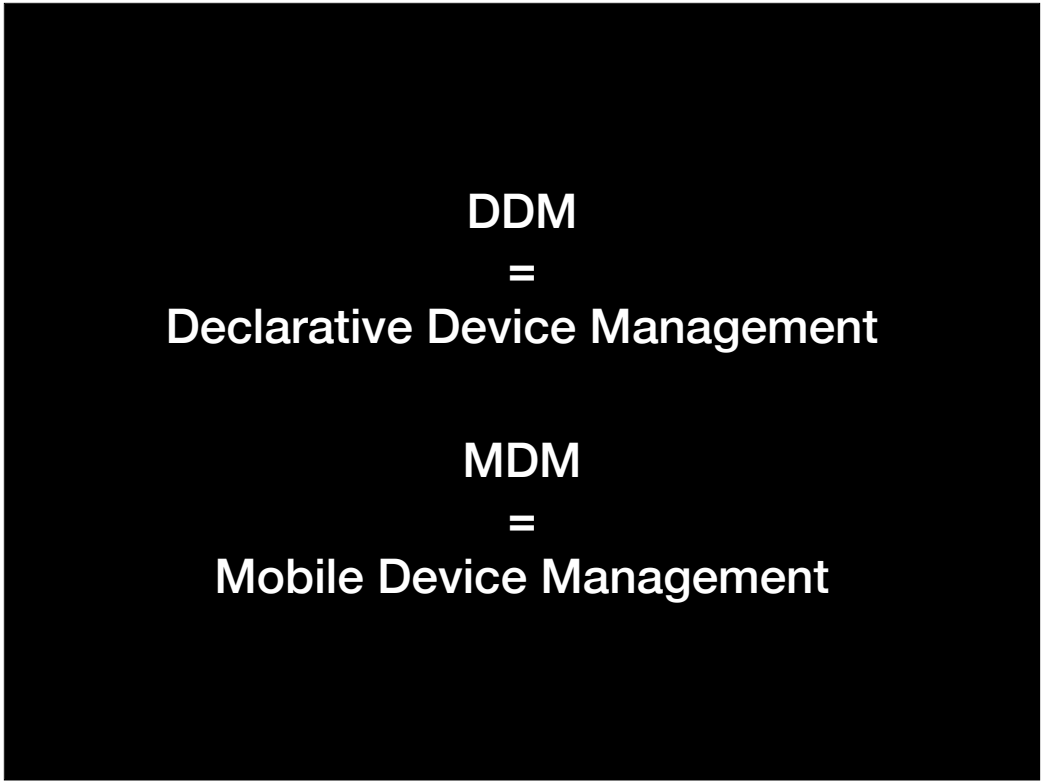
Introduction to Declarative MDM

Before we get started, there's two things I'd like to mention. The first is that, all of the sides, speakers' notes and the demos are available for download and I'll be providing a link at the end of the talk. I tend to be one of those folks who can't keep up with the speaker and take notes at the same time, so for those folks in the same situation, no need to take notes. Everything I'm covering is going to be available for download.



I’m also going to be using the following acronyms frequently during this talk, DDM and MDM. Here’s what they stand for. If you don’t understand what they mean now, hopefully you will by the end of the talk.

Let’s start by getting on the same page. What is MDM? In the context of Apple devices, Mobile Device Management or MDM is a delivery mechanism for distributing data and settings to Apple devices. This includes Macs, iPhones, iPads and Apple TVs, with Apple Watch now being added.

- **API for sending device management commands**
  - **For iOS, usable on iOS 4.x and later.**
  - **For macOS, usable on 10.7.x and later.**
  - **For tvOS, usable on tvOS 10.x and later.**
- **Not all MDM commands are backwards-compatible**

More specifically, MDM is an API for sending device management commands. Not all of these commands are backwards-compatible, so MDM commands for a specific OS version and later won't work on earlier OS versions.

When did MDM enter the picture? Apple first rolled it out starting with iOS 4 and it came to the Mac platform starting with Lion. tvOS is an outgrowth of iOS 9, so the ability to use MDM has been there on tvOS from the start.

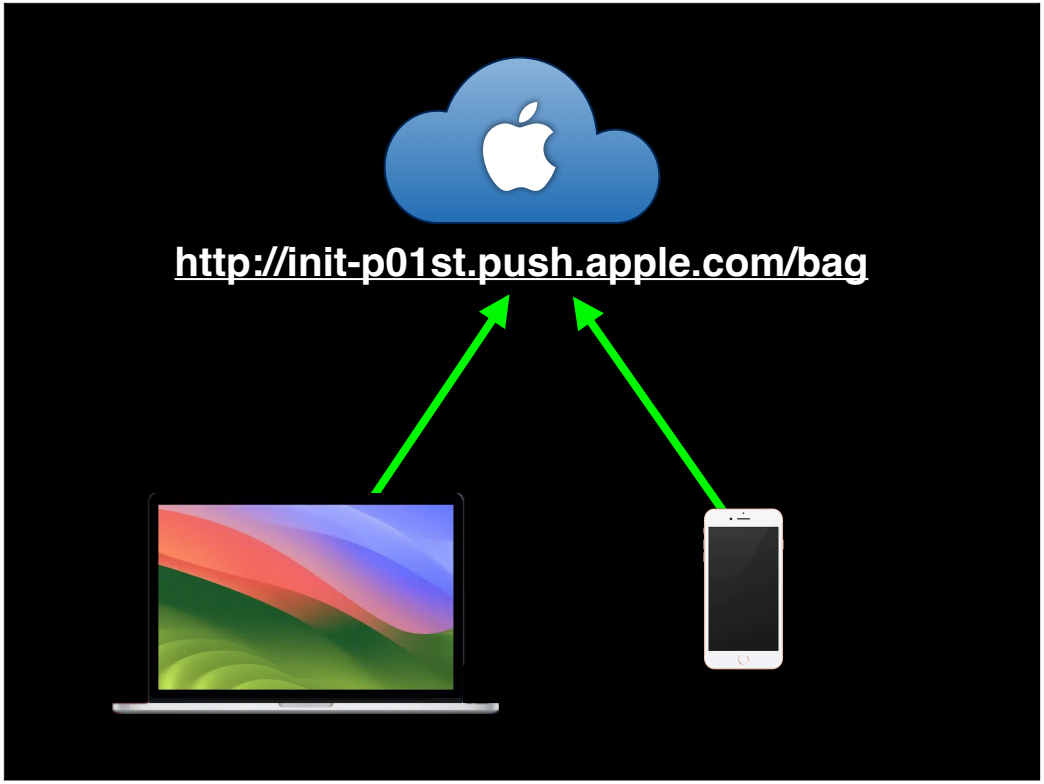
You can use MDM to push email, security settings, applications, app settings, certificates and even media content to Apple devices.

How does MDM management work?

- **Apple's push notification services (APNS)**
- **Mobile device management (MDM) server**

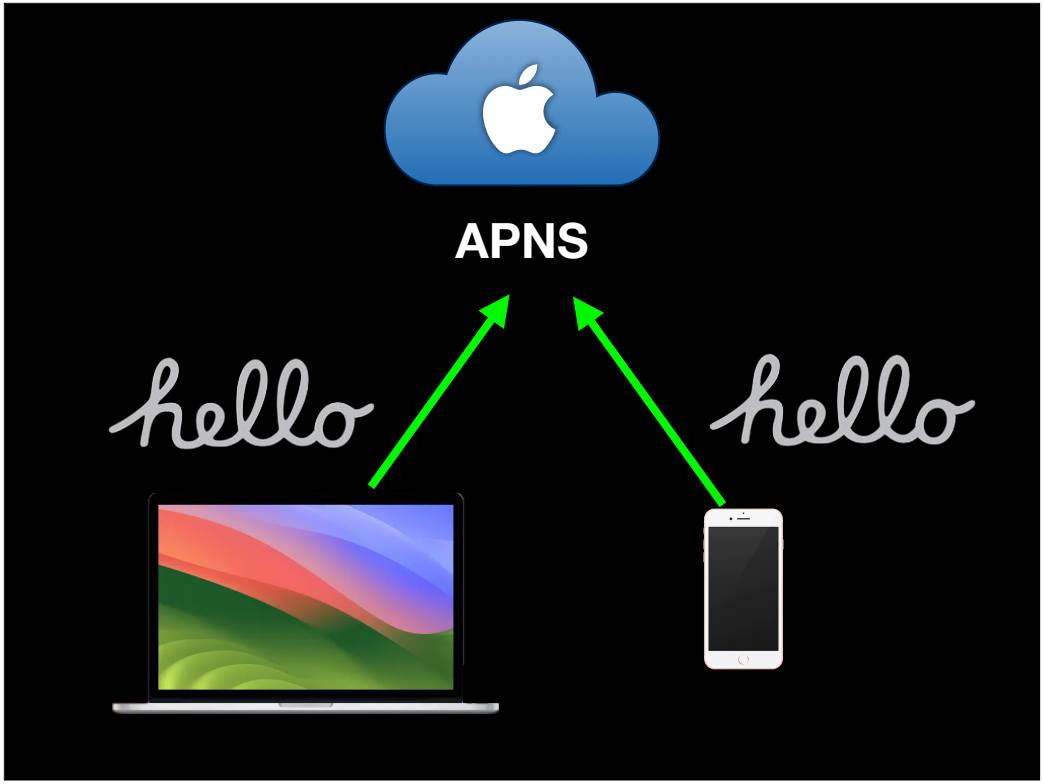
There are two major components to MDM. There’s Apple’s Push Notification Service and then there’s the MDM server used to manage those devices. Let’s take a look at APNS first.

When you take your new Apple device out of its box and power it on for the first time, it doesn’t yet know how to connect to APNS. This is because it may have been sitting in its box for a while and Apple wants to make sure that it’s getting the latest information.

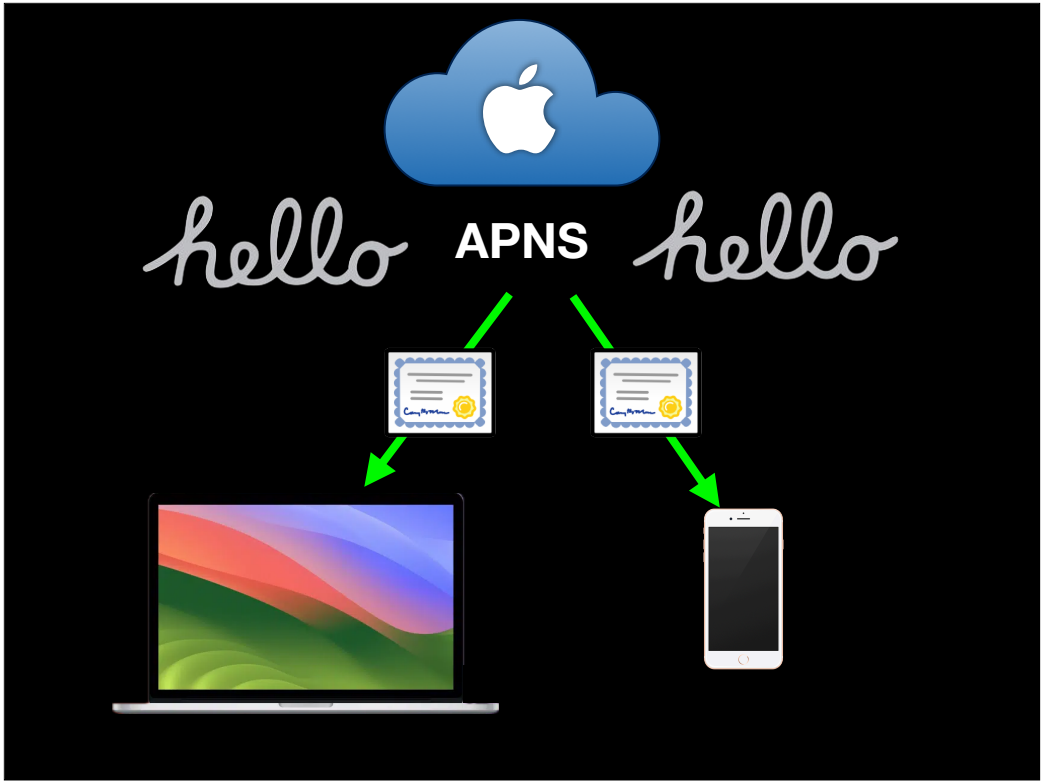


What the Apple device does know is the address of the initialization server it's supposed to talk to and that it's supposed to download a bag file.

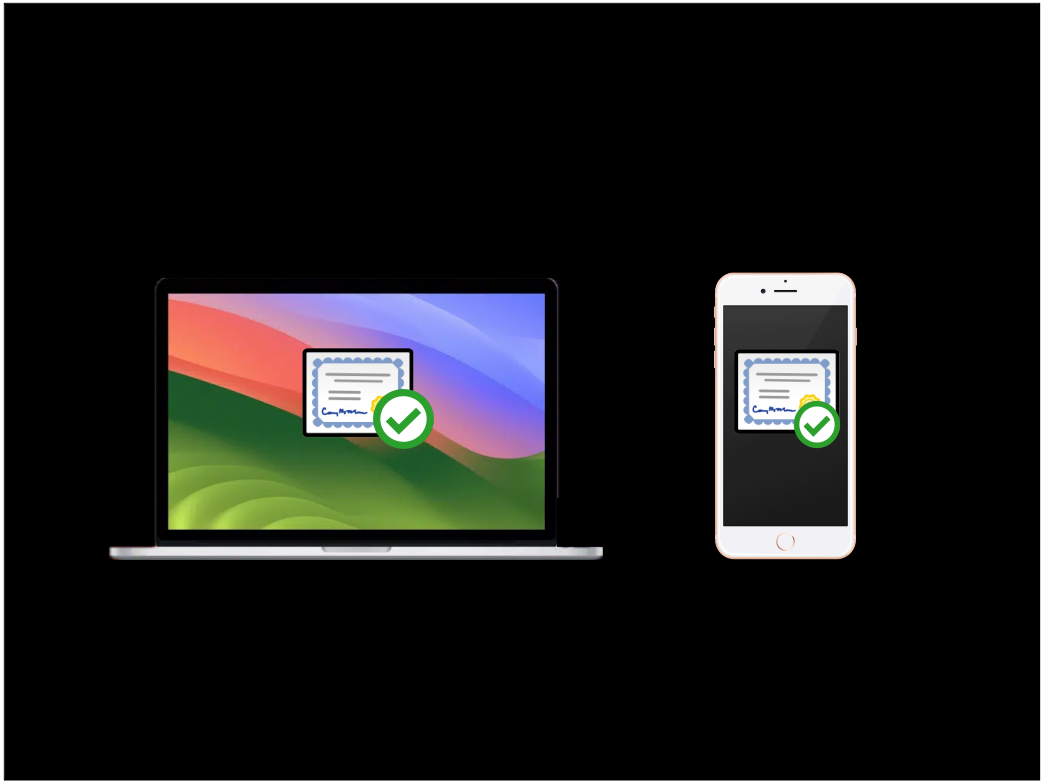
The bag file is a plist which contains the necessary data for the Apple device to start talking to APNS.



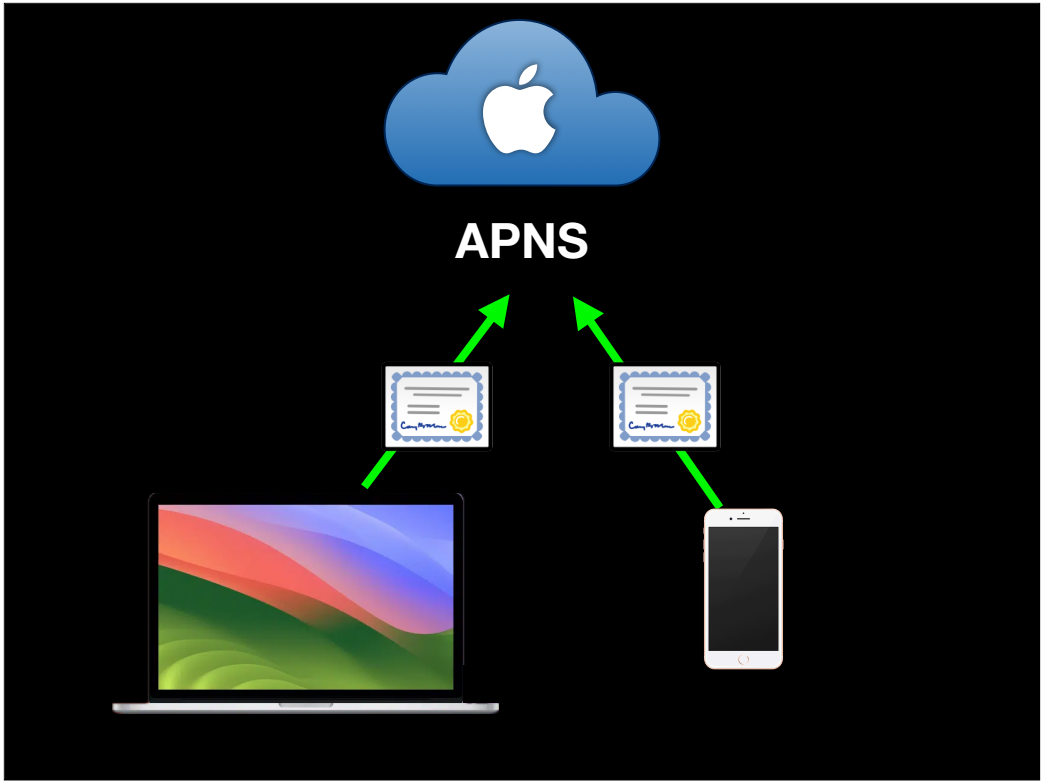
Now that the Apple device knows how and where to start talking to APNS, it opens a connection and says hello.



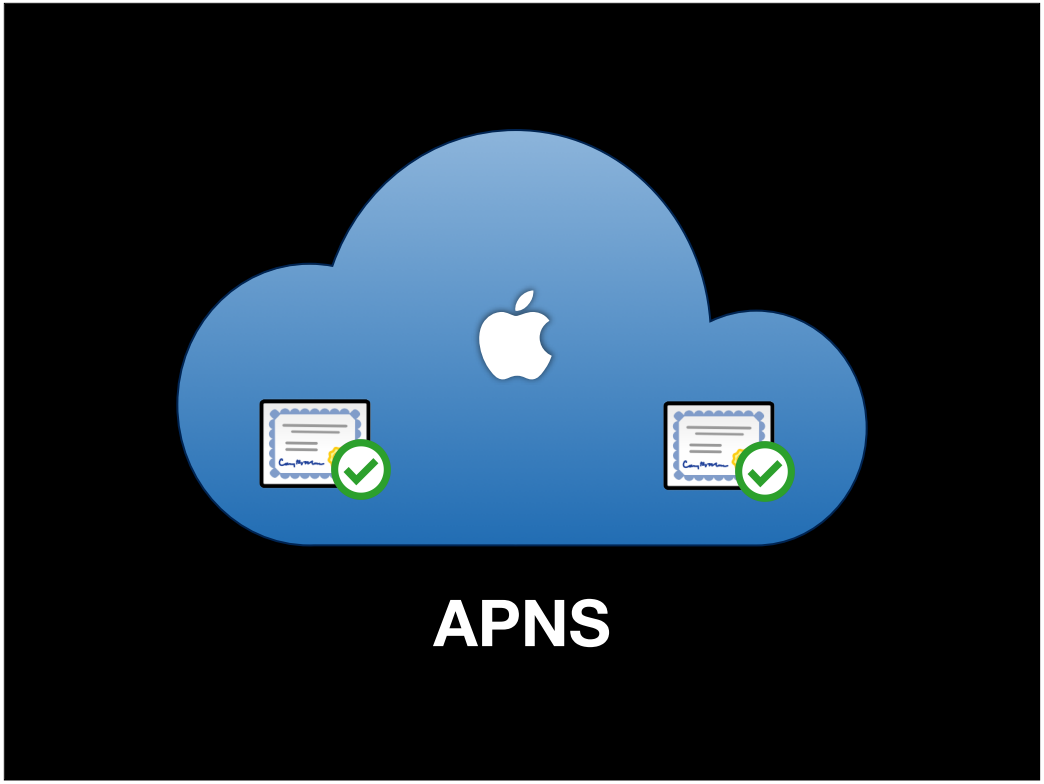
APNS says hello back and provides a certificate.



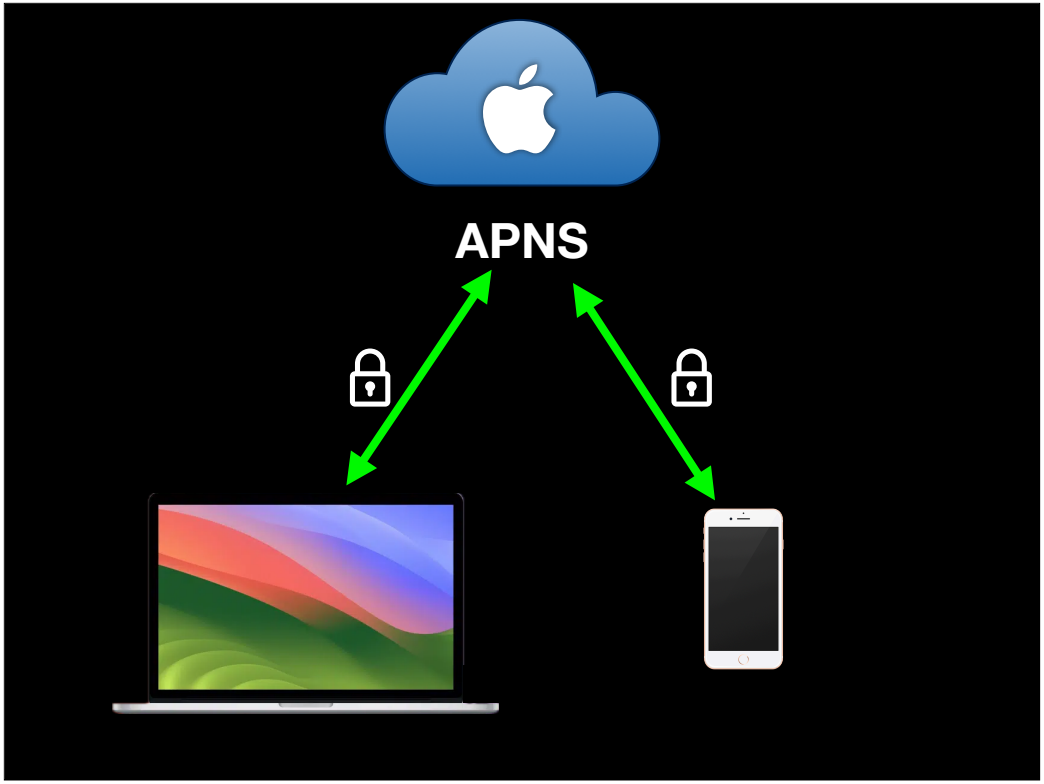
The device checks the certificate and verifies that it’s valid, telling the device that APNS can be trusted to communicate with.



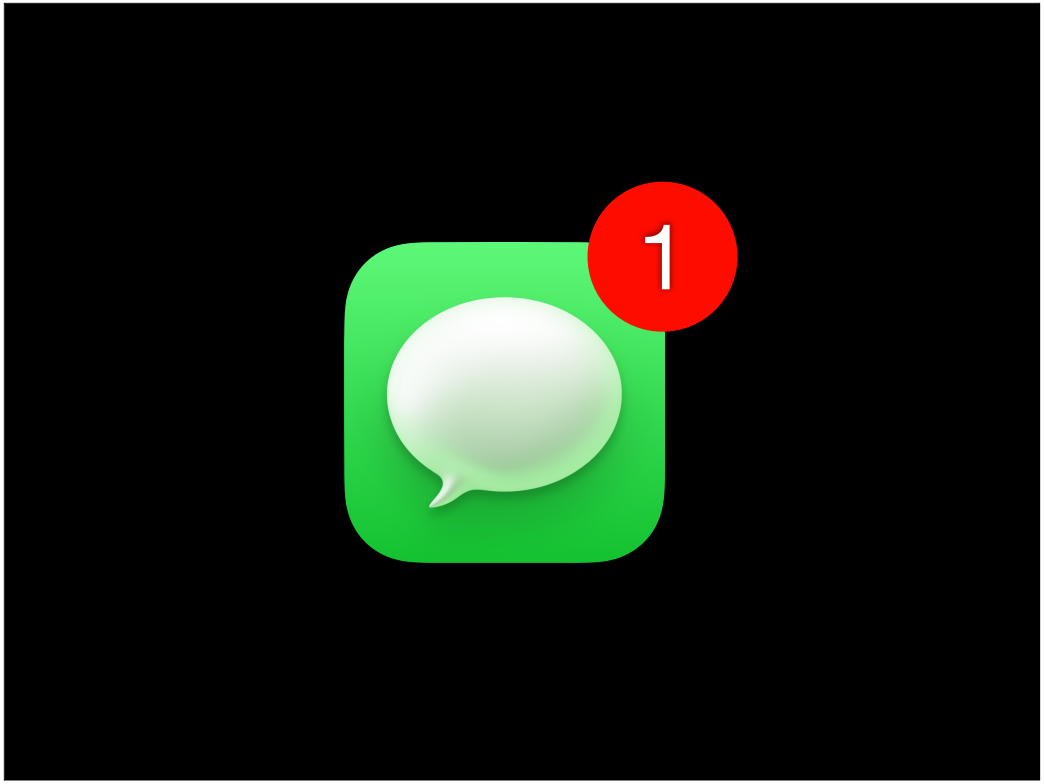
The device has an Apple-issued device certificate, so once the device has verified that the certificate provided by APNS is valid, it sends that device certificate up to APNS.



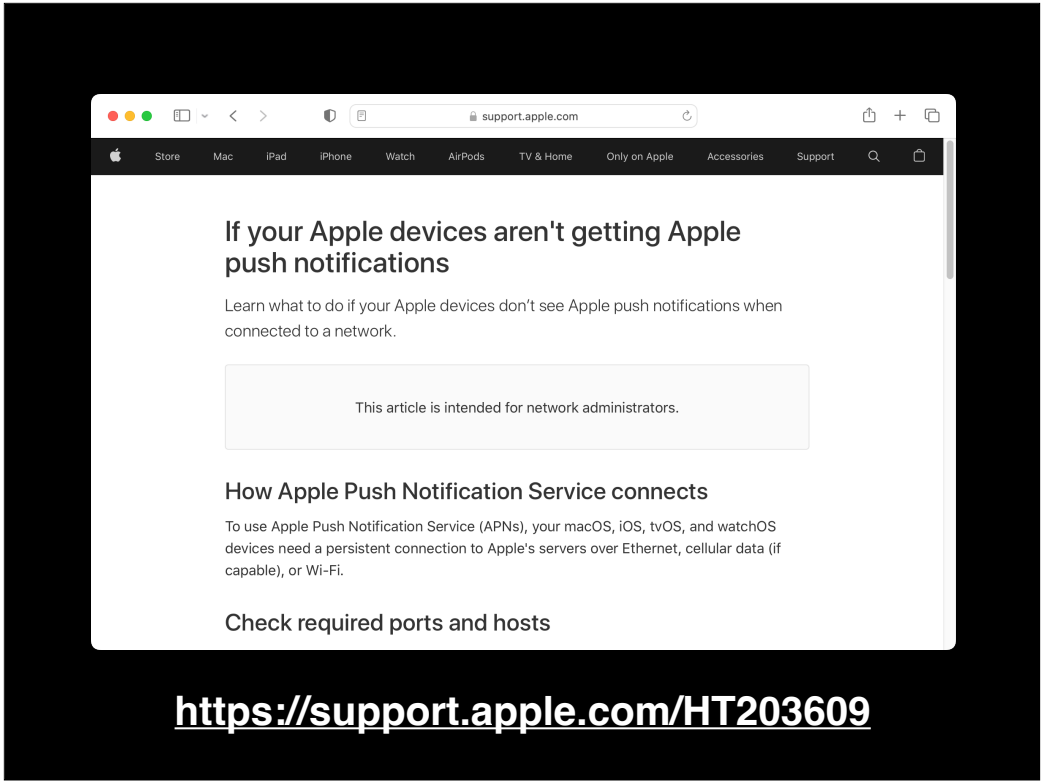
APNS checks the certificate it’s been sent by the device and verifies that it’s valid, telling APNS that the device can be trusted to communicate with.



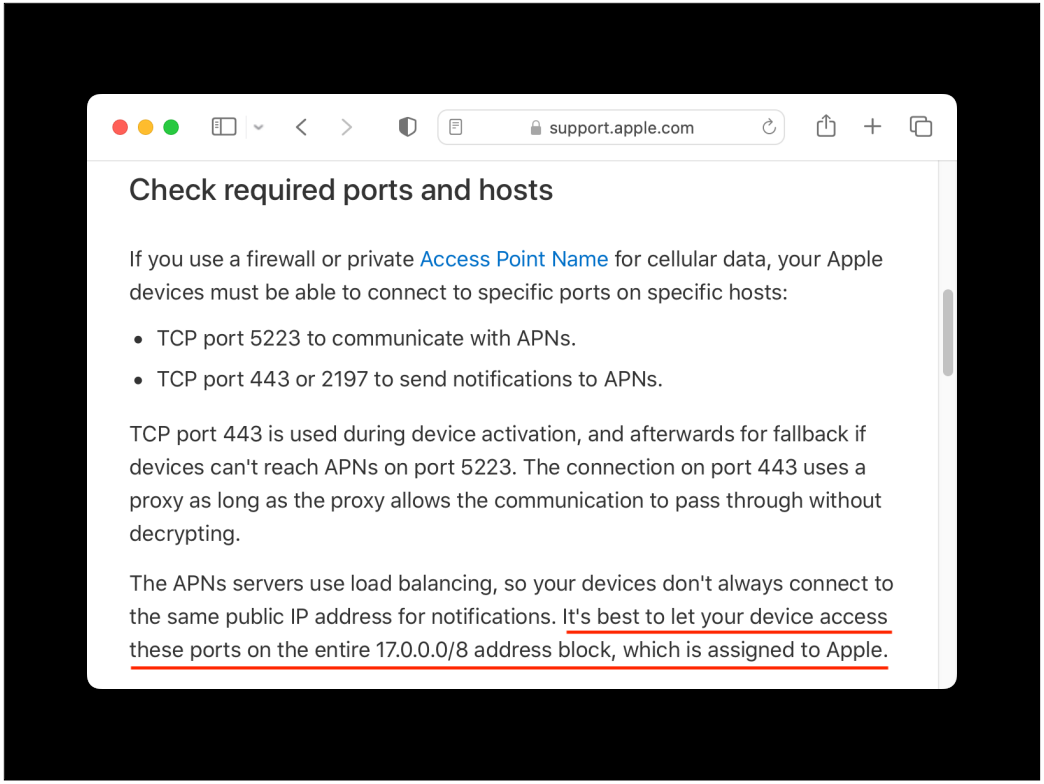
Once both sides know that the opposite end can be trusted to communicate with, they exchange cryptographic ciphers and begin a secure conversation.



APNS will be used by Apple devices regardless of whether they’re enrolled in a mobile device management service or not. For the various apps on your Apple device that use push notifications, APNS will be how those notifications get sent to your device.



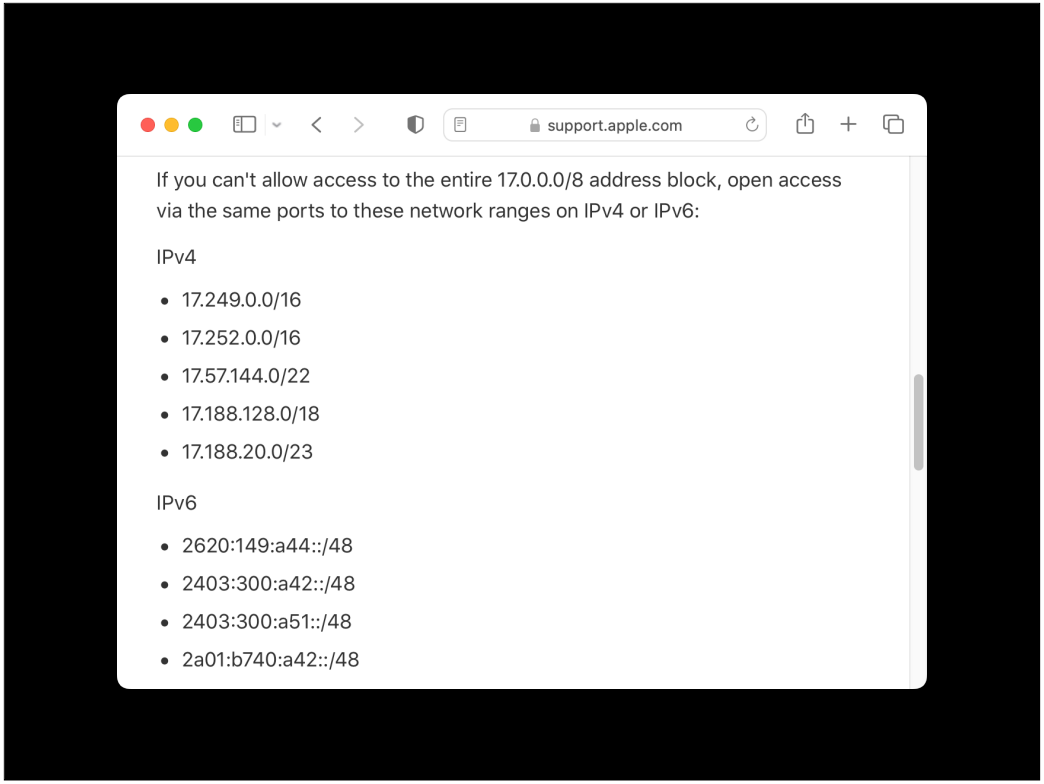
APNS also requires having certain ports opened outbound. Details are available via link on the screen.



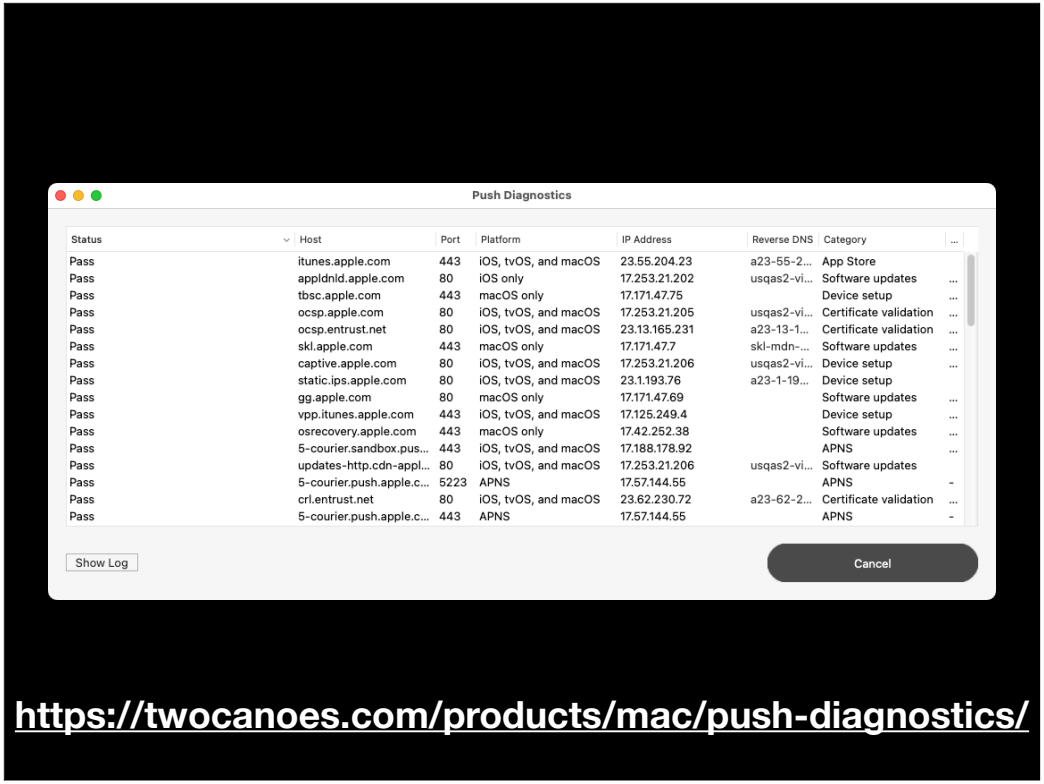
This is the part of the KBase article where your Infosec and firewall folks may start shaking their heads and saying things like “No, no”, and “Can’t do it”.

- **APNS never needs an inbound network connection on your network.**
- **Only *outbound* connections from your network to Apple's network (17.0.0.0 / 8) are needed for APNS.**
- **APNS never makes unsolicited connections.**
- **APNS uses TLS 1.2**
- **APNS authenticates all transactions with device tokens and payload tokens, and validates all SSL certificates.**

For those folks having that argument with their security or network folks, here are some important facts to know.

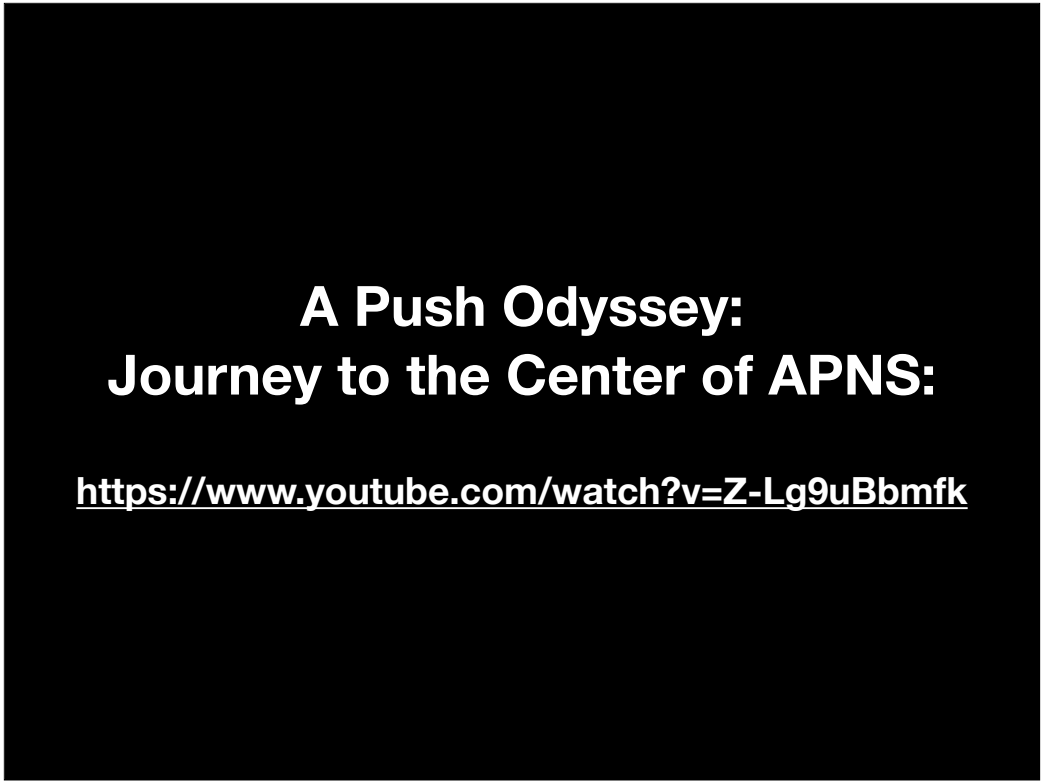


In the event that your network or security folks absolutely won’t do it, Apple does provide a more restricted set of IPv4 and IPv6 ranges to use.



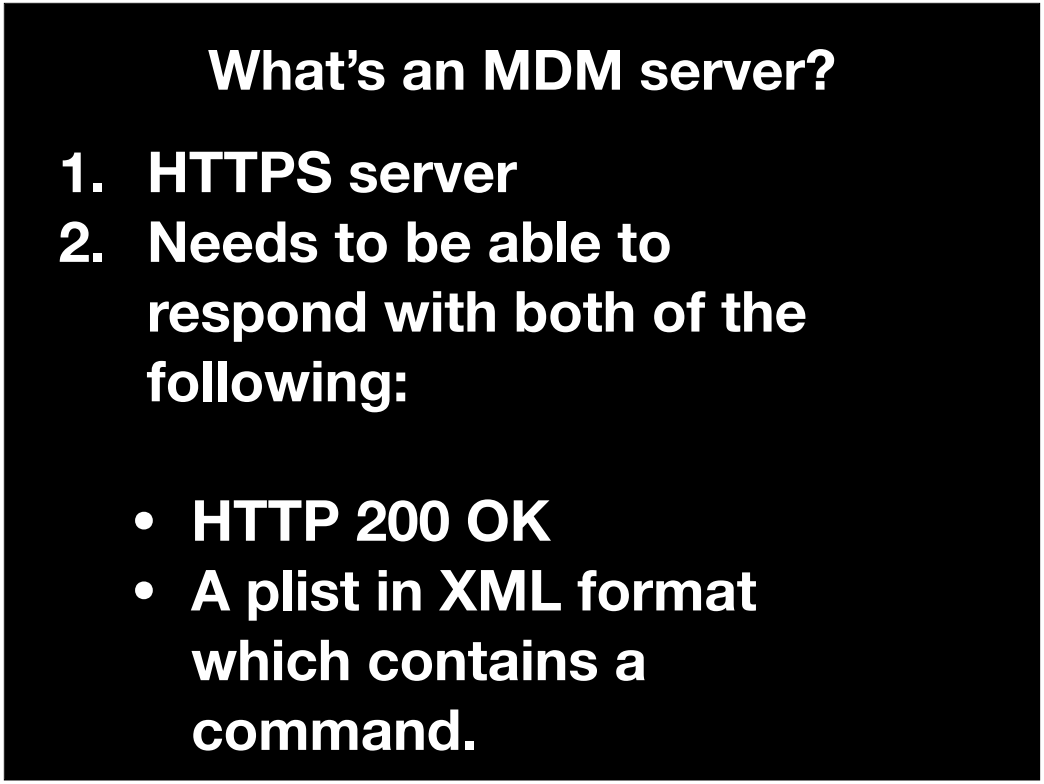
https://twocanoes.com/products/mac/push-diagnostics/

Once the argument is over and they've agreed to open up the needed ports, you may find that they didn't actually open up all the necessary ports. To check this, Two Canoes software makes a great diagnostic tool.

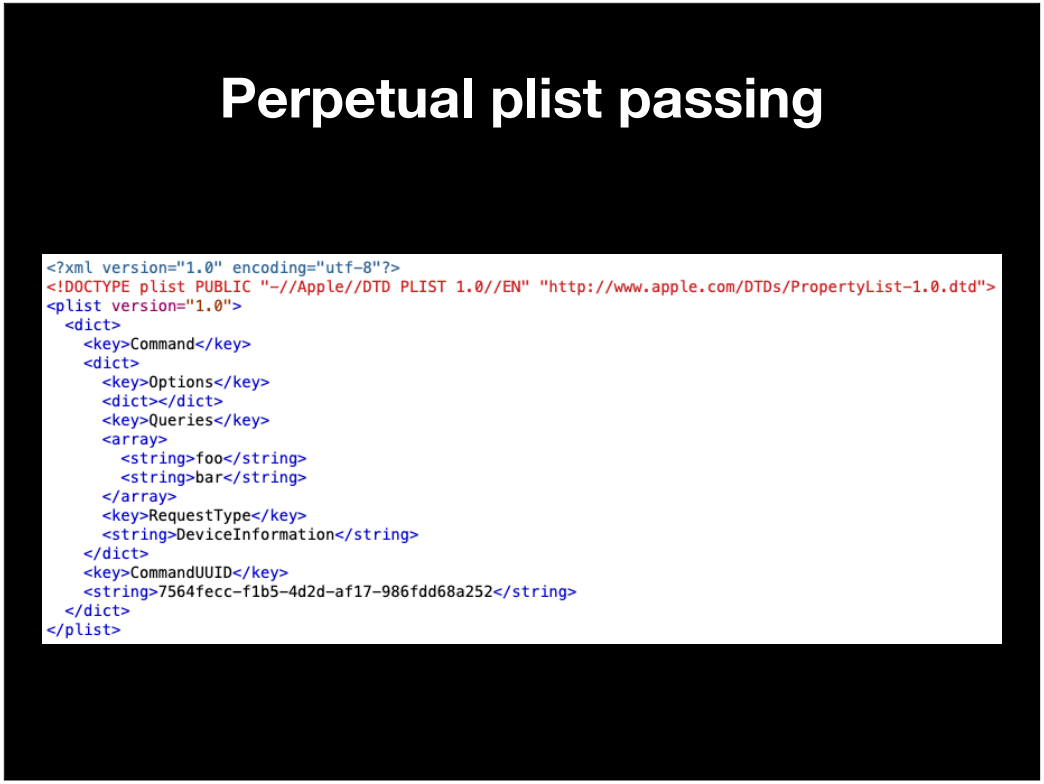


For more information about APNS and how it works, I encourage you to check out Brad Chapman's talk on the subject. It was given at the Jamf Nation User Conference in 2017 and is available via the link on the screen.

Now let's turn our attention to the other half of mobile device management, which is the MDM server.



So, what is an MDM server? When you get down to it, it's an HTTP server which responds to communication with either a plist containing a command, or HTTP 200, which is the OK response.



Most traffic between a device and the MDM server it's enrolled with are going to be plists being sent back and forth. Depending on the contents of the plist, some of what's being passed will be digitally signed or encrypted. For example, this is what a request from the MDM server for a device inventory may look like.

Perpetual plist passing

```
<?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>Command</key>
        <dict>
            <key>Payload</key>
            <data>Zm9vYmFyYmF6</data>
            <key>RequestType</key>
            <string>InstallProfile</string>
        </dict>
        <key>CommandUUID</key>
        <string>7e761ec5-9e20-42d1-9072-3d5a611e3ac5</string>
    </dict>
</plist>
```

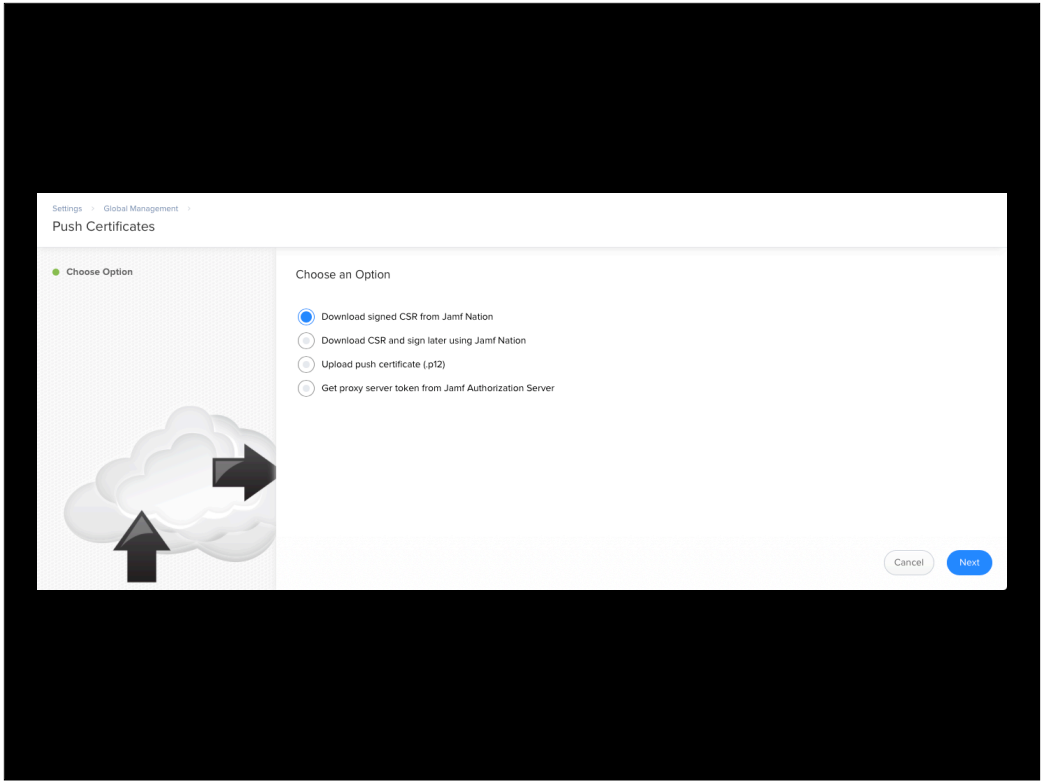
This is what a command from the MDM to install a certain management profile may look like.

MDM Server Certificates

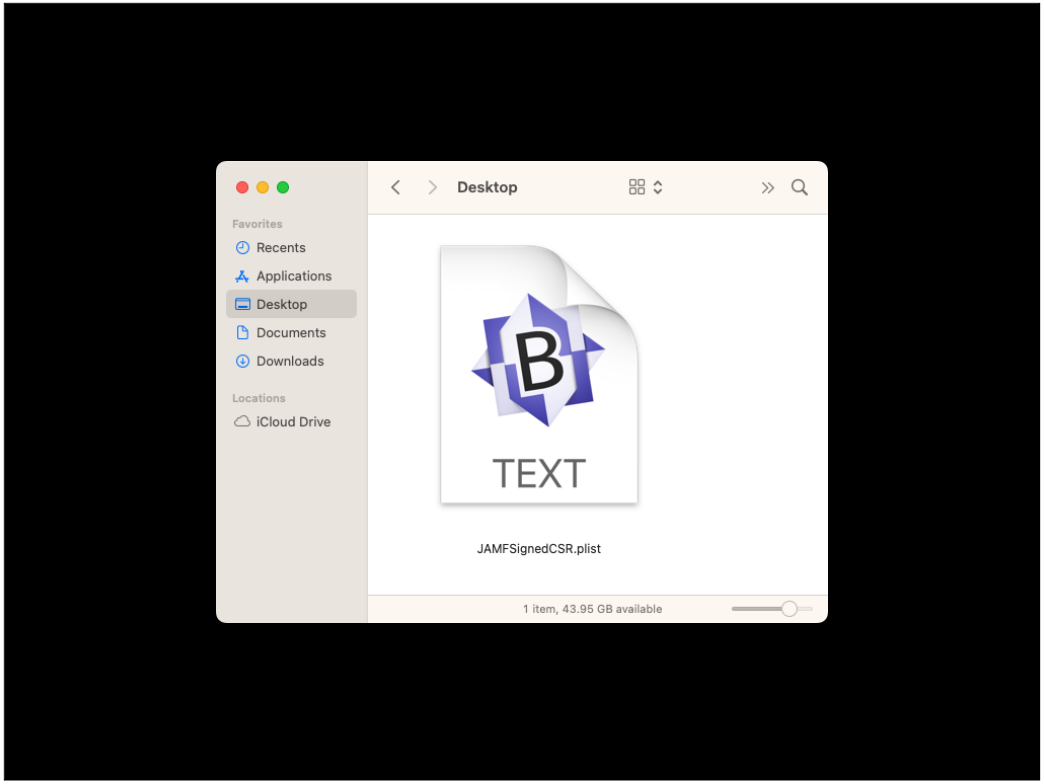
- **APNS Vendor Certificate**
- **APNS Push Certificate**

An MDM server also needs certificates. There are two types of certificates specifically used by MDM servers, the APNS Vendor certificate and the APNS Push certificate

The first is the APNS vendor certificate. This is a special certificate which is used to generate the certificate signing request for an APNS push certificate, prior to uploading the request to Apple for an APNS certificate.

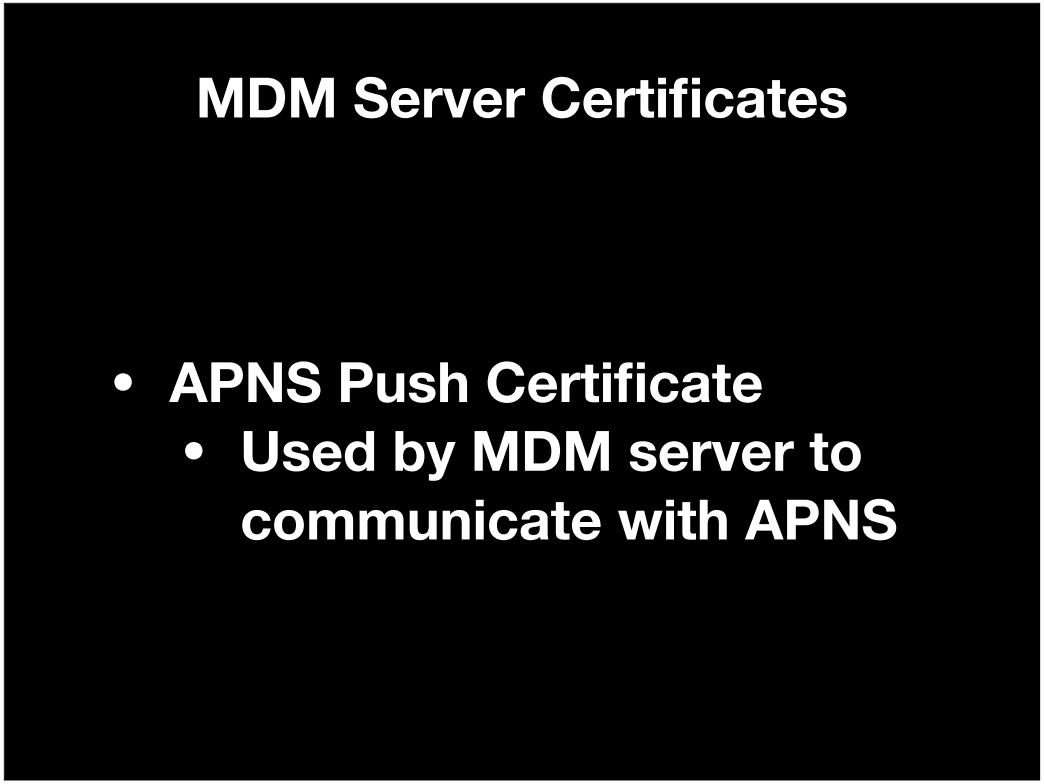


For folks familiar with Jamf Pro, you may have seen this as part of setting up or renewing your APNS push certificate. As part of the APNS setup or renewal, you can select the option of downloading a signed CSR from Jamf Nation.



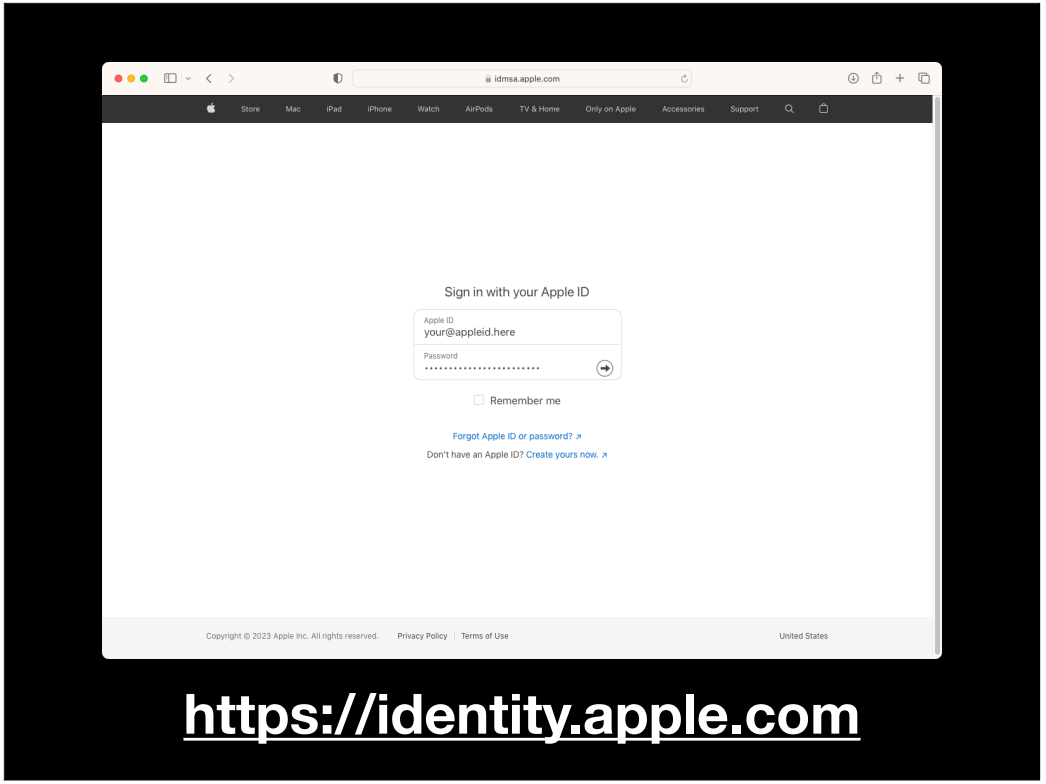
Once you've provided the necessary credentials, a signed certificate request for APNS is then downloaded to your Mac. What's happening in the background is that your Jamf Pro server is communicating with a backend system with access to the APNS vendor certificate to generate that certificate request for your Jamf Pro server.

What is generated is an encrypted plist, ready for uploading to Apple’s push certificates portal.

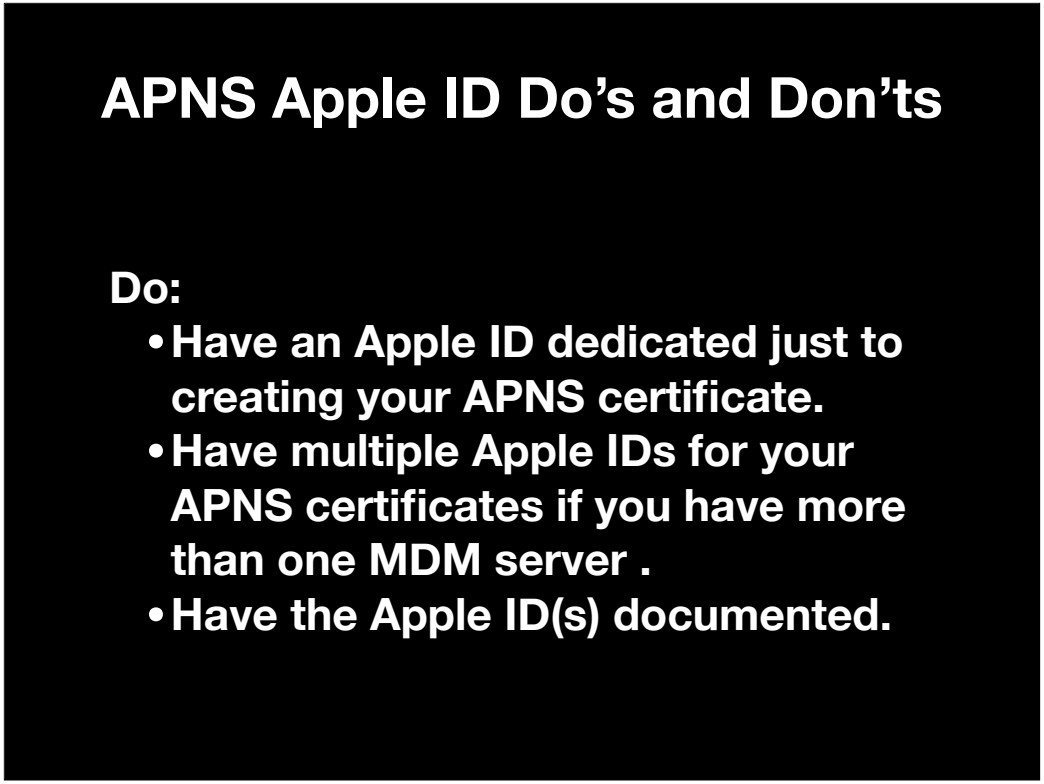


That brings us to the APNS push certificate. This is a special certificate used by MDM servers to communicate with APNS.

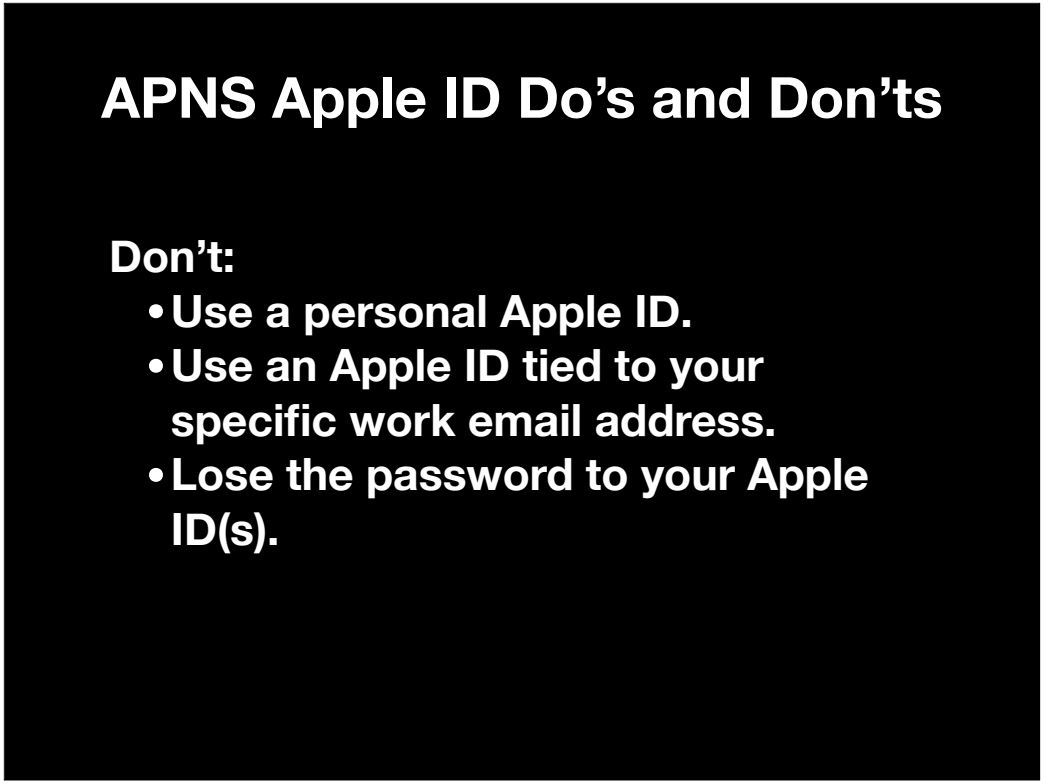
If you've ever set up or renewed an APNS push certificate, you've used the site referenced on the screen. It's where you take the certificate request generated by your MDM and use it to get a push notification certificate.



Your ability to generate a push notification certificate is tied to an Apple ID, which you use to log into the portal site.



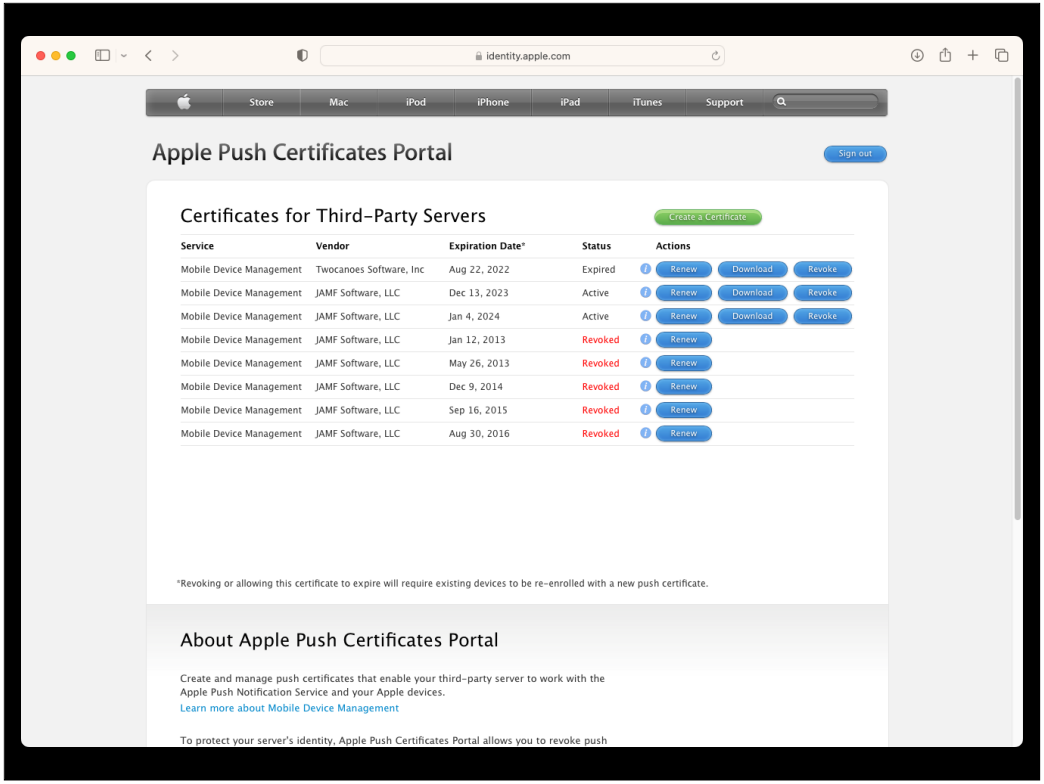
When it comes to the Apple ID used for your APNS certificates, I have some short do’s and don’ts to keep in mind. Where possible, you should try to have one Apple ID per APNS certificate and make sure you have them documented.



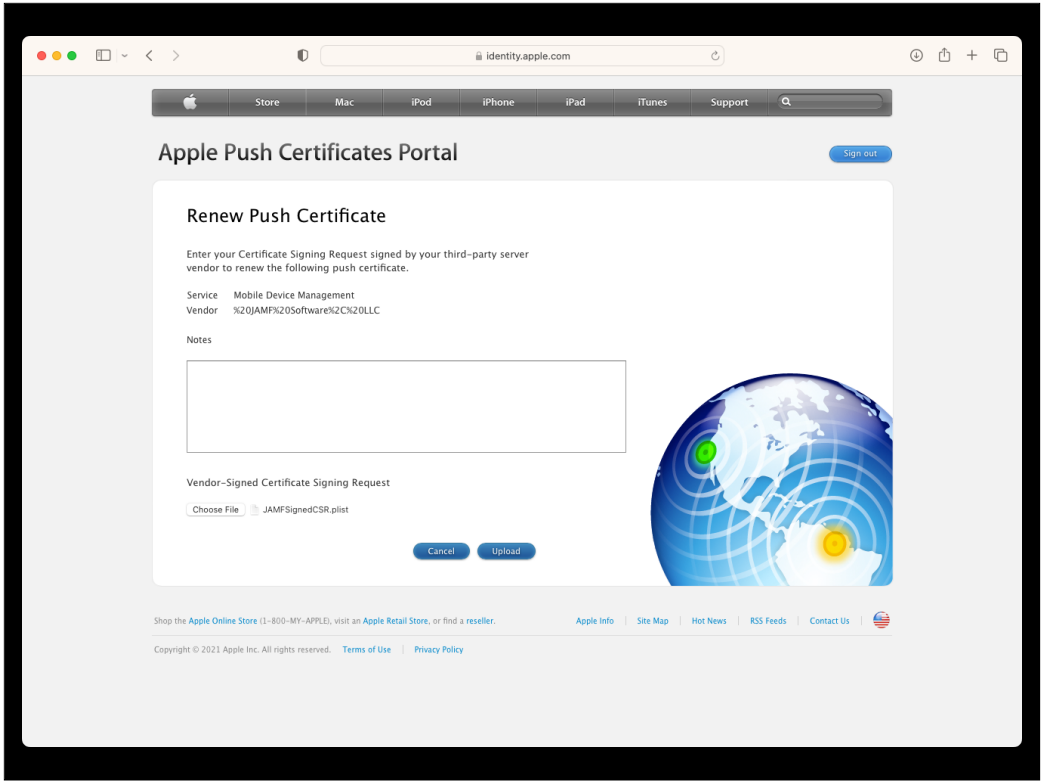
You should also make sure that you’re not using a personal Apple ID or one tied to your work email address. If you can, have a dedicated email mailbox set up just for the Apple IDs involved. Also, really important, do not lose the password for the Apple ID in question.

https://tinyurl.com/fixapns

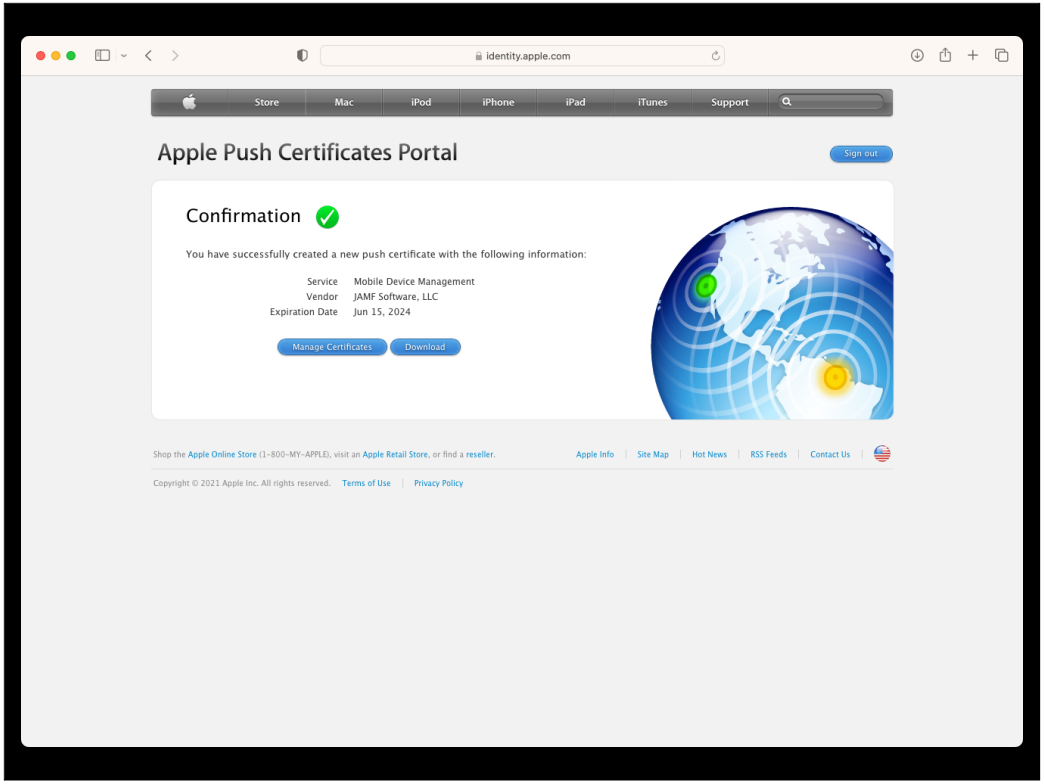
That said, if you do lose the password to the Apple ID in question, Apple does have a process to transfer APNS certificates to another Apple ID. I have a blog post on how to do this available via the link on the screen.



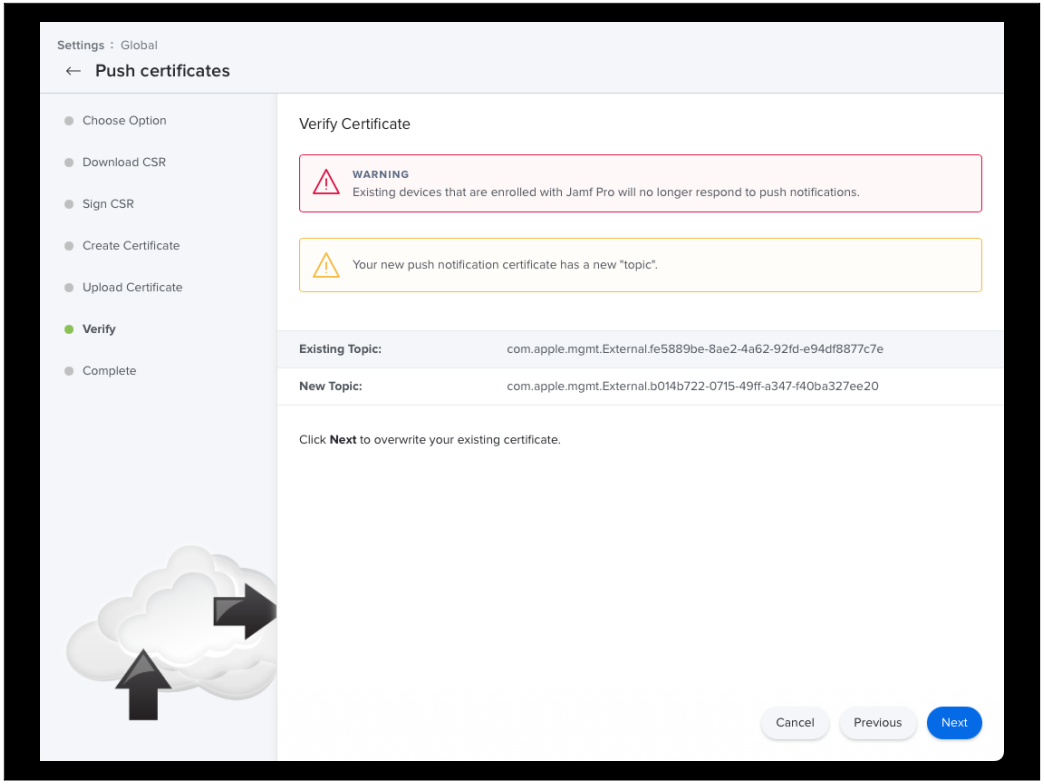
To show what can happen if you don’t follow those rules, here’s what the portal looks like when I log in with my personal Apple ID. I’ve got two active APNS certificates and now I have to figure out which specific APNS certificate goes with my MDM server. This is important, for reasons I’ll discuss in a bit.



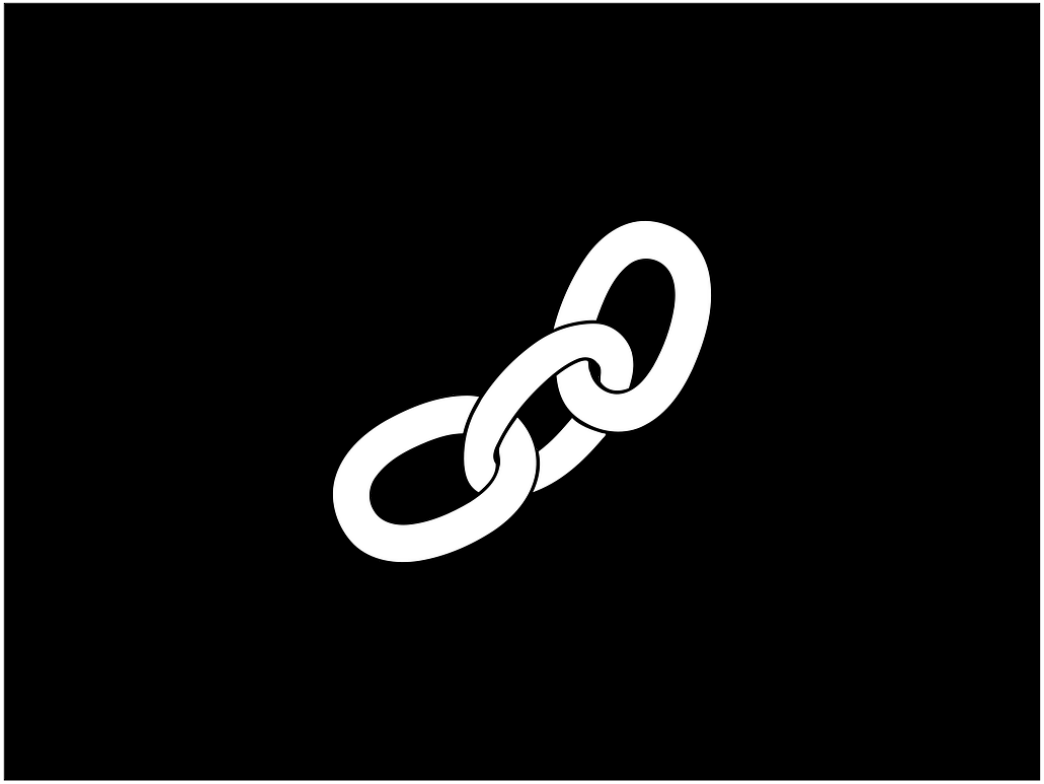
I think I’ve got the right one figured out, so let’s upload that CSR.



Great, I've got a new APNS push certificate. Let's download it and get it over to my MDM server.

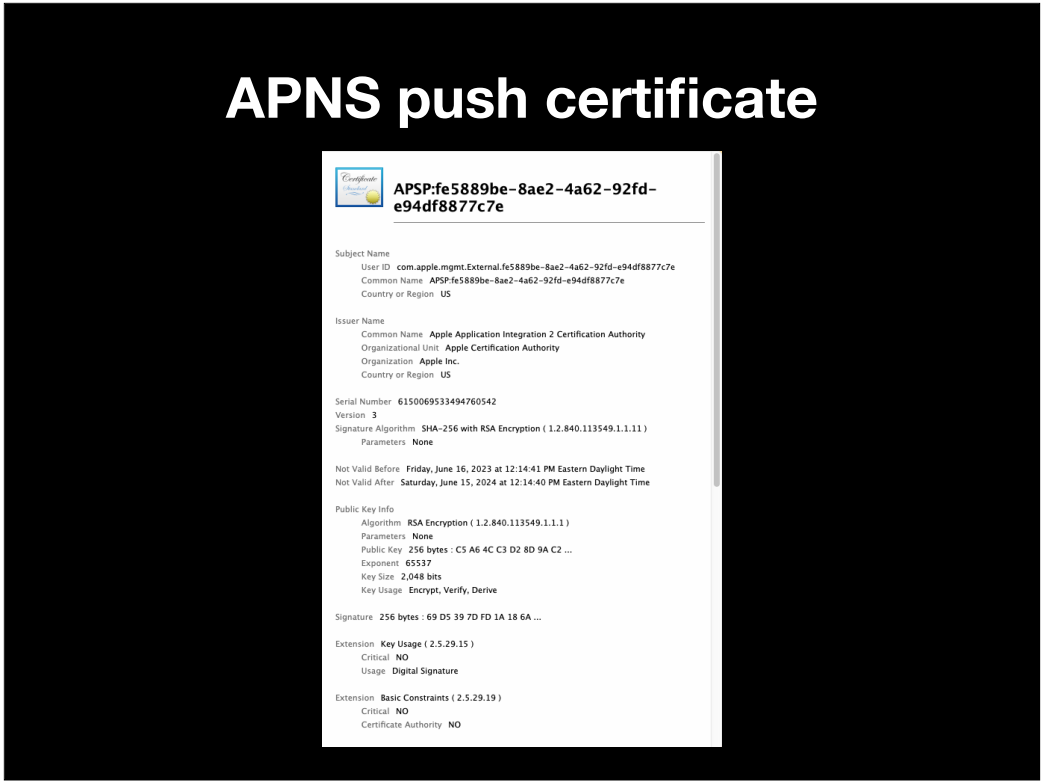


Well, this looks bad. What happened?

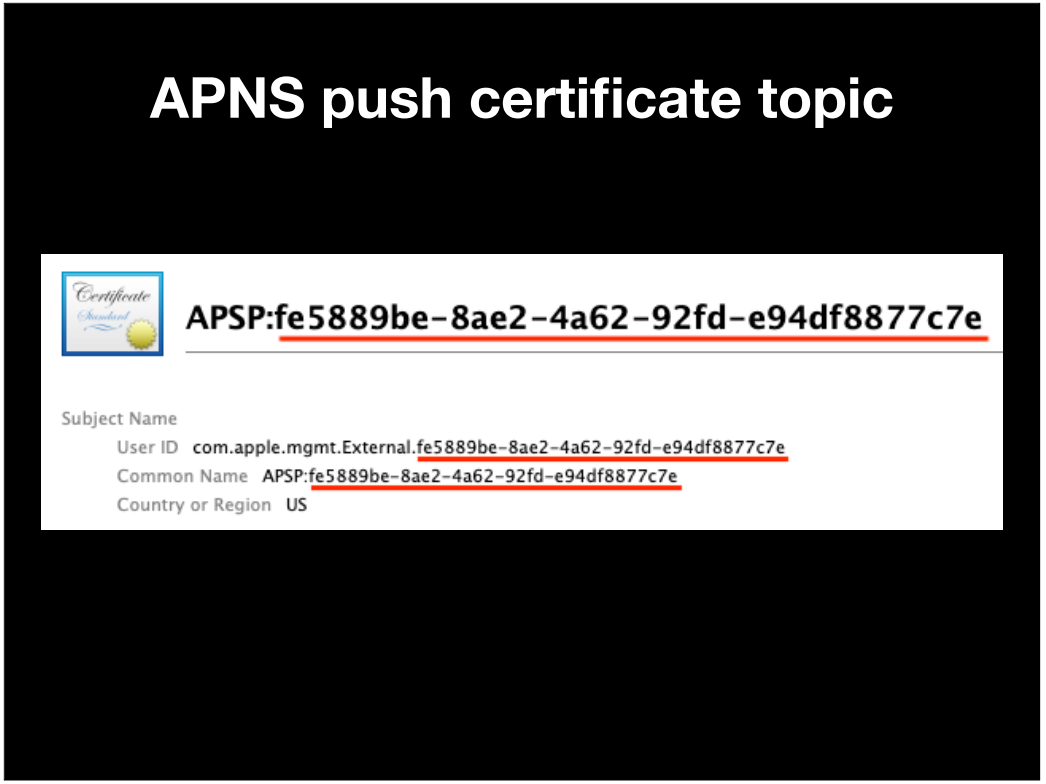


Apple’s push notification service is a chain of trust, with multiple security factors built into that chain.

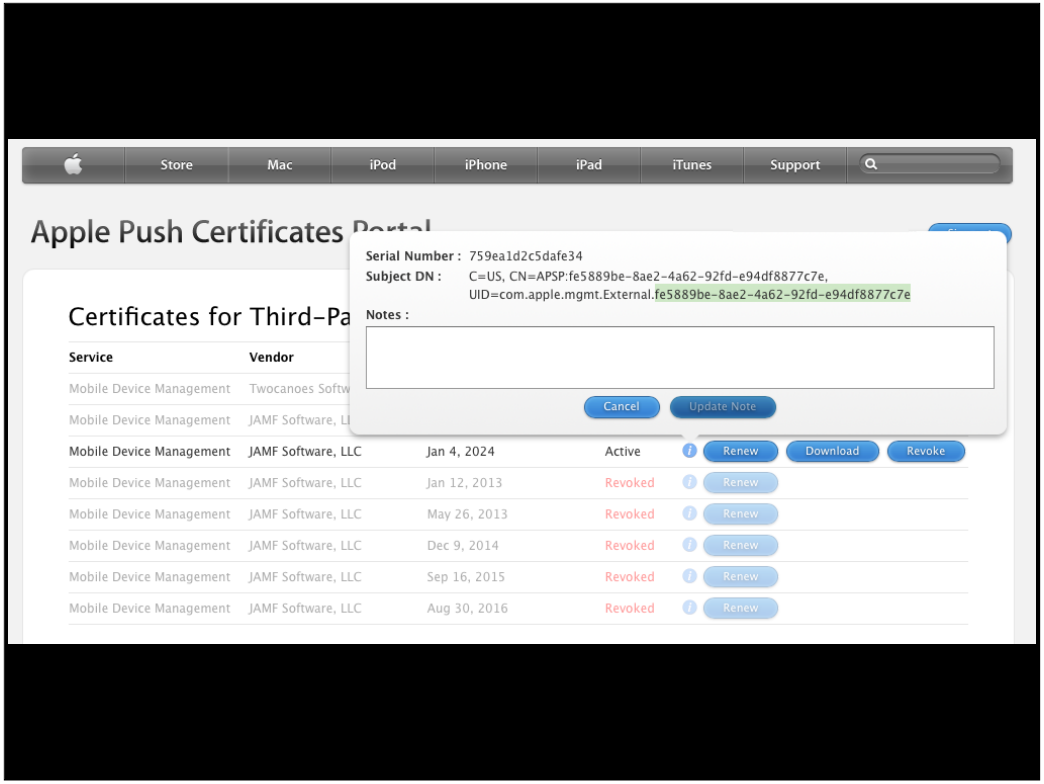
APNS push certificate



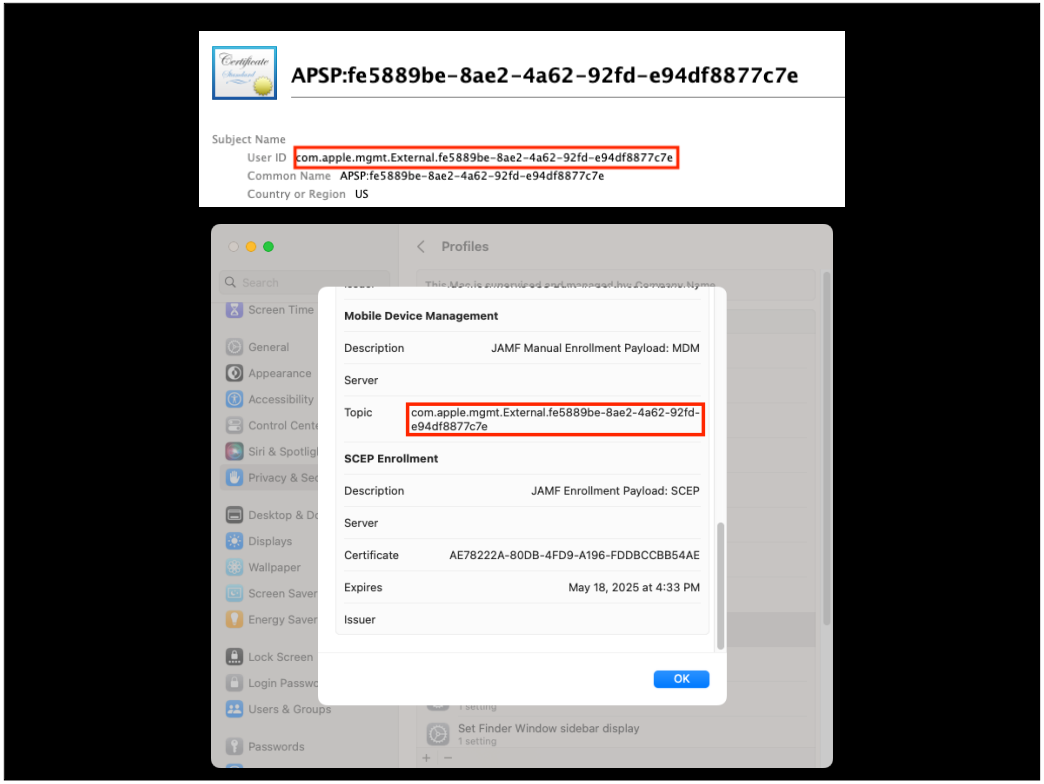
The first part is the certificate that I received from the push certificate portal. When inspected, the certificate looks like this.



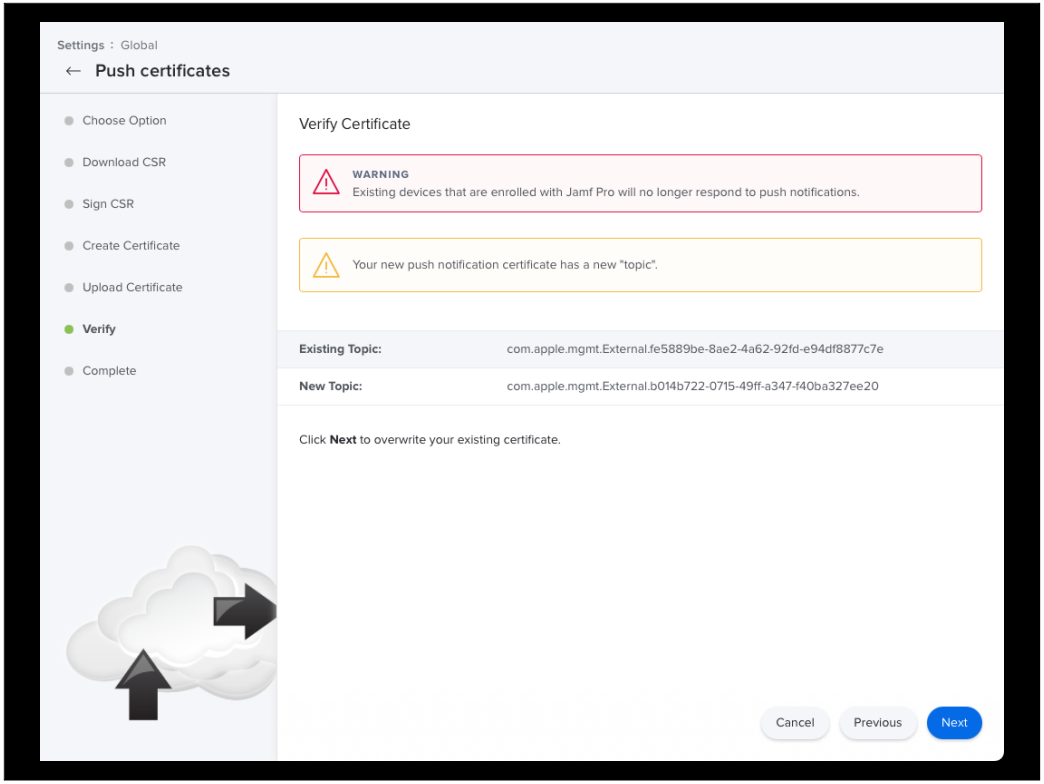
The second part is what's known as the topic, which is a specific UUID identifier associated with your MDM server. The certificate generated by the push notification portal will include the topic in both the Common Name and User ID fields of the certificate.



This information is also visible if you click the info button next to the relevant certificate on the portal page.



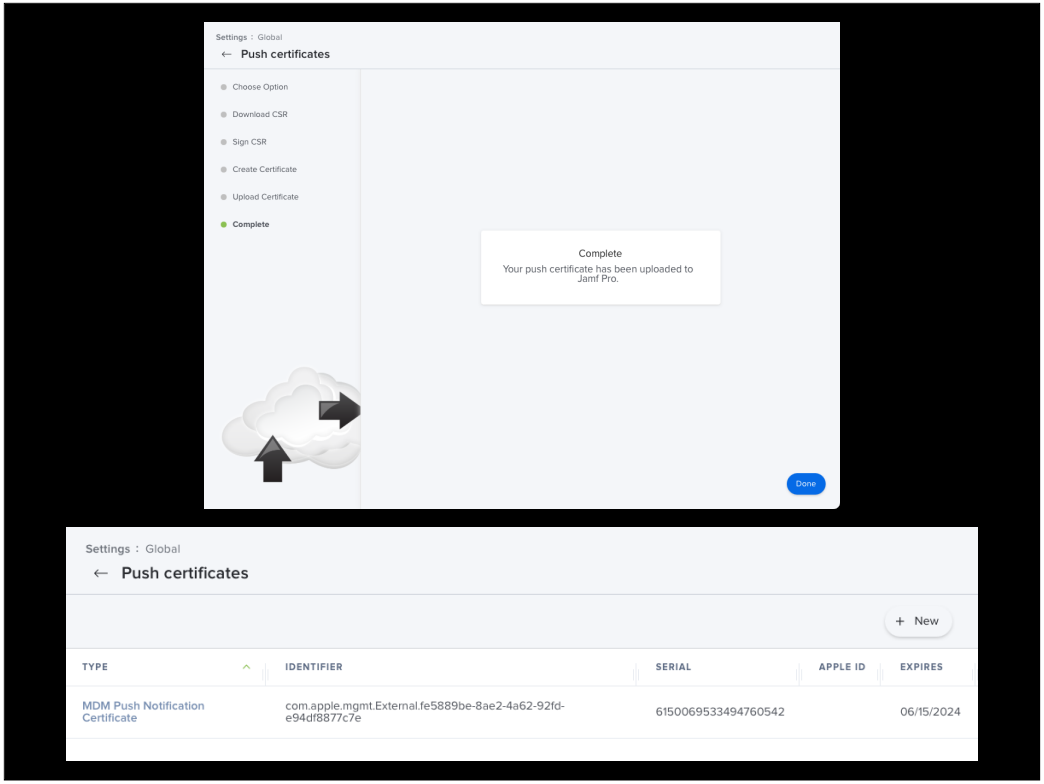
The topic is also embedded in the MDM profile that your MDM server issues to your Apple device, with the Topic field listing the User ID field of the APNS push certificate.



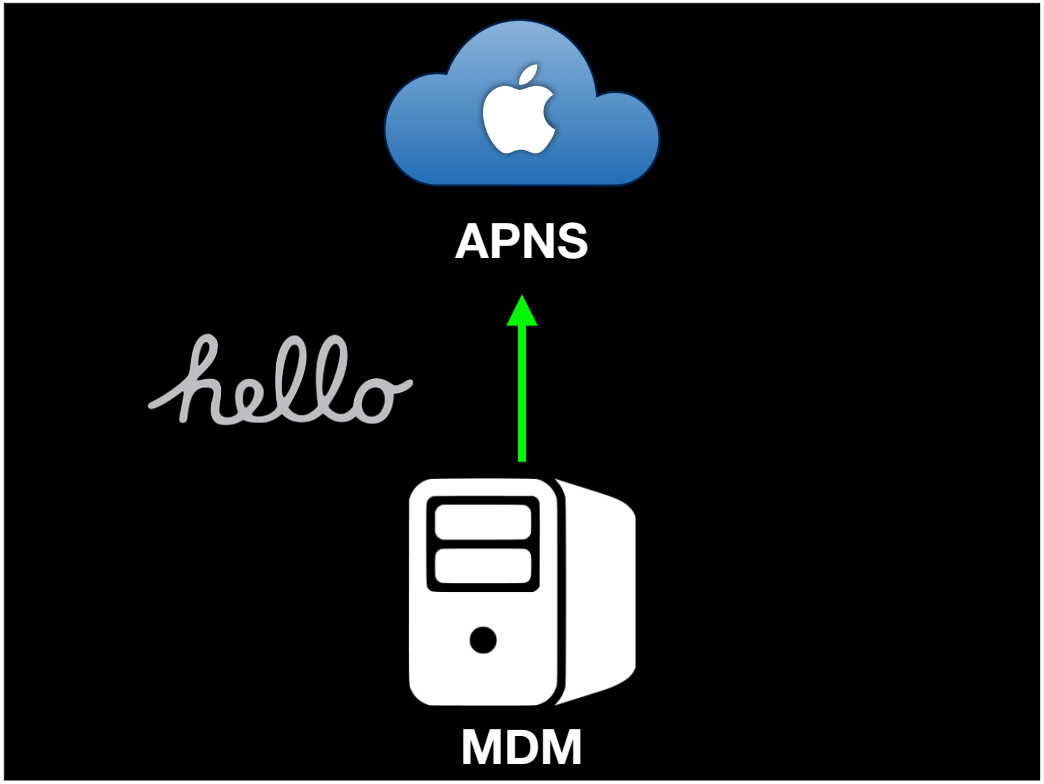
To circle back to my earlier problem with the push notification certificate, I had accidentally created a new push certificate for my MDM instead of renewing my existing one. Because the topic is a UUID, it will be unique per MDM push notification certificate and can only be included in push notification certificates if the certificate is being renewed.

Oops. Fortunately my MDM had warned me before I actually uploaded the new certificate with the different topic.

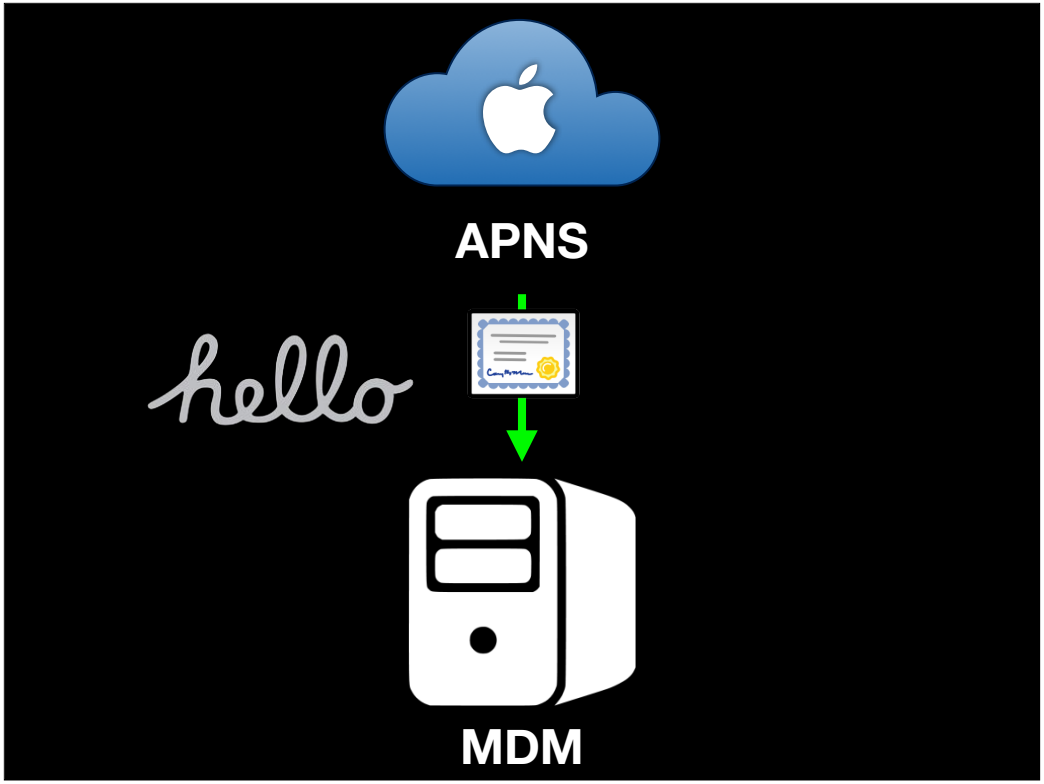
If I had gone ahead and uploaded it, I would have broken the middle of my trusted chain of certificates. All devices enrolled with my MDM would have instantly lost the ability to communicate with the server because APNS would no longer be able to point them to the correct MDM server.



Once I went back, created an new CSR and made sure to renew the correct APNS certificate, everything was fine.

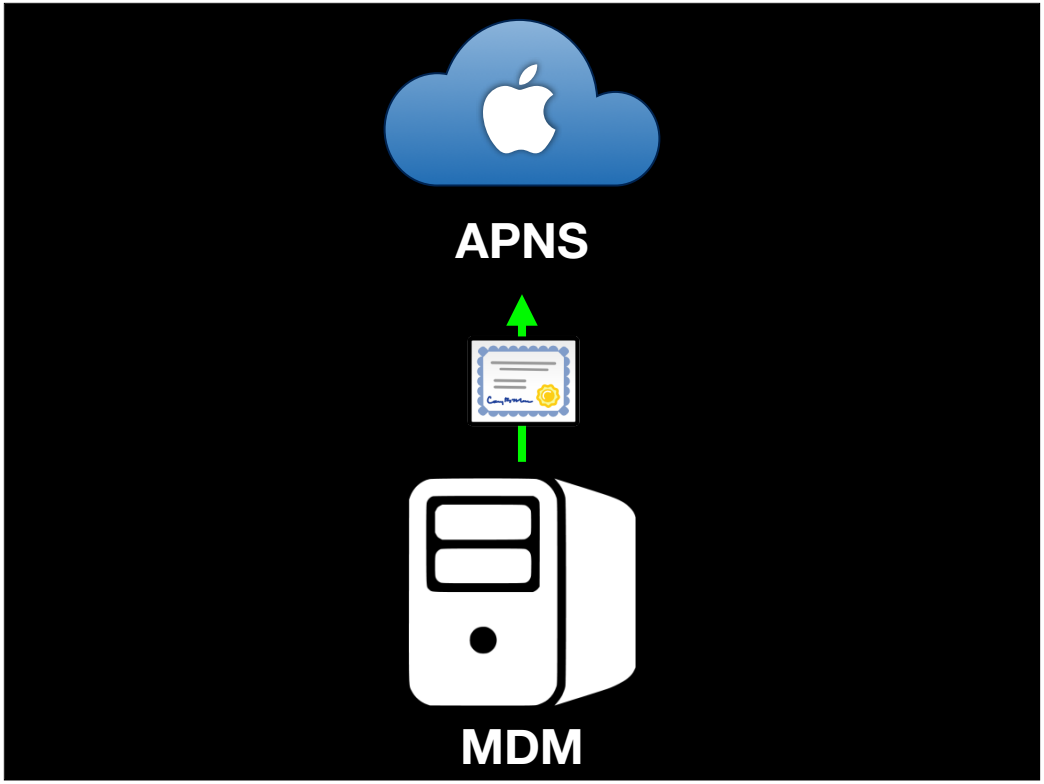


With that knowledge, let’s take a look at how the MDM server talks to APNS. Like an Apple device, it opens a connection and says hello.

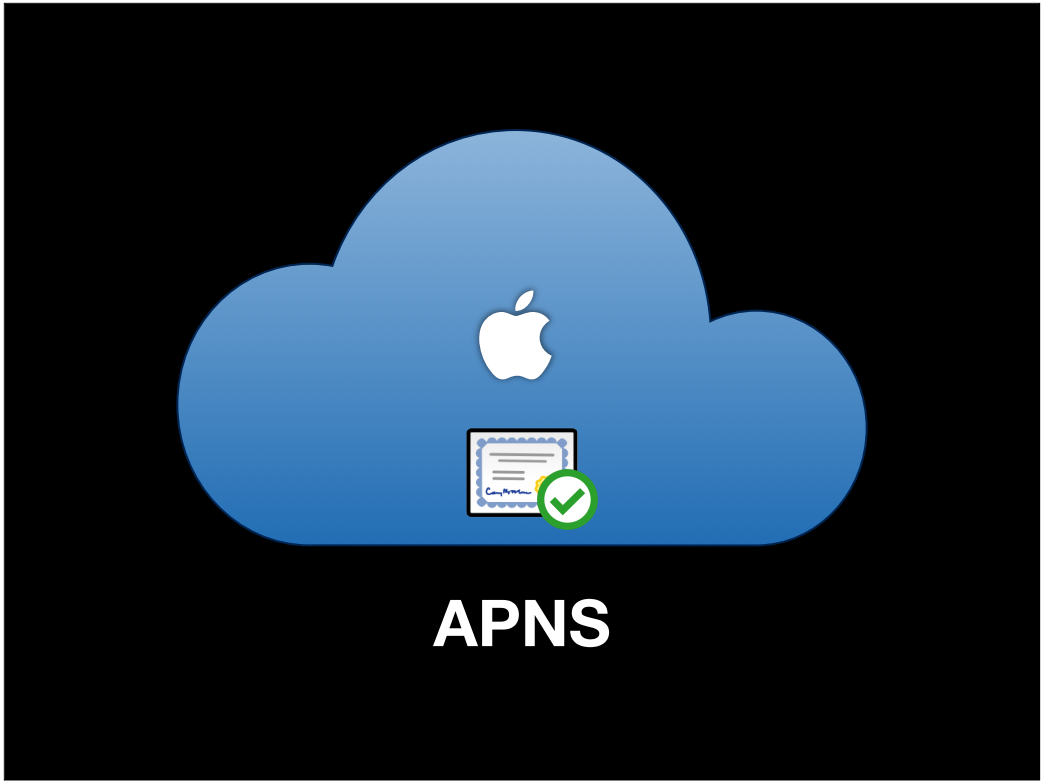


APNS says hello back and responds with a certificate.

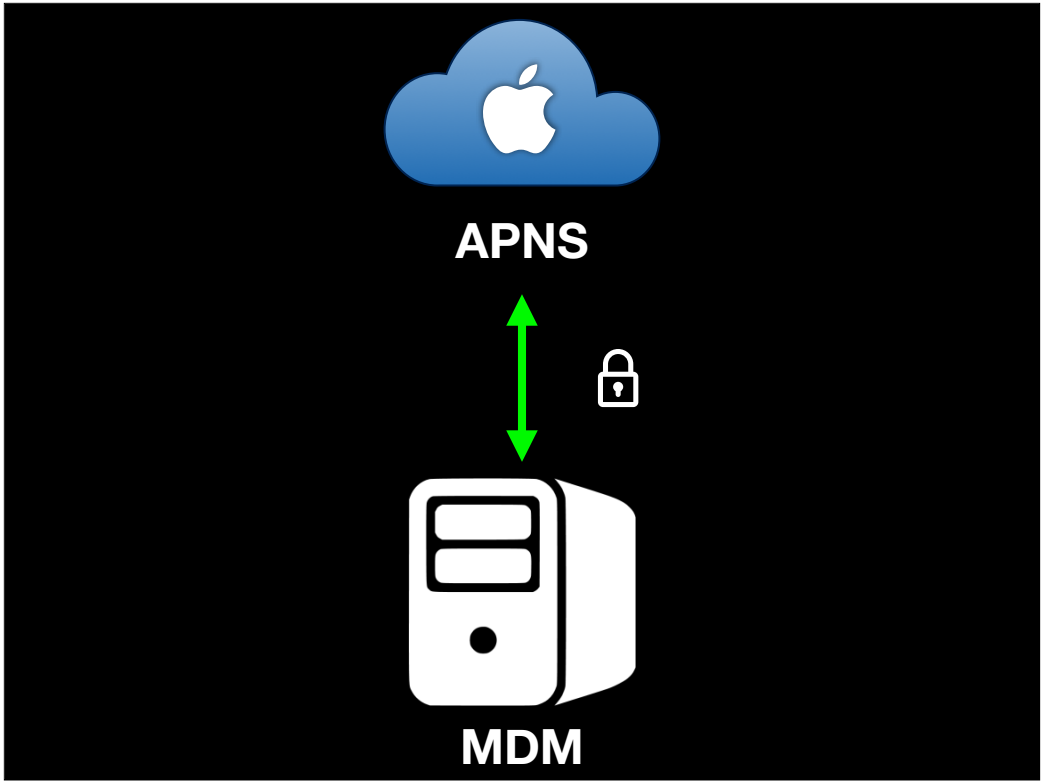
The MDM server checks the certificate and verifies that it’s valid, telling the server that APNS can be trusted to communicate with.



The MDM server now sends the APNS push certificate that it was issued by the Apple push certificate portal.

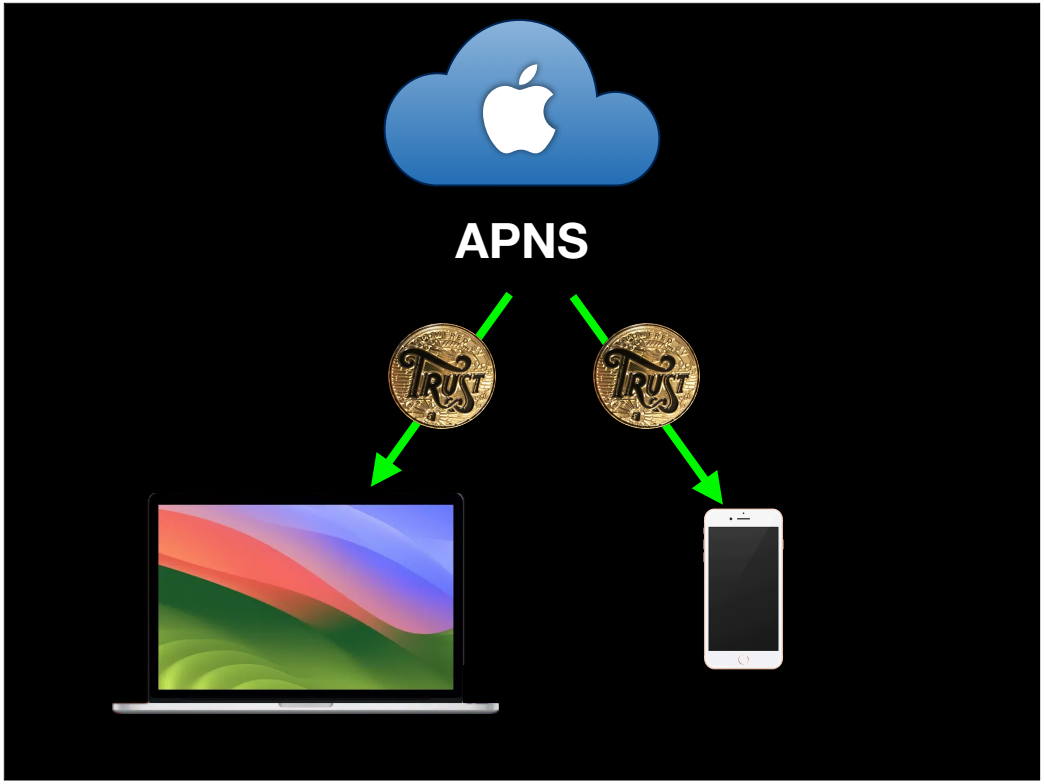


APNS checks the push notification certificate it’s been sent and verifies that it’s valid, telling APNS that the MDM can be trusted to communicate with.

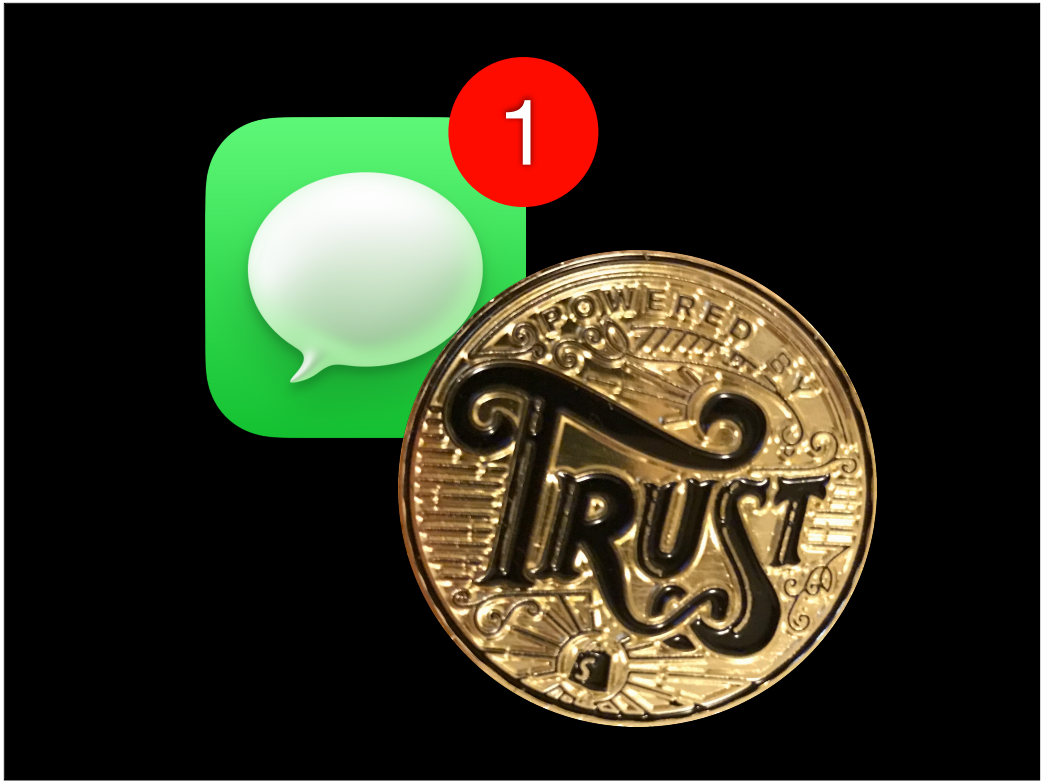


Once both sides know that the opposite end can be trusted, they exchange cryptographic ciphers and begin a secure conversation.

However, the push certificate and the topic are not the end of the chain of certificates. The last link for devices is what’s known as the device token.

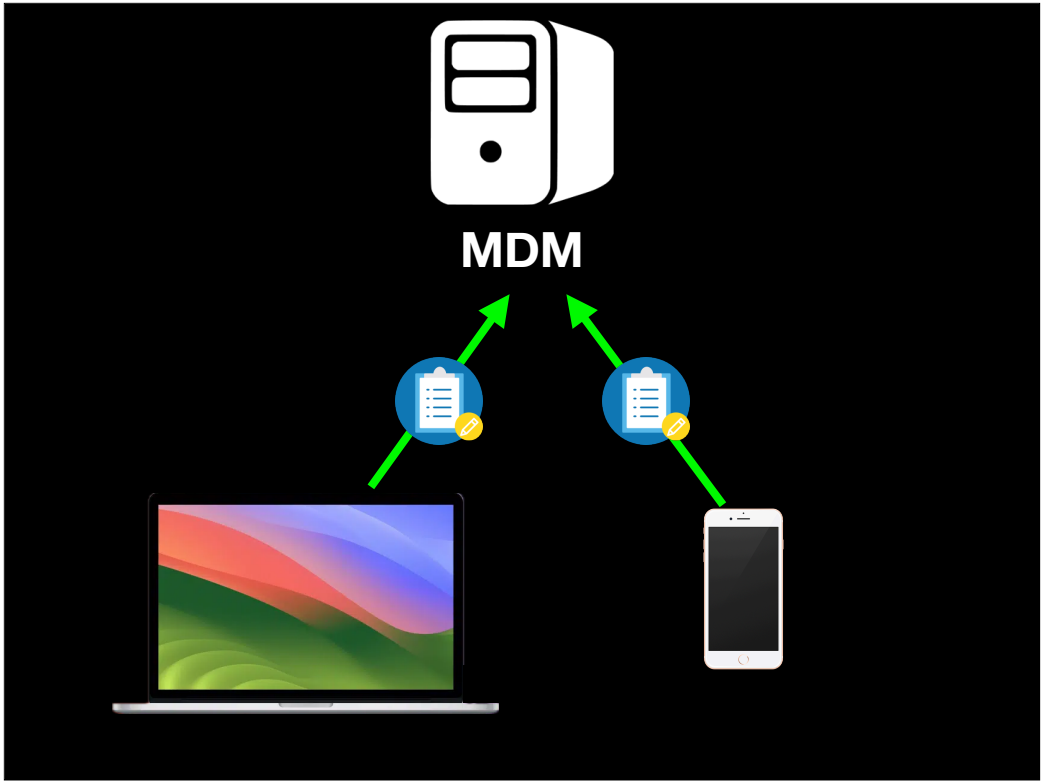


This token is generated by APNS and is unique to the device.

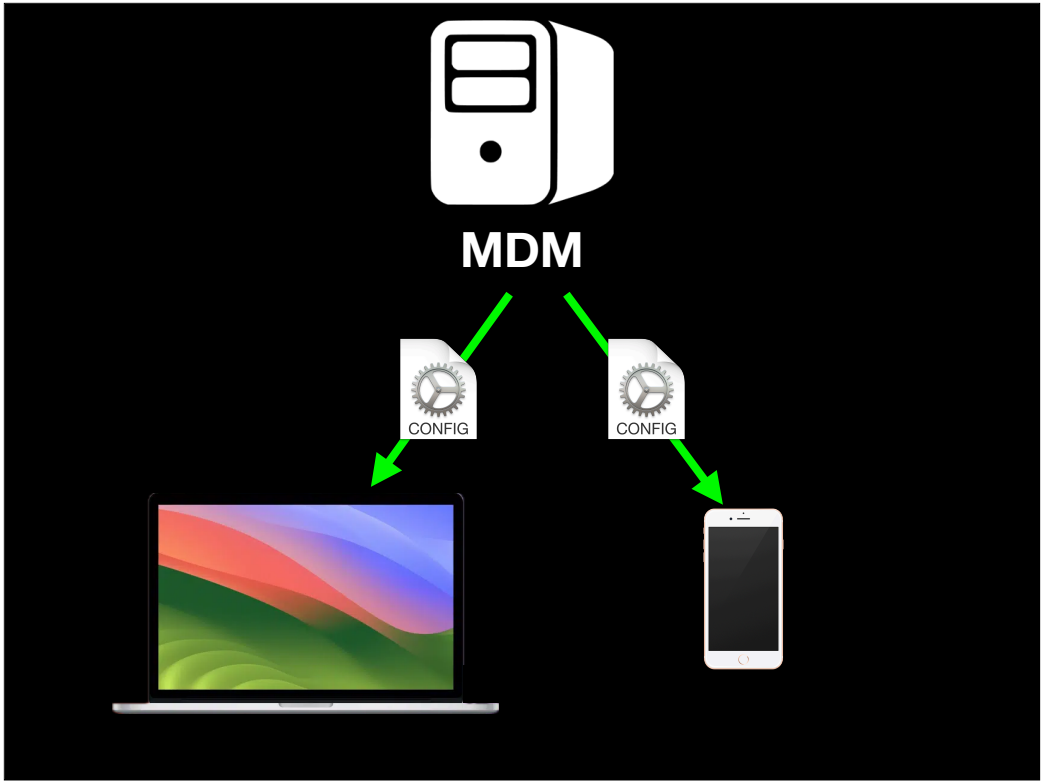


These tokens are the same kind of tokens that APNS assigns to applications which use push notifications.

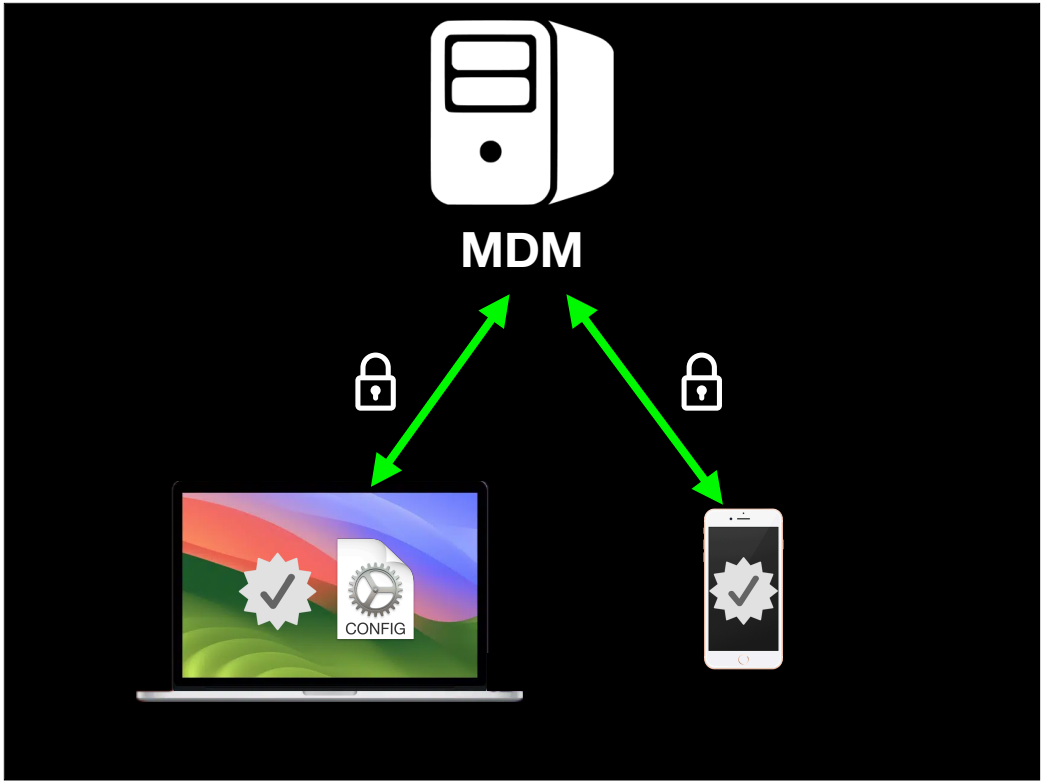
In this case, the “application” in question is your MDM server. But how does the device get the token from APNS and know to forward it to the MDM server? This happens as part of MDM enrollment.



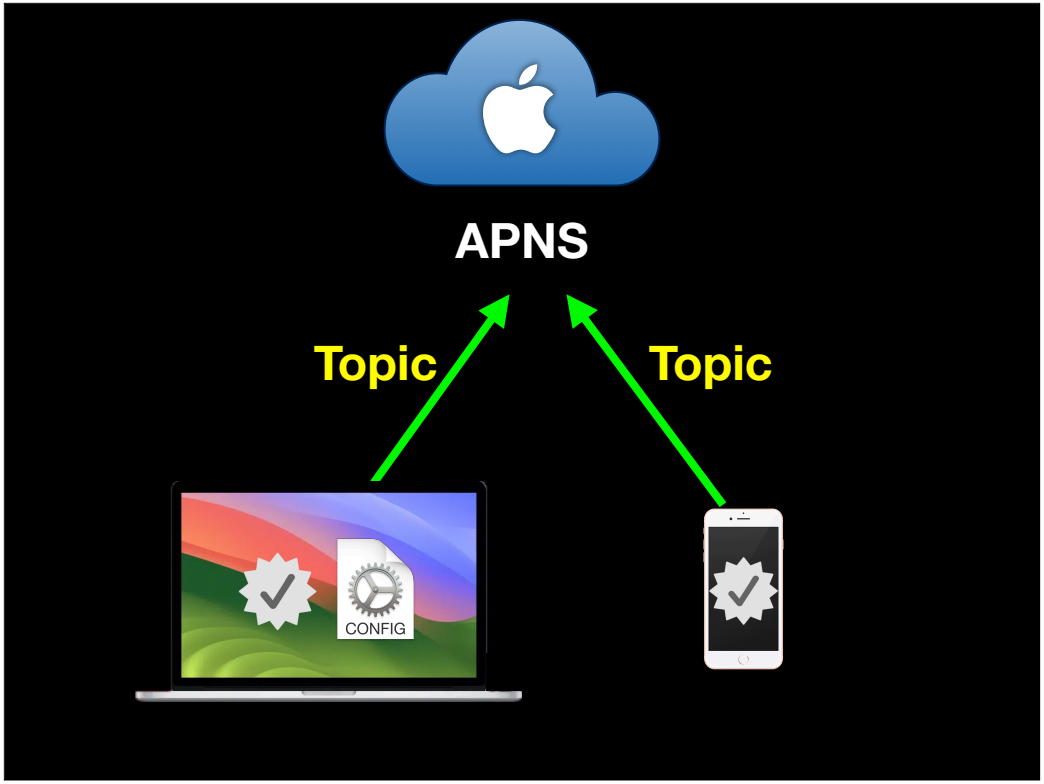
The device connects to the MDM server and sends in an enrollment request.



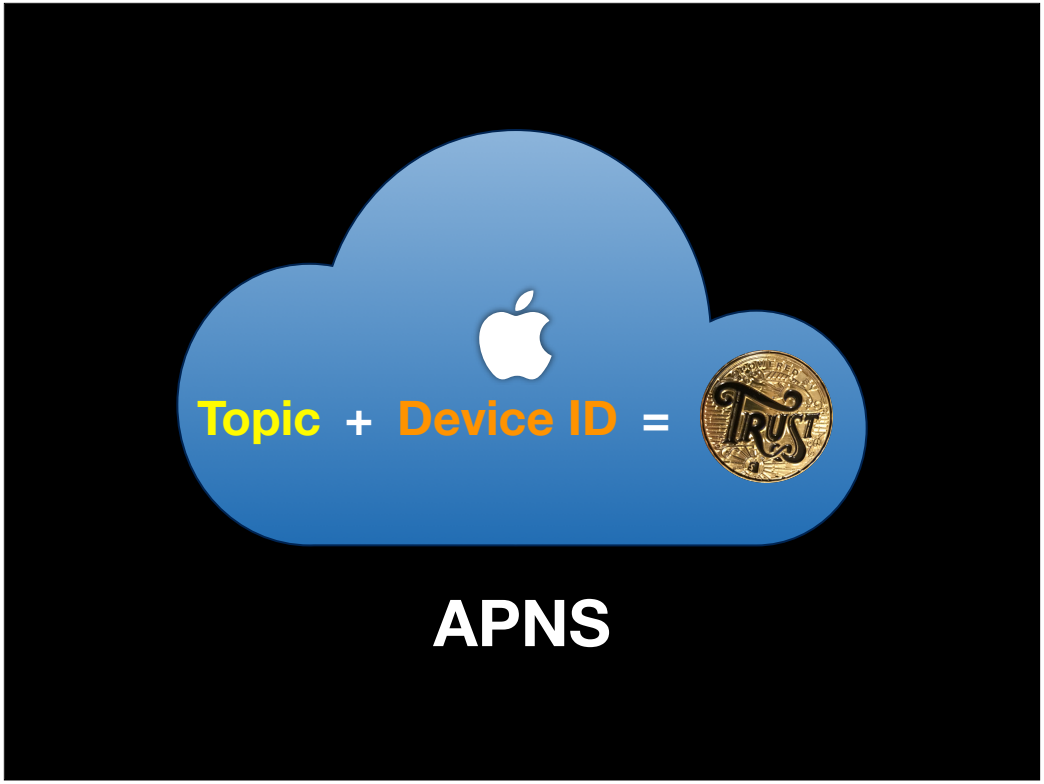
As part of the enrollment process, the MDM configuration profile is downloaded to the device.



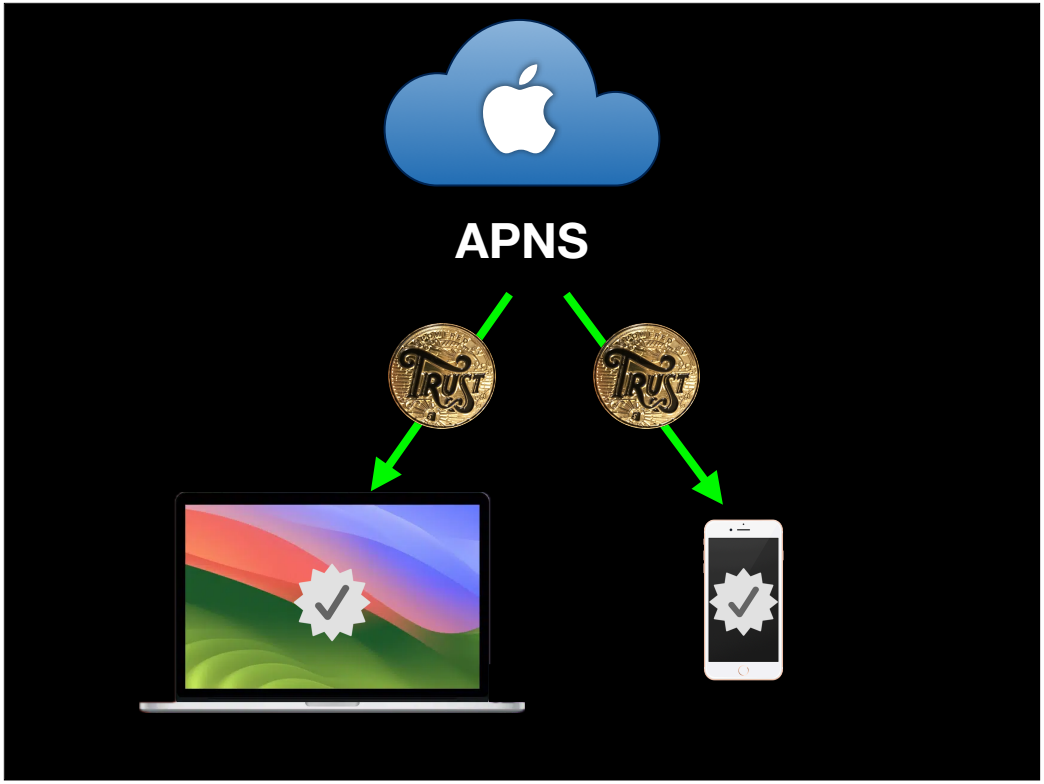
The device is now managed by the MDM server and is able to communicate securely.



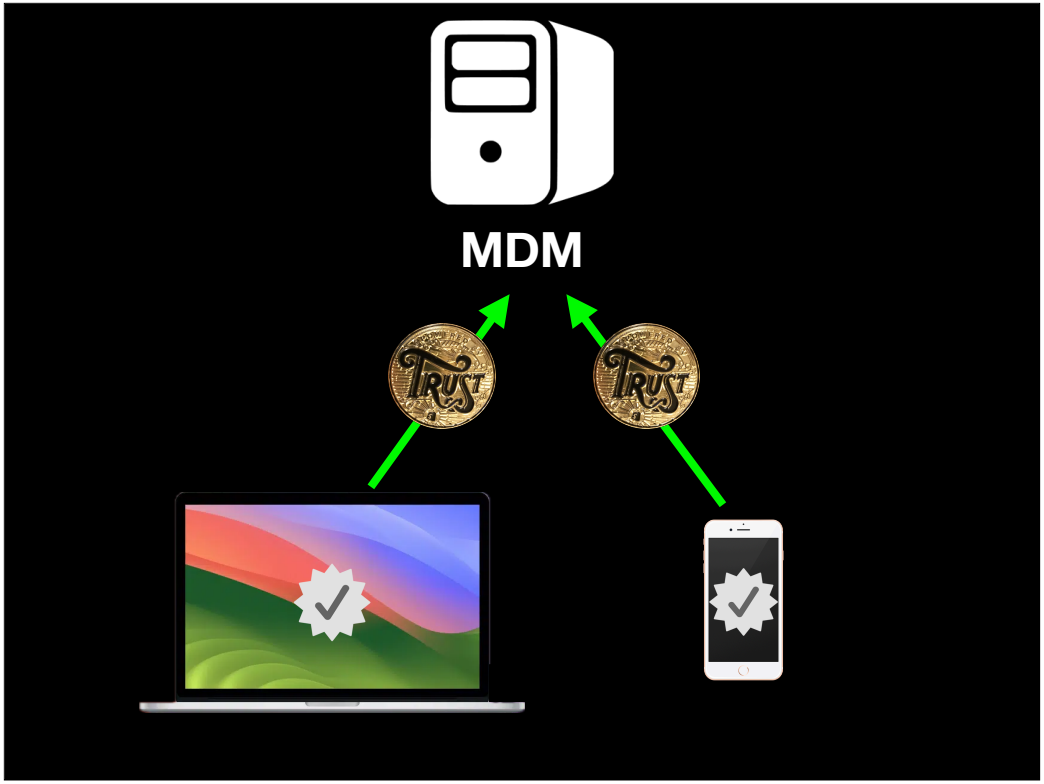
Now that the MDM profile is in place on the device, the device registers itself with APNS using the topic



APNS then generates a token, based on the combination of the device ID and the MDM topic.



Once APNS has generated the token, it’s sent down to the device.

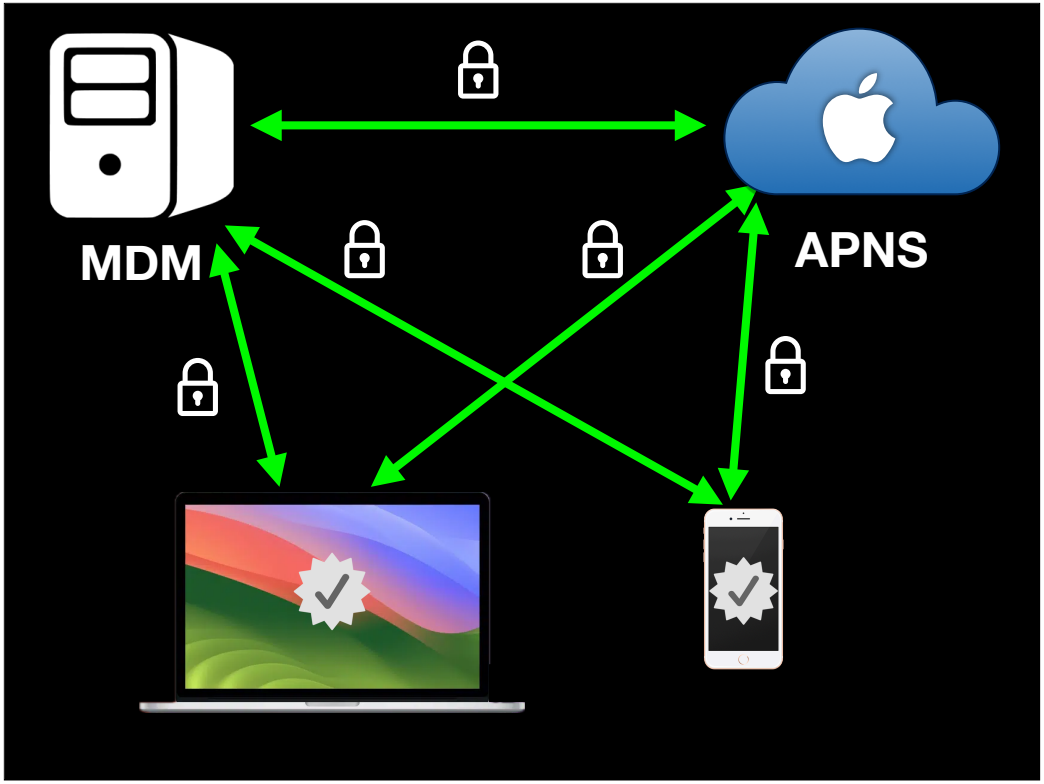


Once the token is sent to the device by APNS, the device then forwards it to the MDM server. Once the MDM server has it, the token is associated with the MDM's record for that device.

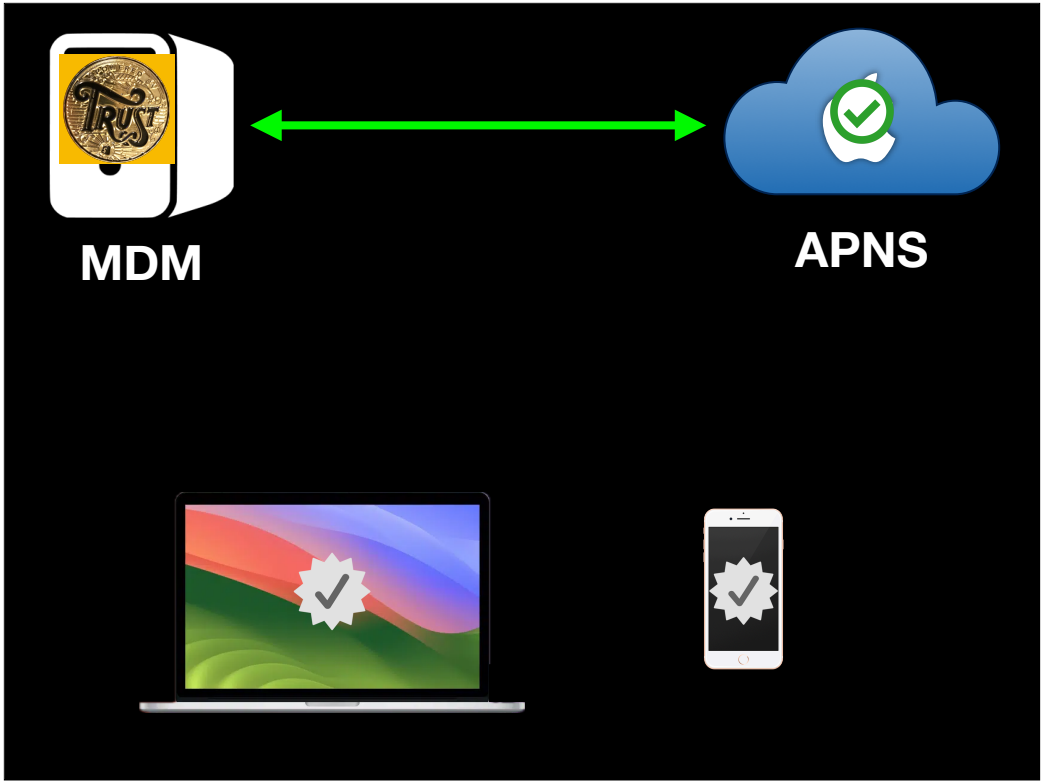
The device token may change occasionally. When a change is detected, the device automatically checks in with the MDM server to report the new token.

PushMagic

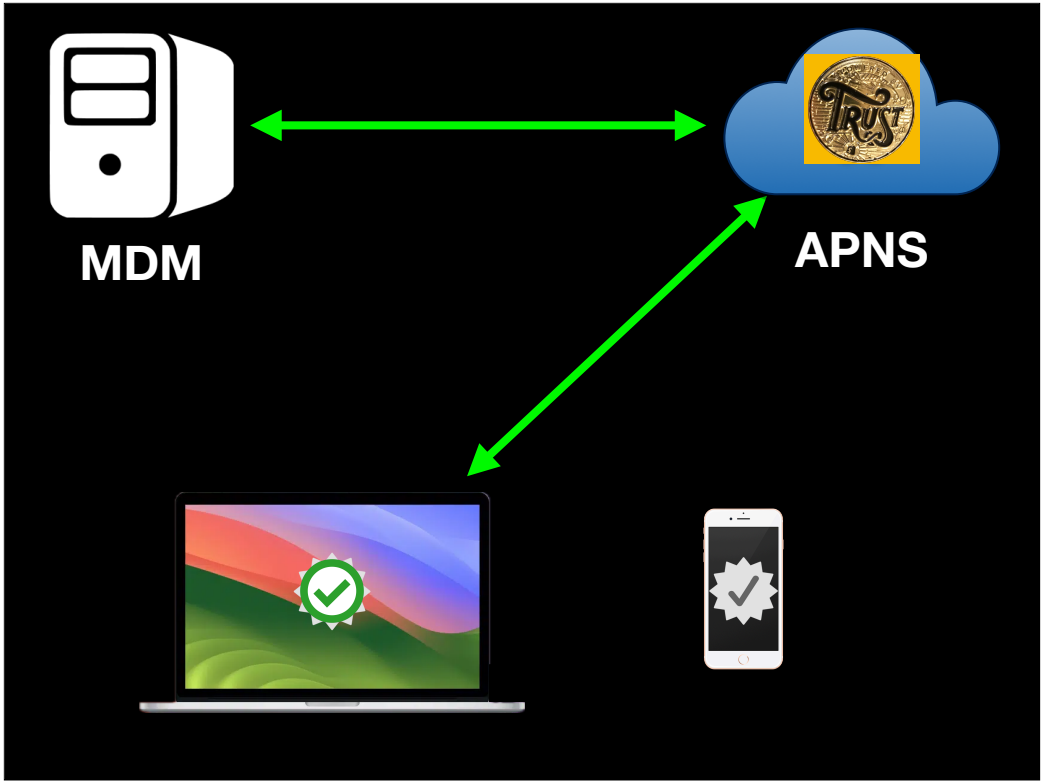
Along with the token generated by APNS, the device itself also generates a unique string known as PushMagic. This is sent to the MDM server and is associated with the MDM's record for that device. This PushMagic value is a UUID which ensures that the MDM server is talking to the right device when it sends commands.



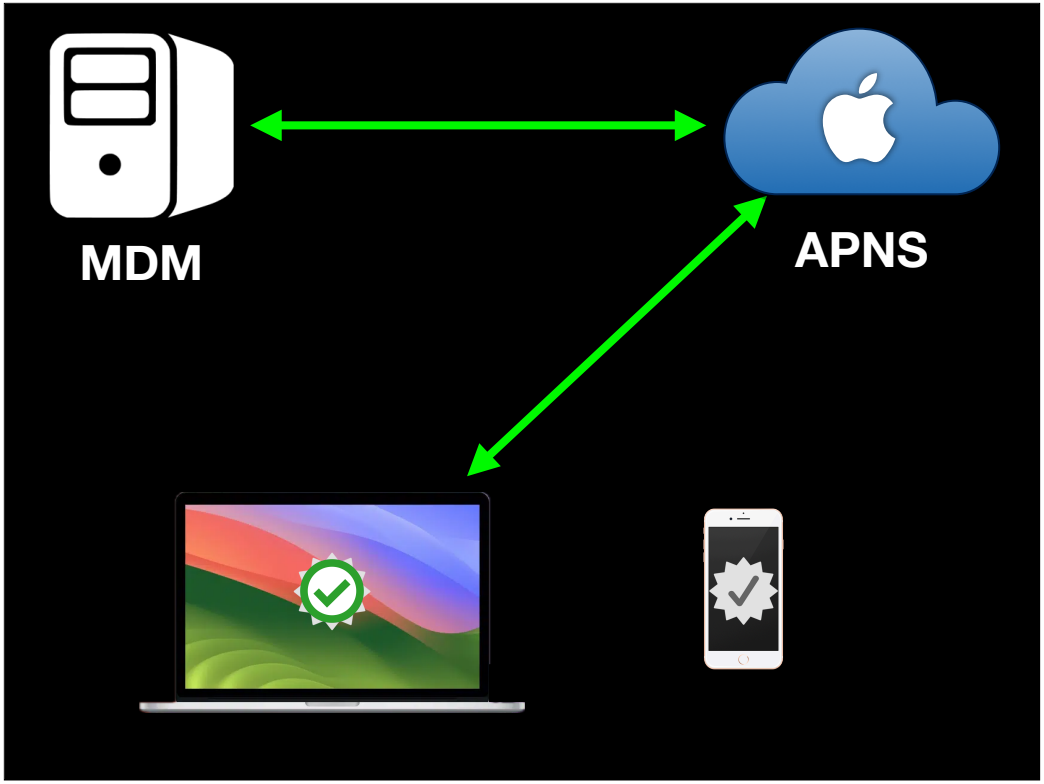
So now everything involved has the right cryptographic trust relationships in place to communicate securely with each other. Let’s use that to send commands to our managed devices.



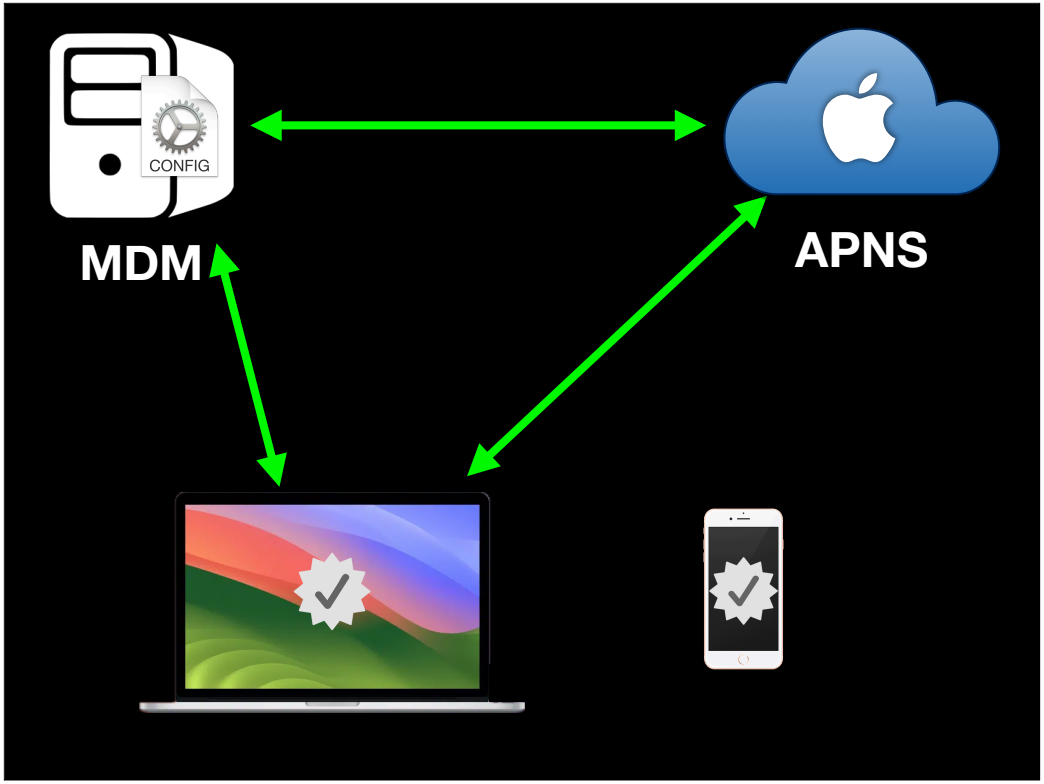
First step is the MDM server creating a notification payload using the token and PushMagic of the device it wants to send something to.



Next, APNS forwards the notification to the device. The device verifies that the topic, token and PushMagic match, then accepts the notification.

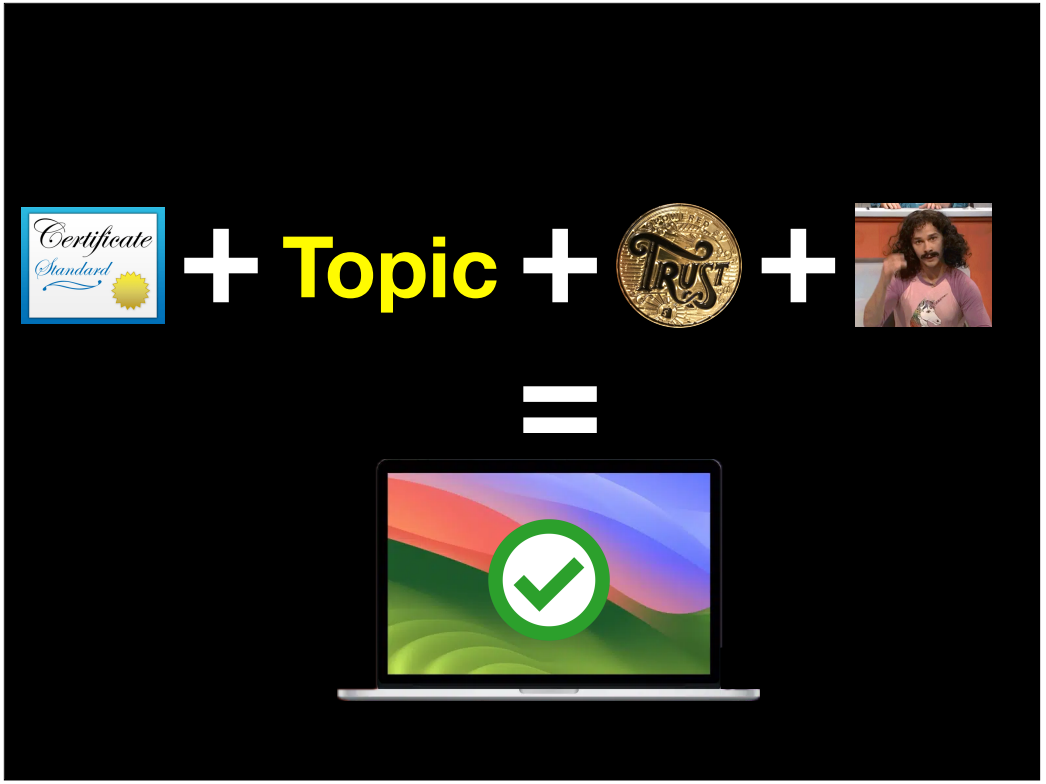


The device sends feedback via APNS that it successfully received and decoded the notification.

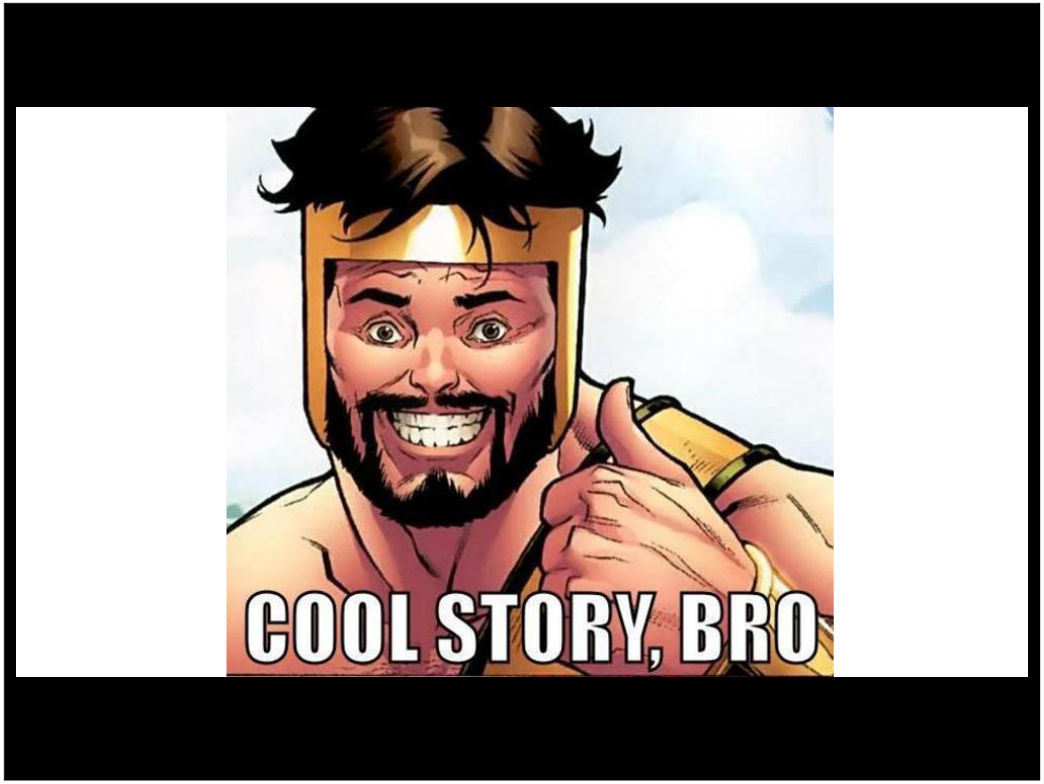


In response to the notification, the device checks in with the MDM and gets whatever command or configuration settings that the MDM has for it.

One thing that’s important to understand is that no management data goes over APNS. APNS is only used as a trigger mechanism to make the device ask its MDM for commands to run. This genericness is what allows APNS to be used by everybody who needs to send push notifications, because APNS is only ever telling the device to check in with whatever originally generated the notification.



APNS is also pretty secure because of its verification scheme. By the time an APNS notification reaches its target device, it relies on the push notification certificate being valid, the topic being valid, the token being valid and the push magic string being valid. If any of those are not valid, the notification is automatically discarded by the system.



So that’s MDM, which we’ve all been using for over a decade to manage Apple devices.

What is DDM?

So what’s Declarative Device Management or DDM?

What it is:

New data management paradigm:

- **Avoids common performance and scaling issues seen with MDM**

Here's what it is - a new data management paradigm which uses a declarative data model which is designed to address and avoid performance and scaling issues commonly seen with MDM.

What it is not:

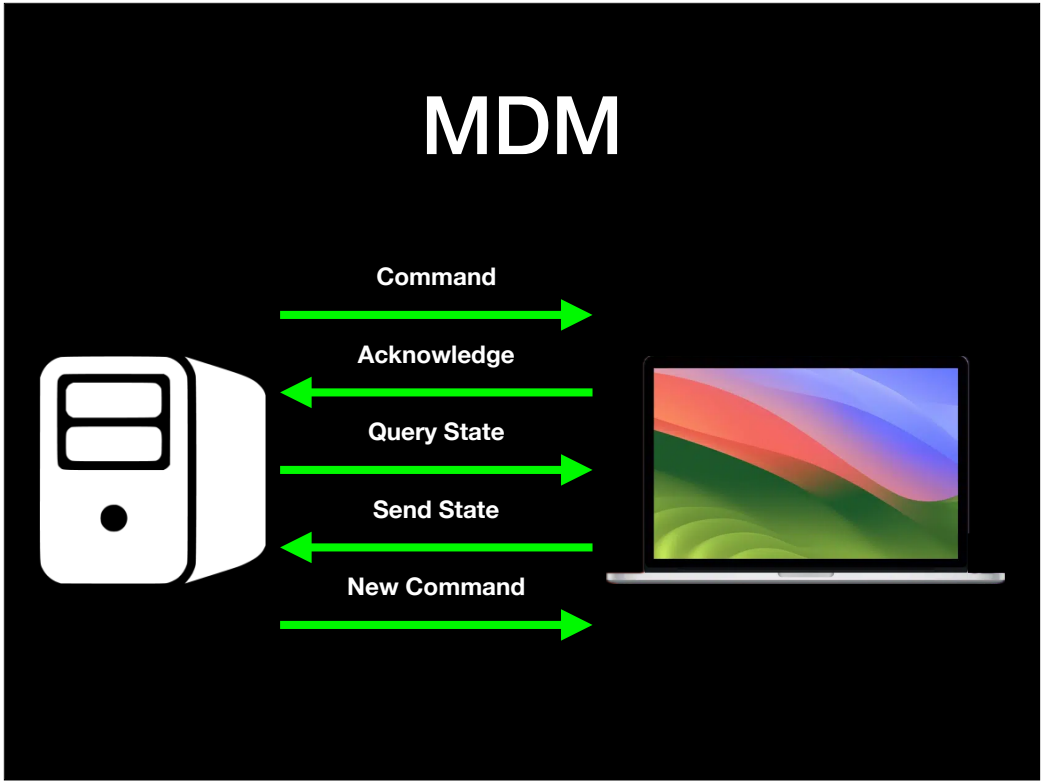
New management protocol:

- **DDM has been added to the existing MDM protocol to make DDM adoption easier.**

Here’s what it is not: a new protocol. DDM has been added to the MDM management protocol to make switching to DDM easier.

Why use DDM?

Why do we want to use it?



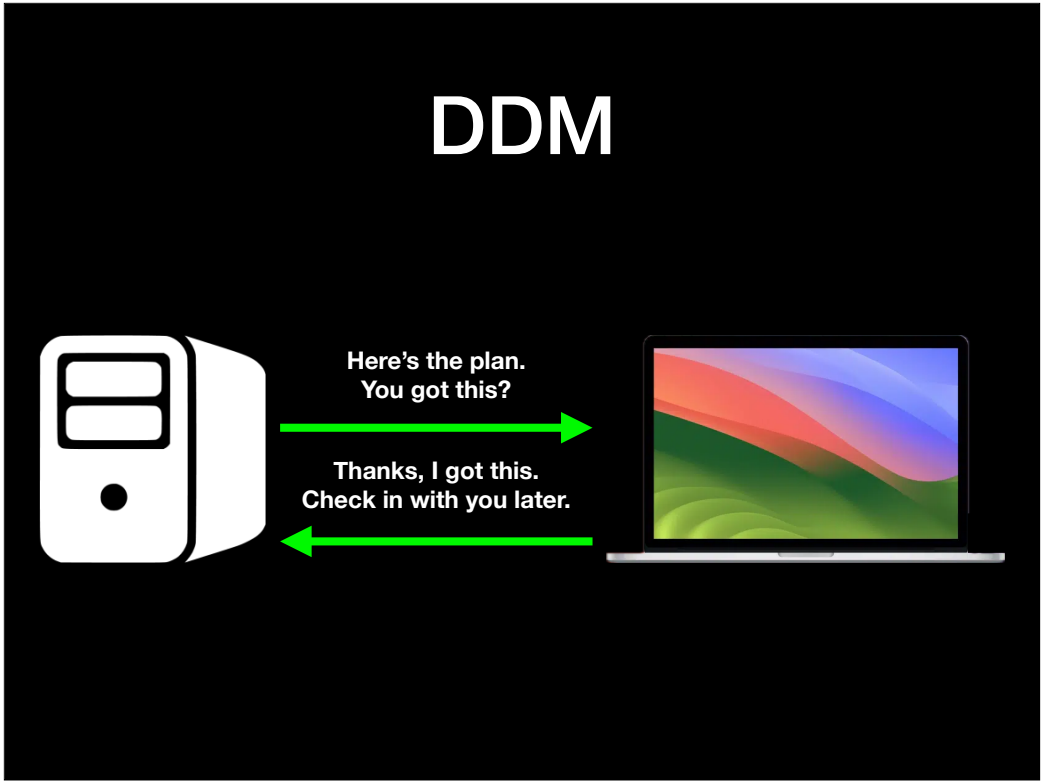
To make a long story very short, MDM without DDM is a very server-focused management protocol. The endpoints are told to check in with their MDM server, the endpoints do and the server tells them to do something. Over and over, endpoint checks in, server tells the endpoint to do something, the endpoint does it.

The server in this scenario is doing pretty much all the work of figuring out what to do. The endpoint's role is doing what the server tells it to do.

MDM

- **Each management workflow takes time and multiple round trips between the MDM server and the managed device**
- **Performance challenges grow larger as the number of devices being managed increases across the organization**

Apple wants to ensure that the MDM protocol remains responsive and scales to meet a growing population of devices, so changes are needed.



DDM is designed to take some of the load off of the server by moving some of the management to the endpoints. With DDM, the endpoint is being given more responsibility for managing itself and deciding when to report information back to the management server. This change does three things right away:

1. It reduces the network resources needed for MDM communication.
2. It reduces load on the management server.
3. It saves time overall.

DDM Data Model

- **Declarations**
- **Status Channel**
- **Extensibility**

Like MDM, DDM uses a structured data model. Let's take a look at what it looks like, starting with declarations.

Declarations

```
{
  "Type":"com.apple.configuration.passcode.settings",
  "Identifier":"4AE28C9F-7082-4521-B435-2550E8B4D57A",
  "Server Token":"EAB5142C-7E86-4C8E-B576-B7F94D0F6CEA",
  "Payload": {stuff goes here}
}
```

Declarations are the policies sent to a device and they have a structure which looks like this. And surprise! Now it's JSON in place of MDM's XML.

Declarations

```
{
  "Type":"com.apple.configuration.passcode.settings",
  "Identifier":"4AE28C9F-7082-4521-B435-2550E8B4D57A",
  "Server Token":"EAB5142C-7E86-4C8E-B576-B7F94D0F6CEA",
  "Payload": {stuff goes here}
}
```

Type: Identifies the type of policy

Type identifies the policy that the declaration is using.

Declarations

```
{
  "Type":"com.apple.configuration.passcode.settings",
  "Identifier":"4AE28C9F-7082-4521-B435-2550E8B4D57A",
  "Server Token":"EAB5142C-7E86-4C8E-B576-B7F94D0F6CEA",
  "Payload": {stuff goes here}
}
```

Identifier: Unique identifier for declaration

Identifier provides a unique identifier for the declaration in the form of a UUID

Declarations

```
{
  "Type":"com.apple.configuration.passcode.settings",
  "Identifier":"4AE28C9F-7082-4521-B435-2550E8B4D57A",
  "Server Token":"EAB5142C-7E86-4C8E-B576-B7F94D0F6CEA",
  "Payload": {stuff goes here}
}
```

ServerToken: Unique version identifier, based on the Identifier value

ServerToken provides a unique identifier for the current version of the declaration, based on the Identifier value in the declaration. This will also be in the form of a UUID.

Declarations

```
{
  "Type":"com.apple.configuration.passcode.settings",
  "Identifier":"4AE28C9F-7082-4521-B435-2550E8B4D57A",
  "Server Token":"EAB5142C-7E86-4C8E-B576-B7F94D0F6CEA",
  "Payload": {stuff goes here}
}
```

Payload: Settings for the policy

Payload contains the settings for the policy being sent by the declaration.

Declarations

- **Type: Identifies the type of policy**
- **Identifier: Unique identifier for declaration**
- **ServerToken: Unique version identifier, based on the Identifier value.**
- **Payload: The settings for the policy.**

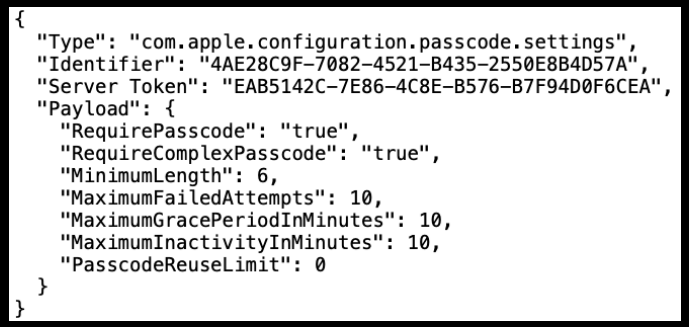
As a quick review, here's all that makes up the structure of a declaration.

Declaration Types

- **Configurations**
- **Assets**
- **Activations**
- **Management**

Now let’s dig into the various declaration types.

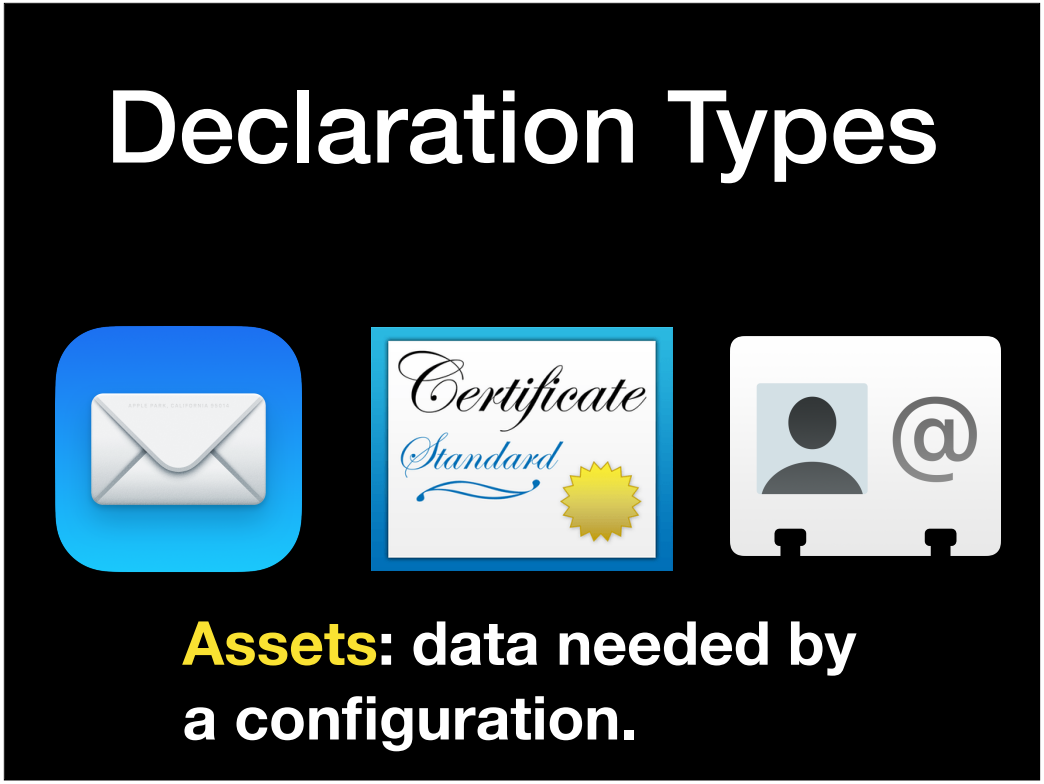
Declaration Types



```
{
  "Type": "com.apple.configuration.passcode.settings",
  "Identifier": "4AE28C9F-7082-4521-B435-2550E8B4D57A",
  "Server Token": "EAB5142C-7E86-4C8E-B576-B7F94D0F6CEA",
  "Payload": {
    "RequirePasscode": "true",
    "RequireComplexPasscode": "true",
    "MinimumLength": 6,
    "MaximumFailedAttempts": 10,
    "MaximumGracePeriodInMinutes": 10,
    "MaximumInactivityInMinutes": 10,
    "PasscodeReuseLimit": 0
  }
}
```

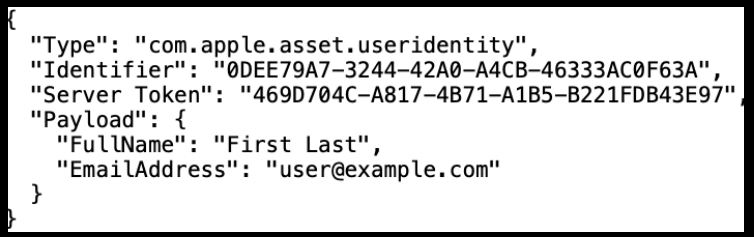
Configuration: represents policies being applied to the device.

The first is the configuration type and this should be pretty familiar. It's a policy definition, similar to what's used in MDM profiles.



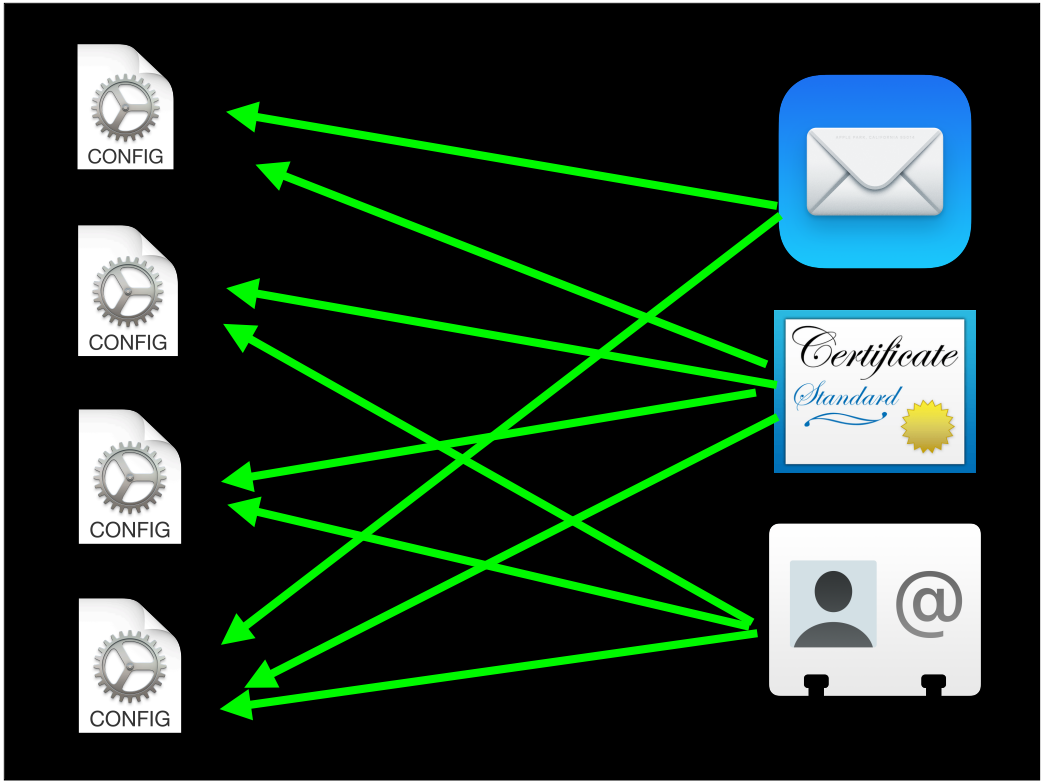
Next, there are assets and these are data needed by a configuration, like MDM URLs, certificates, email addresses or contact information.

Declaration Types



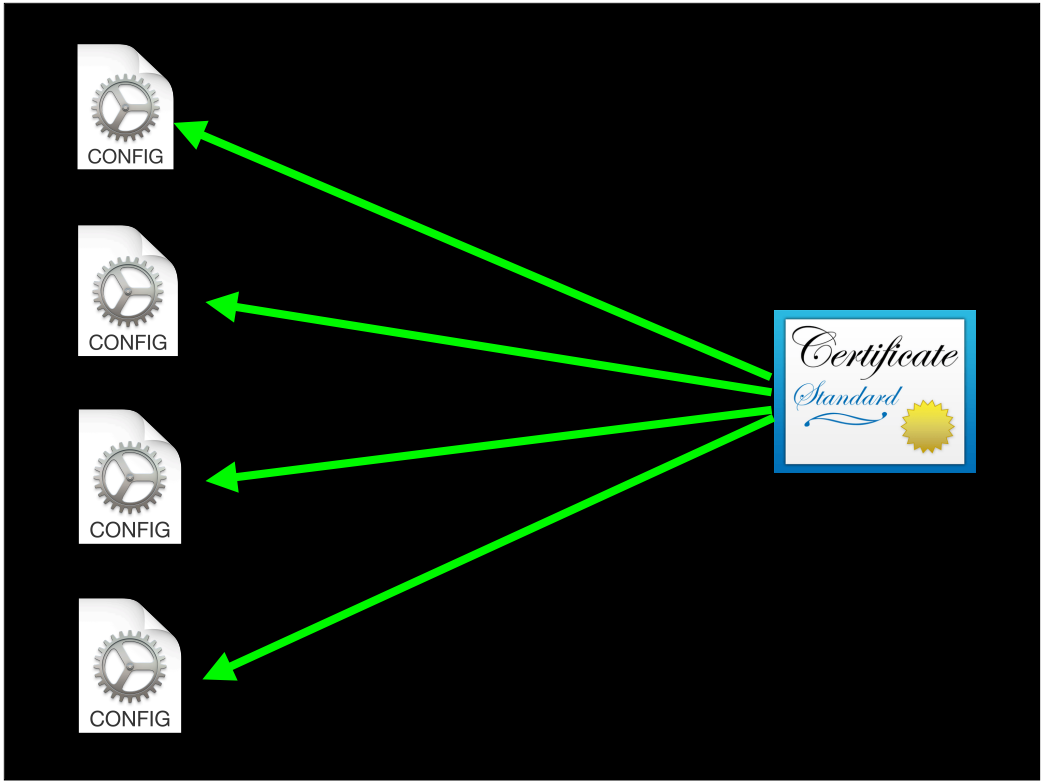
Assets: data needed by a configuration.

Here’s how an asset looks as a declaration.



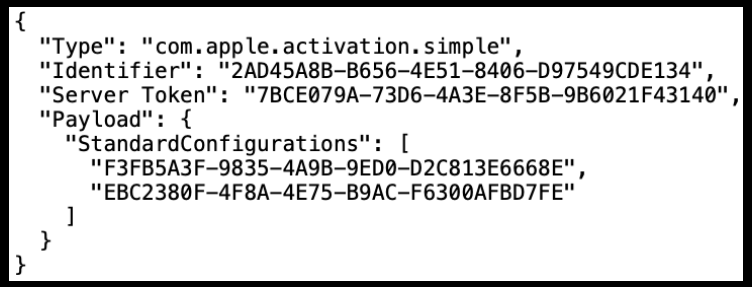
The great thing about assets in DDM is that they can have a one-to-many relationship. One asset can be used for multiple configurations.

This is a great advantage for DDM configurations over MDM profiles, as MDM currently requires that resources for that profile need to be available in the profile. For example, if you need to have the same certificates for authentication for both WiFi and VPN, this either means you need to have your WiFi and VPN configuration in the same profile as your certificates, or you're duplicating the same certificates in the profiles used to configure WiFi and VPN on your devices. In addition, any time we would need to update a certificate in this scenario, it's necessary to remove and reinstall the profile. Especially in the case of a WiFi profile update, this may mean that removing the profile means removing the Mac's only way of communicating back to the MDM server. So a removal of the profile may be the only step the device accomplishes because now it can't communicate with the MDM to put the WiFi profile back.



With this one to many relationship, you can change one asset and it'll update in all of the configurations which reference it. This is a huge step forward in functionality because now we can make incremental updates to a configuration without having to re-push the profile. Going back to the WiFi profile issue I just mentioned, this should mean that WiFi stays up and working the whole time during the profile update process.

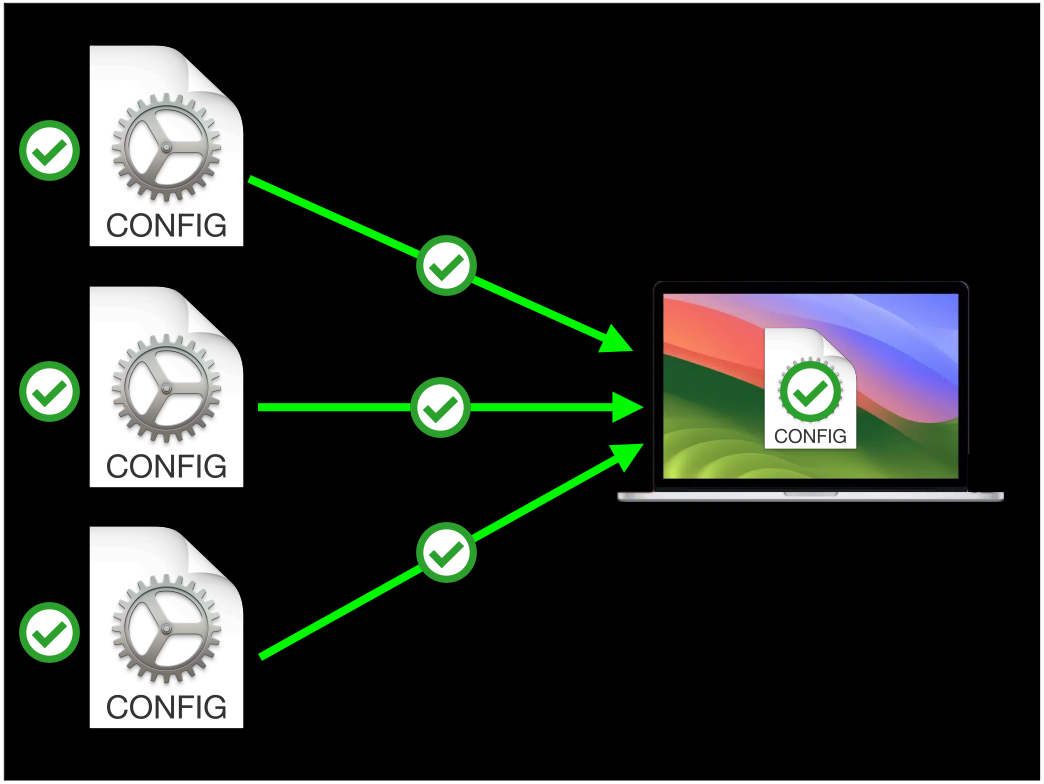
Declaration Types



```
{
  "Type": "com.apple.activation.simple",
  "Identifier": "2AD45A8B-B656-4E51-8406-D97549CDE134",
  "Server Token": "7BCE079A-73D6-4A3E-8F5B-9B6021F43140",
  "Payload": {
    "StandardConfigurations": [
      "F3FB5A3F-9835-4A9B-9ED0-D2C813E6668E",
      "EBC2380F-4F8A-4E75-B9AC-F6300AFBD7FE"
    ]
  }
}
```

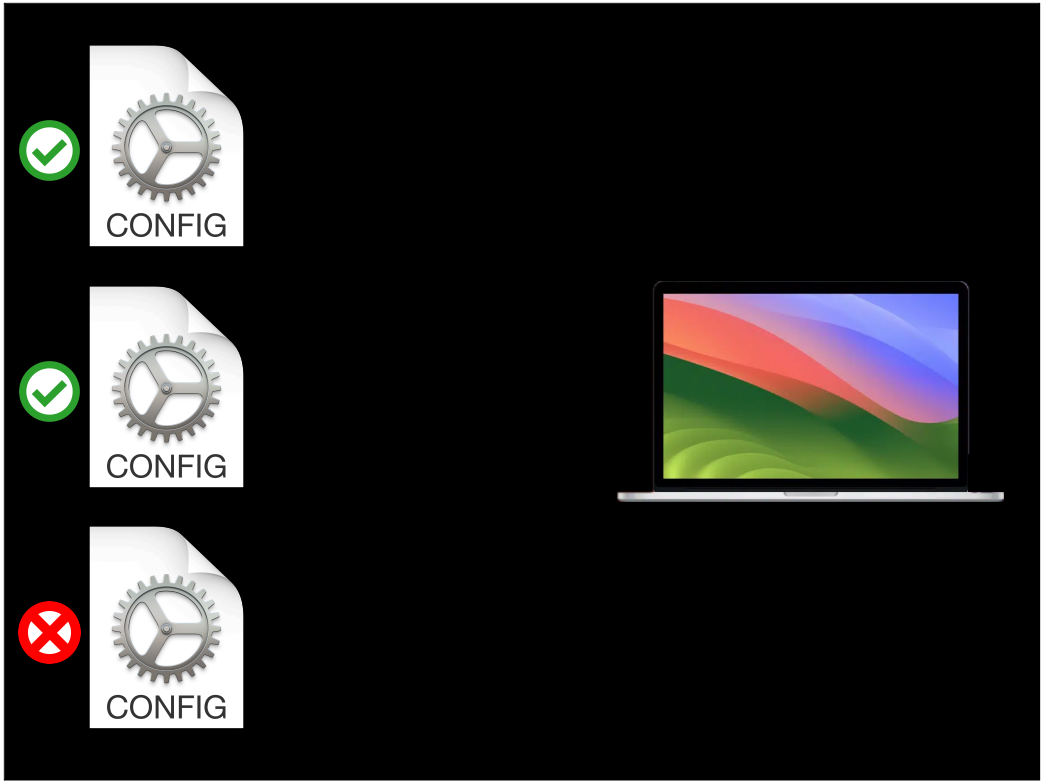
Activation: group of configurations

Activations are a group of configurations which get applied to a device. They also specify the logic that determines how and when the policies defined by configurations get applied to the managed device.

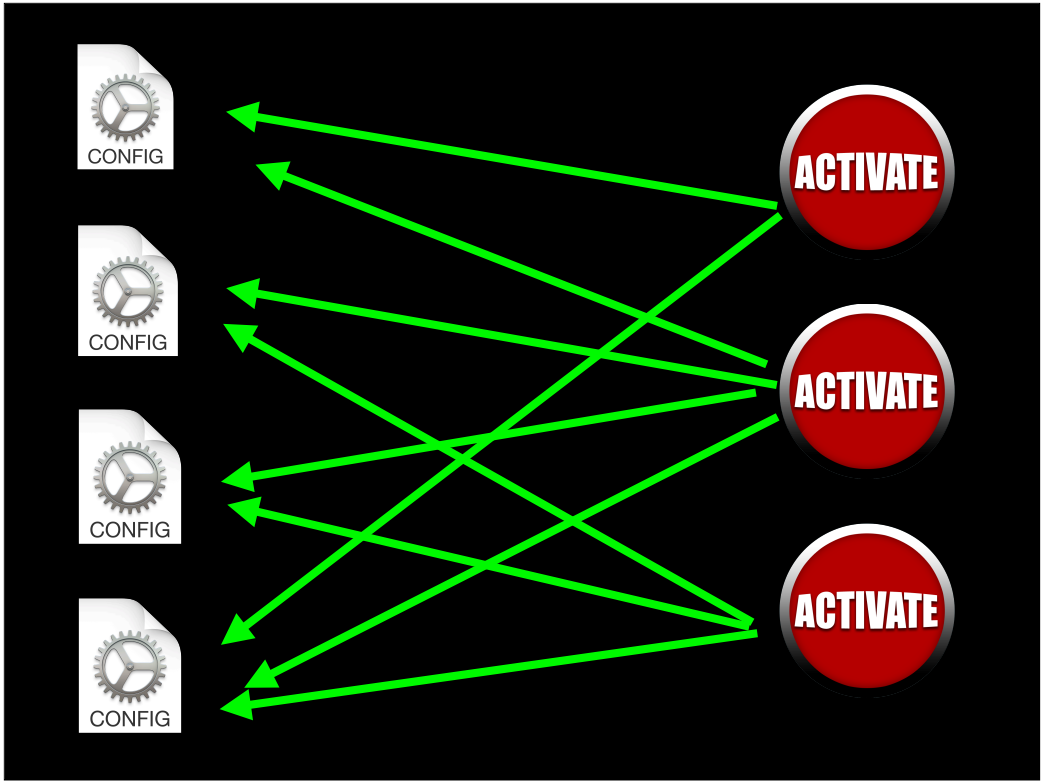


These configurations are also applied atomically to the device, such that the system applies all referenced configurations together, or it applies none of them.

All of the configurations, and assets referenced by those configurations, must be valid in order for the system to apply the activation.



If you have an invalid configuration, or an invalid asset defined in a configuration, nothing happens and none of the configurations get applied.



Activations in DDM can have a many-to-many relationship, where configurations can be referenced by multiple activations. One activation can hit multiple configurations at once and a configuration can be used by multiple activations. This means you can have your devices process complex business logic, while DDM means that the devices are handling this logic autonomously in place of the server having to do all the work.

Activation Predicates

- **Determines activation behavior**
  - **Boolean logic**
  - **Device will only process activation if predicate evaluates to TRUE**

Along with activations, we get predicates. These determine the activation state (if active or not) on a device. This is going to use boolean logic and an activation will only be processed by a device if the predicate is evaluated to be true.

Activation Predicates

```
{
  "Type": "com.apple.activation.simple",
  "Identifier": "2AD45A8B-B656-4E51-8406-D97549CDE134",
  "Server Token": "7BCE079A-73D6-4A3E-8F5B-9B6021F43140",
  "Payload": {
    "StandardConfigurations": [
      "F3FB5A3F-9835-4A9B-9ED0-D2C813E6668E",
      "EBC2380F-4F8A-4E75-B9AC-F6300AFBD7FE"
    ]
  },
  "Predicate": "(device.model.family == 'iPad')"
}
```

Here’s an example of our previous activation declaration where the predicate is set to only evaluate to true on iPads.

Declaration Types

Management: properties of the overall management state on the device.

The final kind of declaration is a management declaration. Management declarations are used to send overall management state to the device.

Management Declarations

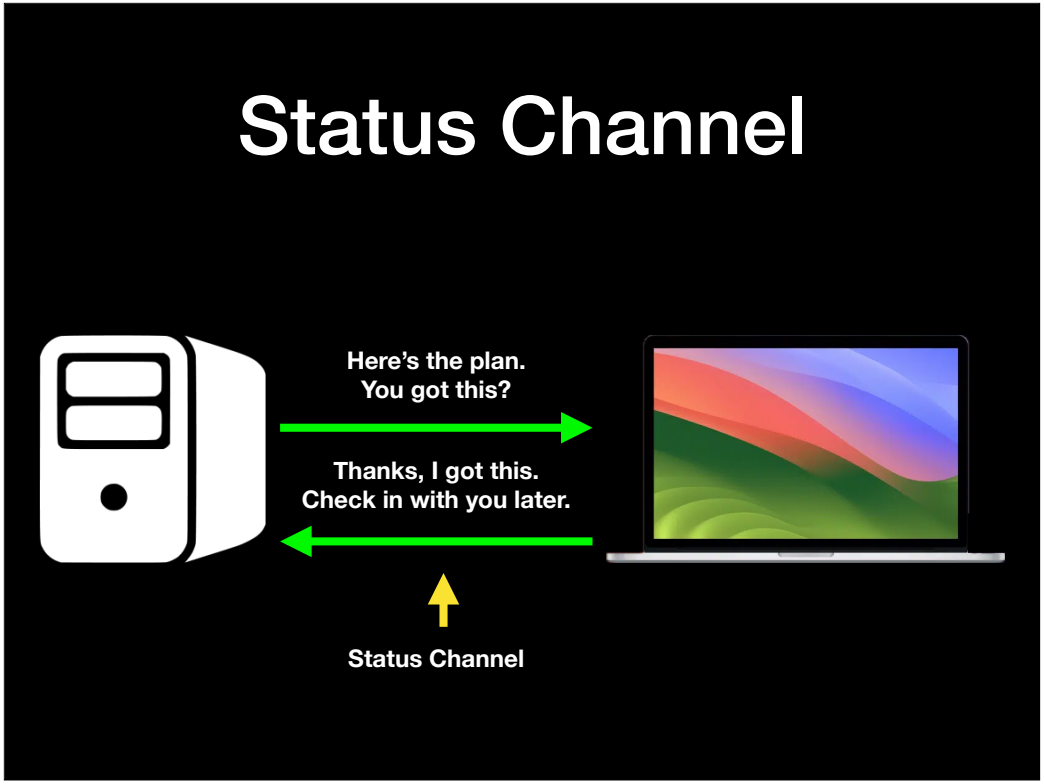
- **Represent properties of the overall management state**
- **Organization information**
- **MDM server capabilities**
- **Conveys static information to the device.**

This includes a declaration describing details about the organization as well as a declaration that describes the capabilities of the MDM server. These declarations are helpful for conveying static data to the device which shouldn't change.

Declarations

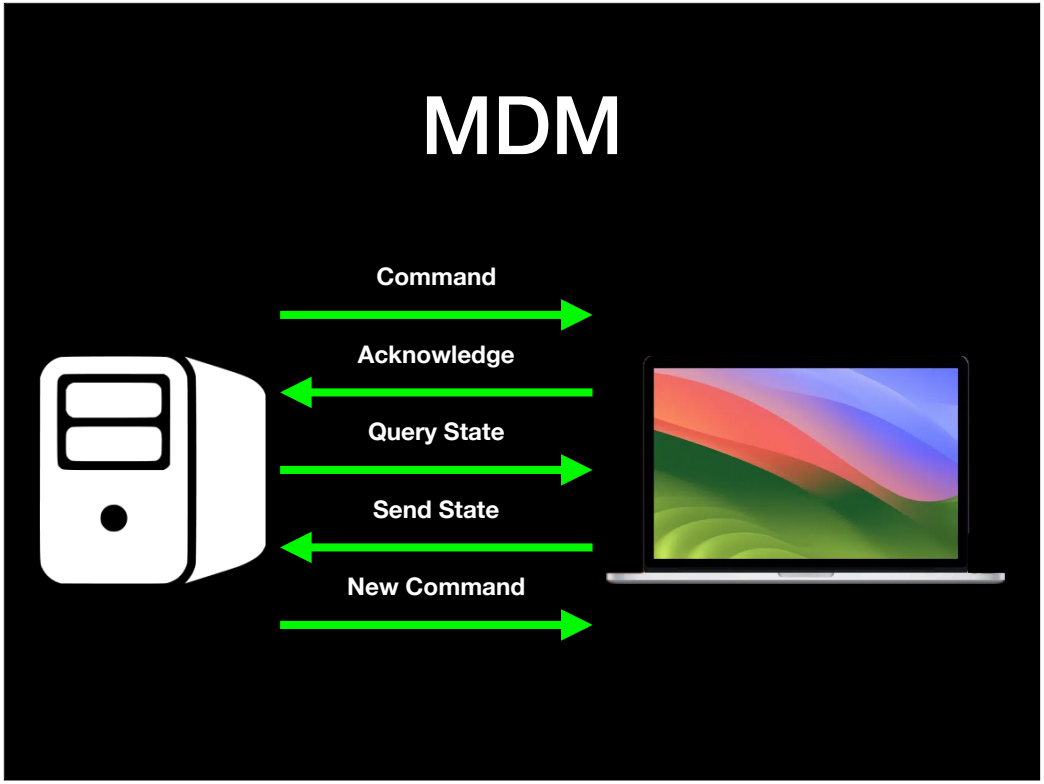
- **Configuration: represents policies being applied to the device.**
- **Assets: data needed by a configuration.**
- **Activation: group of configurations**
- **Management: properties of the overall management state on the device.**

So those are our four types of declarations. But there's more to dig into, so next let's look at DDM's status channel.



In the DDM communication model, there's communication between the management server and the device going on, but there's less communication because of the Status Channel. Status Channel is a way for the server subscribe to only that information your server needs.

Once the server has subscribed to specific status items, the client will report to the server only on those status items.



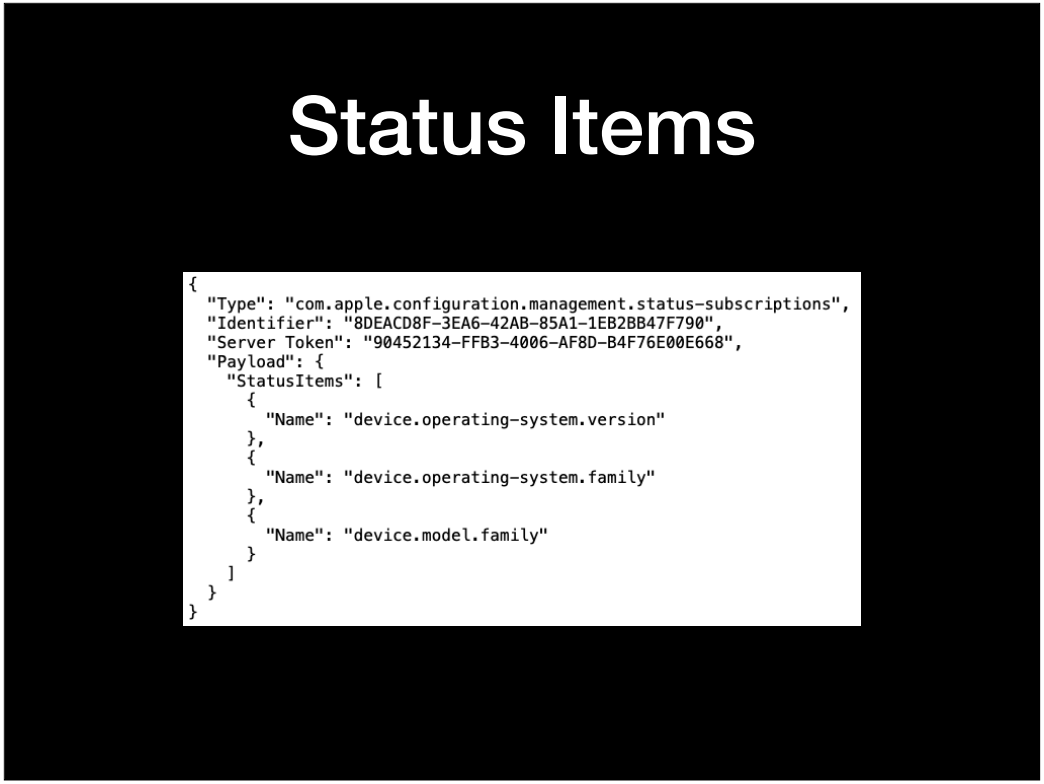
This model of endpoint to server communication cuts way back on the unneeded chatter which happens with MDM and provides the server only that information that it actually needs.

Status Items

- **device.operating-system.family**
- **device.operating-system.version**
- **device.model.family**
- **device.model.identifier**

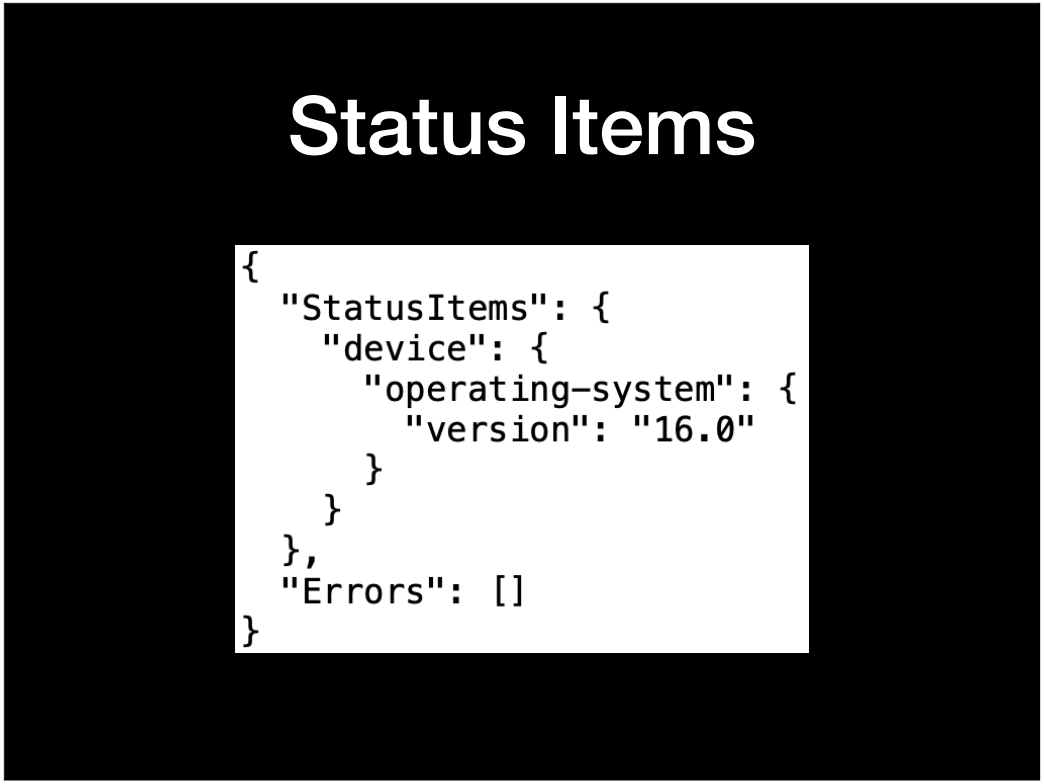
Status items are identified by key-paths, which consist of period-separated string tokens. Examples of these are shown on the screen.

Status Items



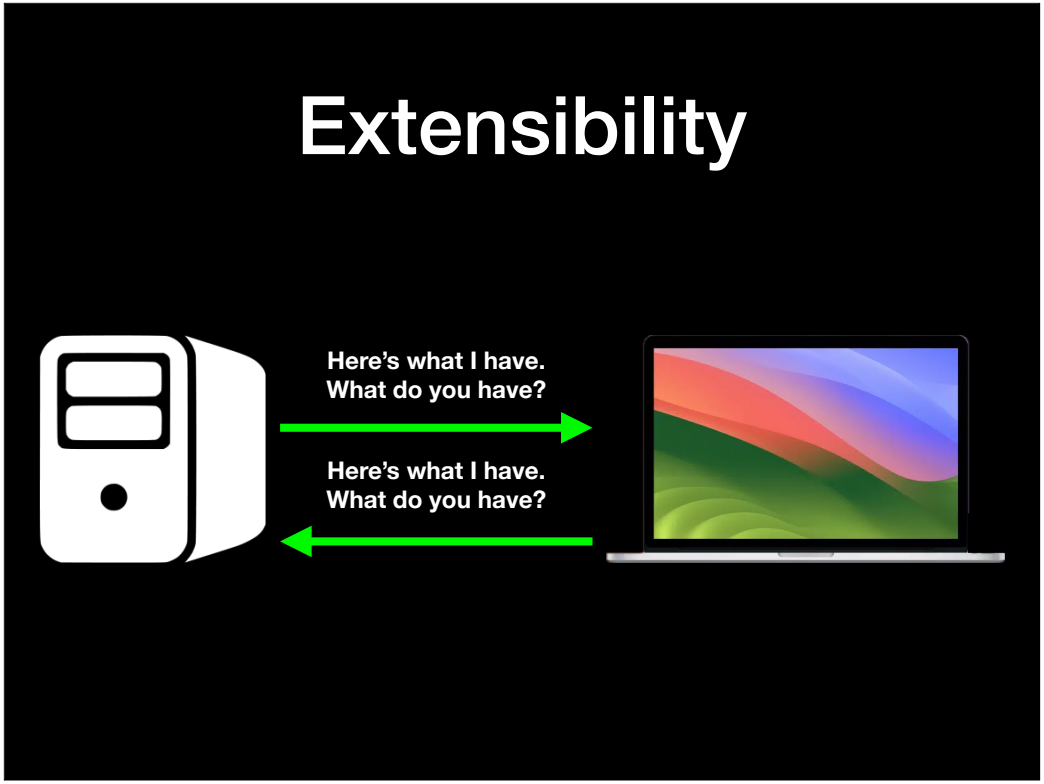
Here’s how a status channel subscription may look.

Status Items



```
{
  "StatusItems": {
    "device": {
      "operating-system": {
        "version": "16.0"
      }
    }
  },
  "Errors": []
}
```

A response from the endpoint may look like this.



Apple's products have a long life cycle, so it's essential to maintain compatibility between different versions of an MDM solution with Apple devices of varying ages and capabilities.

Declarative management means that both the device and the MDM server now advertise supported capabilities to each other. Each side of the management system is now aware of what the other side supports because each side advertises their supported capabilities to each other. Each side knows when it can start taking advantage of new features. These advertisements include a list of supported features, representing both major and minor protocol updates.

The endpoint will advertise supported payloads, including the full set of declarations and

status items supported by that device. Meanwhile, the MDM server advertises its capabilities to the endpoint via a management declaration.

Extensibility

- **Both endpoint and MDM server advertise what they support**
  - **Supported features**
  - **Supported payloads**
- **MDM server indicates support in management declaration.**
- **Client indicates support as specific status item.**

When the MDM server is upgraded, it synchronizes all new capabilities with the device like it would any other declaration, which allows the device to start working with the MDM server's new features. Likewise, the endpoint sends new capabilities to the MDM server as a specific status item when that device's capabilities change.

DDM Integration with MDM

- **DDM is integrated into the MDM protocol for enrollment, transport and authentication.**
  - **Declarations and MDM commands / profiles coexist**
  - **Unenrolling from MDM management also removes all DDM.**

DDM is integrated into the MDM protocol and uses it for managing the enrollment / unenrollment process, for handling the HTTP transport communication, and for device and user authentication.

DDM Integration with MDM

- **DDM is not disruptive to MDM.**
  - **Declarations and MDM commands / profiles coexist.**
  - **DDM can send and install MDM profiles as configurations.**

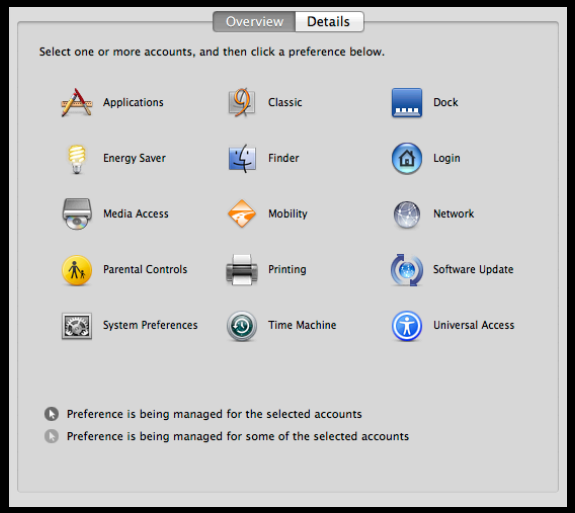
DDM should not interfere with existing MDM behavior, except where Apple says it does; more on that in a bit. Declarations in fact leverage existing MDM behaviors using an MDM command for activation and an MDM CheckIn request for synchronization and status reports. This is intended to ease the migration process from MDM to DDM.

DDM Integration with MDM

```
{
  "Type": "com.apple.configuration.legacy",
  "Identifier": "4B0E572A-7188-4249-87B3-C58F2340ECF8",
  "Server Token": "815357CC-6571-4992-A8E4-AFDB9DB0CD89",
  "Payload": {
    "ProfileURL": "https://mdm.server.goes.here/profiles/importantsetting.mobileconfig"
  }
}
```

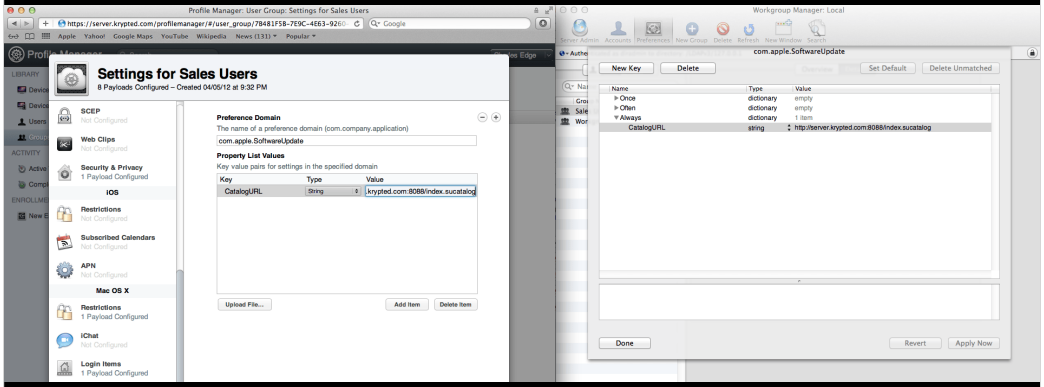
In fact, you can send an existing MDM profile as a configuration. This will allow profiles to be used with DDM, which in turn means you can shift profile-based logic to an endpoint without having to rework your profiles.

DDM Integration with MDM



Just to show how it's turtles all the way down, this means that in some cases you could have added a setting as a managed preference to Workgroup Manager in Mac OS 10.3 Panther Server in 2003.

DDM Integration with MDM

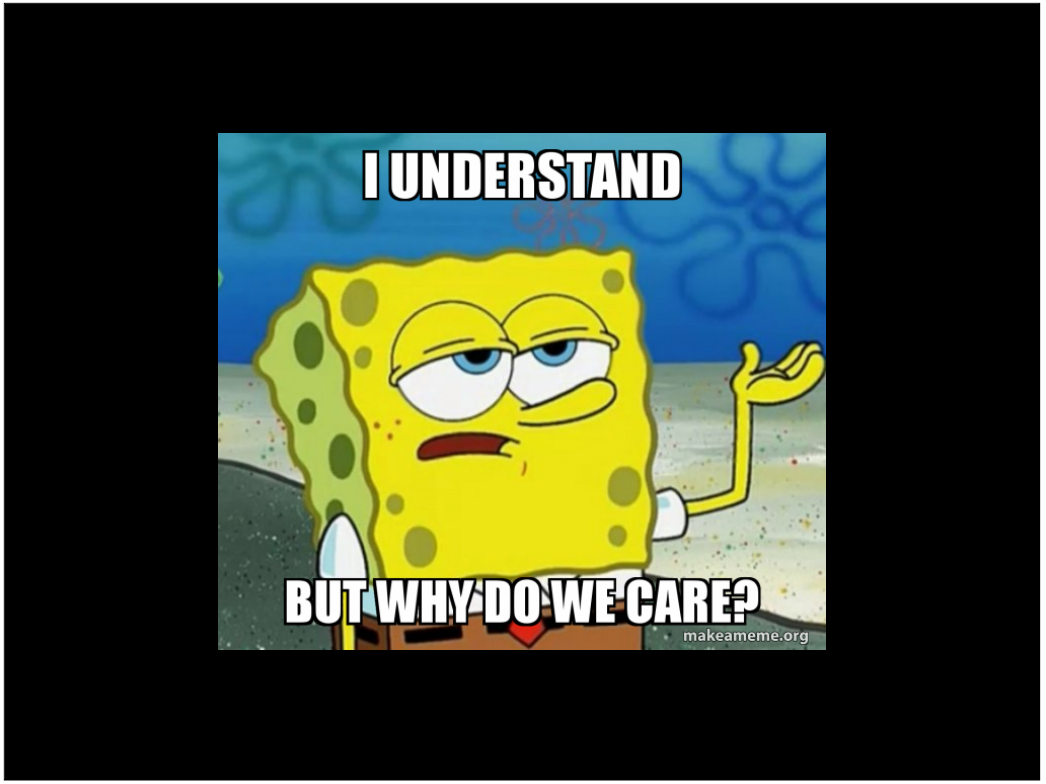


Later moved that managed preference setting into a profile in Mac OS X Lion, using profiles' support for MCX settings.

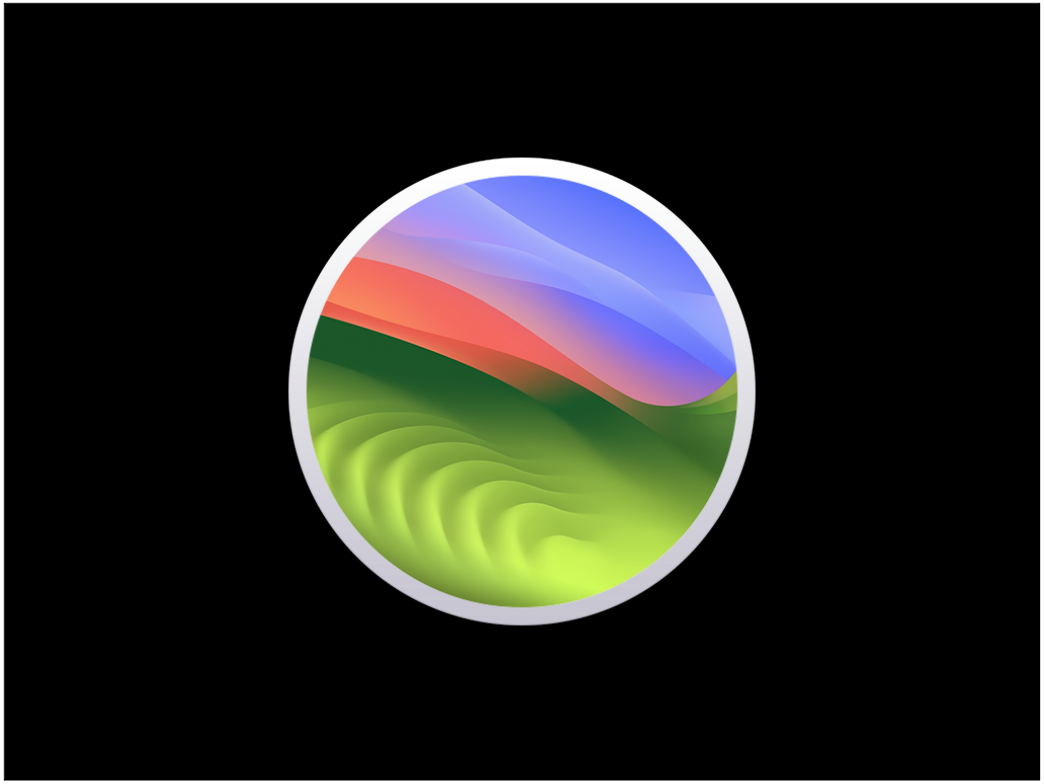
DDM Integration with MDM

```
{
  "Type": "com.apple.configuration.legacy",
  "Identifier": "4B0E572A-7188-4249-87B3-C58F2340ECF8",
  "Server Token": "815357CC-6571-4992-A8E4-AFDB9DB0CD89",
  "Payload": {
    "ProfileURL": "https://mdm.server.goes.here/profiles/importantsetting.mobileconfig"
  }
}
```

Now twenty years after first adding that setting to Workgroup Manager, you can use DDM to deploy that profile with that MCX setting.



OK, so we've looked at MDM and DDM. Why do we care about DDM? What's in it for us as admins?



For that, let’s look at some of what is available in macOS Sonoma and iOS 17.

New DDM-managed software updates

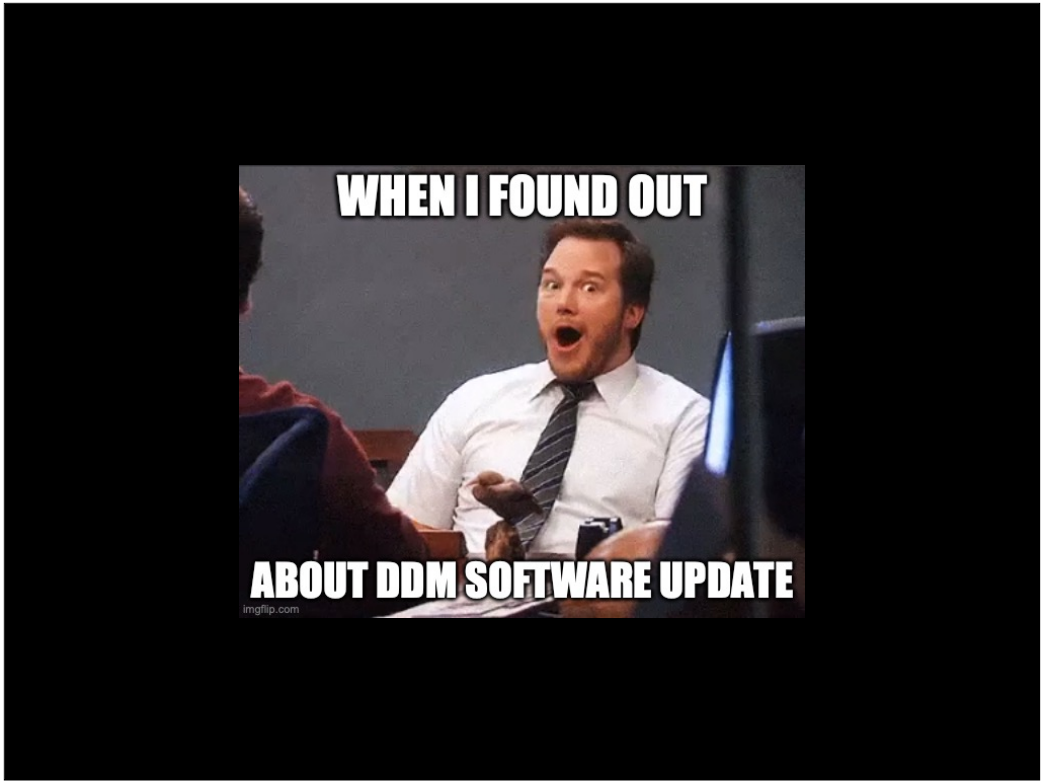
- **Can enforce software updates for a specified OS version and build at specified time.**
- **Available for the following platforms:**
  - **macOS**
  - **iOS**
  - **iPadOS**

Software updates are manageable via DDM and can enforce deploying a specified OS version and build which must be installed by a specified time.

New DDM-managed software updates

- **DDM software update configurations can coexist with MDM software update commands**
  - **Software updates enforced by DDM will take precedence over MDM commands or profiles**

To help ease migration, DDM software update commands can coexist with MDM software update commands. That said, DDM management is going to take precedence in this case over MDM commands. This is one of those situations where DDM should not interfere with existing MDM behavior, except where Apple says it does.



I mean, I don’t know about you but this is what I looked like when I found out about DDM software updates. Forget the rest of it, that alone is huge. I’m looking forward to seeing how this works on Sonoma and I expect a lot of you are too.

DDM-managed app deployment

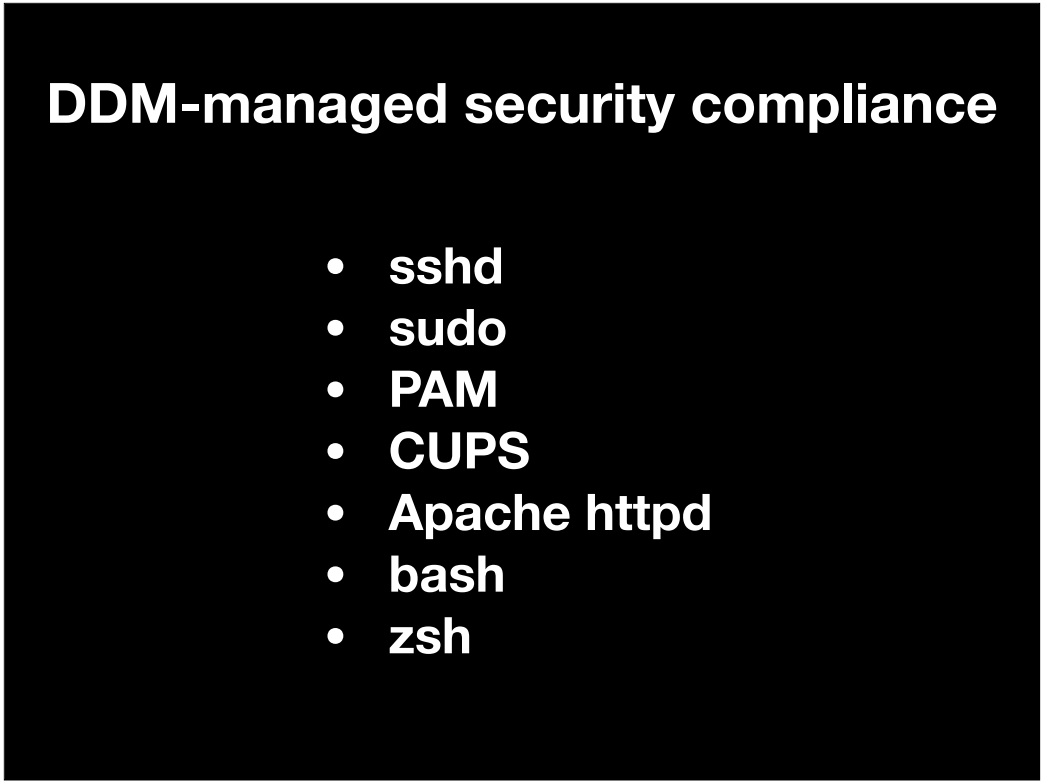
- **DDM configuration can specify an app be available on a device at a desired time.**
  - **App can be sent to the device ahead of time, then made available when needed.**
  - **Administrators can switch between sets of apps as needed.**

DDM can also help with app deployments, with options for apps being sent to the device before a specified time and only made available once the app is needed.

DDM-managed app deployment

- **App can be shown to user without the app being installed, so that the user can choose when to install it.**
  - **Since user is choosing to install, no consent prompt appears.**
- **Asynchronous reporting keeps the admin up to date on changes to managed apps on the endpoints.**

DDM also enables the option for the app being shown without actually being installed, giving the user the choice of when to install it. One benefit of this is that since the user is choosing to install, there’s no extra consent dialog which appears.



On macOS, DDM configurations can be used to specify sets of tamper-resistant system configuration files for different system services. In their session at WWDC, Apple mentioned that the built-in services listed on the screen could be managed this way.

DDM-managed security compliance

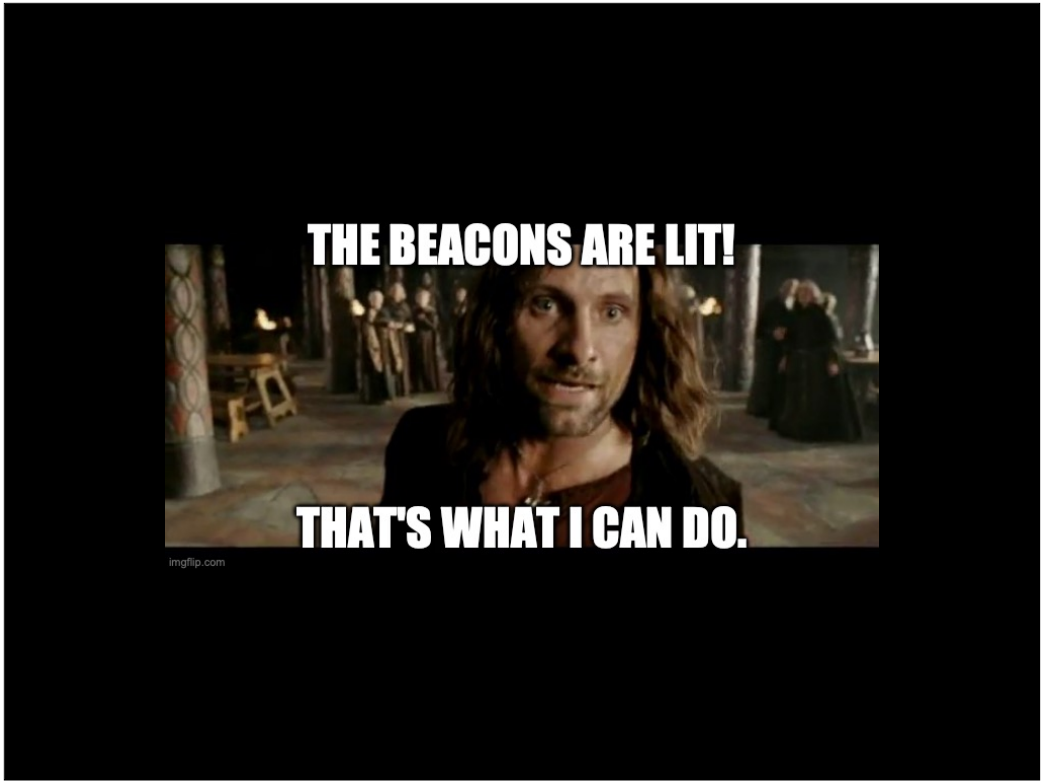
- **FileVault status monitoring**
  - **Status item: diskmanagement.filevault.enabled**
    - **Returns a boolean value to indicate whether FileVault is enabled or not**

DDM also provides improved FileVault status monitoring, with a status item which will return true or false boolean values.

Now, do you as a Mac admin need to know all this stuff about DDM? Largely no, this is stuff your MDM vendor needs to know and figure out. However, understanding what's going on with both MDM and DDM means we know what to ask for from our vendors in terms of new features and that's valuable on its own.

I see DDM as providing us as Mac admins with a deeper and wider toolkit for management than we've had before, which is a great thing.

One thing it is not as of yet is configuration management akin to that provided by Puppet, Ansible, Salt or similar solutions. With those solutions, the configuration management server provides a plan to the managed endpoint and the endpoint makes sure that configuration is applied no matter what. This includes automatically reverting changes to managed settings made by the end user on the endpoint.



As of now, DDM’s ability to report changes via Status Channel means that automatic reporting of changes is there, but automatic remediation which would be handled by the endpoint is not.

I really hope this is coming to DDM, but we are not there yet.

Useful Links

Apple Mobile Device Management: https://developer.apple.com/documentation/devicemanagement

Apple Device Management documentation: https://developer.apple.com/documentation/devicemanagement

A Push Odyssey - Journey to the Center of APNS: https://www.youtube.com/watch?v=Z-Lg9uBbmfk

Getting MicroMDM working and working with MicroMDM: https://youtube.com/watch?v=WGKT-PyHz6I

Demystifying MDM: open source endeavours to manage Macs: https://youtube.com/watch?v=6DBGlDcBKFw

Useful Links

WWDC 2023 What's New in managing Apple devices: https://developer.apple.com/wwdc23/10040

WWDC 2023 Explore advances in declarative device management: https://developer.apple.com/wwdc23/10041

WWDC 2022 Adopt declarative device management: https://developer.apple.com/wwdc22/10046

WWDC 2021 Meet Declarative Device Management: https://developer.apple.com/wwdc21/10131

Downloads

PDF available from the following link:

https://tinyurl.com/MacSysAdmin2023PDF

Keynote slides available from the following link:

https://tinyurl.com/MacSysAdmin2023Keynote