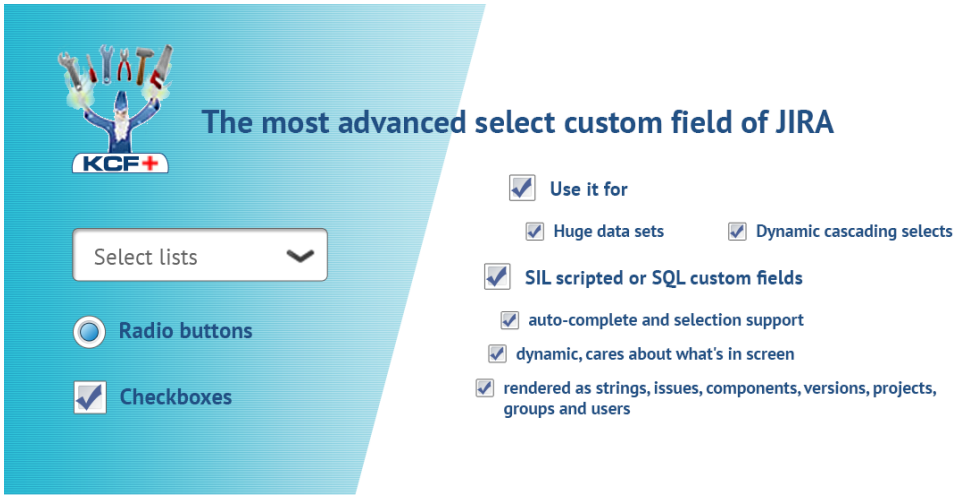
# Home




The KCF-PRO is the ultimate dynamic select list custom-field. It covers multiple ways of rendering the select lists & radio buttons as issues, users, project, etc and allows you to create autocomplete fields for just about anything. Plus, it is scripted, and SIL means unlimited flexibility.

## Recently Updated

Kepler Custom Fields Pro 1.0
Feb 28, 2017 • created by Alexandru Geageac

KCFPRO11
Feb 28, 2017 • attached by Alexandru Geageac

Uninstall via Atlassian Universal Plugin Manager
Feb 27, 2017 • updated by Confluence Administrator • view change

Kepler Custom Fields Pro
Feb 23, 2017 • created by Confluence Administrator

Installation
Feb 23, 2017 • created by Confluence Administrator

install_upm_1.png
Feb 23, 2017 • attached by Confluence Administrator

install_upm_2.png
Feb 23, 2017 • attached by Confluence Administrator

Installing the KCF-PRO add-on
Feb 23, 2017 • created by Confluence Administrator

How to use SQL Datasource on several databases
Nov 29, 2016 • created by Alexandra Topoloaga

Examples
Nov 04, 2016 • created by Confluence Administrator

Example 2 - Issue Links
Apr 01, 2016 • created by Alexandra Topoloaga

Example 4 - Dependent custom fields
Dec 14, 2015 • created by Alexandra Topoloaga

Example 7- (Video) - How to use argv["query"]
Dec 08, 2015 • created by Alexandra Topoloaga

Configuration
Dec 08, 2015 • created by Alexandra Topoloaga

Example 6 - (Video) - More Dependent Fields

Dec 02, 2015 • created by Confluence Administrator

***Navigate space***

# Rationale & How It Works

## Kepler Custom Fields Pro

This add-on came out as a necessity in a real large project. The idea of the add-on has been in the back of our minds since we first started to customize JIRA for our clients, however we didn't have the time to implement it, until now.

### The Problem

Let's talk about select lists, for instance. When dealing with select lists in JIRA, you have the possibility to create **static** select lists via the usual custom fields. This is awesome, but not enough. Select lists may be **dynamic in nature**, in which case the usual custom fields are really not helpful.

Imagine that your field is actually a select from the database. Or a list of components that should be determined at runtime based on user roles in the project.

We tried to solve such several problems in JJupin Live Fields by hiding select lists options elements. However, this is usable only for select lists which are known (static), because we could not determine easily at runtime what values have been shown in the UI.

## Entering the Stage: KCF - Pro

KCF-Pro resolves all the above problems of the dynamic select lists, in our own way: data for the CF is obtained through SIL (although we plan to add more options on it); at save time, we allow one more optional interaction with the CF through SIL, enabling you to update dependent fields or issues (or maybe just log it).

The data is obtained through a SIL script, **allowing you to intervene in the process of data build-up even in the stages of the autocomplete**. This is a huge advantage for JIRA-addicts (like we are) because it allows you to filter more efficiently based on what user types in the field (car parts, computer parts, airplane parts, ...)

**Selection of the data is scripted**, too. This means that you can intercept in SIL the selection the user has made and act on it.

Data may be presented to the user as:

- Select lists (simple or with auto-complete, single or multiple)
- Radio buttons (if you have only a handful of choices)
- Checkboxes (for multiple, but few, choices)

To please the eye of the user and to make the lists look like the JIRA standard lists, we added special renderers for:

- Issue
- Component
- Project Version
- Project
- Group
- User

The default renderer is used just to show general data.

### When To Use KCF-Pro

We see two immediate usage of KCF-Pro:

- Dynamic choices
- Huge Datasets, when the user must filter via autocomplete.

The plugin functionality overlaps with UGP-Pro and partially with DBCF.

# 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.



## Installing the KCF-PRO add-on

On this page:



### Requirements

KCF-PRO 2.0 is compatible with JIRA 7 only, starting with JIRA version 7.0; JDK version should be at level 8.

KCF-PRO 2.0 bundles the Warden and KATL-Commons plugins. It is upward compatible meaning it works with any Warden version 4.0 and up and with any KATL-Commons version 4.0 and up. Please see **Installation Notes** chapter below for an explanation.

### Install via Atlassian UPM (Universal Plugin Manager)

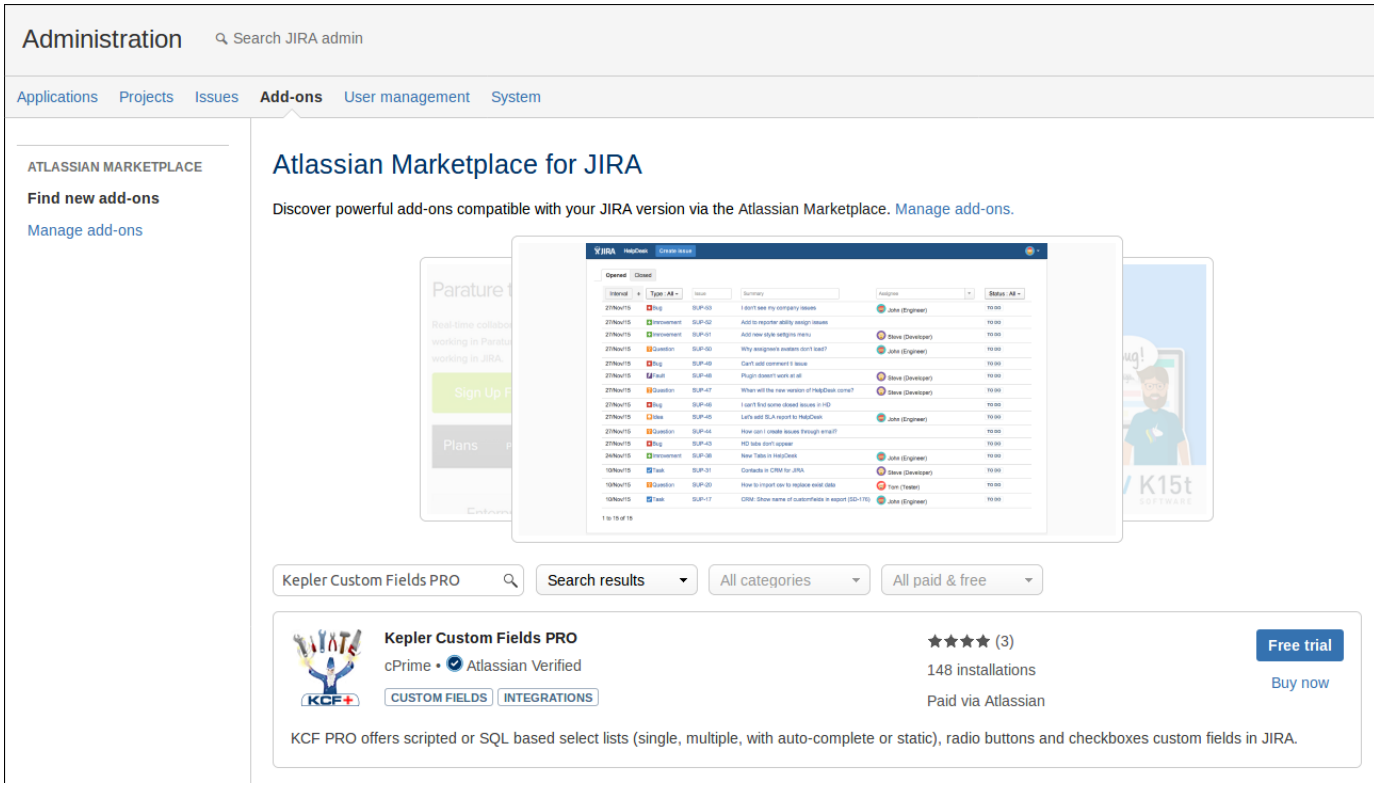
Installing the add-on follows the general guidelines outlined by Atlassian in the JIRA Documentation.

#### Steps to install

Step 1: Go on the *Administration menu  Add-ons  Find new add-ons*

Step 2: Search for our add-on by entering "Kepler Custom Fields PRO" in the search box

Step 3: Press on "Buy now" or "Free trial" to install



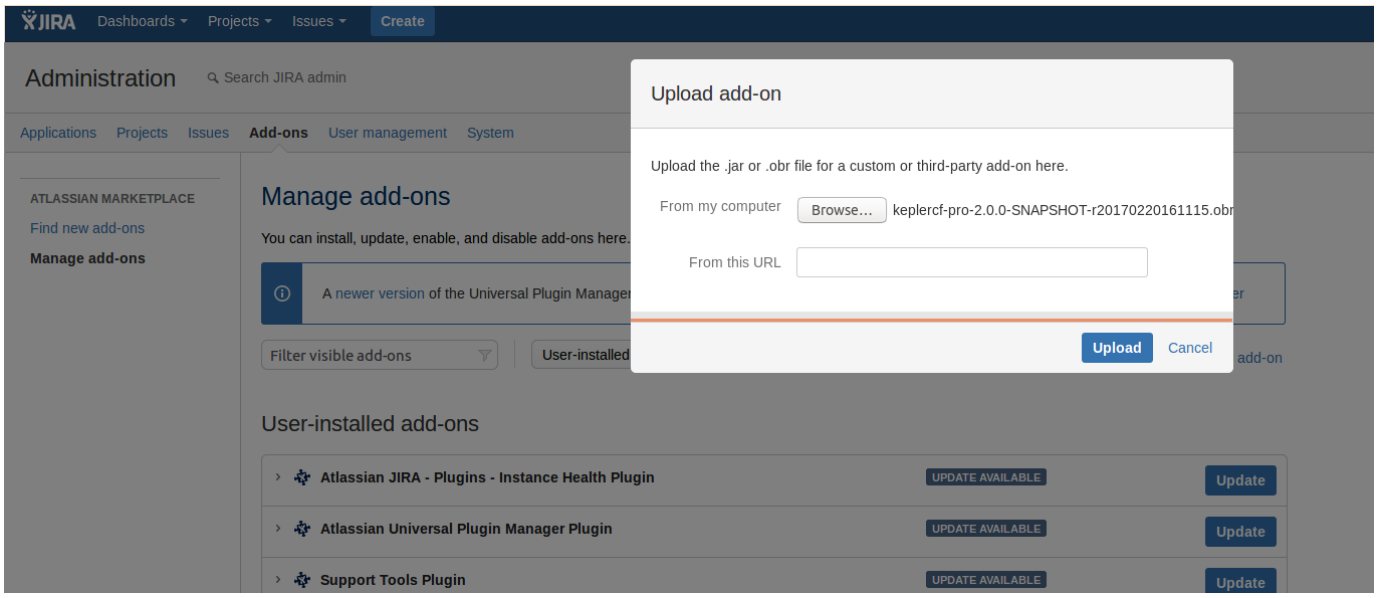

## Manual install

Step 1: Download the correct kepler-custom-fields-pro obr file from Atlassian Marketplace

Step 2: Go to *Administration  Add-ons  Manage add-ons*. Install the previously downloaded OBR file by using '*Upload add-on*' link.

Step 3: [Optional, but highly recommended]: If not already done, 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
```




## Installation Notes

KCF-PRO packages together 3 JIRA add-ons:

1. Warden add-on, responsible for licensing of the cPrime products. This add-on is not published on Atlassian Marketplace, however is it bundled in all our paid add-ons,
2. KATL-Commons, providing all SIL functionalities and common services built over JIRA, and
3. KCF-PRO add-on, providing the select lists you're looking for

KATL-Commons and Warden are shared across our line of products. If any of those become disabled, the functionality built on top of them will cease to work. Unlike Warden, KATL-Commons is published on Atlassian Marketplace and updates that are compatible for this add-on should be installed as soon as they appear, since they contain bug fixes and enhancements.

### What should I do if I installed an incompatible version?

1. Remove (uninstall) KCF-PRO
2. Uninstall Warden add-on
3. Uninstall KATL-Commons add-on
4. Install the right version of KCF-PRO (the one compatible with your JIRA). Make sure you use the OBR file you downloaded or use directly the UPM search & install feature (recommended)
5. KATL-Commons and Warden should now have the right versions as well
6. Check any dependent add-ons, such as JJUPIN or Blitz-Actions, etc. These plugins must be re-enabled manually ⚠

## Licensing

### Dual Licensing support

All versions support both Kepler and Atlassian licenses, but you only need one valid license to run the plugin, which can either be provided as the **keplercf-pro.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.

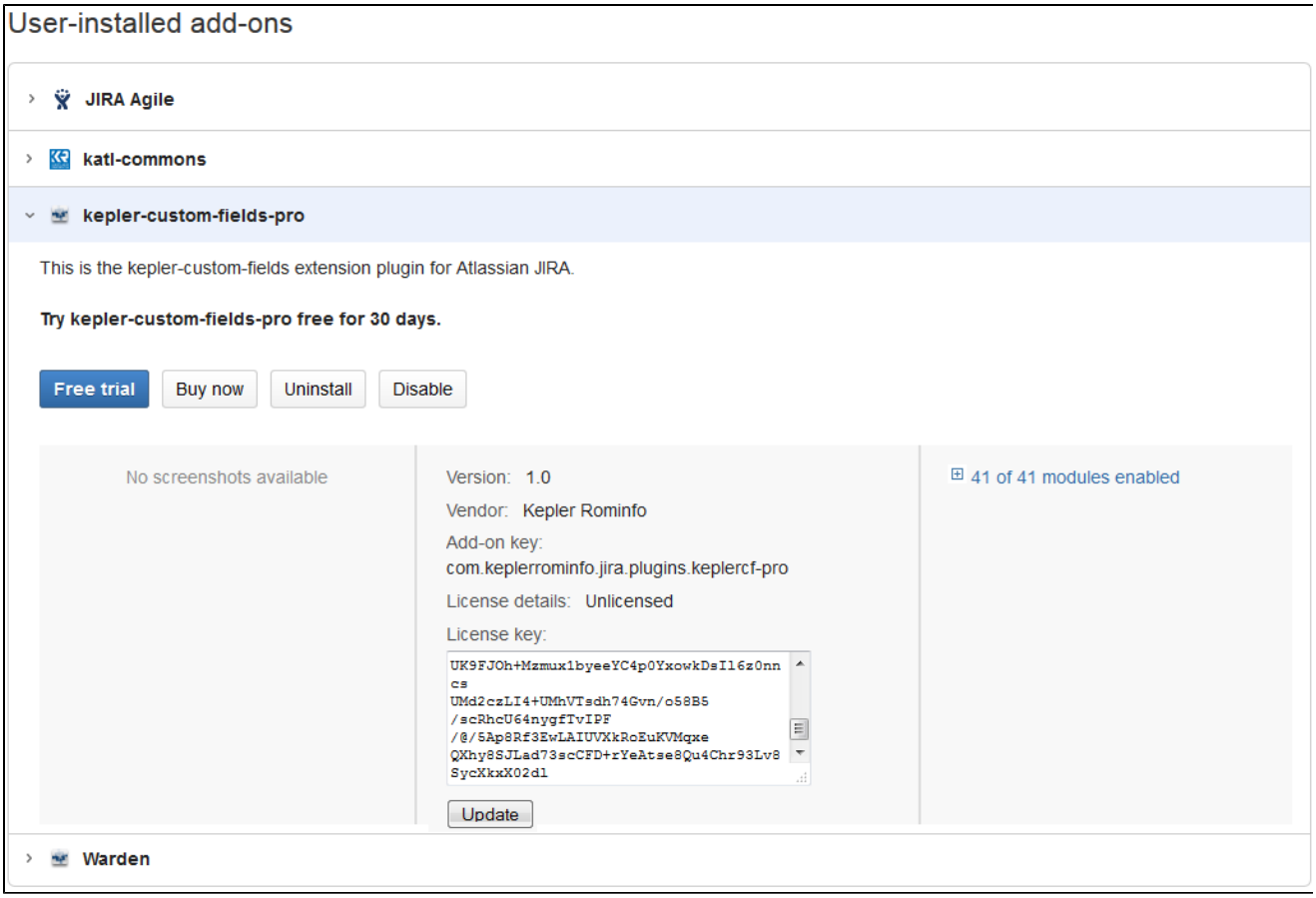
### Atlassian Licensing

> **Note**
> To support Atlassian licenses you need to install **katl-commons 3.0+** *before* installing Kepler Custom Fields Pro.

The Atlassian Marketplace allows you to easily purchase or generate an evaluation license for **Kepler Custom Fields Pro**.

### Using Universal Plugin Manager 2.0.1+

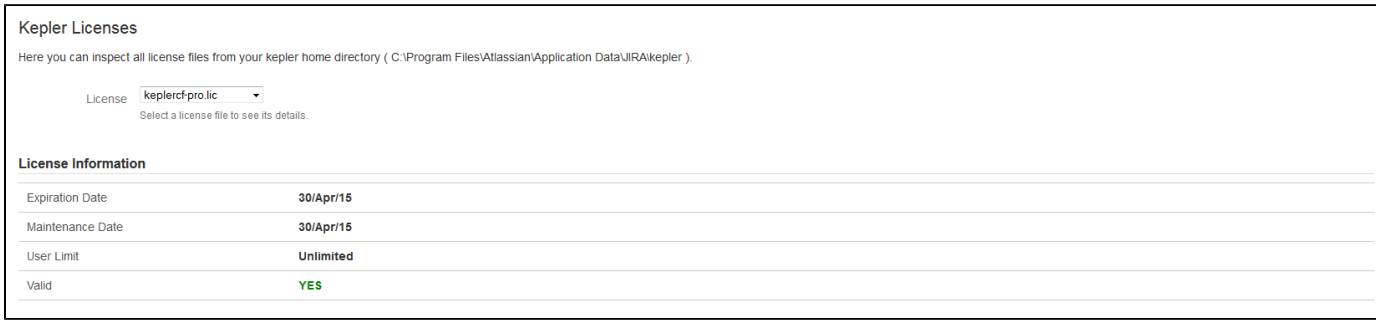
After generating the license key, all you have to do is access the **Administration-> Plugins** section in your JIRA instance and paste the key into the corresponding plugin textbox.




## Kepler Licensing

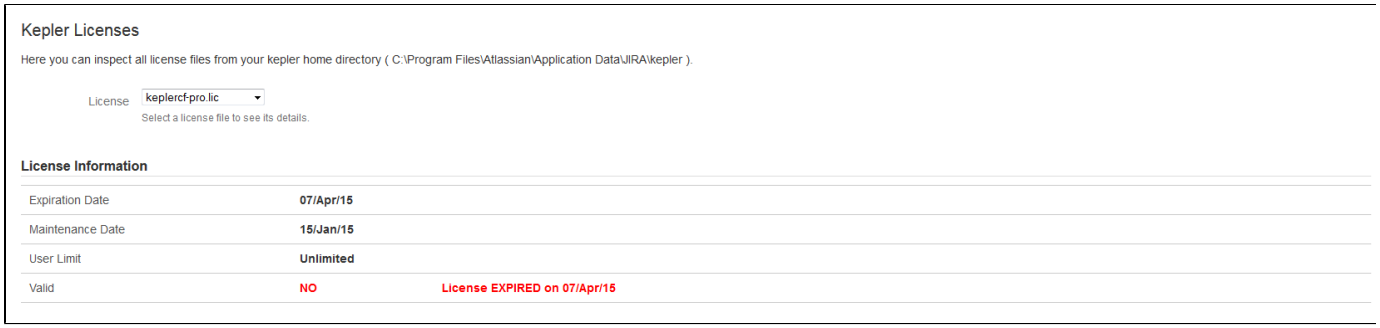
The Kepler license is a file (**keplercf-pro.lic**) which must be placed in the <JIRA_HOME>/kepler folder. You can either generate and download a free evaluation license by registering on our site and accessing the **Licenses** section, or you can purchase the plugin by following these instructions.

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:




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

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

**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.

## 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).

Kepler Custom Fields Pro 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 |
|---|---|
| katl-commons | kplugins<br>kpluginscfg<br>kissuestate<br>kstatevalues |
| warden | - |

## Uninstall

### Uninstall via Atlassian Universal Plugin Manager

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

## Manual Install

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 manually

> 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 `Kepler Custom Fields PRO` plugin.

### Restart the server

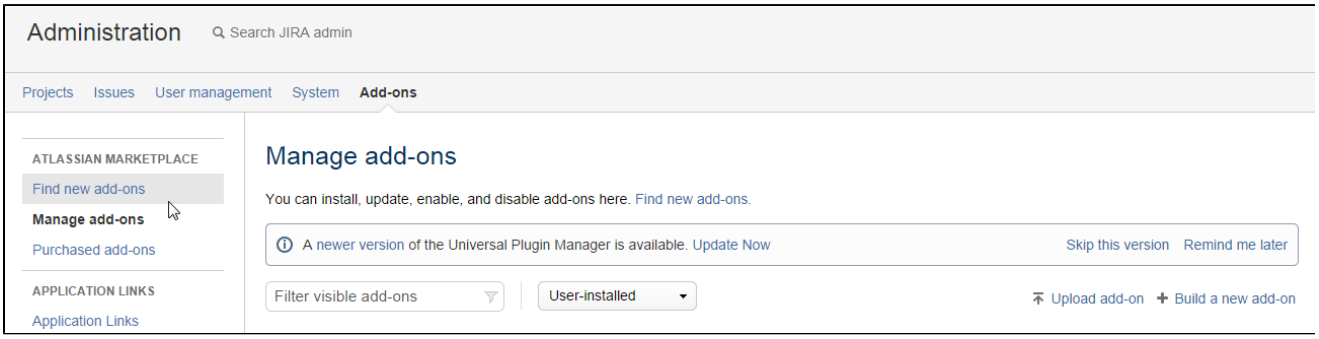
Now you can restart Jira server

## Uninstall via Atlassian Universal Plugin Manager

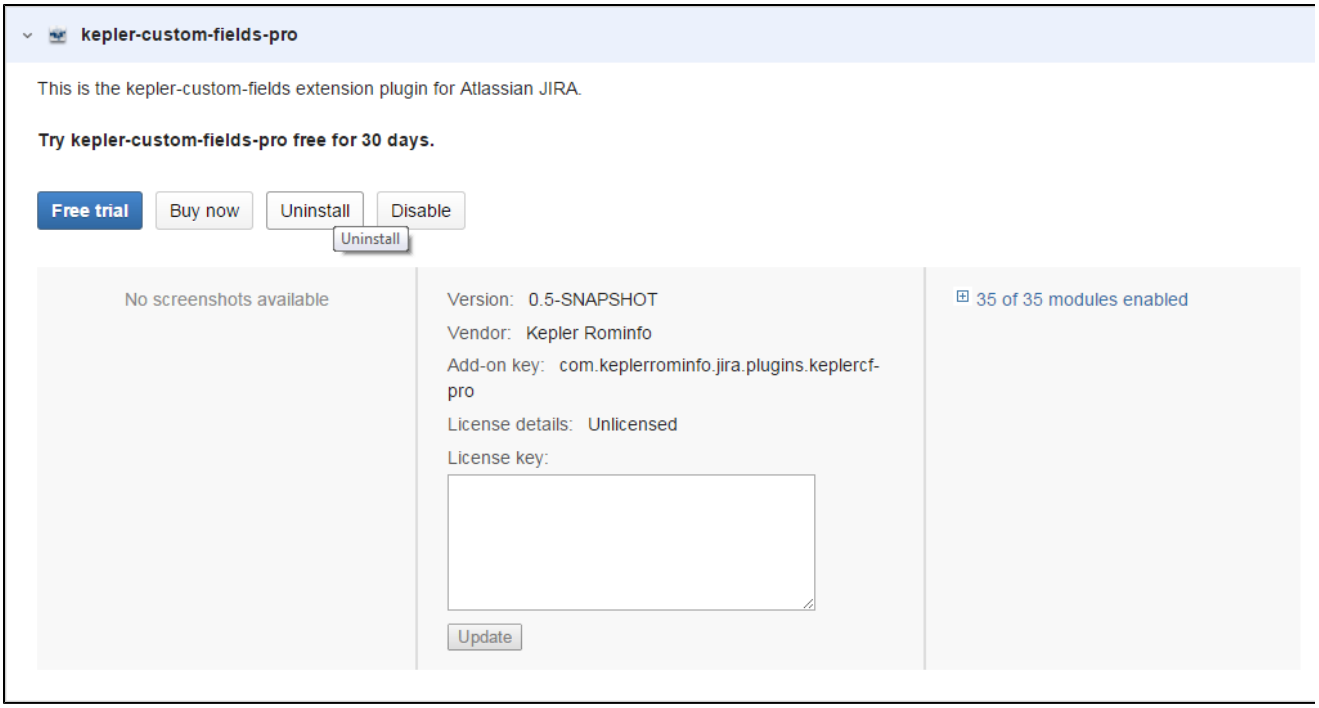
### Uninstall via Atlassian Universal Plugin Manager

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

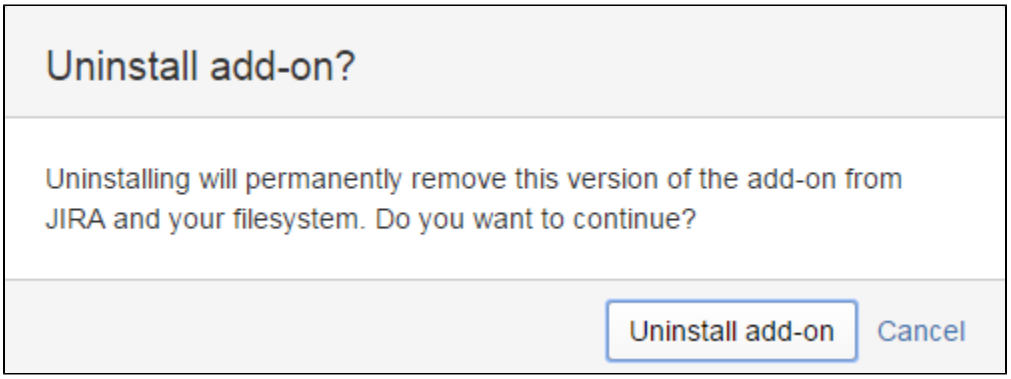
1) Log in as administrator and go to Administration->Add-ons->Manage add-ons



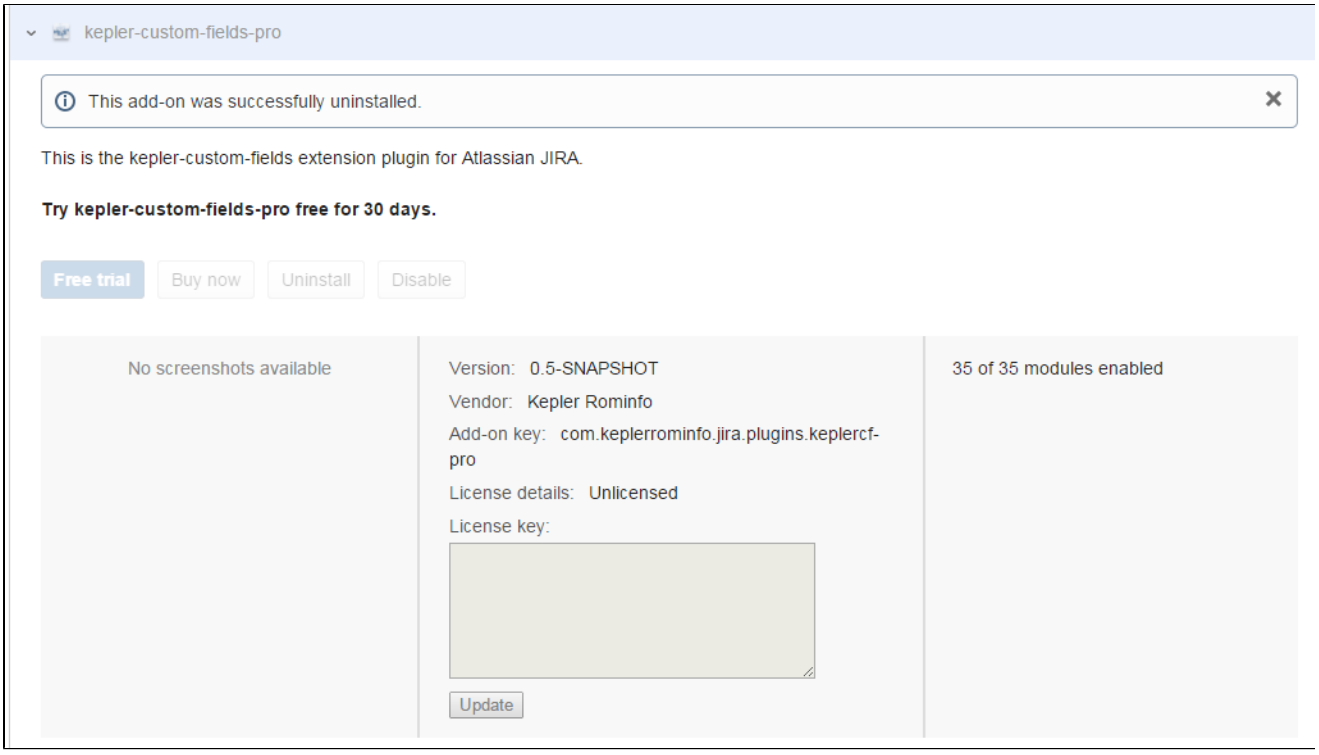

2) Search for `Kepler Custom Fields PRO` plugin in `User-installed add-ons` section and click on `Uninstall` button




3) Press `Uninstall add-on` when the uninstall confirmation dialog box appears




4) A message "successfully uninstalled" should appear




# Configuration

## How to configure a KCF-PRO CustomField

Go to the **Administration** page, choose **Custom Fields** link and add a type of available KCF Pro Custom Fields.

## Configure Custom Field

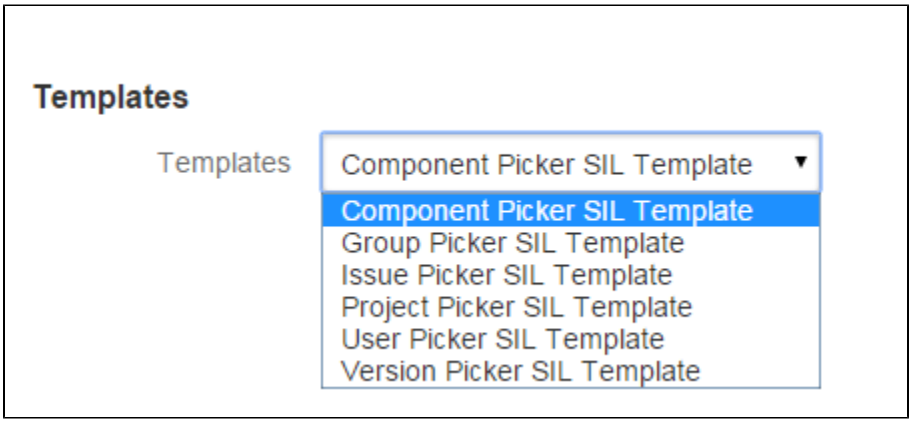
After you've created the field, go to the corresponding **Configure** link. A page like in the image below will appear**:**


## Default value

Depending on the type of the custom field, the default value can be set in two ways. Both of them are explained here.

## Template Configuration

In order to make Kepler Custom Field PRO more easy to use we have added a template for each option type, as you can see in the picture below:

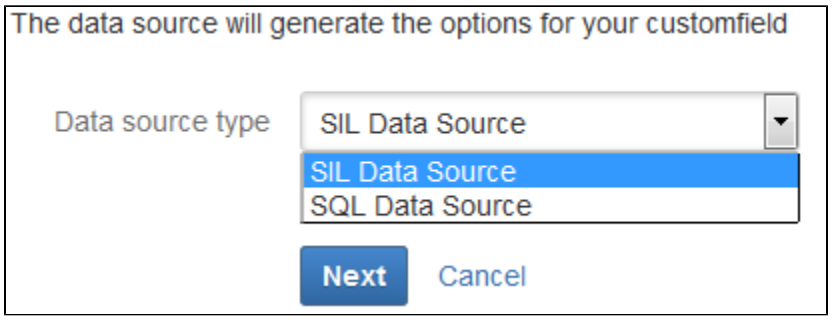



So, if you don't want to configure all the other settings for such a custom field, you don't have to. All you have to do is select a template from the ones above and the required properties will be automatically set. This way, you can simply enjoy the goodies KCF Pro offers you.

> For the templates regarding components and versions, you will need to edit the SIL script so it uses your own project key.
>
> You can find the scripts used by the templates in "silprograms" folder, in the sub-folder named like your custom field id.

## Edit Data Source

You have to select a **Edit Data Source** for populating the options of the custom field.




The data source is used to obtain the values which will populate the custom field.

A script used as data source can either return a string array or an option array. If a string array is returned, its elements it will be saved as pairs (value, value); if an option array is returned, the elements will be saved as KPOption structure, as you can see here: Predefined Structure Types#KPOption.

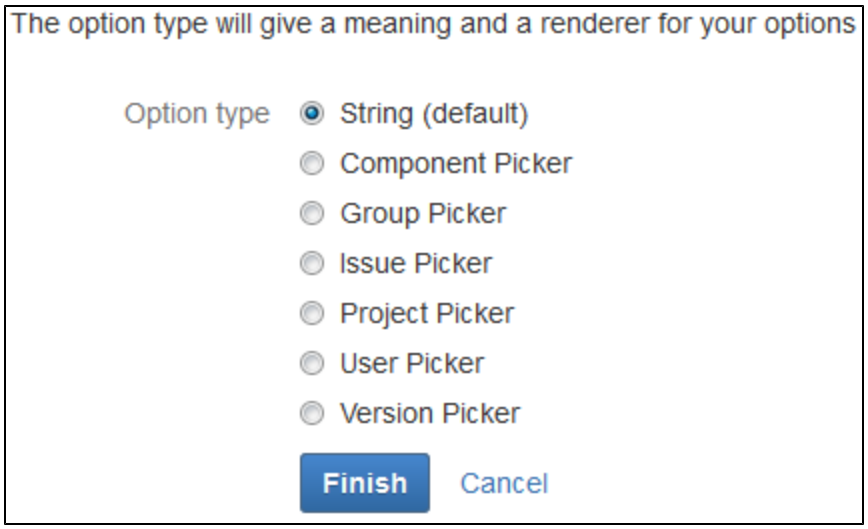
The data source script is also invoked when JQL search is on going.

As you can see in the screenshot above, there are 2 types of data sources. You can find out more details about each type of data source on the pages : SIL Data Source and SQL Data Source .

You can find some examples of data sources in our Examples page.

## Edit Option Type

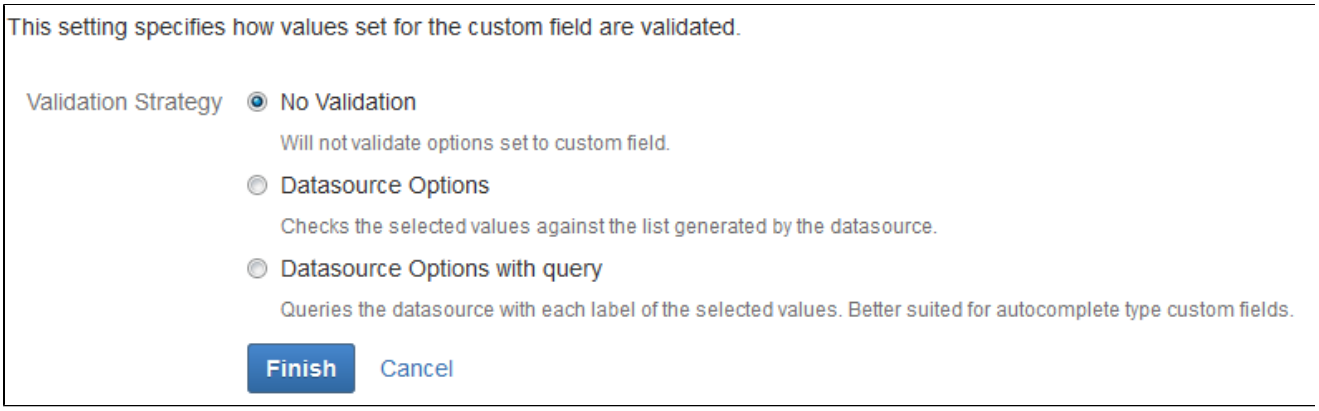
You have to select the **Edit Option Type** link for the types of option that you want to use.




The default option type is rendered as a string. You can find more examples regarding the option types here.

## Edit Validation Strategy

You have to select the **Edit Validation Strategy** link for deciding the type of validation for the custom fields values which you have set.

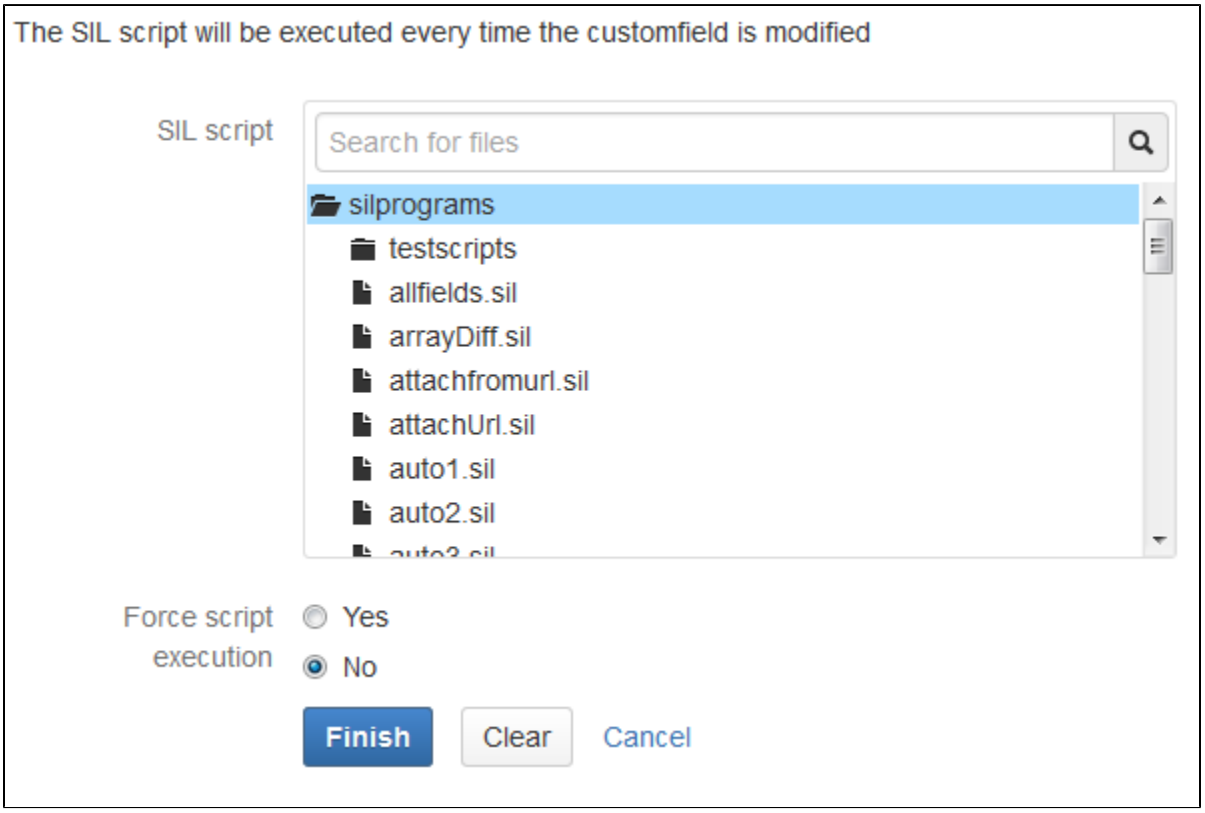



If you choose :

- **No Validation** - the options set to custom field won't be validated
- **Datasource Options** - the options set to custom field will be checked by the list generated by the datasource
- **Datasource Options with query** - the options set to custom field will be validate by the datasource which is queried with each label of the selected values. This option is recommended for autocomplete type custom fields.

## Edit Select Script

You have to select the **Edit Select Script** link for selecting the SIL script which will be run each time the custom field is modified.

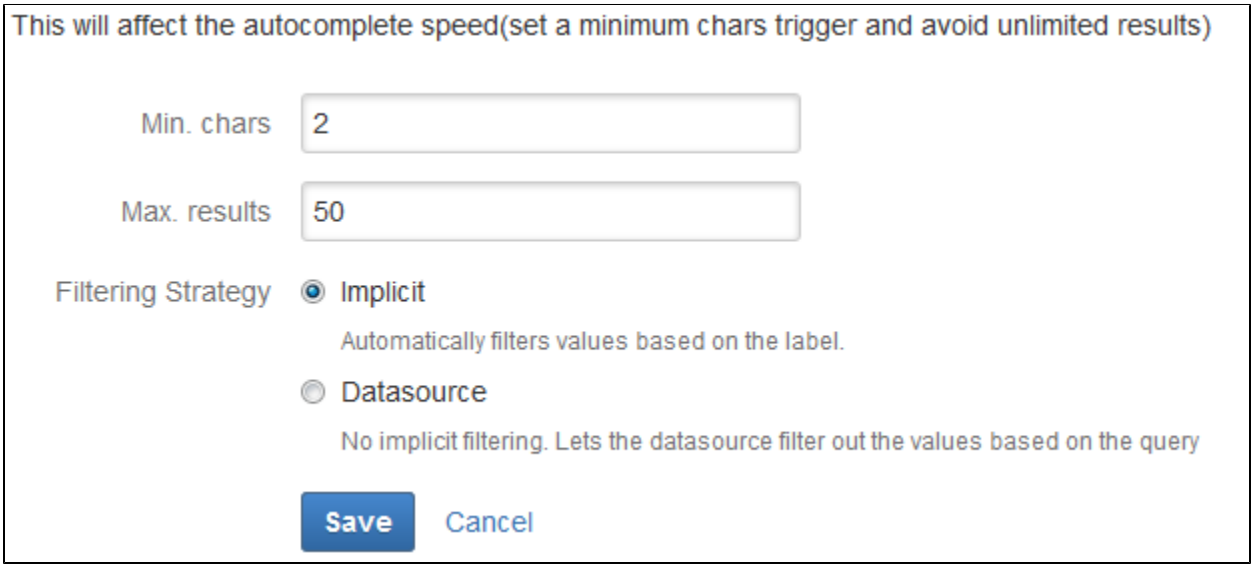



When the custom field value is changed (either using inline edit or using the edit issue screen and saving the changes), the argv value changes. The argv contains the old value of the custom field and the new value of the custom field.

When the radio button is set to "Yes", the script will be executed even if the value of the field does not change; otherwise, the script will not be execute if the value is not changed.

## Edit Autocomplete Settings

The **Edit Autocomplete Settings** link helps you to configure the autocomplete settings for the custom field.




With the **Min. chars** option you can decide which is the minimum number of characters that can be used for autocomplete.

**Max. results** represents the number of the results that can result from autocomplete. In the example above the script will show maximum 50 results.

The **Filtering Strategy** options are : Implicit and Datasource.

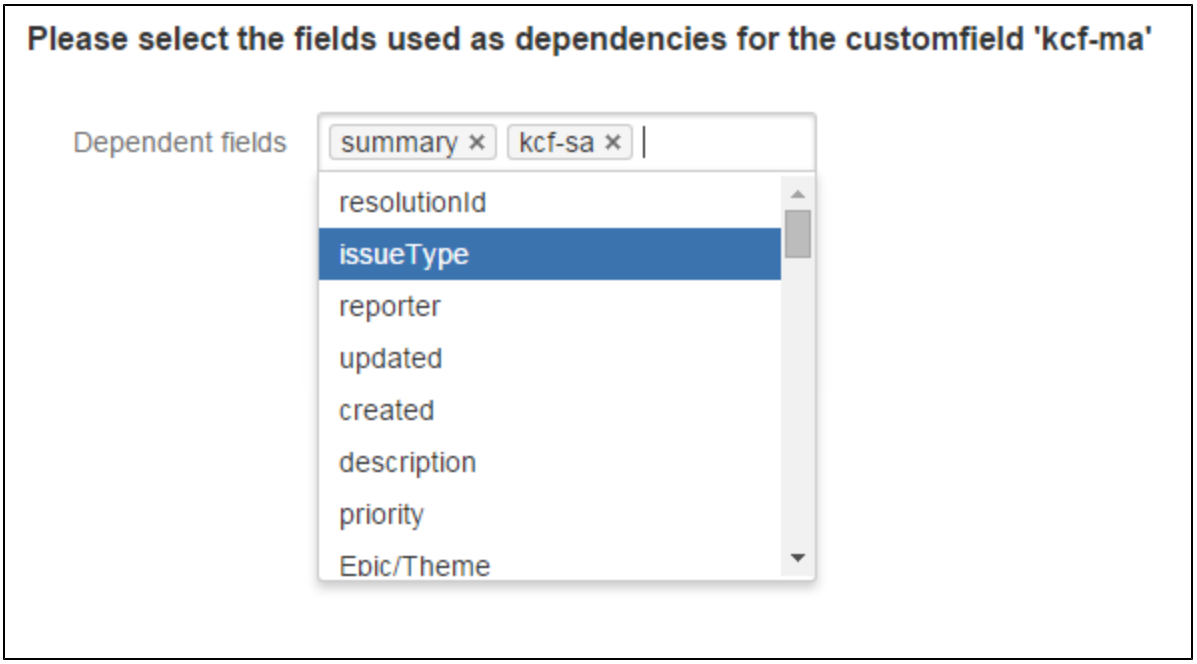
- Implicit - this option filters the values by the label
- Datasource - this option doesn't use an implicit filtering. It calls the datasource to decide what filter should be applied based on the query.

> If you select the SQL Data Source the default filtering strategy will be set on Datasource.

## Dependencies

> In order to use dependent fields you must have **JJUPIN** plugin installed and enabled.

In order to use the screen values for a field in the data source, you have to add it as dependent field.




You can find an example regarding dependent fields here.

### Related Content

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

## Option Type

Our new plugin, Kepler Custom Fields PRO offers you seven (including the default option type, which shows you the information as plain text) option types.

The seven option types are:

- String (default)
- Component Picker
- Group Picker
- Issue Picker
- Project Picker
- User Picker
- Version Picker

All the option types above are available to be used with any type custom field.

You can find more information about our new custom fields in User Guide page.

**Component Picker**

An option is rendered as a project component.

This is the **SIL** script to get project components:

```
string projectKey = "TEST";
string[] componentNames = admGetProjectComponents(projectKey);
number[] componentId;
for(string componentName in componentNames){
    JComponent componentNames = admGetProjectComponent(projectKey,
componentName);
    componentId += componentNames.id;
}
return componentId;
```

It gets all the components existing in the project "TEST". The components will be added as options to the custom field which uses this SIL script as data source.

This is the **SQL** script to get all the components:

```
select cname, id from component;
```

You can see it in action here: KCF - Checkbox

**Group Picker**

An option is rendered as a group.

This is the **SIL** script to get groups:

```
return userGroups(currentUsername());
```

It gets all the groups to which current user belong. The groups will be added as options to the custom field which uses this SIL script as data source.

This is the **SQL** script to get all the groups:

```
select group_name from cwd_group;
```

You can see it in action here: KCF - Multi Select

**Issue Picker**

An option is rendered as an issue.

This is the **SIL** script to get issues:

```
return selectIssues("created < now()");
```

It gets all the issues previously created. The issues will be added as options to the custom field which uses this SIL script as data source.

This is the **SQL** script to get all issue keys.

```
select  p.project_key ||'-'||i.issuenum as "KEY"
from jiraissue i
inner join project_key p on p.project_id = i.project
```

You can see it in action here: KCF - Single Autocomplete

**Project Picker**

An option is rendered as a project.

This is the **SIL** script to get projects:

```
string [] projects = allProjects();
return projects;
```

It gets all the projects previously created. The projects will be added as options to the custom field which uses this SIL script as data source.

This is the **SQL** script to get all projects:

```
select pkey from project;
```

You can see it in action here: KCF - Radio Buttons

**User Picker**

An option is rendered as an user.

This is the **SIL** script to get users:

```
function getUsers(string [] groups){
  string [] users;
  for(string group in groups){
    string [] currentGrp;
    currentGrp = addElement(currentGrp, group);
    for(string user in usersInGroups(currentGrp)){
      users = addElementIfNotExist(users, user);
    }
  }
  return users;
}


string [] groups = {"jira-users"};
string [] users = getUsers(groups);
string [] res;
for (string user in users) {
    if (contains(user, argv["query"])) {
    res = addElementIfNotExist(res, usernameToUserKey(user));
    }
}
return res;
```

It gets all users existing in some groups (in this case, "jira-users") and adds the usernames that match the search as options to the custom fields using this data source.

This is the **SQL** script to get all users:

```
select user_name from cwd_user;
```

You can see it in action here: KCF - Multiple Autocomplete

**Version Picker**

An option is rendered as a version.

This is the **SIL** script to get versions:

```
string projectKey = "TEST";
string[] versionNames = admGetProjectVersions(projectKey);
number[] versionId;
for(string versionName in versionNames){
    JVersion version = admGetProjectVersion(projectKey, versionName);
    versionId += version.id;
}
return versionId;
```

It gets all the versions existing in the project "TEST" . The versions will be added as options to the custom field which uses this SIL script as data source.

This is the **SQL** script to get all versions.

```
select vname, id from projectversion;
```

You can see it in action here: KCF - Single Select

### Related Content

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

## SIL Data Source

### SIL Data Source

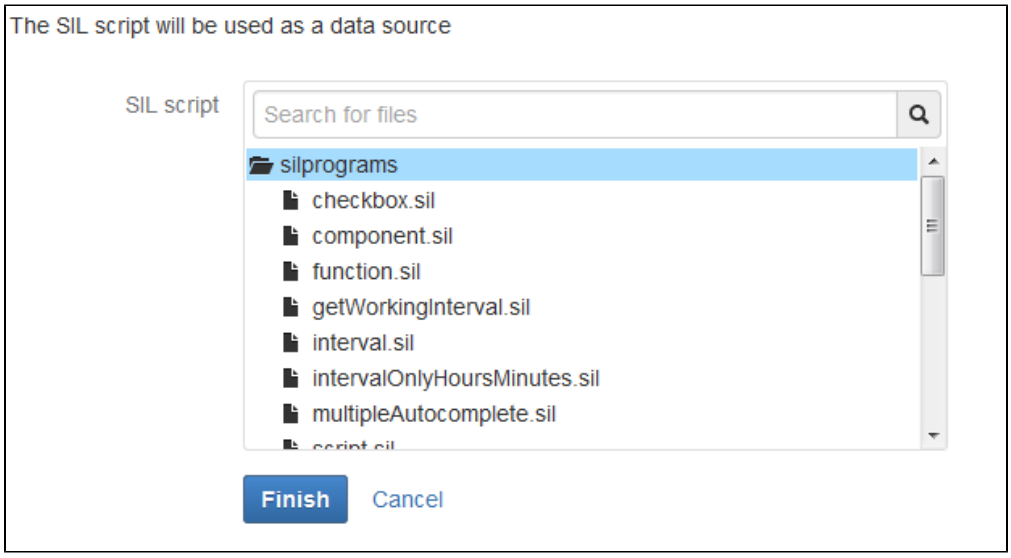
**Description**

This type of data source helps you obtain the values which will populate the custom field, using SIL scripts.

The purpose of SIL Data Source is to bring fast results on the screen for your custom fields, writing your own SIL scripts.

For this type of data source it is very helpful the sql routine . The script will bring you the results you need for your custom field.

After you have selected the SIL Data Source, you have to choose the SIL Script that you want to use from the list of silprograms .




You can find out more about using this type of data source from our Examples and Option Type pages.

### Related Content

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

# SQL Data Source

### SQL Data Source

**Description**

This type of data source helps you obtain the values which will populate the custom field, using SQL scripts.

The purpose of SQL Data Source is to bring fast results on the screen for your custom fields. Writing the **SQL Initial Script** for the custom fields with **no autocomplete** or with **autocomplete** and value **0** set for **Min. chars**, you can configure the details for the information that you need from database.

Also, for the custom fields with **autocomplete** , you have to configure your own **SQL Autocomplete Script**.

> **Important**
> If you want to use 2 columns at the **select** statement, the first column will be considered the label, and the second will be the value. If you have only one column at the select statement, this will be interpreted as label and value at the same time.

The **SQL Data Source** has the following fields :

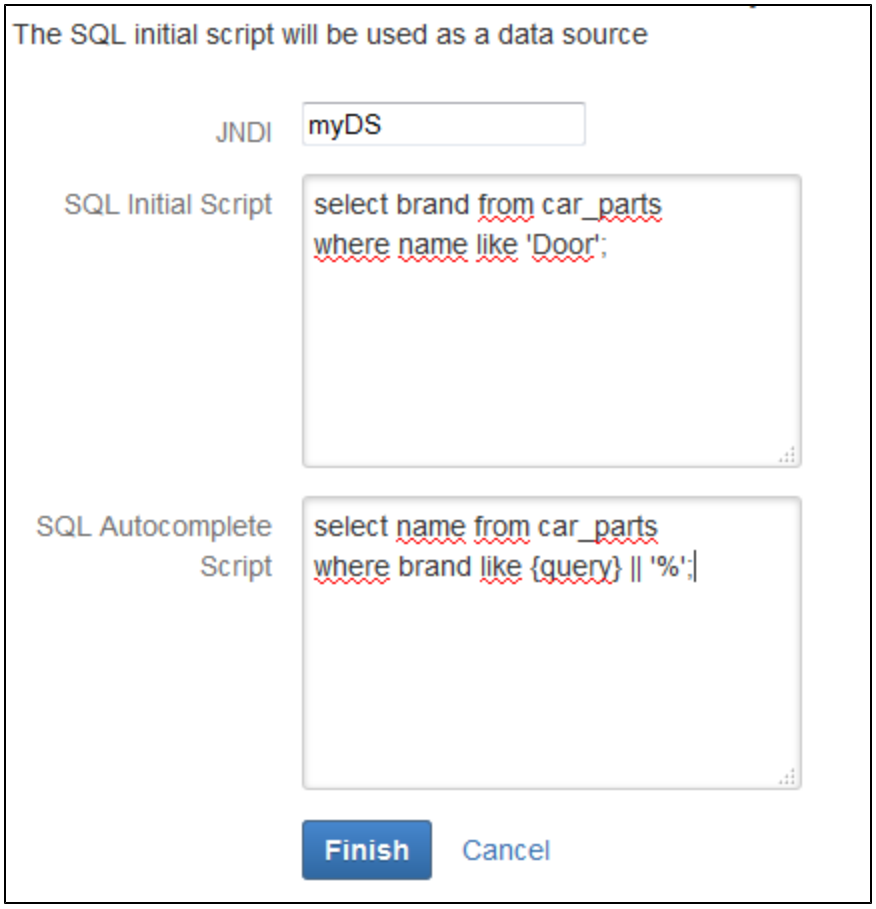
- **JNDI :** represents the resource name of database used. You have to type the name of the database used.

> Before setting the **JNDI**, configure it as described here.

- **SQL Initial Script :** represents the initial script which filters the result from database for custom fields with **no autocomplete** or with **auto complete** and value **0** set for **Min. chars.**
- **SQL Autocomplete Script :** represents the script made by user for custom fields with **autocomplete** .

After you have selected the SQL Data Source, you will see one of the 3 images below.

- If you have a custom field - with **autocomplete** and you want to set **Min. chars** on **0**, you need to write 2 sql scripts : **SQL Initial Script** an d **SQL Autocomplete Script**



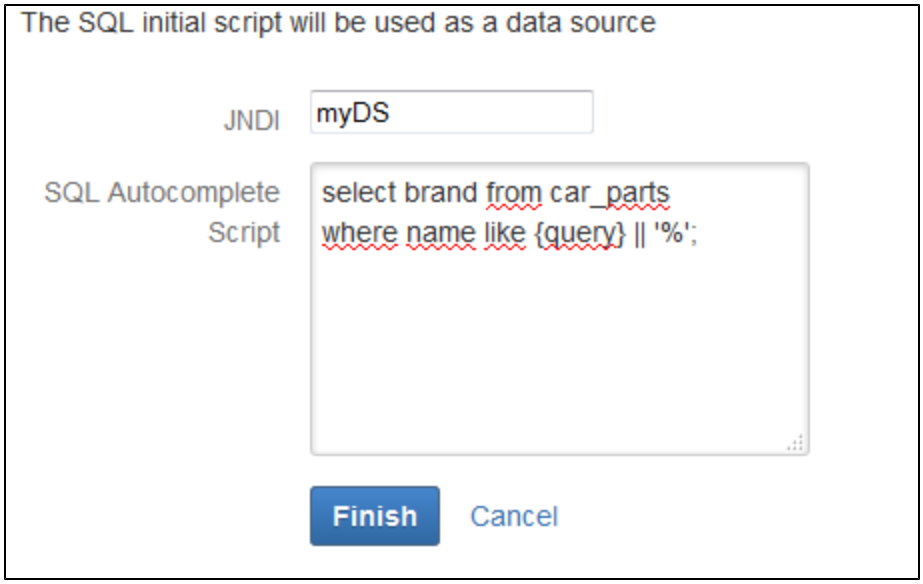

**SQL Initial Script**

```
select brand from car_parts
where name like 'Door';
```

**SQL Autocomplete Script**

```
select name from car_parts
where brand like {query} || '%';
```

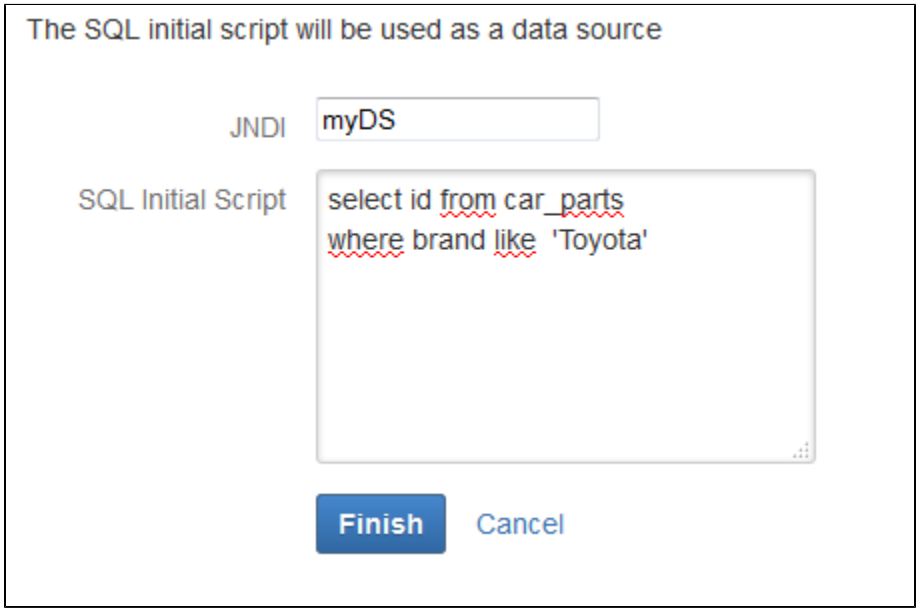
- with **autocomplete** and you want to set **Min. chars** on a value **grater than 0**, you need to type the **SQL Autocomplete Script**




**SQL Autocomplete Script**

```
select brand from car_parts
where name like {query} || '%';
```

- with **no autocomplete**, you need to type the **SQL Initial Script**



**SQL Initial Script**

```
select id from car_parts
where brand like 'Toyota';
```

**Note:** If you want to use "like"  operator in your sql scipts , you have to use the concatenation symbol between query and "%" symbol.

**Related Content**

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

## How to use SQL Datasource on several databases

On most of the databases, the way described before works just fine.

```
select name from TestTable
where name like {query} || '%'
```

However, for Oracle databases we have three special cases:

- In case of searching at the beginning of the text:

```
select name from TestTable
where name like {query%}
```

- In case of searching at the end of the text:

```
select name from TestTable
where name like {%query}
```

- In case of searching anywhere in the text:

```
select name from TestTable
where name like {%query%}
```

**Note**
We tested the above scripts using ojdbc6.jar.

It seems that there is a bug that addresses the issue where the LIKE operator causes a problem in the parser, existing in ojdbc7.jar, that makes the scripts unusable for Oracle.

We suggest to use as an workaround the concat operator:

- In case of searching at the beginning of the text:

```
select name from TestTable
where name like concat({query}, '%')
```

- In case of searching at the end of the text:

```
select name from TestTable
where name like concat('%', {query})
```

- In case of searching anywhere in the text:

```
select name from TestTable
where name like concat(concat('%', {query}), '%')
```

# User Guide

This documentation refers to Kepler Custom Fields PRO, version 1.0.

> Because we are using some JIRA dependencies which are present since JIRA 6.2, Kepler Custom Fields PRO, version 1.0 is supported from JIRA 6.2 and up.
> Step-by-step guides, previews, demo images and screenshots were made under JIRA 6.x.

*Table of Contents*



## Introduction

## KEPLER Custom Fields PRO

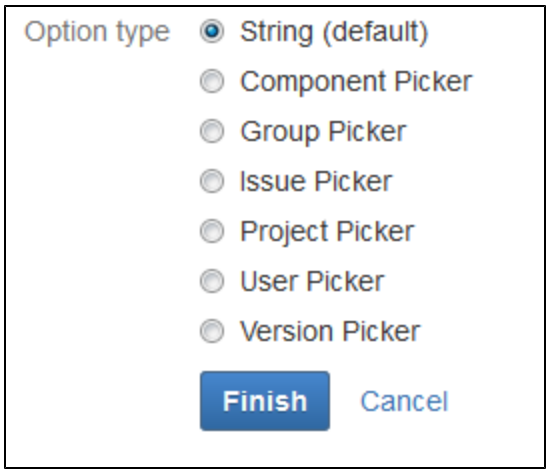
Using our new plugin, you can create 6 new types of custom fields, which can be configured as you wish. These will improve the user experience and will allow you to customize JIRA to better suit your needs.

The six types of custom fields are:

- KCF Checkbox
- KCF Radio Buttons
- KCF Single Select
- KCF Multi Select
- KCF Single Autocomplete
- KCF Multiple Autocomplete

Each of these custom fields can be configured to behave like virtually any type of pickers (issue, version, project and user among them) and if not enough the String option may offer you the together with the data source configuration (SQL or SIL scripts) provide you the means to select anything.

The complete list to date of selection options is:



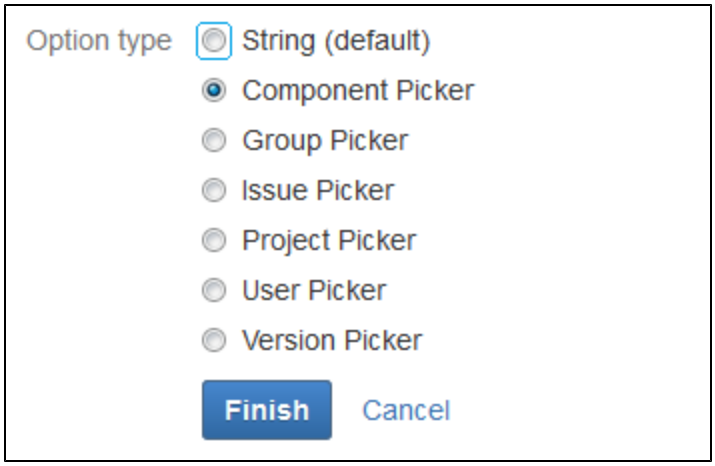

# KCF - Checkbox

## How to configure

Go to the **Administration** page, choose **Custom Fields** link and add an KCF - Checkbox.


In order to see how you can configure all the options above, check this: Configuration

This custom field can be used for many types of options:




If you go to an issue and try to edit this field, you will see an image like this:


Save the changes and the selected values should appear on the issue screen:


### Example for SIL Data Source

**Get Project Components**

```
string projectKey = "TEST";
string[] componentNames = admGetProjectComponents(projectKey);
number[] componentId;
for(string componentName in componentNames){
    JComponent componentNames = admGetProjectComponent(projectKey,
componentName);
    componentId += componentNames.id;
}
return componentId;
```

This example is used to generate the pictures above. It gets all the components existing in the project "TEST". The components will be added as options to the custom field which uses this script as data source.

### Example for SQL Data Source

> First of all, in order to use SQL Data Source, you have to set the **JNDI** as described here.

**SQL Initial Script**

```
select cname, id from component;
```

This example returns all component names.

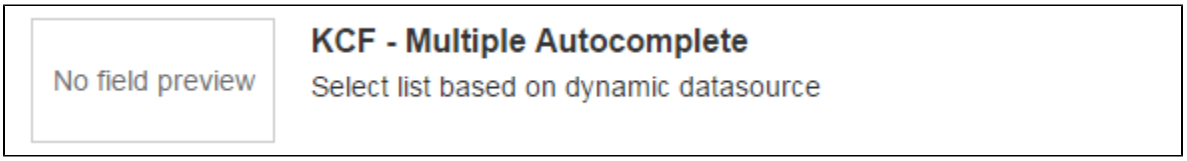
### Related Content

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

# KCF - Multiple Autocomplete

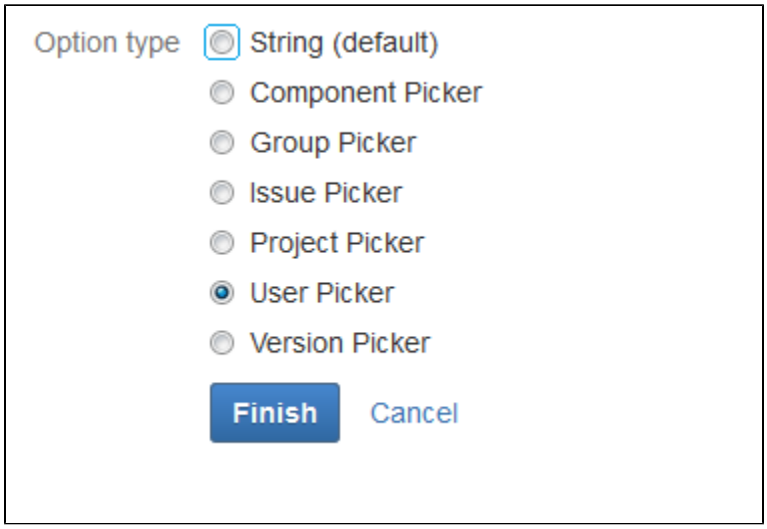
## How to configure

Go to the **Administration** page, choose **Custom Fields** link and add an KCF - Multiple Autocomplete.

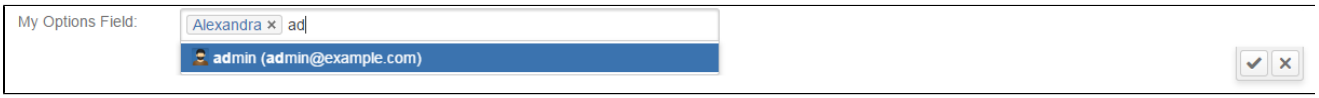



In order to see how you can configure all the options above, check this: Configuration

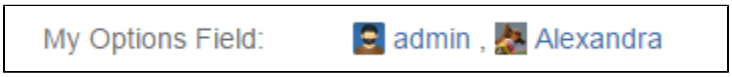
This custom field can be used for many types of options:



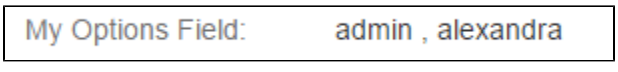

If you go to an issue and try to edit this field, you will see an image like this:




Save the changes and the values should appear on the issue screen:




If the option type you selected is "Default", the picture above will look like this:




### Example for SIL Data Source

**Get Users**

```
function getUsers(string [] groups){
  string [] users;
  for(string group in groups){
    string [] currentGrp;
    currentGrp = addElement(currentGrp, group);
    for(string user in usersInGroups(currentGrp)){
      users = addElementIfNotExist(users, usernameToUserKey(user));
    }
  }
  return users;
}


string [] groups = {"jira-users"};
string [] users = getUsers(groups);
string [] res;
for (string user in users) {
    if (contains(user, argv["query"])) {
     res = addElementIfNotExist(res, user);
    }
}
return res;
```

This example is used to generate the pictures above. It gets all users existing in some groups (in this case, "jira-users") and adds the usernames that match the search as options to the custom fields using this data source.

#### Example for SQL Data Source

> First of all, in order to use SQL Data Source, you have to set the **JNDI** as described here.

**SQL Autocomplete Script**

```
select user_name from cwd_user;
```

This example returns the user names of all users.

#### Related Content

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

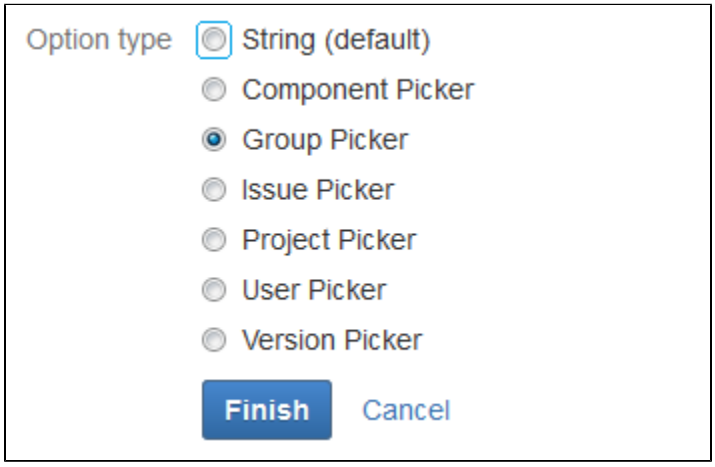
## KCF - Multi Select

### How to configure

Go to the **Administration** page, choose **Custom Fields** link and add an KCF - Multi Select.


In order to see how you can configure all the options above, check this: Configuration

This custom field can be used for many types of options:




If you go to an issue and try to edit this field, you will see an image like this:


Save the changes and the selected values should appear on the issue screen:


If the option type you selected is "Default", the picture above will look like this:


### Example for SIL Data Source

**Get Groups**

```
return userGroups(currentUsername());
```

This example is used to generate the pictures above.

### Example for SQL Data Source

First of all, in order to use SQL Data Source, you have to set the **JNDI** as described here.

**SQL Initial Script**

```
select group_name from cwd_group;
```

This example will return the list of group names.

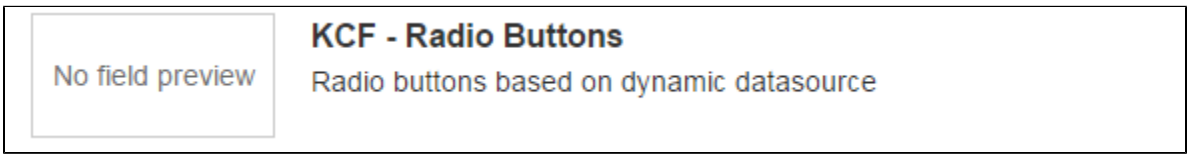
### Related Content

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

## KCF - Radio Buttons

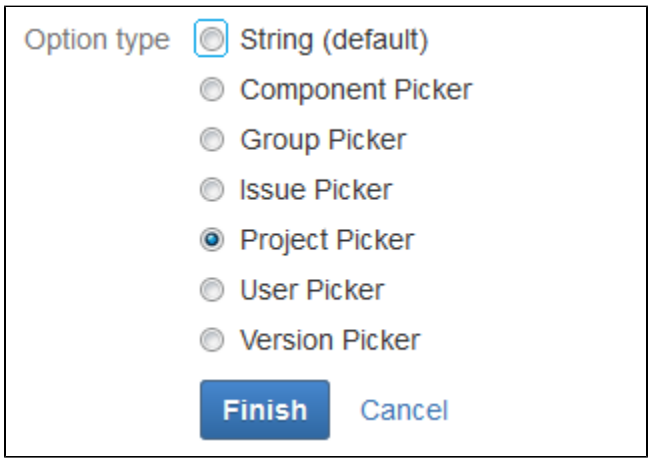
### How to configure

Go to the **Administration** page, choose **Custom Fields** link and add an KCF - Radio Buttons.

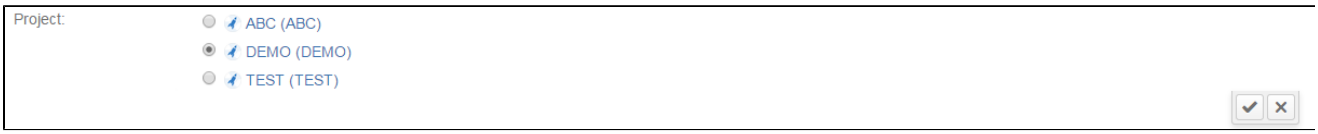



In order to see how you can configure all the options above, check this: Configuration

This custom field can be used for many types of options:

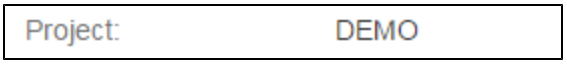



If you go to an issue and try to edit this field, you will see an image like this:




Save the changes and the selected value should appear on the issue screen:


If the option type you selected is "Default", the picture above will look like this:




### Example for SIL Data Source

**Get Projects**

```
string [] projects = allProjects();
return projects;
```

This example is used to generate the pictures above.

### Example for SQL Data Source

> First of all, in order to use SQL Data Source, you have to set the **JNDI** as described here.

**SQL Initial Script**

```
select pkey from project;
```

This example is also used to generate the pictures above. It returns the project keys from table "project".

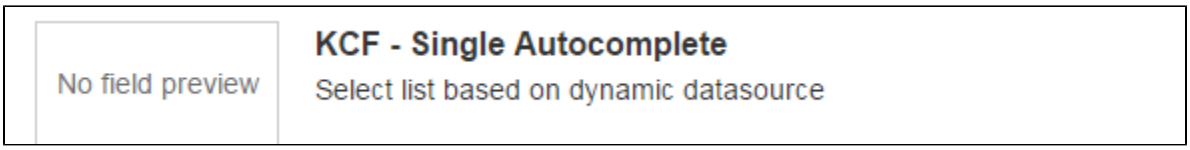
### Related Content

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

## KCF - Single Autocomplete

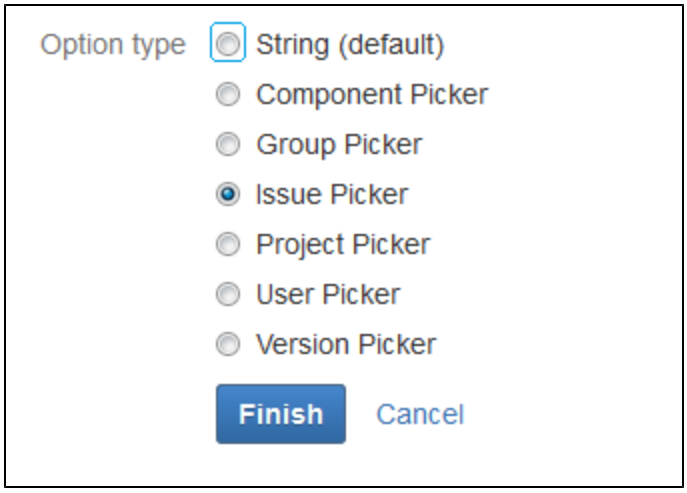
### How to configure

Go to the **Administration** page, choose **Custom Fields** link and add an KCF - Single Autocomplete.




In order to see how you can configure all the options above, check this: Configuration

This custom field can be used for many types of options:




If you go to an issue and try to edit this field, you will see an image like this:


Save the changes and the selected value should appear on the issue screen:


### Example for SIL Data Source

**Get Issues**

```
string [] issues = selectIssues("created < now()");
string [] res;
for (string iss in issues) {
    if (contains(iss.summary, argv["query"])) {
     res = addElementIfNotExist(res, iss);
    }
}
return res;
```

This example is used to generate the pictures above. It gets all the created issues with the summary matching the search.

### Example for SQL Data Source

> First of all, in order to use SQL Data Source, you have to set the **JNDI** as described here.

**SQL Autocomplete Script**

```
select  p.project_key ||'-'||i.issuenum as "KEY"
from jiraissue i
inner join project_key p on p.project_id = i.project
```

This example returns all the issue keys .

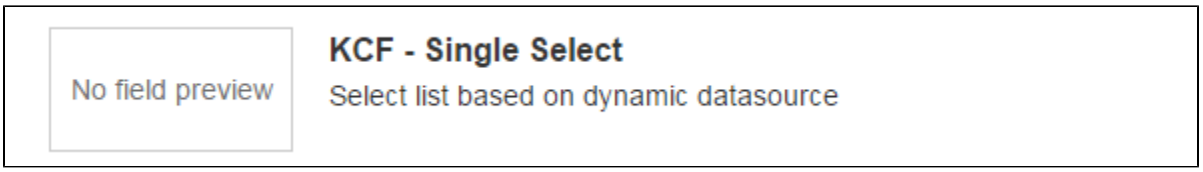
### Related Content

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

## KCF - Single Select

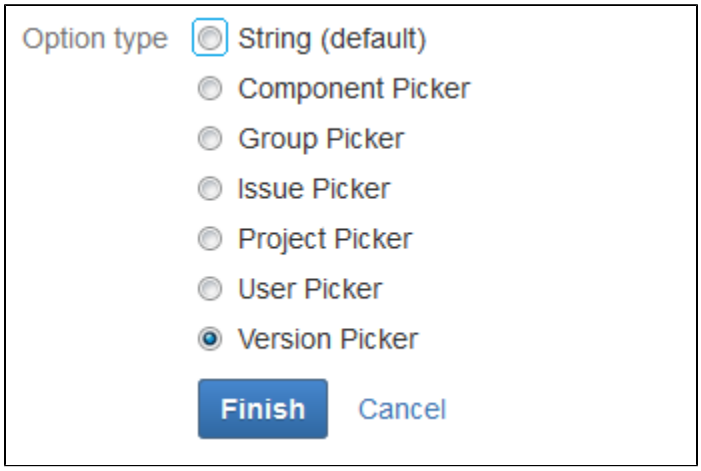
### How to configure

Go to the **Administration** page, choose **Custom Fields** link and add an KCF - Single Select.

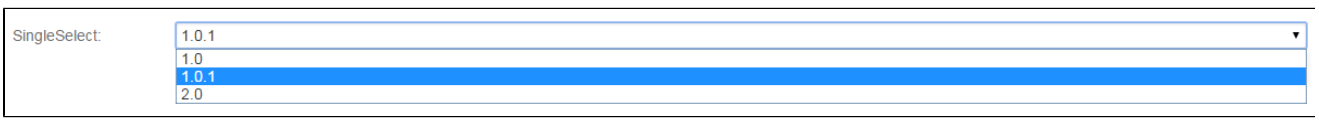



In order to see how you can configure all the options above, check this: Configuration

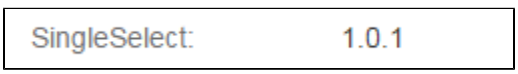
This custom field can be used for many types of options:




If you go to an issue and try to edit this field, you will see an image like this:




Save the changes and the selected value should appear on the issue screen:




### Example for SIL Data Source

**Get Project Versions**

```
string projectKey = "TEST";
string[] versionNames = admGetProjectVersions(projectKey);
number[] versionId;
for(string versionName in versionNames){
    JVersion version = admGetProjectVersion(projectKey, versionName);
    versionId += version.id;
}
return versionId;
```

This example is used to generate the pictures above.

#### Example for SQL Data Source

> First of all, in order to use SQL Data Source, you have to set the **JNDI** as described here.

**SQL Initial Script**

```
select vname, id from projectversion;
```

This example returns the versions names.

#### Related Content

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

## Predefined Structure Types

Here is a list of predefined structures that are used throughout Kepler Custom Field - PRO.

| Name | Field | Type |
|---|---|---|
| ***KPOption*** | label | string |
| | value | string |

## JJUPIN integration

- Scripting support
- Live Fields Support

Of course, our solutions are tightly integrated. The following pages help you if you already have JJUPIN installed.

We pushed a lot of effort in offering full support in JJUPIN for KCF-PRO. For any given KCF-PRO field, you will be able to exploit it as a normal JIRA customfield, using the same syntax and the same JJUPIN functions.

### Scripting support

As usual with our plugins, the scripting support is natural. Once you install the KCF PRO plugin, you can access the custom fields via the usual syntax: *customfield_<id>*.

The value of the field is always an array of **KPOption** (see: Predefined Structure Types) representing the selected options in the field.

Example:

```
//simple example on how to access (get/set) KCF-PRO fields

//we assume we have an issue context here
KPOption [] options = customfield_12345;

if(size(options) > 0) {
    runnerLog("First option id = " + options[0].value + " and the label is
:" + options[0].label);
}

//we'll assume this is a multi-select field, because otherwise it will not
work
KPOption nopt;
nopt.label = "Extra topping";
nopt.value = "99";
options += nopt;

customfield_12345 = options; //here's the set
```

## Live Fields Support

> Live Fields support for these custom fields is available since JJUPIN 3.0.6.

You can also use the Live Fields feature of JJUPIN with our new custom fields.

The routines you can apply to a custom field from Kepler Custom Field PRO plugin are:

- lfHide
- lfShow
- lfDisable
- lfEnable
- lfSet
- lfWatch
- lfAllowSelectOptions
- lfRestrictSelectOptions

> **For the technical minded**
> If you are using the last three routines on a different option type than "default", you should keep in mind that you must provide for each option type, the corresponding property from the table below:

| Option Type | Property |
| --- | --- |

| | |
|---|---|
| Component | component name |
| Group | group name |
| Issue | issue key |
| Project | project key |
| User | user key |
| Version | version name |

## Examples

Here you can find examples of Kepler Custom Field usage.

**Examples:**



### Example 1 - Huge Data Sets

#### Huge data sets

Let's consider we are working with some **huge data sets**. For example, we own a store and we sell car parts. Obviously, in our data base are thousands of products. If we would had to go through them all to select one, it wouldn't be nice. Our newest plugin, **KCF Pro**, allows you to intervene in the process of data build-up even in the stages of the autocomplete. You can see this in the picture below:

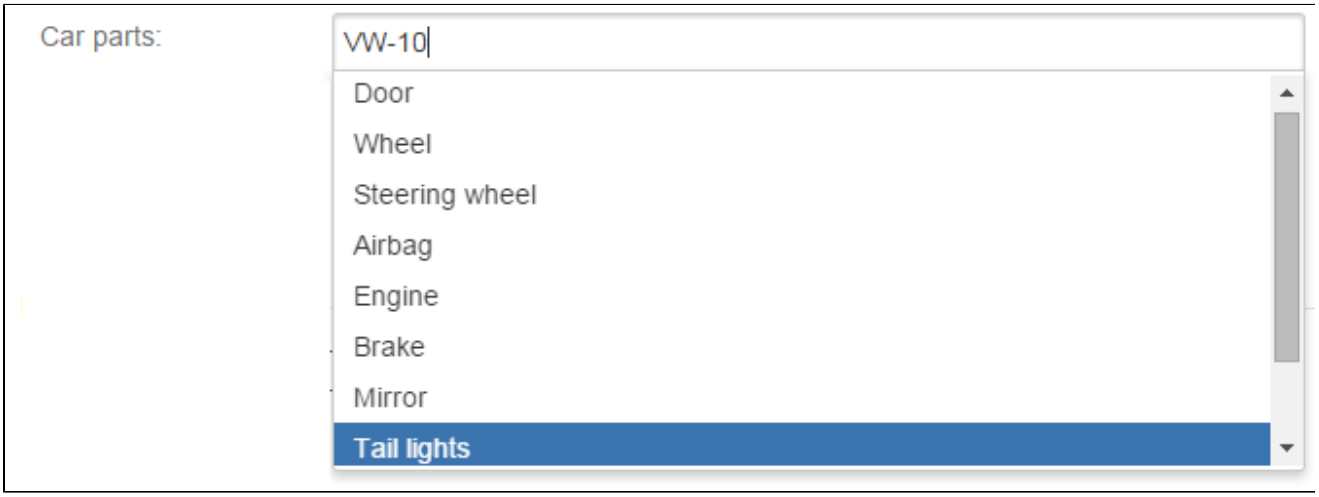

Considering we have a KCF - Multi select configured with one SIL Data Source, one Selection script, minimum characters for autocomplete set to 5, the maximum results number set to 50 and the filtering strategy set to "DataSource". We want to select from our big data set the items using the ids they have.

The **SIL data source** used:

```
string s = argv["query"];
KPOption [] res;
if(length(s) > 2) {
    KPOption[] options = sql("myDB", "select name, id from car_parts where
id like '%" + s + "%' limit 100" );
    for (int r = 0; r < size(options); r++) {
        if (contains(options[r].value, s)) {
          res = addElementIfNotExist(res, options[r]);
        }
    }
 }
 return res;
```

In the script above, "myDB" represents the resource name of the data base used, "car_parts" represent the table from the data base we want to select from and ("name", "id")  are the columns, converted to an object of type KPOption and represented by a pair (label, value).

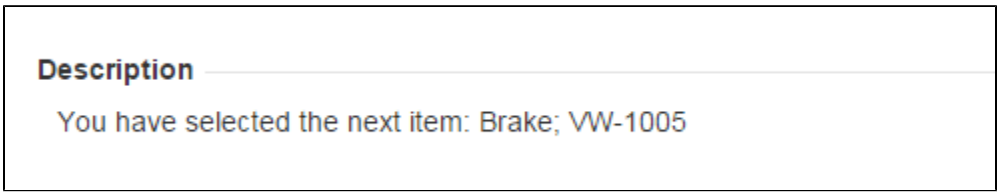
If you use this script as data source, your custom field will be populated like this:




We can use a **select script** to do some stuff when an item from the autocompleted list is selected. The example used is:

```
KPOption[] selectedOptions = customfield_12101;
description += "You have selected the next item: " +
selectedOptions[size(selectedOptions)-1].label +
              "; " + selectedOptions[size(selectedOptions)-1].value + "\n
";
```

The example above sets the description like this:



***Related Content***

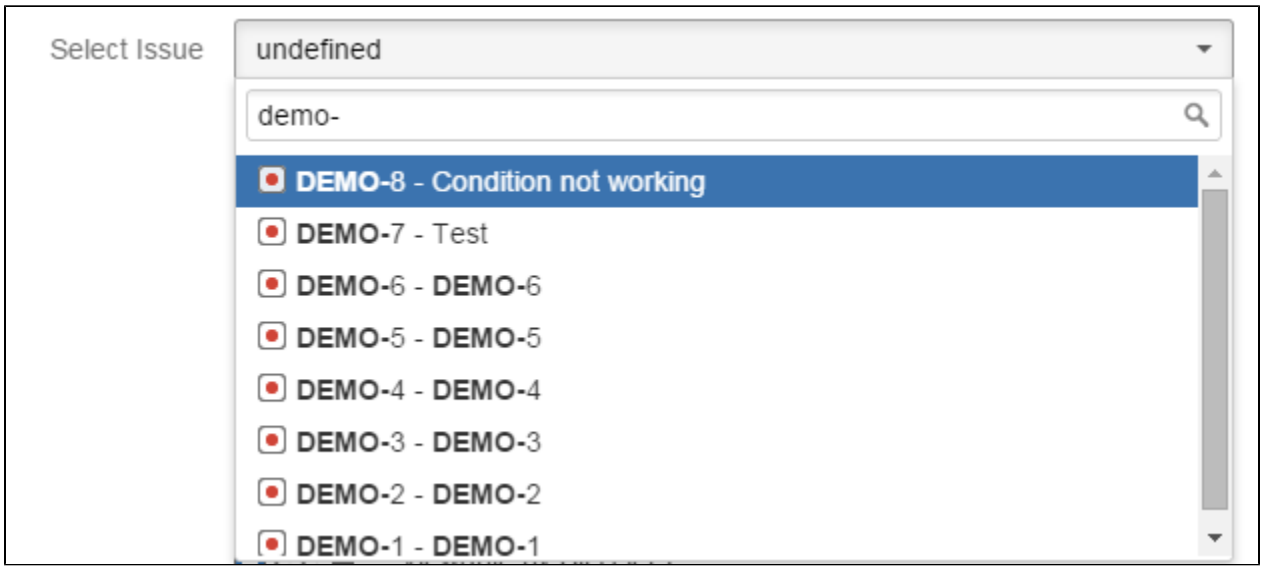
Error formatting macro: contentbylabel: com.atlassian.confluence.api.service.exceptions.BadRequestException: Could not parse cql : null

## Example 2 - Issue Links

### Issue Links

Let's consider the next example: we want to link the current issue to an issue selected. Using **KCF PRO** you can do this very simple.

Considering we have a KCF - Single select configured with one SIL Data Source, one Selection script, minimum characters for autocomplete set to 5 and the maximum results number set to 50. We want to select an issue using its key or summary. The custom field looks like this:



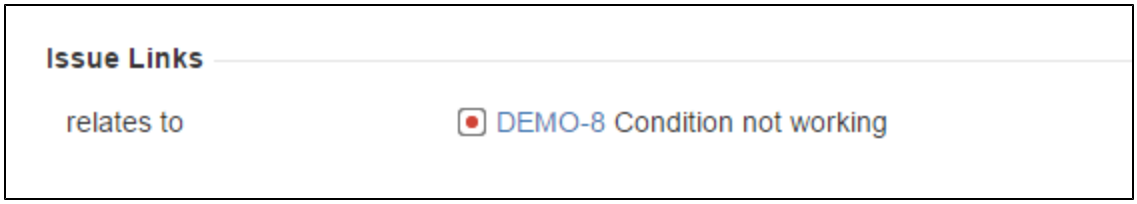

The **SIL data source** used to populate the custom field above:

```
string [] issues = selectIssues("created < now()");
string [] res;
for (string iss in issues) {
    if (contains(iss.summary, argv["query"])) {
     res = addElementIfNotExist(res, iss);
    }
}
return res;
```

The **select script** used is:

```
KPOption[] selectedOption = customfield_12209;
linkIssue(key, selectedOption[0].value, "Relates");
```

It creates a link of type "Relates" between the current issue and the selected issue.




Please notice that you'll have to use the name of the link issue, as you find it in Issue linking page (in our case, "Relates" and not "relates to").

***Related Content***

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

## Example 3 - SQL Data Source

### SQL Data Source

The **SQL Data Source** has the following fields :

- **JNDI :** represents the resource name of database used. You have to type the name of the database used.

Before setting the **JNDI**, configure it as described here.

- **SQL Initial Script :** represents the initial script which filters the result from database for custom fields with **no autocomplete** or with **auto complete** and value **0** set for **Min. chars.**
- **SQL Autocomplete Script :** represents the script made by user for custom fields with **autocomplete** .

All the next 3 examples works for huge data sets.

We own a store and we sell car parts. Obviously, in our data base are thousands of products. If we would had to go through them all to select one, it wouldn't be nice. Our newest plugin, **KCF Pro**, allows you to intervene in the process of data build-up even in the stages of the autocomplete.

***Example A***

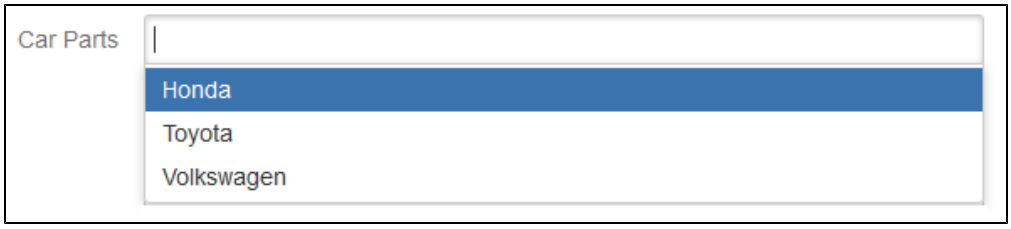
**Description**:

- Type Custom Field : KCF - Multiple Autocomplete
- Datasource : SQL Data Source
- Min. chars : 0

The **SQL Initial Script** :

```
select brand from car_parts
where name like 'Door';
```

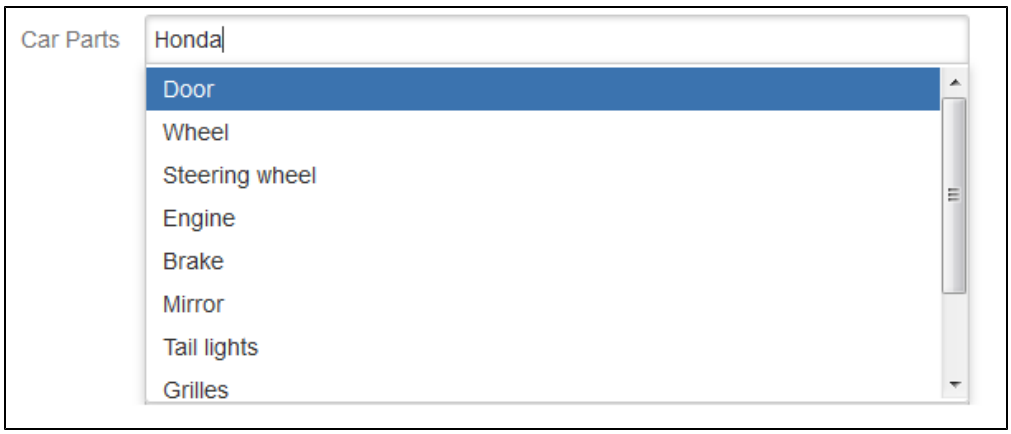
The script above returns a result as you can see to the image below:




The **SQL Autocomplete Script** :

```
select name from car_parts
where brand like {query} || '%';
```

The script for autocomplete will return the following results:




### *Example B*

**Description**:

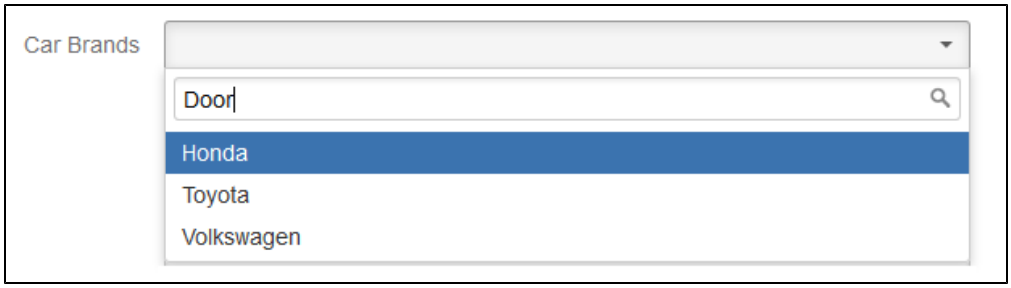
- Type Custom Field : KCF - Single Autocomplete
- Datasource : SQL Data Source
- Min. chars : 3

**Note:** Set the **JNDI** as described here .

The **SQL Autocomplete Script** :

```
select brand from car_parts
where name like {query} || '%';
```

The script for autocomplete will return the following results:




### *Example C*

**Description**:

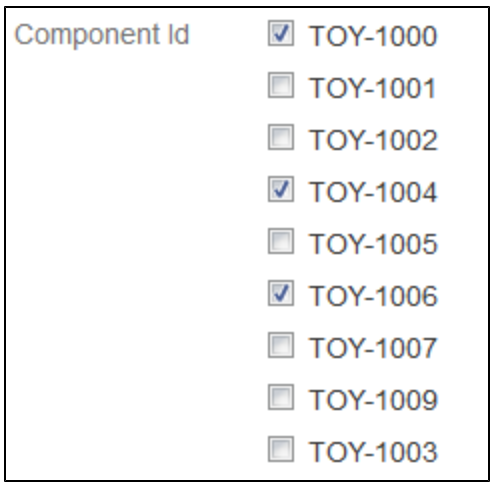
- Type Custom Field : KCF - Checkbox
- Datasource : SQL Data Source

**Note:** Set the **JNDI** as described here .

The **SQL Initial Script** :

```
select id from car_parts
where brand like  'Toyota'
```

The script will return the following results:



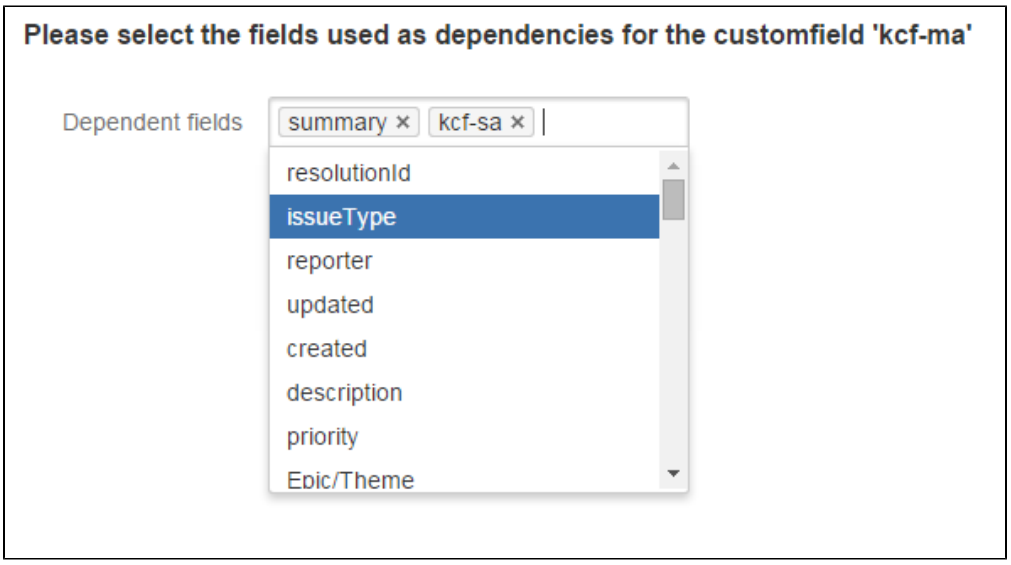

### *Related Content*

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

## Example 4 - Dependent custom fields

In order to use dependent fields you must have **JJUPIN** plugin installed and enabled.

In order to use the screen values for a field in the data source, you have to add it as dependent field.




**Example:** Let's suppose we have two custom fields: one KCF - Single Autocomplete and one KCF - Multiple Autocomplete, the second one is populated depending on the value of the first one.

An easy example would be: the first custom field has five options, representing the five continents and the second one contains the countries from some table ("countries"), depending on the value selected on the first one.

You can see the corresponding data sources below:

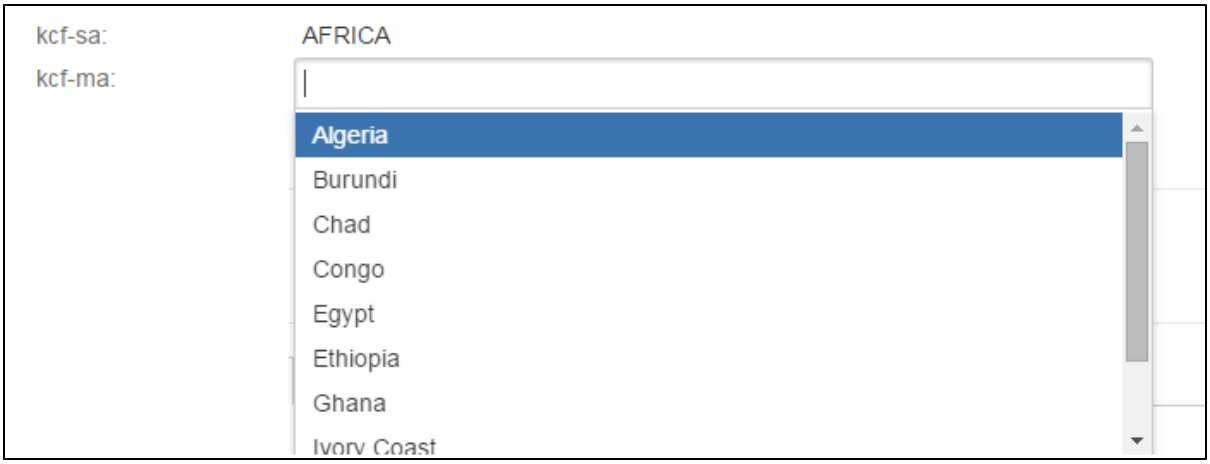
The data source for the first custom field:

```
return {"AFRICA", "ASIA", "EUROPE", "AMERICA", "OCEANIA"};
```

The data source for the second custom field:

```
string x = argv["customfield_10101"];
string[] res = sql("myDB", "select country from countries where continent
like '" + x + "'");
return res;
```

In the issue view screen, the result will look like this:



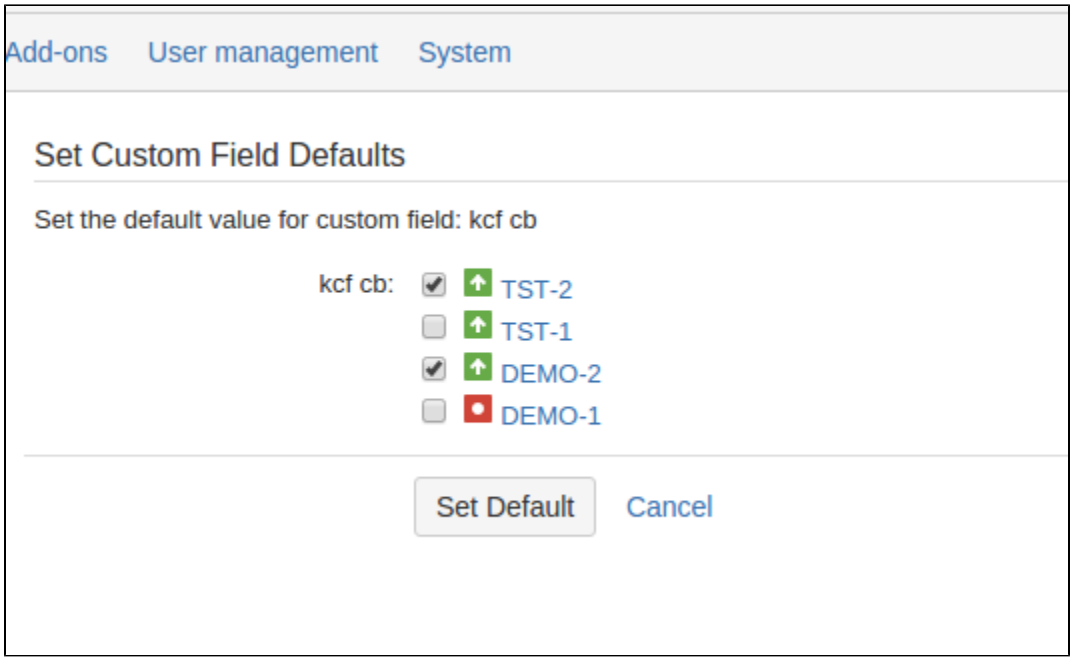

A similar example can be found here.

> For the moment, dependent fields can only be used by the two custom fields with autocomplete, and can only be referred in SIL data sources.
>
> In order to get the value set on the screen for one custom field, you will have to refer it like this: "**argv[customfield_id]**"

## Example 5: Setting default values for KCF Pro fields

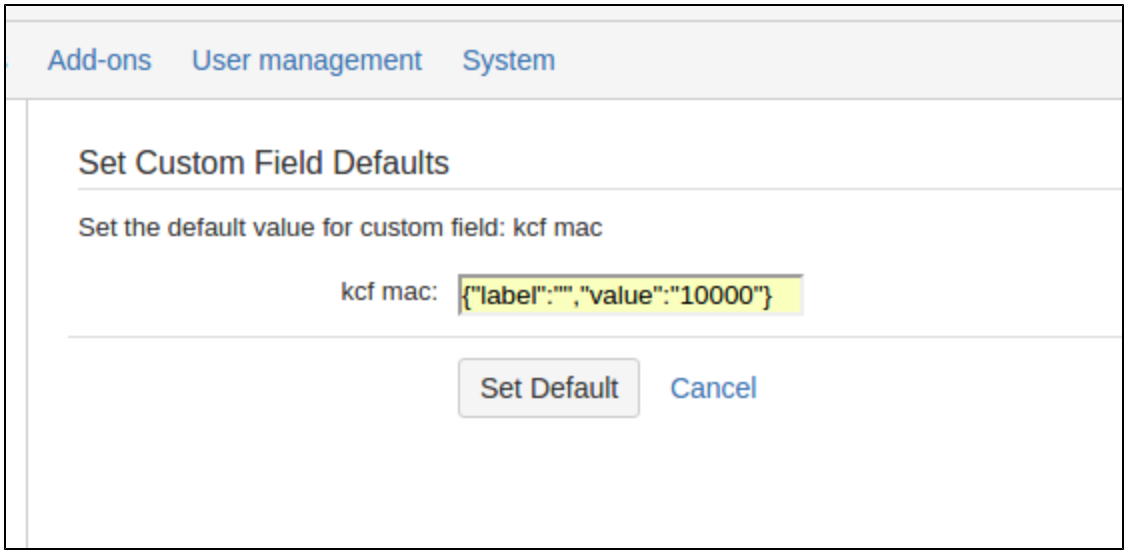
Setting a default value for one custom field without autocomplete (checkbox, radio buttons, single or multi select) is very simple. All you have to do is to check/select the desired option(s).

Example:




Setting the values for the select custom fields with autocomplete is not that easy, since the options are of type KPOption, so they need to have defined a label and an options.

The configuration for such a field would be like follows:




In the example above, custom field is rendered as an issue picker and the value is the id of the issue I want to select.

After the value will  be set to default, the configuration will be updated by itself according to the renderer. In our case, it became:

```
{"option":{"label":"","value":"10000"},"meta":{"longLabel":"TST-1 - update
the documentation","tooltip":"update the
documentation","shortLabel":"TST-1","iconUrl":"/secure/viewavatar?size=xsm
all&avatarId=10310&avatarType=issuetype","href":"http://localhost:8070/bro
wse/TST-1","labelOverride":"TST-1"}}
```

### Example 6 - (Video) - More Dependent Fields

In this second example  we have taken the options for the Continent custom field from the database as well. The script used in this example are:

**Continent data source**

```
string[] continents = sql("myDB", "select distinct continent from
countries");
return continents;
```

**Country data source**

```
string continent = argv["customfield_10109"]; //the dependent field
string[] countries = sql("myDB", "select country from countries where
continent = '" + continent + "'");
return countries;
```

For the moment, dependent fields can only be used by the two custom fields with autocomplete, and can only be referred in SIL data sources.

In order to get the value set on the screen for one custom field, you will have to refer it like this: "**argv[customfield_id]**"

## Example 7- (Video) - How to use argv["query"]

As we have said before, KCF Pro can be used for huge data sets. Still, showing all the options, without being able to select them by name can be a torture. So, you can restrict the values by name, selecting the ones which contains those characters.

> In order to do this, you can use argv["query"] only if you set the **Filtering Strategy** (from Autocomplete settings) to **Datasource**.

The example below shows us how to do this:

The script used in the video is:

```
string [] issues = selectIssues("created < now()");
string [] res;
for (string iss in issues) {
    if (contains(iss.summary, argv["query"])) {
        res = addElementIfNotExist(res, iss);
    }
}
return res;
```