JJupin Documentation

JJUPIN

Power to the workflow people!

> reduces the time of implementation by more than 50%

> consistent over Jira versions

> simple, intuitive, Java like syntax

> easy to express ideas, instead of concentrating on Jira internals

Gallery

JJUPIN provides virtually unlimited power to your Jira workflows. Forget about adding tens of plugins to your JIRA installation just to express yourself: this is all you need to create any post-function, validator or condition in your workflows. Our philosophy was to empower the customer and to create a JIRA installation that will adapt very easily to the actual needs without any special knowledge of the JIRA internals; for that we created a JIRA adapted language, named **Simple Issue Language 4.0**, or simply SIL.

SIL is very easy to learn yet powerful and extensible: it's a Java-like language and it is independent of the JIRA version; furthermore SIL has made its way through our Database Custom Field and Kepler Custom Fields, Blitz Actions plugins as well as in our newest plugin family member, KCF PRO , by specific extensions using the same language. All for one purpose: power through simplicity and flexibility.

With Atlassian JIRA at base and with our SIL-enabled plugins on top, we managed to put big smiles on our customer's faces: JJUPIN made possible incredible integrations and customizations of JIRA.

Whenever you have heavy workflows, integration with your payment systems or you simply want better awareness for your teams, JJUPIN is here to help. If you want to use Jira as a helpdesk solution, JJUPIN can update your inventory tables directly from JIRA, while your teams are responding to user requests. If you have a tight SLA, JJUPIN can send intelligent mail, helping the programmers focus on the priorities, and not being flooded with spam email about trivial modifications in issues.

Common use cases are:

- Complex workflows
- Integration with legacy systems
- Integration with your enterprise systems (relational databases, files, mail systems, LDAP, SMSC)
- Integration with other JIRA systems, not necessary the same version.
- Smart notifications
- Automatically charging for support, when you employ this business model

Besides post-functions, validators and conditions, which are linked directly in your workflow, JJUPIN offers a full environment:

- SIL listeners - so you can react when an issue is changed,
- SIL services & job scheduling - a way to implement batch updates and notifications to your issues and automate tasks,
- A gadget so that regular users can run their own scripts (useful for example by project leads to automate tasks),
- A nice editor, with common functionalities such as autocomplete,
- A comprehensive view of the workflow actions, screens and their fields and, of course, attached SIL code,
- A SIL Manager, so that you can easily browse for and edit scripts,
- Live Fields - SIL routines for hiding, disabling, attaching messages or setting values for issue fields in any screen.

We tried hard to minimize JIRA customization time because this is usually something that comes into aid of the real productive activities; minimizing the time for these customizations means that your teams can benefit faster from them. Our approach was pragmatic, therefore:

- We introduced aliases for custom fields so that one can develop on some test environment then move scripts directly into production,
- We introduced environment variables for SIL,
- All our routines are lenient regarding common user mistakes (e.g. asking for a greater substring than the string has to offer does not result in error).

Of course, we did not forget extensibility. Registering new routines and adding support for additional custom fields is easy. JJUPIN gives you:

- Include scripts / User defined routines - so you can create libraries of routines
- Mappings for custom fields to a known descriptor, by using Custom Field Descriptors or programatically.
- An easy way to write java routines and hook them into the language. Our Javadoc is available to our customers.

Recently Updated

JJUP30
Feb 28, 2017 • attached by Alexandru Geageac

Supported fields and graphic elements
Dec 21, 2016 • updated by Confluence Administrator • view change

SIL Manager
Dec 15, 2016 • updated by Confluence Administrator • view change

Backup and restore
Feb 05, 2016 • updated by Alexandru Geageac • view change

Licensing
Jan 28, 2016 • updated by Florin Haszler • view change

IfShow
Jan 27, 2016 • updated by Alexandra Topoloaga • view change

IfHide
Jan 27, 2016 • updated by Alexandra Topoloaga • view change

What should I do if I installed an incompatible version?
Dec 02, 2015 • created by Alexandra Topoloaga

Requirements
Nov 16, 2015 • updated by Alexandra Topoloaga • view change

Install notes for JIRA 7
Nov 16, 2015 • created by Alexandra Topoloaga

runnerLog
Sep 17, 2015 • updated by Alexandra Topoloaga • view change

Parameters in SIL Runner Gadget
Sep 15, 2015 • updated by Alexandra Topoloaga • view change

Selection_001.png
Sep 15, 2015 • attached by Alexandra Topoloaga

IfHideAllExcept
Sep 09, 2015 • updated by Alexandra Topoloaga • view change

Routines
Sep 09, 2015 • updated by Alexandra Topoloaga • view change

Introduction

JJupin

JJupin is a OSGI enabled JIRA plugin that offers scripting capabilities to JIRA.

SIL

The scripting language, named SIL (Simple Issue Language 4.0), helps you improving JIRA work flows, by extending them with new conditions, post-functions and validators, while keeping you free from the changes of the JIRA API.

This language offers conditional behaviour in post-functions (which otherwise would require new states in the workflow – thus simplifying your workflow), string manipulation routines for JIRA fields, SQL access, operating system access (command line, email, ...) and many more.

The language is intended for people who do not want to enter into implementations details of JIRA, users who do not know Java, but not only. The purpose of the language was to make things as simple as possible , so a person without (many) programming abilities can use it, but retaining as much flexibility as possible and exposing what we believe to be the basics of work flow customization.

If the standard functionality is not enough, you may extend the language with your own functions and your own custom field support. For details, please contact us.

What's new in JJUPIN 3.0

- **Updated to work with Simple Issue Language 3.0 and all the goodies it brings.**
- Dropped support for SOAP remote calls. All remote calls are now done via REST.
- Updated SIL Listener configuration UI
- Updated SIL Services & Scheduler configuration UI

Requirements

A fully installed JJupin consists of multiple jar files. You are advised to use the bundle installer when installing JJupin. Please refer to the Install Guide for explanations and details.

At the minimal level JJUPIN consists from 2 dependencies (jar files): katl-commons (a library having countless utility routines, but also - most important - the SIL language parser) and JJUPIN jar file, which contains JJUPIN specific routines plus the user interface : script editor, gadgets, JIRA specific hooks, etc.

SMS functionality is achieved still through the same jar file (jjupin-integration jar).

Compatibility

JJUPIN Version	JIRA	katl-commons
3.0	6.x	3.0
3.0.1	6.x	3.0.1
3.0.2	6.x	3.0.2
3.0.3	6.x	3.0.3
3.0.4	6.x	3.0.4
3.0.5	6.x	3.0.5
3.0.6	6.x	3.0.6
3.0.7	6.x	3.0.7
3.0.8	6.x	3.0.8
3.0.9	6.x	3.0.9
3.0.10	6.x	3.0.10
3.1	7.x	3.1

Installation & Configuration

This is the JJUPIN administration section. Please refer to each subsection for details on how you should configure the product.

- Installation
- Install notes for JIRA 7
- Administration Page
- SIL Configuration
- Mail Configuration
- Remote Systems
- SQL Configuration
- LDAP Configuration
- Configuring a SIL JIRA Service
- Configure JIRA Logging
- Licensing
- Uninstall

Installation

Installation via Atlassian Universal Plugin Manager

This page points the simple steps to follow for installing the plugin using the Universal Plugin Manager. This method requires an internet connection.

Manual Install

It may seem more complicated, but a manual install is quite easy to do. After all, all you have to do is to copy some files. Here's how.

Installation via Atlassian Universal Plugin Manager

Installation via Atlassian Universal Plugin Manager

If you are not familiar with Universal Plugin Manager (UPM), please read this document before we begin.

Steps are simple:

1. Enter the administration screen and go to *Add-ons->Find new add-ons*.

2. Search for **jjupin** plugin and install it.

That's all.

> **Note**
> The **jjupin-integration provider** jar cannot be installed via the UPM, since it will need a JIRA cold restart. If you need remoting capabilities, you will have to download it from our site (jira-plugins.kepler-rominfo.com) and placed manually in the **JIRA_INSTALL_DIR /atlassian-jira/WEB-INF/lib** directory

Manual Install

Manual Install

Do not worry, it's a simple task to install it manually:

1. Download the correct jjupin obr file from Atlassian Marketplace or from our site: Kepler Products.

2. Go to Administration->Add-ons->Manage add-ons. Install the previously downloaded obr file by using 'Upload add-on' link.

3. [Optional] Copy **jjupin-integration-provider** jar into **JIRA_INSTALL_DIR/atlassian-jira/WEB-INF/lib**. This is optional and it is needed only if you plan to do remoting on JIRA - you can call scripts at a distance, on your JIRA instance (so you can better integrate JIRA with other apps, for instance). This step needs a JIRA restart.

4. Install a license for jjupin, which can either be provided as the **jjupin.lic** file, or as the key generated via the Atlassian Marketplace. See more details about this in Licensing.

5. [Optional, but highly recommended]: Enable logging on our modules. Open with a text editor of your choice the JIRA log4j configuration file *JIRA_INSTALL_DIR/atlassian-jira/WEB-INF/classes/log4j.properties* and add these 2 lines at the end of it. Restart Jira.

```
log4j.logger.com.keplerrominfo=INFO, filelog
log4j.additivity.com.keplerrominfo=false
```

Installing a New License

To install a new license, there are four easy steps you must follow:

1. Acquire the license file. (**jjupin.lic**)
2. Stop JIRA.
3. Copy (or overwrite) the **jjupin.lic** file to **JIRA_HOME/kepler/**. If the **kepler** folder does not exist, create it.
4. Start JIRA.

Install notes for JIRA 7

When upgrading from an older version of JIRA to JIRA 7, you must update all our plugins as well.

As you can see on this page, the versions compatible with JIRA 7 are the 3.1.x versions.

What should I do if I installed an incompatible version?

As we have said before, **3.0.x** versions are compatible with **JIRA 6.x** and **3.1.x** versions are compatible with **JIRA 7.x**.

If you have installed JJUPIN 3.0.x on JIRA 7.x or JJUPIN 3.1.x on JIRA 6.x, you should do the next steps :

1. Uninstall warden
2. Uninstall katl-commons
3. Uninstall JJUPIN
4. Install the right version of JJUPIN (the one compatible with your JIRA)
5. katl-commons and warden should now have the right versions as well

> After you uninstall katl-commons and warden, some plugins may remain disabled, so you may need to re-enable them manually.

Administration Page

Introduction

To allow for better customization of JJupin to suit your needs, we have created an administration area where you can configure various parameters.

Navigate to **Administration ➤ Add-ons** -> **JJupin** to get to the **JJupin Administration Page**:
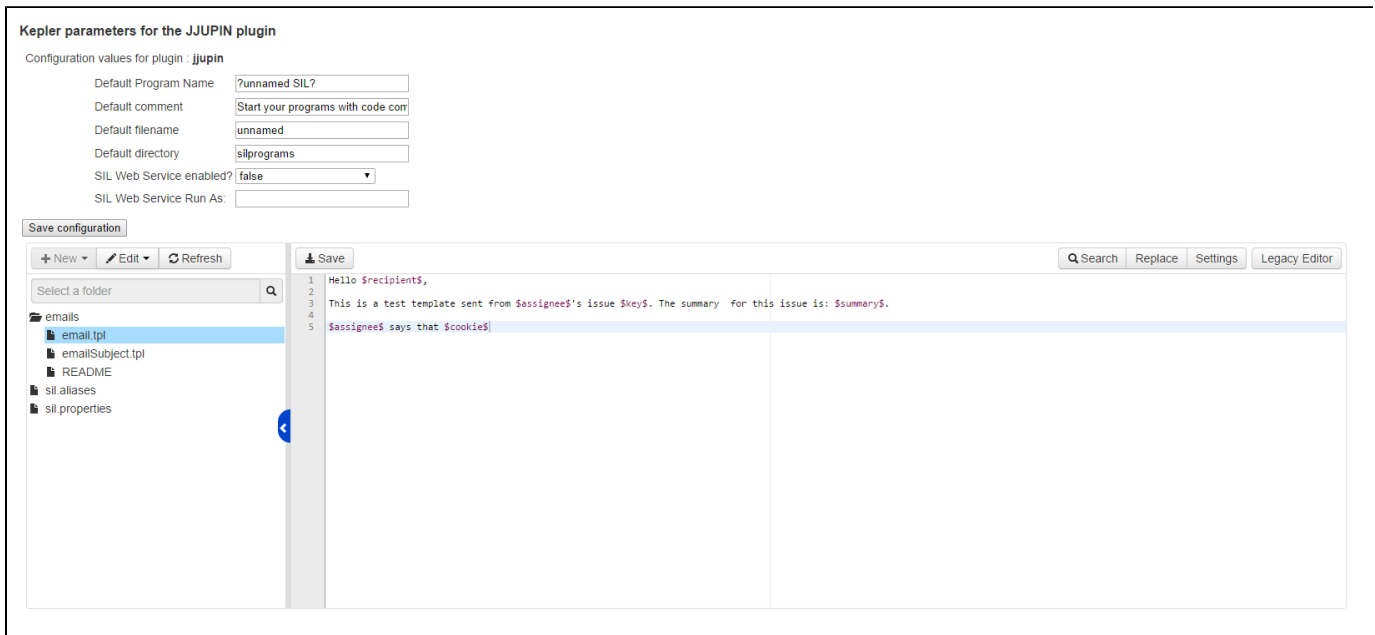
1. Configuration Values for JJupin Plugin

Default Program Name - the name of the default SIL script name that appears in SIL editor when you create a new condition, validator or post-function.

Default comment - the default comment that appears in SIL editor when adding or editing a new SIL script. The first (max. 3) commented lines will appear as a description for your condition, validator or post-function.

Default filename - the default SIL file name to save your program as.

Default directory - the default directory where SIL scripts will be saved. If no absolute path indicated, then will be considered the relative path to **<JIRA_HOME>**

SIL Web Service enabled? - indicates if the SIL Web Service is enabled or not

SIL Web Service Run As - the username to run the service as.

2. Email Templates and Configuration Manager

The left side of the editor includes the file tree for the **email template directory** and the configuration directory where you can define aliases for custom fields and properties for SIL environment.

> **Info**
> See Mail Configuration for instructions on how to use email templates.

You can create, delete, rename or edit email templates using this editor. It also offers search/replace capability.

Advanced Config

In this section of **JJupin Administration** you can configure the next parameters:

Advanced configuration for JJupin

Asynchronous Runner

| Threads | 3 ‡ |
The number of threads to be initialized by the thread pool for minor asynchronous operations

| Time To Live (TTL) | 1h ‡ |
Maximum running time for a SIL script. A script that takes more than the maximum allowed time to run will be killed and end with no result.

| Checkpoint Interval | 5m ‡ |
Time interval between cleanup actions on the threads that exceeded the TTL.

Asynchronous Runner

This are the parameters for SIL Runner Gadget (up to the 3.0.8 version). Since version 3.0.8, the pool only cares about a limited set of functionality in the SIL manager (calculating usage, etc)

Threads - the number of running threads (number of sil scripts running in the same time).

Time to Live (TTL) - time to live (running time for a sil script). If you run a script that takes more than TTL configured, the script will end with no result.

Checkpoint Interval - interval to clean up the expired threads (the sil scripts that exceeds the TTL configured).

Startup Script

Allows you to configure a startup script; this script is run every time JJUPIN gets started, either for administrative reasons (i.e. update of the plugin) or at JIRA's own startup. This feature appears at 3.0.8 version.

Startup Script

| SIL File | /opt/atlassian/jira63home/silprograms/job.sil |
The startup script will be run at the start of the plugin.

SMS Provider Configuration

SMS Provider Configuration

1. Copy JJupin-integration-provider.jar file into <InstallPath>/atlassian-jira/WEB-INF/lib
2. Configure the EnmsProvider:
   Create `enms.properties` file into <JIRA_HOME>/kepler folder. It's the same path where the Kepler licenses (e.g. for jjupin) are located.
   Provide the configuration for SMS provider:

enms.user=<username>

enms.password=<passwd>

enms.endpoint=http://ems.kepler.ro/emsws/service.asmx?wsdl

enms.default.sender=<phone number>

enms.unicode=false

> Restart JIRA to put into effect this configuration. Each change needs restart.

Configuration Details

enms.user is the username to authenticate against the web service end point

enms.password is the password to authenticate against the web service end point

enms.endpoint is the URL where the web service is located

enms.default.sender is the default sender phone number that is used by the web service to send messages with

enms.unicode enables or disables the unicode text representation

The integration jar can be retrieved from this location: https://www.kepler-rominfo.com/static/downloads/com.keplerrominfo.jira.plugins.jjupin/resources/jjupin-integration-provider-2.0.2.jar

SIL Manager

SIL Manager

The SIL Manager allows you to create, delete, edit and view all the SIL programs used in the JIRA environment (Conditions, Validators, Post-Functions, SIL Service and SIL Runner Gadget).

SIL scripts browser:

The script browser allows you to see and manage all the SIL programs. By default, when opened, the browser will show all existing SIL scripts.

You can click on 'Hide unused' button to filter the view so only the programs which are in use in a condition, validator, post-function, service, gadget or live fields configuration are visible.

The available operations are:

New - creates a new SIL file or folder under the selected directory.

> **Note**
> You can only create new files and folders in the non-filtered view. You can toggle this by clicking **Show(Hide) Unused.**

Delete - deletes the selected file or directory

> **Warning**
> Deleting files or folders will also delete them **from the disk**, so all the contents are lost and **cannot be recovered**.

Rename - renames the selected folder or file

Refresh - reloads the SIL file tree. This is useful when the usage of some programs changes.

Show/Hide unused - shows or hides the unused SIL files scripts.

> **Tip**
> To hide the file browser, click on the vertical bar that separates it from the editor. Additionally you can resize it by dragging the border between the file picker and editor.

> Right-click items in the script browser to see a contextual menu of available actions for the selected file.

Searching

When a folder is selected, you can start typing in the search box to find files by name. Note that this search is recursive and will also look inside folders under the currently selected one.

Editor

When a file is selected in the script browser, its contents will be loaded into the editor.

The toolbar allows for quick actions such as checking the script for errors, saving changes, find/replace functionality. Additionally you can fine-tweak some settings of the editor by clicking the **Settings** button and changing the values in the pop-up dialog.

Clicking the **Show Usage** button will toggle the editor between showing file contents and where the script is used, such as workflow actions, listeners, services, etc. You can also look inside other scripts for declarations where the current file is included. Note that this may take a while if you have a large number of scripts. Note that while editing a script, pressing **Ctrl+Space** will open up an auto-complete suggestions menu.
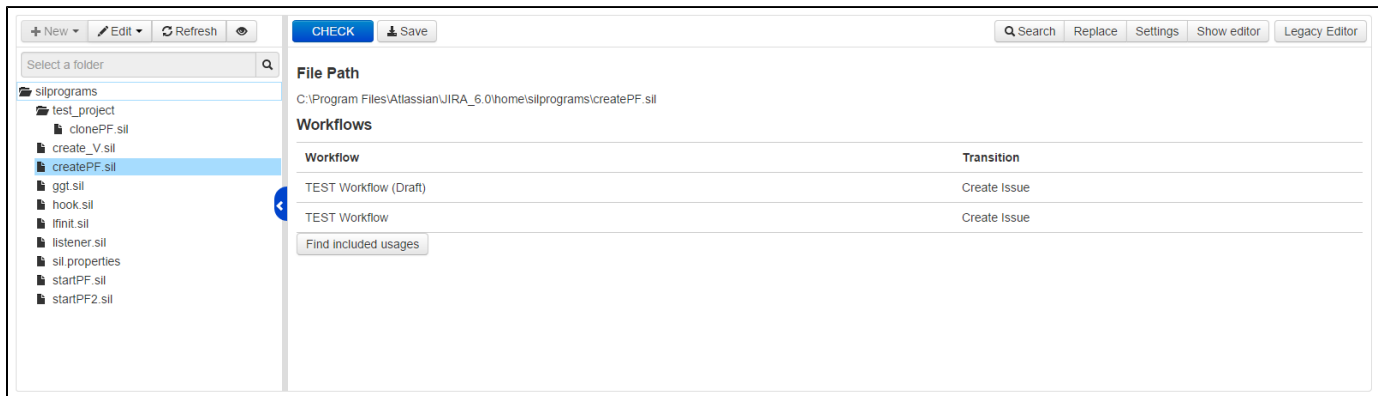
> Credits for the editor go to the Ace editor.

SIL Services & Scheduler

Managing SIL Services

The SIL Services feature allows users to run SIL scripts periodically. Each SIL Service has the following fields/properties:

- Name - short name to explain what the service does
- Run As - the user to impersonate when running the service
- Interval - the interval between two consecutive runs. The input requires a "JIRA-style" formatted interval (e.g. 3d 12h 30m). However, note that, as opposed to intervals used throughout JIRA, this representation assumes a 24h/day 7days/week timeframe.
- Script - the script to run. Note that these scripts do NOT have an issue context.

> **Info**
> To edit the actual scripts, please use the SIL Manager.

Managing SIL Scheduler

> **Availability**
> This feature is available since **jjupin 3.0.8 .**

The SIL Schedulers feature allows users to run SIL scripts after a valid JIRA interval or using a CRON expression.

The jobs are not persistent and they use run once per cluster policy. Please read about scheduling jobs here: Scheduling Routines. Since this mechanism uses the same scheduling engine, the same notes apply.

Each scheduled job has the following properties:

- Schedule - a valid JIRA interval or a CRON expression
- Repeatable - if you use a JIRA interval you can choose if the job repeats every interval
- SIL File - the SIL Script that will run using the schedule defined by the user
- Arguments - the arguments of the job

Add Job

Scheduler Type *   Interval ▼

Schedule *   2h 30m

Add a valid JIRA interval or a CRON expression

Repeatable ☑

If not cron, is this job repeatable?

SIL File *   Select a folder   🔍

📂 silprograms
    📁 testscripts
    📄 1.sil
    📄 a.sil
    📄 addGroupFromRole.sil
    📄 allfields.sil
    📄 arrayFind.sil

Choose a job SIL file from the file tree above.

Arguments   param1 param2

Arguments for the above job, separated by space

Add job   Cancel

Add Job

Scheduler Type* [Cron ▼]

Schedule* [0 0 12 * * ?]

Add a valid JIRA interval or a CRON expression

SIL File*

[Select a folder   🔍]

📂 silprograms
    🗀 testscripts
    📄 1.sil
    📄 a.sil
    📄 addGroupFromRole.sil
    📄 allfields.sil
    📄 arrayFind.sil

Choose a job SIL file from the file tree above.

Arguments [param1 param2]

Arguments for the above job, separated by space

[Add job] Cancel

SIL Listener

The SIL Listener allows users to execute a script when certain events are triggered. Each entry for the SIL Listener represents a script the will run for an event and has the following fields/properties:

- Event - mandatory - the event to react to
- Run As - optional - user to impersonate when running the script. If left empty, the script will be run by the currently logged in user. This setting may be necessary if certain scripts require additional privileges than regular users.
- Script - mandatory - the script to run when the event is received

Multiple listener entries can be added for the same event.

Infinite Loop
When selecting a script for an event, please make sure that the script does not use the raiseEvent routine to raise the same event, as this will cause a loop and crash your JIRA instance.

SIL Context

When writing the SIL script that will handle an event, the username of the user who triggered the event will be available as the first element in the **argv** variable and it can be used like this:

```
string callingUser = getElement(argv, 0);
```

Also, the issue context (all the standard variables and custom field values) will be set to those of the issue where the event was triggered from. For example, if a SIL script is triggered by an event launched from the issue "TST-123", all the standard variables and custom fields used in the SIL script will point to the issue "TST-123", unless specified otherwise using the construction %otherIssueKey%.variable.

Info
To edit the actual scripts, please use the SIL Manager.

Aside from the issue events that are configurable from the JIRA UI, the SIL Listener also allows you to react to other events. Note that these events, since they're not related to an issue will not run in an issue context and using issue standard variables without qualifying them with the key of the issue does not make sense. Additionally, each event may add additional information to the **argv** variable, aside from the information that is common for all events.

The first three elements in the **argv** array (string array) are (note that indexing in the array starts from 0) :

1. The user that triggered the event
2. An internal id for the event that was triggered. Normally you should not need this.

3. The name of the event as specified in the dropdown list that configures the listener.

The next elements in the array after these are event-specific and are detailed in the table below

Event	Additional Parameters	Observations/Example
Version Created Version Archived Version Moved Version Released Version Unarchived Version Unreleased	4. version ID 5. the string representation of a Version structure. You can retrieve this value and then cast it to a "Version" structure.	Version v = (Version) argv[4];
Version Merged	4. version ID 5. the string representation of a Version structure. You can retrieve this value and then cast it to a "Version" structure. 6. merged version ID 7. the string representation of the merged version (castable to Version, similar to 5).	Even though you can merge multiple versions into one at once, the JIRA API only provides reference to a single merged version.
Version Deleted	4. version ID 5. empty (the version is already deleted at this point and details are no longer available)	
Project Created	4. project ID 5. the string representation of a Project structure. You can retrieve this value and then cast it to a "Project" structure	
Project Deleted	4. project ID 5. project KEY (the project is already deleted at this point and details are no longer available)	

SIL Custom Field Descriptors

SIL Custom Field Descriptors

Sil custom field descriptors are used to translate the custom field value into a valid SIL value. Most of them are already mapped to a certain descriptor, but there are some of them which are not.

For the custom fields that are not mapped to a descriptor, the default descriptor is used. This does not guarantee that the descriptor is the proper one for the field, but it will attempt to determine the correct type.

Steps for the configuration

If you have such fields, here is how to configure the custom field descriptors:

- Navigate to **Administration -> Kepler General Parameters -> Custom Fields** to get to the **SIL Custom Field Descriptors**.
- Search for the custom fields that you use in the SIL scripts and choose the right descriptor.(fields already registered have the list of descriptors disabled)

- Choose the proper descriptor.
- The "**Saved**" message should appear.
- Now you can run a test SIL script to verify the behavior of the custom field.

List of Descriptors

Here is the list of descriptors you can choose from:

Custom Field Key	Descriptor
com.atlassian.jira.plugin.system.customfieldtypes:datetime	Date -> date
com.atlassian.jira.plugin.system.customfieldtypes:datepicker	Date -> date
com.keplerrominfo.jira.plugins.blitz-actions:blitzactions-cf	Default
com.atlassian.jira.plugin.system.customfieldtypes:readonlyfield	Default
com.atlassian.jira.plugin.system.customfieldtypes:radiobuttons	Option -> string
com.atlassian.jira.plugin.system.customfieldtypes:multigrouppicker	Group[] -> string[]
com.atlassian.jira.plugin.system.customfieldtypes:float	Number -> number
• Business Value	
• Story Points	
com.atlassian.jira.plugins.jira-importers-plugin:bug-importid	Default
com.pyxis.greenhopper.jira:gh-sprint	Default
• Sprint	
com.atlassian.jira.plugin.system.customfieldtypes:version	Version[] -> string []
com.atlassian.jira.plugin.system.customfieldtypes:textarea	String -> string
com.pyxis.greenhopper.jira:gh-global-rank	Default
com.keplerrominfo.jira.plugins.usergrouppicker-pro:siluserpicker	User -> string
• silUgp	
com.atlassian.jira.plugin.system.customfieldtypes:url	String -> string
com.atlassian.jira.plugin.system.customfieldtypes:multiselect	Option[] -> string[]
com.keplerrominfo.jira.plugins.keplercf:regexcf	String -> string
com.atlassian.jira.plugin.system.customfieldtypes:multiuserpicker	User[] -> string[]

Default
Default
Boolean -> boolean
Interval -> interval
User[] -> string[]
Group -> string
Number[] -> number[]
Option -> string
Number -> number
Boolean[] -> boolean[]
Date[] -> date[]
Date -> date
User -> string
Label[] -> string[]
String -> string
Interval[] -> interval[]
Group[] -> string[]
Cascade -> string []
Collection(String) -> string [

1. **Default**

   This descriptor will try to determine the SIL representation of the custom field based on the type of its value.
2. **Boolean -> boolean**

   Translates a boolean to its SIL internal representation.
3. **Interval -> interval**

   Translates an interval to its SIL internal representation.

   The custom field value can either be the user friendly string representation (1d 2h) or the number of seconds.
4. **User[] -> string []**

   Translates a collection of Users to a string array.

   The collection can also be represented as a single string with values separated by a pipe (|).
   The values represent the usernames.
5. **Group -> string**

   Translates a group to a string value representing the group name.
6. **Number[]-> number []**

   Translates a collection of numbers to a number array.

The collection can also be represented as a single string with values separated by a pipe (|).

7. **Option-> string**
Translates an option to a string value.

8. **Number -> number**

Translates a number to its SIL internal representation.

9. **Boolean[]-> boolean[]**

Translates to an array of boolean values.

Custom field value can be a collection of boolean values or their string representation separated by a pipe (|).

10. **Date[] -> date []**

Translates a collection of dates to a date array.

The collection can also be represented as a single string with values separated by a pipe (|).

11. **Date -> date**

Translates a date value to its SIL internal representation.

12. **User -> string**

Translates a user to a string value representing the username.

13. **Label[] -> string []**

Translates labels custom field to a string array containing the labels as strings.

14. **String -> String**
Translates to a string value.

15. **Interval[] -> interval []**

Translates a collection of intervals to an interval array.

The collection can also be represented as a single string with values separated by a pipe (|).

The interval values can be represented either in a user-friendly string form (1d 2h) or in seconds.

16. **Group[] -> string []**

Translates a collection of Groups to a string array.

The collection can also be represented as a single string with values separated by a pipe (|).

The values represent the group names.

17. **Cascade -> string []**

Translates to an array of string values. First is key, second is value.

18. **Collection(String) -> string []**

Translates a collection of string values to a string array.
The collection can also be represented as a single string with values separated by a pipe (|).

Table of custom fields

For every type of custom field available, a list of existing custom fields of the respective type is displayed like in the picture below:

That's it! However, if any exceptions occur, check the log for details on what went wrong.

Live Fields Configuration

If you want your JIRA fields do whatever you want whenever you want, you have to make a **Live Fields Configuration** and associate it with a project.

You can do this from **Administration -> Add-ons - >Live Fields.**

> **Info**
> For more information see Live Fields.

Add configuration

To add a configuration you have to click the **Add Configuration** button.

In the displayed dialog box you have to enter the configuration name and description and you have to choose a SIL File for the Live Fields Configuration.
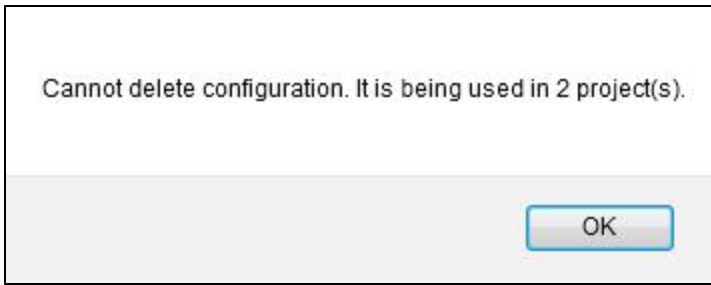

where:

Name - Live Fields configuration name

Description - Live Fields configuration description

SIL File - the SIL script that will be executed on every issue page for the associated projects.

You can also edit and remove a configuration by clicking the **Edit/Delete** link. The configuration will be removed if there aren't projects associate with it.

Cannot delete configuration. It is being used in 2 project(s).
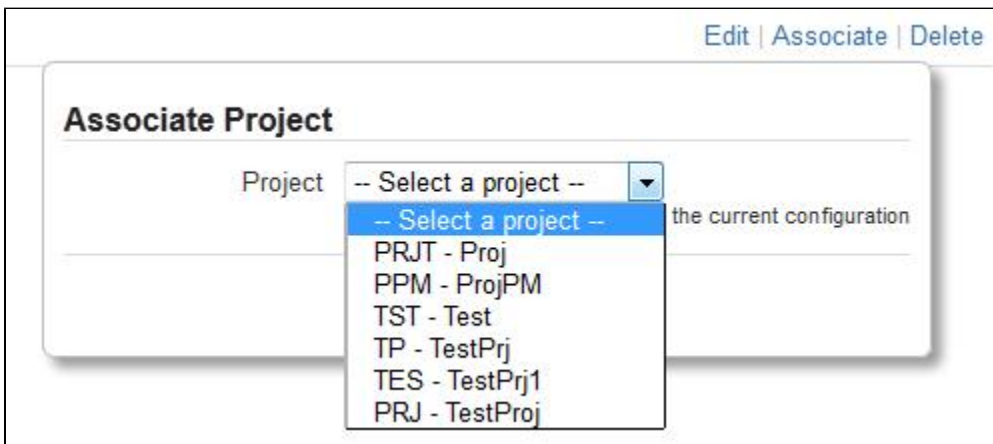
OK

Associate project

Now, you have to associate the configuration with a project. You can do this by clicking the **Associate** button from the configuration row.

Here you can manage all your live fields configurations.

Add Configuration

Name	Description	SIL File	Projects	Operations
lftest1		C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\LiveField2.sil	• PPM ⊗	Edit | Associate | Delete
lftest		C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\LiveField1.sil	• PRJ ⊗ • TST ⊗	Edit | Associate | Delete
lftest2		C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\LiveFieldTest.sil		Edit | Associate | Delete

A dialog will be displayed from where you can choose the project to associate with.

Edit | Associate | Delete

Associate Project

Project   -- Select a project --

-- Select a project --
PRJT - Proj
PPM - ProjPM
TST - Test
TP - TestPrj
TES - TestPrj1
PRJ - TestProj

the current configuration

You can also remove an association by clicking the **Remove project** red icon at the right of the project you want to remove.

Associate project from Project page

You can also associate a Live Fields Configuration with a project from the Project page, in Administration. You can do this in the Live Fields Config tab from the project page.

Versions

Components

Issue Collectors

Live Fields Config

From the **Configuration** select you can choose the Live Fields Configuration for the project.

Live Fields Configuration

lftest

Select and save a live fields configuration for the current project from the select list below. The configuration SIL script will get executed on page load for all issues in the current project.

Configuration [lftest   ▾] [Save]

Name	Description	SIL File
lftest		C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\LiveField1.sil

After associating a project with a Live Fields Configuration, the SIL file from that configuration will be executed on every issue page of that project.

SIL Configuration

In this page you can configure specific parameters for the SIL language, such as email options, SIL Tree caching and programming warnings.

Navigate to **Administration -> Add-ons -> SIL Configuration** to get to the administration page for the SIL language configuration:

Configuration for SIL language

Generic Configuration

Charset [windows-1252 ▾]
Charset that will be used to read from files on disk.

Email Templates Directory [emails]
Path for the directory where the email templates will be stored. Can be either absolute, or relative to the Kepler directory in JIRA_HOME.

Default Email language [Sender language ▾]
Default language to use with internationalized email templates.

Email Sender [SMTP Direct ▾]
Email channel to use when sending emails.

SIL Tree Caching

Caching Enabled ⦿ ON   ○ OFF
If enabled, caches the parsed SIL trees for reuse. This will reduce the time needed to run a cached script by ~50%.

Cache Size [100 ▾] scripts
Provides the number of scripts to hold in the cache. When full, the cache will replace existing entries based on a LRU algorithm.

Clear Cache [Clear]
Flushes all cached scripts. New scripts will need to be parsed before being added into the cache.

SIL Programming Warnings

Enable Warning Report ○ ON   ⦿ OFF
If enabled, will print a report in the logs showing any warnings we found during execution of the script.
This will be useful especially when developing new scripts or debugging old ones.
Note: Since the warnings are useful mostly for debugging purposes, this setting will not be kept between restarts.

[Save]

1. Generic Configuration

Here you can configure the following parameters:

Charset - charset that will be used to read from files on disk.

Email Templates Directory - path for the directory where the email templates will be stored.

Default Email language - default language to use with internationalized email templates (sender language or receiver language)**.**

Email Sender - see Mail Configuration

For more details regarding email configuration check the Mail Configuration section.

2. SIL Tree Caching

For caching your scripts and reduce the time needed to run them you have to configure the next parameters:

Caching Enabled - if enabled, caches the parsed SIL trees for reuse, reducing the time needed to run a cached script by ~50%.

Cache Size - the number of scripts to hold in the cache.

Clear Cache - removes all the scripts from the cache.

3. SIL Programming Warning

This parameter is useful especially when developing new scripts or debugging old ones.

Enable Warning Report - if enabled, will print a report in the logs showing any warnings that has been found during execution of the script (useful especially when developing new scripts or debugging old ones).

Mail Configuration

The SMTP server used by JJupin is the same JIRA is using. You do not have to configure anything special here.

Email templates

Configuration for SIL language

Generic Configuration

Charset	windows-1252 ▼
	Charset that will be used to read from files on disk.
Email Templates Directory	emails
	Path for the directory where the email templates will be stored. Can be either absolute, or relative to the Kepler directory in JIRA_HOME.
Default Email language	Sender language ▼
	Default language to use with internationalized email templates.
Email Sender	SMTP Direct ▼
	Email channel to use when sending emails.

SIL Tree Caching

Caching Enabled	○ ON ● OFF
	If enabled, caches the parsed SIL trees for reuse. This will reduce the time needed to run a cached script by ~50%.
Cache Size	100 ▼ scripts
	Provides the number of scripts to hold in the cache. When full, the cache will replace existing entries based on a LRU algorithm.
Clear Cache	Clear
	Flushes all cached scripts. New scripts will need to be parsed before being added into the cache.

SIL Programming Warnings

Enable Warning Report	○ ON ● OFF
	If enabled, will print a report in the logs showing any warnings we found during execution of the script. This will be useful especially when developing new scripts or debugging old ones. **Note:** Since the warnings are useful mostly for debugging purposes, this setting will not be kept between restarts.

Save

The email templates folder, as well as the email language are configurable and can be set from the SIL Configuration admin page under Generic Configuration.

At runtime, when a template is requested, it looks for templates in a locale folder within the default template folder. (Ex:*mydefaultfolder/en_US/template.tpl*, then it will look for *mydefaultfolder/template.tpl* - assuming you configured *mydefaultfolder* as a template directory).

Within the templates, any standard or custom field defined in the issue that called the routine can be referenced using the notation **$field$.**

Example:

```
Hello $recipient$!, the sender $sender$ announces you that the assignee for
issue $key$ is $assignee$
```

At runtime, the plugin will replace with real values the body of the email.

> **Warning**
> *$recipient$* and *$sender$* will only work if the addresses belong to JIRA users and not some external email addresses.

> **Tip**
> You can create, edit, delete and upload email templates using the built-in browser and editor in the administration page.

The **Email Sender** selection box contains two options:

- **SMTP Direct** (default) - will attempt to connect directly to the default SMTP server and immediately send emails
- **JIRA Mail Queue** - will create an item in the default JIRA Mail Queue and will be sent along with other JIRA email notifications when the queue is flushed.

Remote Systems

Remote SIL configuration

If you enable the remote SIL, you will be able to execute SIL programs on some other JIRA instance running SIL using REST. It does not require any other library file and it has a lower encoding footprint. The configuration is kept in the *rest-client.properties* file.

REST:

- Requires only katl-commons, this gives you liberty
- Documentation is here

Resolution when calling a remote system

As you know, you can call a remote system via a call() routine invocation. The resolution of the system is as follows:

1. Try to see if the name of the system is empty ('') or the string 'local'. If yes, it will call a local script
2. Next, try to find the name of the system as defined by REST. If it is defined, it calls the REST remote system
3. If it is not defined, error

REST Remote Systems

REST Remote Systems

Using REST remote systems you will be able to execute SIL programs on some other JIRA instances running SIL.

A file named '*rest-client.properties*' should be placed in the kepler directory (along with the licenses). For each system, you should configure the URL and the connection details.

Example:

The following defines two remote systems: 'REMOTE' and 'ANOTHER':

```
REMOTE=http://192.168.17.112:8080
REMOTE.user=admin
REMOTE.password=admin

ANOTHER=http://192.168.17.113:8080
ANOTHER.user=admin1
ANOTHER.password=admin123
```

An easier way is to manage the REST remote systems from the special administration page at **Administration > Add-ons > REST Remote Systems**

Remote Systems Configuration

REST Remote Systems

Here you can configure the remote systems that you can access via their local SIL REST service provided by katl-commons. Once you have set up the systems, you will be able to run remote scripts with the **call** routine using the systems name.

Name	URL	Username	Password	Actions
REMOTE	http://192.168.17.112:8080	admin	Hidden	Edit | Delete
ANOTHER	http://192.168.17.113:8080	admin1	Hidden	Edit | Delete
				Add

Local REST Service Parameters

Allowed REST Users   admin, developer1, developer2, developer3, developer4

Specify the users that are allowed to call SIL scripts using the REST service. Note that this also applies to the **call** routine for REST clients.

Save

Here you can add, edit and delete REST remote systems in an instance.

In the **Local REST Service Parameters** section you can specify the Allowed REST Users for calling SIL scripts using the REST service.

SQL Configuration

SQL Configuration

To execute the SQL function, one must define first the datasource. By default, JIRA runs in Tomcat, so the following example applied to Tomcat only. The user should refer to the application server manual on how to define a datasource.

1. First, make sure you have the SQL driver in JIRA_INSTALL_DIR/lib directory.

2. Then, open with your favourite text editor **JIRA_INSTALL_DIR/conf/context.xml** file. Enter your datasource there, for instance:

```
<Context>
  <Resource name="TestDB" auth="Container" type="javax.sql.DataSource"
            username="sa" password=""
            driverClassName="org.hsqldb.jdbcDriver"
            url="jdbc:hsqldb:/tmp/somedb;create=true;"
  />
<Context>
```

3. Restart JIRA.

4. Check the tomcat logs for errors

You should be now ready to use the datasource you just defined via the sql() calls, the JNDI datasource name you just created is named "**TestDB**".

> **Note**
> The above example works for HSQL DB, which is the embedded JIRA database. You should use correct driver class and a correct URI syntax for your database. More examples for different databases can be found at Data Source Configuration.

> **Note**
> The Guide on how you should configure a datasource is here: Apache Tomcat: Configuring a Datasource

LDAP Configuration

LDAP configuration

We had the option to read the LDAP parameters values from osuser.xml file, but some customers wanted lookaside LDAPs (and not integrated ones), thus we'll keep this config aside (and most possibly duplicated, but what can we do?).

Go to **Administration -> Add-ons -> LDAP Configuration**. The following screen appears:

Example settings:

```
URL = ldap://localhost:389
BIND USER = cn=binduser,ou=IT Group,dc=alpha,dc=local
PASSWORD = passw0rd
BASEDN = dc=alpha,dc=local
```

> **Warning**
> Right now, only the Microsoft Active Directory is supported, though it might work with other systems too (e.g. works with OpenDS). However, we are eagerly waiting for requests to extend this functionality to different LDAP servers.

Configuring a SIL JIRA Service

Configuring a SIL JIRA Service

If you would like to periodically run a SIL script, you will have to install JJupin as a service. Here is how to do that:

- Go to **Administration -> Services**
- Give your SIL service a name
- Under "Class" put **com.keplerrominfo.jira.plugins.jjupin.services.SILService**
- Set the delay.

Add Service

Add a new service by entering a name and class below. You can then edit it to set properties.

Name: My SIL Service

Class: com.keplerrominfo.jira.plugins.jjupin.services.SILService
> Built-in Services

Delay: 100
Delay between running time, in minutes.

[Add Service]

Next, you will be presented with a configuration screen, where you should:

- Enter the **absolute path** to the file containing the SIL script
- Choose a user the service will run as (if the user doesn't have administrator privileges, some SIL routines might not work).

Edit Service: My SIL Service

Description:
This service will run the specified SIL program.

Enter text values for service properties below. Any empty fields will be set to NULL in the Service's initialization.

User: admin
Start typing to get a list of possible matches.
Run as this user

SIL Script: C:\program.sil
file to run

Delay: 100
Delay - in minutes
You can also adjust the delay period of this service. Note that if you adjust this delay, the service will be restarted.

[Update] Cancel

That's it! However, if any exceptions occur, check the log for details on what went wrong.

Tip

You can also try out our SIL Services & Scheduler page.

Configure JIRA Logging

Configure JIRA Logging

JIRA uses Log4J as a logging system. We're interested in output messages produced by our plugins so you will need to configure logging.

1. Open *JIRA_HOME/atlassian-jira/WEB-INF/classes/log4j.properties* with your favorite text editor.
2. Append the following lines (add them at the end):

```
log4j.category.com.keplerrominfo = INFO, console, filelog
log4j.additivity.com.keplerrominfo = false
```

Note
If you do not perform this configuration step, some routines such as print() will not output messages.

For debug add these 2 lines into *JIRA_HOME/atlassian-jira/WEB-INF/classes/log4j.properties* file:

```
log4j.category.com.keplerrominfo = DEBUG, console, filelog
log4j.additivity.com.keplerrominfo = false
```

Warning
Setting the level to DEBUG will output a lot of messages and it will hurt your performance. Do this only when instructed by the Kepler Team.

Info
For more information see JIRA Documentation: Logging and Profiling.

Licensing

Dual Licensing support

Versions 2.0.4 and up support both Kepler and Atlassian licenses, but you only need one valid license to run the plugin, which can either be provided as the **jjupin.lic** file, or as the key generated via the Atlassian Marketplace.

The order in which the licenses are checked is:

1. Atlassian License
2. Kepler License

It is **strongly recommended** that you do not install both licenses at once, as this might yield unwanted results. For example, consider that you have an Atlassian License with the support date expired and one valid Kepler License. In this case you cannot update the plugin, because the Atlassian License is checked first and its support date is expired.

Atlassian Licensing

Note
To support Atlassian licenses you need to install **katl-commons 2.0.4+** *before* installing JJupin.

The Atlassian Marketplace allows you to easily purchase or generate an evaluation license for **JJupin**.

Using Universal Plugin Manager 2.0.1+

After generating the license key, all you have to do is access the **Administration-> Plugins** section in your JIRA instance and paste the key into the corresponding plugin textbox.

Kepler Licensing

The Kepler license is a file (**jjupin.lic**) which must be placed in the <JIRA_HOME>/kepler folder. You can either generate and download a free evaluation license by registering on our site and accessing the **Licenses** section, or you can purchase the plugin by following these instructions.

You can view details for all the license files situated in the kepler folder, by accessing the **Kepler Licenses page** from **Administration > Add-ons > Kepler Licenses** menu:

The page shows the expiration and maintenance date, user limit and validity message for each selected kepler license.

If the license is expired, user limit is exceeded or license is targeted for a different JIRA server id, a red colored message shows the status:

Kepler Licenses

Here you can inspect all license files from your kepler home directory (D:\Atlassian\JIRA 5.2.4\Application Data\kepler).

License kontinuum.lic ▼
Select a license file to see its details.

License Information

Expiration Date	31/Jan/13	
Maintenance Date	31/Jan/13	
User Limit	3	
Valid	NO	License EXPIRED on 31/Jan/13

If kepler license is close to expiration date (less than 10 days remaining), a warning message is displayed, showing the remaining time:

Kepler Licenses

Here you can inspect all license files from your kepler home directory (D:\Atlassian\JIRA 5.2.5\Application Data\kepler).

License jjupin.lic ▼
Select a license file to see its details.

License Information

Expiration Date	22/Feb/13	
Maintenance Date	22/Feb/14	
User Limit	Unlimited	
Valid	YES	Only 7 day(s) remaining

> **Reminder**
> Don't forget that you need only *one* valid license to run the plugin.

> **Technical info**
> Starting with **katl-commons version 2.5.5** an new plugin, called **Warden**, will be automatically installed by any paid add-on (including JJupin 2.5). This plugin is responsible with the management of licenses (both JIRA and Kepler). Do not attempt to uninstall it without removing first all the Kepler paid add-ons.

> **Removing an unused license**
> If you want to remove a no longer used Atlassian license, this can be done in UPM (for UPM 2.0.1+) by removing the old license key and clicking Update. To remove a Kepler license, you have to delete the correspondent .lic file from the kepler folder. Note that any change to the Kepler license requires a server restart.

Uninstall

Uninstall via Atlassian Universal Plugin Manager

This page shows the steps to uninstall the plugin using the Universal Plugin Manager.

Manual Uninstall

At first sight, this seem a little bit complicated but actually it isn't. All it has to be done is to remove the plugin manually and delete its tables from the internal database.

- Manual Uninstall
- Uninstall via Atlassian Universal Plugin Manager

Manual Uninstall

Uninstall manually

At first we will uninstall the plugin manually and finally we'll remove the corresponding tables in the internal database.

Uninstall the plugin

> You need to have access where the Jira server has been installed.

Goto the folder where Jira server has been installed.

Access **<JIRA_APPLICATION_DATA>/plugins/installed-plugins** and manually delete JJupin plugin.

Remove the tables

You can go to the internal database administration tool.

You can use a visual tool or a command line tool and remove the following tables in your database:

- krunnablesils
- klistenersils
- jjlf_config
- jjlf_project
- jjlf_category

Restart the server

Now you can restart Jira server

Uninstall via Atlassian Universal Plugin Manager

Uninstall via Atlassian Universal Plugin Manager

At first we will uninstall the plugin and finally we'll remove the corresponding tables in the internal database.

Uninstall the plugin

If you are not familiar with Universal Plugin Manager (UPM), please read this document before we begin.

1) Log in as administrator and go to Administration->Add-ons->Manage add-ons
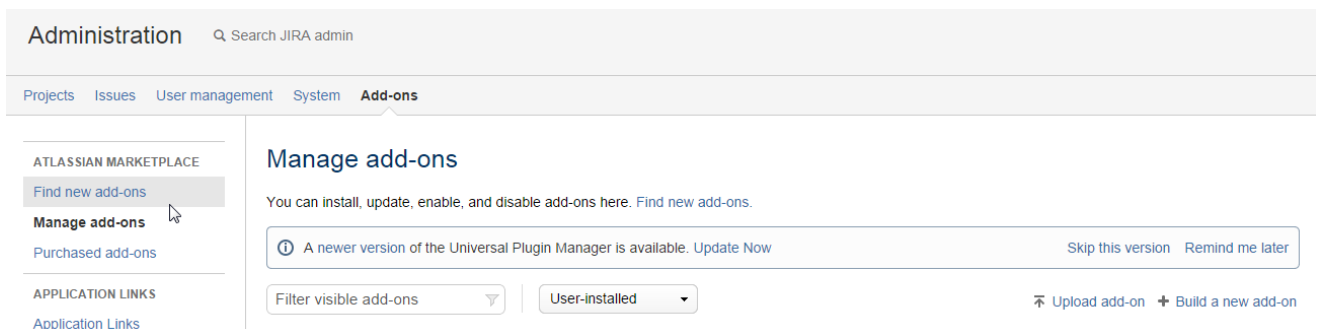
2) Search for JJupin plugin in `User-installed add-ons` section and click on `Uninstall` button

3) Press `Continue` when the uninstall confirmation dialog box appears

4) A message "successfully uninstalled" should appear

Now you can delete JJupin corresponding tables.

Remove the tables

You can go to the internal database administration tool.

You can use a visual tool or a command line tool and remove the following tables in your database:

- krunnablesils
- klistenersils
- jjlf_config
- jjlf_project

- jjlf_category

User guide

In this section, you will learn about the friendly user interface that JJupin offers and its capabilities.

> **Note**
> Step-by-step guides, previews, demo images and screenshots were made under JIRA 6.x.

> **Info**
> Before using JJupin check out the Simple Issue Language documentation for a better grasp of SIL usage and capabilities.

Writing Validators, Postfunctions and Conditions

Writing Validators, Postfunctions and Conditions

After you installed JJUPIN plugin, you should go to your JIRA's *Administration->Workflows* page and create a workflow, associated with a project. Since you cannot modify the workflow while it's active, you must create a draft workflow, as specified in the JIRA documentation, by pressing the "Create draft" link. The result should look like in the below excerpt.

Clinking on a transition will show the transition's conditions, validators and postfunctions.

You should always keep in mind that:

1. Whenever an issue advances from one state to another the postfunction will be called.

2. The transition is made possible only if the conditions are fulfilled. Therefore, a condition **must** return true or false to signal that the condition is fulfilled or not

3. The validators must validate the data before transition is fired. Subsequently, a validator is entitled to return true or false and optionally the field and the error message you want to show in the interface.

An important consequence of the above model is that conditions and validators should not have side-effects. In fact, JJupin is discarding modifications of the issues, allowing them to occur in the postfunction only, but it cannot discard modifications made on another database, for instance, applied using the sql routine (see sql() routine for details).

To create conditions, validators and postfunctions, press the corresponding add link, found at the top of the workflow management tab.

The following image shows the creation of a test post-postfunction:

Add Post Function To Transition

Name	Description
⦿ (k) SIL Post-function	Runs a SIL program as a postfunction
○ Assign to Current User	Assigns the issue to the current user if the current user has the 'Assignable User' permission.
○ Assign to Lead Developer	Assigns the issue to the project/component lead developer
○ Assign to Reporter	Assigns the issue to the reporter
○ Create Perforce Job Function	Creates a Perforce Job (if required) after completing the workflow transition.
○ Update Issue Field	Updates a simple issue field to a given value.

Add   Cancel

After you'll press the "Add" button, you will be ready to write your SIL (in this case, a SIL postfunction)

Update parameters of the (k) SIL Post-function Function for this transition.

Update parameters of the (k) SIL Post-function Function for this transition.

Go to: Plugin configuration | Workflow Viewer |

Edit options: ⦿ Update script
                    ○ Reuse existing script

Name: clone pf   ⑦

The file of your program is stored as: C:\Program Files\Atlassian\JIRA_6.0\home\silprograms\startPF.sil

Click here to change the default path.

Sil code:

CHECK   Q Search   Replace   Settings   Legacy Editor

```
1  // Create one or more sub-tasks on transition, and optionally reopen existing sub-tasks
2  string[] SUMMARIES = {"First sub-task for approval", "Second sub-task for approval"};
3  string[] subtasks = subtasks(key);
4
5  for(string summ in SUMMARIES) {
6      boolean exists = false;
7      for(string subtask in subtasks) {
8          if(%subtask%.summary == summ) {
9              autotransition("Reopen Issue", subtask);
10             exists = true;
11         }
12     }
13     if(!exists) {
14     createIssue(project, key, "Sub-task", summ);
15     }
16 }
```

> You can create a new script or pick a script that was already created (or even used for other purposes) in the silprograms folder.

By providing a meaningful name to your program and by pressing the "Add" button, you are now ready to extend your JIRA Workflow:

Returning into the transition screen, your newly added post-function will be reflected in the view:

Transition: Start Progress

> ℹ You are editing a draft workflow. View the original workflow or publish this draft.

Transition View: None - it will happen instantly

- View **workflow steps** of **ATestWorkflow (Draft)**.
- **Edit** this transition.
- **Delete** this transition.
- View **properties of this transition**

All **Conditions** (1) **Validators** (0) **Post Functions** (7)

☐ **Add** a new post function to the unconditional result of the transition.

This will run the **test** SIL program
The file of your program is stored as: C:\jira\home\silprograms\test.sil
```
//My SIL program wich prints the log
```

Edit | Move Down | Delete

— THEN

The **Resolution** of the issue will be **cleared**.
Edit | Move Up | Move Down | Delete

— THEN

Set issue status to the linked status of the destination workflow step.

— THEN

Add a comment to an issue if one is entered during a transition.

— THEN

Update change history for an issue and store the issue in the database.

— THEN

Re-index an issue to keep indexes in sync with the database.

— THEN

Fire a **Work Started On Issue** event that can be processed by the listeners.
Edit

Warning:

> **Modifying Issues**
> You should **avoid** modifying issues in conditions and validators, as they are supposed to be read-only. Do not yield to that temptation! You should modify issue values (or create new issues, or change anything) in the postfunction only.

> **Info**
> Your SIL program will be saved on the disk in the folder specified in the configuration. The filename is obtained by removing any invalid characters from the program name you entered and appending a number to help you browse through different versions of the same file.

Changing the Default Save Path

When editing or creating a new condition, validator or post-function, you will notice a link saying "Click here to change the path". By default, JJupin saves all the programs in a (configurable) folder, but you also have the ability to select a different one. By clicking on the link, you will be presented with the following view:
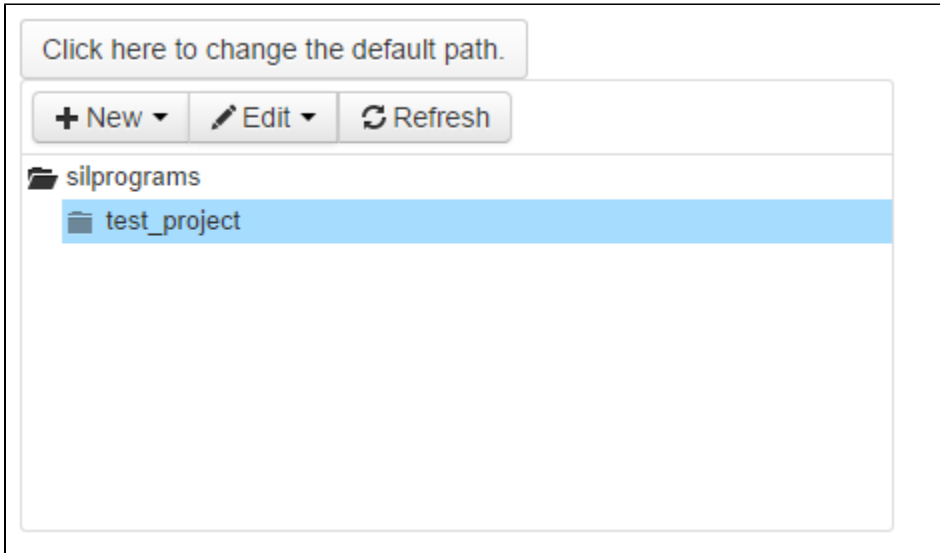
Here you can choose an already existing folder to save your file into and even create and delete new ones.

The name of the file your program will be saved as is calculated by removing any invalid characters from the program name and appending a version number at the end.

> **Note**
> For best experience, we recommend Google Chrome or Mozilla Firefox.

Return codes:

Returns codes are different for validators, conditions and postfunctions

For validator:

```
return false, "assignee", "We have failed, assignee is not ok";
```

The first field tells us that we have failed, the second indicates the field, the third is the message that will be set on the user interface. For the moment, the filed name must be a "bare" name. That means that it should comply with the name given to the HTML objects displayed (e.g: for customfields it will be customfield_xxxxx). One can easily inspect the HTML source of the edit screen to see the "bare" name of a field.

For condition:

```
return false; //to signal that condition is not fullfilled.
```

Just tell JIRA this condition is not fullfiled.

For postfunctions:

```
return;
```
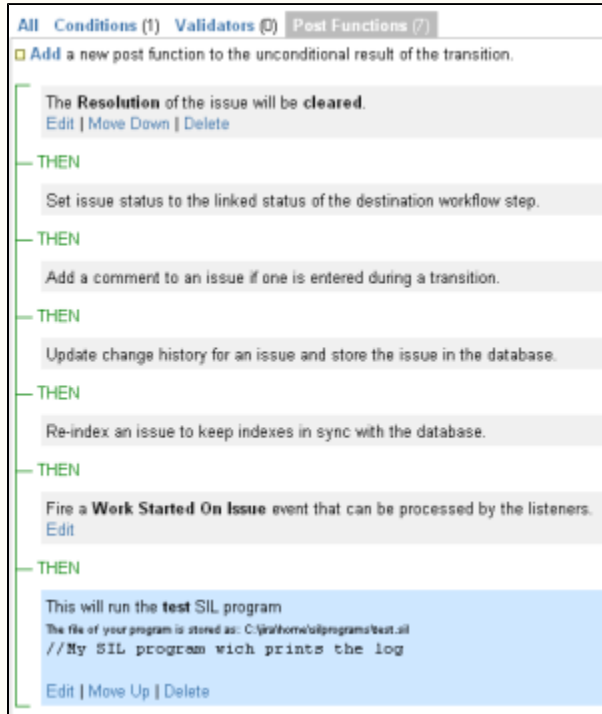
return ends the program, any values are ignored.

See also:

[JIRA Documentation: Configuring Workflows](#)

[JJUPIN Tutorials](#)

Transition View

This page is restricted since I'm not sure whether we should keep it or not.

On this page:

Introduction

The user-friendly interface offers a number of visual aids that will help you write complex SIL programs in no time.

Editing Code

> **Help Tip**
> When writing SIL programs, click the **Help!** button for a shorter version of this guide.

Highlighting

SIL-specific syntax highlighting can considerably improve the readability of your code.

There are four types of highlighting:

1. **keywords** - words like if, do, while, else, return, etc. are colored in **brown**
2. **datatypes** - datatypes like string, number, boolean, etc are colored in **blue**
3. **constants** - numbers, strings, etc. are colored in **green**
4. **brackets** - when the cursor is near a bracket, its background will become green or red depending on whether that bracket has a closing pair or not.

Auto-completion

The SIL Editor also offers auto-completion capabilities by pressing **Ctrl+Space**.

The list of suggestions contains standard variables, routines as well as UDRs and local variables defined up to that point. The **scope** of the variables is also taken into account. In the example above, you can see that the **user** and **name** variables are defined in the **for** statement, so their scope is the **for** block. Variables that are not in scope (for example the parameters of the doSomething() function) will not be shown in the list. Therefore, if you write outside the **for** block, variables **name** and **user** will not be shown in the list.

Another useful feature is the dynamic population if the suggestions list and the auto-selection if the list has only one entry.

> **Tip**
> Notice that the suggestions list is NOT case-sensitive.
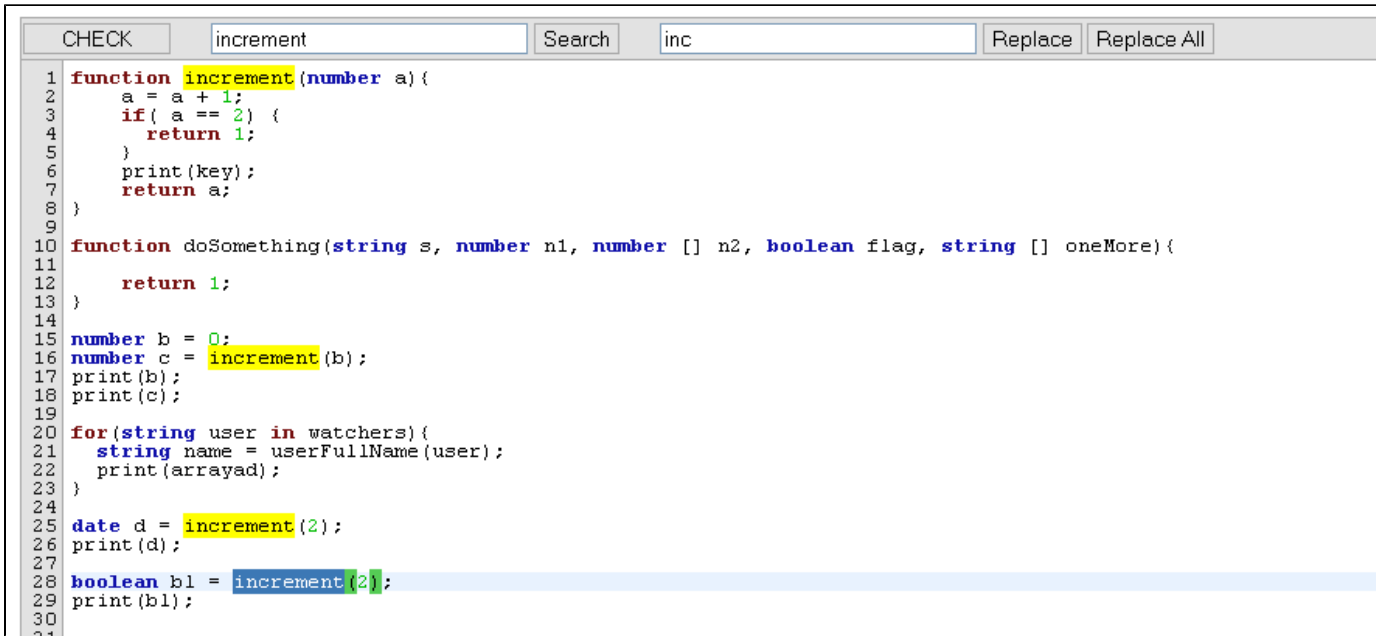
> **Note**
> UDRs and variables defined inside programs included with the include statement, will not be visible in the suggestions list.

Indenting

Selecting a whole block of code and pressing **Tab** will indent it further to the right. To decrease indentation (move it to the left), select the block and press **Shift+Tab**.

Search and Replace

The SIL Editor also offers search and replace capabilities using the panel on the upper side of the editor.

Notice that all the occurrences of the searched term are highlighted in yellow. You can replace the current occurrence (highlighted in blue) by pressing **Replace** or you can replace all of them using the **Replace All** button. You can cycle through the search results by repeatedly pressing **S earch**.

> **Tip**
> To un-highlight search results press **Esc**.

Checking the Program

To help you through each step of writing a SIL program, the editor also offers a live checking capability using the **CHECK** button.

Feel free to use this to check for errors at any time.

If the program is correct, a message written in green will apprear saying "Syntax Ok.". Otherwise, an error written in red and containing detailed information will appear. You will also notice that if there are errors the line where they occurred will be highlighted in red.

CHECK		Search		Replace	Replace All

```
Syntax Ok! increment(number a){
2       a = a + 1;
3       if( a == 2) {
4          return 1;
5       }
6       print(key);
7       return a;
8  }
9
10 function doSomething(string s, number n1, number [] n2, boolean flag, string [] oneMore){
11
12      return 1;
13 }
14
15 number b = 0;
16 number c = increment(b);
17 print(b);
18 print(c);
19
20 for(string user in watchers){
21    string name = userFullName(user);
22    print(name);
23 }
24
25 date d = increment(2);
26 print(d);
27
28 boolean bl = increment(2);
29 print(bl);
30
```

CHECK	

```
Encountered " "print "" at line
2       a = a + 1;
3       if( a == 2) {
4          return 1;
5       }
6       print(key);
7       return a;
8  }
9
10 function doSomething(string
11
12      return 1;
13 }
14
15 number b = 0;
16 number c = increment(b);
17 print(b)
18 print(c);
19
20 for(string user in watchers)
21    string name = userFullName
22    print(name);
23 }
24
25 date d = increment(2);
26 print(d);
27
28 boolean bl = increment(2);
29 print(bl);
30
```
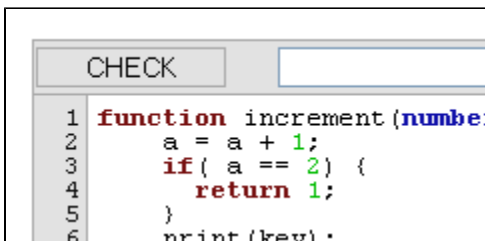
> **Tip**
> To discard the error notifier, just click it.

Changing the Default Save Path

When editing or creating a new condition, validator or post-function, you will notice a link saying "Click here to change the path". By default, JJupin saves all the programs in a (configurable) folder, but you also have the ability to select a different one. By clicking on the link, you will be presented with the following view:


```
?unnamed SIL?

The file of your program is stored as: C:\Program Files\Atlassian\JIRA 4.3.4\home\silprograms\includes\unnamed_16.sil

Click here to change the path.

[New] [Delete]
⊞ 📂 C:\Program Files\Atlassian\JIRA 4.3.4\home
⊞ 📂 E:\
   📂 C:\silprograms
```
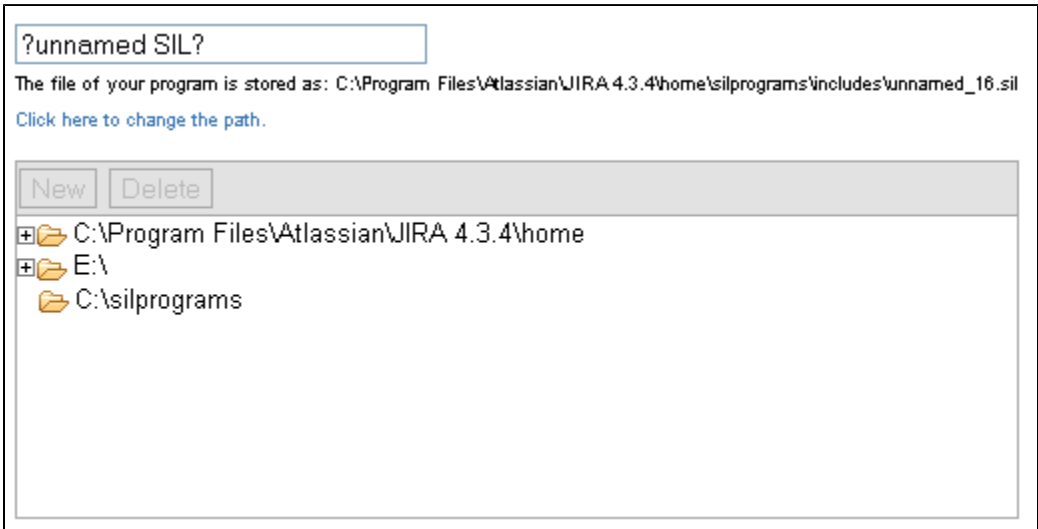
Here you can choose an already existing folder to save your file into and even create and delete new ones.

The name of the file your program will be saved as is calculated by removing any invalid characters from the program name and appending a version number at the end.

> **Note**
> For best experience, we recommend Google Chrome or Mozilla Firefox.

See Also

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

Note: Our SIL editing capabilities are based on a modified version of CodeMirror.

Workflow View

Workflow View

This view will help you browse through your workflow without having to open the program every time to see what it does.

- On the first line we have the **name of the SIL program**
- The second line shows the **path** of the file which contains the actual code.
- After that, you have a short **description** of the program, which you can write by commenting on the first lines (max. 3 lines) in your code. For example, the program you see on the right contains "//Your SIL code should go in here" on the first line.
- Finally, you have the **error notifier** which tells you if the program is correct. If there are any errors, open the program for a more detailed description of the cause. If the program is correct, this line will be blank.

Workflow Viewer

Workflow Viewer

Displays all information needed for a workflow or a draft workflow, (optional) associated to a given project. To get access to workflow viewer, you have to be an administrator.

Usage

Navigate to **Administration Page** Administration -> Add-ons -> Workflow Viewer.

You can choose the **JIRA project** to see the associated workflows, including draft workflows (if any), for each issue type associated OR you can choose directly the workflows or draft workflows to display all information about.

The report look like this:

- **Initial state** - indicates the **initial** step name
- **Transition** - the name of the **linked status**
- **Final state** - the **destination** step name
- **Conditions** - the text representation of the **conditions tree** including SIL Conditions and their detailed information
- **Input data** - the **input screens** when executing transitions. An input screen can contain multiple tabs and the information displayed contains all input data, e.g. Custom Fields names and the associated IDs
- **Validators** - the list of **validators** on the current transition, including SIL Validators and their detailed information

- **Post functions** - the list of **post functions** on the current transition, including SIL post-functions and their detailed information

At the end of the report you'll find the additional information about the Validators, Conditions and Post Functions present in the edit or view screens included in the issue screen scheme, associated to the current issue type:

| - | View Issue | - | - | Issue Type(s): [Bug, New Feature]

Default Screen

Field Tab
• Summary (*summary*)
• Issue Type (*issuetype*)
• Security Level (*security*)
• Assignee (*assignee*)
• Description (*description*) | - | - |
| - | Edit Issue | - | - | Issue Type(s): [Bug, New Feature]

Default Screen

Field Tab
• Summary (*summary*)
• Issue Type (*issuetype*)
• Security Level (*security*)
• Assignee (*assignee*)
• Description (*description*) | - | - |

> **Info**
> If there is no project selected for workflow viewer, then there will be no associated issue type to a given workflow. So there will be no such issue screen scheme associated and the information about the edit or view screen will be missing.

When clicking on **Show All SIL programs** you will see the contents of all SIL programs associated to the Validators, Conditions and Post Functions for each transitions. You can also show the contents of each SIL program by clicking the **show** link near the SIL indicated location. You can collapse all the SIL programs at once by clicking the **Hide All SIL programs** link, or one by one by clicking the **hide** link for each of them.

SIL Runner Gadget

> **Important!**
> Since JJUPIN 3.0.8, you can now customize the gadget to be more user friendly, asking **parameters** more nicely. See more details [her e](#).

Another useful feature of JJupin is the ability to randomly run SIL scripts **on demand** using the SIL Runner Gadget. This allows you to configure a list of scripts that can be run at any time directly from your Dashboard.

> **Important!**
> Bare in mind that scripts ran this way **do not have an issue context**! Therefore, constructs and keywords like 'key' do not have a meaning here (they are undefined!). You need to first select the issues to work with, and prefix any standard variables with the issue key!

Configuration

The **configuration screen** is only available to JIRA Administrators and System Administrators and allows them to manage the list of available SIL scripts. They can add and delete scripts or edit the parameters of a **runnable SIL script**.

To add a script to the runner you must give it a name, description and select an already existing file containing the script.

The gadget also offers the ability to restrict script usage to specific users or groups by choosing a security option:

Public - the script will be available for any user

Group - the script will be available only if the currently logged in user is a member of the specified group (will display a group picker)

User - the script will be available only if the currently logged in user is the same as the specified one (will display a user picker)

 Project role - the script will available only if the currently logged in user is in a specific role on a specific project (will display a project picker) - **(available since JJUPIN 3.0.8)**

Info
To edit the actual scripts, please use the SIL Manager.

Usage

Example

When you select a script from the list, its **description** will automatically be filled in below.

The **Parameters** field is used to pass values **into** your SIL program. To add a parameter click the **Add Parameter** button.

> Parameter names must be unique, otherwise the most recent definition will overwrite previous ones. This includes parameters with no name.

The parameters will be passed into the program using the **argv** variable. The values will be available using a construct like **argv["parameter_name"]** or **argv[position].** For the above example, the number of rockets can be retrieved using argv["rockets"] or argv[2].

> **Tip**
> Parameters can be reordered using drag and drop.

Once you run the script, the program console will be displayed.

SIL Runner Gadget

Reset

Console

Running script Start a war
Preparing to start a war...
Building tanks...
Built 6121 tanks.
Gathering infantry...
Gathered 15246 brave men.
Fueling rockets...
a big one ready.
Dispatching orders...
Done. Program returned: Good job! The world is now at war!

> You can use the runnerLog routine to print info in the console as the program runs. Note that the console buffer is limited to 512 lines every ~0.5 sec and the console will only display the latest 512 lines.

Example code

```
runnerLog("Preparing to start a war...");

runnerLog("Building tanks...");
runnerLog("Built " + argv["tanks"] + " tanks.");

runnerLog("Gathering infantry...");
runnerLog("Gathered " + argv["infantry"] + " brave men.");

runnerLog("Fueling rockets...");
runnerLog(argv["rockets"] + " ready.");

runnerLog("Dispatching orders...");
return "Good job! The world is now at war!";
```

> **Tip**
> You can return as many values as you need, regardless of their type.

Parameters in SIL Runner Gadget

Parameters in SIL Runner Gadget

Since **JJUPIN 3.0.8**, SIL Runner Gadget can be customized in a much user friendly way, asking parameters more nicely.

In order to configure such an entry, you will need to set up the following components:

- **Name** - the name of the configuration
- **Description** - the description of the configuration
- **Execution script** - the script that will be executed
- **Parameter script** - an optional script that will dynamically insert advanced parameter fields on the configuration's run screen.

If no parameter script is given, the user will be able to add simple input fields for text parameters (we maintained the old functionality).

Example

The execution script is the script that will be executed. If there are any parameters declared in the parameter script, their values will be received and interpreted here. In order to get the values of the parameters, you will need to use the parameter retrieval routines.

In our case, the execution script uses the runnerLog routine and can return as many values as you need, regardless of their type.

Example code

```
date start_date = gadget_getDateValue(argv, "Start Date");
string tanks = gadget_getSingleValue(argv, "Tanks");
string infantry = gadget_getSingleValue(argv, "Infantry");
string rockets = gadget_getMultiValues(argv, "Rockets");
runnerLog("Preparing to start a war...");
runnerLog("The war will start at this date: " + start_date);
runnerLog("Building tanks...");
runnerLog("Built " + tanks + " tanks.");
runnerLog("Gathering infantry...");
runnerLog("Gathered " + infantry + " brave men.");
runnerLog("Fueling rockets...");
runnerLog(rockets + " ready.");
runnerLog("Dispatching orders...");
return "Good job! The world is now at war!";
```

The parameter script contains the declaration of the parameters that will be used in the execution script. In order to declare the parameters you will need to use the input type routines.

Example code

```
gadget_createDatePicker("Start Date", currentDate(), true, "Choose a start
date");
gadget_createInput("Tanks", "500", true, "The number of tanks");
gadget_createInput("Infantry", "1600", true, "The number of tanks");
gadget_createCheckboxGroup("Rockets", {"A big one", "A lot of small ones"},
"", false, "Do you want to use rockets?");
```

The parameters can be set to a default values, which can be edited before running the execution script. Using the scripts above, the SIL Runner Gadget will look like this:

The execution of the script above produces the output below:

Live Fields

Live Fields

You want to restrict some users visibility on issue fields? Or you want to change issues fields values automatically when a field value changes? Or, you just want to execute your own javascript code? Now, you can do this with Live Fields.

See how Live Fields work.

How 'Live Fields' work

- Live Fields
- What's the idea?
- Screens where we can use Live Fields
- Example
    - Writing the code

Live Fields

You want to restrict some users visibility on issue fields? Or you want to change issues fields values automatically when a field value changes? Or, you just want to execute your own javascript code? Now, you can do this with Live Fields.

Live Fields is a JJupin extension that contains several routines for SIL used for example to hide, disable or set the values for JIRA fields. This actions happen automatically, while editing or viewing the issue.

> You can call also the Live Fields routines from postfunctions, and not only from the main configuration script or hooks!

What's the idea?

To understand how Live Fields work,we have to define some notions first:

- An **action** is the action executed on the screen. It can be *hide*, *show*, *disable*, etc; each action can be called from SIL using its corresponding routine (**IfHide**, **IfShow**, **IfDisable**, etc)
- A **hook script** (or, if you prefer, the **callback script**) represents a SIL script file that is executed when an event is triggered. You can create a hook using the **IfWatch** routine.
- The **main script** (or **configuration script**) is the initial script executed when the view issue / edit issue is called. This is actually your entry point in JJupin **Live Fields**.

Since we like UML, please take a look at the following sequence diagram:
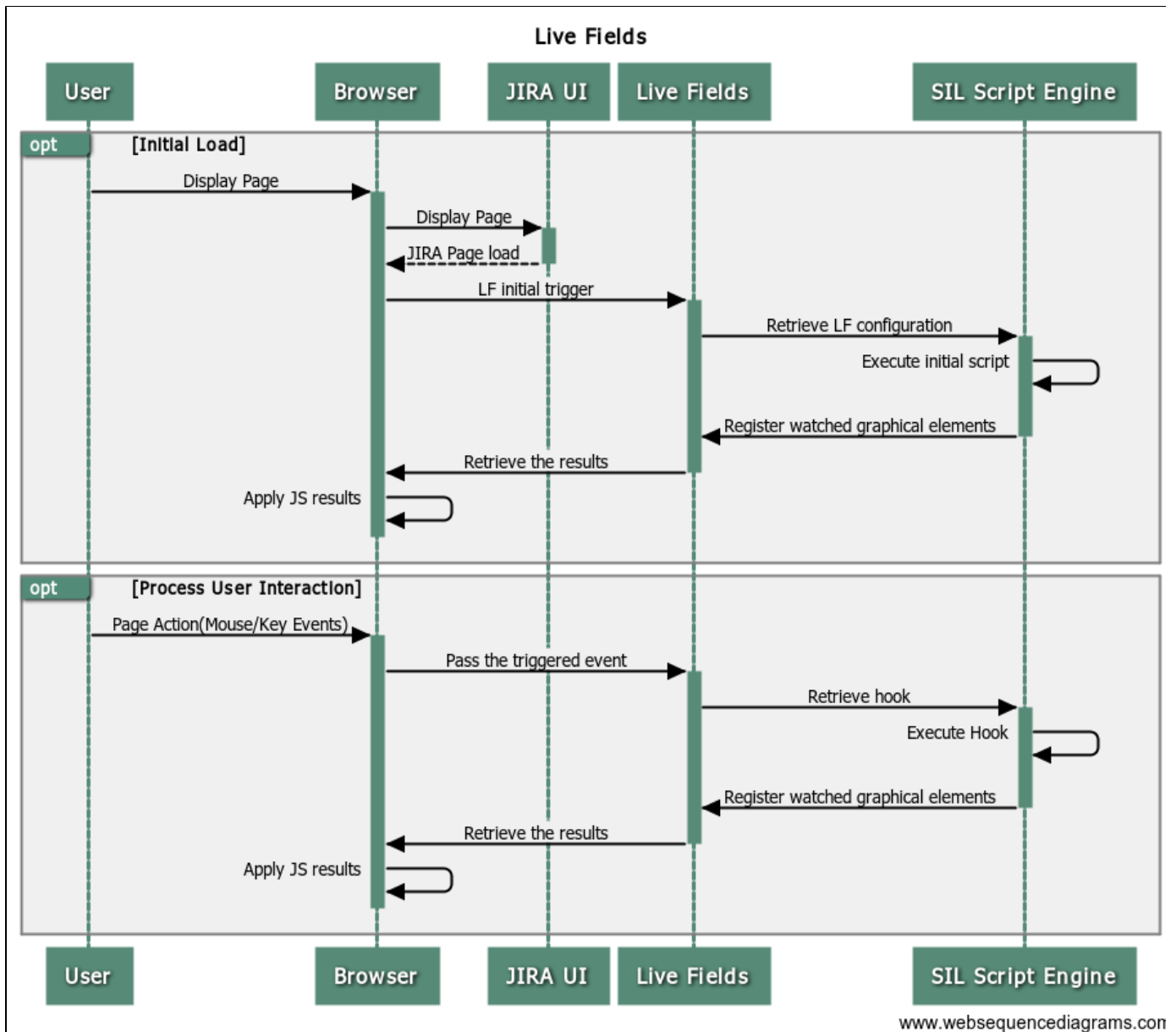
Live Fields

As you can see, the sequence of operations is actually very simple

1. When the JIRA issue page is loaded the Live Fields Configuration for the issue project is retrieved and the **Main Script** is executed.

   Remember: the **Main Script** represents the SIL Script file from the Live Fields Configuration of the issue project. The **Main Script** contains the actions and the hooks that will be executed every time an issue page is loaded. The main script can be associated with many projects, to ease the configuration for projects having similar screens. Not all pages in JIRA trigger Live Fields main script (see below).

2. After the **Main Script** is executed, the hooks will be registered and the results will be sent to the browser where the actions will apply.

   Remember: An **action** is represented by the live fields routines that changes the fields state, like **IfHide/IfShow, IfDisable/IfEnable.** A hook represents a SIL Script file that is executed when an event is triggered; hooks are created using **IfWatch** routine**.**

3. When an user interacts with an element (JIRA field) that has a hook registered for it, the event is triggered and the **Hook Script** is executed.

   Take care: The **Hook Script** can also contain actions and hooks. The difference is that the actions from the **Hook Script** are executed only in the current screen (the screen where the event was triggered).

Screens where we can use Live Fields

It is of paramount importance to understand that Live Fields can only be used in certain screens. You cannot use Live fields for the administration pages of JIRA, for instance (we could do it, but has really very little importance in our mind ..)

The following table summarizes the screens loading the Live Fields **main script** (configuration):

Screen category	Screen	Notes
Issue screens	View issue	Is the normal screen for viewing the issue. With the introduction of inline edit, please see the note below
	Edit issue	Here you should be able to implement on-screen logic, e.g. if the customer importance combo-box goes on important, increase priority.
	Create issue	Same comment apply here
	Transition screen	
Issue Navigator	Issue navigator	Issue navigator screens are supported as the above
	Issue navigator->Edit	
	Issue navigator->Transition	

As a general note, you should not worry if you're requesting an action for a field that is not on the screen. Live Fields is smart enough to skip over the non-existent fields.

If necessary, arbitrary javascript, residing in the **silprograms** folder on the server, can be executed on any above screen. However, try to minimize the amount of javascript, since it makes your JIRA install non-portable across versions of JIRA.

Example

Let's take the following example for you to understand better the Live Fields concept. Let's say you want to set the priority of the issue at Major when the summary contains the "important" word.

Writing the code

First of all, let's create the **Main Script**.

Go to **Administration** -> **Add-ons -> SIL Manager.**

Click the **silprograms** folder and the **New-> New file** button. Create a new SIL file and name it MainScript.sil like in the image below.
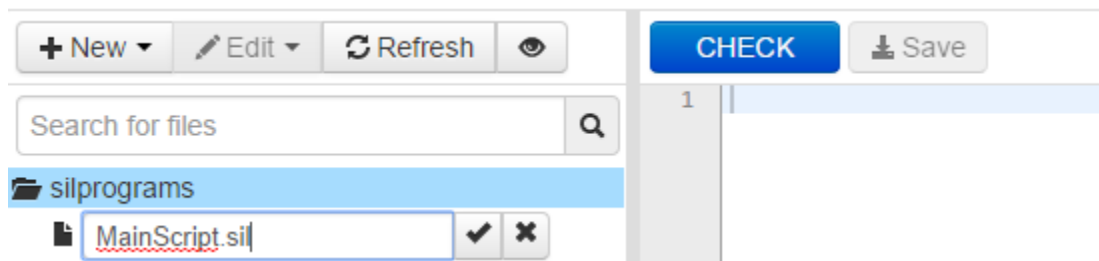
This will be the **Main Script** and we will configure it later.

In the MainScript.sil file that you create write the following code.

<div align="center">

MainScript.sil

</div>

```
lfWatch("summary", {"summary"}, "HookScript.sil" , {"keyup"});
```

When entering on the issue page, the *MainScript.sil* will run and attaches a listener for the **keyup** event, for the **summary** field. When the event is triggered the script from the *HookScript.sil* will run.

Next, let's create the *HookScript.sil* file in the way you created the *MainScript.sil*. For our example, you should write the following code in the *HookScript.sil* file:

<table>
<tr><th colspan="1" align="center">HookScript.sil</th></tr>
</table>

```
if (contains(argv["summary"], "important")) {
    lfSet("priority", "Major");
    lfShowFieldMessage("priority", "Priority changed", "INFO");
}
```

Info
See more information about managing your SIL Scripts.

Create a Live Field Configuration

So, we created the two scripts, but before testing it we have to create a Live Field Configuration and associate it to a project.

To do this you have to follow the next steps:

- Go to **Administration** -> **Add-ons -> Live Fields**
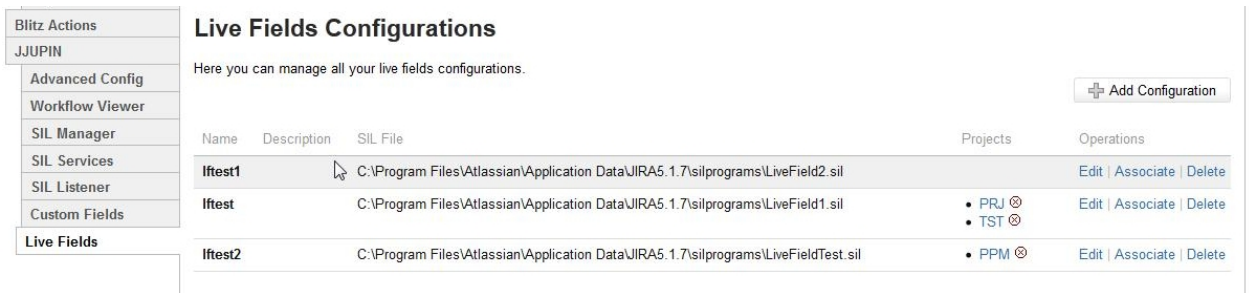
- Click the **Add Configuration** button

- In the displayed dialog box you have to enter the configuration name and description and you have to choose a SIL File for the Live Fields Configuration. As we said before, the **Main Script** represents the SIL Script file from the Live Fields Configuration, so let's choose it for our configuration:

Add Live Fields Configuration

Name * LiveFieldConfig

Provide a name for the configuration.

Description My first Live Field Configuration

SIL File C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\MainS

Choose a SIL file for the configuration from the file tree below.

- C:\Program Files\Atlassian
- silprograms
  - LiveField1.sil
  - LiveField2.sil
  - LiveFieldTest.sil
  - **MainScript.sil**
  - SILtesting.incl
  - ScriptHide.js

Add   Cancel

- Click the **Add** button and the Live Field Configuration will be created

LiveFieldConfig   My first Live Field Configuration   C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\MainScript.sil   Edit | Associate | Delete

- Now, you have to associate this configuration to a project. To do that, click the **Associate** link.
- From the displayed dialog you can choose the project(s) to associate the configuration with.

LiveFieldConfig   My first Live Field Configuration   C:\Program Files\Atlassian\Application Data\JIRA5.1.7\silprograms\MainScript.sil   • PPM ⊗   Edit | Associate | Delete

In short:

You created two scripts, the *MainScript.sil* and the *HookScript.sil*.

You created a **Live Fields** Configuration that contains the *MainScript.sil* and associated it with a project (with ProjPM, in our case).

Result:

Every time we enter on an issue page of ProjPM project, the *MainScript.sil* is executed and the hook is registered. When we edit the issue summary, the keyup event is triggered and the *HookScript.sil* is executed.
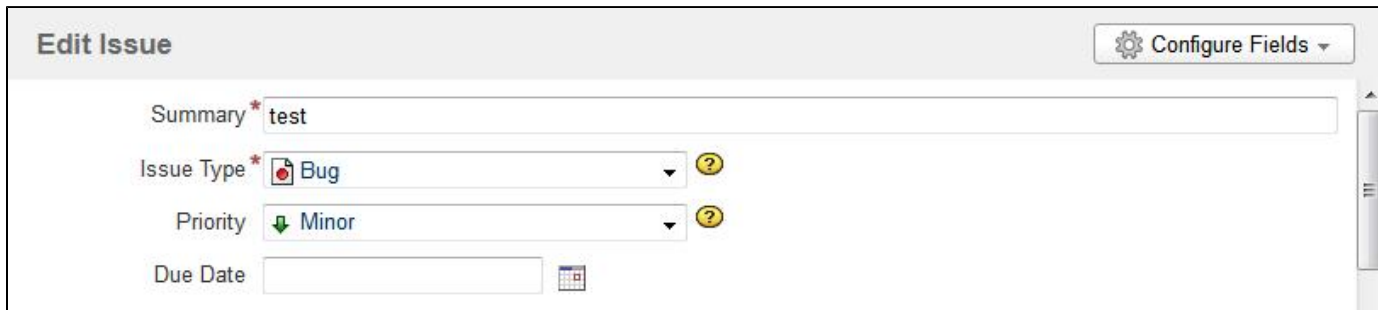
> **Info**
> You can find here more information about **Live Fields Configuration**.

Let's test it !
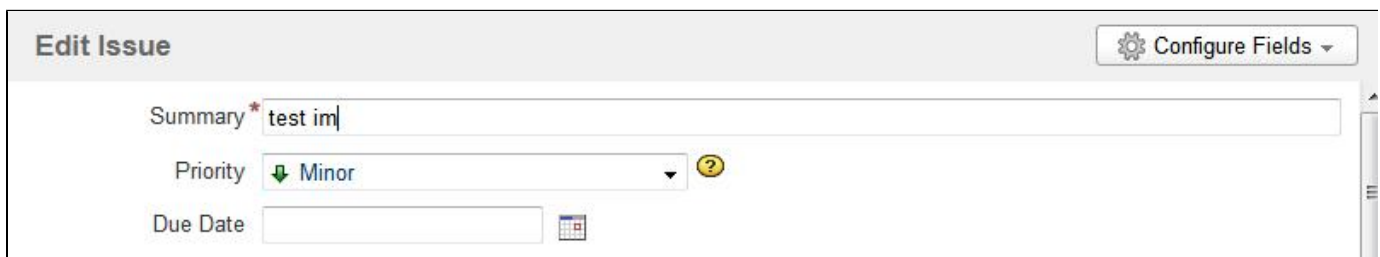
On the edit screen of the issue you start typing the summary for the issue. Every time the event **keyup** is triggered a call to the server is made and the hook.sil is executed. The server will receive the values of the related fields (the second parameter from the IfWatch routine), in our case the value of the summary field.

For example, we have the following issue:
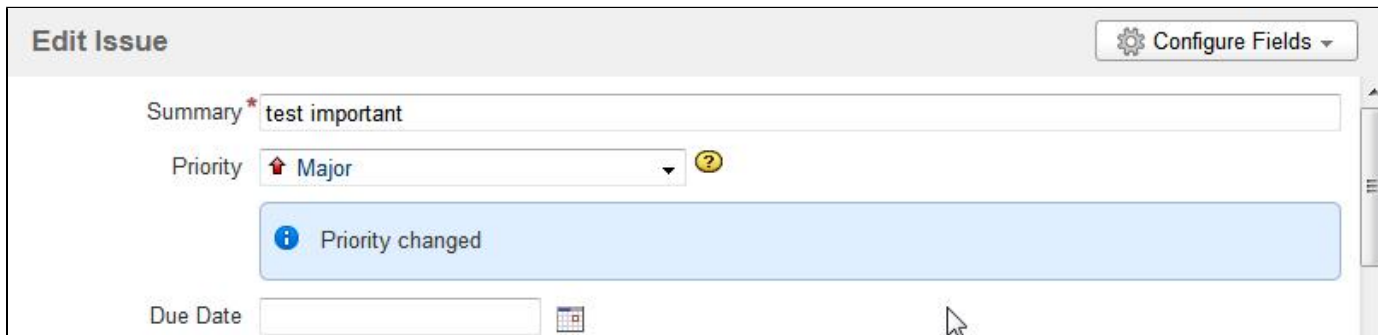
We start editing the summary field.

We typed " im", so the event was triggered three times. That means the hook.sil was executed three times, and it received the following values for summary field: "test ", "test i", "test im". This are the values passed from argv["summary"].

When the **summary** field will contain the **"important"** word, the priority will be set as Major and a message will be displayed.

> **Info**
> You can find the Live Fields routines here.

> **Info**
> More information about SIL programs in Simple Issue Language documentation.

Supported fields and graphic elements

On this page:

- Supported JIRA fields

Supported JIRA fields

Here is a list of JIRA fields supported by Live Fields:

Field	Explanation	Usage
Project	The project name for the issue. Available since v. 2.6.1 (for JIRA 6.x).	project
Summary	The summary field of the issue.	summary
Type	The issue type field.	issueType
Priority	The priority field of the issue.	priority
Status	The status field of the issue.	status
Resolution	The resolution field of the issue.	resolution
Affects Version/s	The affected versions field of the issue.	affectedVersions
Fix Version/s	The fix versions field of the issue.	fixVersions
Security Level	The security level field of the issue.	security
Component/s	The component field of the issue.	components
Labels	The labels field of the issue.	labels
Environment	The environment field of the issue.	environment
Description	The description field of the issue.	description
Assignee	The assignee field of the issue.	assignee
Reporter	The reporter field of the issue.	reporter
Due	The due date field of the issue.	dueDate
Created	The created field of the issue.	created
Updated	The updated field of the issue.	updated
Resolved	The resolved field of the issue.	resolved
Estimated	The issue original estimate.	originalEstimate
Remaining	The issue remaining estimate.	estimate
Logged	The issue time spent.	timeSpent
Votes	The vote field of the issue.	votes
Watchers	The watchers field of the issue.	watchers
Edit Submit	Submit button from Edit screen.	editSubmit
Transition Submit	Submit buttons on Transition screens.	transitionSubmit
Cancel	Cancel link from Edit screen, Transition screen, Create screen.	cancel
Create Issue Submit	Submit button from Create Issue screen.(the pop-up screen) Available since v. 3.0.3 (for JIRA 6.x).	createIssueSubmit

Issue Create Submit	Submit button from Create Issue screen.(with the Transition screen) Available since v. 3.0.3 (for JIRA 6.x).	issueCreateSubmit
Attach Files	The Attach Files drop-down item from button More from the view screen of an issue. Also, the drag and drop from all screens of an issue. Available since v. 3.0.5 (for JIRA 6.x).	attachFiles
Attach Screenshot	The Attach Screenshot drop-down item from button More from the view screen of an issue. Available since v. 3.0.5 (for JIRA 6.x).	attachScreenshot
Delete attachments	The Delete icon from the View screen of an issue and from the Manage Attachments screen. Available since v. 3.0.5 (for JIRA 6.x).	deleteAttachment
Attachments	The Attachments module from the view screen of an issue. Available since v. 3.0.5 (for JIRA 6.x).	attachments
Add Attachments	The Add Attachments icon from the view screen of an issue. Available since v. 3.0.5 (for JIRA 6.x).	addAttachments
Manage Attachments	The Manage Attachments drop-down item from Attachments module. Available since v. 3.0.5 (for JIRA 6.x).	manageAttachments
Viewable by	The Viewable By option of the Comment field from View, Edit, Transition screens. Available since v. 3.0.8 (for JIRA 6.x).	viewable_by

> **Important!**
> You have to use them in Live Fields Routines with the key indicated in the **Usage** column of the above table.

> **Important!**
> **Create Issue Submit** and **Issue Create Submit** - both of them are used for the Create button from the create screen.
>
> If the button is accessed from an issue then it will appear the Create Issue pop-up and you have to use **Create Issue Submit**.
>
> If the button is accessed for example from the Manage Add-ons screen, first it will appear a transition screen - you have to access the Next button if you want to create a new issue. For this situation you have to use **Issue Create Submit**.
>
> For **Create Issue Submit** and **Issue Create Submit** you have to use both of them if you want to control all the Create Issue screens using live fields.

> **Tip**
> Some Live Fields routines can interact with other elements of the issue as well. Check out each routine's page to see any additional elements it can interact with.

Examples

An example for it would be to hide the **Estimated** field. You do that, using the lfHide function like this:

```
lfHide("originalEstimate"); //this will hide the estimated field
```

The image shows time tracking before hidding the estimated field(on the left side) and after hidding it(on the right side).
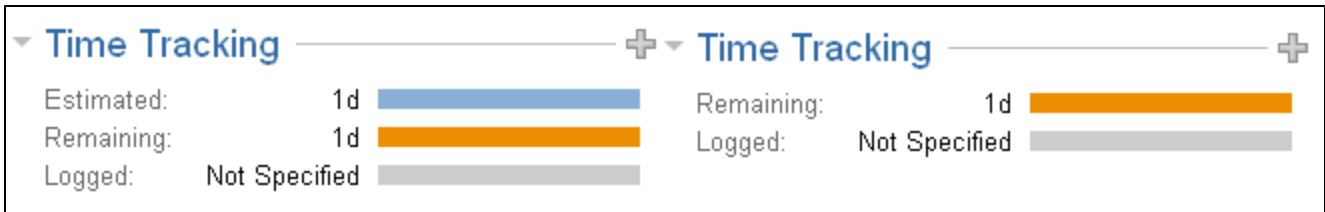
If you want to set the **Type** of the issue, use the lfSet function like this:

```
lfSet("issueType", "Task"); //this will set the issue type with the Task
value
```

Supported JIRA custom field types

Live Fields also supports the following custom fields:

- Number Field
- Text Field
- Free Text Field
- URL Field
- Labels
- Single Version Picker
- Version Picker
- Cascading Select
- Radio Buttons
- Date Picker
- Date Time
- User Picker
- Multi User Picker
- Group Picker
- Multi Group Picker
- Multi Checkboxes
- Multi Select
- Select List
- Project Picker

> **Warning**
> When using custom field name make sure you don't have more than one custom field with the same name. The action will apply only on the first created custom field.

Example

Let's say you have a **Number Field** custom field named **count** . The custom field id is 10100. You can hide it like:

```
lfHide("customfield_10100") //hide the custom field by its id
lfHide("count") //hide the custom field by its name
```

> **Info**
> You can also use aliases to apply actions to custom fields. More about custom fields aliases see here.

Supported JIRA Software custom field types

Since JJUPIN 4.0.0, we also support the next JIRA Software custom fields:

- Epic Name
- Epic Colour
- Epic Label
- Epic Link

- Epic Status
- Rank
- Sprint
- Story Points
- Flagged
- Business Value

Example

Let's analyze the "Epic Link" custom field. The custom field id is 10001. You can hide it as follows:

```
lfHide("customfield_10001") //hide the custom field by its id
lfHide("Epic Link") //hide the custom field by its name
```

Events

Watching events seems simple (check lfWatch routine). But what exactly are the events ? The answer to that is actually very simple: all of them are **JavaScript events**. We decided to use them directly because

1. People already know these events
2. We wanted to offer you a broad range of events to watch on
3. People may need to add additional JS in the page. Mixing them would mean that the programmer would have to mentally map events from JJupin to JS and the other way around. Yak!

Now let's see what we can do with these fields and events here.

Accessing the current screen

> **Availability**
> This feature is available since
>
> 1. **JJUPIN 2.5.3**
> 2. **katl-commons 2.5.5**

Starting from version 2.5.3 of JJupin, the "screen" argument was passed to the Live Fields scripts, so you can easily filter your actions based on which issue screen you are operating.

Syntax

argv["screen"]

Description

Returns the actual screen on which the current Live Fields SIL script is executed (for the initial script, as well as for any hooked script).

Returns

A string from the following list of predefined values, corresponding to the actual issue screen:

Screen	Argument Value
View Issue	view
Edit Issue	edit
Create Issue	create

Create Sub-Task	create-subtask
Transition Screen	trans_<transition_id>

> Support for the "Create Sub-Task" screen is available since **JJUPIN 2.5.5** and **katl-commons 2.5.8**.
>
> You can easily determine a particular transition's id, by checking your workflow administration page. They are listed as: Transition (id).

Here are the Jira's transitions and their correspondent ids:

Workflows
DEMO: Simple Issue Tracking Workflow ACTIVE SHARED BY 1 PROJECT   Edit ⊙

ⓘ This workflow was last edited by **you** at 05/Mar/15 2:01 PM.

Diagram   Text   Export ▾

Step Name (id)	Linked Status	Transitions (id)	Operations
To Do (1)	TO DO	Start Progress (11) >> IN PROGRESS Done (21) >> DONE	View Properties
In Progress (2)	IN PROGRESS	Stop Progress (31) >> TO DO Done (41) >> DONE	View Properties
Done (3)	DONE	Reopen (51) >> TO DO Reopen and start progress (61) >> IN PROGRESS	View Properties

Example

This can be useful when you want to apply certain Live Fields actions only on editable screens for example, and not on view issue page.

Let's say that you want the assignee to always be set to user "x" when creating a new issue with a custom field *Defect* having the value *Developm ent*, without letting the user modify it.

At the same time, you want to set by default the current user as assignee whenever it accesses the "Resolve Issue" transition screen, but keep it editable.

This can easily be achieved by checking the "screen" argument and applying the live fields actions on field assignee, based on the argument's value, in the initial script, as well as in the hook script:

<div style="text-align:center">

init.sil

</div>

```
if(argv["screen"] == "trans_5") {
    //on Resolve Issue screen
    lfSet("assignee", currentUser());
}
//set the hook script for the Defect custom field
lfWatch("Defect", "Defect", "hook.sil");
```

<div style="border: 1px solid #ccc">

hook.sil

```
if(argv["screen"] == "create") {
    //on Create Issue screen
    if(argv["Defect"] == "Development") {
        lfSet("assignee", "x");
        lfDisable("assignee");
    } else {
        lfEnable("assignee");
    }
} else {
    //other functionality based on Defect field value for other screens
here....
}
```

</div>

Routines

Standard routines

The standard routines are listed in our [SIL space](#). These routines are available to all our SIL-enabled plugins, namely:

- **JJUPIN** (this plugin)
- **JJUPIN Agile** - with the power of SIL and JJupin for the Agile ninjas
- **Blitz Actions** - creates a non-transition screen. The companion of JJupin
- **KCF - Kepler Custom Fields** - varia CF for your fun time (free)
- **DBCF - Database Custom Field** - The only free plugin getting data from databases
- **Kontinuum** - Our time-tracking solution

> **For the technical minded**
> There is just one routine registry, and that belongs to the katl-commons plugin. This makes sharing of the routines possible among plugins !

Routines added by JJupin

The following routines are JJupin specific.

Routine	Description	Syntax
lfHide	Hides a field.	lfHide(field)
lfShow	Shows a field.	lfShow(field)
lfDisable	Disables a field.	lfDisable(field)
lfEnable	Enables a field.	lfEnable(field)
lfHideAllExcept	Hides all the given fields, panels and tabs except the ones given as parameters.	lfHideAllExcept(fields_tabs_and_panels)
lfShowAll	Shows the given fields, panels and tabs.	lfShowAll(fields_tabs_and_panels)
lfShowFieldMessage	Displays a message for the given field.	lfShowFieldMessage(field, message, messageClass)
lfHideFieldMessage	Hides a message for the given field.	lfHideFieldMessage(field)

lfGlobalMessage	Displays a global message.	lfGlobalMessage(message, messageClass)
lfDialogMessage	Displays a global message in a dialog box.	lfDialogMessage(message, messageClass)
lfSet	Sets a field with the given values.	lfSet(field, value)
lfWatch	Attach listeners for the given events.	lfWatch(field, relatedFields, scriptPath[,javaScriptEvents])
lfExecuteJS	Gives you the possibility to run your own javascript code.	lfExecuteJS(jsFilePath)
lfRestrictSelectOptions	Restricts the list of given options from the options of the field.	lfRestrictSelectOptions(field, options)
lfRefreshScreen	Performs a page reload.	lfRefreshScreen()
lfRedirect	Redirects to a given URL.	lfRedirect(url)
lfInstantHook	Executes the given SIL script, passing the screen values for relatedFields as parameters to the script.	lfInstantHook(relatedFields, scriptPath)

lfAllowSelectOptions

> **Availability**
> This routine is available since
>
> 1. **JJUPIN 3.0.2**

Syntax

lfAllowSelectOptions(field, options[, triggerChange])

Description

Restricts the list of given options of the field to the list of options given as parameter.

Parameters

Parameter	Type	Required	Description
field	String	Yes	The field to restrict options for.
options	String	Yes	The list of remaining options.
triggerChange	Boolean	No	If set to true, triggers the change event on the field when routine is used.

Example

The following code example restricts anything but Major and Minor from the options of the priority standard field.

```
lfAllowSelectOptions("priority", {"Major", "Minor"}); //where field =
"priority" and options = "Major" and "Minor"
```

If you want to trigger the change event on the field when using lfAllowSelectOptions, you can use the optional triggerChange parameter set to true:

```
lfAllowSelectOptions("customfield_10000", {"option1", "option2"}, true);
//where field = "customfield_10000" of type select list, options =
"option1" and "option2" and triggerChange = true
```

lfDialogMessage

> **Availability**
> This routine is available since
>
> 1. **JJUPIN 2.5**
> 2. **katl-commons 2.5**

Syntax

lfDialogMessage(message, messageClass);

Description

Displays a global message in a dialog box.

Parameters

Parameter	Type	Required	Description
message	String	Yes	The message to display.
messageClass	String	Yes	The message type.

The **messageClass** parameter can be:

- **ERROR**: will display an error message.
- **WARNING**: will display a warning message.
- **SUCCESS**: will display a success message.
- **INFO**: will display an info message.
- **HINT**: will display a hint message.

Example

```
lfDialogMessage("This is a dialog message!", "WARNING");// where message =
"This is a dialog message!" and messageClass = "WARNING"
```

The message will be displayed like in the image below.

lfDisable

Syntax

lfDisable(field)

Description

Disables the given field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the field to disable.

Example

Let's assume that the field issueType once set should not be changed by anyone, but by the admin user. To prevent other users from changing it, being only able to view it, use lfDisable.

```
if(assignee != "admin") {
    lfDisable("issueType");
  }
```

Here's how the issue type looks like on the issue screen after calling lfDisable.

Issue Type* [Task] ?
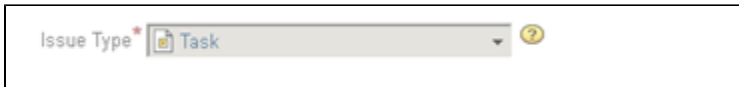
Known Issues
When updating an issue the values for some of the disabled fields will not be saved. For example, you will not be able to update an issue that has the **summary** field disabled.

If you update an issue that has a **Text Field** disabled, the custom field will not be anymore visible on the issue page because it was saved with an empty value. This applies to most custom fields.

If a field is disabled and you want to enable it, use the next routine: lfEnable.

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfDisableTab

Syntax

lfDisableTab(field)

Description

Disables the given tab.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the tab to disable.

Example

If the assignee is not admin, disable the Field Tab from the issue.

```
if(assignee != "admin") {
     lfDisableTab("Field Tab");
   }
```

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfEnable

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

lfEnable(field)

Description

Enables the given field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the field to enable.

Example

If the field is disabled for all the users and the user admin, for example, should change the value, use lfEnable.

```
if(assignee == "admin") {
    lfEnable("issueType");
   }
```

The image shows the enabled field and the value of the assignee field.

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

IfEnableTab

Availability
This routine is available since

1. **JJUPIN 2.5.12+ / 2.6.7+**
2. **katl-commons 2.5.16+ / 2.6.8+**

Syntax

IfEnableTab(field)

Description

Enables the given tab.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the tab to enable.

Example

If the assignee is admin, enable the Field Tab from the issue.

```
if(assignee == "admin") {
      lfEnableTab("Field Tab");
   }
```

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

IfExecuteJS

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

IfExecuteJS(jsFilePath);

Description

Gives you the possibility to run your own javascript code.

Parameters

Parameter	Type	Required	Description

jsFilePath	String	Yes	The script source to run that contains your javascript code. The file is resolved relative to **silprograms** path.

Example

Let's first create a file which contains the following javascript code:

```
AJS.$('#summary-val').get(0).childNodes[0].nodeValue = "Executing my
javascript";
AJS.$('#descriptionmodule').hide();
```

Save it on the disk as hook.js and call the lfExecuteJS routine like in the code block below:

```
lfExecuteJS("hook.js"); // jsFilePath = "hook.js"
```

For the jsFilePath parameter you can either give the relative path (as in the example above) or the absolute path.

When calling this routine, the javascript code from hook.js is executed.

This will set the summary value on the issue page and will hide the description.

Important
The file designated by the jsFilePath patameter must contain only JavaScript code. Note that this code will be inlined, so **DO NOT USE SINGLE LINE COMMENTS**!

```
var v = "a";
// let's show an alert
alert(v);
```

The above script will be evaluated to

```
var v = "a"; // let's show an alert alert(v);
```

So the alert() will never be called.

For the technical minded
The above routine gives you virtually all the power on JIRA UI. However, this may **NOT BE PORTABLE** across versions of JIRA.

lfGlobalMessage

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

lfGlobalMessage(message, messageClass);

Description

Displays a global message.

Parameters

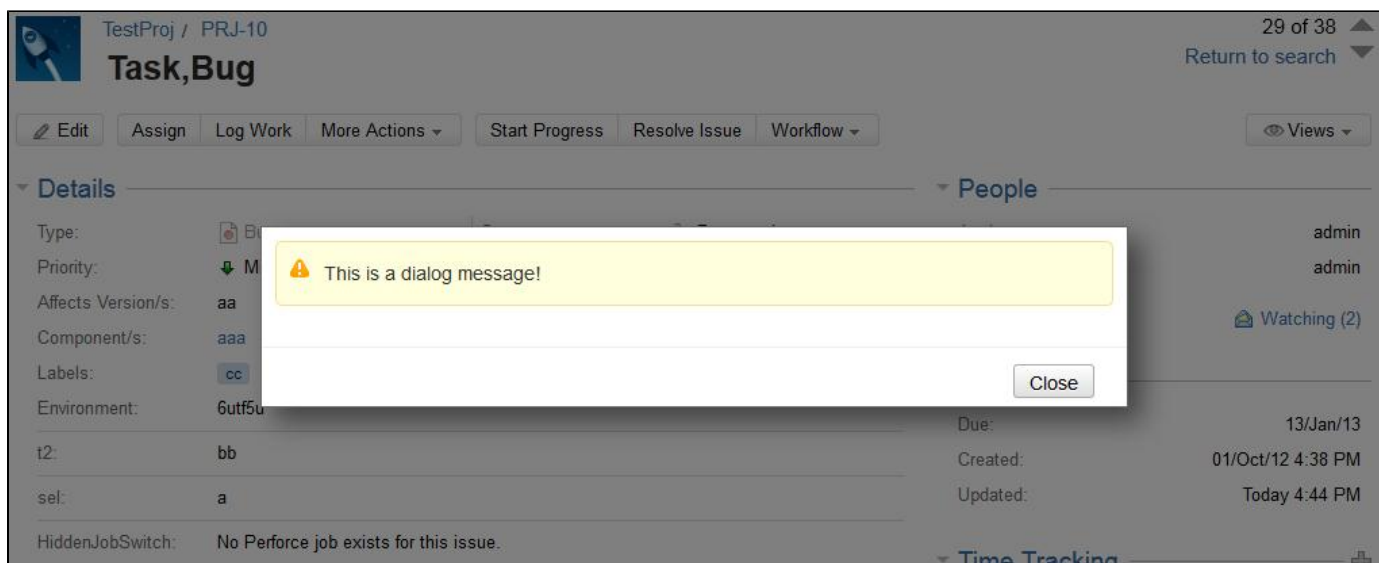| Parameter	Type	Required	Description
message	String	Yes	The message to display.
messageClass	String	Yes	The message type.
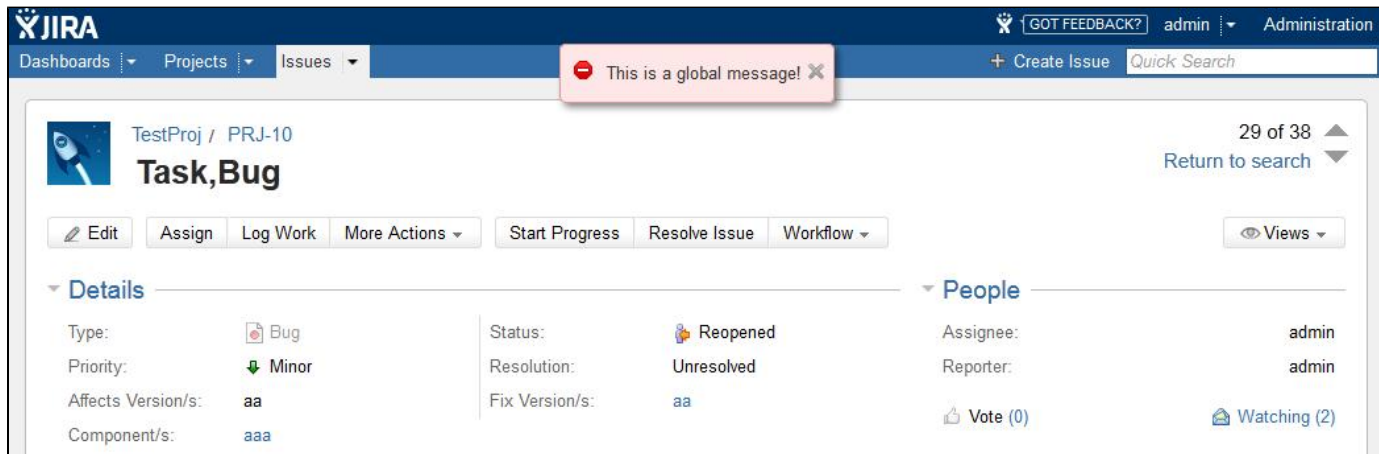
The **messageClass** parameter can be:

- **ERROR**: will display an error message.
- **WARNING**: will display a warning message.
- **SUCCESS**: will display a success message.
- **INFO**: will display an info message.
- **HINT**: will display a hint message.

Example

```
lfGlobalMessage("This is a global message!", "ERROR"); // where message =
"This is a global message!" and  messageClass = "ERROR"
```

The message will be displayed on the issue screen like in the image below:

lfHide

> **Availability**
> This routine is available since
>
> 1. **JJUPIN 2.5**
> 2. **katl-commons 2.5**

Syntax

lfHide(field)

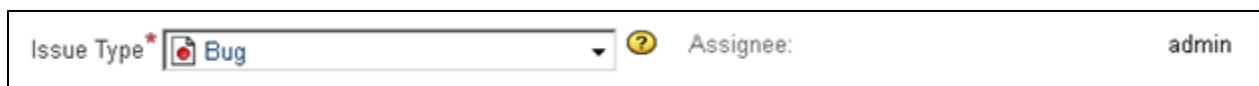### Description

Hides the given field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the field to hide.

Example

If the assignee is not admin, hide the issue type field from the issue.

```
if(assignee != "admin") {
      lfHide("issueType");
   }
```

The image shows on the left side that the issue type is hidden and on the right side that the assignee is set to "test 1".
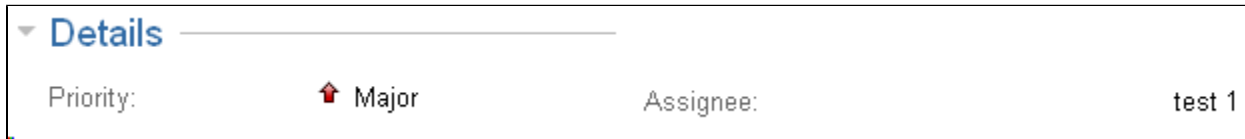
Now that the field is hidden, you can use IfShow to display it on the issue screen.

Hiding fields and security
Hiding fields on the screen is not secure ! This is not a security solution, the field is present in HTML and can still be inspected via a simple "Show page source".

This feature is only used to put some logic in the screen !

Additional Fields

Availability
Feature available since JJupin 2.5.2.

In addition to the Supported fields and graphic elements accepted by all Live Fields routines, IfHide can also handle:

Element	Field (to be used in routine)
Details Panel	details_panel
People Panel	people_panel
Dates Panel	dates_panel
Timetracking Panel	timetracking_panel
Activity Panel	activity_panel
Comments Tab	comments_tab
History Tab	history_tab
Worklog Tab	worklog_tab
Activity Tab	activity_tab
All Tab	all_tab
Add Comment	addComment

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfHideAllExcept

Syntax

lfHideAllExcept(fields_tabs_and_panels)

Description

Hides all the given fields, panels and tabs except the ones given as parameters.

Parameters

Parameter	Type	Required	Description
fields_tabs_and_panels	String	Yes	Specifies the fields/panels/tabs to hide.

Example

Please note that this routine hides **all** the elements that are not specified as parameters; so, if you want to show a field, don't forget to add the tab or panel it belongs to as a parameter.

```
lfHideAllExcept("details_panel", "issueType", "priority", "activity_panel",
"comments_tab",  "Field Tab", "customfield_10101");
```

This is equivalent with:

```
lfShow("details_panel");
lfShow("issueType");
lfShow("activity_panel");
lfShow("comments_tab");
lfShowTab("Field Tab");
lfShow("customfield_10101");
//for all the other fields, tabs and panels: lfHide(element);
```

Additional Fields

In addition to the Supported fields and graphic elements accepted by all Live Fields routines, lfHideAllExcept can also handle:

Element	Field (to be used in routine)
Details Panel	details_panel
People Panel	people_panel
Dates Panel	dates_panel

Timetracking Panel	timetracking_panel
Activity Panel	activity_panel
Comments Tab	comments_tab
History Tab	history_tab
Worklog Tab	worklog_tab
Activity Tab	activity_tab
All Tab	all_tab

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfHideFieldMessage

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

lfHideFieldMessage(field)

Description

Hides a message for the given field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	The field to hide the message for.

Example

```
lfHideFieldMessage("assignee");//where field = "assignee"
```

lfHideTab

Availability
This routine is available since

1. **JJUPIN 2.5.12+ / 2.6.7+**
2. **katl-commons 2.5.16+ / 2.6.8+**

Syntax

lfHideTab(field)

Description

Hides the given tab.

This routine only handles field tab and tabs defined by the user. If you want to hide the tabs from Activity panel, see lfHide routine.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the tab to hide.

Example

If the assignee is not admin, hide the Field Tab from the issue.

```
if(assignee != "admin") {
    lfHideTab("Field Tab");
}
```

Now that the field is hidden, you can use IfShowTab to display it on the issue screen.

Hiding fields and security
Hiding fields on the screen is not secure ! This is not a security solution, the field is present in HTML and can still be inspected via a simple "Show page source".

This feature is only used to put some logic in the screen !

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

IfInstantHook

Availability
This routine is available since

1. **JJUPIN 2.5.6 (for JIRA 5.x) and 2.6.1 (for JIRA 6.x)**
2. **katl-commons 2.5.8 (for JIRA 5.x) and 2.6.1 (for JIRA 6.x)**

Syntax

IfInstantHook(relatedFields, scriptPath);

Description

Executes the given SIL script, passing the screen values for the specified relatedFields as parameters to the script.

This is especially useful in the create issue screen, where you don't have access to the issue standard variables.

Using an instant hook, you can access in the hook script the screen values for the desired fields as **argv[field]**.

Parameters

Parameter	Type	Required	Description
relatedFields	String Array	Yes	The dependent fields required for the given field.
scriptPath	String	Yes	The script source to run when the event is triggered.
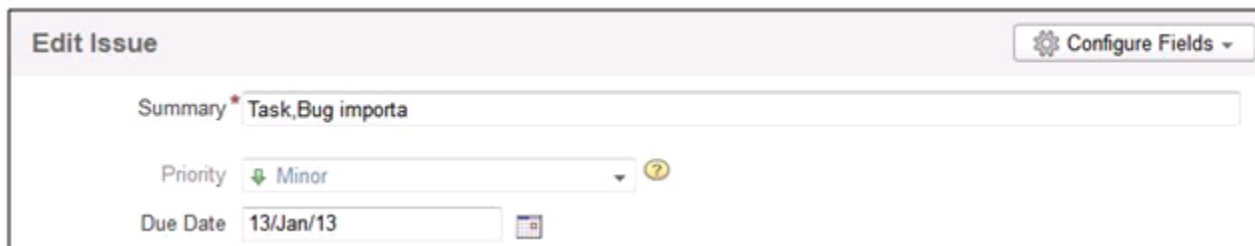
Example

```
lfInstantHook({"summary", "customfield_13706", "components"}, " hook.sil");
```

For the scriptPath parameter you can either give the relative path (as in the example above), or the absolute path as: "C:/Program Files/Atlassian/Application Data/JIRA/silprograms/hook.sil".

```
// hook.sil :
 if (contains(argv["summary"], "important")) {
 lfSet("priority", "Critical");
 lfShowFieldMessage("priority", "Priority changed", "INFO");
 }
```
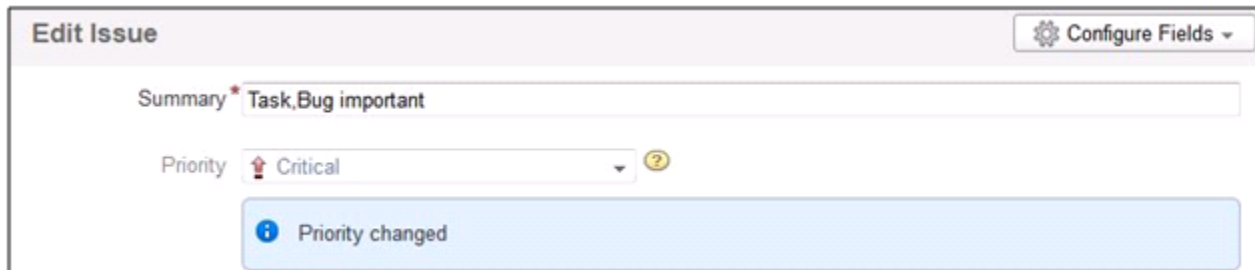
Every time when the initial script is triggered, the hook.sil is executed. When the summary field contains the word "important", priority field is set to Critical and a message will be displayed for the priority field.

The first image shows the initial value of the priority for the current issue, the next one shows the value it is changed to, after executing the code from hook.sil.

As we said before, the values from the relatedFields are accessed as **argv[field]**. For multiple values fields like **components** or **affectedVersions** the value returned is in the following format: **val1|val2|val3**.

> **Info**
> For more information, see How 'Live Fields' work.

lfRedirect

> **Availability**
> This routine is available since
>
> 1. **JJUPIN 2.5.6 (for JIRA 5.x) and 2.6.1 (for JIRA 6.x)**
> 2. **katl-commons 2.5.8 (for JIRA 5.x) and 2.6.1 (for JIRA 6.x)**

Syntax

lfRedirect(url);

Description

Redirects to the specified URL.

If the url parameter represents a project or issue key, will redirect to its page, that is "<jira_base_url>/browse/<issue_or_project_key>".

Parameters

Parameter	Type	Required	Description
url	String	Yes	The redirect URL.

The **url** parameter can be**:**

- a Jira **relative path** (eg. "/secure/Dashboard.jspa")
- a **issue key** (eg. "DEMO-1")
- a **project key** (eg. "DEMO")
- a **full path URL** (eg. "http://www.google.com")

Example

Redirecting to our Kepler's products site:

```
lfRedirect("http://jira-plugins.kepler-rominfo.com");
```

Redirecting to Jira dashboard:

```
lfRedirect("/secure/Dashboard.jspa");
```

Redirecting to project "DEMO" page:

```
lfRedirect("DEMO");
```

Redirecting to issue "DEMO-1" page:

```
lfRedirect("DEMO-1");
```

lfRefreshScreen

> **Availability**
> This routine is available since
>
> 1. **JJUPIN 2.5.5**
> 2. **katl-commons 2.5.8**

Syntax

lfRefreshScreen();

Description

Performs a page reload.

Example

This routine cane be used for example to refresh information on view issue after performing an auto-transition when issue is viewed for the first time:

```
if(argv["screen"] == "view" && status == "New") {
    autotransition("Move to Open", key);
    lfRefreshScreen();
}
```

Issue is created in status New. When first accessed it is auto-transitioned to status Open and page is refreshed by means of the lfRefreshScreen routine to reflect the updated info.

lfRestrictSelectOptions

Availability
This routine is available since

1. **JJUPIN 2.5.2**
2. **katl-commons 2.5.3**

Syntax

lfRestrictSelectOptions(field, options, [triggerChange]);

Description

Restricts the list of given options from the options of the field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	The field to restrict options for.
options	String	Yes	The list of options to restrict.
triggerChange	Boolean	No	If set to true, triggers the change event on the field when routine is used. Available since v. 2.5.6 for Jira 5.x and v. 2.6.1 for Jira 6.x.

Example

The following code example restricts Major and Minor from the options of the priority standard field.

```
lfRestrictSelectOptions("priority", {"Major", "Minor"}); //where field =
"priority" and options = "Major" and "Minor"
```

If you want to trigger the change event on the field when using lfRestrictSelectOptions, you can use the optional triggerChange parameter set to true:

```
lfRestrictSelectOptions("customfield_10000", {"option1", "option2"}, true);

//where field = "customfield_10000" of type select list, options =
"option1" and "option2" and triggerChange = true
```

lfSet

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

lfSet(field, value, [triggerChange]);

Description

Sets the field with the given values.

This sets the value **in the screen only**. It does not set the value on the issue (setting it on the issue require direct access to the field)

Parameters

Parameter	Type	Required	Description
field	String	Yes	The field to set the value for.
value	String	Yes	The value to set. It can be a string value or an array with string values.
triggerChange	Boolean	No	If set to true, triggers the change event when lfSet is used on a field. Available since v. 2.5.6 for Jira 5.x and v. 2.6.1 for Jira 6.x.

Examples

The following code example sets the priority standard field as Major.

```
lfSet("priority", "Major"); // where field = "priority" and value = "Major"
```

Warning
The value will not be saved in the database. To save value in the database you should do something like:

priority = "Major"; // this saves into the database the value

However, please make sure you're not on the create screen!

As we said before, you can set multiple values to a field that can have multiple values. For example, let's set components field to comp1, comp2.

```
lfSet("components" , {"comp1", "comp2"});
```

Warning
You can't set a field if the values are not available for the given field. For example, in order to set components field to comp1, comp2, you have to make sure that comp1 and comp2 are valid components for that issue.

If you try to set, for example, issue type field using an array like the code below, lfSet will take into account only the first value from the array. So, this will set the issue type to "Task".

```
lfSet("issueType", {"Task", "Bug"});
```

If you want to trigger the change event on the field when using lfSet, you can use the optional triggerChange parameter set to true:

```
lfSet("customfield_10000", "updated val", true);
```

Known Issues
There are some fields, from the list provided in Supported fields and graphic elements, that couldn't set the value for. These fields are:

- Labels, on Edit, Transition, Create screens;

- Estimate (remaining estimate), on Transition screens;
- Votes
- Watchers

On the view screen, when you want to edit a field will be displayed the last value saved for that field.

You can't set fields that are not editable. For example, on the issue view screen you **can't set** status or resolution fields.

lfShow

> **Availability**
> This routine is available since
>
> 1. **JJUPIN 2.5**
> 2. **katl-commons 2.5**

Syntax

lfShow(field)

Description

Shows the given field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the field to show.

Example

If the assignee is admin, show the issue type field on the issue.

```
if(assignee == "admin") {
    lfShow("issueType");
}
```

The image shows on the left side that the issue type is displayed on the issue screen and that the assignee is set to "admin".

Additional Fields

> **Availability**
> Feature available since JJupin 2.5.2.

In addition to the Supported fields and graphic elements accepted by all Live Fields routines, lfShow can also handle:

Element	Field (to be used in routine)
Details Panel	details_panel
People Panel	people_panel
Dates Panel	dates_panel

Timetracking Panel	timetracking_panel
Activity Panel	activity_panel
Comments Tab	comments_tab
History Tab	history_tab
Worklog Tab	worklog_tab
All Tab	all_tab
Add Comment	addComment

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfShowAll

Availability
This routine is available since

1. **JJUPIN 3.0.7**
2. **katl-commons 3.0.7**

Syntax

lfShowAll(fields_tabs_and_panels)

Description

Shows the given fields, panels and tabs.

Parameters

Parameter	Type	Required	Description
fields_tabs_and_panels	String	Yes	Specifies the fields/panels/tabs to show.

Example

```
lfShowAll("issueType", "comments_tab", "activity_panel", "Field Tab");
```

This is equivalent with:

```
lfShow("issueType");
lfShow("comments_tab");
lfShow("activity_panel");
lfShowTab("Field Tab");
```

Additional Fields

Availability
Feature available since JJupin 3.0.7.

In addition to the Supported fields and graphic elements accepted by all Live Fields routines, lfShowAll can also handle:

Element	Field (to be used in routine)
Details Panel	details_panel
People Panel	people_panel
Dates Panel	dates_panel
Timetracking Panel	timetracking_panel
Activity Panel	activity_panel
Comments Tab	comments_tab
History Tab	history_tab
Worklog Tab	worklog_tab
All Tab	all_tab

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfShowFieldMessage

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

lfShowFieldMessage(field, message, messageClass)

Description

Displays a message for the given field.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the field for displaying the message.
message	String	Yes	The message.
messageClass	String	Yes	The message type.

The **messageClass** parameter can be:

- **ERROR**: will display an error message.
- **WARNING**: will display a warning message.
- **SUCCESS**: will display a success message.
- **INFO**: will display an info message.
- **HINT**: will display a hint message.
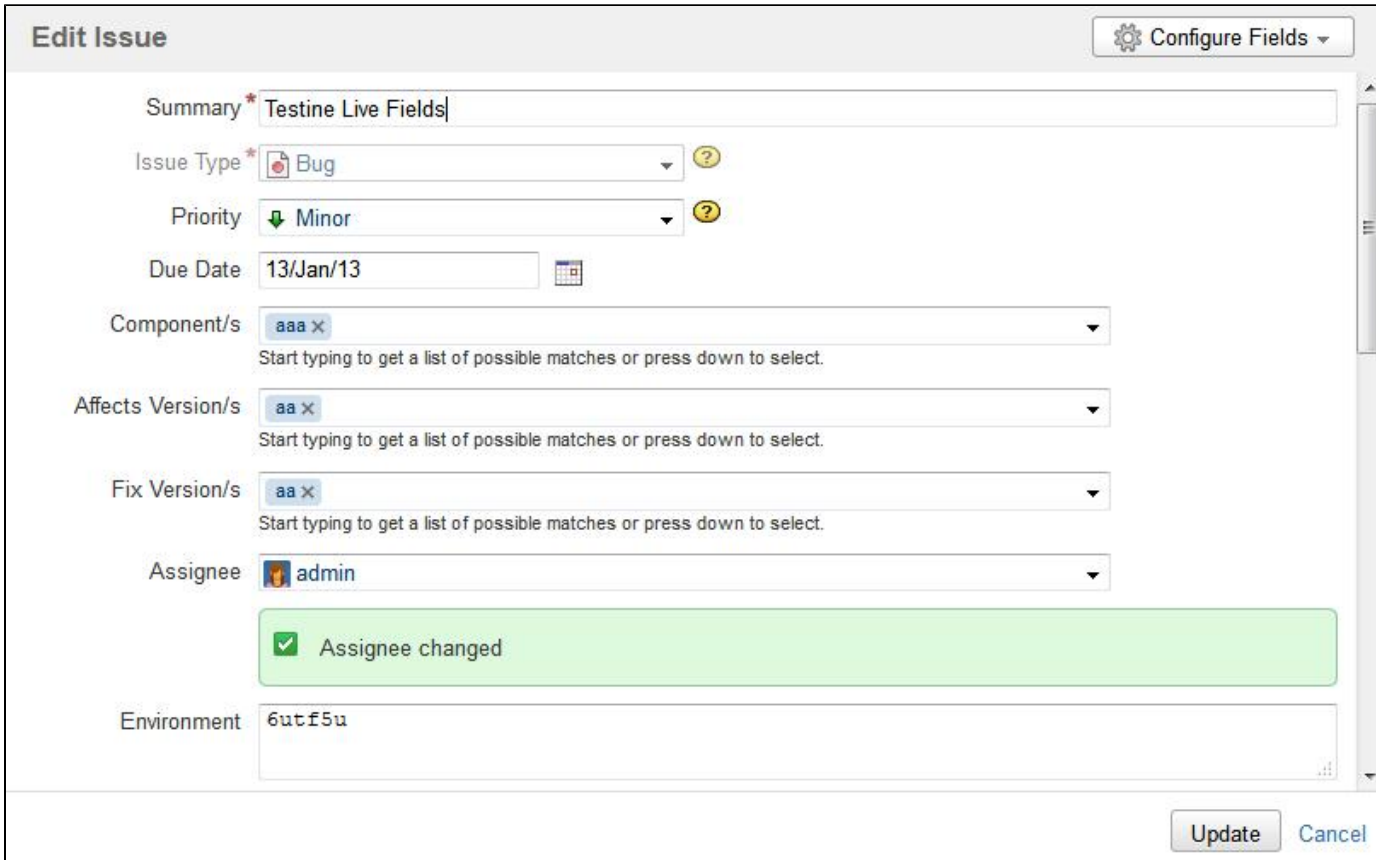
Example

```
lfShowFieldMessage("assignee", "Assignee changed", "SUCCESS");
```

On the issue screen, the message will be displayed like in the image below:
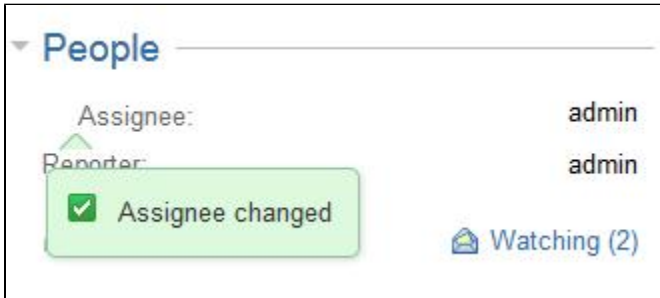
On the edit screens the message will be displayed like in the image below:

IfShowTab

Availability
This routine is available since

1. **JJUPIN 2.5.12+ / 2.6.7+**
2. **katl-commons 2.5.16+ / 2.6.8+**

Syntax

IfShowTab(field)

Description

Shows the given tab.

Parameters

Parameter	Type	Required	Description
field	String	Yes	Specifies the tab to show.

Example

If the assignee is admin, show the Field Tab from the issue.

```
if(assignee == "admin") {
      lfShowTab("Field Tab");
    }
```

Now that the field is shown, you can use lfHideTab to hide it on the issue screen.

See Also:

Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

lfWatch

Availability
This routine is available since

1. **JJUPIN 2.5**
2. **katl-commons 2.5**

Syntax

lfWatch(field, relatedFields, scriptPath[,javaScriptEvents]);

Description

Attach listeners for the events that you give as parameters in the function.

If you don't give any event, it attaches listeners to "change" event (triggered when the issue is updated).

Every time the event is triggered, the SIL script from scriptPath parameter runs.

This SIL script receives the values for the relatedFields and you can use them as: **argv[field]**.

Parameters

Parameter	Type	Required	Description
field	String	Yes	The field to listen.
relatedFields	Array String	Yes	The dependent fields required for the given field.
scriptPath	String	Yes	The script source to run when the event is triggered.
javaScriptEvents	Array	No	The events to listen to. It's any JavaScript event (check this list for references)

"change" event
When using the "change" event on a "labels type" field (Fix Versions, Affected Versions, Labels, Components, etc.), the event will never trigger when a label is deleted, but only when labels are added. We have noticed that for these fields the "focusin" event closely matches the behavior expected for the "change" event.
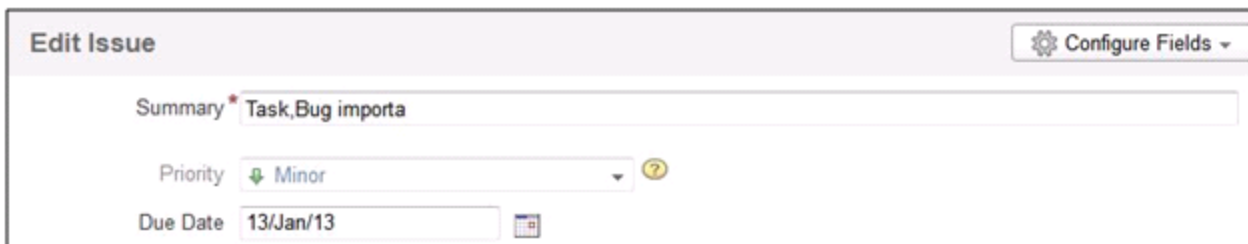
Example

```
lfWatch("summary", {"summary", "customfield_13706","components"}, "
hook.sil", {"keyup"});
//where field = "summary";relatedFields = {"summary",
"customfield_13706","components"};scriptPath = " hook.sil";javaScriptEvents
= {"keyup"}
```
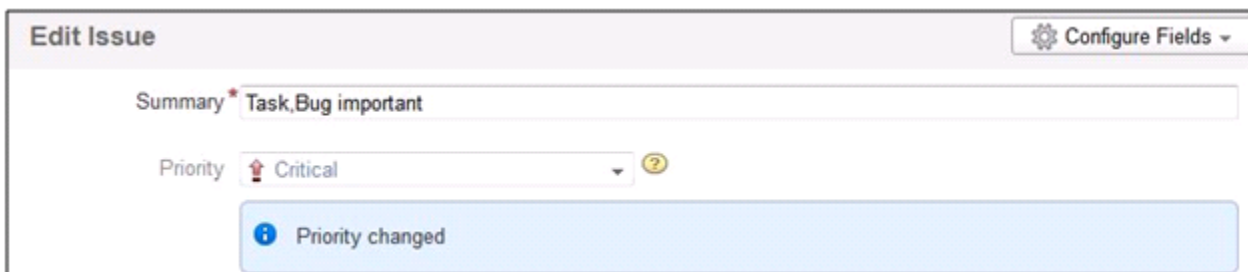
For the scriptPath parameter you can either give the relative path (as in the example above), or the absolute path as: "C:/Program Files/Atlassian/Application Data/JIRA/silprograms/hook.sil".

```
// hook.sil :
 if (contains(argv["summary"], "important")) {
 lfSet("priority", "Critical");
 lfShowFieldMessage("priority", "Priority changed", "INFO");
 }
```

Every time when the keyup event is triggered, the hook.sil is executed. When the summary field contains the word "important", priority field is set to Critical and a message will be displayed for the priority field.

The first image shows the initial value of the priority for the current issue, the next one shows the value it is changed to, after executing the code from hook.sil.

As we said before, the values from the relatedFields are accessed as **argv[field]**. For multiple values fields like **components** or **affectedVersion s** the value returned is in the following format: **val1|val2|val3**.

> **Info**
> For more information, see How 'Live Fields' work.

Additional Routines

Starting with JJupin 2.5.5, there are additional routines implemented into JJupin, concerning the display on SIL Runner. The UI has been dramatically changed (we hope it's for the best!) and now you can put messages in your long-running scripts so you can watch the progress on the runner.

- runnerLog

runnerLog

Availability
This routine is available since

1. **JJUPIN 2.5.5**

Syntax

runnerLog(message)

or

runnerLog(message, percent, action) **(Since JJUPIN 3.0.10)**

Description

Puts the message 'message' on the console of a runner gadget. This is a special routine making sense only in JJupin/SIL Excel Reporting and only for the runner. The use of it has no effect whatsoever besides for the runner.

Since JJUPIN 3.0.10, runnerLog routine can also render a progress bar by specifiing the percent we want to be set.

Parameters

Parameter	Type	Required	Description
message	string	Yes	Specifies the message to be put on the runner console
percent	number	No	Specifies the percent to be updated on the progress bar
action	string	No	Specifies the action to be executed (so far, the only action considered is init_progressBar - to initialize the progress bar; everything else will be ignored)

Return type

string, can be always ignored

Example

Let's modify the example used here:

The scripts would look like below:

execution_script.sil

```
date start_date = gadget_getDateValue(argv, "Start Date");
string tanks = gadget_getSingleValue(argv, "Tanks");
string infantry = gadget_getSingleValue(argv, "Infantry");
string rockets = gadget_getMultiValues(argv, "Rockets");
runnerLog("Preparing to start a war...", 0, "init_progressBar");
runnerLog("The war will start at this date: " + start_date, 10);
runnerLog("Building tanks...");
runnerLog("Built " + tanks + " tanks.", 30);
runnerLog("Gathering infantry...");
runnerLog("Gathered " + infantry + " brave men.", 60);
runnerLog("Fueling rockets...");
runnerLog(rockets + " ready.", 90);
runnerLog("Dispatching orders...", 100);
return "Good job! The world is now at war!";
```

parameter_script.sil

```
gadget_createDatePicker("Start Date", currentDate(), true, "Choose a start
date");
gadget_createInput("Tanks", "500", true, "The number of tanks");
gadget_createInput("Infantry", "1600", true, "The number of tanks");
gadget_createCheckboxGroup("Rockets", {"A big one", "A lot of small ones"},
"", false, "Do you want to use rockets?");
```

In this case, using the new runnerLog routine, when the script execution is done, the runner will look like this:

Development

Making the life of the SIL developer bearable

This page is dedicated to you, the developer who needs to accomplish tasks using what we have done.

We plan to do some improvements on the interface, but while you are waiting for them, let us give you some hints.

Log Custom Field - How to use it

Many SIL messages are put directly into the JIRA log. This is becoming a problem, your transition does not get executed, and you would like to know why.

So here it comes: if you want to display the log messages directly on the issue screen, use **Log Custom Field**, provided by Kepler Custom Field plugin.

In order to see the logs which refers to the JJUPIN plugin, you have to install Kepler Custom Field and add a custom field of type Log Custom Field. We'll assume JJUPIN is already installed.

> For more information about how to configure this custom field and how to test it, please see the documentation from here.

Example

Let's create a sil script using the IfSet routine, which sets the value for a JIRA field on the current issue.
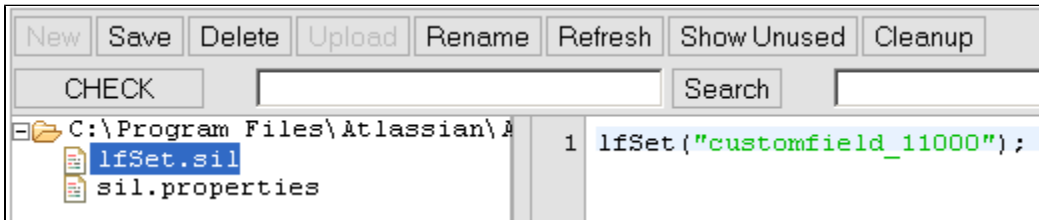
Suppose for some reason we forget to write the value this field should be set to and use the code block displayed below:

```
lfSet("customfield_11000");//which is plain wrong, we expect 2 parameters
here
```

But we expect that this should change the value for a field of type version picker, identified in JIRA by the id: customfield_11000, with the value: v1 and when we open the issue we see that the value of it is the old one.

So now we know something is wrong and we should look in the log file to see what happened, because we don't know what the cause is.

But we have the log custom field on the screen and we know that the log messages will appear as the value for this like in the image below.
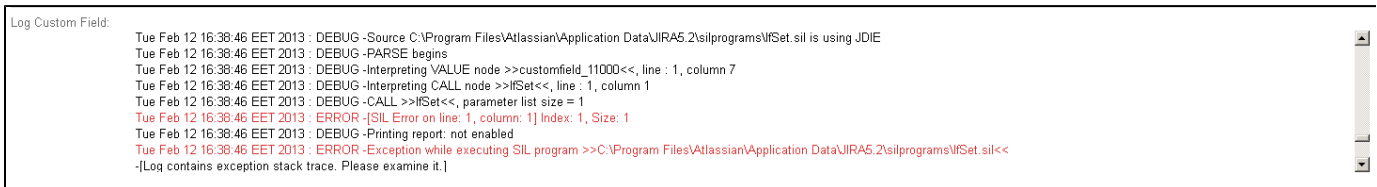
To search the messages, please scroll down and look for the messages which refers to the routine to be executed.

> It's just easier sometimes to use this field, than to search the log file provided by JIRA.
>
> The log on the screen **ONLY** contains main / important messages (not everything !) . However, it may reduce the development time quite dramatically !

The log custom field indicates that the script was scheduled to be executed but encountered an error at line 1 while executing it. And the advice is to examine the exception stack trace from the log file, but we think that

most probably the syntax is not correct, we search for lfSet routine in the documentation and we discover that we were right about it, so we change it to:

```
lfSet("customfield_11000", {"v1"});//which is correct
```

Now we go back to the issue and the value for the version picker field is set to v1 like in the image below and in the log custom field the messages does not contain any error regarding the usage of this routine:

Version Picker:	v1
Multi Checkboxes:	aaa, bbb

Log Custom Field:

```
Tue Feb 12 16:54:18 EET 2013 : DEBUG -Source C:\Program Files\Atlassian\Application Data\JIRA5.2\silprograms\lfSet.sil is using JDIE
Tue Feb 12 16:54:18 EET 2013 : DEBUG -PARSE begins
Tue Feb 12 16:54:18 EET 2013 : DEBUG -Interpreting VALUE node >>customfield_11000<<, line : 1, column 7
Tue Feb 12 16:54:18 EET 2013 : DEBUG -Interpreting VALUE node >>"v1"<<, line : 1, column 28
Tue Feb 12 16:54:18 EET 2013 : DEBUG -Interpreting CALL node >>lfSet<<, line : 1, column 1
Tue Feb 12 16:54:18 EET 2013 : DEBUG -CALL >>lfSet<<, parameter list size = 2
Tue Feb 12 16:54:18 EET 2013 : DEBUG -Detected field customfield_11000 to be custom field. Rewriting to customfield_11000
Tue Feb 12 16:54:18 EET 2013 : DEBUG -PARSE ends, no return encountered.
Tue Feb 12 16:54:18 EET 2013 : DEBUG -Printing report: not enabled
```

This is only an example of how log custom field is used with JJUPIN.

SIL Programming Warnings

Introduction

What you do when the script you are running doesn't have the expected result? Your first thought is to look in the Log files. But where?

Since JJupin 2.5 and katl-commons 2.5 we came in your help with a powerful tool that is useful when developing new scripts or debugging old ones.

How to use it

What you have to do? First of all you have to enable this feature. To do this you have to go to **Administration -> Add-ons -> SIL Configuration.**

SIL Programming Warnings

Enable Warning Report ⦿ ON ○ OFF
If enabled, will print a report in the logs showing any warnings we found during execution of the script.
This will be useful especially when developing new scripts or debugging old ones.

[Save]

> **Info**
> For more information about JJUPIN Configuration, see Administration Page.

How it works

Once you enabled it let's see how it works.

Every time a SIL Script is executed a warning report is created that shows the warnings that were found during the script execution.

Assume we have the following script.

```
function f(string s){
 description = s;
}
f(1);
```

Running this script will generate the following WARN logs in the Log file.

```
[jira.commons.sil.SILProgrammingWarnings] =========================================================================
[jira.commons.sil.SILProgrammingWarnings] == Found 1 code problems (potential: 0 serious, 1 style)(fatal: 0)
[jira.commons.sil.SILProgrammingWarnings] =========================================================================
[jira.commons.sil.SILProgrammingWarnings] [1] STYLE, line 5: Programming style: the routine f accepts on position 0 a type STRING, but you are calling it with NUMBER [f]
[jira.commons.sil.SILProgrammingWarnings] =========================================================================
```

This is the Warning Report that displays, in the first row, the number and the type of the problems that were found during the script execution.Then, it displays a detailed report of each problem found saying the line of the script that generated the problem and the problem message.

In our example was found only one problem of type STYLE, at line 5 that warn us about the parameter type we call the f function with.The routine was expected on the first position a String parameter but we gave it a Number.

There are three types of problems that may occur.

>   **STYLE**- a style problem

>   **SERIOUS** - a medium problem

>   **FATAL** - plain error

> **Info**
> You can also have the possibility to view the log messages on the issue screen, using Log Custom Field, provided by Kepler Custom Field plugin.

> **Important**
> Using SIL Programming Warning will not affect your SIL script execution.

What does this script? Sets the description with the value 1.

Look what happens on the issue after the script is executed.

The description is set to 1.

Let's see another example.

```
function f(string s){
 description = s;
}
f(1);
number a;
a += 2;
```

The execution of this script will generate the following warnings:

```
[jira.commons.sil.SILProgrammingWarnings] =================================================================
[jira.commons.sil.SILProgrammingWarnings] == Found 2 code problems (potential: 1 serious, 1 style)(fatal: 0)
[jira.commons.sil.SILProgrammingWarnings] =================================================================
[jira.commons.sil.SILProgrammingWarnings] [1] STYLE, line 5: Programming style: the routine f accepts on position 0 a type STRING, but you are calling it with NUMBER [f]
[jira.commons.sil.SILProgrammingWarnings] [2] SERIOUS, line 8: ADD operator called with uninitialized LHS operator. [a]
[jira.commons.sil.SILProgrammingWarnings] =================================================================
```

Calling SIL Scripts from Remote Systems

- Problem
- Solution 1 - REST
    - Step 1 - Create the Script
    - Step 2 - Add the Script to the Gadget
    - Step 3 - Identifying the Script ID
    - Step 4 - Calling the Script
- Solution 2 - SOAP
    - Step 1 - Authentication
    - Step 2 - Calling the Script

Problem

Complex integrations with external systems might require your JIRA instance to react somehow to external events. And there's no better way to express "somehow" than using a SIL Script.

Solution 1 - REST

Deprecation
As of JJUPIN 2.5.5 this method of calling SIL scripts is **deprecated and no longer supported**. Use the Common REST Service instead.

Required plugins
You will need the following JIRA plugins:

1. **JJUPIN**

Level: **ADVANCED**

The first solution is to use the same REST service that the SIL Runner Gadget uses to call scripts. This requires that the script is first added to the gadget. We will also use a special user that the external system will use to authenticate against JIRA, and we will restrict the Script from the gadget to this specific user.

Step 1 - Create the Script

We will create the script using the SIL Manager from Administration -> Add-ons -> SIL Manager. Select the folder where you want the file to be created and click **New->New file**.

For the purpose of this guide, we will use the following script:

```
    print(argv[0]);
    return "Hello World!";
```

Step 2 - Add the Script to the Gadget

SIL Runner Gadget

Name ⬝ test
A suggestive name for the program

Description
A detailed description of what the program does and maybe some usage tips.

Path

Select a folder 🔍

📂 silprograms
  📄 gadget.sil
  📄 liveFieldsTest.sil
  📄 validator1.sil

Select the file containing the SIL script to run

Security ○Public ◉User ○Group
Choose the security level. Only the selected entity and administrators will be allowed to see and run the script.

User admin
Select the user which is allowed to see and run the script

Save   Cancel

For the purpose of this guide, the special user I mentioned earlier will be the generic "admin".

Step 3 - Identifying the Script ID

Using your favorite browser's Developer tools, inspect the select list from the **Runner** tab and look for the newly created script. Since we restricted it to the special user, we'll have to be logged in as "admin" to see it.

You should find something like this:

```
<select name="silid" id="silid" class="select6">
    <option value="10001">test</option>
</select>
```

You guessed it! The ID is 10001.

> **Alternative**
> You can also find the script ID by analyzing the **krunnablesils** table from the JIRA database.

Step 4 - Calling the Script

Calling the script is done using a HTTP GET to the REST resource behind the SIL Runner Gadget.

> **Authentication**
> Don't forget to use basic authentication with your request.

In the URL, we need to specify 2 parameters:

- silid - the script ID
- silparams - the comma-separated list of additional parameters

To actually call the script, we will use this URL

```
<your_base_url>/rest/keplerrominfo/jjupin/latest/rungadget/run?silid=10001
&silparams=abc
```

This will return a JSON object containing 2 important parameters:

- key
- starthour

Example return value
```
{"key": "1","starthour": "1363342987392","message": "Sil script runnig.
Please wait..."}
```

The script is now running. This pair uniquely identifies your running script. You'll need to make another request to get the results, using the values you received in the first response.

```
<your_base_url>/rest/keplerrominfo/jjupin/latest/rungadget/verifyResponse?
key=1&starthour=1363342987392
```

Now you can have one of two types of responses:

1. if the **response** contains the **key** and **starthour** parameters (they will have the same values as the ones that were sent), this means that your script is still running and you'll have to do the call again and again until you find response 2.

Example return value
```
{"key": "1","starthour": "1363342987392","message": "Sil script runnig.
Please wait..."}
```

2. if the response contains the **returns** parameter, your script is done and the value of the parameter specifies the list of returned values from the script.

Example return value
```
{"returns": ["Hello World!"]}
```

That's all there is to it using the REST resource!

Solution 2 - SOAP

> **Required plugins**
> You will need the following JIRA plugins:
>
> 1. **JJUPIN**
>
> Level: **ADVANCED**

The JJupin exposes a web service that can be used to call remote scripts. That's what the call routine actually calls. The WSDL is available at **<your_base_url>/rpc/soap/sil?wsdl**. You will need to enable the web service by going to Administration -> Kepler General Parameters -> JJupin and setting WSEnabled to true.

The web service provides a "execute" method which takes 3 parameters

- in0 - String - authentication token
- in1 - String - the path of the file containing the script
- in2 - String [] - parameters to be sent to the script

Step 1 - Authentication

To authenticate your request, you can either use the JiraSoapService to generate an authentication token and pass it to the execute method, or use Basic Authentication or OAuth to authenticate your request. Note that when using basic authentication or OAuth, you will still need to provide a non-null, non-empty bogus token.

Step 2 - Calling the Script

All that's left to do now is to call the web service.

Additional Documentation

Before using JJupin check out the Simple Issue Language documentation for a better grasp of SIL usage and capabilities.

Here you will find some useful tutorials that will help you get started with JJupin.

If you would like to share your idea, please notify us.

Known problems (and their resolutions)

We strive for perfection. However some things really do not depend on us. For some we consider there's room for improvement but we didn't have the time to achieve them.

So here's a list of common problems (we will update the page with each finding):

No	Affected Functionality	Problem	Explanations and the Resolution
1	listeners, service	Upgrading **katl-commons** to a superior version, but the behavior is the same on listeners and services.	There are classloader issues on JIRA, disabling the plugin does not clear completely the classes used by the listeners and services. This is somehow normal, since the listeners and services are loaded in the "superior" layer, not in the plugin OSGI framework. Since this is a JIRA behavior, we cannot do too much about it. **Resolution 1:** disable-enable the listeners and services**.** This may or may not work, it depends on the version of JIRA. **Resolution 2**: cold restart of JIRA (this for sure works).

2	configuration pages	Upgraded JJUPIN, but configuration pages are looking odd	We are providing **backward compatibility only**, so when you upgrade a plugin, make sure you upgrade the dependencies as well.
			This happens when you install for instance, JJUPIN v2.5.5 directly from jar file (not .obr file) but you preserve katl-commons v.2.5.7.
			The version of katl-commons is unable to provide the services JJUPIN requests, therefore there are errors in the logs and pages are looking odd
			Resolution: upgrade katl-commons to the latest level offered by the corresponding dependent plugin / JIRA version (for our example, minimal 2.5.8)
3	plugin installation	Plugin fails to (re)enable after it was disabled, or some components remain disabled (UPM shows "x of y modules enabled" where x < y).	**Resolution:** Re-install the plugin. If you're uploading the plugin file from your local disk, uninstalling the previous version is not required. Just upload over the existing version. If you're installing the plugin from the marketplace, you'll need to uninstall first since there's no option to "Install" plugins that are already installed on your JIRA instance.

4	plugin installation	`OsgiContainerException: Cannot start plugin`
		
		caused by:
		
		`org.osgi.framework.BundleException: Unresolved constraint in bundle`
			Resolution: Install correct katl-commons or warden, as explained in the exception. You need to provide the correct dependency. Even if we provide the .obr archive, sometimes, in some containers, this is not enough and a reinstall is needed.

5	checking scripts	A number of messages are logged to ERROR, but the Check button says the check is OK.	Affects version 3.0. The messages are wrongfully logged to ERROR and will be downgraded to DEBUG in a future version.

Previous versions documentation

If you have an older version of JJupin here is the documentation for JJupin 2.5 and 2.6.

License & Pricing

Info

This product requires a **Kepler** license, which can either be provided as the **jjupin.lic** file, or as the key generated via the Atlassian Marketplace. You can find more about licenses here.

You can find pricing details on Atlassian Marketplace or visiting our site: Kepler Products.

Contact

Software Development and Services

Florin Haszler
Phone: + 4021 233 10 80
Email: fhaszler@kepler-rominfo.com

http://www.kepler-rominfo.com

JIRA Plugins Support

Please see Getting Support.

Backup and restore

At Restore: install first the plugins

Mundane operations as backup and restore may pose some problems to the unsuspecting JIRA administrator. Since all the Kepler plugins create some tables in the JIRA schema - we created this mechanism long before Active Objects was introduced into Atlassian's framework - you need to take some precautions at restore.

Specifically, at restore you need to create the tables used by our plugins. You do not need to copy schema from the previous JIRA or fill it with data, you just need to **simply install the plugins into JIRA before restoring** (enabling the plugins would create the needed tables).

JJUPIN has two dependencies:

1. katl-commons (core support)
2. warden (used for licensing)

For reference, these are the tables created by each add-on

Plugin	Tables
JJUPIN	krunnablesils
	krssecurity
	klistenersils
	jjlf_config
	jjlf_project
	jjlf_category
katl-commons	kplugins
	kpluginscfg
	kissuestate
	kstatevalues
warden	-