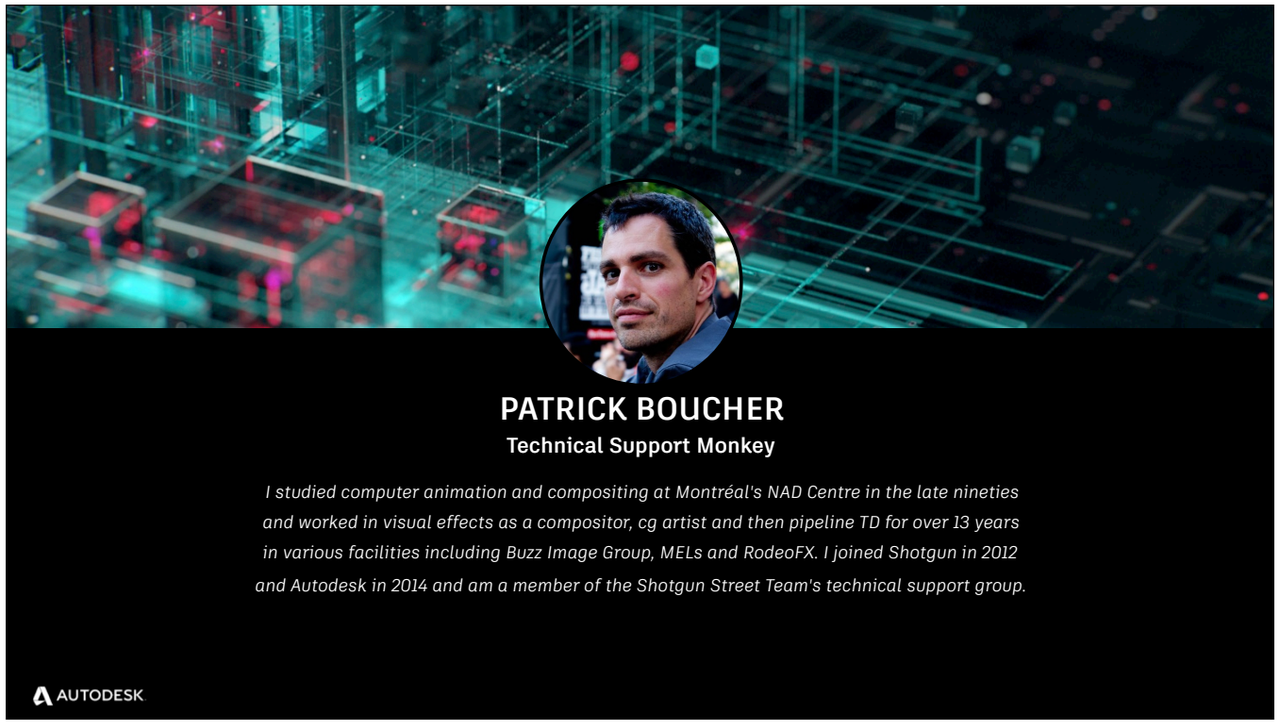
Hi everyone, in this talk we’re going to discuss data management in Shotgun using its APIs.



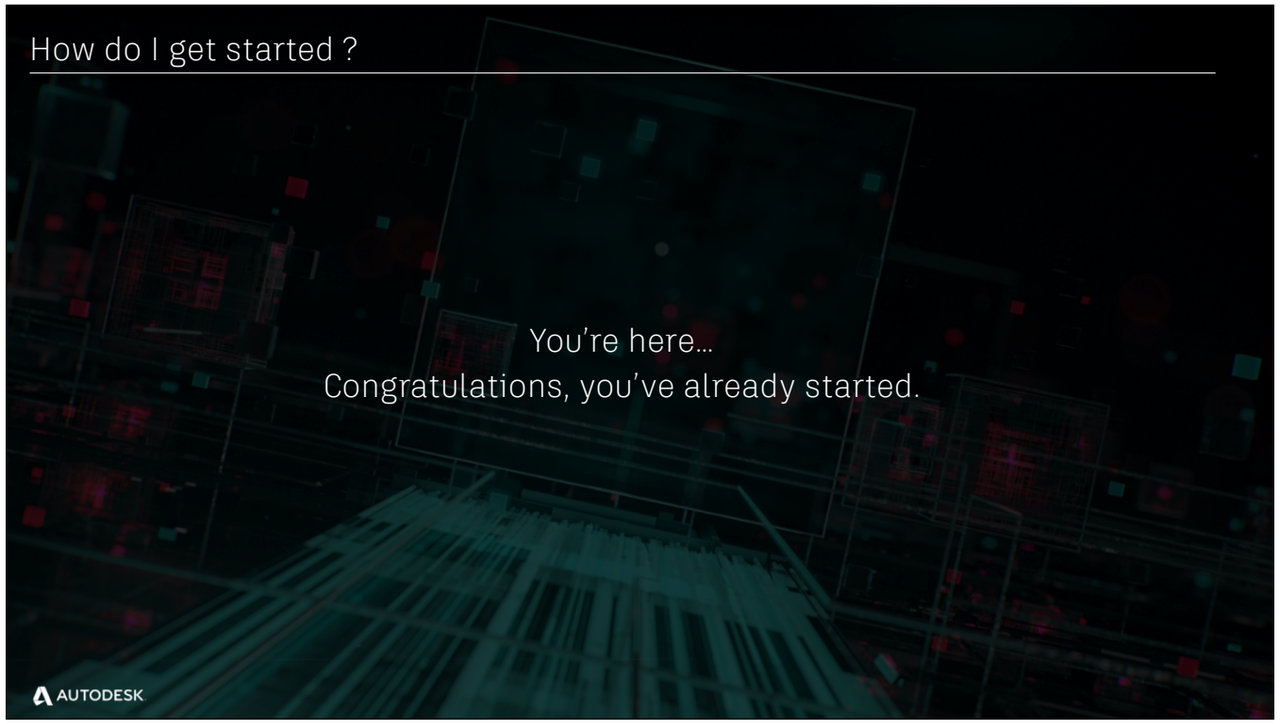
My name is Patrick Boucher and I’ve been with Shotgun for 7 years most recently as a technical support engineer. Prior to that I was a pipeline engineer for various facilities for 13 years.

The material from this talk will be available online so no need to take pictures of the slides… except this one. Take a picture of this slide, now.

Getting ready to chat

https://app.sli.do/event/10wljptx

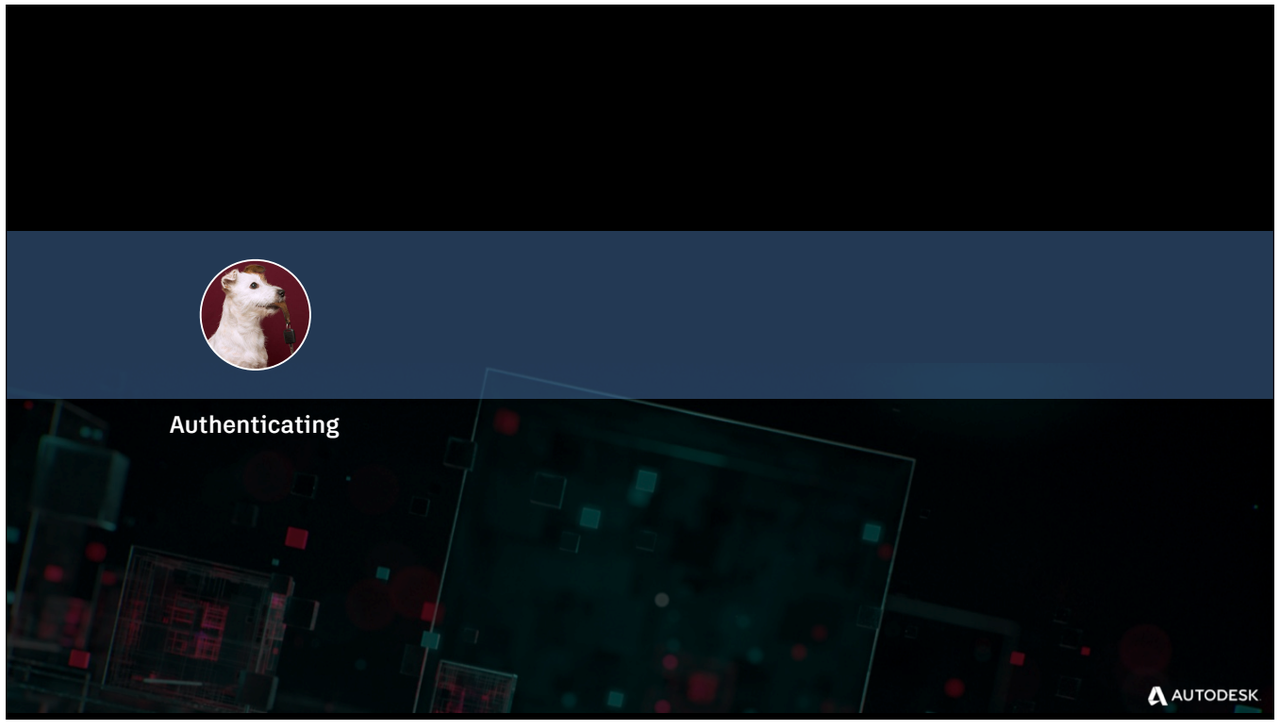
AUTODESK



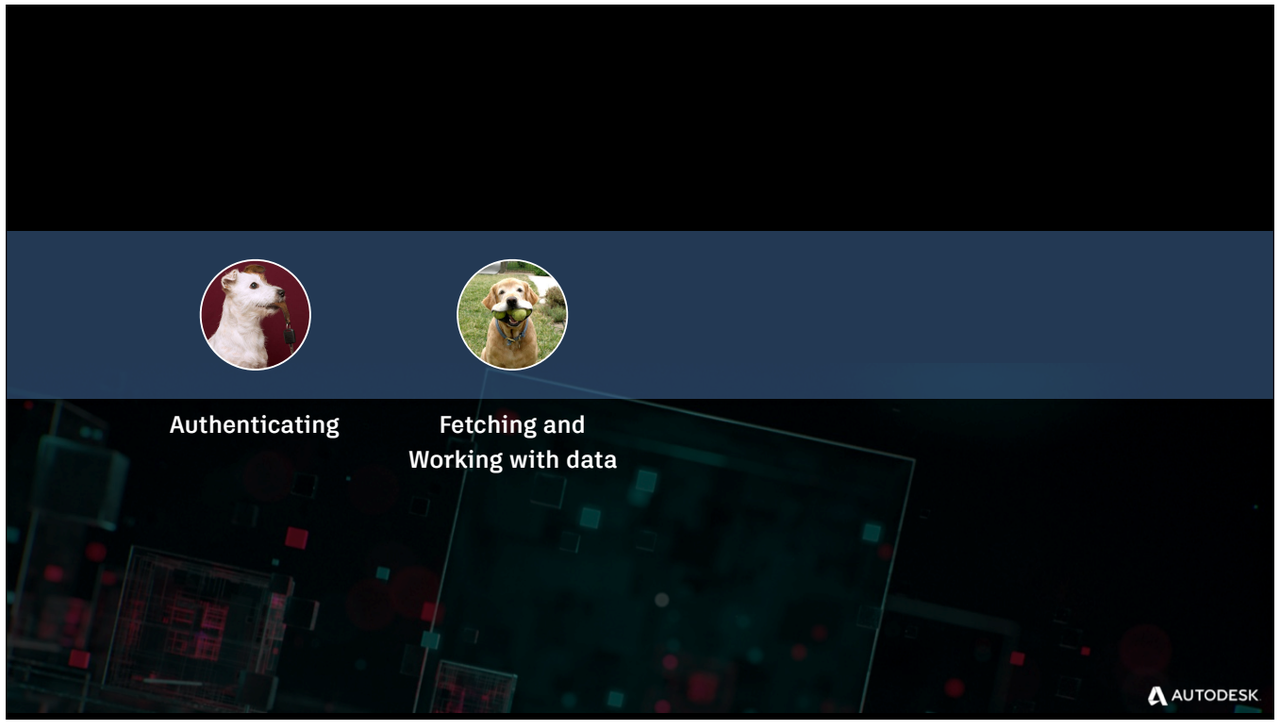
- When people are new to programming with Shotgun they often wonder how and where to get started...
- Well... You’re here, so you’ve already taken the best first step you could. Actually, where I like to suggest people start is with the Python API.

The Python API is definitely some of what we’ll be looking at today but we’ll also cover some other topics and some useful general Shotgun tips and tricks.

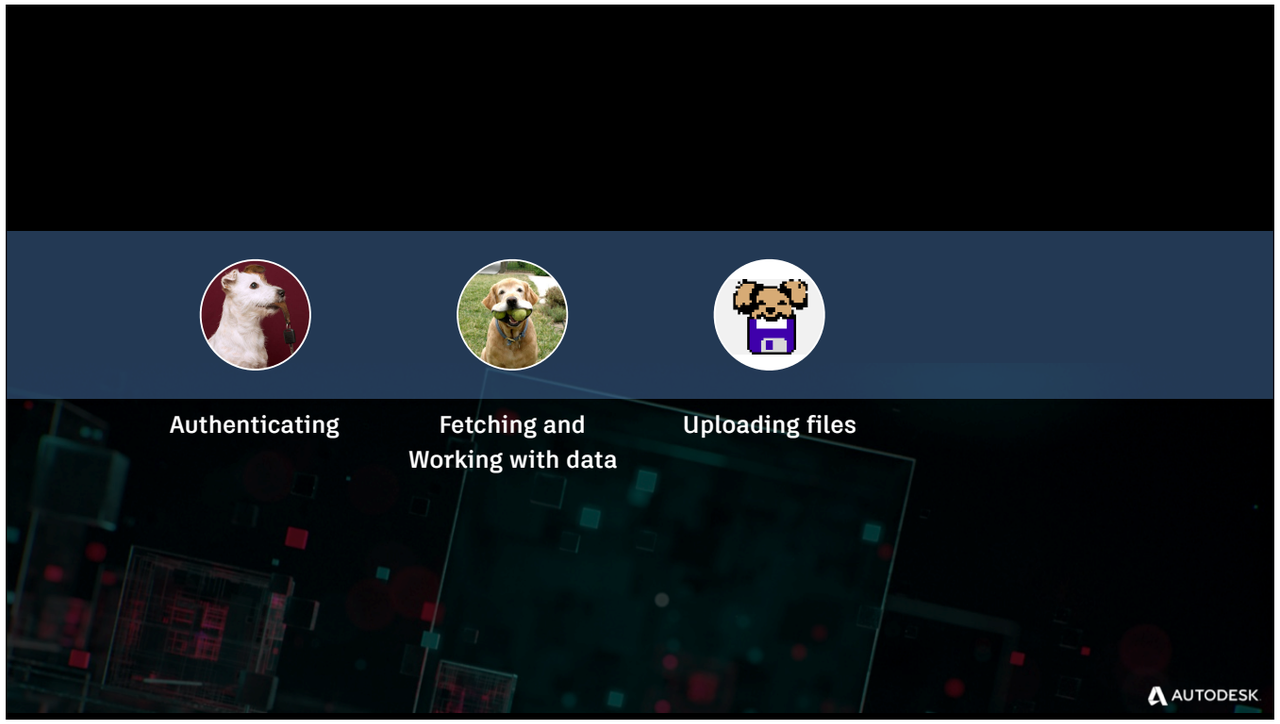
Here’s what we’ll be looking at and what I hope your main takeaways from this talk are going to be.



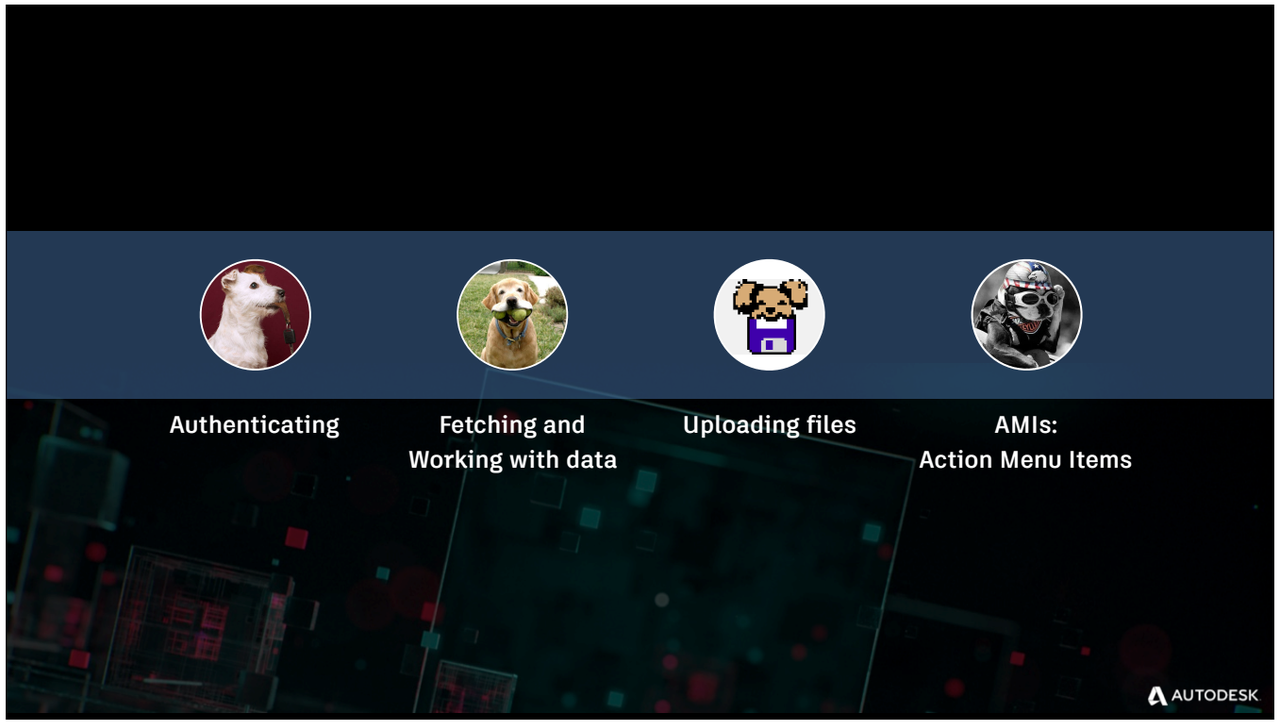
- We’ll be looking at how to setup a script entity in Shotgun and connect via the Python API...



- Querying, creating and updating data
- What are entity dictionaries and the special name key in linked entity dictionaries
- Linked field syntax (aka bubbled field syntax)



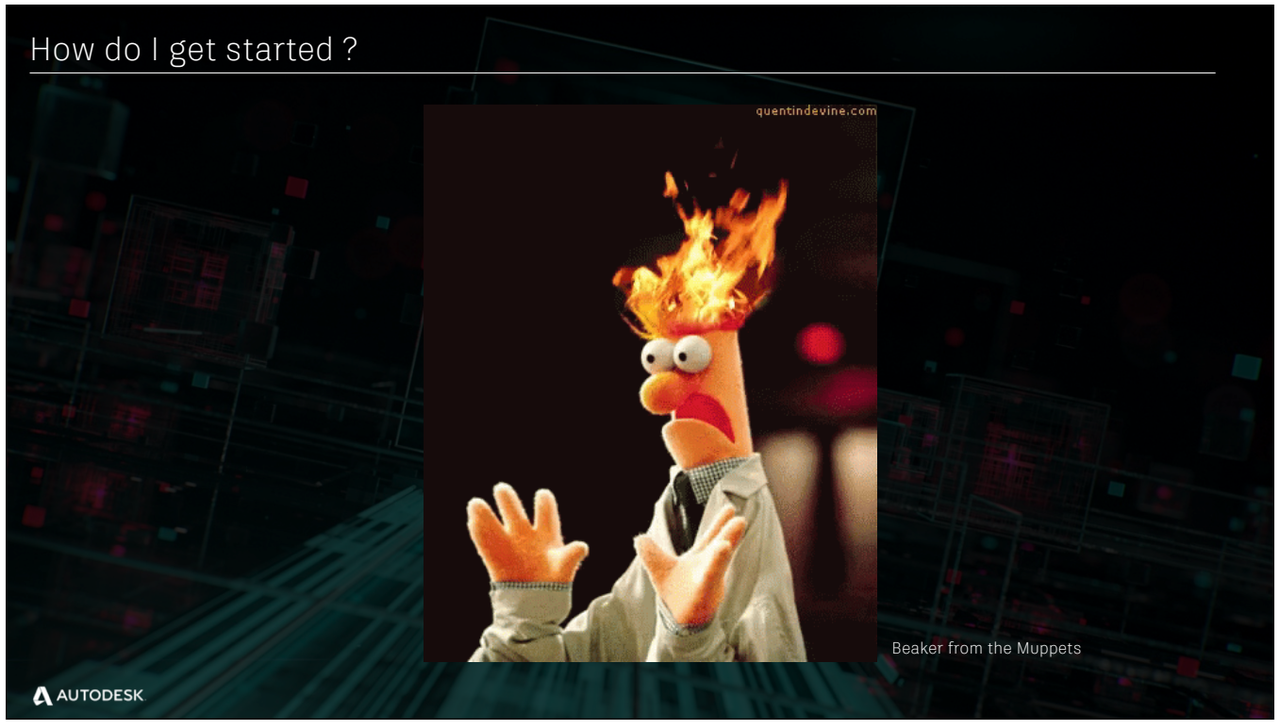
- Uploading via the Python API



- How AMIs work, how to set them up and how to use Flask to write a simple one. Here we'll also be looking at the REST API.

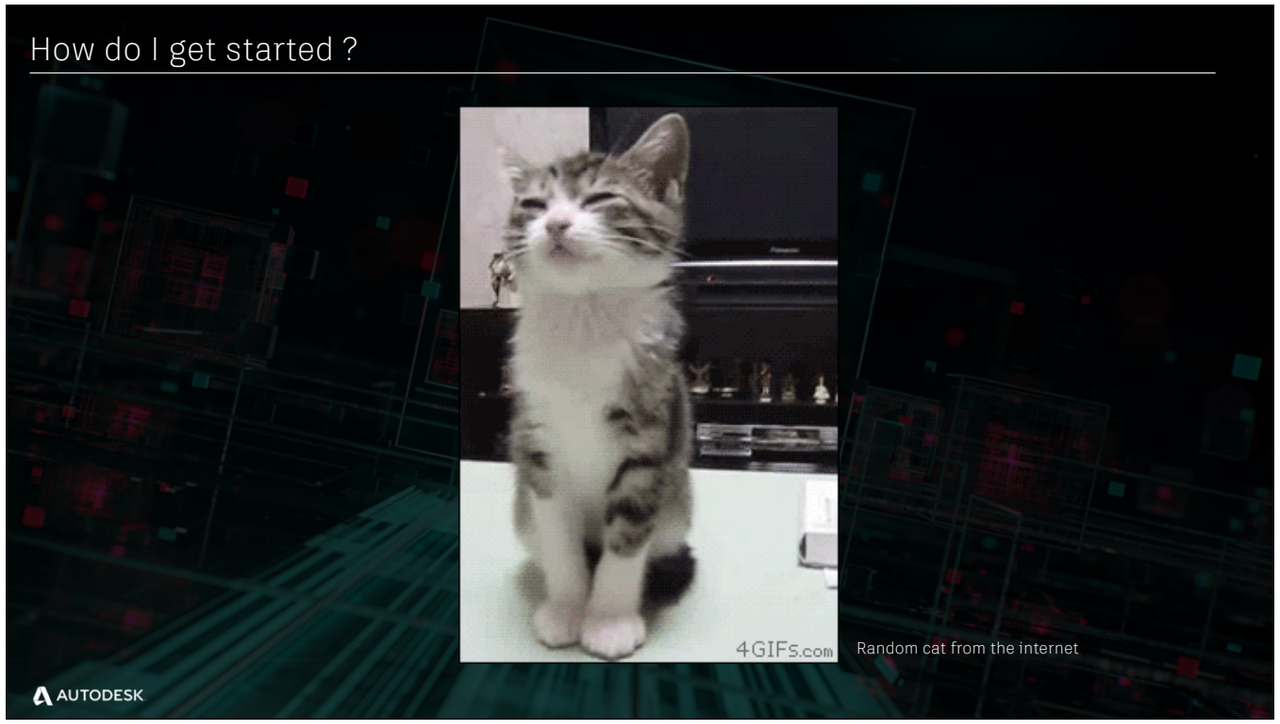
Uhmmmm... Concerning that last icon... Did you know that AMI is also the acronym for the American Motorcycle Institute? So you get a puppy on a motorbike. I mean, finding a dog pic for AMI was hard. I'll give you this: it is a bit far fetched...

Hey, throw me a bone.
... Ruff crowd.

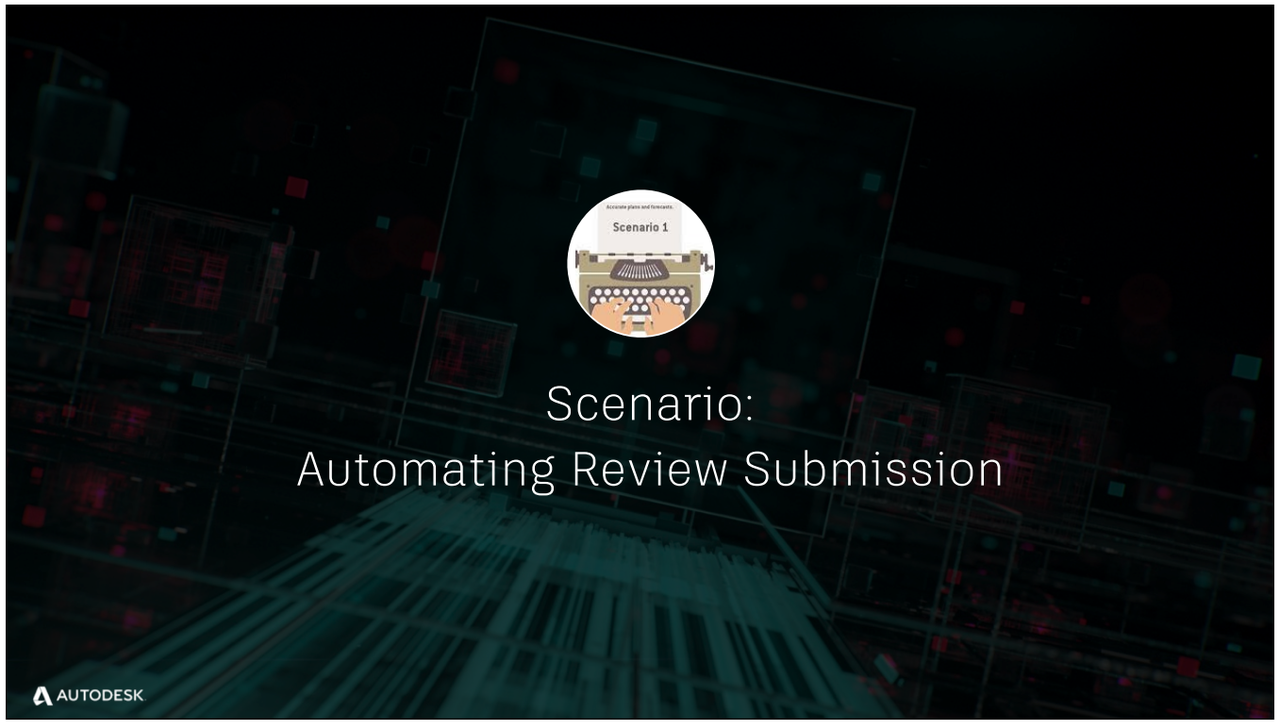


This is Beaker, from the Muppets.

If at any point you feel like this this picture of Beaker is you - that things are going too quickly and are getting out of hand, remember all the friendly Shotgun staff is here to help you, answer your questions and maybe even give you a hug if you need one. Also, come and join the community at community.shotgunsoftware.com .

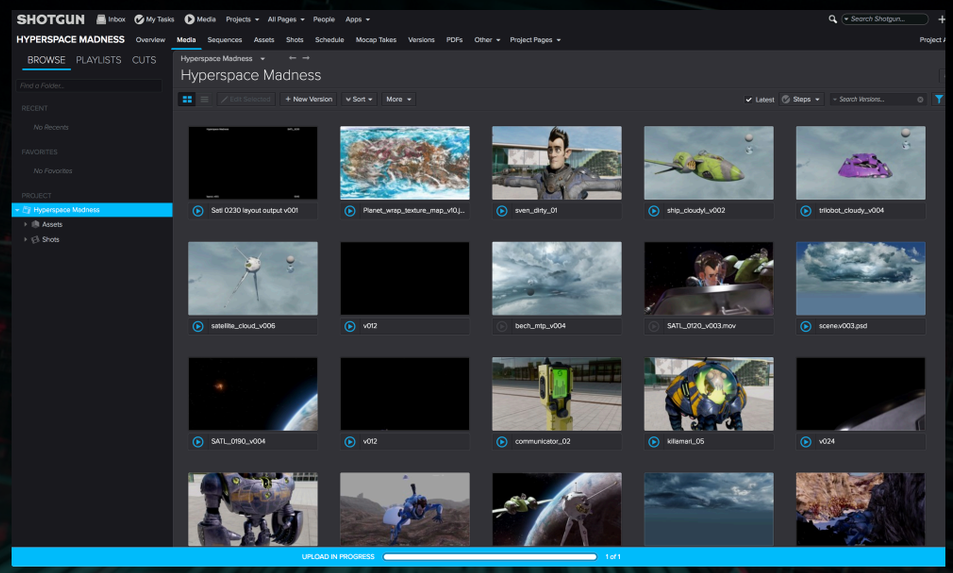


Alternatively, if you get bored and feel like this is you… Don’t worry… Brandon and Manne will be presenting advance development topics later so you’ve got more than enough time to start looking like Beaker.



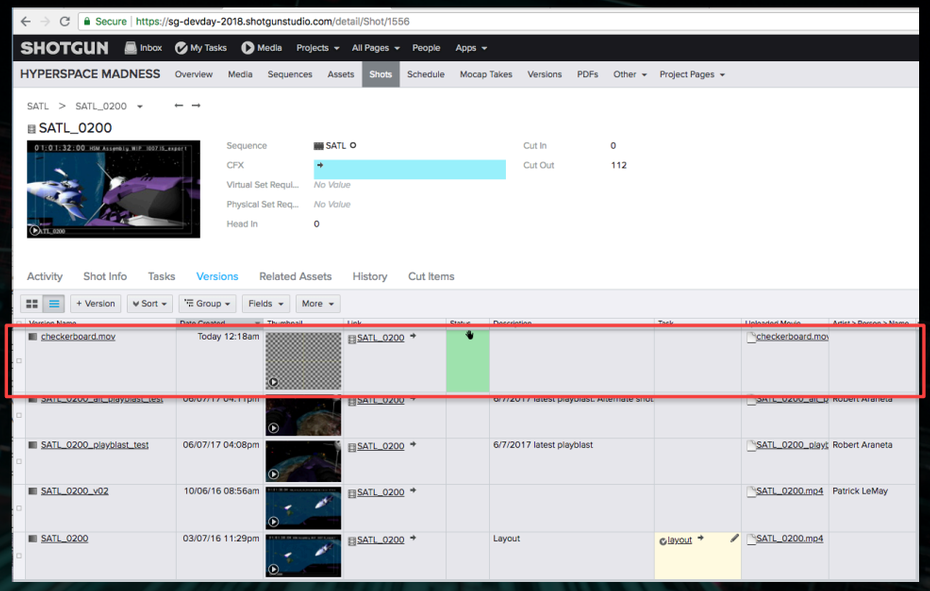
Alright, for the sake of argument, let’s imagine a scenario where you’re using Shotgun for your review process.

Automating Review Submission



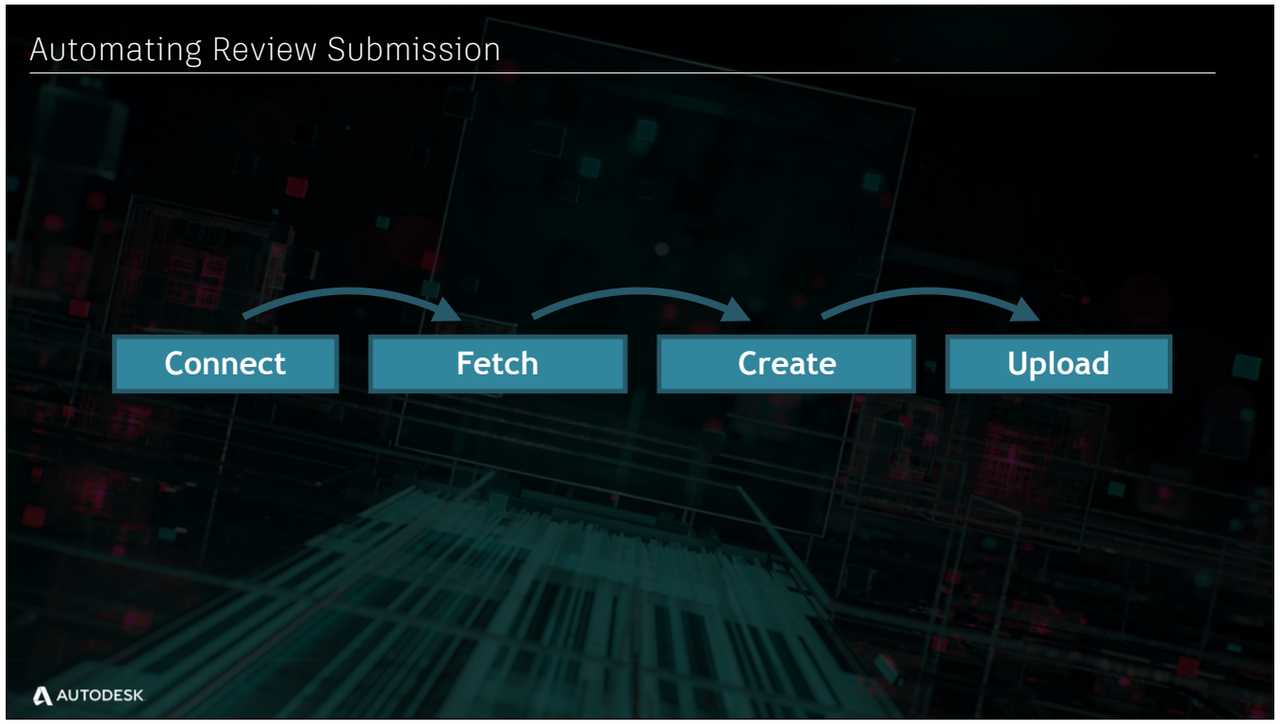
Usually version submission starts out with users uploading content manually via drag and drop in the web app. This can be pretty time consuming, especially if you have 100 versions to submit. This is where the API can come in handy as tools can be written to automate the review submission process. Our tool isn't going to be a complex one. We'll keep it to its simplest form possible.

Automating Review Submission

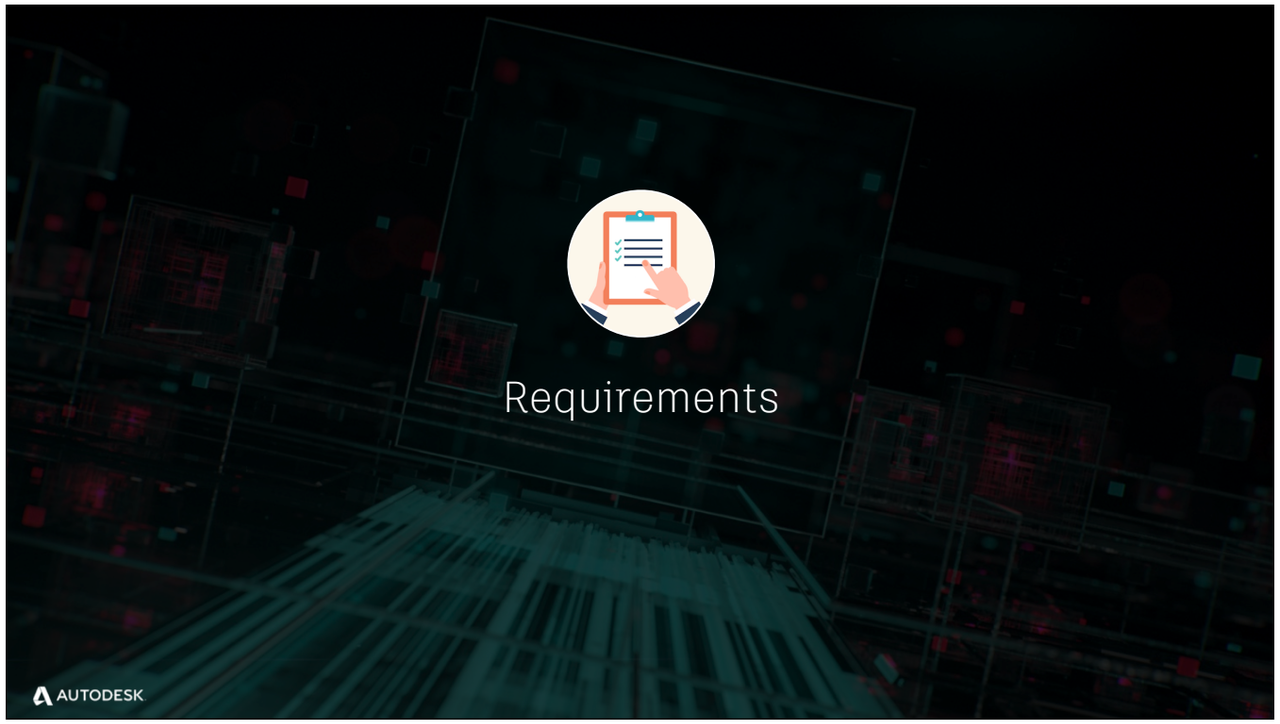


AUTODESK

Once the tool runs, here’s what we’ll be left with: A version with a clip uploaded to it. For this to work we’ll need to:

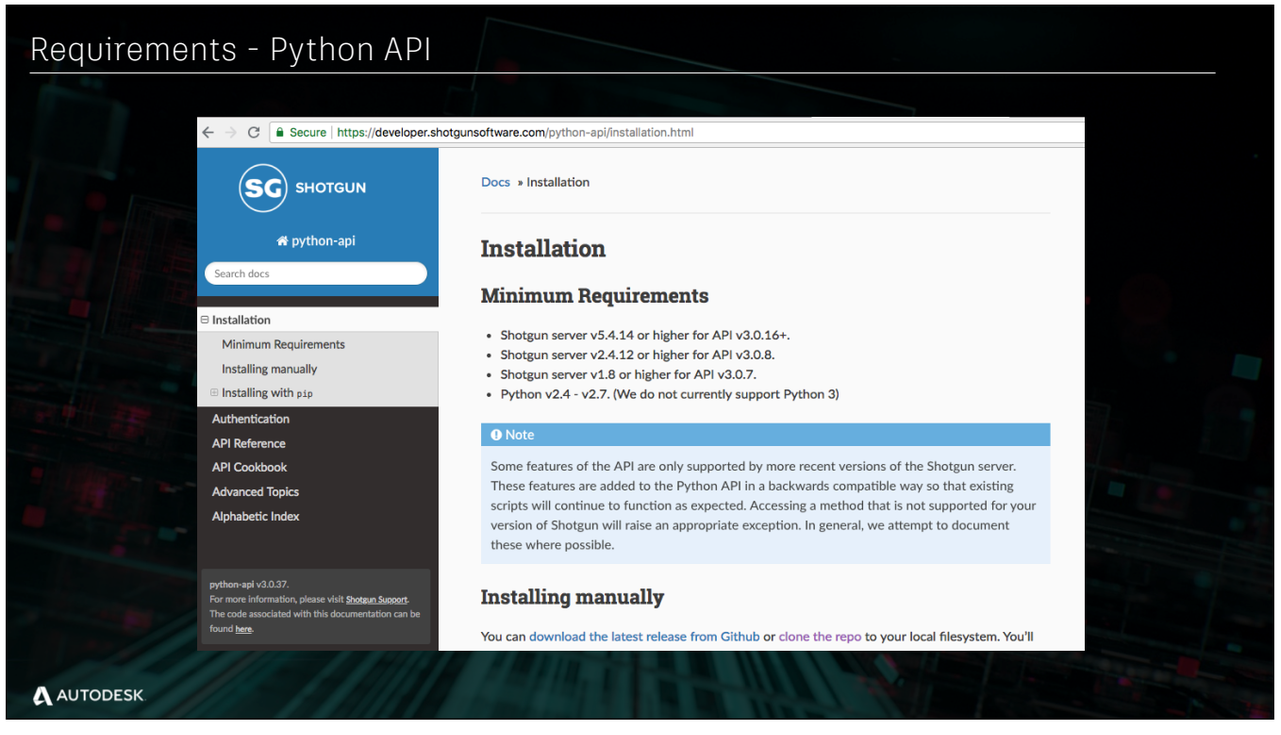


- Connect to Shotgun
- Fetch the project and shot the version will be linked to
- Create the version
- and finally upload the movie.



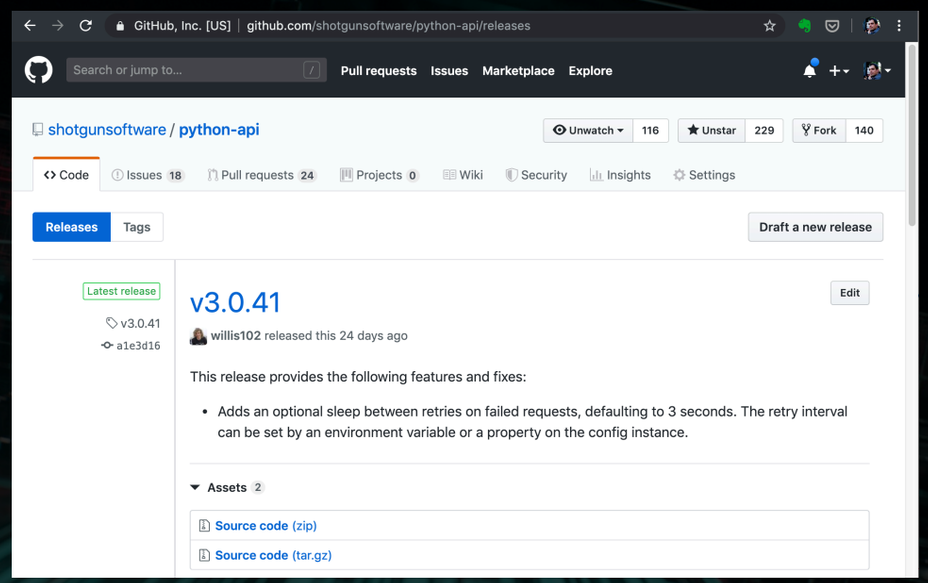
Let’s talk about what you’ll need to make this work.

- Obviously a key requirement for the Python API will be Python itself.
- If you’re on Windows you can opt for ActivePython which will come with pywin32 and a bunch of other goodies.
- If you’re using the REST API, any language with an HTTP or REST library will do. For Python users I like to recommend the requests library.



You'll also, obviously, need the Python API itself. You can get it via a link at developper.shotgunsoftware.com. developper.shotgunsoftware.com/python-api will take you directly to the Python API documentation.

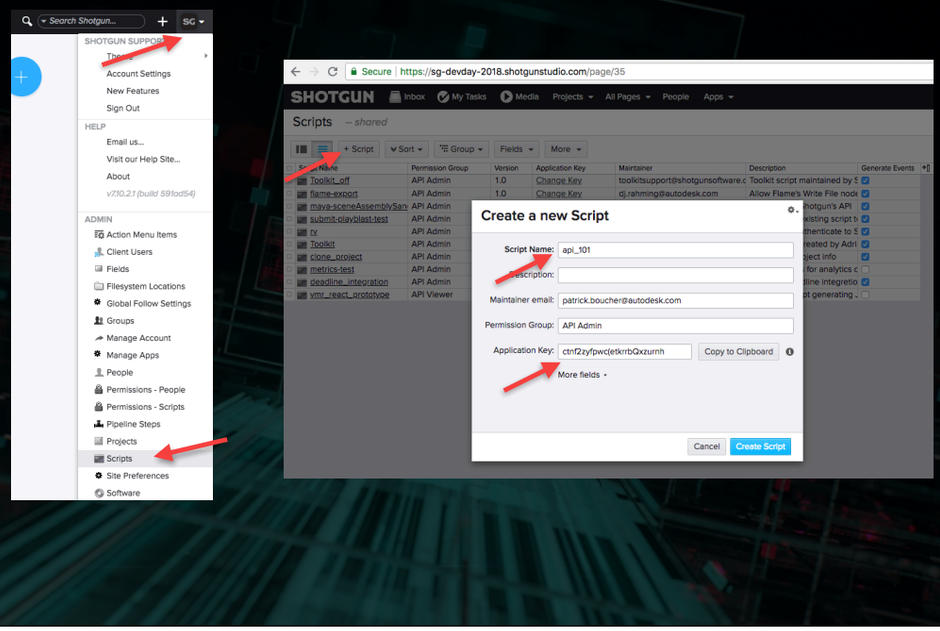
Requirements - Python API



AUTODESK

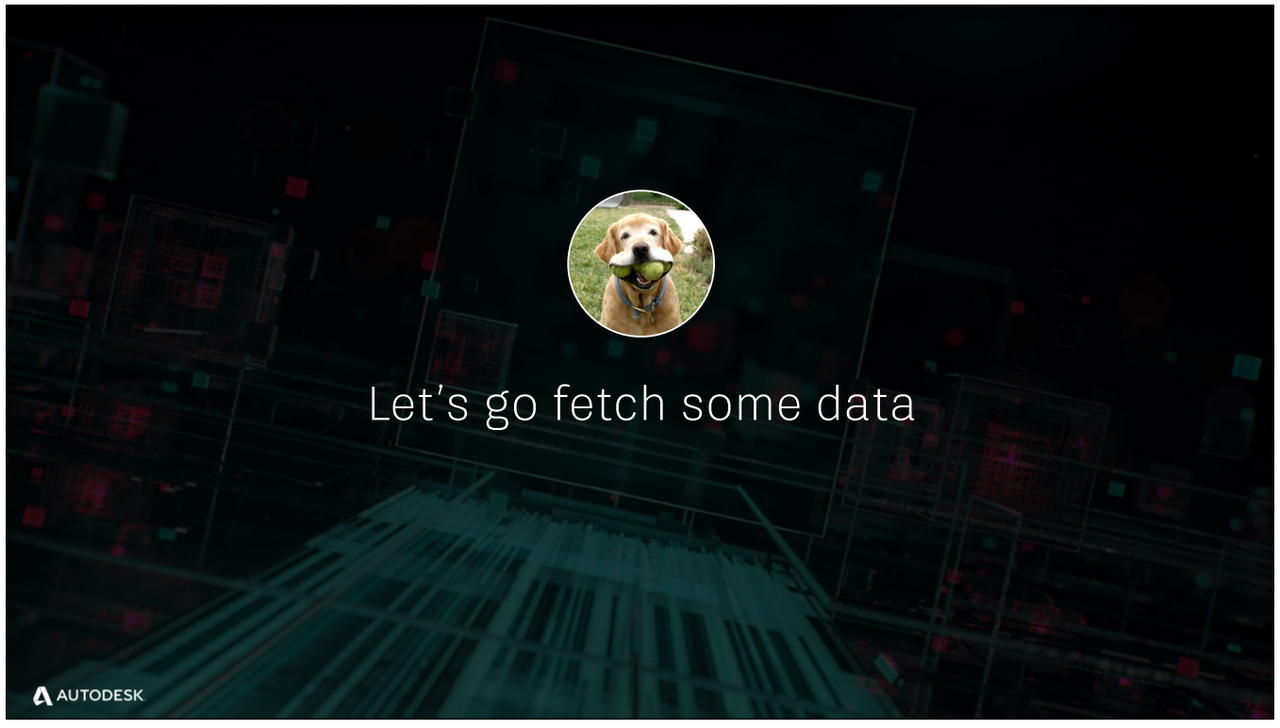
You can also go directly to GitHub where you can grab the latest tag. It’s really easy to download and install.

Requirements - Script Key



AUTODESK

- Finally, you'll need a script name and a script key. There are many ways to authenticate using the Python API but this is the simplest and the one you'll see here.
- To create your script key you'll press the user menu icon on the top right
- and you'll select "scripts" - you need to be an administrator for this.
- In the scripts page you can create a new script, give it a name and jot down in a secure location the application key
- You probably noticed that there are multiple script keys in the screenshot. Why do we have multiple script keys? Well, having more than one key, and preferably one key per tool or script, will allow for better control over your tools and better auditing of which tool performed which change in Shotgun. As your pipeline grows, troubleshooting it will be much easier if you have multiple keys instead of a single one.
- On the support team we see a lot of clients that create wrappers around the Shotgun API. This isn't a bad practice, however, including your authentication in the wrapper is. If you wrap the API, make sure that you provide a mechanism for each tool to supply their credentials and that you don't bundle all tools under a single set of authentication credentials in the wrapper.



OK. Let’s start our tool and go fetch some data with the API.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

This is the first part of our tool where we connect to Shotgun and go retrieve data.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

First off, in our imports we'll import the Shotgun API so that we can create a connection

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

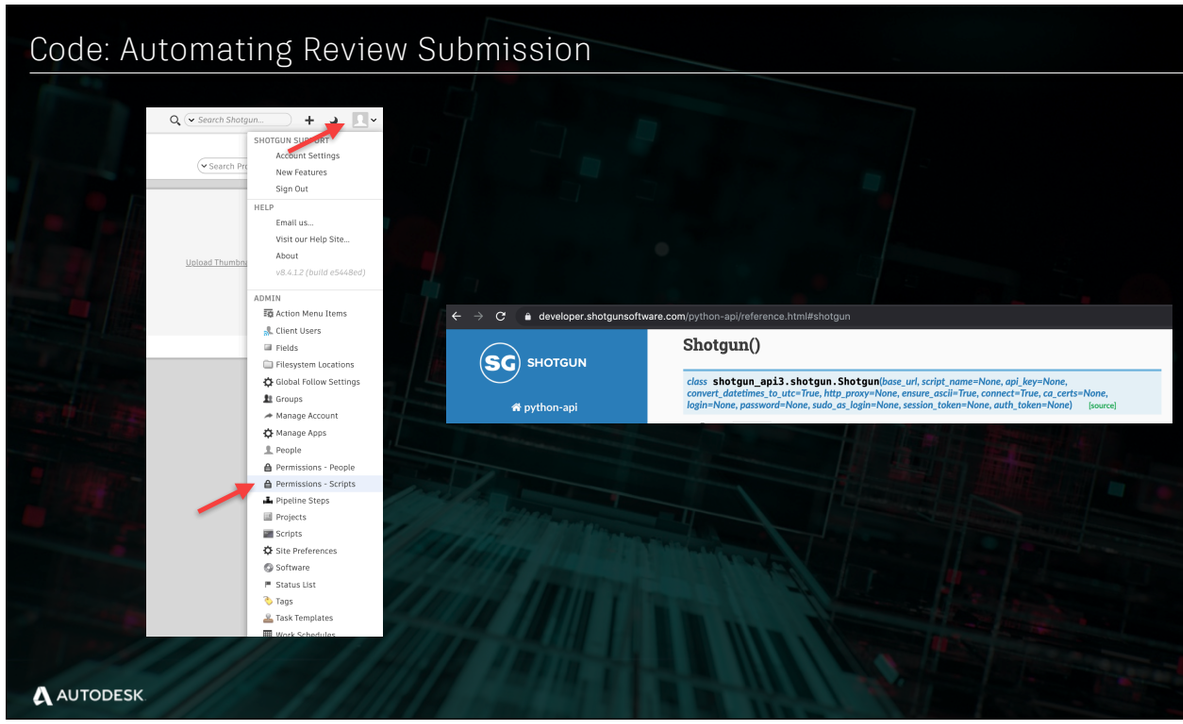
This second block of our code stores our script name and script key information that we created in Shotgun along with our site's URL. You should _not_ store this information directly in your script as this is not secure. Preferably use something like environment variables or other more secure techniques.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

Next we create a connection to our Shotgun site’s URL and authenticate using the script name and key.



If you use script key based authentication, please remember that you can tweak your script's permissions via the admin options in the user menu. This will allow you to control which script keys can access which entity types and fields in Shotgun and then you can provide the appropriate keys to your developers.

You can also authenticate with a normal user's username and password, with or without 2FA, with a session token and a bunch of other options but I'll let you read the documentation for that.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

Finally we get to the fun part. Reading data from Shotgun.
On this line we use the `find_one` method to fetch a single project whose name is `Hyperspace Madness`.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

The second argument of the method, known as the filter, is a list of conditions that define which project we're looking for. Here we're looking at an example of a basic filter. Each condition in the list has three parts:

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

the field we’re filtering on

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

the kind of filter, known as the relation...

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

and the value we’re looking for.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

This last bit of code uses the same pattern as above to find the shot to which we'll link our version. Notice that this time around we have two conditions in our filter. By default, the filter operator is an `AND` operator meaning that both conditions need to be true to find a Shot. This can be changed with an argument called `filter_operator` so that all conditions are `OR`ed together or you can even make complex filters with a mix of AND and OR operators between conditions.

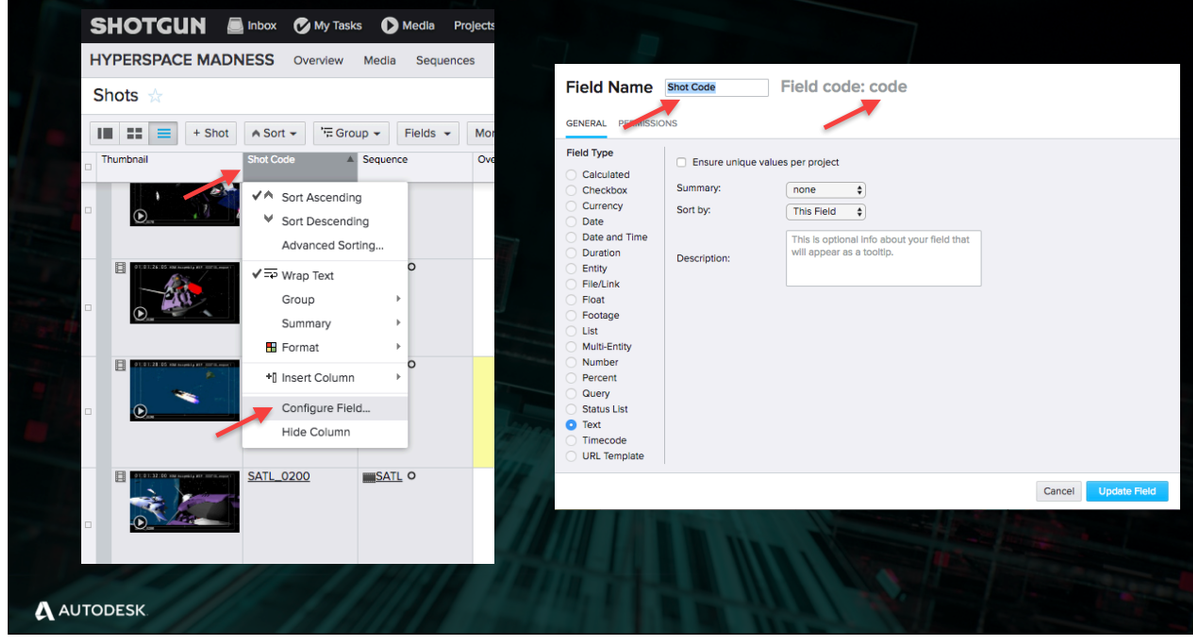
Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

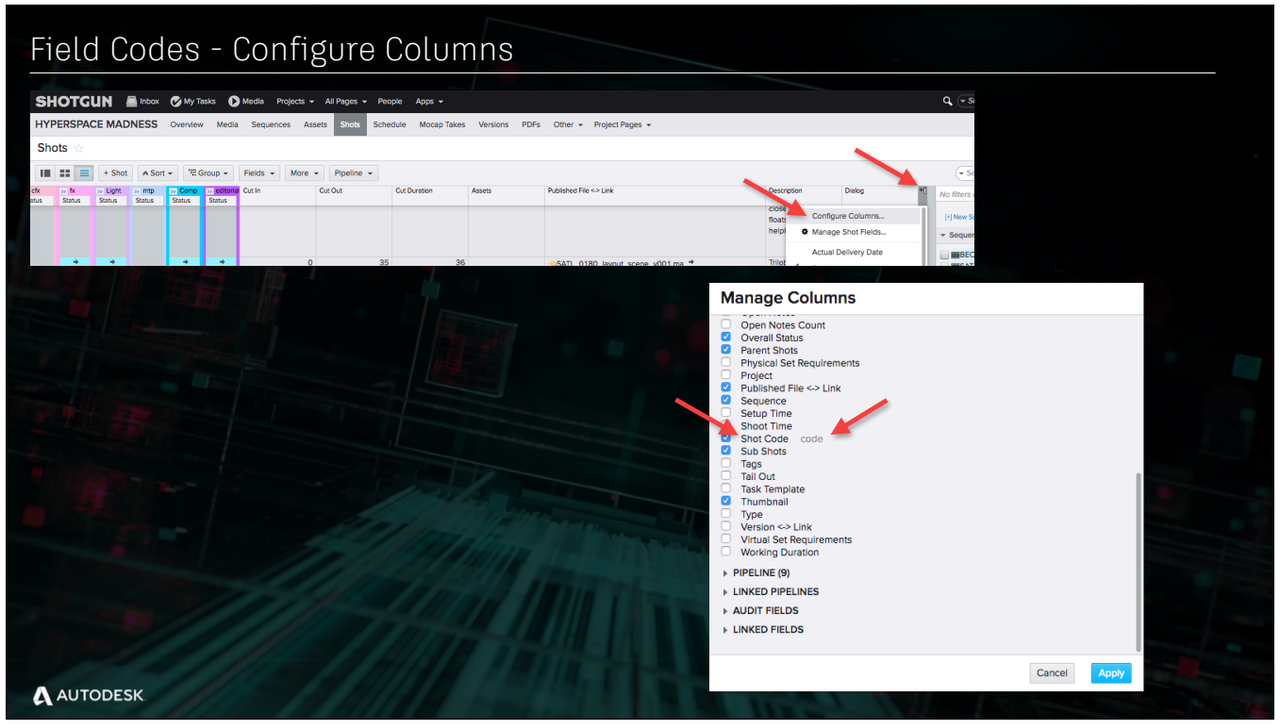
Did you notice that in our filter's first condition here we're searching on a field called `code` but for the project it was called `name`? These fields codes correspond to grid columns in the Shotgun UI but may have different names for accessing them in the API. Here are few tips to figure out what the field code is for a given grid column.

Field Codes - Configure field



First...
- In a grid in Shotgun, right click on the column header and...
- click `Configure Field...`.
- In the configure field dialog you'll see
- The column name for the UI
- And the field code you need to use in the API

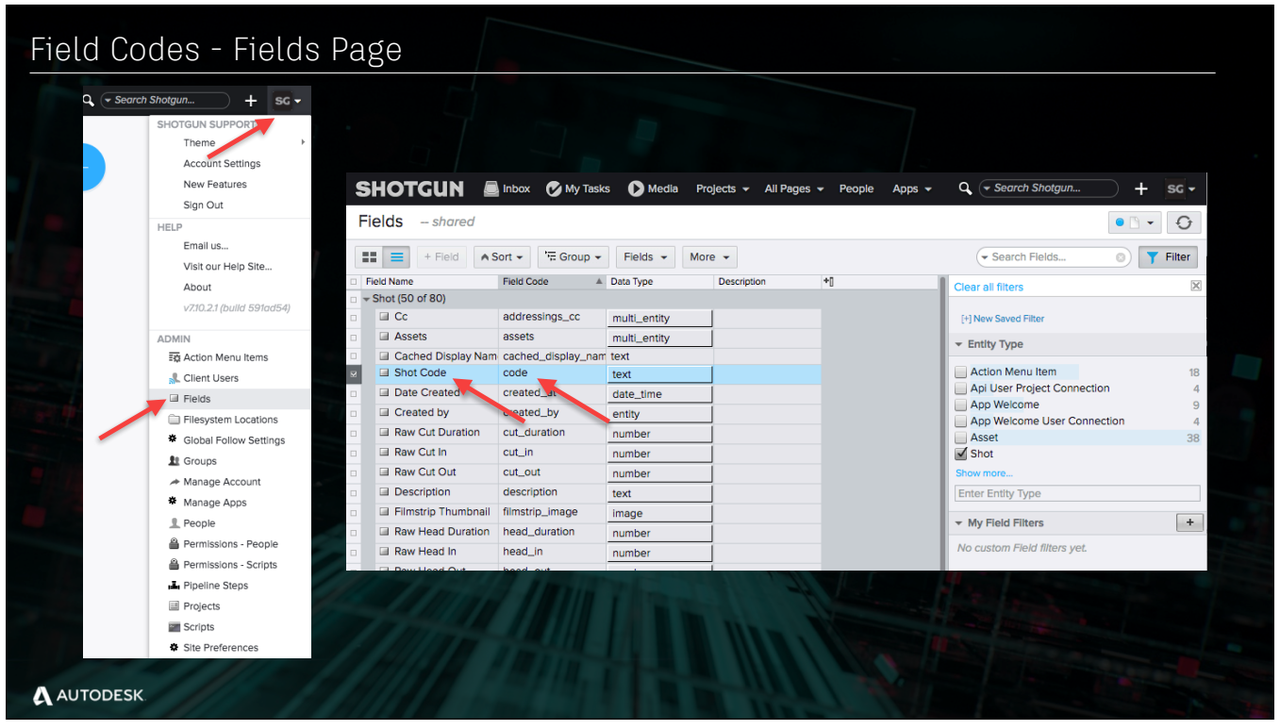
Field Codes - Configure Columns



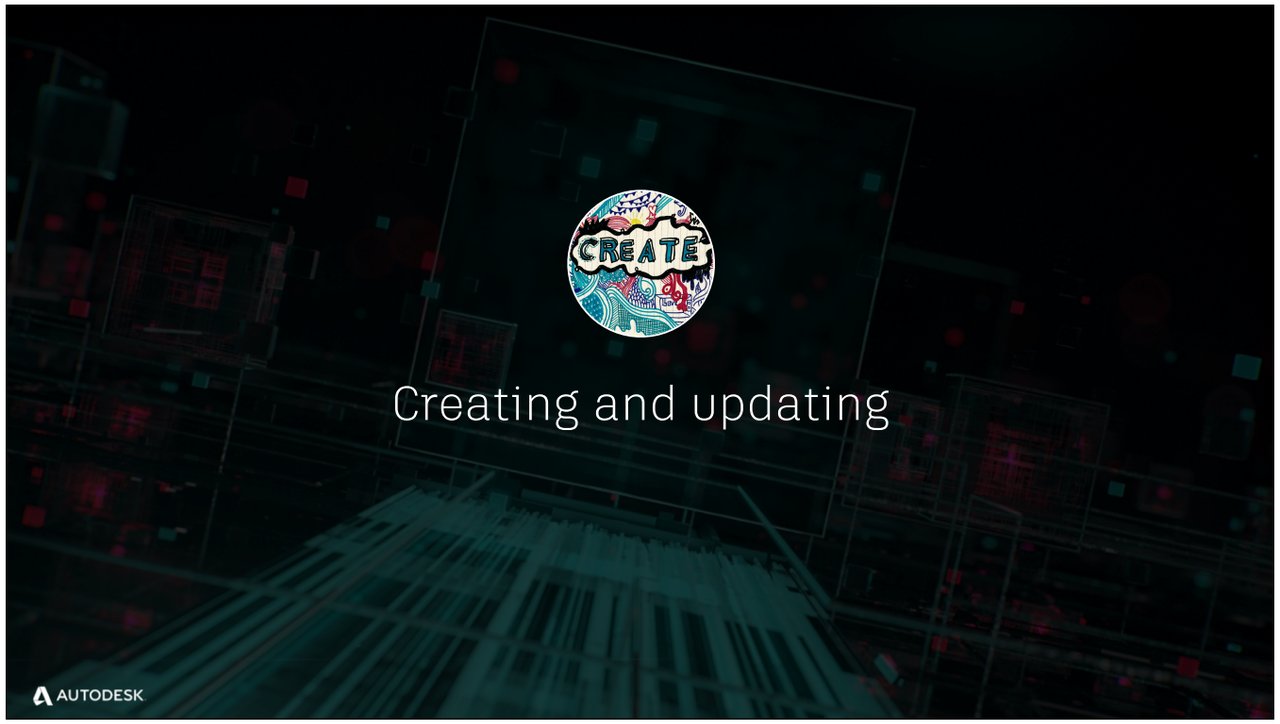
Second
- In a grid in Shotgun, click on the column management button...
- click `Configure Columns...`.
- In the dialog
- You'll see the column name for the UI
- And the field code you need to use in the API

You'll need to hover your mouse over a column name to see the field code.

Field Codes - Fields Page



Finally
- By clicking the user menu
- And choosing Fields
- You can access the fields page which lists all fields on all entities in the system
- You can then find the appropriate field
- And the API field code



Ok… Let’s keep going with our script… Now let’s add code to create our version.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])
```

AUTODESK

Our code as it stands… and we’ll add a single line

You'll notice that the first argument to our `create` method is the entity type of the record we want to create. The second argument is a dictionary which includes field codes and their respective values. Here's where we're reusing the project and shot entities we found previously in order to link our version to the right locations.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])

# Create our version attached to the project
version = sg.create('Version', {'project':project, 'entity':shot, 'code':'checkerboard.mov'})

# Update the version to set it to review
sg.update('Version', version['id'], {'sg_status_list':'rev'})
```

AUTODESK

Next we'll update our version in order to set its status to in review, or `rev`. Like the find and create methods you'll notice a pattern in the arguments which include an entity type and a dictionary of field values.

Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])

# Create our version attached to the project
version = sg.create('Version', {'project':project, 'entity':shot, 'code':'checkerboard.mov'})

# Update the version to set it to review
sg.update('Version', version['id'], {'sg_status_list':'rev'})
```

AUTODESK

One extra bit of information that needs to be supplied to the update method is the id of the entity you’re editing. In this case we’re reusing the id that was given to us as a result of the `create` call. Let’s take a minute to look at that version variable more closely.

Shotgun records in the Python API are ordinary dictionaries with two mandatory keys, a `type` which determines the entity type of the record you’re looking at and an `id` for that record.

This is the simplest form of an entity dictionary. Because of the dictionary nature of this object, you now see how we can use the id of a record we've fetched or created in Shotgun in order to update it later.
Now, I said this was the simplest form possible. In reality, our `create` call returned the following...

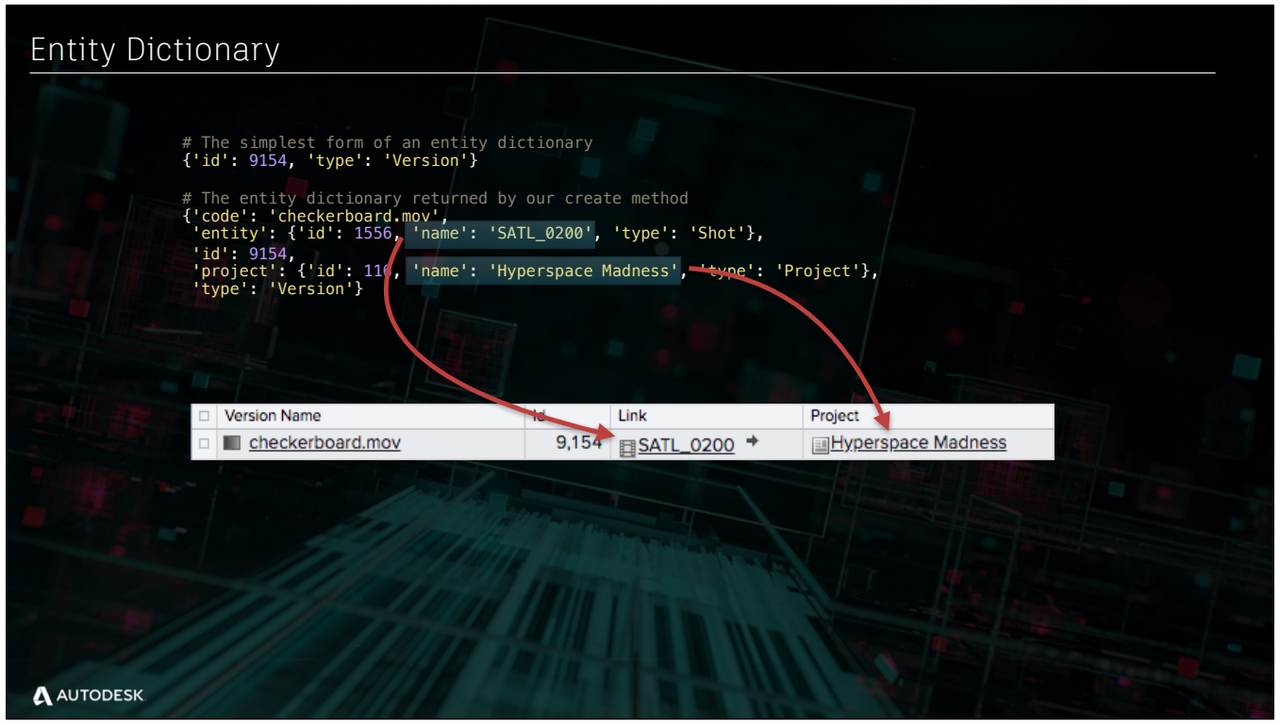
Entity Dictionary

```
# The simplest form of an entity dictionary
{'id': 9152, 'type': 'Version'}

# The entity dictionary returned by our create method
{'code': 'checkerboard.mov',
 'entity': {'id': 1556, 'name': 'SATL_0200', 'type': 'Shot'},
 'id': 9152,
 'project': {'id': 116, 'name': 'Hyperspace Madness', 'type': 'Project'},
 'type': 'Version'}
```

AUTODESK

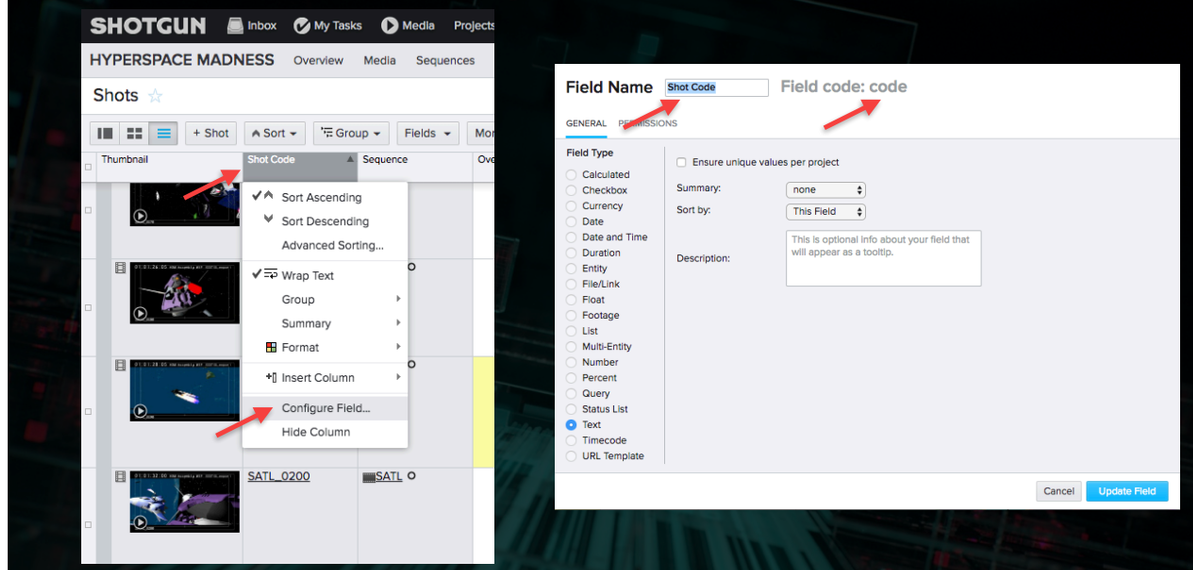
When you create a record in Shotgun you get an object back that includes all the fields you supplied in your creation. So in this case, not only do we have the type and id but we also have the name of our version in the code field as well as entity dictionaries representing the shot and project in the respective entity and project fields.

Let’s take a second to explore those shot and project hashes



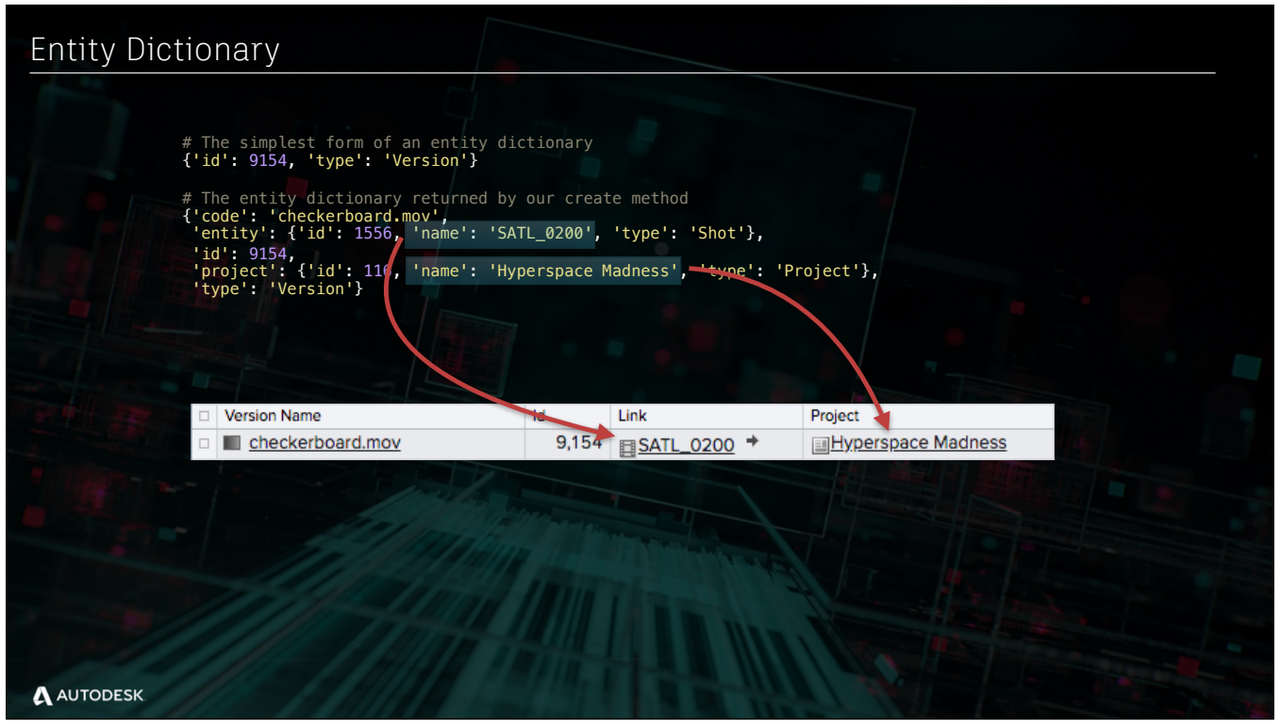
These hashes represent values in entity fields of our version. Let’s called these linked entities.
They both have a `name` key, which makes sense for our project but in the case of our shot we would have expected the shot name to be in a field called code.

Field Codes - Configure field

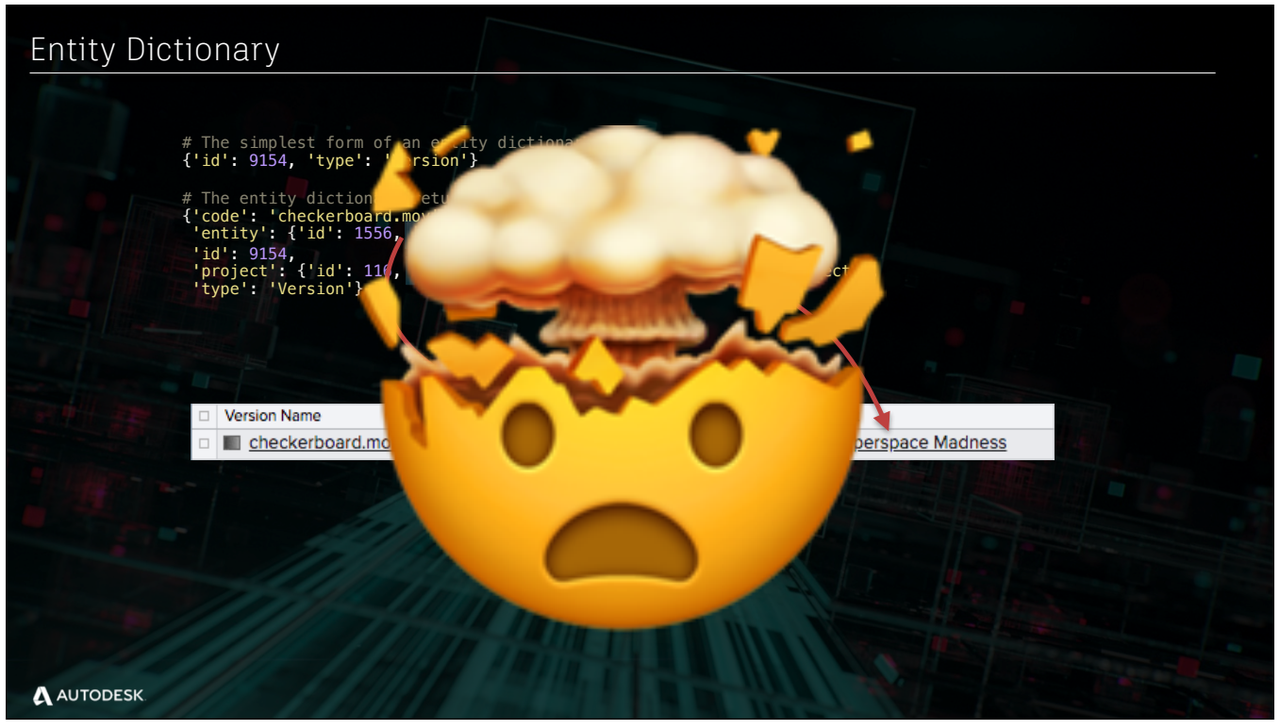


AUTODESK

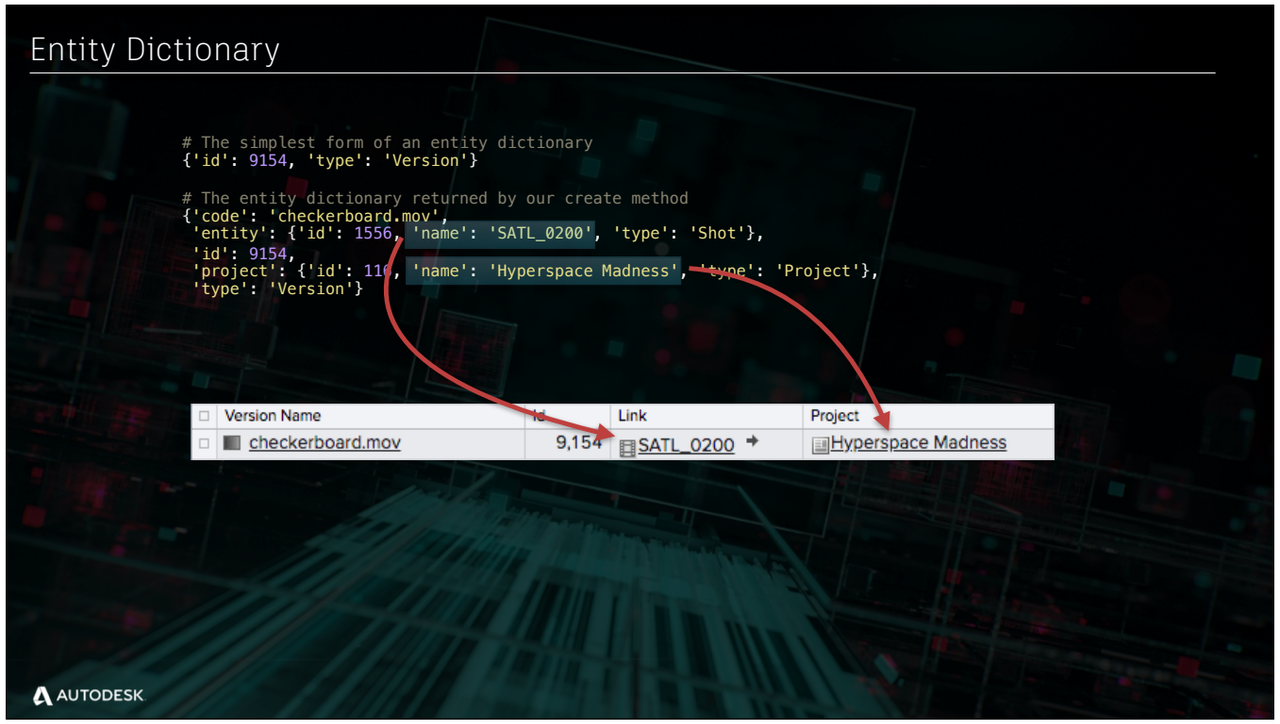
Here’s a reminder...



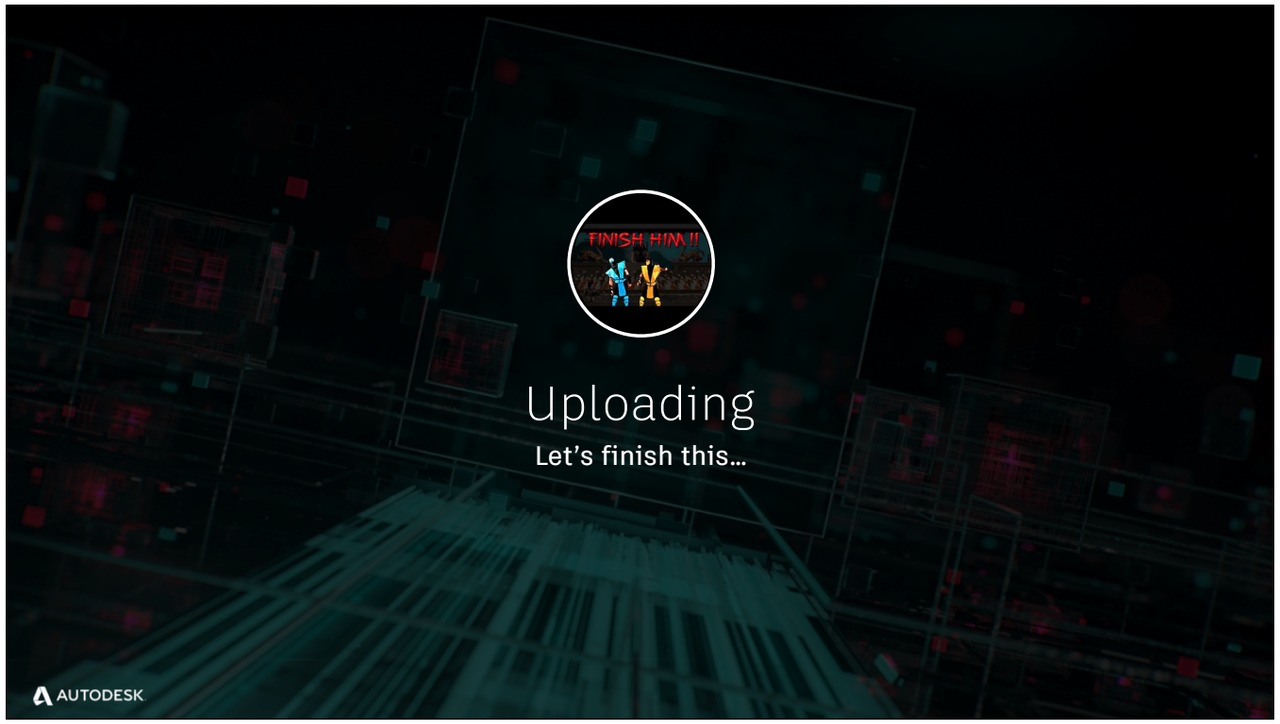
What gives?
Well... for linked entities, this is a third piece of standard information the API returns along with `type` and `id`. We've generalized this as being the `name` of the entity so you can reliably access the key but it represents a human readable name for the records, whichever field really holds that information, like `name` for a project or `code` for a shot.


Entity Dictionary
The simplest form of an entity dictiona
{'id': 9154, 'type': 'Version'}
The entity diction
{'code': 'checkerboard.mo
'entity': {'id': 1556,
'id': 9154,
'project': {'id': 11
'type': 'Version'}
Version Name
checkerboard.mo
berspace Madness
AUTODESK

Whaaaa???



Alright. Put simply... If the only thing you need from a linked entity is a human readable name for a UI of some sort, this mechanism avoids having to do extra find calls for the linked entities and gives you those names in a standardized manner for free. It doesn't necessarily mean there's a field called name on those records in Shotgun.



Ok, let’s finish this by uploading our media...

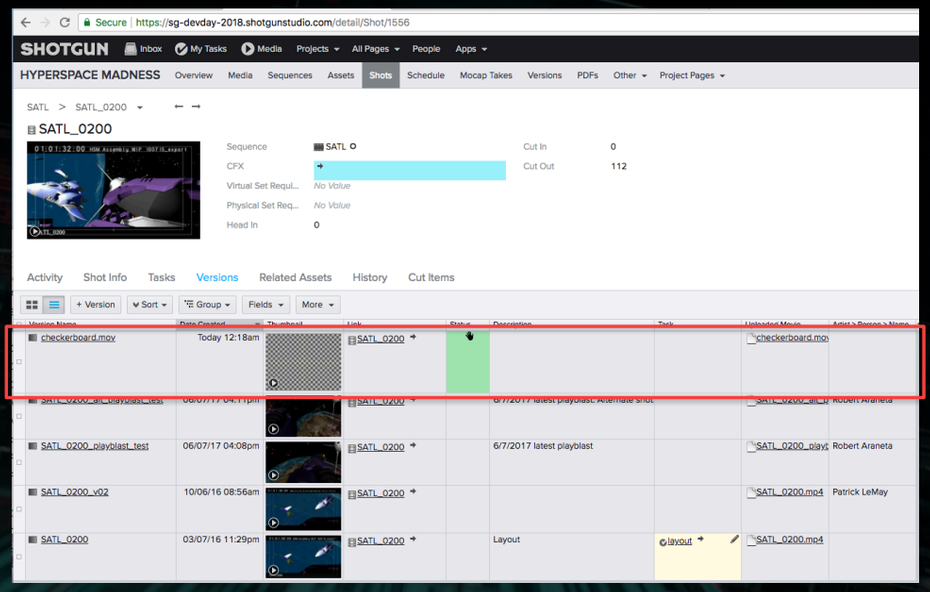
Code: Automating Review Submission

```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])

# Create our version attached to the project
version = sg.create('Version', {'project':project, 'entity':shot, 'code':'checkerboard.mov'})

# Update the version to set it to review
sg.update('Version', version['id'], {'sg_status_list':'rev'})

# Upload your media for approval
sg.upload('Version', version['id'], '/shotgun/media/checkerboard/checkerboard.mov', 'sg_uploaded_movie')
```

AUTODESK

This last line is again very simple. It’s format is fairly similar to the update call in that you need to supply the id of the record you’re uploading to as well as the entity type. The last two arguments are the path to the file you wish to upload as well as the field on the entity you’ll be uploading to.

Automating Review Submission



AUTODESK

And voilà. We've got our version record, ready for approval.

Code: Automating Review Submission

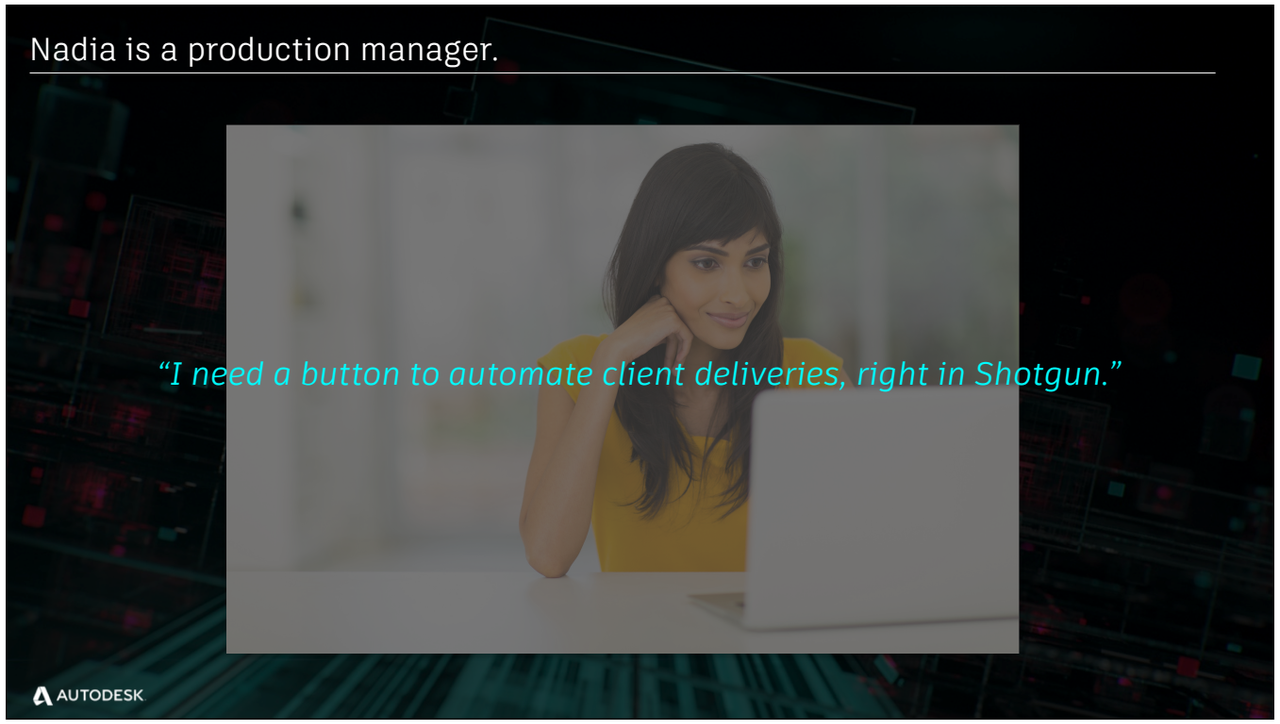
```python
import os
import shotgun_api3

SITE_URL = os.environ['SG_HOST']
SCRIPT_NAME = os.environ['SG_SCRIPT_NAME']
SCRIPT_KEY = os.environ['SG_SCRIPT_KEY']

# Create a connection to Shotgun
sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

# Find the project we'll be attaching our version to / simple filter
project = sg.find_one('Project', [['name', 'is', 'Hyperspace Madness']], ['id'])

# Find the shot we'll be attaching our version to / compound AND filter
shot = sg.find_one('Shot', [['code', 'is', 'SATL_0200'], ['project', 'is', project]], ['id'])

# Create our version attached to the project
version = sg.create('Version', {'project':project, 'entity':shot, 'code':'checkerboard.mov'})

# Update the version to set it to review
sg.update('Version', version['id'], {'sg_status_list':'rev'})

# Upload your media for approval
sg.upload('Version', version['id'], '/shotgun/media/checkerboard/checkerboard.mov', 'sg_uploaded_movie')
```

AUTODESK

The eagle eyed here (or anyone still mildly paying attention) might ask why there's the update call in the script to set the status to review. Couldn't that have happened in the create call. Yes, it could but then I wouldn't have had an excuse to cover the update call, entity dictionaries and the name key on linked entities.

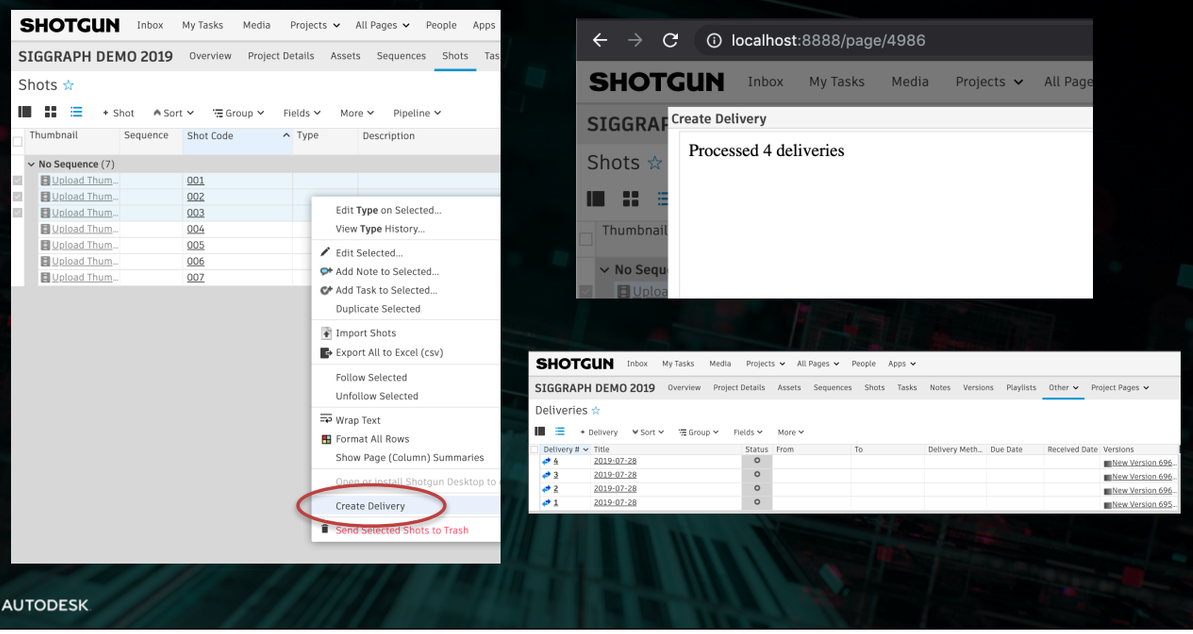
Alright, now that you've got versions for review, here's another scenario for you. You want to create a tool that will prep a delivery for a client. It lets you select multiple shots, creates one delivery per shot and adds the appropriate versions whose status were pending to the delivery.

This is a perfect opportunity to use an AMI. This could also be implemented as a client side toolkit app and if you want to know more about that, make sure to check out the advanced talk later today.



Actually, if you remember Tannaz’s talk this is exactly the tool that Nadia the production manager was wanting.

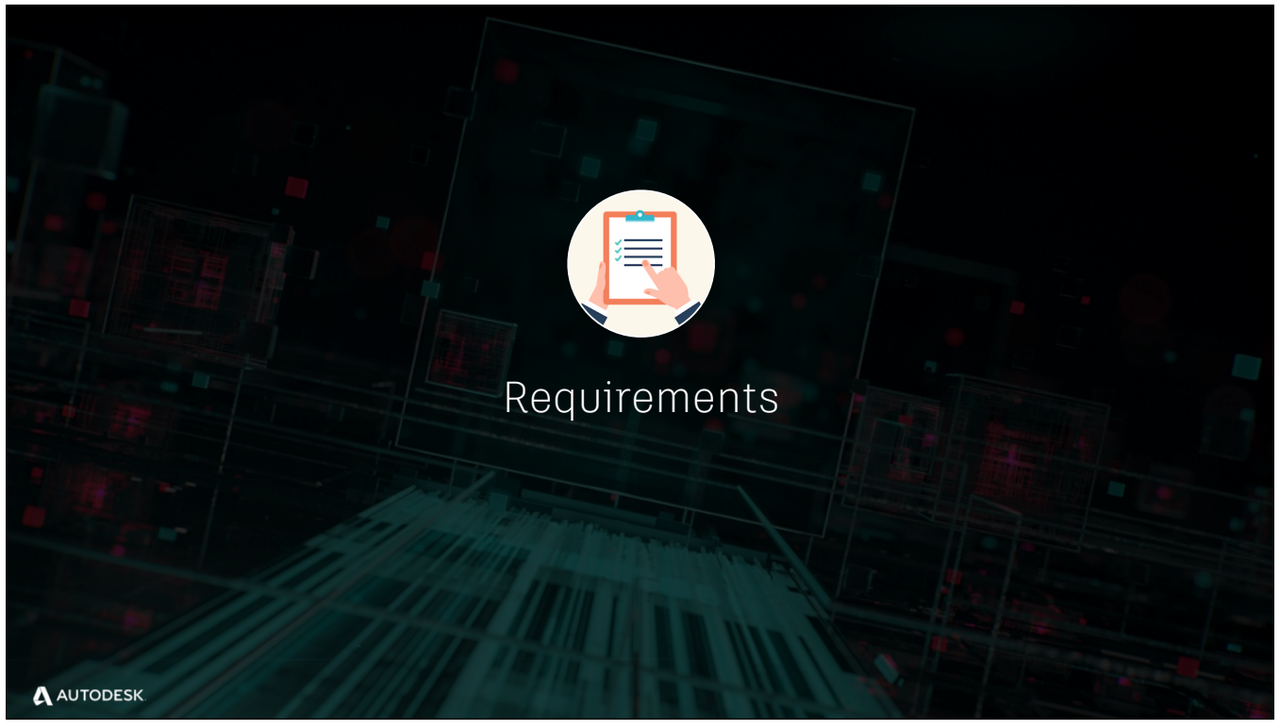
Create Delivery AMI



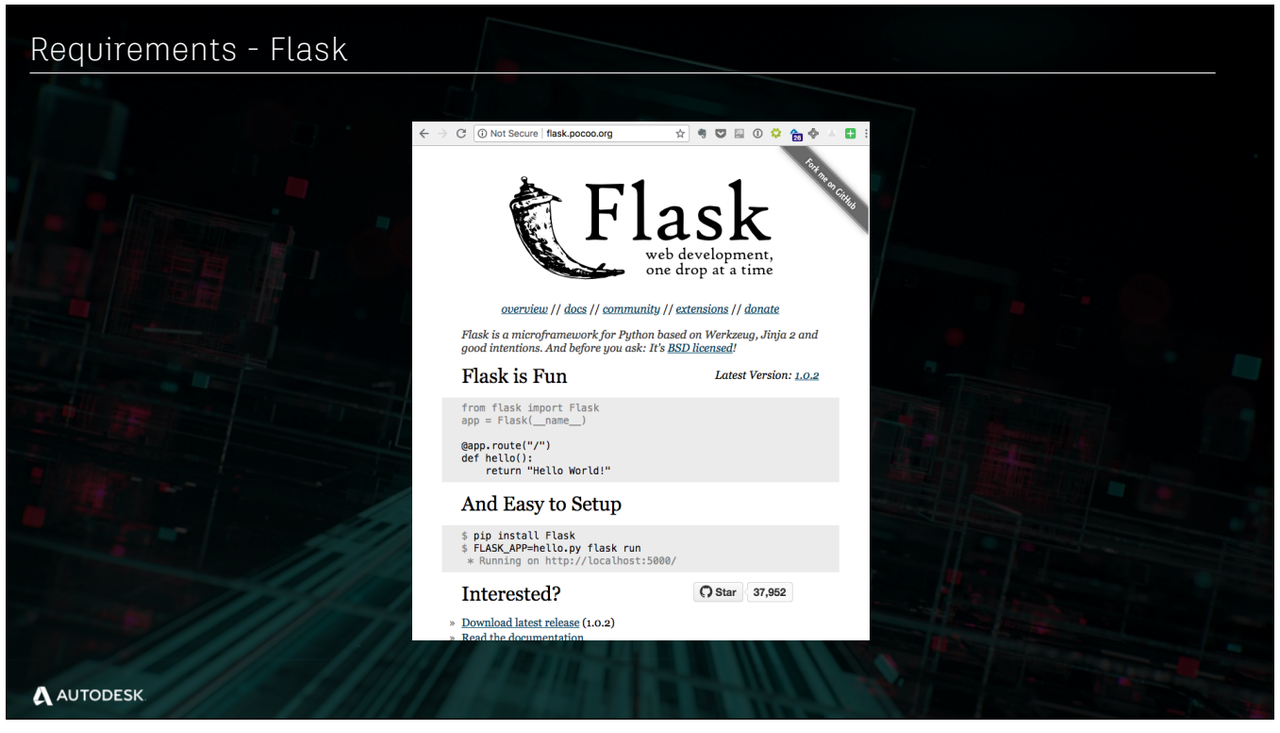
At the end of this scenario here’s what you can expect

- Once you click on your AMI which will be accessible when right clicking on one or more shots
- you can expect a small web server to process your request
- and create a set of delivery records accessible in Shotgun
- Each selected shot will have a delivery made for it which will contain links to all versions with a pending status for that shot

This, again, is a very simple implementation that doesn’t pretend to cover all possible usage scenarios or error cases - take it as it is, an excuse for learning about the API.

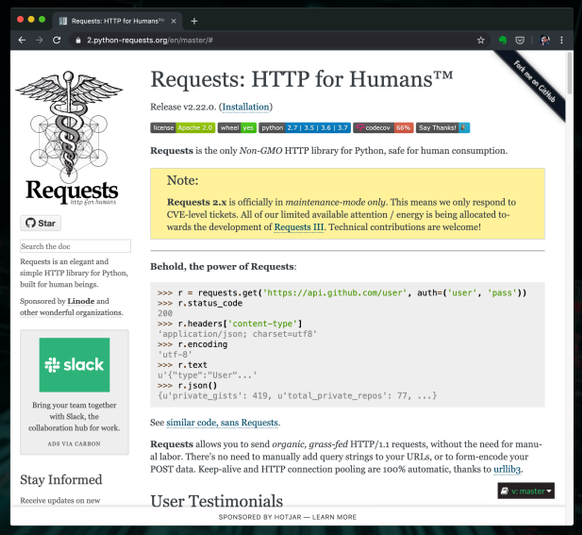


Let’s talk about what you’ll need to make this work.



In this use case I'll be showing you how an AMI can interact with a web service you've written and how that web service can interact in turn with Shotgun.
To implement the web service I'll be using Flask which is a lightweight web framework for Python. It is installable like any other Python module and is freely available.

Requirements - Requests

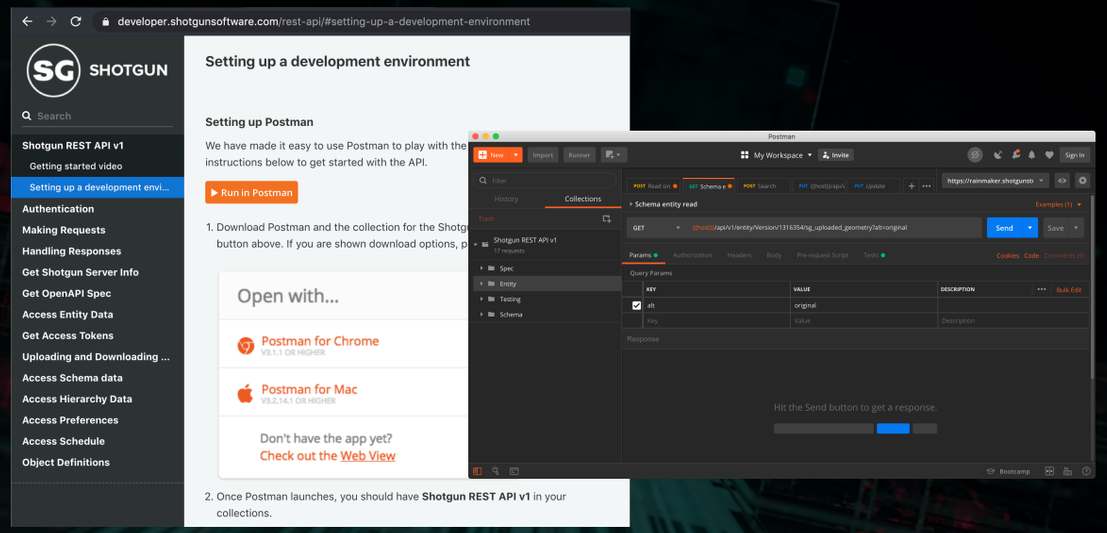


AUTODESK

You'll also need the Python Requests library because we'll be writing our interactions with Shotgun with the REST API.

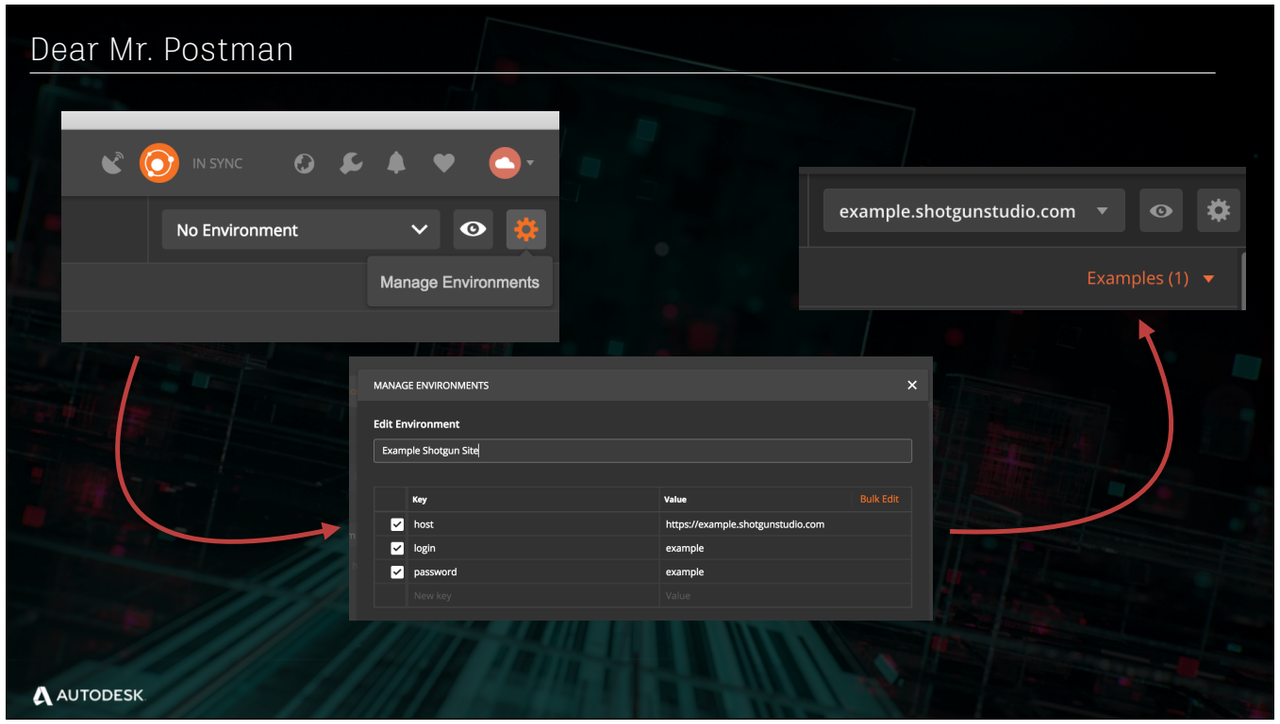

I digress...
Experimenting with Postman
AUTODESK

Dear Mr. Postman



Postman is a tool for API development accessible at getpostman.com

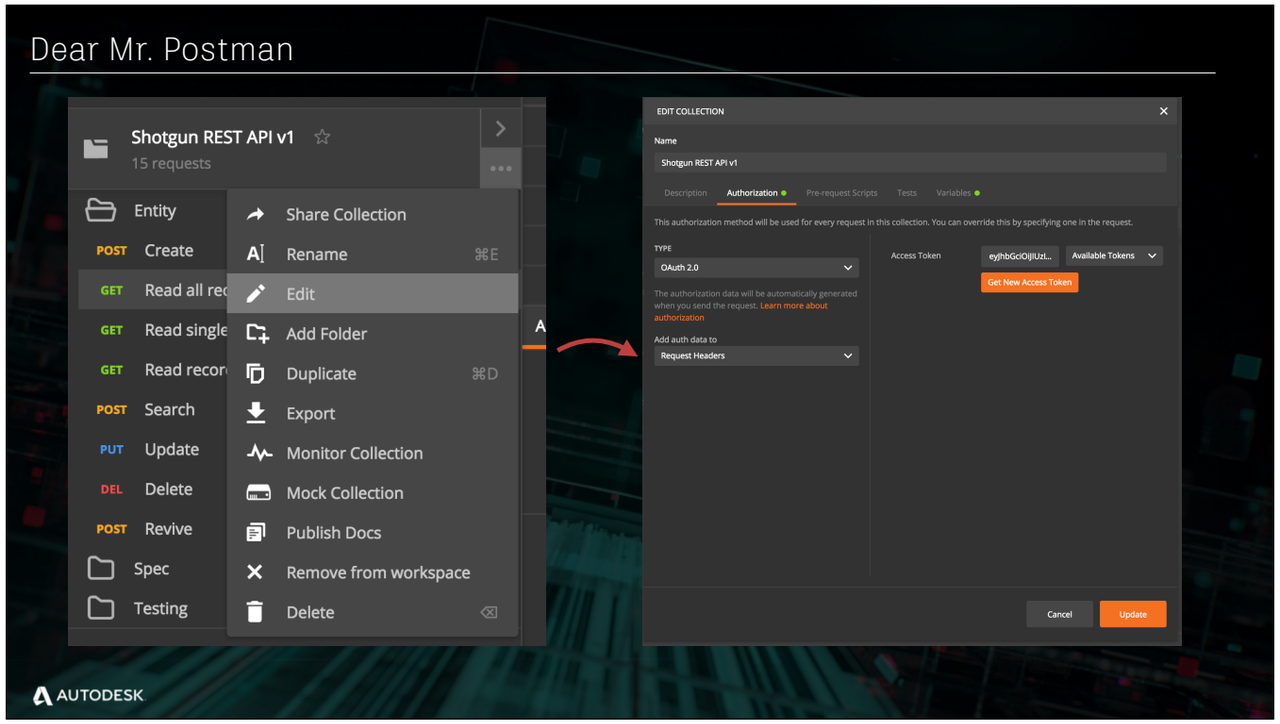
From our REST documentation, in the "setting up a development environment" section you'll find a "Run in Postman" button which will load a collection of API calls in Postman. This collection will allow you to experiment and familiarize yourself with our REST API.



To connect to Shotgun, we need to setup an environment that has the site, login, and password to use. Start by clicking on the gear icon in the upper right of Postman to manage environments.

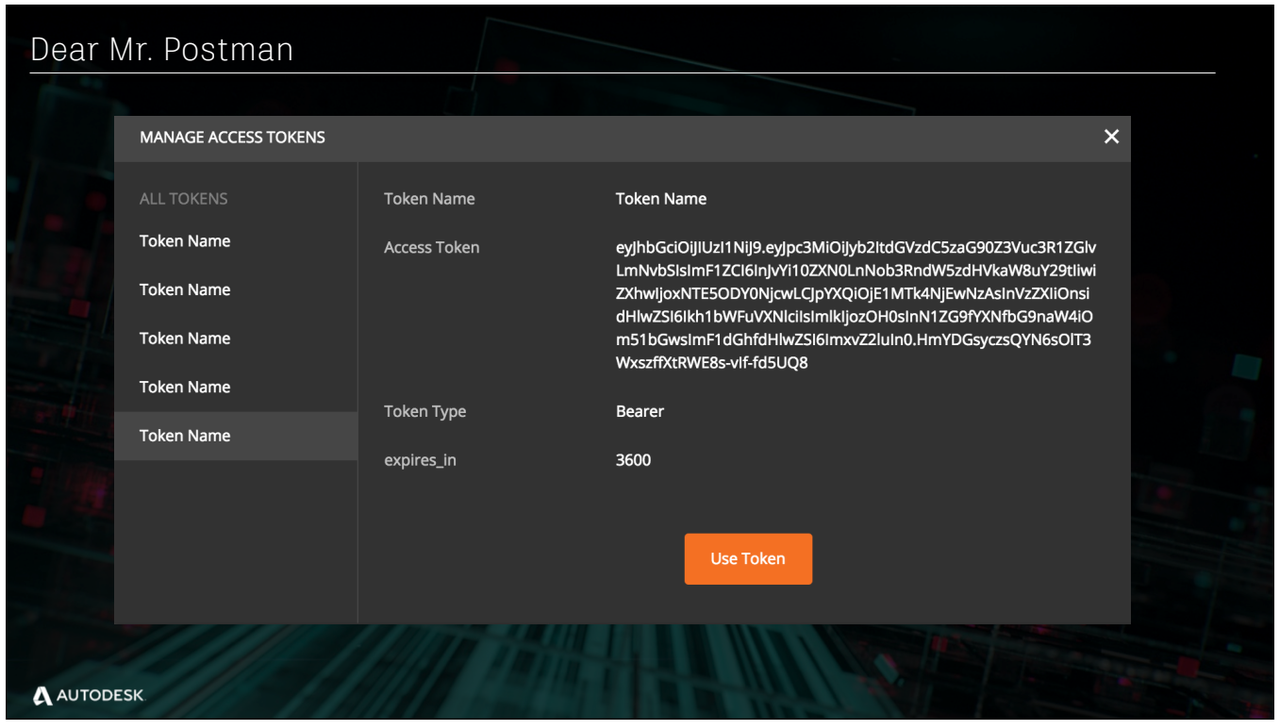
Add a new environment and add keys for host, login, and password. Fill out the values with the account you want to connect as.

Select the environment you just created as the active environment.



Now that credentials are available via the environment, we will use them to get an authentication token. The auth token will be used by the API to talk to your Shotgun site. To do this click on the ellipsis ("...") menu next to the collection and pick edit.

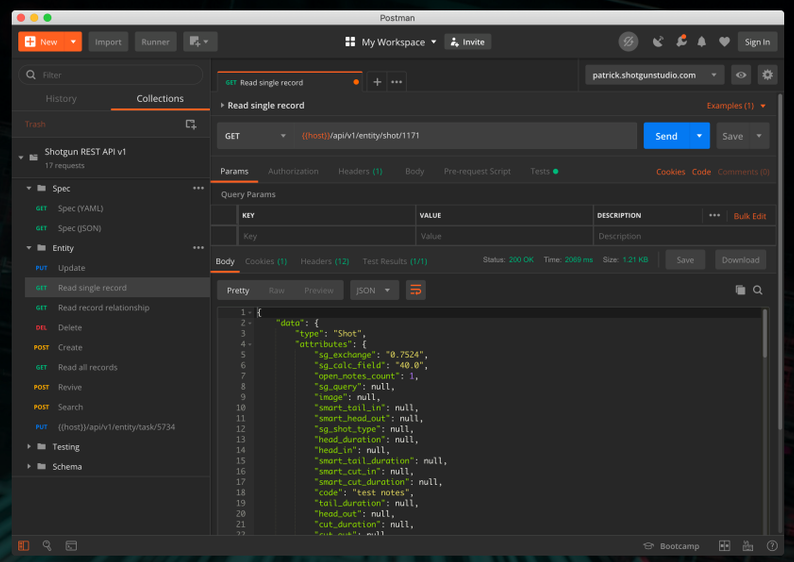
Go to the authorization tab of the collection dialog and click on Get New Access Token.



You should see a dialog like this one that displays the resulting token. Click on Use Token to start being able to use the API. Note: These tokens are only good for a fixed period of time (as determined by a site preference). You will have to repeat the above steps to refresh your token as needed.

Postman's built-in authorization system doesn't support refresh tokens currently so we have built a Postman "Pre-request Script" to help with this. I’m not covering this here but you can check out our documentation.

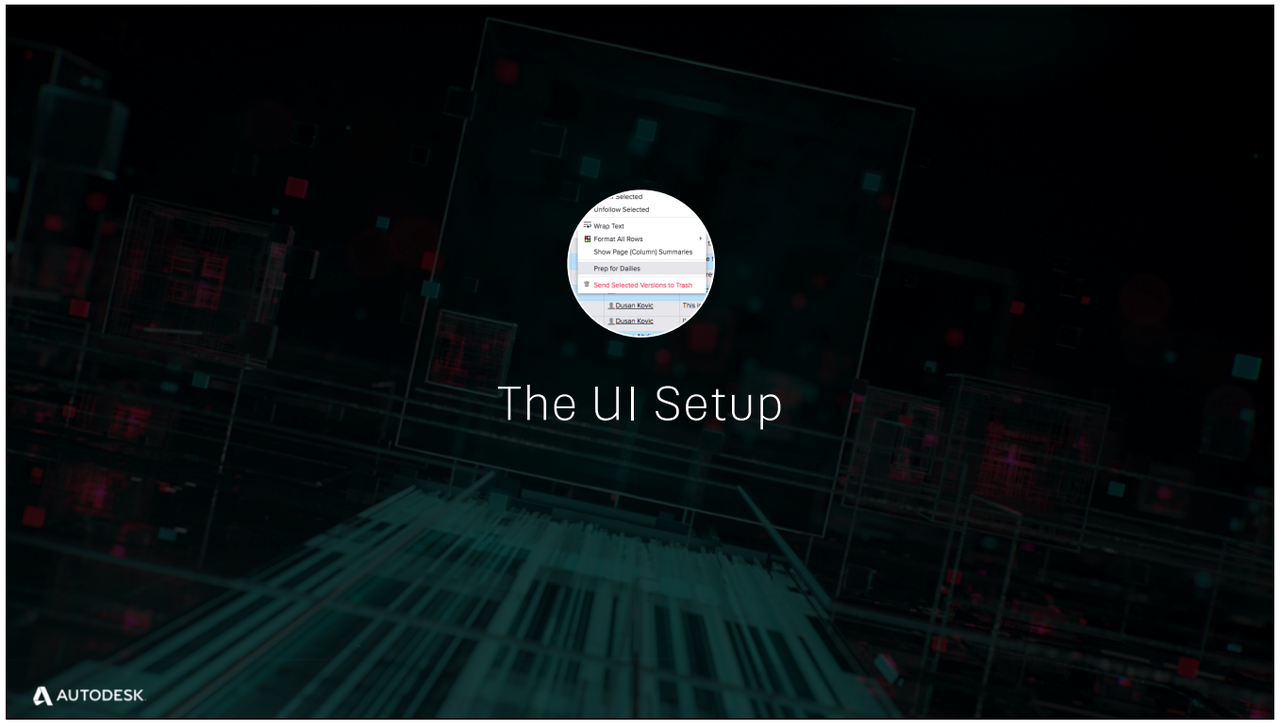
Dear Mr. Postman



AUTODESK

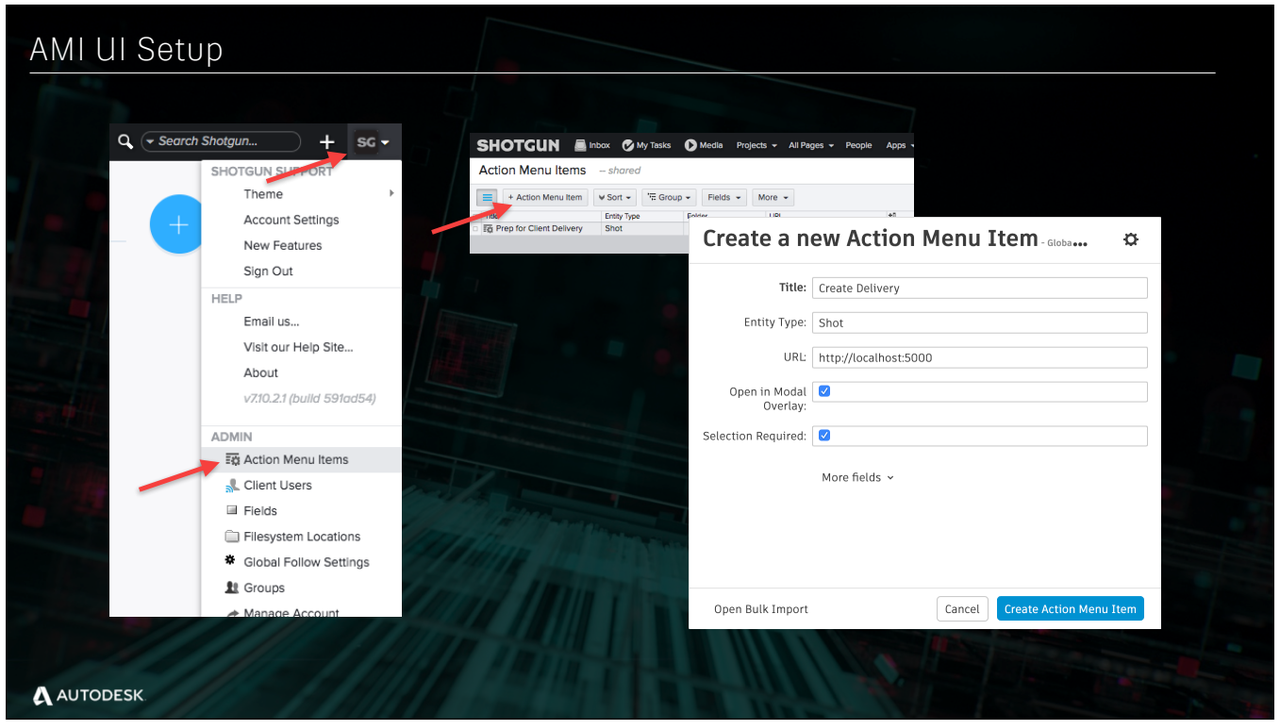
Once all of this is done you'll be able to test any one of our 17 request examples as well as your own.

You'll be able to test requests before you code them into your scripts and applications making your process much more interactive.



Let’s get back to our Action Menu Item scenario: a tool that lets you select multiple shots, creates one delivery per shot and adds the appropriate pending versions to that delivery.

First off, setting up the UI.

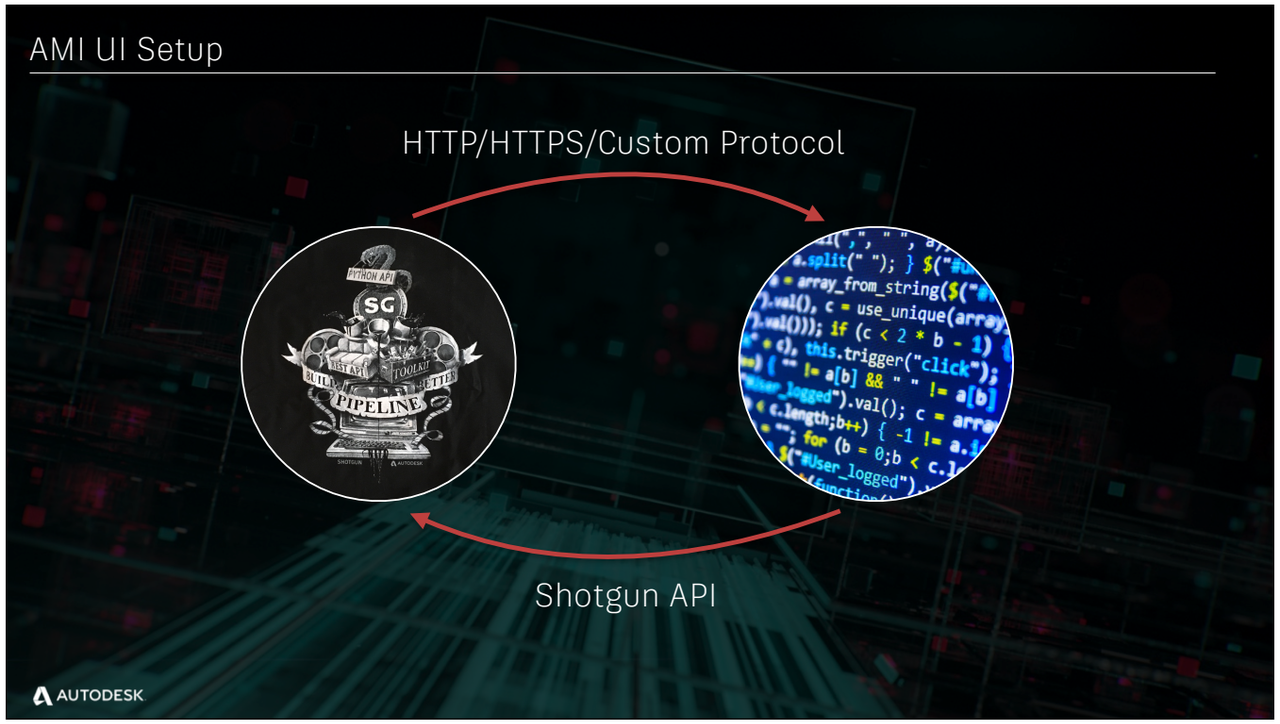


To create your AMI
- As an admin you'll open the user menu
- and click on `Action Menu Items`
- In the `Action Menu Items` page you'll
- add a new action menu item
- And populate it as follow

Because we're setting up our AMI to work on shot entities, only in this context will the menu item appear when you right click. Additionally we're making sure that the system will require you have shots selected before you can invoke the menu item. By enabling the "Modal Overlay" option, we make sure the AMI's output is shown in a dialog box on the page where the user clicked the AMI instead of opening a new page. Finally, the URL is the location that the browser will send information to when the menu is clicked.

This URL can be http/https but could also be a custom protocol, for example shotgun://, that you've configured an application on your local system to handle. This is a very useful trick but a bit outside the scope of this talk so let's talk after if you want to know more about it.

At this point the only thing you've given Shotgun is a location to send information to...



At a very high level, everything starts in Shotgun when you click the AMI.
- Shotgun then sends a special payload via HTTP/HTTPS or a custom protocol at the URL you asked it to
- Your code then kicks in and does what its supposed to do
- which most probably includes communicating information back to Shotgun

AMI UI Setup

```
# This is a Flask request object's form converted to a dict for easy pretty printing
{
    'server_hostname': [u'sg-devday-2019.shotgunstudio.com'],
    'project_name': [u'Hyperspace Madness'],
    'user_login': [u'shotgun_admin'],
    'sort_column': [u'created_at'],
    'cols': [u'code,image,entity,sg_status_list,user,description,created_at'],
    'sort_direction': [u'desc'],
    'user_id': [u'24'],
    'column_display_names': [u'Version Name,Thumbnail,Link,Status,Artist,Description,Date Created'],
    'entity_type': [u'Version'],
    'referrer_path': [u'/detail/HumanUser/24'],
    'title': [u'undefined'],
    'ids': [u'9155'],
    'selected_ids': [u'9155'],
    'session_uuid': [u'e8cd610a-9bf3-11e8-bb83-0242ac110004'],
    'project_id': [u'116'],
    'page_id': [u'2279'],
    'view': [u'Default']
}
```


Here’s an example of the data you can expect Shotgun to send to your code in a slightly modified form of the Flask request object.

AMI UI Setup

```
# This is a Flask request object's form converted to a dict for easy pretty printing
{
    'server_hostname': [u'sg-devday-2019.shotgunstudio.com'],
    'project_name': [u'Hyperspace Madness'],
    'user_login': [u'shotgun_admin'],
    'sort_column': [u'created_at'],
    'cols': [u'code,image,entity,sg_status_list,user,description,created_at'],
    'sort_direction': [u'desc'],
    'user_id': [u'24'],
    'column_display_names': [u'Version Name,Thumbnail,Link,Status,Artist,Description,Date Created'],
    'entity_type': [u'Version'],
    'referrer_path': [u'/detail/HumanUser/24'],
    'title': [u'undefined'],
    'ids': [u'9155'],
    'selected_ids': [u'9155'],
    'session_uuid': [u'e8cd610a-9bf3-11e8-bb83-0242ac110004'],
    'project_id': [u'116'],
    'page_id': [u'2279'],
    'view': [u'Default']
}
```

AUTODESK

I’d like to call to your attention some information which is very useful for custom report building like:

- the field codes of the columns that were visible on the grid
- the human readable name of those columns
- the sorting and / or grouping of the columns

AMI UI Setup

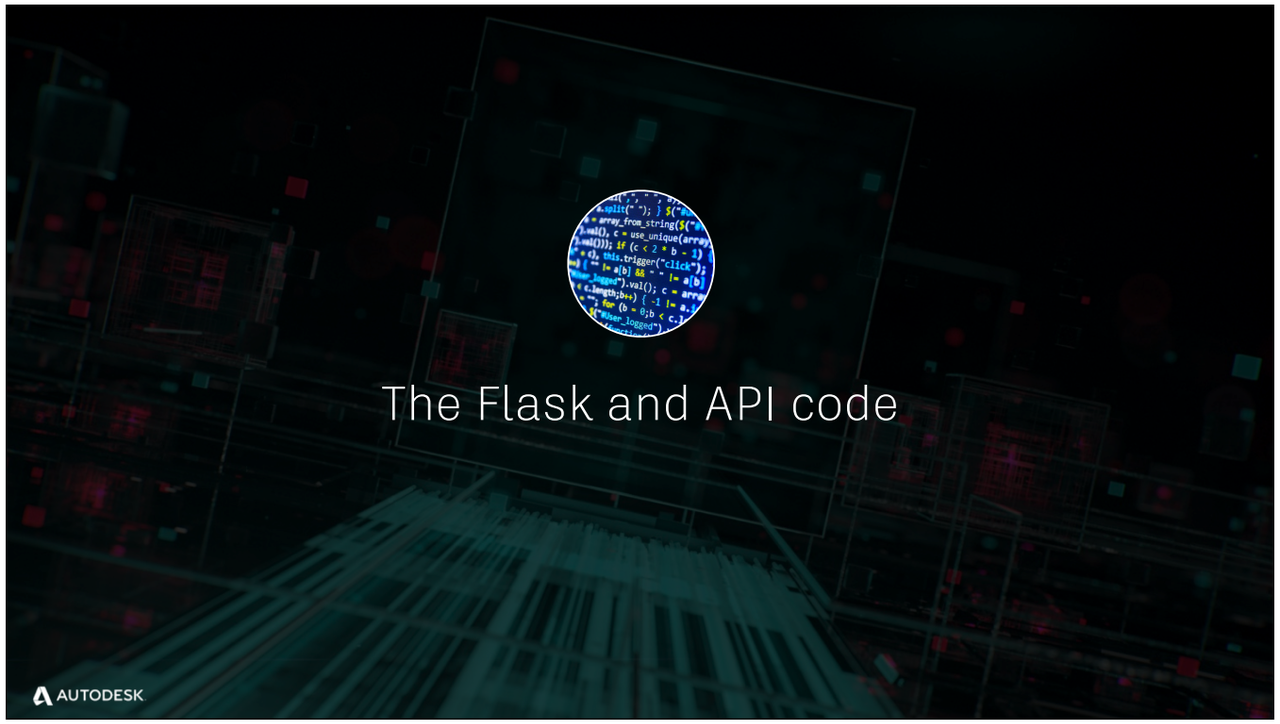
```
# This is a Flask request object's form converted to a dict for easy pretty printing
{
    'server_hostname': [u'sg-devday-2019.shotgunstudio.com'],
    'project_name': [u'Hyperspace Madness'],
    'user_login': [u'shotgun_admin'],
    'sort_column': [u'created_at'],
    'cols': [u'code,image,entity,sg_status_list,user,description,created_at'],
    'sort_direction': [u'desc'],
    'user_id': [u'24'],
    'column_display_names': [u'Version Name,Thumbnail,Link,Status,Artist,Description,Date Created'],
    'entity_type': [u'Version'],
    'referrer_path': [u'/detail/HumanUser/24'],
    'title': [u'undefined'],
    'ids': [u'9155'],
    'selected_ids': [u'9155'],
    'session_uuid': [u'e8cd610a-9bf3-11e8-bb83-0242ac110004'],
    'project_id': [u'116'],
    'page_id': [u'2279'],
    'view': [u'Default']
}
```

AUTODESK

As well as some information that is particularly useful for general purpose tools like:

- the entity_type of the grid you were on
- the ids of selected records
- the id of the project you were in

In our case we'll be very interested in these



Alright... On with the meaty bits.

Flask Boilerplate

```python
import datetime
import json
import os
from flask import Flask, request
import requests
import shotgun_api3

SITE_URL = os.environ.get('SG_HOST') + '/api/v1'
SCRIPT_NAME = os.environ.get('SG_SCRIPT_NAME')
SCRIPT_KEY = os.environ.get('SG_SCRIPT_KEY')

app = Flask(__name__)

@app.route("/", methods=['GET', 'POST'])
def ami_endpoint():
    delivery_count = process_shots()
    return "Processed %d deliveries" % delivery_count

if __name__ == "__main__":
    app.run(debug=True)
```

AUTODESK

For context I wanted to show you the Flask code but this is by no means a course in Python web frameworks. I just wanted to make obvious the minimal amount of non-API code that is required for this kind of integration. Here we're just:
- importing our necessary Python modules which includes the Shotgun API
- setting up our connection and authentication information
- Setting up a single route on our server that responds to GET and POST requests
- And starting the server

All the fun stuff happens in `process_shots` and that's what we're going to look at next.

Processing Shots

```python
def process_shots():
    delivery_count = 0
    auth_header = get_auth_header()
    for shot_id in [int(i) for i in request.form.get('selected_ids').split(',')]:
        versions = get_versions_for_shot(auth_header, shot_id)
        if versions:
            deliveries = create_delivery_with_versions(auth_header, versions)
            if deliveries:
                delivery_count += 1
    return delivery_count
```

AUTODESK

Process shots implements the business logic of our AMI

Processing Shots

```python
def process_shots():
    delivery_count = 0
    auth_header = get_auth_header()
    for shot_id in [int(i) for i in request.form.get('selected_ids').split(',')]:
        versions = get_versions_for_shot(auth_header, shot_id)
        if versions:
            deliveries = create_delivery_with_versions(auth_header, versions)
            if deliveries:
                delivery_count += 1
    return delivery_count
```

AUTODESK

First it authenticates and gets an authentication token using the REST API in get_auth_header().

We'll break this down shortly...

Processing Shots

```python
def process_shots():
    delivery_count = 0
    auth_header = get_auth_header()
    for shot_id in [int(i) for i in request.form.get('selected_ids').split(',')]:
        versions = get_versions_for_shot(auth_header, shot_id)
        if versions:
            deliveries = create_delivery_with_versions(auth_header, versions)
            if deliveries:
                delivery_count += 1
    return delivery_count
```

AUTODESK

Then using the information passed in through the AMI and the requests module we loop through the ids of each select shot...

Processing Shots

```python
def process_shots():
    delivery_count = 0
    auth_header = get_auth_header()
    for shot_id in [int(i) for i in request.form.get('selected_ids').split(',')]:
        versions = get_versions_for_shot(auth_header, shot_id)
        if versions:
            deliveries = create_delivery_with_versions(auth_header, versions)
            if deliveries:
                delivery_count += 1
    return delivery_count
```

AUTODESK

For each shot id, we will fetch all the appropriate pending versions that we wish to deliver...

Processing Shots

```python
def process_shots():
    delivery_count = 0
    auth_header = get_auth_header()
    for shot_id in [int(i) for i in request.form.get('selected_ids').split(',')]:
        versions = get_versions_for_shot(auth_header, shot_id)
        if versions:
            deliveries = create_delivery_with_versions(auth_header, versions)
            if deliveries:
                delivery_count += 1
    return delivery_count
```

AUTODESK

And then, now that we have the interesting versions we can go ahead and create the delivery in the `create_delivery_with_versions` function.

Finally we'll tally the number of created deliveries and report it back to our flask server so it can be displayed in our UI.

As you can see, business logic doesn't need to be complicated for it to be usefully automated and for its automation to be a big time saver in your facility.

Let's take a deeper look at the three key steps, authenticating with REST, searching for the versions and creating the delivery.

First, authentication.

To authenticate with a site using the REST API you need to provide credentials to the `access_token` endpoint. Like the Python API you can provide a script name and script key combination or a username and password. What's important is that after you send these credentials the API will return an authorization token which you will send back on all your subsequent REST API commands.

You should keep in mind that this authorization token will eventually expire. This is a standard security measure. It is your responsibility to renew the authorization token whenever appropriate for your application in order for you to always have a valid authorization token. In our case, our AMI will be able to process our deliveries well before the token expires so we will not worry about this.

The last part of our auth function sets up a header dictionary for future calls to the REST API. Notice the `Authorization` header which contains our access token as returned from the call to the `access_token` endpoint. This auth header will be reused by our version fetching and delivery creating code.

Searching in REST

```python
def get_versions_for_shot(auth_header, shot_id):
    # Get all pndng versions for a shot
    headers = {
        'Content-Type': 'application/vnd+shotgun.api3_array+json'
    }
    headers.update(auth_header)
    filters = {
        'filters': [
            ['entity.Shot.id', 'is', shot_id],
            ['sg_status_list', 'is', 'pndng']
        ]
    }
    resp = requests.post(SITE_URL + '/entity/version/_search', headers=headers, data=json.dumps(filters))
    return resp.json().get('data')
```

AUTODESK

After authenticating our next step was to search for the appropriate versions for our delivery. This is accomplished with the `_search` REST API endpoint

The search endpoint requires we send it some search filters using the request body. This filter syntax should be somewhat familiar to you as it is extremely similar to the Python API filter syntax.

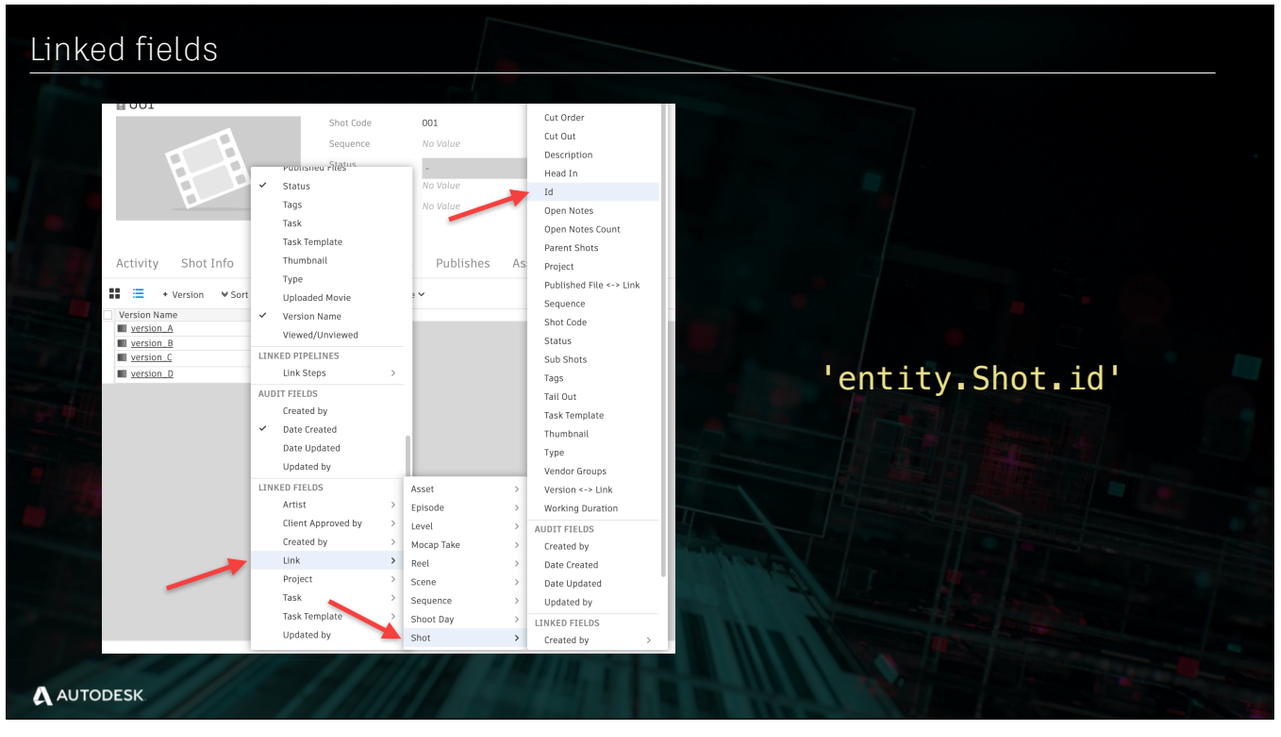
There is one thing I'd like to draw your attention to which I've intentionally planted here.

Searching in REST

```python
def get_versions_for_shot(auth_header, shot_id):
    # Get all pndng versions for a shot
    headers = {
        'Content-Type': 'application/vnd+shotgun.api3_array+json'
    }
    headers.update(auth_header)
    filters = {
        'filters': [
            ['entity.Shot.id', 'is', shot_id],
            ['sg_status_list', 'is', 'pndng']
        ]
    }
    resp = requests.post(SITE_URL + '/entity/version/_search', headers=headers, data=json.dumps(filters))
    return resp.json().get('data')
```

AUTODESK

It is called linked field syntax



When you're showing a column in a grid and you go to the linked fields section and you drill down through the hierarchy of entities and fields until you get to some deeply linked information, that's the UI equivalent of the linked field syntax.
You start off by specifying a field on the entity type you're searching for. In our case this is the entity field on the Version. Then you specify the entity type you're expecting back from that field. You then specify a field on that entity and so on and so forth until you get to the information you're looking for. This is an extremely powerful tool. In our case, somewhat reading backwards, we're searching for the `id` of the Shot that is present in the entity field of specific versions.

What's the point of this... Using linked field syntax often reduces the number of searches required to get data. This way, sometimes expensive combinations of calls can be merged into one which is more efficient.

Digression: If you've noticed that the field names in the UI don't match the field codes in the linked field syntax, remember that field names and field codes can differ.

Remember when I said I planted this linked field syntax to draw your attention to it… That’s simply so we could talk about the concept of linked field syntax.

Truth be told, this could be even more efficiently written as…

Searching in REST

```python
def get_versions_for_shot(auth_header, shot_id):
    # Get all pndng versions for a shot
    headers = {
        'Content-Type': 'application/vnd+shotgun.api3_array+json'
    }
    headers.update(auth_header)
    filters = {
        'filters': [
            ['entity', 'is', {'type':'Shot', 'id':shot_id}],
            ['sg_status_list', 'is', 'pndng']
        ]
    }
    resp = requests.post(SITE_URL + '/entity/version/_search', headers=headers, data=json.dumps(filters))
    return resp.json().get('data')
```

AUTODESK

This! Notice how we artificially construct an entity dictionary to represent the Shot. This is perfectly legitimate. You can manually build entity dictionaries like this and pass them onto Shotgun Python API or REST API methods because that's what they expect, just standard dictionaries. As long as you supply the prerequisite `type` and `id` keys in the dictionrary, you'll be fine.

Searching for specific records in entity and multi-entity fields like this should be a reflex as its way more efficient than searching via linked field for an id or even worse for a text match on the linked entity when you already know the id.

Searching in REST

```python
def get_versions_for_shot(auth_header, shot_id):
    # Get all pndng versions for a shot
    headers = {
        'Content-Type': 'application/vnd+shotgun.api3_array+json'
    }
    headers.update(auth_header)
    filters = {
        'filters': [
            ['entity', 'is', {'type':'Shot', 'id':shot_id}],
            ['sg_status_list', 'is', 'pndng']
        ]
    }
    resp = requests.post(SITE_URL + '/entity/version/_search', headers=headers, data=json.dumps(filters))
    return resp.json().get('data')
```

AUTODESK

The last thing I want to draw your attention to in this function is the reuse of the access token headers in the headers that we build and pass to the REST API call.

Now let’s look at our REST API call that creates the delivery…

It’s an oddly similar pattern as before, right?

First, notice the access token headers being reused again. In this simple example I’m just redoing similar work as before. In a more complex application you’d probably refactor this and extract some of this work in common functions.

Creating a record in REST

```python
def create_delivery_with_versions(auth_header, versions):
    headers = {
        'Content-Type': 'application/json'
    }
    headers.update(auth_header)
    data = {
        'title': datetime.datetime.now().strftime('%Y-%m-%d'),
        'sg_versions': [{'type':'Version', 'id':version['id']} for version in versions],
        'project': {'type':'Project', 'id':int(request.form['project_id'])}
    }
    resp = requests.post(SITE_URL + '/entity/delivery', headers=headers, data=json.dumps(data))
    return resp.json().get('data')
```

AUTODESK

If we're going to create a record we're going to need a data payload that describes the record. The REST API simply expects a hash where each key is a field in Shotgun and each field's value is the value that should be in the database. These can be simple datatypes or Shotgun records for entity and multi-entity fields.

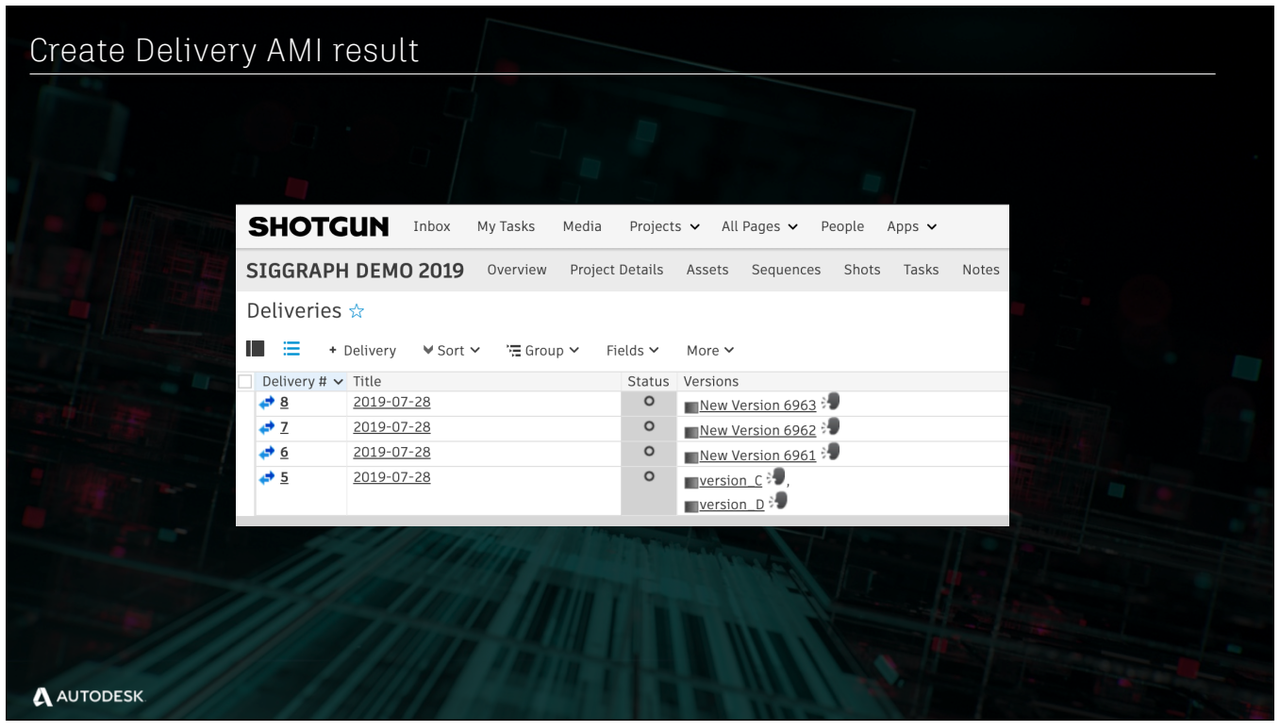
You'll notice that again this is very reminiscent of the Python API. We wanted the path to REST to be as simple as possible if you were already familiar with key Python API concepts.

Creating a record in REST

```python
def create_delivery_with_versions(auth_header, versions):
    headers = {
        'Content-Type': 'application/json'
    }
    headers.update(auth_header)
    data = {
        'title': datetime.datetime.now().strftime('%Y-%m-%d'),
        'sg_versions': [{'type':'Version', 'id':version['id']} for version in versions],
        'project': {'type':'Project', 'id':int(request.form['project_id'])}
    }
    resp = requests.post(SITE_URL + '/entity/delivery', headers=headers, data=json.dumps(data))
    return resp.json().get('data')
```

AUTODESK

Finally, calling the `entity` endpoint with the proper entity type name in the URL and passing our new record information as document body will complete the REST call to create our delivery.



Once all this back and forth between Shotgun, your AMI code and back to Shotgun has happened, you're left with a list of deliveries that your production crew can use to track communications with the client.

I hope this scenario has helped you appreciate the usefulness of AMIs and the power and relative simplicity of the REST API. Remember, we used Python here but the beauty with REST just being standard HTTP calls is that you can implement this in whatever language of your choosing whether it have a REST or HTTP libraries to further streamline things or not. Hint: Most languages do have these libraries.

There are tons of things we weren't able to cover here, like more advanced AMI options, REST API methods or Python API methods. Not to mention the Shotgun Event Daemon, Webhooks and Toolkit. Stick around for the Advanced topics class with Manne and Brandon for some of these.

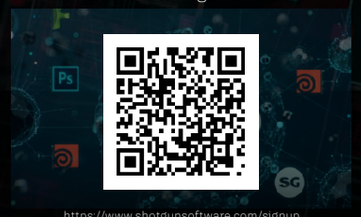
Let’s open this to Q&A

https://app.sli.do/event/10wljptx

AUTODESK

Thank you!

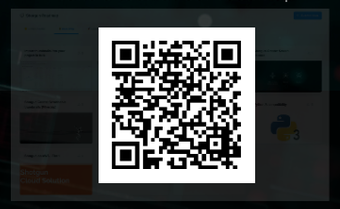
Start a Shotgun eval



https://www.shotgunsoftware.com/signup

If you haven't started playing around with Shotgun, but like what you're seeing, take it for a test drive.

Take a look at our roadmap



https://www.shotgunsoftware.com/roadmap

If you want to know what's cooking or what we are thinking about tackling next, this should be your first stop.

Checkout our new forums

https://community.shotgunsoftware.com

We've just launch a new community forum where you can chat with us and other Shotgun users.


AUTODESK