Create your Own Actions

A Quick Reference Guide

Action(s), the powerful application by app.jbbres.com, helps you streamline repetitive everyday manual tasks quickly, efficiently, and effortlessly without programming. You can easily automate tasks such as renaming a large group of files, manipulating dozens of images, or download new version of documents from the Internet. Once automated, you can repeat those tasks again and again.

The basic building block in *Action(s)* is an **action**. Each action is designed to perform a single task, such as downloading images on a Web page or copying files from one folder to another. Instead of being a do-it-all tool, an action is purpose-designed to perform a single task well. The power in *Action(s)* comes from sequencing actions into a **workflow**. By combining several actions into a workflow, you can quickly and easily accomplish a specific task that no one action can accomplish on its own. This idea of building functionality from small, discrete components dates back to the early days of computers programming. The breakthrough that *Action(s)* brings is allowing you to assemble small tools in an easy, intuitive graphic user interface. Anyone with development skills is a prime candidate for creating actions, and in most cases they will be able to do it much faster than scripting by hand.

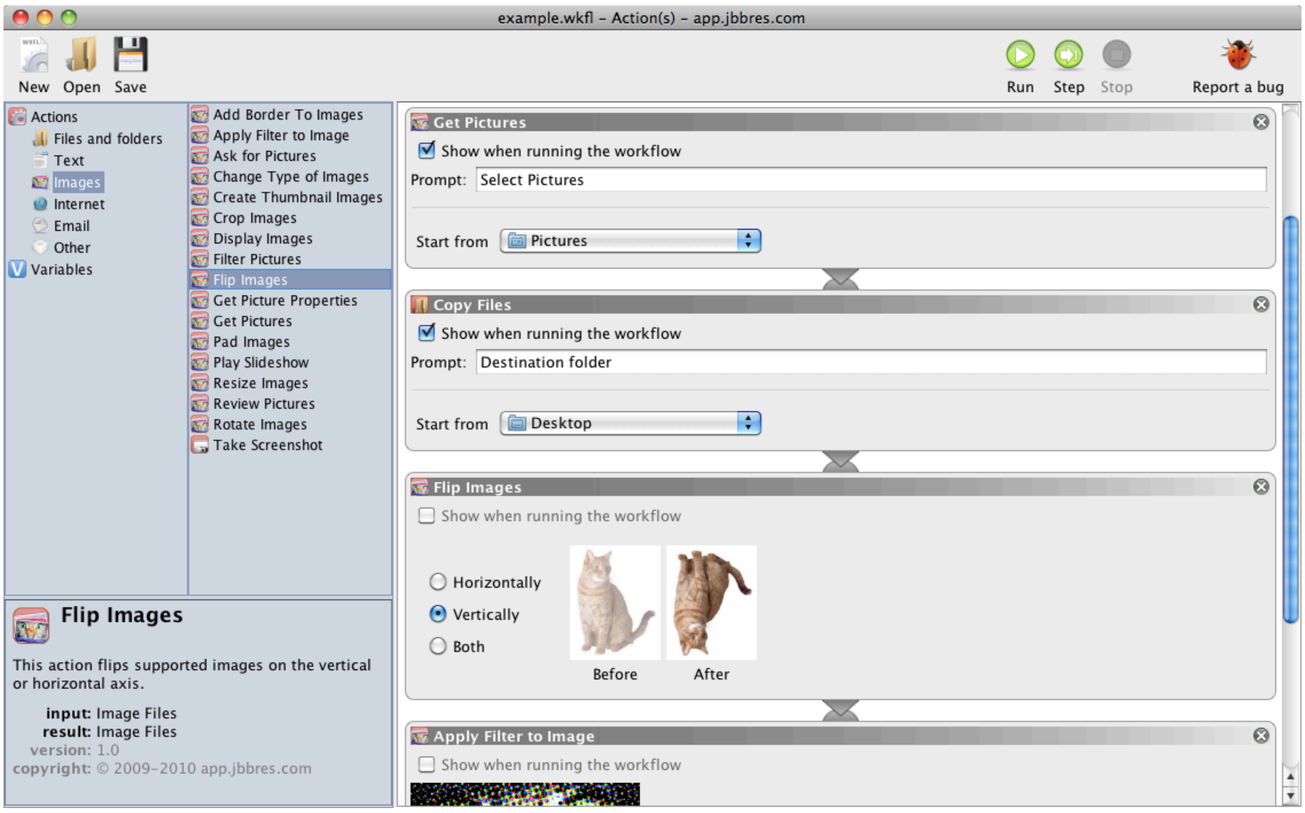
The beauty of an action is that it can take advantage of many sources of functionality. Actions can access the functionality of the core Java frameworks, leverage the command-line tool environment, or take advantage of the specialized features of a particular application. Even better, it's easy to create your own actions with Java. If you are an application developer, creating actions for your users is a great opportunity: it allows you to provide extended functionality in small, bite-sized units for use in their workflows.

Creating great actions and workflows can help expose key features of your application, making it even more valuable to your users. Creating actions and workflows that access functionality in other applications on a user's system is another tremendous opportunity to extend the value of your application.

This article shows you how *Action(s)* works, how you can build your own actions to use in *Action(s)*, and how you can distribute those actions to users.

Using Action(s)

Action(s) comes preloaded with a large library of actions. These actions expose functionality built into the operating system, such as dealing with files and directories or taking a screenshot. There are also actions that let you write shell scripts. You can use these actions to rename a group of files or to resize an image. This library of actions is accessed and organized on the left hand side of the *Action(s)* window.



Flip Images

This action flips supported images on the vertical or horizontal axis.

input: Image Files
result: Image Files
version: 1.0
copyright: © 2009–2010 app.jbbres.com

Get Pictures
[✓] Show when running the workflow
Prompt: Select Pictures
Start from: Pictures

Copy Files
[✓] Show when running the workflow
Prompt: Destination folder
Start from: Desktop

Flip Images
[ ] Show when running the workflow
Horizontally [✓ Vertically] Both
Before
After

Apply Filter to Image
[ ] Show when running the workflow

To assemble a workflow from a set of actions, simply drag-and-drop the actions from the library into the sequence in which you want them to run. Each action in the workflow corresponds to an individual step that you would need to perform manually. Each action comes with a small GUI panel that lets you tweak its options and settings. *Action(s)* shows these panels connected, along with the types of data that are flowing from one action to another.

Once you have created a workflow, you can save it as an editable document. This lets you re-open, edit, and run the workflow inside of *Action(s)*. Another option is to save the workflow as non-editable document (called an executable workflow). This will create a file that will run the workflow when you double-click on it.

Creating an Action

As you can see, *Action(s)* is a powerful and flexible tool for users of all experience levels. If you are an application developer, it should be clear that letting users access the functionality of your application from

Action(s) will enhance its value to them. Providing Actions is a powerful additional path to the features and services that make your products unique.

Actions are easy to create. A public API is available for creating Java based actions. With this API, you can create **actions**, **variables** and loadable bundles (called **collections**) that *Action(s)* uses.

There are three general types of actions that you can create:

- Actions that control an application to get something done.
- Actions that use system frameworks or other system resources to get something done.
- Actions that perform a "bridge" function, such as prompting the user for input, writing output to disk, or converting data from one type to another.

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

API: http://app.jbbres.com/actions/developers/

javadoc: http://app.jbbres.com/actions/developers/javadoc/

To illustrate this article, we will create an action that, once placed in a workflow, returns the path of the file it receives from the previous action. The code of this example is available within the *Action(s)* API.

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

- **`getFileName.properties`**: The information property list includes the action description.
- **`GetFileName.java`**: The action class. This class extends `com.jbbres.lib.actions.tools.elements.SimpleAction`.

Creating the Project

Depending on the EDI you are using for developing in Java, the process to create a new project 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.

Creating the Action Class

The `com.jbbres.lib.actions.tools.elements.SimpleAction` class is a very basic implementation of the action class. 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.

`SimpleAction` 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.

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 `Void`. On the opposite, if the action is able to handle any type of input data, the first parameterized argument will be `Object`.

These conventions also 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 `SimpleAction<Void, Void>`. *Action(s)* will recognize this specific case and will route the flow of data around the action.

In our example, we expect to receive a file and return a text. However, most of the actions return and receive arrays, so we know that we are most likely to receive an array of files (`File[]`) instead of a single file. Having say that, your action will have, for each item in the array, to return its path, so the result will be an array of text (`String[]`).

Furthermore, if the action preceding ours in the workflow does not return an array of files but a single file, *Action(s)* will automatically transform the result into an array (with a single item) so we do not have to worry about the different type of data we might have to deal with.

```
public class GetFileName
    extends SimpleAction<File[], String[]> {

    /**
     * Instantiates a new action.
     */
    public GetFileName(Workflow workflow) {
        super(workflow);
    }
}
```

Constructing the User Interface

The user interface of your action is displayed in the workflow definition panel in *Action(s)*. The user can use it to define the settings of the action.

The user interface must be provided by the `getUI()` method in the action class.

In our example, the UI will provide a combo box allowing the user to choose if he wants the action to return the full path of the files that the action received, or only their file names:

```
JPanel uiPanel = new JPanel();
JLabel resultLabel = new JLabel("Result:");
String[] pathTypes =
    {"Full path", "File name only"};
JComboBox pathTypeComboBox =
    new JComboBox(pathTypes);

/**
 * Instantiates a new action.
 */
public GetFileName(Workflow workflow) {
    super(workflow);
    uiPanel.add(resultLabel);
    uiPanel.add(pathTypeComboBox);
}

/**
 * The action UI.
 */
public Component getUI() {
    return uiPanel;
}
```

The User Interface can be created using AWT, Swing or any other compatible toolkit.

Implementing the `execute` method

The most important step in creating an action is writing the `execute` method that implements the logic for your action.

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

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.

```
public String[] execute(File[] input)
    throws ActionExecutionException {
    if (input == null)
          return null;

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

    for (int i = 0; i < input.length; i++){
          if (pathTypeComboBox.getSelectedItem()
                      .equals("Full path")){
                result[i] = input[i].getPath();
          } else {
                result[i] = input[i].getName();
          }
    }
    return result;
}
```

Saving and Restoring User Settings

The `getParameters()` and `setParameters(Parameters)` methods generate and set the parameters associated to the User Interface. The `Parameters` object that the `getParameters()` method returns is used by *Action(s)* to store the state of the action within the .wkfl file when the user saves the workflow. The same object is given to the `setParameters(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.

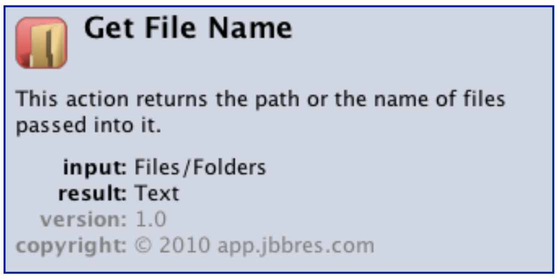
In our example, the `getParameters()` method must store in a `Parameters` object the value selected by the user in the UI combo box, and the `setParameters(Parameters)` method should restore the combo box state based on the value stored in the Parameters object it received as an argument.

```
/**
 * Returns the action settings as
 * a Parameter object.
 */
public Parameters getParameters() {
    Parameters parameters = new Parameters();
    parameters.setParameter("result.path.type",
          (String)pathTypeComboBox.getSelectedItem());
    return parameters;
}

/**
 * Sets the action settings from a
 * Parameters object.
 */
public void setParameters(Parameters parameters)
    throws InvalidParametersException {
    String pathType =
          parameters.getParameter("result.path.type");
    pathTypeComboBox.setSelectedItem(pathType);
}
```

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 action. This information includes:


Get File Name

This action returns the path or the name of files passed into it.

input: Files/Folders
result: Text
version: 1.0
copyright: © 2010 app.jbbres.com

- The name of the action
- The icon for the action
- The category 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 our example the action class base name is `GetFileName`, therefore the properties file is called `getFileName.properties`. This file contains the following lines:

```
type=action
description.title=Get File Name
description.summary=This action returns the path or \
the name of files passed into it.
description.categories=#FilesCategory
description.input=Files/Folders
description.output=Files/Folders
```

Those 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.

A description has several parts:

- **`type`**: the type of element that the properties file defines. For an action, the value of this property must always be `action`.
- **`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 produce as a result.

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

Deploying actions

After having created your own actions, 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 to perform correctly into a single archive file.

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

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.

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 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:

```
Manifest-Version: 1.0
ActionsElements:
com.jbbres.examples.actions.GetFileName
```

***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.*

All 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 actions included in the collection file. If you want to declare more that one action, each action class name should be separated by a space.

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).

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 collection 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 JAR archive recursively.

The command to create a collection file names `GetFileName.actc` of our project is:

```
jar cfm GetFileName.actc MANIFEST.MF bin
```

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 can use are very different.

The easiest deployment strategy you can implement is to let your 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 from the new collection his workflows.

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. 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.

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

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 requiring an user intervention.

Action(s)' 3rd party collections are stored in the *Action(s)*' library folder, a specific directory on the 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.

Action Design Guidelines

Since many actions are used with each other in *Action(s)*, they should have a consistent look and feel. Probably the most important piece of advice is to keep an action as simple and as discrete as possible. An action should not attempt to do too much. An action that resembles a multi-tool will be too specialized to be useful and will limit the user. If the functionality that you want to provide is complicated, consider breaking it up into many small independent actions.

The user interface should also be kept as simple as possible. You should try to minimize the use of vertical space. In particular, use combo list instead of radio buttons and avoid tab views. Probably the best strategy to adopt is to look at the other actions in *Action(s)*. The more you can make yours look like the ones that are already there, the better off your users will be.

Conclusion

As you have seen, *Action(s)* is a powerful tool. It let users take control of tasks that they need to do every day. And it also let developers allow users to extend the functionality of their applications in ways that they could never anticipate.

For More Information

- Using *Action(s)* to expand the market of your software
- *Action(s)* API
- Action(s) Programming Guide
- *Action(s)* JavaDoc