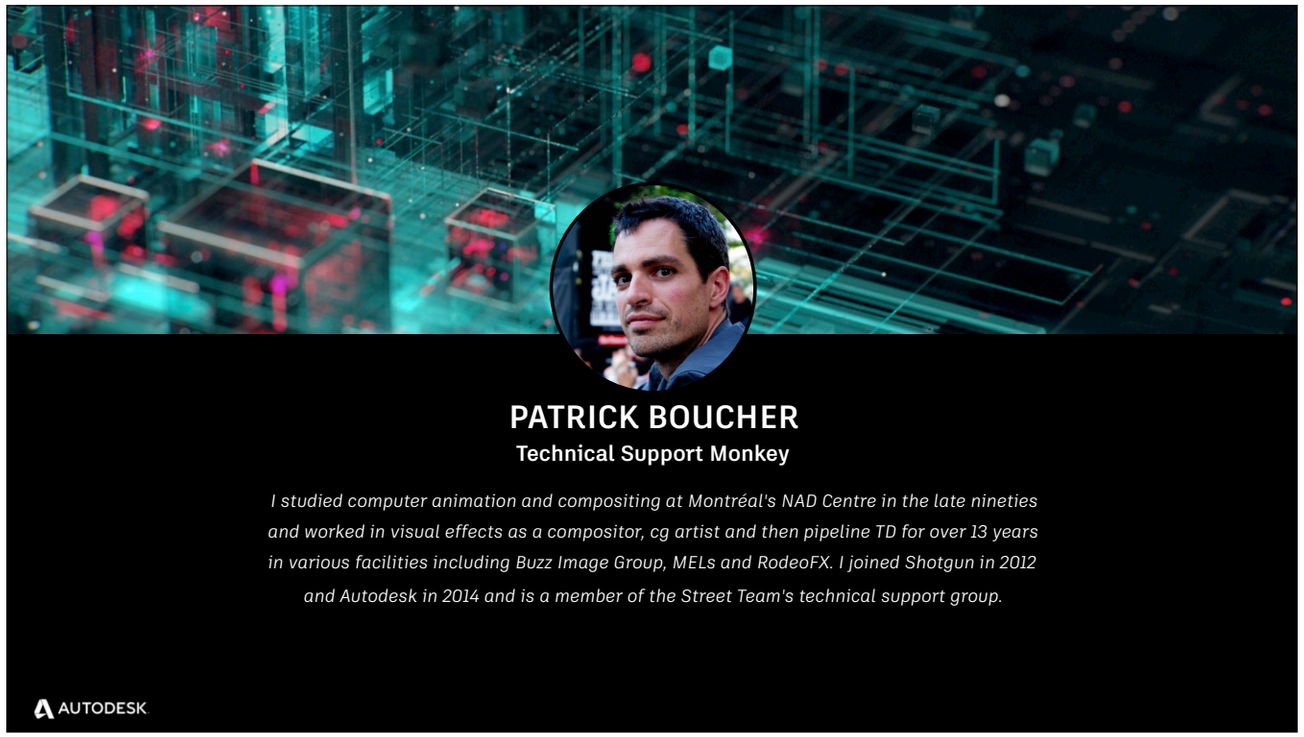
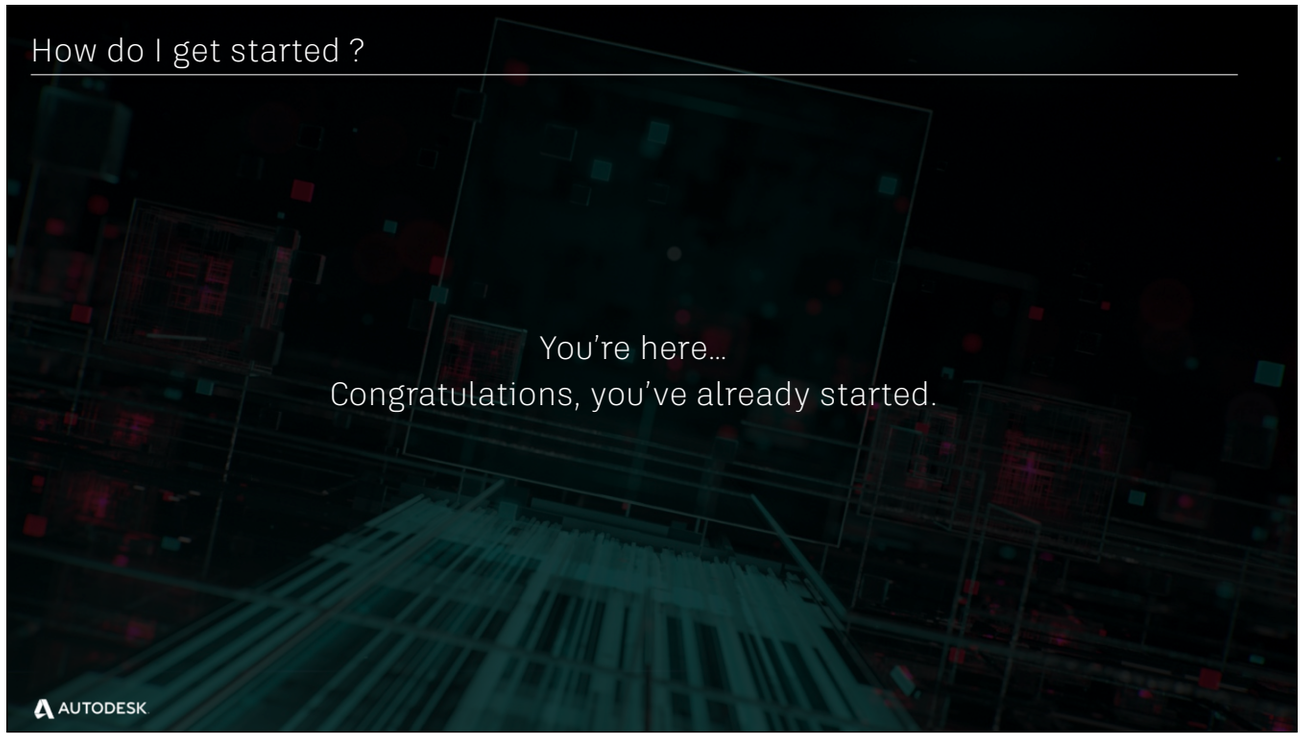
Hi everyone, in this talk we're going to discuss data management in Shotgun using its APIs.

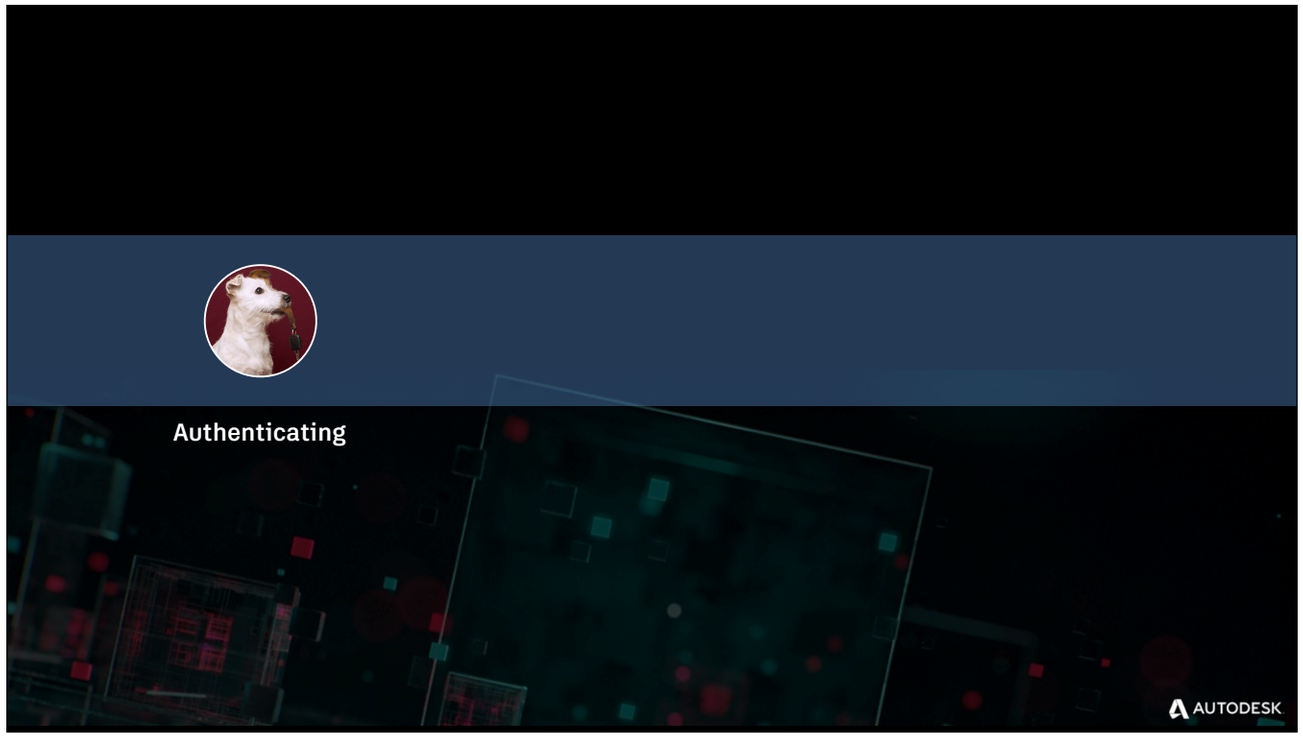


My name is Patrick Boucher and I’ve been with Shotgun for 6 years most recently as a technical support engineer. Prior to that I was a pipeline engineer for various facilities for 13 years.

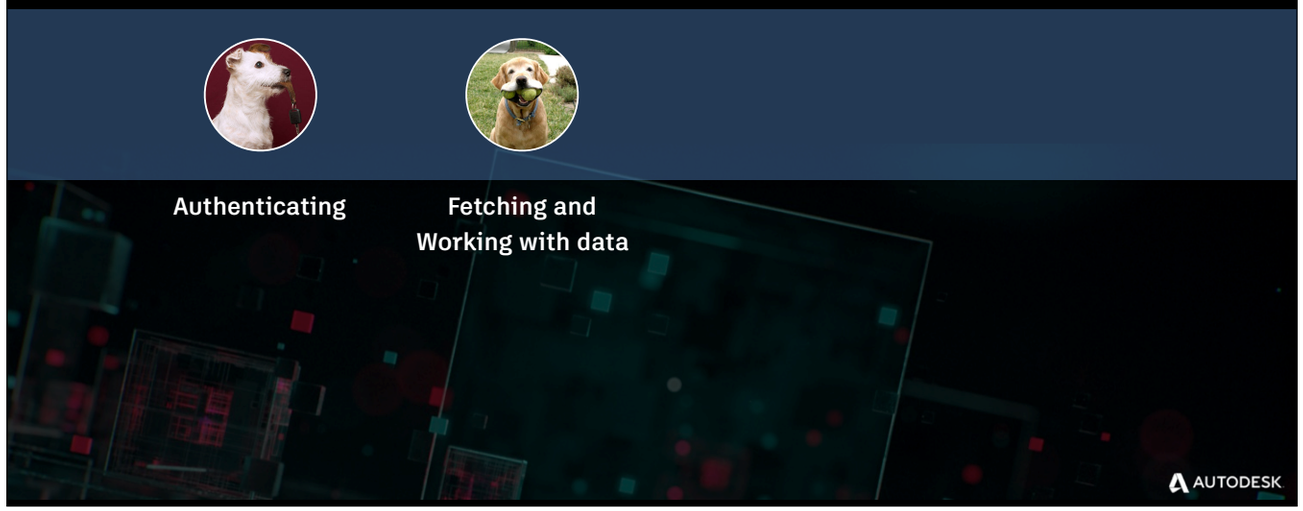


- When people are new to programming with Shotgun they often wonder how and where to get started…
- Well… You’re here, so you’ve already started. Actually, where I like to suggest people start is with the Python API.
- As Tannaz mentioned earlier we do have a REST api but in the interest of time I’m going to be referring you to a great webinar one of our engineers, Brandon Ashworth, recently did about REST and will instead be covering our native Python API. If you’re a beginner, a lot of the concepts of the Python API are easily transferable to the REST api. We’ll also be skipping over the Shotgun Event daemon due to time constraints but I’m here all day and it’ll be my absolute pleasure to talk to you about it and show you how it works if you’re so inclined.

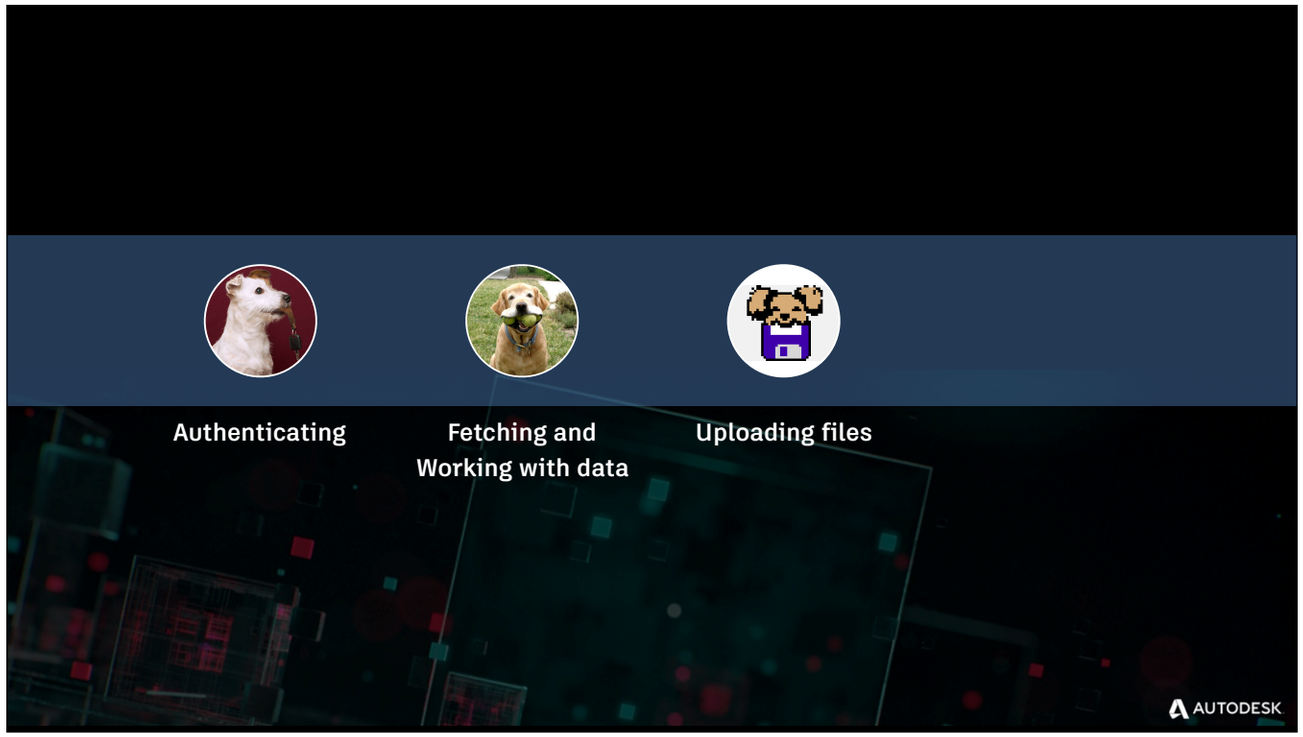
So what’s left? Well, here’s what we’ll be looking at and what I hope your main takeaways from this talk are going to be.



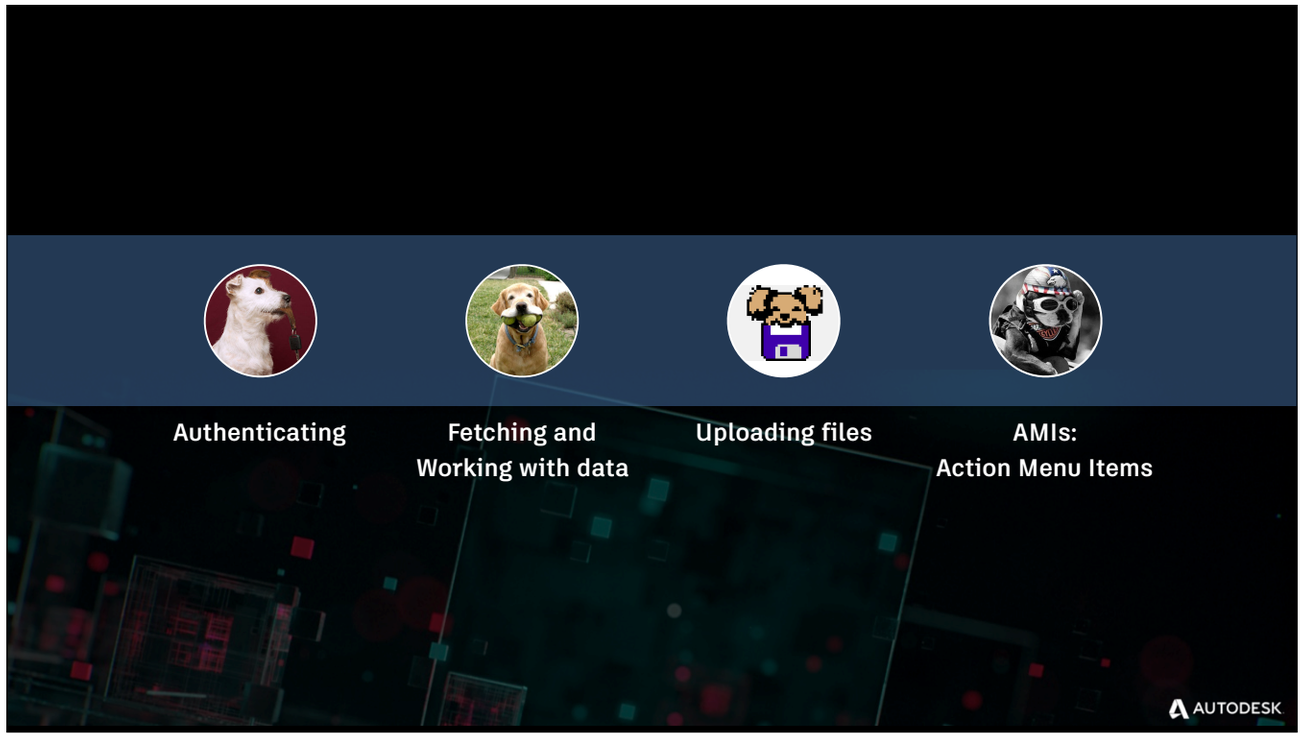
- How to setup a script entity in Shotgun and connect via the Python API…



- 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)



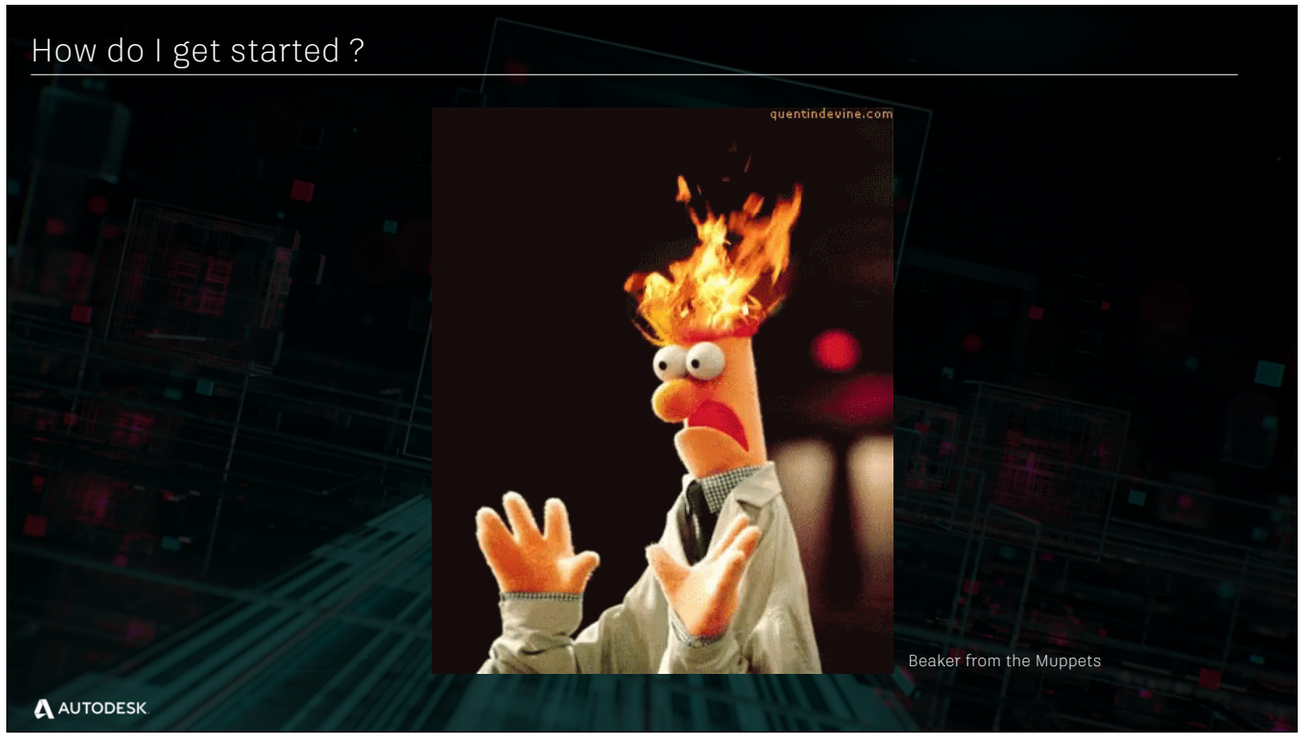
- Uploading via the Python API



- How AMIs work, how to set them up and how to use Flask to write a simple one.

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.



Beaker from the Muppets

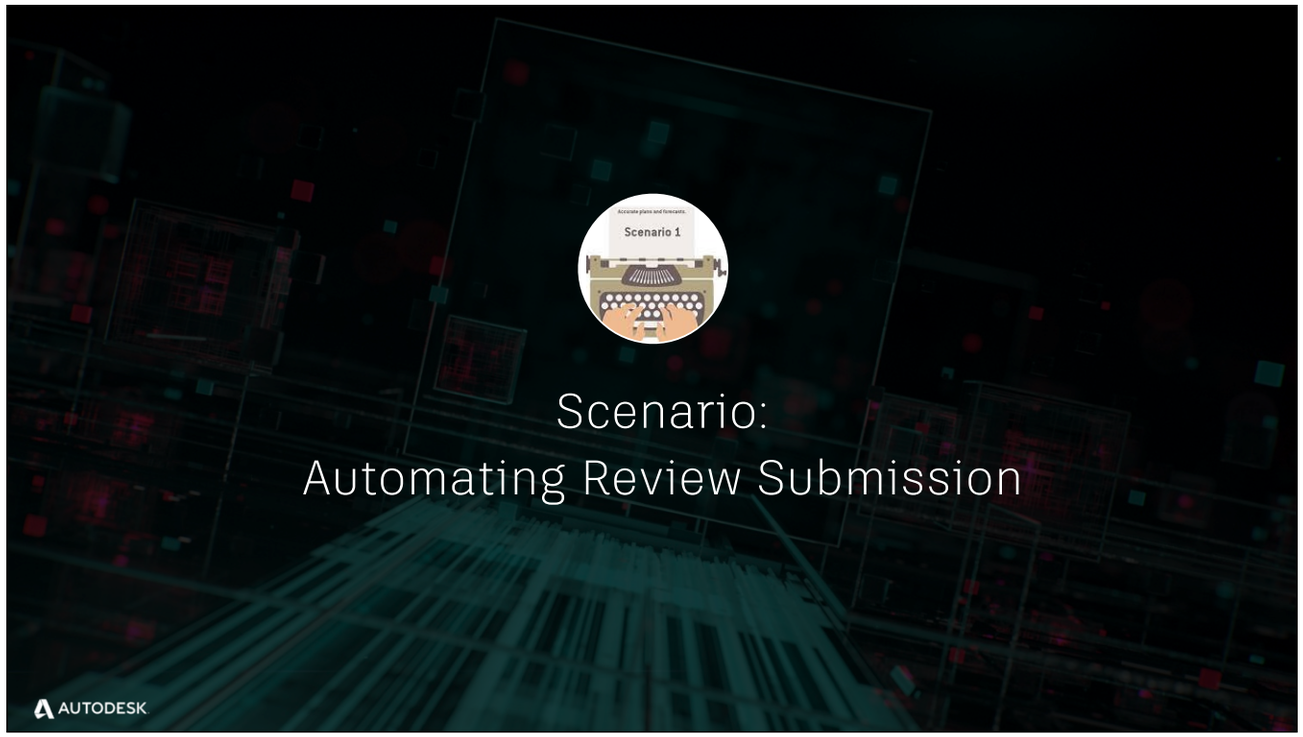
If at any point you feel like this this picture of Beaker is you - that things are going too quickly and are getting out of hand, remember all the friendly Shotgun staff is here to help you, answer your questions and maybe even give you a hug if you need one. Also, the support@shotgunsoftware.com email address is your friend. And don’t forget, all this content will be available in a GitHub repo, just grab a picture of the QR code you’ll be seeing throughout the day.

How do I get started ?

Random cat from the internet

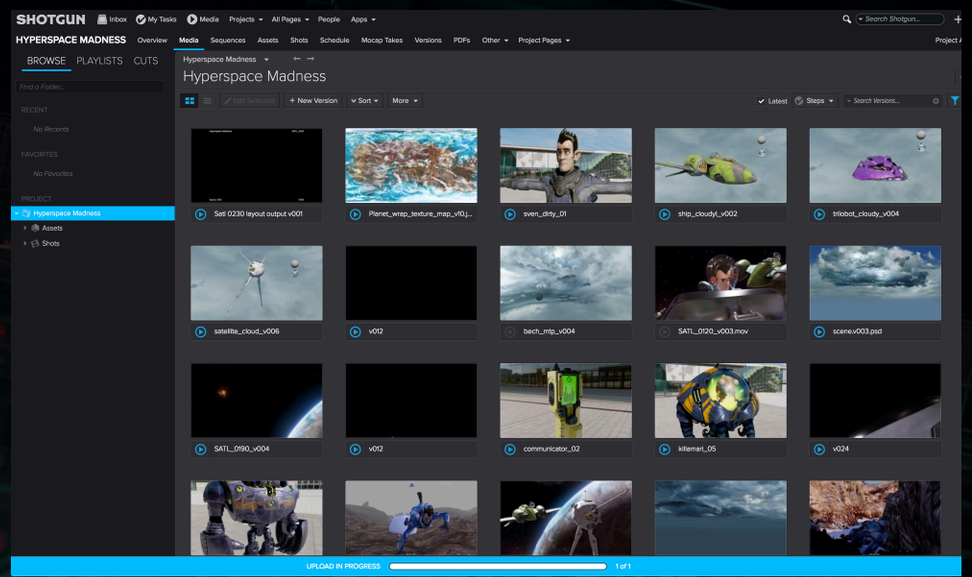
AUTODESK

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



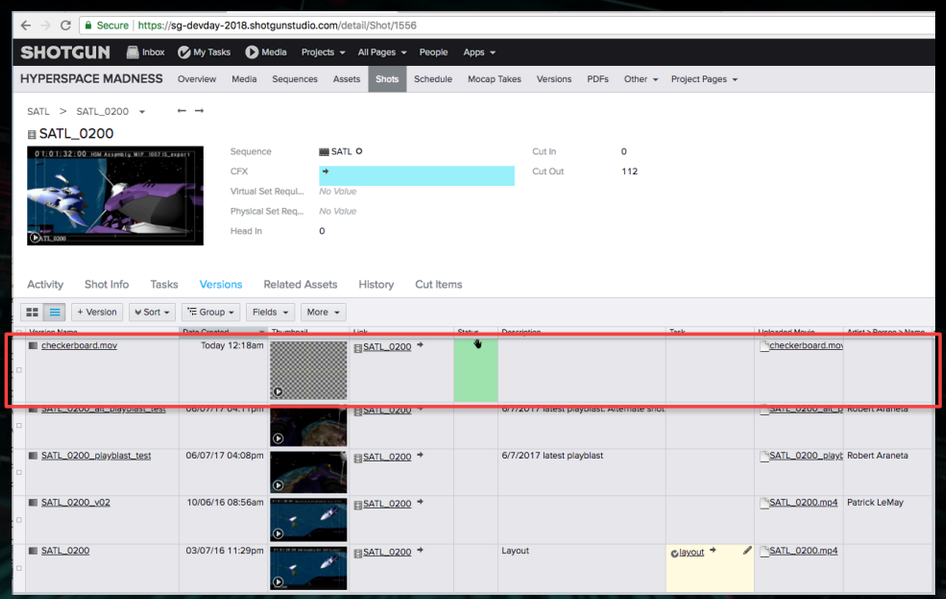
Alright, for the sake of argument, let’s imagine a scenario where you’re using Shotgun for your review process.

Automating Review Submission

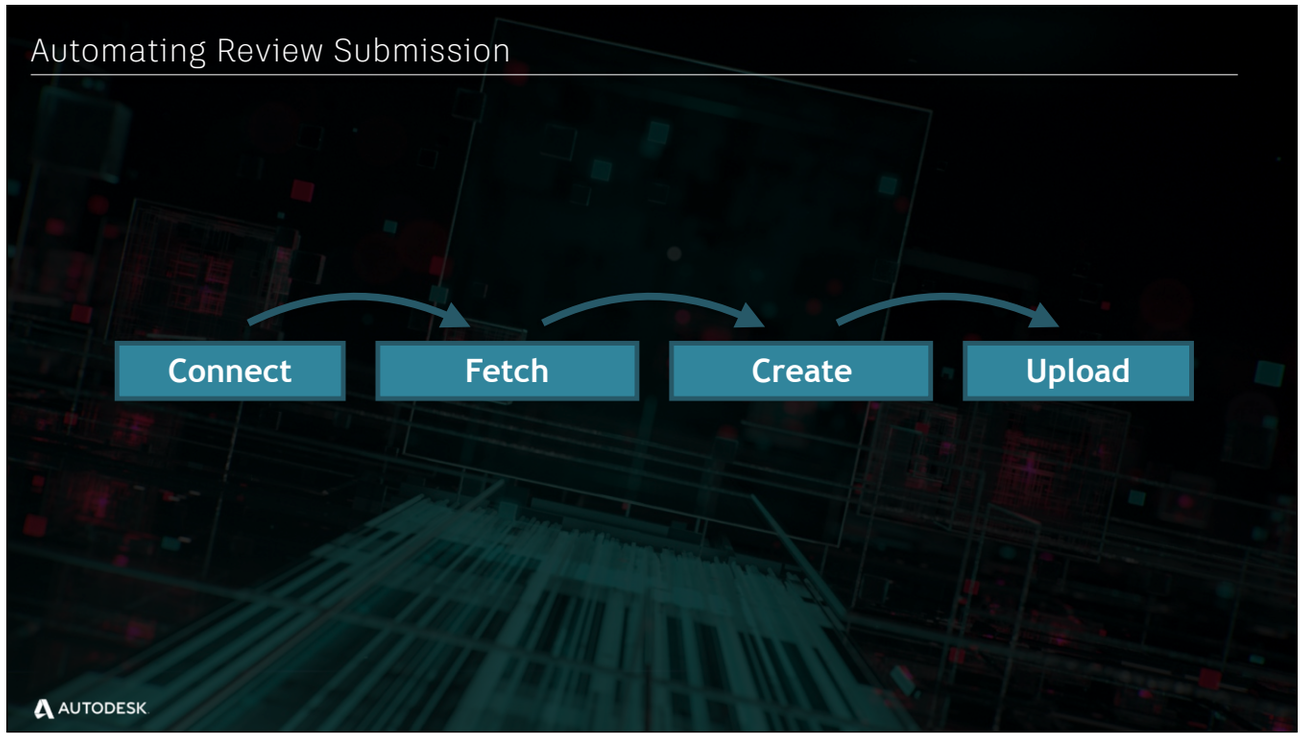


Usually version submission starts out with users uploading content manually via drag and drop in the web app. This can be pretty time consuming, especially if you have 100 versions to submit. This is where the API can come in handy as tools can be written to automate the review submission process. Our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

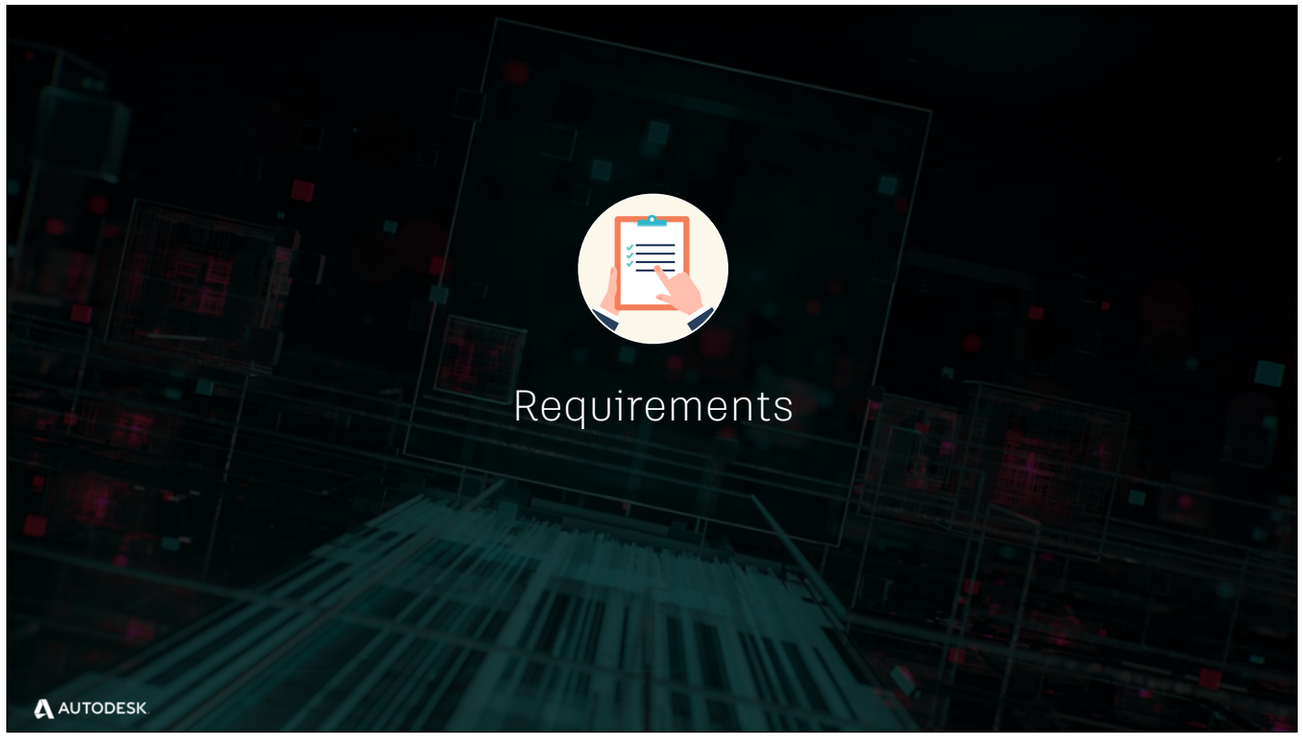
Automating Review Submission



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:



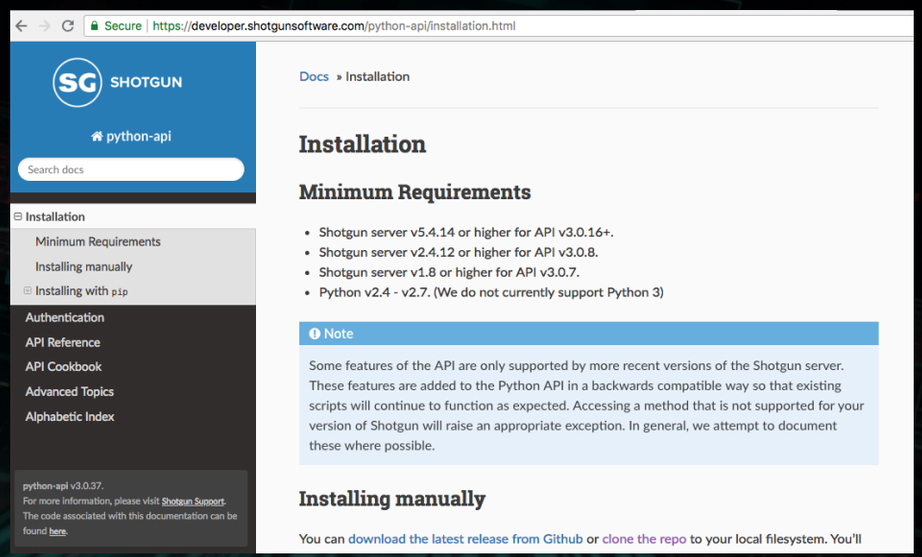
- Connect to Shotgun
- Fetch the project and shot the version will be linked to
- Create the version
- and finally upload the movie.



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.

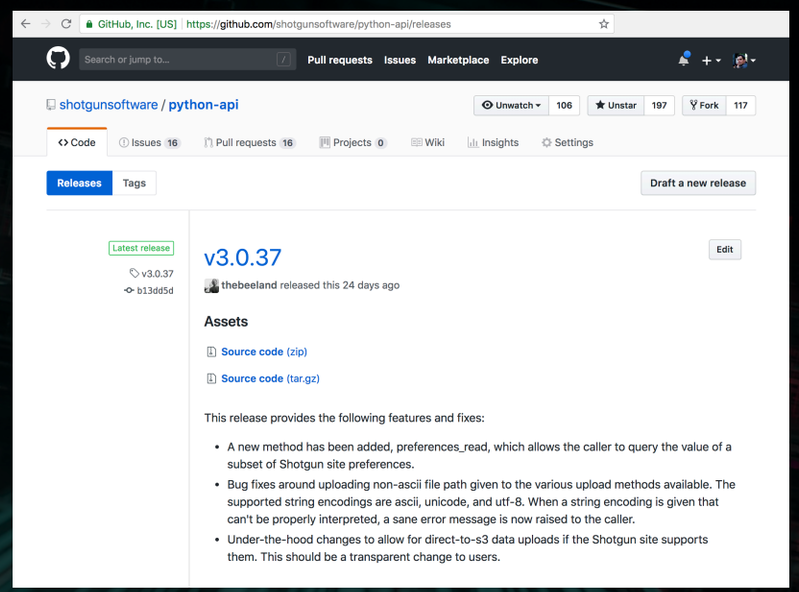
Requirements - Python API



AUTODESK

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.

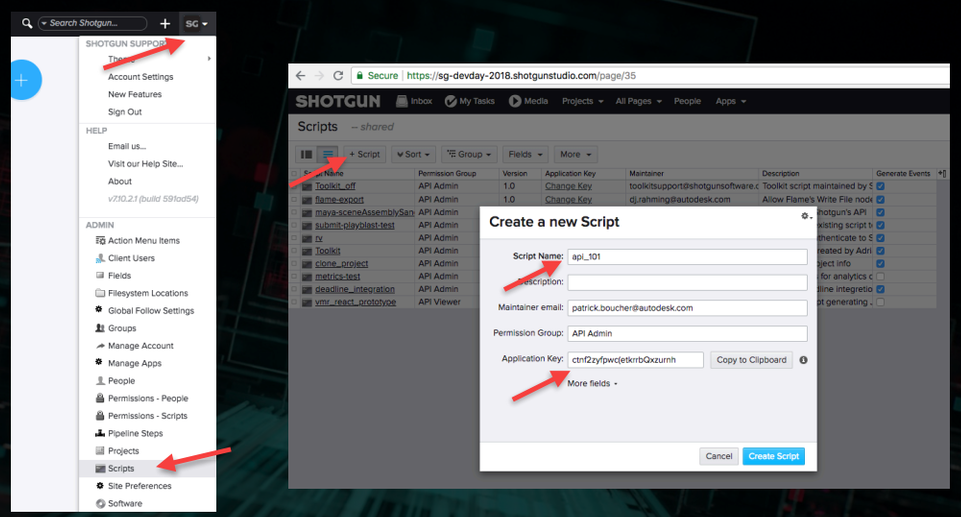
Requirements - Python API



AUTODESK

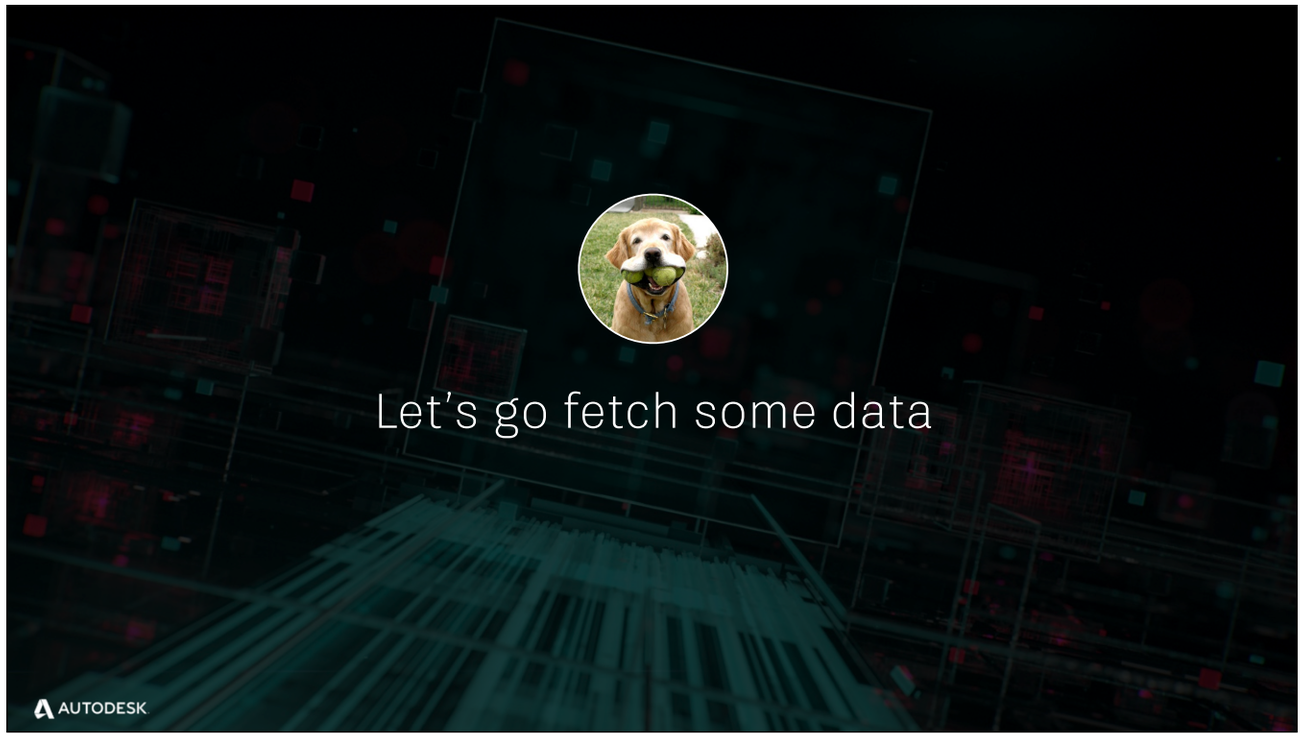
You can also go directly to GitHub when you can grab the latest tag. It’s really easy to download and install.

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.



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

Code: Automating Review Submission

```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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

```
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 we import the Shotgun API so that we can create a connection

Code: Automating Review Submission

```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 - this is just example shorthand. Also, yeah, that script key is bogus, don't worry.

Code: Automating Review Submission

```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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. 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 filters.

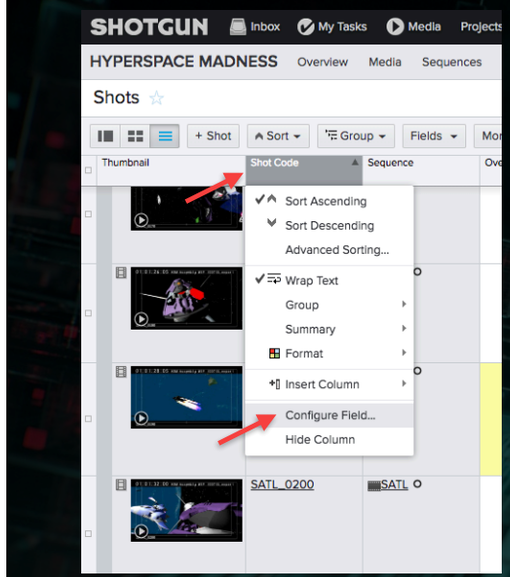
Code: Automating Review Submission

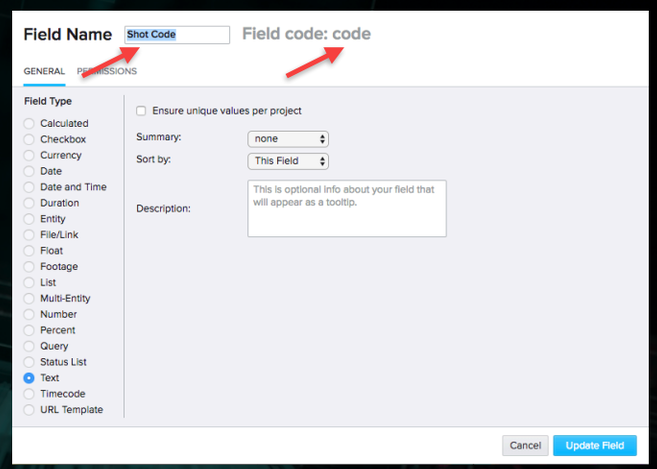
```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 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

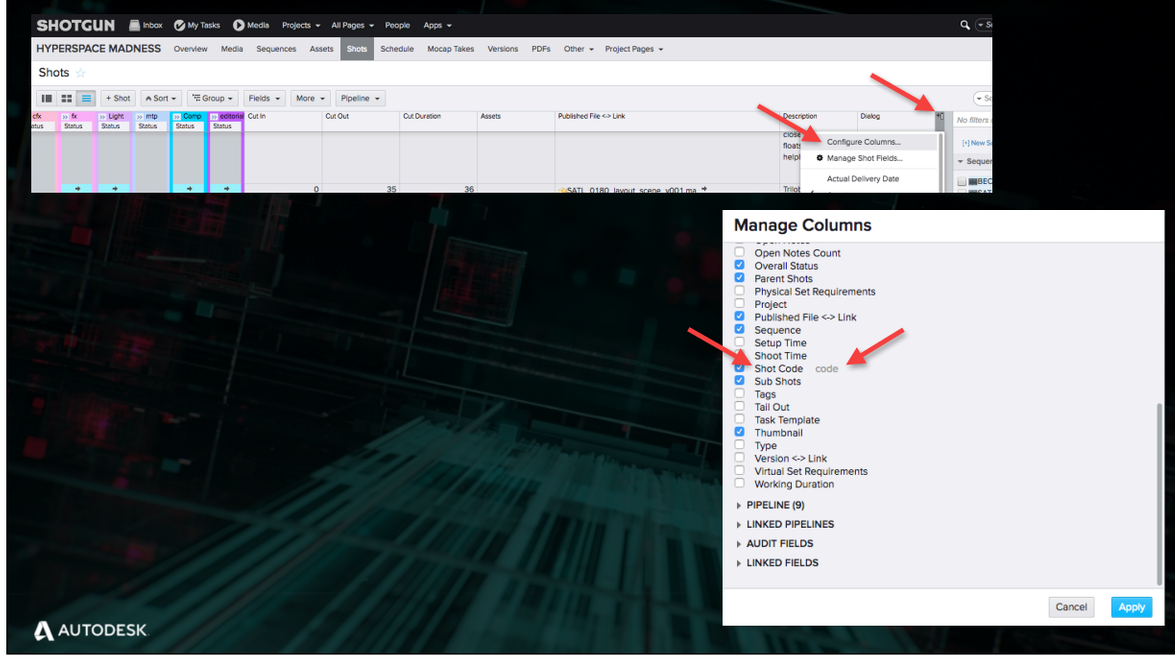




AUTODESK

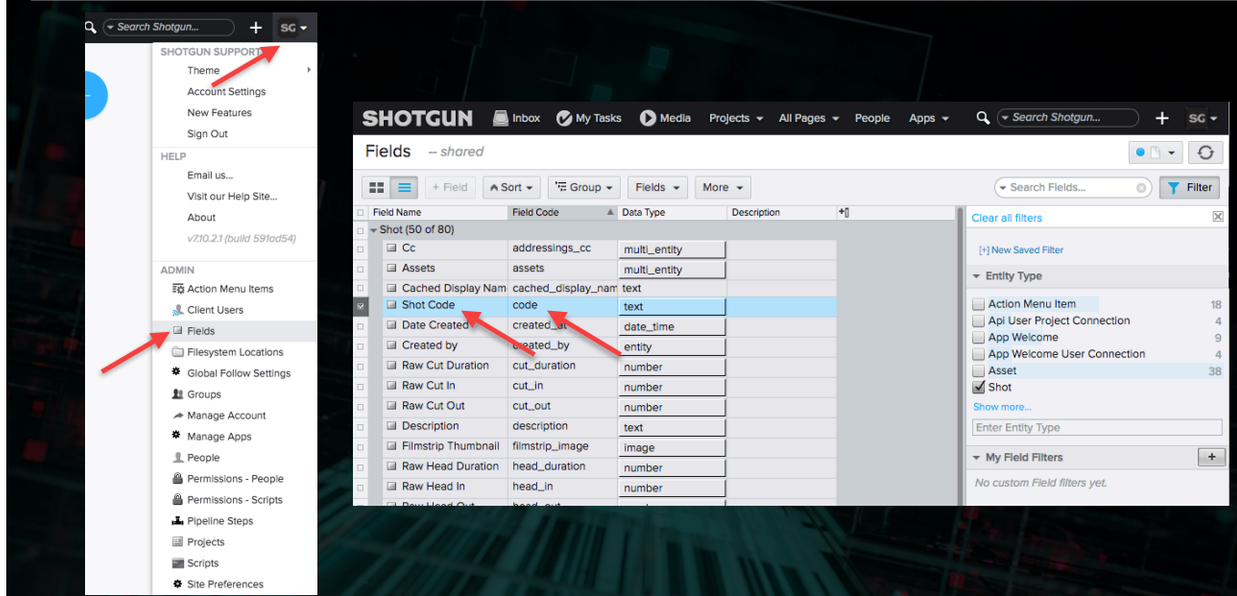
First…
- **In a grid in Shotgun, right click on the column header and…**
- **click `Configure Field…`.**
- **In the configure field dialog you'll see**
- **The column name for the UI**
- **And the field code you need to use in the API**

Field Codes - Configure Columns

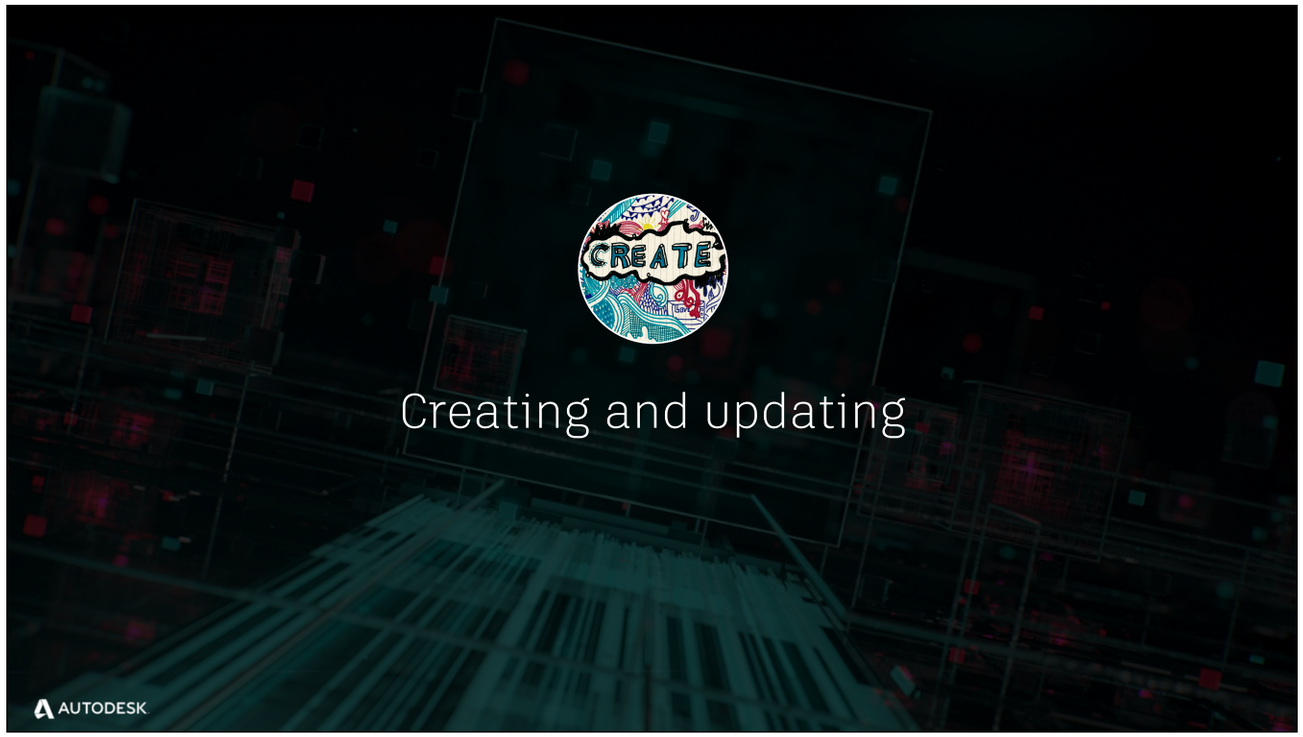


Second
- **In a grid in Shotgun, click on the column management button…**
- **click `Configure Columns…`.**
- **In the dialog**
- **You'll see the column name for the UI**
- **And the field code you need to use in the API**

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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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

Code: Automating Review Submission

```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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'})
```

AUTODESK

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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 it's 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 shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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 as a result of the `create` call. Let's take a minute to look at that version variable more closely.

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'}
```

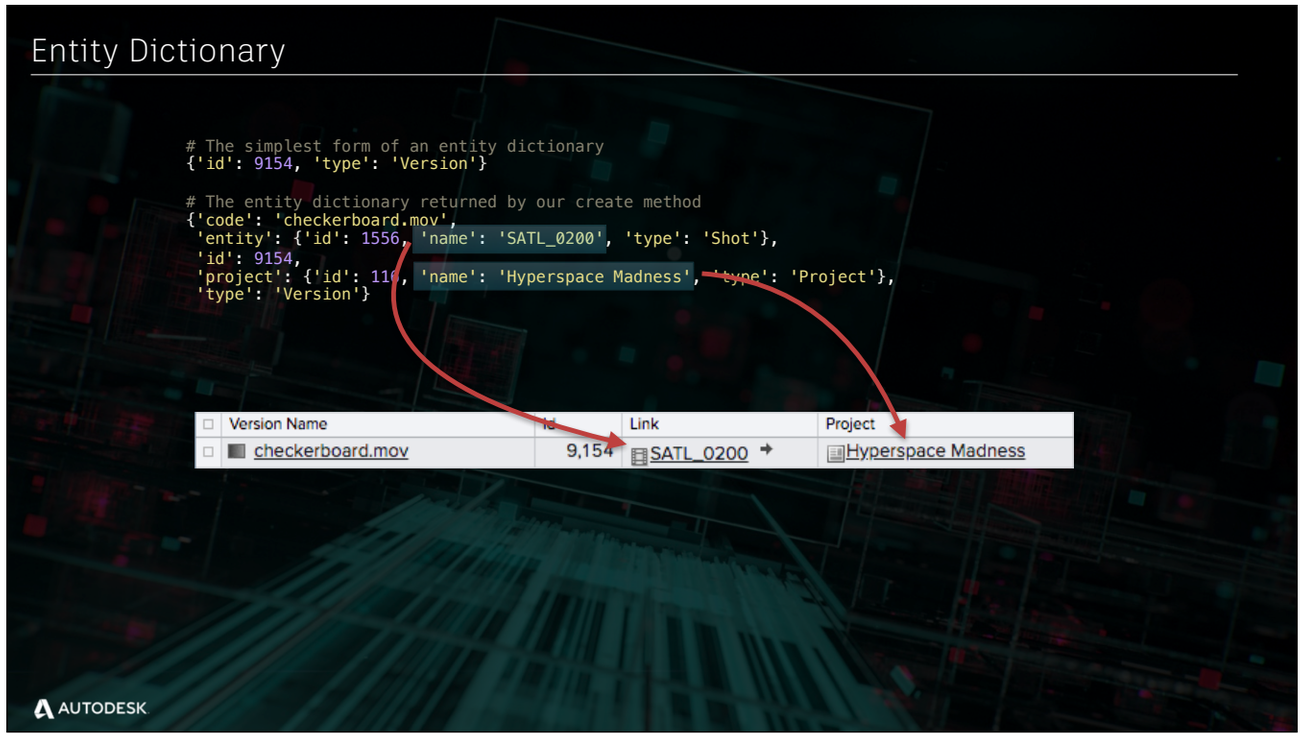
AUTODESK

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 an entity dictionary. Because of the dict 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…

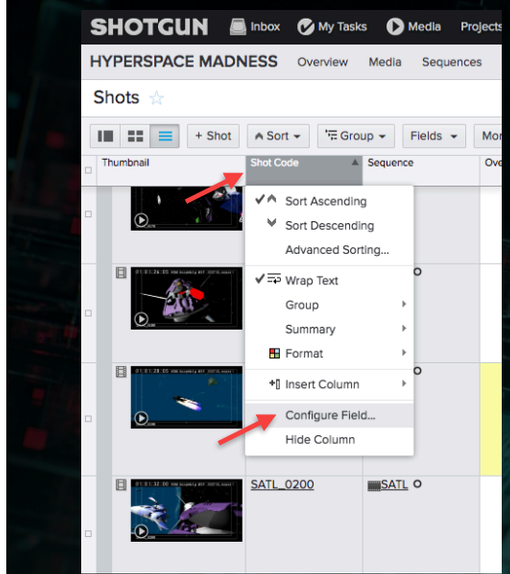
When you create a record in Shotgun you get an object back that includes all the fields you supplied in your creation. So in this case, not only do we have the type and id but we also have the name of our version in the code field as well as entity dictionaries representing the shot and project in the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

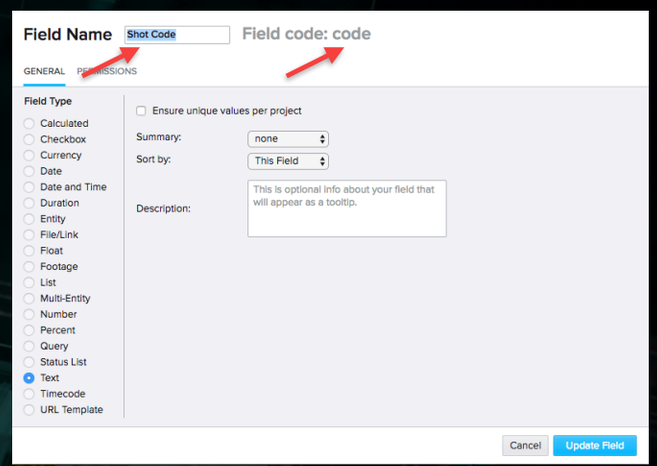
Let’s take a second to explore those shot and project hashes



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.

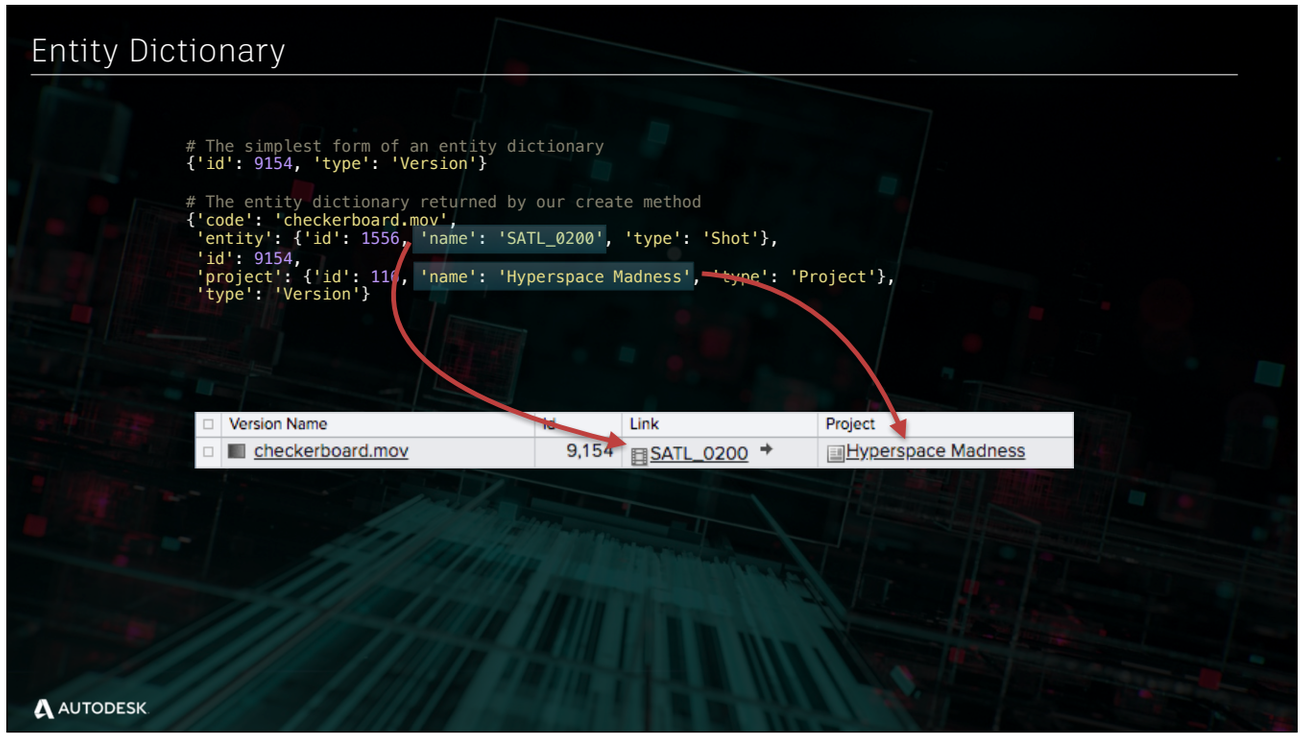
Field Codes - Configure field



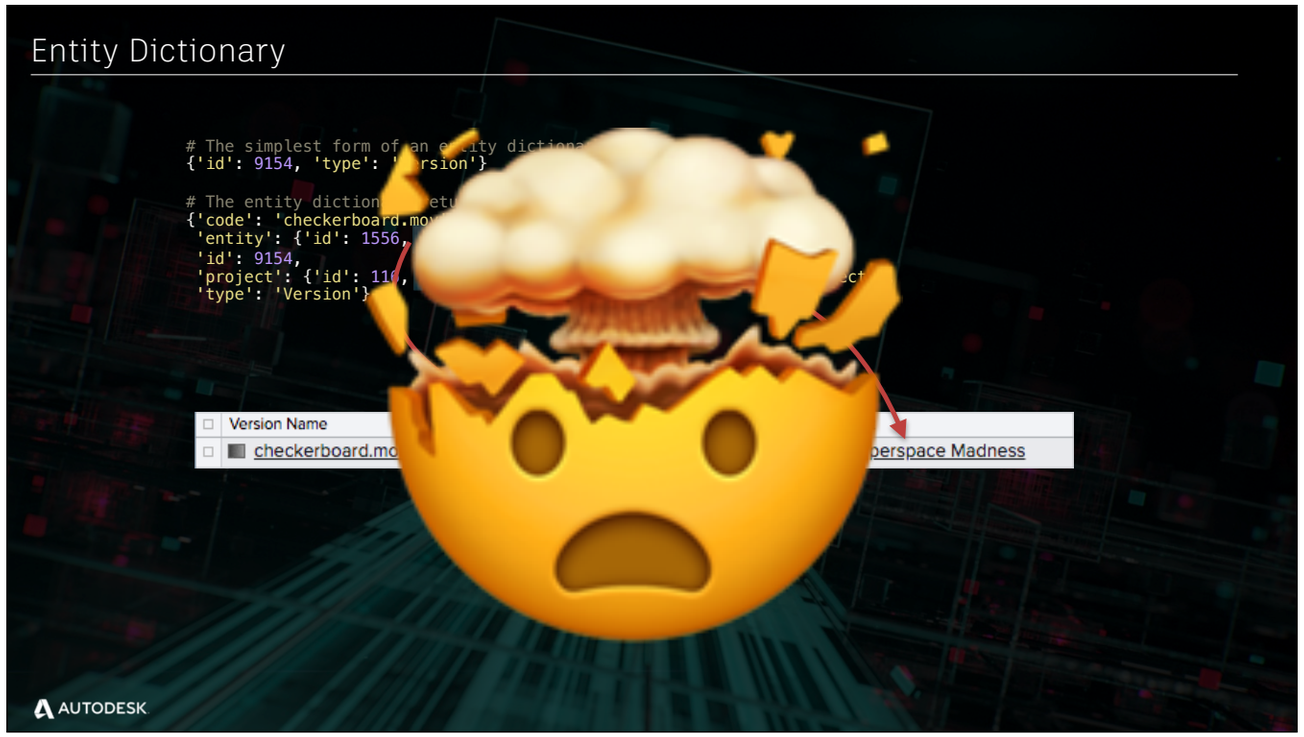


AUTODESK

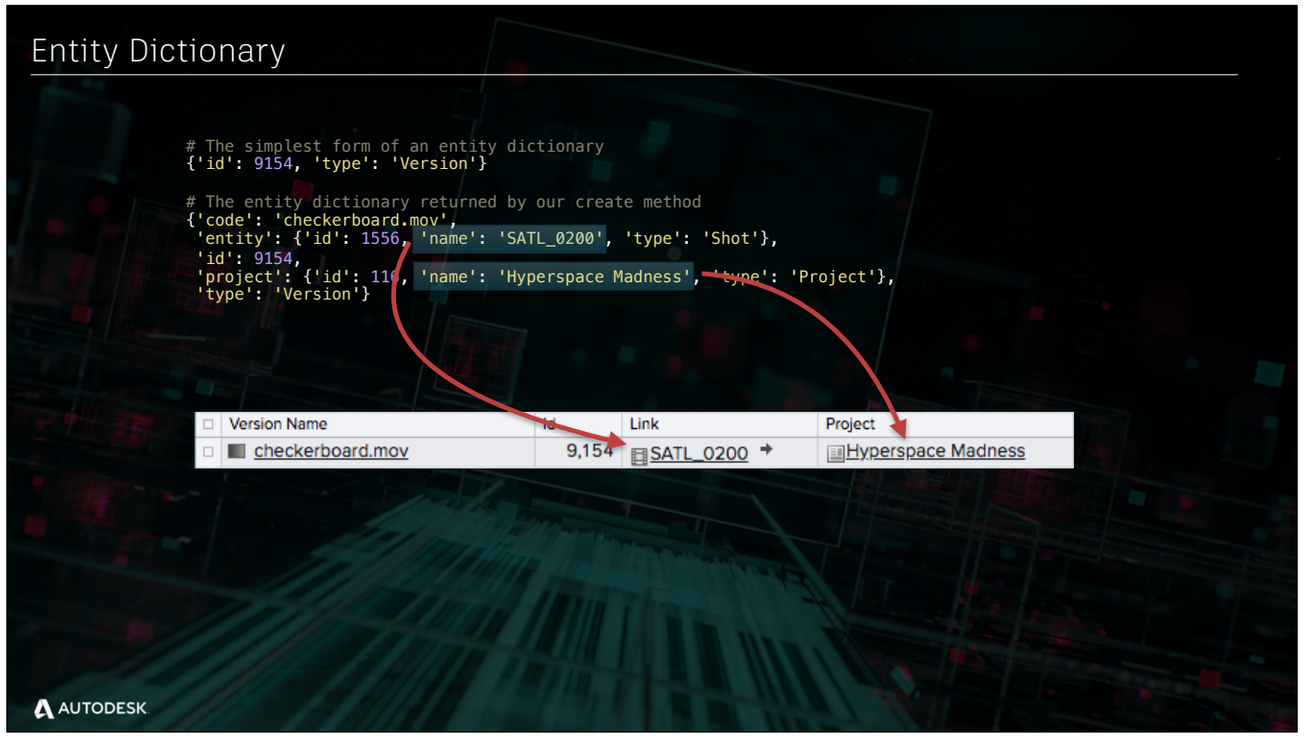
Here’s a reminder…



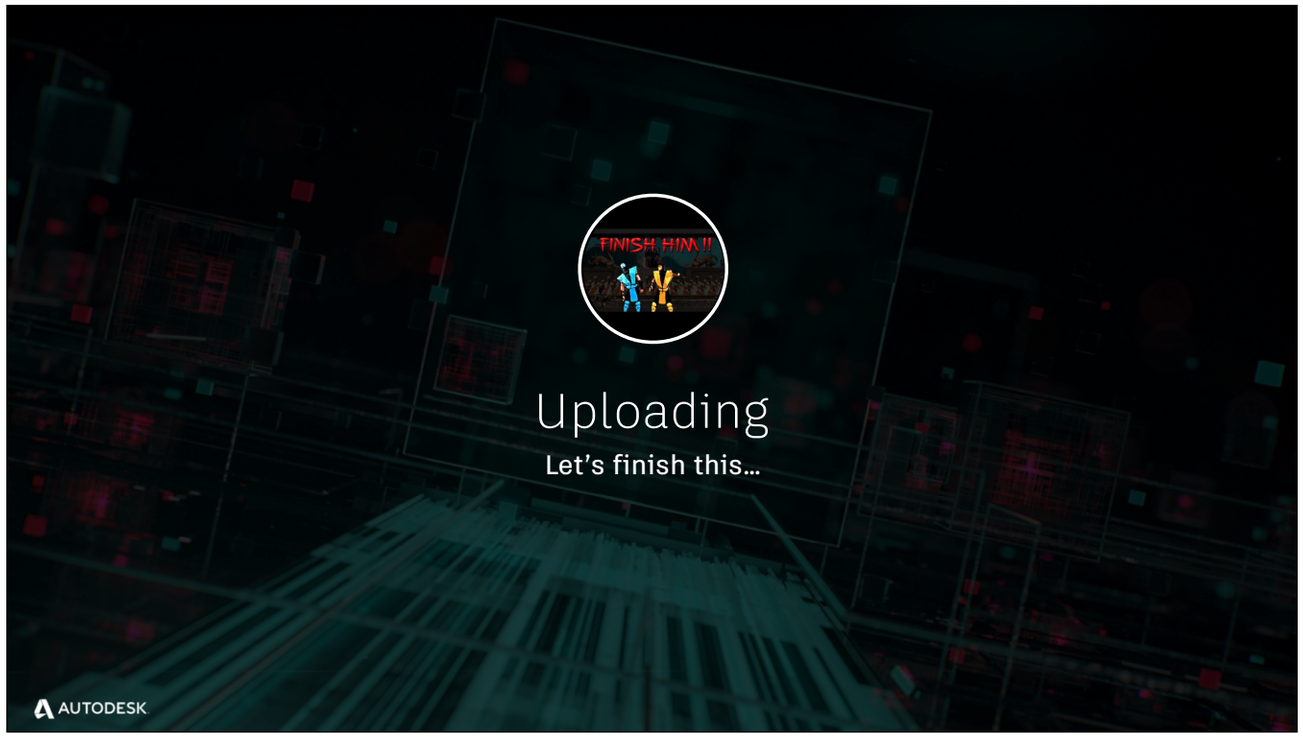
What gives?
Well… for linked entities, this is a third piece of standard information the API returns along with `type` and `id`. We've generalized this as being the `name` of the entity so you can reliably access the key but it represents a human readable name for the records, whichever field really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

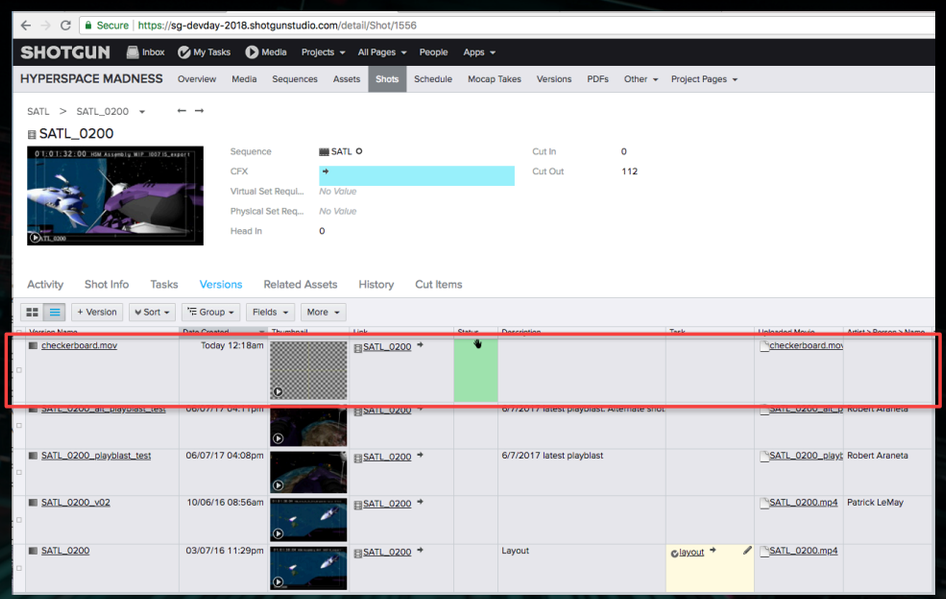
Code: Automating Review Submission

```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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



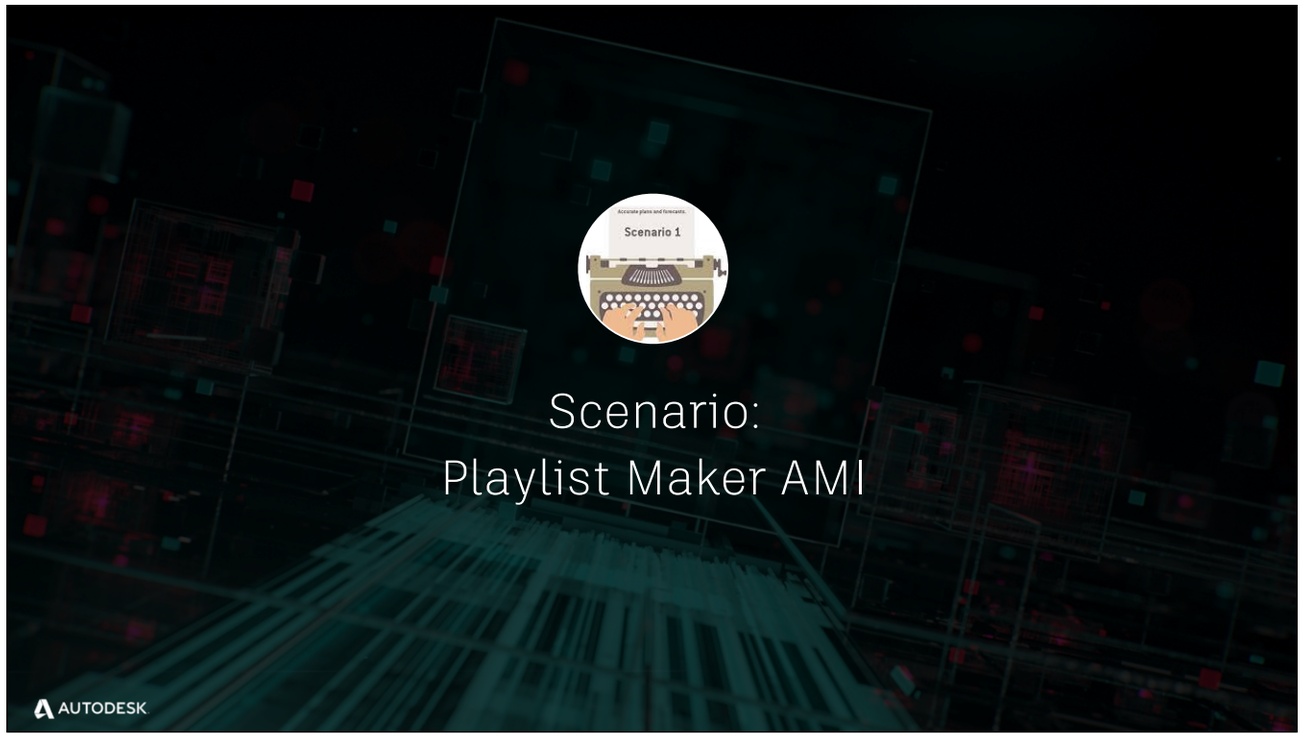
And voilà. We've got our version record, ready for approval.

Code: Automating Review Submission

```python
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

# 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

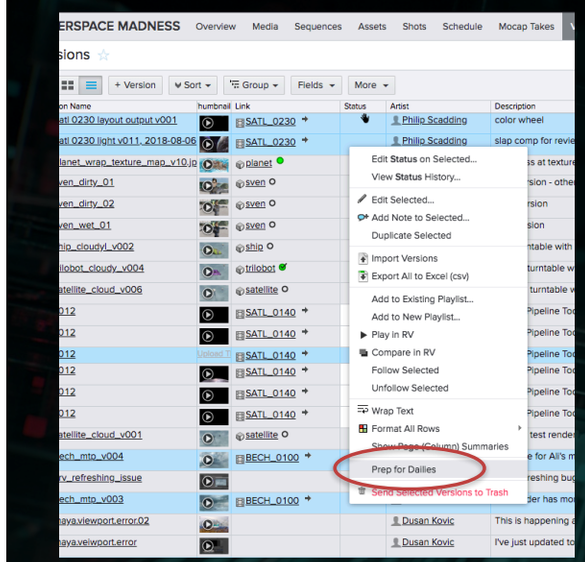
The eagle eyed here (or anyone still mildly paying attention) might ask why there's the update call in the script to set the status to review. Couldn't that have happened in the create call. Yes, it could but then I wouldn't have had an excuse to cover the update call, entity dictionaries and the name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

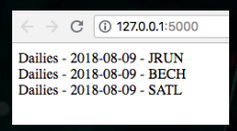


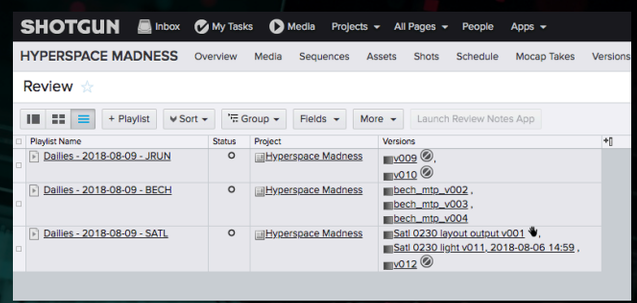
Alright, now that you've got versions for review, here's another scenario for you. You want to create a tool that will prep a dailies session. It lets you select multiple versions, creates one playlist per sequence and adds the appropriate versions to that playlist.

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 Jeff's talk later today.

Playlist Maker AMI





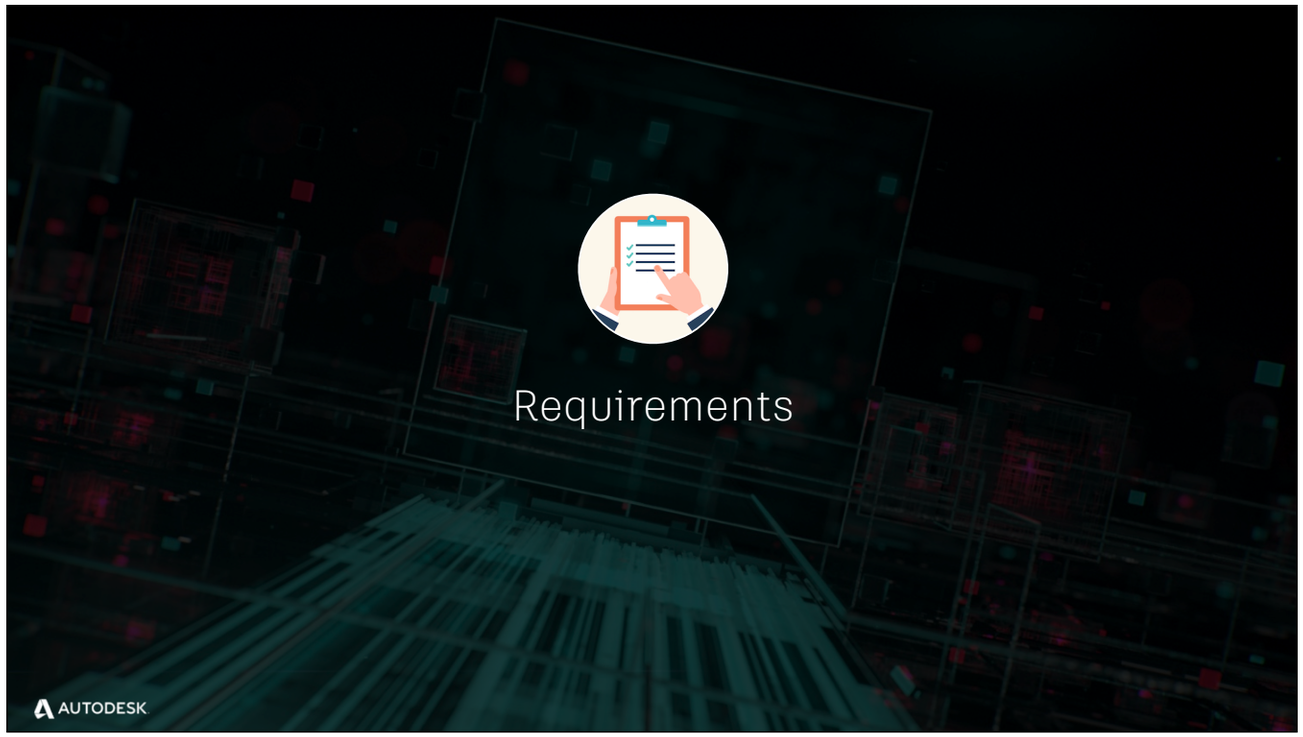


AUTODESK

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 a version in the Hyperspace Madness project**
- **you can expect a small web server to process your request**
- **and create a set of playlists for you that will be visible in Shotgun and playable in Shotgun's review tools - even shareable with clients.**

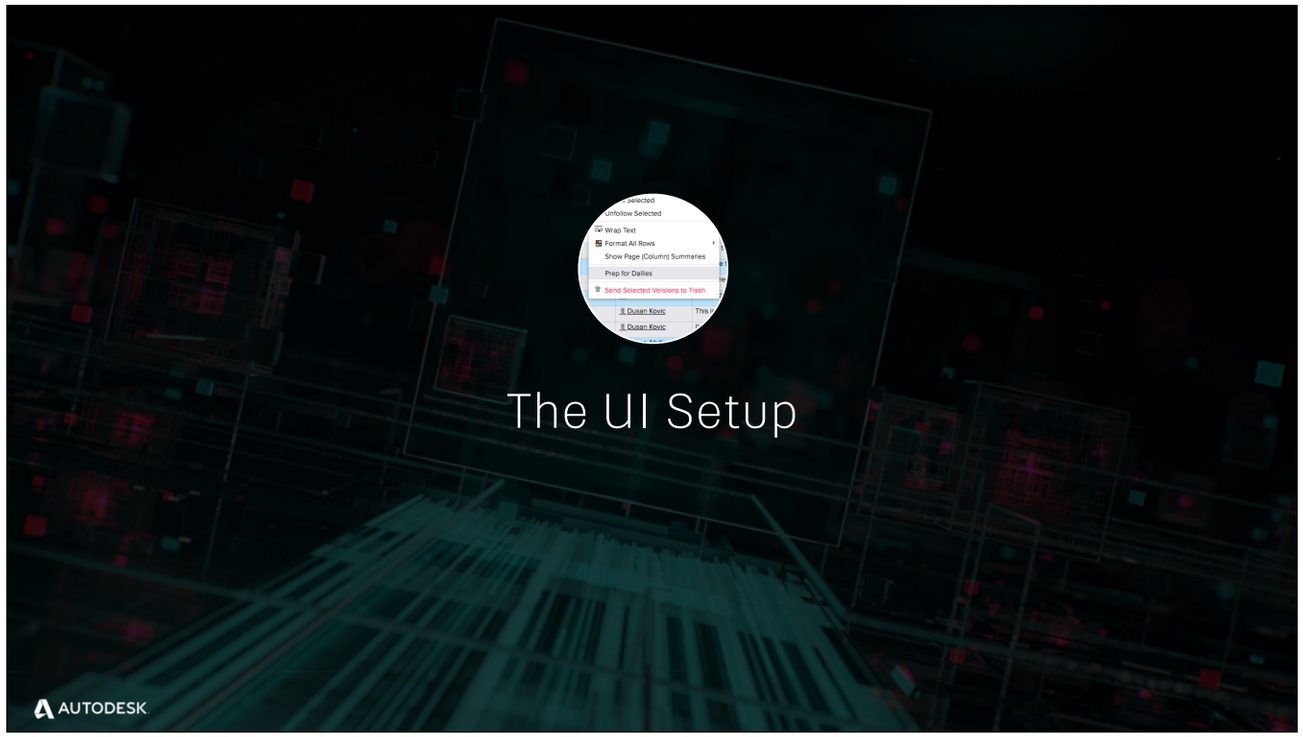
This, again, is a very simple implementation that doesn't pretend to cover all possible usage scenarios or error cases - take it as it is, an excuse for learning about the API.

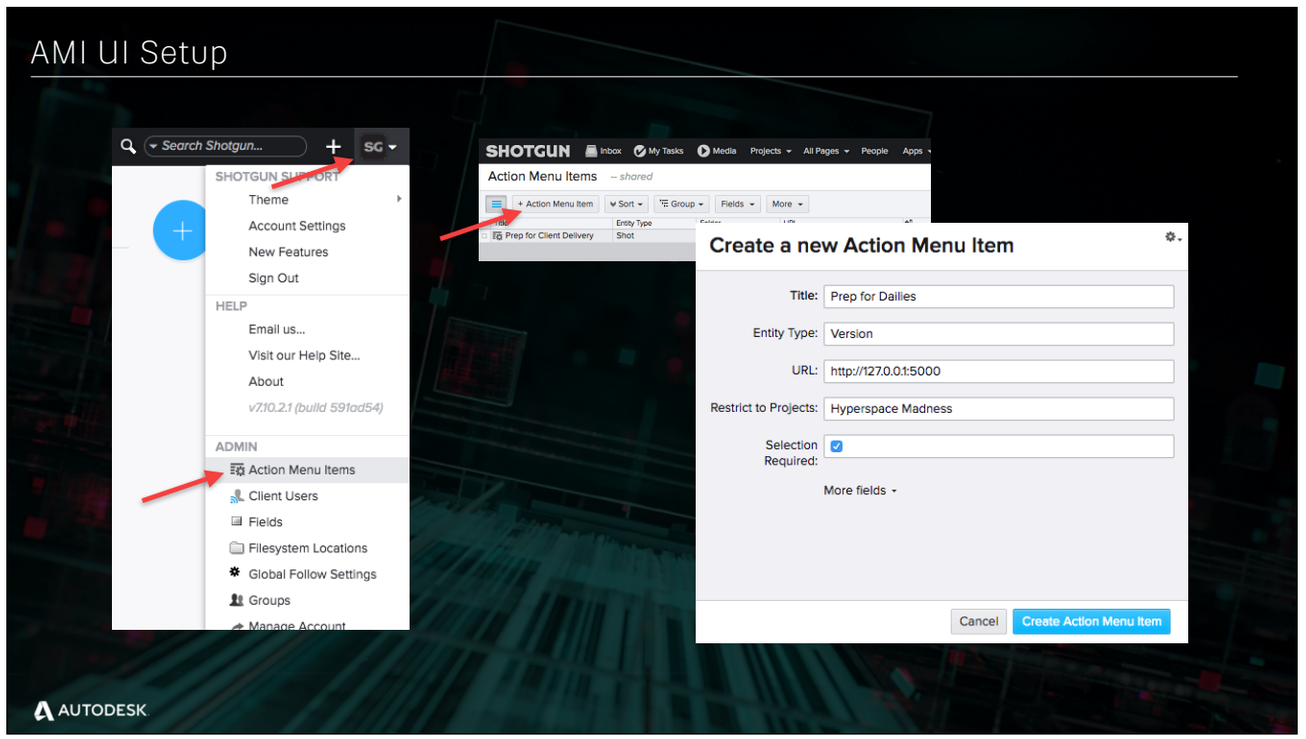


Let’s talk about what you’ll need to make this work.

Requirements - Flask

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.


Unfollow Selected
Wrap Text
Format All Rows
Show Page (Column) Summaries
Prep for Dailies
Send Selected Versions to Trash
Dusan Kovic
Dusan Kovic
The UI Setup
AUTODESK

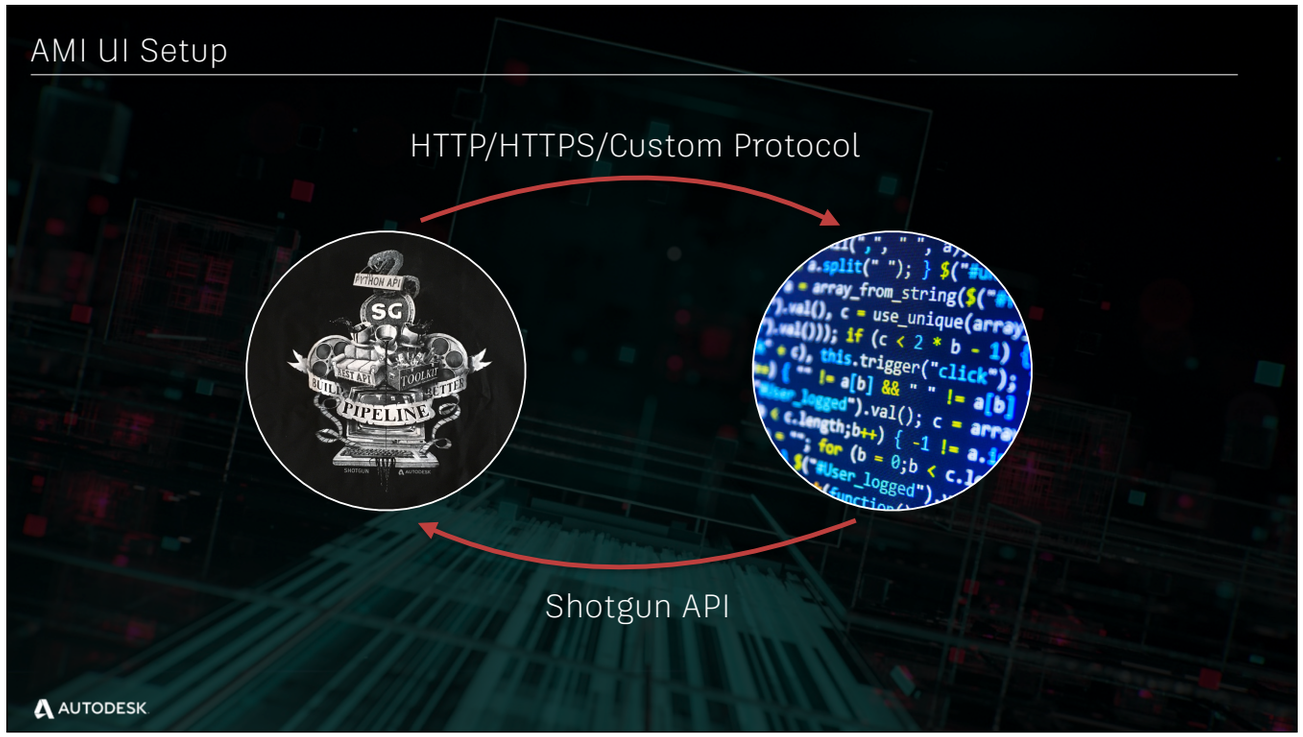


To create your AMI
- As an admin you'll open the user menu
- and click on `Action Menu Items`
- In the `Action Menu Items` page you'll
- add a new action menu item
- And populate it as follow

Because we're setting up our AMI to work on version entities and in the Hyperspace madness project, 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 versions selected before you can invoke the menu item. 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 back information 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-2018.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

Here’s an example of the data you can expect to be sent 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-2018.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**

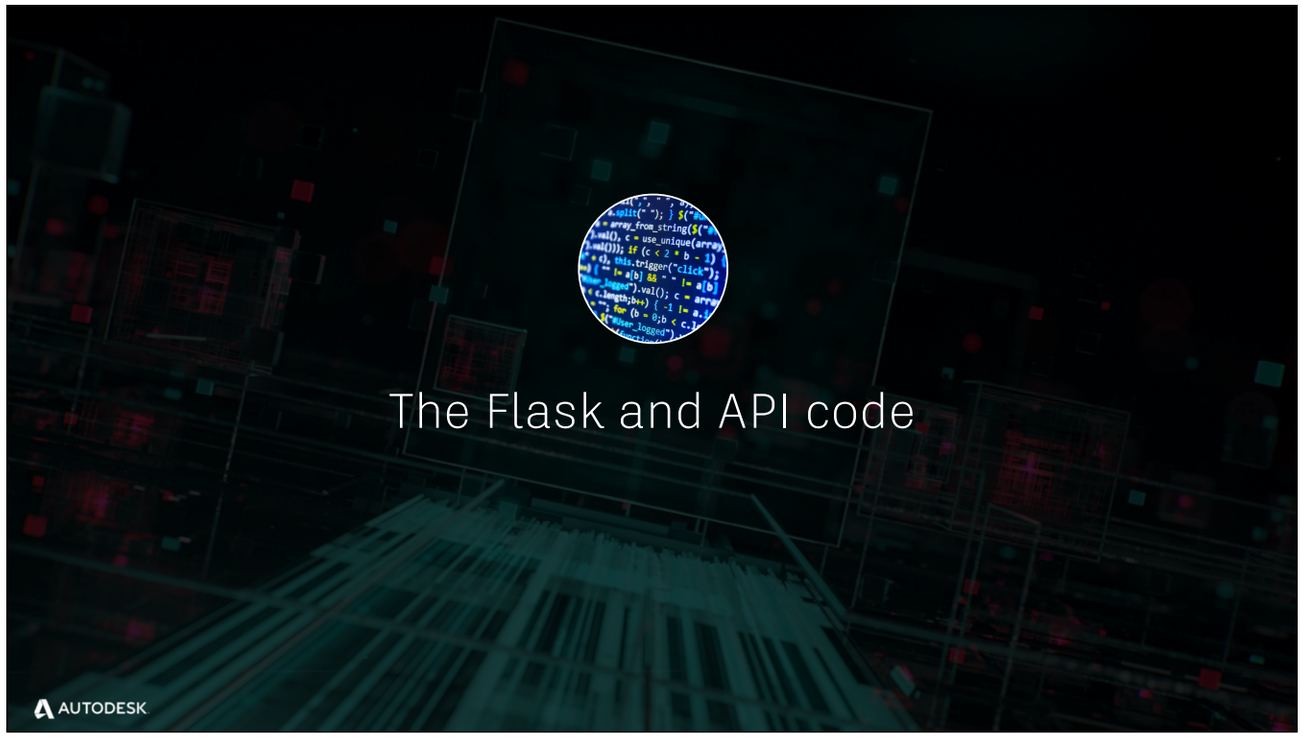
AMI UI Setup

```
# This is a Flask request object's form converted to a dict for easy pretty printing
{
    'server_hostname': [u'sg-devday-2018.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
from flask import Flask, request
import shotgun_api3

SITE_URL = 'https://sg-devday-2018.shotgunstudio.com'
SCRIPT_NAME = 'api_101'
SCRIPT_KEY = 'x~eglc2jpzrhghtcpbzvyymvM'

app = Flask(__name__)


@app.route("/", methods=['GET', 'POST'])
def ami_endpoint():
    return process_versions()


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 - again, this isn't secure, don't write auth credentials in your code**
- **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_versions` and that's what we're going to be concentrating on from here on in.

Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

This should all feel pretty familiar to you by now but while I'll gloss over a few things, like creating the Shotgun connection, I'll take a few seconds to call out one particularly interesting thing. Let's start at the top…

Processing Versions

```python
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

First we start by extracting from the incoming AMI payload the entity type we're working with. In this case, we know it would be a Version because we setup our AMI to work on versions but hardcoding it would have been bad practice.

Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

Then we extract the project id from the AMI payload and artificially construct an entity dictionary to represent the project. This is perfectly legitimate. You can manually build entity dictionaries like this and pass them onto Shotgun API methods because that what they expect, just standard dictionaries. As long as you supply the prerequisite `type` and `id` keys, you'll be fine.

Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

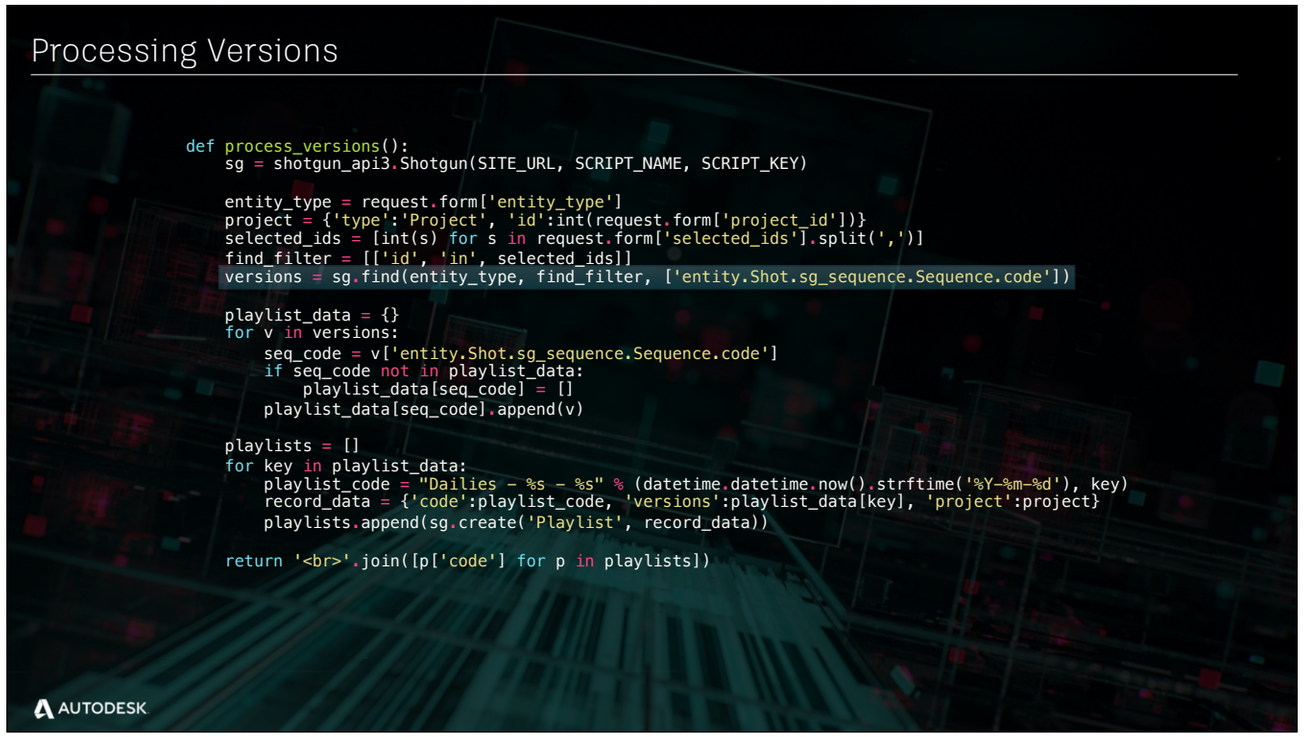
The next step is to extract the ids of the selected versions from the AMI payload. These are the ids of the records that were highlighted in the grid at the time the AMI was invoked. Because of the nature of information being passed in a POST request as standard form data, these are strings and need to be converted to int.

Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

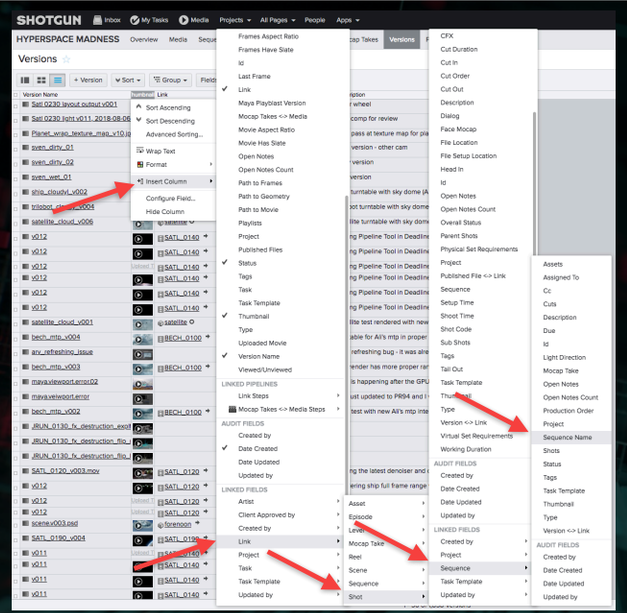
AUTODESK

We then build a simple single condition filter which we'll use in an API find call to get all the versions we want to organize into playlists.



Now here is what I find the most interesting in the AMI scenario and what should be one of your big takeaways. I've previously glossed over the third argument to a `find` or `find_one` call… The third argument is a list of fields whose information you want returned to you in the entity dictionaries. You can specify any field on the entity itself like in the case of a version you could ask for `code`, `project`, or `description`, but you can also use linked field syntax.

Linked fields



'entity.Shot.sg_sequence.Sequence.code'

AUTODESK

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 looking for the `code` of the sequence that is linked to the Shot our version is linked to.
What's the point of this… Well we want to create a playlist for each sequence our versions belong to, so that means we need to understand which shot they're linked to and which sequences those shots are linked to. Doing all those find calls (at the very least 3 if you've optimized your stuff really well) would be very expensive, now we can get to all our information in a single call using linked field syntax.

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.

Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

Alright, back to our code. The next step here is all bout organizing our versions in sequence groups based on the information we retrieved with our linked field syntax.

Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

Next, is a block of code where we take all the organized playlist information and call out to Shotgun via the `create` method you already know about.
What is interesting to note here is that the versions field on the playlist entity is a multi-entity field. That means the field can hold many versions. It should be noted that it's as simple as supplying a list of entities in the data dictionary and the multi-entity field will be populated with more than one item. Easy peasy.

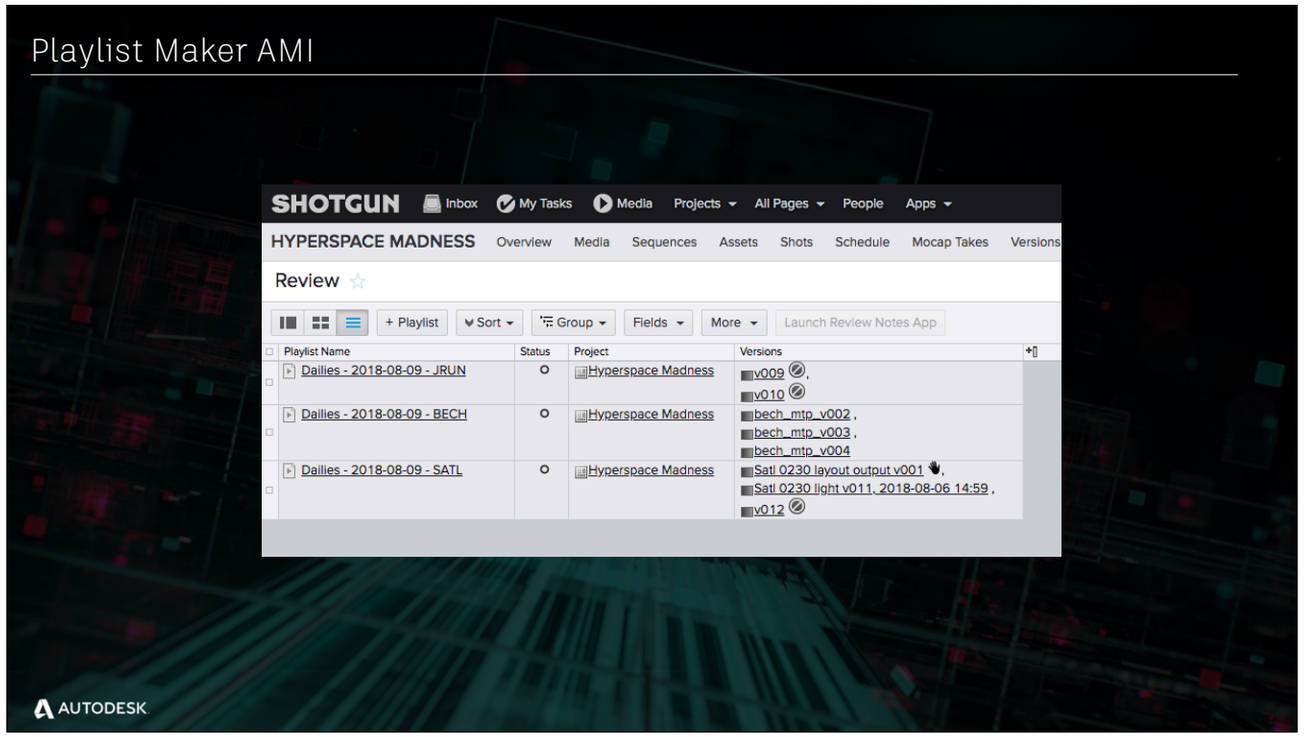
Processing Versions

```
def process_versions():
    sg = shotgun_api3.Shotgun(SITE_URL, SCRIPT_NAME, SCRIPT_KEY)

    entity_type = request.form['entity_type']
    project = {'type':'Project', 'id':int(request.form['project_id'])}
    selected_ids = [int(s) for s in request.form['selected_ids'].split(',')]
    find_filter = [['id', 'in', selected_ids]]
    versions = sg.find(entity_type, find_filter, ['entity.Shot.sg_sequence.Sequence.code'])

    playlist_data = {}
    for v in versions:
        seq_code = v['entity.Shot.sg_sequence.Sequence.code']
        if seq_code not in playlist_data:
            playlist_data[seq_code] = []
        playlist_data[seq_code].append(v)

    playlists = []
    for key in playlist_data:
        playlist_code = "Dailies - %s - %s" % (datetime.datetime.now().strftime('%Y-%m-%d'), key)
        record_data = {'code':playlist_code, 'versions':playlist_data[key], 'project':project}
        playlists.append(sg.create('Playlist', record_data))

    return '<br>'.join([p['code'] for p in playlists])
```

AUTODESK

Finally we just return a bit of information for our Flask server to return some content.

Playlist Maker AMI



Once all this back and forth between Shotgun, your AMI code and back to Shotgun has happened, you're left with a list of playlists that your users can view and use in dailies.

What's left?

- Deleting / Reviving
- Batch calls
- Summarizing
- Following
- Schema methods
- REST API specifics
- The Shotgun Event Daemon
- And so much more...


Question, comments,
verbal abuse?
AUTODESK


Q & A
AUTODESK


AUTODESK

DEV DAY RESOURCES

github.com/shotgunsoftware/sg-devday-2018