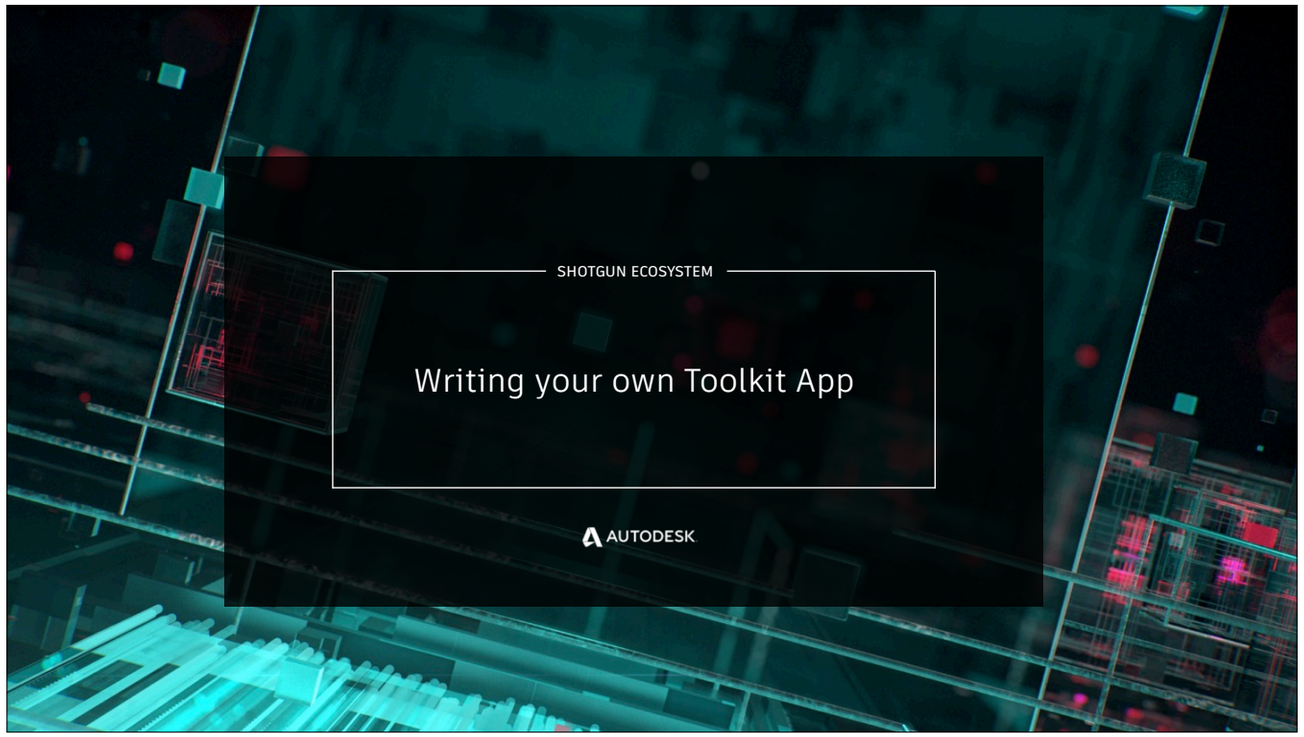
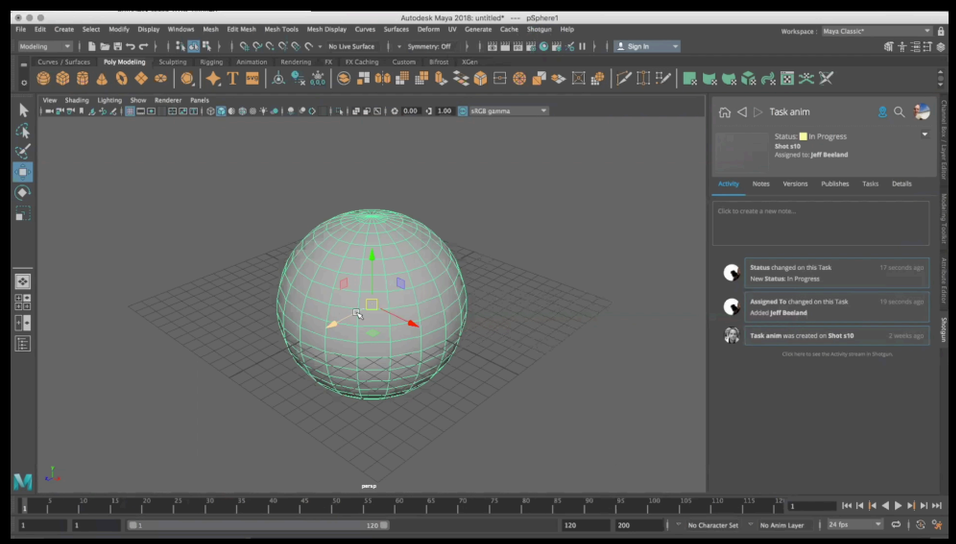

SHOTGUN ECOSYSTEM
Writing your own Toolkit App
AUTODESK.

Jeff Beeland

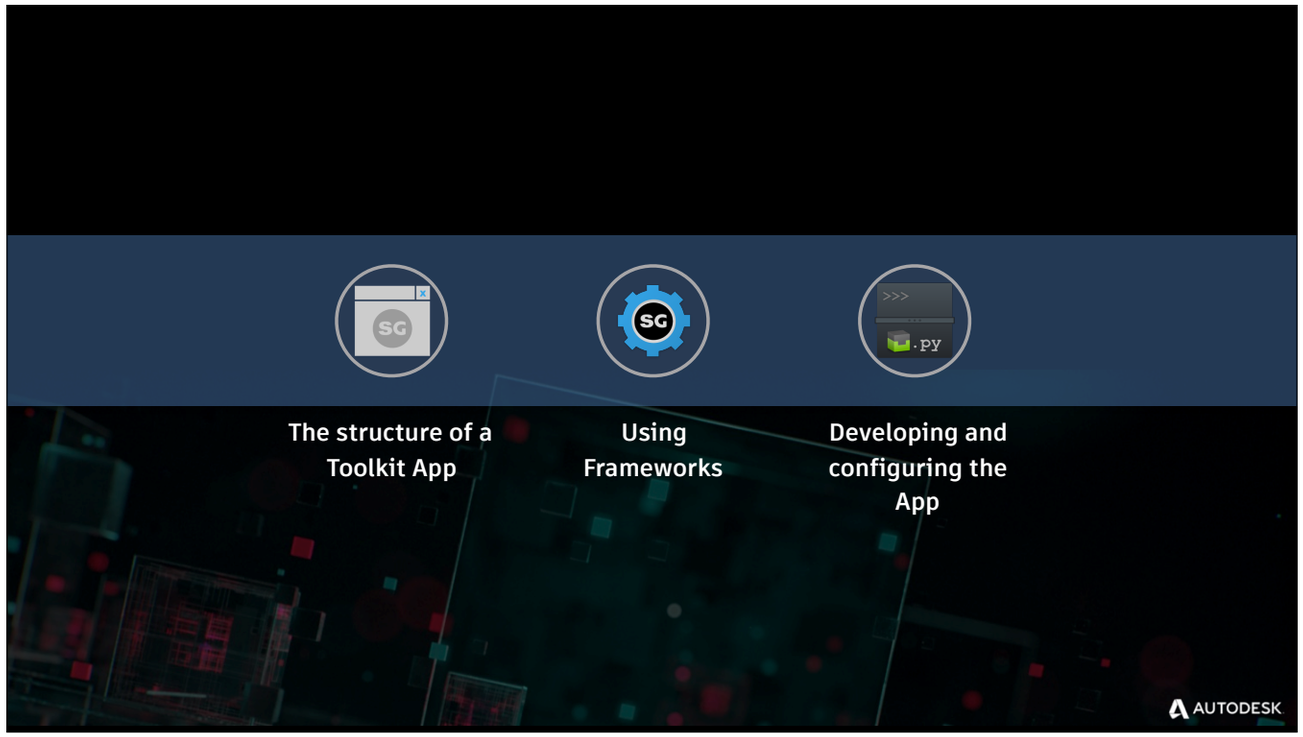
Software Development Manager

Jeff spent 11 years developing and supporting VFX and Animation pipelines at Rhythm and Hues and Blur Studio. For the past 3 years, he has worked as an engineer working on Shotgun as part of the Toolkit and Ecosystem teams.

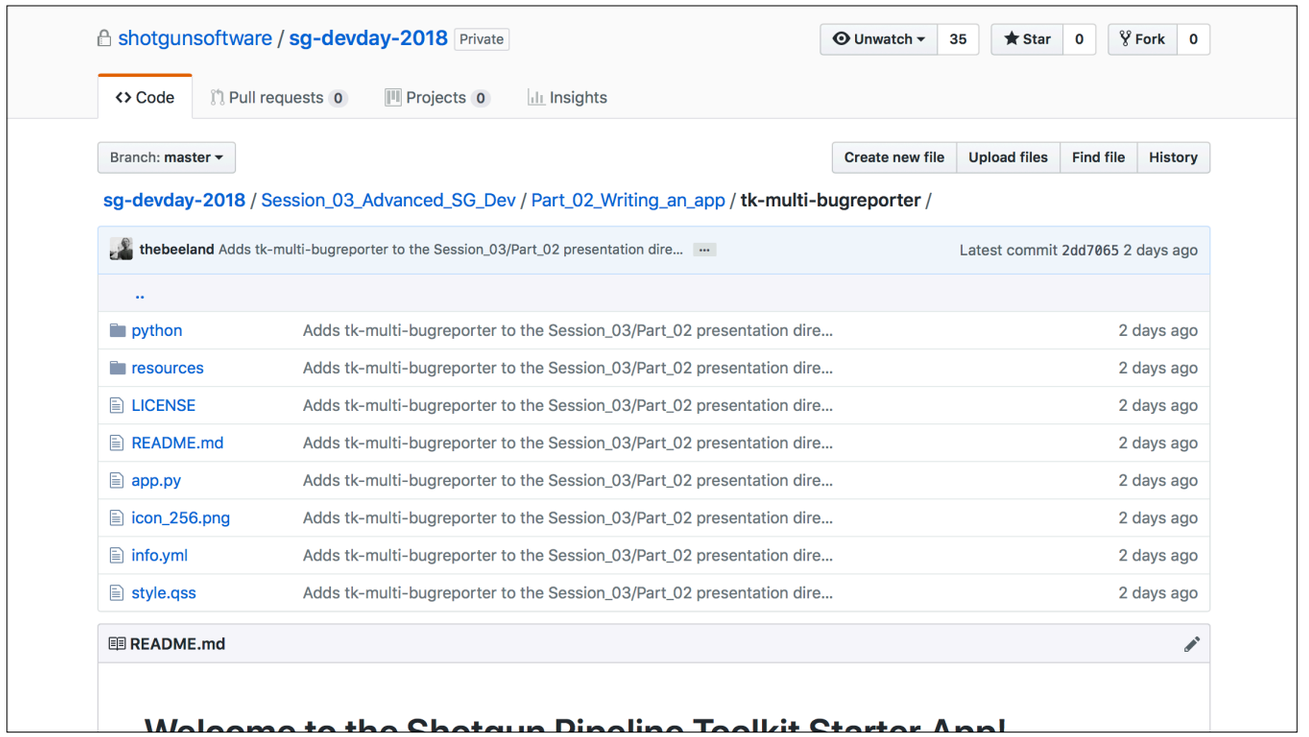
AUTODESK



There’s a story to tell, which is that an artist has found something wrong while they’re working:
*** They report the issue using the Bug Reporter app**
*** Mister Support was on the CC list, so they get an email notification**
*** Mister Support goes to Shotgun and assigns the ticket to himself, and gives it a quick reply**

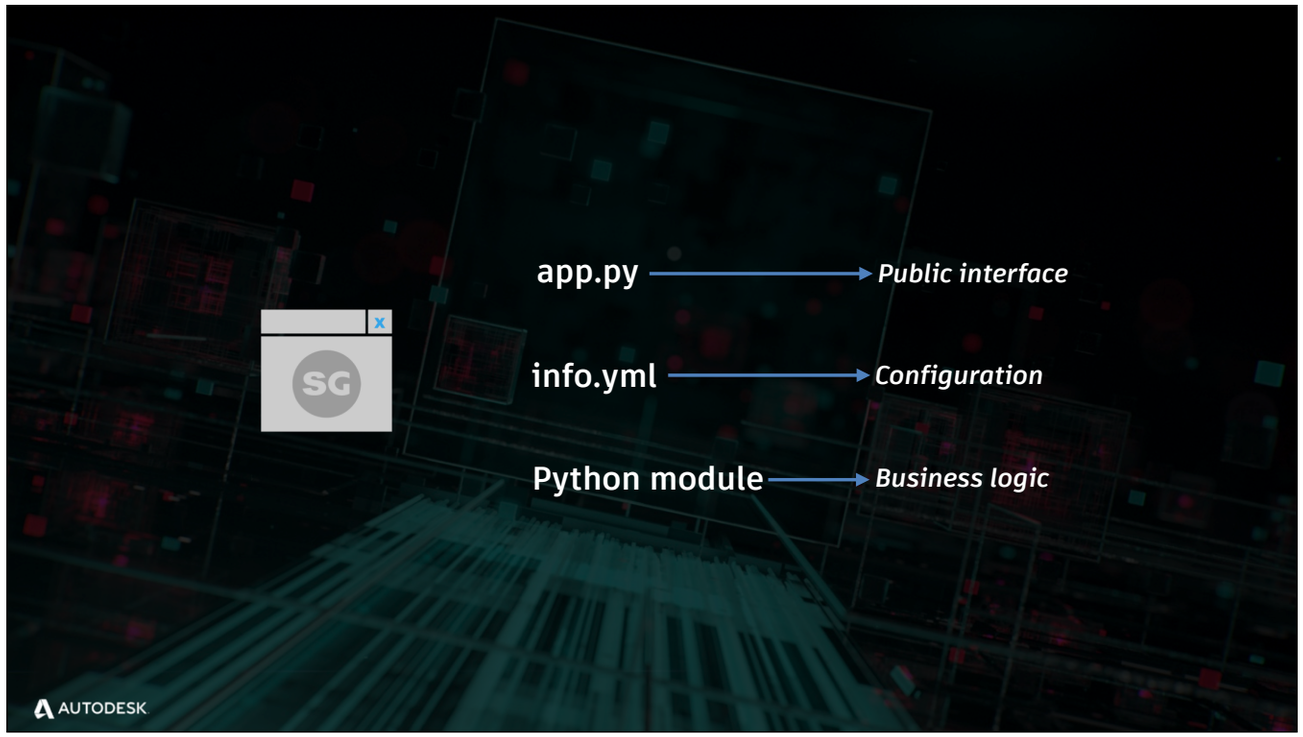


We’re going to break this down into three chunks, as shown here



*** All of the code for the bug reporter app is available in the sg-devday-2018 repo in github!**


SG
The structure of a Toolkit App
AUTODESK



There are three primary components to an App that we're going to look at:

* The app.py contains the public interface (from a code standpoint) and fulfills the contract that any Toolkit App is responsible for
* The info.yml file contains a description of the configuration of the App
* The python module within the App bundle contains the "business logic", where user interface and features are coded

Contents of an **app.py**

```python
from sgtk.platform import Application

class BugReporter(Application):
    """
    An App that can be used to submit a bug ticket to support team in Shotgun.
    """
    def init_app(self):
        """
        Called as the application is being initialized
        """
        # first, we use the special import_module command to access the app module
        # that resides inside the python folder in the app. This is where the actual UI
        # and business logic of the app is kept. By using the import_module command,
        # toolkit's code reload mechanism will work properly.
        app_payload = self.import_module("app")

        # now register a *command*, which is normally a menu entry of some kind on a Shotgun
        # menu (but it depends on the engine). The engine will manage this command and
        # whenever the user requests the command, it will call out to the callback.

        # first, set up our callback, calling out to a method inside the app module contained
        # in the python folder of the app
        menu_callback = lambda : app_payload.dialog.show_dialog(self)

        # now register the command with the engine
        self.engine.register_command("Report Bugs!", menu_callback)
```

AUTODESK

The bare minimum:
* **A class subclassing sgtk.platform.Application**
* **An init_app method**

Contents of an **info.yml**

```yaml
# expected fields in the configuration file for this engine
configuration:
  cc:
    type: str
    default_value: ""
    description: A comma-separated list of Shotgun user names to CC by default.
```

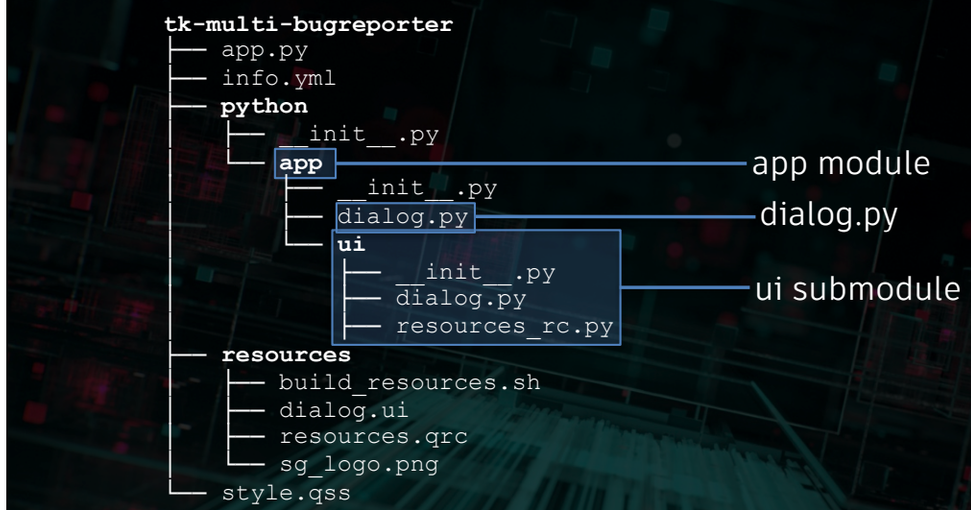
```yaml
# More verbose description of this item
display_name: "Shotgun Toolkit Bug Reporter"
description: "An App that can be used to quickly submit a new bug ticket to the support team."
```

```yaml
# the frameworks required to run this app
frameworks:
  - {"name": "tk-framework-qtwidgets", "version": "v2.x.x", "minimum_version": "v2.7.0"}
```

AUTODESK

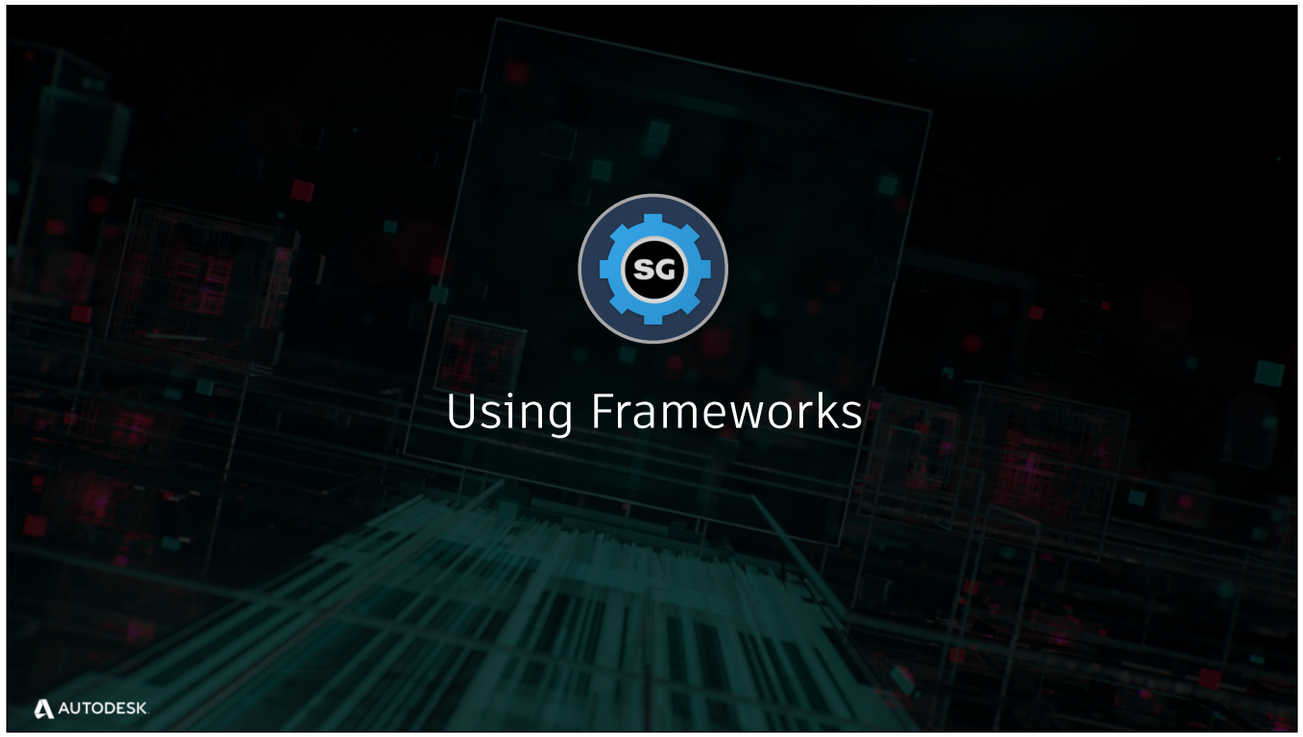
* Configuration: contains the definition of any config settings for the app
* Name/Desc: Pretty name and description for users
* Frameworks: The app's required frameworks (more later)

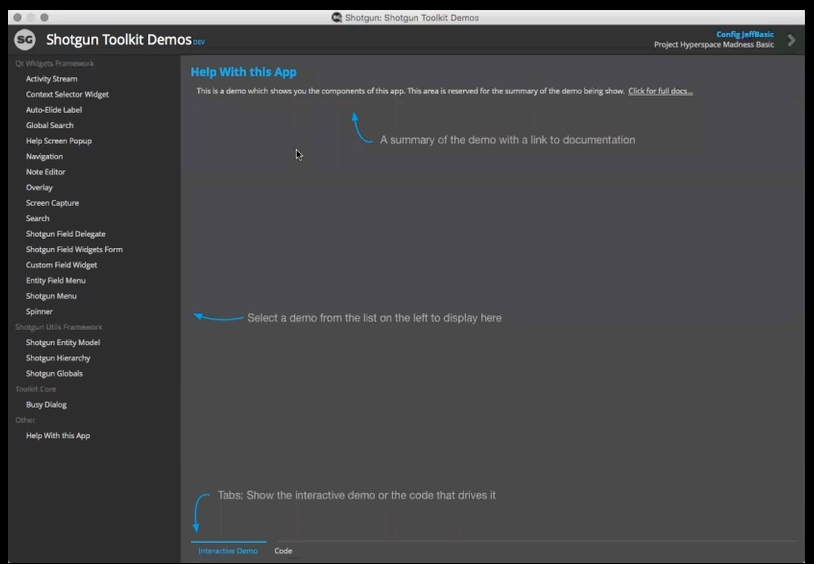
Contents of the **Python module**



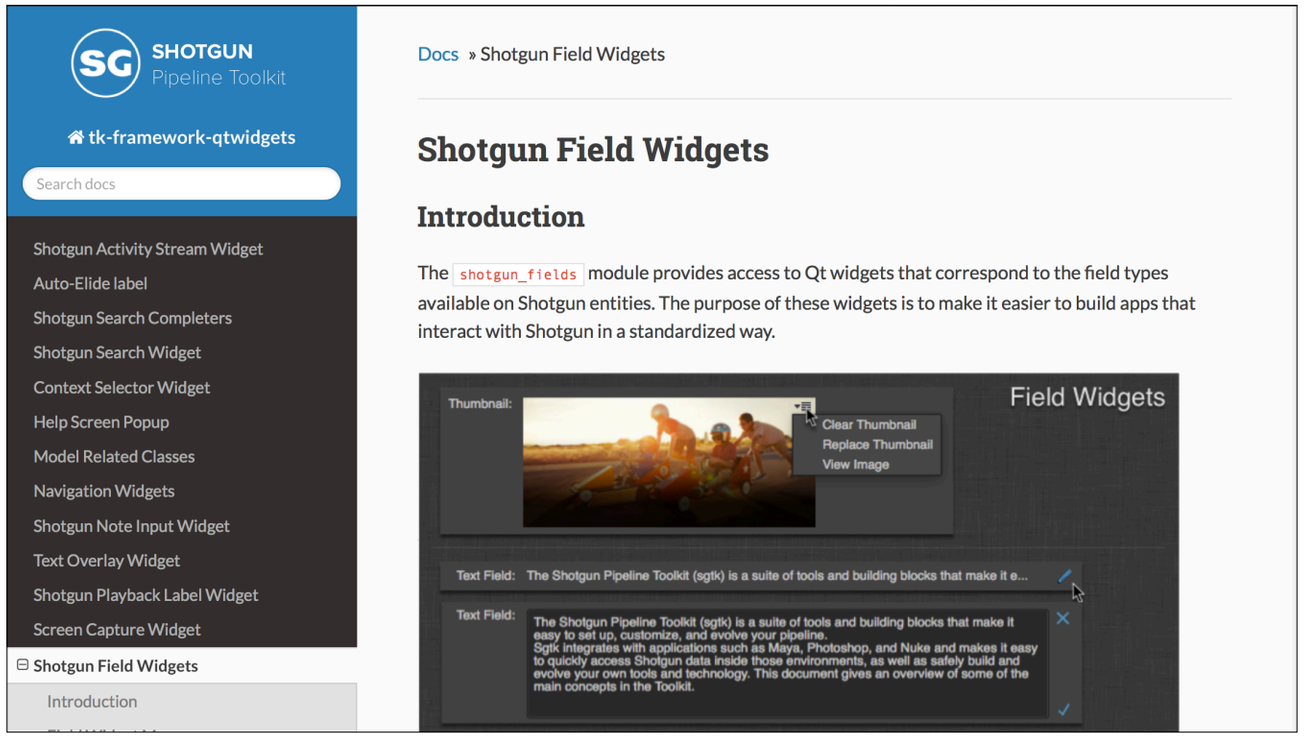
AUTODESK

* App module: what's imported using import_module
* UI: submodule that contains code generated by build_resources.sh
* Dialog.py: The business logic of the UI


SG
Using Frameworks
AUTODESK



* We provide a BUNCH of stuff out of the box, and the demo app is a good place to start
* We're going to be focusing the qtwidgets framework for this app



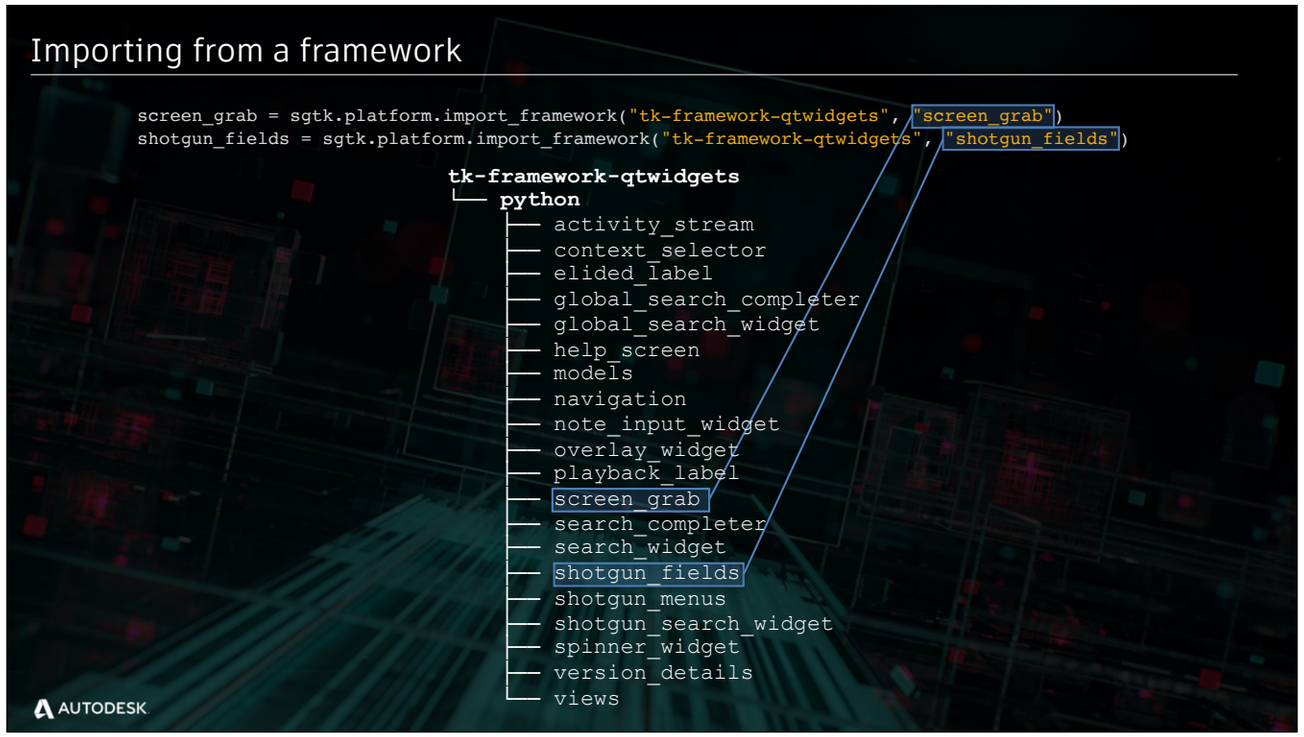
More info in the above links that will explain more about what we offer in the QtWidgets framework
*** The demo app was what was just shown in the previous video**
*** The repo for the qtwidgets framework is public — have a look! Contribute!**
*** The qtwidgets framework also has a lovely developer page with lots of documentation**

Required frameworks in **info.yml**

```yaml
# expected fields in the configuration file for this engine
configuration:
  cc:
    type: str
    default_value: ""
    description: A comma-separated list of Shotgun user names to CC by default.

# More verbose description of this item
display_name: "Shotgun Toolkit Bug Reporter"
description: "An App that can be used to quickly submit a new bug ticket to the support team."

# the frameworks required to run this app
frameworks:
  - {"name": "tk-framework-qtwidgets", "version": "v2.x.x", "minimum_version": "v2.7.0"}
```

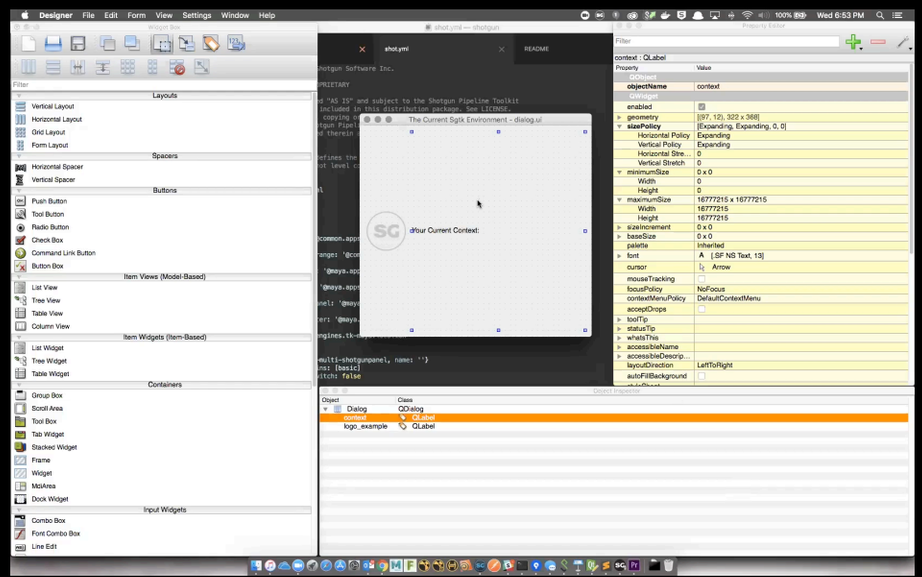
AUTODESK

* For the app's business logic to have access to a framework, it must be included in the frameworks structure in info.yml
  * The name of the framework
  * The required version: we allow for wildcards for non-major version components
  * Optionally a bare minimum version requirement

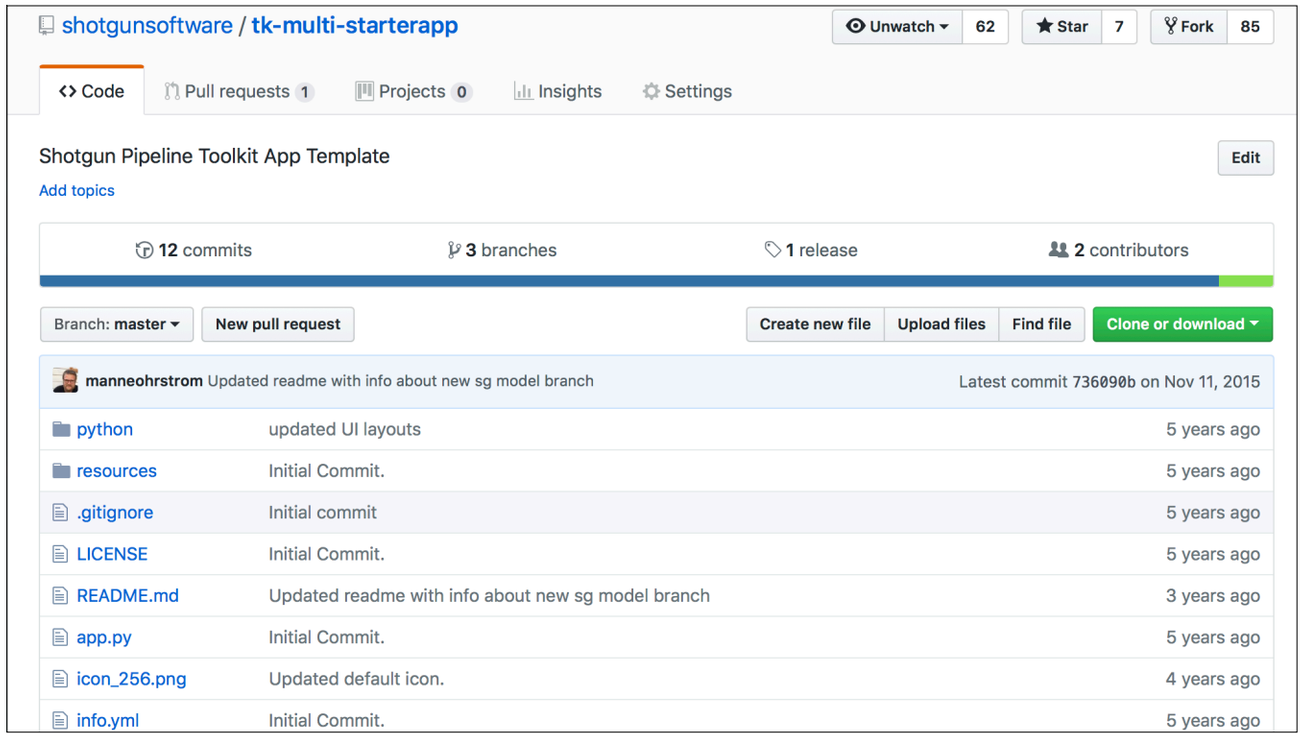


When importing a submodule from a framework, reference it by the submodule name you see in the framework's structure, then reference individual components in those submodules as any other python module/class reference

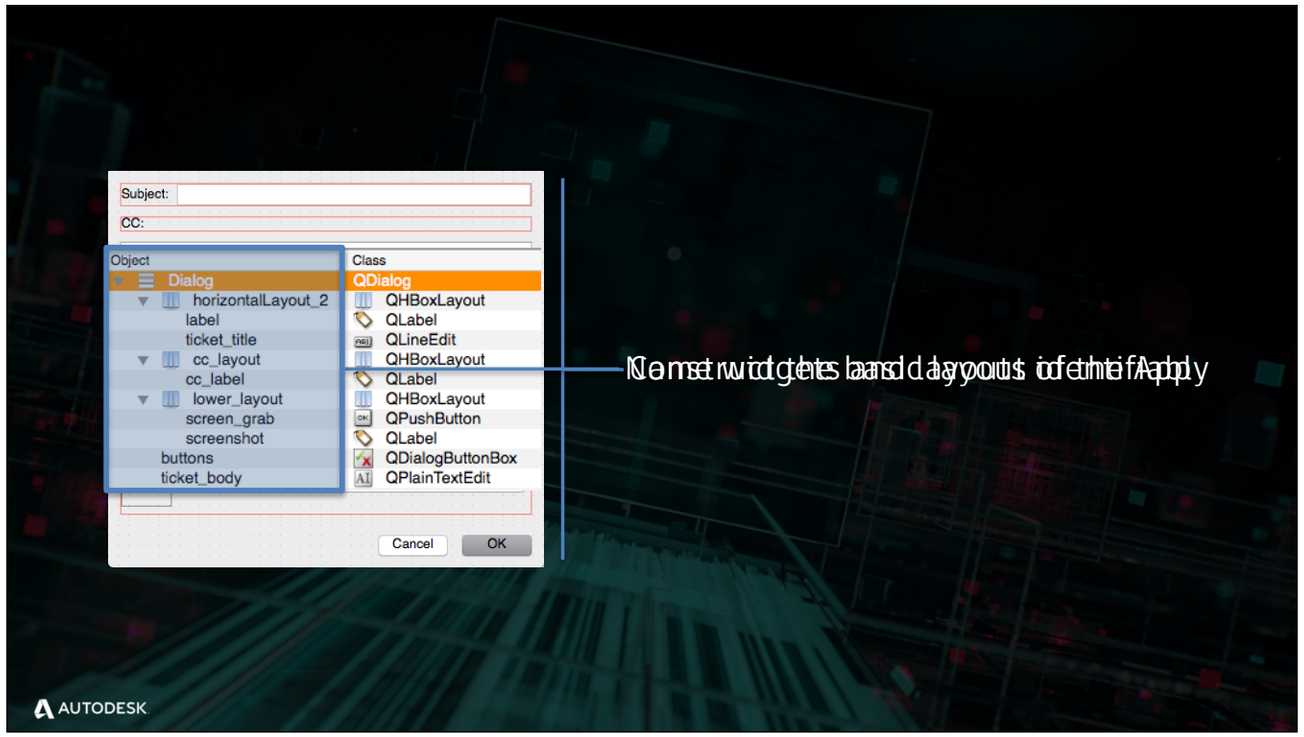

>>>
.py
Developing and Configuring
the App
AUTODESK



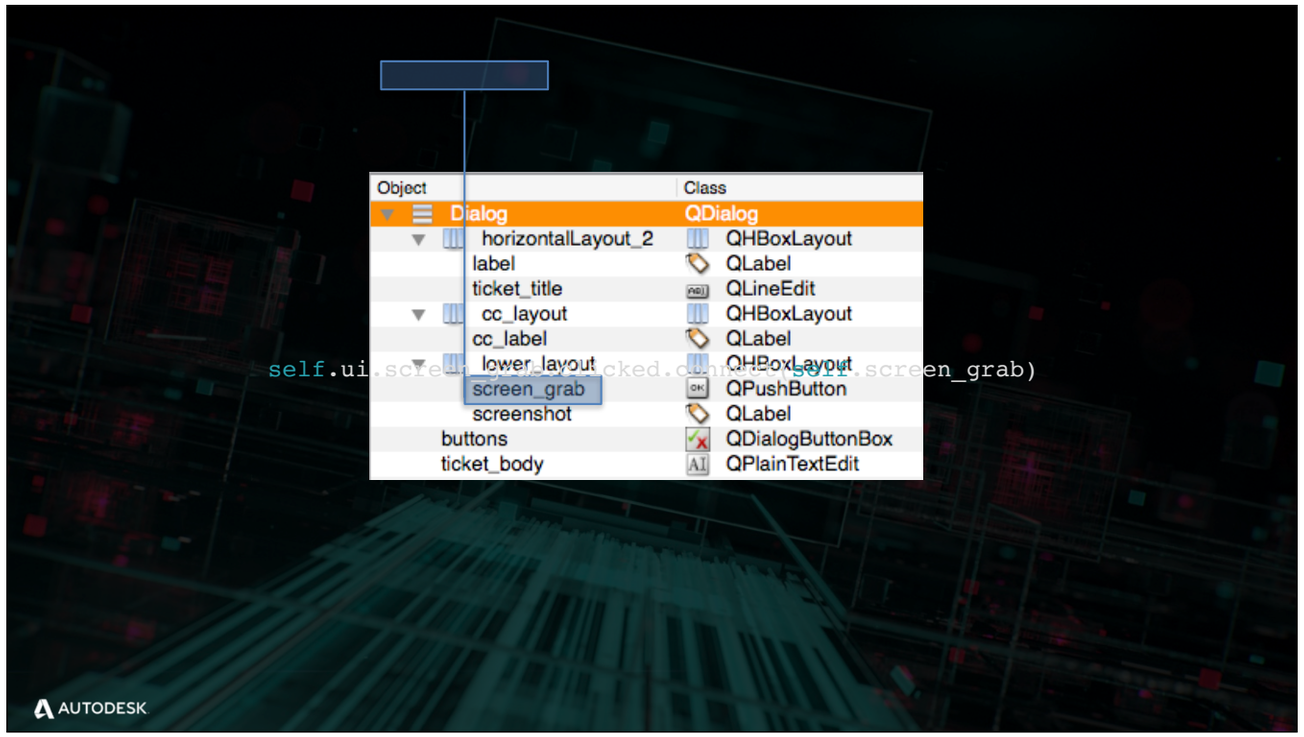
* We started with tk-multi-starterapp
  * It gives us a very basic .ui file that can be used in Qt Designer as a starting place
  * It provides the basic scaffolding required of a Toolkit App, which can be extended to do what you want your app to do



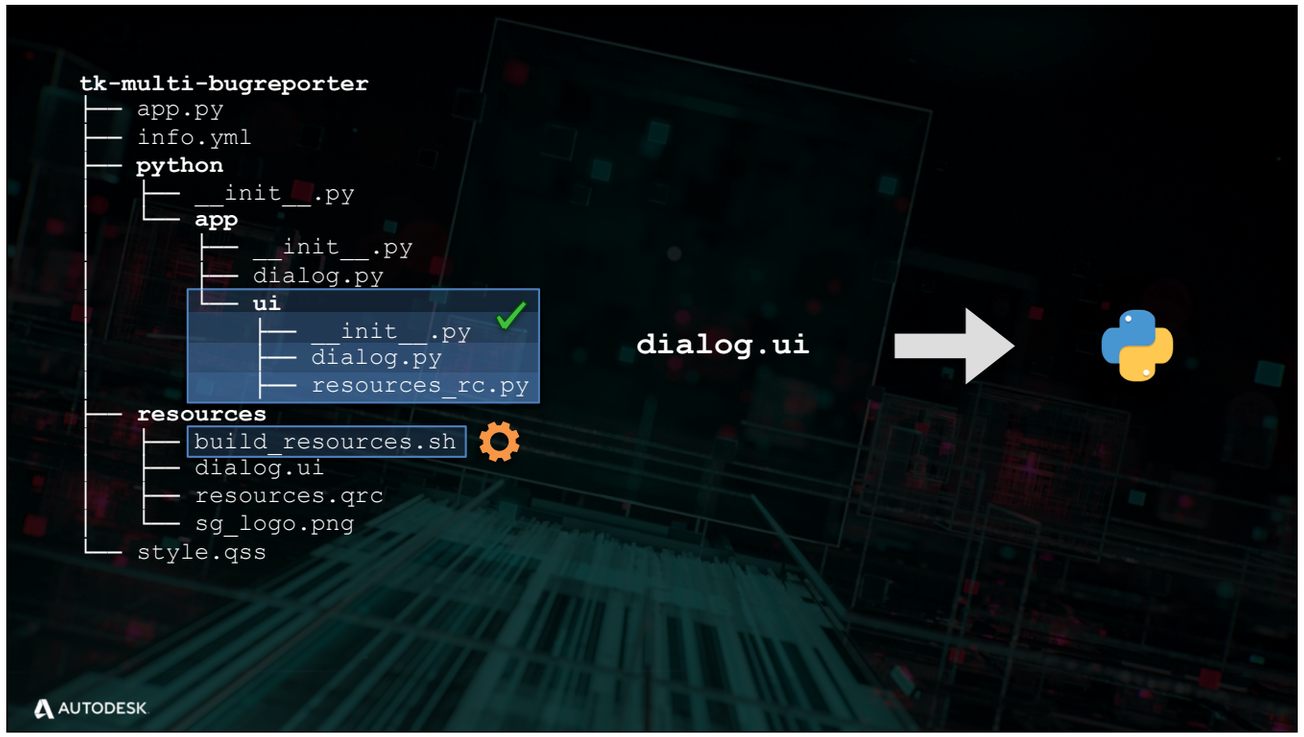
*** A good place to start is by using the starter app that we provide in the GitHub repo here**



* We won't be going into detail about how to use Qt Designer
  * It's not required: if you prefer to code all of your UI layout, then go for it, but we find Designer to be a faster workflow
  * Qt Designer is bundled with Shotgun Desktop, so if you look in the install location for that, you can find what you need without installing any additional software
* Constructing the layout is largely a drag-and-drop affair
* Naming the widgets/layouts you add makes them easy to reference from code later — this is KEY, as we'll discuss further as we continue



Accessing the widgets created in Qt Designer is done via the ui submodule and by the name specified in Designer.



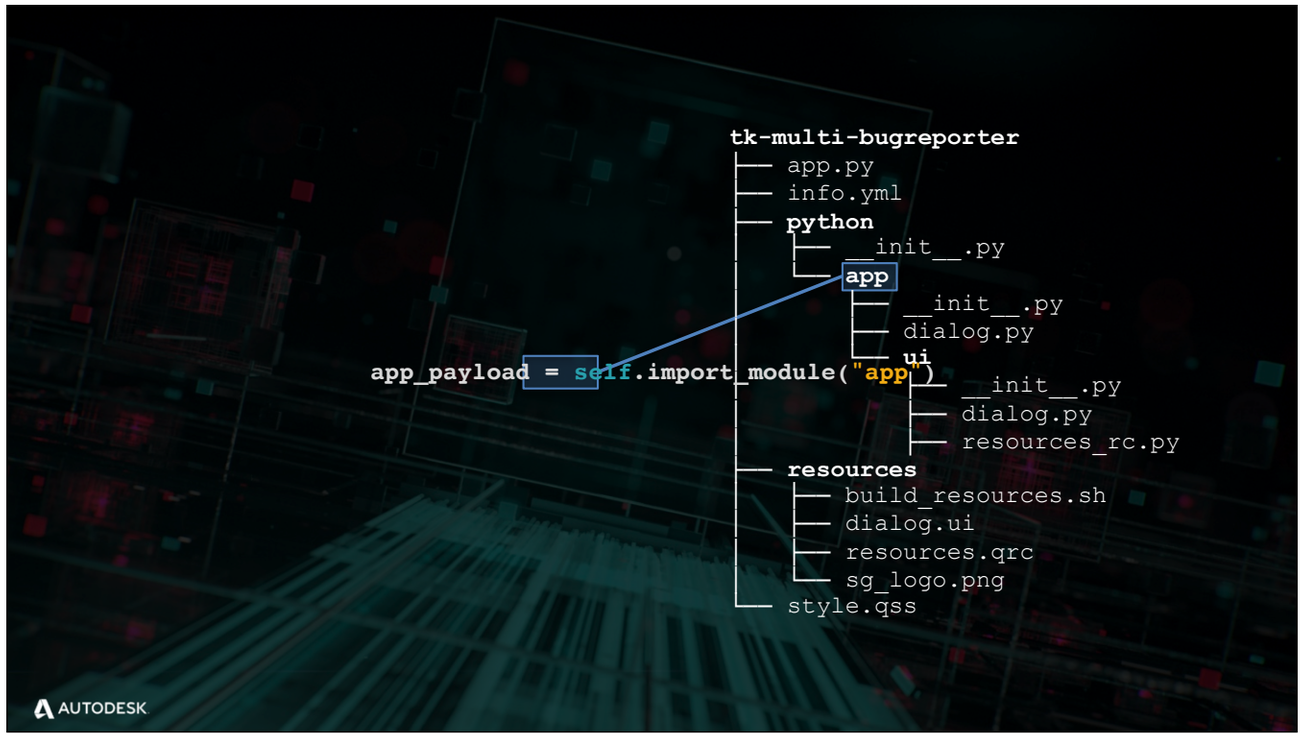
* build_resources.sh converts the work you did in Qt Designer into Python
* That python that is generated goes into the app's ui submodule

What's happening in **app.py**?

```python
from sgtk.platform import Application

class BugReporter(Application):
    """
    An App that can be used to submit a bug ticket to support team in Shotgun.
    """
    def init_app(self):
        """
        Called as the application is being initialized
        """
        # first, we use the special import_module command to access the app module
        # that resides inside the python folder in the app. This is where the actual UI
        # and business logic of the app is kept. By using the import_module command,
        # toolkit's code reload mechanism will work properly.
        app_payload = self.import_module("app")

        # now register a *command*, which is normally a menu entry of some kind on a Shotgun
        # menu (but it depends on the engine). The engine will manage this command and
        # whenever the user requests the command, it will call out to the callback.

        # first, set up our callback, calling out to a method inside the app module contained
        # in the python folder of the app
        menu_callback = lambda : app_payload.dialog.show_dialog(self)

        # now register the command with the engine
        self.engine.register_command("Report Bugs!", menu_callback)
```

AUTODESK

* import_module will import the named modules from the python area in the app's bundle. The returned value
* Build the menu callback and register it as a command -> sgtk builds menu items in any integration based on these registered commands



Importing the bundle’s app module corresponds to importing the “app”module under python in the bundle

*** the lambda builds a callable referencing the show_dialog function defined in the “dialog” submodule, as shown**

What's happening in **dialog.py**?

```
# by importing QT from sgtk rather than directly, we ensure that
# the code will be compatible with both PySide and PyQt.
from sgtk.platform.qt import QtCore, QtGui
from .ui.dialog import Ui_Dialog

screen_grab = sgtk.platform.import_framework("tk-framework-qtwidgets", "screen_grab")
shotgun_fields = sgtk.platform.import_framework("tk-framework-qtwidgets", "shotgun_fields")

def show_dialog(app_instance):
    """
    Shows the main dialog window.
    """
    # in order to handle UIs seamlessly, each toolkit engine has methods for launching
    # different types of windows. By using these methods, your windows will be correctly
    # decorated and handled in a consistent fashion by the system.

    # we pass the dialog class to this method and leave the actual construction
    # to be carried out by toolkit.
    app_instance.engine.show_dialog("Report Bugs!", app_instance, AppDialog)
```

AUTODESK

* this is the business logic of the UI
* Import Qt stuff from sgtk.platform.qt because of the shim
* Import the auto-gen dialog from ui for future use (more to come)
* Import frameworks (already explained earlier)
* show_dialog implementation, which makes use of AppDialog (more to come)

What's happening in **AppDialog**?

```
class AppDialog(QtGui.QWidget):
    """
    Main application dialog window
    """
    def __init__(self):
        """
        Constructor
        """

    def _get_shotgun_fields(self):
        """
        Populates the CC list Shotgun field widget.
        """

    def screen_grab(self):
        """
        Triggers a screen grab to be initiated.
        """

    def create_ticket(self):
        """
        Creates a new Ticket entity in Shotgun from the contents of the dialog.
        """
```

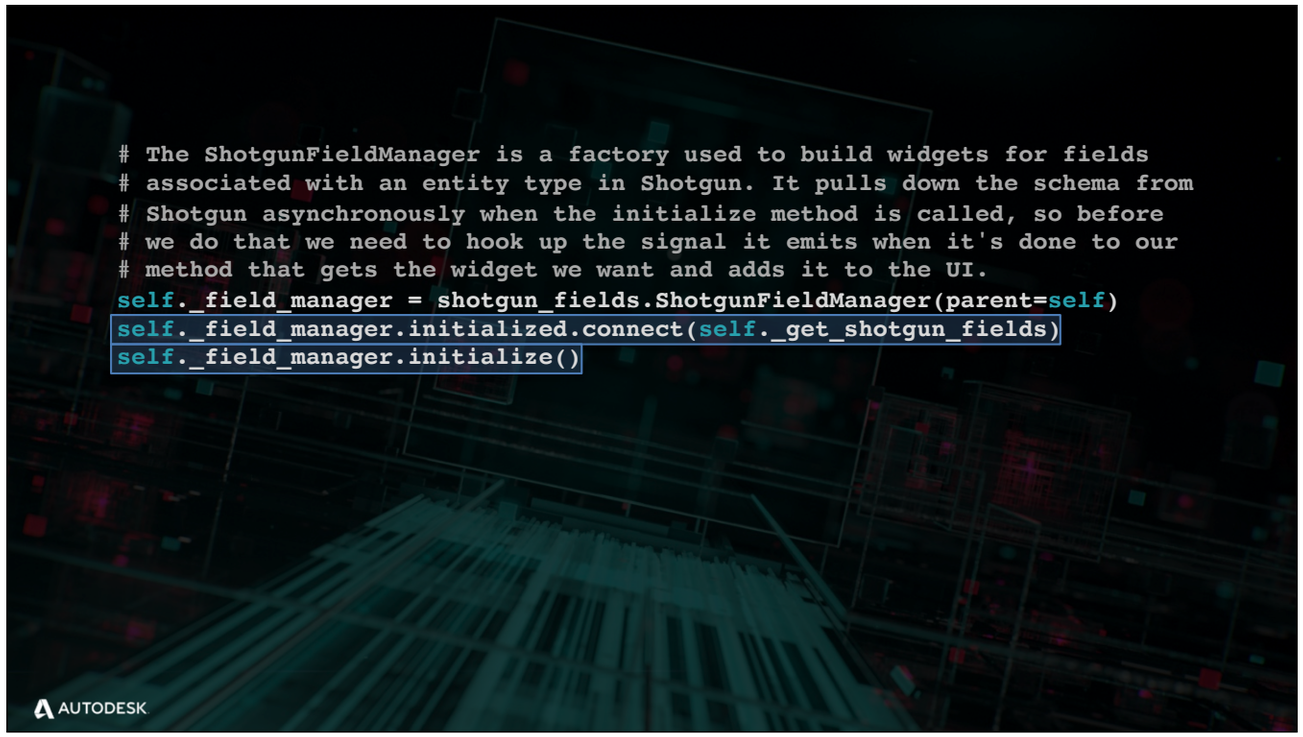
AUTODESK

* It's a QWidget!
* four methods, we'll go through each one by one

```python
def __init__(self):
    """
    Constructor
    """
    QtGui.QWidget.__init__(self)

    # now load in the UI that was created in the UI designer
    self.ui = Ui_Dialog()
    self.ui.setupUi(self)

    self._app = sgtk.platform.current_bundle()

    self.ui.buttons.accepted.connect(self.create_ticket)
    self.ui.buttons.rejected.connect(self.close)
    self.ui.screen_grab.clicked.connect(self.screen_grab)

    self._screenshot = None
    self._cc_widget = None

    # The ShotgunFieldManager is a factory used to build widgets for fields
    # associated with an entity type in Shotgun. It pulls down the schema from
    # Shotgun asynchronously when the initialize method is called, so before
    # we do that we need to hook up the signal it emits when it's done to our
    # method that gets the widget we want and adds it to the UI.
    self._field_manager = shotgun_fields.ShotgunFieldManager(parent=self)
    self._field_manager.initialized.connect(self._get_shotgun_fields)
    self._field_manager.initialize()
```

AUTODESK

* we're using the auto-gen dialog from Designer and calling its setupUi method to populate everything
* We can then reference any of the widgets/layouts we created in Designed via self.ui
* The fields manager will require some explanation, so we'll look at it on its own now

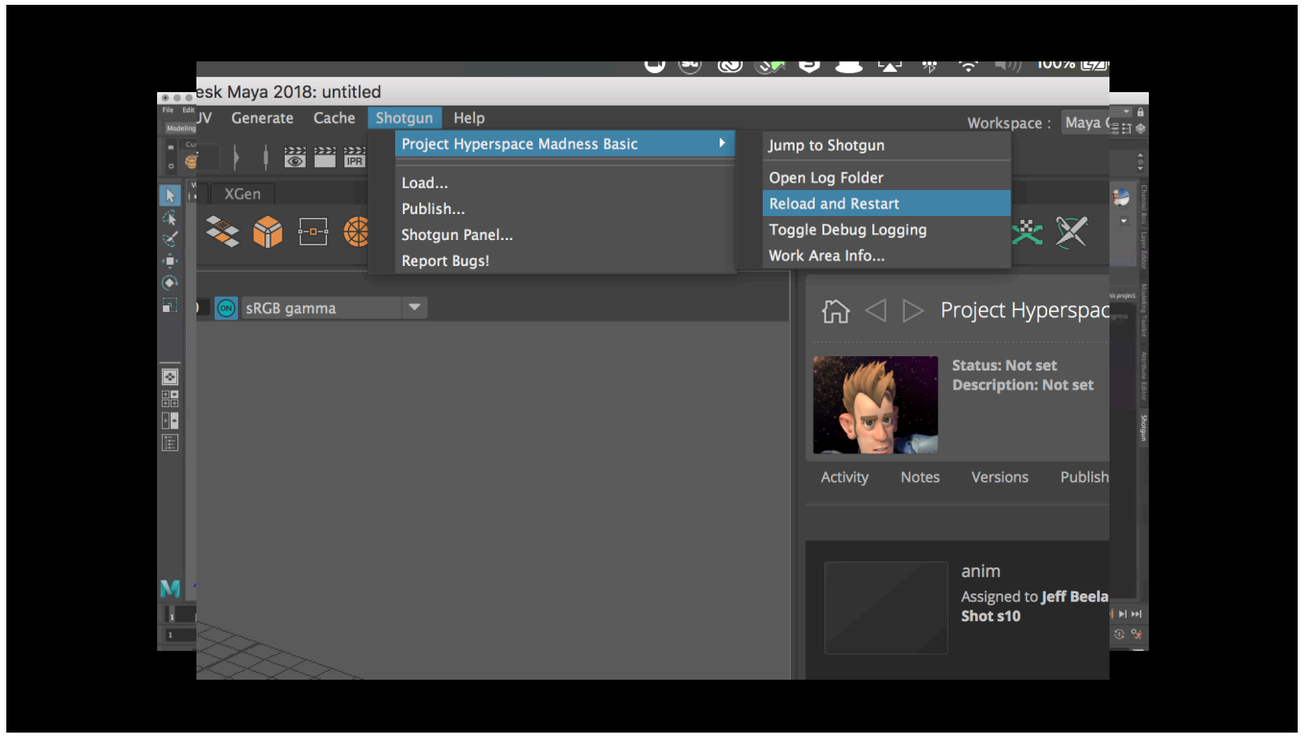



```
# The ShotgunFieldManager is a factory used to build widgets for fields
# associated with an entity type in Shotgun. It pulls down the schema from
# Shotgun asynchronously when the initialize method is called, so before
# we do that we need to hook up the signal it emits when it's done to our
# method that gets the widget we want and adds it to the UI.
self._field_manager = shotgun_fields.ShotgunFieldManager(parent=self)
self._field_manager.initialized.connect(self._get_shotgun_fields)
self._field_manager.initialize()
```

* the initialize call will trigger loading the schema from Shotgun, which can take some time if the data isn’t already cached
* When it’s done, it emits the initialized signal, which we’ll use to trigger some logic (more to come)

* Accessing our “cc” setting from the config to get the default set of users to CC from SG
* Creating the field widget for the “addressings_cc” field that is on the Ticket entity in SG
* Setting the value of that widget to the list of users we want to CC by default
* Add the cc widget to the layout we previously built in Designer where we wanted it to go

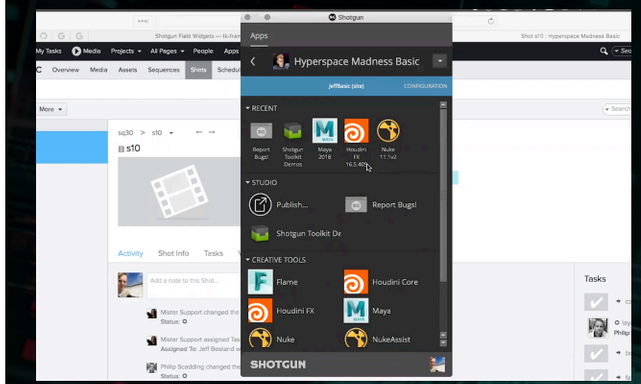
* This one is simple: we call the screen_capture static method from ScreenGrabber, which gives us a pixmap object
* We keep track of the pixmap because we'll need it later to create a png we can upload to Shotgun
* We set our screenshot QLabel that we created in Qt Designer as the label's pixmap at the correct resolution



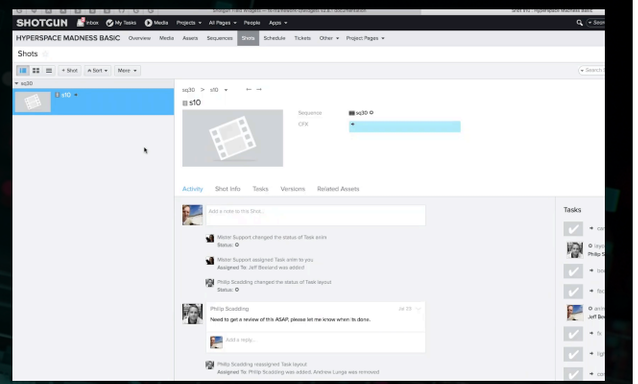
* We'll take a moment to step away from explaining code and look at the development workflow
* An important and super-useful aspect of developing apps and integrations with Toolkit is the reload and restart menu action.
* We try our app without the screen_grab slot hooked up to the button and it doesn't do anything, as expected
* We make the code change, save the file, and use the reload and restart menu action in Maya — now the button triggers a screen grab!
* This highlights how quickly a developer can iterate when writing an app or modifying existing code

* NOTE: Comments have been removed here to conserve screen space, but we'll be going over each chunk of logic here now
* First: create the new Ticket entity:
  * We get the project entity from the current context
  * Title and description from the widgets we added for those in Qt Designer
  * The cc widget we got from the shotgun fields manager from the qtwidgets framework will give us back a value that's a list of entities that we can use directly, no modification required!
* If we had a screenshot taken, then we need to write that out as a png to a temp file and then upload it as an attachment to our Ticket

Launch from SG Desktop!



Launch from Shotgun!



AUTODESK

Potential **additional features**

* Gather information about the DCC session and include it in the Ticket:
  * Which DCC is being used, and which version?
  * Collect the Toolkit log file and upload it as a Ticket attachment?
  * Include the user's current context in the Ticket body?
* Allow for multiple screenshots
* Support for arbitrary file attachments
* User-settable priority/severity
* Progress bar when creating the new Ticket and uploading attachments


Q&A
AUTODESK

https://github.com/shotgunsoftware/sg-devday-2018

AUTODESK


AUTODESK