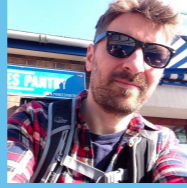


Making package uploading and deployment easier with JamfUploader



[Technical Note: The SF Mono font is required for proper playback of this deck. It is available from Apple at <https://developer.apple.com/fonts/>]

[Graham] Hello everyone, and welcome to our session about making package uploading and deployment easier with JamfUploader



Graham Pugh

ETH Zürich (Switzerland)

ETH zürich



Anthony Reimer

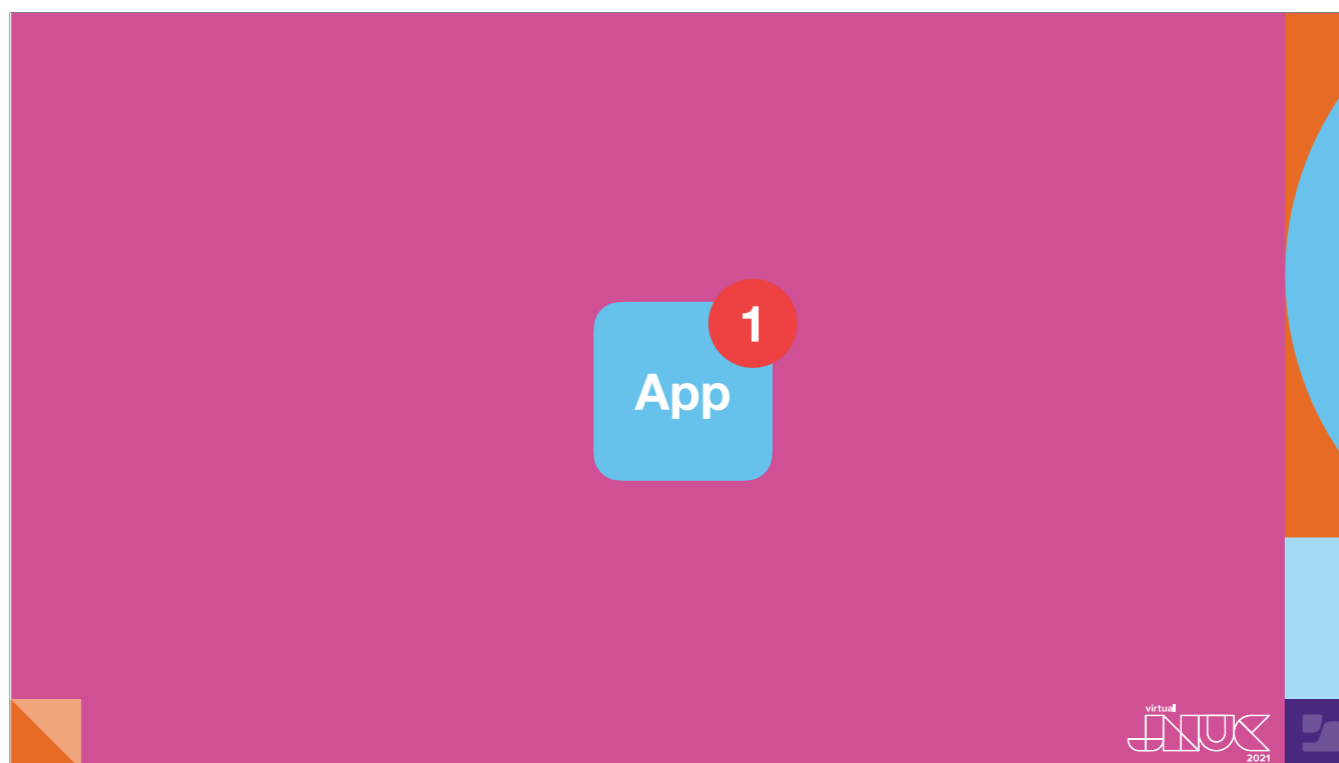
University of Calgary (Canada)

 UNIVERSITY OF
CALGARY

virtual
ENUC
2021

[Graham] My name is Graham Pugh, and I'm a Mac Client Engineer from the UK, working at ETH, a technical university in Zürich Switzerland.

[Anthony] And my name is Anthony Reimer, and I run the computer labs for Art, Music, Drama, and Dance at the University of Calgary in Canada.



[Anthony continues] Here's a common scenario for Jamf Pro administrators: you hear about an update of an App you deploy, either directly or via Self Service. It might even be urgent because of a security update or an important bug patch. What do you do now? Here's a typical manual workflow:

Steps to Update

1. Go to vendor web site
2. Download update
3. Turn into a pkg installer (if necessary)
4. Upload pkg to Jamf Pro Server
5. Add Category Info to uploaded pkg
6. Go to the policy that deploys it
7. Update the pkg info for the policy
8. Trigger as required

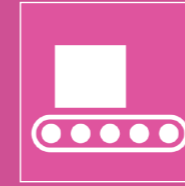


[Go through points:]

1. Go to the vendor's website or download portal
 2. Find the link to the downloader and download it.
 3. If the downloaded installer is not a package, you need to convert it to a package to use in Jamf Pro.
 4. Then, upload the package to Jamf Pro (using your Jamf Admin app or the admin console if you're using Jamf Cloud).
 5. Add a category to the uploaded package.
 6. Now find the policy that deploys that app, which might be a Patch Management policy, and
 7. switch out the package.
 8. Then flush the policy if it is a Run Once policy, or change the target version if it is a Patch Management or dynamically-scoped policy.
- If this is what you are doing now, you are viewing the right session! AutoPkg can help us automate some or all of these steps.

Steps to Update

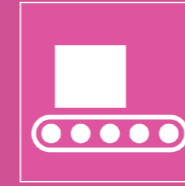
1. Go to vendor web site
2. Download update
3. Turn into a pkg installer (if necessary)
4. Upload pkg to Jamf Pro Server
5. Add Category Info to uploaded pkg
6. Go to the policy that deploys it
7. Update the pkg info for the policy
8. Trigger as required



AutoPkg can take care of the first three steps using common pkg recipes. In fact, it can even take care of Step 0, determining that an update is available in the first place. We won't go over the basics of using AutoPkg here, but there are some great conference session videos linked in the resources section of the AutoPkg wiki that can help. I'd like to highlight Greg Neagle's session from MacSysAdmin 2019 and my co-presenter's session from JNUC 2019 as great places to start. For most people, just adopting AutoPkg is a huge gain.

Steps to Update

1. Go to vendor web site
2. Download update
3. Turn into a pkg installer (if necessary)
4. Upload pkg to Jamf Pro Server
5. Add Category Info to uploaded pkg
6. Go to the policy that deploys it
7. Update the pkg info for the policy
8. Trigger as required



JamfUploader

virtual
JAMF
2021

This session will focus on using the group of AutoPkg processors in the JamfUploader project, which leverages the Jamf Pro Classic API to automate some or all of those remaining steps. In fact, JamfUploader can do additional tasks such as uploading scripts, Extension Attributes, and configuration profiles, but to begin with, we are just going to focus on automating the workflow you see on screen.



[Auto-build] This session will look at JamfUploader from two different perspectives. I have a lot of AutoPkg experience but am fairly new to using Jamf Pro for deployment workflows, so I have no history with tools like Munki or JSSImporter. I also deploy exclusively to shared use Mac labs, so I don't use Self Service. 🍏

[Graham:] I have been a heavy user of AutoPkg for a few years now, initially with Munki, and for the past 6 with Jamf Pro. My deployment environment has multiple Jamf Pro instances, mainly to single user computers, with scoping controlled by departmental admins in their own Jamf Pro instance, many of whom choose to use Self Service, though others also auto-deploy like in Anthony's labs.





JamfUploader: A Brief History



[Graham continues] There have been other attempts at using AutoPkg to automate things in Jamf Pro using the Classic API.

Before JamfUploader...

virtual
JNUK
2021

JSSImporter is the most notable. I've used JSSImporter for package deployment since we migrated to Jamf Pro at ETH, and I became its maintainer when its developer Shea Craig moved jobs and stopped using Jamf Pro. So why have I created a new tool to do the same job?

🍏 JSSImporter requires additional python packages to function, 🍏 so JSSImporter must be installed separately to AutoPkg itself. 🍏 It is optimised toward a standard recipe design. You can change certain aspects of the recipes to fit your needs, but 🍏 this gets complex, or sometimes not possible, if you veer too far away from the envisaged standard.

Despite making some changes to JSSImporter, the complex underlying frameworks and the limits to its flexibility have remained a challenge. I wanted to use the power of AutoPkg not just for creating the package testing policies, but to deploy all our policies and scripts. I needed something more flexible and, 🍏 as maintainer, easier to understand.

Before JamfUploader...

JSSImporter

Before JamfUploader...

JSSImporter

- Custom AutoPkg processor based on python-jss framework

Before JamfUploader...

JSSImporter

- Custom AutoPkg processor based on python-jss framework
- Requires installation via package

Before JamfUploader...

JSSImporter

- Custom AutoPkg processor based on python-jss framework
- Requires installation via package
- Standardised templates and recipes

Before JamfUploader...

JSSImporter

- Custom AutoPkg processor based on python-jss framework
- Requires installation via package
- Standardised templates and recipes
- Limited design flexibility

Before JamfUploader...

JSSImporter

- Custom AutoPkg processor based on python-jss framework
- Requires installation via package
- Standardised templates and recipes
- Limited design flexibility
- Lots of tech debt in the code

Enter JamfUploader

virtual
JNUK
2021

AutoPkg's core processors tend to do one specific task, are simple to configure, and can be used as many times as required. Taking inspiration from these, I decided to make a set of smaller, simpler processors to do each of the specific tasks that JSSImporter tries to do in a single processor. 🍏 Collectively, I call these processors JamfUploader, but they are actually 6 different processors, based on what they are intended to upload.

Enter JamfUploader

JamfCategoryUploader

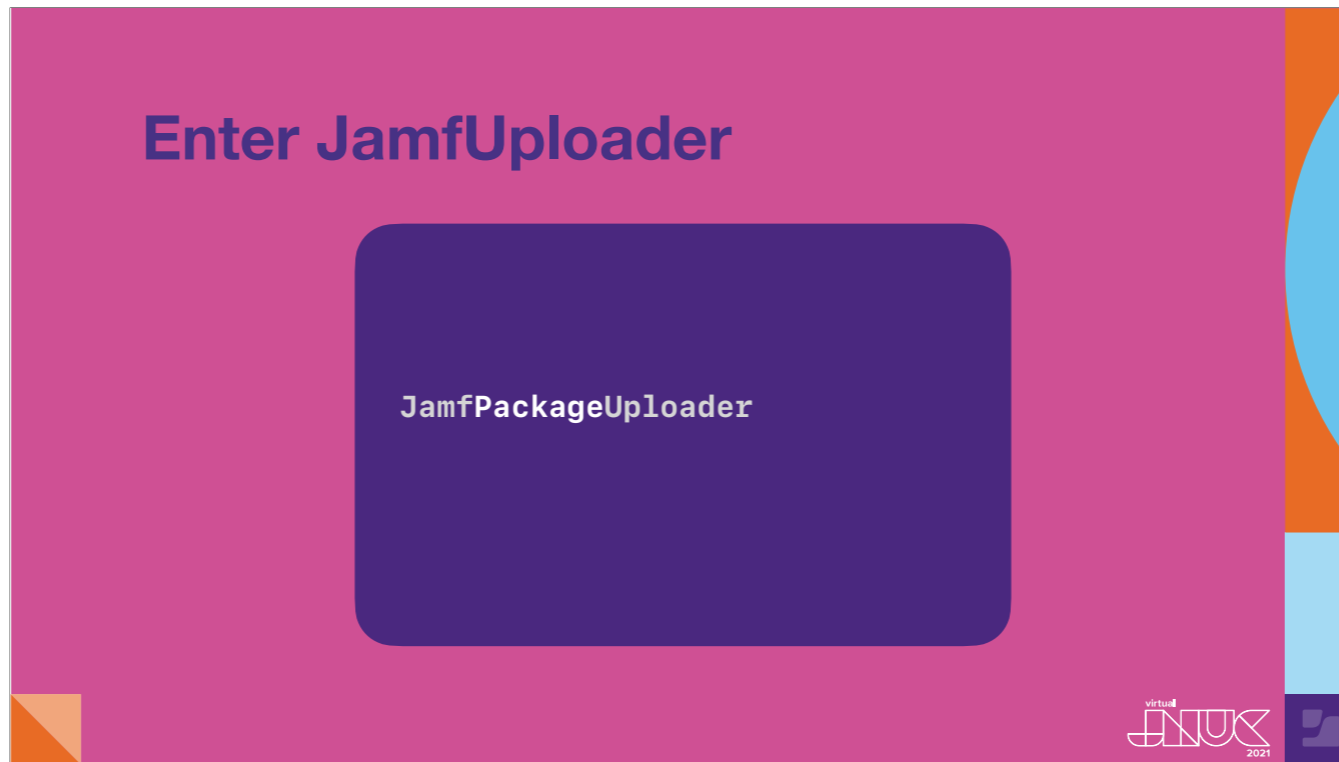
JamfExtensionAttributeUploader

JamfPackageUploader

JamfScriptUploader

JamfComputerGroupUploader

JamfPolicyUploader



You only need to add the processors to your recipe that are required for the items you want to upload. If you just have a package to upload, then just add the JamfPackageUploader processor to your recipe.

Enter JamfUploader

JamfCategoryUploader

JamfPackageUploader

If you want to assign a category to your package, then add the JamfCategoryUploader processor.

Enter JamfUploader

JamfCategoryUploader

JamfPackageUploader

JamfComputerGroupUploader

JamfPolicyUploader

virtual
JNUK
2021

If you want to automate the creation of a policy and smart group with which to push the package to clients, or offer it in Self Service, then use the JamfPolicyUploader and JamfComputerGroupUploader processors.

And so on. You only use the processors you need, and you can use them multiple times in a single recipe.

"Installing" JamfUploader

```
autopkg add-repo grahampugh-recipes
```

Unlike JSSImporter, there's nothing to install - just add my AutoPkg repo to your repo-list as you would to use one of my recipes.

I hope we've piqued your interest in JamfUploader.
Anthony is going to show you how to get started.

Starting Simple: JamfCategoryUploader & JamfPackageUploader

virtual
JNUK
2021

[06:30]
[Anthony] Let's begin by looking at these two processors from the project, which can give you an immediate benefit.

Steps to Update

1. Go to vendor web site
2. Download update
3. Turn into a pkg installer (if necessary)
4. Upload pkg to Jamf Pro Server
5. Add Category Info to uploaded pkg
6. Go to the policy that deploys it
7. Update the pkg info for the policy
8. Trigger as required



As their names imply, they can be used to automate uploading packages and adding categories to Jamf Pro. As with most AutoPkg processors, these processors have some necessary input variables.

JamfPackageUploader – Input

```
pkg_path  
version  
JSS_URL  
API_USERNAME  
API_PASSWORD  
SMB_URL  
SMB_USERNAME  
SMB_PASSWORD
```

Here are the ones for the JamfPackageUploader processor. 🍏 Normally, the package path and version would be supplied by the parent recipe, so you wouldn't usually need to add them manually. 🍏 The processor needs to know how to connect to the Jamf Pro Server. Because the password for this account will be stored in plain text on your system, I recommend setting up a separate AutoPkg user account that has just the privileges you need. Details on which privileges are needed are available in the JamfUploader wiki. 🍏 If you are using an on-prem Jamf Pro Server, you will also need to specify the location of the share. I recommend using the same AutoPkg account. The JamfCategoryUploader processor requires similar information for the Jamf Pro Server.

JamfPackageUploader – Input

pkg_path
version
JSS_URL
API_USERNAME
API_PASSWORD
SMB_URL
SMB_USERNAME
SMB_PASSWORD

} from pkg recipe

JamfPackageUploader – Input

pkg_path

version

JSS_URL

API_USERNAME

API_PASSWORD

SMB_URL

SMB_USERNAME

SMB_PASSWORD

} from pkg recipe

} **Jamf Pro server**
(cloud or on-premises)

JamfPackageUploader – Input

pkg_path	}	from pkg recipe
version		
JSS_URL	}	Jamf Pro server (cloud or on-premises)
API_USERNAME		
API_PASSWORD		
SMB_URL	}	Jamf Pro fileshare (on-premises)
SMB_USERNAME		
SMB_PASSWORD		

```
<key>Input</key>
<dict>
  <key>JSS_URL</key>
  <string>https://yourjamfproserver.org:8443</string>
  <key>API_USERNAME</key>
  <string>autopkguser</string>
  <key>API_PASSWORD</key>
  <string>password</string>
  <key>SMB_URL</key>
  <string>smb://yourjamfproserver.org/jamfproshare</string>
  <key>SMB_USERNAME</key>
  <string>autopkguser</string>
  <key>SMB_PASSWORD</key>
  <string>password</string>
</dict>
```

So how do you specify that Jamf Pro Server info? One way is to add the necessary Input variables to each recipe. Since you are likely to have many packages you wish to upload, this could become quite tedious.

```
defaults write com.github.autopkg JSS_URL  
https://yourjamfproserver.org:8443  
  
defaults write com.github.autopkg API_USERNAME autopkguser  
defaults write com.github.autopkg API_PASSWORD password  
  
defaults write com.github.autopkg SMB_URL  
smb://yourjamfproserver.org/jamfproshare  
  
defaults write com.github.autopkg SMB_USERNAME autopkguser  
defaults write com.github.autopkg SMB_PASSWORD password
```

Another method is to make these variables available whenever AutoPkg runs by adding them to AutoPkg's preferences. The *defaults write* command is your friend here. All six of these commands are one-liners. As mentioned earlier, if your Jamf Pro Server is in the cloud, you only need to set the first three variables. Every example we provide from here on out will assume that these values have already been set.

Using JamfPackageUploader as a Post-processor

```
autopkg run FirefoxSignedPkg.pkg  
--post=com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
```

If you are used to running AutoPkg from the command line, you can actually use the JamfPackageUploader processor as a post-processor to any pkg recipe and it will upload any new package to the Jamf Pro Server. Just use the --post flag. You'll see here that we are using the syntax for shared processors, since JamfUploader is not part of the core AutoPkg repo. Again, this command is all on one line. JamfPackageUploader also lets you attach a Category to the package, which you can specify using the -k or --key flag. 🍏 The only caveat is that the category must already exist if you run it this way. Note that the key flag is a general AutoPkg option for the run verb, which can be handy if you need to supply an input variable for a recipe run on a one-off basis. While this works perfectly well, I suspect most of you will want to use an AutoPkg recipe to do this work. Let's look at how that would be done.

Using JamfPackageUploader as a Post-processor

```
autopkg run FirefoxSignedPkg.pkg  
--post=com.github.grahampugh.jamf-upload.processors/JamfPackageUploader  
--key pkg_category=Browsers
```

Example -pkg-upload recipes

[https://github.com/autopkg/
graahmpugh-recipes/tree/master/
Jamf_Package_Only_Recipes](https://github.com/autopkg/graahmpugh-recipes/tree/master/Jamf_Package_Only_Recipes)

Graham has shared a couple of recipes that you can use as a model in his recipes repo in the AutoPkg project. I used those as a template to create the next recipe I am going to show you, one that uploads the Firefox signed package installer to a Jamf Pro Server.

```
<?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>Description</key>
  <string>This recipe downloads the signed installer package from Mozilla
and then uploads the pkg to your Jamf Pro Server/Distribution Point using
variables set in the environment.
The grahamugh-recipes repo is required.
</string>
  <key>Identifier</key>
  <string>com.github.jazzace.jamf.FirefoxSignedPkg-pkg-upload</string>
  <key>Input</key>
  <dict>
    <key>CATEGORY</key>
    <string>Browsers</string>
  </dict>
  <key>MinimumVersion</key>
  <string>2.0</string>
  <key>ParentRecipe</key>
  <string>com.github.autopkg.download.FirefoxSignedPkg</string>
</dict>
</plist>
</xml>
```

FirefoxSignedPkg-pkg-upload.jamf.recipe



Let's take a look at the top half of the recipe. The description describes what we are going to do, but also points out that it expects the Jamf Pro Server variables to be set and notes that you need the grahamugh-recipes repo. The identifier should have your Github username or other unique identifier and should specify that this is a pkg upload recipe. The current convention is to add -pkg-upload to the recipe name and identifier. The only Input key we need is for the Category we want to assign to the pkg in the Jamf Pro Server. At the bottom of the screen, you will see the parent recipe from the core recipes repo.

```
<key>Process</key>
<array>
  <dict>
    <key>Processor</key>
    <string>com.github.grahampugh.jamf-upload.processors/
JamfCategoryUploader</string>
    <key>Arguments</key>
    <dict>
      <key>category</key>
      <string>%CATEGORY%</string>
    </dict>
  </dict>
  <dict>
    <key>Processor</key>
    <string>com.github.grahampugh.jamf-upload.processors/
JamfPackageUploader</string>
    <key>Arguments</key>
    <dict>
      <key>pkg_category</key>
      <string>%CATEGORY%</string>
    </dict>
  </dict>
</array>
```

FirefoxSignedPkg-pkg-upload.jamf.recipe

virtual JNUK 2021

This is followed by two simple processor steps. The first creates that category in Jamf Pro for the package we are about to upload and the second uploads the package. Note that we use the same identifier / processor name syntax as we did with the post-processor option so that AutoPkg can find the JamfUploader processors. That is the entire child recipe. You can now override it in preparation for running it in production.

```
Description: |
  This recipe downloads the signed installer package from Mozilla and then
  uploads the pkg to your Jamf Pro Server/Distribution Point using variables
  set in the environment.
  The grahampugh-recipes repo is required.
Identifier: com.github.jazzace.jamf.FirefoxSignedPkg-pkg-upload
ParentRecipe: com.github.autopkg.download.FirefoxSignedPkg
MinimumVersion: '2.3'

Input:
  CATEGORY: Browsers

Process:
- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
  Arguments:
    category_name: '%CATEGORY%'

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
  Arguments:
    pkg_category: '%CATEGORY%'
```

FirefoxSignedPkg-pkg-upload.jamf.recipe.yaml



As of version 2.3 of AutoPkg, you can also write your recipes in YAML, short for Yet Another Markup Language. Just add .yaml to the filename and AutoPkg will know how to handle it. Here is what the recipe we just wrote looks like in YAML. Because it's a lot easier to read and we can fit more info onto one screen, this is what we will use for the rest of our recipe-writing examples.

Steps to Update

1. Go to vendor web site
2. Download update
3. Turn into a pkg installer (if necessary)
4. Upload pkg to Jamf Pro Server
5. Add Category Info to uploaded pkg
6. Go to the policy that deploys it
7. Update the pkg info for the policy
8. Trigger as required

} **AutoPkg**
(-pkg-upload.jamf recipe)

} **Manual**

So just by doing that little bit of work, we already have profit! Doing this was a great way for me to get started with the JamfUploader project. Even with manual work remaining in my workflow, my workflows were already much more efficient.

```
Description: |
  This recipe downloads the signed installer package from Mozilla and then
  uploads the pkg to your Jamf Pro Server/Distribution Point using variables
  set in the environment.
  The grahampugh-recipes repo is required.
Identifier: com.github.jazzace.jamf.FirefoxSignedPkg-pkg-upload
ParentRecipe: com.github.autopkg.download.FirefoxSignedPkg
MinimumVersion: '2.3'

Input:
  CATEGORY: Browsers

Process:
- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
  Arguments:
    category_name: '%CATEGORY%'

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
  Arguments:
    pkg_category: '%CATEGORY%'
```

FirefoxSignedPkg-pkg-upload.jamf.recipe.yaml



As Graham mentioned earlier, the JamfCategoryUploader processor will create any category that does not already exist on the Jamf Pro Server. But what if you want to limit yourself to categories that already exist?

```
Description: |
  This recipe downloads the signed installer package from Mozilla and then
  uploads the pkg to your Jamf Pro Server/Distribution Point using variables
  set in the environment.
  The grahampugh-recipes repo is required.
Identifier: com.github.jazzace.jamf.FirefoxSignedPkg-pkg-upload
ParentRecipe: com.github.autopkg.download.FirefoxSignedPkg
MinimumVersion: '2.3'

Input:
  CATEGORY: Browsers

Process:
- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
  Arguments:
    pkg_category: '%CATEGORY%'
```

FirefoxSignedPkg-pkg-upload.jamf.recipe.yaml

virtual JNUK 2021

You can choose to completely omit the `JamfCategoryUploader` processor step. The `JamfPackageUploader` will assign the category without issue if it exists but will cause a failure if it doesn't. This is actually what I want for my personal workflows, so my `-pkg-upload` recipes only need one processor step. Simple!

Adding a Policy using JamfPolicyUploader

virtual
JNUK
2021

[12:00]

But what if you want to add or update a policy for that shiny new package you now have in Jamf Pro? The JamfUploader project can help here, too.

Steps to Update

1. Go to vendor web site
2. Download update
3. Turn into a pkg installer (if necessary)
4. Upload pkg to Jamf Pro Server
5. Add Category Info to uploaded pkg
6. Go to the policy that deploys it
7. Update the pkg info for the policy
8. Trigger as required



The `JamfPolicyUploader` processor can take care of Steps 6 and 7, or could even be used to create a *new* policy which would automatically be triggered. Most of the heavy lifting is done using an XML template. There are a couple of template examples in Graham's repo that you can learn from, but we're going to build one from scratch so that you can include the options you want.

My Typical Policy

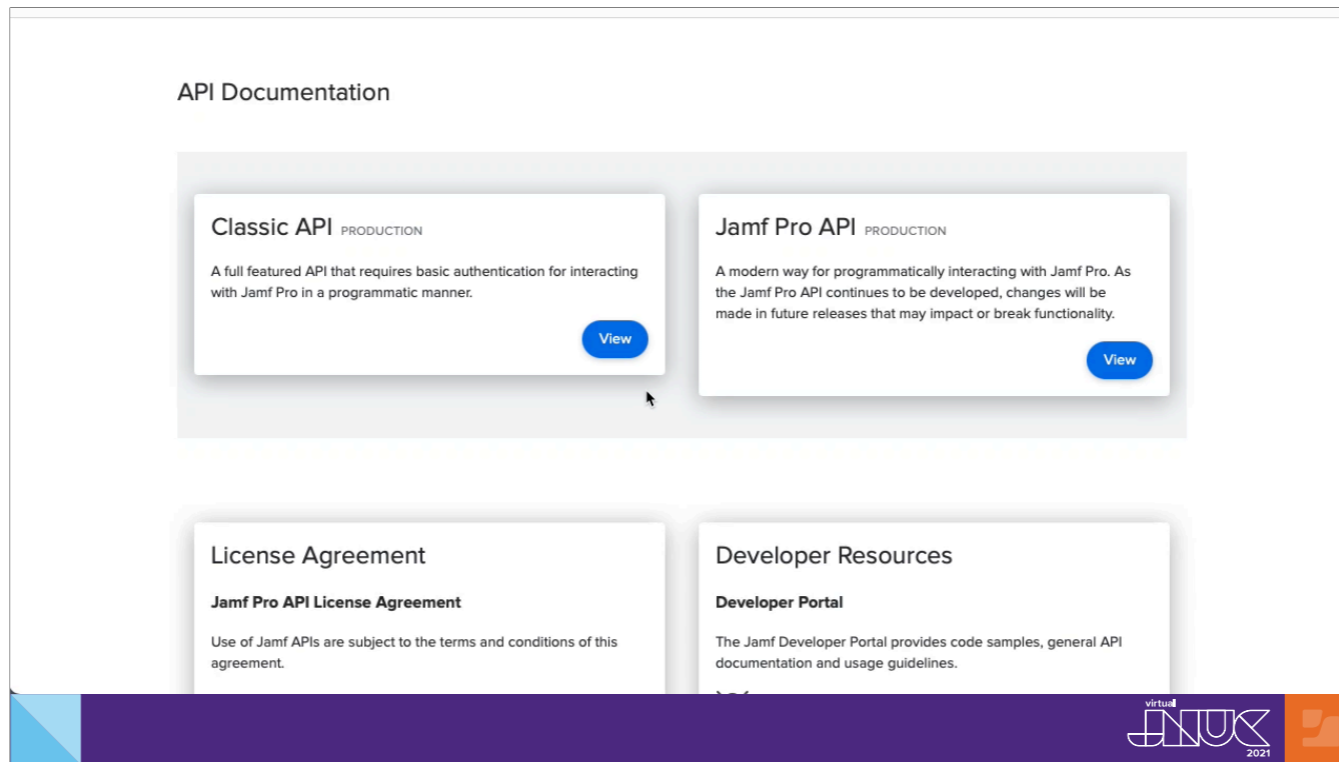
<i>Category</i>	<i>Varies</i>
<i>Triggers</i>	Enrollment Complete Recurring Check-in
<i>Frequency</i>	Once per computer
<i>Scope</i>	All computers
<i>Self Service</i>	No
<i>Payload</i>	<i>Single Package</i>

Here are the most common attributes in the policies I write, since I am deploying to shared lab computers. I'm going to take an existing policy with these attributes and use this as the basis of my template.

<https://yourjamfproserver.org/api/>



I do this by downloading the policy as an XML file using the Jamf Pro Classic API. Just take the address of your Jamf Pro server and add /api to get there.



[Video Auto-plays][Narrate demo video]

As you saw when we scrolled through the policy in XML form, it had a lot of stuff that was just a bunch of self-closing tags or defaults that don't apply. We can get rid of those bits. We also need to get rid of any tags that give an ID number, since that is something that Jamf Pro will take care of for us when we use the API to create the policy. That leaves us with something like this:

API Documentation

Classic API PRODUCTION

A full featured API that requires basic authentication for interacting with Jamf Pro in a programmatic manner.

[View](#)

Jamf Pro API PRODUCTION

A modern way for programmatically interacting with Jamf Pro. As the Jamf Pro API continues to be developed, changes will be made in future releases that may impact or break functionality.

[View](#)

License Agreement

Jamf Pro API License Agreement


Use of Jamf APIs are subject to the terms and conditions of this agreement.

Developer Resources

Developer Portal

The Jamf Developer Portal provides code samples, general API documentation and usage guidelines.

```
<?xml version="1.0" encoding="utf-8"?>
<policy>
  <general>
    <name>Policy Template</name>
    <enabled>true</enabled>
    <frequency>Once per computer</frequency>
    <trigger_checkin>true</trigger_checkin>
    <trigger_enrollment_complete>true</trigger_enrollment_complete>
    <category>
      <name>Testing</name>
    </category>
    <site>
      <name>None</name>
    </site>
  </general>
  <scope>
    <all_computers>true</all_computers>
  </scope>
  <package_configuration>
    <packages>
      <size>1</size>
      <package>
        <name>App Name.pkg</name>
        <action>Install</action>
      </package>
    </packages>
  </package_configuration>
  <scripts>
    <size>0</size>
  </scripts>
  <self_service>
    <use_for_self_service>>false</use_for_self_service>
  </self_service>
</policy>
```



[Pause to let viewer look at the screen for a moment.] This is all the critical information needed to create this policy. But if the goal is to create a reusable template, 🍏 our policy will not really be called Policy Template, nor will it always use the same category, nor will the package payload have the same name. This is where we put the power of AutoPkg to work with variable substitution.

```
<?xml version="1.0" encoding="utf-8"?>
<policy>
  <general>
    <name>Policy Template</name>
    <enabled>true</enabled>
    <frequency>Once per computer</frequency>
    <trigger_checkin>true</trigger_checkin>
    <trigger_enrollment_complete>true</trigger_enrollment_complete>
    <category>
      <name>Testing</name>
    </category>
    <site>
      <name>None</name>
    </site>
  </general>
  <scope>
    <all_computers>true</all_computers>
  </scope>
  <package_configuration>
    <packages>
      <size>1</size>
      <package>
        <name>App_Name.pkg</name>
        <action>Install</action>
      </package>
    </packages>
  </package_configuration>
  <scripts>
    <size>0</size>
  </scripts>
  <self_service>
    <use_for_self_service>>false</use_for_self_service>
  </self_service>
</policy>
```



```
<?xml version="1.0" encoding="utf-8"?>
<policy>
  <general>
    <name>%POLICY_NAME%</name>
    <enabled>true</enabled>
    <frequency>Once per computer</frequency>
    <trigger_checkin>true</trigger_checkin>
    <trigger_enrollment_complete>true</trigger_enrollment_complete>
    <category>
      <name>%POLICY_CATEGORY%</name>
    </category>
    <site>
      <name>None</name>
    </site>
  </general>
  <scope>
    <all_computers>true</all_computers>
  </scope>
  <package_configuration>
    <packages>
      <size>1</size>
      <package>
        <name>%pkg_name%</name>
        <action>Install</action>
      </package>
    </packages>
  </package_configuration>
  <scripts>
    <size>0</size>
  </scripts>
  <self_service>
    <use_for_self_service>>false</use_for_self_service>
  </self_service>
</policy>
```

All-Computers-Policy-Template.xml

virtual JNUK 2021

pkg_name is supplied automatically through the parent .pkg recipe, while the others are supplied as input variables. You will probably have to do some trial and error to get your template perfect, and you may need more than one template; Graham is going to go into much greater depth about that in a few minutes. So now that we have a policy template, let's take the recipe we wrote earlier for Firefox and add the JamfPolicyUploader processor step.

```
Input:
NAME: Firefox
CATEGORY: Testing
POLICY_CATEGORY: Testing
POLICY_NAME: "Install Latest %NAME%"
POLICY_TEMPLATE: All-Computers-Policy-Template.xml


Process:
- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
Arguments:
category_name: "%CATEGORY%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
Arguments:
pkg_category: "%CATEGORY%"

- Processor: StopProcessingIf
Arguments:
predicate: "pkg_uploaded == False"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
Arguments:
policy_name: "%POLICY_NAME%"
policy_template: "%POLICY_TEMPLATE%"
replace_policy: "True"
```

FirefoxSignedPkg.jamf.recipe.yaml



Here's what the Input and Process sections look like.

We add input variables related to the policy we are creating, including the name of the policy template. Just as with the previous recipe, we upload category and package information with the first two processors.

Next is StopProcessingIf. That is a core AutoPkg processor, which checks if a particular event happened or not in the previous processor, and stops the recipe from proceeding if the predicate matches. The JamfPackageUploader processor outputs a true or false value depending on whether a package was uploaded, so we check against that value. This allows us to stop the recipe if there's no new package, which speeds up our workflow, and prevents us from overwriting a policy erroneously. We finish with JamfPolicyUploader, which builds the policy in Jamf Pro from our template and input variables.

In my situation, this is all I need to automate most of what I need to do to deploy apps in Jamf Pro. Graham has other, more complex use cases that can also benefit from the JamfUploader project processors, so let me hand the rest of the presentation over to him.

Putting multiple processors into a single recipe

virtual
JNUK
2021

[17:00]

[Graham] As Anthony has demonstrated, your recipe only need the processors for each specific object you want to upload into Jamf.

In this section, I'll show you recipes which create dynamically scoped self service policies, allowing you to fully automate your package deployment workflow.

Jamf recipe design - processor order

Processors need to be added to recipes in the order that items need to be created.

🍏 Categories and extension attributes can be created first because these do not depend on anything else.

🍏 Packages and scripts may have an assigned category, so should come after category.

🍏 Computer Groups may include extension attributes and other computer groups, so should come after EAs, and if you are creating multiple groups, you have to consider the order based on any group dependencies in the criteria.

🍏 Policies often depend on all of the above, so should go last.

Jamf recipe design - processor order

JamfCategoryUploader
JamfExtensionAttributeUploader

Jamf recipe design - processor order

JamfCategoryUploader
JamfExtensionAttributeUploader
JamfPackageUploader
JamfScriptUploader

Jamf recipe design - processor order

JamfCategoryUploader
JamfExtensionAttributeUploader
JamfPackageUploader
JamfScriptUploader
JamfComputerGroupUploader

Jamf recipe design - processor order

JamfCategoryUploader
JamfExtensionAttributeUploader
JamfPackageUploader
JamfScriptUploader
JamfComputerGroupUploader
JamfPolicyUploader

Jamf recipe design - required files

virtual
JNUK
2021

Any jamf recipe which includes a policy and a specific scope, will need at least 3 files. 🍏 That is, the recipe itself, 🍏 an XML template file to populate the policy, and 🍏 at least one smart group template file.

Jamf recipe design - required files

- `Firefox.jamf.recipe.yaml`

Jamf recipe design - required files

- `Firefox.jamf.recipe.yaml`
- `PolicyTemplate.xml`

Jamf recipe design - required files

- `Firefox.jamf.recipe.yaml`
- `PolicyTemplate.xml`
- `SmartGroupTemplate.xml`

Software Testing Policy

<i>Category</i>	<i>Varies</i>
<i>Triggers</i>	Recurring Check-in
<i>Frequency</i>	Ongoing
<i>Scope</i>	"Testing" static group - <i>and</i> - Current version of Firefox not installed
<i>Self Service</i>	Yes
<i>Payload</i>	<i>Single Package</i>

Here, we are going to construct a recipe that creates a Self Service policy scoped to any computer in a static group named Testing, which does not already have the version we are uploading installed. This dynamic method of scoping means that you don't need to flush policies, because as a new version of Firefox is released, the computer will fall back into scope, because it still has the older version.

Let's look at the recipe and templates required.

```
Description: |
  Downloads the latest version of Firefox and makes a pkg. Then, uploads the
  package to the Jamf Pro Server and creates a Self Service Policy and Smart
  Group.
Identifier: com.github.grahampugh.recipes.jamf.Firefox
MinimumVersion: "2.3"
ParentRecipe: com.github.autopkg.pkg.Firefox_EN

Input:
  NAME: Firefox
  CATEGORY: Productivity
  GROUP_NAME: "%NAME%-update-smart"
  GROUP_TEMPLATE: SmartGroup-update-smart.xml
  VERSION_CRITERION: Application Version
  TESTING_GROUP_NAME: Testing
  POLICY_CATEGORY: Testing
  POLICY_TEMPLATE: Policy-install-latest.xml
  POLICY_NAME: "Install Latest %NAME%"
  POLICY_RUN_COMMAND: 'chown -R "$(stat -f%Su /dev/console):staff" "/
Applications/%NAME%.app" && echo "Corrected permissions for %NAME%."'
  SELF_SERVICE_DISPLAY_NAME: "Install Latest %NAME%"
  SELF_SERVICE_DESCRIPTION: Mozilla Firefox is a free and open source web
  browser.
  SELF_SERVICE_ICON: "%NAME%.png"
  INSTALL_BUTTON_TEXT: "Install %version%"
  REINSTALL_BUTTON_TEXT: "Install %version%"
  UPDATE_PREDICATE: "ake_uploaded == false"
```

Here is the recipe.

🍏 The Input list has grown considerably, because we need to supply values for the smart group name and criteria, self service details such as description and icon. This looks complex, but you'll find that there are few differences between recipes of standard applications, so creating a new recipe is normally easily done by duplicating an existing one and changing the Name and Category.

🍏 In the process list, the JamfComputerGroupUploader processor needs to be inserted before the policy.

ParentRecipe: com.github.autopkg.pkg.Firefox_EN

Input:

```
NAME: Firefox
CATEGORY: Productivity
GROUP_NAME: "%NAME%-update-smart"
GROUP_TEMPLATE: SmartGroup-update-smart.xml
VERSION_CRITERION: Application Version
TESTING_GROUP_NAME: Testing
POLICY_CATEGORY: Testing
POLICY_TEMPLATE: Policy-install-latest.xml
POLICY_NAME: "Install Latest %NAME%"
POLICY_RUN_COMMAND: 'chown -R "$(stat -f%Su /dev/console):staff" "/
Applications/%NAME%.app" && echo "Corrected permissions for %NAME%."'
SELF_SERVICE_DISPLAY_NAME: "Install Latest %NAME%"
SELF_SERVICE_DESCRIPTION: Mozilla Firefox is a free and open source web
browser.
SELF_SERVICE_ICON: "%NAME%.png"
INSTALL_BUTTON_TEXT: "Install %version%"
REINSTALL_BUTTON_TEXT: "Install %version%"
UPDATE_PREDICATE: "pkg_uploaded == False"
```

Process:

```
- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
Arguments:
  category_name: "%CATEGORY%"
```

```
- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
```



```
UPDATE_PREDICATE: "pkg_uploaded == false"
```

Process:

- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
Arguments:
category_name: "%CATEGORY%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader

- Processor: StopProcessingIf
Arguments:
predicate: "%UPDATE_PREDICATE%"

- Processor: com.github.grahampugh.jamf-upload.processors/
JamfComputerGroupUploader
Arguments:
computergroup_template: "%GROUP_TEMPLATE%"
computergroup_name: "%GROUP_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
Arguments:
policy_template: "%POLICY_TEMPLATE%"
policy_name: "%POLICY_NAME%"
icon: "%SELF_SERVICE_ICON%"


```
<?xml version="1.0" encoding="UTF-8"?>
<policy>
  <general>
    <name>%POLICY_NAME%/name>
    <enabled>>true</enabled>
    <frequency>Ongoing</frequency>
    <category>
      <name>%POLICY_CATEGORY%/name>
    </category>
  </general>
  <scope>
    <computer_groups>
      <computer_group>
        <name>%GROUP_NAME%/name>
      </computer_group>
    </computer_groups>
    <exclusions>
      <computers/>
      <computer_groups/>
    </exclusions>
  </scope>
  <package_configuration>
    <packages>
      <size>1</size>
      <package>
        <name>%pkg_name%/name>
        <action>Install</action>
```



Here is our new Policy Template. The template is longer than Anthony's example, because it includes a scope, 🍏 the Self Service details, and because in this template we have provided the ability to define a policy run command.

```
</scripts>
<self_service>
  <use_for_self_service>true</use_for_self_service>
  <install_button_text>%SELF_SERVICE_INSTALL_BUTTON%</install_button_text>
  <reinstall_button_text>%SELF_SERVICE_REINSTALL_BUTTON%</
reinstall_button_text>
  <self_service_display_name>%SELF_SERVICE_POLICY_NAME%</
self_service_display_name>
  <self_service_description>%SELF_SERVICE_DESCRIPTION%</
self_service_description>
</self_service>
<files_processes>
  <search_by_path/>
  <delete_file>>false</delete_file>
  <locate_file/>
  <update_locate_database>>false</update_locate_database>
  <spotlight_search/>
  <search_for_process/>
  <kill_process>>false</kill_process>
  <run_command>%POLICY_RUN_COMMAND%</run_command>
</files_processes>
<maintenance>
  <recon>>true</recon>
</maintenance>
</policy>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<computer_group>
  <name>%GROUP_NAME%</name>
  <is_smart>true</is_smart>
  <criteria>
    <criteria>
      <name>Application Title</name>
      <priority>0</priority>
      <and_or>and</and_or>
      <search_type>is</search_type>
      <value>%JSS_INVENTORY_NAME%</value>
      <opening_paren>true</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>%VERSION_CRITERION%</name>
      <priority>1</priority>
      <and_or>and</and_or>
      <search_type>is not</search_type>
      <value>%version%</value>
      <opening_paren>>false</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>Application Title</name>
      <priority>2</priority>
      <and_or>or</and_or>
```

And here is the XML smart group template file, which defines that any computer is in scope if it either does not have Firefox installed at all, or does not have the current version. 🍏 JSS_INVENTORY_NAME is a special key name in JamfPolicyUploader processor that I borrowed from JSSImporter. By default it adds "dot-app" to the value of the NAME variable, for example Firefox.app. But you can specify an alternative in the recipe's Input list if necessary. 🍏 I also set the criterion that is normally "Application Version" into a variable, so that if we need to use an extension attribute to determine the version, we can still use the same template, but you could use a different template instead.

```
<?xml version="1.0" encoding="UTF-8"?>
<computer_group>
  <name>%GROUP_NAME%</name>
  <is_smart>true</is_smart>
  <criteria>
    <criteria>
      <name>Application Title</name>
      <priority>0</priority>
      <and_or>and</and_or>
      <search_type>is</search_type>
      <value>%JSS_INVENTORY_NAME%</value>
      <opening_paren>true</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>%VERSION_CRITERION%</name>
      <priority>1</priority>
      <and_or>and</and_or>
      <search_type>is not</search_type>
      <value>%version%</value>
      <opening_paren>>false</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>Application Title</name>
      <priority>2</priority>
      <and_or>or</and_or>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<computer_group>
  <name>%GROUP_NAME%</name>
  <is_smart>true</is_smart>
  <criteria>
    <criteria>
      <name>Application Title</name>
      <priority>0</priority>
      <and_or>and</and_or>
      <search_type>is</search_type>
      <value>%JSS_INVENTORY_NAME%</value>
      <opening_paren>true</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>%VERSION_CRITERION%</name>
      <priority>1</priority>
      <and_or>and</and_or>
      <search_type>is not</search_type>
      <value>%version%</value>
      <opening_paren>>false</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>Application Title</name>
      <priority>2</priority>
      <and_or>or</and_or>
```

```
<search_type>is not</search_type>
<value>%version%</value>
<opening_paren>>false</opening_paren>
<closing_paren>>false</closing_paren>
</criterion>
<criterion>
  <name>Application Title</name>
  <priority>2</priority>
  <and_or>or</and_or>
  <search_type>is not</search_type>
  <value>%JSS_INVENTORY_NAME%</value>
  <opening_paren>>false</opening_paren>
  <closing_paren>>true</closing_paren>
</criterion>
<criterion>
  <name>Computer Group</name>
  <priority>3</priority>
  <and_or>and</and_or>
  <search_type>member of</search_type>
  <value>%TESTING_GROUP_NAME%</value>
  <opening_paren>>false</opening_paren>
  <closing_paren>>false</closing_paren>
</criterion>
</criteria>
</computer_group>
```

Software Testing Policy with script

<i>Category</i>	<i>Varies</i>
<i>Triggers</i>	Recurring Check-in
<i>Frequency</i>	Ongoing
<i>Scope</i>	"Testing" static group - <i>and</i> - Current version of Word not installed
<i>Self Service</i>	Yes
<i>Payload</i>	<i>Single Package</i> <i>Single Script (priority: before)</i>

Some recipes need a script to be added to the policy, because some action is required before or after the installation of the app. For this, we need to change the recipe and the policy template.

Let's take a look at such an example - for my Microsoft Office 365 recipe, I need a preinstall script.

```
Description: Downloads the latest version of Microsoft Office 365 and makes a
pkg. Then, uploads the package to the Jamf Pro Server and creates a Self Service
Policy and Smart Group.
Identifier: com.github.grahampugh.recipes.jamf.MicrosoftOffice365
MinimumVersion: "2.3"
ParentRecipe: com.github.grahampugh.recipes.pkg.MicrosoftOffice365

Input:
NAME: Microsoft Office 365
JSS_INVENTORY_NAME: Microsoft Word.app
INSTALLED_REGEX_MATCH: '^(16\.1[7-9]\.|16\.[2-9]\d\.|17\.)'
OS_EXCLUDE_MIN: 10.0.0
OS_EXCLUDE_MAX: 10.13.6
OS_LIMITS_GROUP_NAME: "macOS %OS_EXCLUDE_MAX% or less"
EXCLUSION_GROUP_TEMPLATE: SmartGroup-OSVersionLimits.xml
CATEGORY: Productivity
GROUP_NAME: "%NAME%-update-smart"
GROUP_TEMPLATE: SmartGroup-update-smart-MicrosoftOffice365.xml
TESTING_GROUP_NAME: Testing
POLICY_CATEGORY: Testing
POLICY_TEMPLATE: Policy-install-latest-MicrosoftOffice365.xml
POLICY_NAME: "Install Latest %NAME%"
SCRIPT_NAME: Microsoft Office License Removal Tool.sh
SCRIPT_PRIORITY: Before
PARAMETER4_LABEL: "--All or --0365 or --Volume"
PARAMETER5_LABEL: "--ForceClose"
PARAMETER6_LABEL: "--jamfUser"
```



🍏 In the recipe for Microsoft Office 365, we set the script name and priority, and we have three script parameters so we provide the labels and values for these. 🍏 In the process list, we have the JamfScriptUploader processor in the recipe to set the default script category, priority, and the parameter labels.


```
INSTALLED_REGEX_MATCH: '^(16\.1[7-9]\.|16\.[2-9]\d\.|17\.)'  
OS_EXCLUDE_MIN: 10.0.0  
OS_EXCLUDE_MAX: 10.13.6  
OS_LIMITS_GROUP_NAME: "macOS %OS_EXCLUDE_MAX% or less"  
EXCLUSION_GROUP_TEMPLATE: SmartGroup-OSVersionLimits.xml  
CATEGORY: Productivity  
GROUP_NAME: "%NAME%-update-smart"  
GROUP_TEMPLATE: SmartGroup-update-smart-MicrosoftOffice365.xml  
TESTING_GROUP_NAME: Testing  
POLICY_CATEGORY: Testing  
POLICY_TEMPLATE: Policy-install-latest-MicrosoftOffice365.xml  
POLICY_NAME: "Install Latest %NAME%"  
SCRIPT_NAME: Microsoft Office License Removal Tool.sh  
SCRIPT_PRIORITY: Before  
PARAMETER4_LABEL: "--All or --0365 or --Volume"  
PARAMETER5_LABEL: "--ForceClose"  
PARAMETER6_LABEL: "--jamfUser"  
PARAMETER4_VALUE: "--Volume"  
PARAMETER5_VALUE: "--ForceClose"  
PARAMETER6_VALUE: "--jamfUser"  
SELF_SERVICE_DISPLAY_NAME: "Install Latest %NAME%"  
SELF_SERVICE_DESCRIPTION: |  
    Microsoft Office 365 includes the following applications:  
    - Microsoft Word  
    - Microsoft Excel  
    - Microsoft PowerPoint  
    - Microsoft Outlook  
    - Microsoft OneNote
```

```
Process:
- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
  Arguments:
    category_name: "%CATEGORY%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader

- Processor: StopProcessingIf
  Arguments:
    predicate: "%UPDATE_PREDICATE%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfScriptUploader
  Arguments:
    script_category: "%CATEGORY%"
    script_path: "%SCRIPT_NAME%"
    script_priority: "%SCRIPT_PRIORITY%"
    script_parameter4: "%PARAMETER4_LABEL%"
    script_parameter5: "%PARAMETER5_LABEL%"
    script_parameter6: "%PARAMETER6_LABEL%"

- Processor: com.github.grahampugh.recipes.commonprocessors/
  VersionRegexGenerator

- Processor: com.github.grahampugh.jamf-upload.processors/
  JamfComputerGroupUploader
  Arguments:
    computergroup_template: "%GROUP_TEMPLATE%"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<policy>
  <general>
    <name>%POLICY_NAME%/name>
    <enabled>>true</enabled>
    <frequency>Ongoing</frequency>
    <category>
      <name>%CATEGORY%/name>
    </category>
  </general>
  <scope>
    <computer_groups>
      <computer_group>
        <name>%GROUP_NAME%/name>
      </computer_group>
    </computer_groups>
    <exclusions>
      <computers/>
      <computer_groups>
        <computer_group>
          <name>%EXCLUSION_GROUP_NAME%/name>
        </computer_group>
      </computer_groups>
    </exclusions>
  </scope>
  <package_configuration>
    <packages>
```

In our policy template, 🍏 we add Input variables for script name, priority and values of Parameters 4, 5 and 6.

Note that any variable added to a template or a process **do** need to be set in the Input list, even if set as explicitly blank. If the variable name is not defined at all, the recipe will fail.

```
<size>1</size>
<package>
  <name>%pkg_name%</name>
  <action>Install</action>
</package>
</packages>
</package configuration>
<scripts>
  <size>1</size>
  <script>
    <name>%SCRIPT_NAME%</name>
    <priority>%SCRIPT_PRIORITY%</priority>
    <parameter4>%PARAMETER4_VALUE%</parameter4>
    <parameter5>%PARAMETER5_VALUE%</parameter5>
    <parameter6>%PARAMETER6_VALUE%</parameter6>
    <parameter7/>
    <parameter8/>
    <parameter9/>
    <parameter10/>
    <parameter11/>
  </script>
</scripts>
<self_service>
  <use_for_self_service>>true</use_for_self_service>
  <install_button_text>%SELF_SERVICE_INSTALL_BUTTON%</install_button_text>
```

Software Testing Policy with EA

<i>Category</i>	<i>Varies</i>
<i>Triggers</i>	Recurring Check-in
<i>Frequency</i>	Ongoing
<i>Scope</i>	"Testing" static group - <i>and</i> - Current version of Edge not installed
<i>Self Service</i>	Yes
<i>Payload</i>	<i>Extension Attribute</i> <i>Single Package</i>

We might also need to use an Extension Attribute to define the scope, if the version string being determined by AutoPkg is different to the one being collected by Jamf's inventory.

```
Description: Downloads the latest version and makes a pkg. Then, uploads the
package to the Jamf Pro Server and creates a Self Service Policy and Smart
Group.
Identifier: com.github.grahampugh.recipes.jamf.MicrosoftEdge
MinimumVersion: "2.3"
ParentRecipe: com.github.rtrouton.pkg.microsoftedge

Input:
NAME: Microsoft Edge
CATEGORY: Productivity
GROUP_NAME: "%NAME%-update-smart"
GROUP_TEMPLATE: SmartGroup-update-smart-EA-regex.xml
EXTENSION_ATTRIBUTE_NAME: "%NAME% Version"
EXTENSION_ATTRIBUTE_SCRIPT: ExtensionAttribute-CFBundleShortVersionString.sh
TESTING_GROUP_NAME: Testing
POLICY_CATEGORY: Testing
POLICY_TEMPLATE: Policy-install-latest.xml
POLICY_NAME: "Install Latest %NAME%"
POLICY_RUN_COMMAND: "echo 'Installation of %NAME% complete'"
SELF_SERVICE_DISPLAY_NAME: "Install Latest %NAME%"
SELF_SERVICE_DESCRIPTION: Microsoft Edge is a fast, simple, and secure web
browser, built for the modern web.
SELF_SERVICE_ICON: "%NAME%.png"
INSTALL_BUTTON_TEXT: "Install %version%"
REINSTALL_BUTTON_TEXT: "Install %version%"
UPDATE_PREDICATE: "pkg_uploaded == False"
```



In this recipe for Microsoft Edge, we have added keys for an extension attribute script called ExtensionAttribute-CFBundleShortVersionString. There is a key for the name we want the EA to have in Jamf Pro, and a key for the path to the EA script itself. The second key is only necessary if we want to automate the upload of the EA script rather than manually add it in the Jamf console beforehand.

🍏 In this recipe, we do want to upload the script, so we have added the JamfExtensionAttributeUploader process to this recipe to do so.

Description: Downloads the latest version and makes a pkg. Then, uploads the package to the Jamf Pro Server and creates a Self Service Policy and Smart Group.

Identifier: com.github.grahampugh.recipes.jamf.MicrosoftEdge

MinimumVersion: "2.3"

ParentRecipe: com.github.rtrouton.pkg.microsoftedge

Input:

NAME: Microsoft Edge

CATEGORY: Productivity

GROUP_NAME: "%NAME%-update-smart"

GROUP_TEMPLATE: SmartGroup-update-smart-FA-regex.xml

EXTENSION_ATTRIBUTE_NAME: "%NAME% Version"

EXTENSION_ATTRIBUTE_SCRIPT: ExtensionAttribute-CFBundleShortVersionString.sh

TESTING_GROUP_NAME: Testing

POLICY_CATEGORY: Testing

POLICY_TEMPLATE: Policy-install-latest.xml

POLICY_NAME: "Install Latest %NAME%"

POLICY_RUN_COMMAND: "echo 'Installation of %NAME% complete'"

SELF_SERVICE_DISPLAY_NAME: "Install Latest %NAME%"

SELF_SERVICE_DESCRIPTION: Microsoft Edge is a fast, simple, and secure web browser, built for the modern web.

SELF_SERVICE_ICON: "%NAME%.png"

INSTALL_BUTTON_TEXT: "Install %version%"

REINSTALL_BUTTON_TEXT: "Install %version%"

UPDATE_PREDICATE: "pkg_uploaded == False"

```
- Processor: StopProcessingIf
  Arguments:
    predicate: "%UPDATE_PREDICATE%"

- Processor: com.github.grahampugh.recipes.commonprocessors/
  VersionRegexGenerator

- Processor: com.github.grahampugh.jamf-upload.processors/
  JamfExtensionAttributeUploader
  Arguments:
    ea_script_path: "%EXTENSION_ATTRIBUTE_SCRIPT%"
    ea_name: "%EXTENSION_ATTRIBUTE_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/
  JamfComputerGroupUploader
  Arguments:
    computergroup_template: "%GROUP_TEMPLATE%"
    computergroup_name: "%GROUP_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
  Arguments:
    policy_template: "%POLICY_TEMPLATE%"
    policy_name: "%POLICY_NAME%"
    icon: "%SELF_SERVICE_ICON%"
```



```
#!/bin/sh

CFBundleShortVersionString=""

if [ -f "/Applications/%JSS_INVENTORY_NAME%/Contents/Info.plist" ]; then
    CFBundleShortVersionString=$(defaults read "/Applications/
%JSS_INVENTORY_NAME%/Contents/Info.plist" CFBundleShortVersionString)
fi

echo "<result>${CFBundleShortVersionString}</result>"

exit 0
```

And here is the script itself. We can add this script anywhere within our Recipe Repo. As you can see, you can add variable substitution to scripts, just as you do in templates and recipes. These will be populated during the recipe run. So you can create "template" scripts and EAs that can be used again and again for different policies and smart group criteria.

Advanced tips

I want to mention four advanced tips for using JamfUploader processors that can make your testing and automation even better.

1. Replacing an existing package

```
autopkg run Firefox.jamf
```

First: If I need to replace an existing package due to an upload failure, or while testing the package construction in the pkg recipe, 🍏 I can override the default of the *replace_pkg* key, and the existing package in the repo will be deleted and a new upload will be made. Since the package has changed, the StopProcessingIf predicate will be false and recipe will then go to completion.

1. Replacing an existing package

```
autopkg run Firefox.jamf  
--key replace_pkg=True
```

2. Overriding StopProcessingIf

Second tip: while testing a recipe, we might want to override the behaviour of the StopProcessingIf process, to force the rewrite of the smart group and policy without replacing the existing package.

```
INSTALL_BUTTON_TEXT: "Install %version%"
REINSTALL_BUTTON_TEXT: "Install %version%"
UPDATE_PREDICATE: "pkg_uploaded == False"

Process:
- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
  Arguments:
    category_name: "%CATEGORY%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader

- Processor: StopProcessingIf
  Arguments:
    predicate: "%UPDATE_PREDICATE%"

- Processor: com.github.grahampugh.jamf-upload.processors/
JamfComputerGroupUploader
  Arguments:
    computergroup_template: "%GROUP_TEMPLATE%"
    computergroup_name: "%GROUP_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
  Arguments:
```

For this to work, we need to make the 🍏 **StopProcessingIf** processor overridable. This is done by moving the predicate variable into the Input list.

```
INSTALL_BUTTON_TEXT: "Install %version%"  
REINSTALL_BUTTON_TEXT: "Install %version%"  
UPDATE_PREDICATE: "pkg_uploaded == False"
```

Process:

- Processor: com.github.grahampugh.jamf-upload.processors/JamfCategoryUploader
Arguments:
category_name: "%CATEGORY%"
- Processor: com.github.grahampugh.jamf-upload.processors/JamfPackageUploader
- Processor: StopProcessingIf
Arguments:
predicate: "%UPDATE_PREDICATE%"
- Processor: com.github.grahampugh.jamf-upload.processors/
JamfComputerGroupUploader
Arguments:
computergroup_template: "%GROUP_TEMPLATE%"
computergroup_name: "%GROUP_NAME%"
- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
Arguments:

2. Overriding StopProcessingIf

```
autopkg run Firefox.jamf
```

With the key now overridable, 🍏 We can override the key at the command line like this. The string FALSEPREDICATE will always return False, meaning that the predicate is never met, so the recipe will continue.

2. Overriding StopProcessingIf

```
autopkg run Firefox.jamf  
--key STOP_PROCESSING_IF=FALSEPREDICATE
```

3. Better scoping

<i>Category</i>	<i>Varies</i>
<i>Triggers</i>	Recurring Check-in
<i>Frequency</i>	Ongoing
<i>Scope</i>	"Testing" static group - <i>and</i> - Current or newer version of Firefox not installed
<i>Self Service</i>	Yes
<i>Payload</i>	<i>Single Package</i>

Thirdly, let's look at a better way of scoping.

Jamf Pro cannot compare version strings to tell which are older or newer. But if we only want to scope a version of Firefox to computers that have an older version installed, and avoid those that have a newer version, we really need that version comparison logic. This single failing in Jamf Pro is why many people use Munki with Jamf, or spend a lot of time maintaining Jamf Patch definitions.

```
- Processor: StopProcessingIf
  Arguments:
    predicate: "%UPDATE_PREDICATE%"

- Processor: com.github.grahampugh.recipes.commonprocessors/
VersionRegexGenerator

- Processor: com.github.grahampugh.jamf-upload.processors/
JamfExtensionAttributeUploader
  Arguments:
    ea_script_path: "%EXTENSION_ATTRIBUTE_SCRIPT%"
    ea_name: "%EXTENSION_ATTRIBUTE_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/
JamfComputerGroupUploader
  Arguments:
    computergroup_template: "%GROUP_TEMPLATE%"
    computergroup_name: "%GROUP_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
  Arguments:
    policy_template: "%POLICY_TEMPLATE%"
    policy_name: "%POLICY_NAME%"
    icon: "%SELF_SERVICE_ICON%"
```

Going back to the Edge recipe, you may have noticed another processor in the Process list called VersionRegexGenerator. This single line provides that comparison logic we need.

```
- Processor: StopProcessingIf
  Arguments:
    predicate: "%UPDATE_PREDICATE%"

- Processor: com.github.grahampugh.recipes.commonprocessors/
VersionRegexGenerator

- Processor: com.github.grahampugh.jamf-upload.processors/
JamfExtensionAttributeUploader
  Arguments:
    ea_script_path: "%EXTENSION_ATTRIBUTE_SCRIPT%"
    ea_name: "%EXTENSION_ATTRIBUTE_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/
JamfComputerGroupUploader
  Arguments:
    computergroup_template: "%GROUP_TEMPLATE%"
    computergroup_name: "%GROUP_NAME%"

- Processor: com.github.grahampugh.jamf-upload.processors/JamfPolicyUploader
  Arguments:
    policy_template: "%POLICY_TEMPLATE%"
    policy_name: "%POLICY_NAME%"
    icon: "%SELF_SERVICE_ICON%"
```

VersionRegexGenerator

virtual
JNUK
2021

The VersionRegexGenerator processor is based on a script made by Jamf's own 🍏 William Smith. It calculates a regular expression that matches the version of the app in the package being uploaded, plus any plausible higher version string. 🍏 Here is the string generated by the processor for Firefox version 90.0. I won't attempt to break it down, but this string represents version 90.0 and any possible higher version.

🍏 Using this string instead of the actual version allows me to scope my install policy only to computers that have an older version of the app installed, rather than the normal way with Jamf which is to only match an exact value.

The value is added as a smart group criterion using the "Application Version matches regex", or "Application Version does not match regex", depending on your smart group composition.

VersionRegexGenerator



VersionRegexGenerator

```
^(\d{3,}.*|9[1-9].*|90\.\d{2,}.*|90\.[1-9].*|90\.0.*)$
```

VersionRegexGenerator

`^(\\d{3,}.*|9[1-9].*|90\\.\\d{2,}.*|90\\. [1-9].*|90\\.0.*)$`

Computers : Smart Computer Groups
← Firefox current version installed

Computer Group	Criteria																				
	<table border="1"><thead><tr><th>AND/OR</th><th>CRITERIA</th><th>OPERATOR</th><th>VALUE</th></tr></thead><tbody><tr><td></td><td>Application Title</td><td>is</td><td>Firefox.app</td></tr><tr><td>and</td><td>Application Version</td><td>matches regex</td><td>^(\\d{3,} 9[1-9] 90\\.\\d{2,} 90\\. [1-9] 90\\.0).*</td></tr><tr><td>or</td><td>Application Version</td><td>matches regex</td><td>^\$</td></tr><tr><td>and</td><td>Computer Group</td><td>member of</td><td>Firefox users</td></tr></tbody></table>	AND/OR	CRITERIA	OPERATOR	VALUE		Application Title	is	Firefox.app	and	Application Version	matches regex	^(\\d{3,} 9[1-9] 90\\.\\d{2,} 90\\. [1-9] 90\\.0).*	or	Application Version	matches regex	^\$	and	Computer Group	member of	Firefox users
AND/OR	CRITERIA	OPERATOR	VALUE																		
	Application Title	is	Firefox.app																		
and	Application Version	matches regex	^(\\d{3,} 9[1-9] 90\\.\\d{2,} 90\\. [1-9] 90\\.0).*																		
or	Application Version	matches regex	^\$																		
and	Computer Group	member of	Firefox users																		

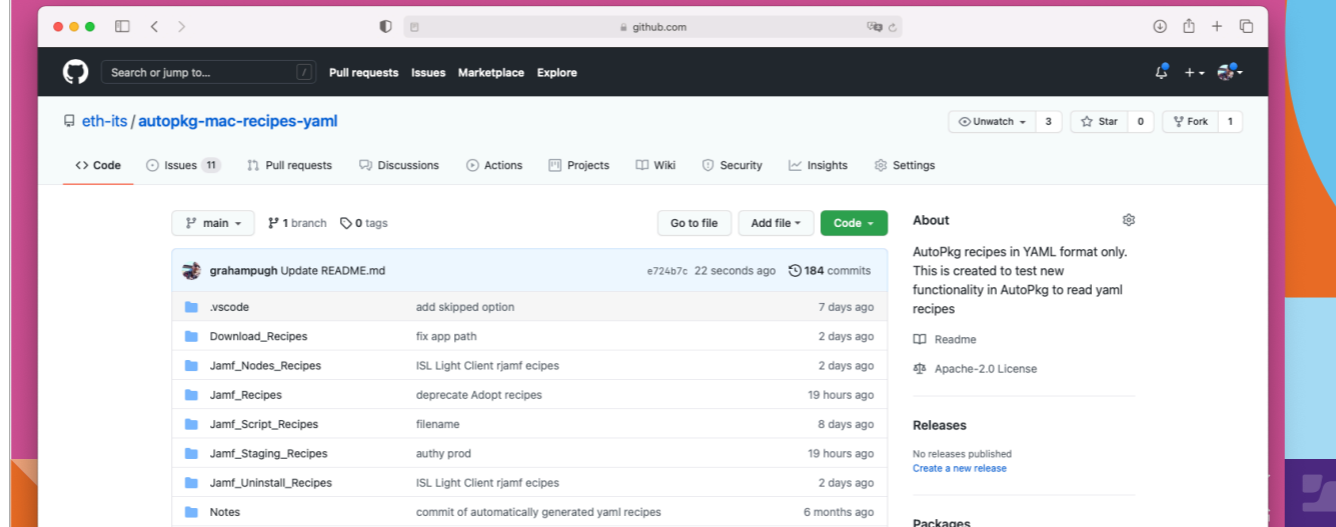

```
<?xml version="1.0" encoding="UTF-8"?>
<computer_group>
  <name>%GROUP_NAME%</name>
  <is_smart>true</is_smart>
  <criteria>
    <criteria>
      <name>Application Title</name>
      <priority>0</priority>
      <and_or>and</and_or>
      <search_type>is</search_type>
      <value>%JSS_INVENTORY_NAME%</value>
      <opening_paren>true</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>%VERSION_CRITERION%</name>
      <priority>1</priority>
      <and_or>and</and_or>
      <search_type>does not match regex</search_type>
      <value>%version_regex%</value>
      <opening_paren>>false</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>Application Title</name>
      <priority>2</priority>
      <and_or>or</and_or>
```

In our Smart Group Template, we change to using the "does not match regex" search type, and set the value to a variable named version_regex, which is outputted from the processor.

Using this simple processor in my Jamf recipes makes scoping equivalent to using Munki or Patch. It means I don't need to maintain Patch definitions to figure out which versions are newer or older than the installed version, and can stick with using regular policies instead.

```
<?xml version="1.0" encoding="UTF-8"?>
<computer_group>
  <name>%GROUP_NAME%</name>
  <is_smart>true</is_smart>
  <criteria>
    <criteria>
      <name>Application Title</name>
      <priority>0</priority>
      <and_or>and</and_or>
      <search_type>is</search_type>
      <value>%JSS_INVENTORY_NAME%</value>
      <opening_paren>true</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>%VERSION_CRITERION%</name>
      <priority>1</priority>
      <and_or>and</and_or>
      <search_type>does not match regex</search_type>
      <value>%version_regex%</value>
      <opening_paren>>false</opening_paren>
      <closing_paren>>false</closing_paren>
    </criteria>
    <criteria>
      <name>Application Title</name>
      <priority>2</priority>
      <and_or>or</and_or>
```

4. Repo organisation



My fourth tip concerns repo organisation.

JSSImporter is quite strict about where you have to place templates and files associated with a recipe for them to be seen, but JamfUploader is more flexible. JamfUploader processors will search the entire repo that contains the recipe for a file of matching name to that specified in the recipe. That means that in most circumstances, you do not need to provide a path, and they do not need to be in the same folder as the recipe. Nor do you need to copy them to the RecipeOverrides folder, though you can if you like. In my repos, 🍏 I put all my templates into a single folder named Templates. 🍏 All my scripts and Extension Attributes go into a folder named Scripts, and 🍏 all the icons needed for Self Service go in a single folder, meaning they don't need to be replicated if used in more than one recipe.

You can organise all the additional files how you like - so long as they are somewhere in the same repository, or in your RecipeOverrides folder, JamfUploader will find them.

4. Repo organisation

The screenshot shows a GitHub repository page for 'eth-its/autopkg-mac-recipes-yaml'. The browser address bar shows the URL 'https://github.com/eth-its/autopkg-mac-recipes-yaml/tree/main/Templates'. The repository name is 'eth-its / autopkg-mac-recipes-yaml'. The page shows the 'Templates' directory with a commit history table.

File Name	Commit Message	Time
..		
Policy-all-computers-once-per-computer.xml	commit of automatically generated yaml recipes	5 months ago
Policy-prod-autoinstall-OS-exclude-max.xml	fix templates	4 months ago
Policy-prod-autoinstall-package.xml	commit of automatically generated yaml recipes	5 months ago
Policy-prod-autoinstall.xml	commit of automatically generated yaml recipes	5 months ago
Policy-prod-autoupdate-OS-exclude-max.xml	fix templates	4 months ago
Policy-prod-autoupdate.xml	commit of automatically generated yaml recipes	5 months ago

4. Repo organisation

The screenshot shows a web browser displaying the GitHub repository page for 'eth-its/autopkg-mac-recipes-yaml'. The browser's address bar shows the URL 'https://github.com/eth-its/autopkg-mac-recipes-yaml/tree/main/Scripts'. The repository page includes a search bar, navigation links for Pull requests, Issues, Marketplace, and Explore, and repository statistics: 3 Unwatch, 0 Stars, and 1 Fork. The main content area shows the 'Scripts' directory with a commit history table.

File Name	Commit Message	Commit Hash	Time Ago
..			
AdobeAir-EA.sh	commit of automatically generated yaml recipes	2172a22	3 days ago
AdobeAir-uninstall.sh	NiceUpdater2		5 months ago
AdobeFlashPlayer-EA.sh	commit of automatically generated yaml recipes		3 months ago
AdobeFlashPlayer-uninstall.sh	commit of automatically generated yaml recipes		5 months ago
AdoptOpenJDK-EA.sh	commit of automatically generated yaml recipes		5 months ago
AdoptOpenJDK-uninstall.sh	fix uninstaller script		5 months ago
Application-uninstall.sh	stata 17 uninstaller		2 months ago

4. Repo organisation

The screenshot displays a GitHub repository page for 'eth-its / autopkg-mac-recipes-yaml'. The browser address bar shows the URL 'https://github.com/eth-its/autopkg-mac-recipes-yaml/tree/main/Scripts'. The repository page includes navigation links for 'Code', 'Issues (5)', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. The current view is the 'Scripts' directory, showing a commit history table.

Commit Message	Author	Time
..		
commit of automatically generated yaml recipes	Graham Pugh	5 months ago
commit of automatically generated yaml recipes		5 months ago
commit of automatically generated yaml recipes		5 months ago
app store app recipes		4 months ago
commit of automatically generated yaml recipes		5 months ago

Final Thoughts and Resources

That's some advanced tips covered.
Let's finish with some conclusions.

Conclusions



[28:50]

[Graham starts:]

🍏 From my side, I've designed JamfUploader to replace JSSImporter in my organisation, though they can be used side-by-side. Over the course of this year I am replacing all my old jss recipes with jamf recipes.

🍏 It's designed to be simpler to install and use, with more intuitive recipe construction.

🍏 Everything the processors need to do are based on the python 3 distribution that is bundled in with your AutoPkg installer package.

🍏 By removing the dependency on the python-jss framework, I've got rid of all the tech debt - for now at least - and the code should be much easier to understand for anyone who wants to contribute to the project.

[Anthony] 🍏 For me, this was much less to learn than using JSSImporter or Munki, and 🍏 I could start small and scale up, because it is flexible that way. JamfUploader opens up more ways to use AutoPkg for more workflows and lets you eliminate any complexity you don't need for your situation.

Conclusions

1. JamfUploader is designed as a replacement for JSSImporter

Conclusions

1. JamfUploader is designed as a replacement for JSSImporter
2. Simpler installation and recipe construction

Conclusions

1. JamfUploader is designed as a replacement for JSSImporter
2. Simpler installation and recipe construction
3. Python 3 from day 1

Conclusions

1. JamfUploader is designed as a replacement for JSSImporter
2. Simpler installation and recipe construction
3. Python 3 from day 1
4. Tech debt - not yet

Conclusions

1. JamfUploader is designed as a replacement for JSSImporter
2. Simpler installation and recipe construction
3. Python 3 from day 1
4. Tech-debt - not yet
5. Shallower learning curve

Conclusions

1. JamfUploader is designed as a replacement for JSSImporter
2. Simpler installation and recipe construction
3. Python 3 from day 1
4. Tech-debt - not yet
5. Shallower learning curve
6. Flexible and scalable

Resources

MacAdmins Slack: [#jamf-upload](#)

<https://grahamrpugh.com/2021/10/21/jnuc-presentation-jamfuploader-session.html>



[Graham] I hope you have seen something of interest, and that you'd like to give the JamfUploader processors a try! If you want to continue the discussion after this event, come to the MacAdmins Slack and join the Jamf-upload channel.

Rather than trying to provide links to everything that we've mentioned today in a slide, I've published a blog post containing all the links. So just visit grahamrpugh.com and find the post about JNUC 2021. Thanks for joining us!