins—When the client wins, the mobile server automatically applies client changes to the server. And if you have a record that is set for INSERT, yet a record already exists, the mobile server automatically modifies it to be an UPDATE.
- Server wins—If the server wins, the client updates are not applied to the application tables. Instead, the mobile server automatically composes changes for the client. The client updates are placed into the error queue, just in case you still want these changes to be applied to the server—even though the winning rules state that the server wins.
- Custom—You have created your own callbacks to resolve the conflict resolution.

All synchronization errors are placed into the error queue. For each publication item created, a separate and corresponding error queue is created. The purpose of this queue is to store transactions that fail due to unresolved conflicts. The administrator can attempt to resolve the conflicts, either by modifying the error queue data or that of the server, and then attempt to re-apply the transaction.

See Section 2.10, "Resolving Conflicts with Winning Rules" for more information.

DML Callback

A user can use Java to specify a customized PL/SQL procedure which is stored in the mobile server repository to be called in place of all DML operations for this publication item. There can be only one mobile DML procedure for each publication item. See Section 2.4.1.13, "Callback Customization for DML Operations" for more information on how to specify a DML Callback.

Enter a string for the schema and package of the DML callback, such as `schema.package_name`.

Grouping Function

If you know that two tables should share a map, but Oracle Database Mobile Server would not normally associate these tables, provide a grouping function that denotes the shared publication item data between the tables.

> **Note:** The mobile server schema owner needs to be granted execute privilege on the defined grouping function.

The grouping function is a PL/SQL function with the following signature.

```
(
CLIENT in VARCHAR2,
PUBLICATION in VARCHAR2,
ITEM in VARCHAR2
) return VARCHAR2.
```

The returned value must uniquely identify the client's group.

In this field, provide the PL/SQL grouping function fully-qualified, either with `schema.package.function_name` or `schema.function_name`.

See the Section 1.2.7, "Shared Maps" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for more information.

Priority Condition

Provide a string that is to be added to the publication item query statement to limit what is returned based on priority.

See Section 1.2.10, "Priority-Based Replication" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for more information.

MyCompose Class

Provide a string with the full path and classname of the location and name of the `MyCompose` Class. See Section 2.6, "Customize the Compose Phase Using MyCompose" for more information on this class.

Weight

You can rate the order in which each publication item in this publication is executed by specifying the weight. This should be a number. Each publication item must have a unique number in ascending order. The first publication item executed is the one with the weight of one.

4.8.3 Sequence Tab Associates Existing Sequences With the Publication

You can only associate an existing sequence with the publication on this screen. To add an existing sequence, click **Add**. Create a sequence through the **File->New->Sequence** screen.

> **Note:** Sequences are supported only within Berkeley DB.

Click on the drop-down list and select one of the existing sequences to add to the publication. Click **OK** to add the sequence; click **Cancel** to go back to the previous screen.

4.8.4 Script Tab Associates Existing Scripts With the Publication

You can only associate an existing script with the publication on this screen. To add an existing script, click **Add**.

> **Note:** You can import a script through the **File->New->Script** screen.

Click on the drop-down list and select one of the existing scripts to add to the publication. Click **OK** to add the script; click **Cancel** to go back to the previous screen.

It is important that all scripts follow the instructions listed in Section 4.7.1, "Writing SQL Scripts".

4.8.5 Event Tab Configures Automatic Synchronization Rules for this Publication

When you select the Event Tab, you can configure data event rules for this publication, which apply to all automatic synchronization enabled publication items associated in this publication.

Data events define when an automatic synchronization is triggered.

- Client Data Events—Synchronize if the client database contains more than `<number>` modified records, where you specify the `<number>` of modifed records in the client database to trigger an automatic synchronization.
- Server Data Events—Synchronize if the out queue contains more than `<number>` modified records, where you specify the `<number>` of modifed records in the client database to trigger an automatic synchronization.

The lowest value that can be provided in these fields is 1. Specify a high value if you want the synchronization to occur based upon other rules. Click **Apply** when finished.

4.9 Import Existing Publications and Objects from Repository

You can import existing publications, publication items, sequences, or scripts that already exist within the repository by choosing the **Project->Add From Repository** option, as described in the following sections:

- Section 4.9.1, "Import Existing Publication from Repository"
- Section 4.9.2, "Import Existing Publication Item From the Repository"
- Section 4.9.3, "Import Existing Sequence From the Repository"
- Section 4.9.4, "Import an Existing Script From the Repository"

4.9.1 Import Existing Publication from Repository

You can add an existing publication that already exists in the repository to this project by selecting **Project->Add From Repository->Publication**. All associated objects—publication items, sequences, or scripts—are also pulled into the project with the publication.

To view all publications in the repository, click **Search**. All publications are shown in the left-hand screen. To limit the displayed publications to only those with a certain string as part of the name, provide this string in the Filter and then click **Search**. Only those publications that match the filter are shown.

Note: In the Search Filter, you can use the same pattern matching characters in a valid `SQL WHERE` clause. The filter is case-sensitive; use upper-case characters.

Select the desired publications and either double-click or select the right arrow to move them to the right window. Once all desired publications are in the right window, click **OK** to move these publications into the project.

4.9.2 Import Existing Publication Item From the Repository

You can add an existing publication item that already exists in the repository to this project by selecting **Project->Add From Repository->Publication Item**.

To view all publication items in the repository, click **Search**. All publication items are shown in the left-hand screen. To limit the displayed publication items to only those with a certain string as part of the name, provide this string in the Filter and then click **Search**. Only those publication items that match the filter are shown.

> **Note:** To search only for objects that match a condition, designate the condition in the Filter box and click **Search**. You can use the same pattern matching characters in a valid SQL WHERE clause. The filter is case-sensitive; use upper-case characters.

Select the desired publication items and either double-click or select the right arrow to move them to the right window. Once all desired publication items are in the right window, click **OK** to move these publication items into the project.

Once added into the project, you still must associate them with the publication if you want to test the synchronization of the publication item. See Section 4.8.2, "Publication Item Tab Associates Publication Items With the Publication" for more information.

4.9.3 Import Existing Sequence From the Repository

You can add an existing sequence that already exists in the repository to this project by selecting **Project->Add From Repository->Sequence**.

> **Note:** Sequences are supported only within Berkeley DB.

To view all sequences in the repository, click **Search**. All sequences are shown in the left-hand screen. To limit the displayed sequences to only those with a certain string as part of the name, provide this string in the Filter and then click **Search**. Only those sequences that match the filter are shown.

> **Note:** To search only for objects that match a condition, designate the condition in the Filter box and click **Search**. You can use the same pattern matching characters in a valid `SQL WHERE` clause. The filter is case-sensitive; use upper-case characters.

Select the desired sequences and either double-click or select the right arrow to move them to the right window. Once all desired sequences are in the right window, click **OK** to move these sequences into the project.

Once added into the project, you still must associate them with a publication if you want to test it with a synchronization. See Section 4.8.3, "Sequence Tab Associates Existing Sequences With the Publication" for more information.

4.9.4 Import an Existing Script From the Repository

You can add an existing script that already exists in the repository to this project by selecting **Project->Add From Repository->Script**.

To view all scripts in the repository, click **Search**. All scripts are shown in the left-hand screen. To limit the displayed scripts to only those with a certain string as part of the name, provide this string in the Filter and then click **Search**. Only those scripts that match the filter are shown.

> **Note:** To search only for objects that match a condition, designate the condition in the Filter box and click **Search**. You can use the same pattern matching characters in a valid SQL WHERE clause. The filter is case-sensitive; use upper-case characters.

Select the desired scripts and either double-click or select the right arrow to move them to the right window. Once all desired scripts are in the right window, click **OK** to move these scripts into the project.

> **Note:** All scripts added to the project must follow the guidelines as described in Section 4.7.1, "Writing SQL Scripts".

Once added into the project, you still must associate them with a publication if you want to test it with a synchronization. See Section 4.8.4, "Script Tab Associates Existing Scripts With the Publication" for more information.

4.10 Create a Virtual Primary Key

For fast refresh, you must have a primary key. If the table, view, or synonym does not currently have a primary key, you can designate one of the columns as the virtual primary key through this screen, as follows:

> **Note:** Any virtual primary key must be unique and not NULL.

1. Using the drop-down lists, choose the following:
   - Schema name
   - Object type: table, view or synonym type
   - Any string that exists within the object name, if desired
2. Click **Search**, which brings up a list of available objects.
3. From the object list, choose the appropriate table, view, or synonym. Once chosen, the available columns are listed.
4. Select the columns that you wish to be the primary key and click **OK**.

If you have a composite primary key, iteratively add each column within the composite primary key.

4.11 Test a Publication by Performing a Synchronization

You can create a test to perform a synchronization of the designated publication. Click **Project->Test Publication**. When you create the test, MDW automatically creates the subscription for the user.

1. Click **Create** to design the test and provide the following information:

- Name: If the test is remote, then the user name is populated with the registered owner of the remote target device. If the test is local, then the user name should be a valid mobile user in the repository.
- Publication: From the drop-down list, select one of the available publications in this project for this test.
- Client type: Designate if the client is local or remote. Default is local. If Active Sync is not installed, the remote option is not available.
- Specify a user that is defined in Mobile Manager.

Click **OK** to save the test; click **Cancel** to revert back to the previous screen.

Note: To remove any tests, select the test and click **Remove**.

---

2. Once created, click **Synchronize** to perform a synchronization for the designated publication. On the pop-up dialog, provide the password for the given user name and the URL of the mobile server. The URL for the mobile server should be the `hostname/mobile`.

   Click **Option** to specify priority of the publication items, as follows:

   - High Priority: Limits synchronization to server tables flagged as high priority, otherwise all tables are synchronized.

     Note: You can only use fast refresh with a high priority restricting predicate. If you use any other type of refresh, the high priority restricting predicate is ignored.

     See Section 1.2.10, "Priority-Based Replication" in the *Oracle Database Mobile Server Troubleshooting and Tuning Guide* for more information.

   - Push Only: Upload changes from the client to the server only, do not download. This is useful when data transfer is one way, client to server.
   - Complete Refresh: All data is refreshed from the server to the client.
   - Debug: Turn on debugging when synchronizing.
   - Selective Synchronization: Determine which publication and publication items are allowed to synchronize. When you click this option, move the publication items that you want to synchronize from the left window to the right window using the arrow buttons. For details on how selective synchronization performs, see Section 3.1.3.1.8, "Manage What Tables Are Synchronized With ocSetTableSyncFlag" and Section 3.1.1.1.4, "Selective Synchronization".

   Click **OK** to save the synchronization options or **Cancel** to return to the previous screen.

4.12 Deploy the Publications in the Project to the Repository

You can deploy one or more of the publications in the current project from the development/test mobile server repository to a target production mobile server repository by clicking **File->Deploy**. You should adequately test all publications before deploying to the production mobile server repository.

All available publications are displayed in the project publications section. To limit the displayed publications to only those with a certain string as part of the name, provide this string in the Filter and then click **Search**. Only those publications that match the filter are shown.

> **Note:** In the Search Filter, you can use the same pattern matching characters in a valid `SQL WHERE` clause. The filter is case-sensitive; use upper-case characters.

Select the desired publications and click **OK** to deploy these publications into the repository. A dialog appears where you specify the remote database connection information, as follows:

- User name and password for database connection authentication.
- JDBC Driver type: Based on the type of the JDBC driver, different information is required. At this time, you can only use the JDBC Thin driver. Provide the host name, port, and SID for the remote database.

Click **OK** to accept the input values for the remote database; click **Cancel** to return to the previous screen.

5

Using the Packaging Wizard

The following sections enable you to package and publish your mobile application definitions using the Packaging Wizard.

5.1 Using the Packaging Wizard

After you have completed the code implementation for your application, you need to define the SQL commands that retrieve the data for the user snapshot—also known as a publication. MDW (as described in Chapter 4, "Using Mobile Database Workbench to Create Publications") is a graphical tool that enables you to define the publications for your application. Then, use the Packaging Wizard to package the application and publish the final application product to the mobile server to complete the subscription.

In general, you can create a publication—or components of a publication—using one of the following methods:

- SQL on the back-end Oracle database
- Consolidator APIs
- MDW
- Packaging Wizard

> **Note:** If you create your publication using the Packaging Wizard, you cannot use remote databases for your application.

If you create the publication using any method other than the Packaging Wizard, you can import the definition into the Packaging Wizard. However, these tools and the Packaging Wizard are separate. Thus, once the publication is published by the Packaging Wizard, you can only modify it through the Packaging Wizard.

Important: If you modify the publication or any component of the publication using any method other than the Packaging Wizard, then it not show up in your published application.

The following is the recommended method for creating the publication for the application:

- Create a new mobile application definition—An application definition is more than the code that you have implemented. It consists of the implementation, the publication with its publication items, and other components. Use the Mobile

Database Workbench (MDW) tool (as described in Chapter 4, "Using Mobile Database Workbench to Create Publications" for performing an iterative approach to defining your publications.

- Edit an existing mobile application definition within the Packaging Wizard—You can always go back and edit an existing mobile application definition for tuning purposes, to modify the publication, or other reasons.
- Package a mobile application definition for easy deployment within the Packaging Wizard—Once the application is finished with development, you need to package the components into a JAR file before you can publish the application definition.
- Publish an application definition to the mobile server—You can either publish your application definition to the mobile server with the Packaging Wizard or through the Mobile Manager.

The following sections describe how to use the Packaging Wizard tool:

5.1.1 Starting the Packaging Wizard

To launch the Packaging Wizard, enter the following using a Command Prompt window.

```
runwtgpack
```

Note: If you enable the mobile server to be SSL-Enabled, then you have to change the configuration on the host where the Packaging Wizard is located in order for it to successfully communicate with the mobile server.

In order for Packaging Wizard to be SSL-Enabled, set the `SSL` parameter to `TRUE` in the `mobile.ora` file located on the host where the MDK is installed.

Figure 5–1 shows the Welcome screen for the Packaging Wizard, which enables you to create, edit, or remove the mobile application definition as described fully in Table 5–1.

Figure 5–1 Packaging Wizard - Make A Selection Dialog

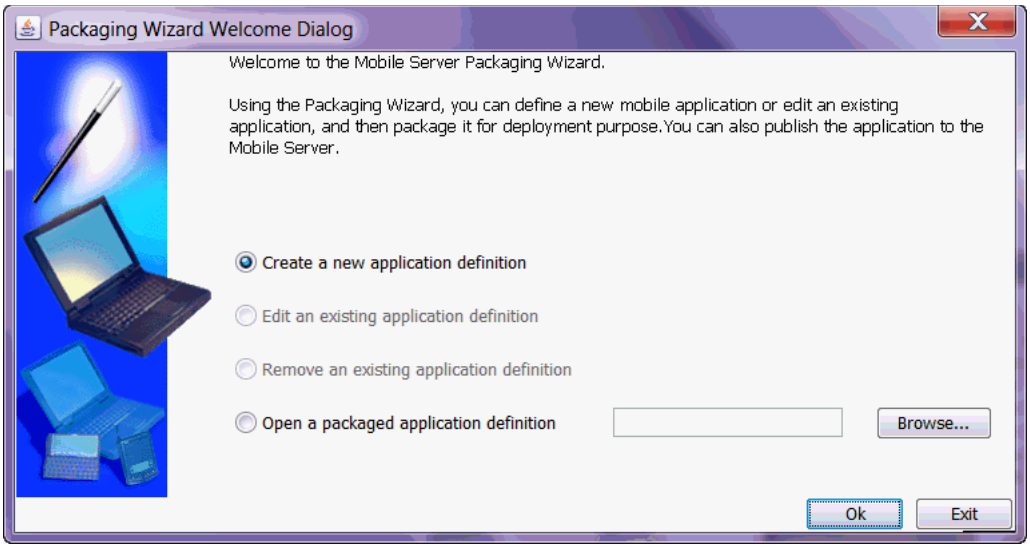


Table 5–1 Make a Selection Dialog

Feature	Description
Create a new application definition	Define a new mobile application definition with the application implementation, publication items, and so on.
Edit an existing application definition	Edit an existing mobile application definition. When selected, all existing application definitions are presented in a drop-down box. Users can select the desired mobile application definition from the list. All applications listed in this list have been created or published using the Packaging Wizard. Any application definition created by MDW not appear in this list.
Remove an existing application definition	Remove an existing mobile application definition. When selected, all existing application definitions are presented in a drop-down box. Users can select the desired mobile application definition from the list. This option removes the application definition from the Packaging Wizard; it does not delete the application from within the mobile server.
Open a Packaged application definition	Select an application definition that has been packaged a JAR file. You can enter the name of the packaged application or locate it using the 'Browse' button.

Click OK. This brings up the 'Select the client application attributes' dialog, with which you want to package your application definition. As Figure 5–2 displays, this dialog has a radio button group from which you can specify the database type, a pull-down menu from which you can specify the platform, and a pull-down menu from which you can specify the locale. The available platforms in the Platform pull-down menu only provide those options which are applicable to the selected database type.

For Berkeley DB, the available platform options are as follows:

- Java All Platforms stands for all Java platforms where Berkeley DB Java client is available
- Windows 32/64 stands for Berkeley DB WIN32 in the Mobile Manager
- Linux 32/64 stands for Berkeley DB Linux x86 in the Mobile Manager

- Windows CE/Mobile ARMv4i stands for Berkeley DB PPC50 and Berkeley DB PPC60 ARMV4I in the Mobile Manager
- Android stands for Berkeley DB Android in the Mobile Manager

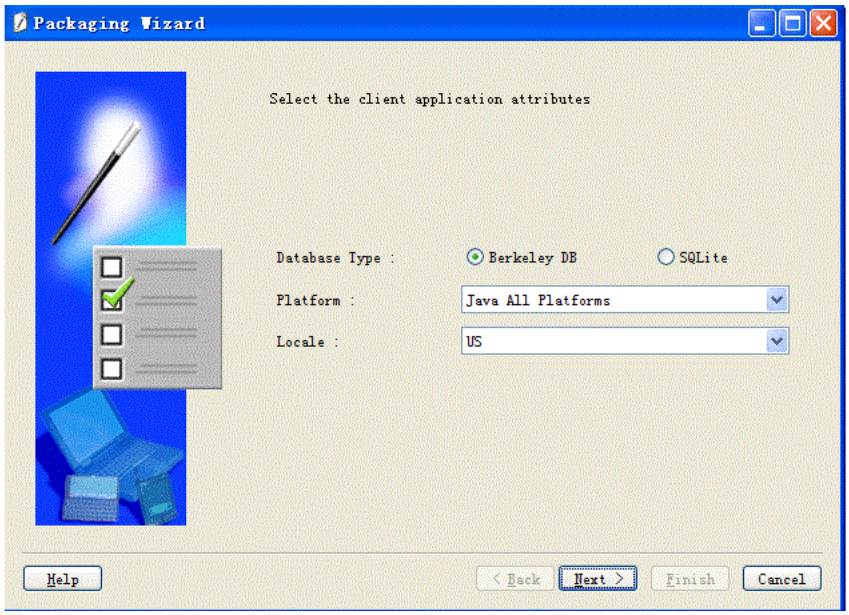
For SQLite, the available platform options are as follow:

- Java All Platforms stands for all Java platforms where SQLite Java client is available
- Windows 32/64 stands for SQLite WIN32 in the Mobile Manager
- Linux 32/64 stands for SQLite Linux x86 in the Mobile Manager
- Windows CE/Mobile ARMv4i stands for SQLite PPC60 ARMV4I in the Mobile Manager
- Android stands for SQLite Android in the Mobile Manager
- BlackBerry stands for SQLite BlackBerry in the Mobile Manager

Once selected, click Next.

Note: For iOS client, select Linux 32/64 option.

Figure 5–2 Select a Platform Dialog



5.1.2 Specifying New Application Definition Details

The Application dialog enables you to name a new application and specify its storage location on the mobile server.

Figure 5–3 Application Dialog

Table 5–2 describes the Application dialog.

Table 5–2 Application Dialog Description

Field Name	Description	Required
Application Name	The name of the new mobile application definition.	Yes
Virtual Path	A path that is mapped from the root directory of the server repository to the mobile application itself. The virtual path eliminates the need to refer to the application entire directory structure. It indicates that all of the subdirectories and all of the files that are in the virtual path be uploaded exactly as they are in the directory structure to the mobile server repository when the application is published. It also provides the application with a unique identity. The name that you enter as the virtual path of the application becomes the **application root directory** within the mobile server repository, when the application is published. Consequently, you can specify the application root directory by the name that you enter in the virtual path field. This name can be different from the application name, but should not contain spaces. For example, your application name can be 'Sales Office' and your virtual path '/Admin'. In this case, '/Admin' becomes the name of the application root directory within the mobile server repository. The application root directory is the location where the actual application files are stored within the mobile server repository. When the administrator publishes the application, the Packaging Wizard automatically uses the name that you entered in the virtual path as the name of the application root directory in the mobile server repository. However, the administrator can change the name of the application root directory in the mobile server repository by entering a different name for it when the administrator publishes the application.	Yes

Table 5–2 (Cont.) Application Dialog Description

Field Name	Description	Required
Description	A brief description of the mobile application.	Yes
Local Application Directory	The directory on the local machine that contains all components of the application. You can type this location or locate it using the 'Browse' button. During development, the application root directory is set to the local application directory.	Yes

Table 5–2 (Cont.) Application Dialog Description

Field Name	Description	Required
Publication Name	Publication name of an existing application in the Mobile Server repository. You can enter the publication name or locate it using the "Browse" button. If you choose to enter the publication name, ensure that the publication already exists in the Mobile Server repository where you will publish the application. If you choose to locate the publication using the "Browse" button, ensure that it satisfies either of the following conditions: ■ the publication exists in the Mobile server repository where you will publish the application ■ the publication exists in another Mobile server repository and it was originally created with Mobile Database Workbench the publication exists in the Mobile server repository where you will publish the application Note: these two Mobile servers should have the same base tables for your publication For example, you have two Mobile servers with matching repository user and application schema names: ■ one in the production environment, with MOBILEADMIN as its repository user and MASTER as its application schema user in database instance A ■ another in the test environment, with MOBILEADMIN as its repository user and MASTER as its application schema user in another database instance B If you have a publication based on base table MASTER.T1 in the test environment and it was created with Mobile Database Workbench, you do the following to publish an application associated with this publication to the production environment: ■ Locate the publication using the "Browse" button ■ Enter a JDBC connection to MOBILEADMIN in database instance A. This is to get the mada data of the publication in MOBILEADMIN and create mada.xml. Mada.xml is then used to create the publication in the production environment. This is done when you publish the application associated with the publication to the production environment. When you locate a publication using the "Browse" button, you need to specify schema name, password and JDBC url. The schema name, password and JDBC url is the mobile server repository user name, password (with administrator privilege) and JDBC url where the publication already exists. The default mobile server repository user name is MOBILEADMIN. For the example above, the schema name is MOBILEADMIN and the JDBC url is the connection to MOBILEADMIN in database instance A.	No

Click Next to advance to the next screen.

5.1.3 Listing Application Files

Use the Files screen to list your application files and to specify their location on the local machine. The Packaging Wizard analyzes the contents of the Local Application Directory and displays each file's local path.

Figure 5–4 displays the Files screen.

Figure 5–4 Files Tab

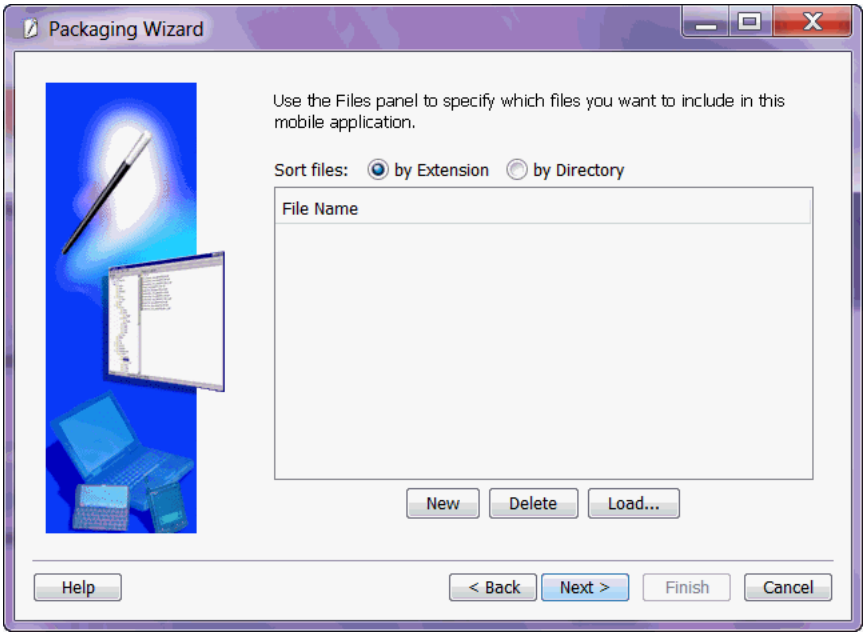


Table 5–3 Files Tab Description

Field	Description	Required
Local Path	The absolute path of each mobile application file. Each entry on the list includes the complete path of the individual file or directory.	Yes

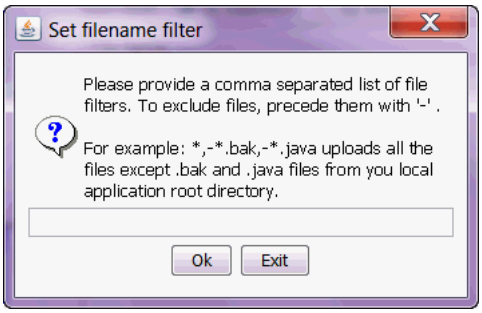
You can add, remove, or load any of the files that are listed in the 'Files' dialog. If you are creating a new application, the Packaging Wizard automatically analyzes and loads all files listed under the local directory when you proceed to the 'Files' dialog. If you are editing an existing application, upload your individual application files using the 'Load' button.

When you click the 'Load' button, the 'Input' dialog appears. You can use the 'Input' dialog to create a comma-separated list of filters that either include or exclude application files from the upload process. To exclude a file, type a preceding minus sign (-) before the file name. For example, to load all files but exclude files with the `.bak` and `.java` suffixes, enter the following.

```
*,-*.bak,-*.java
```

Figure 5–5 displays the Input dialog.

Figure 5–5 Input Dialog



Once you have loaded all relevant files, click **Finish** to complete the packaging and start the publishing of your application.

5.1.4 Publish the Application

Once the packaging is complete, the Packaging Wizard brings up the following screen, so you can choose whether to start the publishing or to complete the packaging to be published at another time.

Figure 5–6 Publish the Application

On this screen, you can choose one of the following:

- Create files: You can package all the information you have gathered for the application and its files into a JAR file. This JAR file can be used at a future date to be published to any mobile server repository. If chosen, you are asked for a location on where to store the JAR file.

  The Packaging Wizard has an option to generate SQL scripts for database objects. This option has been deprecated as of release 11.1.

- Publish the current application: This publishes all of the information you have gathered for the application and its files directly to the mobile server repository.

  If chosen, you be prompted for the URL, user name and password for the mobile server repository. You also be asked for the repository directory and if you want to make this application public.

- Restart wizard: This does not save any of your work for this application and starts the Packaging Wizard without any saved data.

5.1.5 Editing Application Definition

You can edit application definitions by launching the Packaging Wizard and selecting "Edit an existing application definition."

5.1.6 Troubleshooting

The Packaging Wizard also supports development mode. In this mode, the Packaging Wizard only enables you to define application information, list the application files, and make registry changes. Since the application is packaged to your local machine, it requires neither connectivity nor database information.

To launch the Packaging Wizard in development mode, enter the following using the Command Prompt.

```
runwtgpack -d
```

5.2 Packaging Wizard Synchronization Support

The Packaging Wizard and the Mobile Manager provide the ability to perform the most commonly used functions of the publish and subscribe model, package and publish applications, create or drop users, and create or drop subscriptions. More sophisticated functionality is provided by the Consolidator Manager and Resource Manager APIs. Table 5–4 describes basic features.

Table 5–4 Packaging Wizard Synchronization Support

Function	Packaging Wizard	Mobile Manager	API
Open Connection	No	No	Yes
Create User	No	Yes	Yes
Drop User	No	Yes	Yes
Create Publication	Yes	No	Yes
Create Publication Item	Yes	No	Yes
Create Publication Item Index	Yes	No	Yes
Drop Publication	No	Yes	Yes
Drop Publication Item	Special - See the Packaging Wizard documentation for more details.	No	Yes
Drop Publication Item Index	Yes	No	Yes
Create Sequence	Yes	No	Yes
Create Sequence Partition	Yes	No	Yes
Drop Sequence	Yes	No	Yes
Drop Sequence Partition	Yes	No	Yes
Add Publication Item	Yes	No	Yes
Remove Publication Item	No	No	Yes
Create Subscription	No	Yes	Yes
Deinstantiate Subscription	No	No	Yes
Set Subscription Parameter	No	Yes	Yes

Table 5–4 (Cont.) Packaging Wizard Synchronization Support

Function	Packaging Wizard	Mobile Manager	API
Drop Subscription	No	Yes	Yes
Commit Transaction	No	No	Yes
Rollback Transaction	No	No	Yes
Close Connection	No	No	Yes

More advanced features of Data Synchronization are only generally available by using the Consolidator Manager and Resource Manager APIs. Table 5–5 describes these features.

Table 5–5 Data Synchronization Advanced Function Description

Function	Packaging Wizard	Mobile Manager	API
Create Virtual Primary Key Column	Yes	No	Yes
Drop Virtual Primary Key Column	Yes	No	Yes
Add Mobile DML Procedure	Yes	No	Yes
Remove Mobile DML Procedure	Yes	No	Yes
Reinstantiate Publication Item	No	No	Yes
Parent Hint	Yes	No	Yes
Dependency Hint	Yes	No	Yes
Remove Dependency Hint	Yes	No	Yes
Enable Publication Item Query Cache	No	No	Yes
Disable Publication Item Query Cache	No	No	Yes
Primary Key Hint	Yes	No	Yes
Purge Transaction	No	No	Yes
Execute Transaction	No	No	Yes
Complete Refresh	Yes	Yes	Yes
Execute Statement	No	No	Yes
Generate Metadata	No	No	Yes
Reset Cache	No	No	Yes
Cache Dependencies	No	No	Yes
Remove Cache Dependencies	No	No	Yes
Get Current Time	No	No	Yes
Authenticate	No	Yes	Yes
Set Restricting Predicate	No	No	Yes
Alter Publication	Yes	No	Yes

6

Create and Manage Jobs with APIs

The following sections describe how you can manage and create jobs with ConsolidatorManager APIs:

6.1 Managing Scheduled Jobs Using ConsolidatorManager APIs

Application developers can define, submit, and manage jobs programmatically based on a pre-determined time and interval. For example, jobs can be scheduled to run repeatedly for a specified duration on any specified day or days of the week or month. Administrators can schedule jobs to run repeatedly for a specified number of months, weeks or specified days of the month or week.

The Job Scheduler API schedules and executes jobs using a job engine. It is a generic component which enables apply and compose functions for MGP, device manager jobs, and custom jobs.

- Using the class `oracle.lite.sync.ConsolidatorManager`, application developers can register or de-register a job class, create, drop, enable or disable a job, search, and delete a job execution log.
- Use other supporting classes, such as `Job`, `Schedule`, `ExecutionResult` and `ExecutionLog` in the `oracle.lite.sync.job` package to manage your scheduled jobs.

For more information on these classes and their methods, refer to the *Oracle Database Mobile Server JavaDoc*.

6.2 Start a Standalone Job Engine In Separate JVM

If you want to execute a Standalone Job engine in a separate JVM from any of the mobile servers in the farm, then perform the following:

1. Retrieve a connection to the database with the Consolidator Manager `openConnection` method. Pass in the Mobile Manager administrator user name, password and optionally, the JDBC URL to the back-end Oracle database.
2. Create a new Job engine with the `JobEngine` class and start it with the `startUp` method. The Standalone Job engine executes in a separate thread, which you can terminate from the main thread.
3. Define how long the thread is to sleep between execution of all jobs.

4. Terminate the Standalone Job engine when you have completed all activities.

> **Note:** The following example demonstrates how to start up a Standalone Job engine in its own thread. It executes all of the jobs that have been scheduled either through the API or through the Mobile Manager Job Scheduler screens, because the Job Scheduler retrieves the scheduled job information from the repository.

```
JobEngine JobEngine = new JobEngine();
JobEngine.startUp();
if (JobEngine.runnerThreadException != NULL){
  System.out.println("runnerThreadException:");
  JobEngine.runnerThreadException.printStackTrace();
}

Thread.currentThread().sleep(60*1000);

if (JobEngine.runnerThreadException != NULL){
  System.out.println("runnerThreadException:");
  JobEngine.runnerThreadException.printStackTrace();
}
JobEngine.shutDown();
```

6.3 Using the ConsolidatorManager APIs to Create Jobs

Within the `oracle.lite.sync.ConsolidatorManager` class, there are several APIs, which are documented fully in the *Oracle Database Mobile Server JavaDoc*, that enable you to create, register, and schedule your job.

While these methods are described fully in the *Oracle Database Mobile Server JavaDoc*, the following demonstrates the order in which you would execute the methods:

1. Create your job class by implementing the `oracle.lite.job.Job` interface. Implement the Job interface methods, as follows:
   - `init` method—This method is invoked by the Job Scheduler when the job is loaded.
   - `execute` method—This method is invoked by the Job Scheduler when the job is scheduled to execute. Put a call into your application within this method. The Job Scheduler passes in the input parameter that was provided when the job is created—either with the `createJob` method or within the Mobile Manager Job Scheduler screen. When finished, the `execute` method returns an object of class type `ExecutionResult` containing whether the job was a success or failure.
   - `destroy` method—This method is invoked after the job completes.
2. After you have created your job class, register it with the `registerJobClass` method.
3. Create the job in the Job Scheduler by executing the `createJob` method. One of the input parameters is an object of class type `Schedule`, which defines when the job is executed. There are also other management methods that correspond to the Mobile Manager GUI, such as `dropJob`, `enableJob`, and `disableJob`.
4. If you want to retrieve any logs, execute the `getJobExecutionLogs` method, which retrieves objects of `ExecutionLog` class.

7

Customizing Oracle Database Mobile Server Security

Managing the provided security within Oracle Database Mobile Server is described in Chapter 9, "Configuring Security in the Oracle Database Mobile Server" in the *Oracle Database Mobile Server Administration and Deployment Guide*. This chapter describes how to customize authentication to provide your own mechanisms to be used within Oracle Database Mobile Server.

The following section details security issues for Oracle Database Mobile Server:

- Section 7.1, "Providing Your Own Authentication Mechanism for Authenticating Users for the Mobile Server"

7.1 Providing Your Own Authentication Mechanism for Authenticating Users for the Mobile Server

You can provide an external authenticator for the mobile server to authenticate users with passwords as well as their access privileges to applications. For example, in an enterprise environment, you may have your user data, such as employee information, stored in a LDAP-based directory service. The mobile server can retrieve the user information from the LDAP directory—or from any custom User Management System—if configured with your own implementation of an external authenticator. The mobile server links the external user information to the mobile server repository.

The following sections describe how to implement and use an external authentication method for Oracle Database Mobile Server:

- Section 7.1.1, "Implementing Your External Authenticator"
- Section 7.1.2, "Registering External Authenticator"
- Section 7.1.3, "User Initialization Scripts"

7.1.1 Implementing Your External Authenticator

In order to use an external authenticator, you must implement the `oracle.lite.provider.Authenticator` Java interface and configure the implementation in the `mobile.ora` file.

Implement the following methods in your external authenticator. The mobile server invokes each of these methods as appropriately.

- Section 7.1.1.1, "Initialization for the External Authenticator"
- Section 7.1.1.2, "Destruction of the External Authenticator"

7.1.1.1 Initialization for the External Authenticator

The mobile server invokes the `initialize` method before calling any other method of provider class. This method be called only once when the provider is initialized.

```
Method: void initialize (String metaData) throws Exception
Parameter: String metaData (Reserved for future use)
```

7.1.1.2 Destruction of the External Authenticator

The mobile server invokes the `destroy` method when the system shutdowns. Provider implementation should implement all the cleanup code in this method.

```
Method: void destroy() throws Exception
Parameter: None
```

7.1.1.3 The Authentication Method for the External Authenticator

Authenticate a user and return a session handle with the `authenticate` method. The returned session handle is passed to the `logOff` method when the user logs off from the system. Note that the `logOff` method may not be called for each successful `authenticate` method call. Some of the mobile server clients may use the `authenticate` method to verify the user credential and not for logging on to the system.

```
Method: Object authenticate (String uid, String pwd) throws SecurityException
Parameter: User Id (or User Name) and password string
Return: Session handle or NULL
```

You can pass error and warning information, as follows:

- Failure: Pass along any error information, such as why the authentication failed. Use the `AuthException` class, available in the package `oracle.lite.provider.auth`, to pass along failure information.
- Warning: Pass along any warnings, such as the situation when the user's password is about to expire. Use the `ExtAuthResult` class, available in the package `oracle.lite.provider.auth`, to pass along warning information.

Refer to the *Oracle Database Mobile Server API Specification* for more details on these exception classes.

7.1.1.4 The User Instantiation Method for the External Authenticator

If the user has not been instantiated in the mobile server repository, then the mobile server invokes the `getInitializationScripts` method—after authenticating the user—to retrieve the initialization scripts for the user. The mobile server uses the initialization scripts to instantiate the user in the mobile server and assign access rights to applications and data. See Section 7.1.3, "User Initialization Scripts" for more information.

```
Method: StringBuffer getInitializationScripts (Object sid)
Parameter: Session handle returned by 'authenticate' method
```

```
Return: 'StringBuffer' containing User's initialization scripts
```

7.1.1.5 Retrieve the User Name or the User Global Unique ID

Return the user name or GUID (Globally Unique Id) of the user if there is one. Usually, LDAP-based User Management systems maintain a GUID for each user. In case your authentication mechanism does not support GUID, then the `getUserGUID` method returns `NULL`.

```
Method: String getFullName (Object sid)
Parameter: Session handle returned by 'authenticate' method
Return: User's full name

Method: String getUserGUID (Object sid)
Parameter: Session handle returned by 'authenticate' method
Return: User's GUID or NULL
```

7.1.1.6 Log Off User

Log off the User from the back-end system. Note that the `logOff` method may not be called for each successful `authenticate` method call. Some of the mobile server clients may use the `authenticate` method to verify the user credential and not for logging on to the system.

```
Method: void logOff (Object sid) throws SecurityException
Parameter: Session handle returned by 'authenticate' method
```

7.1.1.7 Change User Password

```
Method: void changePassword (Object sid, String pwd) throws SecurityException
Parameter: Session handle returned by the authenticate method and new password
string
```

7.1.2 Registering External Authenticator

The `EXTERNAL_AUTHENTICATION` section in the `mobile.ora` file facilitates the authentication of existing external users with the specified external authenticator class. To register your external authenticator class, modify the `mobile.ora` file and set your external `Authenticator` JAVA class name in the `EXTERNAL_AUTHENTICATION` section, as follows:

```
[EXTERNAL_AUTHENTICATION]
CLASS  = SampleAuthenticator
EXPIRATION = 1800
```

The mobile server caches the user instantiated through the external authenticator for a period of time in order to improve efficiency. The default expiration time for the cached user object is 30 minutes (or 1800 seconds). Customize this value by setting a new value for the `EXPIRATION` parameter.

In addition, you must configure the `EXTERNALUSER` parameter in the `WSH.INI` script, which notifies the server that the user being created is external and does not require a password in the `WSH.INI` script. Instead, the new user be authenticated by the external authenticator specified in the `mobile.ora` file. For more information on `EXTERNALUSER` parameter, see Appendix B, "Write Scripts for the Mobile Server with the WSH Tool" in the *Oracle Database Mobile Server Administration and Deployment Guide*.

Alternatively, you can create the external user with the `ResourceManager` APIs, which notifies the server that the user being created does not require a password.

Instead, the new user be authenticated by the external authenticator specified in the `mobile.ora` file.

7.1.3 User Initialization Scripts

The mobile server invokes the `getInitializationScripts` method to retrieve the user initialization script that instantiates user-specific objects in the mobile server repository. The external authenticator can perform the following actions during the initialization process:

1. Assign access rights to applications
2. Set data subscription parameters.
3. Optionally, add the user to a user group.

The syntax of the initialization script is based on the `INI` format. The first section in the script is as follows.

```
[MAIN]
VERSION=2
```

The following example performs these actions for a user whose id is `USER1`.

1. Assigning access rights to applications.

   Assign access rights to `Application1` and `Application2` for `USER1`, where `Application1` has two publication items and three subscription parameters.

   ```
   # List the applications we want access to
   #
   [ACL]
   Application1
   Application2
   # List Access details for 'Application1'
   #
   [ACL.Application1]
   NAME=USER1
   TYPE=USER
   DATA=LOCATION, ITEMS
   # List Access details for 'Application2'
   #
   [ACL.Application2]
   NAME=USER1
   TYPE=USER
   ```

2. Setting data subscription parameters.

   ```
   [SUBSCRIPTION.USER1.Application1.LOCATION]
   NAME=ZIP, USR_ID
   VALUE=12345, USER1
   [SUBSCRIPTION.USER1.Application1.ITEMS]
   NAME=WEIGHT
   VALUE=20
   ```

3. Adding a User to a User Group

   ```
   [GROUP]
   User's Group
   [GROUP.User's Group]
   USER=USER1
   ```

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W