

# Action(s)

## Developer Guide

Edited and written by Jean-Baptiste Brès.

Copyright ©2010 app.jbbres.com. All rights reserved.

The owner or authorized user of a valid copy of the *Action(s)* API may reproduce this publication for the purpose of learning to use such API. No part of this publication may be reproduced or transmitted for commercial purposes, such as selling copies of this publication or for providing paid for support services.

Every effort has been made to ensure that the information in this manual is accurate. app.jbbres.com is not responsible for printing or clerical errors. Because *Action(s)* frequently releases new versions and updates to its applications and Internet sites, images shown in this book may be slightly different from what you see on your screen.

Information in this document, including URL and other Internet Web site references, is subject to change without notice.

Other company and product names mentioned herein are trademarks of their respective companies. Mention of third-party products is for informational purposes only and constitutes neither an endorsement nor a recommendation. app.jbbres.com assumes no responsibility with regard to the performance or use of these products.

Even though app.jbbres.com has reviewed this document, APP.JBBRES.COM MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APP.JBBRES.COM BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED.

## Contents

<b>Introduction to Action(s) programming guide .....</b>	<b>4</b>
Who Should Read This Document .....	4
Organization of This Document .....	4
Additional documentation .....	5
Feedback and Bug Reporting .....	5
<b>Chapter 1   Action(s) and the Developer .....</b>	<b>7</b>
Constructing Workflows with Action(s) .....	7
Developing for Action(s) .....	8
<b>Chapter 2   How Action(s) works .....</b>	<b>10</b>
Loadable plug-in Architecture .....	10
Threading Architecture.....	13
The Action(s) Classes .....	13
The Element Information Property List .....	16
<b>Chapter 3   Design Guidelines for Action(s) .....</b>	<b>18</b>
What Makes a Good Element?.....	18
Action Input and Output.....	18
Variable data .....	19
Naming an Action.....	19
Naming a Variable.....	19
The Element icon .....	20
The User Interface of an Action .....	21
<b>Chapter 4   Developing an Action .....</b>	<b>23</b>
Creating the Project .....	24
Creating the Action Main Class .....	24
Constructing the User Interface .....	24
Writing the Action Service .....	33
Specifying Action Properties .....	36
Internationalizing the Action .....	38
Testing and Debugging the Action .....	39

<b>Chapter 5   Developing a Variable .....</b>	<b>41</b>
Creating the Project .....	42
Creating a Runtime Variable .....	42
Creating a storage variable.....	43
Creating the Variable's Renderer and Editor.....	44
Specifying Variable Properties .....	45
Internationalizing the Variable .....	46
Testing and Debugging the Variable.....	47
<b>Chapter 6   Creating and Deploying a Collection File .</b>	<b>48</b>
Creating a Collection File.....	48
Deploying a Collection File.....	51
<b>Chapter 7   Element Property Reference .....</b>	<b>53</b>
Property keys and values .....	53
<b>Revision History .....</b>	<b>58</b>

## Introduction to Action(s) programming guide

*Action(s)* is an application designed by app.jbbres.com that automates repetitive procedures performed on a computer.

With *Action(s)* users can construct arbitrarily complex workflows from modular units called actions. An **action** performs a discrete task, such as opening a file, cropping an image, or sending a message. A **workflow** is a number of actions in a particular sequence; when the workflow executes, data is piped from one action to the next until the desired result is achieved. Workflows can also contain **variables**: virtual containers that temporarily hold information, and pass it to the workflow when requested.

In *Action(s)*, actions and variables are grouped under the **element** denomination. They are enclosed into plug-in components named **collections**, which can be easily added or removed from *Action(s)*.

*Action(s)* is provided with a suite of ready-made elements, but developers are encouraged to contribute their own elements. Developers can easily create elements using Java.

*Action(s)* is designed to run with Java SE 6 or higher. It might work well Java SE 5, but some of the features like click-and-drag tools, and some pre-designed elements do not work or are limited when running the application with Java SE 5. When developing elements for *Action(s)*, it is highly recommended to use Java SDK 6 or higher. More information on Java and useful downloads can be found at <http://java.sun.com>.

### Who Should Read This Document

Any developer can create elements for *Action(s)*, as indeed can system administrators or "power users" who are familiar with Java. But application developers have a particular motivation for developing elements. They can create actions that access the features of their applications, and then install these elements along with their applications. Users of *Action(s)* can then become aware of the applications and what they have to offer.

Developers can also contribute to *Action(s)* by making their applications scriptable or by providing a programmatic interface (via an API) that developers use to create their elements.

### Organization of This Document

*Action(s)* Programming Guide consists of the following articles:

- [Action\(s\) and the Developer](#) describes what *Action(s)* can do and discusses the characteristics and types of elements,
- [How Action\(s\) Works](#) gives an overview of the architecture of *Action(s)* and the Java interfaces and classes of the *Action(s)* API.
- [Design Guidelines for Elements](#) lists guidelines for element development, including recommendations regarding I/O, naming, and the user interface.

- [Developing an Action](#) guides you through the major steps required to develop an action.
- [Developing a Variable](#) guides you through the major steps required to develop a variable.
- [Creating and Deploying a Collection File](#) describes the steps required to create a collection of elements and to deploy it.
- [Action\(s\) Element Property Reference](#) defines the types and expected values for the properties specified in an element's information property list.

## Additional documentation

As this document provides all the necessary documentation to create and deploy your own elements, a good understanding of the Java language is a pre-requisite to reading this document.

### Website

The *Action(s)*' website is all about *Action(s)*. It features lots of up-to-date information about the application, including new and updated features, development news, and tutorials. ↪ <http://app.jbbres.com/actions>

### Java Sun Website

The Java Sun Website gets you code samples, developer tools, downloads, open-source projects, resource centres, and support to Java. ↪ <http://java.sun.com>

### The Java Tutorials

The Java Tutorials are practical guides for programmers who want to use the Java programming language. They include hundreds of complete, working examples, and dozens of lessons. ↪ <http://java.sun.com/docs/books/tutorial/>

## Feedback and Bug Reporting

Though the *Action(s)* Team developers invest a lot of time and hard work making sure *Action(s)* and the *Action(s)* API are high-quality applications, as with any other software, bugs still sometimes find their way into the application or the API. That is why it is very important to keep developers informed about any bugs you are experiencing or any improvements that you think should be done in order to improve the application. And the sooner developers are informed, the faster they can act to fix or improve a feature.

Feedback about the Programming Guide you are reading now is also very welcome.

## app.jbbres.com

Use the *Report a Bug or Provide feedback* form at <http://app.jbbres.com/support/feedback> to let the *Action(s)* Team developers know about any bugs or feature requests you have.

## Chapter 1 | Action(s) and the Developer



*Action(s)* is an application designed by app.jbbres.com that lets users automate repetitive procedures performed on a computer.

With *Action(s)*, users can quickly construct arbitrarily complex sequences of configurable tasks that they can set in motion with a click of a button. And they do not need to write a single line of code to do so.

All kinds of users, including system administrators and developers, can derive huge benefits from *Action(s)*. It enables them to accomplish in a few seconds a complex or tedious procedure that manually might take many minutes. Developers can contribute to what *Action(s)* offers in two ways: by making their applications scriptable and by creating loadable modules specifically designed for *Action(s)*.

### Constructing Workflows with Action(s)

As a developer, you can best appreciate how to integrate your software products with *Action(s)* by understanding how users might use the application. You can download and install the application at <http://app.jbbres.com>. Figure 1 shows a typical *Action(s)* workflow.

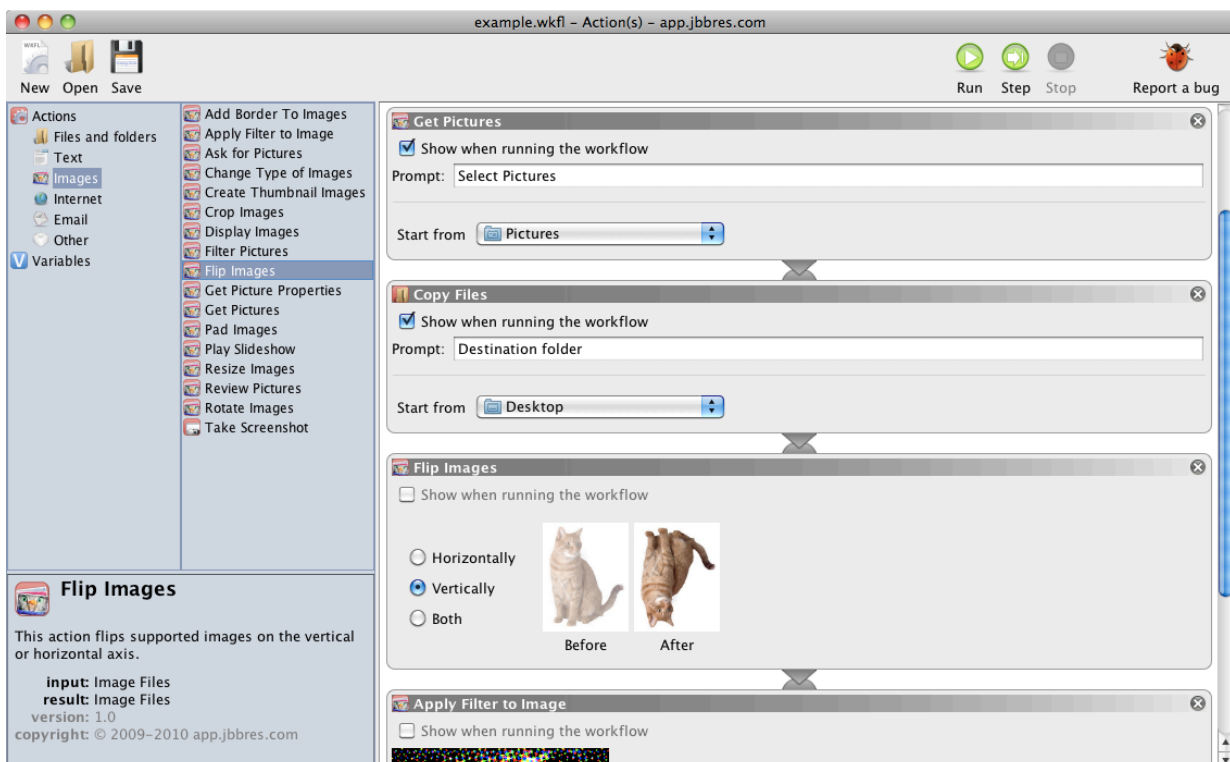


Figure 1 | A typical Action(s) workflow

A workflow in *Action(s)* consists of a sequence of discrete tasks called **actions**. An action is a kind of functional building block. A **workflow** hooks the actions together so that – in most cases – the output of one action is the input of the subsequent action.

Clicking the run button in the upper-right corner of the window causes the application to invoke each action of the workflow in turn, passing it (usually) the output of the previous action.

To create a workflow, users choose each action they want from the browser on the left side of the application window and drag it into the layout view in the right side of the window. They can search for specific actions by category, or keyword. Once dropped, an action presents a view which, in almost all cases, contains text fields, buttons, and other user-interface elements for configuring the action. In addition to configuring each action, users can move them around in the workflow to put them in the desired sequence.

Actions can be virtually anything that you can do on a computer. You can have an action that copies files, an action that crops an image, an action that send an email, or an action that builds a project in Java or C++. Actions can either interact with applications or draw on system resources to get their work done. By stringing actions together in a meaningful sequence – that is, in a workflow – a user can accomplish complex and highly customized procedures with a click of a button. *Action(s)* come with dozens pre-installed of actions.

Users can also use **variables** to temporary store some data and results. Actions are able to access to the variables if they are designed to do so, and users can also retrieve the value of a variable at any time within the workflow and use it as an input to any action.

They can be variables storing any type of data: file, text, images, date...

Actions and variables are referred as the **elements** of the workflow.

Once their workflow is defined, users can then save the workflow as a document that can be run again and again in *Action(s)*.

## Developing for Action(s)

*Tip: Consider making your applications accessible to 3<sup>rd</sup> parties elements even if you do not provide any Action(s) elements of your own. Other developers may decide to create elements that access your application's services*

Given an automation tool as powerful and flexible as *Action(s)*, it's clear that any element you develop for it benefits both you and the user, especially if that element accesses what your application has to offer. Users have an additional path to the features and services that make your products unique, and this in turn enhances the visibility and reputation of your products.

You don't even have to be a programmer (in the traditional sense) to develop elements for *Action(s)*.

As we saw in the previous section, elements can be either actions or variables. There are three general types of actions:

- An action that controls an application to get something done.
- An action that uses system APIs and other system resources to get something done.
- An action that performs some small but necessary “bridge” function such as prompting the user for input, writing output to disk, or informing the user of progress.

And two general types of variables:

- A variable that allow access to data related to your application, such as a specific folder.
- A variable that can store specific data used by your application or your actions.

From a developer’s perspective, actions and variables have a common programmatic interface and a common set of properties.

The programmatic interface is defined by a Java interface: `Element`. Two sub-interfaces inheriting from `Element`: `Action` and `Variable`, are available to describe respectively actions and variables.

Each element is associated to a description file known as the **element information property list**. It defines characteristics such as:

- The name and description of the element
- The icon associated to it.
- The categories of task it performs
- The description of the data the actions accepts and provides.

and can be accessed at any time without having to create an instance of the element. See the *Action(s) Element Property Reference* section for further information on *Action(s)* properties.

The *Action(s)* API provides support for element developers. It includes all the interfaces and classes required to create your own elements, additional classes to quickly create actions and variables and Swing objects that interact directly with the *Action(s)*.

You can download the *Action(s)* API at <http://app.jbbres.com/actions/developers>.

## Chapter 2 | How Action(s) works

The following sections describe the development environment, runtime architecture, and class hierarchy for *Action(s)* elements.

### Loadable plug-in Architecture

The *Action(s)* application is based on a loadable plug-in architecture. Elements are packaged as a loadable plug-in called **collection**. A collection contains resources of various kinds and usually binary code, but it is not capable of executing that code on its own. The internal structure of a collection is similar to the java JAR files one.

When it launches, *Action(s)* immediately scans the currently installed collections and extracts from each collection manifest the information necessary to display and prepare the elements for use (see Figure 2). Collections are stored in a standard file-system location: *[Users Preferences folder]/app.jbbres.com/Actions/plugins*.

See [Creating and Deploying a Collection File](#) for information on installing collections.

[Users Preferences folder]/com.jbbres.app/Action(s)/plugins

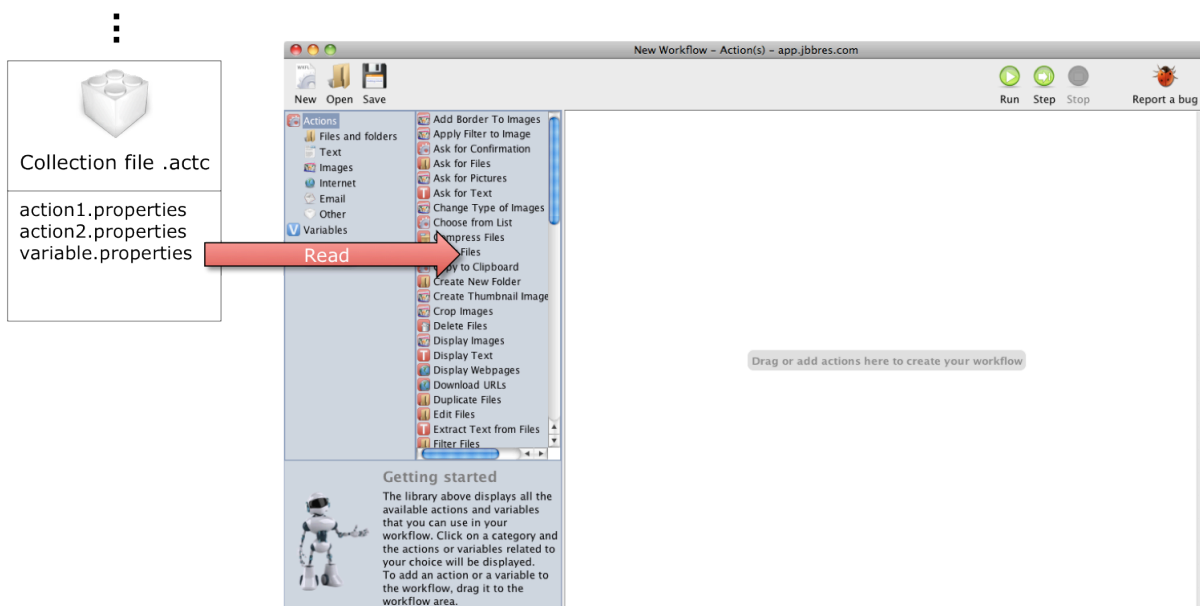


Figure 2 | When launched, Action(s) gets information about available elements

When it launches, *Action(s)* also loads any Java code and resources it finds in the collection.

For each element of the collection, *Action(s)* gathered all the information required to display the element description through the element information property list – a .properties file stored with the element class. However, no instance of the element has been created at this stage.

When a user drags an action or a variable into the workflow area the application creates an instance of the element and displays the action's view or adds it in the variable list, depending if the element is an action or a variable (see Figure 3).

[Users Preferences folder]/com.jbbres.app/Action(s)/plugins

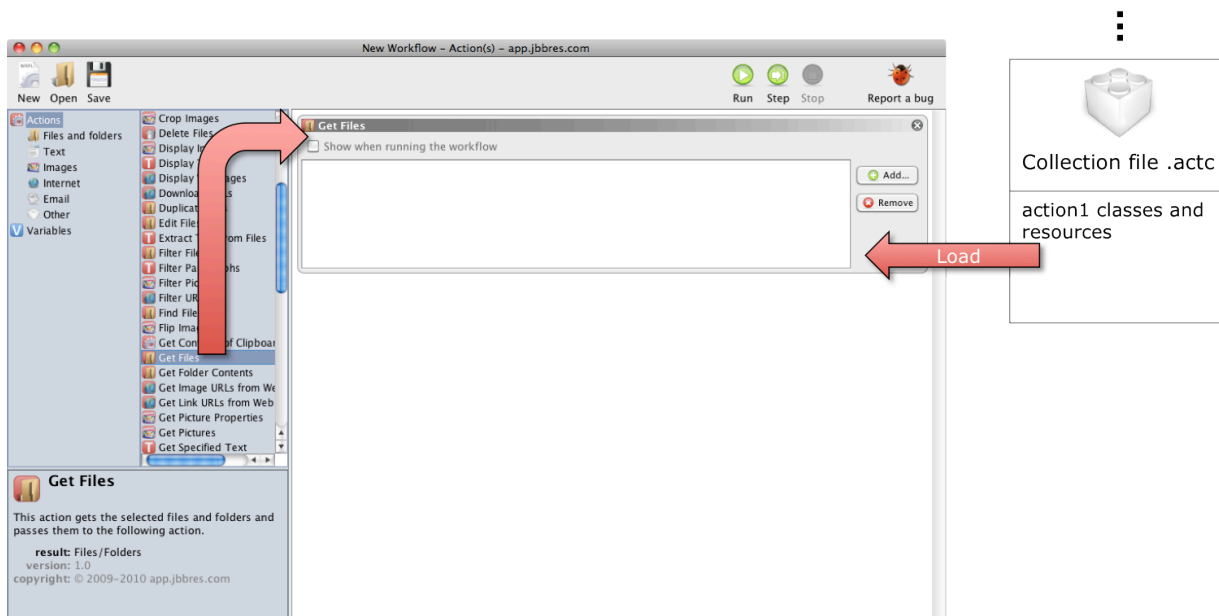


Figure 3 | When user drags action or variable into workflow, Action(s) creates an instance of the element.

**Note:** A new instance of the element is created for each time the user drag the element in the workflow. This means that the same element can have multiple instances simultaneously in the same workflow.



## A New Workflow

When the user creates a new workflow by dragging one or more elements into the workflow layout view, Action(s) does a couple of things:

- It creates an instance of the element by calling the default constructor `ElementName(Workflow workflow)`, passing the workflow object as an argument.
- If the element is an action, it gets its User Interface by calling the `getUI()` method of the Action instance and displays it within an action view in the workflow area.

Users modify the parameters of an action by choosing pop-up items, clicking buttons, entering text into text fields, and so on. When the workflow is ready, they click the run button to execute the workflow. With Action(s) acting as a coordinator, the application and each action perform the following steps in the workflow sequence:

- *Action(s)* invokes the `execute(Object input, Parameter parameter)` method of the action, passing it the output of the previous action as input via the `input` object. The settings made in the user interface of the action can be propagated via the `parameter` object.
- The action object (in most cases) takes the input and, based on the parameters, transforms it or does whatever its stated role is (such as importing it into an application or displaying output).
- As its last step, the action returns the result of its work (its output). If it does not affect the data passed it as input, it simply returns it unchanged.

While each action is busy, *Action(s)* displays a spinning progress indicator in the action view. If an error occurs in an action, the action view displays a red  and an error message. If the action successfully completes, its view displays a green  mark. When the last action has finished its task, the workflow execution is over.

### An Un-Archived Workflow

When users save a workflow, the workflow and all of its elements are archived. *Action(s)* invokes the `getParameters()` method of each of the elements in the workflow. The combined parameters of each action of the workflow are encoded and archived in a `.wkfl` file.

When *Action(s)* reads a workflow file from disk, it re-creates an instance of all the elements in the workflow and invokes the `setParameters(Parameters)` method of each elements. Based on the parameters given, the element can re-creates its state, particularly the last-selected parameters.

Once all elements in the workflows have been reinitialized from the file, the workflow is ready to use. Users can change settings in elements and run the workflow. Things continue on at this point as described in [A New Workflow](#).

### Programming Implications of Plug-in Architecture

The loadable plug-in architecture of *Action(s)* has some implications for developers willing to create their own actions or variables.

The main implication concern namespaces: a Java class defines a namespace for the methods and instance variables it declares. Because of this, identically named methods and instance variables in other classes do not cause conflicts within a process. However the name of a class itself exists in a namespace occupied by all classes loaded by a process.

For *Action(s)*, with its loadable plug-in architecture, the potential for namespace collisions – and hence runtime exceptions – is significant. *Action(s)* can potentially load hundreds of elements from different sources and, for example, if two action classes have the same name, there is a potential for namespace conflict when those actions are loaded by *Action(s)*.

To avoid namespace collisions, it is recommended that you declare your classes within packages as distinctive as possible. For example, if your company's name is Acme, you might define your element classes within a package `com.acme`.

You can find more information about package and how to name them at <http://java.sun.com/docs/books/tutorial/java/package/namingpkgs.html>.

## Threading Architecture

To improve runtime stability, *Action(s)* has a threading architecture that puts different types of program activity on separate threads.

*Action(s)* starts a workflow on a secondary thread (that is, a thread other than the main thread).

This threading architecture imposes restrictions:

- The user interface does not belong to the same thread than the one executing the action, which means that the user can potentially interact with the action UI while the action is executed. However, the `setEnabled(Boolean)` method of the UI component is called before and after the action execution. By overriding this method you can make sure that the user will not be able to change the settings of your action while it is running.
- The user cannot cancel the execution of an action while it is running. If the user click on the stop or pause button, the workflow will stop (or pause) only when the current action is executed.

## The Action(s) Classes

*Action(s)* as a technology includes not only the application and its elements but also the *Action(s)* API. The API provides public interfaces and classes that implement much of the common behaviour of elements and workflows.

### Elements, Actions and Variables

The *Action(s)* public model for creating elements (either actions or variables) is defined in the `com.jbbres.lib.action.elements` package. Figure 4 describes this model.

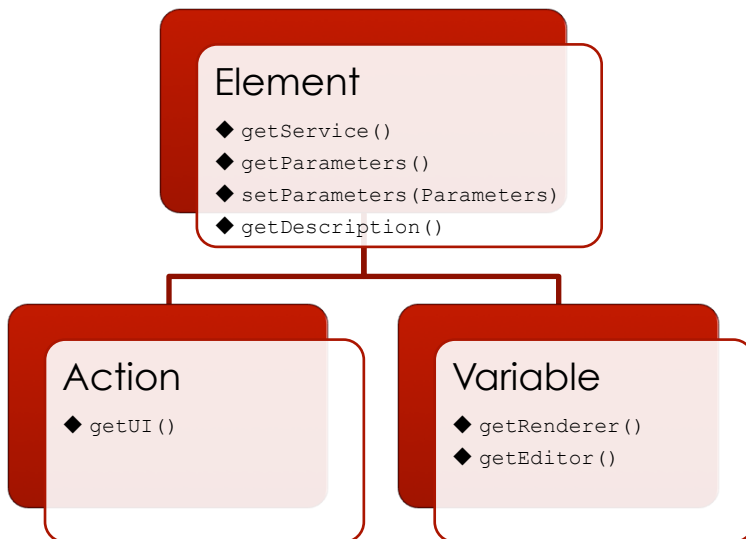


Figure 4| The Action(s) public model

**Element** is an interface that specifies all methods essential to all elements (actions or variables). An implementation of `Element` has 4 major methods:

- **getService()** returns the service associated to the element. The service defines how the element performs within the workflow. For example, if the element is an action, the service will contain the executable methods that the action is designed to perform. If the element is a variable, the service will contain methods related to data access.
- **getParameters()** and **setParameters(Parameters)**, respectively stores and loads the state of the element, as presented quickly previously.
- **getDescription()**, a method that returns a dictionary derived from the element description, usually the information specified in the element information property list.

↪ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/Element.html>

## User Interface

The **Action** interface inherits of all `Element` methods. In addition, it specifies a new method essential to all actions:

- **getUI()** returns the User Interface displayed in the workflow definition panel in *Action(s)*, and that user can use to define the settings of the action.

↪ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/Action.html>

**Tip:** *Action(s) 1.0 does not support the `getRenderer()` and `getEditor()` methods. Their support will be added in Action(s) very soon.*

In a similar way than the `Action` interface, the `Variable` interface has two User Interface methods in order to provide a better user experience when working with variables:

- `getRenderer()` defines the how to render the variable value into the variable table.
- `getEditor()` returns an editor that the user will be able to use to changes the variable value during the workflow definition.

↪ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/Variable.html>

### ElementService, ActionService and VariableService

The `getService()` method of an `Element` returns an instance of `ElementService` (an `ActionService` object if the element is an action, a `VariableService` object if the element is a variable).

The service is a heart of an element as it defined its behaviour within the workflow. It describes how the element performs during the workflow execution.

For an action, the `ActionService` describes:

- the object types that are accepted as an input for the action execution. This is the purpose of the `isValidInputClass(Class)` method.
- the object types that the action execution may produce. This is the purpose of the `outputClass(Class)` method.
- what the action does if it is executed by the workflow. This is the purpose of the `execute(Object, Parameters)` method.

↪ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/ActionService.html>

For a variable, the `VariableService` describes:

- the object type that the variable can store via the `valueClass()` and `isValidValueClass(Class)` methods.
- The stored value with the `getValue()` and `setValue(Object)` methods.
- The variable name (also called variable instance name), via the `getInstanceName()` and `setInstanceName(String)` methods.

↪ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/VariableService.html>

## Parameters

The parameters of an element serve a double objective:

1. They are used to save and restore the setting associated to an element. When the user saves a workflow in a file, *Action(s)* calls the `getParameters()` method of each element and convert the received object into an xml document stored inside the .wkfl file. When, later on, the user re-open the workflow, *Action(s)* extract the xml document associated to the element, converts it into a `Parameters` object, and assigns it to the element calling the `setParameters(Parameters)` method.
2. If the element is an action, the parameters are passed as an argument of the service's method `execute(Object, Parameters)`. The method can extract from the received object the settings to apply during its execution.

↳ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/Parameters.html>

## ElementDescription, ActionDescription and VariableDescription

The `ElementDescription` class – and its 2 subclasses `ActionDescription` and `VariableDescription`, used respectively by the `Action` and `Variable` classes – gives information regarding the element, such as its name, a short description, its version number etc.

That information might or might not be the same as the one provided by the [element information property list](#) and have various usages within the workflow definition process. It is interesting to note that one of the strength of the `ElementDescription` over the element information property list is that it can provide information specific to an instance of the element, whereas the element information property list only provide general information regarding the `Element` class.

↳ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/elements/ElementDescription.html>

## The Element Information Property List

The element information property list is a .properties file stored within the same package as the element class and providing characteristics such as:

- The name and description of the element.
- The icon it is associated to.
- The categories of task it performs
- The description of the data the action accepts and provides.

Contrary to the `ElementDescription` object associated to an element instance, the element information property list is accessible at anytime, even if no instance of the element is available. It is used to gather

information regarding the element when *Action(s)* is launching and display the extracted information in the element library.

## Chapter 3 | Design Guidelines for Action(s)

As with other parts of the human interface, *Action(s)*'s elements and especially actions should have a consistent look and feel so users can easily use them. The following guidelines will help you achieve that consistent look and feel.

### What Makes a Good Element?

Perhaps the most important guideline is to keep your element as simple and focused as possible. An element should not attempt to do too much; by doing so it runs the risk of being too specialized to be useful in different contexts. For example, an action should perform a narrowly defined task well. If an action you're working on seems like it's unwieldy, as if it's trying to do too much, consider breaking it into two or more actions. Small, discrete actions are better than large and complex actions that combine several different functionalities.

An element should inform the user what is going on, and if it encounters errors, it should tell users about any corrective steps that they might take. If an action takes a particularly long period to complete, consider displaying a determinate progress indicator. (*Action(s)* displays a circular indeterminate progress indicator when an action runs.)

You should provide an element in as much localization as possible. See [Developing an Action](#) and [Developing a Variable](#) for further information on internationalizing elements.

### Action Input and Output

Interoperability is critical in the implementations of elements. An action's usefulness is limited by the types of data it can accept from other actions and give to other actions in a workflow.

You specify these data types through the `isValidInputClass(Class)` and `outputClass(Class)` methods in the action's service. The following guidelines apply to action input and output.

- Make the types of data the action accepts as the less specific possible. For example, if the action accepts an image, try to use a `java.awt.Image` object instead of a `java.awt.image.BufferedImage`.
- Specify multiple accepted types, unless that is not appropriate for the action.
- Ideally, an action should accept and provide a list (or array) of the specified types.
- If your action doesn't require input and doesn't provide output (such as the Pause action), it should use the `java.lang.Void` type. The API will then route the flow of data around the action.
- If the output class of your action is undetermined (for example if it depends on the content of the action input), you can use the `java.lang.Object` class.

- Even if your action requires input it should still be prepared to handle gracefully the case where it doesn't get input (the received value is `null`).

The usage of the following classes is recommended in order to maximize the compatibility of your actions:

Object	Class
Date	<code>java.util.Date</code>
File and Folder	<code>java.io.File</code>
Image	<code>java.awt.Image</code>
Image File	<code>java.io.File</code>
Number	<code>java.lang.Number</code>
Text	<code>java.lang.String</code>
URL	<code>java.net.url</code>
Object	<code>java.lang.Object</code>
N/A	<code>java.lang.Void</code>

## Variable data

The guidelines provided for [action input and output](#) apply to variable data too.

## Naming an Action

The following guidelines apply to the names of actions:

- Use long, fully descriptive names (for example, *Add Attachments to Front Mail Message*).
- Start the name with a verb that specifies what the action does.
- Use plural objects in the name – actions should be able to handle multiple items, whether that be URLs or files. However, you may use the singular form if the action accepts only a single object (for example *Add Date to File Names*, where there can be only one date).
- Don't use "(s)" to indicate one or more objects (for example, *Add Attachment(s)*). Use the plural form.

## Naming a Variable

The following guidelines apply to the names of variable:

- If your variable can store a value, start the name with "New", followed by a short description of the type the variable can store. A good name for a storage variable can be *New audio file*.
- If your variable provides access to an external data, such as the default folder used by your application or the number of customers recorded in your database, simply states what the data is (for example: *Pictures folder* or *Number of customers*).

- Use plural objects if the variable may returns more than one object. Otherwise use singular.
- Don't use "(s)" to indicate one or more objects (for example, *Customer(s)*). Use the plural form.

## The Element icon

This section describes the overall philosophy behind element icons.

*Action(s)* offers an illustrative icon style to convey a lot of information in a small space. Anti-aliasing makes curves and no rectilinear lines possible. Alpha channels and translucency allow for complex shading and dimensionality. All of these qualities allow you to create lush, vibrant icons that capture the user's attention.

To represent your element in *Action(s)*, it's essential to create high-quality icons that scale well in the various places the icon appears. *Action(s)* uses two version of the element icon: a 32 x 32 pixels version when displaying the element in the element description panel, and a 16 x 16 pixels version when displaying the element in the library. If the provided icons do not match with those sizes, *Action(s)* automatically rescale the icons to fit with these requirements.

**Tip:** The standard bit depth for icons and images is 24 bits (8 bits each for red, green, and blue), plus an 8-bit alpha channel. The PNG format is recommended, because it preserves colour depth and supports an embedded alpha channel.



A 16 x 16 pixels icon



A 32 x 32 pixels icon

Figure 5 | Icons sizes

Traditionally, an element icon looks like a colour square with rounded corners. It includes an image representing the element functionality.

The square colour depends on the type of element the icon represents: a red square represents an action, a light blue square represents a storage variable and a purple square represents a runtime variable.



An action icon



A storage variable icon



A runtime variable icon

Figure 6 | Icons' background colours

The image is not centred in the middle of the square but in the middle of the lower right quarter of the square, as shown in the following example.

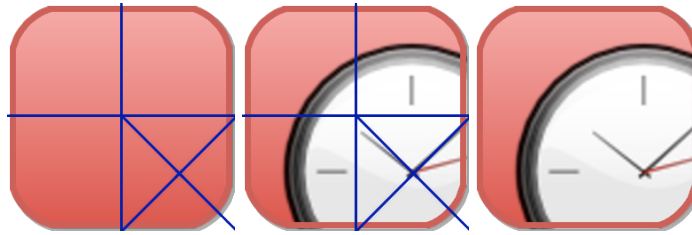


Figure 7 | Image position within the square

Element icons should make their associated functionality obvious. If the element provides access to functionalities from an external application, using the application icon is recommended.

Templates of element icons are provided in the API to help you create your own icons for *Action(s)*.

## The User Interface of an Action

The user interface of an action should adhere to the following guidelines:

### Keep it simple

- Refrain from using boxes.
- Minimize the use of vertical space; in particular, use *combo boxes* instead of radio buttons even if there are only two choices.
- Avoid tab views; instead, use hidden tab views to swap alternate sets of controls when users select a top-level choice.
- Don't have labels repeat what's in the action title or description.

### Keep it small and consistent

- Use 10-pixel margins.
- Use small controls and labels.
- Follow the Java Look and Feel guidelines – visit <http://java.sun.com/products/jlf/> for more information.
- Implement behaviour expected in a user interface – for example, tabbing between multiple text fields.

### Provide feedback and information to users

- Use determinate progress indicators when a user-interface element needs time to load its content; for example, an action that presents a list of data extracted from a database load them.
- Present examples of what the action will do when possible. For instance, the *Make Sequential File Names* action has an area of the view labelled "Example" that shows the effect of action options; the examples have the same font size and colour as the rest of the action's user interface. Figure 8 shows an action that uses images for its example.

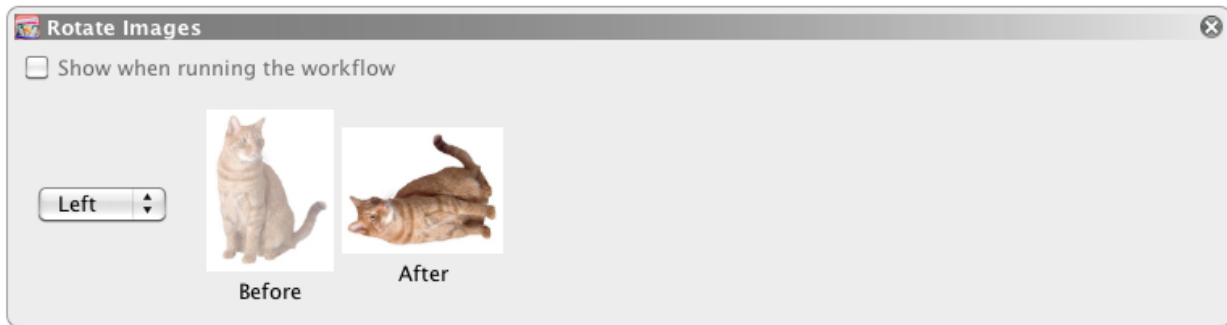


Figure 8 | An action that includes an example of its effect

### **Streamline file and folder selection and display**

- Insert a *combo box* that includes standard file-system locations, such as Home, Start-up Disk, Documents, Desktop, and so on.
- The *Action(s)* API utilities includes pre-configured *combo boxes* for selecting directories, and files; use these objects where appropriate.

## Chapter 4 | Developing an Action

It's easy to develop an *Action(s)* element. Because an action is a loadable plug-in, its scope is limited and hence the amount of code you need to write is limited. app.jbbres.com also eases the path to developing an element because of all the resources it places at your disposal. Various abstract element classes and Swing objects are at your disposal. You just need to follow certain steps – described in this document – to arrive at the final product.

The steps for developing an element don't necessarily have to happen in the order given below. For example, you can write an element description at any time and you can specify the User Interface at any time.

The *Action(s)*'s class structure is described in section [The Action\(s\) Classes](#). This structure is very efficient and allows a maximum of flexibility for the developer. However, if you want to develop a simple action, you might not want to code all the methods and functionalities declared in the interface.

app.jbbres.com provides some abstract classes that you can easily implement to quickly develop your own elements. Those classes are available in the `com.jbbres.lib.actions.tools.elements` package of the *Action(s)* API.

↪ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/tools/elements/package-summary.html>

You'll become more familiar with many of the files that will constitute your action in the sections that follow. But here is a summary of the more significant items:

- **actionName.properties:** The information property list includes the action description. See [Specifying Action Properties](#) for further information.
- **ActionName.java:** The java class that controls the action. This class extends [com.jbbres.lib.actions.tools.elements.AbstractAction](#).
- **ActionNameService.java:** The java class that provides the service associated to the action. This class extends [com.jbbres.lib.actions.tools.elements.AbstractActionService](#).
- **ActionNameUI.java:** The java class that provides the User Interface associated to the action. This class extends [com.jbbres.lib.actions.tools.elements.AbstractActionUI](#).

## Creating the Project

Depending on the EDI you are using for developing in Java, the process to create a new action might be different. Refer to your EDI documentation for more information on how to create a project.

You might need to include the *Action(s)* API as a known library of your tool in order to be able to develop and compile your action. You can download this file at: <http://app.jbbres.com/actions/developers/>

## Creating the Action Main Class

The action main class controls the action by providing a centralized access to all components of the action, such as the UI, the service class, the action description and the parameters.

The `com.jbbres.lib.actions.tools.elements.AbstractAction` class is the default implementation of the main action class. It does most of the work for you as most of the methods that this class should contain are pre-defined by this abstract class.

In your project, creates a new Java file named `ActionName.java` where `ActionName` is the name of your action. The content of your class will look like the following example:

```
package com.acme;

import java.io.IOException;
import com.jbbres.lib.actions.tools.elements.AbstractAction;
import com.jbbres.lib.actions.workflow.Workflow;

/**
 * The action main class.
 */
public class AddTextToFileName extends AbstractAction {

    /**
     * Instantiates a new <code>AddTextToFileName</code>.
     *
     * @param workflow - the workflow
     */
    public AddTextToFileName(Workflow workflow)
        throws IOException {
        super(workflow);
    }
}
```

## Constructing the User Interface

In your project, creates a new Java file named `ActionNameUI.java` where `ActionName` is the name of your action. This class extends `com.jbbres.lib.actions.tools.elements.AbstractActionUI`. It should have a constructor receiving a single `AbstractAction` object as argument, and two methods `getUIParameters()` and `setUIParameters(Parameters)`:

```

package packageName;

import com.jbbres.lib.actions.elements.*;
import com.jbbres.lib.actions.tools.elements.*;

public class ProjectNameUI extends AbstractActionUI {

    public ProjectNameUI(AbstractAction parent) {
        super(parent);
    }

    protected Parameters getUIParameters() {
        return null;
    }

    protected void setUIParameters(Parameters parameters)
        throws InvalidParametersException {
    }
}

```

You will create and design your User Interface within this class. The User Interface can be created using AWT, Swing or any other compatible toolkit. More information regarding how to use AWT and Swing can be found <http://java.sun.com/javase/6/docs/technotes/guides/awt/> online:

and <http://java.sun.com/javase/6/docs/technotes/guides/swing/>.

The User Interface components should be added to the content panel of the class. The content panel is a `JPanel` object that you can access by calling the `getContentPane()` method.

The `getUIParameters()` and `setUIParameters(Parameters)` generate and set the parameters associated to the User Interface. The parameter object that the `getUIParameters()` method returns is used by *Action(s)* to store the state of the User Interface within the `.wkfl` file when the user saves the workflow. The same object is given to the `setUIParameters(Parameters)` method when the user opens a `.wkfl` file, so the User Interface can be reconstructed at a similar state as it was when the workflow had been saved.

The `Parameters` object is also given to the action service as an argument of the `execute(Object, Parameters)` method. The service will use the parameters provided to know what settings the user want to apply to the action when it is executed.

The following code is a good example of a well-designed User Interface:

```

package com.jbbres.examples.actions;

import java.awt.FlowLayout;
import javax.swing.JComboBox;
import javax.swing.JLabel;
import javax.swing.JTextField;
import com.jbbres.lib.actions.elements.*;
import com.jbbres.lib.actions.tools.elements.*;

/**
 * The User Interface of the AddTextToFileName action.
 */

```

```

    */
    public class AddTextToFileNameUI
        extends AbstractActionUI {

        private JLabel addTextPanel;

        /**
         * The text field that the user will use to specify the
         * text to be added to the file name
         */
        private JTextField textField;

        /**
         * A combo box that the user will use to choose the
         * position where the text is to be added - before or
         * after the file name.
         */
        private JComboBox positionComboBox;

        /**
         * Instantiates a new AddTextToFileNameUI.
         *
         * @param parent - the parent action
         */
        public AddTextToFileNameUI(AbstractAction parent) {
            super(parent);

            addTextPanel = new JLabel("Add text:");
            textField = new JTextField(50);
            String[] positions = { "before file name",
                                   "after file name" };
            positionComboBox = new JComboBox(positions);

            getContentPane().setLayout(
                new FlowLayout(FlowLayout.LEADING));
            getContentPane().add(addTextPanel);
            getContentPane().add(textField);
            getContentPane().add(positionComboBox);
        }

        /**
         * Returns the current state of the UI.
         */
        protected Parameters getUIParameters() {
            Parameters parameters = new Parameters();

            // saves the current value of the components into
            // the parameters object.
            parameters.setParameter("text",
                                   textField.getText());
            parameters.setParameter("position",
                                   (String) positionComboBox.getSelectedItem());

            return parameters;
        }

        /**
         * Sets the current state of the UI.
         */
        protected void setUIParameters(Parameters parameters)
            throws InvalidParametersException {

```

```

        // restores the components based on the content of
        // the parameters object.

        textField.setText(
            parameters.getParameter("text"));
        positionComboBox.setSelectedItem(
            parameters.getParameter("position"));
    }

    /**
     * Sets whether or not this component is enabled. This
     * method is overridden to make sure that when the state
     * of the UI changes all its components are updated.
     */
    public void setEnabled(boolean enabled) {
        super.setEnabled(enabled);
        addTextPanel.setEnabled(enabled);
        textField.setEnabled(enabled);
        positionComboBox.setEnabled(enabled);
    }
}

```

### Show when run

Using an `AbstractActionUI` object is a great way to let users define the behaviour of the action they are willing to use. However, the user defines this setting when he creates the workflow. If the workflow is meant to be re-used with different settings every time it is executed, the user will have to update them manually within the workflow every time, which will depreciate its experience of the application.

To get around this problem, `AbstractActionUI` includes a **Show When Run** feature. If this feature is turned on, when *Action(s)* executes the workflow it displays the user interface for the action when execution reaches that point. The user can then make the required settings before the action proceeds.

The User Interface of actions using the Show When Run feature includes an additional section with a *Show when running the workflow* check box in their top section. When a user clicks this control, this section expands to expose a *Prompt* text field.

Figure 9 illustrates the "Show When Run" control sets.

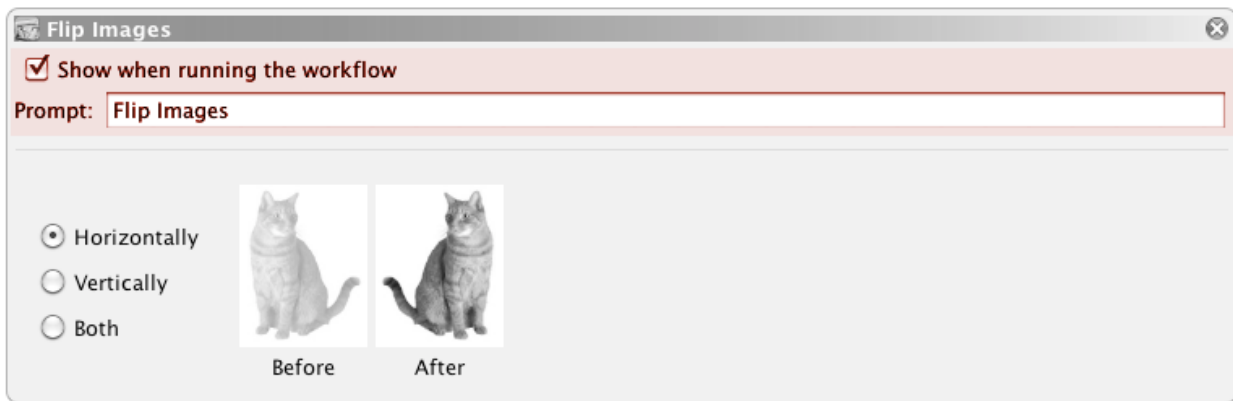


Figure 9 | The "Show when running the workflow" option

If the check box is selected and the workflow is run, the action displays its User Interface in a separate window, as shown in Figure 10. The value inputted in the prompt text field is used as title of the window.

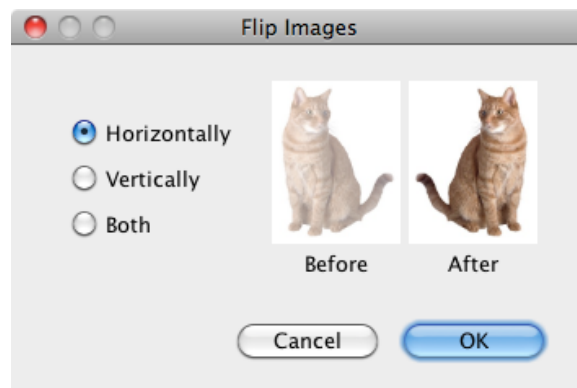


Figure 10 | Action displaying User Interface in a window

The user makes selections and fills in information in this window and clicks OK to have the action proceed.

The `setShowWhenRunAvailable(boolean)` method in the `AbstractActionUI` class activates or deactivates the Show When Run feature for the User Interface. It is also possible to define the default state of the "Show when running the workflow" check box by using the `setShownWhenRun(boolean)` method, and the default value of the prompt with the `setPrompt(String)` method. Alternatively, the Show When Run feature can be set within the action property list. See the [Element Property Reference](#) section for more details.

### Refining Show When Run

In the default Show When Run configuration, *Action(s)* displays the User Interface in a separate window. Developers can refine the default

behaviour of the Show When Run feature in two ways: by providing a specific User Interface for the workflow creator when he activates the Show When Run functionality or by defining a specific behaviour that will replace the display of the User Interface window for the workflow user.

An example of a refined Show When Run behaviour is the *Get Pictures* action provided by default with *Action(s)*. As shown in Figure 11, the default action's User Interface provides four components: a list area displaying the selected files, two buttons "add" and "remove", and a spinner updating the size of the image preview in the list.

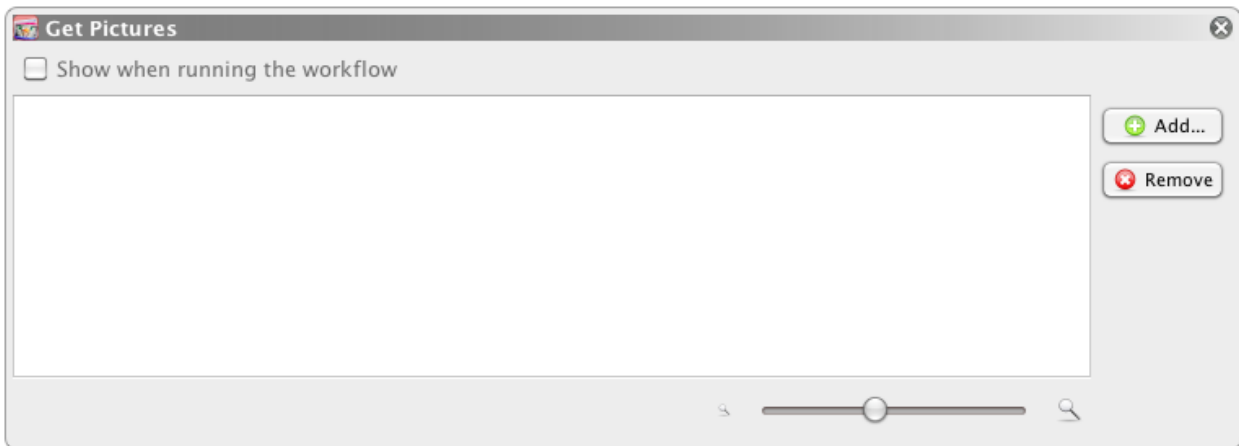


Figure 11 | User Interface when the "Show when running the workflow" option is not active.

When "Show when running the workflow" option is selected, the User Interface changes and provides a single combo box for selecting the starting directory.

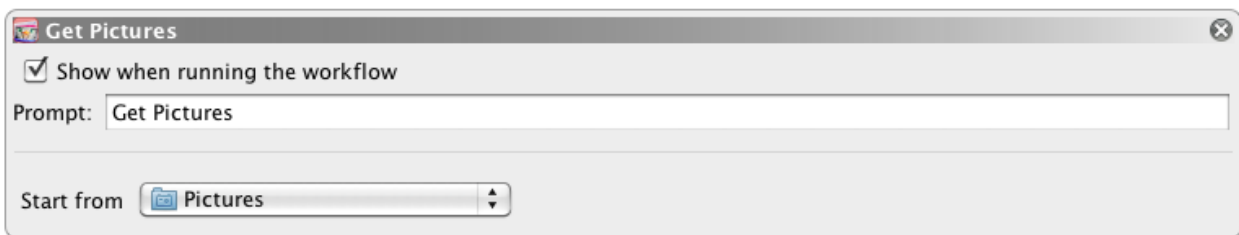


Figure 12 | User Interface when the "Show when running the workflow" option is active.

In this case, when the action is run, a file chooser dialog is displayed, showing the content of the starting directory.

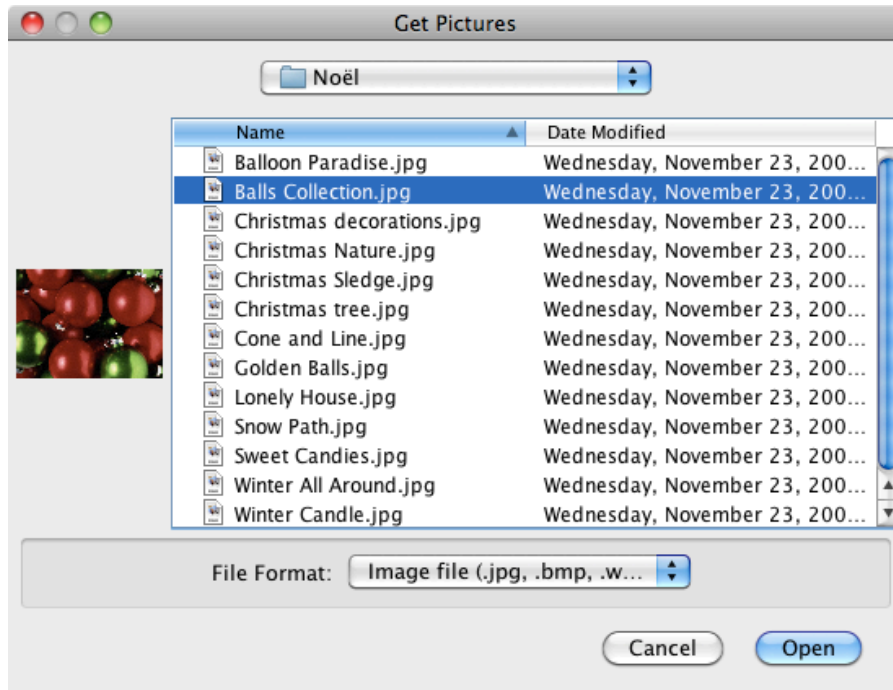


Figure 13 | Action specific behaviour when run.

You can easily change the content of the User Interface when the workflow creator select the "Show when running the workflow" by listening to `AbstractActionUI` events with an `ActionUIListener`. The `showWhenRunOptionChanged` method of a `ActionUIListener` is called every time the "Show when running the workflow" is checked or unchecked. You can update the User Interface within this method, based on the state of the check box, as shown in the example below.

```
package packageName;

import javax.swing.JCheckBox;
import javax.swing.JTextField;
import
com.jbbres.lib.actions.elements.InvalidParametersException;
import com.jbbres.lib.actions.elements.Parameters;
import com.jbbres.lib.actions.tools.elements.AbstractAction;
import
com.jbbres.lib.actions.tools.elements.AbstractActionUI;
import com.jbbres.lib.actions.tools.elements.ActionUIEvent;
import
com.jbbres.lib.actions.tools.elements.ActionUIListener;

public class ActionNameUI extends AbstractActionUI
implements ActionUIListener{

    public JCheckBox component1;
    public JTextField component2;

    public ActionNameUI(AbstractAction parent) {
        super(parent);
        addActionUIListener(this);
    }
}
```

```

        // Creates 2 components. component1 will be
        // visible when the Show When Run option is not
        // selected, otherwise component2 will be visible.
        component1 = new JCheckBox("This component is " +
            "displayed if the Show When Run option " +
            "is not selected");
        component2 = new JTextField("This component is" +
            "displayed if the Show When Run option " +
            "is selected");
        getContentPane().add(component1);
        getContentPane().add(component2);

        // turn on the Show When Run feature
        this.setShowWhenRunAvailable(true);
        // set the Show When Run option to not selected
        setShownWhenRun(false);
    }

    protected Parameters getUIParameters() {
        return null;
    }

    protected void setUIParameters(Parameters parameters)
        throws InvalidParametersException {
    }

    public void showWhenRunOptionChanged(ActionEvent e) {
        // The value of the Show When Run option has
        // changed
        component1.setVisible(!e.isShownWhenRun());
        component2.setVisible(e.isShownWhenRun());
    }

    public void promptChanged(ActionEvent e) {
    }
}

```

Changing the behaviour of the User Interface when the action is run and the Show When Run option selected requires you to override the `showWhenRun()` method of your `AbstractActionUI` instance.

The `showWhenRun()` method is called at the very beginning of the action execution. Use this method to display a personalized frame where the user can define the settings of the action. In order to have those settings processed by the action service, make sure that the result of the `getUIParameters()` method reflects the selection made by the user in your personalized frame. The example below shows how a User Interface can take advantage of the `JOptionPane` functionalities when using the Show When Run feature.

```

package packageName;

import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;
import
com.jbbres.lib.actions.elements.InvalidParametersException;

```

```

import com.jbbres.lib.actions.elements.Parameters;
import com.jbbres.lib.actions.tools.elements.AbstractAction;
import
com.jbbres.lib.actions.tools.elements.AbstractActionUI;
import com.jbbres.lib.actions.tools.elements.ActionUIEvent;
import
com.jbbres.lib.actions.tools.elements.ActionUIListener;

public class ActionNameUI extends AbstractActionUI
    implements ActionUIListener{
    JTextField textField;
    JLabel label;
    String input;

    public ActionNameUI(AbstractAction parent) {
        super(parent);
        addActionUIListener(this);

        textField = new JTextField();
        label = new JLabel("Text:");
        getContentPane().add(label);
        getContentPane().add(textField);

        setShowWhenRunAvailable(true);
        setShownWhenRun(false);
    }

    protected Parameters getUIParameters() {
        Parameters result = new Parameters();
        String text;
        if (isShownWhenRun())
            text = input;
        else
            text = textField.getText();

        if (text == null)
            text = "";

        result.setParameter("text", text);
        return result;
    }

    protected void setUIParameters(Parameters parameters)
        throws InvalidParametersException {
        String text = parameters.getParameter("text");
        if (text != null)
            textField.setText(text);
        else
            textField.setText("");
    }

    public void showWhenRun() {
        input = JOptionPane.showInputDialog(getPrompt(),
            textField.getText());
    }

    public void showWhenRunOptionChanged(ActionUIEvent e) {
        if (e.isShownWhenRun())
            label.setText("Default text:");
        else

```

```

        label.setText("Text:");
    }

    public void promptChanged(ActionEvent e) {}
}

```

## Writing the Action Service

The most important step in creating an action is writing the Service object that implements the logic for your action.

In your project, creates a new Java file named `ActionNameService.java` where `ActionName` is the name of your action. This class extends `com.jbbres.lib.actions.tools.elements.AbstractActionService`.

`AbstractActionService` is a parameterized abstract class with two type-variables, which means that all implementation of this class requires two type-arguments:

1. The first argument represents the type of object that the action accepts as an input.
2. The second argument is the type of object that the action generates as a result.

For example, if your action is supposed to receive an array of files (`File[]`) and return a text (`String`), the content of your service class file will look like the following example:

```

package packageName;

import java.io.File;
import com.jbbres.lib.actions.elements.*;
import com.jbbres.lib.actions.tools.elements.*;

public class ActionNameService extends
AbstractActionService<File[], String> {

    public ActionNameService(AbstractAction parent) {
        super(parent);
    }

    @Override
    public String executeAction(File[] input, Parameters
parameters)
        throws ActionExecutionException {
        // The logic for the action goes here
        return "Result of the action";
    }
}

```

If an action does not have to deal with the input data handed it – for example, its role is to select some items in the file system – the first parameterized argument should be `java.lang.Void`. On the opposite, if the action is able to handle any type of input data, the first parameterized argument will be `java.lang.Object`.

**Tip:** For more information regarding generic classes, visit <http://java.sun.com/docs/books/tutorial/java/generics/index.html>.

These conventions apply to the result data type too. A second parameterized argument sets to `java.lang.Void` means that the action is not expected returning a result. `java.lang.Object` informs that the result of the action can be any type of object.

A particular case in action is a service not receiving input nor generating result. Following the convention, such a service should extend `AbstractActionService<java.lang.Void, java.lang.Void>`. `Action(s)` will recognize this specific case and will route the flow of data around the action.

### Implementing `executeAction`

In your custom subclass of `AbstractActionService` you must override the method `executeAction`. Aside from a constructor, this method implementation is the only requirement for creating a service.

The `executeAction` method has two parameters, `input`, and `parameters`.

- The `input` parameter contains (in most instances) the output of the previous action in the workflow; it is almost always in the form of a list. The type of `input` is the first parameterized argument of the service class (in our example above, a `File[]`). Remember that if the previous action result is not compatible, or if this action is the first one in the workflow, the `input` parameter will be `null`. Make sure that you handle this case properly to avoid a `NullPointerException`.
- The `parameters` parameter contains the settings made in the action's user interface.

The method finally returns an output, which should be an object compatible with the second parameterized argument of the class (in our example above, a `String`). The method should always return an output, even if it is `null` or the input object.

Most action services operate on the input data given to them from the previous action. `Action(s)` includes an internal technology called *Smart-Cast* that converts, when possible, the output of an action into an acceptable input type for the next action. For example, if an action returning an `Image` is followed by an action that requires a `File`, `Action(s)` will create a temporary file containing the image result of the first action and pass it as the input of the second action. *Smart-Cast* can also convert a single object into an array to ensure action's compatibility. This means that, as a developer, you do not have to worry too much about the type of object you receive as an input or generate as an output.

The input object and the output object for an action are almost always array objects. This is why many action services implement `executeAction` using the following general approach:

1. Prepare an output array for later use.

2. Iterate through the elements of the input array and for each perform whatever operation is required and write the resulting data item to the output array.
3. Return the output array.

```

public String[] executeAction(final File[] input,
    final Parameters parameters)
    throws ActionExecutionException {
    if (input == null)
        return null;

    final String[] result = new String[input.length];

    for (int i = 0; i < input.length; i++) {
        try {
            // read the file content
            final FileInputStream fis =
                new FileInputStream(f);
            final FileChannel fc = fis.getChannel();
            final long sz = fc.size();
            final MappedByteBuffer bb =
                fc.map(FileChannel.
                    MapMode.READ_ONLY, 0, sz);

            final Charset charset =
                Charset.forName("ISO-8859-1");
            final CharsetDecoder decoder =
                charset.newDecoder();
            final CharBuffer cb = decoder.decode(bb);

            result[i] = cb.toString();
            fc.close();
            fis.close();
        } catch (final IOException e) {
            throw new ActionExecutionException(e);
        }
    }
    return result;
}

```

If your action service encounters an error that prevents it from proceeding, it should give information describing the error to *Action(s)*, which then stops executing the workflow and displays an error message.

To report errors, you must throw an `ActionExecutionException` exception. In the example above, you can see how the code stops and informs *Action(s)* if an I/O exception occurs when trying to read the file.

An action service has only two restrictions related to its implementation of `executeAction`:

1. It cannot return until it has completely finished whatever processing it has initiated. For instance, if an action instructs a camera to take a picture (an asynchronous process) it cannot

return from `executeAction` method until the picture is taken. So the action has to implement whatever blocking algorithm, timeout logic, or threading strategy is necessary until the picture is taken.

2. The second restriction has to do with *Action(s)*'s threading architecture. Because `executeAction` is run on a secondary thread, if they want to display a dialog window it must be done on the main thread. Especially, `javax.swing.JDialog` and `javax.swing.JOptionPane` objects require a dialog owner to display correctly. The `getWorkflow().dialogOwner()` method is adequate for this purpose; for example:

```
JOptionPane.showInputDialog (getWorkflow().dialogOwner(),
    "Input a text", "default text");
```

## Specifying Action Properties

The *Action(s)* application uses special properties in an information property list to get various pieces of information it needs for presenting and handling the action. This information includes:

- The name of the action
- The icon for the action
- The categories for the action
- The description of types of data the action accepts and the types of data it provides
- The description of the action

A properties file is a simple text file. You can create and maintain a properties file with just about any text editor.

The name of this file begins with the base name of your action, but start with a lower case, and ends with the `.properties` suffix. In your example the action class base name is `AddTextToFileName`, Therefore the properties file is called `addTextToFileName.properties`. This file contains the following lines:

```
# This is the default addTextToFileName.properties file
type=action
service=com.acme.AddTextToFileNameService
action.ui=com.acme.AddTextToFileNameServiceUI
description.title=Add Text To File Name
description.summary=This action add a text before or after \
the names of the files or folders passed into it.
description.icon=icon.png
description.categories=#FilesCategory
description.input=Files/Folders
description.output=Renamed Files/Folders
description.company.name=Acme
description.company.website=http://www.acme.com/
description.company.support=http://www.acme.com/support
description.version=1.0
description.copyright=© 2010 Acme
```

[Element Property Reference](#) describes the element properties, including their purpose, value types, and representative values.

Note that in the preceding file the comment lines begin with a pound sign (#). The other lines contain key-value pairs. The key is on the left side of the equal sign and the value is on the right. For instance, `type` is the key that corresponds to the value `action`.

The 3 first properties of the file are used by *Action(s)* to generate a new instance of the action when requires.

- **type:** the type of element that the properties file defines. For an action, the value of this property must always be `action`.
- **service:** the full name (including package address) of the action's service class.
- **action.ui:** the full name (including package address) of the action's user interface class, if any.

The other properties are used by *Action(s)* to displays the description in its lower-left view whenever the user selects the action. The description briefly describes what the action does and tells users anything else they should know about the action. Figure 14 shows what a typical description looks like.



Figure 14 | A sample action description

**Tip:** Because the values of some properties appear in the user interface, you should include translations of them using a located properties file for each localization you provide. See [Internationalizing the Action](#) for further information.

Because the description fits into a relatively small area of the *Action(s)* window, you should make it as concise and brief as possible. Ideally the user should not have to scroll the description view to see all of the text.

A description has several parts:

- **description.icon:** a 32 x 32 pixel image displayed in the upper-left corner of the description. In the properties file you should provide the relative path of the image within the package. Accepted formats are PNG, JPEG, GIF and BMP.
- **description.title:** the name of the action.
- **description.summary:** a sentence or two directly under the title that succinctly states what the action does.
- **description.input** and **description.output:** states respectively the type of data that the action accepts as an input and the type of data that the action produces as a result.

A description's icon, title, summary, input and output are required or strongly recommended.

## Internationalizing the Action

Most polished applications that are brought to market feature multiple localizations. These applications not only include the localizations – that is, translations and other locale-specific modifications of text, images, and other resources – but have been internationalized to make those localizations immediately accessible.

Java provides a very powerful method to developer to identify and separate culturally sensitive or locale-dependent objects from their source code: the resource bundle. As Oracle provides very good documentation to understand and implement internationalization in Java, this section will only focus what you must do to internationalize the action properties. For more information about internationalization in Java, visit <http://java.sun.com/docs/books/tutorial/i18n/index.html>.

### Localizing the Action Properties

*Action(s)* automatically detects and uses localized version of your action properties file if you provide them.

For each language translation of your action, you need to create a new version of your initial action properties.

For example, if you have created the default properties file *myAction.properties* to store all the properties in English, you will create a similarly named file to store the properties in French.

Localized properties files use a naming convention that distinguishes the potentially numerous versions of essentially the same properties. Each properties file name consists of a base name and an optional locale identifier. Together, these two pieces uniquely identify a bundle in a package.

The local identifier is determined by the language code and by country identifier. Some examples are provided in the following table:

Language	Country	Identifier
English (en)	United States (US)	_en_US
French (fr)	France (FR)	_fr_FR
French (fr)	Canada (CA)	_fr_CA
Japanese (jp)	Japan (JP)	_jp_JP
French (fr)	Any country	_fr

Using the *myAction.properties* example, the French version of the properties file would be named *myAction\_fr\_FR.properties*. The Canadian French version would be *myAction\_fr\_CA.properties*. You can also have a French translation independently of the country by providing a *myAction\_fr.properties* file.

The content of the localized file is similar to the content of the default file. It contains the same keys, but the values associated to them are translated into the destination language.

#### **myAction.properties**

```
# This is the default properties file
#[...]
description.title=Get Text from File
description.income=Files
description.outcome=Text
```

#### **myAction\_fr\_FR.properties**

```
# This is the French properties file
#[...]
description.title=Extraire le texte du fichier
description.income=Fichiers
description.outcome=Texte
```

You do not need to provide a translation of all entries from the default properties file, especially for entries that are not supposed to be translated (such as `element` or `action.ui`). If an entry cannot be found in the translation file, *Action(s)* will extract the value from the default properties file.

*Action(s)* will automatically identify and use the properties file that corresponds to the user language and country. If no translation is available, it will use the one in the default properties file.

## **Testing and Debugging the Action**

Testing and debugging an action you have created require you to create an *Action(s)*'s collection file and to load it into *Action(s)*. For more information about creating collection files, read the [Creating and deploying a collection file](#) section.

Once your collection available in *Action(s)*, you should create a workflow which contains your action. Run the workflow and observe how your action performs. To get better data from testing, consider the following steps:

- Use the *View Results* action between after each action you want to see the result.
- Use the *Pause*, *Step* and *Cancel* buttons to control the execution of the workflow.

Anything your action wrote to `stderr` (by calling the `System.out.printerr(String)` method) will show up in the *Action(s)* error log. The *Action(s)* error log can be display by choosing *Report Bug or Enhancement...* in the *Help* menu, then by clicking on the *View Error Log* button.

You can also use `stdout` (`System.out.println(String)` method) to write a trace into the standard output.

## Chapter 5 | Developing a Variable

In a lot of ways, developing a variable is very similar to developing an action. Both are based on the common element object foundation.

Variables have however a much more restricted range of functionalities as they are limited to provide a value on request and, for some of them, to store a given value for future access.

There are two types of variables that you might be willing to create:

- **A runtime variable:** a variable that allow access to data related to your application, such as a specific user folder.
- **A storage variable:** a variable that can store specific data used by your application and/or actions.

Runtime variables are not editable: they provide a value that is not stored nor managed by *Action(s)* and whose has its own life cycle. Value can be static (for example the path to the user's picture folder) or dynamic (for example the result of a SQL request on a database).

Storage variable is usually editable: they allow the user to set their content and retrieve it at anytime during the workflow execution cycle life. Once the execution ends, storage variables' original values are restored and all unsaved modification is lost.

In *Action(s)*, runtime variables are represented by a purple icon and storage variable by a blue icon.



*A storage variable icon    A runtime variable icon*

*Figure 15 | Variable's icons*

app.jbbres.com provides some abstract classes that you can easily implement to quickly develop your own variables. Those classes are available in the `com.jbbres.lib.actions.tools.elements` package of the *Action(s)* API.

↳ javadoc:

<http://app.jbbres.com/actions/developers/javadoc/com/jbbres/lib/actions/tools/elements/package-summary.html>

You'll become more familiar with many of the files that will constitute your variable in the sections that follow. But here is a summary of the more significant items:

- **variableName.properties:** The information property list includes the variable description. See [Specifying Variable Properties](#) for further information.

- **VariableName.java:** The variable java class. This class extends [com.jbbres.lib.actions.tools.elements.RuntimeVariable](#) if the variable is a runtime variable, or [com.jbbres.lib.actions.tools.elements.StorageVariable](#) if the variable is a storage variable.

## Creating the Project

Depending on the EDI you are using for developing in Java, the process to create a new variable might be different. Refer to your EDI documentation for more information on how to create a project.

You might need to include the *Action(s)* API as a known library of your tool in order to be able to develop and compile variable action. You can download this file at: <http://app.jbbres.com/actions/developers/>

## Creating a Runtime Variable

The `com.jbbres.lib.actions.tools.elements.RuntimeVariable` class is the default implementation of a runtime variable. It does most of the work for you as most of the functions and methods that this class should contain are pre-defined by this abstract class.

`RuntimeVariable` is a parameterized abstract class with a single type variable, representing the type of object that the variable provides when calling the `getValue()` method.

For example, if your variable provides a file object (`File`), the content of your variable class file will look like the following example:

```
package packageName;

import java.io.File;
import
com.jbbres.lib.actions.tools.elements.RuntimeVariable;
import com.jbbres.lib.actions.workflow.Workflow;

public class VRuntime extends RuntimeVariable<File> {

    public VRuntime(Workflow workflow) {
        super(workflow);
    }

    public File getValue() {
        return new File("my file path");
    }

}
```

The `getValue()` method should return the content of the variable. The `getValue()` method is called every time the user tries to access the

variable by adding a *Get Variable Value* action in its workflow. Its results are passed as an input to the next action.

## Creating a storage variable

The `com.jbbres.lib.actions.tools.elements.StorageVariable` class is the default implementation of a storage variable. It does most of the work for you as most of the functions and methods that this class should contain are pre-defined by this abstract class.

As for `RuntimeVariable`, `StorageVariable` is a parameterized abstract class with a single type variable, representing the type of object that the variable can store.

For example, if your variable is designed to store a date object (`Date`), the content of your variable class file will look like the following example:

```
package packageName;

import java.text.*;
import java.util.Date;
import com.jbbres.lib.actions.elements.*;
import com.jbbres.lib.actions.tools.elements.StorageVariable;
import com.jbbres.lib.actions.workflow.Workflow;

public class VStorage extends StorageVariable<Date> {

    private DateFormat dateFormat =
        new SimpleDateFormat("yyyyymmdd");

    public VStorage(Workflow workflow) {
        super(workflow);
    }

    public Parameters getParameters() {
        Parameters parameters = new Parameters();
        String formattedValue =
            dateFormat.format(getValue());
        parameters.setParameter("value", formattedValue);
        return parameters;
    }

    public void setParameters(Parameters parameters)
        throws InvalidParametersException {
        try {
            Date value = dateFormat.parse(
                parameters.getParameter("value"));
            this.setVariableValue(value);
        } catch (Exception e) {
            throw new InvalidParametersException(this,
parameters);
        }
    }
}
```

**Tip:** The default value setting functionality is not available in Action(s) 1.0 but will be added in an upcoming version.

The `getParameters()` and `setParameters(Parameters)` methods are used by *Action(s)* to save and restore the default value of the variable when saving and opening a workflow. The user can define the default value when he creates the workflow by editing the variable within the variable table at the bottom of the workflow. As a developer, you have to make sure that the `getParameters()` method returns the value currently stored in your variable, and that the `setParameters(Parameters)` replaces the current value of the variable by the one described in the `Parameters` argument. *Action(s)* makes sure that the `Parameters` argument received by the `setParameters(Parameters)` method is always similar to the one that the `getParameters()` method has produced.

## Creating the Variable's Renderer and Editor

In *Action(s)*, the variable list, at the bottom of the workflow area, display the variables that the user is currently using, their name, their type and their current value. If the variable is a storage variable, the user can double-click on the value and edit it. This allows him to define the default value of a variable for the workflow.

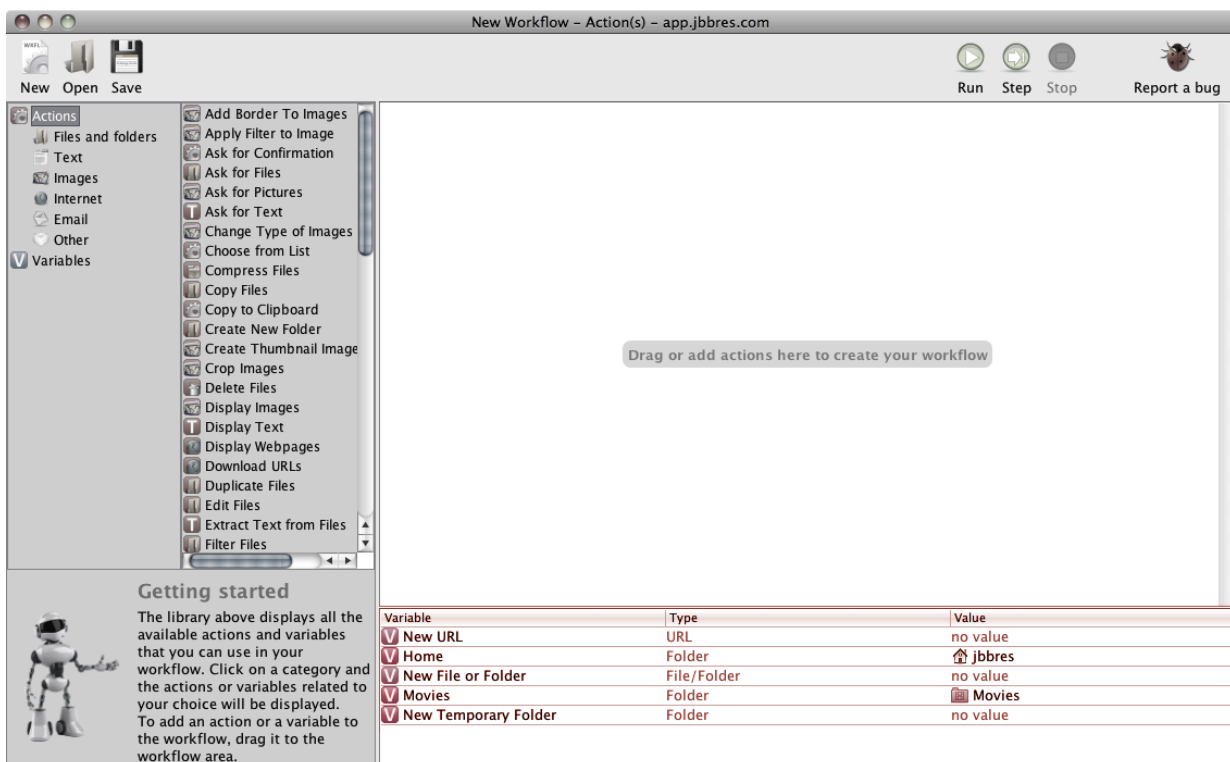


Figure 16 | The variable section in Action(s)

*Action(s)* renders the variable value differently depending on the value type. A `String` will be rendered as a label. A `File` will appear as the file icon followed by the file name. Common variable value types have a default renderer that *Action(s)* will use. If *Action(s)* failed finding a compatible renderer, it will display the result of the `toString()` method of the value object.

Similar process occurs for the value editor. When the user double-clicks on the variable value, *Action(s)* will try to identify the default editor

**Tip:** *Action(s) 1.0 does not support customized editors and renderers, but support will be added in an upcoming version.*

associated to the variable value type. However, if the identification failed, no alternative editor will be provided and the user will not be able to edit the value.

As a developer, you can define your own renderers and editors for the variables you create. You simply need to override the `getRenderer()` and `getEditor()` methods within your `Variable` object. The expected results of those methods are respectively a `javax.swing.table.TableCellRenderer` and a `javax.swing.table.TableCellEditor` object.

The *How to Use Tables* Java Tutorial presents how to create `TableCellRenderer` and `TableCellEditor`. See <http://java.sun.com/docs/books/tutorial/uiswing/components/table.html#editrender> for more information.

## Specifying Variable Properties

Specifying variable properties is very similar to specifying action properties. The *Action(s)* application uses special properties in an element information property list to get various pieces of information it needs for presenting and handling the variable. This information includes:

- The name of the variable
- The icon of the variable
- The description of the variable
- The description of types of data the variable can store or provide

As for an action properties file, the name of this file begins with the base name of your action, but start with a lower case, and ends with the `.properties` suffix. If your variable class base name is `MyVariable`, Therefore the properties file is called `myVariable.properties`. This file contains the following lines:

```
# This is the default myVariable.properties file
type=variable
description.title=New Date
description.icon=icon.png
description.summary=Creates a new variable that can \
contain a date.
description.content=Date
description.company.name=Acme
description.company.website=http://www.acme.com/
description.company.support=http://www.acme.com/support
description.version=1.0
description.copyright=© 2010 Acme
```

[Element Property Reference](#) describes the element properties, including their purpose, value types, and representative values.

The first property of the file is used by *Action(s)* to generate a new instance of the variable when requires.

- **type:** the type of element that the properties file defines. For a variable, the value of this property must always be `variable`.

The other properties are used by *Action(s)* to displays the description in its lower-left view whenever the user selects the variable.

Because the description fits into a relatively small area of the *Action(s)* window, you should make it as concise and brief as possible. Ideally the user should not have to scroll the description view to see all of the text.

A description has several parts:

- **description.title:** the name of the variable.
- **description.summary:** a sentence or two directly under the title that succinctly states what the variable represents.
- **description.content:** states the type of data that the variable can store or provide.

A description's title, summary and content are required or strongly recommended.

## Internationalizing the Variable

As variable properties files and action properties files use the same format, localization methods are the same for both type of files. *Action(s)* automatically detects and uses localized version of your variable properties file if you provide them.

For each language translation of your action, you need to create a new version of your initial action properties.

Each localized properties file name consists of a base name and an optional locale identifier. Together, these two pieces uniquely identify a properties file. For example the *myVariable.properties* contains the default properties of the variable written in English. The French version of the properties file would be named *myVariable\_fr\_FR.properties*. The Canadian French version would be *myVariable\_fr\_CA.properties*. You can also have a French translation independently of the country by providing a *myVariable\_fr.properties* file.

The content of the localized file is similar to the content of the default file. It contains the same keys, but the values associated to them are translated into the destination language.

*myVariable.properties*

```
# This is the default properties file
#[...]
description.title=New File
description.content=File
```

*myVariable\_fr\_FR.properties*

```
# This is the French properties file
#[...]
description.title=Nouveau fichier
description.content=Fichier
```

You do not need to provide a translation of all entries from the default properties file, especially for entries that are not supposed to be translated (such as `element`). If an entry cannot be found in the translation file, *Action(s)* will automatically take the value from the default properties file.

*Action(s)* will automatically identify and use the properties file that corresponds to the user language and country. If no translation is available, it will use the one in the default properties file.

## Testing and Debugging the Variable

Testing and debugging a variable you have created require you to create an *Action(s)*'s collection file and to load it into *Action(s)*. For more information about creating collection files, read the [Creating and deploying a collection file](#) section.

Once your collection available in *Action(s)*, you should create a workflow which contains your variable. Run the workflow and observe how your variable performs. To get better data from testing, consider the following steps:

- Use the *View Results* action between after each action you want to see the result.
- Use the *Pause*, *Step* and *Cancel* buttons to control the execution of the workflow.

Anything your variable wrote to `stderr` (by calling the `System.out.printerr(String)` method) will show up in the *Action(s)* error log. The *Action(s)* error log can be display by choosing *Report Bug or Enhancement...* in the *Help* menu, then by clicking on the *View Error Log* button.

You can also use `stdout` (`System.out.println(String)` method) to write a trace into the standard output.

## Chapter 6 | Creating and Deploying a Collection File

After having created your own actions and variables, the last step is to deploy them so everybody can use them into *Action(s)*. *Action(s)* collection file (.actc) format enables you to bundle all the files requires by your actions and variables to perform correctly into a single archive file.

Typically a collection file contains the class files and auxiliary resources associated with actions and variables.

The *Action(s)* collection file format is based on the Java Archive (JAR) file format. If you are familiar with JAR file creation, you will see that creating a collection file follow the same mechanisms. Even better, if you have created a JAR file containing your actions and variables, creating a collection file is as simple as changing its extension from .jar to .actc and adding a few lines in its manifest.

The collection file format provides many benefits:

1. Decreased download time: all your actions, variables and associated resources can be downloaded to a browser in a single HTTP transaction without the need for opening a new connection for each file.
2. Compression: the collection format allows you to compress your files for efficient storage.
3. Package versioning: a collection file can hold data about the files it contains, such as vendor and version information.
4. Portability: collection files can be used on any platform supporting *Action(s)*.

### Creating a Collection File

#### Writing the Collection Manifest

The manifest is a special file that can contain information about the files packaged in a collection file. It is used to identify the actions and variables available within the collection package. There can be only one manifest file in a collection file.

The manifest file is a simple text file. You can create and maintain it with just about any text editor.

The name of this file should be `MANIFEST.MF`. This file contains the following lines:

```
# This is the manifest file
Manifest-Version: 1.0
ActionsElements: com.acme.MyAction com.acme.MyVariable
```

Note that in the file the comment lines begin with a pound sign (#). The other lines contain key-value pairs. The key is on the left side of the equal sign and the value is on the right. For instance, `Manifest-Version` is the key that corresponds to the value `1.0`.

- The first mandatory key is `Manifest-Version`. Its value should always be 1.0 as the manifest is conform with version 1.0 of the manifest specification.
- The second key is `ActionsElements`. Its value is the full class name (including package name) of all elements (actions and/or variables) included in the collection file. Element class names should be separated by a space. You only need to include the element class name (the instance of `AbstractAction`, `RuntimeVariable` or `StorageVariable`). Services, User Interface and other associated classes and resources are automatically identified by *Action(s)* using the element properties file.

**Warning:** The text file from which you are creating the manifest must end with a new line. The last line will not be parsed properly if it does not end with a new line.

You can have additional `ActionsElements` key-value pairs if it is convenient for you. You simply need to add a number to the key name and increment it for each line. For example:

```
# This is the manifest file
Manifest-Version: 1.0
ActionsElements: com.acme.MyAction
ActionsElements2: com.acme.MyVariable
ActionsElements3: com.acme.AnotherAction
ActionsElements4: com.acme.AnotherVariable
```

### Creating the Collection File

Collection files are packaged with the ZIP file format. To create a collection file, you can use the Java Archive Tool provided as part of the Java Development Kit (JDK). Visits <http://java.sun.com/javase/downloads> to download the Java Development Kit.

If your IDE provides a built-in jar creation tool, you can generate a collection file by creating a JAR file and changing its extension from `.jar` to `.actc`. However, you will need to make sure that your IDE includes your manifest file into the JAR file created.

The basic format of the command for creating a collection file is:

```
jar cfm actc-file collection-manifest input-file(s)
```

The options and arguments used in this command are:

- The `c` option indicates that you want to create a can file.
- The `m` option indicates that you want to include your own manifest file within the collection file.
- The `f` option indicates that you want the output to go to a file (the collection file you're creating) rather than to standard output.
- `collection-manifest` is the name (or path and name) of the manifest file you have created for this collection.
- `actc-file` is the name that you want the resulting collection file to have (extension should be `.actc`).
- The `input-file(s)` argument is a space-separated list of one or more files that you want to be placed in your collection file. The `input-file(s)` argument can contain the wildcard `*` symbol. If any of the "input-files" are directories, the contents of those directories are added to the collection archive recursively.

### An Example

Let us look at an example. Let consider two actions `com.acme.ConvertToJpeg` and `com.acme.ConvertToPng`. Once compiled, the project is having this structure:

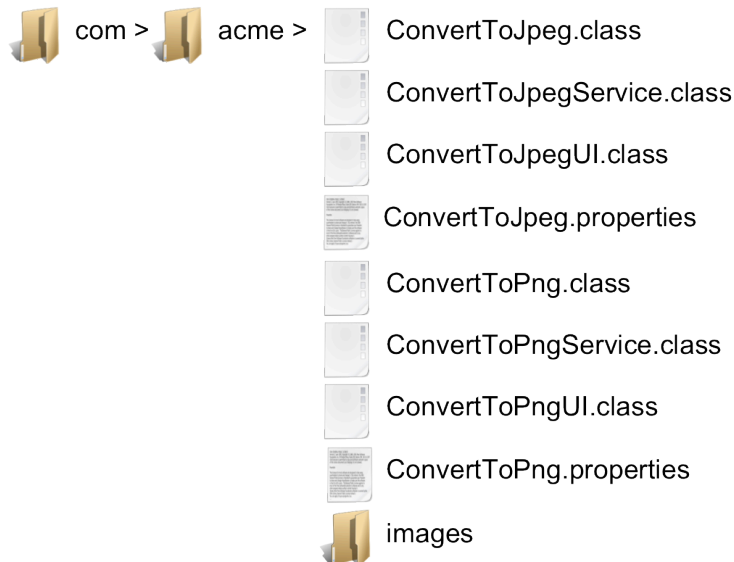


Figure 17 | Example of project structure

The images subdirectory contains JPEG and PNG images used by the actions.

The first step is to create the manifest file associated to the collection. It will be named `MANIFEST.MF` and will be located in the root directory. Its content would be:

```
# This is the manifest file for ImageConverter.actc
Manifest-Version: 1.0
ActionsElements: com.acme.ConvertToJpeg
ActionsElements2: com.acme.ConvertToPng
```

To package all the files into a single collection file named `ImageConverter.actc`, you would run this command from inside the root directory:

```
jar cfm ImageConverter.actc MANIFEST.MF com
```

The `com` arguments represent a directory, so the Jar tool will recursively place it and its content in the collection file. The generated collection file `ImageConverter.actc` will be placed in the current directory.

The Jar tool will accept arguments that use the wildcard `*` symbol. As long as there weren't any unwanted files in the root directory, you could have used this alternative command to construct the JAR file:

```
jar cfm ImageConverter.actc MANIFEST.MF *
```

## Deploying a Collection File

Your collection file created, your final step is now to make sure that people are able to download and install it on their computer. Depending on the audience you are targeting, the deployment strategy you might be willing to use might be very different.

The two deployment strategies describe in this section are the most common ones but you can easily adapt them to your own needs or creates your owns.

### Manual installation

The easiest deployment strategy you can implement is to let the final user installs the collection file himself.

From a user perspective, installing a collection in *Action(s)* is a very simple operation. Opening a collection file (.actc) will trigger the collection installation within *Action(s)*, copying the collection file within the user library. The user will immediately be able to see and use the actions and variables from the new collection in *Action(s)*.

As a developer, neither specific coding nor setting is required. You simply need to provide the collection file to the user, for example via a download section on your website.

**Tip:** The <http://app.jbbres.com/actions/more> webpage provides a free listing of 3rd parties *Action(s)* actions and variables. Visit it to get your collection file listed.

However, this strategy requires the user to have *Action(s)* installed on its computer, otherwise the collection file will not be recognized by the system and opening the file will result on an error message.

If you're providing the collection file via a website, you can easily add an *Action(s)* launch button so your user can download and install *Action(s)* instantly. Simply copy and paste the following lines within your webpage:

```
<script src="http://www.java.com/js/deployJava.js"></script>
<script>deployJava.launchButtonPNG='http://app.jbbres.com/jws/icons/actions-jws-launch-button.png';var url =
"http://app.jbbres.com/jws/app/actions.jnlp";
deployJava.createWebStartLaunchButton(url,
'1.6.0');</script>
```

### Automatic installation

If your strategy is to provide your actions and variables as an additional feature of your application, it is possible to create an automatic installation of your collection files within *Action(s)* without user intervention.

*Action(s)*' 3<sup>rd</sup> party collections are stored in the *Action(s)*' library folder, a specific directory on the final user computer. Any .actc file stored in this directory is automatically loaded into the library next time the user starts *Action(s)*.

The *Action(s)*' library folder is located in the application library folder within the user's directory. Depending on the operating system used by your user, this folder might be located at a different path. The common paths are:

- **Windows XP:** C:\documents and settings\%username%\local settings\application data\app.jbbres.com\Actions\plugins\
- **Windows Vista & Windows 7:**  
C:\Users\%username%\AppData\Local\app.jbbres.com\Actions\plugins\
- **Mac OS X:**  
~/Library/Preferences/app.jbbres.com/Actions/plugins/

By providing a script copying your collection file within the adequate folder you deploy your actions and variables transparently for the final user.

## Chapter 7 | Element Property Reference

When you develop an element for *Action(s)*, one of the steps is specifying the properties of the element in its information property file (see [Specifying Action Properties](#) and [Specifying Variable Properties](#)). *Action(s)* uses these properties for various purposes, among them creating the element instance and getting the element name, icon and description.

You must specify some of the *Action(s)* properties described below in an element information property file (.properties), and optionally may specify the others.

### Property keys and values

#### **type**

**This property is required.** A string that specifies the concrete type of the element. Accepted values are:

- `action`: if the element is an action.
- `variable`: if the element is a variable

This property is used to display the element within the correct section in the library (*Actions* or *Variables*).

Example:

```
type=action
```

#### **description.title**

**This property is required.** A string giving the name of the element to be displayed in the *Action(s)* user interface. Example names are "Add Attachments to Front Message", "Copy Files", and "Profile Executable".

Example:

```
description.title=Copy Files
```

See the guidelines for naming elements in [Design Guidelines for Action\(s\)](#).

#### **description.icon**

A string that specifies a file that contains the 32-by-32 pixel image that *Action(s)* displays to the left of the element name in its user interface. The icon name string should include the extension. It refers to a custom image file (PNG, JPG, JPEG, GIF or BMP) in the collection file. Its location is relative to the properties file.

Example:

```
description.icon=icons/copy_files.png
```

See the guidelines for creating element icons in [Design Guidelines for Action\(s\)](#).

#### **description.summary**

**This property is required.** A string giving a succinct description of what the element does. It appears directly under the element title.

Example:

```
description.summary=This action copies the specified \
files or folders to the specified location.
```

#### **description.input**

**For actions only.** A string giving the types of data the action accepts. The subheading "input:" precedes this text in the displayed description.

Example:

```
description.input=Files/Folders
```

#### **description.output**

**For actions only.** A string giving the types of data the action provides. The subheading "result:" precedes this text in the displayed description.

Example:

```
description.output=Files/Folders (copied files)
```

#### **description.content**

**For variables only.** A string giving the types of data the variable can store or provide. The subheading "data:" precedes this text in the displayed description.

Example:

```
description.content=Files
```

#### **description.version**

A string giving the version number of the element. The subheading "version:" precedes this text in the displayed description.

Example:

```
description.version=1.0.5
```

#### **description.categories**

A string that *Action(s)* uses to group the action with similar elements in terms of their effects or the objects they operate on. *Action(s)* presents categories in its user interface and also uses an element category as a search criterion.

Category names are simply strings, with values such as Find, Music, Pictures, System, Terminal, Text, and Utility. You could also specify names for new categories. Categories should be separated by semicolons (;).

Category codes are also available for major categories. You should use the code names shown in the table below.

Code Name	Category
#EmailCategory	Email
#FilesCategory	Files and Folders
#ImagesCategory	Images
#InternetCategory	Internet
#OtherCategory	Other
#TextCategory	Text

Example:

```
description.categories=System;#FilesCategory
```

#### **description.support.website**

A string giving the URL of the element support website.

Example:

```
description.support.website=http://www.acme.com/support
```

#### **description.company.name**

A string giving the name of the company providing the element.

Example:

```
description.company.name=Acme Inc.
```

#### **description.company.website**

A string giving the URL of the company website.

Example:

```
description.company.website=http://www.acme.com/
```

#### **description.copyright**

A string giving the copyright notice for the element. The subheading "copyright:" precedes this text in the displayed description.

Example:

```
description.copyright=© 2010 Acme Inc.
```

#### **service**

**For instance of `AbstractAction` and `AbstractVariable` only – This property is required.** A string giving the full java name of the service class for this element.

Example:

```
service=com.jbbres.actions.files.CopyFilesService
```

#### **action.ui**

**For instance of `AbstractAction` only.** A string giving the full java name of the UI class for this element.

Example:

```
action.ui=com.jbbres.actions.files.CopyFilesUI
```

#### **action.canShowWhenRun**

**For instance of `AbstractAction` only.** A Boolean (accepted values are `true` or `false`) that controls whether the Show When Run feature is enabled for the action. It overrides any setting done within the action code.

Setting this property to `true` causes the *Show When Running the Workflow* check box to be displayed in the top-left corner of the action view. When the check box is selected, the view is extended to expose a *Prompt:* text field.

Properties `action.showWhenRun.selected` and `action.showWhenRun.prompt` can be used to define the default state of the *Show When Running the Workflow* check box and the default prompt text.

Setting `action.canShowWhenRun` to `false` removes *Show When Running the Workflow* check box.

Example:

```
action.canShowWhenRun=true
```

#### `action.showWhenRun.selected`

**For instance of `AbstractAction` only.** A Boolean (accepted values are `true` or `false`) that controls whether the *Show When Running the Workflow* is by default selected or not for the action. It overrides any setting done within the action code.

Example:

```
action.showWhenRun.selected=false
```

#### `action.showWhenRun.prompt`

**For instance of `AbstractAction` only.** A String giving the default prompt text for the Show When Run feature.

Example:

```
action.showWhenRun.prompt=Select a destination directory
```

## Revision History

This table describes the changes to *Action(s) Developer Guide*.

Version	Notes
1.0.1 (20/10/2010)	Update library folders.
1.0 (12/06/2010)	New document that explains how to create actions-loadable files that perform discrete task for the Action(s) application.