# Administering and Configuring Sgtk

manne Öhrström
posted this on May 31, 2013 19:48



## An introduction to how to administer and configure Toolkit

This document covers some of the more detailed aspects of configuration and administration. It explains how to update and install apps, how the environments work and how to test new versions of apps and the Core API in a sandbox environment prior to production install.

Table of Contents:


- Introduction
- Using the tank command
  - Running in debug mode
  - Running tank commands via the Tookit API
  - Useful tank commands
    - setup_project
    - core
    - configurations
    - updates
    - install_app, install_engine
    - app_info
    - switch_app
    - push_configuration
    - folders, preview_folders
    - shell
  - Advanced tank commands
- The Toolkit Python API
- Pipeline Configurations and Sandboxes
  - Running the tank command for a sandbox
  - Using the Core API from a sandbox
  - Accessing a sandbox from Shotgun
  - Localizing a Pipeline Configuration
  - Deleting a cloned configuration
- Checking for Updates
  - Creating a staging sandbox
- Configuring how Apps are launched
- Creating folders on disk with Sgtk
  - Deferred Creation and User Sandboxes
- Configuring Templates
  - @include syntax in the template file

Folder creation and templates
Hooks
App level hooks
Core level hooks
Studio level hooks
Project name hook
Shotgun connection hook
Configuring Apps and Engines
Each App has a Location setting
Including files
Configuring Template settings (file paths)
Using Hooks to customize App Behaviour

# Introduction

Welcome to the Shotgun Toolkit Admin guide! This document explains how to work with the Shotgun Pipeline Toolkit from an administrative point of view - installing and updating apps, setting up new projects, and managing your studio configuration. The Toolkit is still relatively technical, so we imagine that the person handling the configuration and administration is a sysadmin, pipeline/tools developer or TD. Before we start getting into details, we would recommend that you read through the following document if you haven't already! It covers a number of the basic concepts and gives a brief introduction to configuration management and updates:

 > An Introduction to the basic Concepts in Shotgun Pipeline Toolkit.

If you are not set up with the Shotgun Toolkit yet, you may want to read through the steps how to set up and configure your very first test setup:

 > First Steps after you have installed Shotgun Pipeline Toolkit.

# Using the tank command

When managing the Shotgun Toolkit, and using the Toolkit in general, the `tank` command is your friend! The `tank` command lets you run both administrative commands and actual Apps quickly and easily from a command shell. When the Toolkit is installed for the first time, a *studio level* tank command is installed as part of the Core API install. We recommend that this is added to your `PATH` so that you can reach it from anywhere. The studio level tank command can be used to reach all different projects and execute command and run apps associated with them.

In addition to the studio level tank command, each pipeline configuration has its own tank command. Running this tank command will give you access to the methods that are associated with that project only.

You can use the tank command in many different ways. The basic idea is that you first tell the tank command *where* you want to operate and then *what* you want to do. If you don't tell it *what* you want to do, it will display a list of all the available commands. If you don't tell it *where* you want to operate, it will try to use your current directory. You can also use the tank command to list things in Shotgun.

Click to see a basic overview of how you can use the `tank` command

## Running in debug mode

Sometimes it can be useful to see what is going on under the hood. You can pass a `--debug` flag to the `tank` command which will enable verbose output and timings, sometimes making it easier to track down problems or understand why something isn't doing what you expected it to.

## Running tank commands via the Tookit API

Most Tank commands are also fully supported to run via the API. This makes it easy to perform toolkit related maintenance operations as part of more extensive scripted workflows. For more information how to

do this, see the Core API Reference:

 > Toolkit Core API Reference.

## Useful tank commands

Here's a brief list of useful tank commands that you can use when you are administering the Shotgun Toolkit.

### setup_project

Set up a new project with the Shotgun Toolkit. This is where you start when you have a project in Shotgun and you want to extend this to use the Toolkit. The command will guide you through the process and ask you for various pieces of information, such as which configuration to use and which project to set up.

Re-run setup_project on the Same Project

Toolkit provides you with a project name suggestion as part of the project setup process. If you are happy with the name, you can just press Enter to continue or alternatively type in another name by hand.

If you have a special studio naming convention for projects, it is also possible to control the default value that the setup project process suggests. This is done via an advanced studio level hook - read more about it in the studio level hooks section further down in this document.

### core

Checks for Core API Updates. This will connect to the App Store and see if there is a more recent version of the Core API available. If there is, the command will ask you if you want to update.

Click to see example output from this command

### configurations

Gives an overview of all the configurations for a project. This can be handy when you want to get a breakdown of the current activity for a project.

Click to see example output from this command

### updates

This command will go through all the environments associated with the project and check if there are more recent versions of apps or engines available. Depending on how the apps and engines have been installed, this updater may check against a local git repository, Github or the Shotgun Toolkit App Store. If a more recent version is detected, you will get asked if you want to update your setup. If any new configuration parameters have been introduced in the new version of the App, you may be prompted for values.

General syntax:

```
> tank updates [environment_name] [engine_name] [app_name]
```

The special keyword ALL can be used to denote all items in a category. Examples:

- Check everything: `tank updates`
- Check the Shot environment: `tank updates Shot`
- Check all maya apps in all environments: `tank updates ALL tk-maya`
- Check all maya apps in the Shot environment: `tank updates Shot tk-maya`
- Make sure the loader app is up to date everywhere: `tank updates ALL ALL tk-multi-loader`
- Make sure the loader app is up to date in maya: `tank updates ALL tk-maya tk-multi-loader`

### install_app, install_engine

Installs a new engine or App in one of the environments associated with the project. You can use this command to install something either from the Toolkit app store or from git.

### app_info

See a breakdown of all the apps and their key configuration settings

### switch_app

Switch the location for an app between git, app store and local paths. This is very handy when you are doing development.

### push_configuration

Push the configuration contained in the current pipeline configuration to another pipeline configuration.

### folders, preview_folders

Creates folders on disk for an item.

Click to see example output from this command

### shell

Starts an interactive python shell in your currently selected location. Sets up handy referenecs to pre-initialized APIs, context and engine objects.

Click to see example output from this command

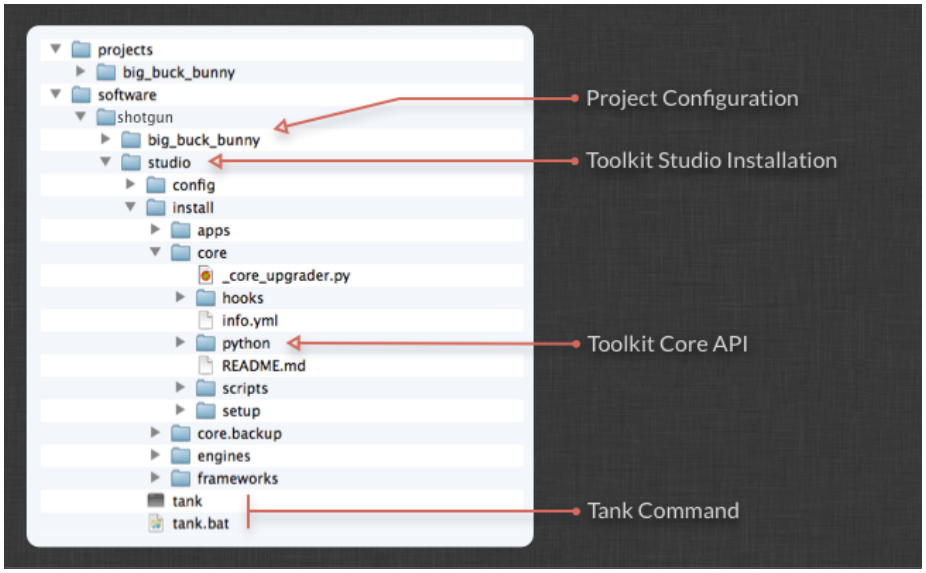
## Advanced tank commands

Here's a brief list of advanced tank commands that you can use when you are administering the Shotgun Toolkit.

- `tank Project XYZ move_configuration` - This command will move a pipeline configuration from one location on disk to another. This is useful if you want to change the structure on disk for your project.
- `tank move_studio_install` - This command will move a studio install from one location on disk to another. This is useful if you want to change the structure on disk for your project.
- `tank Project XYZ localize` - This command will download the Core API to a particular pipeline configuration location. This is useful if you want to test a new version of the Core API inside a Toolkit staging area. This process is described in detail further down in this document.
- `tank Project XYZ clear_cache` - Clears some of the Toolkit's caches. This can be useful if menu items aren't showing up inside Shotgun for some reason.

# The Toolkit Python API

If you are launching applications using the tank command or using Shotgun, the Toolkit will automatically get added to the `PYTHONPATH` and initialize. Sometimes it is useful to manually source and run the Toolkit API. This can be done by adding the Toolkit Core API to the pythonpath and then importing it.




In the example above, we have installed sgtk into `/mnt/software/sgtk/studio` and the Sgtk Core API is

located in `/mnt/software/sgtk/studio/install/core/python`. Once added to the `PYTHONPATH`, you can import it and initialize it:

```
import sgtk

# create a Sgtk API object for a shotgun entity
tk = sgtk.sgtk_from_entity("Shot", 123)

# Create a Sgtk API object based on a file system location
tk = sgtk.sgtk_from_path("/mnt/projects/hero/assets/chair")
```

# Pipeline Configurations and Sandboxes

A Pipeline Configuration has the same basic structure as the studio installation. Most importantly, it contains a both a `tank` command and a Toolkit API inside of `install/core/python`. The reason the API and the tank command is duplicated across each configuration is to make it easy to do development and to do work outside of the production configuration.

When a new project is set up in the Shotgun Toolkit, the project setup creates a *Primary Pipeline Configuration* for the project. This is always called 'Primary' and represents the main configuration for the project. You can see the configuration represented inside of Shotgun in the form of a Pipeline Configuration entity for the project. When you are doing development or are making changes to the configuration, you typically don't work in the Primary configuration - if you accidentally break something, everyone working on the project will be affected. Instead, you can *clone* the configuration inside of Shotgun. You now have your own parallel configuration where you can make changes without anyone else being affected.

## Running the tank command for a sandbox

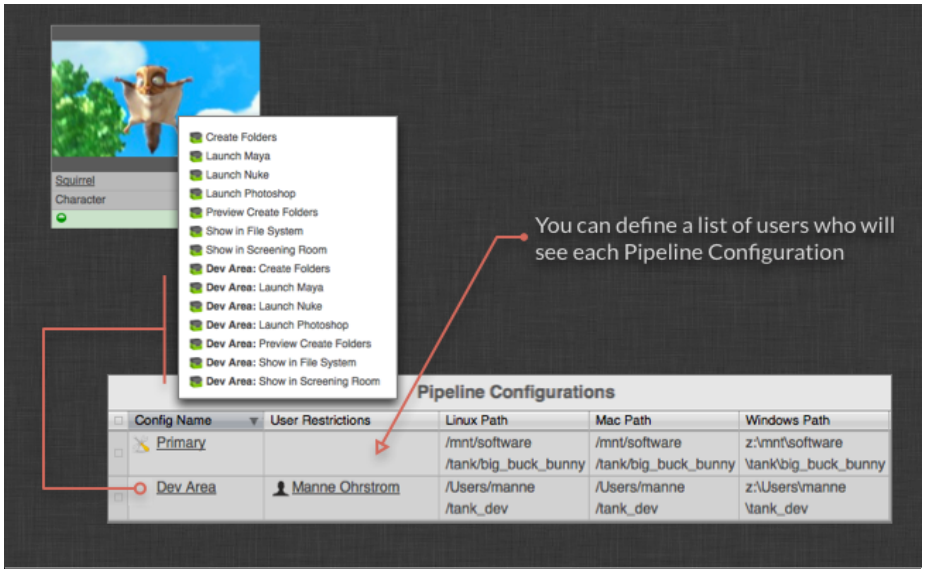
If you use the studio level `tank` command to run an app, it will always use the Primary configuration for a project. So if you just type in `tank Shot ABC123 launch_maya`, the Shotgun Toolkit will find the project that shot ABC123 belongs to, find its primary pipeline configuration and use those configuration settings when it launches maya. If you instead want to use the experimental configuration in your development sandbox you instead use the specific tank command inside the dev sandbox to launch maya: `~/sgtk_dev_sandbox/tank Shot ABC123 launch_maya`. The Toolkit will now use the config in the dev sandbox instead of the Primary configuration for the project.

## Using the Core API from a sandbox

Similarly, if you want run the Toolkit API from inside your pipeline configuration and not from the studio install location, you can add your dev sandbox to the `PYTHONPATH` rather than the studio python API.

## Accessing a sandbox from Shotgun

Inside Shotgun, it is a lot simpler. Each Pipeline Configuration for a Project has a list of users that can see that Configuration. By leaving the field blank, everyone will see the configuration.




When you clone a configuration (which you can do by right clicking on it in Shotgun), you will automatically

be associated with that configuration, effectively granting you exclusive visibility to the configuration. If you are developing new tools in Maya and want an artist to test them out, simply add the artist to your pipeline configuration dev sandbox and they can launch maya from your sandbox and will then have access to your work in progress tools.

### Localizing a Pipeline Configuration

By default, a Pipeline Configuration will pick up its code from the Shotgun Toolkit Studio install. The Toolkit studio installation will contain a cache of all the app and engine code that the Shotgun Toolkit is using and the Toolkit Core API installation. Each Pipeline configuration will share the Core API and the apps cache in the studio location. This is often useful because you can roll out core API updates to all projects at the same time. Simply update the studio location, and all projects will be affected.

Sometimes, however, it is useful to be able to cut off a Pipeline Configuration and make it independent. Examples when this makes sense include:

- You have a project that is about to wrap up and want to freeze updates and make sure nothing is changing.
- You have a pipeline configuration that you want to *test* a new version of the Toolkit Core API in.
- You are assembling a minimal Toolkit bundle that you will use when working from home.

This process of making a pipeline configuration completely independent from the studio location is called *localizing* the Configuration and basically means that the Core API is copied into the Pipeline Configuration. You do this by running the `tank localize` command.

Note that once you have localized a configuration, you cannot necessarily run the studio tank command anymore - the general rule is that once a project has been localized, you need to use its local tank command and Python API.

### Deleting a cloned configuration

If you don't want your cloned configuration or dev area anymore, simply delete the record and then delete the configuration from disk.

## Checking for Updates

Checking if there are any updates to apps or engines is easy! Simply run the `tank updates` command for a project. The Shotgun Toolkit will check for updates and ask you if you want to update. If there are any new parameters that don't have default values, the update script will prompt you for values. Each update presents a url link to a release notes page, in case you want to check the details of what has changed. You can exit the process at any point.

Updating the Toolkit Core API is equally easy. Just run the `tank core` command!

### Creating a staging sandbox

While it is often perfectly safe to simply run `tank updates` on your primary configuration, it is sometimes better to test things prior to rolling it out in production. In this case, you just clone the Primary pipeline configuration and run the update commands there. In the case of a Core API upgrade, make sure you *localize* the sandbox before running the `core` command (see above for more information about localize). Once you have verified that the update works, run it again on the primary configuration. For more detailed information how to do this, see the following document:


> Managing your Configuration

## Configuring how Apps are launched

A part of the Toolkit configuration that is often necessary to configure right after you have installed it is the app launch configuration. We have tried to make this flexible and configurable, knowing that this is an area where studios often already have customization and tools in place.

When you are launching an application (such as maya or nuke) either from inside of Shotgun or using the tank command, you are invoking an app that is responsible for starting up the application and initializing the Toolkit. This app is called `tk-multi-launchapp` can you can read about it here:


> The Toolkit Application Launcher

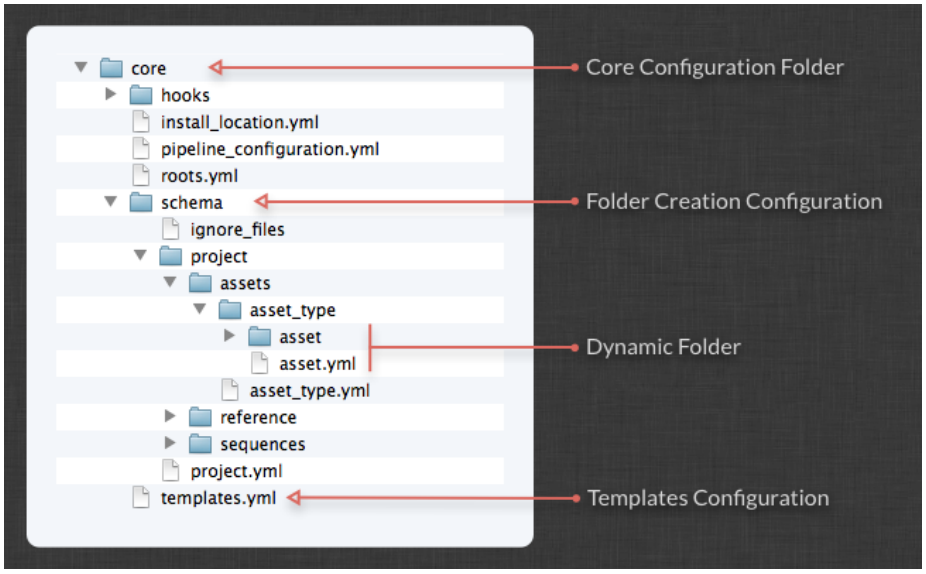
When you setup a new project with one of the Toolkit starter configurations, it will display instructions how to change the application paths that are used with the Launch App. You are not required to use this app - if you have an alternative system in place that works just as well. The Launcher application typically carries out the following steps:

1. It figures out a context to use. The context represents the current work area. If you are right clicking a task, asset or shot inside shotgun, the context is created based on this. If you are using the tank command, it will be something you specify as part of the command line or is picked up from your current directory.
2. It then launches the application based on settings in the app configuration. You can configure the launch app in several ways - the path to the application, the command line arguments to pass, the actual code that executes the application and environment variables that needs setting prior to application launch.
3. It sets the Pythonpath to ensure that the Toolkit API can be initialized later.
4. Once the application has started, the Toolkit API is imported and initialized.
5. Finally, the engine is launched.

Example: A more hands on example of how the Shotgun Toolkit starts up.

## Creating folders on disk with Sgtk

One key part of the Toolkit Configuration is the file system configuration. The Shotgun Toolkit can help create folders on disk in a consistent fashion and the creation process is driven from shotgun. Below is an overview of the core folder inside a configuration:




Setting up a folder configuration is relatively easy. The configuration is basically a template folder structure that you create, with some configuration files to indicate that a folder should be representing for example an asset or a shot. When you configure you file system structure, you can always use the *folder preview* functionality to get a listing of what will be created. This command is available both in shotgun and in the tank command.

Start by outlining your scaffold using normal folders. If you have a level of dynamic folders in your scaffold, for example representing assets, shots or pipeline steps, just ignore that for the moment. Once you are happy with the result, add dynamic functionality step by step to each dynamic folder. This is done by adding a yml file that has the same name as the folder.

Inside the yaml file, use a special syntax to define how a folder is created. The Shotgun Toolkit supports a number of different dynamic behaviours, but the normal one is a dynamic node that represents a shotgun entity. In this case, the configuration file could look something like this:

```
# the type of dynamic content
type: "shotgun_entity"

# the shotgun field to use for the folder name
name: "{code}_{sg_prefix}"

# the shotgun entity type to connect to
entity_type: "Asset"
```

```
# shotgun filters to apply when getting the list of items
# this should be a list of dicts, each dict containing
# three fields: path, relation and values
# (this is std shotgun API syntax)
# any values starting with $ are resolved into path objects
filters:
    - { "path": "project", "relation": "is", "values": [ "$project" ] }
    - { "path": "sg_asset_type", "relation": "is", "values": [ "$asset_type"] }
```

This will inform that that the dynamic folder should create folders named using two shotgun fields on the Asset entity. Using standard Shotgun API query syntax, we also define constraints based on parent folders; only consider assets for the current project and asset type.

For a full reference of what types of nodes are supported, please see the reference documentation:

 > File System Configuration Reference.

## Deferred Creation and User Sandboxes

It is also possible to set up the Toolkit folder creation so that it runs in two phases: One phase whenever someone runs the folder creation command, one phase just before an application is launched. This is a behaviour that is built into the Toolkit Application Launcher (which is simply calling a standard API method to carry out the folder creation). With deferred folder creation you can handle the following use cases:

- If you have multiple different content creation applications in your pipeline and don't want to include a complete folder scaffold for every single one of them until they are actually needed, you can set up the folder creation so that each content creation app has its own deferred subtree in the config. When a production person or admin creates folders for the Shot, it will stop just before it starts creating work areas for maya, nuke, mari etc. Then, when an application is launched these folders are created just before the application starts up.
- If you want to create user based sandboxes in the file system, these will need to be created immediately prior to the work taking place. With deferred folder creation, you can add a special user node that makes this process easy. In the templates config, you then refer the to user node as `HumanUser` since this is the way the Shotgun API denotes it.

For more information about deferred creation, please see the reference documentation.

# Configuring Templates

Once you have created the file system structure it is time to configure a set of file system locations based on the folder structure above. These locations are called *templates* and are an integral part of the Shotgun Toolkit. The template file contains three parts: a *keys* section, where you define what each field means, a *paths* section where you can define template paths and a *strings* section where you can define string expressions. There are two syntaxes you can use in your file - one simple form that you can use for configs that have a single storage root and an advanced syntax that you can use for multi-root configs.

Example: Single Root Template Format

Example: Multi Root Template Format

There are a number of configuration options available for the templates file - you can find a complete reference here:

 > File System Configuration Reference.

## @include syntax in the template file

To cut down on repetition in the templates file, you can reuse fields:

```
paths:
    asset_root: 'assets/{sg_asset_type}/{Asset}/{Step}'
    maya_asset_work: '@asset_root/work/maya/@maya_asset_file'

strings:
    maya_asset_file: '{name}.v{version}.ma'
```

You can also split the templates across multiple files and include files in other files. For full details, see the reference documentation.

## Folder creation and templates

When you create a template that refers to folders created by the folder creation system, you need to specify the fields using 'Shotgun API' style notation! This a subtle detail which can be easily missed. The example above is a good illustration of this. In the folder creation, we have set up a configuration which first groups items by their asset type and then by their asset name. For example:

```
/mnt/projects/my_project/assets/character/Hero
```

We then want to create a template in the Toolkit that matches this path. In order for the Shotgun Toolkit to be able to match up the template with the path and the context, the fields need to be named the same way you would name them if you were using the shotgun API - the asset type folder level needs to be called `sg_asset_type` since this is the field name for this field in Shotgun, and the asset level folder needs to be called `Asset` (with a capital A) since this is how you refer to the asset entity type when using the Shotgun API.

# Hooks

Hooks are flexible pieces of the toolkit configuration. Normally, when you configure an app, engine or the core API, you specify a collection of parameters to define the behaviour of something. However, sometimes this is not powerful enough and this is where *hooks* come into play. Hooks are small chunks of Python code that you can use to customize an aspect of an app, engine or indeed the core. We have tried to design the hooks to be light weight and atomic. There are three different levels at which hooks appear in Toolkit. Read more about each level in the sections below.

## App level hooks

Each Toolkit app (and engine for that matter) comes with a collection of settings, some of which can be hooks. Each app carries a collection of default hooks which will be automatically used unless you specifically override them. Hooks are typically used to customize something very application specific. For example, for a Toolkit which loads images into Maya, the UI code and all the interaction logic is inside the app, but the little piece of business logic which actually loads the image into maya is inside a hook. This makes it possible for a studio to customize the behaviour; the default hook may simple create a standard texture node in maya, but a studio which wants to use a different node type can override the hook and thereby easily change the behaviour of the entire app - without having to rewrite any of the code!

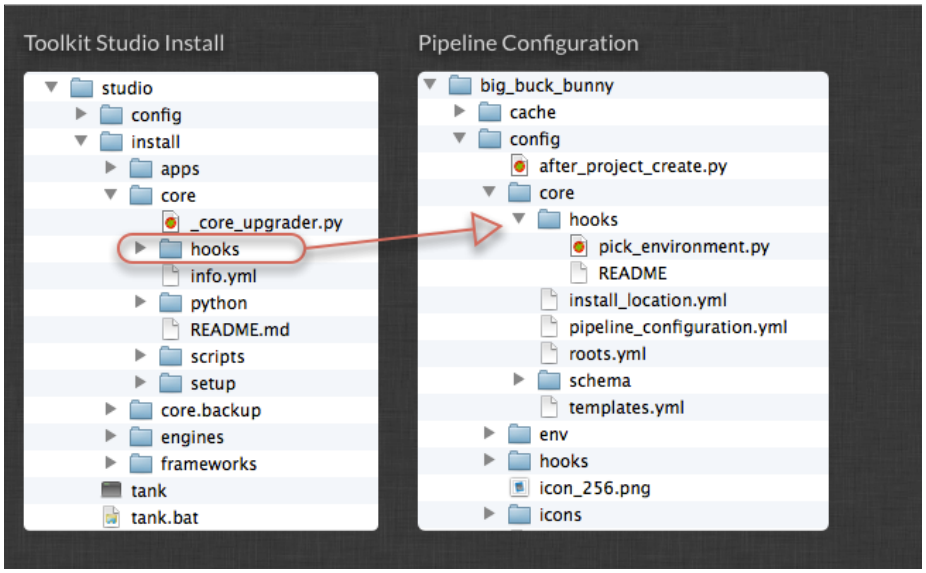
When customizing an app hook, you typically copy the default hook from inside the hooks folder in the app into your project's hooks folder. Next, you need to update the app settings inside the environment file so that it will read your new hook and not the default one. Your custom hook will automatically inherit from the default hook provided by the app, making it easy to add tweaks and small adjustments yet keep most of the business logic in the default hook. For more information about hook inheritance, please see the environment configuration reference and the hook section:

 > Environment Configuration Reference.

## Core level hooks

Core hooks makes it possible to override system wide behaviour in toolkit. Core level hooks are all overridden per project, so each project needs to set up individually with overrides. (If you keep re-using the same configuration when you set up new projects, this is generally straight forward).

The Core configuration area contains a special `hooks` folder. This folder is where you can put your own implementations of certain core hooks. Core hooks are similar to the hooks that you find in apps - it is a way to extract a code snippet from the Toolkit and be able to customize it. The core API allows you to override a number of different core behaviours, including file system I/O, folder creation and validation of file system structure.




By default, the Toolkit will pick up the core hooks it needs from with the API folder itself. If you want to customize the behaviour, take the hook file and copy it into the `config/core/hooks` area in your configuration. Then modify the code.

For a list of what core hooks are available, have a look at the hooks folder inside the Core API. Each hook contains extensive documentation about what it does and how it can be modified.

## Studio level hooks

There are also a couple of very special hooks which we call *studio level hooks*. These hooks are global and will affect everything. These hooks control aspects of Toolkit which sits outside of any specific project.

### Project name hook

The project setup process will prompt you for a 'disk name' for your project and suggest a name based on the project name in Shotgun, but with spaces and other non-file system friendly replaced with underscores. The disk name will be the name of the folder under which the project data and configuration is stored.

It is possible to use slashes when specifying the disk name. This will produce a project root point which spans over several folders in depth and can sometimes be useful if the studio is organizing its projects based on for example discipline (commercials, vfx, etc) or if the sheer volume of projects in a studio is so large that a single level in the file system would make it difficult to overview. Note that you should always use forward slashes ('/'). Toolkit will make the necessary adjustments on Windows.

In conjunction with the multi-level folders described above, it is also possible to customize the name that Toolkit suggests as part of the setup process. This is done in a special studio-level hook. If you want to customize this behaviour, create a file named `project_name.py` inside of the studio API location, in the `config/core` folder. This folder should already contain files such as `install_location.yml`, `app_store.yml` and `shotgun.yml`.

The `project_name.py` hook file can for example look like this:

```
from tank import Hook
import os

class ProjectName(Hook):

    def execute(self, sg, project_id, **kwargs):
        """
        Gets executed when the setup_project command needs a disk name preview.
        """

        # example: create a name based on both the sg_type field and the name field

        sg_data = sg.find_one("Project", [["id", "is", project_id]], ["name", "sg_t
ype"])

        # create a name, for example vfx/project_x or commercials/project_y
        name = "%s/%s" % ( sg_data["sg_type"], sg_data["name"] )

        # perform basic replacements
        return name.replace("_", "/").replace(" ", "/")
```

### Shotgun connection hook

Toolkit stores connection settings so that it can connect to its associated Shotgun instance. Sometimes it may be useful to control these connection settings in a dynamic fashion. In this case, create a hook file named `sg_connection.py` inside of the studio API location, in the `config/core` folder. This folder should already contain files such as `install_location.yml`, `app_store.yml` and `shotgun.yml`.

This hook that is called after shotgun connection settings have been read in from the `shotgun.yml` and `app_store.yml` configuration files. It makes it easy to modify connection settings procedurally, for example set up a proxy server which depends on some external environment variable.

The following three parameters are passed to the hook:

- `config_data` is a dictionary containing the settings in the shotgun config file that has been read in. It typically contains the keys `host`, `api_script`, `api_key` and `http_proxy`
- `user` is the user profile which the connection information is associated with. This is an expert setting and is almost always set to `defualt`.
- `cfg_path` is the path to the configuration file from which the config_data was loaded.

The hook needs to return a dictionary on the same form as config_data.

If you are customizing proxy settings, note that the proxy string returned should be on the same form as is expected by the Shotgun API constructor, e.g. 123.123.123.123, 123.123.123.123:8888 or username:pass@123.123.123.123:8888.

Below is an example implementation that can be used as a starting point:

```
from tank import Hook
import os

class CustomShotunConnection(Hook):
    """
    Allows for post processing of Shotgun connection data prior to connection
    """
    def execute(self, config_data, user, cfg_path, **kwargs):

        # explicitly set the proxy server setting
        config_data["http_proxy"] = "123.123.123.123"
        return config_data
```

# Configuring Apps and Engines

Now that we have a templates file set up that defines all the key locations on disk, we can start deciding which apps and engines to include in our configuration. As explained in other parts of the introductory documentation, the apps and engines configuration is broken down into series of *environments*. Environments are essentially alternative configurations - and this is useful since you most likely need to provide a different suite of apps, configured differently, for example shot work and asset work. For a more complex pipeline, you may want to break it down further in departments, so that modeling have a different setup than rigging. This is all handled via environments.

The environment file defines a number of possible engines, and depending on the application you are running, one of these sections will be used. For example, if you are running maya, you tell the Toolkit to launch the 'tk-maya' engine. The Shotgun Toolkit will first determine which environment to use (based on the current work area) and then look for a `tk-maya` engine inside this environment. When it finds it, it will load in all the apps defined for this section.

Each app has a number of settings that needs to be configured. When you install or upgrade an app, the Shotgun Toolkit will ask you to configure any setting that doesn't have a default value. The Toolkit Apps are often designed to be reusable, so depending on how you set it up, it can be used in many different ways and workflows. You can even have the same app defined several times in the same environment, for example you may want to have two publishers appear on the maya menu - one for a rig publish and one for a model publish - both using the same publishing app but with different configurations.

Example: An environment file

## Each App has a Location setting

Each item in the environment file has a special `location` token. This token defines where the Toolkit should pick up the app code from and how it should check for new versions of the app. For example, a location token may be defined like this:

```
location: {name: tk-multi-setframerange, type: app_store, version: v0.1.2}
```

The type indicates that this app comes from the app store and that a particular version is being used. When you run the update check, the Shotgun Toolkit will connect to the app store and check if there is a version more recent than `v0.1.2` available and if this is the case, ask you if you want to upgrade. The Toolkit supports a couple of different location types, including git and github, so you can build your own apps, use git to track them and when you create a new tag in git, the update will detect this handle it correctly in the upgrades check. For detailed information, see the reference documentation:

 > Environment Configuration Reference.

## Including files

You can include external files into you environment files. This is often useful when you try to centralize settings or manage overrides:

- You can organize your configuration so that you manage all the file paths to applications (maya, nuke) in one place even though you launch maya from multiple environments.
- Apps that are used with the same settings in many environments can be defined in a single place.
- You could maintain a 'central' pipeline configuration that is shared across multiple projects, and when this is updated, all projects will benefit from the update. Each project could then potentially override or extend the central config that is included if they need a particular behaviour.
- You can include files from your context as overrides, meaning that you could reconfigure settings parameters on a per-shot or per-asset basis. This is illustrated in our default config, where you can set override the paths to maya, nuke, etc. on a per shot or asset basis.

Example: Shot specific overrides for app launch

For more details on how the includes work, see the reference documentation.

## Configuring Template settings (file paths)

An important type of setting that many Apps use is the `template` setting. Apps needing to specify a file system location will use these settings. Apps are developed to be generic or flexible, designed to work with any file system structure or naming convention. The templates are the key piece which makes it possible for and app to be independent of the underlying file system. For more information about this, see the concepts introduction:

 > An Introduction to the basic Concepts in the Shotgun Toolkit.

When configuring apps and are coming across a template setting, you will need to specify a template that contains the right set of fields. Fields can be either required or optional, with required being fields that have to be included in the template and optional meaning that a field can be part of the template, however the app will also work if that field is not defined in the template.


When an app runs, it will create paths from the template that you specify in the configuration. These paths are created based on the current context plus a set of fields provided by the app logic. This means that if your template contains any fields that are not part of the context nor part of the optional or required fields for the app, the app doesn't know how to set a value for that field and hence wont work. This situation is

prevented by the Toolkit validating the configuration at startup.

When a template setting is validated, the Toolkit will first check the context - and compare the fields given by the context against the fields in the template. If the list of fields once the context has been compared isn't matching the required/optional parameter definition for the app, a validation error will be raised.

Practical Example: The snapshot app

## Using Hooks to customize App Behaviour

Another type of setting that apps use frequently is the `hook` settings type. A hook is a piece of code that the app will run as part of its normal execution. Hooks makes it possible to do very flexible customization of apps and means that some of the business logic of the app can be separated out from the actual app code.

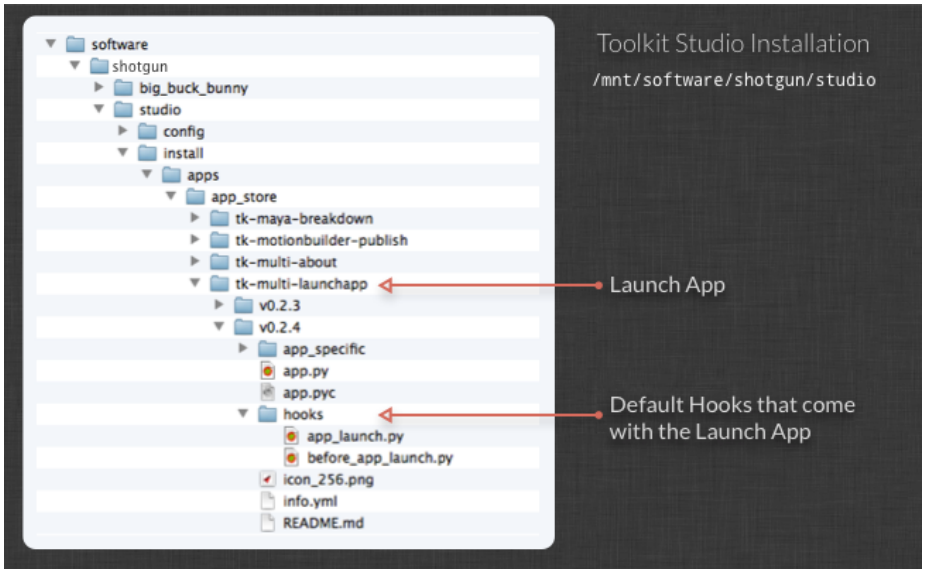
For example, imagine a breakdown app that needs to scan the scene for various references and file inputs. For maya, we can provide a default behaviour that handles the standard reference nodes that maya provides, but what if a studio is using a custom type of reference node? Of course the studio could always take the app and fork it in github, but that's a pretty drastic action given that all they really want to change is to add their custom node type to the code that scans the scene for reference nodes.

Instead, the breakdown could implement the scene scanning code snippet as a hook. This means that it is effectively a setting, a part of the app configuration. It will come with a default value, which will handle the plain-vanilla maya case, so it will work out of the box, but it is also easy for someone to configure the Shotgun Toolkit to change this behaviour completely if they wish.

When an App is installed, the hook settings will all show up in the configuration as *default*. This means that the app will use the built in hook implementation that comes with the App. For example, here's the launcher app configuration as an example:

```
launch_maya:
  engine: tk-maya
  extra: {}
  hook_app_launch: default
  hook_before_app_launch: default
  linux_args: ''
  linux_path: '@maya_linux'
  location: {name: tk-multi-launchapp, type: app_store, version: v0.2.4}
  mac_args: ''
  mac_path: '@maya_mac'
  menu_name: Launch Maya
  windows_args: ''
  windows_path: '@maya_windows'
```

We can see that there are two hooks here, `hook_app_launch` and `hook_before_app_launch`, both using the default app implementation. These hooks have been created to allow studios to customize the launch process, set environment variables etc.




In order to customize these hooks, first you need to find their original implementation. Each App has a hooks folder in which any hooks will be registered. Now grab the hook you wish to customize and *copy the hook into the hooks folder in your configuration*. Make the necessary code changes.

Now the environment configuration still has the hook set to `default` and as long as it has that, it will pick up the default hook that comes with the app and nothing else. In order to pick up your new settings, change `default` to the name of the python hook file you have inside the configuration hooks folder.

### Example: How to customize a hook

Here's a quick rundown of how to customize the `hook_before_app_launch` hook for an app launcher.

1. Copy the default hook implementation. See the image above for the location of the default hook that comes with the app. Now copy this file into your configuration area, in the hooks folder. For example, if your Pipeline Configuration is located in `/mnt/software/sgtk/big_buck_bunny`, you want to copy the file to `/mnt/software/sgtk/big_buck_bunny/config/hooks/before_app_launch.py`
2. Make the necessary changes to the python code.
3. Finally, update the environment configuration to use your new code:

   :::yaml launch_maya: engine: tk-maya extra: {} hook_app_launch: default hook_before_app_launch: before_app_launch # <-- uses custom hook! linux_args: '' linux_path: '@maya_linux' location: {name: tk-multi-launchapp, type: app_store, version: v0.2.4} mac_args: '' mac_path: '@maya_mac' menu_name: Launch Maya windows_args: '' windows_path: '@maya_windows'

The next time you run the launch app, your code will be executed instead of the default hook code.

Big Buck Bunny - footage courtesy of (CC) Blender Foundation, www.blender.org