Jabuka Terminology

This is an outline of the key terms in Jabuka, and how they apply to the Image Engine pipeline.

Contents [hide]

Basic terms [edit | edit source]

Entity [edit | edit source]

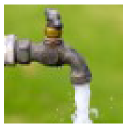
An **entity** is the most fundamental “thing” inside Jabuka. It is a tracked collection of data. Jabuka preserves each entity's entire version history, including previous file versions, for backup and archival purposes.

Everything in Jabuka is an entity of one type or another, and, like file types in a filesystem, each type is intended for a particular use.

Some top-level entity types include:

- Library
- Asset
- Component
- Show
- Sequence
- Shot

Entity source [edit | edit source]



A **entity source**, sometimes referred to simple as a **source**, is another entity from which an entity derives some or all of its data. For example, an animator's scene in Maya will typically reference an asset's modelling_modelGeo component and a rigging_animationRig component when animating its movement. When the scene is cached and the animationGeo component is published, Jabuka registers these referenced components as source entities.

Library [edit | edit source]

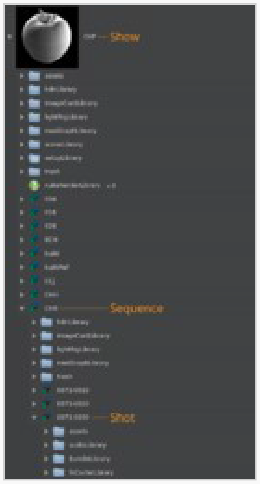
A **library** is an entity that organizes other entities, those entities being related to a particular topic or use. Think of it like a folder or directory in a file system.

Some examples:

- Asset library
- Scene library

- Image sequence library
- Bundle library

Location [edit | edit source]



Hierarchy levels of a shot.

A **location** is a position in Jabuka's hierarchy. In Jabuka, there are four main levels in the hierarchy:

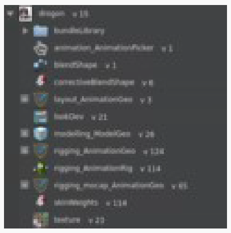
- Root
- Show
- Sequence
- Shot

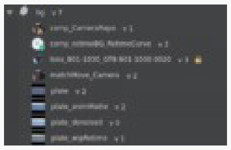
These levels correspond to the standard show-sequence-shot paradigm. They can be thought of as levels in a file system path. However, **do not** confuse them for Jabuka's file system, because its structure follows a more granular hierarchy scheme.

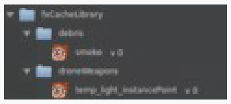
Component [edit | edit source]

A component is an entity that tracks a working element of production content. A component stores one or more content files and metadata values related to them. These are the entities artists work on and produce.

Components are typically organized by their intended function, into an asset or a shot library.


Example asset components in an asset.


Example cine asset components in a cine asset.


Example FX components (caches) in a shot's FX cache library.

Example image sequence components in a shot's image sequence library.

Version [edit | edit source]

animationRig v 171

A **version** is an iteration of an entity, most commonly a component. When an existing component is re-published, a new version of it is registered with Jabuka. A version is the smallest meaningful unit of artistic output in the pipeline and in an artist's workflow. At any given step of any task, an artist will be working on a version.

Publishing [edit | edit source]

To **publish** a component is to register a new version of a component (either an existing one, or from scratch) and make its data available in the pipeline. Publishing a component increases its version number by 1. Publishing is how Jabuka keeps track of components: saving files is not sufficient on its own. Publishing is the most important action in the workflow, and the one artists will be committing the most often. Every artist is expected to publish what they work on.

Depending on the farm and file system resources the version publishing process requires, it can take some time for a publish to complete.

Asset [edit | edit source]

An **asset** in Jabuka is an organizing container for components related to an asset, and represents the asset as a whole in the pipeline. Assets are in effect just libraries for asset components.

Cine asset [edit | edit source]

A **cine asset** is an organizing container for the constituent content in a 2D layer, primarily plate- and camera-related components. It is similar to a regular asset in that it organizes and contains components. Cine assets are in effect just libraries for cine asset components.

Cine assets were invented at Image Engine to reduce confusion in the workflow and streamline throughput when dealing with numerous 2D-related components over multiple layers.

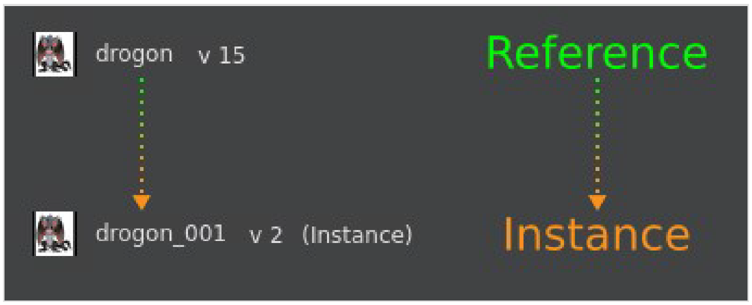
For more information on cine assets, see Cine Asset.

Copied entity [edit | edit source]

A **copied entity**, or just a **copy**, has the same meaning as the common filesystem sense of the word *copy:* an identical but independent duplicate of an entity. After the copy is made, changes can be made to both the original entity and the copy with no effect on the other.

A caveat to this is that to reduce disk space use, a copy in Jabuka only actually duplicates the files from the latest version of an entity, and ignores those from prior versions.

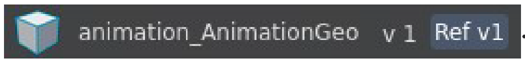
Asset instances and references [edit | edit source]



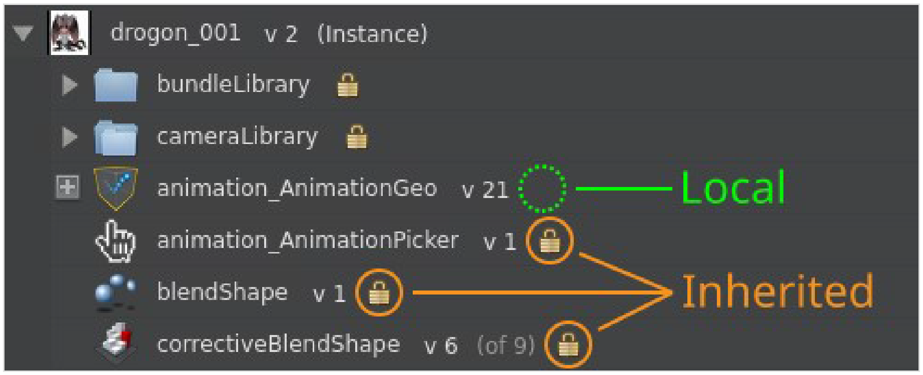
An **asset instance**, often shortened to just **instance**, is a copy of an asset, but instead of containing copies of the original asset's components, it only contains links to them.

Asset instances propagate assets from the root location of a show to its shots, so they are the most common type of instance in Image Engine's pipeline. Other entity types can be instanced, but they are rarer.

A **reference** is the complement of an instance: it is the original entity from which the instance is sourced.

While the term *reference* is applicable to the source of any instance, it typically only comes up with regards to individually instanced components, such as when a component is instanced from one asset to another. Referenced components will have tags in the interface that indicate the reference's version, like  animation_AnimationGeo v 1 Ref v1. The references of asset instances do not have such tags.

Local and inherited entities [edit | edit source]


drogon_001 v 2 (Instance)
bundleLibrary
cameraLibrary
animation_AnimationGeo v 21
Local
animation_AnimationPicker v 1
blendShape v 1
Inherited
correctiveBlendShape v 6 (of 9)

A **local** entity is an entity that has its data and files at the location it appears in.

An **inherited entity** is an entity that is available at a location, but any files it has are in fact stored at location higher up in the hierarchy. Inherited entities exist as links, and are created when an entity is instanced. When an inherited entity is accessed, Jabuka will point to the files in the original entity higher up in the hierarchy.

For example, if an asset at...

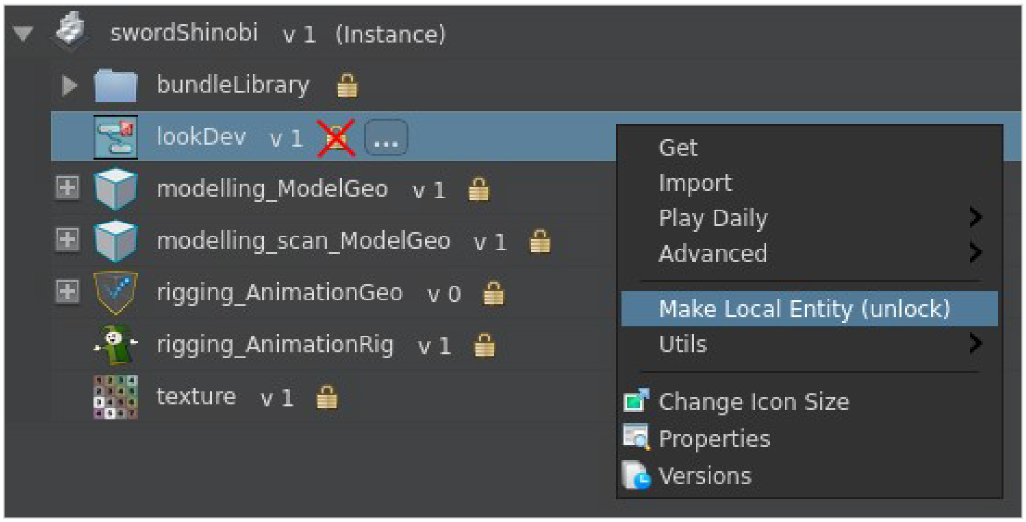
```
GT8/assets/character/drogon
```

... is instanced to...

```
GT8/801/801-1030/assets/character/drogon
```

... all of the instanced asset's components will be links to their originals, and no actual component data will be copied (unless made local; see below).

Localizing [edit | edit source]



An inherited entity can be **localized** or **made local**, which unlinks it from its reference and actually copies its content over. Just like a regular copy, once an entity is made local, it can be changed without affecting the original.

Bundle [edit | edit source]

A **bundle** is a container for all the input or output content for one task in the pipeline. A task here could be an entire department's output for a shot or asset, or a sub-task within a department. Fundamentally, a bundle is a simply a tracked list entity versions, which, like other entities, can be updated and has versions of its own.

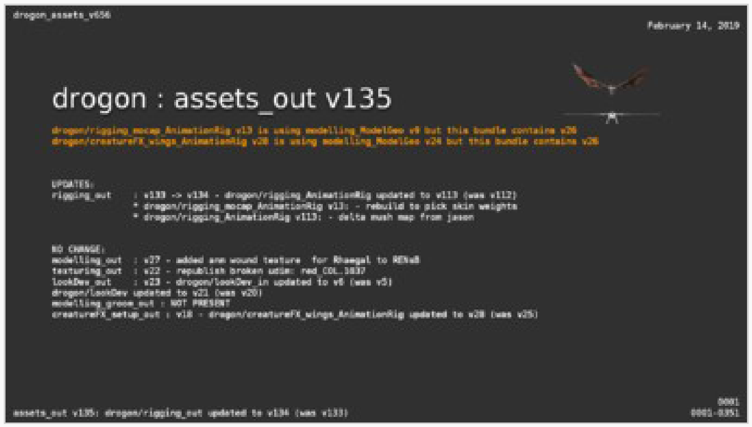
Departments use bundles to pass work to each other, and they are fundamental to Image Engine's pipeline. You can think about the Out bundle as an iteration of the total output of a task in the pipeline. The use of bundles for department sub-tasks allows us to have multiple artists from the same department each working on a task on the shot or asset at the same time.If at every step of the process we make sure to include the correct data in our bundles, then no department will ever be missing the content they need.

There are two types of bundles:

- **In** bundle
- **Out** bundle

An **In** bundle lists all the approved entity versions from the previous departments that are safe for use, and informs the artist of what is available to use in their task. The artist then works with this data, and in turn generates outputs, which they publish into an **Out** bundle, to be used by an artist in a downstream task.

Bundle render [edit | edit source]

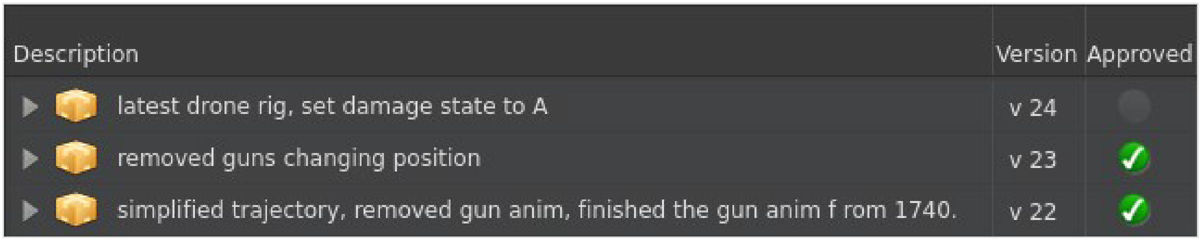


```
drogon_assets_v656
                                                                 February 14, 2019

drogon : assets_out v135
drogon/rigging_mocap_AnimationRig v13 is using modelling_ModelGeo v9 but this bundle contains v26
drogon/creatureFX_wings_AnimationRig v28 is using modelling_ModelGeo v24 but this bundle contains v26


UPDATES:
rigging_out     : v133 -> v134 - drogon/rigging_AnimationRig updated to v113 (was v112)
                * drogon/rigging_mocap_AnimationRig v13: - rebuild to pick skin weights
                * drogon/rigging_AnimationRig v113: - delta mush map from jason


NO CHANGE:
modelling_out   : v27 - added arm wound texture  for Rhaegal to REN#8
texturing_out   : v22 - republish broken udim: red_COL.1037
lookDev_out     : v23 - drogon/lookDev_in updated to v6 (was v5)
drogon/LookDev updated to v21 (was v20)
modelling_groom_out : NOT PRESENT
creatureFX_setup_out : v18 - drogon/creatureFX_wings_AnimationRig updated to v28 (was v25)


                                                                              0001
assets_out v135: drogon/rigging_out updated to v134 (was v133)               0001-0351
```

The slate frame of the bundle render based on the 135th version of Drogon's assets_out bundle. Notice the breakdown of changes described by the burn-in.

A **bundle render** is a render made using the entities from a version of a bundle. Since a bundle represents the output of a pipeline task, a bundle render offers a guaranteed look at the output of that task. During review and QC, a lead or sup will use the bundle render as a daily, and approve its bundle if it passes. By approving the bundle, they automatically approve all of the task's component versions inside that bundle.

Approving bundles [edit | edit source]



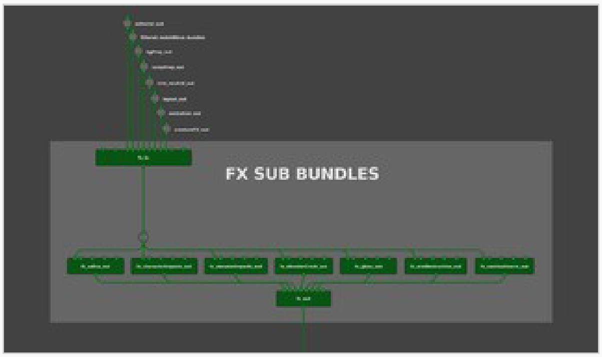
Description	Version	Approved
latest drone rig, set damage state to A	v 24	●
removed guns changing position	v 23	✓
simplified trajectory, removed gun anim, finished the gun anim f rom 1740.	v 22	✓

To **approve** a bundle means that a lead or sup QC'd a bundle render, and marked the bundle's contents for use in the pipeline. Keep in mind that *approved* does not mean *final*, but that other departments can use it. Once a bundle is approved, all of the entities it contains become automatically approved as well, and downstream department(s) can use those entities. An approved Out bundle is, in effect, a collection of entities that are guaranteed to work together.

Since an approved bundle is a list of entities guaranteed to work together, its corresponding bundle render offers a guaranteed snapshot of those entities. With both the bundle and its bundle render tracked and linked in Jabuka, we can maintain a full history of the changes we make, and recreate any bundle render we have generated on an active show.

Advanced terms [edit | edit source]

Sub-bundle [edit | edit source]

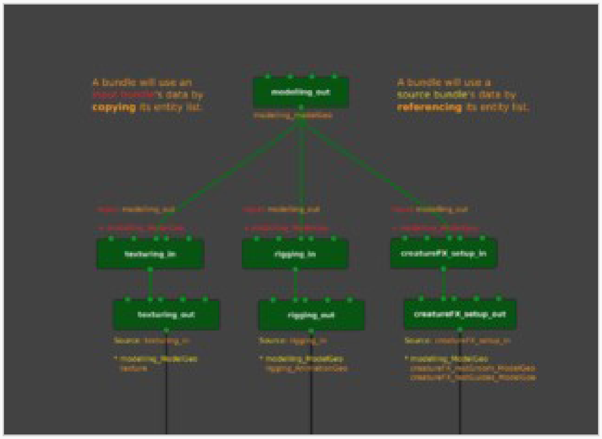


Sub-bundles for the FX department used on BLD.

A **sub-bundle** is an Out bundle for a sub-stage inside a department's workflow, used to help isolate work on different components and account for more granular pipelines. With them, multiple artists from the same department can track and work on different elements of a shot at the same time.

The most common examples of sub-bundles are in FX's workflows on FX-heavy shows. So that the individual FX elements on a shot can be separately cached, QC'd, and tracked in Shotgun, typically there is one Out bundle per element. For example: fx_glass_out, fx_smoke_out, fx_saliva_out. Each are fed by fx_in, but the separate artists working on those elements publish to each. Then, the sub-bundles feed into the main fx_out bundle for the pass-off of the summed state of all FX elements to downstream departments.

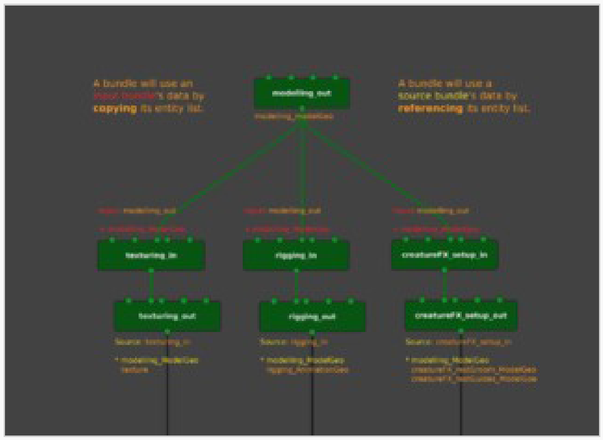
Input bundle [edit | edit source]


A bundle will use an input bundle's data by copying its entity list.
modelling_out
A bundle will use a source bundle's data by referencing its entity list.
texturing_in
rigging_in
creatureFX_setup_in
texturing_out
rigging_out
creatureFX_setup_out
An input bundle in the asset workflow template.

An **input bundle** is a bundle from which another bundle **copies** a list of entities. As a basic example, say that bundle A is an input bundle of bundle B. If A's entity list is `1,2,3`, and B's list is `4,5,6`, then the next time B updates, its list will become `1,2,3,4,5,6`.

In practice in our workflow templates, the most common input bundles are department Out bundles connected to downstream In bundles through inputBundles plugs. For example, modelling_out is an input bundle for texturing_in. Here, modeling_out adds its list of entities to texturing_in, resulting in both lists containing the asset's model cache.

Source bundle [edit | edit source]



Source bundles in the asset workflow template.

A **source bundle** is a bundle from which another bundle **references** its list of entities. This reference can be thought of like a scene reference in 3D tools like Maya. The primary purpose of a source bundle is so that a bundle can dynamically load another bundle's list of entities at a moment in time without modifying its own list. In the database, source bundles are just like source entities, and are defined in a bundle's `source` property.

In practice in our workflow templates, the most common source bundles are department In bundles connected to their Out bundles through Sources plugs. For example, texturing_in is a source bundle for texturing_out. During a process like a bundle render, texturing_out will reference texturing_in's entities, meaning it will read its components, but texturing_out's own list of entities **will not** change.

See also [edit | edit source]

- Jabuka
- Bundle Basics 1: Bundle Pipeline

Categories: Jabuka | Pipeline