Mobile Device Management Protocol Reference

Developer

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 iOS devices running iOS 4 and later, OS X devices running OS X 10.7 and later and Apple TV devices running iOS 7 (Apple TV software 6.0) 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.

> **Note:** For information about about SCEP, see the draft SCEP specification located at datatracker.ietf.org/doc/draft-nourse-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 automatically using the Device Enrollment Program. 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 configuration profiles or provisioning profiles, it may only remove apps, configuration profiles, and provisioning profiles that it originally installed. Accounts installed using managed profiles are called Managed Accounts.

In addition to Managed Profiles, you can also use MDM to install apps. Apps installed through the MDM service are called Managed Apps. The MDM service has additional control over how Managed Apps and their data are used on the device.

Devices running iOS 5 and later can be designated as supervised when preparing it for deployment with Apple Configurator. Additionally, devices running iOS 7 and later can be supervised using the Device Enrollment Program. A supervised device provides an organization with additional control over its configuration and restrictions. In this document, if any configuration option is limited to supervised devices, its description notes that limitation.

Unless the profile is installed using the Device Enrollment Program, a user may remove the profile containing the MDM payload at any time. The MDM server can always remove its own profile, regardless of its access rights. In OS X 10.8 and later and iOS 5, the MDM client makes a single attempt to contact the server with the `"CheckOut"` command when the profile is removed. On earlier OS versions, the device does not contact the MDM server when the user removes the payload. Please see "MDM Best Practices" (page 140) for recommendations on how to detect devices that are no longer managed.

A profile containing an MDM payload cannot be locked unless it is installed using the Device Enrollment Program. 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 devices in enterprise environments.

The MDM Check-In Protocol Lets a Device Contact Your Server

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.

Related Chapter: "MDM Check-in Protocol" (page 12)

The MDM Protocol Sends Management Commands to the Device

The (main) MDM protocol uses push notifications to tell the managed device to perform specific functions, such as deleting an app or performing a remote wipe.

Related Chapter: "Mobile Device Management (MDM) Protocol" (page 16)

The Way You Design Your Payload Matters

For maximum effectiveness and security, you should install a base profile that contains little more than the most basic MDM management information, then install other profiles to the device after it is managed.

Related Chapter: "MDM Best Practices" (page 140)

The Device Enrollment Program Lets You Configure Devices With the Setup Assistant

The HTTP-based Device Enrollment Program addresses the mass configuration needs of organizations purchasing and deploying devices in large quantities, without the need for factory customization or pre-configuration of devices prior to deployment.

The cloud service API provides profile management and mapping. With this API, you can create profiles, update profiles, delete profiles, obtain a list of devices, and associate those profiles with specific devices.

Related Chapter: "Device Enrollment Program" (page 72)

The Volume Purchase Program Lets You Assign App Licenses to Users

The Volume Purchase Program provides a number of web services that MDM servers can call to associate volume purchases with a particular user.

Related Chapter: "VPP App Assignment" (page 104)

Apple Push Notification Certificates Can Be Generated Through the APN Portal

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. Then, you must use that certificate to sign your customers' certificates.

Related Chapter: "MDM Vendor CSR Signing Overview" (page 150)

See Also

For discussions about Mobile Device Management, visit the MDM Developer Forum.

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.

> **Note:** MDM configuration profiles can be stored in and read from Apple Open Directory servers.

Structure of a Check-in Request

When the MDM payload is installed, 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 39) command. This feature is not available in OS X.

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, and in OS X 10.9, 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.

In OS X v10.8, the device attempts to send a `CheckOut` message when the MDM profile is removed regardless of the value of this key (or its absence).

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

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

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 administrator tells the device to install an MDM payload. The structure of this payload is described in "Structure of MDM Payloads" (page 17).
- The device connects to the check-in server. The device presents its identity certificate for authentication, along with its UDID and push notification topic.

  Note: Although UDIDs are used by MDM, the use of UDIDs is deprecated for iOS apps.

  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 push notification 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.

Mobile Device Management, as its name implies, was originally developed for embedded systems. To support environments where a computer is bound to an Open Directory server and various network users may login, extensions to the MDM protocol were developed to identify and authenticate the network user logging in so that any network user will also be managed by the MDM server (via their user profiles). The extensions made to the MDM protocol are described in "MDM Protocol Extensions" (page 24).

> **Note:** Login may be blocked momentarily while the MDM server is contacted for its latest settings. Device enrollment can also be performed later, after the computer is connected to the Internet.

Structure of MDM Payloads

The Mobile Device Management (MDM) payload, a simple property list, 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 144) 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`. Note: OS X v10.8 acts as though this setting is always `true`. **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—Allow manipulation of settings. **Availability:** Available in iOS 5.0 and later. Available in OS X 10.9 for certain commands. • 4096—Allow app management. **Availability:** Available in iOS 5.0 and later. Available in OS X 10.9 for certain commands. May not be zero. If 2 is specified, then 1 must also be specified. If 128 is specified, then 64 must also be specified. **Availability:** The OS X MDM client ignores flags above 1024.
`UseDevelopmentAPNS`	Bool	*Optional.* If `true`, the device uses the development APNS servers. Otherwise, the device uses the production servers. Defaults to `false`. Note that this property must be set to false if your Apple Push Notification Service certificate was issued by the Apple Push Certificate Portal (identity.apple.com/pushcert). That portal only issues certificates for the production push environment.

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 *Configuration Profile Reference*.

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

> **Note:** Profile payload dictionary keys that are prefixed with “Payload” are reserved key names and must never be treated as managed preferences. Any other key in the payload dictionary may be considered a managed preference for that preference domain.

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":"PushMagicValue"}
```

In place of `PushMagicValue` above, you should substitute the actual `PushMagic` string that the device sends to the MDM server in the `TokenUpdate` message. 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.

> **Note:** An empty response body must be zero bytes in length, not an empty JSON dictionary.

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 23).
`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 27).

The `Status` key will contain one of the following strings:

Table 1 MDM status codes

Status value	Description
`Acknowledged`	Everything went well.

Status value	Description
`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.

MDM Protocol Extensions

To support environments where a computer is bound to an Open Directory server and various network users may login, extensions to the MDM protocol were developed to identify and authenticate the network user logging in. This way, any network user is also managed by the MDM server via their user profiles.

At login time, if the user is a network user or has a mobile home, the MDM client issues a request to the server to authenticate the current user to the MDM server and obtain an `AuthToken` value that is used in subsequent requests made by this user to the server.

The authentication happens using a transaction similar in structure to existing transactions with the server, as an HTTP `PUT` request to `/mdm/checkin` at the `CheckInURL` address specified in the MDM payload.

The first request to the server is sent to `/mdm/checkin` with the same identity used for all other MDM requests. The message body contains a property list with the following keys:

Key	Type	Content
`MessageType`	String	`UserAuthenticate`
`UDID`	String	UDID used on all MDM requests
`UserID`	String	User's GUID from Open Directory Record

The response from the server should contain a dictionary with:

Key	Type	Content
`DigestChallenge`	String	`Standard HTTP Digest`

If the server provides a `200` response but a zero-length `DigestChallenge` value then the server does not require any AuthToken to be generated for this user.

Otherwise, with a `200` response and `DigestChallenge` value that is non-empty, the client generates a digest from the user's shortname, the user's clear-text password, and the `DigestChallenge` value obtained from the server. The resulting digest is sent in a second request to the server, which validates the response and returns an `AuthToken` value that is sent on subsequent requests to the server.

The second request to the server is also sent to `/mdm/checkin` and sent with the same identity used for all other MDM requests. The message body contains:

Key	Type	Content
`MessageType`	String	`UserAuthenticate`
`UDID`	String	UDID used on all MDM requests
`UserID`	String	User's GUID from Open Directory Record
`DigestResponse`	String	Obtained from generating digest above

The response from the server should contain a dictionary with:

Key	Type	Content
`AuthToken`	String	The token used for authentication

If the server responds with a `200` response and a non-empty `AuthToken` value is present, then the `AuthToken` value is sent to the server on subsequent requests. The `AuthToken` value is included in the message body of subsequent requests along with the additional keys:

Key	Type	Value
`UDID`	String	Device ID
`UserID`	String	GUID attribute form user's Open Directory record
`UserShortName`	String	Record name from user's Open Directory record
`UserLongName`	String	Full name from user's Open Directory record
`AuthToken`	String	Token obtained from above

If the server rejects the `DigestResponse` value because of an invalid password, then the OS X server returns a `200` response and an empty `AuthToken` value.

For push notifications, the client uses different push tokens for device and user connections. Each token will be sent to the server using the `TokenUpdate` request. The server can tell for whom the token is intended based on the `UDID` and `UserID` values in the request. If the user is a network/mobile user, the `AuthToken` is provided.

Warning: These push tokens should not be confused with the "AuthToken" mentioned above.

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 on iOS, and never return `NotNow`:

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

- `Restrictions`

The OS X MDM client may respond with `NotNow` when:

- The system is in Power Nap (dark wake) and a command other than `DeviceLock` or `EraseDevice` is received.
- An `InstallProfile` or `RemoveProfile` request is made on the user connection and the user's keychain is locked.

In OS X, the client may respond with `NotNow` if it is blocking the user's login while it contacts the server, and if the server sends a request that may take a long time to answer (such as `InstalledApplicationList` or `DeviceInformation`).

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. However, for reference, the current codes are listed in the sections that follow.

Request Types

This section describes the MDM protocol request types for Apple devices that run iOS. Support for the equivalent request types used with Apple computers that run OS X is summarized in "Support for OS X Requests" (page 57).

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 *Configuration Profile Reference*.

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

If you want to update a profile in place, by installing a new one where there is already one existing, follow these rules:

- The new profile must replace an existing MDM profile.
- You cannot change the topic or server URL of the profile.
- You cannot add rights to a profile that replaces an existing one.

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.

Note that in the definition of the InstallProfile command the Payload is of type Data, meaning that the entire Payload must be base64-encoded, including the XML headers. This is true for any Data type items in a property list. Please see "Understanding XML Property Lists" in *Property List Programming Guide* for more information.

> **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.

Note: The OS X MDM client will respond with an empty `ProvisioningProfileList` array.

InstallProvisioningProfile Commands Install Provisioning Profiles

To send an `InstallProvisioningProfile` command to an iOS device, 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 to an iOS device, 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 31).

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 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`

Key	Type	Content
`Identifiers`	Array	Optional. An array of app identifiers as strings. If provided, the response contains only the status of apps whose identifiers appear in this array. Available in iOS 7 and later.
`ManagedAppsOnly`	Boolean	Optional. If `true`, only managed app identifiers are returned. Available in iOS 7 and later.

The device replies with:

Key	Type	Content
`InstalledApplicationList`	Array	Array of installed applications. Each entry is a dictionary as described in Table 4 (page 32).

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. **Availability:** Available in iOS 5.0 and later.

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`

Key	Type	Content
`Queries`	Array	Array of strings. Each string is a value from Table 5 (page 33), Table 7 (page 34), or Table 8 (page 36).

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 33), Table 7 (page 34), or Table 8 (page 36). 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 device.
`Languages`	Array	Array of strings. The first entry in this array indicates the current language. Available in iOS 7 (Apple TV software 6.0) and later, on Apple TV only.
`Locales`	String	Array of strings. The first entry in this array indicates the current locale. Available in iOS 7 (Apple TV software 6.0) and later, on Apple TV only.
`DeviceID`	String	The Apple TV device ID. Available in iOS 7 (Apple TV software 6.0) and later, on Apple TV only.
`OrganizationInfo`	String	The contents (if any) of a previously set `OrganizationInfo` setting. Available in iOS 7 and later.

iTunesStoreAccountIsActive Commands Tell Whether An iTunes Account is Logged In

The query in Table 6 are available if the MDM host has an Install Applications access right:

Table 6 iTunes Store account queries

Query	Reply Type	Content
`iTunesStoreAccount-IsActive`	Boolean	`true` if the user is currently logged into an active iTunes Store account. Available in iOS 7 and later and in OS X 10.9.

Device Information Queries Provide Information About the Device

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

Table 7 Device information queries

Query	Reply Type	Comment
`DeviceName`	String	The name given to the iOS device via iTunes or the OS X device via System Preferences.
`OSVersion`	String	The version of iOS the device is running.
`BuildVersion`	String	The build number (8A260b, for example).
`ModelName`	String	Name of the device model, e.g. "MacBook Pro".
`Model`	String	The device's model number (`MC319LL`, for example).
`ProductName`	String	The model code for the device (iPhone3,1, for example).
`SerialNumber`	String	The device's serial number.
`DeviceCapacity`	Number	Floating-point gigabytes (base-1024 gigabytes).
`AvailableDeviceCapacity`	Number	Floating-point gigabytes (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.

Query	Reply Type	Comment
`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. **Availability:** Not supported in OS X.
`MEID`	String	The device's MEID number. Ignored if the device does not support CDMA. **Availability:** Not supported in OS X.
`ModemFirmwareVersion`	String	The baseband firmware version. **Availability:** Not supported in OS X.
`IsSupervised`	Boolean	If `true`, the device is supervised. **Availability:** Available in iOS 6 and later.
`IsDeviceLocator-ServiceEnabled`	Boolean	If `true`, the device has a device locator service (such as Find My iPhone) enabled. **Availability:** Available in iOS 7 and later.
`IsActivation-LockEnabled`	Boolean	If true, the device has Activation Lock enabled.
`IsDoNotDisturb-InEffect`	Boolean	If `true`, Do Not Disturb is in effect. This returns `true` whenever Do Not Disturb is turned on, even if the device is not currently locked. **Availability:** Available in iOS 7 and later.
`DeviceID`	String	Device ID. **Availability:** Available in iOS 7 (Apple TV software 6.0) and later on Apple TV only.
`EASDeviceIdentifier`	String	The Device Identifier string reported to Exchange Active Sync (EAS).

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

The queries in Table 8 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. The OS X MDM client will only respond to `BluetoothMAC`, `WiFiMAC`, and `EthernetMACs`.

Table 8 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.
`EthernetMACs`	Array of strings	Ethernet MAC addresses. **Availability:** Available in OS X v10.8 and later, and in iOS 7 and later.
`CurrentCarrier-Network`	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.)
`SubscriberCarrier-Network`	String	Name of the home carrier network. (Replaces `SIMCarrierNetwork`.) **Availability:** Available in iOS 5.0 and later.
`CarrierSettings-Version`	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.

Query	Reply Type	Comment
IsRoaming	Bool	Returns whether the device is currently roaming. **Availability:** Available in iOS 4.2 and later. See note below.
PersonalHotspotEnabled	Bool	True if the Personal Hotspot feature is currently turned on. This value is available only with certain carriers. **Availability:** iOS 7.0 and later.
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.
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.

In iOS only, 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 9 (page 38).
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.

Note: In OS X 10.9, the OS X client replies to a SecurityInfo request with only the current state of Full Disk Encryption. The response contains the keys FDE_Enabled, FDE_HasPersonalRecoveryKey, and FDE_HasInstitutionalRecoveryKey. All these values are Boolean.

Hardware encryption capabilities are described using the logical OR of the values in Table 9. Bits set to 1 (one) indicate that the corresponding feature is present, enabled, or in effect.

Table 9 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.

Note: The OS X MDM client will respond with an empty `SecurityInfo` dictionary.

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`
`PIN`	String	The Find My Mac PIN. Must be 4 or 6 characters long. Note that this field is provided only for OS X, not iOS.
`Message`	String	Optional. If provided, this message is displayed on the lock screen. Available in iOS 7 and later.
`PhoneNumber`	String	Optional. If provided, this phone number is displayed on the lock screen. Available in iOS 7 and later.

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	The `UnlockToken` value that the device provided in its "TokenUpdate" (page 13) check-in message.

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

Note: The OS X MDM client will generate an `Error` response to the server.

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`
`PIN`	String	The Find My Mac PIN. Must be 4 or 6 characters long. Note that this field is provided only for OS X, not iOS.

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.

RequestMirroring and StopMirroring control AirPlay Mirroring

In iOS 7 and later, the MDM server can send the `RequestMirroring` and `StopMirroring` commands to start and stop AirPlay mirroring.

Note: The `StopMirroring` command is supported in supervised mode only.

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

Key	Type	Content
`RequestType`	String	`RequestMirroring`
`DestinationName`	String	*Optional.* The name of the AirPlay mirroring destination. For Apple TV, this is the name of the Apple TV.

Key	Type	Content
`DestinationDeviceID`	String	*Optional.* The device ID (hardware address) of the AirPlay mirroring destination, in the format "xx:xx:xx:xx:xx:xx". This field is not case sensitive.
`ScanTime`	String	*Optional.* Number of seconds to spend searching for the destination. The default is 30 seconds. This value must be in the range 10–300.
`Password`	String	*Optional.* The screen sharing password that the device should use when connecting to the destination.

Note: Either `DestinationName` or `DestinationDeviceID` must be provided. If both are provided, `DestinationDeviceID` is used.

In response, the device provides a dictionary with the following key:

Key	Type	Content
`MirroringResult`	String	The result of this request. The returned value is one of: `Prompting`—The user is being prompted to share his or her screen. `DestinationNotFound`—The destination cannot be reached by the device. `Cancelled`—The request was cancelled. `Unknown`—An unknown error occurred.

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

Key	Type	Content
`RequestType`	String	`StopMirroring`

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 *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.

Note: Restrictions commands are not supported on the OS X MDM client.

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.

In iOS 7 and later, an MDM server can provide a configuration dictionary to third-party apps and can read data from a feedback dictionary provided by third-party apps. See "Managed App Configuration and Feedback" (page 54) for details.

To install a managed app on an iOS device, the MDM server sends an installation command to the user's device. Unless the device is supervised, 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, if the app was not purchased using App Assignment, 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.

The OS X MDM client does not support managing applications. However, it does support the parts of the `InstallApplication`, `InstallMedia`, and `InviteToProgram` MDM commands related to VPP enrollment and installation.

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`

Key	Type	Content
`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.
`Identifier`	String	Optional. The application's bundle identifier. Available in iOS 7 and later.
`Options`	Dictionary	Optional. App installation options. The available options are listed below. Available in iOS 7 and later.
`ManifestURL`	String	The `https` URL where the manifest of an enterprise application can be found. Note that in iOS 7 and later, this URL *must* begin with `https`.
`ManagementFlags`	Integer	The bitwise OR of the following flags: 1—Remove app when MDM profile is removed. 4—Prevent backup of the app data.
Configuration	Dictionary	Optional. If provided, this contains the initial configuration dictionary for the managed app. For more information, see "Managed App Configuration and Feedback" (page 54).
Attributes	Dictionary	Optional. If provided, this dictionary contains the initial attributes for the app. For a list of allowed keys, see "ManagedApplicationAttributes Queries App Attributes" (page 55).

The request must contain exactly one of the following fields: `Identifier`, `iTunesStoreID`, or `ManifestURL` value.

The options dictionary can contain the following keys:

Key	Type	Content
`NotManaged`	Boolean	If true, the app is queued for installation, but is not managed. OS X app installation must set this value to `true`.
`PurchaseMethod`	Integer	One of the following: 0: Volume Purchase Program 1: Volume Purchase Program App Assignment

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 it is PromptingForUpdate, the process is waiting for the user to approve an app update.

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` • `AppAlreadyQueued` • `NotSupported` • `CouldNotVerifyAppID` • `AppStoreDisabled` • `NotAnApp` • `PurchaseMethodNotSupported` (iOS 7 and later)

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`
`Identifiers`	Array	Optional. An array of app identifiers as strings. If provided, the response contains only the status of apps whose identifiers appear in this array. Available in iOS 7 and later.

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.

Key	Type	Content
		`PromptingForLogin`—The user is being prompted for App Store credentials. `Installing`—The app is being installed. `ValidatingPurchase`—An app purchase is being validated. `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. `PromptingForUpdate`—The user is being prompted for an update PromptingForUpdateLogin - the user is being prompted for App Store credentials for an update `Updating`—The app is being updated `ValidatingUpdate`—An app update is being validated. `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. `UpdateRejected`—The user rejected the offer to update the app. `Failed`—The app installation has failed.
`ManagementFlags`	Integer	Management flags. (See InstallApplication command above for a list of flags.)

Key	Type	Content
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.
HasConfiguration	Boolean	If true, the app has a server-provided configuration. For details, see "Managed App Configuration and Feedback" (page 54). Available in iOS 7 and later.
HasFeedback	Boolean	If true, the app has feedback for the server. For details, see "Managed App Configuration and Feedback" (page 54). Available in iOS 7 and later.

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.

InviteToProgram Lets the Server Invite a User to Join a Volume Purchasing Program

In iOS 7 and later, this command allows a server to invite a user to join a volume purchasing program. This command issues the invitation, but does not allow the server to monitor whether the user has joined the program.

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

Key	Type	Content
RequestType	String	InviteToProgram
ProgramID	String	The program's identifier. One of the following: com.apple.cloudvpp: Volume Purchase Program App Assignment

Key	Type	Content
`InvitationURL`	String	An invitation URL provided by the program.

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

Key	Type	Content

`InvitationResult`	String	`One of the following:`
		`Acknowledged`
		`InvalidProgramID`
		`InvalidInvitationURL`

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.

The OS X MDM client does not support managing settings.

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.
`Identifier`	String	Optional. The app identifier to which this error applies. **Availability:** Available in iOS 7 and later.

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.

PersonalHotspot Modifies the Personal Hotspot Setting

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

Key	Type	Content
`Item`	String	`PersonalHotspot`
Enabled	Boolean	If `true`, enables Personal Hotspot. If `false`, disables Personal Hotspot. The Personal Hotspot setting is only available on certain carriers.

Note: This query requires the Network Information right.

Wallpaper Sets the Wallpaper

A wallpaper change is a one-time setting that can be changed by the user at will. This command is supported in supervised mode only.

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

Key	Type	Content
`Item`	String	`Wallpaper`
`Image`	Data	A Base64-encoded image to be used for the wallpaper. Images must be in either PNG or JPEG format.
`Where`	Number	Where the wallpaper should be applied. 1—Lock screen 2—Home (icon list) screen 3—Lock and home screens

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.

ApplicationAttributes Sets Or Updates the App Attributes for a Managed Application

To set or update the attributes for a Managed Application, send a `Settings` command with the following dictionary as an entry:

Key	Type	Content
`Item`	String	`ApplicationAttributes`

Key	Type	Content
Identifier	String	The app identifier.
Attributes	Dictionary	Optional. Attributes to be applied to the app. If this member is missing, any existing attributes for the app is removed.

Note: This setting requires the App Management right.

The keys that can appear in the Attributes dictionary are listed below:

Key	Type	Content
VPNUUID	String	Per-App VPN UUID assigned to this app.

Language Sets the Device's User Interface Language

To send an Language command (available only on supervised Apple TV devices running iOS 7—Apple TV software 6.0—and later), the server sends a dictionary containing the following keys:

Key	Type	Content
Item	String	Language
Language	String	The desired language. This language must be one of the languages returned by a Languages query.

Locale Sets the Device's User Interface Locale

To send a Locale command (available only on supervised Apple TV devices running iOS 7—Apple TV software 6.0—and later), the server sends a dictionary containing the following keys:

Key	Type	Content
Item	String	Locale
Language	String	The desired language. This language must be one of the languages returned by a Locales query.

OrganizationInfo Sets Information About the Organization That Is Operating the MDM Server

In iOS 7 and later, to send an `OrganizationInfo` command, the server sends a dictionary containing the following keys:

Key	Type	Content
`Item`	String	`OrganizationInfo`
`OrganizationInfo`	Dictionary	*Optional.* A dictionary, as described below. If this field is missing, the current org info is removed.

The `OrganizationInfo` dictionary can contain the following keys:

Key	Type	Content
`OrganizationName`	String	A string describing the organization that is operating the MDM server. If provided, this name is used in MDM-related alerts.
`OrganizationAddress`	String	*Optional.* The organization's address. You can embed line feed characters (&#10) to insert line breaks.
`OrganizationPhone`	String	*Optional.* A telephone number to contact the organization for support.
`OrganizationEmail`	String	*Optional.* An email address to contact the organization for support.
`OrganizationMagic`	String	*Optional.* A string that uniquely identifies various services that are managed by a single organization.

Note: In OS X 10.9, MDM clients can receive the `OrganizationInfo` command, but only to support the VPP `InviteToProgram` command.

MDMOptions Sets Options Related to the MDM Protocol

To send an `MDMOptions` command (available only in iOS 7 and later), the server sends a dictionary containing the following keys:

Key	Type	Content
`Item`	String	`MDMOptions`
`MDMOptions`	Dictionary	A dictionary, as described below.

The `MDMOptions` dictionary can contain the following keys:

Key	Type	Content
`ActivationLock-AllowedWhile-Supervised`	Boolean	*Optional.* If `true`, a supervised device registers itself with Activation Lock when the user enables Find My iPhone. Defaults to `false`. This setting is ignored on unsupervised devices.

Managed App Configuration and Feedback

In iOS 7 and later, an MDM server can use configuration and feedback dictionaries to communicate with and configure third-party managed apps.

Important: The managed app configuration and feedback dictionaries are stored as unencrypted files. Do not store passwords or private keys in these dictionaries.

The configuration dictionary provides one-way communication from the MDM server to an app. An app can access its (read-only) configuration dictionary by reading the key `com.apple.configuration.managed` using the `NSUserDefaults` class. A managed app can respond to new configurations that arrive while the app is running by observing the `NSUserDefaultsDidChangeNotification` notification.

A managed app can also store feedback information that can be queried over MDM. An app can store new values for this feedback dictionary by setting the `com.apple.feedback.managed` key using the `NSUserDefaults` class. This dictionary can be read or deleted over MDM. An app can respond to the deletion of the feedback dictionary by observing the `NSUserDefaultsDidChangeNotification` notification.

ManagedApplicationConfiguration Retrieves Managed App Configurations

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

Key	Type	Content
`RequestType`	String	`ManagedApplicationConfiguration`
`Identifiers`	Array	Array of managed application identifiers, as strings.

Note: The `ManagedApplicationConfiguration` command requires that the server have the App Management right.

Queries about apps that are not managed are ignored.

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

Key	Type	Content
`ApplicationConfigurations`	Array	An array of dictionaries, one per app.

Each member of the `ApplicationConfigurations` array is a dictionary with the following keys:

Key	Type	Content
`Identifier`	String	`The application's identifier`
`Configuration`	Dictionary	Optional. The current configuration. If the app has no managed configuration, this key is absent.

ApplicationConfiguration Sets Or Updates the App Configuration for a Managed Application

In iOS 7 and later, to set or update the App Configuration for a Managed Application, send a `Settings` command with the following dictionary as an entry:

Key	Type	Content
`Item`	String	`ApplicationConfiguration`
`Identifier`	String	The app identifier.
`Configuration`	Dictionary	Optional. Configuration dictionary to be applied to the app. If this member is missing, any existing managed configuration for the app is removed.

Note: This setting requires the App Management right.

ManagedApplicationAttributes Queries App Attributes

In iOS 7 and later, attributes can be set on managed apps. These attributes can be changed over time.

Key	Type	Content
RequestType	String	ManagedApplicationAttributes
Identifiers	Array	Array of managed application identifiers, as strings.

The device replies with a dictionary that contains the following keys:

Key	Type	Content
ApplicationAttributes	Array	Array of dictionaries.

Each member of the ApplicationAttributes array is a dictionary with the following keys:

Key	Type	Content
Identifier	String	The application's identifier.
Attributes	Dictionary	Optional. The current attributes for the application.

The keys that can appear in the Attributes dictionary are listed below:

Key	Type	Content
VPNUUID	String	Per-App VPN UUID assigned to this app.

ManagedApplicationFeedback Retrieves Managed App Feedback

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

Key	Type	Content
RequestType	String	ManagedApplicationFeedback
Identifiers	Array	Array of managed application identifiers, as strings.
DeleteFeedback	Boolean	Optional. If true, the application's feedback dictionary is deleted after it is read.

Note: The `ManagedApplicationFeedback` command requires that the server have the App Management right.

Queries about apps that are not managed are ignored.

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

Key	Type	Content
`ManagedApplicationFeedback`	Array	An array of dictionaries, one per app.

Each member of the `ApplicationConfigurations` array is a dictionary with the following keys:

Key	Type	Content
`Identifier`	String	`The application's identifier`
`Feedback`	Dictionary	Optional. The current feedback dictionary. If the app has no feedback dictionary, this key is absent.

Support for OS X Requests

Table 10 (page 57) lists the MDM protocol request types that are available for Apple devices that run OS X. The interfaces of these requests to OS X are similar to the iOS interfaces described above.

Table 10 OS X Support for MDM Requests

Command	Min OS	User/Device	Coments
ProfileList	10.7	both	
InstallProfile	10.7	both	
RemoveProfile	10.7	both	
ProvisioningProfileList	10.7	both	Supported, but always returns empty list.
CertificateList	10.7	both	
InstalledApplicationList	10.7	both	
InstallApplication	10.9	user	

Command	Min OS	User/Device	Coments
DeviceInformation	10.7	both	**General:** UDID **DeviceInfo:** OSVersion, BuildVersion, ProductName, Model, ModelName, DeviceCapacity, AvailableDeviceCapacity, DeviceName, SerialNumber. **NetworkInfo:** BluetoothMAC, WiFiMAC, EthernetMAC.
	10.9	both	**General:** added Languages, Locales. **AppManagement:** added iTunesStoreAccountIsActive.
SecurityInfo	10.7	both	Supported, but always returns empty list.
	10.9	device	Returns FileVault state in FDE_Enabled, FDE_HasPersonalRecovery Key, FDE_HasInstitutionalRecoveryKey.
DeviceLock	10.7	device	
EraseDevice	10.7	device	
Restrictions	10.7	both	Supported, but always returns empty list.
InviteToProgram	10.9	both	
Settings	10.9	varies	DeviceName (device), OrganizationInfo (device).

Error Codes

The following sections list the error codes currently returned by iOS and OS X devices. Your software should *not* depend on these values, because they may change in future operating system releases. They are provided solely for informational purposes.

MCProfileErrorDomain

Code	Meaning
1000	Malformed profile

Code	Meaning
1001	Unsupported profile version
1002	Missing required field
1003	Bad data type in field
1004	Bad signature
1005	Empty profile
1006	Cannot decrypt
1007	Non-unique UUIDs
1008	Non-unique payload identifiers
1009	Profile installation failure
1010	Unsupported field value

MCPayloadErrorDomain

Code	Meaning
2000	Malformed Payload
2001	Unsupported payload version
2002	Missing required field
2003	Bad data type in field
2004	Unsupported field value

MCRestrictionsErrorDomain

Code	Meaning
3000	Inconsistent restriction sense (internal error)
3001	Inconsistent value comparison sense (internal error)

MCInstallationErrorDomain

Code	Meaning
4000	Cannot parse profile
4001	Installation failure
4002	Duplicate UUID
4003	Profile not queued for installation
4004	User cancelled installation
4005	Passcode does not comply
4006	Invalid input
4007	Unrecognized file format
4008	Mismatched certificates
4009	Device locked
4010	Updated profile does not have the same identifier
4011	Final profile is not a configuration profile
4012	Profile is not updatable
4013	Update failed
4014	No device identity available
4015	Replacement profile does not contain an MDM payload
4016	(Internal error)
4017	Multiple global HTTPProxy payloads
4018	Multiple APN or Cellular payloads
4019	Multiple App Lock payloads
4000	UI installation prohibited
4001	Profile must be installed non-interactively
4002	Profile must be installed using MDM

Code	Meaning
4003	Unacceptable payload
4004	Profile not found
4005	Invalid supervision

MCPasscodeErrorDomain

Code	Meaning
5000	Passcode too short
5001	Too few unique characters
5002	Too few complex characters
5003	Passcode has repeating characters
5004	Passcode has ascending descending characters
5005	Passcode requires number
5006	Passcode requires alpha characters
5007	Passcode expired
5008	Passcode too recent
5009	(unused)
5010	Device locked
5011	Wrong passcode
5012	(unused)
5013	Cannot clear passcode
5014	Cannot set passcode
5015	Cannot set grace period

MCKeychainErrorDomain

Code	Meaning
6000	Keychain system error
6001	Empty string
6002	Cannot create query

MCEmailErrorDomain

Code	Meaning
7000	Host unreachable
7001	Invalid credentials
7002	Unknown error occurred during validation
7003	SMIME certificate not found
7004	SMIME certificate is bad
7005	IMAP account is misconfigured
7006	POP account is misconfigured
7007	SMTP account is misconfigured

MCWebClipErrorDomain

Code	Meaning
8000	Cannot install Web Clip

MCCertificateErrorDomain

Code	Meaning
9000	Invalid password

Code	Meaning
9001	Too many certificates in a payload
9002	Cannot store certificate
9003	Cannot store WAPI data
9004	Cannot store root certificate
9005	Certificate is malformed
9006	Certificate is not an identity

MCDefaultsErrorDomain

Code	Meaning
10000	Cannot install defaults
10001	Invalid signer

MCAPNErrorDomain

Code	Meaning
11000	Cannot install APN
11000	Custom APN already installed

MCMDMErrorDomain

Code	Meaning
12000	Invalid access rights
12001	Multiple MDM instances
12002	Cannot check in
12003	Invalid challenge response
12004	Invalid push certificate

Code	Meaning
12005	Cannot find certificate
12006	Redirect refused
12007	Not authorized
12008	Malformed request
12009	Invalid replacement profile
12010	Internal inconsistency error
12011	Invalid MDM configuration
12012	MDM replacement mismatch
12013	Profile not managed
12014	Provisioning profile not managed
12015	Cannot get push token
12016	Missing identity
12017	Cannot create escrow keybag
12018	Cannot copy escrow keybag data
12019	Cannot copy escrow secret
12020	Unauthorized by server
12021	Invalid request type
12022	Invalid topic
12023	Could not validate app iD
12024	Could not validate app manifest
12025	App already installed
12026	App already queued
12027	Not an app
12028	Not waiting for redemption

Code	Meaning
12029	App not managed
12030	Invalid URL
12031	App installation disabled
12032	Too many apps in manifest
12033	Invalid manifest
12034	URL is not HTTPS
12035	App cannot be purchased
12036	Cannot remove app in current state
12037	Invalid redemption code
12038	App not managed
12039	(unused)
12040	iTunes store login required
12041	Unknown language code
12042	Unknown locale code

MCWiFiErrorDomain

Code	Meaning
13000	Cannot install
13001	Username required
13002	Password required
13003	Cannot create WiFi configuration
13004	Cannot set up EAP
13005	Cannot set up proxy

MCTunnelErrorDomain

Internal only—do not publish

Code	Meaning
14000	Invalid field
14001	Device locked
14002	Cloud configuration already exists

MCVPNErrorDomain

Code	Meaning
15000	Cannot install VPN
15001	Cannot remove VPN
15002	Cannot lock network configuration
15003	Invalid certificate

MCSubCalErrorDomain

Code	Meaning
16000	Cannot create subscription
16001	No host name
16002	Account not unique

MCCalDAVErrorDomain

Code	Meaning
17000	Cannot create account
17001	No host name
17002	Account not unique

MCDAErrorDomain

Code	Meaning
18000	Unknown error
18001	Host unreachable
18002	Invalid credentials

MCLDAPErrorDomain

Code	Meaning
19000	Cannot create account
19001	No host name
19002	Account not unique

MCCardDAVErrorDomain

Code	Meaning
20000	Cannot create account
20001	No host name
20002	Account not unique

MCEASErrorDomain

Code	Meaning
21000	Cannot get policy from server
21001	Cannot comply with policy from server
21002	Cannot comply with encryption policy from server
21003	No host name
21004	Cannot create account

Code	Meaning
21005	Account not unique
21006	Cannot decrypt certificate
21007	Cannot verify account

MCSCEPErrorDomain

Code	Meaning
22000	Invalid key usage
22001	Cannot generate key pair
22002	Invalid CAResponse
22003	Invalid RAResponse
22004	Unsupported certificate configuration
22005	Network error
22006	Insufficient CACaps
22007	Invalid signed certificate
22008	Cannot create identity
22009	Cannot create temporary identity
22010	Cannot store temporary identity
22011	Cannot generate CSR
22012	Cannot store CACertificate
22013	Invalid PKIOperation response

MCHTTPTransactionErrorDomain

Code	Meaning
23000	Bad identity

Code	Meaning
23001	Bad server response
23002	Invalid server certificate

MCOTAProfilesErrorDomain

Code	Meaning
24000	Cannot create attribute dictionary
24001	Cannot sign attribute dictionary
24002	Bad identity payload
24003	Bad final profile

MCProvisioningProfileErrorDomain

Code	Meaning
25000	Bad profile
25001	Cannot install
25002	Cannot remove

MCDeviceCapabilitiesErrorDomain

Code	Meaning
26000	Block level encryption unsupported
26001	File level encryption unsupported

MCSettingsErrorDomain

Code	Meaning
28000	Unknown item

Code	Meaning
28001	Bad wallpaper image
28002	Cannot set wallpaper

MCChaperoneErrorDomain

Code	Meaning
29000	Device not supervised
29003	Bad certificate data

MCStoreErrorDomain

Code	Meaning
30000	Authentication failed
30001	Timed out

MCGlobalHTTPProxyErrorDomain

Code	Meaning
31000	Cannot apply credential
31001	Cannot apply settings

MCSingleAppErrorDomain

Code	Meaning
32000	Too many apps

MCSSOErrorDomain

Code	Meaning
34000	Invalid app identifier match pattern
34001	Invalid URL match pattern
34002	Kerberos principal name missing
34003	Kerberos principal name invalid

MCFontErrorDomain

Code	Meaning
35000	Invalid font data
35001	Failed font installation
35002	Multiple fonts in a single payload

MCCellularErrorDomain

Code	Meaning
36000	Cellular already configured
36000	(internal error)

Device Enrollment Program

In iOS 7 and later and OS X v10.9 and later, the Device Enrollment Program helps to address the mass configuration needs of organizations purchasing and deploying devices in large quantities, without the need for factory customization or pre-configuration of devices prior to deployment.

A device enrolled in the Device Enrollment Program prompts the user to enroll in MDM during the initial device setup process. Additionally, devices enrolled in the program can be supervised over the air. Although Apple's servers store information about the device's participation in this program, the MDM profile and login challenge are served by the organization's server.

The cloud service API provides profile management and mapping. With this API, you can obtain a list of devices, obtain information about those devices, and associate MDM enrollment profiles with those devices.

Device Management Workflow

A typical MDM device management workflow contains the following steps:

1. Set up an account for your MDM server if you have not already done so.
2. Use the Fetch Devices endpoint to obtain devices associated with the MDM server's account.

   > **Note:** Your server should periodically use the Sync Devices endpoint to obtain updated information about existing devices and new devices.

3. Assign a profile to the device. You can do this in one of the following ways:
   - Use the Define Profile endpoint to create a new MDM server profile and associate it with one or more devices.
   - Use the Assign Profile endpoint to associate an existing MDM server profile with one or more devices.
4. Remove the profile from the device when appropriate by using the Remove Profile endpoint.
5. When your organization is no longer responsible for the device, use the Disown Devices endpoint to permanently disassociate it from your MDM server.

Authentication and Authorization

To obtain OAuth access credentials for a server, download a server token file while the server is being created on the portal. The token file will contain a JSON object similar to the one shown below:

```
{
  "consumer_key": "CK_00fadb3d36c6094cf479838455321b7c",
  "consumer_secret": "CS_5fb17e5676db0cf875211937e5166d0f662ea1f9",
 "access_token": "AT_021092790220e03b641fd6f07d7face7894211d521fd8bef09c30137392",
  "access_secret": "AS_837c228d968ff303837086a5a54be645314ef755"
  "access_token_expiry": "2013-09-09T02:24:28Z"
}
```

Each service request to the MDM enrollment service must include an `X-ADM-Auth-Session` header.

If the request does not have a valid `X-ADM-Auth-Session` header, or the auth token has expired, the server returns an HTTP `401 Unauthorized` error.

```
HTTP/1.1 401 Unauthorized
Content-Type: text/plain;Charset=UTF8
Content-Length: 9
WWW-Authenticate: ADM-Auth-Token
Date: Thu, 31 May 2012 21:23:37 GMT
Connection: close

UNAUTHORIZED
```

Requesting a New Session Authorization Token

A new `X-ADM-Auth-Session` can be requested by using the https://mdmenrollment.apple.com/session endpoint. This endpoint supports the OAuth 1.0a protocol for accessing protected resources. When you sign up for the Device Enrollment Program, your server is assigned four pieces of information:

- consumer_key
- consumer_secret
- access_token
- access_secret

Your OAuth request must provide these pieces of information along with a timestamp (in seconds since January 1, 1970 00:00:00 GMT) and a cryptographically random nonce that must be unique for all requests made with a given timestamp. The server's time should be synchronized using time.apple.com or another trusted NTP provider.

The request must be signed using HMAC-SHA1, as described in http://oauth.net/core/1.0a/#signing_process.

For example:

```
GET /session HTTP/1.1
Authorization: OAuth realm="ADM",
       oauth_consumer_key="CK_00fadb3d36c6094cf479838455321b7c",
      oauth_token="AT_O21092790220e03b641fd6f07d7face7894211d521fd8bef09c30137392",
       oauth_signature_method="HMAC-SHA1",
       oauth_signature="wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%3D",
       oauth_timestamp="137131200",
       oauth_nonce="4572616e48616d6d65724c61686176",
       oauth_version="1.0"
```

For more information about the OAuth specification, see http://oauth.net/core/1.0a/.

Response Payload

The token service validates the request and replies with a JSON payload containing a single key, `auth_session_token`, that contains the new `X-ADM-Auth-Session` token. For example:

```
HTTP/1.1 200 OK
Date: Thu, 28 Feb 2013 02:24:28 GMT
Content-Type: application/json;charset=UTF8
Content-Length: 47
Connection: close

{
    "auth_session_token" : "87a235815b8d6661ac73329f75815b8d6661ac73329f815"
}
```

> **Note:** The Device Enrollment Program service periodically issues a new `X-ADM-Auth-Session` in its response to a service call; the MDM server can use this new header value for any subsequent calls.

After a period of time, this token expires, and the service returns a `401` error code. At this point, the MDM server must obtain a new session token from the https://mdmenrollment.apple.com/session endpoint.

Authentication Error Codes

An authentication error commonly results in either a `400`, `401`, or `403` error code.

An HTTP `400 Bad Request` error indicates one of the following:

Unsupported oauth parameters

Unsupported signature method

Missing required authorization parameter

Duplicated OAuth Protocol Parameter

An HTTP `401 Unauthorized` error indicates one of the following:

Invalid Consumer Key

Invalid or expired Token

Invalid signature

Invalid or already-used nonce

An HTTP `403 Forbidden` error indicates one of the following:

The MDM server does not have access to perform the specific request or the MDM server's consumer key or token does not have authorization to perform the specific request. In this case, the request body contains `ACCESS_DENIED`.

The organization has not accepted latest Terms and Conditions of the program. In this case, the request body contains `T_C_NOT_SIGNED`.

For example, the following is the response when the MDM server is not authorized to perform a given request.

```
HTTP/1.1 403 Forbidden
Content-Type: text/plain;Charset=UTF8
Content-Length: 13
Date: Thu, 31 May 2012 21:23:57 GMT
```

```
Connection: close


ACCESS_DENIED
```

Web Services

This section lists the services that Apple's servers provide to your MDM server. Except where otherwise specified, all requests must be sent with the following HTTP headers:

Header	Value
`User-Agent`	Your MDM server's user agent string.
`X-Server-Protocol-Version`	Integer, 1 currently.
`X-ADM-Auth-Session`	An authentication token value. This header may be omitted when requesting an authentication token.
`Content-Type`	`application/json;charset=UTF8` This header may be omitted for requests that do not include a request body.

For example:

```
GET /account HTTP/1.1
User-Agent: ProfileManager-1.0
X-Server-Protocol-Version: 1
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
```

The sections below describe the available commands.

Account Details

Each MDM server must be registered with Apple. This endpoint provides details about the server entity to identify it uniquely throughout your organization. Each server can be identified by either its system-generated UUID or by a user-provided name assigned by one of the organization's users. Both the UUID and server name must be unique within your organization.

URL

`https://mdmenrollment.apple.com/account`

Query Type

GET

Request Body

This request does not require a request body.

For example, your MDM server might make the following request:

```
GET /account HTTP/1.1
User-Agent: ProfileManager-1.9
Content-Length: 0
X-Server-Protocol-Version: 1
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`server_name`	An identifiable name for the MDM server
`server_uuid`	A system-generated server identifier
`facilitator_id`	Apple ID of the person who generated the current tokens that are in use
`org_name`	The organization name
`org_email`	The organization email address
`org_phone`	The organization phone
`org_address`	The organization address

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 28 Feb 2013 02:24:28 GMT
```

```
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 640
Connection: close

{
     "server_name" : "IT Department Server",
     "server_uuid" : "677cab70-fe18-11e2-b778-0800200c9a66",
     "facilitator_id" : "facilitator1@example.com",
     "org_name" : "Sample Inc",
     "org_phone" : "111-222-3333",
     "org_email" : "orgadmin@example.com",
     "org_address": "12 Infinite Loop, Cupertino, California 95014"
}
```

Fetch Devices

This request fetches a list of all devices that are assigned to this MDM server at the time of the request. This service should be used for loading an initial list of devices into the MDM server's data store. Once the list of devices is loaded, device sync requests should be used to synchronize the list with any further changes.

This request provides a limited number of entries per request, using cursors to provide position information across requests.

> **Note:** The server accepts only the `application/json` content type for this request.

URL

`https://mdmenrollment.apple.com/server/devices`

Query Type

POST

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
`cursor`	*Optional.* A hex string that represents the starting position for a request. This is used for retrieving the list of devices that have been added or removed since a previous request. On the initial request, this should be omitted.
`limit`	*Optional.* The maximum number of entries to return. The default value is 100, and the maximum value is 1000.

For example, your MDM server might make the following request:

```
POST /server/devices HTTP/1.1
User-Agent:ProfileManager-1.9
X-Server-Protocol-Version:1
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
"limit": 100,
"cursor": "1ac73329f75815"
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`cursor`	Indicates when this request was processed by the enrollment server. The MDM server can use this value in future requests if it wants to retrieve only records added or removed since this request.
`devices`	An array of dictionaries providing information about devices, sorted in chronological order of enrollment from oldest to most recent.
`fetched_until`	A timestamp indicating the progress of the device fetch request, in ISO 8601 format.
`more_to_follow`	A Boolean value that indicates whether the request's limit and cursor values resulted in only a partial list of devices. If `true`, the MDM server should then make another request (starting from the newly returned cursor) to obtain additional records.

Each device dictionary contains the following keys:

Key	Value
`serial_number`	The device's serial number (string).
`model`	The model name (string).
`description`	A description of the device (string).
`color`	The color of the device (string).
`asset_tag`	The device's asset tag (string), if provided by Apple.
`profile_status`	The status of profile installation—either `"empty"`, `"assigned"`, or `"pushed"`.
`profile_uuid`	The unique ID of the assigned profile.
`profile_assign_time`	A time stamp in ISO 8601 format indicating when a profile was assigned to the device. If a profile has not been assigned, this field may be absent.
`profile_push_time`	A time stamp in ISO 8601 format indicating when a profile was pushed to the device. If a profile has not been pushed, this field may be absent.
`device_assigned_date`	A time stamp in ISO 8601 format indicating when the device was enrolled in the Device Enrollment Program.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 02:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 640
Connection: Keep-Alive

{
       "devices" : [
             {
                 "serial_number" : "C8TJ500QF1MN"
                 "model" : "IPAD",
```

```
                "description" : "IPAD WI-FI 16GB",
                "color" : "black",
                "asset_tag" : "304214",
                "profile_status" : "empty",
                "device_assigned_date" : "2013-04-05T14:30:00Z"
            },
            {
                "serial_number" : "C8TJ500QF1MN"
                "model" : "IPAD",
                "description" : "IPAD WI-FI 16GB",
                "color" : "white",
                "profile_status" : "assigned",
                "profile_uuid" : "88fc4e378fea4021a94b2d7268fbf767",
                "profile_assign_time" : "2013-01-01T00:00:00Z",
                "device_assigned_date" : "2013-04-05T15:30:00Z"
            }
        ]
        "fetched_until" : "2013-05-09T02:24:28Z",
        "cursor" : "1ac73329f75815",
        "more_to_follow" : "false"
}
```


Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error with `INVALID_CURSOR` in the response body indicates that an invalid cursor value was provided.
- A `400` error with `EXHAUSTED_CURSOR` in the response body indicates that the cursor had returned all devices in previous calls.

Sync Devices

The sync service depends on a cursor returned by the fetch device service. It returns a list of all modifications (additions or deletions) since the specified cursor. The cursor passed to this endpoint should not be older than 7 days.

This service may return the same device more than once. You must resolve duplicates by matching on the device serial number and the `op_type` and `op_date` fields.

> **Note:** The server accepts only the `application/json` content type for this request.

URL

`https://mdmenrollment.apple.com/devices/sync`

Query Type

POST

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
`cursor`	A hex string returned by a previous request that represents the starting position for a request. The request returns results that describe any changes or additions to devices that happened after this starting position.
`limit`	*Optional.* The maximum number of entries to return. The default value is 100, and the maximum value is 1000.

For example, your MDM server might make the following request:

```
POST /devices/sync HTTP/1.1
User-Agent:ProfileManager-1.9
X-Server-Protocol-Version:1
Content-Type: application/json;charset=UTF8
Content-Length: 50
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
    "cursor": "1ac73329f75815",
    "limit" : 200
```

```
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`cursor`	Indicates when this request was processed by the server. The MDM server can use this value in future requests if it wants to retrieve only records added or removed since this request.
`more_to_follow`	Indicates that the request's limit and cursor values resulted in only a partial list of devices. The MDM server should immediately make another request (starting from the newly returned cursor) to obtain additional records.
`devices`	An array of dictionaries providing information about devices, sorted in chronological order by the time stamp of the operation performed on the device.
`fetched_until`	A date stamp indicating the progress of the device fetch request, in ISO 8601 format.

Each device dictionary contains some of the following keys:

Key	Value
`serial_number`	The device's serial number (string).
`model`	The model name (string).
`description`	A description of the device (string).
`color`	The color of the device (string).
`asset_tag`	The device's asset tag (string).
`profile_status`	The status of profile installation—either `"empty"`, `"assigned"`, or `"pushed"`.
`profile_uuid`	The unique ID of the assigned profile.
`profile_assign_time`	A time stamp in ISO 8601 format indicating when a profile was assigned to the device.
`profile_push_time`	A time stamp in ISO 8601 format indicating when a profile was pushed to the device.

Key	Value
op_type	Indicates whether the device was added (assigned to the MDM server), updated, or deleted. Contains one of the following strings: added, updated, or deleted.
op_date	A time stamp in ISO 8601 format indicating when the device was added, updated, or deleted. If the value of op_type is added, this is the same as device_assigned_date.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 03:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 640
Connection: Keep-Alive

{
        "devices" : [
                {
                    "serial_number" : "C8TJ500QF1MN"
                    "model" : "IPAD",
                    "color" :    "black",
                    "description" : "IPAD WI-FI 16GB",
                    "asset_tag" : "304214",
                    "profile_status" : "empty",
                    "op_type" : "added",
                    "op_date" : "2013-05-09T14:30:00Z"
                },
                {
                    "serial_number" : "C8TJ500QF1MN"
                    "model" : "IPAD",
                    "color" :    "white",
                    "description" : "IPAD WI-FI 16GB",
                    "op_type" : "deleted",
```

```
                    "op_date" : "2013-05-09T14:30:00Z"
                }
            ],
        "more_to_follow" : false,
        "cursor" : "2ac73329f75815"
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error with `CURSOR_REQUIRED` in the response body indicates that no cursor value was provided.
- A `400` error with `INVALID_CURSOR` in the response body indicates that an invalid cursor value was provided.
- A `400` error with `EXPIRED_CURSOR` in the response body indicates that the provided cursor is older than 7 days.

Device Details

Returns information about an array of devices.

> **Note:** The server accepts only the `application/json` content type for this request.

URL

`https://mdmenrollment.apple.com/devices`

Query Type

POST

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
`devices`	An array of strings containing device serial numbers.

For example, your MDM server might make the following request:

```
POST /devices HTTP/1.1
User-Agent:ProfileManager-1.0
X-Server-Protocol-Version:1
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
    "devices":["C8TJ500QF1MN", "B7CJ500QF1MA"]
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary of dictionaries. The outer dictionary keys are the serial numbers from the original request. Each value is a dictionary with the following keys:

Key	Value
`response_status`	A string indicating whether a particular device's data could be retrieved—either `SUCCESS` or `NOT_FOUND`.
`model`	The model name (string).
`description`	A description of the device (string).
`color`	The color of the device (string).
`asset_tag`	The device's asset tag (string).
`device_assigned_by`	The Apple ID of the person who assigned the device.
`device_assigned_date`	A time stamp in ISO 8601 format indicating when the device was assigned to the MDM server.
`profile_status`	The status of profile installation—either `"empty"`, `"assigned"`, or `"pushed"`. If `"empty"`, no other profile fields are present.
`profile_uuid`	The unique ID of the assigned profile.
`profile_assign_time`	A time stamp in ISO 8601 format indicating when a profile was assigned to the device.

Key	Value
profile_push_time	A time stamp in ISO 8601 format indicating when a profile was pushed to the device.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 03:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 259
Connection: Keep-Alive

{
  "devices":
  {
    "C8TJ500QF1MN" :
    {
      "response_status" : "SUCCESS",
      "model" : "IPAD",
      "description" : "IPAD WI-FI 16GB",
      "color": "BLACK",
      "asset_tag" : "304214",
      "device_assigned_by" : "facilitator1@sampleinc.com"
      "device_assigned_date" : "2013-01-01T14:30:00Z",
      "profile_uuid" : "88fc4e378fea4021a94b2d7268fbf767",
      "profile_assign_time" : "2013-01-01T00:00:00Z",
      "profile_push_time" : "2013-02-01T00:00:00Z"
    },
    "B7CJ500QF1MA" : {
      "response_status" : "NOT_FOUND"
    }
   }
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `200` error with `NOT_FOUND` in the response body indicates that the specified device is not accessible by the MDM server.
- A `400` error with `DEVICE_ID_REQUIRED` in the response body indicates that the request did not contain any devices.

Disown Devices

Tells Apple's servers that your organization no longer owns one or more devices.

Warning: Disowning a device is a permanent action. After a short grace period, a disowned device cannot be reassigned to an MDM server in your organization.

Note: The server accepts only the `application/json` content type for this request.

URL

`https://mdmenrollment.apple.com/devices/disown`

Query Type

POST

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
`devices`	Array of strings containing device serial numbers.

For example, your MDM server might make the following request:

```
POST /devices/disown HTTP/1.1
User-Agent:ProfileManager-1.0
X-Server-Protocol-Version:1
```

```
Content-Type: application/json;charset=UTF8
Content-Length: 30
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
    "devices":["C8TJ500QF1MN", "B7CJ500QF1MA"]
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`devices`	A dictionary of devices. Each key in this dictionary is the serial number of a device in the original request. Each value is one of the following values: • `SUCCESS`—Device was successfully disowned. • `NOT_ACCESSIBLE`—A device with the specified ID was not accessible by this MDM server. • `FAILED`—Disowning the device failed for an unexpected reason. If three retries fail, the user should contact Apple support. If no devices were provided in the original request, this dictionary may be absent.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 03:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 160
Connection: Keep-Alive

{
    "devices": {
        "C8TJ500QF1MN":"SUCCESS",
        "B7CJ500QF1MA":"NOT_ACCESSIBLE"
```

```
    }
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error code with `DEVICE_ID_REQUIRED` in the response body indicates that no device IDs (serial numbers) were provided.

Define Profile

Tells Apple's servers about a profile that can then be assigned to specific devices. This command provides information about the MDM server that is assigned to manage one or more devices, information about the host that the managed devices can pair with, and various attributes that control the MDM association behavior of the device.

URL

`https://mdmenrollment.apple.com/profile`

Query Type

POST

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
`profile_name`	String. A human-readable name for the profile.
`url`	String. The URL of the MDM server.
`allow_pairing`	*Optional.* Boolean. Default is `true`.
`is_supervised`	*Optional.* Boolean. If `true`, the device must be supervised. Defaults to `false`.

Key	Value
`is_mandatory`	*Optional.* Boolean. If `true`, the user may not skip applying the profile returned by the MDM server. Default is `false`.
`is_mdm_removable`	If `false`, the MDM payload delivered by the configuration URL cannot be removed by the user using the UI on the device—that is, the MDM payload is locked onto the device. Defaults to `true`.
`support_phone_number`	String. A support phone number for the organization.
`org_magic`	A string that uniquely identifies various services that are managed by a single organization.
`anchor_certs`	*Optional.* Array of strings. Each string should contain a DER-encoded certificate converted to Base64 encoding. If provided, these certificates are used as trusted anchor certificates when evaluating the trust of the connection to the MDM server url. Otherwise, the built-in root certificates are used.
`supervising_host_certs`	*Optional.* Array of strings. Each string contains a DER-encoded certificate converted to Base64 encoding. If provided, the device will continue to pair with a host possessing one of these certificates even when `allow_pairing` is set to `false`.
`skip_setup_items`	*Optional.* Array of strings. A list of setup panes to skip. The array may contain one or more of the following strings: • `Passcode`—Hides and disables the passcode pane • `Location`—Disables Location Services • `Restore`—Disables restoring from backup • `AppleID`—Disables signing in to Apple ID and iCloud • `TOS`—Skips Terms and Conditions • `Siri`—Disables Siri • `Diagnostics`—Disables automatically sending diagnostic information
`department`	String. The user-defined department or location name.
`devices`	Array of strings containing device serial numbers. (May be empty.)

For example, your MDM server might make the following request:

```
POST /profile HTTP/1.1
User-Agent:ProfileManager-1.0
X-Server-Protocol-Version:1
Content-Type: application/json;charset=UTF8
Content-Length: 350
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
   "profile_name": "Test Profile",
   "url":"https://mdm.acmeinc.com/getconfig",
   "is_supervised":false,
   "allow_pairing":true,
   "is_mandatory":false,
    "is_mdm_removable":    false,
   "department": "IT Department",
   "org_magic": "913FABBB-0032-4E13-9966-D6BBAC900331",
   "support_phone_number": "1-555-555-5555",
   "anchor_certs":[
      "MIICkDCCAfmgAwIBAgIJAOAeuvyohALaMA0GCSqGSIb3DQEBBQUAMGExCzAJBgNVBAYT..."
   ],
   "supervising_host_certs:[
        "AlVTMQswCQYDVQQIDAJDQTESMBAGA1UEBwwJQ3VwZXJ0aW5vMRowGAYDVQQKDBFB…"
    ],
   "skip_setup_items":[
      "Location",
      "Restore",
      "AppleID",
      "TOS",
      "Siri",
      "Diagnostics"
   ],
    "devices":["C8TJ500QF1MN", "B7CJ500QF1MA"]
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`profile_uuid`	The profile's UUID (hex string).
`devices`	A dictionary of devices. Each key in this dictionary is the serial number of a device in the original request. Each value is one of the following strings: • `SUCCESS`—Profile was mapped to the device. • `NOT_ACCESSIBLE`—A device with the specified serial number was not accessible by this server. • `FAILED`—Assigning the profile failed for an unexpected reason. If three retries fail, the user should contact Apple support.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 03:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 160
Connection: Keep-Alive

{
    "profile_uuid": "88fc4e378fea4021a94b2d7268fbf767",
    "devices": {
          "C8TJ500QF1MN":"SUCCESS",
          "B7CJ500QF1MA":"NOT_ACCESSIBLE"
    }
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error code with `CONFIG_URL_REQUIRED` in the response body indicates that the MDM server URL is missing in the profile.
- A `400` error code with `CONFIG_NAME_REQUIRED` in the response body indicates that the configuration name is missing in the profile.
- A `400` error code with `DEPARTMENT_REQUIRED` in the response body indicates that department is missing in the profile.
- A `400` error code with `SUPPORT_PHONE_REQUIRED` in the response body indicates support phone is missing in the profile.

Assign Profile

Tells Apple's severs that the specified devices should use a particular profile defined by the "Define Profile" (page 90) command.

URL

`https://mdmenrollment.apple.com/profile/devices`

Query Type

PUT

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
`profile_uuid`	The UUID (string) for the profile that you want to assign to the specified devices. This UUID was returned by a previous Define Profile request.
`devices`	Array of strings containing device serial numbers. An empty array is considered a no-op.

For example, your MDM server might make the following request:

```
PUT /profile/devices HTTP/1.1
User-Agent:ProfileManager-1.0
X-Server-Protocol-Version:1
Content-Type: application/json;charset=UTF8
```

```
Content-Length: 38
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
   "profile_uuid": "88fc4e378fea4021a94b2d7268fbf767",
    "devices":["C8TJ500QF1MN", "B7CJ500QF1MA"]
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`profile_uuid`	The profile's UUID (string).
`devices`	A dictionary of devices. Each key in this dictionary is the serial number of a device in the original request. Each value is a string with one of the following values: • `SUCCESS`—Profile was mapped to the device. • `NOT_ACCESSIBLE`—A device with the specified ID was not accessible by this MDM server. • `FAILED`—Assigning the profile failed for an unexpected reason. If three retries fail, the user should contact Apple support.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 03:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 160
Connection: Keep-Alive

{
    "profile_uuid": "88fc4e378fea4021a94b2d7268fbf767",
    "devices": {
                "C8TJ500QF1MN":"SUCCESS",
```

```
                "B7CJ500QF1MA":"NOT_ACCESSIBLE"
            }
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error with `DEVICE_ID_REQUIRED` in the body of the response indicates that the request did not contain any device IDs.
- A `400` error with `PROFILE_UUID_REQUIRED` in the body of the response indicates that the request did not contain a profile ID.
- A `404` error with `PROFILE_NOT_FOUND` in the body of the response indicates that the profile with the specified UUID could not be found.

Fetch Profile

Returns information about a profile.

URL

`https://mdmenrollment.apple.com/profile`

Query Type

GET

Request Query

The query string should contain the following keys:

Key	Value
`profile_uuid`	The UUID of a profile.

For example, your MDM server might make the following request:

```
GET /profile?profile_uuid=3dd2ccafe97bf07130fe3c908a92c870 HTTP/1.1
User-Agent:ProfileManager-1.0
```

```
X-Server-Protocol-Version:1
Content-Length: 0
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
`profile_name`	String. A human-readable name for the profile.
`url`	String. The URL of the MDM server.
`allow_pairing`	*Optional.* Boolean. Default is `true`.
`is_supervised`	*Optional.* Boolean. If `true`, the device must be supervised. Defaults to `false`.
`is_mandatory`	*Optional.* Boolean. If `true`, the user may not skip applying the profile returned by the MDM server. Default is `false`.
`is_mdm_removable`	If `false`, the MDM payload delivered by the configuration URL cannot be removed by the user using the UI on the device—that is, the MDM payload is locked onto the device. Defaults to `true`.
`support_phone_number`	String. A support phone number for the organization.
`org_magic`	A string that uniquely identifies various services that are managed by a single organization.
`anchor_certs`	*Optional.* Array of strings. Each string should contain a DER-encoded certificate converted to Base64 encoding. If provided, these certificates are used as trusted anchor certificates when evaluating the trust of the connection to the MDM server url. Otherwise, the built-in root certificates are used.
`supervising_host_certs`	*Optional.* Array of strings. Each string contains a DER-encoded certificate converted to Base64 encoding. If provided, the device will continue to pair with a host possessing one of these certificates even when `allow_pairing` is set to `false`.

Key	Value
skip_setup_items	*Optional.* Array of strings. A list of setup panes to skip. The array may contain one or more of the following strings: • Location—Disables Location Services • Restore—Disables restoring from backup • AppleID—Disables signing in to Apple ID and iCloud • TOS—Skips Terms and Conditions • Siri—Disables Siri • Diagnostics—Disables automatically sending diagnostic information
department	The user-defined department or location name.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 28 Feb 2013 02:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 160
Connection: Keep-Alive

{
   "profile_uuid": "88fc4e378fea4021a94b2d7268fbf767",
   "profile_name": "Test Profile",
   "url":"https://mdm.acmeinc.com/getconfig",
   "is_supervised":false,
   "allow_pairing":true,
   "is_mandatory":false,
    "is_mdm_removable":    false,
   "department": "IT Department",
   "org_magic": "913FABBB-0032-4E13-9966-D6BBAC900331",
   "support_phone_number": "1-555-555-5555",
   "anchor_certs":[
```

```
      "MIICkDCCAfmgAwIBAgIJAOAeuvyohALaMA0GCSqGSIb3DQEBBQUAMGExCzAJBgNVBAYT..."
   ],
   "supervising_host_certs:[
        "AlVTMQswCQYDVQQIDAJDQTESMBAGA1UEBwwJQ3VwZXJ0aW5vMRowGAYDVQQKDBFB…"
    ],
   "skip_setup_items":[
      "Location",
      "Restore",
      "AppleID",
      "TOS",
      "Siri",
      "Diagnostics"
   ]
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error with `PROFILE_UUID_REQUIRED` in the body of the response indicates that the request did not contain a profile UUID.
- A `404` error with `PROFILE_NOT_FOUND` in the body of the response indicates that a profile cannot be found for the requested profile UUID.

Request to a Configuration URL

If a `ConfigurationURL` is provided, the device will make an HTTPS POST call to the URL. The request will have a Content-Type of `application/pkcs7-signature`. The following dictionary will be sent as the body of the request. The dictionary will be encoded as an XML plist and then CMS-signed and DER-encoded:

Field	Type	Content
UDID	String	The device's UDID.
SERIAL	String	The device's serial number.
PRODUCT	String	The device's product type, e.g. `iPhone5,1`.
VERSION	String	The OS version installed on the device, e.g. 7A182.

Field	Type	Content
IMEI	String	The device's IMEI (if available).
MEID	String	The device's MEID (if available).
LANGUAGE	String	The user's currently-selected language, e.g. `en`

The plist will be CMS-signed with the device identity certificate. The device's certificate and all necessary intermediate certificates will be included. The certificate chain should validate against the Apple Root CA.

The server may respond with a 401 (Unauthorized) status message to prompt the user for a login. If this response is sent, the WWW-Authenticate header must contain the `Digest` authentication method. When the user enters a username and password, the request is retried with the appropriate Authorization header.

If a 401 status is sent, the content of the response will be shown above the prompt for the username and password. If the content is empty, then a default message will be displayed.

The server may respond with a 200 (OK) status to indicate a successful retrieval of the configuration profile. The configuration profile containing the MDM payload and one or more SCEP or certificate payloads must be included in the message body. The profile may be encrypted against the certificate used to sign the dictionary in the request.

Remove Profile

Removes profile mapping from the list of devices from Apple's servers. After this call, the devices in the list will have no profiles associated with them. However, if those devices have already obtained the profile, this has no effect until the device is wiped and activated again.

URL

`https://mdmenrollment.apple.com/profile/devices`

Query Type

DELETE

Request Body

The request body should contain a JSON dictionary with the following keys:

Key	Value
devices	Array of strings containing device serial numbers.

For example, your MDM server might make the following request:

```
DELETE /profile/devices HTTP/1.1
User-Agent:ProfileManager-1.0
X-Server-Protocol-Version:1
Content-Type: application/json;charset=UTF8
Content-Length: 35
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815

{
    "devices":["C8TJ500QF1MN", "B7CJ500QF1MA"]
}
```

Response Body

In response, the MDM enrollment service returns a JSON dictionary with the following keys:

Key	Value
devices	A dictionary of devices. Each key in this dictionary is the serial number of a device in the original request. Each value in this dictionary is one of the following strings: • SUCCESS—Profile was removed from the device. • NOT_ACCESSIBLE—A device with the specified serial number was not found. • FAILED—Removing the profile failed for an unexpected reason. If three retries fail, the user should contact Apple support.

For example, the server might send a response that looks like this:

```
HTTP/1.1 200 OK
Date: Thu, 9 May 2013 03:24:28 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: 160
```

```
Connection: Keep-Alive


{
    "devices": {
                   "C8TJ500QF1MN":"SUCCESS",
                   "B7CJ500QF1MA":"NOT_ACCESSIBLE"
               }
}
```

Request-Specific Errors

In addition to the standard errors listed in "Common Error Codes" (page 102), this request can return the following errors:

- A `400` error with `DEVICE_ID_REQUIRED` in the body of the response indicates that the request did not contain any device serial numbers.

.

Common Error Codes

If the request could not be validated, the server returns one of the following errors.

- An HTTP `400` error with `MALFORMED_REQUEST_BODY` in the response body indicates that the request body was not valid JSON.
- An HTTP `401` error with `UNAUTHORIZED` in the response body indicates that the authentication token has expired. This error indicates that the MDM server should obtain a new auth token from the https://mdmenrollment.apple.com/session endpoint.
- An HTTP `403` error with `FORBIDDEN` in the response body indicates that the authentication token is invalid.
- An HTTP `405` error means that the method (query type) is not valid.

For example, the following is the response when an authentication token has expired.

```
HTTP/1.1 401 Unauthorized
Content-Type: text/plain;Charset=UTF8
Content-Length: 12
```

```
Date: Thu, 31 May 2012 21:23:57 GMT
Connection: close

UNAUTHORIZED
```

> **Note:** The Device Enrollment Program service periodically issues a new `X-ADM-Auth-Session` in its response to a service call; the MDM server can use this new header value for any subsequent calls.

After a period of extended inactivity, this token expires, and the MDM server must obtain a new auth token from the https://mdmenrollment.apple.com/session endpoint.

All responses may return a new `X-ADM-Auth-Session` token, which the MDM server should use in subsequent requests.

VPP App Assignment

In iOS 7 and later or OS X v10.9 and later, VPP App Assignment allows an organization to assign apps to users. At a later date, if a user no longer needs an app, you can reclaim the app license and assign it to a different user.

The Volume Purchase Program provides a number of web services that MDM servers can use to associate volume purchases with a particular user. This chapter describes those services.

Usage

Service Request URL

The service URL has the form of:

https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/<serviceName>

It is recommended that you obtain the service URLs from the `VPPServiceConfigSrv` service rather than using hard-coded values in the client. All service URLs are subject to change except for the `VPPServiceConfigSrv` URL.

If an error with error code `9617` (URL has moved) is received in the response of any web service request, a `VPPServiceConfigSrv` request should be made to refresh the service URLs.

Providing Parameters

Parameters to the service requests should be provided as a JSON string in the request body, and the `Content-Type` header value should contain `application/json`.

The value of a parameter can be in primitive type or string type. When the web services receive input parameters, all primitive types are converted to string type first before they are parsed into primitive types as required by the specific parameter. For example, 'licenseId' requires a long type; the input in JSON format can be either `{"licenseId":1}` or `{"licenseId":"1"}`. The responses of the services use primitive type for non-string values.

Authentication

All services except `VPPServiceConfigSrv` requires an `sToken` parameter to authenticate the client user. This parameter takes a secret token (in string format). A Program Facilitator can obtain such a token by logging into the appropriate VPP website.

- For Education customers: https://volume.itunes.apple.com/
- For Business customers: https://vpp.itunes.apple.com/

In the Account Summary page, click on the Download button to generate and download a text file containing the new token. Each token is valid for one year from the time you generate it.

The MDM server should store the user's token along with its other private, protected properties, and should send this token value in the `sToken` field of all VPP requests described in this chapter.

The `sToken` blob itself is a JSON object in Base64 encoding. When decoded, the resulting JSON object contains three fields: `token`, `expDate`, and `orgName`. For example, the following is an `sToken` value (with line breaks inserted):

```
eyJ0b2tlbiI6InQxWG9VenBMRXRwZGxhK25zeENkd3JjdDBS
andkaWN0aGRreW5STW05VVAyc2hSYTBMUnVGcVpQM0pLQmJU
TWxDSE42ajNta1R6WVlQbVVkVXJXV2x3PT0iLCJleHBEYXRl
IjoiMjAxNC0wOC0xNVQxODoxMzo1Mi0wNzAwIiwib3JnTmFt
ZSI6Ik9SRy4yMDA5MDcxNjAwIn0=
```

After Base64 decoding, this is the JSON string (with line breaks inserted):

```
{"token":"t1XoUzpLEtpdla+nsxCdwrct0RjwdicNhdkynRMm9UP
2shRa0LRuFqZP3JKBbTMlCHN6j3mkTzYYPmUdUrWWlw==",
"expDate":"2014-08-15T18:13:52-0700",
"orgName":"ORG.2009071600"}
```

The `expDate` field contains the expiration date of the token in ISO 8601 format. The `orgName` field contains the name of the organization for which the token is issued.

Service Response

Response content is in JSON format.

As a convention, fields with `null` values are not included in the response. For example, the user object has an `email` field that is optional. The following example doesn't have the `email` field in the user object, so the `email` field value is `null`.

```
"user":{
   "userId":1,
   "clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8",
   "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc=",
   "status":"Associated",
   "licenses":[
      {
         "licenseId":2,
         "adamId":408709785,
         "productTypeId":7,
         "pricingParam":"STDQ",
         "productTypeName":"Software",
         "isIrrevocable":false
      },
      {
         "licenseId":4,
         "adamId":497799835,
         "productTypeId":7,
         "pricingParam":"STDQ",
         "productTypeName":"Software",
         "isIrrevocable":false
      }
   ]
}
```

Note the licenses associated with the user are returned as an array. If the user doesn't have any license, the "licenses" field will not show up. The license object in this context is a subfield of the user object. To avoid a cyclic reference, the user object is not included in the license object. But if the license is the top object returned, it includes a `user` object with `id` and `clientUserIdStr` fields and, if the user is already associated with an iTunes account, an `itsIdHash` field.

JSON escapes some special characters including "/". So a URL returned in JSON looks like: `"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/registerVPPUserSrv"`.

For any service that requires authentication with an `sToken` value, if the provided token is within the expiration warning period (currently 15 days before the expiration date), then the response contains an additional field, `tokenExpDate`. The value of this field is the expiration date in ISO 8601 format. For example:

"tokenExpDate":"2013-07-26T18:12:09-0700"

If this field is present in the response, it should serve as a reminder that it is time to get a new `sToken` blob in order to avoid any service disruption.

Error Codes

When a service request results in error, there will normally be two fields containing the error information in the response - an `ErrorCode` field and an `ErrorMessage` field. There could be additional fields depending on the error. The `ErrorMessage` field is a human readable text explaining the error. The `ErrorCode` field is intended for software to interpret. Any `ErrorMessage` value uniquely maps to an `ErrorCode` value, but not the other way around. The possible `ErrorCode` values are defined as follows:

ErrorCode	Meaning
9600	Missing required argument
9601	Login required
9602	Invalid argument
9603	Internal error
9604	Result not found
9605	Account storefront incorrect
9606	Error constructing token
9607	License is irrevocable
9608	Empty response from SharedData service
9609	Registered user not found
9610	License not found
9611	Admin user not found

ErrorCode	Meaning
9612	Failed to create claim job
9613	Failed to create unclaim job
9614	Invalid date format
9615	OrgCountry not found
9616	License already assigned
9617	URL has moved
9618	The user has already been retired
9619	License not associated
9620	The user has already been deleted
9621	The token has expired. You need to generate a new token online using your organization's account at https://vpp.itunes.apple.com/.
9622	Invalid authentication token
9623	Invalid Apple Push Notification token

Additional error types may be added in the future.

The Services

The following are the web services exposed to the internet that can be requested by your client.

registerVPPUserSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
clientUserIdStr	Required.	"810C9B91-DF83-41DA-80A1-408AD7F081A8"
email	Optional.	"user1@someorg.com"
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

`clientUserIdStr` is a string field. It can be, for example, the GUID of the user. The `clientUserIdStr` strings must be unique within the organization and may not be changed once a user is registered. It should not, for example, be an email address, because an email address might be reused by a future user.

When a user is first registered, the user's initial status is `Registered`. If the user has already been registered, as identified by `clientUserIdStr`:

- If the user's status is `Registered` or `Associated`, that active user account is returned.
- If the user's status is `Retired` and the user has never been assigned to an iTunes account, the account's status is changed to `Registered` and the existing user is returned.
- If the user's status is `Retired` and the user has previously been assigned to an iTunes account, a new account is created.

Thus, it is possible for more than one user record to exist for the same `clientUserIdStr` value—one for each iTunes account that the `clientUserIdStr` value has been associated with in the past (in addition to a currently active record or a retired and never-associated record). Each of these users has a unique `userId` value. Over time, with iTunes store assignment, retirement, and reassignment, it is possible for the `userId` value of the active user for a given `clientUserIdStr` to change.

Further, if two user identifiers exist for a given `clientUserIdStr`, one assigned to a iTunes account and the other unassigned, and a user accepts an invitation to be associated, it is possible for the user to use the same iTunes account that he or she used previously. If the user does, then the unassigned user record gets marked with `Retired` status, and the formerly retired user record gets moved to the `Associated` status.

The response contains some of these fields.

Field Name	Example of Value
`status`	`0` for success, `-1` for error.

Field Name	Example of Value
user	{ "userId": 100014, "email": "test_reg_user11@test.com", "status": "Registered", "inviteUrl": "https: \/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/associateVPPUserWithITSAccount?inviteCode=9e8d1ecc57924d9da13b42b4f772a066&mt=8", "inviteCode": "9e8d1ecc57924d9da13b42b4f772a066", "clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8", }
errorMessage	"\"clientUserIdStr\" or \"email\" is required input parameter"
ErrorCode	9600

getVPPUserSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
userId	One of these is required.	100001
clientUserIdStr		810C9B91-DF83-41DA-80A1-408AD7F081A8
itsIdHash	Optional.	"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

If a value is passed for `clientUserIdStr`, an `itsIdHash` (iTunes Store ID hash) value may be passed, but is optional. If a value is passed for `userId` is passed, that value is used, and `clientUserIdStr` and `itsIdHash` are ignored.

The `getVPPUserSrv` request returns users with any status—`Registered`, `Associated`, `Retired`, and `Deleted`, as described below:

- A `Registered` status indicates the user has been created in the system by making a `registerVPPUserSrv` request, but is not yet associated with an iTunes account.

- An `Associated` status indicates that the user has been associated with an iTunes account. When a user is associated with an iTunes account, an `itsIdHash` value is generated for the user record.
- A `Retired` status indicates that the user has been retired by making a `retireVPPUserSrv` request.
- A `Deleted` status indicates that indicates that a VPP user is retired and its associated iTunes user has since been invited and associated with a new VPP user that shares the same `clientUserIdStr`. Because there are two VPP users with distinct `userId` values but the same `clientUserIdStr` value, the `Deleted` status is used to ensure database consistency.

  This status appears only in the `getVPPUserSrv` service response, and only when a `userId` value is used to get a VPP user instead of a `clientUserIdStr` value. A user with a `Deleted` status, fetched by `userId`, will never change status again; its sole purpose is to ensure that your software can recognize that the `userId` is no longer associated with the `clientUserIdStr` record, and can update any internal references appropriately.

Thus, it is possible for more than one user record to exist for the same `clientUserIdStr` value—one for each iTunes account that the `clientUserIdStr` value has been associated with in the past (in addition to a currently active record or a retired and never-associated record). However, no more than one of these records can be active at any given time.

When a new record is associated with a `clientUserIdStr` value that has previously been associated with a different user, because the `clientUserIdStr` is still associated with the same iTunes user when it is retired and associated again, any irrevocable licenses originally associated with the retired VPP user, if any, are moved to the new VPP user (as identified by `userId`) automatically.

If you use a `clientUserIdStr` value to fetch the VPP user after such a reassociation, the status of that user changes from `Retired` to `Associated`. If you use `userId` values to fetch the VPP users after the association, the status of the first VPP user changes from `Retired` to `Deleted`, and the status of the second VPP user changes from `Registered` to `Associated`.

To obtain only the record for the currently active user matching a `clientUserIdStr` value, your MDM server passes the `clientUserIdStr` by itself. If no users for the `clientUserIdStr` are active (all are retired or no matching record exists), `getVPPUserSrv` returns a "result not found" error code.

To obtain an old, retired user record that was previously associated with an iTunes Store account, your MDM server can pass either the `userId` for that record or the `clientUserIdStr` and `itsIdHash` for that record.

All user record responses for this request include an `itsIdHash` if the user is associated with an iTunes account.

The response contains some of these fields.

Field Name	Example of Value
status	0 for success, -1 for error.

user	{
	"userId": 2,
	"email": "user2@test.com",
	"status": "Associated",
	"clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8",
	"itsIdHash": "C2Wwd8LcIaE2v6f2/mvu82Gs/Lc=",
	"licenses": [
	{
	"licenseId": 4,
	"adamId": 497799835,
	"productTypeId": 7,
	"pricingParam": "STDQ",
	"productTypeName": "Software",
	"isIrrevocable": false
	}
	]
	}

errorMessage	"Result not found"
ErrorCode	9604

The `itsIdHash` field is omitted if the account is not yet associated with an iTunes Store account.

Note the user object returned includes a list of licenses assigned to the user.

getVPPUsersSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
batchToken	No	EkZQCWOwhDFCwgQsUFJZkA oUU0pKLEnOUAIKZOalpFYAR YzA7OSc0pTUoNSSzKLUFJAy Q6CSWgCS88JnkgAAAA==
sinceModifiedToken	No	0zJTU5SAEplpMF4wWCozJy ezGKjS0NjM0tjUwtTA3MzQ 1FqhFgBuLPH3TgAAAA==
includeRetired	No	1
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

The batchToken and sinceModifiedToken values are generated by the server, and the batchToken value can be several kilobytes in size.

Your MDM server can use this endpoint to obtain a list of all known users from the server and to keep your MDM system up-to-date with changes made on the server. To use this endpoint, your MDM server does the following:

- Makes an initial request to getVPPUsersSrv with no batchToken or sinceModifiedToken (optionally with the includeRetired field).

  This request returns all user records associated with the provided sToken.
- If the number of users exceeds a server controlled limit (on the order of several hundred), a batchToken value is included in the response, along with the first batch of users. Your MDM server should pass this batchToken value in subsequent requests to get the next batch. As long as additional batches remain, the server returns a new batchToken value in its response.
- Once all records have been returned for the request, the server includes a sinceModifiedToken value in the response. Your MDM server should pass this token in subsequent requests to get users modified since that token was generated.

  Even if no records are returned, the response still includes a sinceModifiedToken for use in subsequent requests.

The includeRetired value contains 1 if retired users should be included in the results, else 0.

Note: The `batchToken` value encodes the original value of `includeRetired`; therefore, if a `batchToken` value is present on the request, the `includeRetired` field (if passed) is ignored.

If a `sinceModifiedToken` value is included, the `includeRetired` field is ignored, and is treated as if you had passed `1`.

The response contains some of these fields.

Field Name	Example of Value
`status`	`0` for success, `–1` for error.

`users`	`[`
	`{`
	`"userId": 2,`
	`"email": "user2@test.com",`
	`"status": "Associated",`
	`"clientUserIdStr":"810C9B91–DF83–41DA–80A1–408AD7F081A8",`
	`"itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="`
	`},`
	`{`
	`"userId": 3,`
	`"email": "user3@test.com",`
	`"status": "Registered",`
	`"inviteUrl":`
	`"https: \/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/associateVPPUserWithITSAccount?inviteCode=f551b37da07146628e8dcbe0111f0364&mt=8",`
	`"inviteCode": "f551b37da07146628e8dcbe0111f0364",`
	`"clientUserIdStr":"293C9B02–DF83–41DA–20B7–203KD7F083C9"`
	`}`
	`]`
	Note that the `inviteUrl` field is present only for users whose status is `Registered`, not for users whose status is `Associated` or `Retired` status.

Field Name	Example of Value
totalCount	5 Note that this value is returned only for requests that do not include a batchToken value.
errorMessage	"Result not found"
ErrorCode	9604
batchToken	EkZQCWOwhDFCwgQsUFJZkA oUU0pKLEnOUAIKZOalpFYAR YzA7OSc0pTUoNSSzKLUFJAy Q6CSWgCS88JnkgAAAA== Note that this field is present only if there are more entries left to read.
sinceModifiedToken	0zJTU5SAEplpMF4wWCozJy ezGKjS0NjM0tjUwtTA3MzQ 1FqhFgBuLPH3TgAAAA== Note that this field is present only if batchToken is not (that is, only after the last batch of users has been returned).

The itsIdHash field is omitted if the account is not yet associated with an iTunes Store account.

The totalCount field contains the total number of records that will be returned.

getVPPLicensesSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
batchToken	No	EkZQCWOwhDFCwgQsUFJZkA oUU0pKLEnOUAIKZOalpFYAR YzA7OSc0pTUoNSSzKLUFJAy Q6CSWgCS88JnkgAAAA==
sinceModifiedToken	No	0zJTU5SAEplpMF4wWCozJy ezGKjS0NjM0tjUwtTA3MzQ 1FqhFgBuLPH3TgAAAA==

Parameter Name	Required or Not	Example
adamId	No	408709785
pricingParam	No	"PLUS"
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

The batchToken and sinceModifiedToken values are generated by the server, and the batchToken value can be several kilobytes in size.

Your MDM server can use this endpoint to obtain a list of licenses from the server and to keep your MDM system up-to-date with changes made on the server. To use this endpoint, our MDM server does the following:

- Makes an initial request to getVPPUsersSrv with no batchToken or sinceModifiedToken (optionally with the adamId and pricingParam fields).

  This request returns all licenses associated with the provided sToken.
- If the number of licenses exceeds a server controlled limit (on the order of several hundred), a batchToken value is included in the response, along with the first batch of users. Your MDM server should pass this batchToken value in subsequent requests to get the next batch. As long as additional batches remain, the server returns a new batchToken value in its response.
- Once all records have been returned for the request, the server includes a sinceModifiedToken value in the response. Your MDM server should pass this token in subsequent requests to get licenses modified since that token was generated.

  Even if no records are returned, the response still includes a sinceModifiedToken for use in subsequent requests.

> **Note:** The batchToken and sinceModifiedToken encode whether adamId and pricingParam were originally passed; therefore, if the batchToken or sinceModifiedToken is present on the request, the adamId and pricingParam fields (if passed) are ignored.

The value of pricingParam is assumed to be "STDQ" if not specified.

The response contains some of these fields.

Field Name	Example of Value
status	0 for success, –1 for error.

Field Name	Example of Value

licenses	[
	{
	"licenseId": 1,
	"adamId": 408709785,
	"productTypeId": 7,
	"pricingParam": "STDQ",
	"productTypeName": "Software",
	"isIrrevocable": false
	},
	{
	"licenseId": 2,
	"adamId": 408709785,
	"productTypeId": 7,
	"pricingParam": "STDQ",
	"productTypeName": "Software",
	"isIrrevocable": false,
	"userId": 1,
	"clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8",
	"itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
	}
	]

totalCount	10 Note that this value is returned only for requests that do not include a batchToken value.
errorMessage	"Result not found"
ErrorCode	9604

Field Name	Example of Value
batchToken	EkZQCWOwhDFCwgQsUFJZkA oUU0pKLEnOUAIKZOalpFYAR YzA7OSc0pTUoNSSzKLUFJAy Q6CSWgCS88JnkgAAAA== Note that this field is present only if there are more entries left to read.
sinceModifiedToken	0zJTU5SAEplpMF4wWCozJy ezGKjS0NjM0tjUwtTA3MzQ 1FqhFgBuLPH3TgAAAA== Note that this field is present only if batchToken is not (that is, only after the last batch of users has been returned).

Licenses that are assigned to a user contain userId, clientUserIdStr and itsIdHashfield fields, as shown in the second example above. The totalCount field contains the total number of records that will be returned.

retireVPPUserSrv

This service disassociates a VPP user from its iTunes account and releases the revocable licenses associated with the VPP user. Currently, ebook licenses are irrevocable. The revoked licenses can then be assigned to other user in the organization. A retired VPP user can be reregistered, in the same organization, by making a registerVPPUserSrv request.

The request takes the following parameters:

Parameter Name	Required or Not	Example
userId	One of these is required. userId takes precedence.	100001
clientUserIdStr		"810C9B91-DF83-41DA-80A1-408AD7F081A8"
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

If the user passes the userId value for an already-retired user, this request returns an error that indicates that the user has already been retired.

The response contains some of these fields.

Field Name	Example of Value
status	0 for success, –1 for error.
errorMessage	"Result not found"
ErrorCode	9604

The itsIdHash field is omitted if the account is not yet associated with an iTunes Store account.

associateVPPLicenseWithVPPUserSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
userId	One of these is required. userId takes precedence.	100001
clientUserIdStr		"810C9B91–DF83–41DA–80A1–408AD7F081A8"
adamId	One of these is required. licenseId takes precedence.	361285480
licenseId		100002
pricingParam	No. Used in combination with an adamId value. "STDQ" is assumed if not specified.	"PLUS"
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

For apps, pricingParam can only be "STDQ". But if other types of assets are supported by VPP license, the following are the list of pricingParam values that may be used to narrow down the asset when used in combination with an adamId value:

pricingParam	Description	Asset
STDQ	Standard Quality	Apps and Books
PLUS	High Quality	Books

The response contains some of these fields.

Field Name	Example of Value
status	0 for success, –1 for error.
license	{ "licenseId":2, "adamId":408709785, "productTypeId":7, "pricingParam":"STDQ", "productTypeName":"Software", "isIrrevocable": false, "userId":2, "clientUserIdStr":"810C9B91–DF83–41DA–80A1–408AD7F081A8", "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc=" }
user	{ "userId": 2, "email": "user2@test.com", "status": "Associated", "clientUserIdStr":"810C9B91–DF83–41DA–80A1–408AD7F081A8", "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc=" }
errorMessage	"License not found"
ErrorCode	9602

The itsIdHash field is omitted if the account is not yet associated with an iTunes Store account.

disassociateVPPLicenseFromVPPUserSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
licenseId	Yes	100001
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

The response contains some or all of these fields.

Field Name	Example of Value
`status`	`0` for success, `-1` for error.

`license`	`{`
	`"licenseId": 4,`
	`"adamId": 408709785,`
	`"productTypeId": 7,`
	`"pricingParam": "STDQ",`
	`"productTypeName": "Software",`
	`"isIrrevocable": false`
	`}`

`user`	`{`
	`"userId": 2,`
	`"email": "user2@test.com",`
	`"status": "Associated",`
	`"clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8",`
	`"itsIdHash": "C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="`
	`}`

`errorMessage`	`"License not found"`
`ErrorCode`	`9602`

The `itsIdHash` field is omitted if the account is not yet associated with an iTunes Store account.

If the license is already disassociated, this request returns error code `9619` (license not associated).

editVPPUserSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
`userId`	One of these is required. `userId` takes precedence.	`20001`
`clientUserIdStr`		`"810C9B91-DF83-41DA-80A1-408AD7F081A8"`

Parameter Name	Required or Not	Example
email	No	"user1@someorg.com"
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="

The email field will only be updated if the value is provided in the request.

The response contains some of these fields.

Field Name	Example of Value
status	0 for success, –1 for error.
user	{ "userId":100014, "email":"test_reg_user14_edited@test.com", "status":"Registered", "inviteUrl": "https: \/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/ associateVPPUserWithITSAccount?inviteCode= 9e8d1ecc57924d9da13b42b4f772a066&mt=8", "inviteCode":"9e8d1ecc57924d9da13b42b4f772a066", "clientUserIdStr":"810C9B91–DF83–41DA–80A1–408AD7F081A8" }
errorMessage	"Missing \"userId\" input parameter"
ErrorCode	9600

VPPClientConfigSrv

This service allows the client to store some information on the server on a per organization basis. The information currently can be stored is a clientContext string. The clientContext string is any string useful to the client that is less than 256 bytes in length.

The request takes the following parameters:

Parameter Name	Required or Not	Example
clientContext	Optional.	(any string less than 256 bytes)
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ="
apnToken	Optional	"HCBv0d949nWlBQLb86E3Ppb0NK6/Qep5U64vDauQ6dU=" or "1c206fd1df78f675a50502dbf3a1373e96f434aebf41ea7953ae2f0da

If a value is provided for clientContext, then the value is stored by the server, and the response contains the current value of this field. To clear this field value, provide an empty string as the input value—that is, "".

The apnToken value is an Apple Push Notification token. A valid APN token is either a Base64-encoded string 44 characters in length or a hexadecimally encoded string 64 characters in length. Submitting an apnToken value registers the MDM server to receive push notifications. The topic of these notification is "com.apple.manageddistribution". The app assignment service currently sends three types of notifications are sent, as described in the following table.

Table 1 App Assignment push notification types

Alert Message	Badge	Custom Message	Triggered By
"Token submited"	1	type=0	Submitting or Clearing of APN token
"License bought"	1	type=1	VPP license purchase
"User associated"	1	type=2	iTunes user association with VPP user

The type=0 notification serves as a way to verify that push notifications are functioning correctly with the new apnToken value. The type=1 and type=2 notifications are intended to tell the MDM server to pull new license data and user data from the server. These notifications are used to optimize the performance of MDM servers, so they are not required. To clear the apnToken value, provide an empty string ("") as the token value.

The response contains some of these fields.

Field Name	Example of Value
status	0 for success, −1 for error.
clientContext	"abc"
errorMessage	"Login required"

Field Name	Example of Value
ErrorCode	9601

VPPServiceConfigSrv

This service returns the full list of web service URLs, the registration URL used in the user invitation email, and a list of error codes that can be returned from the web services. No parameters or authentication is necessary.

Clients should make a `VPPServiceConfigSrv` request to retrieve the list of service URLs at the appropriate moment (client restart) to ensure they are up-to-date, because the URLs may change under certain circumstances. The `VPPServiceConfigSrv` service exists to provide a level of indirection so that other service URLs can be changed in a way that is transparent to the clients.

The response contains the URLs to be used to register VPP users and other web services.

Field Name	Example of Value
invitationEmailUrl	"https://buy.itunes.apple.com/WebObjects/MZFinance.woa/wa/associateVPPUserWithITSAccount?inviteCode=%inviteCode%&mt=8" Your MDM server should replace `%inviteCode%` with the actual invitation code.
registerUserSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/registerVPPUserSrv"
editUserSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/editVPPUserSrv"
getUserSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPUserSrv"
retireUserSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/retireVPPUserSrv"

Field Name	Example of Value
getUsersSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPUsersSrv"
getLicensesSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPLicensesSrv"
associateLicense-SrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/associateVPPLicenseWithVPPUserSrv"
disassociateLicense-SrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/disassociateVPPLicenseFromVPPUserSrv"

Field Name	Example of Value
errorCodes	[{ "errorMessage":"Missing required argument", "errorNumber":9600 }, { "errorMessage":"Login required", "errorNumber":9601 }, { "errorMessage":"Invalid argument", "errorNumber":9602 }, { "errorMessage":"Internal error", "errorNumber":9603 }, { "errorMessage":"Result not found", "errorNumber":9604 }, { "errorMessage":"Account storefront incorrect", "errorNumber":9605 }, { "errorMessage":"Error constructing token", "errorNumber":9606 }, { "errorMessage":"License irrevocable", "errorNumber":9607 }, { "errorMessage":"Empty SharedData response", "errorNumber":9608 }, { "errorMessage":"User not found", "errorNumber":9609 }, { "errorMessage":"Lincese not found", "errorNumber":9610 }, { "errorMessage":"Admin user not found", "errorNumber":9611 }, { "errorMessage":"Fail creating SAPFeeder job for claim", "errorNumber":9612 }, { "errorMessage":"Fail creating SAPFeeder job for unclaim", "errorNumber":9613 }, { "errorMessage":"Invalid date formate", "errorNumber":9614 }, { "errorMessage":"orgCountry not found", "errorNumber":9615 }, { "errorMessage":"License already assigned to the iTunes account", "errorNumber":9616 }, { "errorMessage":"The URL has been moved. Please call VPPServiceConfigSrv to find the new URL.", "errorNumber":9617 }, { "errorMessage":"User already retired", "errorNumber":9618 }, { "errorMessage":"License not associated", "errorNumber":9619 }, { "errorMessage":"User already deleted", "errorNumber":9620 }, { "errorMessage":"The token has expired. You need to generate a new token online using your organization's account at https:\/\/vpp.itunes.apple.com.", "errorNumber":9621 }, { "errorMessage":"The authentication token is invalid.", "errorNumber":9622 }, { "errorMessage":"The APN token is invalid.", "errorNumber":9623 }]
disassociateLicense-SrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/disassociateVPPLicenseFromVPPUserSrv"

Field Name	Example of Value
clientConfigSrvUrl	"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/VPPClientConfigSrv"

Examples

The following are examples of requests and responses of each service. The requests are made with curl command from the command line. The response JSON are all formatted with beautifier to facilitate viewing. They were one string without line breaks when received from the web services.

Request to VPPServiceConfigSrv

The curl command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/VPPServiceConfigSrv
```

The response:

```
{

"getLicensesSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/getVPPLicensesSrv",

"registerUserSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/registerVPPUserSrv",
   "status":0,

"invitationEmailUrl":"http:\/\/buy.itunes.apple.com\/us\/vpp-associate?inviteCode=%25inviteCode%25&mt=8",

"associateLicenseSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/associateVPPLicenseWithVPPUserSrv",
   "errorCodes":[
      {
         "errorMessage":"Missing required argument",
         "errorNumber":9600
      },
      {
         "errorMessage":"Login required",
         "errorNumber":9601
```

```
},
{
   "errorMessage":"Invalid argument",
   "errorNumber":9602
},
{
   "errorMessage":"Internal error",
   "errorNumber":9603
},
{
   "errorMessage":"Result not found",
   "errorNumber":9604
},
{
   "errorMessage":"Account storefront incorrect",
   "errorNumber":9605
},
{
   "errorMessage":"Error constructing token",
   "errorNumber":9606
},
{
   "errorMessage":"License irrevocable",
   "errorNumber":9607
},
{
   "errorMessage":"Empty SharedData response",
   "errorNumber":9608
},
{
   "errorMessage":"User not found",
   "errorNumber":9609
},
{
```

```
         "errorMessage":"Lincese not found",
         "errorNumber":9610
      },
      {
         "errorMessage":"Admin user not found",
         "errorNumber":9611
      },
      {
         "errorMessage":"Fail creating SAPFeeder job for claim",
         "errorNumber":9612
      },
      {
         "errorMessage":"Fail creating SAPFeeder job for unclaim",
         "errorNumber":9613
      },
      {
         "errorMessage":"Invalid date formate",
         "errorNumber":9614
      },
      {
         "errorMessage":"orgCountry not found",
         "errorNumber":9615
      },
      {
         "errorMessage":"License already assigned to the iTunes account",
         "errorNumber":9616
      },
      {
         "errorMessage":"The URL has been moved. Please call VPPServiceConfigSrv
to find the new URL.",
         "errorNumber":9617
      },
      {
         "errorMessage":"User already retired",
         "errorNumber":9618
```

```
      },
      {
         "errorMessage":"License not associated",
         "errorNumber":9619
      },
      {
         "errorMessage":"User already deleted",
         "errorNumber":9620
      },
      {
         "errorMessage":"The token has expired. You need to generate a new token
online using your organization's account at https:\/\/vpp.itunes.apple.com.",
         "errorNumber":9621
      },
      {
         "errorMessage":"The authentication token is invalid.",
         "errorNumber":9622
      },
      {
         "errorMessage":"The APN token is invalid.",
         "errorNumber":9623
      }
   ],

"editUserSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/editVPPUserSrv",

"retireUserSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/retireVPPUserSrv",
   "vppWebsiteUrl":"https:\/\/vpp.itunes.apple.com\/",

"clientConfigSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/VPPClientConfigSrv",

"disassociateLicenseSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/disassociateVPPLicenseFromVPPUserSrv",

"getUsersSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/getVPPUsersSrv",

"getUserSrvUrl":"https:\/\/vpp.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/getVPPUserSrv"
}
```

Request to getVPPLicensesSrv

Content of the get_licenses.json file used in the curl command next:

```
{"sToken":"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2
a5Rlopo4KDn3MrFKf4CM3OY+WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ="}
```

The curl command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPLicensesSrv
-d @get_licenses.json
```

The response:

```
{
   "status":0,
   "totalCount":3,
   "licenses":[
      {
         "licenseId":1,
         "adamId":408709785,
         "productTypeId":7,
         "pricingParam":"STDQ",
         "productTypeName":"Software",
         "isIrrevocable":false
      },
      {
         "licenseId":2,
         "adamId":408709785,
         "productTypeId":7,
         "pricingParam":"STDQ",
         "productTypeName":"Software",
         "isIrrevocable":false,
         "userId":2,
         "clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8",
         "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
      },
```

```
      {
         "licenseId":3,
         "adamId":497799835,
         "productTypeId":7,
         "pricingParam":"STDQ",
         "isIrrevocable":false,
         "productTypeName":"Software"
      }
   ]
}
```

Request to getVPPUsersSrv

Content of the get_users.json file used in the curl command next:

```
{"sToken":"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2
a5Rlopo4KDn3MrFKf4CM3OY+WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ="}
```

The curl command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPUsersSrv -d
@get_users.json
```

The response:

```
{
   "users":[
      {
         "userId":1,
         "email":"user1@test.com",
         "clientUserIdStr":"200006",
         "status":"Associated"
         "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
      },
      {
         "userId":2,
```

```
         "email":"user2@test.com",
         "clientUserIdStr":"200007",
         "status":"Associated"
         "itsIdHash":"*leSKk3IaE2vk2KLmv2k3/200D3="
      },
      {
         "userId":3,
         "email":"user3@test.com",
         "clientUserIdStr":"user3@test.com",
         "status":"Registered",
         "inviteCode":"f551b37da07146628e8dcbe0111f0364"
       "inviteUrl":"https:\/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/
             associateVPPUserWithITSAccount?inviteCode=
             f551b37da07146628e8dcbe0111f0364&mt=8",
      },
      {
         "userId":4,
         "email":"user4@test.com",
         "clientUserIdStr":"user4@test.com",
         "status":"Registered",
       "inviteUrl":"https:\/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/wa\/
             associateVPPUserWithITSAccount?inviteCode=
             859c5aa3485a48918a5f4f70c5629ec8&mt=8",
         "inviteCode":"859c5aa3485a48918a5f4f70c5629ec8"
      }
   ],
   "status":0,
   "totalCount":4
}
```


Request to getVPPUserSrv

Content of the get_user.json file used in the curl command next:


```
{"userId": 1, "sToken":"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQ
keeb3h2a5Rlopo4KDn3MrFKf4CM3OY+WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ="}
```

The curl command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPUserSrv -d
@get_user.json
```

The response:

```
{
   "status":0,
   "user":{
      "userId":1,
      "email":"user1@test.com",
      "clientUserIdStr":"200006",
      "status":"Associated",
      "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
      "licenses":[
         {
            "licenseId":2,
            "adamId":408709785,
            "productTypeId":7,
            "pricingParam":"STDQ",
            "productTypeName":"Software",
            "isIrrevocable":false
         },
         {
            "licenseId":4,
            "adamId":497799835,
            "productTypeId":7,
            "pricingParam":"STDQ",
            "productTypeName":"Software",
            "isIrrevocable":false
         }
      ]
   }
}
```

Request to registerVPPUserSrv

Content of the reg_user.json file used in the curl command next:

```
{"email": "test_reg_user11@test.com", "clientUserIdStr": "200002", sToken":
"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2a5Rlopo4KDn3MrFKf4CM3OY+
WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ=" }
```

The curl command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/registerVPPUserSrv
-d @reg_user.json
```

The response:

```
{
   "status":0,
   "user":{
      "userId":100014,
      "email":"test_reg_user11@test.com",
      "status":"Registered",
      "inviteUrl": "https:\/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/
          wa\/associateVPPUserWithITSAccount?inviteCode=
          89e8d1ecc57924d9da13b42b4f772a066&mt=8",
      "inviteCode":"9e8d1ecc57924d9da13b42b4f772a066",
      "clientUserIdStr":"200002"
   }
}
```

Request to associateVPPLicenseWithVPPUserSrv

Content of the associate_license.json file:

```
{"userId": 2, "licenseId": 4, "sToken":
"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2a5Rlopo4KDn3MrFKf4CM3OY+
WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ=" }
```

The command:

```
curl
https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/associateVPPLicenseWithVPPUserSrv
 -d @associate_license.json
```

The response:

```
{
   "status":0,
   "license":{
       "licenseId":4,
       "adamId":497799835,
       "productTypeId":7,
       "pricingParam":"STDQ",
       "productTypeName":"Software",
       "isIrrevocable":false,
       "userId": 2,
       "clientUserIdStr":"200007",
       "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
   },
   "user":{
       "userId":2,
       "email":"user2@test.com",
       "clientUserIdStr":"200007",
       "status":"Associated",
       "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
   }
}
```

Request to disassociateVPPLicenseFromVPPUserSrv

Content of the `disassociate_license.json` file:

```
{"userId": 2, "licenseId": 4, "sToken":
"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2a5Rlopo4KDn3MrFKf4CM3OY+
WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ=" }
```

The command:

```
curl
https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/disassociateVPPLicenseFromVPPUserSrv
 -d
@disassociate_license.json
```

The response:

```
{
   "status":0,
   "license":{
      "licenseId":4,
      "adamId":497799835,
      "productTypeId":7,
      "pricingParam":"STDQ",
      "isIrrevocable":false,
      "productTypeName":"Software",
   },
   "user":{
      "userId":2,
      "email":"user2@test.com",
      "clientUserIdStr":"user2@test.com",
      "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
      "status":"Associated",
      "inviteCode":"a5ea54beb2954d4dadc65cf19cee5e58",
   }
}
```

Request to editVPPUserSrv

Content of the edit_user.json file:

```
{"userId": 100014, "email": "test_reg_user15_edited@test.com", "sToken":
"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2a5Rlopo4KDn3MrFKf4CM3OY+
WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ=" }
```

The command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/editVPPUserSrv -d
@edit_user.json
```

The response:

```
{
   "status":0,
   "user":{
      "userId":100014,
      "email":"test_reg_user15_edited@test.com",
      "status":"Registered",
      "inviteUrl": "https:\/\/buy.itunes.apple.com\/WebObjects\/MZFinance.woa\/
          wa\/associateVPPUserWithITSAccount?inviteCode=
          9e8d1ecc57924d9da13b42b4f772a066&mt=8",

      "inviteCode":"9e8d1ecc57924d9da13b42b4f772a066",
      "clientUserIdStr":"200015",
      "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc="
   }
}
```

Request to retireVPPUserSrv

Content of the retire_user.json file:

```
{"userId": 1, "sToken":
"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxefOuQkeeb3h2a5Rlopo4KDn3MrFKf4CM3OY+
WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ=" }
```

The command:

```
curl https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/retireVPPUserSrv -d
 @retire_user.json
```

The response:


```
{
   "status":0,
   "user":{
      "userId":1,
      "email":"user1@test.com",
      "clientUserIdStr":"200006",
      "status":"Retired",
      "licenses":[
          {
              "licenseId":2,
              "adamId":408709785,
              "productTypeId":10,
              "pricingParam":"STDQ",
              "productTypeName":"Publication",
              "isIrrevocatble":true
          }
      ]
   }
}
```

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.

In a single-user environment on OS X, installing an MDM profile will cause the device to be managed by MDM (via device profiles) and the user that installed the profile (via user profiles), but any other local user logging into that machine will not be managed (other than via device profiles).

Multiple network users bound to Open Directory servers can also have their devices managed, assuming the MDM server is configured to recognize them.

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.

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. In iOS 7 and later, profiles can be updated even after expiration, as described in "Updating Expired Profiles" (page 146). In older versions of iOS, when a certificate in the profile is about to expire, an "Update" button appears 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.

Device Enrollment Program

The Device Enrollment Program, when combined with an MDM server, makes it easier to deploy configuration profiles over the air to devices that you own. When performed at the time of purchase, devices enrolled in this program can prompt the user to begin the MDM enrollment process as soon as the device is first activated, removing the need for preconfiguring each device.

Device Enrollment Program allows devices to be supervised during activation. Supervised devices allow an MDM server to apply additional restrictions and to send certain configuration commands that you otherwise cannot send, such as setting the device's language and locale, starting and stopping AirPlay Mirroring, and so on. Also, MDM profiles delivered using Device Enrollment Program cannot be removed by the user.

MDM vendors can take advantage of web services provided by the Device Enrollment Program, integrating its features with their services.

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 144).

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.

Updating Expired Profiles

In iOS 7 and later, an MDM server can replace profiles that have expired signing certificates with new profiles that have current certificates. This includes the MDM profile itself.

To replace an installed profile, install a new profile that has the same top-level `PayloadIdentifier` as an installed profile.

Replacing an MDM profile with a new profile restarts the check-in process. If an SCEP payload is included, a new client identity is created. If the update fails, the old configuration is restored.

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 correctly, 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 7.0 And Later

In iOS 7.0 and later, you can use MDM's app assignment feature to assign app licenses to iTunes accounts. MDM can then be used to push a VPP app to a device that is signed in to that iTunes account. You can later remove those licenses and use them with other iTunes accounts.

Also, in iOS 7.0 and later, an MDM server can provide configuration dictionaries to managed apps and can read response dictionaries from those apps. Apps can take advantage of this functionality to preconfigure themselves in a supervised environment, such as a classroom setting.

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 142), 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.

Managed “Open In”

In iOS 7.0 and later, an MDM server can prevent accidental movement of data in and out of managed accounts and apps on a user's device by installing a profile with a Restrictions payload that specifies the restrictions `allowOpenFromManagedToUnmanaged` and `allowOpenFromUnmanagedToManaged`.

When the `allowOpenFromManagedToUnmanaged` restriction is specified, an Open In sheet started from within a managed app or account will show only other managed apps and accounts. When the `allowOpenFromUnmanagedToManaged` restriction is specified, an Open In sheet started from within an unmanaged app or account will show only other unmanaged apps and accounts.

The Open In sheet shown by Safari and AirDrop continues to show all apps and accounts even when these restrictions are specified.

It is a best practice to use these restrictions to manage data and attachments on a user's device.

MDM Vendor CSR Signing Overview

The process of generating an APNS push certificate can be completed using the Apple Push Notification Portal.

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

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 sign 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 the private key from 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.

   Be certain that the `PushCertCertificateChain` value contains a *complete* certificate chain all the way back to a recognized root certificate (including the root certificate itself). This means it must contain your MDM signing certificate, the WWDR intermediate certificate (available from http://developer.apple.com/certificationauthority/AppleWWDRCA.cer), and the Apple Inc. root certificate (available from http://www.apple.com/appleca/AppleIncRootCertificate.cer).

   Also, be sure that every certificate complies with PEM formatting standards; each line except the last must contain exactly 64 printable characters, and the last line must contain 64 or fewer printable characters.

   It may be helpful to save the certificate and its chain into a file ending in `.pem` and then verify your certificate chain with the `certtool` (`certtool –e < filename.pem`) or `openssl` (`openssl verify filename.pem`) command-line tools. To learn more about certificates and chains of trust, read *Security Overview*.)

   Refer to the code samples in Listing 1 (page 152), Listing 2 (page 153), and Listing 3 (page 153) for additional instructions.

   > **Note:** To minimize the risk of errors, you should use Xcode or the standalone Property List Editor application when editing property lists.
   >
   > Alternatively, on the command line, you can make changes to property lists with the `plutil` tool or check the validity of property lists with the `xmllint` tool.

6. Deliver the PushCertWebRequest file back to the customer and direct them to https://identity.apple.com/pushcert to upload it to Apple.

Be sure to use a separate push certificate for each customer. There are two reasons for this:

- If multiple customers shared the same push topic, they would be able to see each other's device tokens.
- When a push certificate expires, gets invalidated or revoked, gets blocked, or otherwise becomes unusable, any customers sharing that certificate lose their ability to use MDM.

All devices for the same customer should share a single push certificate. This same certificate should also be used to connect to the APNS feedback service.

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
AwwRU0FDSSBUZXN0IFJvb3QgQ0ExEjAQBgNVBAsMCUFwcGxlIElTVDETMBEGA1UE
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDlTCCAn2gAwIBAgIIBInl9fQbaAkwDQYJKoZIhvcNAQEFBQAwXDEkMCIGA1UE
AwwbU0FDSSBUZXN0IEludGVybWVkaWF0ZSBDQSAxMRIwEAYDVQQLDAlBcHBsZSBJ
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDpjCCAo6gAwIBAgIIKRyFYgyyFPgwDQYJKoZIhvcNAQEFBQAwXDEkMCIGA1UE
AwwbU0FDSSBUZXN0IEludGVybWVkaWF0ZSBDQSAxMRIwEAYDVQQLDAlBcHBsZSBJ
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDiTCCAnGgAwIBAgIIdv/cjbnBgEgwDQYJKoZIhvcNAQEFBQAwUjEaMBgGA1UE
AwwRU0FDSSBUZXN0IFJvb3QgQ0ExEjAQBgNVBAsMCUFwcGxlIElTVDETMBEGA1UE
-----END CERTIFICATE-----
</string>
<key>PushCertSignature</key>
<string>
CGt6QWuixaO0PIBc9dr2kJpFBE1BZx2D8L0XH0Mtc/DePGJOjrM2W/IBFY0AVhhEx
</string>
```

Document Revision History

If you find errors in this documentation, please file bugs at bugreport.apple.com in the component `Documentation (developer)` version `X`.

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

Date	Notes
2014-01-15	Updated for iOS 7 and OS X 10.9.
2013-03-13	General revision and updates.
2012-09-20	Fixed a few minor errors.
2012-09-04	Updated document to support OS X.
2011-12-09	Clarified format of certificates.
2011-10-03	Updated for iOS 5.0 and Corrected push cert URL.
2011-02-16	Updated for CDMA support
2010-12-09	Updated for iOS 4.2
2010-09-14	First version

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