Actions Support User Guide

Release 2.1.1

Table of Contents

Introduction

The ActionsSupport NBM is a library which can be used by other NetBeans plugins, to allow the creation of Actions.

One of its major actions is to run external programs. This feature make it easy to integrate tool chains used with particular file types and/or new project types.

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

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 separate thread, so being suitable for any action which takes a period of time (which would otherwise block the processing of UI events).

These Actions 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

NodeActions

NodeActions supports the creation of Actions for a node 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 all node actions, combining various sources of actions to create the Actions array required by a Node.

Structure of the Node Actions Properties file.

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

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

- n.label - defines the label displayed in the popup list (required).
- n.command - defines the CLI command to be executed when the action is selected (required).
- n.commandargs - defines the CLI command arguments to be used when the action is selected (optional). This text is subject to parameter substitution prior to use. The only substitution defined is \${NODEPATH}.
- n.tabname - defines the IO Tab name to be used in the Output Window. Optional, if not defined then the label property will be used for the tab name.
- n.killcommand - defines the command word for the 'kill the external process' action (optional). Commands are typed into the IO Tab.
- n.cleartab = "every execution" - clears the tab between every usage (optional).

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

NbCliDescriptor

NbCliDescriptor is used to define the configuration of the External process that will run. The description includes:

- The CLI command and its arguments.
- The IO Tab configuration (optional)
- file connections for the STDIN, STDOUT and STDERR streams which are accepted or produced by the external program. This can including use of the IO Tab to accept either of the two stream produced.
- Other configuration options.

SaveBeforeAction

Supports the saving of modified files prior to running an external process.

Two Use Cases are supported:

- File is not within a project. In that case the file is saved if it is modified.
- File is within a project. In that case, the project properties file can be used to obtain a save_before_action setting:
  - NO - don’t save
  - YES - save the file which is the subject of the action (either file node or editor)
  - ALL - save all project source files which are currently in a modified state.

The class supports:

- Extracting the required property from the project property file (including replacement of the instance whenever an update of the property file is made).
- Setting the source files root folder.

- SaveIfModified methods:
    - for a file(s) within a project.
    - for a file which is not within a project.

ActionsSupport by Example

NodeActions

NodeActions creation:

- accepts the filename of a properties file
- enables the creation of DynamicAsyncActions 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) {
    .....
    nodeactions = new NodeActions(dir, "projectactions"); ①
}
```

① create a NodeActions object, which will observe a file *projectactions.properties* within folder *dir* for changes. It will use the content of that file to define DynamicAsyncActions, updating these actions whenever the file changes.

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 created by the NodeActions constructor, while the others must be passed to NodeActions (using setNodeBasicActions and setNodeActions methods).

```
    nodeactions.setNodeBasicActions(
            CommonProjectActions.renameProjectAction(),
            CommonProjectActions.copyProjectAction(),
            CommonProjectActions.closeProjectAction()  ①
    );
    nodeactions.setNodeActions(
            new DynamicAsyncAction("Bake")
                    .onAction(() -> new NbCliDescriptor(projectDir, "jbake", "-b")
                    .stderrToIO()
                    .stdoutToIO()
                    .ioTabName("Bake " + projectDir.getName())
                    .exec("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 nodeactions.getAllNodeActions(); ①
}
```

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

NbCliDescriptor

The NbCliDescriptor uses a set of Builder style methods to create the description. Once the description is complete the external process can be started and run to completion.

```
NbCliDescriptor descriptor = new NbCliDescriptor(projectDir, "jbake", "-b") ①
                        .ioTabName("Bake " + projectDir.getName()) ②
                        .stderrToIO() ③
                        .stdoutToIO() ③
                        .exec("Baking")) ④
```

① initialise the CLI command and arguements, and the required working directory

② Request an IOTab, defining the tab label

③ attach an external process stream to the IOTab

④ once configuration is complete, the exec method is used to open or reuse an IOTab, start the external process, connect all the required IO, run to process completion and finally tidy up the resources used.

SaveBeforeAction

SaveBeforeAction is a helper class to support SaveIfModified for documents which are open and modified.

There are two Use Cases:

Documents are not in a Project

In this case there is a simple method that is always enabled which can be called prior to calling the NbCliDescriptor *.exec(...)* method.

```
    SaveBeforeAction.saveIfModified(dataObject); ①
```

① static method, does not need an instance of SaveBeforeAction as it does not have a project property to set the mode. SaveIfModified is always enabled.

This will ensure the document will be saved if it is in a modified state. Only files modified by NetBeans will be considered as save candidates.

Document are part of a Project

In this case there is a need to parse the project properties file in order to extract the mode. This method will create an instance of SaveBeforeAction. Additionally the project source root folder needs to be defined.

```
    savebeforeaction = new SaveBeforeAction(properties, "save_before_publishing",
ALL); ①
    savebeforeaction.setSourceRoot(projectdir.getFileObject("src")); ②
```

① create the SaveBeforeAction, parsing the properties. Parameter are:

- the properties for extracting the required property
- the required property key
- the default mode (as this property is optional)

② set the project source root folder (used with the ALL option)

Parsing the project's properties file will extract the mode, which can be one of NO, YES or ALL.

The mode controls how any modified files are saved. Only files modified by Netbeans will be considered as save candidates.

The possible values for the property are:

- NO - No modified file(s) are saved.
- YES - Only the file selected (see saveIfModifiedByMode(dataObject) below).
- ALL - All modified files that are within the source folder will be saved.

Whenever a change to the properties file is observed, the project properties file will need to be reparsed and a new instance of SaveBeforeAction created. This occurs automatically as the project properties file is normally registered with the NodeActions class.

```
   nodeactionsmanager.registerFile("asciidoc", "properties", fct ->
updateProperties(projectdir));
```

Finally there is a method to apply SaveIfModified to any project files (controlled by mode).

```
    aproject.getSaveBeforeAction().saveIfModifiedByMode(dataObject); ①
```

① depending on the mode saves zero or more modified file(s).

Best practise will encapsulate these actions with the Project's Property Handling class. It then makes the SaveIfModified instance available to the project instance, which can in turn make it available to any project code needing the SaveIfModified functionality.

DynamicAction and DynamicAsyncAction

While the primary use case will be 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.

Using 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>Actions Support
        <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>2.1.0</version>
    </dependency>
```

Adding the NBM to 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 Packages.
- 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)

Useful Links

- Releases are stored in a Packages Maven repository on GitHub
- Source Code is stored in a Git repository on GitHub as part of a monorepro containing NBMs.
- Documentation (included JavaDoc API 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.

- Development Process Guide for The-Retired-Programmer products can be found on the TRP website.