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 v10.7 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 it 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 a webpage, 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 are 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 they are being prepared for deployment with Apple Configurator 2. 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 v10.8 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. In earlier OS versions, the device does not contact the MDM server when the user removes the payload. See MDM Best Practices (page 222) 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 who design 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 13)

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 18)

The Way You Design Your Payload Matters

For maximum effectiveness and security, 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 222)

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 105)

The Volume Purchase Program Lets You Assign App Licenses to Users and Devices

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

Related Chapter: VPP App Assignment (page 151)

Apple Push Notification Certificates Can Be Generated Through the Apple Push Certificates 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 236)

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 push 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</string>
    <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 Message

While the user is installing an MDM payload, the device sends an authenticate message that contains at least 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.

The device may also send the following key-value pairs if it is running iOS 9 or later and if it has the Device Information access right:

Key	Type	Value
`OSVersion`	String	The device's OS version.
`BuildVersion`	String	The device's build version.
`ProductName`	String	The device's product name (e.g., `"iPhone3,1"`).
`SerialNumber`	String	The device's serial number.
`IMEI`	String	The device's IMEI (International Mobile Station Equipment Identity).
`MEID`	String	The device's MEID (mobile equipment identifier).

Server 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 sends a token update message.

TokenUpdate Message

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` message 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. The server should use this updated token when sending push notifications to the device.
`PushMagic`	String	The magic string that must be included in the push notification message. This value is generated by the device (see below).
`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 50) command. This feature is not available in OS X. The data blob may be up to 8 kB in size after Base64 decoding.
`Awaiting-Configuration`	Boolean	Optional. If set to `true`, the device is awaiting a `DeviceConfigured` MDM command before proceeding through Setup Assistant. **Availability:** Available in iOS 9 and later and can only be sent by DEP (see Device Enrollment Program (page 105)).

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. If the device reports that it is `AwaitingConfiguration`, the MDM server is expected to send a `DeviceConfigured` MDM command before the device can allow the user to proceed in Setup Assistant. This gives the MDM server the opportunity to do some setup via MDM commands.

In addition to sending the initial `TokenUpdate` message, the iOS device may now send additional `TokenUpdate` messages to the check-in server at any time while it has a valid MDM enrollment.

The use of `PushMagic` constrains the device to a unique MDM relationship. When a user removes the MDM profile, the device should no longer listen to the former relationship, even if the user reestablishes a management relationship with the same server topic. Note that only the push topic is the same in this case; the server's address could have changed. This also helps when a user restores a device from backup that contains an older relationship. The use of `PushMagic` also ensures that the server that receives the CheckIn message is owned by the same enterprise as the computer sending push notifications. This is important because there is no way of knowing if the push topic belongs to the owner of the checkin server. It is conceivable that Apple could revoke a push token for one party, only to have that party re-enroll people piggybacking on some other topic that's actively pushing. The fact that all MDM push topics reside in the namespace `com.apple.mgmt.*` helps prevent this.

Warning: The `PushMagic` or `UnlockToken` fields of subsequent `TokenUpdate` messages may be identical to those in previous messages or may be different. If different, the server should update its record for the device to the new value provided by the message. Failure to do so results in the server being unable to send push notifications or perform passcode rests.

New `PushMagic` or `UnlockToken` values may differ in size from previous values. While the `UnlockToken` message can be sent multiple times by the device, it is possible it may only be sent once if `PushMagic` or `UnlockToken` values change. Implementations should not rely on repeated messages to update lost server-side data or to recover from a failure to process a previous `TokenUpdate` 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 v10.9, 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`.
`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 19).
- 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 log in, extensions to the MDM protocol were developed to identify and authenticate the network user logging in so that any network user is also managed by the MDM server (via their user profiles). The extensions made to the MDM protocol are described in MDM Protocol Extensions (page 26).

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).

`ServerCapabilities`	Array	Optional. An array of strings indicating server capabilities. If the server manages OS X devices or a shared iPad, this field is mandatory and must contain the value `com.apple.mdm.per-user-connections`. This indicates that the server supports both device and user connections. See MDM Protocol Extensions (page 26).
`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 226) 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, 1 must also be specified. If 128 is specified, 64 must also be specified.
`UseDevelopmentAPNS`	Bool	Optional. If `true`, the device uses the development APNS servers. Otherwise, the device uses the production servers. Defaults to `false`. 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 is JSON-formatted and must contain the `PushMagic` string as the value of the `mdm` key. For example:

```
{"mdm":"PushMagicValue"}
```

In place of `PushMagicValue` above, 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 request to the MDM server. The MDM server can then reply with the next command or end the connection by sending a `200` status (OK) with an empty response body.

Note: An empty response body must be zero bytes in length, not an empty property list.

If the connection is broken while the device is performing a command, the device caches the result of the command and re-attempts 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 Xcode or Apple Configurator 2. 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 required 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 required 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 25).
`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 26).

The `Status` key contains 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` array 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. It will poll the server again in the future. For details, see Error Handling (page 33).

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 Error Codes (page 88).

MDM Protocol Extensions

OS X Extensions

Unlike iOS clients, an OS X client on an MDM server enrolls devices and users as separate entities. OS X supports several extensions to the MDM protocol to allow managing the device and logged-in user independently. When enrolled in this manner, the MDM server receives requests for the device and for each logged-in user.

Device requests are sent from the `mdmclient` daemon, while user requests are sent from the `mdmclient` agent. If multiple users are logged in, there is one instance of an `mdmclient` agent for each logged-in user, and each may be sending requests concurrently in addition to device requests from the daemon.

Devices and users are assigned different push tokens. The server can use this difference to determine whether the device or a specific user is to contact the server with an Idle request.

To indicate that an MDM server supports both device and user connections, its MDM enrollment payload must contain the string `com.apple.mdm.per-user-connections`; see Structure of MDM Payloads (page 19). The MDM enrollment profile should be delivered as any other manually-installed profile, but MDM promotes it to a device profile once it is installed. This will have the following consequences:

- The device will be managed.
- The local user that installed the profile will be managed.
- No other local users will be managed. The server will never get requests from a local user other than the one that installed the enrollment profile.
- Network users logging into the device will be managed if the server responds successfully to their `UserAuthenticate` messages. If the server does not want to manage a network client, it should return a `410` HTTP status code.

During enrollment, the client sends the standard `Authenticate` request to the `CheckInURL` specified in the MDM payload. Once that request completes, the client sends one `TokenUpdate` request for the device and another for the user that performed the enrollment. The same client certificate is used to authenticate both device and user connections.

To help the server differentiate requests coming from a device versus a user, user requests contain additional keys in their request plists:

```
<key>UDID</key>
<string>23EB7CD8-5567-5E97-827F-06E4E4C456B2</string>
<key>UserID</key>
<string>F17C470A-3ADC-47EC-A7CC-D432867F4793</string>
<key>UserLongName</key>
<string>Jimmy Smith</string>
<key>UserShortName</key>
<string>jimmys</string>
<key>NeedSyncResponse</key>
<boolean>true</boolean>
```

Note the following conditions for including the foregoing keys:

- Requests from a device contain only the `UDID` key.
- `NeedSyncResponse` is optional. If it is present and true, it indicates that the client is in a state where the user is waiting for the completion of an MDM transaction. In OS X 10.9 and later versions, this key is added during user login when the login is blocked while the client checks in with the MDM server to ensure it has the latest settings and profiles. The key is meant as a hint to the server that it should send all commands in the current set of Idle/Acknowledged/Error transactions instead of relying on push notifications. During login, the client blocks the transaction only until the server sends an empty response to an Idle/Acknowledged/Error sequence.
- `UserConfiguration` is optional. If it is present and true, it indicates that the OS X client is trying to obtain user-specific settings while in Setup Assistant during Device Enrollment (see Device Enrollment Program (page 105)). After an OS X client obtains device-specific settings, it also attempts to determine if the server has any user-specific settings that may affect Setup Assistant. Currently, only password policies fall into this category. The password policies are used if Setup Assistant prompts to create a local user account. After the client receives a DeviceConfigured command on the device connection, it starts a normal Idle/Acknowledged/Error connection on the user connection. If the server sends commands or profiles during this time, nothing the client receives persists, because the user account hasn't been created on the system yet. The client always responds `NotNow` to any commands it received during this time. It continues to respond with `NotNow` until it receives a reply with no additional commands (an empty body) or a `DeviceConfigured` command on the user connection. The client passes any password policies to Setup Assistant and discards everything else. After Setup Assistant creates the user account and the user logs in, the client initiates a new series of Idle/Acknowledged/Error connections. The server should then resend all commands and profiles. The client processes them normally and they will persist.

Network User Authentication Extensions

To support environments where an OS X computer is bound to an Open Directory server and various network users may log in, extensions to the MDM protocol were developed to identify and authenticate the network user logging in. This way, network users are 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 the `CheckInURL` address specified in the MDM payload.

The first request to the server is sent to the `CheckInURL` specified in the MDM payload, 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	Local user's GUID, or network user's GUID from Open Directory Record (see below).

If the OS X device being enrolled has an owner, the `UserID` key may designate a local user instead of a network user. If the local request succeeds, an `—MDM—is—owned` header is added to the response to all requests to the `checkinURL`, except `CheckOut` requests where it is optional. To this header may be added a value of 1 to indicate the device is owned; this is also the default behavior if the header is omitted. Only if the header is present with a value of 0 will requests from the client be optimized.

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, 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.

If the server does not want to manage this user, it should return a `410` HTTP status code. The client will not make any additional requests to the server on behalf of this user for the duration of this login session. The next time that user logs in, however, the client will again send a `UserAuthenticate` request and the server can optionally return `410` again.

The second request to the server is also sent to the `CheckInURL` specified in the MDM payload 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.

Key	Type	Content
`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, 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 from the 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.

It is assumed that the `AuthToken` remains valid until the next time the client sends a `UserAuthenticate` request. The client initiates a `UserAuthenticate` handshake each time a network user logs in.

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

The following is an example of a `UserAuthenticate` handshake:

```
// UserAuthenticate request from client to server:
<dict>
       <key>MessageType</key>
       <string>UserAuthenticate</string>
       <key>UDID</key>
       <string>23EB7CD8-5567-5E97-827F-06E4E4C456B2</string>
       <key>UserID</key>
```

```
      <string>16C0477E-EB2F-4B5E-AAFD-92B2B91C4B16</string>
</dict>

// Server sends challenge:
<dict>
      <key>DigestChallenge</key>
      <string>Digest nonce="8BrAkk4GZgrG//
            2XaDLMSSSo89VenjV5E8Se73z98RvSW7Rs",realm="fusion.home"<string>
</dict>

// Client sends response:
<dict>
      <key>DigestResponse</key>
      <string>Digest username="net1",realm="fusion.home",
            nonce="8BrAkk4GZgrG2XaDLMSSSo89VenjV5E8Se73z98RvSW7Rs",
            uri="/",response="84db40bbaf5e0d49cabb0ef7d8cac369"</string>
      <key>MessageType</key>
      <string>UserAuthenticate</string>
      <key>UDID</key>
      <string>23EB7CD8-5567-5E97-827F-06E4E4C456B2</string>
      <key>UserID</key>
      <string>16C0477E-EB2F-4B5E-AAFD-92B2B91C4B16</string>
</dict>

// Server responds with AuthToken for client session:
<key>AuthToken</key>
<string>uEOcQRJrXGbMJUDAkDZSCny5e90=</string>

// From this point on, all user requests from that network user will include an
AuthToken key:
<dict>
      <key>AuthToken</key>
      <string>uEOcQRJrXGbMJUDAkDZSCny5e90=</string>
      <key>Status</key>
      <string>Idle</string>
```

```
        <key>UDID</key>
        <string>23EB7CD8-5567-5E97-827F-06E4E4C456B2</string>
        <key>UserID</key>
        <string>16C0477E-EB2F-4B5E-AAFD-92B2B91C4B16</string>
        <key>UserLongName</key>
        <string>Net One</string>
        <key>UserShortName</key>
        <string>net1</string>
</dict>
```

For push notifications, the client uses different push tokens for device and user connections. Each token is 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.

iOS Support for Per-User Connections

A device running iOS 9.3 or later, and its logged-in users, can be managed independently as a Shared iPad, using a technique similar to Network User Authentication Extensions (page 28). The device and its users are assigned different push tokens. The server can use this difference to determine whether the device or a specific user is to contact the server with an `Idle` request.

To indicate that an MDM server supports both device and user connections, the `ServerCapabilities` array in its MDM enrollment payload must contain the string `com.apple.mdm.per-user-connections`, indicating support for shared iPad. Then when a user logs in, the device sends a `TokenUpdate` request on the user channel.

To help the server differentiate requests coming from a device versus a user, user requests must contain additional keys:

Key	Type	Content
`UserID`	String	Always set to `FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF` to indicate that no authentication will occur.
`UserLongName`	String	The full name of the user.
`UserShortName`	String	The Managed Apple ID of the user.

If the server is configured to manage the user, it stores the user push token and returns a 200 response. At this point the device polls the server for a command on the user channel.

If the server is not configured to manage the user, it should return a 410 HTTP status code. The client will not make any additional requests to the server on behalf of this user for the duration of the login session. The next time the user logs in, however, the client will again send a `UserAuthenticate` request and the server can optionally return a 410 code again.

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 sends a `NotNow` status without performing the command. The server may send another command immediately after receiving this status. See "Handling a NotNow Response," below, for more details.

The following commands are guaranteed to execute in 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`).

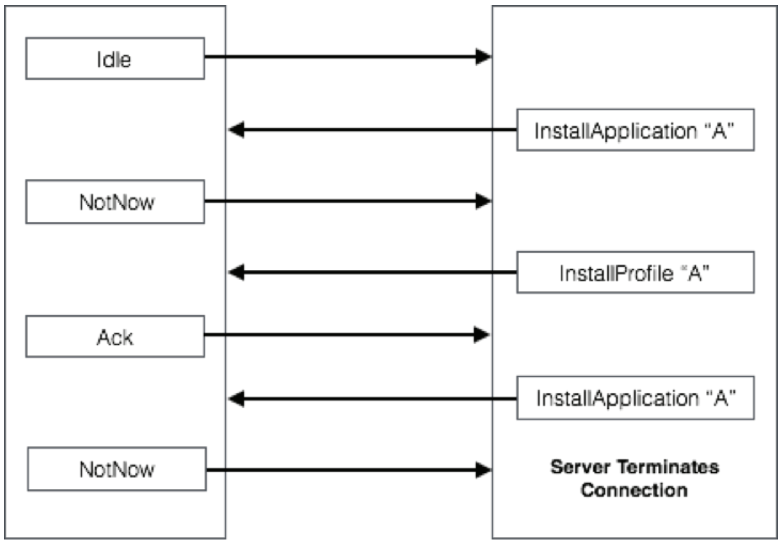
Handling a NotNow Response

If the device's response to the previous command sent has a status of `NotNow`, your server has two response choices:

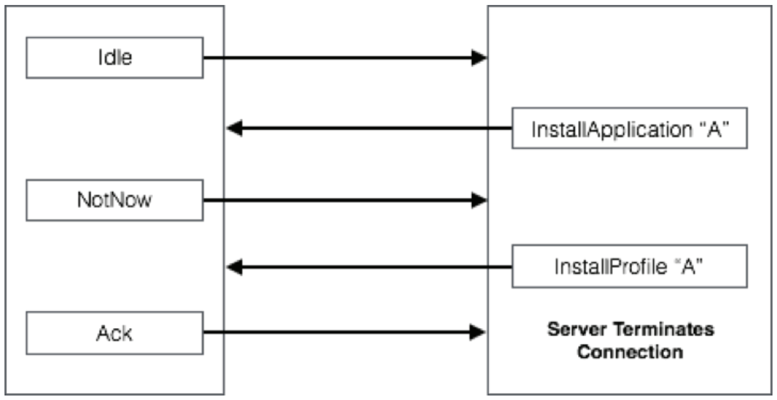
- It may immediately stop sending commands to the device. In this case the device automatically polls your server when conditions change and it is able to process the last requested command. The server does not need to send another push notification in response to this status. However, the server may send another push notification to the device to have it poll the server immediately. The device does not cache the command that was refused. If the server wants the device to retry the command, it must send the command again when the device polls the server.
- It may send another command on the same connection, but if this new command returns anything other than a `NotNow` response, the device will *not* automatically poll the server as it would have with the first response choice. The server must send a push notification at a later time to make the device reconnect. The device polls the server in response to a `NotNow` status only if that is the last status sent by the device to the server.

The three example flowcharts below illustrate the foregoing choices.

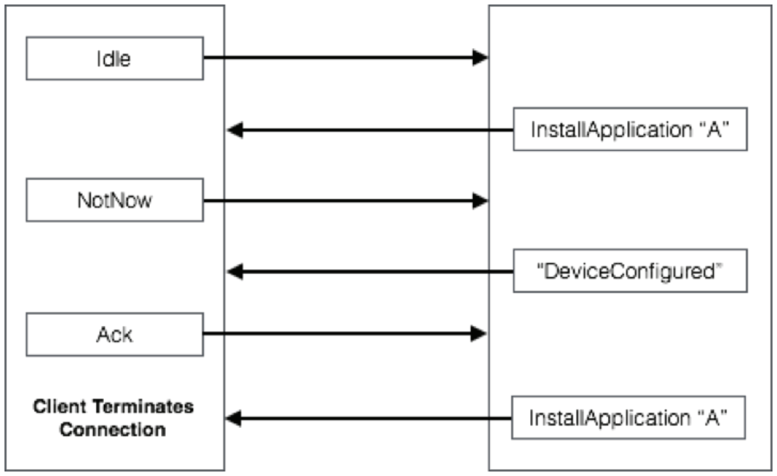
Example 1: The final command results in the server receiving a `NotNow` response. The device will poll the server later, when the `InstallApplication` command might succeed.



Example 2: The final command results in the server receiving something other than a `NotNow` response. The device will not poll the server later, because the last response was not `NotNow`.



Example 3: The connection to the device is unexpectedly interrupted. Because the last status the server received was not `NotNow`, the server should send a push notification to the device to retry the `InstallApplication` command. The server must not assume that the device will automatically poll the server later.



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 86).

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 an existing one, follow these rules:

- The new MDM profile must be signed with the same identity as the existing 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. 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.

Key	Type	Content
`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 responds 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 39).

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 40).

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.
`IsValidated`	Boolean	If `true`, the app has validated as allowed to run and is able to run on the device. If an app is enterprise-distributed and is not validated, it will not run on the device until validated. **Availability:** Available in iOS 9.2 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`
`Queries`	Array	Array of strings. Each string is a value from Table 5 (page 41), Table 7 (page 43), or Table 9 (page 46).

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 41), Table 7 (page 43), or Table 9 (page 46). 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. **Availability:** Available in Apple TV software 6.0 and later only.
`Locales`	String	Array of strings. The first entry in this array indicates the current locale. **Availability:** Available in Apple TV software 6.0 and later only.
`DeviceID`	String	The Apple TV device ID. Available in iOS 7 (Apple TV software 6.0) and later, on Apple TV only.
`OrganizationInfo`	Dictionary	The contents (if any) of a previously set `OrganizationInfo` setting. Available in iOS 7 and later.

Query	Reply Type	Comment
`LastCloudBackupDate`	Date	The date of the last iCloud backup. **Availability:** Available in iOS 8.0 and later.
`Awaiting-Configuration`	Boolean	If `true`, device is still waiting for a DeviceConfigured message from MDM to continue through Setup Assistant. **Availability:** Available in iOS 9 and later and the response is only generated by devices enrolled in MDM via DEP (see Device Enrollment Program (page 105)).
`AutoSetupAdmin-Accounts`	Array of Dictionaries	Returns the local admin users (if any) created automatically by Setup Assistant during DEP enrollment via the `AccountConfiguration` command. **Availability:** Available in OS X 10.11 and later and the response is only generated by devices enrolled in MDM via DEP (see Device Enrollment Program (page 105)). Each dictionary in the array contains two keys: a key `GUID` with a string value of the Global Unique Identifier of a local admin account, and a key `shortName` with a string value of the short name of the admin account.

iTunesStoreAccountIsActive Commands Tell Whether an iTunes Account Is Logged In

The queries 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.
`iTunesStoreAccount-Hash`	String	Returns a hash of the iTunes Store account currently logged in. This string is identical to the `itsIdHash` returned by the VPP App Assignment web service. **Availability:** Available in iOS 8.0 and later and OS X 10.10 and later.

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 iOS device name or the OS X hostname.
`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).
`AvailableDevice-Capacity`	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.
`CellularTechnology`	Number	Returns the type of cellular technology. `0`: none `1`: GSM `2`: CDMA `3`: both **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.

Query	Reply Type	Comment
`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. **Availability:** Available in iOS 7 and later and OS X 10.9 and later.
`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 Apple TV software 6.0 and later only.
`EASDeviceIdentifier`	String	The Device Identifier string reported to Exchange Active Sync (EAS). **Availability:** Available in iOS 7 and later and OS X 10.9 and later.
`IsCloudBackupEnabled`	Boolean	If true, the device has iCloud backup enabled. **Availability:** Available in iOS 7.1 and later.
`OSUpdateSettings`	Dictionary	Returns the OS Update settings (see Table 8 (page 45)). **Availability:** Available in OS X 10.11 and later.
`LocalHostName`	String	Returns the local host name as reported by Bonjour. **Availability:** Available in OS X 10.11 and later.
`HostName`	String	Returns the host name. **Availability:** Available in OS X 10.11 and later.
`SystemIntegrity-ProtectionEnabled`	Boolean	**Availability:** Available in macOS 10.11 and later. Whether System Integrity Protection is enabled on the device.

Query	Reply Type	Comment
`ActiveManagedUsers`	Array of strings	Returns an array of the directory GUIDs (as strings) of the logged-in managed users. This query can be sent only to a device. An additional key, `CurrentConsoleManagedUser`, is sent in the reply; its string value is the GUID of the managed user active on the console. If no user listed in the `ActiveManagedUsers` array is currently active on the console, this additional key is omitted from the reply. **Availability:** Available in OS X 10.11 and later.
`IsMDMLostModeEnabled`	Boolean	If true, the device has MDM Lost Mode enabled. Defaults to false. **Availability:** Available in iOS 9.3 and later.
`MaximumResidentUsers`	Integer	Returns the maximum number of users that can use this Shared iPad mode device. **Availability:** Available in iOS 9.3 and later.

Table 8 OS update settings

Key	Type	Content
`CatalogURL`	String	The URL to the software update catalog currently in use by the client.
`IsDefaultCatalog`	Bool	
`PreviousScanDate`	Date	
`PreviousScanResult`	String	
`PerformPeriodicCheck`	Bool	
`AutomaticCheckEnabled`	Bool	
`BackgroundDownloadEnabled`	Bool	
`AutomaticAppInstallationEnabled`	Bool	
`AutomaticOSInstallationEnabled`	Bool	
`AutomaticSecurityUpdatesEnabled`	Bool	

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

The queries in Table 9 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 responds only to `BluetoothMAC`, `WiFiMAC`, and `EthernetMACs`.

Table 9 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.
`PersonalHotspot-Enabled`	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 10 (page 49).
`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.
`PasscodeLockGracePeriod`	Integer	The user preference for the amount of time in seconds the device must be locked before unlock will require the device passcode.
`PasscodeLockGrace-PeriodEnforced`	Integer	The current enforced value for the amount of time in seconds the device must be locked before unlock will require the device passcode.
`FDE_Enabled`	Boolean	**Availability:** Available in macOS 10.9 and later. Device channel only. Whether Full Disk Encryption (FDE) is enabled or not.
`FDE_HasPersonalRecoveryKey`	Boolean	**Availability:** Available in macOS 10.9 and later. Device channel only. If FDE has been enabled, returns whether a personal recovery key has been set.
`FDE_-HasInstitutionalRecoveryKey`	Boolean	**Availability:** Available in macOS 10.9 and later. Device channel only. If FDE has been enabled, returns whether an institutional recovery key has been set.

Key	Type	Content
`FirewallSettings`	Dictionary	(macOS 10.12 and later): the current Firewall settings. This information will be returned only when the command is sent to the device channel. The response is a dictionary with the following keys: • `FirewallEnabled` (Boolean): Whether firewall is on or off. • `BlockAllIncoming` (Boolean): Whether all incoming connections are blocked. • `StealthMode` (Boolean): Whether stealth mode is enabled. • `Applications` (array of dictionaries): Blocking status for specific applications. Each dictionary contains these keys: ◦ `BundleID` (string) : identifies the application ◦ `Allowed` (Boolean) : specifies whether or not incoming connections are allowed ◦ `Name` (string) : descriptive name of the application for display purposes only (may be missing if no corresponding app is found on the client computer).
`SystemIntegrity-ProtectionEnabled`	Boolean	**Availability:** Available in macOS 10.9 and later. Device channel only. Whether System Integrity Protection is enabled on the device. In macOS 10.11 or later, this information may also be retrieved using a `DeviceInformation` query.

Hardware encryption capabilities are described using the logical OR of the values in Table 10. Bits set to `1` (one) indicate that the corresponding feature is present, enabled, or in effect.

Table 10 HardwareEncryptionCaps bitfield values

Value	Feature
`1`	Block-level encryption.

Value	Feature
2	File-level encryption.

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

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

DeviceLock Command Locks the Device Immediately

The `DeviceLock` command is intended to lock lost devices remotely; it should not be used for other purposes. To send one, the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`DeviceLock`
`PIN`	String	The Find My Mac PIN. Must be 6 characters long. **Availability:** Available in OS X 10.8 and later.
`Message`	String	Optional. If provided, this message is displayed on the lock screen and should contain the words "lost iPad." 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.

If a passcode has been set on the device, the device is locked and the text and phone number passed with the `DeviceLock` command are displayed on the locked screen. The device returns a `Status` of `Acknowledged` and a `MessageResult` of `Success`. If a passcode has not been set on the device, the device is locked but the message and phone number are not displayed on the screen. The device returns a `Status` of `Acknowledged` and a `MessageResult` of `NoPasscodeSet`.

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 15) check-in message.

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

Note: The OS X MDM client generates 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 6 characters long. **Availability:** Available in OS X 10.8 and later.

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.
`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 reports 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	A 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.
`intersection`	Dictionary	A dictionary of intersected restrictions.
`union`	Dictionary	A dictionary of unioned 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 Key Reference.

Each entry in the dictionary contains the following keys:

Key	Type	Content
restriction_name	Dictionary	Restriction parameters.

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.

The `intersection` and `union` dictionaries have the following keys:

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

The restriction names (keys) in the dictionary correspond to the keys in the Restriction and Passcode Policy payloads.

Each entry in the dictionary contains the following keys:

Key	Type	Content
`values`	Array of strings	The values of the restriction.

With intersected restrictions, new restrictions can only reduce the number of strings in the set. With unioned restrictions, new restrictions can add to the set.

Clear Restrictions Password

The `ClearRestrictionsPassword` command allows the server to clear the restrictions password and restrictions set by the user on the device. Supervised only. **Availability:** Available in iOS 8 and later.

Key	Type	Content
`RequestType`	String	`ClearRestrictionsPassword`.

Shared iPad User Commands Manage User Access

Three MDM Protocol commands—`UsersList`, `LogOutUser`, and `DeleteUser`—let the MDM server exercise control over the access of users to MDM devices in an educational environment. These commands are all available in iOS 9.3 and later and may be used only in Shared iPad mode.

UsersList

This command allows the server to query for a list of users that have active accounts on the current device.

Key	Type	Content
`RequestType`	String	`UserList`.

The device replies with either an error response of code 12070 if the device cannot return a list of users, or the following response dictionary:

Key	Type	Content
`Users`	Array	Array of dictionaries containing information about active users.

Each entry in the Users array contains the following dictionary:

Key	Type	Content
`UserName`	String	The user name of the user.
`HasDataToSync`	Boolean	Whether the user has data that still needs to be synchronized to the cloud.
`DataQuota`	Integer	The data quota set for the user in bytes.
`DataUsed`	Integer	The amount of data used by the user in bytes.
`IsLoggedIn`	Boolean	If `true`, the user is currently logged onto the device.

LogOutUser

This command allows the server to force the current user to log out.

Key	Type	Content
`RequestType`	String	`LogOutUser`.

DeleteUser

This command allows the server to delete a user that has an active account on the device.

Key	Type	Content
`RequestType`	String	`DeleteUser`.
`UserName`	String	Required. The user name of the user to delete.
`ForceDeletion`	Boolean	Optional. Whether the user should be deleted even if they have data that needs to be synced to the cloud. Defaults to false.

The status of the response to `DeleteUser` is either Acknowledged, or Error with code 12071 if the specified user does not exist, 12072 if the specified user is logged in, 12073 if the specified user has data to sync and `ForceDeletion` is false or not specified, or 12074 if the specified user could not be deleted.

MDM Lost Mode Helps Lock and Locate Lost Devices

Three MDM Protocol commands—`EnableLostMode`, `DisableLostMode`, and `DeviceLocation`—let the MDM server help locate supervised devices when they are lost or stolen. These commands are all available in iOS 9.3 and later and may be used only in supervised mode.

When a device is erased, Lost Mode is disabled. To re-enable Lost Mode on the device, the MDM server should store the device's Lost Mode state before erasing it. If the device is enrolled again, the MDM server can then restore the correct Lost Mode state.

EnableLostMode

This command allows the server to put the device in MDM lost mode, with a message, phone number, and footnote text. A message or phone number must be provided.

Key	Type	Content
`RequestType`	String	`EnableLostMode`.
`Message`	String	Required if `PhoneNumber` is not provided; otherwise optional. If provided, this message is displayed on the lock screen.

Key	Type	Content
`PhoneNumber`	String	Required if `Message` is not provided; otherwise optional. If provided, this phone number is displayed on the lock screen.
`Footnote`	String	Optional. If provided, this footnote text is displayed in place of “Slide to Unlock.”

The response status is either Acknowledged or it is Error with code 12066 if MDM Lost Mode could not be enabled.

DisableLostMode

This command allows the server to take the device out of MDM lost mode.

Key	Type	Content
`RequestType`	String	`DisableLostMode`.

The response status is either Acknowledged or it is Error with code 12069 if MDM Lost Mode could not be disabled.

DeviceLocation

This command allows the server to ask the device to report its location if it is in MDM lost mode.

Key	Type	Content
`RequestType`	String	`DeviceLocation`.

The device replies with either an error response with code 12067 if the device is not in MDM Lost Mode, code 12068 if the location could not be determined, or the following response dictionary:

Key	Type	Content
`Latitude`	Double	The latitude of the device’s current location.
`Longitude`	Double	The longitude of the device’s current location.

Managed Applications

Running iOS 5 and later, 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 76) for details.

On devices running iOS earlier than iOS 9, apps from the App Store cannot be installed on a user's device if the App Store has been disabled. With iOS 9 and later, VPP apps can be installed even when the App Store is disabled (see VPP App Assignment (page 151)).

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 (that is, if the original `InstallApplication` request's `Options` dictionary contained a `PurchaseMethod` value of 0), 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/.

The OS X MDM client does not support managed 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`.
`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.

Key	Type	Content
`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: In iOS 7 and later, this URL and the URLs of any assets specified in the manifest 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 76).
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 78).
`ChangeManagement-State`	String	Optional. Currently the only supported value is the following: `Managed`: Take management of this app if the user has installed it already. Available in iOS 9 and later.

If the application is not already installed and the `ChangeManagementState` is set to `Managed`, the app will be installed and managed. If the application is installed unmanaged, the user will be prompted to allow management of the app on unsupervised devices and, if accepted, the application becomes managed.

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: Legacy Volume Purchase Program (iOS only) 1: Volume Purchase Program App Assignment

iOS App Installation

Here is an example of an iOS `InstallApplication` command for a per-device VPP app that uses the `ChangeManagementState` option:

```
<?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>ChangeManagementState</key>
     <string>Managed</string>
   <key>ManagementFlags</key>
     <integer>1</integer>
   <key>Options</key>
    <dict>
     <key>PurchaseMethod</key>
     <integer>1</integer>
    </dict>
   <key>RequestType</key>
     <string>InstallApplication</string>
   <key>iTunesStoreID</key>
     <integer>361309726</integer>
</dict>
</plist>
```

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)

OS X App Installation

OS X apps are installed through MDM as packages. Using `productbuild`, each package must be signed with an appropriate certificate (such as a TLS/SSL certificate with signing usage) and must be md5 hashed into 10 MB chunks. Only the package needs to be signed, not the app; Apple's Gatekeeper doesn't check apps installed through MDM.

The command lines to install an OS X app package should look like this:

```
$ sudo pkgbuild —component ~/Desktop/MyApp.app —install—location /Applications
 —sign myserver.myenterprise.com /tmp/myPackage.pkg
$ split —b 10485760 myPackage.pkg myPackage.pkg.
$ md5 —r myPackage.pkg.*
```

The manifest file included in the foregoing installation should contain:

the (HTTP) path to the package

the (HTTP) path to the display icons

the md5 hash size (10 MB as defined by CommerceKit)

the md5 hash information

the size of the download (package) in bytes

a unique bundle identifier to identify the package

bundle identifiers describing the items inside the package

descriptive titles for display purposes

The following lists a typical `Manifest.plist` file:

```
<?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>items</key>
   <array>
    <dict>
     <key>assets</key>
     <array>
      <dict>
       <key>kind</key>
       <string>software-package</string>
       <key>md5-size</key>
       <integer>10485760</integer>
       <key>md5s</key>
       <array>
        <string>d519a84e907a088f7e77381e8ce265e5</string>
        <string>0c4ea856a1b18ea7e24124d41fad3cc1</string>
        <string>5a6b17332bf258e77956ac0d7a69ff8a</string>
       </array>
       <key>url</key>
       <string>http://myserver.myenterprise.com/MDM_Test/MyApp.pkg</string>
      </dict>
      <dict>
       <key>kind</key>
       <string>full-size-image</string>
       <key>needs-shine</key>
       <false/>
       <key>url</key>
       <string>http://myserver.myenterprise.com/MDM_Test/Server.png</string>
      </dict>
      <dict>
```

```
        <key>kind</key>
        <string>display-image</string>
        <key>needs-shine</key>
        <false/>
        <key>url</key>
        <string>http://myserver.myenterprise.com/MDM_Test/Server.png</string>
       </dict>
      </array>
      <key>metadata</key>
      <dict>
       <key>bundle-identifier</key>
       <string>com.myenterprise.MyAppPackage</string>
       <key>bundle-version</key>
       <string>1.0</string>
       <key>items</key>
       <array>
        <dict>
         <key>bundle-identifier</key>
         <string>com.myenterprise.MyAppNotMAS</string>
         <key>bundle-version</key>
         <string>1.7.4</string>
        </dict>
       </array>
       <key>kind</key>
       <string>software</string>
       <key>sizeInBytes</key>
       <string>26613453</string>
       <key>subtitle</key>
       <string>My Enterprise</string>
       <key>title</key>
       <string>Example Enterprise Install</string>
      </dict>
     </dict>
    </array>
</dict>
```

```
</plist>
```

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; see Table 11 (page 65) for possible values.
`ManagementFlags`	Integer	Management flags. (See InstallApplication command above for a list of flags.)
`UnusedRedemptionCode`	String	If the user has already purchased a paid app, the unused redemption code is reported here. This code can be used again to purchase the app for someone else. This code is reported only once.
HasConfiguration	Boolean	If `true`, the app has a server-provided configuration. For details, see Managed App Configuration and Feedback (page 76). 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 76). Available in iOS 7 and later.
IsValidated	Boolean	If `true`, the app has validated as allowed to run and is able to run on the device. If an app is enterprise-distributed and is not validated, it will not run on the device until validated. Available in iOS 9.2.1 and later.

Table 11 Managed app statuses

Value	Description
`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.
`PromptingForLogin`	The user is being prompted for App Store credentials.

Value	Description
`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.
`PromptingForManagement`	The user is being prompted to change an installed app to be managed.
`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.
`ManagementRejected`	The user rejected management of an already installed app.
`Failed`	The app installation has failed.

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`.

Key	Type	Content
`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 the Volume Purchase Program for per-user VPP app assignment. After this command issues an invitation, you can use the `iTunesStoreAccountIsActive` query to get the hash of the iTunes Store account currently logged in.

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
`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`

This command yields a `NotNow` status until the user exits Setup Assistant.

ValidateApplications Verifies Application Provisioning Profiles

This command allows the server to force validation of the free developer and universal provisioning profiles associated with an enterprise app. **Availability:** Available in iOS 9.2 and later.

Key	Type	Content
`RequestType`	String	`ValidateApplications`.
`Identifiers`	Array of Strings	Optional. An array of app identifiers. If provided, the enterprise apps whose identifiers appear in this array have their provisioning profiles validated. If not, only installed managed apps have their provisioning profiles validated.

Installed Books

Books obtained from the iBooks Store can be installed on a device. These books will be backed up, will sync to iTunes, and will remain after the MDM profile is removed. Books not obtained from the iBooks Store will not sync to iTunes and will be removed when the MDM profile is removed.

Books obtained from the iBooks Store must be purchased using VPP Licensing. Installing a book from the iBooks Store on a device that already has that book installed causes the book to be visible to the MDM server.

Installation of books requires the App Installation right. The App Store must be enabled for iBooks Store media installation to work. The App Store need not be enabled to install books retrieved using a URL.

InstallMedia Installs a Book onto a Device

To send an `InstallMedia` command (in iOS 8 or later), the server sends a dictionary containing the following keys:

Key	Type	Content
`RequestType`	String	`InstallMedia`.
`iTunesStoreID`	Integer	Optional. The media's iTunes Store ID.
`MediaURL`	String	Optional; not supported in OS X. The URL from which the media will be retrieved.
`MediaType`	String	`Book`.

The request must contain either an `iTunesStoreID` or a `MediaURL`.

If a `MediaURL` is provided, the URL must lead to a PDF, gzipped epub, or gzipped iBooks document. The following fields are provided to define this document:

Key	Type	Content
`PersistentID`	String	Persistent ID in reverse-DNS form, e.g., `com.acme.manuals.training`.
`Kind`	String	Optional. The media kind. Must be one of the following: • `pdf`: PDF file • `epub`: A gzipped epub • `ibooks`: A gzipped iBooks Author-exported book If this field is not provided, the file extension in the URL is used.
`Version`	String	Optional. A version string that is meaningful to the MDM server.
`Author`	String	Optional.
`Title`	String	Optional.

Installing a book not from the iBooks Store with the same `PersistentID` as an existing book not from the iBooks Store replaces the old book with the new. Installing an iBooks Store book with the same `iTunesStoreID` as an existing installed book updates the book from the iBooks Store.

The user is not prompted for book installation or update unless user interaction is needed to complete an iBooks Store transaction.

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

Key	Type	Content
`iTunesStoreID`	Integer	The book's iTunes Store ID, if it was provided in the command.
`MediaURL`	String	The book's URL, if it was provided in the command.
`PersistentID`	String	Persistent ID, if it was provided in the command.
`MediaType`	String	The media type.

Key	Type	Content

`State`	String	The installation state of this media. This value can be one of the following:
		`Queued`
		`PromptingForLogin`
		`Updating`
		`Installing`
		`Installed`
		`Uninstalled`
		`UserInstalled`
		`Rejected`
		
		The following states are transient and are reported only once:
		`Failed`
		`Unknown`

If the book cannot be installed, an `Error` status is returned, which may contain an error chain. In addition, a `RejectionReason` field of type `String` is returned, containing one of these values:

- `CouldNotVerifyITunesStoreID`
- `PurchaseNotFound`: No VPP license found in the user's history
- `AppStoreDisabled`
- `WrongMediaType`
- `DownloadInvalid`: URL doesn't lead to valid book

ManagedMediaList Returns a List of Installed Media on a Device

To send a `ManagedMediaList` command, the server sends a dictionary containing the following key:

Key	Type	Content
`RequestType`	String	`ManagedMediaList`.

If the request is accepted, the device responds with an `Acknowledged` response and the following field:

Key	Type	Content
Books	Array	Array of dictionaries.

Each entry in the `ManagedMedia` array is a dictionary with the following keys:

Key	Type	Content
iTunesStoreID	Integer	The item's iTunes Store ID, if the item was retrieved from the iTunes Store.
State	String	The installation state of this media. This value can be one of the following: `Queued` `PromptingForLogin` `Updating` `Installing` `Installed` `Uninstalled` `UserInstalled` `Rejected`
PersistentID	String	Provided if available.
Kind	String	Provided if available.
Version	String	Provided if available.
Author	String	Provided if available.
Title	String	Provided if available.

RemoveMedia Removes a Piece of Installed Media

This command allows an MDM server to remove installed media. This command returns `Acknowledged` if the item is not found.

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

Key	Type	Content
`RequestType`	String	`RemoveMedia`.
`MediaType`	String	`Book`.
`iTunesStoreID`	Integer	Optional. iTunes Store ID.
`PersistentID`	String	Optional. Persistent ID of the item to remove.

Upon success, an `Acknowledged` status is returned. Otherwise, an error status is returned.

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 (in iOS 8 or later) 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 are 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.

DeviceName Sets the Name of the Device

To send a `DeviceName` command (available only on supervised devices or devices running OS X v10.10 or later), the server sends a dictionary containing the following keys:

Key	Type	Content
`Item`	String	`DeviceName`.
`DeviceName`	String	The requested name for the device.

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.

MaximumResidentUsers

Shared iPad Mode only. Sets the maximum number of users that can use a shared iPad. This can be set only when the iPad is in the `AwaitingConfiguration` phase, before the `DeviceConfigured` message has been sent to the device. If `MaximumResidentUsers` is greater than the maximum possible number of users supported on the device, the device is configured with the maximum possible number of users instead.

Key	Type	Content
`Item`	String	`MaximumResidentUsers`.
`MaximumResidentUsers`	Integer	The maximum number of users that can use a Shared iPad.

Availability: Available in iOS 9.3 and later.

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 bundle 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 bundle 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 application's bundle identifier.

Key	Type	Content
`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 bundle 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 bundle 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 bundle 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 bundle identifier.
`Feedback`	Dictionary	Optional. The current feedback dictionary. If the app has no feedback dictionary, this key is absent.

AccountConfiguration

When an OS X (v10.11 and later) device is configured via DEP to enroll in an MDM server and the DEP profile has the `await_device_configuration` flag set to true, the `AccountConfiguration` command can be sent to the device to have it create the local administrator account (thereby skipping the page to create this account in Setup Assistant). This command can only be sent to an OS X device that is in the `AwaitingConfiguration` state.

The `AccountConfiguration` command replaces the `SetupConfiguration` command, which is deprecated. While both commands remain supported, new software should use `AccountConfiguration`.

Key	Type	Content
`SkipPrimarySetup-AccountCreation`	Boolean	(Optional, default=false). If true, skip the UI for setting up the primary accounts. Setting this key to true requires that an entry be specified in `AutoSetupAdminAccounts`. Setting this value to true also prevents auto login after Setup Assistant completes.
`SetPrimarySetup-AccountAsRegularUser`	Boolean	(Optional, default=false). If true, the primary accounts are created as regular users. Setting this to true requires that an entry be specified in `AutoSetupAdmin-Accounts`.
`AutoSetupAdmin-Accounts`	Array of Dictionaries	(Required if either of the above options are true) Describes the admin accounts to be created by Setup Assistant (see below). Currently, OS X creates only a single admin account. Array elements after the first are ignored.
`SetAutoAdminPassword`	Array of Dictionaries	(Optional; available in OS X v10.11 and later) Allows changing the password of a local admin account that was created by Setup Assistant during DEP enrollment via the `AccountConfiguration` command.

The `AutoSetupAdminAccounts` dictionaries contain the specifications of local administrator accounts to be created before Setup Assistant finishes:

Key	Type	Content
`shortName`	String	The short name of the user.
`fullName`	String	(Optional) string of full user name. This defaults to `shortName` if not specified.
`passwordHash`	Data	Contains the pre-created salted PBKDF2 SHA512 password hash for the account (see below).
`hidden`	Boolean	(Optional, default=false) If true, this sets the account attribute to make the account hidden to `loginwindow` and Users&Groups. OD attribute: `dsAttrTypeNative: IsHidden`.

The `SetAutoAdminPassword` dictionaries contain these keys:

Key	Type	Content
GUID	String	The Globally Unique Identifier of the local admin account for which the password is to be changed. If this string does not correspond to the GUID of an admin account created during DEP enrollment, the command returns an error.
passwordHash	Data	Contains the pre-created salted PBKDF2 SHA512 password hash for the account (see below).

The `passwordHash` data objects should be created on the server using the CommonCrypto libraries or equivalent as a salted SHA512 PBKDF2 dictionary containing three items: `entropy` is the derived key from the password hash (an example is from `CCKeyDerivationPBKDF()`), `salt` is the 32 byte randomized salt (from `CCRandomCopyBytes()`), and `iterations` contains the number of iterations (from `CCCalibratePBKDF()`) using a minimum hash time of 100 milliseconds (or if not known, a number in the range 20,000 to 40,000 iterations). This dictionary of the three keys should be placed into an outer dictionary under the key `SALTED-SHA512-PBKDF2` and converted to binary data before being set into the configuration dictionary `passwordHash` key value.

DeviceConfigured

`DeviceConfigured` informs the device that it can continue past DEP enrollment. It works only on devices in DEP that have their cloud configuration set to await configuration.

Key	Type	Content
RequestType	String	`DeviceConfigured`.

Software Update

The Software Update commands allow an MDM server to perform software updates. In OS X, a variety of system software can be updated. In iOS, only OS updates are supported.

Note: If the device has a passcode, it must be cleared before an iOS update is performed.

Only Supervised DEP-enrolled iOS devices and DEP-managed Mac computers are eligible for software update management. However, the `AvailableOSUpdates` query is available to non-DEP managed devices.

The MDM server must have the App Installation right to perform these commands.

ScheduleOSUpdate

Key	Type	Content
`RequestType`	String	`ScheduleOSUpdate`.
Updates	Array	An array of dictionaries specifying the OS updates to download or install. If this entry is missing, the device applies the default behavior for all available updates.

The Updates array contains dictionaries with the following keys and values:

Key	Type	Content
`ProductKey`	String	The product key of the update to be installed.
`InstallAction`	String	One of the following: • `Default`: Download and install the software update. • `DownloadOnly`: Download the software update without installing it. • `InstallASAP`: Install an already downloaded software update. • `NotifyOnly`: Download the software update and notify the user via the App Store (OS X only). • `InstallLater`: Download the software update and install it at a later time (OS X only).

The device returns the following response:

Key	Type	Content
`UpdateResults`	Array	Array of dictionaries.

The `UpdateResults` dictionary contains the following keys and values:

Key	Type	Content
`ProductKey`	String	The product key.

Key	Type	Content
`InstallAction`	String	The install action that the device has scheduled for this update. One of the following: • `Error`: An error occurred during scheduling. • `DownloadOnly`: Download the software update without installing it. • `InstallASAP`: Install an already downloaded software update. • `NotifyOnly`: Download the software update and notify the user via the App Store (OS X only). • `InstallLater`: Download the software update and install it at a later time (OS X only).
`Status`	String	The status of the software update. Possible values are: • `Idle`: No action is being taken on this software update. • `Downloading`: The software update is being downloaded. • `DownloadFailed`: The download has failed. • `DownloadRequiresComputer`: The device must be connected to a computer to download this update (iOS only). • `DownloadInsufficientSpace`: There is not enough space to download the update. • `DownloadInsufficientPower`: There is not enough power to download the update. • `DownloadInsufficientNetwork`: There is insufficient network capacity to download the update. • `Installing`: The software update is being installed. • `InstallInsufficientSpace`: There is not enough space to install the update. • `InstallInsufficientPower`: There is not enough power to install the update. • `InstallPhoneCallInProgress`: Installation has been rejected because a phone call is in progress. • `InstallFailed`: Installation has failed for an unspecified reason.
`ErrorChain`	Array	Array of dictionaries describing the error that occurred.

The device may return a different `InstallAction` than the one that was requested.

Note: Responding to the `ScheduleOSUpdate` request currently relies on the device not being passcode-protected.

Because software updates may happen immediately, the device may not have the opportunity to respond to an installation command before it restarts for installation. When this happens, the MDM server should resend the `ScheduleOSUpdate` request when the device checks in again. The device returns a status of `Idle` because the update has been installed and is no longer applicable.

ScheduleOSUpdateScan

`ScheduleOSUpdateScan` requests that the device perform a background scan for OS updates.

Key	Type	Content
`RequestType`	String	`ScheduleOSUpdateScan`.
`Force`	Boolean	If set to `true`, force a scan to start immediately. Otherwise, the scan occurs at a system-determined time. Defaults to `false`.

The device returns the following response:

Key	Type	Content
`ScanInitiated`	Boolean	Returns true if the scan was successfully initiated (OS X only).

This command is needed by OS X only. iOS devices respond with an `Acknowledged` status on success.

AvailableOSUpdates

`AvailableOSUpdates` queries the device for a list of available OS updates. In OS X, a `ScheduleOSUpdateScan` must be performed to update the results returned by this query.

Key	Type	Content
`RequestType`	String	`AvailableOSUpdates`.

The device returns the following dictionary:

Key	Type	Content
`AvailableOSUpdates`	Array	Array of dictionaries.

Each element in the AvailableOSUpdates array contains a dictionary with the following keys and values:

Key	Type	Content
`ProductKey`	String	The product key that represents this update.
`HumanReadableName`	String	The human-readable name of the software update, in the current user's current locale.
`ProductName`	String	The product name: e.g., iOS.
`Version`	String	The version of the update: e.g., 9.0.
`Build`	String	The build number of the update: e.g., 13A999.
`DownloadSize`	Number	Storage size needed to download the software update. Floating point number of bytes.
`InstallSize`	Number	Storage size needed to install the software update. Floating point number of bytes.
`AppIdentifiersToClose`	Array	Array of strings. Each entry represents an app identifier that is closed to install this update (OS X only).
`IsCritical`	Boolean	Set to `true` if this update is considered critical. Defaults to `false`.
`IsConfiguration-DataUpdate`	Boolean	Set to `true` if this is an update to a configuration file. Defaults to false (OS X only).
`IsFirmwareUpdate`	Boolean	Set to `true` if this is an update to firmware. Defaults to `false` (OS X only).
`RestartRequired`	Boolean	Set to `true` if the device restarts after this update is installed. Defaults to `false`.
`AllowsInstallLater`	Boolean	Set to `true` if the update is eligible for InstallLater. Defaults to `true`.

A total of `DownloadSize + InstallSize` bytes is needed to successfully install a software update.

OSUpdateStatus

`OSUpdateStatus` queries the device for the status of software updates.

Key	Type	Content
`RequestType`	String	`OSUpdateStatus`.

The device responds with the following dictionary:

Key	Type	Content
`OSUpdateStatus`.	Array	Array of dictionaries.

Each entry in the `OSUpdateStatus` array is a dictionary with the following keys and values:

Key	Type	Content
`ProductKey`	String	The product key.
`IsDownloaded`	Boolean	Set to `true` if the update has been downloaded.
`DownloadPercent-Complete`	Number	Percentage of download that is complete. Floating point number (0.0 to 1.0).
`Status`	String	The status of this update. Possible values are: • `Idle`: No action is being taken on this software update. • `Downloading`: The software update is being downloaded. • `Installing`: The software update is being installed. This status may not be returned if the device must reboot during installation.

Support for OS X Requests

The table below 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 in the rest of this chapter.

Command	Min OS	User/Device	Comments
AccountConfiguration	10.11	Device	Valid only during DEP enrollment.
AvailableOSUpdates	10.11	Device	

Command	Min OS	User/Device	Comments
CertificateList	10.7	Both	
DeviceConfigured	10.11	Both	Valid only during DEP enrollment.
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.
	10.11	Device	**DeviceInfo:** Added ActiveManagedUsers.
DeviceLock	10.7	Device	
EraseDevice	10.7	Device	
InstallApplication	10.9	User	For VPP (`iTunesStoreID`, `Identifier`).
	10.10	Device	`ManifestURL`.
	10.11	Both	
InstalledApplicationList	10.7	Both	
InstallProfile	10.7	Both	
InviteToProgram	10.9	Both	
OSUpdateStatus	n/a	n/a	Not yet supported in OS X.
ProfileList	10.7	Both	
ProvisioningProfileList	10.7	Both	Supported, but always returns empty list.
RemoveProfile	10.7	Both	
Restrictions	10.7	Both	Supported, but always returns empty list.
ScheduleOSUpdate	10.11	Device	Requires DEP enrolled computer.

Command	Min OS	User/Device	Comments
ScheduleOSUpdateScan	10.11	Device	
SecurityInfo	10.7	Both	Supported, but always returns empty list.
	10.9	Device	Returns FileVault state in FDE_Enabled, FDE_HasPersonalRecovery Key, FDE_HasInstitutionalRecoveryKey.
SetAutoAdminPassword	10.11	User	
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
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

Code	Meaning
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
2005	Internal Error

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

Code	Meaning
4006	Profile removal date is in the past
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
4020	UI installation prohibited
4021	Profile must be installed non-interactively
4022	Profile must be installed using MDM
4023	Unacceptable payload
4024	Profile not found
4025	Invalid supervision
4026	Removal date in the past
4027	Profile requires passcode change
4028	Multiple home screen layout payloads
4029	Multiple notification settings layout payloads

Code	Meaning
4030	Unacceptable payload in ephemeral multi-user
4031	Payload contains sensitive user information

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
5016	Cannot set fingerprint unlock
5017	Cannot set fingerprint purchase
5018	Cannot set maximum failed passcode attempts

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	The iTunes Store ID of the application could not be validated
12024	Could not validate app manifest
12025	App already installed
12026	License for app "<app bundle ID>" could not be found
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
12043	Media download failure
12044	Invalid media type
12045	Invalid media replacement type
12046	Cannot validate media ID
12047	Cannot find VPP assignment
12048	No update available
12049	Device passcode must be cleared
12050	Update scan failed
12051	Update download in progress
12052	Update download complete

Code	Meaning
12053	Update download requires computer
12054	Insufficient space for update download
12055	Insufficient power for update download
12056	Insufficient network for update download
12057	Update download failed
12058	Update install in progress
12059	Update install requires download
12060	Insufficient space for update install
12061	Insufficient power for update install
12062	Update install failed
12063	User rejected
12064	License not found
12065	System app
12066	Could not enable MDM lost mode
12067	Device not in MDM lost mode
12068	Could not determine device location
12069	Could not disable MDM lost mode
12070	Cannot list users
12071	Specified user does not exist
12072	Specified user is logged in
12073	Specified user has data to sync
12074	Could not delete user
12075	Specified profile not installed
12076	Per-user connections not supported

MCWiFiErrorDomain

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

MCTunnelErrorDomain

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
15004	Internal error
15005	Cannot parse VPN payload

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
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

Code	Meaning
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
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
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

Code	Meaning
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
34004	Kerberos identity certificate cannot be found

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
36001	Internal error

MCKeybagErrorDomain

Code	Meaning
37000	Internal error
37001	Internal error

MCDomainsErrorDomain

Code	Meaning
38000	Invalid domain matching pattern

MCWebContentFilterErrorDomain

Code	Meaning
40000	Internal error
40001	Invalid certificate

MCNetworkUsageRulesErrorDomain

Code	Meaning
41000	Internal error
41001	Invalid configuration
41002	Internal error

MCOSXServerErrorDomain

Code	Meaning
42000	Cannot create account
42001	No hostname
42002	Account not unique

MCHomeScreenLayoutErrorDomain

Code	Meaning
43000	Multiple Home screen layouts

MCNotificationSettingsErrorDomain

Code	Meaning
44000	Multiple notification settings

MCEDUClassroomErrorDomain

Code	Meaning
45000	Cannot install
45001	Student already installed
45002	Cannot find certificate
45003	Bad identity certificate

MCSharedDeviceConfigurationErrorDomain

Code	Meaning
46000	Multiple shared device configurations

Device Enrollment Program

In iOS 7 and later and OS X v10.9 and later, the Device Enrollment Program (DEP) 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.

> **Note:** The Device Enrollment Program API is being upgraded to X-Server-Protocol-Version 2. X-Server-Protocol-Version 1 will continue to be supported as a default. The Web Services Header specified in Web Services (page 111) should be passed with all requests, because the default X-Server-Protocol-Version may change in the future.

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.

DEP Server Tokens

The MDM Device Enrollment Program (DEP) uses a server token to allow an MDM server to securely connect to the DEP web service.

Obtaining a Server Token

To obtain a DEP server token, the user must complete the steps outlined below. Your MDM server product can help by automating specific steps.

1. Generate a public/private key pair in PEM format for the MDM server, and store the private key securely on the server.
2. The user then must:
    a. Sign into the Device Enrollment Program web portal.
    b. Create a new virtual MDM server.
    c. Upload a PEM-encoded X.509 certificate containing the PEM public key that was generated in Step 1.
    d. Download the S/MIME encrypted token file generated by the program web portal.
3. Decrypt the S/MIME encrypted server token.
4. Upload the token file to the MDM server.

Using DEP Server Tokens

DEP server tokens can be deployed either automatically or manually.

Automatically

The MDM (physical) server must automatically decrypt this token file when it's uploaded into the system, using the private key for the DEP web service.

Manually

Use the private key and provide an S/MIME encryption utility to manually decrypt the encrypted token file before it is uploaded to the MDM server. The MDM server then ingests a plain text token file for use with the DEP web service.

Server Token Example

Following is a S/MIME encrypted server token:

```
Content-Type: application/pkcs7-mime; name="smime.p7m"; smime-type=enveloped-
data
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="smime.p7m"
Content-Description: S/MIME Encrypted Message

MIAGCSqGSIb3DQEHA6CAMIACAQAxggGeMIIBmgIBADCBgTB1MQswCQYDVQQGEwJVUzESMBAGA1UE
ChMJWmlwcG8gSW5jMSgwJgYDVQQDEx9Qcm9maWxlIE1hbmFnZXIgUy9NSU1FIElkZW50aXR5MSgw
JgYJKoZIhvcNAQkBFhlsb2NhbEB6aXBwb2luYzIuYXBwbGUuY29tAgiDS17MvQ95HDANBgkqhkiG
9w0BAQEFAASCAQC/ukglifm8tk/OjyKBWwPbm+uDNHPG+sXLRrwfTlHKRo1jnvYrqKx1bRrpV/GR
mN7WJPBZLOkFat+LoiEmrBUiUs3PnZ+U1FUAnHR66hnomKoX0JBgfuHBGYz9jeyiu1chQShgdOOe
bYQdaFPJ/P57r98yQ2ZmyqcYOWwE0lOcqa77bfRab/YmMsMx2ZE1wUwnFPM71Yq3+vLIGLBRyvAb
4pBxDlRtgGbxs+2gZwEe0MZ4tx/97RGnZbkJt/26v5P4njGiyCvq2hZUwbria7THhMEvmJRjpZNZ
x5BfTjU8a0EHwvvwnYb67LRnjoSMn/JgelRP70O9fhdZ5Y56xhs6MIAGCSqGSIb3DQEHATAdBglg
hkgBZQMEAQIEEF5d7PQ1O8lxOLjSwjNHwFaggASCAeC9GWg9EDLpyO2g6eoOmeIVYXbWXrRt4JRY
TqCB2dWDqc9BJqOYuX5lnULjvkJ8btlBfAMhUXUb/lFF5xNXGxLTtVHvyVK9FUyhikJFweRohWqM
/xtu+7/1rPT9Nmlssla9wcTAh8GsWbs9ZyM7Pnok+o1XOwRLgh1dGvW8EGxlaPWjcHolleFBStV6
lGKJUrUyzgyBvSWo/6Y/Ojb/kfzq/kzS6H7h4YZI69/Js604rpOL6FAeOwKaJLISfUUp/yNHMBr6
wj772MNnoIdVEQs14/Fk+XVDb4xghD1zzeDow+eseb+qEfY7FkgYi2jpdebk9X4BpJ1WGvy4WiA8
biyKpst6zJb0jdJ4TE0zyIcjuVeOXuV/cD1c7YrYQty1Sh3nBsjFwVOsHq33YjapcHf2wuhXW+hh
HNzpkyMKrNcsEK1HpJva2O6vBtxtYZIn5/4kGDeALUiXxVjtvio1gS37lry5YKEwhYJ+cKKe3exZ
xhLfD67AINahDm868kEuKuHIl8gku+gSKAWlUVGNrPNt/M2rM+y4+4cm23R2f3VXYuNncnFFbulF
7VQuGd3wwtKncIACU5rze4b366rRBG1PCvB7abuRcmw9UrgzkRlH8tbOhORZ0Dgimd5knujsbKMA
AAAAAAAAAAAA
```

Following is the decrypted server token in plain text:

```
Content-Type: text/plain;charset=UTF-8
Content-Transfer-Encoding: 7bit

{"consumer_key":"CK_9dcd8190dde27dfddd9272c657e011f7bec9761676b1b11e46a2f61d3c
b1e482ef22093c7d54b23252f3bdb4d19b4d49","consumer_secret":"CS_27c083df1ab7271e
129cb23325dabaf0de95d087","access_token":"AT_036558709508247e0b5288abcdee25642
311746d67b5858ea5c01389734861908","access_secret":"AS_8c3313de9a3462014c6c96f3
9dd7c3d4342b8cea","access_token_expiry":"2015-01-14T21:27:41Z"}
```

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 contains 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_021092790220e03b641fd6f07d7face7894211d521fd8bef09c30137392",
       oauth_signature_method="HMAC-SHA1",
       oauth_signature="w0JI09A2W5mFwDgiDvZbTSMK%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`	1, 2, or 3.
`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.

Note: Apple servers now run X-Server-Protocol-Version 2, which may include additional keys in the response body. Clients running X-Server-Protocol-Version 1 should be programmed to ignore these keys.

For example:

```
GET /account HTTP/1.1
User-Agent: ProfileManager-1.0
X-Server-Protocol-Version:2
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:3
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.
`admin_id`	Apple ID of the person who generated the current tokens that are in use.
`facilitator_id`	Legacy equivalent to the `admin_id` key. **This key is deprecated and may not be returned in future responses.**
`org_name`	The organization name.
`org_email`	The organization email address.

Key	Value
`org_phone`	The organization phone.
`org_address`	The organization address.
`urls`	The list of dictionaries (see below) containing URLs available in MDM service. This key is valid in X-Server-Protocol-Version 3 and later.
`org_type`	Possible values: `edu` or `org`. Available only in protocol version 3 and above.
`org_version`	Possible values: `v1` or `v2`. `v1` is for ADP organizations and `v2` is for ASM organizations. Currently `v2` is applicable only to educational organizations. This key is available only in protocol version 3 and above.

Each `url` dictionary contains the following keys:

Key	Value
`uri`	URI for the API.
`http_method`	Possible values: GET, POST, PUT, DELETE.
`limit`	Optional: Dictionary for limit parameter (see below).

Each `limit` dictionary contains the following keys:

Key	Value
`default`	Default value of limit.
`maximum`	Maximum value of limit.

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
X-Server-Protocol-Version: 3
Connection: close
```

```
{
     "server_name" : "IT Department Server",
     "server_uuid" : "677cab70-fe18-11e2-b778-0800200c9a66",
     "admin_id" : "facilitator1@example.com",
     "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",
     "urls" : [
       {"uri":"/account","http_method":["GET"]},
       {"uri":"/server/devices","http_method":["POST"],
         "limit":{"default":100,"maximum":1000}},
       {"uri":"/devices/sync","http_method":["POST"]},
         "limit":{"default":100,"maximum":1000}},
       {"uri":"/devices","http_method":["POST"]},
       {"uri":"/devices/disown","http_method":["POST"]},
       {"uri":"/profile","http_method":["POST"]},
       {"uri":"/profile/devices","http_method":["POST"]},
       {"uri":"/profile","http_method":["POST"]},
       {"uri":"/profile/devices","http_method":["GET"]},
       {"uri":"/profile/devices","http_method":["DELETE"]},
     ],
     "org_type":"edu",
     "org_version":"v2"
}
```

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:2
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"`, `"pushed"`, or `"removed"`.
`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.
`device_assigned_by`	The email of the person who assigned the device.

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",
                  "device_assigned_by" : "facilitator1@sampleinc.com"
              },
              {
                  "serial_number" : "C8TJ500QF1MN",
                  "model" : "IPAD",
                  "description" : "IPAD WI-FI 16GB",
                  "color" : "white",
                  "profile_status" : "assigned",
                  "profile_uuid" : "88fc4e378fea4021a94b2d7268fbf767",
                  "profile_assign_time" : "2013-05-01T00:00:00Z",
                  "device_assigned_date" : "2013-04-05T15:30:00Z",
                  "device_assigned_by" : "facilitator1@sampleinc.com"
              }
        ]
        "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 149), 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.

Key	Value
`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:2
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"`, `"pushed"`, or `"removed"`.
`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.
`op_type`	Indicates whether the device was added (assigned to the MDM server), modified, or deleted. Contains one of the following strings: `added`, `modified`, 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`.
`device_assigned_by`	The email 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.

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",
                    "device_assigned_by" : "facilitator1@sampleinc.com",
                    "device_assigned_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",
                    "device_assigned_by" : "facilitator1@sampleinc.com",
                    "device_assigned_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 149), 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:2
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`.
`os`	The device's operating system: `iOS` or `OSX`. This key is valid in X-Server-Protocol-Version 2 and later.
`device_family`	The device's Apple product family: `iPad`, `iPhone`, `iPod`, or `Mac`. This key is valid in X-Server-Protocol-Version 2 and later.
`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).
`device_assigned_by`	The email 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`, `pushed`, or `removed`. 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.
`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" :
      {
        "serial_number":"C8TJ500QF1MN",
        "response_status" : "SUCCESS",
        "os" : "iOS",
        "device_family" : "iPad",
        "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 149), 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. **This request is deprecated and may not be supported in the future.**

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:2
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 149), 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.

Activation Lock

Find My iPhone Activation Lock is a feature of iCloud that makes it harder for anyone to use or resell a lost or stolen iOS device that has been enrolled under DEP.

The Activation Lock request is available in X-Server-Protocol-Version 2 and later to organizations that have enrolled through the Apple School Manager portal.

Request

To lock a device, POST an HTTP request in application/json format to the following URL: `https://mdmenrollment.apple.com/device/activationlock`. The request header must follow this format:

```
POST /device/activationlock HTTP/1.1
User-Agent:<client-software-information>
X-Server-Protocol-Version: <Integer, 2 or higher>
X-ADM-Auth-Session:<AUTH-TOKEN>
Content-Type: application/json;charset=UTF8
Content-Length: <Content_Length>
...
```

Immediately following the request header, send these content keys and values in application/json format:

Key	Type	Content
`device`	String	Serial number of the device (required).
`escrow_key`	String	Escrow key (optional). If the escrow key is not provided, the device will be locked against the person who created the MDM server in the portal. For information about creating an escrow key see Escrow Keys and Bypass Codes (page 132).
`lost_message`	String	Lost message to be displayed on the device (optional).

A typical request might look like this:

```
POST /device/activationlock HTTP/1.1
```

```
User-Agent:ProfileManager-1.0
X-Server-Protocol-Version:2
Content-Type: application/json;charset=UTF8
Content-Length: 122
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
{
   "device": "C8TJ500QF1MN",
   "escrow_key": "30a3449822ae82b94f1839ee0248a9e2350247d4
                  b325071e6deb84285a6bfb34",
   "lost_message": "Please phone 1-800-555-1212"
}
```

Response

The Apple server responds to the Activation Lock request with the following two keys:

Key	Type	Content
`serial_number`	String	Serial number of the device.
`response_status`	String	`SUCCESS` or one of the failure responses listed below.

Activation lock failure responses include the following:

Response	Reason
`NOT_ACCESSIBLE`	A device with this serial number is not accessible by this user.
`ORG_NOT_SUPPORTED`	A device with this serial number is not supported because it is not present in the new program.
`DEVICE_NOT_SUPPORTED`	Device type is not supported like Mac.
`DEVICE_ALREADY_LOCKED`	Device is already locked by someone.
`FAILED`	Activation lock of the device failed for unexpected reason. If retry fails, the client should contact Apple support.

A successful activation lock response typically 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
{
     "serial_number" : "B7CJ500QF1MA",
     "response_status" : "SUCCESS"
}
```

Server failures during activation lock attempts typically look like one of the following two examples:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain;charset=UTF8
Content-Length: 0
Date: Thu, 31 May 2012 21:23:57 GMT
Connection: close

HTTP/1.1 503 Service Unavailable
Content-Type: text/plain;charset=UTF8
Retry-After: 120
Content-Length: 0
Date: Thu, 31 May 2012 21:23:57 GMT
Connection: close
```

A client failure during an activation lock attempt may look like this:

```
HTTP/1.1 4xx <Error Reason>
Content-Type: text/plain;Charset=UTF8
Content-Length: 10
Date: Thu, 31 May 2012 21:23:57 GMT
Connection: close

<ERROR_CODE>
```

The combination of the ERROR_CODE in the response body shown above and the HTTP error typically indicates one of the following reasons for a client failure during an activation lock attempt:

- UNAUTHORIZED + HTTP 401: The auth token has expired. The client should retry with a new auth token.
- FORBIDDEN + HTTP 403: The auth token is invalid.
- MALFORMED_REQUEST_BODY + HTTP 400: The request body is malformed.

Activation Lock Bypass

iOS 7.1 adds support for Activation Lock Bypass. This allows organizations to remove the Activation Lock from supervised devices prior to device activation without knowing the user's personal Apple ID and password.

When an iOS device is configured as supervised it can generate a device-specific Activation Lock bypass code. A cryptographically-secure hash of the bypass code is stored by Apple's activation server. This hash allows the activation server to verify that the correct bypass code has been provided to the device. For further information see Escrow Keys and Bypass Codes (page 132).

When the device creates a bypass code and hash, they're stored in the device's keychain and marked as available after first unlock and non-exportable.

To retrieve the bypass code, the MDM server uses the `ActivationLockBypassCode` query:

Key	Type	Content
`RequestType`	String	`ActivationLockBypassCode`.

Note: The activation lock bypass code must be requested before the device receives the MDMOptions (page 75) setting that enables Activation Lock. If this sequence is not followed the user may lock the device before MDM installs the bypass, in which case the bypass code will not work.

If a bypass code has never been created on the device, a new one is created when this query is received. Either way the bypass code is returned, if possible:

Key	Type	Content
`ActivationLockBypassCode`.	String	The activation lock bypass code, if it's available.

Once retrieved and stored by the MDM server, the bypass code can be removed from the device using the `ClearActivationLockBypassCode` command:

Key	Type	Content
`ClearActivation-LockBypassCode`	String	Supervised only. Clears the activation lock bypass code from the device.

If the command is successful, an Acknowledged status is returned. If not removed, the bypass code is automatically deleted from the device after 15 days.

Once a device is erased, the bypass code can be manually entered when prompted by the Setup Assistant, leaving the username field empty. However, it's recommended that an MDM server should clear the activation lock, using the web service described below, prior to erasing a device.

Authentication

The MDM server must provide its APNS certificate when establishing the SSL connection with the web service.

Request

To remove an activation lock, provide the device's bypass code to the web service. The request should be a standard HTTPS POST on port 443 to `https://deviceservices-external.apple.com/deviceservicesworkers/escrowKeyUnlock`. The request must also have the `contentType` header set to `application/x-www-form-urlencoded`.

The following arguments must be provided as part of the URL request string:

Argument	Description
serial	The device's serial number (required).
imei	Device IMEI (omitted for non-carrier devices).
meid	Device MEID (omitted for non-carrier devices).
productType	Example: iPod4,1 (required).

The following arguments must go into the message body:

Argument	Description
orgName	Client-supplied value for auditing purposes: a string such as the name of the organization.
guid	Client-supplied value for auditing purposes: a string that identifies the user requesting the removal (email, LDAP ID, name, etc.).

Argument	Description
escrowKey	The device's bypass code. For further information see Escrow Keys and Bypass Codes (page 132).

Arguments provided in the message body should be formatted as parameters in a form submission. For example:

```
escrowKey=abcdefg&orgName=Acme+Inc&guid=123456
```

The arguments string should comprise the entire message body.

HTTP Response Codes

The services can return any of the HTTP status codes, and the client is expected to handle the range of status codes. The more common ones include:

Code	Description
200	Success.
400	Failure: bad request; likely cause is a malformed request query or body.
404	Failure: device is not found, or escrowKey is invalid.
500	Unexpected server error; try again later.

Response Body Format

The response body may contain diagnostic information useful when reporting issues to Apple. Do not rely on specific codes, because they may change.

Escrow Keys and Bypass Codes

Your MDM server implementation should store two bypass codes:

- The device-generated bypass code retrieved using the `ActivationLockBypassCode` device query. The server should retain this code until it receives a different, non-empty code from the device.
- The bypass code the server creates when initiating an activation lock through MDM.

The server should try to unlock the device with the bypass code most likely to be active, then try the other code if the first one fails. It is impossible for the server to be certain which code is active at a given time (or even to determine if the device is locked at all) because the device can always be erased and its activation lock removed manually by entering the correct Apple ID or password. The device's `IsActivationLockEnabled` value is not an accurate reflection of its true activation lock state because the device can report either a false positive or a false negative.

Following is a sample of code that generates both an escrow key and a bypass code:

```
#define MCBYPASS_CODE_LENGTH 31 // Excluding terminating null
#define MCBYPASS_CODE_BUFFER_LENGTH 32 // Including terminating null
#define MCBYPASS_RAW_BYTES_LENGTH 16
#define MCBYPASS_HASH_LENGTH CC_SHA256_DIGEST_LENGTH

- (NSString*) _createNewActivationLockBypassCodeOutHash:(NSString**)outHash
{
#define RANDOM_BYTES_LENGTH 16
#define SALT_LENGTH 4
    // Encode raw bytes
    static const char kSymbols[] = "0123456789ACDEFGHJKLMNPQRTUVWXYZ";
                                //  00000000000000001111111111111111
                                //  0123456789abcdef0123456789abcdef

    // Insert dashes after outputting characters at these positions
    static const int kDashPositions[] = { 5, 10, 14, 18, 22 };

    char    rawBytes[MCBYPASS_RAW_BYTES_LENGTH];
    char    code[MCBYPASS_CODE_BUFFER_LENGTH];
    uint8_t hash[MCBYPASS_HASH_LENGTH];
    uint8_t salt[SALT_LENGTH] = {0, 0, 0, 0};

    arc4random_buf(rawBytes, RANDOM_BYTES_LENGTH);
   CCKeyDerivationPBKDF(kCCPBKDF2, rawBytes, RANDOM_BYTES_LENGTH, salt, SALT_LENGTH,
        kCCPRFHmacAlgSHA256, 50000, hash, CC_SHA256_DIGEST_LENGTH);

    if (outHash) *outHash = [NSData dataWithBytes:hash
        length:MCBYPASS_HASH_LENGTH].hexString;

    int         outputCharacterCount = 0;
    const int*  nextDashPosition     = kDashPositions;
    char*       outputCursor         = code;
    uint8_t*    inputCursor          = (uint8_t*)rawBytes;

    // Generate output one symbol at a time
#define INPUT_BITS      128
#define BITS_PER_BYTE   8
#define BITS_PER_SYMBOL 5

    int bitsProcessed     = 0;
    int bitOffsetIntoByte = 0;
    while (bitsProcessed <= (INPUT_BITS - BITS_PER_SYMBOL)) {
        int bitsThisByte = (bitOffsetIntoByte < BITS_PER_BYTE - BITS_PER_SYMBOL
            ? BITS_PER_SYMBOL : BITS_PER_BYTE - bitOffsetIntoByte);
        int bitsNextByte = (bitsThisByte < BITS_PER_SYMBOL ? BITS_PER_SYMBOL
            - bitsThisByte : 0);

        uint8_t value = (((*inputCursor << bitOffsetIntoByte) & 0xff)
        >> (BITS_PER_BYTE - bitsThisByte));

        bitOffsetIntoByte += BITS_PER_SYMBOL;
        if (bitOffsetIntoByte >= BITS_PER_BYTE) {
            bitOffsetIntoByte -= BITS_PER_BYTE;
            inputCursor++;
        }

        if (bitsNextByte) {
            value <<= bitsNextByte;
            value |= (*inputCursor >> (BITS_PER_BYTE - bitsNextByte));
        }

       *outputCursor++ = kSymbols[value];
        if (++outputCharacterCount == *nextDashPosition) {
            ++nextDashPosition;
            *outputCursor++ = '-';
        }

        bitsProcessed += BITS_PER_SYMBOL;
    } // while

    // Process remaining bits
    int bitsRemaining = INPUT_BITS - bitsProcessed;
    if (bitsRemaining) {
        uint8_t value = (((*inputCursor << bitOffsetIntoByte) & 0xff)
            >> (BITS_PER_BYTE - bitsRemaining));
       *outputCursor++ = kSymbols[value];
    }
    *outputCursor = '\0';
    return [NSString stringWithUTF8String:code];
} // -_createNewActivationLockBypassCodeOutHash:
```

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.

Key	Value
`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_multi_user`	Optional. Boolean. If `true`, tells the device to configure for Shared iPad. Default is `false`. This key is valid in X-Server-Protocol-Version 2 and later. Devices that do not meet the Shared iPad minimum requirements do not honor this command. With iOS devices, `com.apple.mdm.per-user-connections` must be added to the MDM enrollment profile's `ServerCapabilities`. See iOS Support for Per-User Connections (page 32).
`is_mandatory`	Optional. Boolean. If `true`, the user may not skip applying the profile returned by the MDM server. Default is `false`.
`await_device_-configured`	Optional. Boolean. If `true`, the device does not continue in Setup Assistant until the MDM server sends a command stating that the device is configured (see DeviceConfigured (page 81)). Default is `false`. This key is valid in X-Server-Protocol-Version 2 and later.
`is_mdm_removable`	If `false`, the MDM payload delivered by the configuration URL cannot be removed by the user via the user interface on the device; that is, the MDM payload is locked onto the device. This key can be set to `false` only if `is_supervised` is set to `true`. Defaults to `true`.
`support_phone_number`	Optional. String. A support phone number for the organization.
`support_email_-address`	Optional. String. A support email address for the organization. This key is valid in X-Server-Protocol-Version 2 and later.
`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.

Key	Value
`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. • `Registration`: Disables registration screen in OS X. • `Location`: Disables Location Services. • `Restore`: Disables restoring from backup. • `AppleID`: Disables signing in to Apple ID and iCloud. • `TOS`: Skips Terms and Conditions. • `Biometric`: Skips Touch ID setup. • `Payment`: Skips Apple Pay setup. • `Zoom`: Skips zoom setup. • `DisplayTone`: Skips DisplayTone setup. • `Android`: If the Restore pane is not skipped, removes Move from Android option from it. • `Siri`: Disables Siri. • `Diagnostics`: Disables automatically sending diagnostic information. • `FileVault`: Disables FileVault Setup Assistant screen (OS X only).
`department`	Optional. 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:2
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,
   "await_device_configured":false,
   "is_mdm_removable":false,
   "department": "IT Department",
   "org_magic": "913FABBB-0032-4E13-9966-D6BBAC900331",
   "support_phone_number": "1-555-555-5555",
   "support_email_address": "org-email@example.com",
   "anchor_certs":[
      "MIICkDCCAfmgAwIBAgIJAOAeuvyohALaMA0GCSqGSIb3DQEBBQUAMGExCzAJBgNVBAYT..."
   ],
   "supervising_host_certs:[
        "AlVTMQswCQYDVQQIDAJDQTESMBAGA1UEBwwJQ3VwZXJ0aW5vMRowGAYDVQQKDBFB…"
    ],
   "skip_setup_items":[
      "Location",
      "Restore",
      "Android",
      "AppleID",
      "TOS",
      "Siri",
      "Diagnostics",
      "Biometric",
      "Payment",
      "Zoom",
      "FileVault"
   ],
    "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`: The 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 149), 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 `FLAGS_INVALID` in the response body indicates that flags have been set incorrectly. Flag `is_mdm_removable` can be set to `false` only if flag `is_supervised` is set to `true`.
- A `400` error code with `CONFIG_URL_INVALID` in the response body indicates that the URL field in the uploaded profile is either empty or has exceeded the maximum allowed length (2000 URL encoded characters). The syntax of the URL is defined by RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, amended by RFC 2732: Format for Literal IPv6 Addresses in URLs.
- A `400` error code with `CONFIG_NAME_INVALID` in the response body indicates that the `profile_name` field in the uploaded profile is either empty or has exceeded the maximum allowed length (125 UTF-8 characters).

  .
- A `400` error code with `DEPARTMENT_INVALID` in the response body indicates that the `department` field in the uploaded profile is either empty or has exceeded the maximum allowed length (125 UTF-8 characters).

  .
- A `400` error code with `SUPPORT_PHONE_INVALID` in the response body indicates that the `support_phone_number` field in the uploaded profile is either empty or has exceeded the maximum allowed length (50 UTF-8 characters).

  .
- A `400` error code with `SUPPORT_EMAIL_INVALID` in the response body indicates that the `support_email_address` field in the uploaded profile is either empty or has exceeded the maximum allowed length (250 UTF-8 characters).

  .
- A `400` error code with `MAGIC_INVALID` in the response body indicates that the `magic` field in the uploaded profile is either empty or has exceeded the maximum allowed length (256 UTF-8 characters).

  .

Assign Profile

Tells Apple's severs that the specified devices should use a particular profile defined by the Define Profile (page 135) 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:2
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).

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 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 149), 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:2
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.
`profile_uuid`	String. The unique ID of the assigned profile.
`url`	String. The URL of the MDM server.
`allow_pairing`	Optional. Boolean. Default is `true`.

Key	Value
`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`.
`await_device_-configured`	Optional. Boolean. If `true`, the device will not continue in Setup Assistant until the MDM server sends a command stating that the device is configured (see DeviceConfigured (page 81)). Default is `false`. This key is valid in X-Server-Protocol-Version 2 and later.
`is_mdm_removable`	If `false`, the MDM payload delivered by the configuration URL cannot be removed by the user using the user interface on the device; that is, the MDM payload is locked onto the device. Defaults to `true`.
`support_phone_number`	Optional. String. A support phone number for the organization.
`support_email_-address`	Optional. String. A support email address for the organization. This key is valid in X-Server-Protocol-Version 2 and later.
`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: • `Passcode`: Hides and disables the passcode pane. • `Registration`: Disables registration screen in OS X. • `Location`: Disables Location Services. • `Restore`: Disables restoring from backup. • `AppleID`: Disables signing in to Apple ID and iCloud. • `Biometric`: Skips Touch ID setup. • `Payment`: Skips Apple Pay setup. • `Zoom`: Skips zoom setup. • `DisplayTone`: Skips DisplayTone setup. • `Android`: If the Restore pane is not skipped, removes Move from Android option from it. • `TOS`: Skips Terms and Conditions. • `Siri`: Disables Siri. • `Diagnostics`: Disables automatically sending diagnostic information. • `FileVault`: Disables FileVault Setup Assistant screen.
`department`	Optional. 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,
   "await_device_configured":false,
   "is_mdm_removable":false,
   "department": "IT Department",
   "org_magic": "913FABBB-0032-4E13-9966-D6BBAC900331",
   "support_phone_number": "1-555-555-5555",
   "support_email_address": "org-email@example.com",
   "anchor_certs":[
      "MIICkDCCAfmgAwIBAgIJAOAeuvyohALaMA0GCSqGSIb3DQEBBQUAMGExCzAJBgNVBAYT..."
   ],
   "supervising_host_certs:[
        "AlVTMQswCQYDVQQIDAJDQTESMBAGA1UEBwwJQ3VwZXJ0aW5vMRowGAYDVQQKDBFB…"
    ],
   "skip_setup_items":[
      "Location",
      "Restore",
      "Android",
      "AppleID",
      "TOS",
      "Siri",
      "Diagnostics",
      "Biometric",
      "Payment",
      "Zoom",
      "FileVault"
   ]
}
```

Request-Specific Errors

In addition to the standard errors listed in Common Error Codes (page 149), 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 Profile URL

When a `url` value is provided in the profile response, the device makes an HTTPS POST call to that URL. The requesthas a Content-Type of `application/pkcs7-signature`. The following dictionary is sent as the body of the request. The dictionary is 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.
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 is CMS-signed with the device identity certificate. The device's certificate and all necessary intermediate certificates are 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. In iOS 7.1, the WWW-Authenticate header may also contain the `Basic` authentication method as outlined in RFC2617. 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 is shown above the prompt for the username and password. If the content is empty, a default message is 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.

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:2
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 149), 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. In iOS 9 and later or OS X v10.11 and later, VPP can assign a license to the device serial number, so no Apple ID is required to download the app.

The Volume Purchase Program provides a number of web services that MDM servers can use to associate volume purchases with particular users or devices. The following services are currently supported:

- Create a user in the iTunes Store representing a user in the MDM system, against which licenses and an iTunes Store account may be linked: registerVPPUserSrv (page 162).
- Determine the current iTunes account status of one or more VPP users: getVPPUserSrv (page 164) or getVPPUsersSrv (page 167).
- Determine the statuses of a VPP user's current licenses for software and other products: getVPPLicensesSrv (page 170).
- List the VPP assets for which an organization has licenses, including counts of assigned and unassigned licenses for each asset: getVPPAssetsSrv (page 174).
- Query the iTunes Store for information about apps and books: contentMetadataLookupUrl (page 176).
- Disassociate a VPP user from their iTunes account and release their revocable licenses: retireVPPUserSrv (page 181).
- Perform batch associations or disassociations of multiple VPP users or devices with their licenses: manageVPPLicensesByAdamIdSrv (page 182).
- Fetch or update a VPP user's email address and optionally link to a Managed Apple ID: editVPPUserSrv (page 188).
- Store and/or return organization-specific information to/from the VPP server: VPPClientConfigSrv (page 190).
- Fetch the current list of VPP web service URLs and error numbers: VPPServiceConfigSrv (page 191).

Using Web Services

You access the services described in this chapter through the MDM payloads described in Structure of MDM Payloads (page 19).

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.

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` require 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 in to https://vpp.itunes.apple.com/.

On the Account Summary page, click 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. Once created, tokens are listed on the Account Summary page.

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 a license, the "licenses" field does 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 slash (`/`). 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.

Retry-After Header

The VPP service may return a 503 Service Unavailable status to clients whose requests result in an unusually high load on the VPP service, or when the VPP service is experiencing loads beyond its current capacity to respond to requests. A Retry-After header may be included in this response, indicating how long the client must wait before making additional requests. Clients who make requests before this time may be rejected for even longer periods of time, or (in extreme cases) may have their VPP account suspended.

Avoid triggering the Retry-After header by setting the `assignedOnly` parameter `true` in calls to `getVPPLicensesSrv`. Also, do not use `getVPPAssetsSrv` for real-time license counts. The client should maintain its own license counts and should call `getVPPAssetsSrv` only when checking for new assets.

The Retry-After response-header field may also be used with any 3xx (Redirection) response to indicate the minimum time the user-agent is asked to wait before issuing the redirected request (see RFC 2616: HTTP/1.1, Section 14.37). The value of this field can be either an HTTP-date or an integer number of seconds (in decimal) after the time of the response.

```
Retry-After = "Retry-After" ":" ( HTTP-date | delta-seconds )
```

Two examples of its use are:

```
Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
```

```
Retry-After: 120
```

In the latter example, the delay is 2 minutes.

VPP Account Protection

It is reasonable behavior for a product that manages VPP app assignments to reset the VPP account by retiring all users and revoking all app assignments when it is first configured to use a VPP account. Therefore, it is very important that your product always sets the `clientContext` data as documented below so that other products that manage VPP accounts can know that the VPP account is being managed by another product and not reset the VPP account without warning.

To ensure that a VPP account is not being managed by another product, follow these steps every time your product starts a VPP session:

- During initial setup, check the `clientContext` attribute returned from the `VPPClientConfigSrv` request.
    - If `clientContext` is empty, create a JSON string with these keys and values:

      ```
      {"hostname":<my.servername.com>, "guid":<random_uuid>}
      ```

      The UUID should be a standard 8-4-4-4-12 formatted UUID string and must be unique for each installation of your product.

      Write this JSON string to `clientContext` to claim this VPP account for your product.
    - If `clientContext` is not empty and does not match the `guid` value of your product, report the `hostname` returned by `clientContext` and confirm that your product should take over from it. Do not rely on `hostname` to confirm that your product still has a proper claim on the VPP account.
- At the start of every subsequent VPP session, check `clientContext` to ensure that it still represents the correct installation of your product.

- If `clientContext` no longer refers to your product, do not make any further requests to the VPP service for that VPP account until the account has been reactivated by administrator commands. Your product should report this isolation action to an administrator, giving the `hostname` of the server that now claims to manage the VPP account.

Initial Import of VPP Managed Distribution Licenses Using getVPPLicensesSrv

It is not necessary to sync every single app license for a specific VPP account. The recommended procedure for importing licenses is to skip importing all of the licenses and instead start importing license counts and then changes. This can be accomplished in the following way:

1. Send a request using `getVPPAssetsSrv` with `includeLicenseCounts : true`. This returns the current license count by `adamID`.
2. Send one request using `getVPPLicensesSrv`. Record the `batchToken` and `totalBatchCount`.
3. Send another request to `getVPPLicensesSrv` using the `batchToken` value from Step 2 and an `overrideIndex` value equal to `totalBatchCount`.
4. Record the `sinceModifiedToken` value and begin syncing license updates and changes instead of all licenses. Set `assignedOnly=true` if you want to get only new license assignments.

Note: Using `sinceModifiedToken` can result in batches with zero records in them. This is not an error or an end signal; just move to the next batch.

For further information, see Parallel getVPP Requests (page 174) and getVPPLicensesSrv (page 170).

productTypeId Codes

Some service requests may return the ID of an Apple product type as a decimal integer, with one of these values:

productTypeId	Meaning
7	Mac OS X software.
8	iOS or OS X App Store app.
10	book.

Managed Apple IDs

Managed Apple IDs were introduced in iOS 9.3. These accounts can be tied to the same organization as the VPP Program Facilitator users who manage licenses. When this is the case, the MDM server may choose to instruct the VPP service to associate Managed Apple IDs with given VPP users. This removes the need to send out an invitation (email or push) to users and wait for them to join by going through an acceptance process.

Managed Apple IDs are implemented through the following services by adding an optional parameter, `managedAppleIDStr`:

- registerVPPUserSrv (page 162)
- editVPPUserSrv (page 188)

Apple uses the Apple ID passed in `managedAppleIDStr` to look up the user's `organizationId`. If the VPP Program Facilitator account associated with the `sToken` making the request is also a Managed Apple ID and that Apple ID's `organizationId` is the same as the user's, the VPP user will be linked to that Apple ID.

If the user cannot be found in the iTunes database, or the user is found but the user's `organizationId` does not match the `organizationId` of the `sToken`'s associated user, the service response returns error 9635, `APPLE_ID_CANT_BE_USED`.

Program Facilitators

As described in Authentication (page 152), Program Facilitators obtain from the iTunes VPP website the `sToken` parameters that must be passed in VPP service requests. Each `sToken` authenticates an organization through the associated Program Facilitator account that generated it.

Managed Apple IDs make it possble for multiple Program Facilitators to be linked together into a group. Each Program Facilitator in the group is assigned a `facilitatorMemberId`. An `sToken` can use this `facilitatorMemberId` to access and change data associated with different Program Facilitators as long as the other Program Facilitators are in the same group. Using VPPClientConfigSrv (page 190), the MDM server can discover member info about all the other Program Facilitators whose data its `sToken` can access, including the `facilitatorMemberId` of each member.

All VPP service calls, except `VPPClientConfigSrv`, accept an optional `facilitatorMemberId` parameter. It is subject to these rules:

- If a Program Facilitator's `facilitatorMemberId` is passed in a service request, the service is executed as if the request had been made with that Facilitator's `sToken` instead.
- If a service request passes the `facilitatorMemberId` of a Program Facilitator that was never associated with the requesting organization, or left it, or is no longer managed, an error is returned.

Here is an example of a `VPPClientConfigSrv` response when the `sToken` passed to it is associated with a group of three users, each of which has a different Program Facilitator:

```
{
    "apnToken": "aJQGSAd+H7FrmIZn9K4IbRbXpge3ySkchugcfYK/ZXg=",
    "appleId": "user1@someorg.com",
    "clientContext": "{\"guid\":\"e91e570f-3eba-4b43-97d3-0f39450c8b92\",
       \"hostname\":\"vpp-integrations2.apple.com\",\"ac2\":1}",
    "countryCode": "US",
    "email": "user1@someorg.com",
    "facilitatorMemberId": 200841,
    "organizationId": 2168850000179778,
    "status": 0,
    "vppGroupMembers": [
        {
            "appleId": "user3@someorg.com",
            "clientContext": "test123test123test123",
            "email": "user3@someorg.com",
            "facilitatorMemberId": 200844,
            "organizationId": 2168850000179778,
            "locationId": 2167975000001686,
            "locationName": "Central School"
        },
        {
            "appleId": "user1@someorg.com",
            "clientContext": "{\"guid\":\"e91e570f-3eba-4b43-97d3-0f39450c8b92\",
               \"hostname\":\"vpp-integrations2.apple.com\",\"ac2\":1}",
            "email": "user1@someorg.com",
            "facilitatorMemberId": 200841,
            "organizationId": 2168850000179778,
            "locationId": 2167975000001686,
            "locationName": "Central School"
        },
        {
            "appleId": "user2@someorg.com",
            "email": "user2@someorg.com",
            "facilitatorMemberId": 200843,
            "organizationId": 2168850000179778,
            "locationId": 2167975000001686,
            "locationName": "Central School"
        }
    ]
}
```

Note that `vppGroupMembers` contains all of the members of the Program Facilitator's group, including the calling member.

Read-Only Access

Using Apple School Manager and Managed Apple IDs, you can tailor different sets of privileges for individual Program Facilitators. This allows a finer range of control on what such users can do. For example, a Program Facilitator that has only the "Read Only" privilege can use the `getVPPUserSrv`, `getVPPUsersSrv`, and `getVPPLicensesSrv` services but not use `retireVPPUserSrv`, `disassociateVPPLicenseSrv`, or `manageVPPLicensesByAdamIdSrv`. You can also assign Program Facilitators "Can Purchase" and/or "Can Manage" privileges, so an individual Program Facilitator could manage licenses but not buy them. (Note that purchasing users and managing users automatically have read privileges.)

Error Codes

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

errorNumber	Meaning
`9600`	Missing required argument
`9601`	Login required
`9602`	Invalid argument
`9603`	Internal error
`9604`	Result not found

errorNumber	Meaning
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
9612	Failed to create claim job
9613	Failed to create unclaim job
9614	Invalid date format
9615	OrgCountry not found
9616	License already assigned (see Error Code 9616 (page 161))
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
9624	License was refunded and is no longer valid.
9625	The sToken has been revoked.
9626	License already assigned to a different user. The MDM server should retry the assignment with a different license.
9628	Ineligible device assignment: MDM tried to assign an item to a serial number but device assignment is not allowed for that item.

errorNumber	Meaning
`9630`	Too many recent already-assigned errors: If MDM gets the same 9616 error from assignments for the same organization, user identifier, and item identifier (license ID, adam ID, or pricing parameter) and does so within too short a time (generally several minutes), it may return this error code.
`9631`	Too many recent no-license errors: If MDM gets the same 9610 error from assignments for the same organization, user identifier, and item identifier (license ID, adam ID, or pricing parameter) and does so within too short a time (generally several minutes), it may return this error code.
`9632`	Too many recent manage-license calls with identical request: If MDM gets precisely the same request to `manageVPPLicensesByAdamIdSrv` too many times within too short a time (generally several minutes), it may return this error code.
`9633`	Data for a batch token passed could not be recovered.
`9634`	Returned when a caller tries to use a formerly deprecated featured that has been removed.
`9635`	Apple ID passed for iTunes Store association cannot be found or is not applicable to organization of the user (see Managed Apple IDs (page 157)).
`9636`	Registered user not found.
`9637`	`sToken` is not allowed to perform the operation requested.
`9638`	Facilitator account that generated `sToken` has no Managed ID organization ID and cannot manipulate the facilitator member requested.
`9639`	No facilitator member could be found for the facilitator member ID requested.
`9640`	Account details of the facilitator member ID requested could not be recovered (likely a transient issue).

Additional error types may be added in the future.

Error Code 9616

Error number `9616` is returned when an attempt is made to assign a license to a user that already has a license for the specified app or book, in which case there is no need to retry the assignment.

Additional information is returned to MDM when a `9616` error occurs. Sometimes it's because the specific user in the request is already assigned to the item in question. When that happens the `9616` error is accompanied by a `licenseAlreadyAssigned` entry with details about the user and the license. For example,

```
{"licenseAlreadyAssigned":{"pricingParam":"STDQ","itsIdHash":
"XuHVGvasXcfEVUUn4EP2wjHEUK00s=","userId":9918783273,"productTypeId":8,
"isIrrevocable":false,"adamIdStr":"778658393","userIdStr":"9918783273",
"licenseIdStr":"99147599840","productTypeName":"Application",
"clientUserIdStr":"xxutt8-e079-4b05-b403-a0792890",
"licenseId":9147599840,"adamId":778658393,"status":"Associated"},
"errorMessage":"License already assigned","errorNumber":9616,"status":-1}
```

Alternatively, a `9616` error may have a `regUsersAlreadyAssigned` entry in the response with information about the one or more other users who already have the item in question. In these cases, the VPP user specified by the user ID or the `clientUserIdStr` does not have the item, but some other users in the organization associated with the same iTunes Store account has the item. If that happens, the server returns `9616` and information about those other users:

```
{"errorMessage":"License already assigned",
"regUsersAlreadyAssigned":[{"itsIdHash":"XXX2CVvZar9YZnpqJxV0SHOUCU=",
"clientUserIdStr":"jjjCXhHHee0e3c-x999-43a9-Xe04-1dcax80ac01x",
"userId":9991992450,"email”:”user@example.apple.com","status":"Associated"}],"
"errorNumber":9616,"status":-1}
```

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`	Not required.	`"user1@someorg.com"`.
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.
`facilitatorMemberId`	Not required.	See Program Facilitators (page 157).

Parameter Name	**Required or Not**	**Example**
`managedAppleIDStr`	Not required.	`"user1@someorg.com"`.

`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`, the following occurs:

- 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 an 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, the unassigned user record gets marked with the `Retired` status, and the formerly retired user record gets moved to the `Associated` status.

The `managedAppleIDStr` parameter is discussed in Managed Apple IDs (page 157).

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".
facilitatorMember	{ "appleId":"user1@someorg.com", "countryCode":"US", "email":"user1@someorg.com", "facilitatorMemberId":200843, "organizationId":2168850000179778, },
errorNumber	9600.

getVPPUserSrv

The request takes the following parameters:

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

Parameter Name	Required or Not	Example
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.
`facilitatorMemberId`	Not required.	See Program Facilitators (page 157).

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 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 number.

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 }] }`

Field Name	Example of Value
facilitatorMember	{ "appleId":"user1@someorg.com", "countryCode":"US", "email":"user1@someorg.com", "facilitatorMemberId":200843, "organizationId":2168850000179778, },
errorMessage	"Result not found".
errorNumber	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	Not required.	EkZQCWOwhDFCwgQsUFJZkA oUU0pKLEnOUAIKZOalpFYAR YzA7OSc0pTUoNSSzKLUFJAy Q6CSWgCS88JnkgAAAA==.
sinceModifiedToken	Not required.	0zJTU5SAEplpMF4wWCozJy ezGKjS0NjM0tjUwtTA3MzQ 1FqhFgBuLPH3TgAAAA==.
includeRetired	Not required.	1.
sToken	Required.	"h40Gte9aQnZFDNM...6ZQ=".
facilitatorMemberId	Not required.	See Program Facilitators (page 157).

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

You 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, otherwise it contains `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.

The response contains some of these fields:

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

Field Name	Example of Value

`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.

`facilitatorMember`	`{`
	`"appleId":"user1@someorg.com",`
	`"countryCode":"US",`
	`"email":"user1@someorg.com",`
	`"facilitatorMemberId":200843,`
	`"organizationId":2168850000179778,`
	`},`

`totalCount`	5 Note that this value is returned only for requests that do not include a `batchToken` value.

Field Name	Example of Value
`errorMessage`	`"Result not found"`.
`errorNumber`	`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 an estimate of the total number of records that will be returned.

getVPPLicensesSrv

The request takes the following parameters:

Parameter Name	Required or Not	Example
`batchToken`	Not required.	`EkZQCWOwhDFCwgQsUFJZkA` `oUU0pKLEnOUAIKZOalpFYAR` `YzA7OSc0pTUoNSSzKLUFJAy` `Q6CSWgCS88JnkgAAAA==`.
`sinceModifiedToken`	Not required.	`0zJTU5SAEplpMF4wWCozJy` `ezGKjS0NjM0tjUwtTA3MzQ` `1FqhFgBuLPH3TgAAAA==`.
`adamId`	Not required.	`408709785`.
`pricingParam`	Not required.	`"PLUS"`.

Parameter Name	Required or Not	Example
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.
`facilitatorMemberId`	Not required.	See Program Facilitators (page 157).
`assignedOnly`	Not required.; defaults to false.	`true`.

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

You 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, your MDM server does the following:

- Makes an initial request to `getVPPUsersSrv` with no `batchToken` or `sinceModifiedToken`.

  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.

If the `assignedOnly` parameter is set to `true`, only licenses currently associated with an Apple ID or a device serial number are returned. When the `assignedOnly` parameter is omitted, all license records are returned regardless of association status.

If a `pricingParam` parameter is not passed in the `getVPPLicensesSrv` request, the VPP service returns all licenses (both PLUS and STDQ `pricingParam` values).

The response contains some of these fields:

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

`licenses`	`[`
	`{`
	`        "licenseIdStr": 1,`
	`        "adamIdStr": 408709785,`
	`        "productTypeId": 7,`
	`        "pricingParam": "STDQ",`
	`        "productTypeName": "Software",`
	`        "isIrrevocable": false`
	`    },`
	`    {`
	`        "licenseIdStr": 2,`
	`        "adamIdStr": 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 token.
`totalBatchCount`	`3` Indicates the total number of round trips that will be necessary to get the full result set.

Field Name	Example of Value

`facilitatorMember`	`{`
	`"appleId":"user1@someorg.com",`
	`"countryCode":"US",`
	`"email":"user1@someorg.com",`
	`"facilitatorMemberId":200843,`
	`"organizationId":2168850000179778,`
	`},`

`errorMessage`	`"Result not found"`.
`errorNumber`	`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).

Licenses that are assigned to a user contain `userId`, `clientUserIdStr`, and `itsIdHashfield` fields, as shown in the second example above. The `totalBatchCount` field contains the total number of round trips that are necessary to get all records in the request. This can be used to provide a progress indicator when compared to the number of batches processed so far.

Note: The `totalCount` value is returned only on the request that started the batch process (the listing request issued without any tokens), because the actual number of licenses or users returned can be different by the time the client has finished.

One of a set of sequential `getVPPLicensesSrv` batch requests may return an error. It is also possible to get a response from a listing call that includes no token but also no error number. Because all listing API requests should return either a batch or `sinceModified` token, do not interpret an error or the lack of a token for an individual batch to mean that the last batch has been received. The last batch is signified by the inclusion of a `sinceModifiedToken`. If an individual batch request fails, the MDM server should retry the same batch using the same `batchToken`.

Receiving a `9603 'Internal Error'` response typically indicates that the VPP server couldn't provide timely processing. Nothing is necessarily wrong with the request. When the MDM server receives this response, it should send the current request again. If it continues to receive `9603` errors after more than five attempts, it may mean that the VPP service is unexpectedly down and further retries should be scheduled for minutes later, instead of seconds.

Parallel getVPP Requests

Both the `getVPPLicensesSrv` and `getVPPUsersSrv` services can accept multiple requests in parallel, instead of sequentially, which can significantly reduce the amount of time required to request all licenses and users. You start by making an initial request to receive a `batchToken`. Subsequent requests can be submitted in parallel by submitting the same `batchToken` and including an `overrideIndex` value from 1 to `totalBatchCount`, which is now returned with `getVPPLicensesSrv` requests. The request in which the `overrideIndex` value is equal to the `totalBatchCount` returns the new `sinceModifiedToken`.

It is advisable not to submit more than five requests simultaneously.

getVPPAssetsSrv

This service returns an enumeration of the assets (`{adamIdStr, pricingParam}` tuples) for which an organization has licenses, along with an optional count of the total number of licenses and the number of licenses available for each asset.

Parameter Name	Required or Not	Example
`includeLicenseCounts`	Not required.; defaults to `false`.	`true`.
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.
`facilitatorMemberId`	Not required.	See Program Facilitators (page 157).

If `includeLicenseCounts` is set to true, the total number of licenses, the number of licenses assigned, and the number of licenses unassigned are included with the response for each asset.

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

Field Name	Example of Value
`assets`	`[{ "adamIdStr":"375380948", "assignedCount":2, "availableCount":8, "deviceAssignable":true, "isIrrevocable":false, "pricingParam":"STDQ", "productTypeId":8, "productTypeName":"Application", "retiredCount":0, "totalCount":10 }, { "adamIdStr":"435160039", "assignedCount":2, "availableCount":8, "deviceAssignable":false, "isIrrevocable":true, "pricingParam":"PLUS", "productTypeId":10, "productTypeName":"Publication", "retiredCount":0, "totalCount":10 }]`

Field Name	Example of Value
`facilitatorMember`	`{ "appleId":"user1@someorg.com", "countryCode":"US", "email":"user1@someorg.com", "facilitatorMemberId":200843, "organizationId":2168850000179778, },`
`totalCount`	4
`errorMessage`	`"Result not found"`
`errorNumber`	`9604`

contentMetadataLookupUrl

The `contentMetadataLookupUrl` in the `VPPServiceConfigSrv` response allows an MDM server to query the iTunes Store for app and book metadata. When the VPP `sToken` is included in the request as a cookie, an MDM server can also get authenticated app metadata for B2B apps already owned by the VPP account, as well as apps that can still be redownloaded but can no longer be purchased.

The URL query string tells the content metadata lookup service what app or book to look up. The VPP `sToken` must be included as a cookie named `itvt` to access the authenticated metadata. Here is an example of the URL to look up an app: https://uclient-api.itunes.apple.com/WebObjects/MZStorePlatform.woa/wa/lookup?version=2&id=361309726&p=mdm-lookup&caller=MDM&platform=itunes&cc=us&l=en.

Here is an example of what a response might look like:

```
{
    "isAuthenticated": false,
    "results": {
        "361309726": {
            "artistId": "284417353",
            "artistName": "Apple",
          "artistUrl": "https://itunes.apple.com/us/artist/apple/id284417353?mt=8",
            "artwork": {
                "bgColor": "ffb800",
                "height": 1024,
                "supportsLayeredImage": false,
                "textColor1": "161616",
                "textColor2": "161616",
                "textColor3": "453712",
                "textColor4": "453712",
                "url": "http://is5.mzstatic.com/image/thumb/
Purple3/v4/72/7d/38/727d38ee-9245-eda6-1188-3458133bd99a/source/{w}x{h}bb.{f}",
                "width": 1024
            },
            "bundleId": "com.apple.Pages",
            "contentRatingsBySystem": {
                "appsApple": {
                    "name": "4+",
                    "rank": 1,
                    "value": 100
                }
            },
            "copyright": "\u00a9 2010 - 2015 Apple Inc.",
            "description": {
               "standard": "Pages is the most beautiful word processor you\u2019ve
 ever seen on a mobile device. This powerful word processor helps you create
gorgeous reports, resumes, and documents in minutes. Pages has been designed
exclusively for the iPad, iPhone, and iPod touch with support for Multi-Touch
gestures and Smart Zoom.\n\nGet a quick start by using one of over 60 Apple-designed
 templates. Or use a blank document and easily add text, images, shapes, and more
 with a few taps. Then format using beautiful preset styles and fonts. And use
advanced features like change tracking, comments, and highlights to easily review
 changes in a document.\n\nWith iCloud built in, your documents are kept up-to-date
 across all your devices. You can instantly share a document using just a link,
giving others the latest version and the ability to edit it directly from
www.icloud.com using a Mac or PC browser.\n\nPages 2.0 is updated with a stunning
 new design and improved performance. And with a new unified file format across
Mac, iOS, and web, your documents are consistently beautiful everywhere you open
them.\n\nGet started quickly\n\u2022 Choose from over 60 Apple-designed templates
 to instantly create beautiful reports, resumes, cards, and posters\n\u2022 Import
 and edit Microsoft Word and plain text files using Mail, a WebDAV service, or
iTunes File Sharing\n\u2022 Quickly browse your document using the page navigator
 and see a thumbnail preview of each page\n\u2022 Turn on Coaching Tips for guided
 in-app help\n\nCreate beautiful documents\n\u2022 Write and edit documents using
 the onscreen keyboard or a wireless keyboard with Bluetooth\n\u2022 Format your
document with gorgeous styles, fonts, and textures\n\u2022 Your most important
text formatting options are right in your keyboard, and always just a tap or two
```

```
away\n\u2022 Easily add images and video to your document using the Media
Browser\n\u2022 Use auto-text wrap to flow text around images\n\u2022 Animate data
 with new interactive column, bar, scatter, and bubble charts\n\u2022 Organize
your data easily in tables\n\nAdvanced writing tools\n\u2022 Turn on change tracking
 to mark up a document as you edit it\n\u2022 Use comments and highlights to share
 ideas and feedback with others\n\u2022 Create footnotes and endnotes and view
word counts with character, paragraph, and page counts\n\u2022 Automatic list
making and spellchecking \n\u2022 Create and view impressive 2D, 3D, and interactive
 bar, line, area, and pie charts\n\u2022 Use Undo to go back through your previous
 changes\n\niCloud\n\u2022 Turn on iCloud so your documents are automatically
available on your Mac, iPad, iPhone, iPod touch, and iCloud.com\n\u2022 Access and
 edit your documents from a Mac or PC browser at www.icloud.com with Pages for
iCloud beta\n\u2022 Pages automatically saves your documents as you make
changes\n\nShare your work\n\u2022 Use AirDrop to send your document to anyone
nearby\n\u2022 Quickly and easily share a link to your work via Mail, Messages,
Twitter, or Facebook \n\u2022 Anyone with a shared document link always has access
 to the latest version of the document and can edit it with you at iCloud.com using
 Pages for iCloud beta\n\u2022 Export your document in ePub, Microsoft Word, and
PDF format\n\u2022 Use \u201cOpen in Another App\u201d to copy documents to apps
such as Dropbox\n\u2022 Print wirelessly with AirPrint, including page range
selection, number of copies, and two-sided printing\n\nSome features may require
Internet access; additional fees and terms may apply.\nPages does not include
support for some Chinese, Japanese, or Korean (CJK) text input features such as
vertical text.\nPages for iCloud beta is currently available in English only."
            },
            "deviceFamilies": [
                "iphone",
                "ipad",
                "ipod"
            ],
            "editorialArtwork": {
                "originalFlowcaseBrick": {
                    "bgColor": "ffb700",
                    "height": 600,
                    "supportsLayeredImage": false,
                    "textColor1": "161616",
                    "textColor2": "161616",
                    "textColor3": "453612",
                    "textColor4": "453612",
                    "url": "http://is4.mzstatic.com/image/
thumb/Features5/v4/22/60/94/226094a4-ed02-a234-7576-6de696ead0ba/source/{w}x{h}{c}.{f}",
                    "width": 3200
            }
        },
        "editorialBadgeInfo": {
            "editorialBadgeType": "staffPick",
            "nameForDisplay": "Essentials"
        },
        "genreNames": [
            "Productivity",
            "Business"
        ],
        "genres": [
            {
                "mediaType": "8",
                "name": "Productivity",
                "url": "https://itunes.apple.com/us/genre/id6007"
            },
            {
                "mediaType": "8",
                "name": "Business",
                "url": "https://itunes.apple.com/us/genre/id6000"
            }
        ],
        "id": "361309726",
        "kind": "iosSoftware",
        "latestVersionReleaseDate": "Sep 15, 2015",
        "name": "Pages",
        "nameRaw": "Pages",
        "offers": [
            {
                "actionText": {
                    "downloaded": "Installed",
                    "downloading": "Installing",
                    "long": "Buy App",
                    "medium": "Buy",
                    "short": "Buy"
                    },
                    "assets": [
                        {
                            "flavor": "iosSoftware",
                            "size": 278782033
                        }
                    ],
                    "buyParams": "productType=C&price=9990&

salableAdamId=361309726&pricingParameters=STDQ&appExtVrsId=813292538",
                    "price": 9.99,
                    "priceFormatted": "$9.99",
                    "type": "buy",
                    "version": {
                        "display": "2.5.5",
                        "externalId": 813292538
                    }
                }
            ],
            "releaseDate": "2010-04-01",
            "shortUrl": "https://appsto.re/us/EysIv.i",
            "url": "https://itunes.apple.com/us/app/pages/id361309726?mt=8",
            "userRating": {
                "ratingCount": 24848,
                "ratingCountCurrentVersion": 236,
                "value": 3.5,
                "valueCurrentVersion": 3
            },
          "whatsNew": "This update contains stability improvements and bug fixes."
        }
    },
    "version": 2
}
```

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 users 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="`.
`facilitatorMemberId`	Not required.	See Program Facilitators (page 157).

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
`facilitatorMember`	`{ "appleId":"user1@someorg.com", "countryCode":"US", "email":"user1@someorg.com", "facilitatorMemberId":200843, "organizationId":2168850000179778, },`
`status`	`0` for success, `-1` for error.
`errorMessage`	`"Result not found"`.
`errorNumber`	`9604`.

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

manageVPPLicensesByAdamIdSrv

This API supersedes the `associateVPPLicenseWithVPPUserSrv` and `disassociateVPPLicenseWithVPPUserSrv` APIs as a more flexible and efficient way of changing license assignments. It offers bulk license association and disassociation in one request, with some optional flags to control back end behavior.

Parameter Name	Required or Not
`adamIdStr`	Required.
`pricingParam`	Required.
`associateClientUserIdStrs`	One (and only one) of these is required to associate licenses.
`associateSerialNumbers`	
`disassociateClientUserIdStrs`	One (and only one) of these is required to disassociate licenses.
`disassociateLicenseIdStrs`	
`disassociateSerialNumbers`	
`notifyDisassociation`	Not required.; defaults to `true`.
`sToken`	Required.
`facilitatorMemberId`	Not required.

Parameter Name	Example
`adamIdStr`	`"408709785"`
`pricingParam`	`"STDQ"`
`associateClientUserIdStrs`	`["810C9B91-...-408AD7F081A8", "d735c1cc-...-c74571007ef6",...]`
`associateSerialNumbers`	`["C17DK6D9DDQW", "DLXL6044FPH8",...]`
`disassociateClientUserIdStrs`	`["810C9B91-...-408AD7F081A8", "d735c1cc-...-c74571007ef6",...]`
`disassociateLicenseIdStrs`	`["2","3","4",...]`
`disassociateSerialNumbers`	`["C17DK6D9DDQW", "DLXL6044FPH8",...]`

Parameter Name	Example
`notifyDisassociation`	`false`
`sToken`	`"h40Gte9aQnZFDNM...6ZQ="`.
`facilitatorMemberId`	See Program Facilitators (page 157).

The request operates on a single asset (specified by the `{adamIdStr, pricingParam}` tuple) for multiple associations and disassociations in a single request. Licenses are disassociated from all users specified by the `disassociateClientUserIdStrs` array, the devices specified by the `disassociateSerialNumbers` array, or the licenses specified by the `disassociateLicenseIdStrs` array (which must only specify licenses assigned to the specified asset). At most one of these `disassociate*` arrays may be specified per request. Then licenses are associated either with the users specified by the `associateClientUserIdStrs` array or the devices specified by the `associateSerialNumbers` array. You must specify either zero or one `associate*` and zero or one `disassociate*` array per request. Specifying more than one of either `associate*` or `disassociate*` arrays result in undefined behavior.

The maximum number of entries allowed in the `associate*` and `disassociate*` arrays are indicated by the `maxBatchAssociateLicenseCount` or `maxBatchDisassociateLicenseCount` fields added to the `VPPServiceConfigSrv` response. Any request that exceeds these limits is immediately rejected with an error.

If `notifyDisassociation` is set to `false`, notifications regarding the disassociation of the license are not sent to devices.

Field Name	Example of Value
`status`	0 for success, -1 if the request failed completely, -3 if any licenses could not be changed as requested.
`adamIdStr`	`"408709785"`
`pricingParam`	`"STDQ"`
`productTypeId`	7 (see productTypeId Codes (page 156))
`productTypeName`	`"Software"`
`isIrrevocable`	`false`

Field Name	Example of Value
`associations`	`[{ "clientUserIdStr":"810C9B91-...-408AD7F081A8", "licenseIdStr":"2" },{ "clientUserIdStr":"d735c1cc-...-c74571007ef6", "licenseIdStr":"3", "errorMessage":"License already assigned", "errorNumber": 9616 },{ "serialNumber":"C17DK6D9DDQW", "licenseIdStr":"4" },{ "serialNumber":"DLXL6044FPH8", "errorMessage":"License not found", "errorNumber": 9610 }, ...]`
`disassociations`	`[{ "clientUserIdStr":"810C9B91-...-408AD7F081A8" },{ "clientUserIdStr":"d735c1cc-...-c74571007ef6", "errorMessage":"Registered user not found", "errorNumber": 9609 },{ "serialNumber":"C17DK6D9DDQW" },{ "serialNumber":"DLXL6044FPH8", "errorMessage":"License not associated", "errorNumber": 9619 }, ...]`

License Counts

The following fields are added to the `VPPServiceConfigSrv` response to indicate the maximum number of entries allowed in the `associateClientUserIdStrs`, `associateSerialNumbers`, `disassociateClientUserIdStrs`, `disassociateSerialNumbers`, or `disassociateLicenseIdStrs` arrays:

Field Name	Example of Value
`maxBatchAssociateLicenseCount`	`20`
`maxBatchDisassociateLicenseCount`	`20`

`VPPServiceConfigSrv` must be checked every 5 minutes to update the current `maxBatchAssociateLicenseCount` and `maxBatchDisassociateLicenseCount` values, which may decrease or increase without notice. Requests that exceed the current limits are rejected with the error code `9602 'Invalid Argument'`, and no work is done. If you receive this error code query `VPPServiceConfigSrv` to retrieve new `maxBatchAssociateLicenseCount` and `maxBatchDisassociateLicenseCount` values, correct the last request that was rejected and resend the request.

associateVPPLicenseWithVPPUserSrv

Note: This request is **deprecated**. Use manageVPPLicensesByAdamIdSrv (page 182) instead.

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"`.

Parameter Name	Required or Not	Example
`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=" }`

Field Name	Example of Value
`user`	`{ "userId":2, "email": "user2@test.com", "status": "Associated", "clientUserIdStr":"810C9B91-DF83-41DA-80A1-408AD7F081A8", "itsIdHash":"C2Wwd8LcIaE2v6f2/mvu82Gs/Lc=" }`
`errorMessage`	`"License not found"`.
`errorNumber`	`9602`.

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

disassociateVPPLicenseFromVPPUserSrv

Note: This request is **deprecated**. Use manageVPPLicensesByAdamIdSrv (page 182) instead.

The request takes the following parameters:

Parameter Name	Required or Not	Example
`userId`	Required.	`100001`.
`licenseId`	Required.	`100002`.
`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.

Field Name	Example of Value

`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"`.
`errorNumber`	`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 number `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"`.
`email`	Not required.	`"user1@someorg.com"`.

Parameter Name	Required or Not	Example
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.
`facilitatorMemberId`	Not required.	See Program Facilitators (page 157).
`managedAppleIDStr`	Not required.	`"user1@someorg.com"`.

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

The `managedAppleIDStr` parameter is discussed in Managed Apple IDs (page 157).

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"` `}`
`facilitatorMember`	`{` `"appleId":"user1@someorg.com",` `"countryCode":"US",` `"email":"user1@someorg.com",` `"facilitatorMemberId":200843,` `"organizationId":2168850000179778,` `},`
`errorMessage`	`"Missing \"userId\" input parameter"`.

Field Name	Example of Value
`errorNumber`	`9600`.

VPPClientConfigSrv

This service allows the client to store some information on the server on a per-organization basis. The information that currently can be stored is a `clientContext` string. The `clientContext` string is any JSON string less than 256 bytes in length. For format information, see Service Response (page 153).

The request takes the following parameters:

Parameter Name	Required or Not	Example
`clientContext`	Not required.	(any string less than 256 bytes)
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.
`verbose`	Not required.	`"true"`.

If a value is provided for `clientContext`, the value is stored by the server and the response contains the current value of this field. To clear the field value, provide an empty string as the input value; that is, "". If `"verbose":true"` is included in the request, the response contains the `appleId` field.

The response to `VPPClientConfigSrv` contains some of these fields:

Field Name	Example of Value
`status`	`0` for success, `-1` for error.
`clientContext`	`"abc"`
`errorMessage`	`"Login required"`.
`errorNumber`	`9601`.
`countryCode`	`"US"`.
`appleId`	`"user1@someorg.com"`.
`email`	`"user1@someorg.com"`.
`facilitatorMemberId`	`"200841"`.
`organizationId`	`"2168850000179778"`.

Field Name	Example of Value
`vppGroupMembers`	See Program Facilitators (page 157).

The `countryCode` value in the response is the ISO 3166-1 two-letter code designating the country where the VPP account is located. For example, "US" for United States, "CA" for Canada, "JP" for Japan, and so on.

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 numbers 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 request takes the following parameters:

Parameter Name	Required or Not	Example
`sToken`	Required.	`"h40Gte9aQnZFDNM...6ZQ="`.

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"`.

Field Name	**Example of Value**
`getUserSrvUrl`	`"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPUserSrv"`.
`retireUserSrvUrl`	`"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/retireVPPUserSrv"`.
`getUsersSrvUrl`	`"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPUsersSrv"`.
`getLicensesSrvUrl`	`"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/getVPPLicensesSrv"`.
`associateLicenseSrvUrl`	`"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/associateVPPLicenseWithVPPUserSrv"`.
`disassociateLicenseSrvUrl`	`"https://vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/disassociateVPPLicenseFromVPPUserSrv"`.

Field Name	Example of Value
`errorCodes`	`[{ "errorMessage":"Missing required argument", "errorCode":9600 }, { "errorMessage":"Login required", "errorCode":9601 }, { "errorMessage":"Invalid argument", "errorCode":9602 }, { "errorMessage":"Internal error", "errorCode":9603 }, { "errorMessage":"Result not found", "errorCode":9604 }, { "errorMessage":"Account storefront incorrect", "errorCode":9605 }, { "errorMessage":"Error constructing token", "errorCode":9606 }, { "errorMessage":"License irrevocable", "errorCode":9607 }, { "errorMessage":"Empty SharedData response", "errorCode":9608 }, { "errorMessage":"User not found", "errorCode":9609 }, { "errorMessage":"Lincese not found", "errorCode":9610 }, { "errorMessage":"Admin user not found", "errorCode":9611 }, { "errorMessage":"Fail creating SAPFeeder job for claim", "errorCode":9612 }, { "errorMessage":"Fail creating SAPFeeder job for unclaim", "errorCode":9613 }, { "errorMessage":"Invalid date formate", "errorCode":9614 }, { "errorMessage":"orgCountry not found", "errorCode":9615 }, { "errorMessage":"License already assigned to the iTunes account", "errorCode":9616 }, { "errorMessage":"User already retired", "errorCode":9618 }, { "errorMessage":"License not associated", "errorCode":9619 }, { "errorMessage":"User already deleted", "errorCode":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.", "errorCode":9621 }, { "errorMessage":"The authentication token is invalid.", "errorCode":9622 }, { "errorMessage":"The APN token is invalid.", "errorCode":9623 } { "errorMessage":"License was refunded and is no longer valid.", "errorCode":9624 } { "errorMessage":"The sToken has been revoked.", "errorCode":9625 } { "errorMessage":"License already assigned to a different user.", "errorCode":9626 }]`
`clientConfigSrvUrl`	`"https: //vpp.itunes.apple.com/WebObjects/MZFinance.woa/wa/ VPPClientConfigSrv"`.

Field Name	Example of Value
`maxBatchAssociate-LicenseCount`	`20`
`maxBatchDisassociate-LicenseCount`	`20`

Examples

The following are examples of requests and responses of each service. The requests are made with the 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:

```
{
    "associateLicenseSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/associateVPPLicenseSrv",
    "clientConfigSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/VPPClientConfigSrv",
    "contentMetadataLookupUrl":"https://uclient-api.itunes.apple.com/
WebObjects/MZStorePlatform.woa/wa/lookup",
    "disassociateLicenseSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/disassociateVPPLicenseSrv",
    "editUserSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/editVPPUserSrv",
    "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 is irrevocable",
            "errorNumber":9607
        },
        {
            "errorMessage":"Empty response from SharedData service",
            "errorNumber":9608
        },
        {
            "errorMessage":"Registered user not found",
            "errorNumber":9609
        },
        {
            "errorMessage":"License not found",
            "errorNumber":9610
        },
        {
            "errorMessage":"Admin user not found",
            "errorNumber":9611
        },
        {
            "errorMessage":"Failed to create claim job",
            "errorNumber":9612
        },
        {
            "errorMessage":"Failed to create unclaim job",
            "errorNumber":9613
        },
        {
            "errorMessage":"Invalid date format",
            "errorNumber":9614
        },
        {
            "errorMessage":"OrgCountry not found",
            "errorNumber":9615
        },
        {
            "errorMessage":"License already assigned",
            "errorNumber":9616
        },
        {
            "errorMessage":"The user has already been retired.",
            "errorNumber":9618
        },
        {
            "errorMessage":"License not associated",
            "errorNumber":9619
        },
        {
            "errorMessage":"The user has already been 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":"Invalid authentication token",
            "errorNumber":9622
        },
        {
            "errorMessage":"Invalid APN token",
            "errorNumber":9623
        },
        {
            "errorMessage":"License was refunded and is no longer valid.",
            "errorNumber":9624
        },
        {
            "errorMessage":"The sToken has been revoked",
            "errorNumber":9625
        },
        {
            "errorMessage":"License already assigned to other user",
            "errorNumber":9626
        },
        {
            "errorMessage":"License disassociation fail due to frequent
reassociation",
            "errorNumber":9627
        },
        {
            "errorMessage":"License not eligible for device assignment.",
            "errorNumber":9628
        },
        {
            "errorMessage":"The sToken is inapplicable to batchToken",
            "errorNumber":9629
        }
    ],
    "getLicensesSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/getVPPLicensesSrv",
    "getUserSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/getVPPUserSrv",
    "getUsersSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/getVPPUsersSrv",
    "getVPPAssetsSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/getVPPAssetsSrv",
    "invitationEmailUrl":"https://buy.itunes.apple.com/
WebObjects/MZFinance.woa/wa/associateVPPUserWithITSAccount?cc=us&inviteCode=%25inviteCode%25&mt=8",
    "manageVPPLicensesByAdamIdSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/manageVPPLicensesByAdamIdSrv",
    "maxBatchAssociateLicenseCount":100,
    "maxBatchDisassociateLicenseCount":100,
    "registerUserSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/registerVPPUserSrv",
    "retireUserSrvUrl":"https://vpp.itunes.apple.com/
WebObjects/MZFinance.woa/wa/retireVPPUserSrv",
    "status":0,
    "vppWebsiteUrl":"https://vpp.itunes.apple.com/"
}
```

Request to getVPPLicensesSrv

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

```
{"sToken":"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxef0uQkeeb3h2
a5Rlopo4KDn3MrFKf4CM30Y+WGAoZ1cD6iZ6yzsMk1+5PVBNc66YS6ZQ="}
```

The curl command:

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

The response:

```
[
    {
        "adamId":408709785,
        "adamIdStr":"408709785",
        "clientUserIdStr":"9a17b450-9820-471e-b232-13a479ddede0",
        "isIrrevocable":false,
        "itsIdHash":"LsrJ6NhzbsOzQXShrpUTWGnD/X8=",
        "licenseId":102547,
        "licenseIdStr":"102547",
        "pricingParam":"STDQ",
        "productTypeId":8,
        "productTypeName":"Application",
        "status":"Associated",
        "userId":10715446,
        "userIdStr":"10715446"
    },
    {
        "adamId":435160039,
        "adamIdStr":"435160039",
        "clientUserIdStr":"9a17b450-9820-471e-b232-13a479ddede0",
        "isIrrevocable":true,
        "itsIdHash":"LsrJ6NhzbsOzQXShrpUTWGnD/X8=",
        "licenseId":795047681,
        "licenseIdStr":"795047681",
        "pricingParam":"PLUS",
        "productTypeId":10,
        "productTypeName":"Publication",
        "status":"Associated",
        "userId":6561022,
        "userIdStr":"6561022"
    },
    {
        "adamId":645859810,
        "adamIdStr":"645859810",
        "isIrrevocable":false,
        "licenseId":967494668,
        "licenseIdStr":"967494668",
        "pricingParam":"STDQ",
        "productTypeId":8,
        "productTypeName":"Application",
        "serialNumber":"C39N3035G68P",
        "status":"Associated"
    }
]
```

Request to getVPPUsersSrv

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

```
{"sToken":"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxef0uQkeeb3h2
a5Rlopo4KDn3MrFKf4CM30Y+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":"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxef0uQ
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":
"h40Gte9aQnZFDNM39IUkRPCsQDxBxbZB4Wy34pxef0uQkeeb3h2a5Rlopo4KDn3MrFKf4CM3OY+
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",
              "isIrrevocable":true
          }
      ]
   }
}
```

Class Rosters

This chapter describes a system of MDM APIs, introduced with iOS 9.3, that retrieve roster information for schools and other personnel-based organizations. These APIs form an extension to the Device Enrollment Program API, so the DEP initial authentication steps are required before sending requests to the roster service.

> **Note:** The following APIs are read-only. It is not possible to change roster information through MDM.

Class Roster Information

This API returns class roster information for an organization at a given location.

Requests

To access this information, POST a request in JSON format and UTF-8 charset to the following URL: `https://mdmenrollment.apple.com/roster/class`. The request body should contain a JSON dictionary with the following keys:

Key	Type	Content
`cursor`	String	Optional. A hex string that represents the starting position for a request. This is used for pagination. On the initial request, this should be omitted.
`limit`	Integer	Optional. The maximum number of entries to return. The default value is 1000 and the maximum value is 1000.

With its required header, a typical request looks like this:

```
POST /roster/class HTTP/1.1
User-Agent:<client-software-information>
Accept-Encoding: gzip, deflate
X-Server-Protocol-Version:2
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
```

```
Content-Type: application/json;charset=UTF8
Content-Length: <Content-Length>
{
"limit": 1000,
"cursor": "1ac73329f75817"
}
```

Responses

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

Key	Type	Content
`cursor`	String	Optional. A hex string that should be used for the next request to paginate. This field data type has a maximum length of 512 UTF-8 characters.
`more_to_follow`	Boolean	Indicates whether the request's limit and cursor values resulted in only a partial list of classes. If true, the MDM server should then make another request (starting from the newly returned cursor) to obtain additional records.
`classes`	Array of dictionaries	Provides information about classes, sorted in lexical order by a class source_system_identifier. The organization must provide this identifier to Apple.

Each dictionary in the `classes` array contains these keys:

Key	Type	Content
`name`	String	Optional. Class name. Maximum length is 1024 UTF-8 characters.
`source`	String	Data source where class was created. Possible values include "iTunes U," "SIS," "CSV," "SFTP," and "MANUAL." Maximum length is 64 UTF-8 characters.
`unique_identifier`	String	Unique identifier for the class. Maximum length is 256 UTF-8 characters.
`source_system_-identifier`	String	Optional. Identifier configured by the organization for its classes. Maximum length is 256 UTF-8 characters. See Note below.

Key	Type	Content
room	String	Optional. Room where class is held. Maximum length is 512 UTF-8 characters.
location	Dictionary	Geographical or organizational location where class is held (see below).
course	Dictionary	Course definition for the class (see below).
instructor_unique_-identifiers	Array of strings	Unique identification for instructors. Each string in the array has a maximum length of 256 UTF-8 characters.
student_unique_-identifiers	Array of strings	Unique identification for students. Each string in the array has a maximum length of 256 UTF-8 characters.

Note: The value of `source_system_identifier` in this and other roster API responses is not guaranteed to be unique and can potentially change.

The `location` dictionary contains the following keys:

Key	Type	Content
name	String	Location name. Maximum length 1024 UTF-8 characters.
unique_identifier	String	Unique identifier for the location. Maximum length 256 UTF-8 characters.

The `course` dictionary contains the following keys:

Key	Type	Content
name	String	Optional. Course name. Maximum length 1024 UTF-8 characters.
unique_identifier	String	Unique identifier for the course. Maximum length 256 UTF-8 characters.

The response contains a list of classes. Each class record contains the location where the class is held and the instructors and students that are registered for that class. It also identifies the course with which the class is associated. The `more_to_follow` Boolean indicates if more class information remains to be fetched. The client should read this flag to determine if subsequent requests are necessary to get the next batch of classes.

The class list could be huge. If modifications are performed while the response is being returned, it will not return any classes created after it started responding. If any updates are applied on any of the entities or attributes, you must send the request again to get the latest snapshot of classes.

One record in a typical response might look like this:

```
{
  "classes": [
    {
      "unique_identifier": "UNICLS1003",
      "source": "SIS",
      "source_system_identifier": "CLSBIO101",
      "name": "Miss Smith's Biology 101",
      "room": "Hall 101",
      "location": {
        "unique_identifier": "UNILOC1003",
        "name": "Biology department"
      },
      "instructor_unique_identifiers": [
        "UNIINSTID1003",
        "UNIINSTID1003"
      ],
      "student_unique_identifiers": [
        "UNISTUDID1003",
        "UNISTUDID1004"
      ],
      "course": {
        "unique_identifier": "UNICOURID1003",
        "name": "Biology 101"
      }
    }
  ],
  "cursor": "1ac73329f75816",
  "more_to_follow": "false"
}
```

Person Roster Information

This API returns roster information for an organization about instructors and students who are registered in any class at a given location.

Requests

To access this information, POST a request in JSON format and UTF-8 charset to the following URL: `https://mdmenrollment.apple.com/roster/class/person`. The request body should contain a JSON dictionary with the following keys:

Key	Type	Content
`cursor`	String	Optional. A hex string that represents the starting position for a request. This is used for pagination. On the initial request, this should be omitted.
`limit`	Integer	Optional. The maximum number of entries to return. The default value is 1000 and the maximum value is 1000.

With its required header, a typical request looks like this:

```
POST /roster/class/person HTTP/1.1
User-Agent:<client-software-information>
Accept-Encoding: gzip, deflate
X-Server-Protocol-Version:2
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Type: application/json;charset=UTF8
Content-Length: <Content-Length>
{
"limit": 1000,
"cursor": "1ac73329f75817"
}
```

Responses

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

Key	Type	Content
`cursor`	String	Optional. A hex string that should be used for the next request to paginate. This field data type has a maximum length of 512 UTF-8 characters.
`more_to_follow`	Boolean	Indicates whether the request's limit and cursor values resulted in only a partial list of persons. If true, the MDM server should then make another request (starting from the newly returned cursor) to obtain additional records.
`persons`	Array of dictionaries	Provides information about persons, both teachers and students, sorted in lexical order by a person source_system_identifier. The organization must provide this identifier to Apple.

Each `persons` dictionary contains the following keys:

Key	Type	Content
`name`	String	Person's name. Maximum length 1024 UTF-8 characters.
`managed_apple_id`	String	Managed Apple ID for the person. Maximum length 1024 UTF-8 characters.
`unique_identifier`	String	Unique identifier for the person. Maximum length 256 UTF-8 characters.
`source`	String	Data source where class was created. Possible values include "iTunes U," "SIS," "CSV," "SFTP," and "MANUAL." Maximum length is 64 UTF-8 characters.
`source_system_-identifier`	String	Identifier configured by organization for the person. Maximum length 256 UTF-8 characters.
`grade`	String	Optional; not used for instructors. Student grade information. Maximum length 256 UTF-8 characters. Value can be null.

The response contains a list of persons. The `more_to_follow` Boolean indicates if more information about persons remains to be fetched. The client should read this flag to determine if subsequent requests are necessary to get the next batch of persons.

The person list could be huge. If modifications are performed while the response is being returned, it will not return any persons enrolled after it started responding. If any updates are applied on any of the entities or attributes, you must send the request again to get the latest snapshot of personnel.

One record in a typical response might look like this:

```
HTTP/1.1 200 OK
Date: Mon,12 Oct 2015 02:25:30 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: ...
Connection: Keep-Alive

{
  "persons": [
    {
      "unique_identifier": "UNIINSTID1003",
      "source": "CSV",
      "source_system_identifier": "INSTID1003",
      "name": "Miss Smith",
      "managed_apple_id": "smith@example.com"
    },
    {
      "unique_identifier": "UNISTUDID1003",
      "source": "SIS",
      "source_system_identifier": "INSTSTUDID1003",
      "name": "John Smith",
      "managed_apple_id": "john@example.com",
      "grade": "K"
    }
  ],
  "cursor": "1ac73329f75816",
  "more_to_follow": "false"
}
```

Location Information

This API returns information for an organization about the locations where any classes are held.

Requests

To access this information, POST a request in JSON format and UTF-8 charset to the following URL: `https://mdmenrollment.apple.com/roster/class/location`. The request body should contain a JSON dictionary with the following keys:

Key	Type	Content
`cursor`	String	Optional. A hex string that represents the starting position for a request. This is used for pagination. On the initial request, this should be omitted.
`limit`	Integer	Optional. The maximum number of entries to return. The default value is 1000 and the maximum value is 1000.

With its required header, a typical request looks like this:

```
POST /roster/class/location HTTP/1.1
User-Agent:<client-software-information>
Accept-Encoding: gzip, deflate
X-Server-Protocol-Version:2
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Type: application/json;charset=UTF8
Content-Length: <Content-Length>
{
"limit": 1000,
"cursor": "1ac73329f75817"
}
```

Responses

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

Key	Type	Content
`cursor`	String	Optional. A hex string that should be used for the next request to paginate. This field data type has a maximum length of 512 UTF-8 characters.
`more_to_follow`	Boolean	Indicates whether the request's limit and cursor values resulted in only a partial list of locations. If true, the MDM server should then make another request (starting from the newly returned cursor) to obtain additional records.

Key	Type	Content
`locations`	Array of dictionaries	Provides information about locations, sorted in lexical order by a location source_system_identifier. The organization must provide this identifier to Apple.

Each `locations` dictionary contains the following keys:

Key	Type	Content
`name`	String	Location name. Maximum length 1024 UTF-8 characters.
`unique_identifier`	String	Unique identifier for the location. Maximum length 256 UTF-8 characters.
`source_system_-identifier`	String	Identifier configured by organization for the location. Maximum length 256 UTF-8 characters.
`source`	String	Data source where class was created. Possible values include “iTunes U,” “SIS,” “CSV,” "SFTP," and “MANUAL.” Maximum length 64 UTF-8 characters.

The response contains a list of locations. The `more_to_follow` Boolean indicates if more information about locations remains to be fetched. The client should read this flag to determine if subsequent requests are necessary to get the next batch of locations.

If modifications to locations are performed while the response is being returned, it will not return any locations rostered after it started responding. If any updates are applied on any of the entities or attributes, you must send the request again to get the latest snapshot of locations in use.

One record in a typical response might look like this:

```
HTTP/1.1 200 OK
Date: Mon,12 Oct 2015 02:25:30 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: ...
Connection: Keep-Alive


{
  "locations": [
    {
      "unique_identifier": "UNILOC1003",
      "source": "SIS",
      "source_system_identifier": "INSTLOCID1003",
      "name": "Biology department"
    }
  ],
  "cursor": "1ac73329f75816",
  "more_to_follow": "false"
}
```

Course Roster Information

This API returns course information for an organization.

Requests

To access this information, POST a request in JSON format and UTF-8 charset to the following URL: `https://mdmenrollment.apple.com/roster/course`. The request body should contain a JSON dictionary with the following keys:

Key	Type	Content
`cursor`	String	Optional. A hex string that represents the starting position for a request. This is used for pagination. On the initial request, this should be omitted.
`limit`	Integer	Optional. The maximum number of entries to return. The default value is 1000 and the maximum value is 1000.

With its required header, a typical request looks like this:

```
POST /roster/course HTTP/1.1
User-Agent:<client-software-information>
Accept-Encoding: gzip, deflate
X-Server-Protocol-Version:2
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Type: application/json;charset=UTF8
```

```
Content-Length: <Content-Length>
{
"limit": 1000,
"cursor": "1ac73329f75817"
}
```

Responses

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

Key	**Type**	**Content**
`cursor`	String	Optional. A hex string that should be used for the next request to paginate. This field data type has a maximum length of 512 UTF-8 characters.
`more_to_follow`	Boolean	Indicates whether the request's limit and cursor values resulted in only a partial list of courses. If true, the MDM server should then make another request (starting from the newly returned cursor) to obtain additional records.
`courses`	Array of dictionaries	Provides information about courses, sorted in lexical order by a course source_system_identifier. The organization must provide this identifier to Apple.

Each `courses` dictionary contains the following keys:

Key	**Type**	**Content**
`name`	String	Optional. Course name. Maximum length 1024 UTF-8 characters.
`unique_identifier`	String	Unique identifier for the course. Maximum length 256 UTF-8 characters.
`source`	String	Data source where class was created. Possible values include “iTunes U,” “SIS,” “CSV,” "SFTP," and “MANUAL.” Maximum length 64 UTF-8 characters.
`source_system_-identifier`	String	Optional. Identifier configured by organization for the course. Maximum length is 256 UTF-8 characters. Value can be null.

The response contains a list of courses. The `more_to_follow` Boolean indicates if more information about courses remains to be fetched. The client should read this flag to determine if subsequent requests are necessary to get the next batch of courses.

If modifications to the course catalog are performed while the response is being returned, it will not return any courses rostered after it started responding. If any updates are applied on any of the entities or attributes, you must send the request again to get the latest snapshot of courses.

One record in a typical response might look like this:

```
HTTP/1.1 200 OK
Date: Mon,12 Oct 2015 02:25:30 GMT
Content-Type: application/json;charset=UTF8
X-ADM-Auth-Session: 87a235815b8d6661ac73329f75815b8d6661ac73329f815
Content-Length: ...
Connection: Keep-Alive

{
  "courses": [
    {
      "unique_identifier": "UNICOURID1003",
      "source": "SIS",
      "source_system_identifier": "INSTCOURSEID1003",
      "name": "Biology 101"
    }
  ],
  "cursor": "1ac73329f75816",
  "more_to_follow": "false"
}
```

Error Responses

Instead of the information responses described earlier in this chapter, MDM roster requests may return system errors. You must read and respond to three kinds of errors:

- Server failures
- Client failures

- MDM errors

Server failures are mainly HTTP 500 and HTTP 503 errors:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain;charset=UTF8
Content-Length: 0
Date: Thu, 22 Oct 2015 21:23:57 GMT
Connection: close,

HTTP/1.1 503 Service Unavailable
Content-Type: text/plain;charset=UTF8
Retry-After: 120
Content-Length: 0
Date: Thu, 22 Oct 2015 21:23:57 GMT
Connection: close
```

Client failures are HTTP 4xx-series or HTTP 429 errors:

```
HTTP/1.1 4xx <Error Reason>
Content-Type: text/plain;Charset=UTF8
Content-Length: 10
Date: Thu, 22 Oct 2015 21:23:57 GMT
Connection: close

<ERROR CODE>

HTTP/1.1 429 <Error Reason>
Content-Type: text/plain;Charset=UTF8
Content-Length: 10
Retry-After: 10
Date: Thu, 22 Oct 2015 21:23:57 GMT
Connection: close

<ERROR CODE>
```

Client failures may return MDM error codes. When combined with HTTP codes, these errors give you the following information:

- UNAUTHORIZED + HTTP 401: Auth token has expired. The client should retry with a new auth token.
- FORBIDDEN + HTTP 403: Auth token is invalid.
- MALFORMED_REQUEST_BODY + HTTP 400: The request body is malformed.
- CURSOR_REQUIRED + HTTP 400: The cursor is missing in the request.
- INVALID_CURSOR + HTTP 400: The cursor in the request is invalid.
- EXPIRED_CURSOR + HTTP 400: The cursor is older than 1 day.
- TOO_MANY_REQUESTS + HTTP 429: Too many requests. Retry after time mentioned in "Retry-After" HTTP response header as per RFC 6585.

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 in OS X, installing an MDM profile causes 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).

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 when returning to the Home screen, but can dismiss the prompt and continue working. After the 60-minute grace period, the user must change the 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.

Here are some general best practices:

- It is best practice to register VPP users and assign apps/books to those users before sending invitations to the users. This makes each assignment faster because it does not need to put the item in the user's purchases at the time of assignment. Also, because an invitation acceptance will likely occur well before an MDM InstallApplication command is issued, the odds are higher that all licenses will have long since propagated to the user's iTunes Store purchase history on the user's clients, which is a necessary step for the `InstallApplication` command to succeed.
- It is best practice to invite an individual user to each VPP organization only once. By checking the `itsIdHash`, MDM servers can detect when a single Apple ID accepts multiple invitations. Attempting to assign licenses for the same item to multiple VPP users with the same `itsIdHash` results in an "Already Assigned" error (code 9616).
- It is best practice to provide a helpful error message when receiving error 403, T_C_NOT_SIGNED, such as "Terms and Conditions must be accepted. Please log into the Device Enrollment Program to accept the new Terms and Conditions on behalf of your organization."

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 228). 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.

The 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 the 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.

You can also install a `trust_profile_url`, as described in Adding MDMServiceConfig Functionality (page 231).

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 226).

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 carries 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. Use this option only if necessary.

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.

Note: Your security report on each managed device should specify whether or not MDM is set to be non-removable. This information is returned by the profile query, as described in Define Profile (page 135).

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 cache your last push notification and 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.

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. MDM service is reinstated only if the backup is restored to the same device. It is not reinstated if the user restores a backup to a new device.

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. 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.

Do 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 9.0 and Later

In iOS 9.0 and later, you can use MDM’s app assignment feature to assign app licenses to device serial numbers. MDM can then be used to push a VPP app to a device regardless of whether an iTunes account is signed in. You can later remove those licenses and use them with other devices.

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 installation. (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 223), *do not* rely solely 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 these 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, 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), 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 shows only other managed apps and accounts. When the `allowOpenFromUnmanagedToManaged` restriction is specified, an Open In sheet started from within an unmanaged app or account shows 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.

Adding MDMServiceConfig Functionality

To simplify administration using Apple Configurator (or other tools in the future) you can add an unauthenticated HTTPS request entry point to your server, labeled with the Uniform Resource Identifier `/MDMServiceConfig`. The resulting URL would have the form `https://mdm.example.com/MDMServiceConfig`. The server code should return in the body of its response a UTF-8 JSON-encoded hash (Content-Type: application/json; charset=UTF8) with some or all of the following keys, the values of which should be fully-functional URLs.

Key	Value
dep_enrollment_url	This is the URL the device should contact to begin MDM enrollment with the MDM server. It should have the same value the server would send for the url key when defining a DEP profile via https://mdmenrollment.apple.com/profile, as described in Define Profile (page 135).
dep_anchor_certs_url	This is the URL that a client can use to obtain the certificates required to trust the URL specified by the dep_enrollment_url key. It is the exact same format as the anchor_certs value in the DEP profile, except the body needs to be UTF-8 JSON-encoded for transfer. The decoded body of the response from this URL should be usable in a DEP profile under the anchor_certs key without any modification. If the MDM server is using a trusted SSL certificate (so no additional certs are required), this URL should still be provided but the body of the response to the URL should either be empty (Content-Length: 0) or the JSON string for an empty array ('[]').
trust_profile_url	This is the URL a client can use to obtain a Trust Profile for the MDM server. This should be a fully formed .mobileconfig profile with only payloads of type com.apple.security.root. If the server is using trusted certificates (so no Trust Profile is required), this key should be omitted from the response. Do not return a URL that would generate an empty profile.

Note: Although the foregoing keys are individually optional, it is recommended that dep_enrollment_url and dep_anchor_certs_url be implemented or not as a pair.

Examples

Below are examples of code that implements /MDMServiceConfig.

The MDMServiceConfig Request

Request Format

```
GET https://mdm.example.com/MDMServiceConfig
```

Response Body

```
{
    "dep_enrollment_url":
"https://mdm.example.com/devicemanagement/mdm/dep_mdm_enroll",
    "dep_anchor_certs_url":
"https://mdm.example.com/devicemanagement/mdm/dep_anchor_certs",
    "trust_profile_url": "https://certs.example.com/mdm/trust_profile"
}
```

It is not required that the URLs refer to the same host as the `/MDMServiceConfig` request, as illustrated by the example for `trust_profile_url`.

The dep_anchor_certs_url Key

Request Format

```
GET https://mdm.example.com/devicemanagement/mdm/dep_anchor_certs
```

Response Body (truncated for clarity)

```
["MIIEKDCCAxCgAwIBAgIEOjznoTALBgkqhkiG9w0BAQswfjEkMCIGA1UEAwwbU3ly
\nYWggQ2VydGlmaWNhd...SVVTo9ll1Lv3OJGqBkxPl9TCC\nfYYnArwzlk4qm1tP\n"]
```

The trust_profile_url Key

Request Format

```
GET https://certs.example.com/mdm/trust_profile
```

Response Body (truncated for clarity)

```
<?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>PayloadContent</key>
```

```
<array>
 <dict>
  <key>PayloadContent</key>
  <data>
  MIIEKDCCAxCgAwIBAgIEOjznoTALBgkqhkiG9w0BAQswfjEkMCIG
  ...
  9TCCfYYnArwzlk4qm1tP
  </data>
  <key>PayloadDescription</key>
  <string>Installs the Root certificate for Example Corp.</string>
  <key>PayloadDisplayName</key>
  <string>Root certificate for Example Corp</string>
  <key>PayloadIdentifier</key>
  <string>com.apple.ssl.certificate</string>
  <key>PayloadOrganization</key>
  <string>Example Corp</string>
  <key>PayloadType</key>
  <string>com.apple.security.root</string>
  <key>PayloadUUID</key>
  <string>B90FA650-5A7D-496A-8C84-0D81C9EBCE6E</string>
  <key>PayloadVersion</key>
  <integer>1</integer>
 </dict>
</array>
<key>PayloadDescription</key>
<string>Configures your device to trust the MDM server.</string>
<key>PayloadDisplayName</key>
<string>Trust Profile for Example Corp</string>
<key>PayloadIdentifier</key>
<string>com.apple.config.mdm.example.com.ssl</string>
<key>PayloadScope</key>
<string>System</string>
<key>PayloadType</key>
<string>Configuration</string>
```

```
    <key>PayloadUUID</key>
    <string>94cdf5c0-bde0-0131-1ed5-005056831d08</string>
    <key>PayloadVersion</key>
    <integer>1</integer>
</dict>
</plist>
```

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, 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 238), Listing 2 (page 239), and Listing 3 (page 239) 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 the customer 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 in to 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
2016-08-05	Made minor updates. Added SFTP as an option for the Source key.
2016-06-10	Made miscellaneous updates and corrections throughout. Added new section Escrow Keys and Bypass Codes (page 132).
2016-01-20	Updated for iOS 9.3. Added new chapter Class Rosters (page 208). Made other updates and corrections throughout.
2015-10-22	Updated for iOS 9 and OS X 10.11. Added new section manageVPPLicensesByAdamIdSrv (page 182). Added new section DeviceConfigured (page 81). Added new section Software Update (page 81). Added new section “Setup Configuration Command.” Added `HostName` queries to Table 7 (page 43). Clarified book installation; see Installed Books (page 68). Added restrictions to `DeviceName` setting; see DeviceName Sets the Name of the Device (page 75).

Date	Notes
	Updated Fetch Profile (page 143). Made miscellaneous updates and corrections.
2015-03-12	Made miscellaneous updates and corrections. Deprecated Disown Devices endpoint; see Disown Devices (page 125). Deprecated `facilitator_id` key; see Account Details (page 112).
2014-11-03	Updated Device Enrollment Program API to X-Server-Protocol-Version 2. Added new section MDM Protocol Extensions (page 26). Added new section Installed Books (page 68). Added new section Adding MDMServiceConfig Functionality (page 231). Made additional updates and corrections throughout.
2014-05-30	Updated for iOS 8.0 and OS X 10.10.
2014-03-19	Updated for iOS 7.1
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.

Date	Notes
2011-02-16	Updated for CDMA support.
2010-12-09	Updated for iOS 4.2.
2010-09-14	First version.