Actions Support User Guide

Early Preview Edition - Release 2.0.1

Table of Contents

Introduction

The ActionsSupport NBM is a set of classes which can be used by other NetBeans plugins, to allow the creation of Actions, with the ability to run external programs.

This document is designed for developers of NetBeans Modules who wish to integrate these features into their code.

These Action can execute:

- NetBeans methods
- Methods coded within the plugin
- External programs which can be executed in a CLI-like manner, allowing the use of such programs from the NetBeans UI

DynamicActions and DynamicAsyncActions

A Dynamic Action is an Action which can be enabled or disabled and is not displayed when disabled.

These are two classes of Dynamic Actions:

- The base class (DynamicAction) which mirrors the Action class.
- The DynamicAsyncAction class which mirrors the DynamicAction class, running the onClick method in a seperate thread, so being suitable for any action which takes a period of time (which would otherwise block the processing of UI events).

DynamicActions can be associated with a particular node, either being for:

- All projects of a particular type
- A specific project, as defined by a properties file using DynamicAyncActions.

NodeActions

NodeActions supports the creation of Actions for a node, by enabling the creation of DynamicCLIActions using a properties file.

NodeActions observes the node folder containing the property file, ensuring the actions are updated whenever changes occur to the properties file.

Additional files within the node folder can be observed, so that changes can trigger updates to any associated objects.

NodeActions provides a method for assembling node actions, combining various sources of actions to create the final dynamic actions array required by a node definition.

ActionsSupport in action

NodeActions example

NodeActions creation:

- accepts the filename of a properties file
- enables the creation of DynamicAyncActions from the information in a properties file
- monitors the properties file for changes and refreshes the actions as necessary
- enables support for assembling a node's actions

In the node constructor, add:

```
XXXXProject(FileObject dir, ProjectState state) {
    .....
    dynamicactions = new NodeActions(dir, "projectactions", "Build"); ①
}
```

① create a DynamicActions object, which will observe a file *projectactions.properties* within folder *dir* for changes. It will use the content of that file to define DynamicAsyncAction objects, updating these actions whenever the file changes. The third parameter defines the root of the default OutputWindow tab which can be used by this node's Actions. The actual tab title is a concatonation of the root and the node's folder name.

In order to be able to create the node's actions, NodeActions must be made aware of all other actions required. It will create the list containing three sections, with separators. Any of the sections may be empty.

The sections are:

- Basic node actions, typically the standard NetBeans actions which a node would want to include
- Node actions, typically DynamicAsyncActions which are node type specific (ie not defined in the node's properties file)
- DynamicAsyncActions which are created using the node's property file

The third group are defined by the NodeActions constructor, while the others must be passed to NodeActions (using setNodeBasicActions and setNodeActions methods).

```
dynamicactions.setNodeBasicActions(
                CommonProjectActions.renameProjectAction(),
                CommonProjectActions.copyProjectAction(),
                CommonProjectActions.closeProjectAction() ①
        );
dynamicactions.setNodeActions(
                new DynamicAsyncAction("Bake")
                        .onAction(
                            () -> new CLIExec(projectDir,"jbake -b")
                                    .stderrToOutputWindow()
                                    .stdoutToOutputWinow()
                                    .executeUsingOutput("Baking")) ②
                );
```

① define 3 actions, in this case standard Netbeans Project Actions.

② define 1 DynamicAsyncAction, which will appear in all projects of this type.

The Node should use the NodeActions getAllNodeActions method to obtain it's Action array. This will combine all valid actions which have been made known to NodeActions.

```
public Action[] getActions(boolean arg0) {
    return dynamicactions.getAllNodeActions(); ①
}
```

① Call NodeActions getAllNodeActions method to get a merged list of all actions.

Use of the DynamicAction/DynamicAsyncAction classes

While the majority of uses of this module will be via the NodeActions API, it is possible to use the DynamicAction classes independently.

DynamicAction examples

Add a DynamicAction to a node's actions.

```
@Override
public Action[] getActions(boolean arg0) {
    return  new Action[]{
                ...
                null,
                new DynamicAction("do something")
                    .onAction(()->do_something()) ① ②
            });
}
```

① Create the DynamicAction

② …and add it to the nodes's actions. The action is enabled so needs no further initialisation.

DynamicAsyncAction example

Add a DynamicAsyncAction to a node's actions.

```
@Override
public Action[] getActions(boolean arg0) {
    return  new Action[]{
                ...
                null,
                new DynamicAsyncAction("do something")
                    .onAction(()->do_something()) ① ②
            });
}
```

① Create the DynamicAsyncAction

② …and add it to the nodes's actions. The action is enabled so needs no further initialisation.

The ActionsSupport API

NodeActions

NodeActions creates DynamicAsyncActions for a node, using a properties file to define these actions.

NodeActions observes the node folder containing the properties file, ensuring the actions are updated whenever changes to the properties file occur.

Additional files within the node folder can be observed, so that changes can trigger updates to any associated objects.

NodeActions provides a method for assembling node actions, combining various sources of actions to create the Actions array required by a node definition.

public static enum FileChangeType {CREATED, CHANGED, RENAMEDTO, RENAMEDFROM, DELETED}

The types of file changes which are observed by NodeActions.

Constructor

public NodeActions(FileObject filefolder, String actionpropertiesfilename, String tabprefix)

Create a NodeActions object:

- *filefolder* The filefolder containing the actionsproperties file (the node folder)
- *actionpropertiesfilename* The filename of the actions property file
- *tabprefix* defines the root of the default OutputWindow tab which can be used by this node's Actions. The actual tab title is a concatination of the root and the node's folder name

Methods

public final void registerFile(String filename, String fileext, Consumer<FileChangeType> callback)

Register an addition file to be observed. This file is expected to be present (when existing) in the same folder as the actions properties file:

- *filename* The filename of the file to be observed
- *fileext* The file extension of the file to be observed
- *callback* The method to be called where a change is observed on this file

public void setBasicNodeActions(Action... basicnodeactions)

define the set of basic actions which are to be included in the Node's actions popup.

public void setNodeActions(DynamicAsyncAction... nodeactions)

define the set of actions which are particular to all projects of this node's project type.

public Action[] getAllNodeActions()

Return an array of enabled actions for the node. This includes the basicNodeActions, the NodeActions and any DynamicAsyncActions that were defined by the properties file.

Structure of the Properties file

The properties file must include a property *COMMANDCOUNT* which indicates the number of actions being defined in this file.

Each action being defined must have two or more property lines defined:

- *n.label* the label displayed in the popup list (required)
- *n.command* the CLI command to be executed when the action is selected (required)
- *n.inputfrom* defines the source for the STDIN stream (optional):
  - *file* a file is to be used as the STDIN stream (see inputfile)
  - *ui* the Output window is to be the STDIN stream, providing keyboard input
  - *noinput* no input is provided for STDIN (default)
- *n.inputfile* defines the input file for STDIN (optional, unless *inputfrom* is set to *file*)
- *n.needscancel* = *yes* add a cancel button to the Output Window sidebar (cancels the Process created by the DynamicAsyncAction)

note that *n* must be an integer between 1 and *COMMANDCOUNT*.

```
1.label = Bake
1.command = jbake -b"
2.label = Copy NBPCG User Guide (html format)
2.command = bash -c "./copynbpcguguide ${NODEPATH}
~/DocumentationProjects/NBPCGDocumentation/UserGuide"
COMMANDCOUNT = 2
```

DynamicAction

DynamicAction is the basic class of the Actions Support Plugin. A DynamicAction is an implementation of the Netbeans AbstractAction class.

A DynamicAction is an Action which can be enabled or disabled, it is not displayed when disabled.

Constructor

public DynamicAction(String label)

Construct a DynamicAction which is enabled:

- *label* The label displayed when this DynamicAction is displayed.

Methods

public DynamicAction onAction(Runnable action)

Set the Action method.

- *action* the method to be run when this action is selected.

public DynamicAction enable(boolean enable)

Enable/Disable this Action.

DynamicAsyncAction

DynamicAsyncAction is a extension of DynamicAction. It's action method runs the Action method on another thread, so not potentialling blocking the Action calling thread.

The constructor and public methods for this Action is identical to the DynamicAction class.

Constructor

public DynamicAction(String label)

Construct a DynamicAction which is enabled:

- *label* The label displayed when this DynamicAction is displayed.

Methods

public DynamicAction onAction(Runnable action)

Set the Action method.

- *action* the method to be run when this action is selected.

public DynamicAction enable(boolean enable)

Enable/Disable this Action.

CLIExec

CliExec is a class which is both a builder for CLI type commands and also its execution. Builder methods include configuration and input and output file options. Execution methods are available to utilise an Output Window tab (either unique or shared), in addition to the basic execution option.

Constructor

public CLIExec(FileObject dir, String clicommand)

Create a CLIExec object:

- *dir* The node folder - will be used as the working directory when executing this object.
- *clicommand* the command to be run when executing this command.

The clicommand includes limited parameter substitution:

- ${NODEPATH} - is replaced with the node folder path.

The clicommand has to be parsed into CLI "phrases", such as options, option parameters, filenames etc. As this constructor never can have a full understanding of every command's syntax, it employs a basic parsing technique (breaking on word breaks ie white space). This works in the vast majority of cases, but there are times where the CLI "phrase" should include multiple words. If this is the case, the phrase should be written with enclosing double quotes. These quotes will be stripped from the generated command prior to it being passed to ProcessBuilder.

Process stdin configuration

public CLIExec stdin(FileObject fo)

- *fo* Use the fileobject as stdin for the Process

public CLIExec stdin(File file)

- *file* Use the file as stdin for the Process

public CLIExec stdin(InputStream is)

- *is* Use the inputstream as stdin for the Process. A thread will be created to copy from the inputsteam to the stdin for the Process.

public CLIExec stdin(Reader rdr)

- *rdr* Use the rdr as stdin for the Process. A thread will be created to copy from the rdr to the stdin for the Process.

public CLIExec stdinFromOutputWindow()

Use the input from the Output Window as stdin for the Process. A thread will be created to copy from the input to the stdin for the Process.

public CLIExec stdinFlushPeriod(int millisecs)

aaa

public CLIExec stdinFlushPeriod()

aaa

Process stdout configuration

public CLIExec stdout(FileObject fo)

aaa

public CLIExec stdout(File file)

aaa

public CLIExec stdout(OutputStream os)

aaa

public CLIExec stdout(Writer wtr)

aaa

public CLIExec stdoutToOutputWindow()

aaa

Process stderr configuration

public CLIExec stderr2stdoutput()

aaa

public CLIExec stderr(FileObject fo)

aaa

public CLIExec stderr(File file)

aaa

public CLIExec stderr(OutputStream os)

aaa

public CLIExec stderr(Writer wtr)

aaa

public CLIExec stderrToOutputWindow()

aaa

Other configuration methods

public CLIExec needsCancel()

aaa

public CLIExec preprocessing(Consumer<OutputWriter> preprocessing)

aaa

public CLIExec postprocessing(Consumer<OutputWriter> postprocessing)

aaa

Execution Methods

public void executeUsingOutput(String startmessage)

aaa

public void executeUsingOutput()

aaa

public void execute()

aaa

public void printerror(String phase, String exmsg)

aaa

Utilising the ActionsSupport NBM

The following provides a quick reference to key information needed to use the ActionsSupport NBM.

Using the Library in a Maven build

The NBM must be added as a dependency in your POM. To ensure Maven can find it in the GitHub Maven repository, you will have add the following in your POM:

```
<repositories>
    <repository>
        <id>github</id>
        <url>https://maven.pkg.github.com/The-Retired-Programmer/NetBeansNBMs</url>
    </repository>
</repositories>
```

and then add a dependency in your POM

```
    <dependency>
        <groupId>uk.theretiredprogrammer</groupId>
        <artifactId>actionssupport</artifactId>
        <version>{release-version}</version>
    </dependency>
```

Installing the NBM into Netbeans

To be able to use the Plugin it must be downloaded and then installed in Netbeans.

There is two ways to download the NBM:

- Releases of this product are stored in a Maven repository on GitHub. It is possible to download an NBM from the Package available .
- However assuming you have added the dependency to you POM and have compiled your code, a copy of the NBM will have been down loaded to your local Maven repository (by default ~/.m2).

Once you have a downloaded copy on your machine, you follow the standard Netbeans process to install the downloaded file (Tools>Plugins)

Links

- Releases are stored in a Maven repository on GitHub
- Source Code is stored in a Git repository on GitHub as part of a monorepro containing NBMs.

- Documentation can be found on the TRP website.
- Documentation source is stored in a Git repository on GitHub as part of a monorepro containing documentation.
- Standard Development Process documentation for The-Retired-Programmer products can be found on the TRP website.