Kabaret Documentation

Release 2.0

Damien Dee Coureau

Feb 08, 2023

Introduction

Kabaret is a Free and Open Source VFX/Animation Studio Framework.

It is made for TDs and Scripters involved in Production Tracking, Asset Management, Workflow and Pipelines used by Production Managers and CG Artists.

CHAPTER 1

Main Features

- Fast and Easy Project modeling.
- No decision made for you, your pipe = your rules !
- Based on 20+ years of experience in the field.
- Generative end-user GUI: zero code *needed*.
- Modular and Extendable pure-python architecture.
- Plugin system for zero/low code modularity.
- Python 2.7+ and 3.6+ compatible.
- Embeddable in PyQt4, PySide, PyQt5, PySide2 and Blender applications.
- Tested under Windows and Linux.
- There's an example project so you can start experimenting in less than 5 minutes !

CHAPTER 2

Status

Kabaret has been used in productions for over 7 years for Commercials, Teasers, TV Shows and Feature Movies.

Among the 20+ releases made since its source opening, only one single (and minor) breaking change was introduced.

2.1 Why and How

2.1.1 Why another Pipeline software ?

There are many existing solutions, both commercial/closed and free/open, to handle the task of "CG Project Management". Notable and recent examples include CGWire, Kurtis, shotgun...

Almost all of them fall in two categories: Meta-Data management or Dataflow Modeling.

The **Meta-Data** Management tools are often *Production Team* oriented and subtitled as "*Better than google docs*™". They manage a more or less flexible "entity" system and their dependencies: assets info, shot list, statuses, frame ranges, etc.

As vital as this is to complete a CG project successfully, it does not give any help in the practical matter: the technical side of an artistic cooperative work.

The **Dataflow** Modeling tools are often *Talent Team* oriented and captioned as "*Automate everything !*©". There is a strong culture of dataflow in the CG world because many of our Digital Content Creation tool use them under the hood, with great success. Automating non-artistic tasks involve things like dependencies, parameters and process execution. It sounds pretty much like a dataflow.

As efficient as they are to manage 3D data or image manipulations, dataflows do come with restrictions. A major one being that the graph needs to be "acyclic". This becomes a real issue when you try to represent the highly iterative day-to-day tasks of an artistic cooperative work.

After years of using and implementing different flavors and mixes of both of those, it is now obvious that the solution is elsewhere.

Hence the need for another approach.

2.1.2 How is Kabaret different ?

Interestingly, the CG world is not flooded by BPM and Workflow concepts, despite the fact that the BPM definition pretty much describes what we are looking for:

> "*Business Process Management (BPM) is a discipline involving any combination of modeling, automation, execution, control, measurement and optimization of business activity flows, in support of enterprise goals, spanning systems, employees, customers and partners within and beyond the enterprise boundaries.*"

The reason might probably be that the "Artistic" world is not keen on being treated as an "Industrial Business" or an "Orchestrated and repeatable pattern of business activity". It is nevertheless what project management aims to bring to the table.

Workflow does a pretty good job as modeling the "*highly iterative day-to-day tasks of an artistic cooperative work*" and **is a better fit than Dataflow**. On the other side it does not deal down to data processing **and does not replace the Dataflow**.

Representing both in a single graph is the idea that led to **Kabaret**.

But there's more !

We also wanted to provide:

- **A framework, not a Solution**

  Every need is different and every project should not deal with decisions made for another project or studio. Any choice you did not make yourself is not the better one. Kabaret gives you an abstract set of tools that you can use as you want. The balance of Workflow / Dataflow you need is up to you.

- **Rapid Prototyping, Fast Development, Live Update, Schema-less**

  This is the only path to happy end-users. Having 100 Artists working on the project for six months should not mean that the workflow can't evolve, and it should not require downtime or migrations to do so.

- **The end of GUI development**

  It can cost more than the implementation of the actual pipeline features. We need automatic default GUI, with configurable behavior. Of course you can extend or even replace the default, but you'll get a pretty good GUI out of the box.

- **Problem isolation, Reusability of solutions**

  Two projects are not the same, but they surely share a lot: Naming conventions, version control, long-running tasks dispatching, etc. Once something is dealt with, it is available for every other project. Once a solution is in use, updating it updates all projects.

- **Modular and Extendable**

  There will always be more. Let's deal with that later by adding blocks :D

Doesn't it whet your appetite ? :D

2.1.3 What Kabaret is not ?

Kabaret is not a Pipeline software, nor an exhaustive Pipeline solution. It is a Framework and it delivers only generic features that may or may not be used by someone to build his very own solution.

That being said, there are a number of generic features that are not available in Kabaret. The reason is that we want to keep it to the bare minimum so that code quality prevails over feature quantity. It does not mean that we won't provide those, on the contrary. We focused on delivering an extensible architecture so that whatever would be the scope of a

missing feature, one can implement it without modifying Kabaret's code, and package the result to share it with the community.

Here are some examples of what we will provide as 'extensions' packages:

- A Script view, with python syntax highlighting and code completion.
- A collection of Flow Objects to handle planning information along with a Gantt view to visualize and edit them.
- Other collections of Flow Objects like BPM Workflow, Shotgun sync, mail automation, etc.
- An Actor to manage subprocess spawned by the flow.
- An Actor to manage users, teams and their preferences.
- Some Actors as alternative key-value stores.

You can look for extensions on the Python Package Index or discuss with the community on the Kabaret Studio discord channel.

We encourage you to share your extensions there too :)

2.2 Quick Start

2.2.1 Demo & Showcase

Goal

This tutorial will let you run and play with a Kabaret standalone application.

Prerequisites

For this tutorial we assume you have installed kabaret in either options described here.

Kabaret uses various functionalities of redis and we will use a local redis-server to continue. You can download one from this page (windows users can download here) and start a server with the default configuration. We will assume it is available at **localhost** on port **6379**.

Preparation

Kabaret is not an application but a framework and it is up to the user to build his very own tools. The convention is to package this code into a "studio" python package, as it will contain everything your studio will need. Before learning how to do this in next tutorials, we will have a look at what kabaret looks like for the end user.

In order to do so, we will run the 'dev_studio' that our developers use to test and showcase their functionalities. This python package installs itself when you install kabaret so you should already be able to import both packages:

```
1 import kabaret
2 import dev_studio
```

Now run python with the following command line options:

```
python -m dev_studio.gui --cluster KABARET_DEMO --session KabaretDemo
```

Note: If you need to connect to a remote redis store, you can use –db, –host and –password. You can also use -h at *the end* of the command line to list all available options.

You should see a Kabaret window like this:

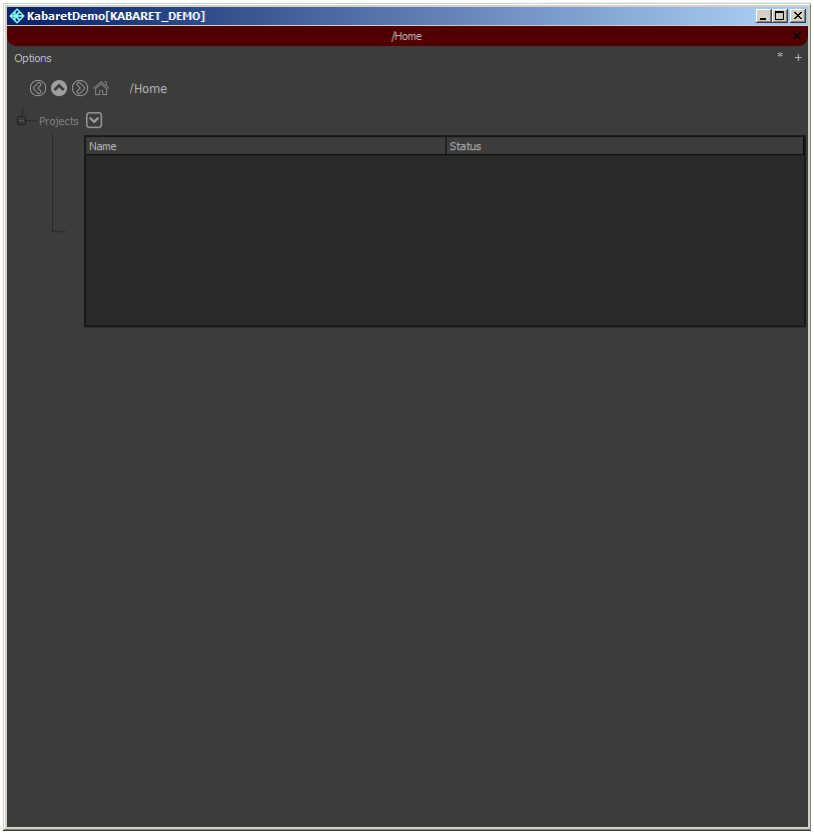


Fig. 1: Your very first encounter with Kabaret GUI \o/

Let's play !

Bravo, you have just run your first kabaret application \o/

This window is Kabaret's default standalone with the default look and style, showing the default view: a project explorer. There is not much to see yet so we will create a project to browse.

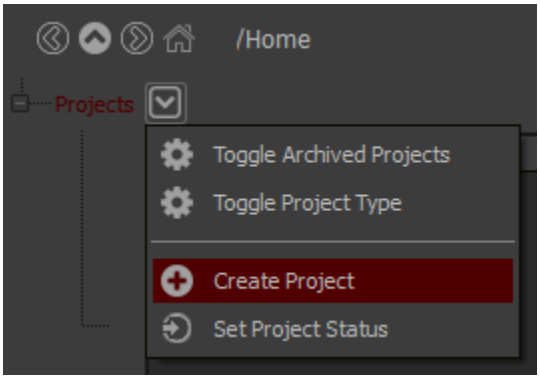
Locate the button on the right of the "Projects" field. It contains the actions you can perform on the list of available projects. Left click it, and select "Create Project"

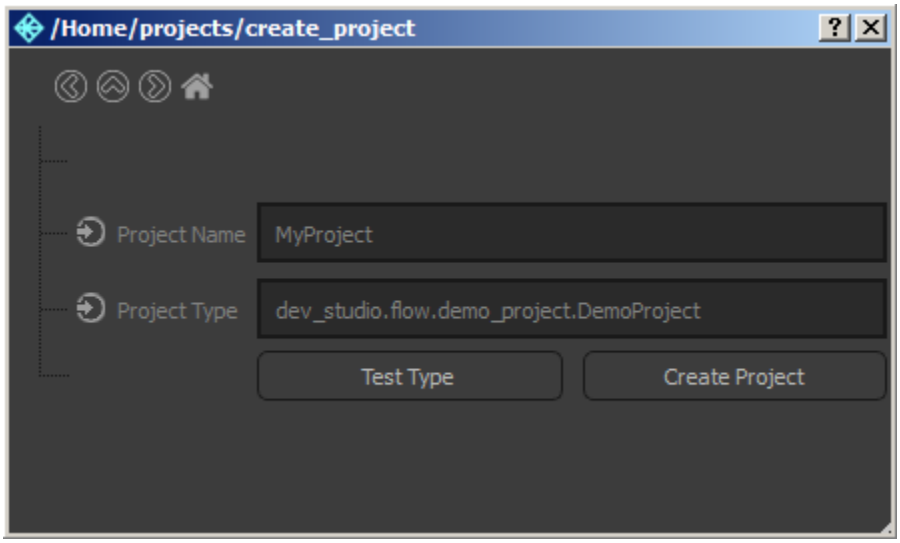
A dialog will appear, with a field for the Project Name and another one for the Project Type. You can leave default values and click the "Create Project" button.

You will now see the "MyProject" entry in the Projects table. Double click on it to open it and have fun :p

Take some time to familiarize yourself with navigation:

- Click the [+] sign on the left of a label, or double click when the label is highlighted to expand it.
- Double click on a label to enter it.


/Home
Projects
Toggle Archived Projects
Toggle Project Type
Create Project
Set Project Status


/Home/projects/create_project
Project Name
MyProject
Project Type
dev_studio.flow.demo_project.DemoProject
Test Type
Create Project

- Double click with the Control key pressed to open in another view (you can rearrange view using drag&drop)
- Use the navigation button at the top-left of the view (use the Home button to go back to the project list)
- Use the navigation address at the top of the view (left or right click on different sections)
- Resize stuff with the middle mouse button.
- ...

You should discover basic possibilities of kabaret GUI:

- Groups
- Action Menus, Actions Buttons, Action Dialogs
- Maps (tables/list)
- Parameter fields with various editors (text, boolean, date, etc.)
- Summaries
- Drag'n'drop
- ...

Everything you see here has been defined by the **dev_studio.flow.demo_project** module.

This project is just a dummy non-functional mockup, but you can challenge yourself into creating a Film and some Shots. Maybe even an asset that you could drag'n'drop into a shot casting...

Note: It you accidentally closed all views, you can right click anywhere and select one of the available views.

Conclusion

We've seen just the tip of the iceberg here, but it hopefully made you want to discover more.

If you're python fluent, you can watch the content of the *dev_studio/flow/demo_project.py* file (start your journey at the end with the DemoProject class definition). You may also want to create a second project with the type *'dev_studio.flow.unittest_project.UnittestProject'*, it is full of interesting inline comments about the *kabaret.flow* usage...

Or just browse to the next tutorials where we will setup your own studio and create a simple project ! :)

2.2.2 Create My Studio

Goal

The Kabaret framework covers many aspects of a TD needs. The most basic one might be to present a GUI to the Artists with some tools to execute.

We are going to build such a GUI with kabaret.

Prerequisites

For this tutorial we assume you have installed kabaret in either options described here, and have a local redis-server as described in the previous tutorial prerequisites.

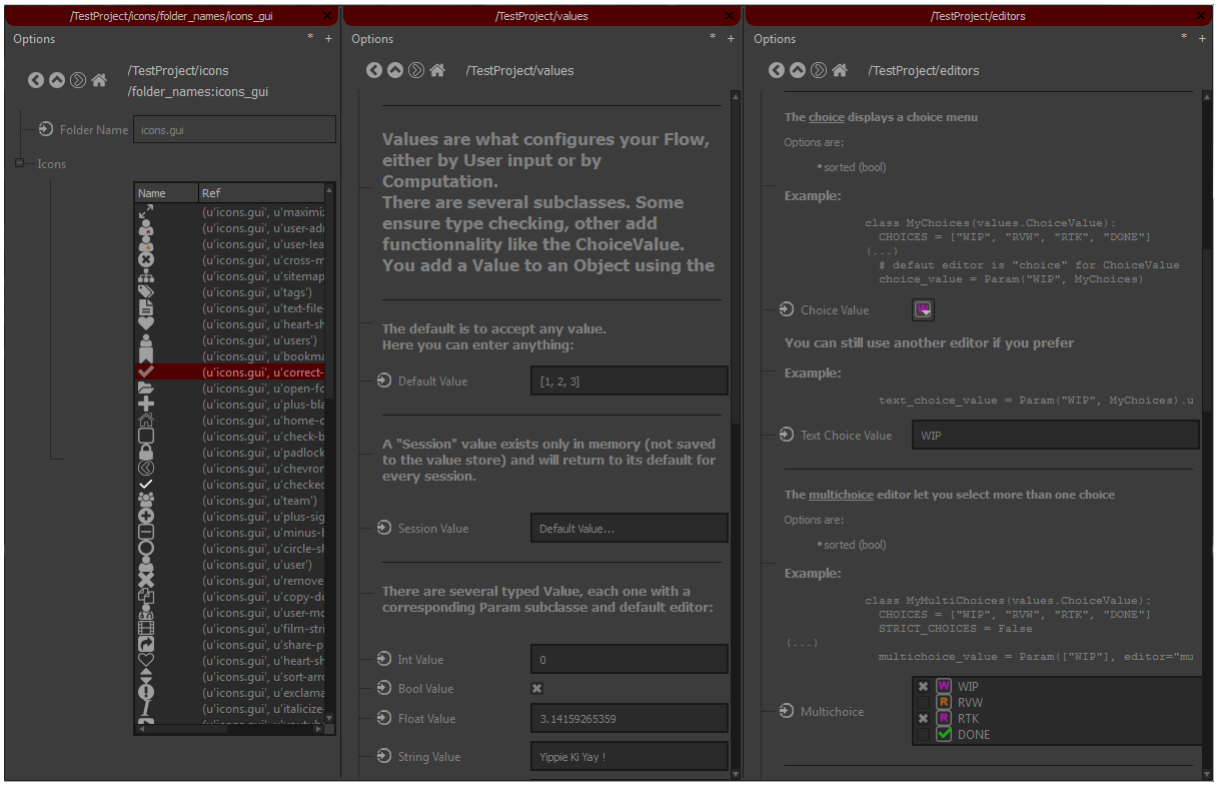


Fig. 2: Here is a sample of what you can find in the UnittestProject

Preparation

Choose a folder where you want to put your code. This location will be referred to as *<BASEDIR>*.

We will need to import python code from there, so you should have it in your PYTHONPATH (or use any other trick you flavor. . .).

Let's do it !

We are going to create a python package containing all the code using kabaret. There is a convention to name such package as *<xxx>_studio* since they tend to contain all the proprietary code you need to run a studio. So let's name ours *'my_studio'*.

Create the *<BASEDIR>/my_studio* folder and add a *__init__.py* file inside it.

Now we are going to create a module that builds and shows our GUI. Let's have it as *my_studio.gui*.

Create the *<BASEDIR>/my_studio/gui.py* file, and open it in your favorite text editor.

Kabaret applications are managed as *'sessions'*. All sessions in the local network communicate with each other so that you can build a truly collaborative application for your users. But you may need to handle more than one studio in a single network so in order to restrict those communications, sessions are organized in *clusters*.

Another purpose of the session is to provide an API to kabaret features and kabaret extensions features. This API is composed by collections of *commands*. A session contains a configurable list of *Actors*, and each actor defines a single collection of commands.

There are a couple of session types available in the framework. One is a *Standalone GUI Session*, and we are going to use it.

In the *gui.py* file, import the kabaret.app.ui.gui module and subclass the *KabaretStandaloneGUISession* it contains:

```
from kabaret.app.ui import gui


class MyStudioGUISession(gui.KabaretStandaloneGUISession):

    pass
```

Now let's have our *gui* module act as a main by adding the classic *__name__* test and create our session. Add those lines at the end of *gui.py*:

```
if __name__ == '__main__':
    session = MyStudioGUISession(session_name="MyStudio")
    session.cmds.Cluster.connect(
        host='localhost',
        port='6379',
        cluster_name='TUTORIALS',
        db_index='1'
    )
    session.start()
    session.close()
```

Here we create our session, giving it a name which will help identify it in the cluster and in logs. We use the *'connect'* command of the *'Cluster'* Actor to configure the communication with other sessions. We start the session and close it after the last window of the GUI get destroyed.

You can now launch you very own application using python's *-m* flag:

```
python -m my_studio.gui
```

Windows users may want to create a *.bat* file containing something like:

```
set PYTHONPATH=%PYTHONPATH%;<BASEDIR>
C:\python27\python.exe -m my_studio.gui
pause
```

You should see the classic default Kabaret window, with a project explorer view:

And in the shell you should be able to see:

```
kabaret - INFO: Registering 'Cluster' Actor from kabaret.app.actors.cluster
kabaret - INFO: Registering 'Flow' Actor from kabaret.app.actors.flow
kabaret - INFO: Connecting to localhost port:'6379', index:'1'
kabaret - INFO: Connected to Cluster 'TUTORIALS'
kabaret - INFO: Configuring Project Registry
kabaret - INFO: Subcribing to flow_touched messages.
kabaret - INFO: [Broadcast Message] u'Cluster joined by Dee:MyStudio-8872@Dee-PC'
```

This may not seem like much but less than 10 python lines you have built a highly configurable and extensible application that can communicate with everyone in the local network. If you click on the **'*'** button on the top right corner of the default view, you will see that this application has a classic multi-view interface where you can drag'n'drop views to move and/or stack them.

Optional fun

Before adding actual useful things into this GUI, let's see how we can customize it, just for fun :)

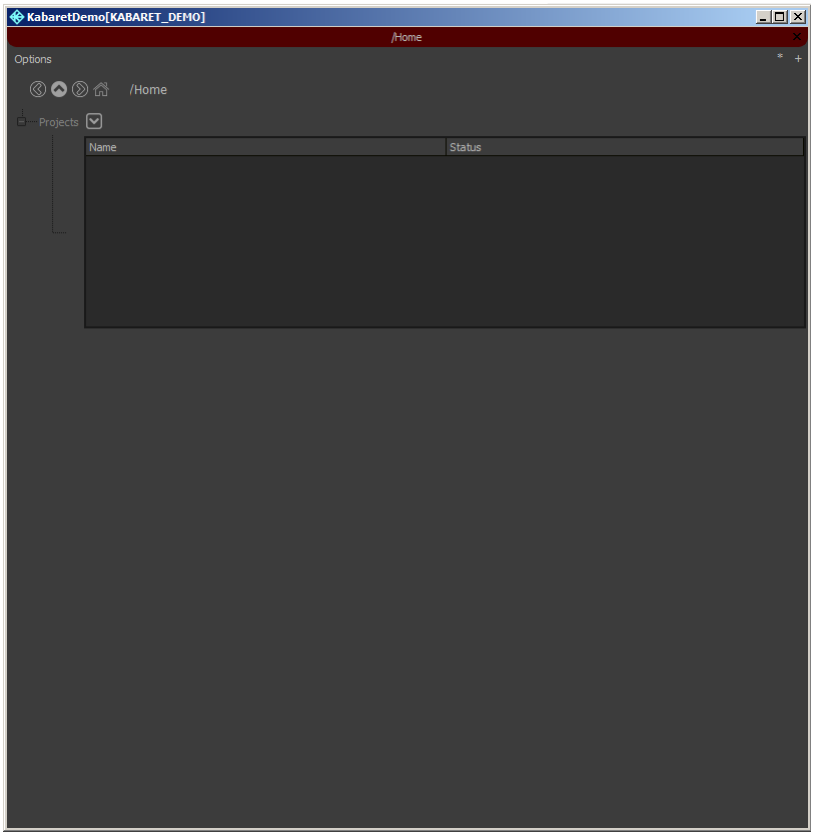


Fig. 3: Your very own GUI \o/

In *gui.py*, add those lines just before the *__name__* test:

```
from kabaret.app.ui.gui.styles import Style


class NoStyle(Style):

    def apply(self, widget):

        pass


NoStyle('NoStyle')
```

We've created and applied a custom style to the gui. This style does nothing in its *apply()* method so if you launch your GUI you will now have something looking like the default for your current Operating System theme.

Now let's do something more interesting by subclassing the default style and rebranding it to a bluish identity. Add those lines just before the *__name__* test:

```
from qtpy import QtGui
from kabaret.app.ui.gui.styles import dark


class MyStyle(dark.DarkStyle):

    def apply(self, widget):
        super(MyStyle, self).apply(widget)

        palette = widget.palette()
        palette.setColor(palette.Window, QtGui.QColor('#556'))
        palette.setColor(palette.Base, QtGui.QColor('#335'))
        palette.setColor(palette.Highlight, QtGui.QColor('#002'))
        palette.setColor(palette.HighlightedText, QtGui.QColor('#88D'))
        widget.setPalette(palette)


MyStyle()
```

We've created a new style based on the default one and we have overridden a few color settings to have a nice (?!) blue ambience.

There's way more you can do with the framework like using stylesheets, replacing or adding icons, etc. But the default theme and icons have been carefully crafted and selected for a nice CG Artist experience.

Conclusion

The philosophy of Kabaret is to provide high-level features but also to reduce the boilerplate to the strict minimum without closing the door to customization and personalization.

Now that you have the environment set up (1 folder and 2 files !) you can build a collaborative application with a classic multi-view GUI.

In the next chapter we will see how convenient and efficient this can be for your workflow/pipeline users.

2.2.3 My First Project

Goal

The Kabaret framework covers many aspects of a TD needs. The most valuable one is probably to build a pipeline and/or workflow for the artists.

Without getting too much in depth into this topic, we are going to give you a hit of what if *feels like* to build something with kaberet.flow, the package responsible for making this task a pleasure.

Prerequisites

For this tutorial we assume that you have successfully walked through the previous one and that you can run a Kabaret standalone session.

Preparation

Get comfy, we need to talk before the fun.

Kabaret's solution to develop pipelines and workflows is named *Flow* and is available in the *kabaret.flow* package. The reasons why *kabaret.flow* is outstanding are beyond the scope of this tutorial, but you should know that one of them is that it's really simple to understand and to use.

The idea is to define a schema of your project using objects and relations between them. That's the whole concept. Nothing more. Anything done with the flow is just some objects related to each other.

kabaret.flow provides a list of different relations and a few specialized object types. You will extend those objects and use the existing relations to create the schema of your project. This is often related to as project "modeling".

Here are the kinds of objects at your disposal:

- **Objects** are the base for everything.
- **Values** are Objects that hold data.
- **Maps** are Objects containing a dynamic list of Objects.
- **Actions** are Objects that execute code.

The most often used relations are:

- **Parent**: the related Object contains this Object
- **Child**: the related Object is inside this Object
- **Param**: the related Object is a Value

We are going to use those Objects and Relations to model a *really* basic project consisting of just a list of shots. let's create this module in our studio:

```
<BASEDIR>/my_studio/my_first_flow.py
```

We will write all this tutorial code in this file. The complete code can be seen *here*.

Note: In real life situation we would probably define our project in a package instead of a module, and it would be a good choice to have all the projects in one package like: *my_studio.flows.my_first_flow*

Let's play !

Now grab your favorite mechanical keyboard, we're diving in !

Foreplay

A project is defined by a *root Object* that *contains* all other Objects. Our project will consist of a list of Shots and a few settings values. Let's add a basic structure for that:

```
from kabaret import flow


class Shots(flow.Map):
    pass


class ProjectSettings(flow.Object):
    pass


class Project(flow.Object):

    shots = flow.Child(Shots)
    settings = flow.Child(ProjectSettings)
```

Flow code is easily read from bottom to top. Let's walk through this code in this order.

The *Project* class is our project definition. It's a *flow.Object* extended with two *Child* relations: the *shots* and the *settings*. The *Child* relation means that *Project* "contains" *shots* and *settings*.

The *ProjectSettings* class is a bare *flow.Ojbect*. We will use it to group settings values.

The *Shots* class is a *flow.Map*. A *Map* can store several objects. We will use it to store our shots.

Let's see how it looks in your application. Start it, create a new project with the name *"MyFirstProject"* and the type *"my_studio.my_first_flow.Project"*. After entering the project you should see something like this:

Ok so if you're not impressed by the GUI built with 8 lines of code, let's see two sugar features of the flow package now. Don't close your application, but go on and swap order of *shots* and *settings* in the *Project* class and save the file:

```
class Project(flow.Object):

    settings = flow.Child(ProjectSettings)
    shots = flow.Child(Shots)
```

Now in the *"Option"* menu at the top-left of your flow view, select *"Activate DEV Tools"*. A new *"[DEV]"* menu should appear. In this menu select *"Reload Project Definitions"*. And voilà !

The order you define your object relations is reflected in the GUI. And you don't need to restart your application to see your changes. We're going to use that a lot !

Now let's keep it nice and swap back those two relations please...

Using Values

It's time to add some values to the ProjectSettings.

All our shots will contain some files, so we're going to need a place to store them. We will be using a *Param* relation to let the user edit this location and a few IntParm for things that would make sense in a real world scenario:

```
class ProjectSettings(flow.Object):

```

(continues on next page)

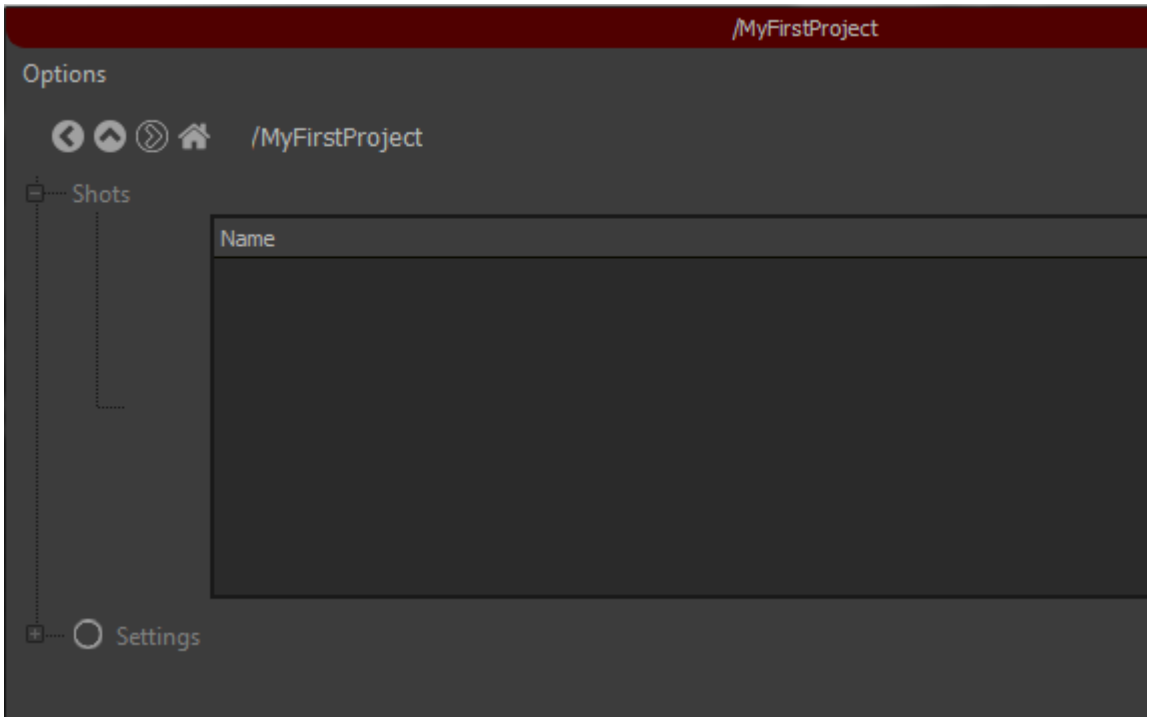


Fig. 4: Hmm... You might not be impressed yet :p

(continued from previous page)

```
10      store = flow.Param('/tmp/PROJECTS')
11      framerate = flow.IntParam(24)
12      image_height = flow.IntParam(1080)
13      image_width = flow.IntParam(1920)
```

Now click *[DEV] -> Reload* and open the *Settings* field.

All fields show the default value defined by our code. If you edit the *Store* field you will see a blue background until press enter and the new value is stored. If you edit the frame rate and try to input something else than an integer, a red background appears in the field: your value was rejected. All those fields accept python expressions so you can enter *3*10* in the *Frame Rate* field and the result will be stored.

Now run another instance of your application and browse to *MyFirstProject/settings*. Change a value and see how the first application reflects the change without any intervention. This works for every instance of your application in the local network.

And now let's click the home button and create a project *"MySecondProject"* with the same type *"my_studio.my_first_flow.Project"*. Browse to its *Settings* and witness how this project uses the default values. You can duplicate the current view by clicking the "*" button on the upper-right corner, and use the new view to show both projects settings side to side:

Now is the time to realise something crucial about Kabaret's Flow: The thing you are modeling is not the project itself but the *schema* of your projects. In fact, your projects are *instances* of your flow. It is a complete different approach than connecting nodes in Nuke or in Maya where you define a graph that is used as a dataflow. Here we are defining a graph that generates the graph that will (or *may*) be used as a dataflow. Each instance of your flow has its own set of values, but the structure is shared. If you comment the *framerate* relation in your *ProjectSettings* class and reload your Project Definitions (on both views), you will see that neither *MyFirstProject* not *MySecondProject* contains a *framerate* field anymore. Another nice feature is that if you un-comment this line and reload, both projects will have their previous value back. And maybe the nicest part is that you did all this without having to worry about how to

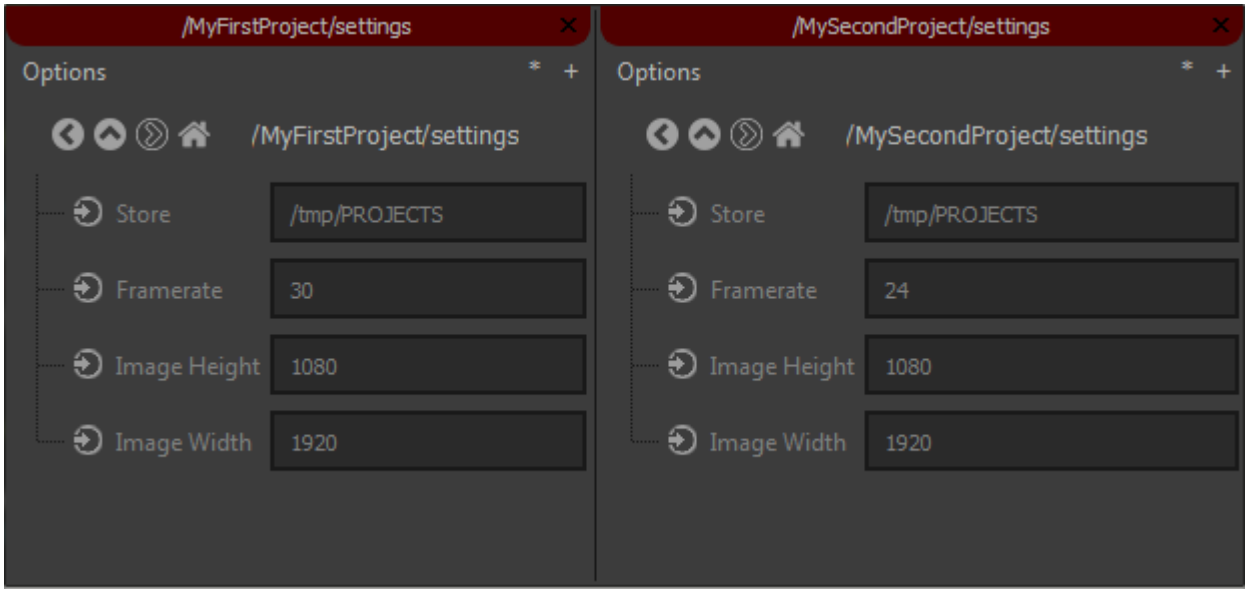


Fig. 5: Hmm... Should you be impressed ?

store the values, and without enduring some migration process to alter the schema of your data. Welcome to the 21st Century ! ;)

Defining the flow instead of the actual project is something borrowed from the "Workflow" world. It has many advantages among which the fact that when you add something, for exemple a batch process between two tasks of a shot, everything is updated at once: all shots will contain this process without the need to update existing graphs or trigger some dark-magic synchronisation machinery.

Now let's forget about *MySecondProject* and focus on building something more interesting.

Using a Map

We've seen how we are defining a structure instead of a concrete project. But all our projects won't have the exact same structure. In our case - a simple shot manager, the list of shots will need to be different from project to project. We can't just use something like:

```
class Project(flow.Object):

    shot001 = flow.Child(Shot)
    shot002 = flow.Child(Shot)
    shot003 = flow.Child(Shot)
    shot004 = flow.Child(Shot)
    shot005 = flow.Child(Shot)
```

That's the reason for the *kabaret.flow.Map* to exist: It provides a per-instance list of things. Let's see how by implementing a few methods on our *Shots* class:

```
4  class Shot(flow.Object):
5
6      first = flow.IntParam(1)
7      last = flow.IntParam(100)
8
9  class Shots(flow.Map):
```

(continues on next page)

(continued from previous page)

```

    @classmethod
    def mapped_type(cls):
        return Shot

    def columns(self):
        return ['Name', 'Ranges']

    def _fill_row_cells(self, row, item):
        row['Name'] = item.name()
        row['Ranges'] = '{}->{}'.format(item.first.get(), item.last.get())
```

We added the *Shot* class definition. It's a simple object with two Params. We also implemented the *Shots' mapped_type* classmethod to return our *Shot* class. This tells the flow that all objects contained in the *Shots* map will be *Shot* objects (or subclasses of *Shot*).

We have also overridden the default implementation of the *columns()* and *_fill_row_cells()* methods. Both are used to configure the information displayed in the views. The *Shots* map will now list the name and the ranges of each shot it contains.

You can notice how *_fill_row_cells()* gets the value from the *Shot* item it receives. We know the *item* argument is a *Shot* because we configured the Map for that. The *Shot* class has a *first* and a *last* relation, so every *Shot* instance has a *first* and a *last* attribute containing the respective Value Object. Every Value in the flow has a *get()* method that returns the data it holds.

Next step is to add some shots in the *Shots* map, and we will need user input for that.

Using Actions

Pipeline is not all about metadata, it is about executing code too. Sometimes the code has to be triggered by some event, sometimes it is up to the user to trigger it. The *kabaret.flow.Action* Object is meant for this second case.

An Action will show up in the GUI as a button and/or as an entry in a menu. By clicking it, the user tells the action to show a dialog if needed, and then execute its *run()* method. Let's add an Action that creates a *Shot* in our *Shots* Map:

```
class AddShotAction(flow.Action):

    def get_buttons(self):
        return ['Create Shot', 'Cancel']

    def run(self, button):
        if button == 'Cancel':
            return
        # create the shot here
```

You create your own Action by extending the *flow.Action* class. The *get_buttons()* method can be implemented to return the list of buttons available in the Action's dialog. When the user clicks one of those buttons, the *run()* method is called with the name of the clicked button. Our *run()* implementation checks that the clicked button was not "Cancel" before doing anything.

In order for this Action to be used by our flow, we need to add it as a relation somewhere. As it will act on the *Shots* map, let's make it a Child there:

```
class Shots(flow.Map):

    add_shot = flow.Child(AddShotAction)
```

(continues on next page)

(continued from previous page)

```
    ...
```

If you reload your project definitions you will see a new menu button on the right of the *Shots* label. This menu contains the *"Add Shot"* entry. Clicking it will show a dialog with the "Create Shot" and "Cancel" buttons.

Now let's implement the *run()* method to actually create a *Shot*. The *flow.Map* Object has an *add()* method accepting a string as the name of the item to add. We will need to call it with a user input value. This input will be handled by a Param in the *AddShotAction*. And, as the Action is a Child of the *Shots* Map, we will use a *Parent* relation to access the *Shots* from within the *AddShotAction*:

```
class AddShotAction(flow.Action):

    # The leading _ tells the GUI that this relation is protected
    # and should not be shown:
    _shots = flow.Parent()

    # Params will show up in the Action dialog:
    shot_name = flow.Param('shot000')
    first_frame = flow.IntParam(1)
    last_frame = flow.IntParam(100)


    def get_buttons(self):
        return ['Create Shot', 'Cancel']

    def run(self, button):
        if button == 'Cancel':
            return

        # Real life scenario should validate this value:
        shot_name = self.shot_name.get().strip()

        # Create the shot using our Parent() relation:
        shot = self._shots.add(shot_name)

        # Configure the shot with requested values:
        shot.first.set(self.first_frame.get())
        shot.last.set(self.last_frame.get())

        # Tell everyone that the Shots list has changed
        # and should be reloaded:
        self._shots.touch()
```

If you reload your project definitions you will now be able to create some shots and even configure them on the fly. A Shot can be browsed by double-clicking on it. CTRL+DoubleClick will open it in a new view.

Computed Value

We discussed earlier the fact that the *kabaret.flow* borrows ideas from the Workflow principles to overcome issues arising when using dataflow to represent a project pipeline. But there's still great power to harvest in dataflow, especially in lazy evaluation dataflow (a.k.a pull dataflow). At some point you will probably want to manage some Value holding the result of a computation using other Values. And this computation should occur when needed (when a dependent Value changes for exemple.)

We will showcase such a need by adding a *length* Value in our *Shot* class. This Value is computed using the *first* and *last* Params, and is updated every time one of them changes.

This is done using the Computed Relation. This Relation defines a Child ComputedValue in the parent Object. The parent Object is responsible for the computation of this Value.

Let's add a *length* Computed relation to our *Shot* class, along with an implementation of is *compute_child_value()* method:

```
class Shot(flow.Object):

    first = flow.IntParam(1)
    last = flow.IntParam(100)
    length = flow.Computed()

    def compute_child_value(self, child_value):
        '''
        Called when a ComputedValue needs to deliver its result.
        '''
        if child_value is self.length:
            self.length.set(
                self.last.get()-self.first.get()+1
            )
```

If you reload your project definitions, you will see the new *"Length"* field in every *Shot*. And it has the correct value, great !

But if you change the *first* or the *last* value of the *Shot*, the *length* does not update. Let's fix this by specifying that we want to react to changes on *first* and *last*. This is done by configuring their relation and implementing *child_value_changed()*:

```
class Shot(flow.Object):

    first = flow.IntParam(1).watched()
    last = flow.IntParam(100).watched()
    length = flow.Computed()

    def child_value_changed(self, child_value):
        '''
        Called when a watched child Value has changed.
        '''
        if child_value in (self.first, self.last):
            # We invalidate self.length whenever self.first or self.last
            # changes:
            self.length.touch()

    def compute_child_value(self, child_value):
        '''
        Called when a ComputedValue needs to deliver its result.
        '''
        if child_value is self.length:
            self.length.set(
                self.last.get()-self.first.get()+1
            )
```

If you reload your project definitions you will notice that the *Length* fields shows an updated value whenever you change the value of the *First* or *Last* field.

Divide, Compose and Conquer

Another advantage of *kabaret.flow* is how it helps you divide your pipelines into components and layers.

A classical dataflow let you encapsulate logic into nodes and connect them together. This is a great way to isolate concerns into components, which is easier to develop, test and manage.

With *kabaret.flow* you can go even further by defining Objects composed of other Objects. It's like having a dataflow inside each node of your dataflow. It gives you the ability to not only isolate concerns into components, but also to encapsulate and compose them into new components. You're actually layering responsibilities, which is known as the good architecture when building complex software.

We are going to illustrate this by adding some content to our *Shot* Object.

Let's say that a *Shot* is composed of consecutive tasks, and that a task has a status and a file that contains the work done for that task. It's an oversimplified case for a CG Pipeline, but it already contains two clearly separable concerns: file naming conventions (Persistence Layer), and task status (Business Layer).

We're going to lay down the structure for that:

```
class File(flow.Object):

    pass


class Task(flow.Object):

    status = flow.Param()
    scene = flow.Child(File)


class Shot(flow.Object):

    first = flow.IntParam(1).watched()
    last = flow.IntParam(100).watched()
    length = flow.Computed()

    anim = flow.Child(Task)
    lighting = flow.Child(Task)
    comp = flow.Child(Task)

    ...
```

We've added some tasks to the *Shot*. Each *Task* has a *status* Value and a *scene* object which is a *File*.

Reload and you will discover a whole hierarchy in *MyFirstProject*. This hierarchy is your Workflow/Pipeline (Business Layer). It is encapsulated in the *Project* component. This component uses the *Task* component, which is also in the Business Layer. The *Task* uses the *File* component which is in the Persistence Layer.

Now you that we've clearly separated concerns, we can implement their functionalities and behavior.

The purpose of the *File* is to provide a filename. This filename is not to be edited by the end user. It must be provided by an authority responsible of applying naming conventions. Let's implement a very simple strategy which consists of a single function that turn some parameters into a filename. This function will be used by a *ComputedValue* in the *File*:

```
import os

...
```

(continues on next page)

(continued from previous page)

```
class File(flow.Object)

    task = flow.Parent()
    filename = flow.Computed()

    def get_filename(self):
        project = self.root().project()
        store = project.settings.store.get()
        task_name = self.task.name()
        name = self.name()
        ext = '.ma'
        return os.path.join(store, project.name(), task_name, name)+ext

    def compute_child_value(self, child_value):
        if child_value is self.filename:
            self.filename.set(self.get_filename())
```

Note: Here we are using self.root().project() to access the project settings. This is a nice alternative to using many *Parent()* relations. If you are wondering why not using something like a *Project()* relation, I'd say you have a point ! We can discuss it in the discord channel :}

After reloading your project definitions you will see how the filename of each *File* has its own value, and this value comes from a well-isolated functional policy.

The purpose of the *File* is to be edited, so let's add an Editor \o/

```
class EditAction(flow.Action):

    _file = flow.Parent()

    def get_buttons(self):
        self.message.set('<h2>Select an Editor</h2>')
        return ['Open with Maya', 'Open in Text Editor']

    def run(self, button):
        # Here we would select an executable depending on the button
        # or the file extension or anything really,
        # and use subprocess to run the editor.
        print('Editing the file:', self._file.filename.get())


class File(flow.Object):

    task = flow.Parent()
    filename = flow.Computed()

    edit = flow.Child(EditAction)


...
```

In this oversimplified example the *File* is used only in the *Task*, but in real life you'd probably use it in many other situations. Defining it as a component let you later extend it with functionalities like the *EditAction* or other *Persistance Layer* features like version management...

Now let's focus on the *Task* to see another example of concern isolation: the *Status*.

Worflows and Pipelines are all about *Statuses*. *Statuses* often contain the information used to trigger automations, reporting, etc. And they need to have value among a defined list of possibilities. Let's add this to our awesome flow !

First, we're going to use a ChoiceValue as we want to restrict the possible values:

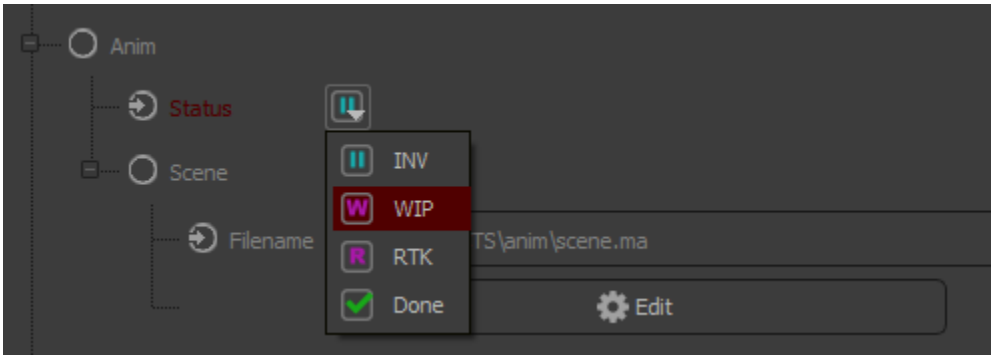
```
class TaskStatus(flow.values.ChoiceValue):

    CHOICES = ['INV', 'WIP', 'RTK', 'Done']


class Task(flow.Object):

    status = flow.Param('INV', TaskStatus)
    scene = flow.Child(File)
```

This changes the GUI representation of the *Task*'s *status* field to a drop down menu:



Note: Kabaret provides default icons for many situations and this is what you see here. This is managed by the *kabaret.app.resources* module and you can override and/or extend the icons as much as you want.

Second, we want to trigger something when the *Status* changes. That's what *Statuses* are meant for. But the particular details of what should be triggered depend on what the *Status* is bound to, so we are going to delegate it to the parent *Task*:

```
class Task(flow.Object):

    status = flow.Param('INV', TaskStatus).watched()
    scene = flow.Child(File)

    def child_value_changed(self, child_value):
        if child_value is self.status:
            self.send_mail_notification()

    def send_mail_notification(self):
        # If this was a real task there would be an assignee
        # that we could send a mail to...
        print(
            'Mailing to santa: status of Task {!r} is now {!r}'.format(
                self.oid(), self.status.get()
            )
        )
```

And last, we want to report the *Tasks* statuses into the *Shot*. This status depends on each *Task*'s status and should be computed every time a *Task* status changes. In order to achieve this, we will have the *Tasks* asking their *Shot* to update their *status* when needed:

```
class Task(flow.Object):

    shot = flow.Parent()
    status = flow.Param('INV', TaskStatus).watched()
    scene = flow.Child(File)

    def child_value_changed(self, child_value):
        if child_value is self.status:
            self.send_mail_notification()
            self.shot.update_status()

    def send_mail_notification(self):
        # If this was a real task there would be an assignee
        # that we could send a mail to...
        print(
            'Mailing to santa: status of Task {!r} is now {!r}'.format(
                self.oid(), self.status.get()
            )
        )

class Shot(flow.Object):

    first = flow.IntParam(1).watched()
    last = flow.IntParam(100).watched()
    length = flow.Computed()

    anim = flow.Child(Task)
    lighting = flow.Child(Task)
    comp = flow.Child(Task)

    status = flow.Param('NYS').ui(editable=False)

    def update_status(self):
        status = 'WIP'
        statuses = set([
            task.status.get()
            for task in (self.anim, self.lighting, self.comp)
        ])
        if len(statuses) == 1:
            status = statuses.pop()

        self.status.set(status)

    ...
```

We could have used a *Computed* relation for the Shot's *status* but this time we chose a simple *Param* (that the user cannot edit) and we update its *Value* directly when a *Task status* changes. The best strategy to use will depend on your case and your fondness...

There's a last thing you may want to do: Give a hint of the *Shot*'s status in the *Shots* list. That requires two lines to add to the *Shots* class:

```
class Shots(flow.Map):
```

(continues on next page)

(continued from previous page)

```
    add_shot = flow.Child(AddShotAction)

    @classmethod
    def mapped_type(cls):
        return Shot

    def columns(self):
        return ['Name', 'Ranges']

    def _fill_row_cells(self, row, item):
        row['Name'] = item.name()
        row['Ranges'] = '{}->{}'.format(item.first.get(), item.last.get())

    def _fill_row_style(self, style, item, row):
        style['icon'] = ('icons.status', item.status.get())
```

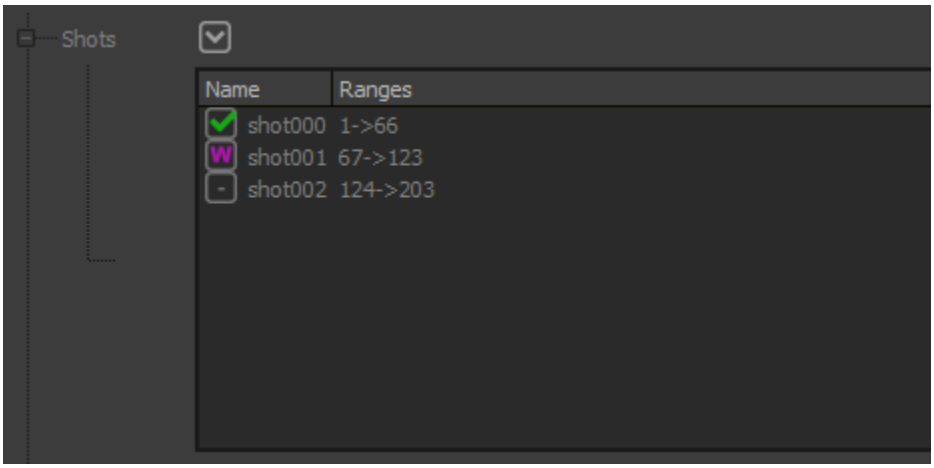


Fig. 6: Wooh ! Icons \o/

Conclusion

We've seen that the concept of extending *Objects* with *Relations* to other *Objects* is pretty simple to understand and to use. And it can efficiently build almost anything using only *Params*, *Actions* and *Maps*.

There's more in the toolbox, like *Refs* which are *Values* pointing to other *Objects*, *ConnectActions* that let you react to Drag'N'Drop in the GUI, *Relation* configuration that let you control their GUI representation, etc.

kabaret.flow has been used in small projects like commercials with ~10 shots and a team of ~10 artists, as well as feature movies with hundreds of shots and complex production tracking tools.

What are you going to use if for ? :D

Here is what we built in less than 150 lines of simple code:

Of course, in real life, pipeline management is about doing quick and dirty stuff. That's the cool thing about *kabaret.flow*: you can do robust and well-prepared things, but it's not mandatory and you can also do bad things whenever you want/need. We'll assume you don't need any tutorial for that ^.^

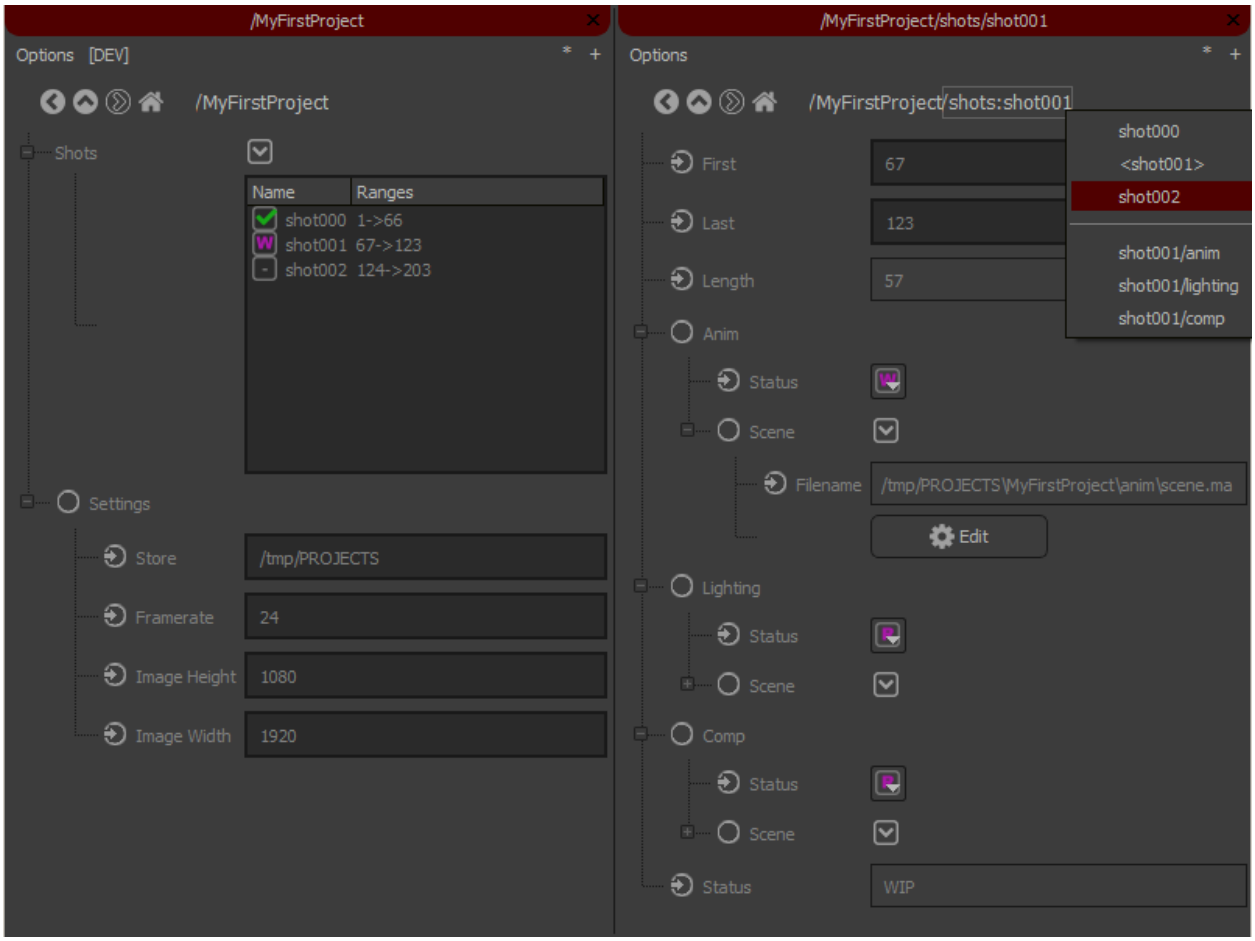


Fig. 7: (the menu you see in the upper-right corner is popped up by a RMB on the page path)

Final Code

```python
from __future__ import print_function

import os

from kabaret import flow


class EditAction(flow.Action):

    _file = flow.Parent()

    def get_buttons(self):
        self.message.set('<h2>Select an Editor</h2>')
        return ['Maya', 'Sublime']

    def run(self, button):
        # Here we would select an executable depending on the button
        # or the file extension or anything really...
        # and use subprocess to run the editor.
        print('Editing the file:', self._file.filename.get())


class File(flow.Object):

    task = flow.Parent()
    filename = flow.Computed()
    edit = flow.Child(EditAction)

    def get_filename(self):
        project = self.root().project()
        store = project.settings.store.get()
        task_name = self.task.name()
        name = self.name()
        ext = '.ma'
        return os.path.join(store, project.name(), task_name, name)+ext

    def compute_child_value(self, child_value):
        if child_value is self.filename:
            self.filename.set(self.get_filename())


class TaskStatus(flow.values.ChoiceValue):

    CHOICES = ['INV', 'WIP', 'RTK', 'Done']


class Task(flow.Object):

    shot = flow.Parent()
    status = flow.Param('INV', TaskStatus).watched()
    scene = flow.Child(File)

    def child_value_changed(self, child_value):
        if child_value is self.status:
            self.send_mail_notification()
```

(continues on next page)

(continued from previous page)

```
            self.shot.update_status()

    def send_mail_notification(self):
        # If this was a real task there would be an assignee
        # that we could send a mail to...
        print(
            'Mailing to santa: status of Task {!r} is now {!r}'.format(
                self.oid(), self.status.get()
            )
        )

class Shot(flow.Object):

    first = flow.IntParam(1).watched()
    last = flow.IntParam(100).watched()
    length = flow.Computed()

    anim = flow.Child(Task)
    lighting = flow.Child(Task)
    comp = flow.Child(Task)

    status = flow.Param('NYS').ui(editable=False)

    def update_status(self):
        status = 'WIP'
        statuses = set([
            task.status.get()
            for task in (self.anim, self.lighting, self.comp)
        ])
        if len(statuses) == 1:
            status = statuses.pop()

        self.status.set(status)

    def child_value_changed(self, child_value):
        '''
        Called when a watched child Value has changed.
        '''
        if child_value in (self.first, self.last):
            # We invalidate self.length whenever self.first or self.last
            # changes:
            self.length.touch()

    def compute_child_value(self, child_value):
        '''
        Called when a ComputedValue needs to deliver its result.
        '''
        if child_value is self.length:
            self.length.set(
                self.last.get()-self.first.get()+1
            )

class AddShotAction(flow.Action):

    _shots = flow.Parent()

    shot_name = flow.Param('shot000')
```

(continues on next page)

(continued from previous page)

```
    first_frame = flow.IntParam(1)
    last_frame = flow.IntParam(100)

    def get_buttons(self):
        return ['Create Shot', 'Cancel']

    def run(self, button):
        if button == 'Cancel':
            return

        # Real life scenario should validate this value:
        shot_name = self.shot_name.get().strip()

        # Create the shot using our Parent() relation:
        shot = self._shots.add(shot_name)

        # Configure the shot with requested values:
        shot.first.set(self.first_frame.get())
        shot.last.set(self.last_frame.get())

        # Tell everyone that the Shots list has changed
        # and should be reloaded:
        self._shots.touch()


class Shots(flow.Map):

    add_shot = flow.Child(AddShotAction)

    @classmethod
    def mapped_type(cls):
        return Shot

    def columns(self):
        return ['Name', 'Ranges']

    def _fill_row_cells(self, row, item):
        row['Name'] = item.name()
        row['Ranges'] = '{}->{}'.format(item.first.get(), item.last.get())

    def _fill_row_style(self, style, item, row):
        style['icon'] = ('icons.status', item.status.get())

class ProjectSettings(flow.Object):

    store = flow.Param('/tmp/PROJECTS')
    framerate = flow.IntParam(24)
    image_height = flow.IntParam(1080)
    image_width = flow.IntParam(1920)


class Project(flow.Object):

    shots = flow.Child(Shots)
    settings = flow.Child(ProjectSettings)

```

2.3 Usage

Soon !..

2.4 Guru

2.4.1 Injection / Inversion of Control

Why ?

Hi-level Objects often need to relate each others.

When building a self contained flow, this is not an issue. But while building flow libraries, you face a situation where you have to deliver Objects for every related high level concepts so that they can interact thru their relations.

The situation is also bad for the library users who will need to adopt all the concepts of the library because they can't change the relation between them.

As an example, let's consider you want to build a Task system:

You use a *flow.Map* to manage a list of *Task* Objects. A *Task* has *start_date* and *duration*, as well as a *assigned_user* which is a *Connection* to a *User*. At this point you need to define what is a *User* and the interface it needs to interact with your *Task*. So you had a *Team* map containing *User* items... You Task system now contains Tasks and Users and in order to use it in a pipeline flow one needs to adopt your Task system as well as your User system. **This sucks !!!**

On this example, an **Inversion of Control** is to give the pipeline flow the control over the relation used by the Task lib flow.

A common way to achieve this is to *inject* the *dependencies* inside the Task flow lib from the pipeline flow instead of letting the Task flow lib define them.

Why is it a problem ?

A flow Object dependencies are defined by the *Child* and *Map* contained in all its children. (*Param* are *Child*, but *Parent* and *Relative* are special and dont apply here).

More precisely, they are defined by the type of Objects each *Relation* relate to and the type of item mapped to each *Map*.

The classic way to change those (in order to change the dependencies) is by inheritance: you inherit the relation owner and override the relation by defining a new one with the same name.

The probleme is that you now need to use this new class instead of the one containing the Relation you altered. So you also need to inherit its parent's class. And this goes up to your Project class !

This sucks !!!

But keep your pants on and read on, we have a **solution** for this situation: **Dependency Injection**

Also, it's worth specifying that dependency injection also makes writing test code extremely easier as it lets you replace/mock-up part of the system to isolate the component you want to test.

How ?

So we want to *"override"* the **related type** of some *Relation*, and/or the **mapped type** of some *Maps*.

We want to do it with different types on different flow (be them pipeline flows or lib flows) because we might use the same lib in different contexts where our dependencies differ.

As alwawy, we want to do from within the flow itsef so that everything is well tight together.

Also, as a lib flow developer, we want to specify which types can be **injected** (*"overidden"*) in my lib.

The first step is specifying that a *Relation* supports injection. This is done by calling *injectable()* on it.

In this example, the *my_studio.flow.lib.foo* lib flow defines a *Foo* Object that contains a *FooChild* Object. Let's say you want the lib users to be able to inject their Object intead of using your 'FooChild':

```
# my_studio/flow/lib/foo.py`

class Foo(flow.Object):
    my_child = flow.Child(FooChild).injectable()
```

The second step is to use the foo lib and inject a new type instead of *FooChild*.

This is done by defining an *_injection_provider()* classmethod in any parent of the *Foo* Object. You can conveniently inherit *kabaret.flow.InjectionProvider* to do so, but it is not mandatory.

In this example, we inject a *FooChild* subclass that extends it with the *is_awesome* Param.

```
from my_studio.flow.lib.foo import Foo, FooChild

class MySuperFooChild(FooChild):
    is_super = flow.BoolParam(True)

class Project(flow.Object, flow.InjectionProvider):

    foo = flow.Child(Foo)

    def _injection_provider(self, slot_name, default_type):
        if default_type is FooChild:
            return MySuperFooChild
```

With this set up, your *Project* flow is using the *foo* lib with your very own FooChild.

This rocks \o/ !!!

More example

There are comprehensive examples in *dev_studio.flow.unittest_project.showcase.injection*.

Create a project with the type *dev_studio.flow.unittest_project.UnittestProject* to see the result, and read the code to learn more !

2.5 Installation

kabaret will run with most python versions (2.7+, 3.3+).

You will need **pip** to install kabaret. Recent versions of python have it pre-installed, you can test its availability by importing it:

```
import pip
```

If nothing happens, you're good to go. But if an *ImportError* is raised, you will need to install **pip**. Download the file get-pip.py then run the following:

```
python get-pip.py
```

That's it, **pip** is now installed. If you want to know more about pip you can read its documentation

As any other package, **kabaret** can be installed in your python's site-package and then used after a simple "*import kabaret*".

This is convenient if you have administrator privileges on your python installation, and if you plan on using kabaret as a standard python package.

Chances are that you will need more than that though:

- Installing kabaret at work for personal use may raise access privileges issues.
- Using kabaret embedded in Blender, Maya or any other extended python interpreter raises even more questions. (Do they even support pip ?)
- Keeping installation up to date on every station is not a fun task.

The *Shared* installation is often the choice to go unless you're just testing.

2.5.1 Local

The local installation is the most straightforward and can be used to discover Kabaret.

```
python -m pip install -U kabaret
```

If you don't have a Qt wrapper installed, you can install PySide2 (or any other one available for your python version):

```
python -m pip install PySide2
```

You can now follow the first tutorial.

2.5.2 Shared

A shared installation puts **kabaret** and all its dependencies in a folder of your choice (most probably one shared across all workstations).

```
mkdir ./KABARET_INSTALL
pip install --install-option="--prefix=./KABARET_INSTALL" -U kabaret
```

Depending on your setup, you may want to install a Qt wrapper there too:

```
pip install --install-option="--prefix=./KABARET_INSTALL" PySide2
```

In order to use this installation, you will need to either configure your PYTHONPATH environment variable:

```
set PYTHONPATH=$PYTHONPATH:/path/to/KABARET_INSTALL
```

Or manage your sys.path before importing kabaret in python

```
import sys
sys.path.append(path_to_kabaret_install)
```

If you use kabaret with several python interpreters (Nuke, Maya, Houdini...), you should create a separate shared installation for each one to avoid issues regarding compiled bytecode.

2.5.3 Dev

Clone the repo, follow instructions in cmds/README.txt (might be outdated).

Don't forget to join us on the discord, that's where the help is !

2.5.4 Dependencies

Automatic

When installing kabaret with pip, those packages will automatically be installed as dependecies:

- redis
- qtpy
- six

Manual

You will need a pre-installed Qt Wrapper: PyQt4, PyQt5, PySide or PySide2. If you don't know which one you to use, you can go with PySide2:

```
python -m pip install PySide2
```

Warning: There is a bug in *PySide2=5.15.1* which crashes python when sorting QTreeWidget items. Use *PySide2=5.15.0* or *PyQt5* instead.

Extra

Kabaret uses various functionalities of redis. You can download one from this page (windows users can download here) and start a server with the default configuration, it will be far enough for testing.

You can run redis in Docker, see https://hub.docker.com/_/redis (be sure to start the image with persistent storage ;)).

You can also get a free online redis instance at redislab.com.

When deploying kabaret into production, we recommand getting acquainted with redis management, most notably with the persistence options. The bare minimum is probably to ensure your server dumps to a file that you backup every now and then.

2.6 Flow Reference Guide

Warning: Documentation in Progress...

2.6.1 Exceptions

2.6.2 Object

2.6.3 Relations

2.6.4 Values

2.6.5 Actions

2.6.6 Maps

2.6.7 Injection

2.7 App Reference Guide

Note: This documentation in still in progress. Your feedback is welcome :)

The **kabaret** framework is a modular system that will help you create standalone or embedded applications.

When building a application, you will start by creating a `kabaret.app.session.KabaretSession` (or more likely, one of its subclasses).

A **Session** contains some **Actors** that each provide specific functionnalities and **commands** to use them.

A **Session** containts some **Views** that use the **commands** to display and modify the informations managed by **Actors**.

A **Session** manages the **communication** between other sessions on the network, deals with **UI** (events, layouts, ...) and **logging**.

Frameword devellopers can extend **kabaret** by providing new **Actors** and/or **Views**. When a User wants to use an extension, he must subclass a **Session** to register the additionnal **Actors / Views**.

Once you session is properly configured, you can use one or more instances of it in a single python interpretor. But you can't share sessions between threads.

You can define multiple session types: a standalone GUI for User, another for administration, a headless one acting as a worker waiting for orders, ...

You can navigate to the Create My Studio tutorial to get you started.

2.7.1 Session

2.7.2 Resources

2.8 Featured Extensions

Kabaret delivers an extensible architecture so that whatever is the scope of a missing feature, one can implement it without modifying Kabaret's code, and package the result to share it with the community.

Here is a list of extensions we like. If you want to be listed here, please contact us on the development forum.

2.8.1 Script View

The **kabaret.script_view** package is a GUI extension that give the user the ability to edit and execute python scripts as if they were methods of a selected flow object.

A geek tool for the mass, a must have for the Geeks :D

2.8.2 InGrid Objects and View

The **kabaret.ingrid** package is a Flow and GUI extension. It contains Flow objects that let you configure a *Grid* of Flow object, and a new View to visualise and edit them.

The users will be able to drop Object on InGrid views to discover and load configurations, as well as use Action in the Flow that opens new InGrid views with a specific configuration.

2.8.3 Subprocess Manager Actor

The **kabaret.subprocess_manager** package is an extension with an Actor and GUI to start and watch subprocesses.

Examples are provided in the documentation to configure and start subprocess from a Flow object.

2.8.4 Gantt Objects and View

Note: This is a work in progress and may not be available at the time you read it.

The **kabaret.gantt** package is a Flow, Session and GUI extension. It contains Flow objects holding time-related parameters to help you build time related entities, an Actor providing commands to manipulate those objects from the GUI, and a Gantt view to visualise and edit them.

2.8.5 Users Actor

Note: This is a work in progress and may not be available at the time you read it.

The **kabaret.users** package is a Session extension with an Actor to manage teams of users and their preferences.

Note that there's a discussion going on about whether this is the job of an Actor or for some Flow object. Depending on the outcome, this actor may not make it to the public...

2.8.6 Naming

kabaret.naming

This package is actually not an extension as it is completely independent from kabaret. Developed at SupamonkS Studio, it was used before kabaret existed.

Somewhat obsolete if you have the chance to use Kabaret, we wanted to share it anyway because many people found it awesome :D

Here is the package on PyPI

2.9 Plugins

2.9.1 What's a Plugin ?

A plugin is a kabaret extension which:

- Is packaged to an index (like PyPI).
- Is automatically active in all Kabaret sessions.
- Reports what it provides.
- Is listed in the Plugin view.
- Can be deactivated using an environment variable.

But most importantly, a plugin is a kabaret extension that the user can install and start using by issuing a single command:

```
pip install my_kabaret_extension
```

2.9.2 Plugin Creation

To create a plugin you just need to implement some of the supported "hooks" using the *kabaret.app.plugin* decorator.

Your hooks implementation must be contained in a module, a static class, or an instance (anything that can be treated as a namespace in fact).

Here is a simple example using a module namespace, all hooks defined here will form the "my_stuff" plugin:

```
'''
Inside "my_stuff.py", this is the "my_stuff" plugin.
'''

from pyqt import QtCore

from kabaret.app import plugin
from .my_stuff.my_view import MyView

@plugin
def install_views(session):
    if not session.is_gui():
        return

    type_name = session.register_view_type(MyView)
```

(continues on next page)

(continued from previous page)

```
    session.add_view(
        type_name,
        hidden=False,
        area=QtCore.Qt.RightDockWidgetArea
    )
```

And here is an example using classes in order to define too plugins in the same module:

```
'''
Inside "best_plugins.py", there is 2 plugins defined.
'''
from pyqt import QtCore

from kabaret.app import plugin
from .best_view_ever import BestViewEver
from .best_pipe_ever import BestPipelineEver


class BestView:
    """This is the `BestView` plugin"""

    @plugin
    def install_views(session):
        if not session.is_gui():
            return

        type_name = session.register_view_type(MyView)
        session.add_view(
            type_name,
            hidden=False,
            area=QtCore.Qt.RightDockWidgetArea
        )

class BestPipe:
    """This is the "BestPipe" plugin"""

    @plugin
    def get_project_types(session):
        return [BestPipelineEver]
```

Pluggable Hooks

Here is the exhaustive list of plugin hooks.

Note that you must respect the given signature.

- ***install_actors(session)*** This let your plugin install some *Actor* in the session.
- ***install_views(session)*** This let your plugin install some *View* in the session.
- ***install_resources(session)*** Kabaret resources don't have an installation procedure since a simple import is enough. But doing you resources import in this hook ensures that it will be done before other hooks get triggered.
- ***install_editors(session)*** The editor factory is static so you can install your editors with a simple import, but using this hook will ensures you that the editors are registered before any view is created.

- ***get_project_types(session)*** Here your plugin can return a list of *kabaret.flow.Object* that are ment to be used as Project structure. Some other extension may use this information to present a list of available project types to the user.
- ***get_flow_objects(session)*** Here your plugin can return a list of *kabaret.flow.Object* that provide packaged features. This is purely informative since you will choose to use them or not in your flow. But this has the advantage of listing in the Plugin View the *Injection points* defined in those objects.

2.9.3 Plugin Activation

Plugins are activated by package distribution entry points in the "kabaret.plugin" group.

To activate the 3 plugins defined in the examples above, your *setup.py* would typically look like:

```
# Inside your extension's "setup.py"

from setuptools import setup, find_packages

setup(
    name="my_extension_name",
    ...
    entry_points = {
        "kabaret.plugin":[
            "my_extension_name.my_stuff = my_extension_name.my_stuff",
            "my_extension_name.best_view = my_extension_name.best_plugins:BestView",
            "my_extension_name.best_pipe = my_extension_name.best_plugins:BestPipe",
        ],
    }
```

2.9.4 Plugin Manual Activation

If some of your extensions are not packaged and installable via pip, you won't get a chance to define entry points.

In this case, creating a custom session class will give you the opportunity to register you plugins programatically. Just override the *register_plugins()* method and use the provided *plugin_manager* to register your plugin modules/classes:

```
# In "run_my_standalone_kabaret.py", a script launching A
# standalone kabaret application

from kabaret.app.ui import gui

from my_extension_name import my_stuff
from my_extension_name.best_plugins import BestView, BestPipe

class MyGUISession(gui.KabaretStandaloneGUISession):

    def register_plugins(self, plugin_manager):
        plugin_manager.register(my_stuff)
        plugin_manager.register(BestView)
        plugin_manager.register(BestPipe)
```

That being said, packaging your code is a good thing for many reasons and you should consider doing it ;)

2.9.5 Plugin Desactivation

All installed plugins are active by default, but sometime you will want to block some of them. This is done with the *KABARET_BLOCKED_PLUGINS* environment variable.

Using the example above, you can desactivate the "my_stuff" and the "BestView" plugins like this:

```
export KABARET_BLOCKED_PLUGINS="my_stuff BestView"
python run_my_standalone_kabaret
```

Note that this is not intended to be used as a "plugin list management" but rather for debuging and corner cases. If you want to manage different sets of plugins, you should use different virtualenvs. They are designed for this and with the *–editable* option of the *pip install* command, you will have the best control and versatility over your plugins installation.

2.9.6 Plugin Order

You will sometimes need a plugin to act depending on other plugins.

A classic example would be a "default plugin" that would install a View type only if no other plugin already did.

A simple approach is to test for the view type name being registered, but this is not enough until you can be sure that this plugin will be called *after* any other plugin wanting to install this view type.

To affect the plugin call order and acheive this, you must configure your plugin with the *trylast* option. All "trylast" plugins are guaranted to be called after plugin without it.

```
from kabaret.app import plugin
from my_custom_stuff import FlowView

@plugin(trylast=True)
def install_views(session):
    if session.has_view_type('Flow'):
        # some other plugin took care
        # of this, let's bail out.
        return

    # The Flow view is mandatory, let's
    # install it and create one:
    session.register_view_type(FlowView)
    session.add_view(type_name)
```

Similarly, you can use the *tryfirst* option to ensure a plugin is called before any plugin without the *tryfist* option.

2.10 FAQ & Fun Facts

- **Where does the name 'Kabaret' come from?** Kabaret was initially developed at Supamonks Studio. Supamonks' in-house software and tools are named with girls first names like:
  - Becassine - she let you bake an animation scene ('bake a scene')
  - Rebeka - she let you bake in batch (Re-Bake)
  - Trinity - she stores tree shaped data on disk.
  - Naomie - she deals the naming conventions
  - Debby - she's the interface to the DataBase

  - Etc.

  We decided not to push those names to the open source version but we wanted to keep a hint of girl power :)

  Also, we like the French Touch it gives.

- **Where does the Kabaret logo come from?** The logo draws a strike-through 'K' symbol.

  It's based on a phonetic pun in French.

  In French "a kabaret" is "un cabaret". It is phonetically the same as "un K barré" which in English means "a strike through 'K'".

  It so happens that a strikeout 'K' is the symbol of the Laos money, the Loa Kip. Laos is a communist society where the concept of possession is unknown and the word for "mine" is the same as the word for "yours". This is a great match with our open source spirit.

  We enjoy the idea that branding the Loa Kip could put Laos in focus and help this great country.

  Laos is the most heavily bombed country in the world. They need your help and you should donate to help with the Lao UXO eradication efforts.

- **Where do I get support?**

  You can join the discord channel here:

  - Kabaret Studio

2.11 Credits

2.11.1 Authors

First versions of Kabaret were conceived and implemented between 2012 and 2015 at SupamonkS Studio, Paris.

The primary authors are (and/or have been):

- Damien 'Dee' Coureau
- Sebastien 'Zwib' Ho
- Valerian Prevost
- Ivans Saponenko
- Steeve 'Firegreen' Vincent

We accumulate experience as Artists, Technical Directors, Developer and Production Director on hundreds of commercial spots and commercial series, as well as on VFX and Full CG features movies like Blueberry, Splice, Irreversible, Despicable Me, The Lorax…

We have a deep faith in the open source philosophy and we wish every CG Talent could focus on the beauty of their work (may it be cost tracking, pixel enhancement, or code magnificence) instead of struggling with the machine. **Kabaret** is our contributions to make this dream get real.

We hope you'll join us in this adventure.

2.11.2 Mentors

Many ideas in Kabaret come from the outstanding people we had the chance to meet or work with.

Most notably:

- Etienne 'Chex' Pecheux, on the dataflow and automation.
- Albert 'Lobo' Bonnefous, on the overall CG world.
- Pierrick 'Ick' Brault, on pipeline and asset exploitation.
- Thierry 'Mamouth' Lauthelier, for ignition.
- Alexis Casas, for understanding and support.
- Nicolas 'Nikko' Brack, for endless higher expectations :)

2.11.3 Contributors

We welcome patches, bug reports and support. If you think your name should appears here, please contact us on the Kabaret Studio discord channel.

CHAPTER 3

Indices

- genindex
- modindex