dkandalov / **live-plugin**

IntelliJ plugin for writing plugins at runtime https://plugins.jetbrains.com/plugin/...

#intellij-plugin #groovy #ide #intellij-api #micro-plugin #groovy-language #groovy-script

817 commits | **6** branches | **18** releases | **4** contributors

Branch: **master** | New pull request | Create new file | Upload files | Find file | Clone or download

dkandalov removed AddKotlinLibsAsDependency action because AddLivePluginLibJars... ... Latest commit `7fdac21` 7 days ago

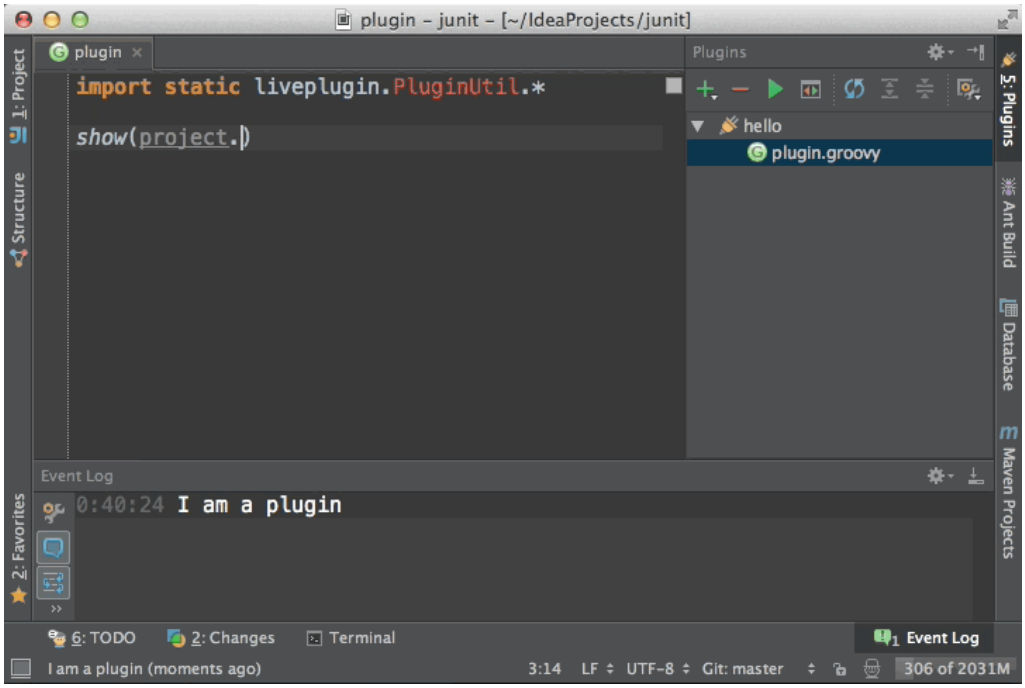
.idea	ide files	5 months ago
gradle/wrapper	added gradle wrapper	9 days ago
plugin_examples/liveplugin/plugine...	helloKotlin example	10 days ago
resources	intention template fix because scratch.ide.Actions doesn't exist anymore	8 days ago
src	removed AddKotlinLibsAsDependency action because AddLivePluginLibJars...	7 days ago
src_groovy/liveplugin	misc	15 days ago
test_groovy/liveplugin	misc fixes to make kotlin support work (mostly)	29 days ago
CONTRIBUTING.md	misc	a month ago
IntellijApiCheatSheet.md	updated readme	2 years ago
LivePlugin.zip	bumped version to 0.5.12 beta; updated zip file	6 months ago
README.md	misc	2 years ago
build.gradle	gradle buildPlugin extension to remove kotlin version from jars zippe...	8 days ago
dependencyLock.json	updated dependencies versions	9 days ago
gradlew	added gradle wrapper	9 days ago
gradlew.bat	added gradle wrapper	9 days ago
helloWorldCode.png	new screenshots	4 years ago
live-plugin-demo.gif	demo gif	3 years ago
scala-clojure.png	new screenshots	4 years ago
settings.gradle	gradle build using "org.jetbrains.intellij" plugin because it's the a...	21 days ago
toolwindow.png	updated toolwindow screenshot	4 years ago

README.md

LivePlugin

This is a plugin for IntelliJ IDEs to write plugins at runtime. It uses Groovy as main language and has experimental support for Scala and Clojure.

To install search for "liveplugin" in `IDE Preferences -> Plugins -> Browse Repositories`. Alternatively, download LivePlugin.zip from GitHub and use `IDE Preferences -> Plugins -> Install plugin from disk`. See also plugin repository page.



Why?

There is great Internal Reprogrammability blog post on this topic by Martin Fowler.

Motivations for LivePlugin are along the same lines:

- **minimal setup to start writing plugin**. Creating new project configured for plugin development feels like too much effort if all I want is to write 20 lines of code. LivePlugins exist outside of normal IDE projects and, therefore, can be modified and run from any project.
- **fast feedback loop**. Typical plugin development involves starting new instance of IDE and restarting it on every code change which cannot be hot swapped. LivePlugins are run in the same JVM instance, so there is no need to restart IDE.
- **customizable IDE**. It is disappointing that most development tools are difficult to customize. After all, developers is the best possible group of people to do it. This plugin is an attempt to improve the situation.

Practical use cases:

- project-specific workflow automation
- running existing shell scripts from IDE
- quick prototyping of IntelliJ plugins
- experimenting with IntelliJ API

Plugin Examples

Hello world

```
import static liveplugin.PluginUtil.show
show("Hello world") // shows balloon message with "Hello world" text
```

Insert New Line Above Action

```
import com.intellij.openapi.actionSystem.AnActionEvent
import static liveplugin.PluginUtil.*

// This action inserts new line above current line.
// It's a follow-up for these posts:
//   http://martinfowler.com/bliki/InternalReprogrammability.html
//   http://nealford.com/memeagora/2013/01/22/why_everyone_eventually_hates_maven.html
// Note that there is "Start New Line Before Current" action (ctrl + alt + enter) which does almost the same thing.

registerAction("InsertNewLineAbove", "alt shift ENTER") { AnActionEvent event ->
        runDocumentWriteAction(event.project) {
                currentEditorIn(event.project).with {
                        def offset = caretModel.offset
                        def currentLine = caretModel.logicalPosition.line
```

```
                        def lineStartOffset = document.getLineStartOffset(currentLine)

                        document.insertString(lineStartOffset, "\n")
                        caretModel.moveToOffset(offset + 1)
                    }
                }
            }
        }
        show("Loaded 'InsertNewLineAbove' action<br/>Use 'Alt+Shift+Enter' to run it")
```

See also Scala plugin example, Clojure plugin example and more examples listed below.

How to start writing plugins

- open `Plugins` tool window
- select one of the plugin entries in the panel
  (entries are folders, and `plugin.groovy` are startup scripts for plugins)
- click `Run` icon to execute plugin (or use keyboard shortcuts `alt+C`, `alt+E` or `ctrl+shift+L`)

If the above worked fine:

- modify `plugin.groovy` and rerun plugin to see results
- add built-in plugin examples and experiment with them (in `Plugins` toolwindow header `+ button -> Examples`)

If something doesn't work, report an issue.

(To use `alt+...` shortcuts on OSX you might need a workaround, please see this wiki page.)

The main idea

LivePlugin basically runs Groovy code in JVM. Conceptually it's quite simple:

```
ClassLoader classLoader = createClassLoader(ideClassloader, ...);
GroovyScriptEngine scriptEngine = new GroovyScriptEngine(pluginFolderUrl, classLoader);
scriptEngine.run(mainScriptUrl, createGroovyBinding(binding));
```

This means that your code is executed in the same environment as IDE internal code. You can use any internal API and observe/change state of any object inside IDE. There are some limitations of course, like `final` fields and complex APIs not designed to be re-initialized.

To simplify usage of IntelliJ API for practical purposes some parts of IntelliJ API are wrapped in PluginUtil class. This is essentially a layer on top standard IntelliJ API. If you find yourself writing interesting IDE scripts, feel free to create pull request or send a gist to include your code into `PluginUtil`. This is experimental API and there is no intention to keep it minimal. `PluginUtil` is not required though and you can always use IntelliJ classes directly.

Also note that:

- plugins are evaluated with new classloader on each run
- plugins are stored in `$HOME/.$INTELLIJ_VERSION/config/live-plugins` (on Mac `$HOME/Library/Application Support/IntelliJIdea15/live-plugins`) Note that you can use `ctrl+shift+C` shortcut to copy file/folder path.
- if available, Groovy library bundled with IDE is used

Misc tips

- if your plugins are stable enough, you can enable `Settings -> Run All Live Plugins on IDE Startup` option. If some of the plugins are not meant to be executed at startup, add `if (isIdeStartup) return` statement at the top.
- it helps to have JetGroovy plugin installed (only available in IDEs with Java support)
- you can get auto-completion and code navigation in plugins code
  - install/enable Groovy plugin
  - `Plugin toolwindow -> Settings -> Add LivePlugin Jar to Project`
    (the jar also includes source code for PluginUtil)
  - `Plugin toolwindow -> Settings -> Add IDEA Jars to Project`
    (adding jars unrelated to your actual project is a hack but there seems to be no major problems with it)
- it helps to be familiar with IntelliJ API
  - get and explore IntelliJ source code

    - look at jetbrains plugin development page
    - PluginUtil might be a good start point to explore IntelliJ API
- when plugin seems to be big enough, you can move it to proper plugin project and still use live plugin See liveplugin as an entry point for standard plugins.

More examples

- intellij-emacs - macros for making IntelliJ more friendly to emacs users (see also blog post)
- Simplistic "compile and run haskell" action - obviously this can be done for other languages/environments
- Google quick search popup - prototype of google popup search mini-plugin
- Scripting a macros - example of finding and invoking built-in actions
- Console filter/transform example - example of filtering and changing console output
- VCS update listener example - example of adding callback on VCS update
- Find class dependencies - simple action to find all class dependencies within current project
- Module transitive dependencies - finds all transitive dependencies for modules in IDEA project
- Show text diff - really lame example of opening IntelliJ text diff window (please don't use it!)
- Find all recursive methods in project (for Java) - quick plugin as a follow up for this talk
- Find all recursive methods in project (for Scala) - find all recursive methods in project
- Watching projects open/close events - an example of reloadable project listener
- Minimalistic view for java code - collapses most of Java keywords and types leaving only variable names
- Symbolize keywords - collapses Java keywords into shorter symbols
- Change List Size Watchdog - micro-plugin to show warning when change list size exceeds threshold (see also Limited WIP plugin)
- Template completion on "Tab" - simplistic prototype for auto-completion on tab key (in case built-in live templates are not enough)
- Completion contributor example - only gives an idea which part of IntelliJ API to use
- Google auto-completion contributor example - same as above but with google search plugged in
- Add custom search example - only gives an idea which part of IntelliJ API to use
- Get files from last commits example - gets VirtualFiles from several last commits
- Show PSI view dialog - one-liner to show PSI viewer dialog. Normally it's only enabled in plugin projects.
- Simplistic "generify return type" - attempt to pattern-match PSI tree
- No copy-paste - disables copy/paste actions
- Text munging actions - simple actions on text (sort, unique, keep/delete lines)
- Wrap selection - micro-plugin to wrap long lines with separator
- Wrap selected text to column width - copy of this plugin https://github.com/abrookins/WrapToColumn
- Create .jar patch file for current change list - that's what it does
- Create .jar patch file for specified favorites list - similar to the above mini-plugin
- Remove getters/setters - removes all setters or getters in a class
- ISO DateTime / Epoch timestamp converter - converts Epoch time to/from ISO format
- Make cursor move in circle - definitely not practical but gives an idea about threading
- Word Cloud - shows world cloud for the selected item (file/package/folder)
- Project TreeMap View - shows project structure (packages/classes) as treemap based on size of classes
- Method History - combines built-in method history based on selection and method history based on method name
- Evaluate selection as Groovy - that's exactly what it does
- Code History Mining - (not a tiny project) allows to grab, analyze and visualize project source code history

Similar plugins

The idea of running code inside IntelliJ is not original. There are similar plugins (some of them might be out-of-date though):

- IDE Scripting Console (experimental feature, bundled with IntelliJ since 14.1)
- PMIP - Poor Mans IDE Plugin (uses Ruby)
- Remote Groovy Console
- Script Monkey
- Groovy Console Plugin

- HotPlugin

Wish list

- try writing plugin for custom language support
- create AST pattern-matching API (this can be useful for writing inspections/intentions to match/replace parts of syntax tree)
- try more languages, e.g. Kotlin, Ruby or Java

Contributing

Please see CONTRIBUTING.md.