Squash Keyword Framework Plugin for IntelliJ-idea Documentation

Release 1.0.0-RELEASE

Minh Qwan Tran, Éric Dégenètais

Apr 01, 2020

Contents

CHAPTER 1

Squash TF IntelliJ IDEA plugin

1.1 Plugin Installation

1.1.1 Requirements

- IntelliJ Community IDE (version *2018.3.2* or later)
- JAVA JDK 1.8+

Note: The plugin can work without installing Maven on your machine thanks to the embedded version. However it is preferable to indicate the path of your own Maven version (3.5.0 recommended).

1.1.2 Install plugin from disk

Download Squash TF plugin for Intellij IDEA here

To install follow the default "install plugin from disk" instructions from IntelliJ :

1. In the `Settings/Preferences` dialog (*Ctrl+Alt+S*), select `Plugins`.
2. In the `Plugins` dialog, click the ⚙ and then click `Install Plugin from Disk`.
3. Select the plugin archive file and click `OK`.
4. Click `OK` to apply the changes and restart the IDE if prompted.

1.2 User Manual

`IntelliJ-Squash-plugin-user-manual-v1.0.8.docx`

CHAPTER 2

Squash TF IntelliJ IDEA plugin

2.1 Guide for Plugin Functional Development

2.1.1 Introduction

1. Purpose

This document is a Technical Design Document for use by **IntelliJ-Squash-plugin** project. It provides not only a global view on the infra-structure between the project modules, but also the technical/functional analysis onto each feature of the plugin project.

2. Intended audience

- Supervisors, to analyse the design and implementation of the plugin project
- Squash TF team members
- Future developers, testers who will work with the plugin

3. Scope

This document describes the structure design and the technical analysis of **IntelliJ-Squash-plugin** project.

4. Acronyms

Acronym	*Definition*
BNF	*Backus-Naur Form*
DSL	*Domain Specific Language*
IDE	*Integrated Development Environment*
ISP	*IntelliJ-Squash-plugin*
SKF	*Squash Keyword Framework*

2.1.2 System overview

The **ISP** project is a member of the Squash TF galaxia project group which is open-source but belongs to @Henix company.

While the whole group purpose is to provide to grand public a toolbox for functional testing automation and is dedicated to the industrialization of automated test execution, the goal of the **ISP** itself is to allow users (i.e. developers, testers…) to compose, execute or even test SKF DSL via IntelliJ Community IDE.

This plugin project consists of 5 modules:

- **IntelliJ-Idea Platform:** creates all dependencies in the IntelliJ-idea binaries to be used as provided dependencies for the build and packaging of the Intellij reactor.
- **IntelliJ-Idea Platform Bundle:** allows the packaging of the IntelliJ-Idea platform dependencies for CI builds and development environment.
- **Squash Keyword Framework Parser for IntelliJ-Idea:** parse the SKF DSL elements into IntelliJ PsiElements for plugin feature development.
- **Intellij-jflex-adapter:** creates an adapter for the version *1.7.0-2* of flex created by Jetbrains for IntelliJ Community IDE, in order to use it with the jflex plugin.
- **Squash Keyword Framework Plugin for IntelliJ-Idea:** provides all kinds of features for SKF DSL in IntelliJ Community IDE such as: autocompletion, syntax annotation/coloration…

2.1.3 Technical requirement

As **ISP** is a Maven project dedicated to create an IntelliJ IDE plugin, it is required in local:

- JAVA JDK 1.8+
- MAVEN (3.5.0 recommended)

Warning: The Maven path MUST contain no spaces to avoid installation problems.

- IntelliJ Community IDE (version from *2018.3.2* to *2019.1.x*)
- Git for cloning the IntelliJ project code sources (version *2018.3.2 Build #IC-183.4886.37*)

PS: The document for setting an environment for the **ISP** development can be obtain `here`

2.1.4 Plugin component descriptions

1. IntelliJ-Idea Platform

To be updated…

2. IntelliJ-Idea Platform Bundle

To be updated…

3. Squash Keyword Framework Parser for IntelliJ-Idea

This is the core of the SKF DSL parsing process in IntelliJ Community IDE. In fact, it helps the IDE not only to identify whether or not a file belongs to a DSL (ex: .ta, .macro) but as well to check if that file content complies with all those DSL syntax conventions.

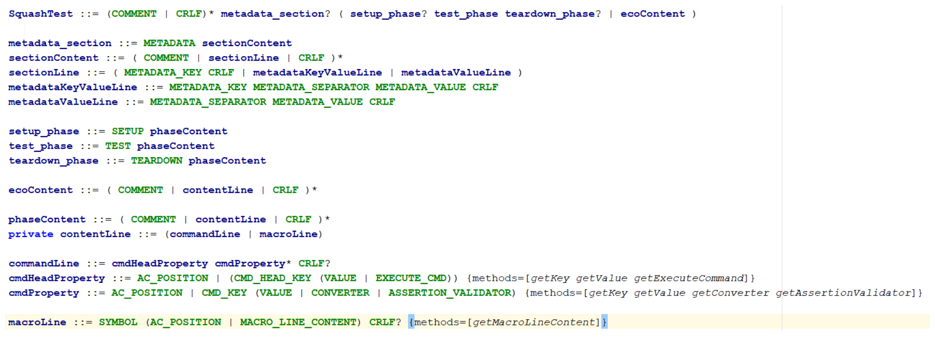
This module consists of 2 principal packages:

- **src/main/bnf:** contains all BNF files of the **ISP**.

  As known by its name, each BNF file is a formal notation for encoding grammars intended for human consumption. More detailed information about BNF can be found here.

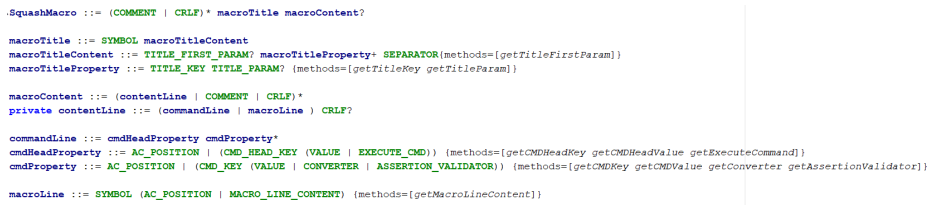
  There are two BNF files of SKF DSLs used in the **ISP**.

  – Squash Test BNF File: for files with `.ta` extension



```
SquashTest ::= (COMMENT | CRLF)* metadata_section? ( setup_phase? test_phase teardown_phase? | ecoContent )

metadata_section ::= METADATA sectionContent
sectionContent ::= ( COMMENT | sectionLine | CRLF )*
sectionLine ::= ( METADATA_KEY CRLF | metadataKeyValueLine | metadataValueLine )
metadataKeyValueLine ::= METADATA_KEY METADATA_SEPARATOR METADATA_VALUE CRLF
metadataValueLine ::= METADATA_SEPARATOR METADATA_VALUE CRLF

setup_phase ::= SETUP phaseContent
test_phase ::= TEST phaseContent
teardown_phase ::= TEARDOWN phaseContent

ecoContent ::= ( COMMENT | contentLine | CRLF )*

phaseContent ::= ( COMMENT | contentLine | CRLF )*
private contentLine ::= (commandLine | macroLine)

commandLine ::= cmdHeadProperty cmdProperty* CRLF?
cmdHeadProperty ::= AC_POSITION | (CMD_HEAD_KEY (VALUE | EXECUTE_CMD)) {methods=[getKey getValue getExecuteCommand]}
cmdProperty ::= AC_POSITION | CMD_KEY (VALUE | CONVERTER | ASSERTION_VALIDATOR) {methods=[getKey getValue getConverter getAssertionValidator]}

macroLine ::= SYMBOL (AC_POSITION | MACRO_LINE_CONTENT) CRLF? {methods=[getMacroLineContent]}
```

  – Squash Macro BNF File: for files with `.macro` extension



```
SquashMacro ::= (COMMENT | CRLF)* macroTitle macroContent?

macroTitle ::= SYMBOL macroTitleContent
macroTitleContent ::= TITLE_FIRST_PARAM? macroTitleProperty+ SEPARATOR{methods=[getTitleFirstParam]}
macroTitleProperty ::= TITLE_KEY TITLE_PARAM? {methods=[getTitleKey getTitleParam]}

macroContent ::= (contentLine | COMMENT | CRLF)*
private contentLine ::= (commandLine | macroLine ) CRLF?

commandLine ::= cmdHeadProperty cmdProperty*
cmdHeadProperty ::= AC_POSITION | (CMD_HEAD_KEY (VALUE | EXECUTE_CMD)) {methods=[getCMDHeadKey getCMDHeadValue getExecuteCommand]}
cmdProperty ::= AC_POSITION | (CMD_KEY (VALUE | CONVERTER | ASSERTION_VALIDATOR)) {methods=[getCMDKey getCMDValue getConverter getAssertionValidator]}

macroLine ::= SYMBOL (AC_POSITION | MACRO_LINE_CONTENT) {methods=[getMacroLineContent]}
```

  The elements in green color is of type **simple**. Their value is defined by a Lexer file that will be discussed in the next chapters of this document.

The elements in blue color is of type **complex**. Their value is composed of more than one **simple** or other **complex** elements.

Each element here will then be generated into a specific JAVA class called **IntelliJ PsiElement** class which is later used for the **ISP** features development.

> **Warning:** As a file violates one or more BNF specifications, the *first-found-troublemaker* element will be annotated as ERROR.
>
> This ERROR will disable any plugin functionality for the rest of the file.
>
> Moreover, it also blocks all possible development interventions as no more **IntelliJ PsiElement** could be identified.

Hint: It is therefore strongly advised to have a discussion between the plugin developers and product owner before inserting or removing a syntax convention at this PARSING level.

- **src/main/java:**

  This package contains JAVA classes that helps IntelliJ Community IDE to identify a file's language by its extension.

  It also creates Element-type and Token-type ofr a DSL, as well as implement methods in order to obtain the value of each PsiElement.

4. Intellij-jflex-adapter

To be updated…

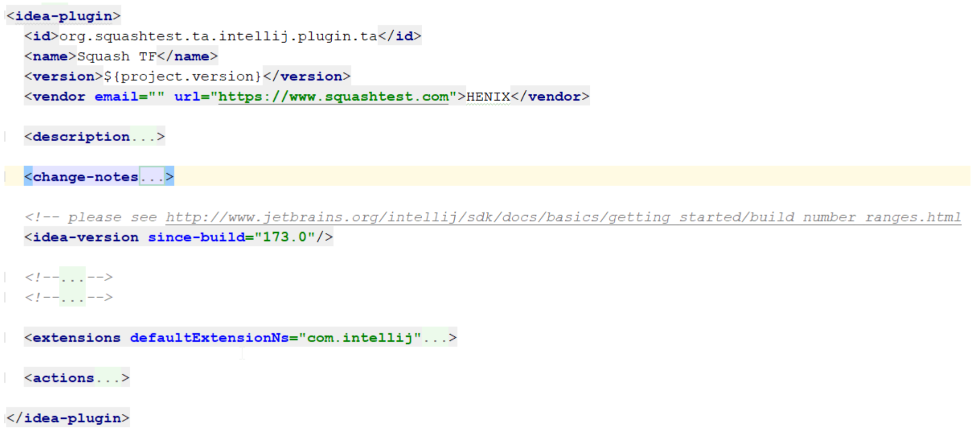
5. Squash Keyword Framework Plugin for IntelliJ-Idea

This module whose type is **IntelliJ Plugin** is the base package of the **ISP**.

Its purpose is to develop all SKF DSL features for IntelliJ Community IDE.

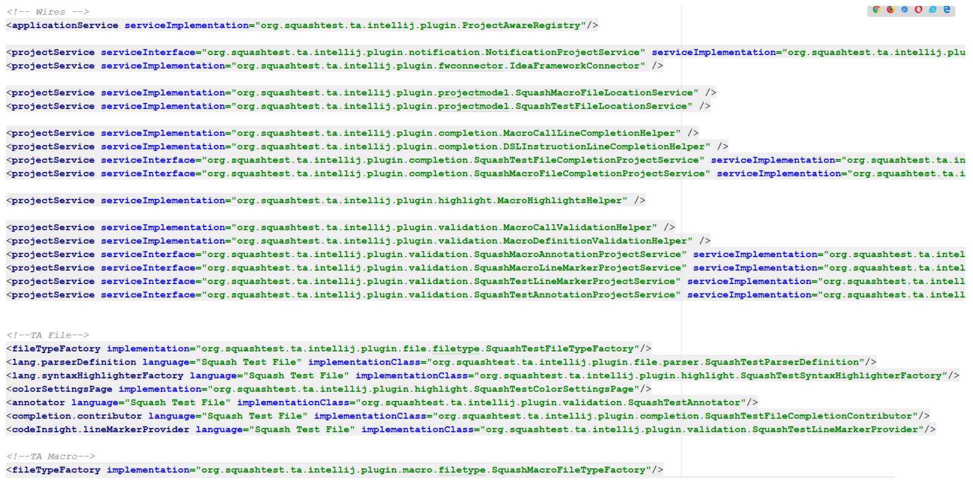
- **sources/META-INF:** contains the plugin.xml file.

  This XML file declares the service, the role, as well as the scale (or life-cycle) of JAVA classes created in the following **src/main/java** package.



```
<idea-plugin>
  <id>org.squashtest.ta.intellij.plugin.ta</id>
  <name>Squash TF</name>
  <version>${project.version}</version>
  <vendor email="" url="https://www.squashtest.com">HENIX</vendor>

  <description...>

  <change-notes...>

  <!-- please see http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started/build_number_ranges.html
  <idea-version since-build="173.0"/>

  <!--...-->
  <!--...-->

  <extensions defaultExtensionNs="com.intellij"...>

  <actions...>

</idea-plugin>
```

The most important part of this file is the *service declaration and implementation* that indicates which JAVA class takes care of which plugin feature of the **ISP**.



```
<!-- Wires -->
<applicationService serviceImplementation="org.squashtest.ta.intellij.plugin.ProjectAwareRegistry"/>

<projectService serviceInterface="org.squashtest.ta.intellij.plugin.notification.NotificationProjectService" serviceImplementation="org.squashtest.ta.intellij.plu
<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.fwconnector.IdeaFrameworkConnector" />

<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.projectmodel.SquashMacroFileLocationService" />
<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.projectmodel.SquashTestFileLocationService" />

<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.completion.MacroCallLineCompletionHelper" />
<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.completion.DSLInstructionLineCompletionHelper" />
<projectService serviceInterface="org.squashtest.ta.intellij.plugin.completion.SquashTestFileCompletionProjectService" serviceImplementation="org.squashtest.ta.in
<projectService serviceInterface="org.squashtest.ta.intellij.plugin.completion.SquashMacroFileCompletionProjectService" serviceImplementation="org.squashtest.ta.i

<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.highlight.MacroHighlightsHelper" />

<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.validation.MacroCallValidationHelper" />
<projectService serviceImplementation="org.squashtest.ta.intellij.plugin.validation.MacroDefinitionValidationHelper" />
<projectService serviceInterface="org.squashtest.ta.intellij.plugin.validation.SquashMacroAnnotationProjectService" serviceImplementation="org.squashtest.ta.intel
<projectService serviceInterface="org.squashtest.ta.intellij.plugin.validation.SquashMacroLineMarkerProjectService" serviceImplementation="org.squashtest.ta.intel
<projectService serviceInterface="org.squashtest.ta.intellij.plugin.validation.SquashTestLineMarkerProjectService" serviceImplementation="org.squashtest.ta.intell
<projectService serviceInterface="org.squashtest.ta.intellij.plugin.validation.SquashTestAnnotationProjectService" serviceImplementation="org.squashtest.ta.intell


<!--TA File-->
<fileTypeFactory implementation="org.squashtest.ta.intellij.plugin.file.filetype.SquashTestFileTypeFactory"/>
<lang.parserDefinition language="Squash Test File" implementationClass="org.squashtest.ta.intellij.plugin.file.parser.SquashTestParserDefinition"/>
<lang.syntaxHighlighterFactory language="Squash Test File" implementationClass="org.squashtest.ta.intellij.plugin.highlight.SquashTestSyntaxHighlighterFactory"/>
<colorSettingsPage implementation="org.squashtest.ta.intellij.plugin.highlight.SquashTestColorSettingsPage"/>
<annotator language="Squash Test File" implementationClass="org.squashtest.ta.intellij.plugin.validation.SquashTestAnnotator"/>
<completion.contributor language="Squash Test File" implementationClass="org.squashtest.ta.intellij.plugin.completion.SquashTestFileCompletionContributor"/>
<codeInsight.lineMarkerProvider language="Squash Test File" implementationClass="org.squashtest.ta.intellij.plugin.validation.SquashTestLineMarkerProvider"/>

<!--TA Macro-->
<fileTypeFactory implementation="org.squashtest.ta.intellij.plugin.macro.filetype.SquashMacroFileTypeFactory"/>
```

Note: Syntax convention for this plugin.xml is modified by IntelliJ from version `2019.2.x`. More information about this can be found here.

- **src/main/java:** consists of all JAVA classes used for the plugin feature development.

  There are **9** main sections in this package:

  5.1. Project model

  This is to defines a model for the SKF project organization in IntelliJ Community IDE as well tools to use it.

  To be specific, it checks whenever the current project is a valid SKF DSL project. On the other hand, it helps IntelliJ Community IDE to get the virtual files of all TA File/Macro files in the working project for further development investigations/interactions.

  5.2. Framework connector

  This module's goal is to connect the **ISP** with the SKF DSL framework (squash-ta-new-engine project) to obtain all the engine components such as built-in macros, converters or command. . .

  5.3. Notification

  This service controls the notification mechanism in the **ISP** (the notif. message content, warning level, showing time. . .).

5.4. Language

All JAVA classes for name, extension, icons, lexer (see `4.5.5 section`) and parser of each SKF DSL are defined in this section.

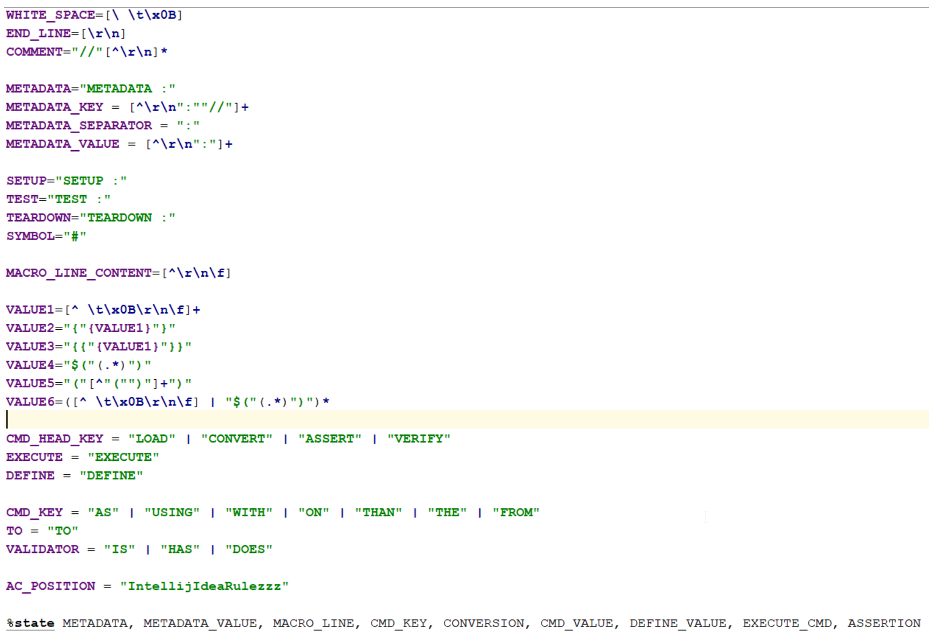
5.5. DSL Lexer

Until the project re-structuring is completed, these lexers temporarily locate in the **file/lexer/** and **macro/lexer/** folders.

IntelliJ Community IDE uses jflex technology to create the lexer for each custom language. In fact, a lexer defines the valid value(s) for every element created by the parser of that DSL. If the current parsed element of a file conforms to the corresponding lexer convention, it will be assigned to an appropriate PsiElement in the **ISP**.

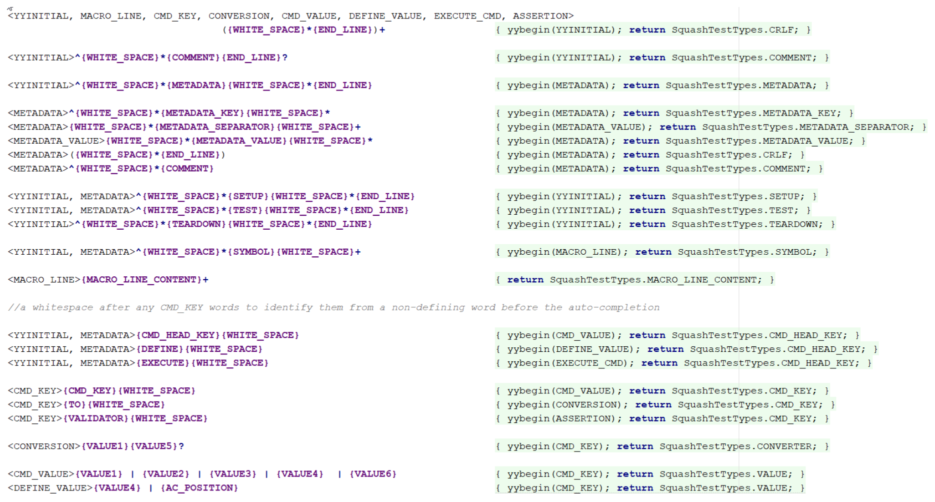
There are 2 principal steps in every lexer file (.flex):

- Declaration of variables and phases



```
WHITE_SPACE=[\ \t\x0B]
END_LINE=[\r\n]
COMMENT="//"[^\r\n]*

METADATA="METADATA :"
METADATA_KEY = [^\r\n":""//"]+
METADATA_SEPARATOR = ":"
METADATA_VALUE = [^\r\n":"]+

SETUP="SETUP :"
TEST="TEST :"
TEARDOWN="TEARDOWN :"
SYMBOL="#"

MACRO_LINE_CONTENT=[^\r\n\f]

VALUE1=[^ \t\x0B\r\n\f]+
VALUE2="{"{VALUE1}"}"
VALUE3="{{"{VALUE1}"}}"
VALUE4="$("(.*)")"
VALUE5="("[^"("")"]+")"
VALUE6=([^ \t\x0B\r\n\f] | "$("(.*)")")*

CMD_HEAD_KEY = "LOAD" | "CONVERT" | "ASSERT" | "VERIFY"
EXECUTE = "EXECUTE"
DEFINE = "DEFINE"

CMD_KEY = "AS" | "USING" | "WITH" | "ON" | "THAN" | "THE" | "FROM"
TO = "TO"
VALIDATOR = "IS" | "HAS" | "DOES"

AC_POSITION = "IntellijIdeaRulezzz"

%state METADATA, METADATA_VALUE, MACRO_LINE, CMD_KEY, CONVERSION, CMD_VALUE, DEFINE_VALUE, EXECUTE_CMD, ASSERTION
```

- Utilisation of variables and phases to identify the parsed element



```
<YYINITIAL, MACRO_LINE, CMD_KEY, CONVERSION, CMD_VALUE, DEFINE_VALUE, EXECUTE_CMD, ASSERTION>
                                    ({WHITE_SPACE}*{END_LINE})+                  { yybegin(YYINITIAL); return SquashTestTypes.CRLF; }

<YYINITIAL>^{WHITE_SPACE}*{COMMENT}{END_LINE}?                                   { yybegin(YYINITIAL); return SquashTestTypes.COMMENT; }

<YYINITIAL>^{WHITE_SPACE}*{METADATA}{WHITE_SPACE}*{END_LINE}                     { yybegin(METADATA); return SquashTestTypes.METADATA; }

<METADATA>^{WHITE_SPACE}*{METADATA_KEY}{WHITE_SPACE}*                            { yybegin(METADATA); return SquashTestTypes.METADATA_KEY; }
<METADATA>{WHITE_SPACE}*{METADATA_SEPARATOR}{WHITE_SPACE}+                       { yybegin(METADATA_VALUE); return SquashTestTypes.METADATA_SEPARATOR; }
<METADATA_VALUE>{WHITE_SPACE}*{METADATA_VALUE}{WHITE_SPACE}*                     { yybegin(METADATA); return SquashTestTypes.METADATA_VALUE; }
<METADATA>({WHITE_SPACE}*{END_LINE})                                             { yybegin(METADATA); return SquashTestTypes.CRLF; }
<METADATA>^{WHITE_SPACE}*{COMMENT}                                               { yybegin(METADATA); return SquashTestTypes.COMMENT; }

<YYINITIAL, METADATA>^{WHITE_SPACE}*{SETUP}{WHITE_SPACE}*{END_LINE}              { yybegin(YYINITIAL); return SquashTestTypes.SETUP; }
<YYINITIAL, METADATA>^{WHITE_SPACE}*{TEST}{WHITE_SPACE}*{END_LINE}               { yybegin(YYINITIAL); return SquashTestTypes.TEST; }
<YYINITIAL>^{WHITE_SPACE}*{TEARDOWN}{WHITE_SPACE}*{END_LINE}                     { yybegin(YYINITIAL); return SquashTestTypes.TEARDOWN; }

<YYINITIAL, METADATA>^{WHITE_SPACE}*{SYMBOL}{WHITE_SPACE}+                       { yybegin(MACRO_LINE); return SquashTestTypes.SYMBOL; }

<MACRO_LINE>{MACRO_LINE_CONTENT}+                                                { return SquashTestTypes.MACRO_LINE_CONTENT; }

//a whitespace after any CMD_KEY words to identify them from a non-defining word before the auto-completion

<YYINITIAL, METADATA>{CMD_HEAD_KEY}{WHITE_SPACE}                                 { yybegin(CMD_VALUE); return SquashTestTypes.CMD_HEAD_KEY; }
<YYINITIAL, METADATA>{DEFINE}{WHITE_SPACE}                                       { yybegin(DEFINE_VALUE); return SquashTestTypes.CMD_HEAD_KEY; }
<YYINITIAL, METADATA>{EXECUTE}{WHITE_SPACE}                                      { yybegin(EXECUTE_CMD); return SquashTestTypes.CMD_HEAD_KEY; }

<CMD_KEY>{CMD_KEY}{WHITE_SPACE}                                                  { yybegin(CMD_VALUE); return SquashTestTypes.CMD_KEY; }
<CMD_KEY>{TO}{WHITE_SPACE}                                                       { yybegin(CONVERSION); return SquashTestTypes.CMD_KEY; }
<CMD_KEY>{VALIDATOR}{WHITE_SPACE}                                                { yybegin(ASSERTION); return SquashTestTypes.CMD_KEY; }

<CONVERSION>{VALUE1}{VALUE5}?                                                    { yybegin(CMD_KEY); return SquashTestTypes.CONVERTER; }

<CMD_VALUE>{VALUE1} | {VALUE2} | {VALUE3} | {VALUE4}  | {VALUE6}                 { yybegin(CMD_KEY); return SquashTestTypes.VALUE; }
<DEFINE_VALUE>{VALUE4} | {AC_POSITION}                                           { yybegin(CMD_KEY); return SquashTestTypes.VALUE; }
```

More detailed information about jflex can be found at this link.

Warning: As in case of parsing process, if a file violates one or more FLEX specifications, the *first-found-troublemaker* element will be annotated as ERROR.

This ERROR will disable any plugin functionality for the rest of the file.

Moreover, it also blocks all possible development interventions as no more **IntelliJ PsiElement** could be identified.

Hint: It is therefore strongly advised to have a discussion between the plugin developers and product owner before inserting or removing a syntax convention at this LEXING level.

5.6. Validation

This is one of the 3 main features of the **ISP** added into IntelliJ Community IDE: defining general validation logic for SKF DSLs.

As you may already know, the first step of validating a file content is provided through each DSL parser and lexer. If passed, that file is then considered of that DSL and then the parsing process in **ISP** will store each captured item in an appropriate PsiElement object.

This is the second validation step which is to check if this *DSL-conformed-file* is in the correct directory, if a SKF DSL basic instruction line is 1 of 6 valid templates, if a macro line is well defined or if a metadata input syntax is correct…

Note: The advantage of this validation step is that developer can choose when, where and how to call an error, warn or info message for each case of violation.

In addition to the syntax validation, this section also provides the **macro signature/line tracing** ability. If a macro signature or a macro line of a file is found in the framework component list or in the project `shortcuts` folder (or its subfolders), a marker will be created at the beginning of that line and eventually provides the navigation to the defining macro file if the latest is in the `shortcuts` folder (or its subfolders).

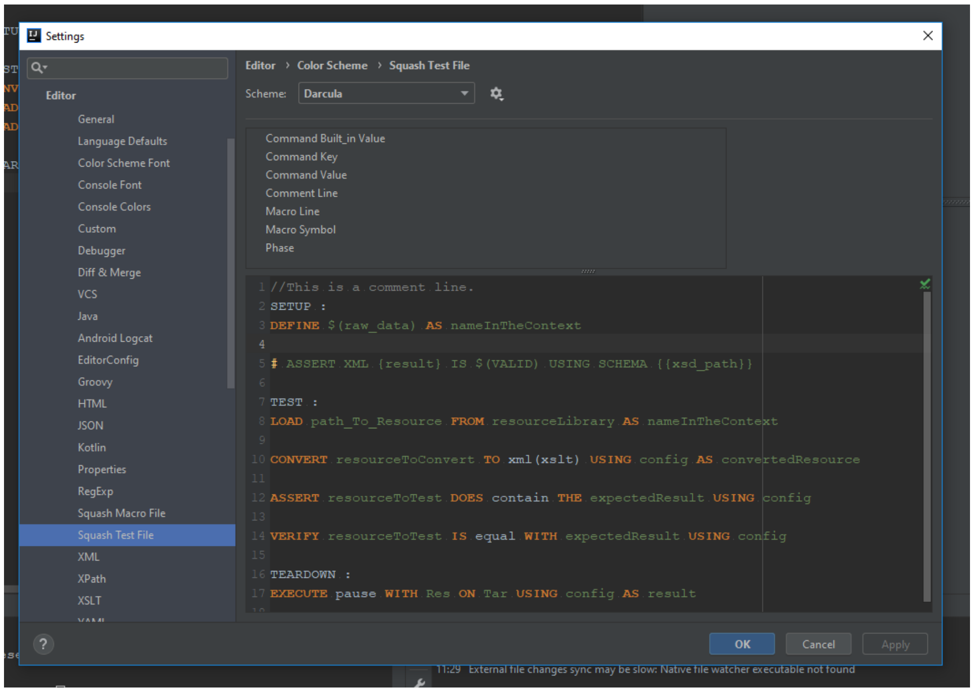
5.7. Highlighting

This is second features of the **ISP** added into IntelliJ Community IDE: providing generic highlight tools for SKF DSLs.

When a file is found in such a DSL by passing that language parsing process, each DSL item of this file will be associated with a corresponding PsiElement element. Those elements are then grouped by type.

This functionality is to assign each element group a specific color for the user visualization.

Besides, it creates also a `color setting page` for each SKF DSL in IntelliJ Community IDE.



```
//This is a comment line.
SETUP :
DEFINE $(raw_data) AS nameInTheContext

# ASSERT XML {result} IS $(VALID) USING SCHEMA {{xsd_path}}

TEST :
LOAD path_To_Resource FROM resourceLibrary AS nameInTheContext

CONVERT resourceToConvert TO xml(xslt) USING config AS convertedResource

ASSERT resourceToTest DOES contain THE expectedResult USING config

VERIFY resourceToTest IS equal WITH expectedResult USING config

TEARDOWN :
EXECUTE pause WITH Res ON Tar USING config AS result
```

```
11:29 External file changes sync may be slow: Native file watcher executable not found
```

5.8. Autocompletion

This is third features of the **ISP** added into IntelliJ Community IDE: implementing general completion logic for SKF DSLs.

This functionality proposes to user the most appropriate SKF DSL elements based on the current file content and the user cursor position. Each SKF DSL (Squash Test and Squash Macro) has a specific autocompletion mechanism.

Note: The common point of these two mechanisms is that the autocompletion result is at least a SKF DSL element and at most to complete the *asking-for-completion* line.

5.8.1. Squash Test autocompletion

There are three different kinds of completions in a Squash Test file:

- Phase/section completion

  When user requests a completion by typing *Ctrl+Space*, the plugin will firstly check the existence of each phase/section (i.e. METADATA, SETUP, TEST, TEARDOWN) in the current file. If this file is not an Ecosystem file (`setup.ta` or `teardown.ta`), the appropriate missing phase(s) will be proposed.

  Warning: The TEST phase must exist in file content or it is the only result to be proposed for completion.

- Line completion

  When a phase is already defined and the user cursor is on an empty line, a line completion will be invoked.

  In fact, user can choose either 1 of 6 available SKF DSL instruction templates or simply their HEAD KEY to insert an `basic instruction`.

  On the other hand, he/she can get the # symbol to start a `macro line`.

- Element completion

  When an autocompletion is asked on a non-empty line, the most appropriate SKF DSL elements will be offered if that line is of type `basic instruction`.

  In case of `macro line`, the element completion functionality is not yet available for this **ISP** version.

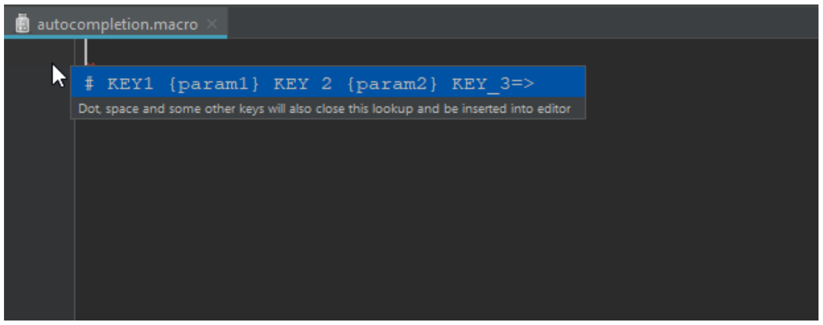
5.8.2. Squash Macro autocompletion

Similarly, there are also three different kinds of completions in a Squash Macro file:

- Macro signature completion

  When user requests a completion for an empty `.macro` , a model of Squash macro signature will be offered. User can then modify the Fixed parts or/and their Parameter to define his/her own macro signature.

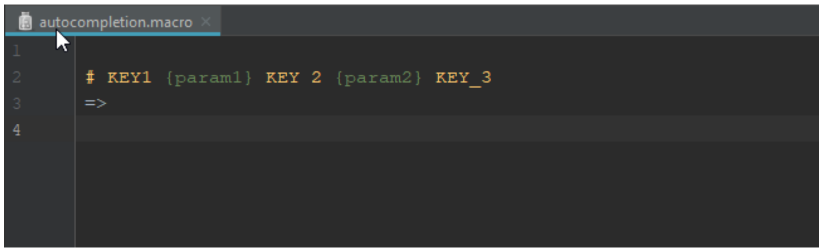
  Before auto-completion:



```
# KEY1 {param1} KEY 2 {param2} KEY_3=>
```

Dot, space and some other keys will also close this lookup and be inserted into editor

  After auto-completion:



```
# KEY1 {param1} KEY 2 {param2} KEY_3
=>
```

- Line completion

  Under reconstruction

- Element completion

  Under reconstruction

5.9. Tools

This is a collection of utils (methods) used for developing the 3 features of the **ISP**.

CHAPTER 3

Summary

This IntelliJ IDEA plugin helps us to write **S** quash **K** eyword **F** ramework, aka SKF, scripts. It brings us :

- autocompletion (instructions / macros)
- syntax highlighting (instructions / macros)
- syntax validation (instructions / macros)

Known limitations

Currently the Squash TF IntelliJ plugin has some limitations:

- The last declared PHASE/SECTION line in a Test script MUST not be empty to be recognised. If you don't have anything to insert in it yet, just simply complete that line by pressing "Enter".
- All plugin's features such as syntax validation, color highlighting, autocompletion… will no longer work properly after the first parsing error (if any) encountered in the file.
- Sometime the IntelliJ's color highlighting doesn't update after a syntax error correction, in that case all you need to do is move back to the beginning of the current line and press "Enter" to refresh the color highlighting.
- When you open the code completion popup for a macro and type some characters, the plugin gets all the macro containing those characters and keeps them to proposal. However, the filtration result contains all macro in which the searched characters existed not only in one block but also being separated by other characters.