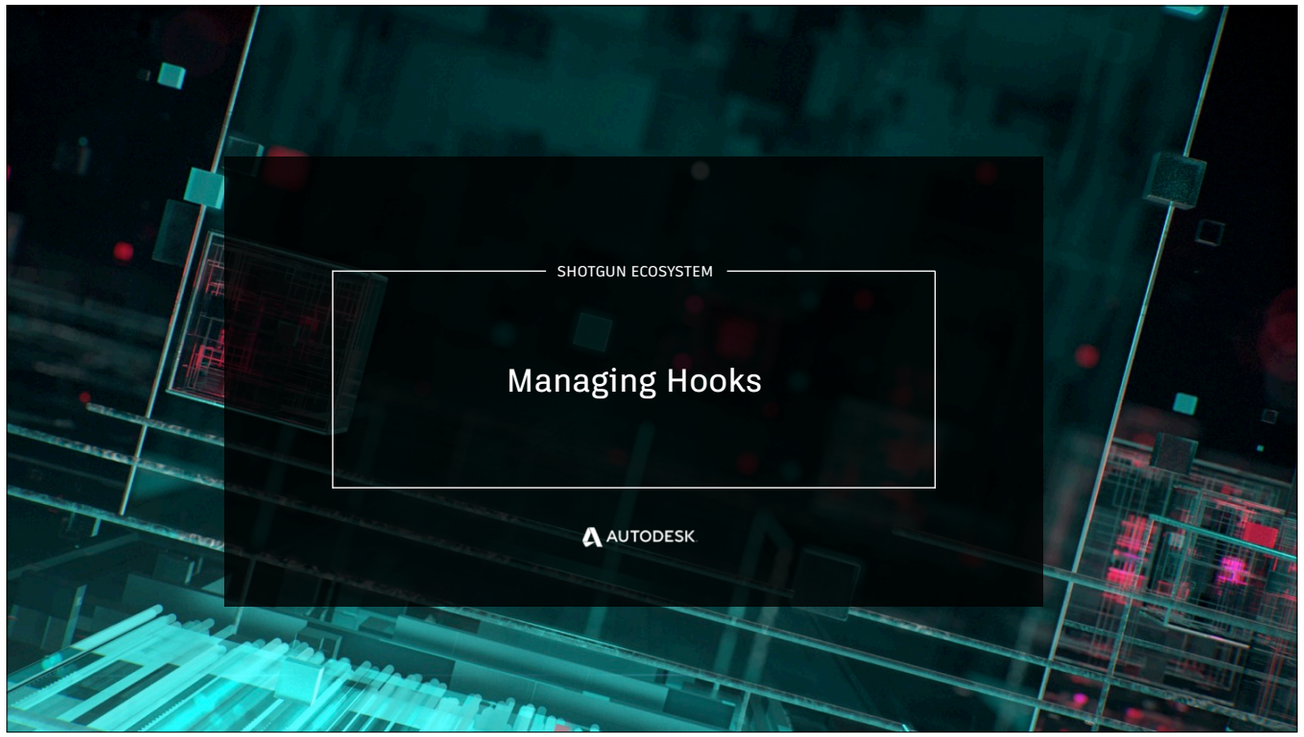
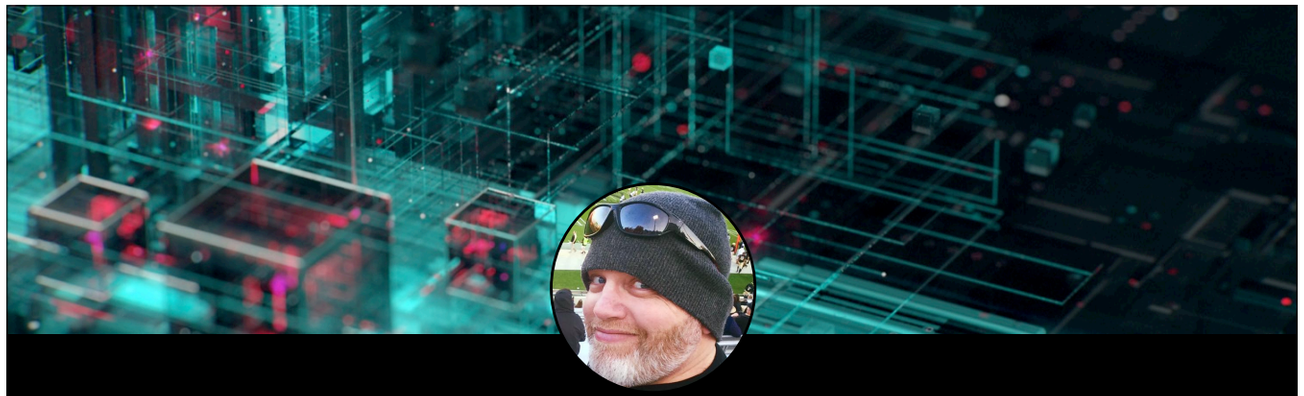


- The goal of this talk is to give you an overview of hook and hook management
- We'll expand on what Tannaz covered and talk specifically about these powerful workflow customizations available in Toolkit

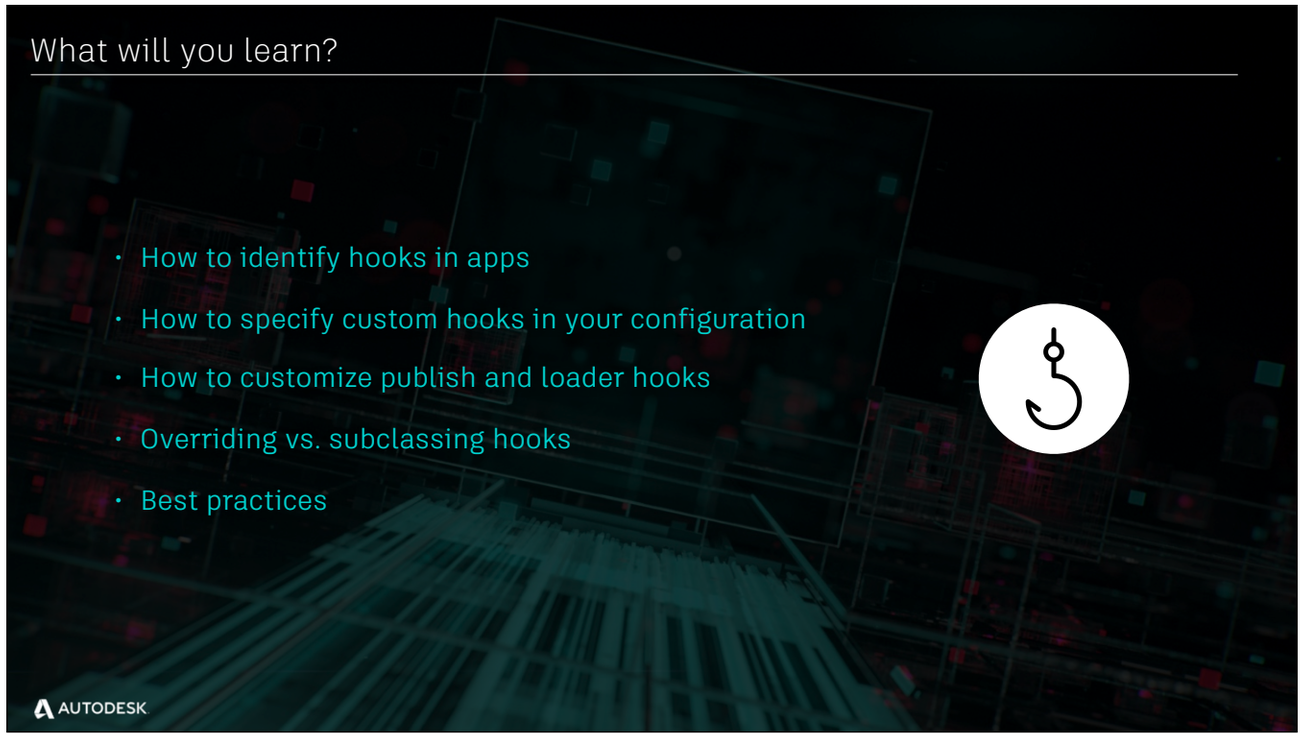


JOSH TOMLINSON

Senior Software Engineer, Autodesk

Josh worked at Rhythm & Hues Studios for 11 years where he helped develop pipeline and workflow solutions for feature film visual effects. He then spent a year and a half building an open source pipeline at Clemson University, his alma mater, before joining Autodesk in 2015. Josh is a Software Engineer as part of the Shotgun Toolkit team and enjoys building tools and APIs used by hundreds of production studios around the world.

AUTODESK.



- What are we going to cover?
- We’ll be doing a pretty broad overview of hooks
- Identifying and customizing them in the configuration
- We’ll look at a couple of publish and load examples
- We’ll talk about overriding vs. subclassing
- And a little about best practices

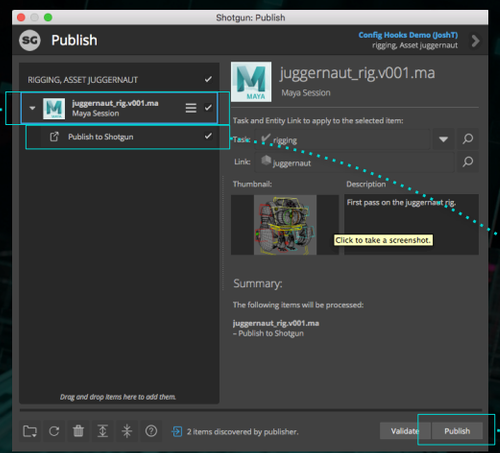
DEV DAY RESOURCES

github.com/shotgunsoftware/sg-devday-2018

- Let’s start with a quick overview/refresher of what hooks are in a practical sense and how they’re specified in the configuration

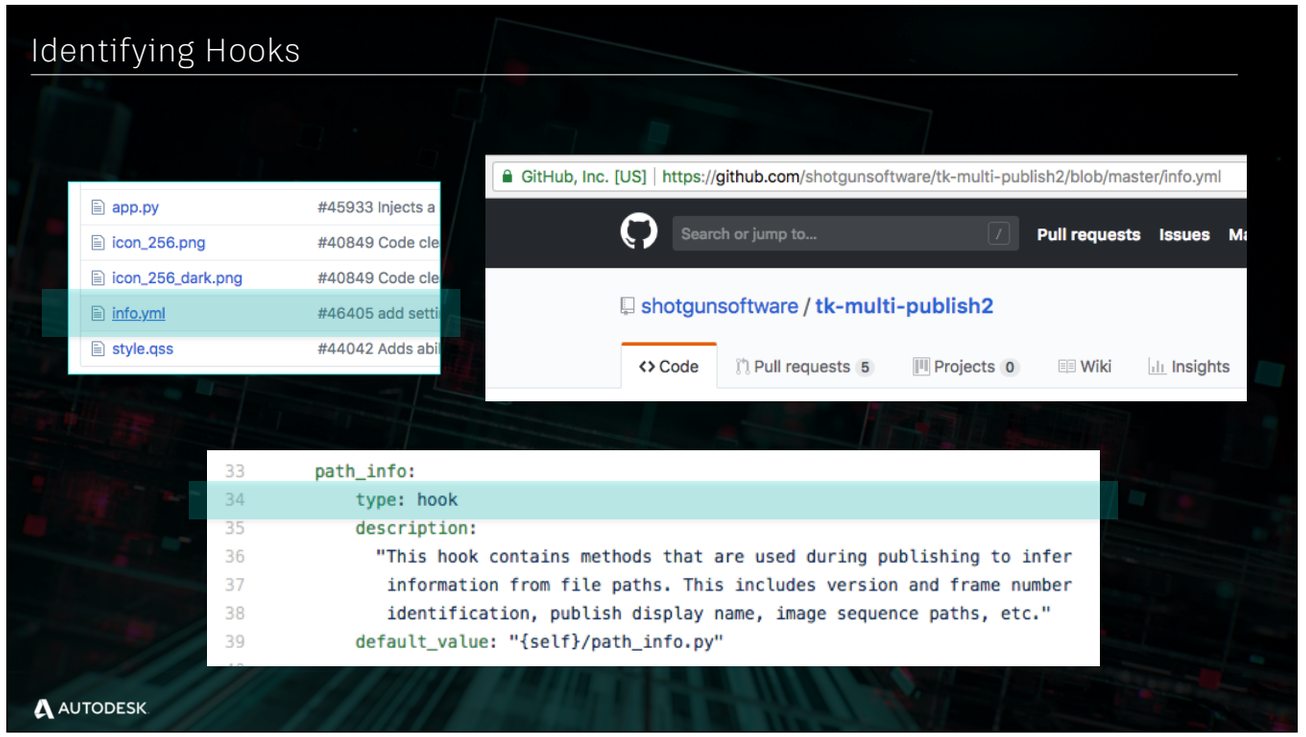
What are Hooks?

what

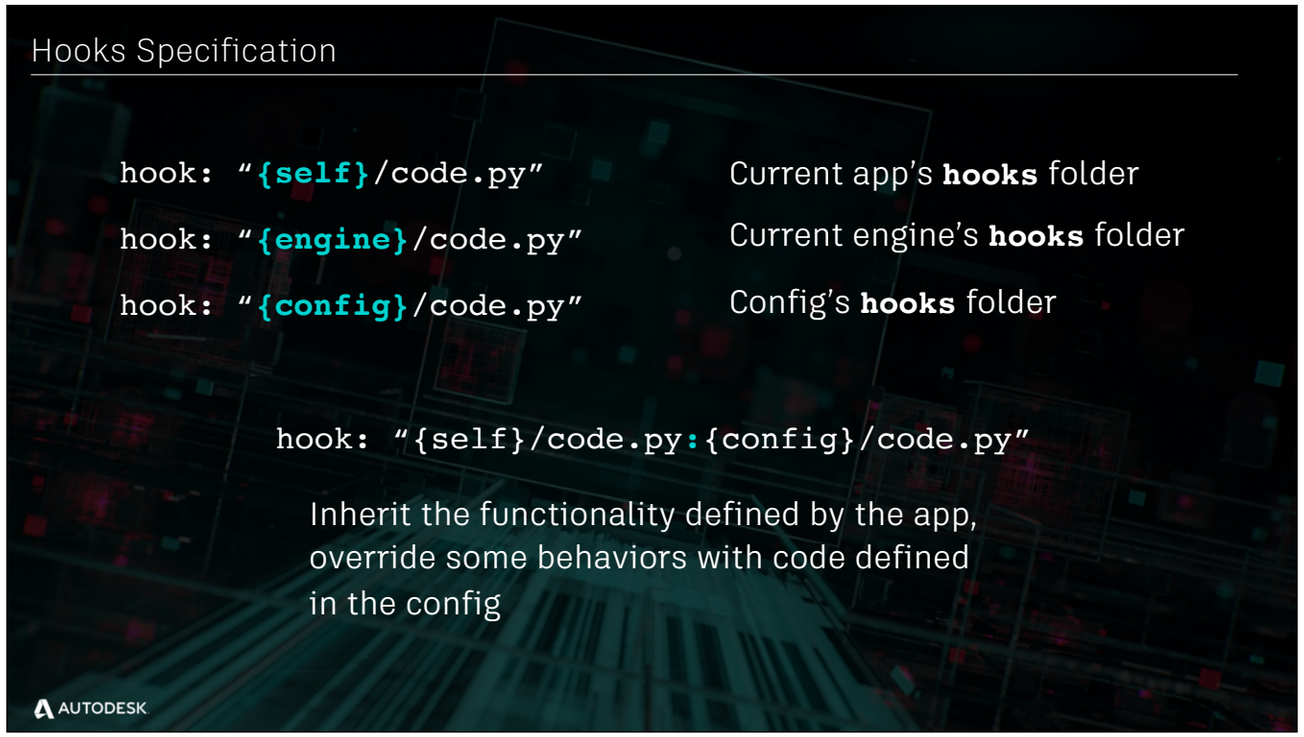


how

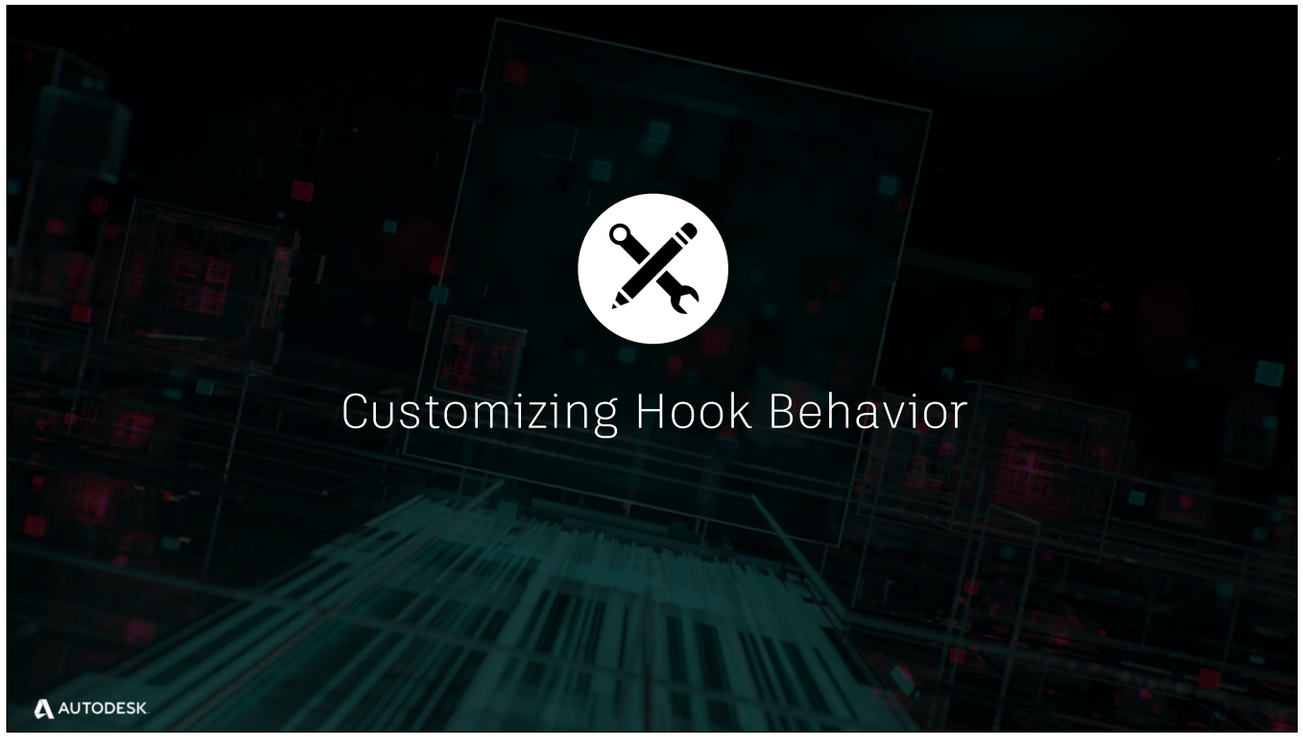
- Hooks are pieces of execution logic that are available for customization, primarily within Toolkit apps
- Hooks are typically exposed in portions of an app that often require studio-specific behaviors
- Take the publisher for example
    - CLICK
    - The logic that defines what items are to be published is exposed via a hook
    - CLICK
    - Similarly, the logic that defines how those items are to be published is also exposed via hooks
- In other words, hooks allow you build your studio's business logic into existing apps without having to roll your own solution



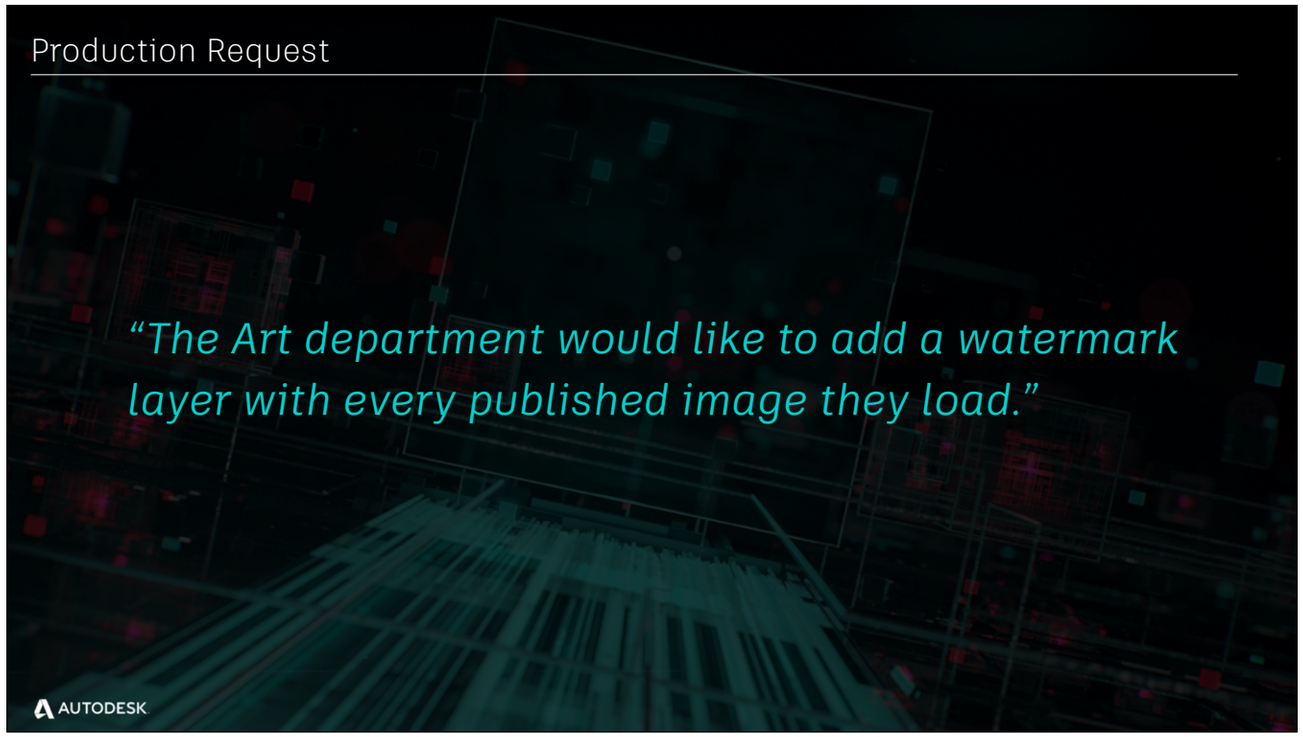
- How do I know what hooks are available for a given app?
- Every Toolkit app includes a top-level info.yml file that defines all the configuration settings available for that app
- CLICK
- All available hooks for the app are defined within this file as settings of type “hook” that can be overridden in a client configuration
- CLICK
- You should check your configuration to see what hooks are used in the various contexts
- These are often engine-specific
- Keep in mind that our basic and default2 configurations are sparse and don’t always define values for available hooks
- So it is important to know how to locate these settings
- CLICK
- All of our shipped apps are available on GitHub, so you can check there if you’re curious about what hooks are available for an app



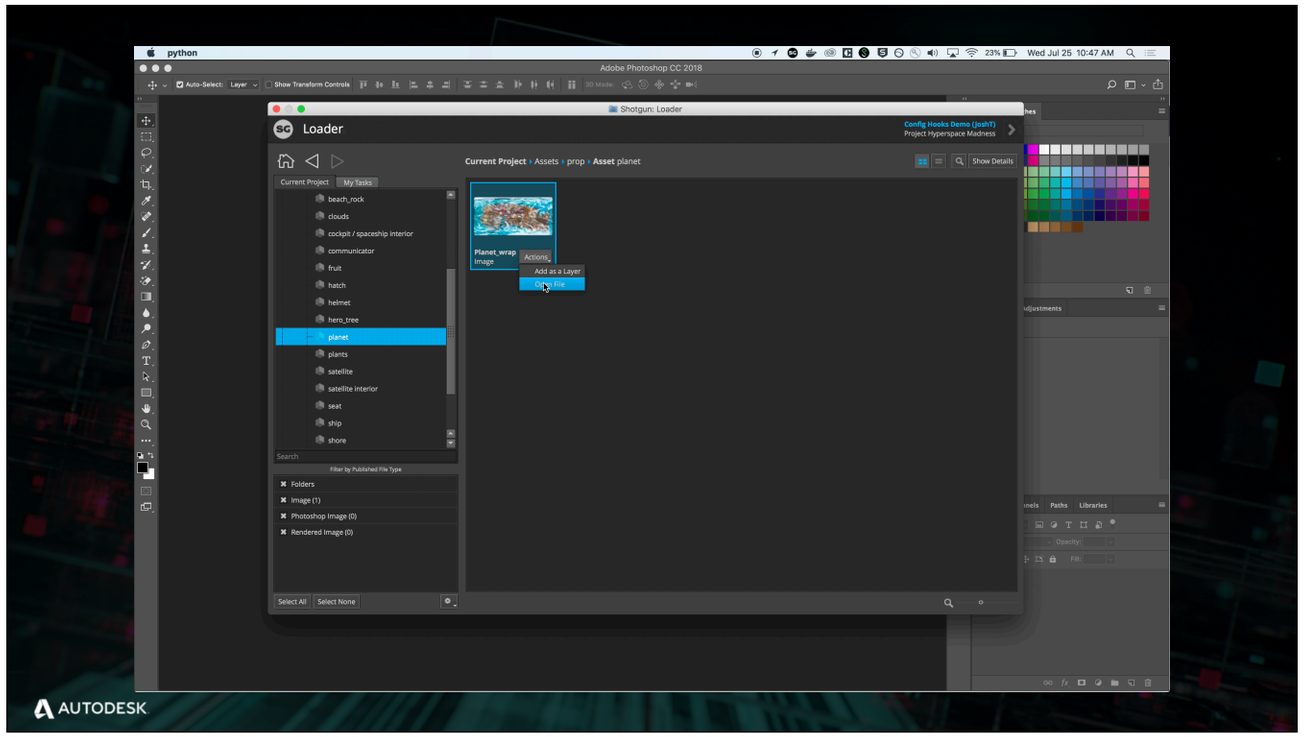
- You can define a hook by simply specifying a path in an app's settings within your configuration
- CLICK
- This is just a string that tells the app where to find the code to execute for a given hook
- CLICK
- CLICK
- Here you can see a few examples of where hook code can be defined
- There are other special tokens as well that you can read about in the documentation
- Hooks allow you to do more than just override behavior
- You can subclass existing hooks to modify only the parts you need
- CLICK



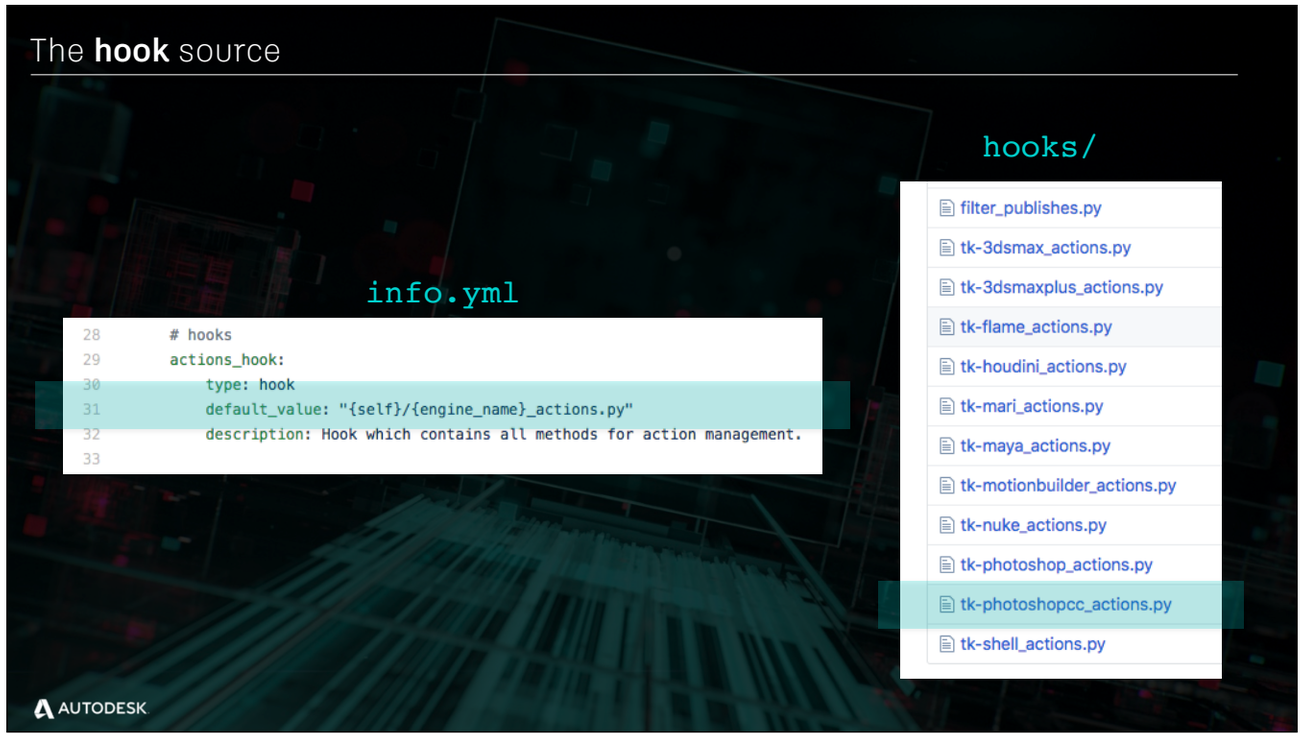
- Now let’s work through a couple of simple production examples of customizing hook behavior



- We’ll start with a request from production
- “The Art department would like to add a watermark layer with every published image they load.”
- How do we address this?
- We know that the SG ships the ability to load images into photoshop
- So maybe we can piggy back on that and just add a little bit of code to add the new layer



- In this video we see the default loader behavior
- We browse assets for the current project
- Find the asset we’re interested in
- We see the published image we’re interested in loading
- When we load it, we see there’s a new image in PS with a single background layer
- Now let’s take that default behavior and customize it



- The first thing we need to do is find the existing hook source code that handles PS imports
- As we saw before we can locate the info.yml via the loader2 app on github
- The hook we need to modify is the actions_hook
- CLICK
- And by default it’s configured with the {engine_name} token
- This is one of the special tokens I mentioned before
- This token will evaluate during the evaluation of the config and the result is a hook file called tk-photoshopcc_actions.py found in the app itself
- CLICK
- Note the {self} token on the left. That’s the indicator that the hook is expected to live within the app itself
- As mentioned before, you should double check your configuration to see if it is overriding this default value
- For this example we’ll assume the starting point is the default2 config which does not override this hook value

The **hook** source

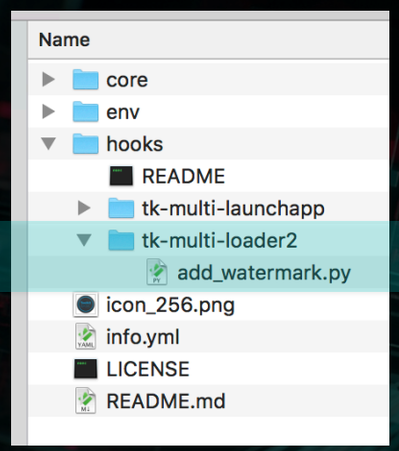
tk-photoshopcc_actions.py

```
def execute_action(self, name, params, sg_publish_data):
    """
    Execute a given action. The data sent to this be method will
    represent one of the actions enumerated by the generate_actions method.

    :param name: Action name string representing one of the items returned
                 by generate_actions.
    :param params: Params data, as specified by generate_actions.
    :param sg_publish_data: Shotgun data dictionary with all the standard
                            publish fields.
    """
    app = self.parent
    app.log_debug("Execute action called for action %s. "
                  "Parameters: %s. Publish Data: %s" % (name, params, sg_publish_data))

    # resolve path
    # toolkit uses utf-8 encoded strings internally and the Photoshop API expects unicode
    # so convert the path to ensure filenames containing complex characters are supported
    path = self.get_publish_path(sg_publish_data).decode('utf-8')

    if not os.path.exists(path):
        raise Exception("File not found on disk - '%s'" % path)

    if name == _OPEN_FILE:
        self._open_file(path, sg_publish_data)
    if name == _ADD_AS_A_LAYER:
        self._place_file(path, sg_publish_data)
```

AUTODESK

- If we look at this file, we can see 2 actions defined: Open File and Add as a Layer
- The request from production was to add an additional layer upon file open
  - So maybe we can modify the file open method to add the new layer
  - We might even be able to reuse the existing “add layer” code to do exactly what we need

Override vs. Subclass



Create the hook

Define it as a subclass of the default PS actions in the config

```
actions_hook: "{self}/tk-photoshopcc_actions.py:{config}/tk-multi-loader2/add_watermark.py"
```

AUTODESK

- So the next question is, do we take over the hook or subclass it
- Since we want to modify the existing behavior, subclassing makes a lot more sense
  - In general, if you can subclass the code it's always a good idea
  - This allows you to receive updates to the default implementations that include new features and bug fixes
  - In this case, if the Toolkit team adds support for new file types or faster loading for example, we want to get those changes
- So let's define our hook subclass
- First, let's create a file in our configuration that will define our custom behavior
- We'll save this file in the hooks folder and call it tk-multi-loader2/add_watermark.py
- In our configuration, we'll then update the loader2 actions_hook to point to our new file
- Note the subclass syntax we saw at the beginning

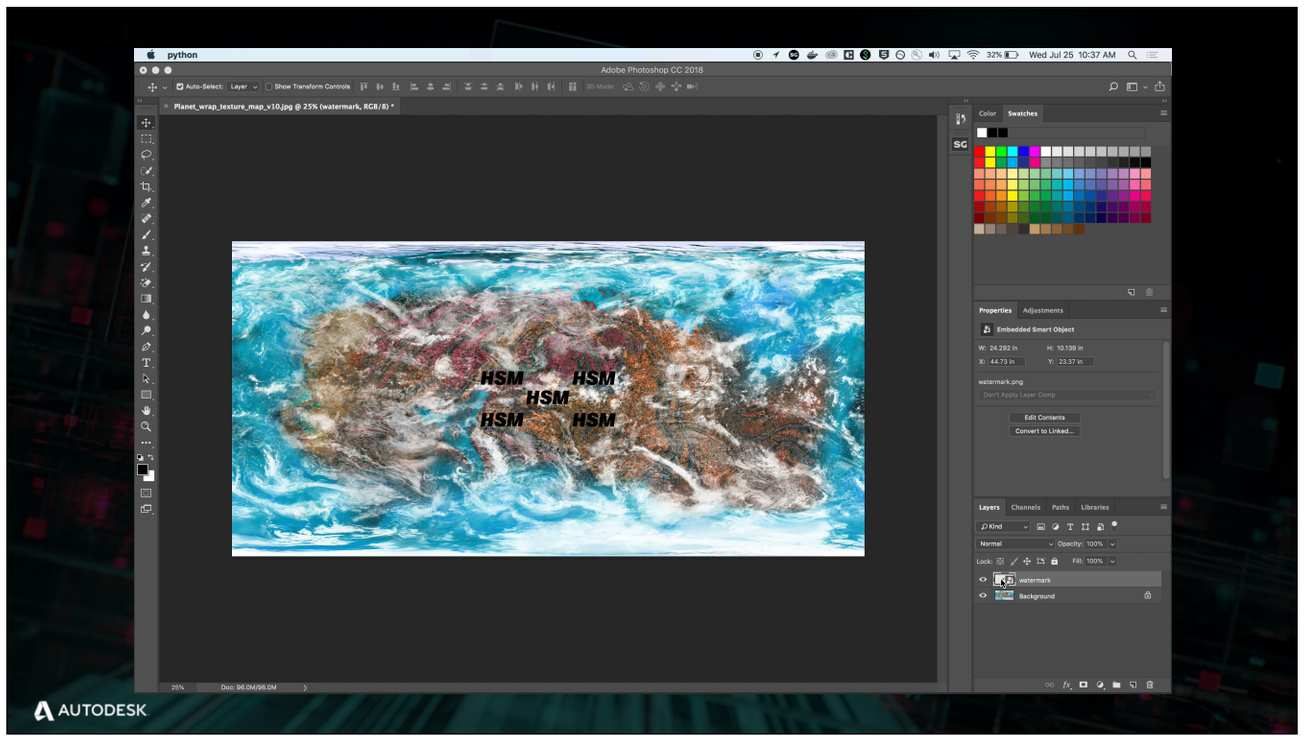
Subclassing a Hook

```python
import sgtk

HookBaseClass = sgtk.get_hook_baseclass()

WATERMARK = "/tmp/resources/watermark.png"

class OpenWithWatermarkAction(HookBaseClass):

    def _open_file(self, path, sg_publish_data):
        super(OpenWithWatermarkAction, self)._open_file(
            path,
            sg_publish_data
        )

        self._place_file(WATERMARK, sg_publish_data)
```

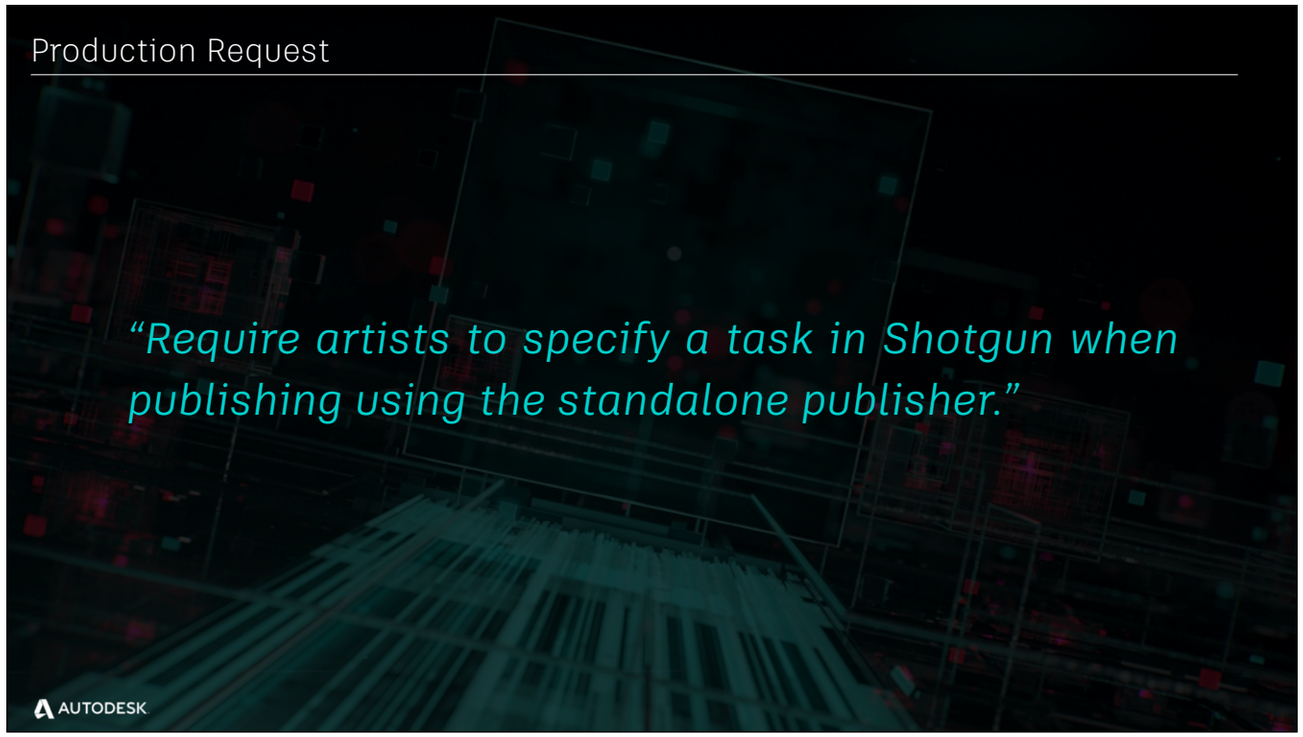
AUTODESK

- What needs to go in this new file?
- This is all the code we need
- We define our subclass, making sure to inherit from the correct base
- CLICK
- Toolkit hooks use a special base class defined by the `get_hook_baseclass` method on sgtk
- This will return the proper class given the configuration
- In our case, this will return the default photoshop actions class
- CLICK
- We override the open file method, calling the base class implementation to actually import the file
- Then we simply load the external watermark file as a new layer, reusing the existing add layer code

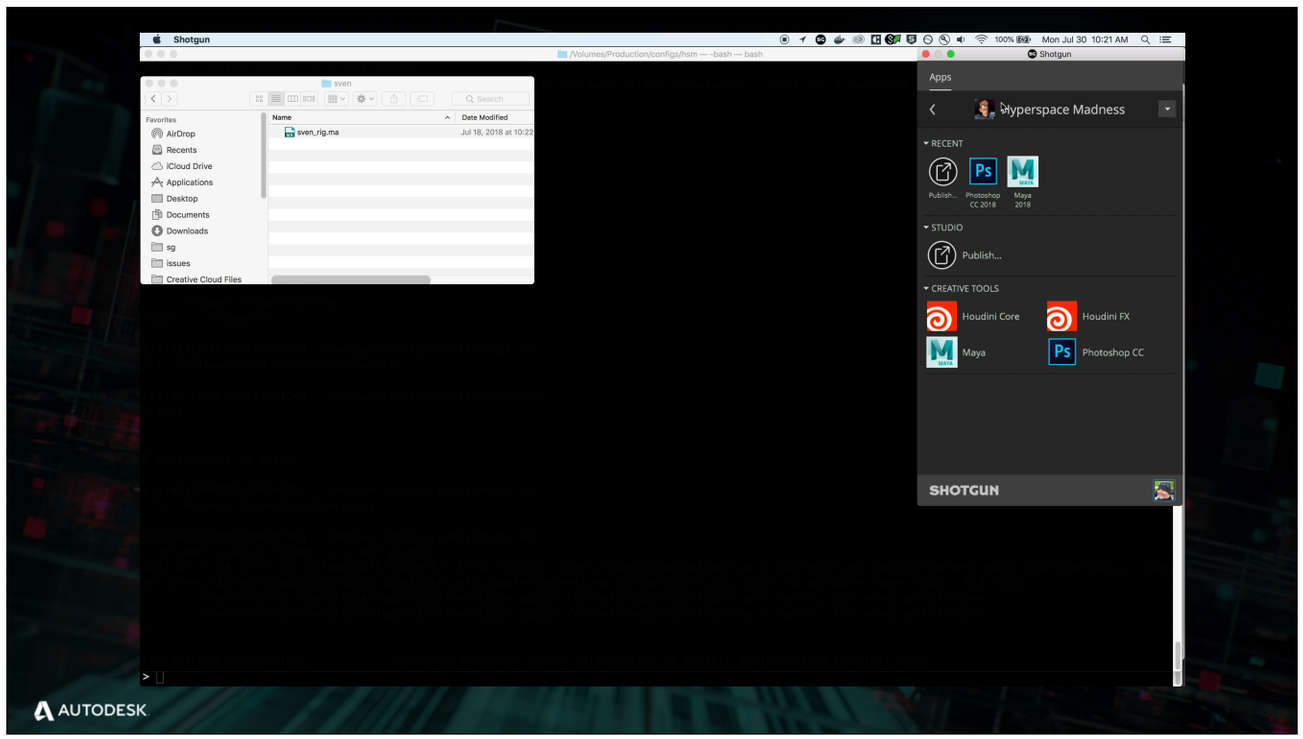
- And for reference, here’s the watermark file that will be added as a layer



- Here's a video that shows the results of the changes we made
- You can see the exact same steps we took before to load the image
- But we've modified the action and now we get the additional watermark layer



- Let’s do another example
- For this one, production has asked that we ensure that for every publish, a task has been specified
- How do we do this
- Well, let’s just see if we can block a publish if a task is not specified



- Let’s take a quick look at what a standalone publish looks like before we customize it
- You can see we drag/drop a file into the publisher and click validate
- Even though we haven’t specified a task on the right, the validation is successful

The **hook** source

info.yml

```
41      publish_plugins:
42          type: list
43          description: "List of publish plugins."
44          values:
45              type: dict
46              items:
47                  name:
48                      type: str
49                  hook:
50                      type: hook
51                  settings:
52                      type: dict
53          allows_empty: True
54          default_value:
55              # Default hook 1: publish any file to SG
56              - name: Publish to Shotgun
57                hook: "{self}/publish_file.py"
58                settings: {}
59              # Default hook 2: upload known, reviewable types to SG
60              - name: Upload for review
61                hook: "{self}/upload_version.py"
62                settings: {}
```

AUTODESK

- Like before, we need to find the existing hook source code that handles publishing files
- Looking in the info.yml via the publish2 app on github
- We see that this app's configuration is a little different
- Parsing this setting definition is beyond the scope of this talk, but it's different than the simple loader hook we saw before.
- Come find me if you have questions about publisher configuration, but just know that the the publish_file hook is the hook file we're interested in
-

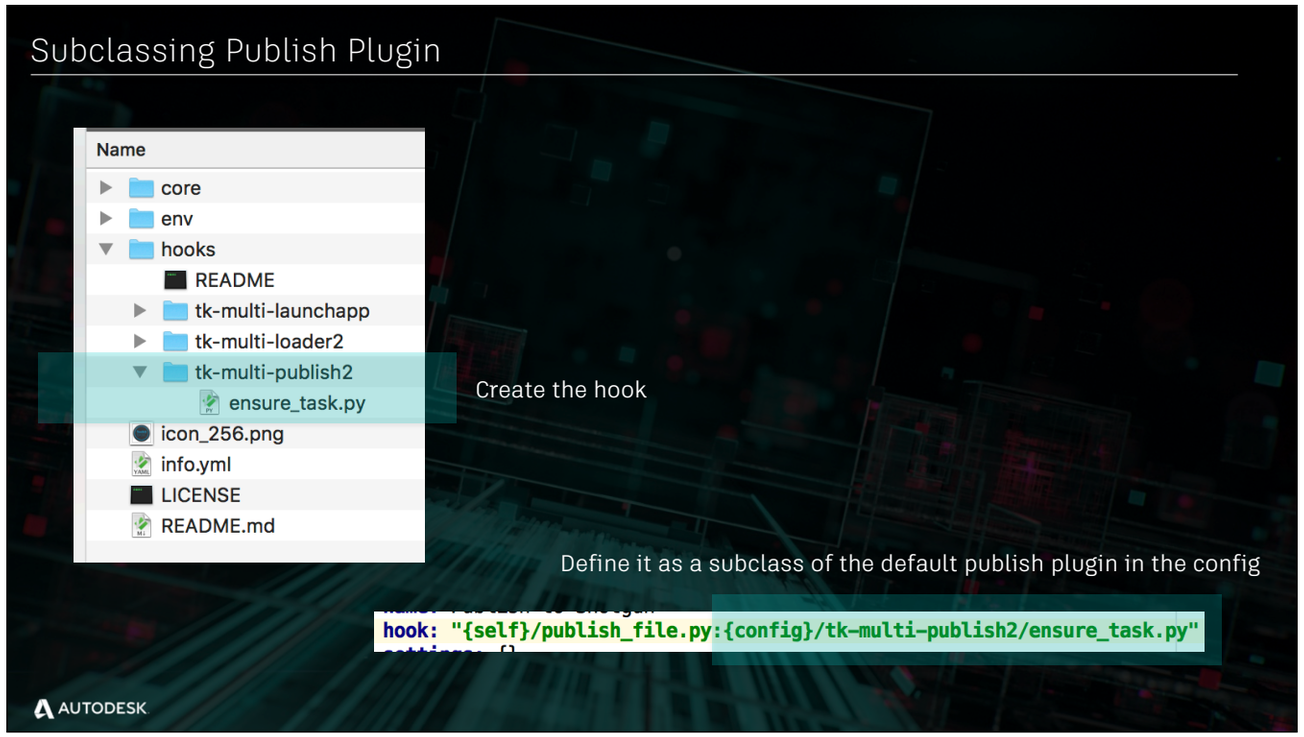
The **hook** source

publish_file.py

```
259     def validate(self, settings, item):
260         """
261         Validates the given item to check that it is ok to publish.
262 
263         Returns a boolean to indicate validity.
264 
265         :param settings: Dictionary of Settings. The keys are strings, matching
266             the keys returned in the settings property. The values are `Setting`
267             instances.
268         :param item: Item to process
269 
270         :returns: True if item is valid, False otherwise.
271         """
272 
273         publisher = self.parent
274         path = item.properties.get("path")
275 
276         # ---- determine the information required to validate
277 
278         # We allow the information to be pre-populated by the collector or a
279         # base class plugin. They may have more information than is available
280         # here such as custom type or template settings.
281 
282         publish_path = self.get_publish_path(settings, item)
283         publish_name = self.get_publish_name(settings, item)
284 
```

AUTODESK

- Each publish plugin has a validate() method than says whether the publish is ready to execute
- Since we’re simply adding onto the existing behavior, let’s see if we can subclass this hook and add the new logic to ensure a task is set for the item



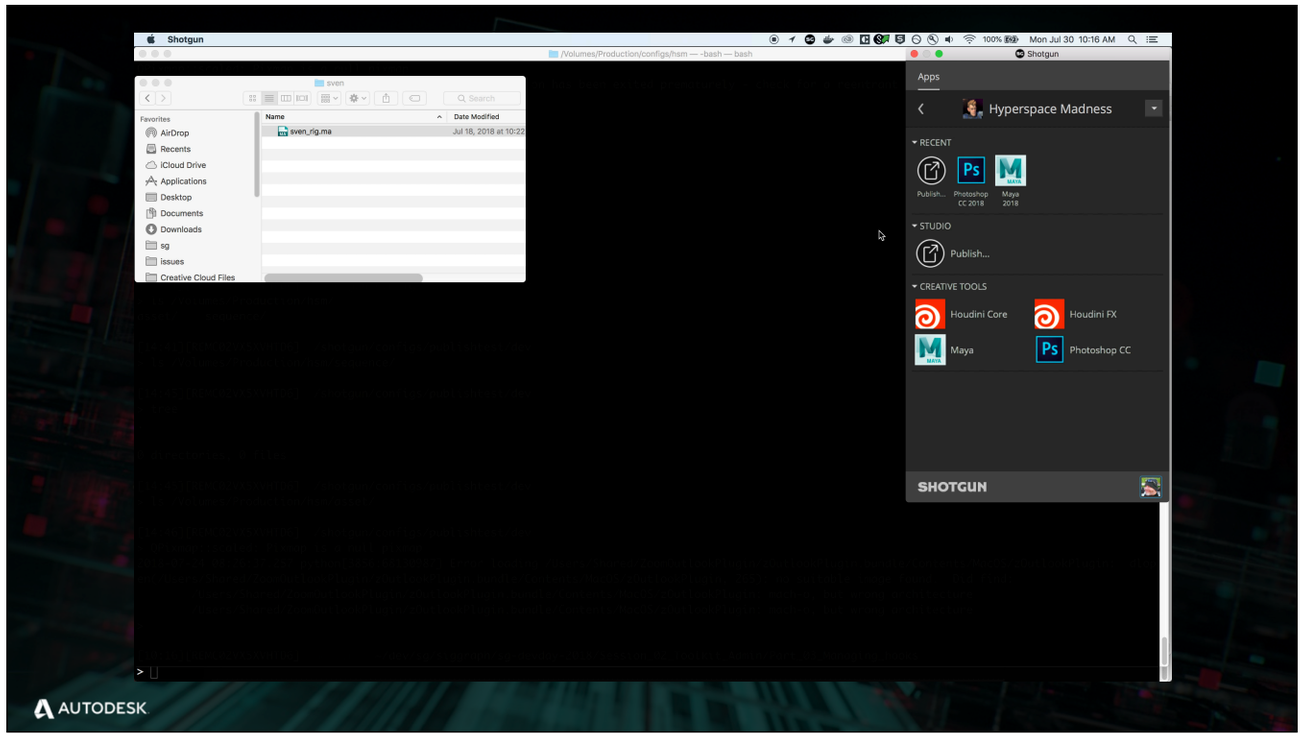
- Once again, we define our hook subclass
- We create a file in our configuration that will define the custom behavior
- We'll save this in the hooks folder and call it tk-multi-publish2/ensure_task.py
- In our configuration, we'll update the publish2 default plugin to include our new hook as a subclass

Subclassing a Hook

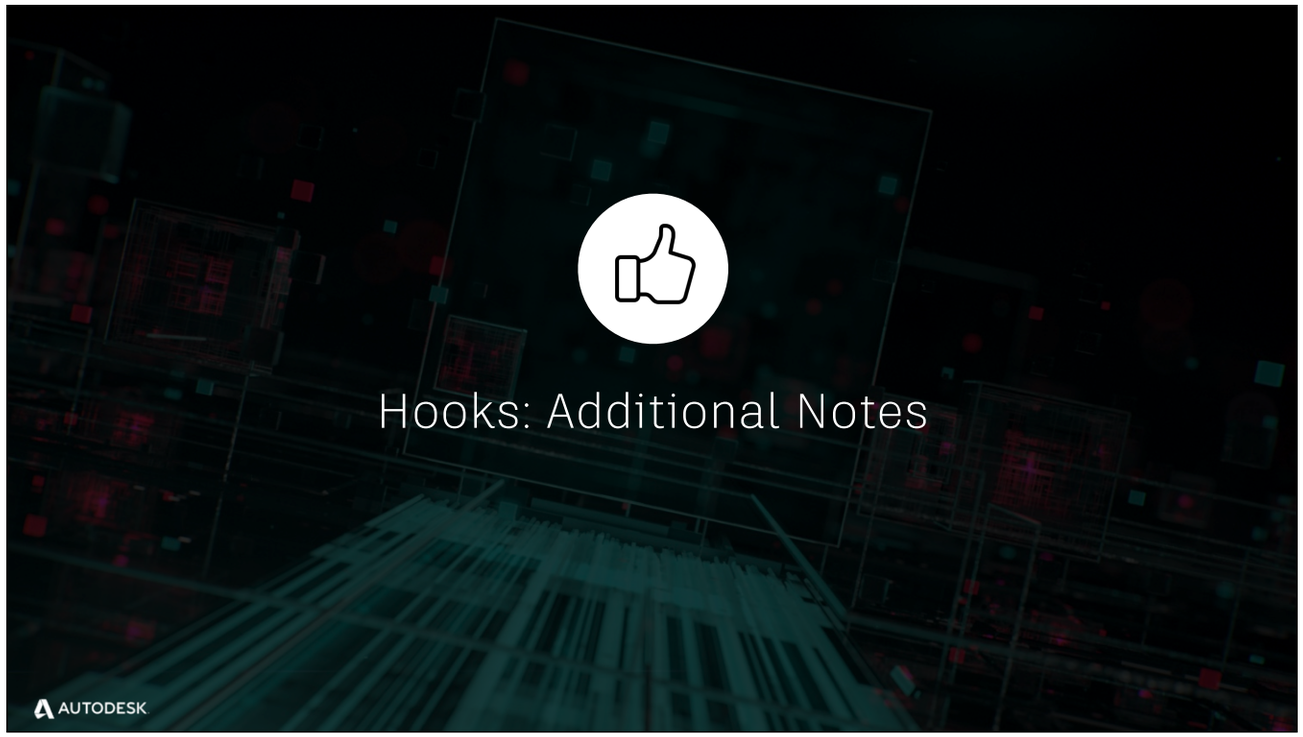
```python
import sgtk

# get the base class to use for the hook. this will process the hook
# specification in the config file and return the appropriate base class
HookBaseClass = sgtk.get_hook_baseclass()

class EnsureTaskPublishPlugin(HookBaseClass):

    def validate(self, settings, item):

        if not item.context.task:
            self.logger.error(
                "Publish item '%s' does not have a task associated. "
                "Please select a task on the right for this item." % (item,)
            )
            return False

        return super(EnsureTaskPublishPlugin, self).validate(
            settings,
            item
        )
```

AUTODESK

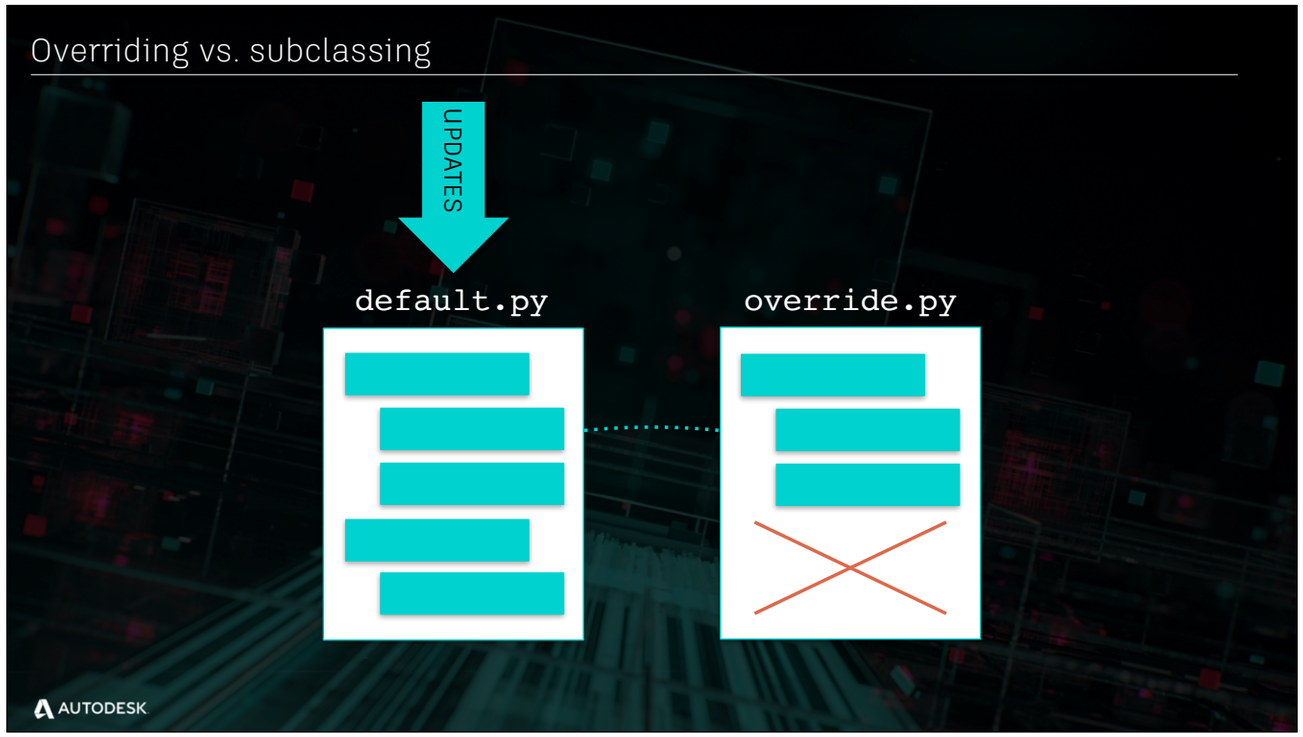
- As with the loader example, we need to define our subclass, making sure to inherit from the correct base
- We’re overriding the validate method here
- All we’re doing is checking to see if the item’s context is defined
- If not, log an error
- If so, continue with the base class’s validation logic



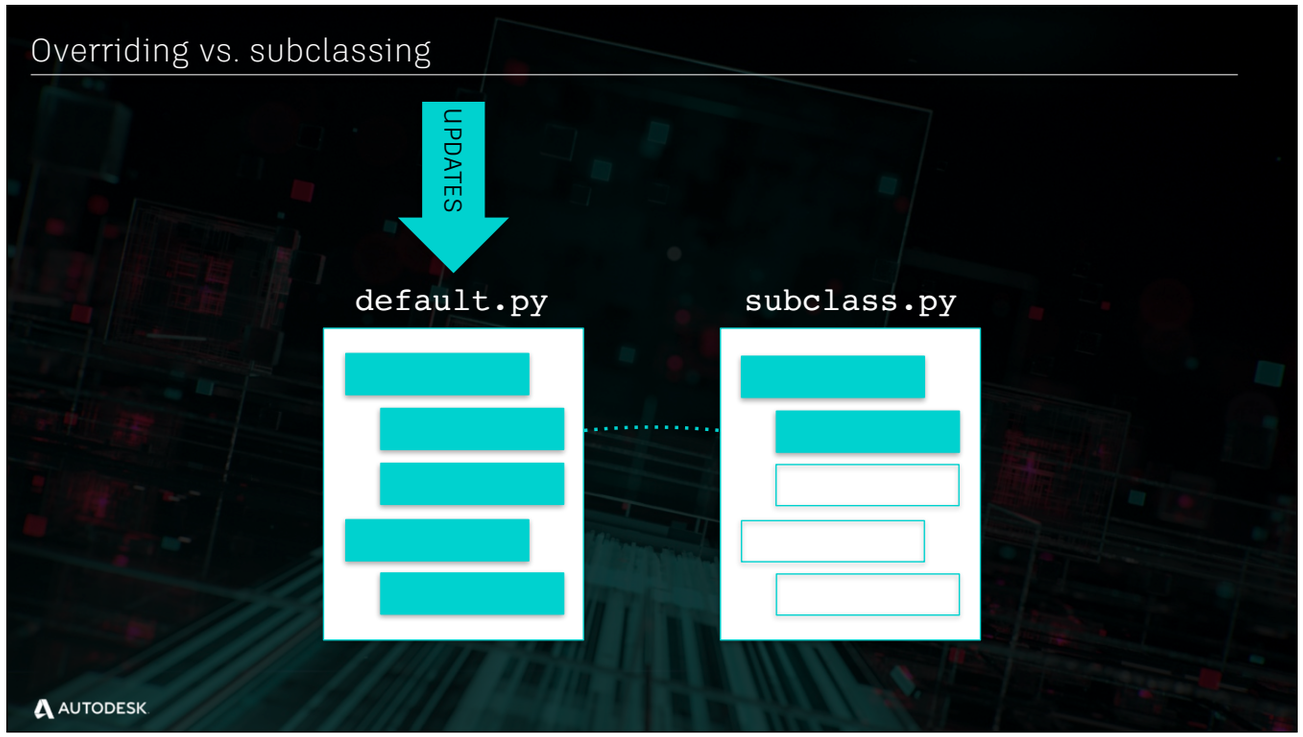
- Here’s a video that shows the results of the changes we made
- And you can see the validation fails if there is no task set
- Once we set the task, the validation is successful



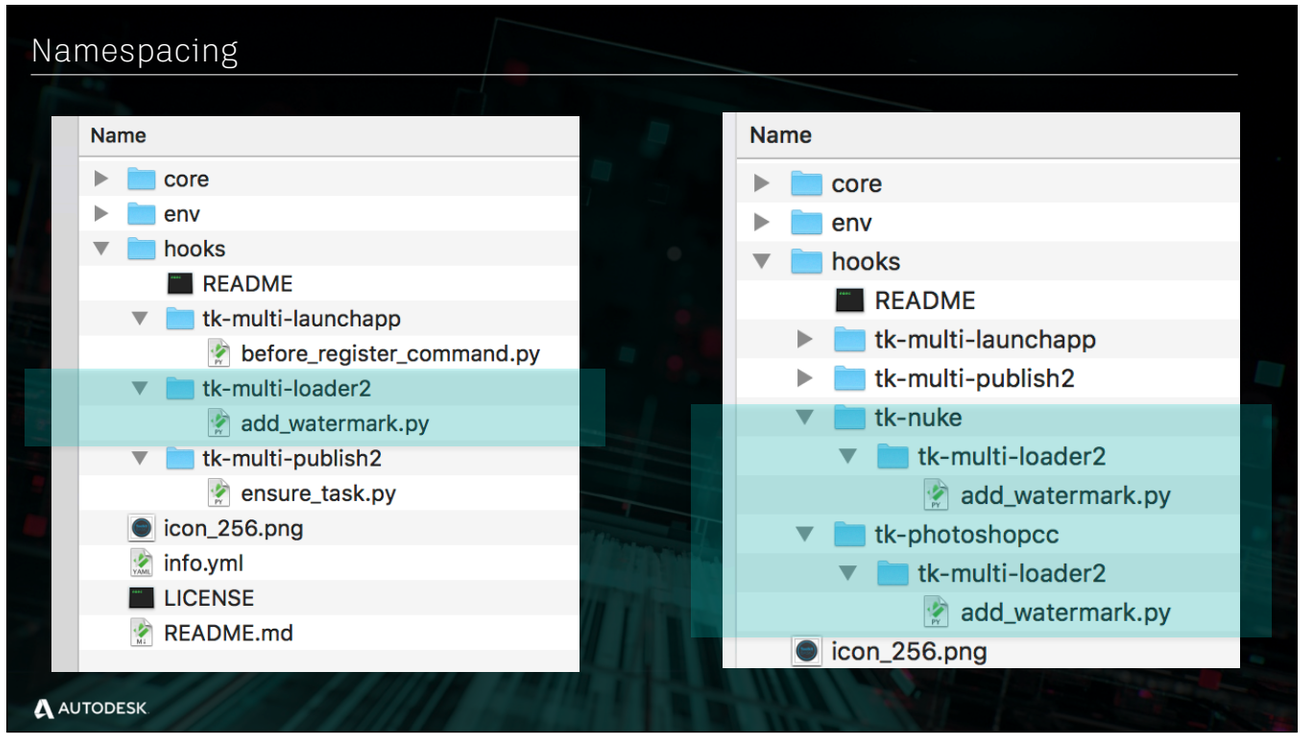
- So, even though these examples were fairly minimal, hopefully you can start to get an idea of how powerful hooks are and how you can really get the existing toolkit apps to behave in a way that works for your studio
- To wrap up this session, I just wanted to point out a few additional things about hooks



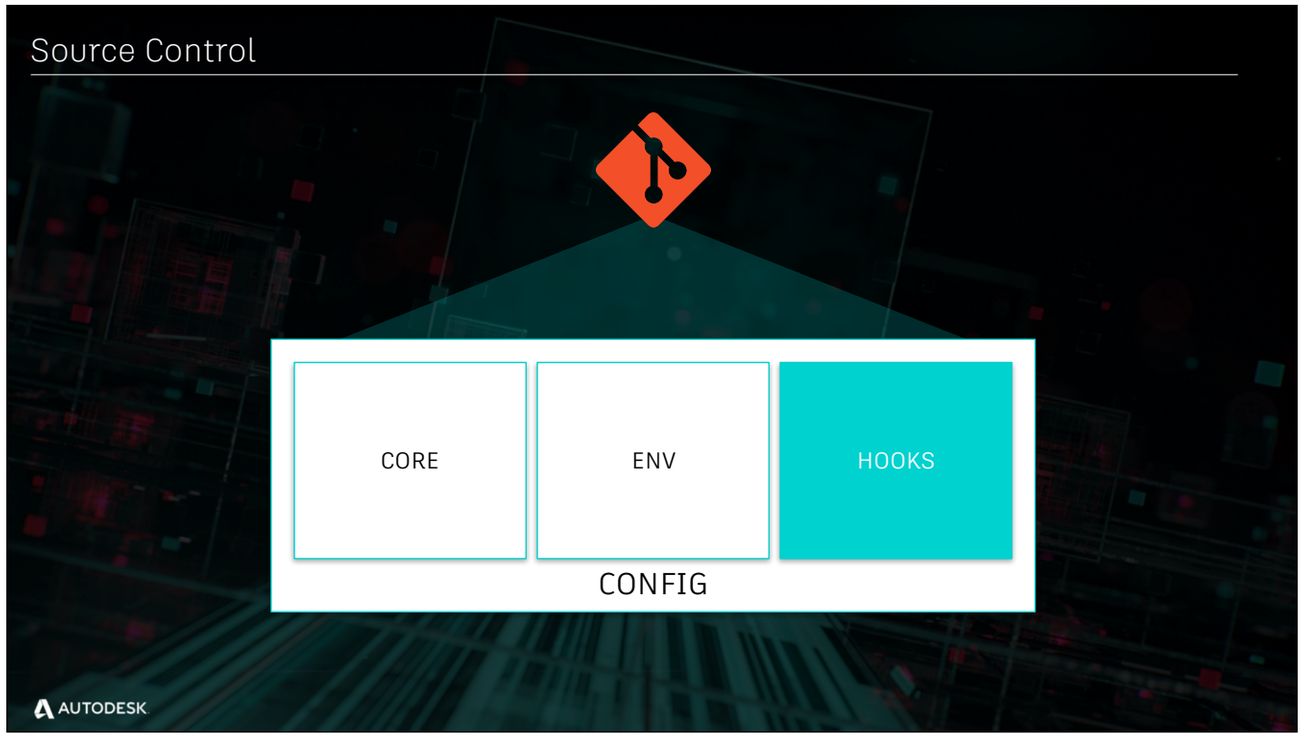
- Overriding vs. subclassing
- When it comes time to alter the behavior of a hook, it’s important to know what makes sense for you in terms of how you write your hooks
- Let’s take an example hook called default
- CLICK
- If we completely override that hook in our configuration, copying over all the source and replacing the reference, we can still get the behavior we want
- If updates are made to the app however, the overridden hook will not receive those updates
- CLICK
- Now that may be perfectly fine. If you’ve completely changed the behavior of the app or you don’t want updates this is fine



- On the flip side, if you subclass your hook and inherit the existing behavior, maintaining the reference to the original
- CLICK
- Then, when updates come, you'll inherit that new behavior as well
- CLICK
- I know these are pretty simple visualizations, but for folks that are new to toolkit or aren't programmers by trade, hopefully this gives you a good understanding of the difference here

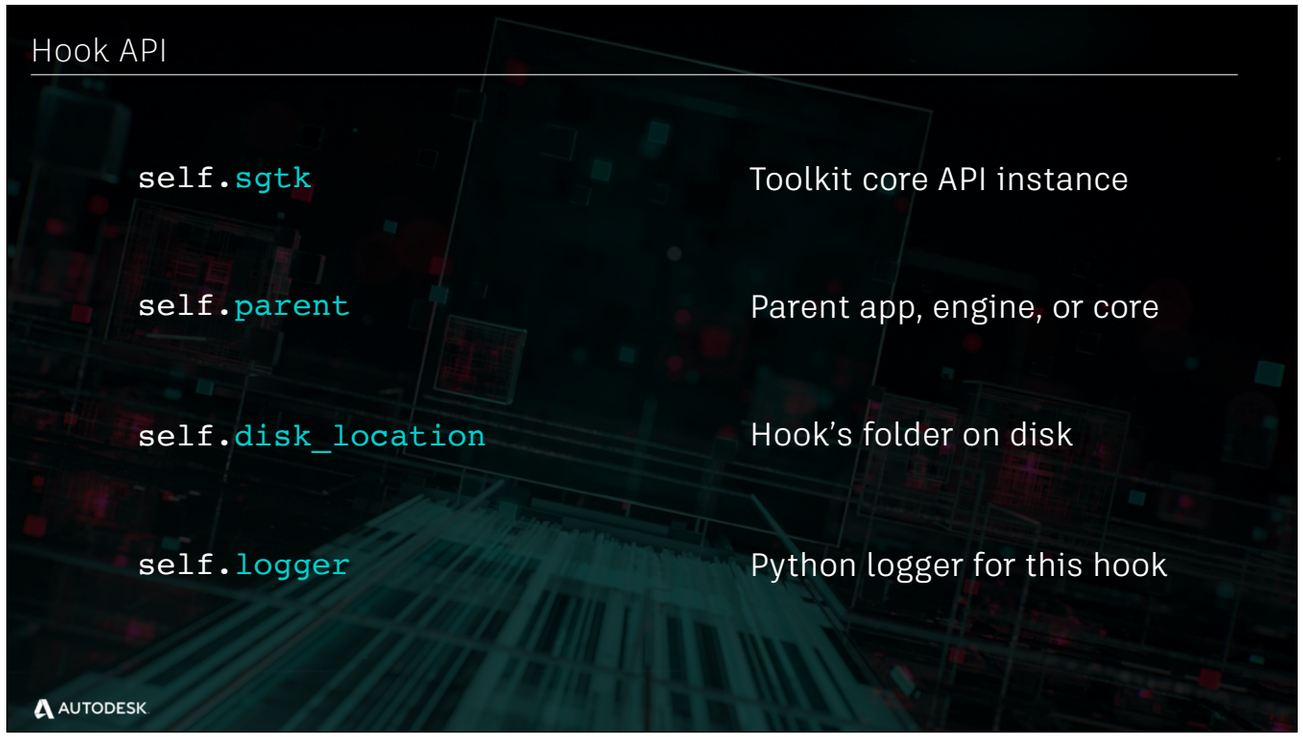


- Namespacing
- It’s important to namespace your hooks
- Toolkit doesn’t really care what you call your hooks so long as you reference their location properly in the config
- In the examples we looked at, we just dropped the hooks into a folder with the name of the app
- CLICK
- But what if we wanted to add similar watermark behavior to the loader in Nuke?
- Sometimes it can be useful to include more directory structure in your hooks folder to make it clear what the scope of the hook’s behavior is
- CLICK
- The structure is up to you, but because Toolkit apps are often used in multiple DCCs, it is important to keep your hooks organized in a way that keeps things clear in your config.



- Source control
- It’s a good idea
- Hopefully you keep your configuration under source control, including your hooks if you find yourself reusing the same ones across projects
- There are other ways to keep your hooks under source control outside of your configuration
- See the developer docs for more info there

https://support.shotgunsoftware.com/hc/en-us/articles/219033168-Configuration-staging-and-rollout
https://developer.shotgunsoftware.com/tk-core/core.html#sgtk.Hook



- I also wanted to point out a couple of properties available to you when writing or modifying a hook
- Each hook class derives from a common base that defines these properties
- So you always have access to them
- CLICK
- There’s the sgtk property which gives you a reference to the core API
- CLICK
- The parent property will give you the parent of the current toolkit bundle
- So if you’re writing an app, the hook’s parent will be the app
- CLICK
- The disk_location property will give you the folder on disk where the hook lives
- This is useful if you have associated files that travel with you hook (publish plugins + icons is one example)
- CLICK
- And there’s the logger property which gives you a logger instance that you can use for debugging for example

Importing Frameworks

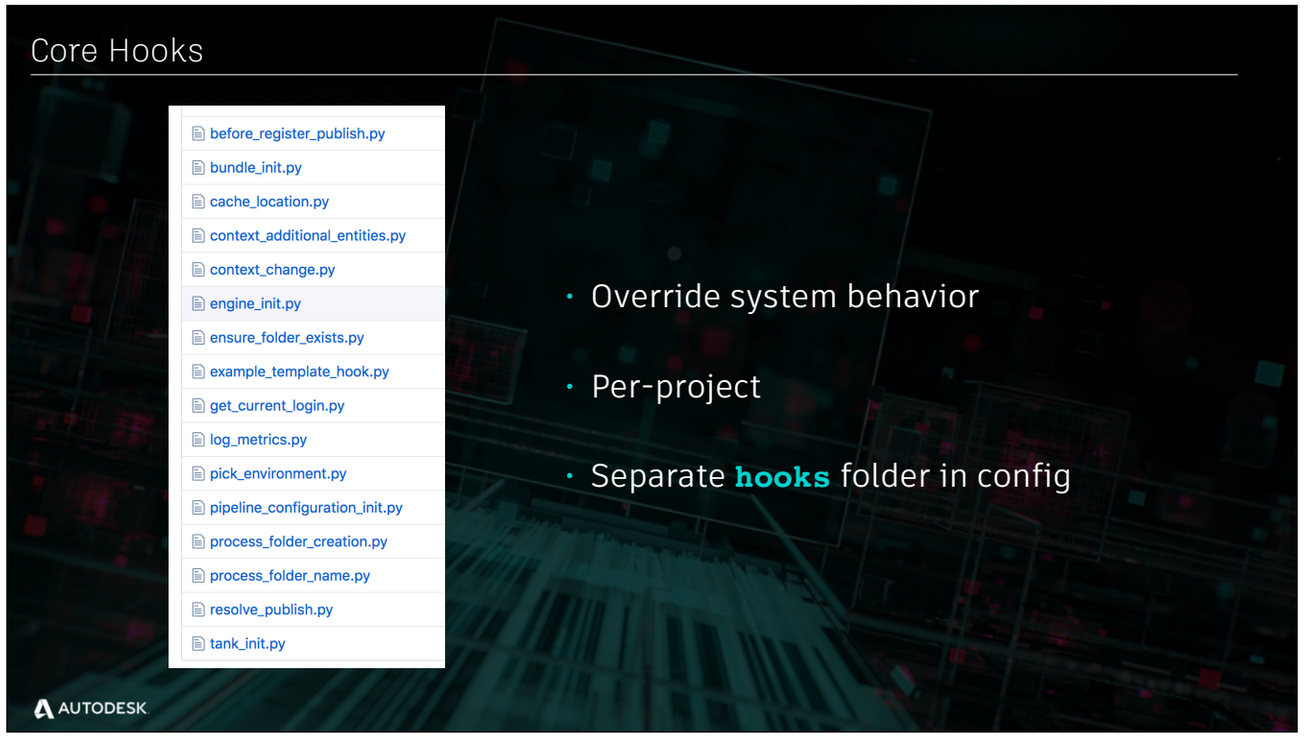
```
# load a required framework
fw = self.load_framework("tk-framework-qtwidgets_v2.x.x")
```

```
31    # qtwidgets - collection of Toolkit related QT Widgets
32    tk-framework-qtwidgets_v2.x.x:
33      location:
34        version: v2.8.1
35        type: app_store
36        name: tk-framework-qtwidgets
```

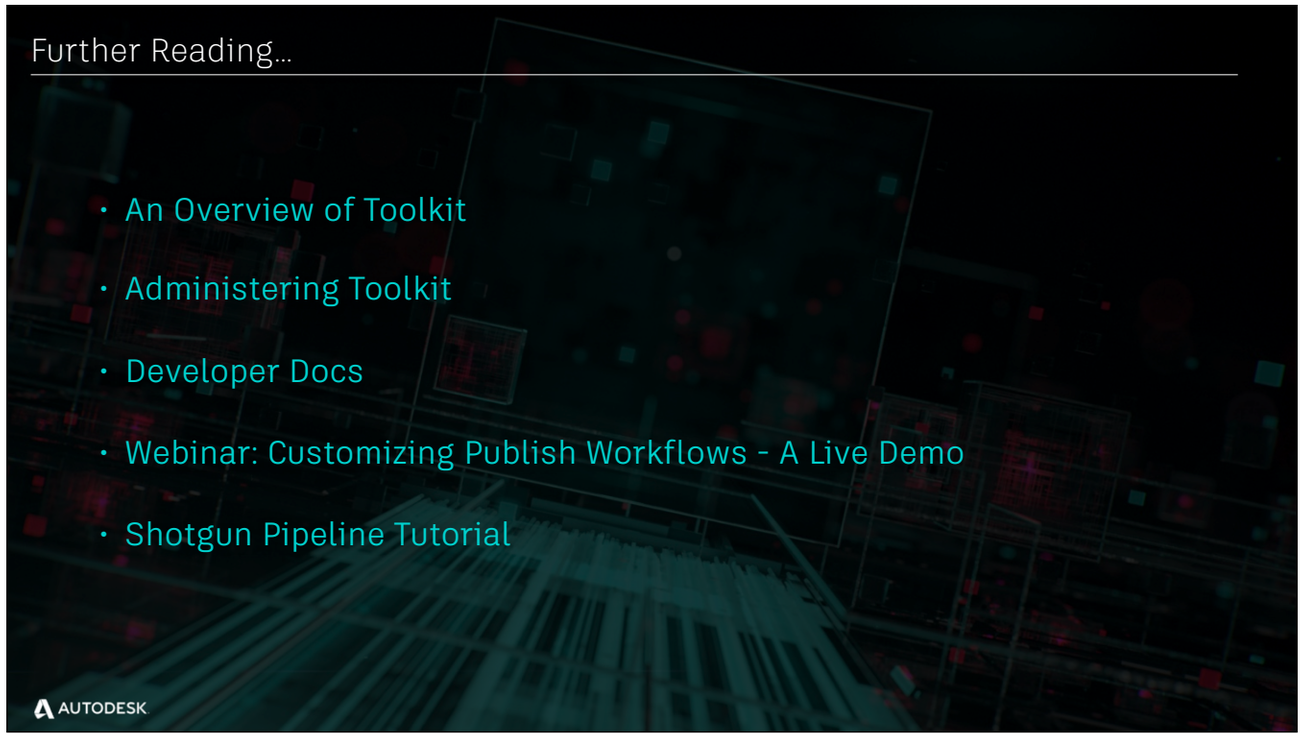
```
# import a module defined by the framework
sg_fields = fw.import_module("shotgun_fields")
```

AUTODESK

- Another good thing to know about is the load_framerork method available to the hook
- This allows you to load and import code that is managed externally, and typically reusable across apps within your hook
- This requires that the framework is required by your app and configured to be available in the environment the app is running
- CLICK
- Here you can see the framework as it is configured in the environment
- CLICK
- Note the long name with the version string appended
- This is the string you want to pass to the load framework method
- This is what it will be called once the engine is running and how this method does the lookup
- One the framework is loaded
- CLICK
- You can import any module defined by the framework and begin using it
- Note this example is using qtwidgets framework, but this could just as easily be your own custom framework that manages code reused by your studio's custom apps



- Finally, I wanted to point out Core hooks
- There are hooks that can alter the behavior of toolkit itself, not just an individual app
- These are defined per project and live in a separate hooks folder in your configuration
- You can find out more about these hooks in the tk-core repo itself on GitHub or in the Toolkit documentation

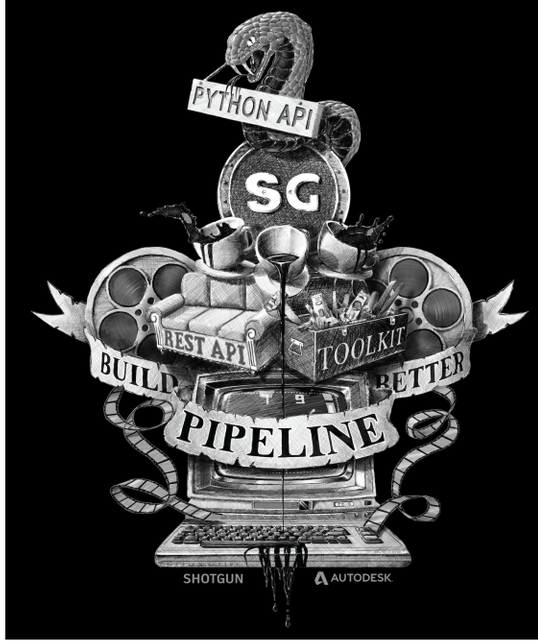


- An Overview of Toolkit: Hooks - support.shotgunsoftware.com/hc/en-us/articles/219040648-An-overview-of-Toolkit#Hooks
- Administering Toolkit: Hooks - support.shotgunsoftware.com/hc/en-us/articles/219033178-Administering-Toolkit#Hooks
- Developer Docs: Hooks - developer.shotgunsoftware.com/tk-core/core.html#hooks
- Shotgun Toolkit Webinar: Customizing Publish Workflows - A Live Demo - youtube.com/watch?v=pH4mylrnktY
- Shotgun Pipeline Tutorial - support.shotgunsoftware.com/hc/en-us/articles/219039938-Pipeline-Tutorial


Questions?
AUTODESK

ADVANCED SHOTGUN DEVELOPMENT

1:30 - 3:00PM



PRESENTED BY


Manne


Jeff


Josh

DEV DAY RESOURCES

WHAT WILL I LEARN?

- How to develop, package up, and distribute toolkit configurations to a remote user base
- How to develop and distribute custom toolkit apps
- How to leverage the standard Toolkit frameworks for UI and data management
- What the Autodesk Forge ecosystem is and how it can be used with Shotgun