Oracle Utilities Mobile Workforce Management

Hybrid Mobile Application Implementation and Development Guide

Release 2.3.0.3.0

F11038-01

November 2018

ORACLE®

Oracle Utilities Mobile Workforce Management Hybrid Mobile Application Implementation and Development Guide, Release 2.3.0.3.0

F11038-01

Preface

Audience

The target audience of this guide is implementers and system administrators responsible for implementation and deployment of mobile applications.

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

http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

Installation, Configuration, and Release Notes

- *Oracle Utilities Mobile Workforce Management Release Notes*
- *Oracle Utilities Mobile Workforce Management Quick Install Guide*
- *Oracle Utilities Mobile Workforce Management Server Application Installation Guide*
- *Oracle Utilities Mobile Workforce Management DBA Guide*
- *Oracle Utilities Mobile Workforce Management Hybrid Mobile Application Installation and Deployment Guide*
- *Oracle Utilities Mobile Workforce Management GEOCODE Data Source Configuration Guide*
- *Oracle Utilities Mobile Workforce Management JMS Setup Guide*

User Guides

- *Oracle Utilities Mobile Workforce Management Administrative User Guide*
- *Oracle Utilities Mobile Workforce Management Business User Guide*
- *Oracle Utilities Mobile Workforce Management Mobile Application User's Guide (Java-based)*
- *Oracle Utilities Mobile Workforce Management Hybrid Mobile Application User's Guide*
- *Oracle Utilities Mobile Workforce Management Hybrid Mobile Contractor Application User's Guide*

Implementation and Development

- *Oracle Utilities Mobile Workforce Management Hybrid Mobile Application Implementation and Development Guide*

Map Editor Installation and User Guides

- *Oracle Utilities Mobile Workforce Management Map Editor User's Guide*

- *Oracle Utilities Mobile Workforce Management Map Editor Installation Guide*

Supplemental Documents

- *Oracle Utilities Mobile Workforce Management Server Administration Guide*

- *Oracle Utilities Mobile Workforce Management Security Guide*

Chapter 1

Overview

This guide provides development and configuration details for the Oracle Utilities Mobile Workforce Management Mobile Application including Oracle Utilities Mobile Library, APIs, development environment setup, customization, and extension methodology.

This section provides a general overview and information about the mobile application components and architecture.
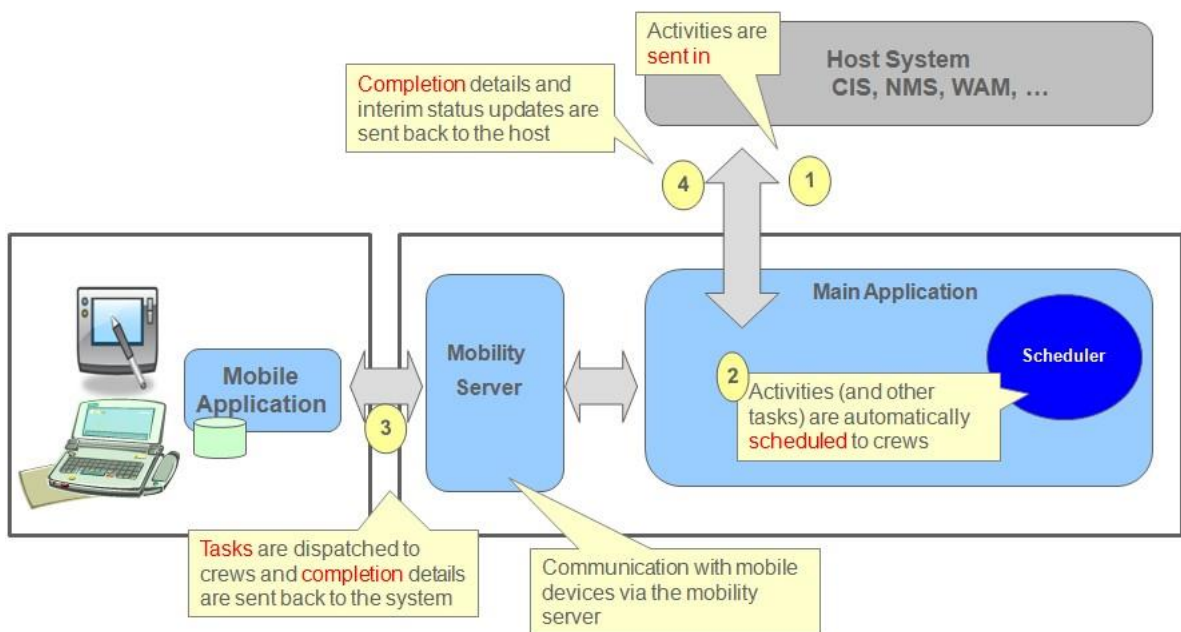
Architecture

Oracle Utilities Mobile Workforce Management simplifies and optimizes the scheduling, dispatching, and tracking of mobile service crews and field activities.
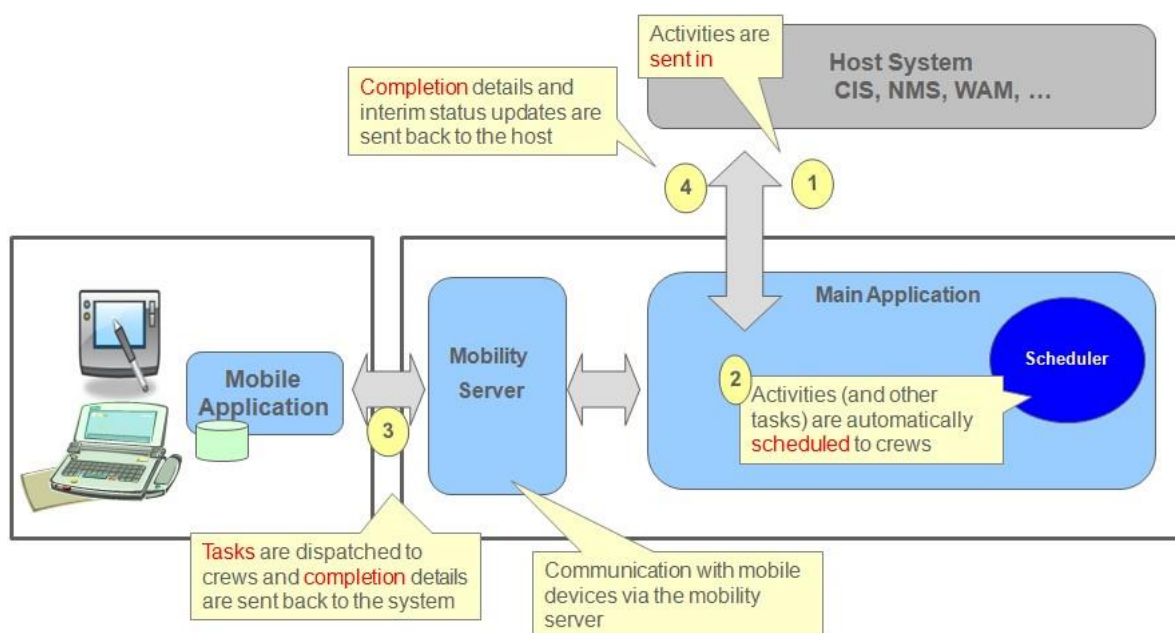
The mobile application consists of the Oracle Utilities Mobile library and application layers responsible for specific business functionality. It uses HTML5 and JavaScript to implement business logic, render the user interface and interact with mobile device services. Web services facilitate communication between the mobile application and the application server.

Oracle Utilities Mobile Library (OUML)

The Oracle Utilities Mobile Workforce Management Mobile Application is based on the Oracle Utilities Mobile Library (OUML) optimized to work with Oracle Utilities Application Framework (OUAF) based services, configurations and metadata. The Oracle Utilities Mobile Library provides a foundation layer and APIs for application development including offline storage, encryption, communication, logging, configuration, UI rendering/navigation, customization, deployment and so on. The Oracle Utilities Mobile Library makes use of third party libraries that are either bundled with the application or listed as pre-requisites.

Please reference Chapter 3: Oracle Utilities Mobile Library for more information on working with the Oracle Utilities Mobile Library.
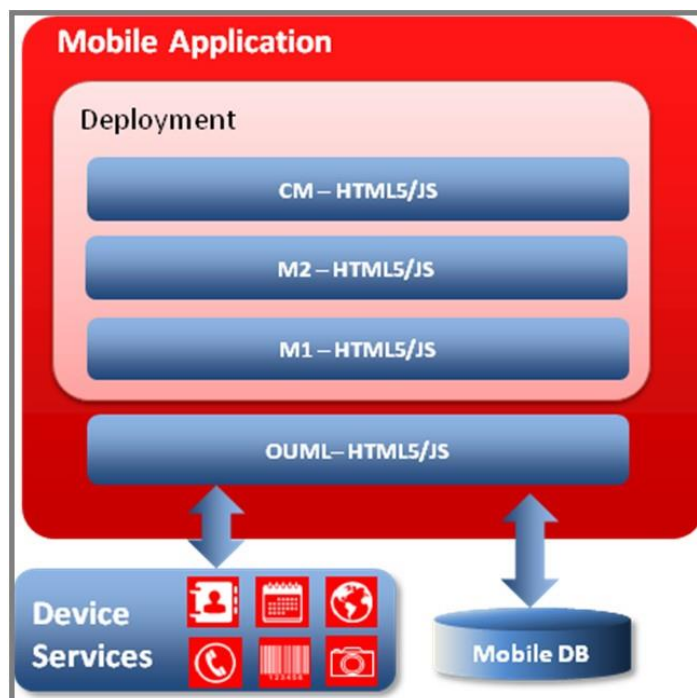
Deployment Models

The mobile application can be packaged and deployed in the format native to one of the supported runtime platforms. Alternately, it can be packaged as a web application and deployed to an application server to be accessed on the mobile device via a web browser.

Please note that certain device specific features are not available when the application is deployed as a web application and accessed via web browser.

The following table lists the features supported by application mode.

Feature	Compiled	Browser Based
GPS	✓	✓
Capture Picture	✓	X
Capture Signature	✓	X
Capture Sound	X	X
Barcode Scanning/Reading	✓	X
Download Attachments from MDT (All File Types)	✓	X
Upload Attachment from MDT to Server (Only Captured Picture and Signature)	✓	X
Maps	✓	✓
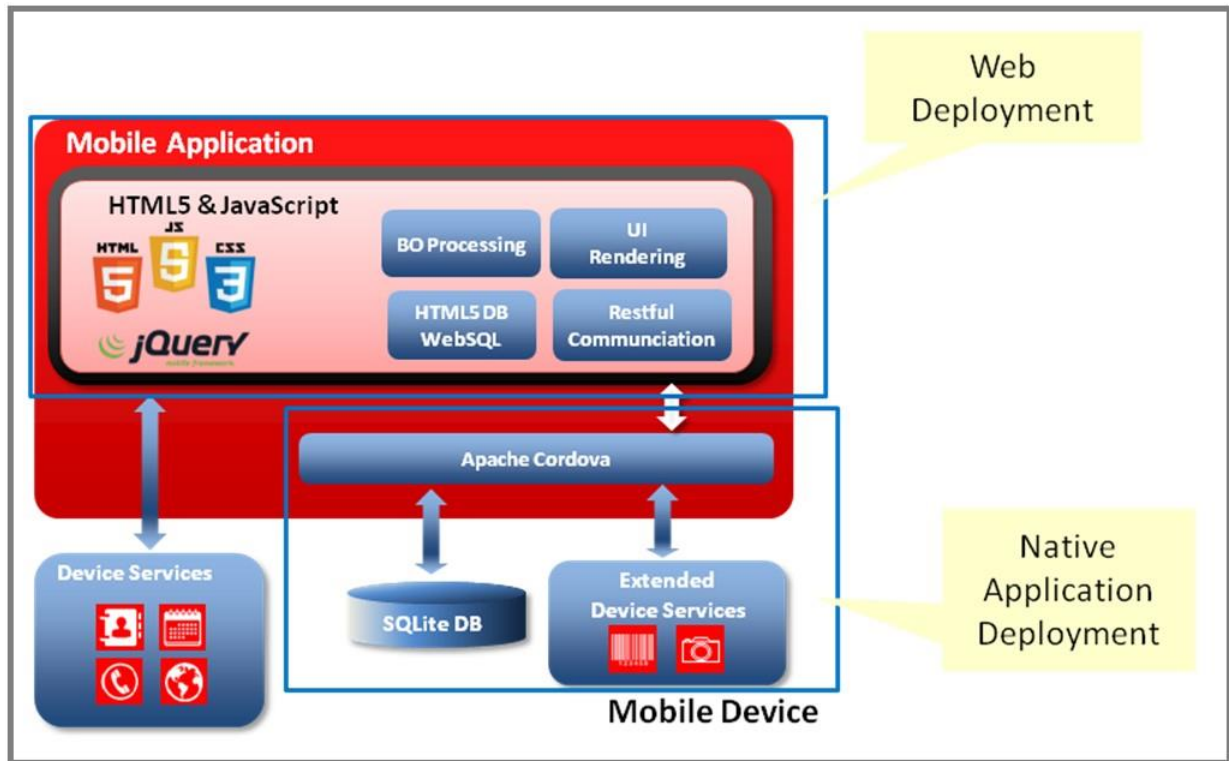
Inbound and Outbound Communication

Inbound and Outbound communication between MWM mobile and server applications is based on RESTFul services and JSON payload. In situations where device is offline at the time of making outbound HTTP request communication modules of MWM Mobile application ensure that delivery of the message when device is back online and communication with server is reestablished.

Please reference Mobile Components for more details on communication between mobile and server applications.

Chapter 2

Development Environment Setup

This section provides information needed to setup the development environment for the mobile application. Implementers can use this environment to add new features and test their code locally or on devices using the steps provided in this section.

Prior to setting up the development environment, you must have completed general server side configuration. Please reference the Oracle Utilities Mobile Workforce Management *Server Administration Guide* for information.

Installing Prerequisite Software

Please reference the chapter on installing prerequisite software in the Oracle Utilities Mobile Workforce Management *Mobile Application Installation and Deployment Guide* for information.

Source Code

Required libraries and source code for development and customization in the local environment can be copied from the <PRODUCT_HOME> directory of the shared build environment that is created as part of the initial install. Please refer to the Oracle Utilities Mobile Workforce Management *Mobile Application Installation and Deployment Guide* for information on the initial install.

The <PRODUCT_HOME>/source/www folder in the shared build environment contains the source files which could be linked to a version control system to enable code contributions from multiple local development environments.

The www directory needs to be copied over or linked to the local Apache Cordova project. This project can be used to locally build native applications.

Apache Cordova Project

The same Apache Cordova project can be used to create native applications for different mobile operating systems.

Complete the following steps to create an Apache Cordova project:

1. Install Cordova.
   Please reference the installation instructions delivered with the Cordova product. The section on "The Command-Line Interface" includes steps to install the CLI tool information about Cordova project commands.

2. Use the Cordova Windows 10 project for building Windows 10 Runtime.

Plugin Installation

Note: The following Cordova instructions apply only to Windows 10.

Cordova includes a set of "core plugins" which are used by the mobile application to access native device features such as the file system, camera, geolocation and so on. Aside from using the Cordova core plugins, implementers can also develop their own plugins or use other available plugins. These plugins are described in the Apache Cordova documentation.

Use the CLI tool to installing/uninstalling plugins. This is done by using the "plugin add" command:

Example:

```
cordova plugin add <path to plugin>
```

Note: Please reference the Oracle Utilities Mobile Workforce Management *Installation and Deployment Guide* in the "Plugin Configurations" section for the list of required plugins for the mobile application including the actual paths for the plugins with the release versions being used. The following section provides an overview of the plugin functions.

Device Plugins

This section provides a high level description of the device plugins used with the Oracle Utilities Mobile Workforce Management.

Device - The Cordova Device plugin defines a global device object, which describes the device's hardware and software.

Camera - The Cordova Camera plugin provides an API for taking pictures and for choosing images from the system's image library.

File - The Cordova File plugin implements a File API allowing read/write access to files residing on the device.

Geolocation - The Cordova Geolocation plugin provides information about the device's location, such as latitude and longitude. Common sources of location information include Global Positioning System (GPS) and location inferred from network signals such as IP address, RFID, WiFi and Bluetooth MAC addresses, and GSM/CDMA cell IDs.

InAppBrowser - The Cordova InAppBrowser plugin provides a web browser view that displays when calling window.open().

Network Information - The Cordova InAppBrowser plugin provides an implementation of an old version of the Network Information API. It provides information about the device's cellular and wifi connection, and whether the device has an internet connection.

Barcode Scanner - This is an external Barcode scanner plugin for Cordova which is optional and can be used with the application. The plugin provides implementation for scanning barcodes and provides the type and the barcode for a scanned item.

SQLite - This is an external SQLite plugin for Cordova which is optional and can be used with the application. The plugin provides implementation for using SQLite Database on the device. The plugin uses the same API as the HTML5 WEBSQL database.

Signature Capture - This is an external signature capture plugin for jQuery. This is a Javascript only plugin and does not require installation using the Cordova add plugin command. The plugin file needs to be included in the www/libs/jSignature folder. It provides a JavaScript widget for simplifying the creation of a signature capture field in a browser window, allowing a user to draw a signature using mouse, pen or finger.

Encryption

The encryption plugin is only used on devices running Android or Windows 10 platforms to:

- Store passwords encrypted on devices
- Store transaction data encrypted on devices (BO, Inbound, Outbound records)

For iOS, encryption is handled with native device encryption. If the Oracle Utilities Mobile Workforce Management Mobile Application is deployed as a web application and is being accessed on the device via web browser, the encryption module is not used, but rather, transaction data is stored in plain text format in offline database. The web application is designed to be used for non-production use cases such as development/testing.

The encryption module is implemented entirely on the device side and there is no associated server side counterpart.

Transaction data generated by the application is securely stored in a non readable encrypted format accessible only to the authenticated user. User credentials are securely stored in private storage of the application in encrypted format for offline authentication. A Symmetric Key for Encryption is generated on the server. This key changes every time a new user session is started.

Encryption features can be enabled or disabled per specific mobile device. They can also be enabled system-wide via Feature Configuration by setting the Encryption value as "Default".

Local Testing

The HTML5 code added to the www directory can be tested locally using a Google Chrome browser. For device-specific features, such as a camera and/or barcode scanner, the testing must be done using native applications.

Use these steps to test the application in non-production mode:

1. If not already installed, install the Google Chrome desktop browser.
2. Create a shortcut to the executable on your desktop.
3. Right-click the shortcut and choose Properties, then append the following to the Target property:

   ```
   --user-data-dir="C:/Chrome dev session" --disable-web-security
   ```

   to disable cross-domain JavaScript security.

4. Start Chrome via the shortcut and load the mobile application (location: www/index.html in your local system).
5. If the login page does not appear or does not work correctly, reopen the Chrome shortcut properties, correct the path specified in the `user-data-dir` parameter (to specify the Chrome location), then reopen Chrome and retry the login.

Alternatively (instead of modifying the shortcut Target property), you can open a command window and enter:

```
chrome.exe --user-data-dir="C:/Chrome dev session" --disable-web-security
```

As with the shortcut, if the login page is not displayed properly, correct the path specified in the `user-data-dir` parameter in the command and rerun it.

Building and Deploying the Mobile Application

This section provides information on how to deploy the Oracle Utilities Mobile Workforce Management Mobile Application on various device types. Please refer to the Oracle Utilities Mobile Workforce Management *Mobile Application Installation and Deployment Guide* in the chapter titled

"Deploying the Mobile Applications" for steps on deploying the mobile client application as a web application.

Chapter 3

Oracle Utilities Mobile Library

This section describes the key modules and APIs that are available for implementing new user interface pages and application features.

Device Communication

This section provides information on communication between the server application and mobile devices. Although we have two categories of messages, inbound and outbound, they are both transferred via HTTP requests initiated by device.

Device Inbound Messages

Configuration

MDT type uses the ASYNC_INTERVAL (seconds) property to configure the interval at which a REST service(M1-SyncData) will be invoked by client. Inbound message is processed by a script which is specified on incoming message (SCRIPT column). Inbound scripts should be mapped to the inboundMsgFiles property in the Product Configuration mobile component.

Message Storage

Messages received are stored in F1_INBOUND intermediate table on device DB. Please reference the Database Schema section for more information.

Inbound Message Event API

Once a message is downloaded and saved to intermediate table it is handed over to inbound processing script. This processing script should be implemented as follows:

Processing Script Code Structure

```
ouml.Inbound["M1-MCPDpAsgn"] = (function (ouml){

function process(msgEvent) {


}

return {
        process: process
    };
})(ouml);
```

M1-MCPDPAsgn is an example script code. This should be replaced with your actual script name. This script needs to implement a process method that is required to be exposed as public method of this module.

The Oracle Utilities Mobile Library invokes a process method with an event object with the following structure.

- **msgEvent.message** – inbound message in JSON format (format of this message is as defined on server, specific to a script)

- **msgEvent.error(ouml.ClientError)** – this method should be called in case an error occurs in processing this message. An instance of ouml.Error should be passed to it. This error message is saved to F1_INBOUND table's error column.

- **msgEvent.complete(transaction)** – this method should be called on successful message processing. Transaction used, if any, should be passed to this method and same will be used by the Oracle Utilities Mobile Library to update the F1_INBOUND'S PROC column. If no transaction is passed then a new transaction is created.

Message Acknowledgements

On successful download and save to the intermediate table, a message delivery acknowledgement is sent back with very next REST service call. This only indicates the delivery part, not the processing. On successful message processing another acknowledgment is sent with a flag to indicate whether or not the processing was successful. If during message processing an error occurs, the same error is also sent back to the server.

The Input to the REST service contains following payload:

```
{
"msgId": msg_id column value from F1_INBOUND,
"isProc": true/flase (true when PROC column value is Y)
"errorData": {error object}, error column value}
}
```

Device Outbound Messages

An outbound message is essentially a RESTful service invocation initiated by client which delivers a message (JSON payload) to that service on server. There are two types of outbound RESTful invocations modes from client:

Online Mode

A service invocation where response from the service is required to proceed further with the business flow. For this type of outbound call, client has to be connected to network as if device is offline we cannot proceed further.:

Online Mode API	Parameters	Description
ouml.AJAX.invokeService	service – service to be executed args – {onSuccess: <callback>, onFailure: <callback>, method: <GET or POST>, contentType: <content type header>, headers: {<all headers passed as is to ajax call>} } payload - JSON Data	Invokes a service immediately (device has to be connected to network) and returns the results via an asynchronous callback. This API adds mandatory headers required for authentication and connecting to server.

Offline Mode

A message is posted to a service however the actual call to service would be made only when the device is connected. Such outbound messages (service calls) are delivered to server as and when the device is connected and client business flow is not dependent on response from server. However it is ensured that no message will be lost and it will be delivered to server eventually. Client ensures that message sent via this outbound module are stored in offline storage and delivered in same sequence they were posted. Application crash or network connectivity should not result into any message loss.

Offline Mode API	Parameters	Description
ouml.OutboundWorker.queueOutbound	args – { transaction:<tx object>, onSuccess: <callback>, onFailure: <callback>, input: { service: <service name>, payload: <JSON data>} }	Message posted via this API will be saved to F1_OUTBOUND table using passed transaction else a new transaction will be used. Transaction object will be returned via success callback so that same can be used to execute the next transaction in case of multiple commits. Whenever device comes online the payload will be delivered to the specified service.

Server Communication

This section describes the outbound and inbound messaging used by the system.

Server Outbound Messages

This section refers to messages that are outbound from the server and inbound to the mobile device.

Outbound messages are maintained through the M1-MessageToDevice business object. The different states that the outbound message can transition to are defined and managed by the business object's lifecycle.

For data synchronization the device sends:

- Device ID

- A list of acknowledgements. Includes Remote Message ID, PROC_SW Y/N (whether it's been processed yet), and optional error details.

The device receives:

- A list of new messages to be processed.
  This includes Remote Message ID, business object, message name, payload, priority.

- Ordered by priority (with high-priority messages first) and then in Created Date-Time order.

- Number of messages is based on bucket size.

The following "rules" apply for client applications that process outbound messages:

- Valid MDT_ID
  Defined in the server application.

- Number of messages received is based on bucket size defined on the MDT Type.
  There may be more or less messages than what is received, however the bucket size limits the number of messages received at one time. The system continues to send the messages in batches until the queue is empty.

- If the device does not acknowledge receipt of the message, the same message will be sent again.
  It is possible to set "callback" settings to cancel messages so that they aren't continually sent.

- If the device does acknowledge receipt, the message must be processed.
  Messages should be processed in order, high-priority first.

- Error details are provided in the outbound message.

If the caller wants to work with the output message IDs from M1-InvokeRSIScript or M1-GetRSIIdsByContext , it could be an RSI_ID (30 chars) OR a REMOTE_MSG_ID (14 chars).

All Callback and Error scripts, ditto. The existing element <rsiMessageId> may be 30 chars or 14 chars.

Callbacks are done only for messages that have not been delivered. If a message is delivered but never processed, it will remain in Queued status forever, unless some other process handles it. You may want a Monitor for that.

Output RSI ID for Various Business Services, Service Scripts, etc.

- Business Services: Invoke Remote Script (M1-InvokeRSIScript) and Get Remote Script Invocation By Context (M1-GetRSIIdsByContext)

- Output message IDs can now be either an RSI ID (30 characters long) or a Remote Message ID (14 characters long).

- Callback and Error Scripts

- Existing schema message IDs can now be either an RSI ID (30 characters long) or a Remote Message ID (14 characters long).

Callback Logic

Callbacks are done only for messages that have not been delivered. If a message is delivered but never processed, it will remain in Queued status forever, unless some other process handles it.

Call back is configured, in seconds, under **Master Configuration** > **Global Configuration**, field: **Remote Script Call Back Seconds**.

This indicates the number of seconds that should pass (from the message's creation date time) before the callback is executed. This works when the Remote Message monitor batch process triggers the remote message's monitor algorithm (which executes callback scripts when applicable).

Remote Message Batch Monitor

The remote message monitor, a timed monitor batch process, can be set to monitor the rules associated with the current state of messages that go between the server application and mobile devices. It is recommended that you set this monitor to run very frequently such as every 5 minutes so that processed messages can be transitioned to a non-queued state (to improve performance on queries for unprocessed messages).

Server Inbound Messages

This section refers to messages that are outbound from the mobile device and inbound to the server.

As described in Client side outbound messages section above, these messages to server are delivered by invoking specific services as per the given context or business logic, e.g., Get Shift, update shift, update task,   etc.

Guaranteed Delivery

A special kind of inbound messaging called **Guaranteed Delivery** ensures that messages from a device are stored in the application database first, and then processed afterwards. This ensures that even though the message cannot be processed immediately because of other factors, the message is at least guaranteed to be delivered to the server.

Guaranteed Delivery Algorithm

The remote message guaranteed delivery algorithm, M1-REMMSG-GD, processes guaranteed delivery requests through remote message creation (through the business object M1-CrewMessage) and state transition. Your implementation must configure the base algorithm on Installation Options/ Guaranteed Delivery. This is configured by navigating to **Installation Options** > **Algorithms**, System Event: **Guaranteed Delivery**.

Remote Message

The **Remote Message** table uses a Device Message ID field that stores a unique ID sent from the mobile device to distinguish inbound messages sent from the server application. This field is later used by the Guaranteed Delivery (M1-REMMSG-GD) algorithm to verify whether or not an inbound message already exists in the Remote Message table before creating a new record (to avoid duplicate entries for inbound messages).

Logging

System logs are sometimes needed to diagnose how the server application is communicating with devices, investigate errors, or for other troubleshooting or informational purposes.

Mobile log files can be accessed in the MDT portal under the **Log** tab.

Changing Log Settings from a Device

Device users can change log settings from the Oracle Utilities Mobile Workforce Management **Mobile Application Settings** page. This includes turning logging on or off, as allowed by the user's permissions, setting the log level, and setting appenders.

Log Appenders

The logging module supports the following types of appenders to display logging messages:

- **Console Appender** (CONSOLE): Writes log messages on the web console.
- **File Appender** (FILE): Writes log messages in a local file on the client. The log files in the client are then sent to the server when requested.
- **Remote Appender** (AJAX): Sends log messages (json/xml/text) to the server with an asynchronous HTTP request.
- **Popup Window Appender** (POPUP): Opens a new window/sub window in the browser and writes log messages in real time.

Users can enable more than one appenders at the same time to write logs from setting page of application

Log Message Format

Log entries use the following format.

[Unique Prefix] - Date Time Log-Level Log Message (Origin Module Line Number)

Log API

The Oracle Utilities Mobile Library Logging module exposes the APIs required by your implementation to facilitate system logging. Any application module that requires logging uses this

module with the single log instance maintained for the complete application. Logs get the appropriate instance from ouml.JSLogger and use the exposed API.

For example to log an info message your implementation would use:

- ouml.JSLogger.info('Your message ');

- Extra public APIs exposed by this object (not part of the Oracle Utilities Mobile Library or parent business object)

    - **mdtdebug(message)**: The module that needs to log a **framework level debug** message calls this method.
    Passes the log message arguments to the methods.

    - **debug(message)**: The module that needs to log a **debug** message calls this method.
    Passes the log message arguments to the methods.

    - **info(message)**: The module that needs to log an **info** message calls this method.
    Passes the log message arguments to the methods.

    - **warn(message)**: The module that needs to log a **warn** message calls this method.
    Passes the log message arguments to the methods.

    - **error(message)**: The module that needs to log an **error** message calls this method.
    Passes the log message arguments to the methods.

    - **perf(message)**: The module that needs to log a **perf** message calls this method.
    Passes the log message arguments to the methods

    - **fatal(message)**: The module that needs to log a **fatal** message calls this method.
    Passes the log message arguments to the methods

    - **setLevel(level)**: These methods set the logging level of the logger instance that the application has acquired initially. The level that is to be set should be within the set of levels supported by Logger. Else default logging level will be used

    - **syncLogFile()** : This method synchronizes the log files to the server.

Offline Database

This section provides information regarding client side offline database tables and APIs available to interact with the offline database.

A WebSQL database is used for local data storage if the application is opened in a web browser. If the application is installed as a native app on a device and "sqliteDB" property is set in the Product Configuration mobile component, then the SQLite DB on the device is used. The database is initialized with an initial size of 5MB.

Database Schema

API

getHandle - Returns the DB handle object. This returns a singleton instance of an object that should be used for any DB transactions.

Tables

At application launch, the tables indicated below are created in the browser database or in SQLite if they do not already exist. You can reference this schema and browse the database during development or debugging.

F1_BIZOBJ

This table stores both deployment and transaction data for all business objects. GEN_COL1 to GEN_COL10 can be used to store specific fields that can be used to query the business object.

Offline Field	Description
BO_KEY	combination of business objects PK1-PK5 (pk1^pk2^pk3^pk4^pk5)
BO_CODE	Business object code
MO_CODE	Maintenance object code
DATA	JSON data for a business object
TYPE	Type of data (DEPLOYMENT or TRANSACTION).
DATE_UPDATED	Timestamp(local) when the data was modified.
VERSION	Version of the record.
GEN_COL1 - GEN_COL10	Generic columns for storing business object attributes used in search and application logic. The default value for the number of columns is set to 10 in the OUML Product Configuration via the property bizObjGenColumns.

F1-Inbound

This table supports inbound messages.

Offline Field	Description
MSG_ID	Unique message ID for the inbound message.
PAYLOAD	JSON Data received in a message.
SCRIPT	Script code (message processing script).
PRIORITY	Priority of the message.
ACK_REQUESTED	Flag to indicate whether acknowledgement is requested.
ACK	Flag to indicate whether acknowledgement was returned.
PROC	Flag to indicate whether the message is processed.
PROC_ACK	Flag to indicate whether processing acknowledgement was sent.
ERROR	Error message received during processing, if any.

F1-Outbound

This business object supports outbound messages.

Offline Field	Description
ID	Unique message ID of the outbound message.
SERVICE	Service name.

PAYLOAD	Service input payload.

Product Configuration

Each application layer has its own Product Configuration mobile component in which a new property can be added. A property defined in the lower layer can also be overwritten by defining a new property with the same name.

Some of the properties that are of type array cannot be overridden completely but values from each layer are merged. Please reference the description of each property.

API -ouml.Config

- **restServerURL** - OUAF REST API URL.

- **mobileAppURL** - Mobile app URL.

- **DEFAULT_MDT_URL** - DEFAULT MDT URL.

- **DEFAULT_DEPLOYMENT_ID** - DEFAULT_DEPLOYMENT_ID.

- **mainMenu** - menu items that should be available on every page menu.

- **applicationFolder** - A folder name used to store the files on local device filesystem.

- **getConfig** - Returns the value of a property (the property available in topmost app).

- **boFiles** – list all the files required by a business object. If the only file that a business object requires is same file as the name of business object and is available in scripts/bo folder then no need to include that. In case of CM config, files are assumed to be present in scripts/bo folder.

- **getBOFiles** - Returns the JS file names required by a business object. This API is internally used by ouml.Loader.loadBO so implementers will not have to ever use this. This property returns the value of boFiles variable after merging it from all layers.

- **pageFiles** – list all the files required by a UI page (business object or non-business object). If the page id (div having data-role =page) is same as file name then no need to include that file. provide absolute path starting from product folder (e.g. cm/taskList.js).

- **getPageFiles** - Returns the JS file names required by a Page(bo pages too) UI. This API is internally used by ouml.Loader.loadPage API so implementers will not have to ever use this. This property returns the value of pageFiles variable after merging it from all layers.

- **commonJSfiles** – List all JS files that should be loaded on successful login. This property is used by login module and it loads all the files defined at different layer, after merging it from all layers. Common JS files like plugins.js or common.js which hosts common APIs not specific to a business object or a Page should be declared in this property. File should contain the path starting from product folder name (e.g. m1/scripts/plugins/plugins.js).

- **inboundMsgFiles** – List mapping between an inbound script and corresponding file containing the script. File MUST be present in scripts/inbound folder. CM can override base.

- **getInboundMsgFiles** - Returns the inbound message handler file names for a given script code.This API reads the value from inboundMsgFiles variable in the Product Configuration mobile component of each app layer and returns the files from appropriate layer. This API is internally used by the Oracle Utilities Mobile Library and implementers will not have to use this.

- **capabilitiesMapping** – define a mapping between a capability type (defined on server) and corresponding script to be executed on client for a given capability. These scripts should be defined in common.js (e.g. cm/ui/common.js) or some JS file that is loaded via commonJSFiles so that whenever a capability request (e.g. scan barcode) is made this file should be already loaded.

- **oracleMapProperties**- Used to configure the Oracle MapViewer properties. The **serverConfig** property is very important. This is the name of the Feature Configuration created for the Oracle MapViewer on the server. The client gets all the MapViewer information like URL, Datasource, Tile Layer etc using the Feature configuration. Besides the serverConfig the images for activities can be changed here. The style for the information window which pops up on clicking an activity marker can also be modified here.

- **sqliteDB** – Set to false by default out of the box. If set to true the SQLite plugin is used to create a SQLite database on the client devices instead of using the HTML5 WEBSQL database. This flag can be set to true only if the SQLite plugin is installed for the Cordova project used to build the native application.

Encryption APIs

For android devices, encryption is provided by a cordova plugin. However, instead of using cordova plugin APIs directly, you should use the APIs indicated in the table below in the ouml.Crypto module.

The Oracle Utilities Mobile Library uses these APIs internally to store data to the F1_BIZOBJ table if the encryption is enabled for devices. Please reference Chapter 2: Encryption for more information.

These APIs return the original input as is if the encryption is not enabled for this device. Callers of the APIs can check the output in success callback to confirm if the data was indeed encrypted (or decrypted).

API	Parameters	Description
ouml.Crypto.encrypt	args = {onSuccess: <success callback>, onFailure:<failure callback>, input: {data: <text string or an array of text strings>}, encryptionKey : <optional, key to be used>}	Encrypted input data will be returned via success callback as {output: <encrypted text>, encrypted: <true|false>}. If the input was an array then output will be an array e.g. {output: []} with each array element corresponding to input array element. Encryption key is not required unless you have to use a different encryption key than what is configured on server. Encrypted property is set to false if no encryption was done in case of iOS device or encryption not enabled for this devce.
ouml.Crypto.decrypt	args = {onSuccess: <success callback>, onFailure:<failure callback>, input: {data: <text string or an array of text strings>}, encryptionKey : <optional, key to be used>}	Decrypted input data will be returned via success callback as {output: <decrypted text>, decrypted: <true|false>}. If the input was an array then output will be an array e.g. {output: []} with each array element corresponding to input array element. Encryption key is not required unless you have to use a different decryption key than what is configured on server. Decrypted property is set to false if no decryption was done in case of iOS device or encryption not enabled for this devce.

Cordova Encryption Plugin APIs

The plugin call takes the following parameters:

1. **success**: Function name of the function to be called on successful execution of the plugin. This function is called with a string parameter depending upon the value of the action parameter.

2. **failure**: Function name of the function to be called on execution failure of the plugin. This function is also called with a string parameter containing the error message of the error which occurred while executing the plugin leading to failure.

3. "**Crypto**": The plugin identifier.

4. **action**: The action parameter passed to the plugin. This includes one of the following values:

   a. **encrypt**
   For this action, the plugin will return the encrypted string of the input text on success. The encryption key will be passed along with the input text as parameter to the plugin in json format.

   b. **decrypt**
   For this action, the plugin will return the decrypted string of the input encrypted text on success. The encryption key will be passed along with the input text as parameter to the plugin in json format

   c. **hash**
   For this action, the plugin will return the hashed value of the input string on success.

5. **json**: The input parameter to plugin in json format. It will contain the input string to be encrypted along with the symmetric encryption key to be used for encryption.

Process Details

1. Users log in to the system in online mode. The user credentials are stored in persistent storage using the hashed value obtained from custom Cordova plugin for offline authentication.

2. After login the following device options are fetched from server in online mode and are stored in local storage:

   a. MDT_ENCRYPTION_FLAG

   b. MDT_ENCRYPTION_KEY

   If the user logs in offline mode then the last stored values of these device options are used in the application.

   If transaction data exists on the device then the new values obtained from the server for these device options are not overwritten in the local storage. Thus the MDT_ENCRYPTION_FLAG and MDT_ENCRYPTION_KEY device options values on the device cannot be changed after transaction data is generated on the device.

3. Using the MDT_ENCRYPTION_FLAG device option the encryption module can be turned on ('M1ON') or off ('M1OF) for a particular device using the MDT portal page.

4. If MDT_ENCRYPTION_FLAG set to 'M1ON' then the transaction data generated on the hybrid client is stored in encrypted format in local storage and its decrypted after reading from local storage to get the original form before use. If MDT_ENCRYPTION_FLAG is 'M1OF' then all transaction data on device is stored in readable text format.

   If the value is set to is M1DF (default), then the value is fetched as per the Master Global Configuration.

5. The MDT_ENCRYPTION_KEY is stored in local storage in encrypted format. It is encrypted using the base64 encoding value of username:password as encryption key and using the same encryption algorithm which is used to encrypt transaction data.

Deployment data is not encrypted on the device as it is not transactional data.

Deployment

The application consists of code and metadata:

- Code is installed (for native apps) or deployed (for webapp) as an application.

- Metadata that is required for the application to work properly, is downloaded on a successful logon in JSON format and stored in the offline database. Deployment metadata is stored in F1_BIZOBJ table with DEPLOYMENT as value in "type" column. The Oracle Utilities Mobile Library provides various APIs to access deployment metadata in simple format Data consist of various Oracle Utilities Application Framework objects including:

  - Labels

  - Lookups and Extended Lookups

  - Messages

  - Business Object Lifecycle

  - Business Objects Data (non transactional objects)

These objects can be configured on the server. Please reference the Oracle Utilities Mobile Workforce Management *Server Administration Guide* for more details.

API (module - ouml.Metadata)

API	Parameters	Description
getLabel	Label/field Id	Returns the label description (should be used instead of hardcoding text strings on UI). Check ViewModel wrapper API for usage on HTML pages.
getLookup	Lookup ID	Returns an array of items containing lookup value and description in the format [{lookupValue: "", description:""},] Check ViewModel wrapper API for usage on HTML pages.
getLookupDesc	Lookup, lookupValue	Returns the description for a specific lookup value of a lookup. Check ViewModel wrapper API for usage on HTML pages.
getExtLookup	Extended Lookup BO Name	Returns an array containing lookup value and description in the format [{lookupValue: "", description:""},] Check ViewModel wrapper API for usage on HTML pages.
getExtLookupDesc	Extended lookup BO Name, lookup Value	Returns the description for a specific lookup value of a extended lookup BO Check ViewModel wrapper API for usage on HTML pages.

API	Parameters	Description
~~getMessage~~	~~Message category,~~ message Id	~~Returns the message. (To get~~ formatted message with parameters, use ouml.ClientError API)
getNextBOStates	Bo name, bo status	Returns a list of next valid states which the business object can transition to from a given state. It returns an array of objects with this format {boNextStatusLabel, boStatus, role}
getStatusReasons	Bo Name, status	Returns an array containing the status reasons valid for a given state. Format of the output is [{description: "", selectability:, "", statusReasonCd: ""},{}]
isFinalBOState	Bo Name, Status	Returns true if there are no next valid states for a given business object and state, otherwise returns false.
getBOInfo	Bo Name	Returns all information(metadata) about a BO. A JSON object with description, owner code, each valid states and related info.
getAncestors	Bo Name	Returns an array of items with business object information for all business objects in the hierarchy. At 0th index is the top most parent and given business object at the end of the array.
getAncestorNames	Bo Name	Returns an array of business object names for all business objects in the hierarchy. At 0th index is the top most parent and given business object at the end of the array.

To read the value of a Business Object in deployment, you can use BOEntity APIs.

Date and Time APIs

API (module ouml.DateTimeUtil

Sr No	API	Parameter	Description
1	getFormattedDate	Base date time object	Converts and returns local device date string value from base date time object , formatted as per DATE DISPLAY Format configured for MDT
2	getFormattedTime	Base date time object	Converts and returns local device time string value from base date time object , formatted as per Time Format configured for the MDT User on the Display profile.
3	getFormattedTimeFromSeconds	Time in seconds	Returns the formatted time value in format HH:mm:ss
4	Time	Base date time object	Converts and returns local device date time string value from base date time object, formatted as per DATE DISPLAY TIME Format configured for the MDT User on the Display profile.
5	getBaseDttmFromLocalDttm	Local device Time in milliseconds	Converts and returns a base date time string value from local device time in YYYY-MM-DD-HH.MM.SS format. If no local device time is passed the current device time value is used
6	getStdBaseDttmFromLocalDttm	Local device Time in milliseconds	Converts and returns a base date time string value from local device time in YYYY-MM-DD-HH.MM.SS format. If no local device time is passed the current device time value is used Same as getBaseDttmFromLocalDttm()
7	getBaseDttm		Converts and returns a base date time string value in YYYY-MM-DD-HH.MM.SS format from current local date time.

Sr No	API	Parameter	Description
8	getLocalDttmFromBaseDttm	Base date time String value	Converts and returns local date time Object from base date time String value. Returns a Date Object.
9	getStdLocalDttmFromBaseDttm	Base date time String value	Converts and returns local date time string value from base date time string value in YYYY-MM-DD-HH.MM.SS format
10	getLocalDttm		Returns current device date time string value in YYYY-MM-DD-HH.MM.SS format
11	getISOFormattedDateTime	Base date time String value	Converts and returns local date time string value from base date time string value in ISO Format (YYYY-MM-DDTHH:mm:ss.sssZ)
12	getBaseDttmFromISOFormattedDateTime	Local date time string value in ISO Format	Converts and returns base date time string value from local date time string value in YYYY-MM-DD-HH.MM.SS format
13	getCurrentDate		Returns current device date string value in yyyy-MM-dd format. Use this instead of new Date(). It gives correct value even if the device clock is out of the sync with the device's time zone. It does so by internally applying device date time correction.
14	getBaseDate	Local device Time in milliseconds	Converts and returns the base date object from local date object. If no local device time is passed the current device time value is used
15	getCurrentDttm		Returns current device date time string value in yyyy-MM-dd-HH.mm.ss format
16	getCurrentDeviceDate		Returns current device date time object

Sr No	API	Parameter	Description
17	getCurrentStdDttm		Converts and returns a base date time string value in YYYY-MM-DD-HH.MM.SS format from current local date time.
18	calculateDttm	Operation "F1AT" or "F1DT", source date time string value, destination date time string value, n no of seconds value.	Calculates and returns a date time string value in YYYY-MM-DD-HH.MM.SS format after performing add or subtract by number n operation

Properties

The following properties are downloaded at login.

- Properties
- KeyValue
- sessionId
- decimalSeparator
- API - ouml.Properties

Business Objects

Please reference the **Oracle Utilities Application Framework** documentation for details of server side implementation and metadata associated with business objects.

Mobile client implementation of a business object includes the following content:

- **JavaScript** – A JavaScript class which can be instantiated to invoke BO APIs (save, update, or change BO state).

- **HTML** – HTML content for the BO UI. This is mostly used as a template in include UI sections.

- **UI JavaScript** – JavaScript content, Knockout ViewModel.

- **boFiles** – Product Configuration property for the BO JS files. This is only specified if the business object name and the JavaScript name are not the same, or if this layer is adding a customization and does not own the BO.

- **pageFiles** – Product Configuration property for the UI JS files. This is only specified if the file name is not same as the Page ID, or if this layer is adding a customization and does not own the BO.

Refer to the base product Business Object and Product Configuration mobile components for examples.

Business object data is received by the Inbound Service and processed by inbound scripts. See the Device Inbound Messages section for more information.

Javascript in the Oracle Utilities Mobile Library uses revealing module pattern or prototype pattern or a combination of both where prototype is wrapped in revealing module. Every JavaScript class/module is attached to a namespace that starts with "ouml".

Please make yourself comfortable with Object Oriented Programming in JavaScript which is a pre-requisite for writing new BO classes. (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model)

Business Object JavaScript (JavaScript)

Every BO JS must extend the GenericBusinessObject class, if there is no parent to this business object. Otherwise, it must extend the parent class. The OUMLTools generates the template code for new content.

GenericBusinessObject APIs

API	Parameters	Description
<constructor>	Data – BO JSON data (optional). Version – offline data version.	Data – If BO JSON data is provided it will be set to this.data property of this instance. To create a new BO instance you should use ouml.BusinessObjectFactory.getBusinessObject Version – The default value is 0 for a new BO record to be added to the offline DB.
getData	JSON path of a file	Returns the value of a JSON path on BO data. Directly accessing a value in a nested JSON structure might result into "undefined is not a function" error so to avoid that use this API.
hasUndefinedOr EmptyField	JSON path of a field	Returns true if the field is undefined or is empty.
setData	Field, Data	When supplied with only one argument, it should be the BO JSON data. If you need to set the value of a specific field pass both field and data.
setFieldData	Field,Data	Set the value of a specific field to given data.

BO Plugins

BO Enter and PostProcessing plugins are defined as JSON object inside a BO class in a private variable named **plugins** (you can name it anything).

```
var plugins  = {
  "POSTPROCESS": {
    "active": [
      {"sequence": 10, "plugin": "M1-MCPTSUpd", "params": {}},
      {"sequence": 20, "plugin": "M1-MCPSndCmp", "params": {}},
      {"sequence": 30, "plugin": "M1-MCPTsLnIn", "params": {}}
    ],
    "inactive": ["M1-TaskUpd"]
  },
  "states": {
    "ENROUTE": {
      "ENTER": {
        "active": [
```

```
              {"sequence": 10,"plugin": "M1-MCPPRCSTL","params": {"key":
         "value"}},
              {"sequence": 20,"plugin": "M1-MCPSTVal","params": {"key":
         "value"}},
              {"sequence": 30,"plugin": "M1-MCPSHSSv","params": {"status":
         "IN-SVC"}}
            ],
            "inactive": []
          }
        },
        "DISPATCHED": {
          "ENTER": {
            "active": [],
            "inactive": []
          }
        }
      }
    }
```

If there are no Post Processing plugins on a BO, active and inactive nodes should both have an empty array ([]), but you *must* define the JSON path for easy addition of a new plugin later (the OUML assumes that the JSON path is present, whether there are any plugins or not).

For state-specific plugins: If your specific state has no plugins the complete node for that state can be deleted, or active/inactive should have an empty array. For example, in the above JSON code, this BO might have other states as well (in addition to ENROUTE and DISPATCHED). If that's the case, OUML will assume that those states have no plugins.

Definition of a Plugin

An active plugin defines its sequence, the plugin script name, and any parameters that will be passed when executing the plugin script.

```
{
"sequence": 10,
"plugin": "M1-MCPPRCSTL",
"params": {"key": "value"}
},
```

All inactive plugins are stored as an array of plugin script names (an array of strings).

The plugins are called for a specific enter or post processing event, top down in the BO hierarchy, and within each BO they are performed in the sequence order.

BO Instance – Implementers APIs

The following APIs (getter methods) must be implemented by each BO class in a BO prototype chain. Each of these methods must return a reference to a module private variable.

getBOName() – BO API (override)

A local private variable, var boName, should be defined in the encapsulating module (before the definition of BO constructor) and returned by this method.

```
var boName = "M1-Assignment"
```

If the BO class is a customization (the BO class, not the owner of the BO), then this method and the variable is not required, since the BO name is the same for a customization class. It is just adding few extra plugins or is used to override other APIs for the base-provided BOs (without creating a child BO).

getParent() – BO API (override)

A local private variable, var parent, *must* be defined in the encapsulating module (before the definition of BO constructor) and returned by this method.

```
var parent = ouml.BOHelper.getPrototypeFor(boName);
```

getPlugins() – BO API (override)

A local private variable, **var plugins**, should be defined in the encapsulating module (before the definition of BO constructor) and returned by this method. This variable is the same as described in BO Plugins.

BO Instance – OUML APIs

> **Important**: The following are the only two APIs that execute plugins attached to a BO or BO state. Every other APIs available on a BO instance use raw data and perform insert/update actions on the database without plugin execution. Thus, additional care is advised when you use APIs other than the two described below.

executeStateChange (args) – BO API

This API transitions the BO to the next state and then saves the data to offline. It will invoke the getPlugins implementation of a BO to get Enter and PostProcessing plugins and execute all the plugins in a single transaction.

args is an object {} with the following properties:

- **onSuccess** – Success callback that is invoked on a successful state transition.
- **onFailure** – Failure callback to be invoked on any error.
- **transaction** – If an existing transaction is to be used, otherwise a new transaction is created.
- **input** – Input that is passed to every plugin script.
- **input.verControl** – Set the value to OVERRIDE or NOVERSION if default behavior is not required. (See the next sections of this document for more information on version control.)

The BO reference is attached to the "input" and is also passed along to each plugin script. Similarly, params on a plugin sequence are also attached to "input" and passed to the respective plugin script.

This API will first execute ENTER plugins, and when successful will execute POSTPROCESS plugins by executing the executeSave API.

executeSave (args) – BO API

This API first executes the POST Processing plugins and then saves the BO data to offline storage.

Arguments for this API are the same as executeStateChange. Thus, this API can be executed without a state transition as well, wherein no ENTER plugins are executed but data will be saved to offline.

Business Object Helper APIs (ouml.BOHelper)

ouml.BOHelper.getPrototypeFor(boName, owner)

This API returns the prototype that a JavaScript class must extend. This API accepts two arguments:

- **boName** – BO name.

- **owner** – Owner code *must* be provided when writing a customization class for another's BO (BO not owned by this owner). Otherwise do not specify (as shown in the diagram above for a BO hierarchy, all green boxes need not specify the owner).

ouml.BOHelper.getViewModelPrototypeFor(boName, owner)

This API returns the prototype that a UI JavaScript class must extend. This API accepts two arguments:

- **boName** – BO name.

- **owner** – Owner code *must* be provided when writing a customization class for another's BO (BO not owned by this owner). Otherwise do not specify (as shown in the diagram above for a BO hierarchy, all green boxes need not specify the owner).

Offline Data Storage and Query APIs

The following APIs are supported for offline storage and retrieval of the BO data on the device.

ouml.BOHelper.getMOQueryFields(moName)

This is a BO helper API. For a given MO name it returns a list of query fields, defined on this MO, in the same sequence as returned by deployment data. This API will not return the associated data types.

ouml.BOHelper.getBOQueryFields(boName)

This is a BO helper API. For a given BO name it returns a list of query fields, defined on this BO. This API will not return the associated data types.

ouml.BOHelper.getQueryFieldTypes(moName)

This is a BO helper API. For a given MO name it returns a dictionary of key value pairs, where key is the query field name and value is the respective data type, as defined on this MO.

getMOQueryFields() – BO API

This is a BO API. It will use the MO name of the current BO to invoke the corresponding BO helper API, and return the query fields defined on the MO.

getBOQueryFields() – BO API

This is a BO API. It will use the BO name of the current BO to invoke the corresponding BO helper API and return the query fields defined on the BO.

getMOQueryFieldTypes() – BO API

This is a BO API. It will use the MO of the current BO to invoke the corresponding BO helper API and return the query field types as defined on the MO.

getPKs() – BO API

This is a BO API. It will return the PKs defined on the BO.

getDTO – BO API for Offline Data Storage

This API will be invoked by OUML when saving (add/update) a BO to offline storage. Query fields defined at the MO will be used to save the specific BO data fields to generic columns (gen_col1 – gen_col10). Query field at index 1 in "queryFields" on the MO will be saved to gen_col1, and so on. However, if a given query field on the MO level is not present on the BO that is being saved, then the same will be ignored. For instance, if a "boStatus" query field is present on MO "queryFields" but is

missing in BO "queryFields", then it will not be saved to an offline generic column, although it is mapped to gen_col1.

ouml.BOHelper.loadBOData – API to Query Offline BO Data

This is an async BO Helper API. Signature of this API is similar to other async APIs.

This API will accept the input in the format shown in the following code, and return the BO data via a success callback.

```
{
  "onSucess": "<success callback function>",
  "onFailure": "<failure callback function>",
  "transaction": "<existing transaction handle>",
  "input": {
    "moCode": "M1-TASK",
   "fetchRowCount": true,
    "offset": "1",
    "limit": "3",
    "queryFields": [
      {
        "field": "boStatus",
        "value": "COMPLETED"
      },
      {
        "field": "scheduleDetails.workSequence",
        "value": "0"
      }
    ],
    "orderBy": [
      {
        "column": "bo_key",
        "order": "DESC"
      },
      {
        "column": "type"
      }
    ]
  }
}
```

- moCode – MO code is mandatory argument.

- fetchRowCount – default is *false*; when set to *true*, the total number of records will be returned as "totalCount" in the callback response.

- offset – default is 0 when not specified – skip first few records as set by this value.

- limit – default is read from config.js – bOQueryLimit (number of records to be returned).

- queryFields – list of query fields to filter the BOs.

  - field – JSON path of the field.

  - value – string value to be compared against.

  - operator – default is '=' when not specified.

  - oolumn – specify a db column name (e.g. bo_code or type; even gen_col1 will work).
    Note: When a column is defined, field will be ignored and db column name will be used.

- orderBy – list of query fields to sort the data.

  - field – json path of the field.

  - order – default is ASC when not specified.

- oolumn – specify a db column name (use only when to be ordered on non query fields).

BO data will be returned in this format. (a list of BOs):

```
{
  "totalCount": <total number of record, without any limit>
  "output": [
    {
      "bo": "<BO_NAME>",
      "version": "<BO RECORD VERSION>"
      "data": "<BO_JSON_DATA>"
    }
  ]
}
```

BO Versioning

OUML's F1_BIZOBJ table has a version column that stores the current version of BO data. This version starts from 1, and by default is incremented with every edit.

When BO data is read on the device via an OUML API, the current version of the BO record is also returned. Every BO instance (JavaScript) is set to store this version number in it. When the BO API to change the BO state or save the BO changes is invoked, this version number is compared against the current version of the BO record in the offline DB. These APIs can accept an argument to control the default behavior of the API. The following options are supported:

- Default behavior - If versions are mismatched, the update command will fail with the appropriate error. If they are same, the version will be incremented by 1 and data will be saved successfully.

- "OVERRIDE" – When this option is set, even if there is a version mismatch, the new BO changes will be saved and it will overwrite he data in offline with a new copy of the BO data.

- "NOVERSION" – Same behavior as default when version mismatches are present. If the version is the same, data will be saved, but the version will not be incremented.

There are different use cases where one can use these extra options to control the BO save operation. For example, the "OVERRIDE" option can be used when saving user changes from the UI so that any important and time-consuming edits on the UI are given preference over any server side updates. Similarly, "NOVERSION" can be used when saving server-side low-importance updates on a BO; this change will not update the version, so the default behavior of the BO to save APIs will overwrite these server-side low-priority edits when a UI edit is occurring in parallel.

Business Object UI (HTML and JavaScript)

Every BO has a top-level UI page (UI Map/HTML file), which is the main landing page when you use the navigateToBOPage API. This page usually has list of sections, with each section pointing to a <DIV> element in either same HTML file or other HTML fragments (files) included via an overridden API (loadPageFragments). The HTML file name must match the BO name and the top-level DIV (which has data-role=page) should have the BO name as the value for the div's id attribute. For exmaple, for the M1-Assignment BO, file name would be M1-Assignment.html and the top level div in that page would be:

```
<div data-role="page" id="M1-Assignment">
<script src="M1-Assignment.js"></script>
```

For a non-BO Page the div Id should also uniquely identify a page, and if the JS file name, HTML file, and the div id all are same, the Oracle Utilities Mobile Library can load the JS file automatically without any script include.

As shown in the above example, a script is included inside the div element. This is important because, with jQuery mobile, you cannot expect any script outside the page div to be loaded. Thus, if you ever need to load a JS file for any page (BO or non-BO), it should be included inside the page div.

Note: If the JS file name is same as the BO name or the id of the page div (for non-business objects), you need not include this file in the HTML code, since the Oracle Utilities Mobile Library automatically includes a file with this name in the same directory as the page.

Page View Model

Every UI JS must extend the BaseViewModel class if there is no parent to this BO; otherwise, it must extend the parent class (see ouml.BOHelper.getViewModelPrototypeFor(boName, owner) for details).

Code for every UI page (bo/non bo) should be attached to a specific namespace (ouml.ViewModel) as shown in this example BO class:

```
ouml.ViewModel["M1-Assignment"] = (function(ouml) {
function m1Assignment() {
    ouml.ViewModel["M1-Common"].call(this);
  model = this;
};
//set the prototype to parent BO, so we extend the parent's
functions.
    m1Assignment.prototype = Object.create(ouml.ViewModel["M1-
Common"].prototype);

    //point the constructor property to this key (in case we need to
make use of it later)
    m1Assignment.prototype.constructor =  m1Assignment;
     return m1Assignment;

})(ouml);
```

> **Note**: In this example we have a common parent for all business objects, which extends BaseViewModel hence we are extending M1-Common here.

This is the basic minimum code that every business object (nonBO) UI must have.

BaseViewModel API Properties

LABELS	A reference to all labels, can be used in HTML as:
LOOKUPS	A reference to all Lookups. usage: <select class="ui-select" id="keepWithCrew" data-role="none" data-bind="value:stateSpecificFields.keepWithCrew, options: LOOKUPS.M1_SAME_CREW_FLG, optionsText: 'description', optionsValue: 'lookupValue', optionsCaption: 'Select One ...'"></select>
EXTLOOKUPS	A reference to all Extended lookups. Usage: <select id="customerContactType" data-bind="options: EXTLOOKUPS['M1-CustomerContactType'], optionsText: 'description', optionsValue: 'lookupValue', value: completionInfo.customerContactDetails.customerContactType, optionsCaption: ""data-role="none"></select>
pageTitle	This is a knockout observable array, so anytime value is changed it will automatically reflect on UI title. model.pageTitle(model.LABELS.M1_SHIFT_LBL)
pageButtons	This is a knockout observable array, all page buttons are stored here. Please reference the setPageButtons API under Page View Model for more information.

pageMenuItems	This is a knockout observable array containing all menu items.
pageIndicators	This is a knockout observable array containing all page indicators.

API	Parameters	Description
showBackButton	Boolean(true|false)	Sets the visibility of back button on a page. Visible when set to true.
showPanicButton	Boolean(true|false)	Sets the visibility of Panic Alert button on a page. Visible when set to true.
showMapButton	Boolean(true|false)	Sets the visibility of Map button on a page. Visible when set to true.
loadPageFragments		This API when implemented by a page will be invoked before displaying the page content. This API must return an array of HTML file names. The content of each file will be appended to the currently active mobile page's content..Use this API to create a UI from multiple HTML files, and reuse same HTMl file in multiple pages. Each HTML file should contain divs which can be navigated to by showSection API. All the Divs should have visibility of none, otherwise they will appear on UI as soon as a fragment (an html file) is loaded. Please reference the UI Layout and Navigation section for more details.
load	args = {onSuccess:<callback>}	This API when implemented by a page will be invoked just after page HTML/JS files are loaded. Implementer must invoke args.onSuccess() on completion of the work of this method. It is assumed that something asynchronous can happen in this overridden method hence the onSuccess callback is provide to indicate the completion of that work. E.g. loading appropriate data from DB and binding the UI via KO.
setPageButtons		Applicable only for business object pages. This API when implemented by a business object page will be invoked during page load process to allow the page to customize the page specific buttons. Overridden method must set pageButtonList property to an array of buttons, each button object should match this structure: {buttonLabel : <string value>, buttonAction: <click handler function on viewModel >}. Default Oracle Utilities Mobile Library implementation of this method adds next valid states of current BO's state to pageButtonList, and set the state name as handler function name, which means for every state the BO UI JS file (ViewModel) should have a corresponding method.

API	Parameters	Description
~~onInboundMessage~~	{inboundMsg } – inbound message received from server	~~This method will be invoked every time a new message~~ is received via InboundWorker. Message received from server will be passed as an argument (inboundMsg.msgData). (each inbound message processing script decides whether or not to notify the current page)
navigateToBOPage	bo – Name of the BO args = {inputArgs: {key:value}, inputData: {key: value}}	Loads the BO JS files, BO UI page/html and BO UI JS files in that order. Even if your HTML file for a business object has no JS included via script tag it will load the matching file (same name as the BO) in the same folder as the main html file. It also loads the pageFiles specified in the product configuration mobile component for a given pageID (div's id).
navigateToPage	bo – Name of the BO args = {inputArgs: {key:value}, inputData: {key: value}suppressUnsavedAlert: true/false}	Same as navigateToBOPage except that this can be used to load any non business object page UI html files. So no BO JS files are loaded. The user must mention when the unsaved alert has to be suppressed. When a user sets this property into the args input, the alert is skipped.
showSection	args = {id:<id of the div>, title: <string for title>, processAction: {icon:< data-icon attribute to be set>, handler:<method in current viewModel to be invoked>}}	This API doesn't switch the currently loaded page however hides currently active section and displays the requested one. Left side icon will be a back button and right side icon will be set accordingly only if processAction is set. On click of right side icon handler will be invoked
goBack		Use this API to back to either previous section or to a previous page. Whatever was displayed before this UI.
showError	error – an instance of ouml.ClientError sectionId – optional, div ID of a section that should be displayed to show this error on	Displays an inline error message in RED color at the top of either currently displayed section or displays the section with given ID first to show the error
setDefaultSection	sectionId- div id of a section that is displayed by default when page loads	This API must be implemented in order for showSection to work. Default section is the Div that is displayed by default when page is loaded.
showDefaultSection		Show the default section and cleans the page's section history stack. So that using goBack on main page should not go back to previously displayed section of that page but previous page in history.
dialNumber		

API	Parameters	Description
~~getFormattedDate~~	~~date – date to be~~ formatted	~~Converts a base date to device date and formats it to a~~ user specific format. Useful for displaying business object data in the UI.
getFormattedTime	dateTime – datetime to be formatted	Converts a base datetime to device datetime and formats it to a user specific format. Useful for displaying business object data in the UI.
getFormattedDateTime	dateTime – datetime to be formatted	Converts a base datetime to device date time and formats it to a user specific format. Useful for displaying business object data in the UI.
setPageMenuItems		Implement and override this method to add page specific menu items. Oracle Utilities Mobile Library calls it at appropriate time to render the page menus. Please reference addMenuItem under Page View Model for more details.
addMenuItem	menuItem – a menu item object , ouml.MenuItem({index:<index of item>,title: <label>, action:<callback function on viewModel>, active:<true|false> });	setPageMenuItems method if overridden must add individual menu items using this API. A Menu item object should be passed to this method. Please reference the Menu section for more details.

Buttons

The Page Buttons API automatically generates life cycle buttons for a BO User Interface. User can override this default behavior by overriding setPageButtons API of the Base View Model present in Oracle Utilities Mobile Library.

For automatically generation of Life Cycle buttons for a Bo UI, page specific model needs to extends ouml.BaseViewModel.

For generation of buttons for a Non BO UI developer needs to override setPageButtons API and either call addButton function of base view model or directly push button JSON into pagebUttons observable array.

JSON for Buttons

```
{
        buttonLabel: "Button Label",
        buttonAction: function
}
```

Menu API has been added as part of the Base View Model and will be available in child view model at different application layers if child view model extends base view model.

1. **setPageButton**

    Description
    Need to override in page specific view model to add page buttons.

    Specified by
    setPageButton in ouml. BaseViewMode()

Parameters
none

Returns
none

Sample Uses
```
cmModel.prototype.setPageButtons = function(){
    var sample = {

buttonLabel: 'Sample',
buttonAction: this.sampleAction
};

            this.addButton(sample);


    }
```

2. **addButton (json)**

Description
Returns a ouml.MenuItem object. This API will help developer/Cm to write there custom API for menu

Specified by
addButton in ouml. BaseViewMode()

Parameters
json: plain json object

Returns
none

Sample Uses Var
```
button = {
buttonLabel: 'Sample',
buttonAction: this.sampleAction
}
        this.addButton(button);
```

Page Level Buttons

To add page level Buttons for a non BO UI, override setPageButtons() API of ouml.BaseViewModel.

```
cmModel.prototype.setPageButtons = function(){

        var sample = {
buttonLabel: 'Sample',
buttonAction: this.sampleAction
};

            this.addButton(sample);


    }
```

Properties

Properties are used to represent configurable values such as Date formats, Sync Interval, Log File Size. These properties are fetched via a REST call to the server every time in online mode. Some of these properties can be set and retrieved at execution time too.

API - ouml.PropertyEntity

Public APIs

- **getMDTProperty** – Fetch the value against the key/name property.
  This is a synchronous call for all the properties except for "PurgeOnNextLogon". This takes in an input of Property name and an optional callback function which is only used in the case of Property Name = 'PurgeOnNextLogon'.

  Method Signature - function getMDTProperty(key, callbackFunc) {}

- **setMDTProperty –** Set the value against the key/name property.
  This is a synchronous call for all the properties except for "PurgeOnNextLogon". The parameters callbackFunc, errorFunc, transaction are optional for properties that aren't stored in the DB.

  Method Signature – function setMDTProperty(key, value , callbackFunc, errorFunc , transaction) {}

- **removeMDTProperty –** Delete a given property from the Property cache. This is a synchronous call for all the properties except for "PurgeOnNextLogon".

  Method Signature - function removeMDTProperty(key , onSuccess ,onFailure ,transaction) {}

Property Names

- **ASYNC_INTERVAL –** Defines the time interval (in seconds) between Device to Server data sync.

- **ATTACHMENT_STORAGE_SIZE –** Maximum (Sum of all the attachments) attachment storage size possible for the current MDT>

- **BASE_TIMEZONE_OFFSET –** Fetches the base time offset against GMT.

- **CURRENCY_CODE –** Preferred currency code as fetched from the user's 'Display Profile'

- **DATE_DISPLAY_FORMAT –** Preferred date display format as fetched from the user's 'Display Profile'

- **DECIMAL_SEPARATOR -** Decimal separator as fetched from the user's 'Display Profile' **(Un-used right now)**

- **DISPLAY_OPTION –** Display option set for the current MDT's MDT Type. This is not used in this framework as the screens are built responsive to deal with both Mobile and Laptop.

- **GPS_LOG_INTERVAL –** Time interval for capturing the device's current location.

- **GPS_SUPPORTED –** GPS Enabled or Disabled on the MDT Type.

- **GPS_SYNC_INTERVAL –** Logged GPS records will be synced across to the server at this time interval (in mins).

- **INITIAL_SERVICE_SCRIPT –** Not used right now but will have the initial script name to be executed. At the moment, the ouml.Config.getConfig("initScript") property **.**

- **IP_UPDATE_INTERVAL –** IP Address update interval. This is not used in the implementation yet as there is no server to device push communication (Only Device to server pull calls are supported).

- LOG_ARCHIVE_DAYS – Un-used – remove.

- **LOG_FILE_COUNT** - Number of active log files to keep before archival.

- **LOG_FILE_SIZE** - The maximal size in kilobytes of a log file. After the log file reaches this size, it's rolled over into a new file.

- **M1_CAPABILITY –** Stores the JSON format of all the capabilities defined on the MDT Type. Value should first be JSON Parsed before use. For using Capabilities use - ouml.Capabilities

- **MDT_ENCRYPTION_KEY –** Data encryption key is used to encrypt any transactional data on the device. The Key itself is encrypted with the user entered - user name and password.

- **MDT_LOG_LEVEL –** MDT's logging framework uses this Log level to conditionally log only selective log statements.

- **MDT_SESSION_ID –** Counter incremented each time a device is registered. This will be used for BO primary generation to ensure unique keys.

- **MONEY_DECIMAL_DIGITS –** Number of allowed decimal digits for Money fields. (Un-used right now)

- **MONEY_FORMAT –** Money format as fetched from the user's display profile. (Un-used right now)

- **NUMBER_FORMAT –** Number format as fetched from the user's display profile.

- **NUMBER_GROUP_SEPARATOR –** Number group separator symbol.

- **TIME_FORMAT –** Time format as defined in the user's display profile.

UI Layout and Navigation

The Oracle Utilities Mobile Library uses jQuery and Knockout APIs for UI Pages. Each UI page is either a single HTML file or a set of files (page fragments) combined together and displayed as one. Oracle Utilities Mobile Library uses jQuery ajax APIs to load HTML and JS content. Knockout is used to bind the JSON data to UI elements. All layout and navigation specific APIs are part of BaseViewModel class and are made available to a page specific ViewModel when it inherits the BaseViewModel.

HTML Content

Each HTML file that can be navigated by a direct link on the menu, an href in html, or via the navigateToPage API should follow standard jQuery page structure:

```
<div data-role="page" id="M1-BreakTask" >
<div data-role="header">...</div>
<div role="main" class="ui-content">...</div>
<div data-role="footer">...</div>
</div>
```

- The ID of the page div should match to the business object name if it is a BO UI otherwise it should be same as the html filename excluding the file extension.

- Each HTML file cannot have more than one div with data-role=page.

- Oracle Utilities Mobile Library uses a single page template structure of jQuery.

Headers

Header elements are automatically injected from the generic header.html on each page load. This forms the content of the jQuery Mobile page header (data-role="header");

If the SDK detects empty header DIVs with data-role="header" and only injects the header.html content, otherwise individual pages can define their own header html that remains untouched by the SDK.

It's advised to use the system headers on most screens with the following APIs to selectively show/ hide them on specific screens.

Contents

The following buttons that appear on the header (from left to right):

- **Back Button -** Displays a back button
- **Maps Button -** Toggles between the Timeline view and the MapView
- **Panic Alert -** Triggers a Panic Alert from a new UI screen
- **Indicator Bar -** Please reference the Indicators section for more information
- **Menu Bar -** Please reference to the Menu section for more information

Public APIs (via ouml.BaseViewModel)

- **showBackButton** – This is a knockout observable object. The default value is set to True. The value changes when in different UI screens.

  Home Page (Shift Start/Task List) → Invisible

  Task List Page → Shift Page → Visible

  Task List Page (Invisible) → Assignment Main Page (Visible) -> Assignment Details Section -→(Visible)

  Back button is visible when the Stack Size is > 2 (Current page occupies a place too)

  This method should be called by over-riding determinePageHeaderButtons() in your UI's ViewModel class.

  Example – model.showBackButton(false); // Would set it to false.

- **showPanicButton** – Controls the visibility of the Panic button. Default value is true. This is a KO Observable object. This button would be displayed on all the UI screens except for the Login screen.

  Example – model.showPanicButton(false); // hides it.

- s**howMapButton** – Controls the visibility of the Map button. Default value is false. This button is only displayed on the Task List page in the application. Any UI requiring this method will have to toggle it ON in the determinePageHeaderButtons API.

- **determinePageHeaderButtons** – API that can be used to control the visibility of the header buttons. This API gets called even when coming out of the section pages using the back button. This is different than the ouml.ViewModel.load() method that is used for the first time initialization. This API can be optionally over-ridden by the child ViewModel UI screens.

API

APIs are covered in the BaseViewModel API section.

Menu

One popup menu appears on each page. Each contains the following two types of menu items:

- Application Level Menu items
- Page Level Menu item.

For menu item we have one observableArray pageMenuItems in BaseViewModel. By default it will be populated with Application Level menu items once View Model is loaded, and it can be extended or appended from child ViewModel with page specific menu items.

Application Level menu items are generated from configuration file of the application and page specific menu will be implemented by the developer in the page specific models.

Menu Items (ouml.MenuItem)

This is menu item object. Create one object for each menu item.

Constructor

```
ouml.MenuItem({
    // Integer a unique id. It's also determines position of menu
item in menu.
    It's a required filed
    index;

    // Label of menu that will appear for menu item on UI. It's a
required property
    title;

    // Icon if we want an icon for menu item optional
    icon;

    // java script method or a URL optional
    Action;

    // Set active true or false if user you want to show hide the
menu item ->   It's
    required
    Active;
})
```

Menu API

Menu API is part of the Base View Model and will be available in the child view model at different application layers if the child view model extends the base view model.

API	Parameters	Description
ouml. BaseViewModel.getMenuItem	index: an Integer value	Returns a ouml.MenuItem object. This API will help write there custom API for menus. Return an object of ouml.MenuItem or **undefined** **var** menuItem = **this**.getMenuItem(201);
ouml. BaseViewModel.addMenuItem	menuItem : an object of ouml.MenuItms	Add menu item in the menu. This API checks for the menu item with the same index value. If menu Item exits its replace the menu items fields' value with new one otherwise add it to the list.
ouml. BaseViewModel. updateMenuItem	menuItem : an object of ouml.MenuItm	Update menu item values example: Label, action handler and visibility. This API checks for the menu item with the same index value. If menu Item exits its update the menu items fields' value with new one. Print an error log input parameter is not a valid ouml.MenuItem object.
ouml. BaseViewModel. showMenuItem	item : an object of **ouml**.MenuItem or index – index of menu item	Display a hidden menu item dynamically e.g. this.showMenuItem(201);
ouml. BaseViewModel. hideMenuItem	item : an object of **ouml**.MenuItem or index – index of menu item	Hide a visible menu item from menu dynamically. e.g. this.hideMenuItem(201);

Application/SDK-Level Menu Items

To add application-level menu items the Developer/CM need to add ouml.MenuItem object in mainMenu Array List of the product configuration mobile component of the application.

```
var mainMenu = [
                new ouml.MenuItem({
                    index:102, title: "Settings",
            action: "ouml/ui/settings.html", active:true}),

            ];
```

Page-Level Menu Item

To add page-level menu items the Developer/CM need to override setPageMenuItems() API of ouml.BaseViewModel().

```
        cmModel.prototype.setPageMenuItems = function(){

                var sample = new ouml.MenuItem({
                            index:201,
```

```
title: 'Sample Menu Item',
action: this.sample,
active:true
});

          this.addMenuItem(sample);

    }
```

Indicators

An indicator bar is displayed on the screen header as a popup. This bar displays various indicators to present device, server, network, user and activities states.

API (module - ouml.BaseViewModel)

API	Parameters	Description
~~ouml.Indicator.addIndicator~~	~~lookusValue|count:~~ integer valueto represent counter if any	~~Add an indicator in the indicator list.~~ Also sets a counter for this indicator. ouml.Indicator.addIndicator('M1NCCON'); ouml.Indicator.addIndicator('M1NR' , 5);
ouml.Indicator.removeIndicator	lookusValue.	Removes an indicator from the indicator list. ouml.Indicator.removeIndicator('M1NCCON');
ouml.Indicator.addUpdateIndicator	lookusValue count: integer valueto represent counter if any	Description Updates an indicator if it exists, otherwise adds an indicator in the same position. ouml.Indicator.addUpdateIndicator('M1NCCON'); ouml.Indicator.addUpdateIndicator('M1NR' , 5);
ouml.Indicator.setCounter	lookusValue count: integer value to represent counter	Sets count value for an Indicator in Indicator list. Example: ouml.Indicator.setCounter('M1NR' , 5);
ouml.Indicator.getCounter	lookusValue	Returns count value for an Indicator in Indicator list. Example: var mailCount = ouml.Indicator.getCounter('M1NR');

Executing a single Javascript Asynchronous function is easy, however executing multiple asynchronous functions (one after another on success of previous one) requires a bit of extra code to manage the callbacks. The extra code is required to prevent recursive callbacks.

The AsyncWorker module can accept a list of functions to be executed. It returns (invokes your callback function) when the last function in the list is executed successfully or any one function fails. To be able to execute any functions using this AsyncWorker module you must follow the pattern below when executing your asynchronous functions:

Asynchronous Functions Pattern

Any method that can do ASYNC work has to accept arguments as a single obj/args ({key: value,…}) instead of fixed arguments. See Business Object JavaScript (JavaScript) or Business Object Helper APIs (ouml.BOHelper) for examples.

- Transaction, input, onSuccess, and onFailure are required keys on this object and will be same for all ASYNC methods.
  Real input required by business logic of the function will be part of "input" , each function can decide what should be in it.

  - onSuccess will always be called with a single argument obj (again {key: value,…})

    - transaction, output are required keys on this obj.

    - output can contain the real response that caller is expecting in as JSON.

  - onFailure will alwas be called with a single argument ouml.ClientError

  - Each such ASYNC API should have its own set of onSuccess and onFailure implementation to intercept the async response of API called by it.
    So that lower-level APIs async response should be first intercepted by your API and formatted in a format that caller of your API can understand

  - AsyncWorker can be used to execute N number of such methods in sequence.

    - Each method will use transaction returned by (via onSucess callback) previous method

Methods that are guaranteed to be executed SYNChronously need not follow the above approach.

AsyncWorker API

BO Plugins

new Ouml.AsyncWorker		Constructor to create an instance of AsyncWorker
addWork	args = {obj: <a reference of object on which a method will be executed>, method: <method name>, args: <arguments to be passed when executing a method on obj instance>}	Use this API to queue an async function to be executed. Function will not be executed right away, this API just collects the data required to execute a function later.
execute	args = {onSuccess:<Success callback >, onFailure:< failure callback >, transaction: <transaction to be used to execute all functions>}	This API must be called to start the execution of queued async functions. On successful execution of 1st function AsyncWorker will execute 2nd function and so on. On Failure of any function in queue the onFailure callback of this function (execute) will be invoked and error object returned by failing function will be passed as argument. On successful completion of all functions onSuccess callback of this function will be invoked.

Plugins approach of the Oracle Utilities Mobile Library is just a convention that is recommended approach to write client side equivalent of server side Enter and PostProcessing plugin scripts.

Enter and PostProcessing algorithms on a BO are written in JavaScript for execution on client. Each plugin is written as a Javascript Class attached to ouml.plugins namespace.<script-code>. This class should implement a process method.

Refer to the various base MCP Enter Plugin and MCP Post Processing Plugin base mobile components for examples.

Structure of a plugin script:

```
ouml.plugins["M1-MCPTSUpd"] = (function (ouml){

var m1SendTaskUpdate = function () {

};
m1SendTaskUpdate.prototype.constructor =  m1SendTaskUpdate;
m1SendTaskUpdate.prototype.process = function(args) {
 }
 return m1SendTaskUpdate;
})(ouml);
Process method should accept an object as argument with following
atributes:
transaction-  transaction to be used for any DB operations
onSuccess – callback for successful execution of plugin
onFailure – callback for failure in execution, any exception or error
scenario

input – {bo: <BO instance reference>, action: <ADD/REPLACE>}
onFailure callback should be called with ouml.ClientError instance
```

Mobile Device APIs

The following sections describe APIs used to access various objects and modules of the mobile application.

File

The file object is a wrapper for the Apache Cordova File. This object is used for read/write access to files residing on the device. It also has some other helper functions for file access/read/write.

API (module - ouml.File)

API	Parameters	Description
openLocalFile	url	Opens the local file from the device file system in native device viewer. The url passed in as the parameter is the complete path of the file to be opened.
base64ToArrayBuffer	base64String	Creates bytes buffer for a given base64 String
getFilesFromDirectory	Directory name, success callback and failure callback	Fetches list of files for a given directory and its sub directories on the device
createDirectoryStructure	Directory path, success callback and failure callback	Creates the structure for given directory on the device. It creates all the directories passed as part of the directory path.
writeFileData	filename, directory path for the file, success callback, failure callback, original callback, file data, appendEOF flag	Writes data to a given file on the Device. A file is created on the device file system and the data passed is written to the file. If original callback is passed it will supersede the success callback. The file url is passed as a parameter to the callback function. If appendEOF flag is passed then the data is appended to the file if the file exists.
deleteFile	Filepath including file name, success callback and failure callback	Deletes given file on the device. The file path is used to locate the file and delete it
getFileSize	Filepath including file name, success callback and failure callback	Returns the size of a given file on the device in the callback function. The file size is returned in bytes.
readLogsFile	filename, directory path for the file, success callback, failure callback	Reads log data from given file on the device
deleteFolder	directory path with directory name, success callback, failure callback	Deletes given folder on the device
fileFailure	error	Default file failure callback. Used if no failure callback is passed as input to the other API functions

Barcoding

The BaseBarCode object is a wrapper for the Apache Cordova Barcode scanner. This object can be used to get the barcode and barcode type for an item.

API (module - ouml.BaseBarcode)

API	Parameters	Description
scan	Success callback, failure callback	This returns the barcode result which has the barcode type and barcode in the success callback

Barcode Support

The application supports the following barcode types on supported Android, iOS, and Windows* devices:

- QR-Code
- Code 128
- Data Matrix ECC200 (* Android and iOS only)
- ITF-14
- UPC-A
- UPC-E

Note: All 12 numbers were returned in UPC-A testing on Android devices.

UI Theme

The UI Theme defines the color scheme used in the mobile application. Please reference Chapter 5: Customization and Extension Methodology for information on working with UI Themes and the JQuery Mobile Theme Editor.

Logging

The Oracle Utilities Mobile Library Logging module exposes the APIs required by your implementation to facilitate system logging. Any application module that requires logging uses this module with the single log instance maintained for the complete application. Logs get the appropriate instance from **ouml.JSLogger** and use the exposed API.

For example to log an info message your implementation would use:

ouml.JSLogger.info('Your message ');

Extra public APIs exposed by this object (not part of the Oracle Utilities Mobile Library or Parent business object).

- **mdtdebug(message):** The module that needs to log a **framework level debug** message calls this method.
  Passes the log message arguments to the methods.

- **debug(message)**: The module that needs to log a **debug** message calls this method.
  Passes the log message arguments to the methods.

- **info(message)**: The module that needs to log an **info** message calls this method. Passes the log message arguments to the methods.

- **warn(message):** The module that needs to log a **warn** message calls this method. Passes the log message arguments to the methods.

- **error(message)**: The module that needs to log an **error** message calls this method. Passes the log message arguments to the methods.

- **perf(message)**: The module that needs to log a **perf** message calls this method. Passes the log message arguments to the methods

- **fatal(message)**: The module that needs to log a **fatal** message calls this method. Passes the log message arguments to the methods

- **setLevel(level)**: These methods set the logging level of the logger instance that the application has acquired initially. The level that is to be set should be within the set of levels supported by Logger. Else default logging level will be used

- **syncLogFile()** : This method synchronizes the log files to the server.

Error Handling

All error messages that gets displayed to user on UI must be created as Message object on server side and downloaded to client as deployment. Any error situation that occurs on client has to create an instance of ouml.ClientError. Please reference the API description below for more information.

All onFailure/error callbacks should return an instance of ouml.ClientError. The Oracle Utilities Mobile Library includes API to display an error on UI in two forms as described in API description below.

API	Parameters	Description
new ouml.ClientError	args = {msgCat:<message category>, msgId:<message id>,params:<parameters to be set on message>}	Create a new instance by passing the message category, id and parameters
ouml.ViewModel.showError	error – an instance of ouml.ClientError sectionId – optional, div ID of a section that should be displayed to show this error on	Displays an inline error message in RED color at the top of either currently displayed section or displays the section with given ID first to show the error
ouml.Notification.showAlert	ouml.ClientError – an instance of this class	Displays the closeable error on a popup at the top of current UI page.

Chapter 4

Mobile Components

The mobile application consists of various types of components, for example

- Standalone pages, for example Task List

- Business objects

- Reusable components

    - BO Status Enter and Post Processing plugins

    - Common services

    - UI sections

    - A library of common functions

- Inbound messages

- Product configuration settings

- Themes and style configuration

The following sections describe how mobile components are implemented and used.

Mobile Component Portal

Mobile components are implemented using a browser-based editor. Use the Mobile Component portal on the Admin menu to author custom mobile components as well as extend base components.

Mobile components are system-owned records stored in a designated maintenance object.

Component functionality includes:

- Only the component's owner is allowed to change/delete the component.

- Some types of components allow custom extensions. Custom content may be added to base components, similar to adding BO options to base BOs.

- Components support the ability to work with revision control.

You may search for all available components using the mobile component query portal.

Packaged by a Batch Process

The source of mobile components is stored in the database and as such can me migrated from one environment to another using standard tools such as Bundling and CMA.

You need to run the "Build Mobile Component Package" (M1-BMCOM) batch process to generate files based on mobile component records and package them into an application bundle file.

The following illustration shows how mobile components are converted to files and packaged along with the OUML components into the applicable bundle used by the mobile device.
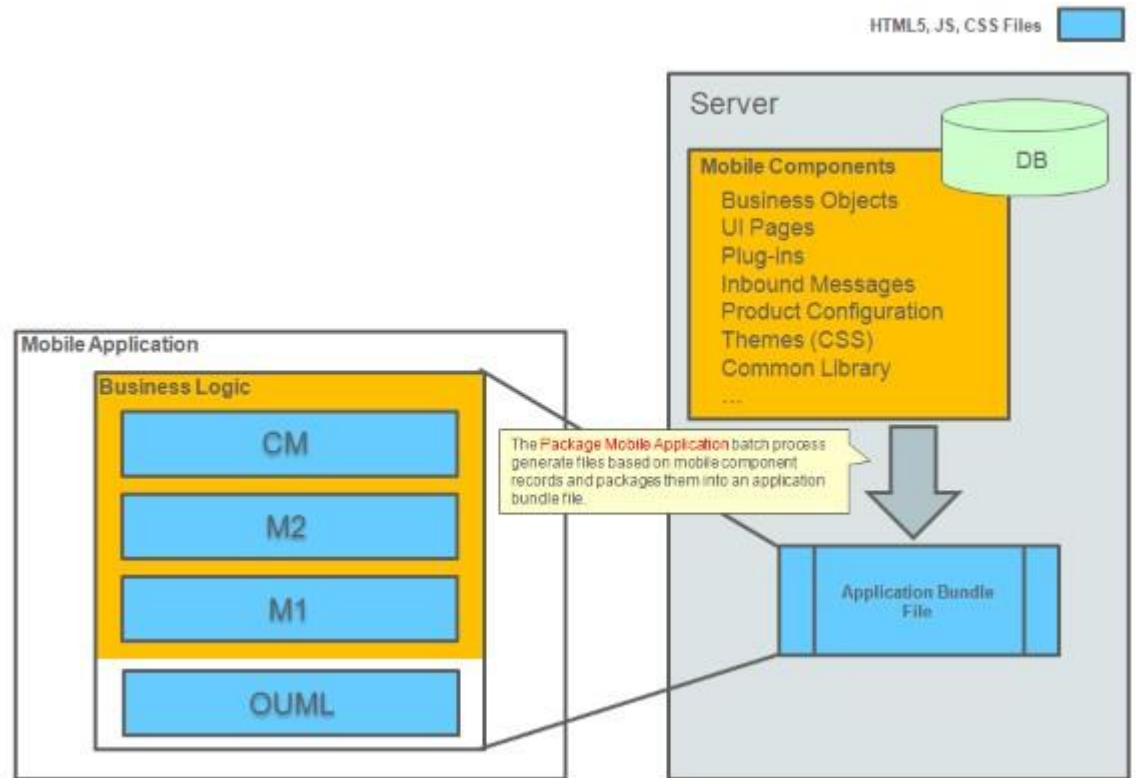

The following table describes how mobile component content is converted to files.

Component Type	Content Type	File	Comment
BO	JS	scripts\bo\<mobile component name>.JS	File name = BO name.
	UI JS	ui\bo\<mobile component name>.JS	File name = BO name
	HTML	ui\bo\<mobile component name>.HTML	File name = BO name
UI Page	UI JS	ui\page\<mobile component name>.JS	
	HTML	ui\page\<mobile component name>.HTML	
UI Section	HTML	ui\library\<mobile component name>.HTML	
MCP Enter	JS	scripts\plugins\BOStatusEnter\plugins.JS	All plugin functions are stored in this file.
MCP Post Processing	JS	scripts\plugins\BOPostProcessing\plugins.JS	All plugin functions are stored in this file.
Inbound Message	JS	scripts\inbound\<mobile component name>.JS	File Name = Message Name

Service	JS	scripts\services\services.JS	All service functions are stored in this file.
Common Library	JS	common\<mobile component name>.JS	
Themes	CSS	ui\themes\<mobile component name>.CSS	
Product Configuration	JSON	Config.JS	The file is config.js regardless of mobile component name, There must be a product configuration record for a product owner if it contributes mobile component content.

Testing

To simplify browser-based testing in a shared application server, content changes made via the portal editor are also automatically synced to the corresponding file in the shared file system. The following illustration describes this process.
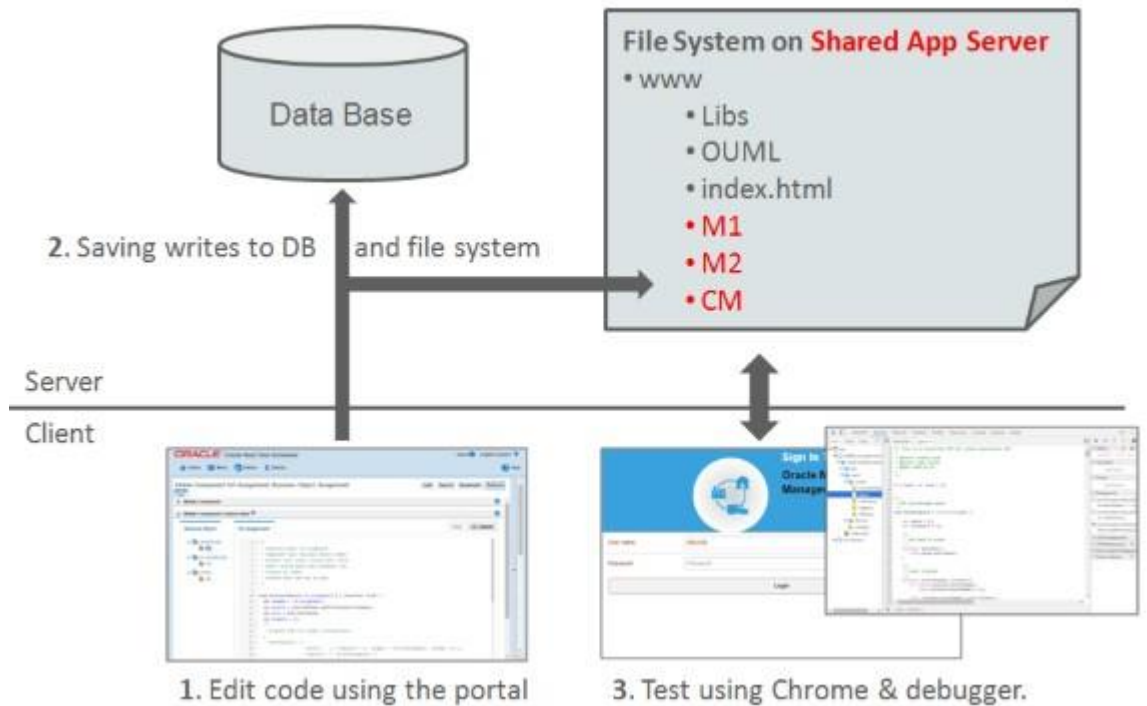
Chapter 5

Customization and Extension Methodology

This chapter provides information on extending the Oracle Utilities Mobile Workforce Management Mobile Application.

Make sure you run the "Build Mobile Component Package (M1-BMCOM)" batch process so that it creates the cm folder under www before you begin customization. You need to run the batch process again when you are done with your customizations to include them in a new mobile application bundle file.

Use the Mobile Component portal to introduce your custom mobile components. Al references in this chapter assume changes are made via this portal only. Refer to Chapter 4 in this document for more information.

CSS and Images

The following sections describe how to change custom themes and images.

Setting Custom Swatch

Oracle's swatch is defined in the www/ouml/themes/theme-ouml.css file.

In your product configuration mobile component, you can set the default swatch as follows: uiTheme: "a"

You can create your own custom swatch using JQuery Mobile Theme Roller (http://themeroller.jquerymobile.com/) as follows:

- Either import the theme-ouml.css file in JQuery Mobile Theme Roller and modify it, or create your own.

- You will need to create a new Theme Mobile Component and then copy/paste the swatch css into the component content.

Overriding Icons

You can override icon properties in Theme Mobile Components as shown in the following examples.

Example 1:

To specify a custom image for the paperclip icon used in Item level Attachment in the Depot Related Assignment screen:

```
.ui-icon-paperclip {
background-image: url("images/example.png");
}
```

Example 2:

To override the logon banner:

```
/* DeskTop version */
#bannerImage {
content:url('images/OracleBanner.png');
}

/* Mobile version */
@media all and (max-width:500px) {

#bannerImage {
content:url('images/OracleBanner_mobile.png');
}
}
```

Changing Map Markers

Map markers are defined as part of a configurable property called **oracleMapProperties** in the product configuration mobile component.

This property can be overridden in your custom the product configuration mobile component to define custom properties for Oracle Map, including images.

Extending Navigation

Application-Level Menu Items

Application level menu items which is shown in all the screens are defined through the **mainMenu** property in the product configuration mobile component. Default menu items include, for example, Settings, Exit, etc.

This property can be extended by adding new application-level menu items to your CM Product Configuration mobile component:

```
var mainMenu = [
            new ouml.MenuItem({"CM_CONTACT_US", index:104, title:
"CM_CONTACT_US_LBL", action: "CM/ui/ContactUs.html", active:true})
            ];
```

Note: For details on creating a CM-ContactUs.html/JS, see the Creating a Custom Page Not Related to a Business Object section.

Application-level menu items can be removed from individual pages by overriding setPageMenuItems in each page JS. Please reference the example in the next section for more information.

Page-Level Menu Items

Menu items for a particular screen are set in the UI JavaScript by overriding the setPageMenuItems function.

In the following example we will hide the "Decline Activity" menu option and introduce a new menu item called "New Function" in the M1-DepotTaskAssignment screen, by extending the UI JavaScript for the M1-DepotTaskAssignment business Object mobile component.

Using the Mobile Component tool, select the M1-DepotTaskAssignment Business Object component. Right click on UI JavaScript and select Add to create CM mobile component. The logic should be as follows:

```
ouml.ViewModel['M1-DepotTaskAssignment'].CM = (function(ouml) {
  var boName = 'M1-DepotTaskAssignment';
  var parent = ouml.BOHelper.getViewModelPrototypeFor(boName,
'CM');
  var model = undefined;

  function cmm1DepotTaskAssignment() {
    parent.call(this);
    model = this;
    model.boData = {};
  };

  cmm1DepotTaskAssignment.prototype =
Object.create(parent.prototype);

  cmm1DepotTaskAssignment.prototype.constructor =
'cmm1DepotTaskAssignment';

  cmm1DepotTaskAssignment.prototype.newFunc = function()
  {
    ouml.Utilities.log("New Function");

    //Custom code for the new function
  }

  cmm1DepotTaskAssignment.prototype.setPageMenuItems = function(){

    parent.prototype.setPageMenuItems.call(this);
    // To hide menu item
    model.hideMenuItem(203);
    // To add new menu item
    var newDTMenuItem = new ouml.MenuItem({index:201,title: 'New
    Function', action:model.newFunc,active:true});
    model.addMenuItem(newDTMenuItem);
  }

  return cmm1DepotTaskAssignment;
})(ouml);
```

Extending Existing Screens and Functions

Hiding Menu Items and Overriding Functionality

Depot-Related Assignments have attachments which are shown at page level on the menu and at item level within the item section. Attachments are shown based on a positive check for Cordova.

In the following example we will add a capability check. The showAttachments functionality is also overridden.

Using the Mobile Component tool, select the M1-DepotTaskAssignment Business Object component, then click on CM UI JavaScript so you can edit the CM UI JavaScript created in the previous step. The logic should be as follows:

```
ouml.ViewModel['M1-DepotTaskAssignment'].CM = (function(ouml) {
  var boName = 'M1-DepotTaskAssignment';
```

```
    var parent = ouml.BOHelper.getViewModelPrototypeFor(boName, 'CM');
    var model = undefined;

    function cmm1DepotTaskAssignment() {
      parent.call(this);
      model = this;
      model.boData = {};
      if(cmm1EnableAttachmentSupport())
        model.showAttachmentIcon(true);
      else
        model.showAttachmentIcon(false);
    };

    cmm1DepotTaskAssignment.prototype = Object.create(parent.prototype);

    cmm1DepotTaskAssignment.prototype.constructor =
'cmm1DepotTaskAssignment';

    cmm1DepotTaskAssignment.prototype.newFunc = function()
    {
      ouml.Utilities.log("New Function");

      //Custom code for the new function
    }

    cmm1DepotTaskAssignment.prototype.setPageMenuItems = function(){

      parent.prototype.setPageMenuItems.call(this);

      // To hide menu item
      model.hideMenuItem(203);

      // To add new menu item
      var newDTMenuItem = new ouml.MenuItem({index:201,title: 'New
      Function', action:model.newFunc,active:true});
      model.addMenuItem(newDTMenuItem);

var input =  {pkValue : ouml.App.getPageContext().inputArgs['taskId']};
var attachmentsMenuItem = new ouml.MenuItem({index:301,title:
model.LABELS.M1_ATTACHMENT,
action:model.showAttachments.bind(this,input), active:true});
      if(enableAttachmentSupport())
        model.addMenuItem(attachmentsMenuItem);
      };
    }

cmm1DepotTaskAssignment.prototype.showAttachments = function (keys) {
      // Custom Attachment logic goes here.
    };

    return cmm1DepotTaskAssignment;
})(ouml);
```

Extending HTML Pages

User Interface components for a particular business object are set in the BO JavaScript by the
userInterface function.

In the following example we will override the userInterface() function to replace the base Activity
Information map fragment with a custom version, by extending the backend BO JavaScript for the M1-
Assignment business Object mobile component.

Using the Mobile Component tool, select the M1-Assignment Business Object component. Right click on JavaScript and select Add to create CM mobile component. The logic should be as follows:

```
ouml.BusinessObject['M1-Assignment'].CM = (function (ouml) {
  var boName = 'M1-Assignment';
  var parent = ouml.BOHelper.getPrototypeFor(boName, 'CM');
  var util = ouml.Utilities;
  var plugins = {
    'POSTPROCESS': {
          'active':   [],
          'inactive': []
                    },
    'states':       {}
  };

  var cmm1Assignment = function(data, version) {
    this.bo = boName;
    parent.call(this, data, version);
  };
  cmm1Assignment.prototype = Object.create(parent.prototype);
  cmm1Assignment.prototype.constructor =  cmm1Assignment;
  cmm1Assignment.prototype.getBOName = function() {
    return boName;
  };
  cmm1Assignment.prototype.getParent = function() {
    return parent;
  };
  cmm1Assignment.prototype.getPlugins = function() {
    return plugins;
  };
  cmm1Assignment.prototype.userInterface = function() {
    var uiConfig = {};
    // Starting in position 0, remove 1 entry and add the custom
entry
    uiConfig.m1AssignmentOnsite.leftSide.splice(0, 1, {
id:'cmActivityInfo' });
    return uiConfig;
  }
  return cmm1Assignment;
})(ouml);
```

Creating Custom Plugins

You can create custom plugins mobile components to validate data or to perform other custom functionality.

In the following example we will create a custom plugin mobile component that will validate that at least one remark type is selected. We will add this plugin to the M1-Assignment so it is performed for all assignments.

1. Using the Mobile Component tool, Add a mobile component. In this example, the component type will be a MCP Enter Plugin. After the component has been added, right click on JavaScript and select Add to create CM mobile component. The logic should be as follows

    ```
    ouml.plugins['CM-RemTypeVal'] = (function (ouml) {

      var cMComplVal = function () {
      };

      cMRemTypeVal.prototype.constructor = cMRemTypeVal;
    ```

```
cMRemTypeVal.prototype.process = function(args) {
 var boData = args.input.bo.data;

    if (boData.completionInformation/remarkTypes/
remarkTypesList.length == 0) {
  return args.onFailure(new ouml.ClientError({
"msgCat" : 90000,
"msgId" : 20008,
"params":[]
  }));
  }
  args.onSuccess(args);
  };

  return cMRemTypeVal;
})(ouml);
```

2. Add the plug-in to the COMPLETED state in the CM M1-Assignment BO JavaScript mobile component. Using the Mobile Component tool, select M1-Assignment business object mobile component. Click on CM JavaScript, so you can edit the CM JavaScript created in a previous step. The logic should be as follows:

```
ouml.BusinessObject['M1-Assignment'].CM = (function (ouml) {
  var boName = 'M1-Assignment';
  var parent = ouml.BOHelper.getPrototypeFor(boName, 'CM');
  var util = ouml.Utilities;
  var plugins = {
    'POSTPROCESS': {
          'active':   [],
          'inactive': []
                    },
        'states': {
"COMPLETED": {
    "ENTER": {
    "active": [
{'sequence': 10, 'plugin': 'CM-RemTypeVal', params: {}}
      ],
    "inactive": []
  }
}
}
    }
  };

  var cmm1Assignment = function(data, version) {
    this.bo = boName;
    parent.call(this, data, version);
  };
  cmm1Assignment.prototype = Object.create(parent.prototype);
  cmm1Assignment.prototype.constructor =  cmm1Assignment;
  cmm1Assignment.prototype.getBOName = function() {
    return boName;
  };
  cmm1Assignment.prototype.getParent = function() {
    return parent;
  };
  cmm1Assignment.prototype.getPlugins = function() {
    return plugins;
  };
  cmm1Assignment.prototype.userInterface = function() {
    var uiConfig = {};
    // Starting in position 0, remove 1 entry and add the custom
entry
```

```
              uiConfig.m1AssignmentOnsite.leftSide.splice(0, 1, {
          id:'cmActivityInfo' });
              return uiConfig;
            }
          return cmm1Assignment;
        })(ouml);
```

Defining a New Initial JS Function

The Initial JS Function is the function executed on the Hybrid MCP after the user has logged on. In the base function, it checks for a current shift and if none, will request the shift from the server.

Some customers might have some other functionality that they want to perform once the user has logged on. They can do this by creating their own custom initial function. Note: The custom initial function MUST call the base initial function in order for the shift check to be performed.

Once created, you need to create a new deployment type and specify your new custom function as the Initial JS Function.

1. Create a mobile component for your custom initial function. Using the Mobile Component tool, Add a new component with a component type of 'Service' and enter appropriate component name (CM-InitialFunction), description, and click Save. After the component has been added, right click on JavaScript and select Add to create CM mobile component. The logic should be similar to the following:

```
ouml.services['CM-InitialFunction'] = (function (ouml) {

  var cMInitialFunction = function () {
  };

  cMInitialFunction.prototype.constructor = cMInitialFunction;

  cMInitialFunction.prototype.process = function(args) {
    // Add custom processing here

    // Call base initial function to check for shift
    new ouml.services["M1-LoadInitialPage"]().process(args);
  };

  return cMInitialFunction;
})(ouml);
```

2. Duplicate the existing deployment type that is being used and change the Initial JS Function to your new Initial JS function just created.

Note: You will need to generate the deployment and download it to your device.

Custom Screens and Functions

Creating a New Custom Child Business Object

In the following example we will create a child assignment BO (which has a new activity details field), for the parent M1-Assignment BO. We will create a new HTML UI section to display the new activity details element.

> **Note:** There are no product configuration changes required when adding a new custom business object.

1. Create a child BO (CM-ApplInspAssignment)on the server side with the required changes and the corresponding changes in the deployment necessary to receive the BO on the client side.

2. Create a mobile component for your HTML UI section to display the new element. Using the Mobile Component tool, Add a new component with a component type of 'UI Section' and enter appropriate component name (cmApplInspDtls), description, and click Save. After the component has been added, right click on HTML and select Add to create CM mobile component. The HTML should be as follows:

```
<div id='cmApplInspDtls' style='display: none' data-inset='true'>
  <div data-role='collapsible' data-bind='ouml_collapse:
isNarrow()' style='width:99%; margin: auto;' data-collapsed-
icon='carat-d' data-expanded-icon='carat-u'>
    <h3>
      <span for="cmApplInspDtls"
            data-bind="text: $root.LABELS.CM_APPLIANCE_DETAILS_LBL"
            id="cmApplInspDtls-lbl">
      </span>
    </h3>
    <ul data-role="listview" data-inset="false" data-filter="false"
data-bind="refresh:true">
      <li data-icon="false">
        <a href="">
          <h2>
            <span data-bind="text: $root.LABELS.CM_APPLIANCE_NAME"
id="cmAIDApplianceName-lbl">
            </span>
          </h2>
          <aside class="ui-li-aside">
            <span data-bind="text:
boData.cmApplianceInspDetails.applianceName"
id="cmAIDApplianceName">
            </span>
          </aside>
        </a>
      </li>
    </ul>
  </div>
</div>
```

3. Create the backend BO JavaScript mobile component for your BO. Using the Mobile Component tool, Add a new component with a component type of 'Business Object' and search for your BO in the Related Business Object field. Enter appropriate description and Save. After the component has been added, right click on JavaScript and select Add to create CM mobile component. We need to change the userInterface() function to display our new UI Section. The logic should be as follows:

```
ouml.BusinessObject['CM-ApplianceInspAssignment'] = (function
(ouml) {
  var boName = 'CM-ApplianceInspAssignment';
  var parent = ouml.BOHelper.getPrototypeFor(boName);
  var util = ouml.Utilities;
  var plugins = {
"POSTPROCESS": {
"active": [
],
```

```
                            "inactive": []
                          },
                          "states": {}
                          };

                            var cMApplianceInspAssignment = function(data, version) {
                              this.bo = boName;
                              parent.call(this, data, version);
                            };

                            cMApplianceInspAssignment.prototype =
                          Object.create(parent.prototype);

                            cMApplianceInspAssignment.prototype.constructor =
                          cMApplianceInspAssignment;

                            cMApplianceInspAssignment.prototype.getBOName = function() {
                              return boName;
                            };

                            cMApplianceInspAssignment.prototype.getParent = function() {
                              return parent;
                            };

                            cMApplianceInspAssignment.prototype.getPlugins = function() {
                              return plugins;
                            };

                            cMApplianceInspAssignment.prototype.userInterface = function() {
                              // Get parent list of section htmls to be used
                              var uiConfig = parent.prototype.userInterface.call (this);
                              // Add sections specific for this BO
                             uiConfig.m1AssignmentOnsite.leftSide.push({ id:'cmApplInspDtls'
                          });

                              return uiConfig;
                            }

                            return cMApplianceInspAssignment;
                          })(ouml);

                          //# sourceURL=cm/scripts/bo/CM-ApplInspAssignment.js
```

4. Create the UI JavaScript mobile component for your BO. Using the Mobile Component tool, select your business object component for your new custom BO. Right click on UI JavaScript and select Add to create CM mobile component. No additional changes are required for this component. The logic should be as follows:

```
ouml.ViewModel['CM-ApplianceInspAssignment'] = (function(ouml) {
  var boName = 'CM-ApplianceInspAssignment';
  var parent = ouml.BOHelper.getViewModelPrototypeFor(boName);
  var model = undefined;

  function cMApplianceInspAssignment() {
    parent.call(this);
    model = this;
    model.boData = {};
  };

  cMApplianceInspAssignment.prototype =
Object.create(parent.prototype);
```

```
    cMApplianceInspAssignment.prototype.constructor =
'cMApplianceInspAssignment';

  return cMApplInspAssignment;
})(ouml);

//# sourceURL=cm/ui/bo/CM-ApplInspAssignment.js
```

Creating a Custom Page Not Related to a Business Object

To create a custom page mobile component not related to a BO and using Oracle Utilities Mobile Library APIs, create your HTML and the corresponding js file as shown in the following examples.

In the following example we will create mobile components for our custom "Contact Us" page. This page is not related to any BO. We will create a new HTML UI page to display our custom contact information.

> **Note**: There are no product configuration changes required when adding a new custom non-BO UI Page.

1. Create a mobile component for your HTML UI page to display our custom contact information. Using the Mobile Component tool, Add a new component with a component type of 'UI Page' and enter appropriate component name (CM-ContactUs), description, and click Save. After the component has been added, right click on HTML and select Add to create CM mobile component. The HTML should be as follows:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <div data-role='page' data-theme='a' id='CM-ContactUs'>
      <script src="CM-ContactUs.js"></script>
      <div data-role='header' data-position='fixed'></div>
      <div data-role='content' id='content'>
        <div id='cMContactUsSections' data-bind='refresh: true'>
          <p>Contact help desk at 1-888-888-HELP</p>
        </div>
      </div>
    </div>
  </body>
</html>
```

2. Create the UI JavaScript mobile component for you non-BO UI Page. Using the Mobile Component tool, select your UI Page component just created. Right click on UI JavaScript and select Add to create CM mobile component. The logic should be as follows:

```
ouml.ViewModel['CM-ContactUs'] = (function(ouml) {
  var model = undefined;

  function cMContactUs() {
    model = this;
    ouml.ViewModel['CM-ContactUs'].call(this);
  };

  cMContactUs.prototype = Object.create(ouml.ViewModel['CM-
ContactUs'].prototype);

  cMContactUs.prototype.constructor = cMContactUs;
```

```
            cMContactUs.prototype.load = function(inputArgs){
              model.pageTitle(model.LABELS.M1_CONTACT_US_LBL);
              this.setDefaultSection("cMContactUsSections");
              inputArgs.onSuccess();
            }

            return cMContactUs;
        })(ouml);
```

Device Plugins

Device plugins can be overridden by implementing custom Service mobile component and mapping them to the appropriate capability in your custom Product Configuration mobile component. Refer to existing base capabilities and their services for an example.

Customizable Indicators

We can add a new indicator and show that inside indicator bar visible at header section of the page. List of indicators shown inside indicator bar is maintained as Extendable Lookup in Oracle Utilities Application Framework. Please follow the steps below to add/hide an indicator in indicator bar.
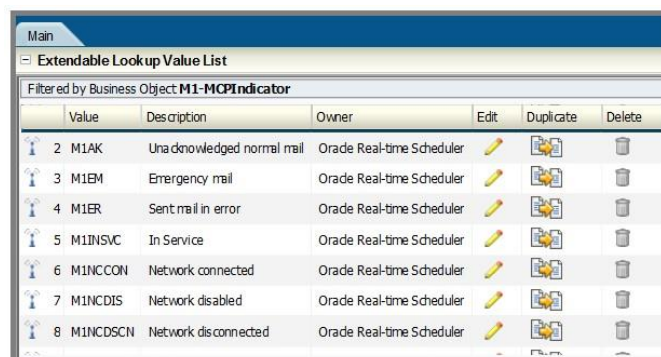
The following examples are for demonstration purpose only and applicable for manipulation of indicators in indicator bar during state transition. These changes will not be persisted. Customization approach may differ based on actual requirement, even though the API for handling indicators in indicator bar will remain the same.

Adding a Custom Indicator

1. Login to Oracle Utilities Application Framework with a CM system user ID and navigate to list of Extendable Lookups following the path:

   ```
   Menu -> Admin Menu -> E -> Extendable Lookup
   ```

2. Search for the business object M1-MCPIndicator.

3. Select the extendable lookup from search result.
   This lookup contains indicator information for those which need to be added to indicator bar.

		Value	Description	Owner	Edit	Duplicate	Delete
	2	M1AK	Unacknowledged normal mail	Oracle Real-time Scheduler			
	3	M1EM	Emergency mail	Oracle Real-time Scheduler			
	4	M1ER	Sent mail in error	Oracle Real-time Scheduler			
	5	M1INSVC	In Service	Oracle Real-time Scheduler			
	6	M1NCCON	Network connected	Oracle Real-time Scheduler			
	7	M1NCDIS	Network disabled	Oracle Real-time Scheduler			
	8	M1NCDSCN	Network disconnected	Oracle Real-time Scheduler			

4. Add new indicators that you want to show in indicator bar.
   For example:

   a. Click the Add link on the Extendable Lookup Value List section.

   b. Add an indicator for showing Crew Onsite.

The indicator code must start with 'CM', which designates these indicators as custom.

c.   Add a second (similar) indicator for showing Crew Enroute.

5.   Add the new indicator icons in the path <base _dir>\m1Mobile\www\cm\images.
The position of the indicator (with respect to the other indicators) depends on the value of: M1-
MCPIndicator -> businessObjectDataArea -> position.

6.   Call the API to add the indicator in the indicator list:

```
ouml.Indicator.addUpdateIndicator(extendable_lookup_value);
//e.g. ouml.Indicator.addUpdateIndicator("CMCRENR");
```

The code for indicator bar customization should be added within the CM layer.

For example, you may want to show the 'Crew Enroute' indicator when crew is working in an
Assignment and is in Enroute state. To implement that, you must first extend the UI
JavaScript content of the M1- Assignment business object mobile component (already done earlier),
followed by overriding the method 'ENROUTE'.

Within the overridden ENROUTE function, invoke the API to add this indicator in indicator
bar.

```
cmm1Assignment.prototype.ENROUTE = function () {
  ouml.Indicator.addUpdateIndicator("CMCRENR");
  parent.prototype.ENROUTE.call(this);
}
```
Now the indicator is added to the indicator bar.
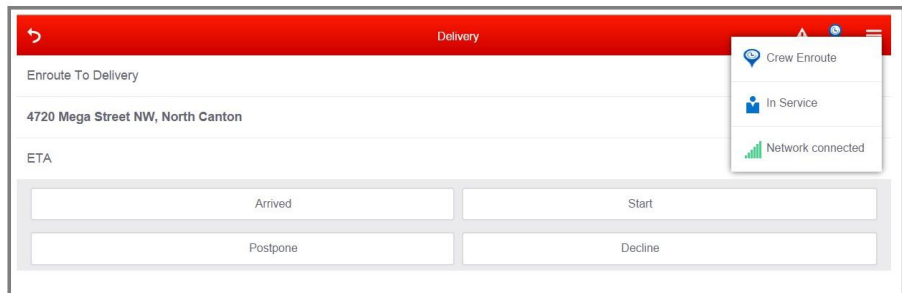
Switching Between Indicators

We can switch between indicators based on specific condition. We need to add the indicators first as
specified in the section above before we can perform switch. Please note that all the indicators should
have the same 'position' value to enable them to switch between themselves.

In this example, we will extend the previous example to switch between the indicators for 'Crew
Onsite' and 'Crew Enroute'.

To show the indicator for 'Crew Onsite', we need to override the ONSITE state transition UI function in the same way we overrode the ENROUTE method in the previous section.

1. Within the overridden ONSITE function, add the API to add indicator for 'Crew Onsite'.

```
cmm1Assignment.prototype.ONSITE = function () {
        ouml.Indicator.addUpdateIndicator("CMCRONS");
    parent.prototype.ONSITE.call(this);
    }
```

2. Since both 'Crew Enroute' and 'Crew Onsite' share same value in 'position', they will replace each other based on the state of the assignment.

Removing an Indicator

You also have the option to remove an indicator. For example, you can hide the 'Network connected / disconnected' indicator.

*   To remove an indicator, call the API to remove an indicator from indicator bar:

```
 ouml.Indicator.removeIndicator("extendable_lookup_value ")
```

*   To remove the network indicator while the crew is enroute, override the ENROUTE state transition method as follows:

```
cmm1Assignment.prototype.ENROUTE = function () {
        ouml.Indicator.addUpdateIndicator("CMCRENR");
         ouml.Indicator.removeIndicator("M1NCCON");
         ouml.Indicator.removeIndicator("M1NCDSCN");
         parent.prototype.ENROUTE.call(this);
        }
```

*   Indicators can also be removed from indicator bar by changing the 'Usage Flag' of this extendable lookup to 'Inactive'.

	Mobile Device Indicator Bar Maintenance
Indicator Code	CMCRENR
Description	Crew Enroute
Usage Flag	Inactive (F1IN)
Indicator Image Name	crewEnroute.png
Indicator Position	210
	Save Cancel