Mobile Device Management Protocol Reference

2011-11-02

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Tables and Listings

About Mobile Device Management

The Mobile Device Management (MDM) protocol provides a way for system administrators to send device management commands to managed iPhone, iPad, and iPod Touch devices running iOS 4 and later. Through the MDM service, an IT administrator can inspect, install, or remove profiles; remove passcodes; and begin secure erase on a managed device.

The MDM protocol is built on top of HTTP, transport layer security (TLS), and push notifications. The related MDM check-in protocol provides a way to delegate the initial registration process to a separate server.

MDM uses the Apple Push Notification Service (APNS) to deliver a "wake up" message to a managed device. The device then connects to a predetermined web service to retrieve commands and return results.

To provide MDM service, your IT department needs to deploy an HTTPS server to act as an MDM server, then distribute profiles containing the MDM payload to your managed devices.

A managed device uses an identity to authenticate itself to the MDM server over TLS (SSL). This identity can be included in the profile as a Certificate payload, or can be generated by enrolling the device with SCEP.

The MDM payload can be placed within a configuration profile (`.mobileconfig`) file distributed using email or web page, as part of the final configuration profile delivered by an Over-The-Air Enrollment service, or installed with iPCU. Only one MDM payload can be installed on a device at any given time.

Configuration Profiles and Provisioning Profiles installed through the MDM service are called Managed Profiles. These profiles will be automatically removed when the MDM payload is removed.

Although an MDM service may have the rights to inspect the device for the complete list of profiles or provisioning profiles, it may only remove profiles and provisioning profiles that it originally installed.

A user may remove the profile containing the MDM payload at any time. Similarly, the MDM server can always remove its own profile, regardless of its access rights. The device does not contact the MDM server when the user removes the payload. Please see "MDM Best Practices" (page 39) for recommendations on how to detect devices that are no longer managed.

A profile containing an MDM payload may not be locked. However, Managed Profiles installed through MDM may be locked. All Managed Profiles installed through MDM are removed when the main MDM profile is removed, even if they are locked.

At a Glance

This document was written for system administrators and system integrators designing software for managing mobile devices in enterprise environments.

This document is divided into three parts:

"Mobile Device Management (MDM) Protocol" (page 9)—describes the main MDM protocol.
"MDM Check-in Protocol" (page 31)—describes the MDM check-in protocol.

"MDM Best Practices" (page 39)—describes how to design payloads for maximum effectiveness.

Mobile Device Management (MDM) Protocol

The Mobile Device Management (MDM) protocol provides a way to tell a device to execute certain management commands remotely. The way it works is straightforward.

During installation:

- The user or SCEP administrator tells the device to install an MDM payload. The structure of this payload is described in "Structure of MDM Payloads" (page 10).
- The device connects to the check-in server. The device presents its identity certificate for authentication, along with its UDID and push notification topic.

  If the server accepts the device, the device provides its push notification device token to the server. The server should use this token to send push messages to the device. This check-in message also contains a `PushMagic` string. The server must remember this string and include it in any push messages it sends to the device.

During normal operation:

- The server (at some point in the future) sends out a push notification to the device.
- The device polls the server for a command in response to the push notification.
- The device performs the command.
- The device contacts the server to report the result of the last command and to request the next command.

From time to time, the device token may change. When a change is detected, the device automatically checks in with the MDM server to report its new device token.

Note: The device polls only in response to a push notification; it does not poll the server immediately after installation. The server must send a push notification to the device to begin a transaction.

The device initiates communication with the MDM Server in response to a push notification by establishing a TLS connection to the MDM Server URL. The device validates the server's certificate, then uses the identity specified in its MDM payload as the client authentication certificate for the connection.

Note: MDM follows HTTP `3xx` redirections without user interaction. However, it does not remember the URL given by HTTP `301 (Moved Permanently)` redirections. Each transaction begins at the URL specified in the MDM payload.

Structure of MDM Payloads

The Mobile Device Management (MDM) payload is designated by the `com.apple.mdm` value in the `PayloadType` field. This payload defines the following keys specific to MDM payloads:

`IdentityCertificate-UUID`	String	*Mandatory.* UUID of the certificate payload for the device's identity. It may also point to a SCEP payload.
`Topic`	String	*Mandatory.* The topic that MDM listens to for push notifications. The certificate that the server uses to send push notifications must have the same topic in its subject. The topic must begin with the `com.apple.mgmt.` prefix.
`ServerURL`	String	*Mandatory.* The URL that the device contacts to retrieve device management instructions. Must begin with the `https://` URL scheme, and may contain a port number (`:1234`, for example).
`SignMessage`	Bool	*Optional.* If `true`, each message coming from the device carries the additional `Mdm-Signature` HTTP header. Defaults to `false`. See "Passing the Client Identity Through Proxies" (page 42) for details.
`CheckInURL`	String	*Optional.* The URL that the device should use to check in during installation. Must begin with the `https://` URL scheme and may contain a port number (`:1234`, for example). If this URL is not given, the `ServerURL` is used for both purposes.
`CheckOutWhenRemoved`	Bool	*Optional.* If `true`, the device attempts to send a `CheckOut` message to the check-in server when the profile is removed. Defaults to `false`. **Availability:** Available in iOS 5.0 and later.

`AccessRights`	Integer, flags	*Required.* Logical OR of the following bit-flags: ● 1 - Allow inspection of installed configuration profiles. ● 2 - Allow installation and removal of configuration profiles ● 4 - Allow device lock and passcode removal ● 8 - Allow device erase ● 16 - Allow query of Device Information (device capacity, serial number) ● 32 - Allow query of Network Information (phone/SIM numbers, MAC addresses) ● 64 - Allow inspection of installed provisioning profiles ● 128 - Allow installation and removal of provisioning profiles ● 256 - Allow inspection of installed applications ● 512 - Allow restriction-related queries ● 1024 - Allow security-related queries ● 2048 - Apply settings. **Availability:** Available in iOS 5.0 and later. ● 4098 - App management. **Availability:** Available in iOS 5.0 and later. May not be zero. If 2 is specified, then 1 must also be specified. If 128 is specified, then 64 must also be specified.
`UseDevelopmentAPNS`	Bool	*Optional.* If `true`, the device uses the development APNS servers. Otherwise, the device uses the production servers. Defaults to `false`.

In addition, four standard payload keys must be defined:

Key	Value
`PayloadType`	`com.apple.mdm`
`PayloadVersion`	`1`
`PayloadIdentifier`	A value must be provided.
`PayloadUUID`	A globally unique value must be provided.

These keys are documented in “Payload Dictionary Keys Common to All Payloads” in *iOS Configuration Profile Reference*.

For the general structure of the payload and an example, see “Configuration Profile Key Reference” in *iOS Configuration Profile Reference*.

Structure of MDM Messages

Once the MDM payload is installed, the device listens for a push notification. The "topic" that MDM listens to corresponds to the contents of the "User ID" parameter in the Subject field of the push notification client certificate.

To cause the device to poll the MDM server for commands, the MDM server sends a notification through the APNS gateway to the device. The message sent with the push notification must contain the `PushMagic` string as the value of the `mdm` key. For example:

```
{"mdm":"<PushMagic>"}
```

Where `PushMagic` is the `PushMagic` string that the device gives to the server during check-in. That should be the whole message. There should not be an `aps` key. (The `aps` key is used only for third-party app push notifications.)

The device responds to this push notification by contacting the MDM server using HTTP PUT over TLS (SSL). This message may contain an `Idle` status, or may contain the result of a previous operation. If the connection is severed while the device is performing a task, the device will try to report its result again once networking is restored.

Listing 1 shows an example of an MDM request payload.

Listing 1 MDM request payload example

```
PUT /your/url HTTP/1.1
Host: www.yourhostname.com
Content-Length: 1234
Content-Type: application/x-apple-aspen-mdm; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>UDID</key>
    <string>...</string>
    <key>CommandUUID</key>
    <string>9F09D114-BCFD-42AD-A974-371AA7D6256E</string>
    <key>Status</key>
    <string>Acknowledged</string>
  </dict>
</plist>
```

The server responds by sending the next command that the device should perform by enclosing it in the HTTP reply.

Listing 2 shows an example of the server's response payload.

Listing 2 MDM response payload example

```
HTTP/1.1 200 OK
Content-Length: 1234
Content-Type: application/xml; charset=UTF-8
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>CommandUUID</key>
    <string>9F09D114-BCFD-42AD-A974-371AA7D6256E</string>
    <key>Command</key>
    <dict>
      ...
    </dict>
  </dict>
</plist>
```

The device performs the command, and sends its reply in another HTTP PUT over the same connection. The MDM server can then reply with the next command, or end the connection by sending a `200 (OK)` status with an empty response body.

If the connection is broken while the device is performing a command, the device will cache the result of the command and re-attempt connection to the server until the status is delivered.

It is safe to send several push notifications to the device. APNS coalesces multiple notifications and delivers only the last one to the device.

You can monitor the MDM activity in the device console using iPhone Configuration Utility. A healthy (but empty) push activity should look like this:

```
Wed Sep 29 02:09:05 unknown mdmd[1810] <Warning>: MDM|mdmd starting...
Wed Sep 29 02:09:06 unknown mdmd[1810] <Warning>: MDM|Network reachability has
 changed.
Wed Sep 29 02:09:06 unknown mdmd[1810] <Warning>: MDM|Polling MDM server
https://10.0.1.4:2001/mdm for commands
Wed Sep 29 02:09:06 unknown mdmd[1810] <Warning>: MDM|Transaction completed.
Status: 200
Wed Sep 29 02:09:06 unknown mdmd[1810] <Warning>: MDM|Server has no commands
for this device.
Wed Sep 29 02:09:08 unknown mdmd[1810] <Warning>: MDM|mdmd stopping...
```

MDM Command Payloads

A host may send a command to the device by sending a plist-encoded dictionary that contains the following keys:

Key	Type	Content
`CommandUUID`	String	UUID of the command
`Command`	Dictionary	The command dictionary.

The content of the `Command` dictionary must include the following key, as well as other keys defined by each command.

Key	Type	Content
RequestType	String	Request type. See each command's description.

MDM Result Payloads

The device replies to the host by sending a plist-encoded dictionary containing the following keys, as well as other keys returned by each command.

Key	Type	Content
Status	String	Status. Legal values are described in Table 1 (page 14).
UDID	String	UDID of the device.
CommandUUID	String	UUID of the command that this response is for (if any)
ErrorChain	Array	*Optional.* Array of dictionaries representing the chain of errors that occurred. The content of these dictionaries is described in Table 2 (page 15).

The Status key will contain one of the following strings:

Table 1 MDM status codes

Status value	Description
Acknowledged	Everything went well.
Error	An error has occurred. See the ErrorChain for details.
CommandFormatError	A protocol error has occurred. The command may be malformed.
Idle	The device is idle (there is no status)
NotNow	The device received the command, but cannot perform it at this time. The device will poll the server again in the future.

Error Handling

There are certain times when the device is not able to do what the server requests. For example, databases cannot be modified while the device is locked with Data Protection. When a device cannot perform a command due to situations like this, it will send the NotNow status without performing the command. The server may send another command immediately after receiving this status, but chances are the following command will be refused as well.

After sending a `NotNow` status, the device will poll the server at some future time. The device will continue to poll the server until a successful transaction is completed.

The device does not cache the command that was refused. If the server wants the device to retry the command, it must send the same command again later, when the device polls the server.

The server does not need to send another push notification in response to this status. However, the server may send another push notification to the device to have it poll the server immediately.

The following commands are guaranteed to execute, and never return `NotNow`:

- `DeviceInformation`
- `ProfileList`
- `DeviceLock`
- `EraseDevice`
- `ClearPasscode`
- `CertificateList`
- `ProvisioningProfileList`
- `InstalledApplicationList`
- `Restrictions`

The `ErrorChain` key contains an array. The first item is the top-level error. Subsequent items in the array are the underlying errors that led up to that top-level error.

Each entry in the `ErrorChain` array contains the following dictionary:

Table 2 ErrorChain array dictionary keys

Key	Type	Content
`LocalizedDescription`	String	Description of the error in the device's localized language
`USEnglishDescription`	String	*Optional.* Description of the error in US English
`ErrorDomain`	String	The error domain
`ErrorCode`	Number	The error code

The `ErrorDomain` and `ErrorCode` keys contain internal codes used by Apple that may be useful for diagnostics. Your host should not rely on these values, as they may change between software releases.

Request Types

ProfileList Commands Return a List of Installed Profiles

To send a `ProfileList` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`ProfileList`

The device replies with a property list that contains the following key:

Key	Type	Content
`ProfileList`	Array	Array of dictionaries. Each entry describes an installed profile.

Each entry in the `ProfileList` array contains a dictionary with a profile. For more information about profiles, see *iOS Configuration Profile Reference*.

Security Note: `ProfileList` queries are available only if the MDM host has an Inspect Profile Manifest access right.

InstallProfile Commands Install a Configuration Profile

The profile to install may be encrypted using any installed device identity certificate. The profile may also be signed.

To send an `InstallProfile` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`InstallProfile`
`Payload`	Data	The profile to install. May be signed and/or encrypted for any identity installed on the device.

Security Note: This query is available only if the MDM host has a Profile Installation and Removal access right.

RemoveProfile Commands Remove a Profile From the Device

By sending the `RemoveProfile` command, the server can ask the device to remove any profile originally installed through MDM.

To send a `RemoveProfile` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`RemoveProfile`
`Identifier`	String	The `PayloadIdentifier` value for the profile to remove.

Security Note: This query is available only if the MDM host has a Profile Installation and Removal access right.

ProvisioningProfileList Commands Get a List of Installed Provisioning Profiles

To send a `ProvisioningProfileList` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`ProvisioningProfileList`

The device replies with:

Key	Type	Content
`ProvisioningProfileList`	Array	Array of dictionaries. Each entry describes one provisioning profile.

Each entry in the `ProvisioningProfileList` array contains the following dictionary:

Key	Type	Content
`Name`	String	The display name of the profile.
`UUID`	String	The UUID of the profile.
`ExpiryDate`	Date	The expiry date of the profile.

Security Note: This query is available only if the MDM host has an Inspect Provisioning Profiles access right.

InstallProvisioningProfile Commands Install Provisioning Profiles

To send an `InstallProvisioningProfile` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`InstallProvisioningProfile`
`ProvisioningProfile`	Data	The provisioning profile to install.

Note: No error occurs if the specified provisioning profile is already installed.

Security Note: This query is available only if the MDM host has a Provisioning Profile Installation and Removal access right.

RemoveProvisioningProfile Commands Remove Installed Provisioning Profiles

To send a `RemoveProvisioningProfile` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`RemoveProvisioningProfile`
`UUID`	String	The UUID of the provisioning profile to remove.

Security Note: This query is available only if the MDM host has a Provisioning Profile Installation and Removal access right.

CertificateList Commands Get a List of Installed Certificates

To send a `CertificateList` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`CertificateList`

The device replies with:

Key	Type	Content
`CertificateList`	Array	Array of certificate dictionaries. The dictionary format is described in Table 3 (page 19).

Each entry in the `CertificateList` array is a dictionary containing the following fields:

Table 3 Certificate dictionary keys

Key	Type	Content
`CommonName`	String	Common name of the certificate
`IsIdentity`	Boolean	Set to `true` if the this is an identity certificate
`Data`	Data	The certificate in DER-encoded X.509 format.

Note: The `CertificateList` command requires that the server have the Inspect Profile Manifest privilege.

InstalledApplicationList Commands Get a List of Third-Party Applications

To send an `InstalledApplicationList` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`InstalledApplicationList`

The device replies with:

Key	Type	Content
`InstalledApplicationList`	Array	Array of installed applications. Each entry is a dictionary as described in Table 4 (page 19).

Each entry in the `InstalledApplicationList` is a dictionary containing the following keys:

Table 4 InstalledApplicationList dictionary keys

Key	Type	Content
`Identifier`	String	The application's ID.
`Version`	String	The application's version.
`ShortVersion`	String	The application's short version. **Availability:** Available in iOS 5.0 and later.
`Name`	String	The application's name.
`BundleSize`	Integer	The app's static bundle size, in bytes.
`DynamicSize`	Integer	The size of the app's document, library, and other folders, in bytes.

DeviceInformation Commands Get Information About the Device

To send a `DeviceInformation` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`DeviceInformation`
`Queries`	Array	Array of strings. Each string is a value from Table 5 (page 20), Table 6 (page 20), or Table 7 (page 21).

The device replies with:

Key	Type	Content
`QueryResponses`	Dictionary	Contains a series of key-value pairs. Each key is a query string from Table 5 (page 20), Table 6 (page 20), or Table 7 (page 21). The associated value is the response for that query.

Queries for which the device has no response or that are not permitted by the MDM host's access rights are dropped from the response dictionary.

General Queries Are Always Available.

The queries described in Table 5 are available without any special access rights:

Table 5 General queries

Query	Reply Type	Comment
`UDID`	String	The unique device identifier (UDID) of the iOS device.

Device Information Queries Provide Information About the Device

The queries in Table 6 are available if the MDM host has a Device Information access right:

Table 6 Device information queries

Query	Reply Type	Comment
`DeviceName`	String	The name given to the device via iTunes.
`OSVersion`	String	The version of iOS the device is running.
`BuildVersion`	String	The iOS build number (8A260b, for example).
`ModelName`	String	Name of the device model, e.g. "iPod Touch".
`Model`	String	The device's model number (`MC319LL`, for example).

Query	Reply Type	Comment
`ProductName`	String	The model code for the device (iPhone3,1, for example).
`SerialNumber`	String	The device's serial number.
`DeviceCapacity`	Number	Floating-point gibibytes (base-1024 gigabytes).
`AvailableDeviceCapacity`	Number	Floating-point gibibytes (base-1024 gigabytes).
`BatteryLevel`	Number	Floating-point percentage expressed as a value between 0.0 and 1.0, or -1.0 if battery level cannot be determined. **Availability:** Available in iOS 5.0 and later.
`CellularTechnology`	Number	Returns the type of cellular technology. `0`—none `1`—GSM `2`—CDMA **Availability:** Available in iOS 4.2.6 and later.
`IMEI`	String	The device's IMEI number. Ignored if the device does not support GSM.
`MEID`	String	The device's MEID number. Ignored if the device does not support CDMA.
`ModemFirmwareVersion`	String	The baseband firmware version.

Network Information Queries Provide Hardware Addresses, Phone Number, and SIM Card and Cellular Network Info

The queries in Table 7 are available if the MDM host has a Network Information access right.

Note: Not all devices understand all queries. For example, queries specific to GSM (IMEI, SIM card queries, and so on) are ignored if the device is not GSM-capable.

Table 7 Network information queries

Query	Reply Type	Comment
`ICCID`	String	The ICC identifier for the installed SIM card.
`BluetoothMAC`	String	Bluetooth MAC address.
`WiFiMAC`	String	Wi-Fi MAC address.
`CurrentCarrierNetwork`	String	Name of the current carrier network.
`SIMCarrierNetwork`	String	Name of the home carrier network. (Note: this query *is* supported on CDMA in spite of its name.)

Query	Reply Type	Comment
SubscriberCarrier-Network	String	Name of the home carrier network. (Replaces SIMCarrierNetwork.) **Availability:** Available in iOS 5.0 and later.
CarrierSettingsVersion	String	Version of the currently-installed carrier settings file.
PhoneNumber	String	Raw phone number without punctuation, including country code.
VoiceRoamingEnabled	Bool	The current setting of the Voice Roaming setting. This is only available on certain carriers. **Availability:** iOS 5.0 and later.
DataRoamingEnabled	Bool	The current setting of the Data Roaming setting.
isRoaming	Bool	Returns whether the device is currently roaming. **Availability:** Available in iOS 4.2 and later. See note below.
SubscriberMCC	String	Home Mobile Country Code (numeric string). **Availability:** Available in iOS 4.2.6 and later.
SubscriberMNC	String	Home Mobile Network Code (numeric string). **Availability:** Available in iOS 4.2.6 and later.
SIMMCC	String	*Deprecated.* Use SubscriberMCC instead. Home Mobile Country Code (numeric string).
SIMMNC	String	*Deprecated.* Use SubscriberMNC instead. Home Mobile Network Code (numeric string).
CurrentMCC	String	Current Mobile Country Code (numeric string).
CurrentMNC	String	Current Mobile Network Code (numeric string).

Note: For older versions of iOS, if the SIMMCC/SMMNC combination does not match the CurrentMCC/CurrentMNC values, the device is probably roaming.

SecurityInfo Commands Request Security-Related Information

To send a SecurityInfo command, the server sends a dictionary containing the following keys:

Key	Type	Content
RequestType	String	SecurityInfo

Response:

Key	Type	Content
SecurityInfo	Dictionary	Response dictionary.

The SecurityInfo dictionary contains the following keys and values:

Key	Type	Content
HardwareEncryptionCaps	Integer	Bitfield. Describes the underlying hardware encryption capabilities of the device. Values are described in Table 8 (page 23).
PasscodePresent	Bool	Set to true if the device is protected by a passcode.
PasscodeCompliant	Bool	Set to true if the user's passcode is compliant with all requirements on the device, including Exchange and other accounts.
PasscodeCompliant-WithProfiles	Bool	Set to true if the user's passcode is compliant with requirements from profiles.

Hardware encryption capabilities are described using the logical OR of the values in Table 8. Bits set to 1 (one) indicate that the corresponding feature is present, enabled, or in effect.

Table 8 HardwareEncryptionCaps bitfield values

Value	Feature
1	Block-level encryption.
2	File-level encryption.

For a device to be protected with Data Protection, HardwareEncryptionCaps must be 3, and PasscodePresent must be true.

Security Note: Security queries are available only if the MDM host has a Security Query access right.

DeviceLock Command Locks the Device Immediately

To send a DeviceLock command, the server sends a dictionary containing the following keys:

Key	Type	Content
RequestType	String	DeviceLock

Security Note: This command requires both Device Lock and Passcode Removal access rights.

ClearPasscode Commands Clear the Passcode for a Device

To send a `ClearPasscode` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`ClearPasscode`
`UnlockToken`	Data	*Optional.* If the device has given an `UnlockToken` value in the "TokenUpdate" (page 32) check-in message, the server must pass the data blob back to the device for this command to work.

Security Note: This command requires both Device Lock and Passcode Removal access rights.

EraseDevice Commands Remotely Erase a Device

Upon receiving this command, the device immediately erases itself. No warning is given to the user. This command is performed immediately even if the device is locked.

Key	Type	Content
`RequestType`	String	`EraseDevice`

The device attempts to send a response to the server, but unlike other commands, the response cannot be resent if initial transmission fails. Even if the acknowledgement did not make it to the server (due to network conditions), the device will still be erased.

Security Note: This command requires a Device Erase access right.

Restrictions Commands Get a List of Installed Restrictions

This command allows the server to determine what restrictions are being enforced by each profile on the device, and the resulting set of restrictions from the combination of profiles.

Key	Type	Content
`RequestType`	String	`Restrictions`
`ProfileRestrictions`	Bool	*Optional.* If `true`, the device will report restrictions enforced by each profile.

The device responds with:

Key	Type	Content
`GlobalRestrictions`	Dictionary	A dictionary containing the global restrictions currently in effect.
`ProfileRestrictions`	Dictionary	An dictionary of dictionaries, containing the restrictions enforced by each profile. Only included if `ProfileRestrictions` is set to `true` in the command. The keys are the identifiers of the profiles.

The `GlobalRestrictions` dictionary and each entry in the `ProfileRestrictionList` dictionary contains the following keys:

Key	Type	Content
`restrictedBool`	Dictionary	A dictionary of boolean restrictions.
`restrictedValue`	Dictionary	A dictionary of numeric restrictions.

The `restrictedBool` and `restrictedValue` dictionaries have the following keys:

Key	Type	Content
restriction name	Dictionary	Restriction parameters.

The restriction names (keys) in the dictionary correspond to the keys in the Restriction and Passcode Policy payloads. For more information, see *iOS Configuration Profile Reference*.

Each entry in the dictionary contains the following keys:

Key	Type	Content
`value`	Bool or Integer	The value of the restriction.

Security Note: This command requires a Restrictions Query access right.

Per-profile restrictions queries require an Inspect Configuration Profiles access right.

Managed Applications

In iOS 5, an MDM server can manage third party applications from the App Store, as well as custom in-house enterprise applications. The server can specify whether the app (and its data) are removed from the device when the MDM profile is removed. Additionally, the server can prevent managed app data from being backed up to iTunes and iCloud.

To install a managed app, the MDM server sends an installation command to the user's device. The managed apps then require a user's acceptance before they are installed.

When a server requests the installation of a managed app from the App Store, the app "belongs" to the iTunes account that is used at the time the app is installed. Paid apps require the server to send in a Volume Purchasing Program (VPP) redemption code that purchases the app for the end user. For more information on VPP, go to http://www.apple.com/business/vpp/.

Apps from the App Store cannot be installed on a user's device if the App Store has been disabled.

InstallApplication Commands Install a Third-Party Application

To send an `InstallApplication` command, the server sends a request containing the following keys:

Key	Type	Content
`RequestType`	String	`InstallApplication`
`iTunesStoreID`	Number	The application's iTunes Store ID. For example, the numeric ID for Keynote is `361285480` as found in the App Store link http://itunes.apple.com/us/app/keynote/id361285480?mt=8.
`ManifestURL`	String	The URL where the manifest of an enterprise application can be found
`ManagementFlags`	Integer	The bitwise OR of the following flags: 1 - Remove app when MDM profile is removed. 4 - Prevent backup of the app data.

The request must contain either an `iTunesStoreID` or `ManifestURL` value, but not both.

If the request is accepted by the user, the device responds with an Acknowledged response, and the following fields:

Key	Type	Content
`Identifier`	String	The app's identifier (Bundle ID)
`State`	String	The app's installation state. If the state is NeedsRedemption, the server needs to send a redemption code to complete the app installation.

If the app cannot be installed, the device responds with an Error status, with the following fields:

Key	Type	Content
`RejectionReason`	String	One of the following: ● `AppAlreadyInstalled` ● `NotSupported` ● `CouldNotVerifyAppID` ● `AppStoreDisabled`

ApplyRedemptionCode Commands Install Paid Applications via Redemption Code

If a redemption code is needed during app installation, the server can use the `ApplyRedemptionCode` command to complete the app installation:

Key	Type	Content
`RequestType`	String	`ApplyRedemptionCode`
`Identifier`	String	The App ID returned by the InstallApplication command
`RedemptionCode`	String	The redemption code that applies to the app being installed.

If the user accepts the request, an acknowledgement response is sent.

Note: It is an error to send a redemption for an app that doesn't require a redemption code.

ManagedApplicationList Commands Provide the Status of Managed Applications

The `ManageApplicationList` command allows the server to query the status of managed apps.

Note: Certain statuses are transient. Once they are reported to the server, the entries for the apps are removed from the next query.

To send a ManagedApplicationList command, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`ManagedApplicationList`

In response, the device sends a dictionary with the following keys:

Key	Type	Content
`ManagedApplicationList`	Dictionary	A dictionary of managed apps.

The keys of the `ManagedApplicationList` dictionary are the app identifiers for the managed apps. The corresponding values are dictionaries that contain the following keys:

Key	Type	Content
`Status`	String	The status of the managed app. One of the following: `NeedsRedemption` - The app is scheduled for installation, but needs a redemption code to complete the transaction. `Redeeming` - The device is redeeming the redemption code. `Prompting` - The user is being prompted for app installation. `Installing` - The app is being installed. `Managed` - The app is installed and managed. `ManagedButUninstalled` - The app is managed, but has been removed by the user. When the app is installed again (even by the user), it will be managed once again. `Unknown` - The app state is unknown. The following statuses are transient, and are reported only once: `UserInstalledApp` - The user has installed the app before managed app installation could take place. `UserRejected` - The user rejected the offer to install the app. `Failed` - The app installation has failed.
`ManagementFlags`	Integer	Management flags. (See InstallApplication command above for a list of flags.)
`UnusedRedemptionCode`	String	If the user has already purchased a paid app, the unused redemption code is reported here. This code can be used again to purchase the app for someone else. This code is reported only once.

RemoveApplication Commands Remove Installed Managed Applications

The RemoveApplication command is used to remove managed apps and their data from a device. Applications not installed by the server cannot be removed with this command. To send a RemoveApplication command, the server sends a dictionary containing the following commands:

Key	Type	Content
`RequestType`	String	`RemoveApplication`
`Identifier`	String	The application's identifier.

Managed Settings

In iOS 5 or later, this command allows the server to set settings on the device. These settings take effect on a one-time basis. The user may still be able to change the settings at a later time. This command requires the Apply Settings right.

Key	Type	Content
`RequestType`	String	`Settings`
`Settings`	Array	Array of dictionaries. See below.

Each entry in the `Settings` array must be a dictionary. The specific values in that dictionary are described in the documentation for the specific setting.

Unless the command is invalid, the `Settings` command always returns an `Acknowledged` status. However, the response dictionary contains an additional key-value pair:

Key	Type	Content
`Settings`	Array	Array of results. See below.

In the response, the `Settings` array contains a result dictionary that corresponds with each command that appeared in the original `Settings` array (in the request). These dictionaries contain the following keys and values:

Key	Type	Content
`Status`	String	Status of the command. Only `Acknowledged` and `Error` are reported.
`ErrorChain`	Array	*Optional.* An array representing the chain of errors that occurred.

Each entry in the `ErrorChain` array is a dictionary containing the same keys found in the top level `ErrorChain` dictionary of the protocol.

VoiceRoaming Modifies the Voice Roaming Setting

To send a `VoiceRoaming` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`Item`	String	`VoiceRoaming`
Enabled	Boolean	If `true`, enables voice roaming. If `false`, disables voice roaming. The voice roaming setting is only available on certain carriers. Disabling voice roaming also disables data roaming.

DataRoaming Modifies the Data Roaming Setting

To send a `DataRoaming` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`Item`	String	`DataRoaming`
`Enabled`	Boolean	If `true`, enables data roaming. If `false`, disables data roaming. Enabling data roaming also enables voice roaming.

MDM Check-in Protocol

The MDM check-in protocol is used during initialization to validate a device's eligibility for MDM enrollment and to inform the server that a device's device token has been updated.

If a check-in server URL is provided in the MDM payload, the check-in protocol is used to communicate with that check-in server. If no check-in server URL is provided, the main MDM server URL is used instead.

Structure of a Check-in Request

The device initiates communication with the check-in server. The device validates the TLS certificate of the server, then uses the identity specified in its MDM payload as the client authentication certificate for the connection.

After successfully negotiating this secure connection, the device sends an HTTP PUT request in this format:

```
PUT /your/url HTTP/1.1
Host: www.yourhostname.com
Content-Length: 1234
Content-Type: application/x-apple-aspen-mdm-checkin

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>MessageType</key>
    <string>Authenticate</key>
    <key>Topic</key>
    <string>...</string>
    <key>UDID</key>
    <string>...</string>
  </dict>
</plist>
```

The server must send a `200 (OK)` status code to indicate success, or a `401 (Unauthorized)` status code to indicate failure. The body of the reply is ignored.

Supported Check-in Commands

Authenticate

The `Authenticate` command contains three key-value pairs in its property list:

Key	Type	Value
MessageType	String	Authenticate
Topic	String	The topic the device will listen to.
UDID	String	The device's UDID.

This message is sent while a user is installing an MDM payload.

Response

On success, the server must respond with a 200 OK status.

The server should not assume that the device has installed the MDM payload at this time, as other payloads in the profile may still fail to install.

When the device has successfully installed the MDM payload, it will send a token update message.

TokenUpdate

A device sends a token update message to the check-in server whenever its device token changes so that the server can continue to send it push notifications.

The TokenUpdate command contains up to six key-value pairs in its property list:

Key	Type	Value
MessageType	String	TokenUpdate
Topic	String	The topic the device will listen to.
UDID	String	The device's UDID.
Token	Data	The Push token for the device.
PushMagic	String	The magic string that must be included in the push notification message. This value is generated by the device.
UnlockToken	Data	Optional. A data blob that can be used to unlock the device. If provided, the server should remember this data blob and send it with the "ClearPasscode Commands Clear the Passcode for a Device" (page 24) command.

The server should use the updated token when sending push notifications to the device.

The device sends an initial token update message to the server when it has installed the MDM payload. The server should send push messages to the device only after receiving the first token update message.

Note: The topic string for the MDM check-in protocol must start with `com.apple.mgmt.*` where `*` is a unique suffix.

CheckOut

In iOS 5.0 and later, if the `CheckOutWhenRemoved` key in the MDM payload is set to `true`, the device attempts to send a `CheckOut` message when the MDM profile is removed.

If network conditions do not allow the message to be delivered successfully, the device makes no further attempts to send the message.

The server's response to this message is ignored.

The `CheckOut` message contains the following keys:

Key	Type	Content
`MessageType`	String	`CheckOut`
`Topic`	String	The topic the device will listen to.
`UDID`	String	The device's UDID.

MDM Vendor CSR Signing Overview

The process of generating a APNS push certificate for use with an MDM server has been updated to make it easier for customers to get up and running with Mobile Device Management. To take advantage of this new process, you must update your MDM product with the capabilities described in this chapter.

Customers can learn how the updated process works at http://www.apple.com/business/mdm.

The existing process for generating an APNS push certificate via the Provisioning Portal in the iOS Developer Enterprise Program will remain in place for a short period of time while you implement these changes. After that, it will be deprecated in favor of using the Apple Push Notification Portal.

Creating a Certificate Signing Request (Customer action)

1. During the setup process for your service, create an operation that generates a Certificate Signing Request for your customer.
2. This process should take place within the instance of your MDM service that your customer has access to.

> **Note:** The private key associated with this CSR should remain within the instance of your MDM service that the customer has access to. This private key is used to validate the full certificate chain of the MDM push certificate. The MDM service instance should not make this private key available to you (the vendor).

Via your setup process, the CSR should be uploaded to your internal infrastructure to be signed as outlined below.

Signing the Certificate Signing Request (MDM Vendor Action)

Before you receive a CSR from your customer, you must download an "MDM Signing Certificate" and the associated trust certificates via the iOS Provisioning Portal.

Next, you must create a script based on the instructions below to sign the customer's CSR:

1. If the CSR is in PEM format, convert CSR to DER (binary) format.
2. Sign the CSR (in binary format) with the private key of the MDM Signing Cert using the `SHA1WithRSA` signing algorithm.

Note: Do not share your MDM Signing Cert with anyone, including customers or resellers of your solution. The process of signing the CSR should take place within your internal infrastructure and should not be accessible to customers.

3. Base64 encode the signature.
4. Base64 encode the CSR (in binary format).
5. Create a Push Certificate Request plist and Base64 encode it. Refer to the code samples in Listing 1 (page 36), Listing 2 (page 37), and Listing 3 (page 37) for additional instructions.
6. Deliver the PushCertWebRequest file back to the customer and direct them to https://identity.apple.com/pushcert to upload it to Apple.

Creating the APNS Certificate for MDM (Customer Action)

Once you have delivered the signed CSR back to the customer, the customer must log into https://identity.apple.com/pushcert using a verified Apple ID and upload the CSR to the Apple Push Certificates Portal.

The portal creates a certificate titled "MDM_<*VendorName*>_Certificate.pem". At this point, the customer returns to your setup process to upload the APNS Certificate for MDM.

Code Samples

The following code snippets demonstrate the CSR signing process.

Listing 1 Sample Java Code

```
/**
* Sign the CSR ( DER format ) with signing private key.
* SHA1WithRSA is used for signing. SHA1 for message digest and RSA to encrypt
the message digest.
*/
byte[] signedData = signCSR(signingCertPrivateKey, csr);

String certChain = "-----BEGIN CERTIFICATE----";
/**
* Create the Request Plist. The CSR and Signature is Base64 encoded.
*/
byte[] reqPlist = createPlist(new String(Base64.encodeBase64(csr)),certChain,
new String(Base64.encodeBase64(signedData)));


/**
* Signature actually uses two algorithms--one to calculate a message digest and
 one to encrypt the message digest
* Here is Message Digest is calculated using SHA1 and encrypted using RSA.
* Initialize the Signature with the signer's private key using initSign().
```

```
* Use the update() method to add the data of the message into the signature.
*
* @param privateKey Private key used to sign the data
* @param data    Data to be signed.
* @return Signature as byte array.
* @throws Exception
*/
private byte[] signCSR( PrivateKey privateKey, byte[] data ) throws Exception{
    Signature sig = Signature.getInstance("SHA1WithRSA");
    sig.initSign(privateKey);
    sig.update(data);
    byte[] signatureBytes = sig.sign();
    return signatureBytes;
}
```

Listing 2 Sample .NET Code

```
var privateKey = new PrivateKey(PrivateKey.KeySpecification.AtKeyExchange,
2048, false, true);
var caCertificateRequest = new CaCertificateRequest();
string csr = caCertificateRequest.GenerateRequest("cn=test", privateKey);

//Load signing certificate from MDM_pfx.pfx, this is generated using
signingCertificatePrivate.pem and SigningCert.pem.pem using openssl
var cert = new X509Certificate2(MY_MDM_PFX,
PASSWORD, X509KeyStorageFlags.Exportable);

//RSA provider to generate SHA1WithRSA
var crypt = (RSACryptoServiceProvider)cert.PrivateKey;
var sha1 = new SHA1CryptoServiceProvider();
byte[] data = Convert.FromBase64String(csr);
byte[] hash = sha1.ComputeHash(data);
//Sign the hash
byte[] signedHash = crypt.SignHash(hash, CryptoConfig.MapNameToOID("SHA1"));
var signedHashBytesBase64 = Convert.ToBase64String(signedHash);
```

Listing 3 Sample Request property list

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>PushCertRequestCSR</key>
<string>
MIIDjzCCAncCAQAwDzENMAsGA1UEAwwEdGVzdDCCASIwDQYJKoZIhvcNAQEBBQAD
</string>
<key>PushCertCertificateChain</key>
<string>
-----BEGIN CERTIFICATE-----
MIIDkzCCAnugAwIBAgIIQcQgtHQb9wwwDQYJKoZIhvcNAQEFBQAwUjEaMBgGA1UE
AwwRU0FDSSBUZXN0IFJvb3QgQ0ExEjAQBgNVBAsMCUFwcGxlIElnTVDETMBEGA1UE
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDlTCCAn2gAwIBAgIIBInl9fQbaAkwDQYJKoZIhvcNAQEFBQAwXDEkMCIGA1UE
AwwbU0FDSSBUZXN0IEludGVybWVkaWF0ZSBDQSAxMRIwEAYDVQQLDAlBcHBsZSBJ
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
```

```
MIIDpjCCAo6gAwIBAgIIKRyFYgyyFPgwDQYJKoZIhvcNAQEFBQAwXDEkMCIGA1UE
AwwbUOFDSSBUZXNOIEludGVybWVkaWFOZSBDQSAxMRIwEAYDVQQLDAlBcHBsZSBJ
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDiTCCAnGgAwIBAgIIdv/cjbnBgEgwDQYJKoZIhvcNAQEFBQAwUjEaMBgGA1UE
AwwRUOFDSSBUZXNOIFJvb3QgQOExEjAQBgNVBAsMCUFwcGxlIElTVDETMBEGA1UE
-----END CERTIFICATE-----
</string>
<key>PushCertSignature</key>
<string>
CGt6QWuixaOOPIBc9dr2kJpFBE1BZx2D8LOXHOMtc/DePGJOjrM2W/IBFYOAVhhEx
</string>
```

MDM Best Practices

Although there are many ways to deploy mobile device management, the techniques and policies described in this chapter make it easier to deploy MDM in a sensible and secure fashion.

Tips For Specific Profile Types

Although you can include any amount of information in your initial profile, it is easier to manage profiles if your base profile provides little beyond the MDM payload. You can always add additional restrictions and capabilities in separate payloads.

Initial Profiles Should Contain Only The Basics

The initial profile deployed to a device should contain only the following payloads:

- Any root certificates needed to establish SSL trust.
- Any intermediate certificates needed to establish SSL trust.
- A client identity certificate for use by the MDM payload (either a PKCS#12 container, or an SCEP payload). An SCEP payload is recommended.
- The MDM payload.

Once the initial profile is installed, your server can push additional managed profiles to the device.

Managed Profiles Should Pair Restrictions With Capabilities

Configure each managed profile with a related pair of restrictions and capabilities (the proverbial carrots and sticks) so that the user gets specific benefits (access to an account, for instance) in exchange for accepting the associated restrictions.

For example, your IT policy may require a device to have a 6-character passcode (stick) in order to access your corporate VPN service (carrot). You can do this in two ways:

- Deliver a single managed profile with both a passcode restriction payload and a VPN payload.
- Deliver a locked profile with a passcode restriction, optionally poll the device until it indicates compliance, and then deliver the VPN payload.

Either technique ensures that the user cannot remove the passcode length restriction without losing access to the VPN service.

Each Managed Profile Should Be Tied to a Single Account

Do not group multiple accounts together into a single profile. Having a separate profile for each account makes it easier to replace and repair each account's settings independently, add and delete accounts as access needs change, and so on.

This advantage becomes more apparent when your organization uses certificate-based account credentials. As client certificates expire, you can replace those credentials one account at a time. Because each profile contains a single account, you can replace the credentials for that account without needing to replace the credentials for every account.

Similarly, if a user requests a password change on an account, your servers could update the password on the device. If multiple accounts are grouped together, this would not be possible unless the servers keep an unencrypted copy of all of the user's other account passwords (which is dangerous).

Managed Profiles Should Not Be Locked

In general, managed profiles should be unlocked. This allows users to remove accounts that they do not need or want on their devices. Alternatively, you may deliver locked profiles and provide a self-service web portal that lets users customize which accounts they want configured on their devices.

Keep in mind that all managed profiles (including managed provisioning profiles) are removed when the user removes the profile containing the MDM payload (which cannot be locked). Consider allowing your users to customize their account selections, knowing that all accounts in managed profiles will be removed when they decide to end their devices' management relationships with your server.

Provisioning Profiles Can Be Installed Using MDM

Third-party enterprise applications require provisioning profiles in order to run them. You can use MDM to deliver up-to-date versions of these profiles so that users do not have to manually install these profiles, replace profiles as they expire, and so on.

To do this, deliver the provisioning profiles through MDM instead of distributing them through your corporate web portal or bundled with the application.

Security Note: Although an MDM server can remove provisioning profiles, you should not depend on this mechanism to revoke access to your enterprise applications for two reasons:

- An application continues to be usable until the next device reboot even if you remove the provisioning profile.
- Provisioning profiles are synchronized with iTunes. Thus, they may get reinstalled the next time the user syncs the device.

Passcode Policy Compliance

Because an MDM server may push a profile containing a passcode policy without user interaction, it is possible that a user's passcode must be changed to comply with a more stringent policy. When this situation arises, a 60-minute countdown begins. During this grace period, the user is prompted to change the passcode whenever he or she returns to the home screen, but can dismiss the prompt and continue working. After the 60 minutes grace period, the user must change his or her passcode in order to launch any application on the device, including built-in applications.

An MDM server can check to see if a user has complied with all passcode restrictions using the `SecurityInfo` command. An MDM server can wait until the user has complied with passcode restrictions before pushing other profiles to the device.

Deployment Scenarios

There are several ways to deploy an MDM payload. Which scenario is best depends on the size of your organization, whether an existing device management system is in place, and what your IT policies are.

Tethered via iPCU

Your IT personnel can connect the device to a computer running iPCU, and install the MDM profile directly. The IT personnel must customize the client identity certificate for each installation by attaching a unique PKCS#12 payload, or by setting a unique challenge for an SCEP payload.

While this approach is straightforward, it is time-consuming, and not scalable to a large number of devices.

OTA Profile Enrollment

You may use over-the air enrollment (described in *Over-the-Air Profile Delivery and Configuration*) to deliver a profile to a device. This option allows your servers to validate a user's login, query for more information about the device, and validate the device's built-in certificate before delivering a profile containing an MDM payload.

When a profile is installed through over-the air enrollment, it is also eligible for updates. When a certificate in the profile is about to expire, an "Update" button will appear that allows the user to fetch a more recent copy of the profile using his or her existing credentials.

This approach is recommended for most organizations because it is scalable.

Vendor-Specific Installation

Third-party vendors may install the MDM profile in a variety of other ways that are integrated with their management systems.

SSL Certificate Trust

MDM only connects to servers that have valid SSL certificates. If your server's SSL certificate is rooted in your organization's root certificate, the device must trust the root certificate before MDM will connect to your server.

You may include the root certificate and any intermediate certificates in the same profile that contains the MDM payload. Certificate payloads are installed before the MDM payload.

Your MDM server should replace the profile that contains the MDM payload well before any of the certificates in that profile expire. Remember: if any certificate in the SSL trust chain expires, the device cannot connect to the server to receive its commands. When this occurs, you lose the ability to manage the device.

Distributing Client Identities

Each device must have a unique client identity certificate. You may deliver these certificates as PKCS#12 containers, or via SCEP. Using SCEP is recommended because the protocol ensures that the private key for the identity exists only on the device.

Consult your organization's Public Key Infrastructure policy to determine which method is appropriate for your installation.

Identifying Devices

An MDM server should identify a connecting device by examining the device's client identity certificate. The server should then cross-check the UDID reported in the message to ensure that the UDID is associated with the certificate.

The device's client identity certificate is used to establish the SSL/TLS connection to the MDM server. If your server sits behind a proxy that strips away (or does not ask for) the client certificate, read "Passing the Client Identity Through Proxies" (page 42).

Passing the Client Identity Through Proxies

If your MDM server is behind an HTTPS proxy that does not convey client certificates, MDM provides a way to tunnel the client identity in an additional HTTP header.

If the value of the `SignMessage` field in the MDM payload is set to true, each message coming from the device will carry an additional HTTP header named `Mdm-Signature`. This header contains a BASE64-encoded CMS Detached Signature of the message.

Your server can validate the body with the detached signature in the `SignMessage` header. If the validation is successful, your server can assume that the message came from the signer, whose certificate is stored in the signature.

Keep in mind that this option consumes a lot of data relative to the typical message body size. The signature is sent with every message, adding almost 2 KB of data to each outgoing message from the device. You should use this option only if needed.

Detecting Inactive Devices

To be notified when a device becomes inactive, set the `CheckOutWhenRemoved` key to `true` in the MDM payload. Doing so causes the device to contact your server when it ceases to be managed. However, because a managed device makes only a single attempt to deliver this message, you should also employ a timeout to detect devices that fail to check out due to network conditions.

To do this, your server should send a push notification periodically to ensure that managed devices are still listening to your push notifications. If the device fails to respond to push notifications after some time, the device can be considered inactive. A device can become inactive for several reasons:

- The MDM profile is no longer installed
- The device has been erased
- The device has been disconnected from the network
- The device has been turned off.

The time that your server should wait before deciding that a device is inactive can be varied according to your IT policy, but a time period of several days to a week is recommended. While it's harmless to send push notifications once a day or so to make sure the device is responding, it is not necessary. Apple's Push Notification servers will cache your last push notification, and will deliver it to the device when it comes back on the network.

When a device becomes inactive, your server may take appropriate action, such as limiting the device's access to your organization's resources until the device starts responding to push notifications once more.

Using the Feedback Service

Your server should regularly poll the Apple Push Notification Feedback Service to detect if a device's push token has become invalid. When a device token is reported invalid, your server should consider the device to be no longer managed, and should stop sending push notifications or commands to the device. If needed, you may also take appropriate action to restrict the device's access to your organization's resources.

The Feedback service should be considered unreliable for detecting device inactivity, because you may not receive feedback in certain cases. Your server should use timeouts as the primary means of determining device management status.

Dequeueing Commands

Your server should not consider a command accepted and executed by the device until you receive the `Acknowledged` or `Error` status with the command UUID in the message. In other words, your server should leave the last command on the queue until you receive the status for that command.

It is possible for the device to send the same status twice. You should examine the `CommandUUID` field in the device's status message to determine which command it applies to.

Handling a NotNow Response

If the device responds with a `NotNow` status, your server has several response choices.

- It may stop sending commands until the device polls it again. Your server should not send another push notification; the device will automatically poll your server when conditions change such that it is able to process more commands. Your server can then resend the same command.
- It may send another command that is guaranteed to execute.

Terminating a Management Relationship

You can terminate a management relationship with a device by performing one of these actions:

- Remove the profile that contains the MDM payload. An MDM server can always remove this profile, even if it does not have the access rights to add or remove configuration profiles.
- Respond to any device request with a `401 Unauthorized` HTTP status. The device automatically removes the profile containing the MDM payload upon receiving a `401` status code.

Dealing with Restores

A user can restore his or her device from a backup. If the backup contains an MDM payload, MDM service is reinstated, and the device is automatically scheduled to deliver a `TokenUpdate` check-in message.

Your server can either accept the device by replying with a 200 status, or reject the device with a `401` status. If your server replies with a `401` status, the device removes the profile that contains the MDM payload.

It is good practice to respond with a `401` status to any device that the server is not actively managing.

Securing the ClearPasscode command

Though this may sound obvious, clearing the passcode on a managed device compromises its security. Not only does it allow access to the device without a passcode, it also disables Data Protection.

If your MDM payload specifies the Device Lock right, the device includes an `UnlockToken` data blob in the `TokenUpdate` message that it sends your server after installing the profile. This data blob contains a cryptographic package that allows the device to be unlocked. You should treat this data as the equivalent of a "master passcode" for the device. Your IT policy should specify how this data is stored, who has access to it, and how the `ClearPasscode` command can be issued and accounted for.

You should not send the `ClearPasscode` command until you have verified that the device's owner has physical ownership of the device. You should *never* send the command to a lost device.

Managing Applications

MDM is the recommended way to manage applications for your enterprise. You can use MDM to help users install enterprise apps, and in iOS 5.0 and later, you can also install App Store apps purchased using the volume purchase program (VPP). The way that you manage these applications depends on the version of iOS that a device is running.

iOS 5.0 And Later

In iOS 5.0 and later, using MDM to manage apps gives you several advantages:

- You can purchase apps for users without manually distributing redemption codes.
- You can notify the user that an app is available for them to install. (The user must agree to installation before the app is installed.)
- A managed app can be excluded from the user's backup. This prevents the app's data from leaving the device during a backup.
- The app can be configured so that the app and its data are automatically removed when the MDM profile is removed. This prevents the app's data from persisting on a device unless it is managed.

An app purchased from the App Store and installed on a user's device is "owned" by the iTunes account used at the time of installation. This means that the user may install the app (not its data) on unmanaged devices.

An app internally developed by an enterprise is not backed up. A user cannot install such an app on an unmanaged device.

In order to support this behavior, your internally hosted enterprise app catalog must use the `InstallApplication` command instead of providing a direct link to the app (with a manifest URL or iTunes Store URL). This allows you to mark the app as managed during installation.

iOS 4.x And Later

To disable enterprise apps, you can remove the provisioning profile that they depend on. However, as mentioned in "Provisioning Profiles Can Be Installed Using MDM" (page 40), you should *not* solely rely on that mechanism for limiting access to your enterprise applications for two reasons:

- Removing a provisioning profile does not prevent the app from launching until the device is rebooted.
- The provisioning profile is likely to have been synced to a computer, and thus will probably be reinstalled during the next sync.

To limit access to your enterprise application, follow the following recommendations:

- Have an online method of authenticating users when they launch your app. Use either a password or identity certificate to authenticate the user.
- Store local app data in your application's Caches folder to prevent the data from being backed up.
- When you decide that the user should no longer have access to the application's data, mark the user's account on the server inactive in some way.
- When your app detects that the user is no longer eligible to access the app, if the data is particularly sensitive, it should erase the local app data.
- If your application has an "offline" mode, you should limit the amount of time users can access the data before reauthenticating online. Ensure that this timeout is enforced across multiple application launches.

  If desired, you can also limit the number of launches to prevent time server forging attacks.

  Be sure to store any information about the last successful authentication in your Caches folder (or in the keychain with appropriate flags) so that it does not get backed up. If you do not, the user could potentially modify the time stamp in a backup file, resync the device, and continue using the application.

These guidelines assume that all the application's data is replicated on your server. If you have data that resides only on the device (including offline edits), you should preserve a copy of the user's changes on the server. Be sure to do so in a way that protects the integrity of the server's data against disgruntled former users.

Document Revision History

This table describes the changes to *Mobile Device Management Protocol Reference*.

Date	Notes
2011-10-03	Corrected push cert URL.
	Updated for iOS 5.0
2011-02-16	Updated for CDMA support
2010-12-09	Updated for iOS 4.2
2010-09-14	First version

Document Revision History