

Project Configurator Integration Toolkit

Developer Guide



© 2020 Adaptavist

Project Configurator Integration Toolkit

Developer Guide

Table of Contents

| | | |
|----------|---|----------|
| 1 | <i>Introduction</i> | 3 |
| 1.1 | Intended Audience | 3 |
| 1.2 | How this Guide is Organised | 3 |
| 2 | <i>Why Create a PC Extension</i> | 4 |
| 2.1 | About PC | 4 |
| 2.2 | Benefits of Creating an Extension | 4 |
| 3 | <i>Types of PC Extensions</i> | 6 |
| 3.1 | Workflows | 6 |
| 3.2 | Dashboard Gadgets | 6 |
| 3.3 | New Custom Entities | 7 |
| 4 | <i>Common Features of Extensions</i> | 8 |
| 4.1 | How to Implement Extensions | 8 |
| 4.2 | How to Identify a PC Extension | 9 |
| 4.3 | Version Compatibility | 9 |
| 4.4 | Maven Dependencies | 11 |
| 4.5 | Importing Packages | 12 |
| 4.6 | The Hollywood Principle | 14 |
| 4.7 | Welcome Lazy Developers! | 15 |

| | | |
|-----|---|----|
| 4.8 | How to Create Objects in an Extension | 15 |
| 4.9 | How to Inject PC Dependencies into Your Extension | 17 |
| 5 | <i>Workflow Extensions</i> | 18 |
| 6 | <i>Dashboard Gadget Extensions</i> | 31 |
| 7 | <i>Custom Entity Extensions</i> | 40 |
| 7.1 | Have a Logical Model of the Information you Want to Support | 40 |
| 7.2 | GlobalCustomEntity | 42 |
| | Names for the Custom Entity | 44 |
| 7.3 | Properties | 46 |
| 7.4 | Identifiers | 51 |
| | Creating identifiers from properties | 54 |
| 7.5 | ChildCustomEntity | 55 |
| 7.6 | Child Collections | 57 |
| 7.7 | ExportTriggerProperty | 59 |
| 8 | <i>Hints for Testing PC Extensions</i> | 61 |
| 8.1 | Test the Actual Migration | 61 |
| 8.2 | Use the Object Dependencies Report | 67 |
| 8.3 | Pro Tip: Create the Extension in Two Stages | 68 |
| 8.4 | Use the Jira Log if Necessary | 68 |

1 Introduction

1.1 Intended Audience

This guide is primarily aimed at developers who want to create or maintain extensions for Project Configurator (PC). The introductory sections may also be useful for product managers who need to decide if and when to create these extensions.

Assumptions

If you are a developer reading this guide, it is assumed that you:

- are reasonably familiar with Java
- have some knowledge of how to create a P2 plugin for Jira

Previous experience with the problems that PC addresses (moving Jira configuration changes or projects between instances) or with PC itself, though beneficial, is not required. In the following sections we explain these problems and how PC works in resolving them.

1.2 How this Guide is Organised

Section 2 is an overview of how PC solves the problem of transferring configuration changes or projects between Jira instances. It also explains how writing an extension for PC will extend those benefits to the configuration and data of other apps. While this is a functional reference, it can also be useful for product managers, consultants, or Jira Administrators.

Section 3 explains what types of extensions are available. This section is not solely for developers; anyone involved with configuring Jira will find it useful.

The remaining sections are more technical and written specifically for developers who will work to integrate other apps with PC:

- Section 4 discusses aspects common to any type of PC extension.
- Sections 5, 6, and 7 discuss workflow, dashboard gadget, and object type extensions respectively. Each contains an integration example with a real app. These sections are complemented by the Javadoc for the "extension-spi" module, however it is best to start reading these three sections and use the Javadoc as a reference.
- Section 8 discusses possible approaches when testing extensions.

We recommend that developers read at least sections 4 through 8, (skipping sections 6 or 7 if not interested in that type of extension). Sections 5, 6, and 7 can also be used as reference material in parallel with the Javadocs while coding the first PC extension.

2 Why Create a PC Extension

2.1 About PC

PC is an app for Jira that automates the process of manually copying projects and configurations from one Jira instance to another. It allows users to specify whether to export and import only specific project configurations or complete projects (configuration and data).

PC provides full migration support giving users the tools to:

- Move the configuration and/or data of projects between Jira instances
- Visualise what configuration objects will collide or cause errors when moving projects between instance
- Simulate an import to assess potential impact changes
- Merge projects from multiple smaller instances to a larger instance
- Plan and execute the splitting and organising of one massive Jira instance into smaller ones
- Enforce best practice in making changes in a staging instance before copying the changes to production

PC is able to handle not only the project-specific configurations but also global objects such as custom fields, schemes, and workflows that are referenced in the project configuration. Other global objects that are not, strictly speaking, required for a project configuration such as filters, dashboards, and agile boards can also be migrated.

2.2. Benefits of Creating an Extension

Because apps are a large and integral part of most Jira installations, it would be extremely useful if Jira Administrators could enjoy the same benefits that PC offers for built-in objects when they are moving configurations or data that belong to the main apps they use in Jira.

For example:

- They might configure an app in a staging environment and then automatically move those changes to production.
- They could have a group of projects which heavily depend on an app in a departmental Jira and then consolidate those projects transparently into a corporate instance of Jira.
- They could find out which projects, workflows, or custom fields are using specific configuration items defined by the app.

All this can be achieved if a PC extension is created to support the app.

Benefits for the Extension Developer

Moving configuration and data to a different Jira instance requires solving several practical problems:

- How to express the content and structure of configuration and data in a way that can be transferred, especially relationships between different objects
- How to map IDs and other instance-specific content to their corresponding values in other instances
- How to collect all items that are required by the projects the user wants to move, but not others that are not related
- What to do when the same items exist in the destination instance
- How to sequence changes in the destination to comply with dependencies among objects
- How to handle and report the myriad of possible errors and warnings

The SPI that PC extensions should implement is designed in a declarative way. As an extension developer, you do not need to worry about all those problems; you only have to express the structure of the information your app manages and the simple operations to create, update, or remove each item. Let PC handle the rest of the export and import problems for you.

3 Types of PC Extensions

PC can be extended in three different ways:

3.1 Workflows

A Jira workflow may have conditions, validators, or post-functions that refer to other Jira objects, such as:

- Offer this transition only to members of group X (condition)
- Fail this transition if the user has not supplied a value for field Y (validator)
- After the transition, update field Z with value V in issues which are linked to this one by link type L (post function)

These workflow features create two challenges when moving the workflow to a different instance:

1. Often the workflow refers (internally) to fields Y, Z or link type L by their IDs. This means these references must be translated to the equivalent IDs that are valid for the corresponding objects at the destination instance.
2. Any of these objects becomes a required part of the new configuration. It must be ensured that their description is extracted from the source instance and that they will be created or updated at the destination.

PC already supports and handles conditions, validators, and post-functions defined in Jira "out of the box" plus the most popular workflow apps at the Atlassian Marketplace, as explained in [Specific Information for Some Object Types](#) (see *Conditions, Validators and Post-functions*).

If it is necessary to support other workflow apps, then it's time to create a PC extension for workflows. This is quite simple; you only have to specify where the references occur and what their content is (perhaps the ID of a field or the name of a group) and PC will take care of the rest. See section 5 for a detailed explanation of how to create a workflow extension for PC.

3.2 Dashboard Gadgets

When users configure dashboards in Jira, they will add gadgets to it. Each of these gadgets provides some functionality to the dashboard, for example, displaying all issues returned by a filter or statistics for a given category of projects. The implication is that gadgets also contain references to other objects in Jira (filters, fields, issue types, categories, projects, etc) and, as outlined in *Workflows*, these references must be handled appropriately during export and import.

These references are created when the user configures the gadget. For each gadget type, the users create a set of user preferences where they specify, for example, from which project or filter the issues should be taken to create a graph, or by which fields to group a given statistic.

PC supports gadgets defined by Jira "out of the box". If you want to migrate gadgets defined in a third-party app to other Jira instances, then you should use a PC extension for gadgets. The approach used to create these extensions is similar to that used workflow extensions. You only have to specify where the references occur (which type of gadget and in which user preference) and their content.

Section 6 provides an in-depth explanation, with examples, of how to create PC extensions for gadgets.

3.3 New Custom Entities

Apps often define new types of *entities* that do not already exist in Jira, for example, ScriptRunner for Jira offers many convenient features including *Behaviours*. Behaviours let you customise the behaviour of fields in the user interface.

A ScriptRunner behaviour is not a custom field, screen, or workflow; it is a completely new type of object that does not exist in Jira out of the box, and only appears if ScriptRunner for Jira is installed in your instance.

In the same way, a test management app might introduce new types of objects like test cases, test plans, or test runs. For administrators that manage different instances of Jira, the objects belonging to new types are part of the content they would like to migrate when they are moving configuration or data between instances. Administrators will want to migrate the configuration of behaviours or their test management data to another instance and, if at all possible, without lots of manual work or having to develop ad-hoc scripts.

Migrating these objects can become more challenging for different reasons:

- The structure of objects can be complex. In most cases, these objects have a number of relationships between them. For example, a test case may be part of a test plan which, in turn, is used in a test run. A behaviour may reference a script and have some specific field configurations or mappings to projects.

- This complexity extends beyond the boundaries of the app because there will be also relationships to/from objects defined by Jira itself. For example, each field configuration of a behaviour applies to a field in Jira, a project uses some behaviours, or test cases derive from specific Jira issues.

In cases like these, it is possible to leverage the power of PC to solve this complex problem if using a *custom entity* extension. These extensions make custom entities, like those described in the examples above, available to PC.

With the right extension, PC is able to export and import those items, managing all their links with other objects in the app or in Jira. Even if this problem is complex, the design of the SPI shields you from most of this complexity. As an SPI developer, you only need to implement the interfaces that describe the structure of the entities they want to handle, in terms of how they are identified, and their properties (including references to other objects). With this information PC can handle all the dependencies and perform all export/import operations required for a successful migration.

Section 7 explains how to write a custom entity extension for PC walking through all the relevant aspects of the SPI.

4 Common Features of Extensions

In this section we address the technical aspects that are common to all three types of extensions: workflow, dashboard, and custom entity.

4.1 How to Implement Extensions

All the extensions are Java code to be deployed in Jira as a P2 Jira plugin (app). The extension can be an additional feature in an existing app or a standalone app. For example, vendor of an app called "X" might develop a PC extension for the app as an added feature that either goes into the same *.jar*, or into a separate *.jar* (called "X extension for Project Configurator") and then distributes it to the Marketplace alongside app "X". The same P2 app can contain an arbitrary combination of workflow, dashboard, or custom entity extensions.

The first option has one technical advantage: it is easier to access and manipulate objects created by app X from the same app, than from a separate one. In this second case, app X would need to export the packages and classes that provide that access/manipulation functionality, and app "X extension for Project Configurator" would have to import and use them. The first option also offers a better, more frictionless user experience. End users can see that by installing compatible versions of X and PC, migration simply works without having to do anything else. On the other hand, if the extension is delivered in a separate app, the end user would have to install it (in addition to PC and X) in order to enable migration of content from X.

In favour of a separate app, if app X is already quite large in terms of project size, you may feel wary of adding a new feature to it.

4.2 How to Identify a PC Extension

All PC extensions will be identified by the key of the P2 app to which they belong.

4.3 Version Compatibility

All PC extensions must declare which version of PC they are compatible with. This is done by adding a parameter to the app's *atlassian-plugin.xml* file, as shown below:

atlassian-plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<atlassian-plugin key="${atlassian.plugin.key}" name="${project.name}" plugins-version="2">
  <plugin-info>
    <description>${project.description}</description>
    <version>${project.version}</version>
    <vendor name="${project.organization.name}" url="${project.organization.url}"/>
    <param name="plugin-icon">images/pluginIcon.png</param>
    <param name="plugin-logo">images/pluginLogo.png</param>
    <param name="min-pc-version-supported">3.2.0</param>
  </plugin-info>
  <!-- add our i18n resource -->
  <resource type="i18n" name="i18n" location="sample-extension-one"/>
</atlassian-plugin>
```

```
<!-- add our web resources -->
<web-resource key="sample-extension-one-resources" name="sample-extension-one Web Resources">
  <dependency>com.atlassian.auiplugin:ajs</dependency>
  <resource type="download" name="images/" location="/images"/>
  <context>sample-extension-one</context>
</web-resource>

</atlassian-plugin>
```

Note the *min-pc-version-supported* parameter. This will be interpreted as the extension in this app is compatible with PC version 3.1.11 or newer. Version strings are compared as defined in the [Semantic Versioning 2.0.0](#) specification.

This parameter is used to filter apps that contain PC extensions from those that do not. So, if this parameter is not added to the *atlassian-plugin.xml* file, this app will not be treated as a valid extension by PC.

4.4 Maven Dependencies

Your app will need access to the *extension-spi* jar to use all the interfaces and classes described in this document. Add this dependency to your `pom.xml` file:

`pom.xml`

```
...  
  
    <dependency>  
        <groupId>com.awnaba.projectconfigurator</groupId>  
        <artifactId>extension-spi</artifactId>  
        <version>${projectconfigurator.version}</version>  
        <scope>provided</scope>  
    </dependency>  
  
...
```

Some extensions will have to use also the class `ObjectAlias`, which is defined in a different jar (*operations-api*). In rare cases, extensions might require other classes from this same jar. In any of these cases, add the following dependency:

`pom.xml`

```
...  
  
    <dependency>  
        <groupId>com.awnaba.projectconfigurator</groupId>  
        <artifactId>operations-api</artifactId>  
        <version>${projectconfigurator.version}</version>  
        <scope>provided</scope>  
    </dependency>  
  
...
```

Where to find the .jar files

Both jars, including their javadoc, are published to the following URLs:

<https://nexus.adaptavist.com/content/repositories/external/com/awnaba/projectconfigurator/extension-spi/>

<https://nexus.adaptavist.com/content/repositories/external/com/awnaba/projectconfigurator/operations-api/>

4.5 Importing Packages

As defined in the dependencies in section 4.4, the extension will need to use some packages provided by Project Configurator. These will be available through OSGI. Functionally speaking, it would be inconvenient if the extension had to resolve those import packages when it is installed, as there is no guarantee that Project Configurator will be installed in Jira before the extension. The most sensible thing would be that those packages are imported when they are actually needed (this means *when an export or import is going to be run by PC*). From the OSGI point of view this implies that those packages must be declared as "DynamicImport-Package" and not as "Import-Package".

This would be specified in the pom.xml file as:

pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>com.atlassian.maven.plugins</groupId>
      <artifactId>jira-maven-plugin</artifactId>
      <version>${amps.version}</version>
      <extensions>true</extensions>
      <configuration>
        .....

      <!-- See here for an explanation of default instructions: -->
      <!-- https://developer.atlassian.com/docs/advanced-topics/configuration-of-instructions-in-atlassian-plugins -->
      <instructions>
        .....

        <!-- Add package import here -->
        <Import-Package>
          org.springframework.osgi.*;resolution:="optional",
          org.eclipse.gemini.blueprint.*;resolution:="optional",
```

```
<!-- Note the packages from PC must be in DynamicImport-Package,
so the following exclusion is added here -->

!com.awnaba.projectconfigurator.*,
*

</Import-Package>

<!-- Packages from Project Configurator must be imported dynamically to allow
the extension to interact with them, even if PC is installed later -->

<DynamicImport-Package>
    com.awnaba.projectconfigurator.*
</DynamicImport-Package>

.....
```

4.6 The Hollywood Principle

Creating an extension consists of implementing some of the interfaces which are defined in the `extension-spi-XXXX.jar`. You do not have to call the methods offered by those interfaces. PC will call them at the right times during the export or import operations.

Remember, "Don't call us; we'll call you!".

4.7 Welcome Lazy Developers!

In addition to the Hollywood Principle, you do not have to create all of the objects those methods return. PC offers you some components that act as factories that create many of those objects. These factories are:

| Factory | Types it will create |
|---|---|
| extensionsservices/ReferenceMarkerFactory | common/ReferenceMarker |
| extensionsservices/TranslatorFactory | common/ParamValueTranslator |
| customentities/references/ObjectReferenceProcessorFactory | customentities/references/ObjectReferenceProcessor customentities/references/MultiObjectReferenceProcessor |
| customentities/IdentifierFactory | customentities/PartialIdentifier customentities/InstanceIndependentIdentifier |

Table 1

4.8 How to Create Objects in an Extension

The lifecycle of objects an extension that implements the interfaces described in this document is very important for a smooth user experience. If their creation is triggered before PC is installed in Jira, that would crash with a *Java ClassNotFoundException*, since those interfaces would not yet be available. On the other hand, we need those objects to have already been created when the export or import is running.

The SPI, and the framework that uses it in PC, are designed so that you can satisfy that lifecycle requirement following very simple guidelines, leveraging the power of the Atlassian Spring Scanner.

You only need to annotate those classes to associate them with a dynamic application context. This context will be started when extensions are required by PC (when export, import, or reporting operations are being run by PC), so that those Spring beans are instantiated only when they are needed, and PC is guaranteed to be present and working at that moment.

Classes in the extension must be linked to the application context identified by *pc4j-extensions*, as shown in this code example.

```
@Profile("pc4j-extensions")
@Component
public class MyGadgetHookPoint implements GadgetHookPoint {
    ....
}
```

More precisely, classes implementing the following interfaces *must* be available as Spring beans within the context *pc4j-extensions*:

- HookPointCollection
- GlobalCustomEntity
- ChildCustomEntity

The rest of classes in the extension may be Spring beans or not.

As this is a development within the context of a P2 Jira app, it is assumed that the extension app is using Spring and Atlassian Spring Scanner version 2.

4.9 How to Inject PC Dependencies into Your Extension

The PC extension will need to get an instance of the factory interfaces mentioned above.

The implementations of these factories are available as Spring components. The usual methods of injecting a dependency in a component in Spring are sufficient. For example:

```
@Profile("pc4j-extensions")
@Component
public class FooWorkflowExtensions implements HookPointCollection {

    ...

    private TranslatorFactory translatorFactory;

    @Inject
    public FooWorkflowExtensions(@ComponentImport TranslatorFactory translatorFactory) {
        this.translatorFactory=translatorFactory;
        ...
    }

    ...
}
```

5 Workflow Extensions

Are you interested in using Project Configurator to migrate workflows that contain conditions, validators or post-functions defined by a third-party app? Let's discover how easily this can be achieved.

Step 0: Check if the workflow app is already supported

First navigate to the list of [supported workflow plugins](#). This list includes the most popular Jira apps for workflows and there are new additions to it from time to time. If your third-party app is already supported, then you do not need to create a new extension and can start moving workflows that use this app immediately.

If the app is not supported, follow the next steps with the help of an example, based on the app [Workflow Essentials for Jira](#).

Example source code

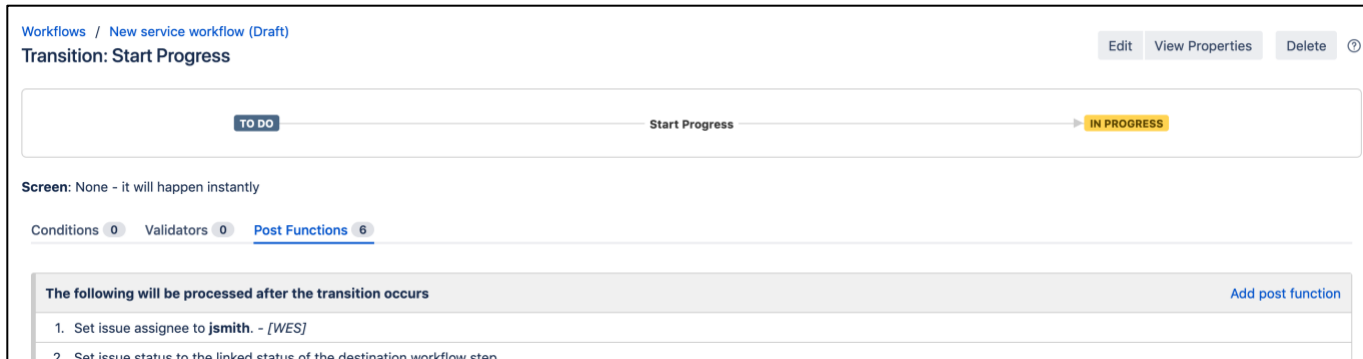
The complete source code for this example is available at <https://bitbucket.org/Adaptavist/example-workflow-extension/src/master/>.

Step 1: Create the workflow feature and export its XML descriptor

Install the third-party app in the Jira instance you will use to develop and test the extension, then create a workflow that has the conditions, validators, or post-functions (collectively, *workflow functions*) that you want to support with Project Configurator. In this guide, we focus on:

- [Assign a specific user post-function](#)
- [Date Compare condition](#)

Add them to a workflow:



Workflows / New service workflow (Draft)

Transition: Start Progress

Edit View Properties Delete ⓘ

TO DO → Start Progress → IN PROGRESS

Screen: None - it will happen instantly

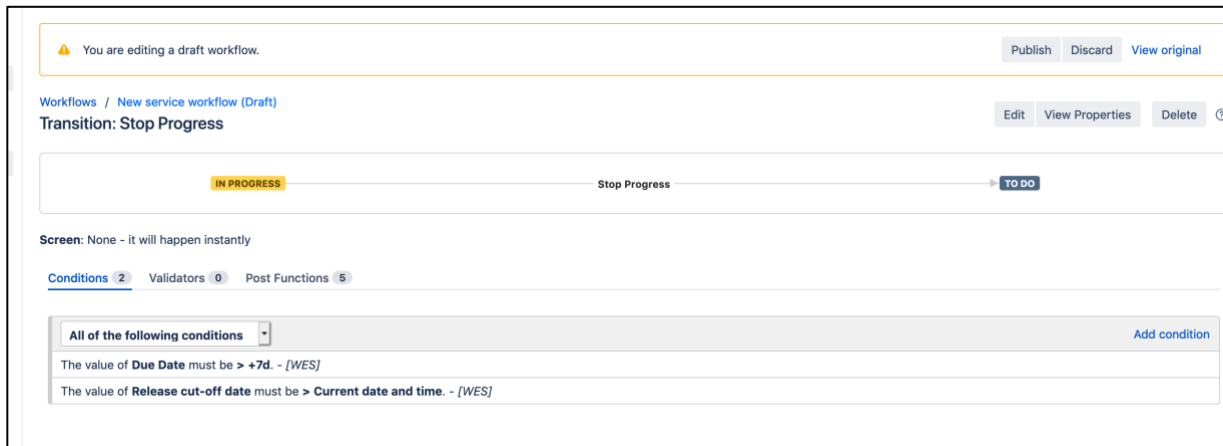
Conditions 0 Validators 0 Post Functions 6

The following will be processed after the transition occurs [Add post function](#)

1. Set issue assignee to jsmith. - [WES]
2. Set issue status to the linked status of the destination workflow step

Figure 1

Depending on the complexity of the chosen workflow function, it may be a good idea to create more than one instance of each to cover cases when the function can be configured in different ways. In the case of the Date Compare condition, you can see that it can work both with system or custom fields, and that it can handle a time expression or the current date and time. It is therefore better to create two instances of this condition to cover those cases.



⚠ You are editing a draft workflow. [Publish](#) [Discard](#) [View original](#)

Workflows / New service workflow (Draft)

Transition: Stop Progress

Edit View Properties Delete ⓘ

IN PROGRESS → Stop Progress → TO DO

Screen: None - it will happen instantly

Conditions 2 Validators 0 Post Functions 5

All of the following conditions [Add condition](#)

- The value of Due Date must be > +7d. - [WES]
- The value of Release cut-off date must be > Current date and time. - [WES]

Figure 2

Once you have the workflow with the desired functions, export it from Jira as XML.

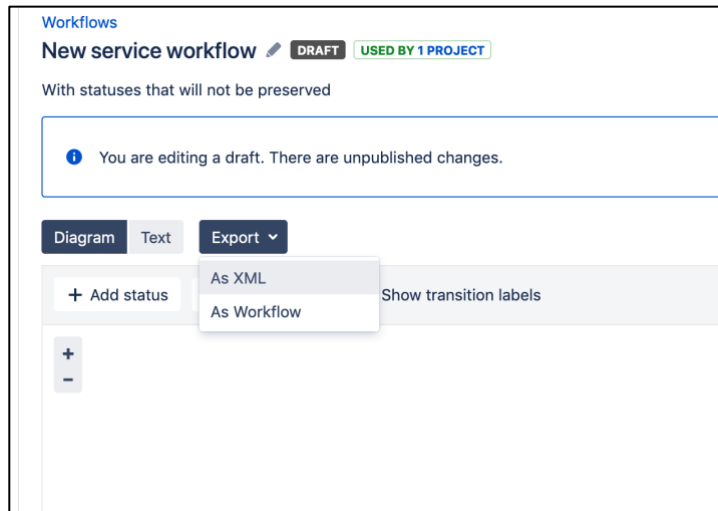


Figure 3

Open the XML file with your preferred editor and navigate to the section of the XML where the functions are defined.

Step 2: Implement interface HookPointCollection

Unless your extension already has another implementation of `com.awnaba.projectconfigurator.extensionpoints.common.HookPointCollection`, to which you can add the extensions for these workflow functions, create a new instance of this interface:

Implementing HookPointCollection

```
@Profile("pc4j-extensions")
@Component
public class WES4JExtensionModule implements HookPointCollection {
```

```
}
```

Step 3: Implement the workflow extensions

a. Date Compare condition

Analyze

Examine the workflow functions in the exported XML file. For example, start with the [Date Compare condition](#). Looking in the XML, you will find the two occurrences of this condition:

First occurrence of the "Date Compare condition"

```
<condition type="class">
  <arg name="EVALUATION_EXPRESSION">+7d</arg>
  <arg name="OPERATOR">></arg>
  <arg name="FIELD_ID">duedate</arg>
  <arg name="SELECT_DATE_COMPARE_OPTION">VARIABLE_EXPRESSION</arg>
  <arg name="class.name">de.codecentric.jira.condition.DateComparisonCondition</arg>
</condition>
```

Second occurrence of the "Date Compare condition"

```
<condition type="class">
  <arg name="EVALUATION_EXPRESSION"></arg>
  <arg name="OPERATOR">></arg>
  <arg name="FIELD_ID">10101</arg>
  <arg name="SELECT_DATE_COMPARE_OPTION">CURRENT_DATETIME</arg>
  <arg name="class.name">de.codecentric.jira.condition.DateComparisonCondition</arg>
</condition>
```

Looking at this part of the workflow descriptor, you see that the argument "FIELD_ID" can contain either the name of a system field ("duedate") or the ID of a custom field ("10101").

Next, consider the following: *Will this argument need to be changed for a successful migration?* If, yes, it has to be changed whenever a custom field is used, as the custom field used by this condition will likely have a different ID at a different Jira instance. As this argument might require being changed ("translated") in some cases for its correct migration to a different instance, you have to create an implementation of the `com.awnaba.projectconfigurator.extensionpoints.workflow.WorkflowTranslationPoint` interface. Create a method with this return type in the class created in step 2.

Implementing HookPointCollection

```
@Profile("pc4j-extensions")
@Component
public class WES4JExtensionModule implements HookPointCollection {

    public WorkflowTranslationPoint getDateCompareConditionHookPoint() {

    }

}
```

The other possibility is that this arg refers to a system field, like "duedate". A system object is completely transparent from the point of view of moving this workflow to another instance, as we expect all Jira instances to have that system field. This means you do not have to consider whether the system field exists or not, or if it has to be created before the workflow.

There are no other references to other entities in Jira, so when you implement support for the "FIELD_ID" argument, you are finished with this condition. The default action for PC when a workflow is migrated is to transfer it to the other instance as it is; for all other elements in the condition that do not reference entities in Jira, you do not need to do anything.

Define location

In order to create the `WorkflowTranslationPoint`, you have to provide information about the location within the workflow descriptor of the string that this extension is dealing with. In this case, the location of this string may be described as "the text node under an 'arg' element that has an attribute called 'name', equal to 'FIELD_ID' under a 'function' element that has another 'arg' with attribute 'name' equal to 'class.name' that is equal to 'de.codecentric.jira.condition.DateComparisonCondition'".

This description must be provided as an XPath v1 expression that identifies this location:

XPath expression

```
//condition[arg[@name='class.name' and  
(text()='de.codecentric.jira.condition.DateComparisonCondition')]]/arg[@name='FIELD_ID']/text()
```

Don't know XPath? No problem!

If you are not familiar with XPath, don't be concerned that you will have to learn an arcane piece of technology. As you will see, all the XPath expressions that you need to cover all your workflow support needs are essentially the same one, just with different values for the 'class.name' or 'name' attributes for the args, and then applying them for condition, validator, or function elements.

Have a look at another example!

Define the content of the reference

You must specify the content of the string. We know the string contains either the internal name of a system field or the numeric ID of a custom field. As discussed before, when a system field is present, we can simply ignore it, as it is not necessary to do anything with it in the migration.

The content of the string handled by this extension point is specified by the method, `ParamValueTranslator getTranslator()` in `WorkflowTranslationPoint`. Except for very specific cases, you do not need to create a `ParamValueTranslator` yourself, as there is a component, the `TranslatorFactory`, that creates these objects. If you browse the Javadoc for this factory class, you will see there is a method

`ParamValueTranslator` `fromOption(TranslateOption option)` that returns one among a set of built in `ParamValueTranslator(s)`. Here are two possible values for `TranslateOption`:

- `CUSTOM_FIELD_ID`: handles a numeric custom field ID
- `VOID_TRANSLATOR`: handles any string that does not reference any object, so this would serve us when a system field is referenced, as the system field can safely be ignored during the migration.

Next, you must specify that one of the two translators describes the string content depending on whether the field is a system field or custom. `TranslatorFactory` has this method:

```
ParamValueTranslator choiceOf(ParamValueTranslator trueTranslator, ParamValueTranslator falseTranslator,  
                             Predicate<String> importSelector, Predicate<String> exportSelector);
```

that expresses that: a string that can be described by two different translators depending on a boolean condition.

Many workflow plugins reference fields in a slightly different way to the one seen here, using the id string provided by `Field.getId()`. This produces the same strings shown here, with the only difference that a custom field would be identified by "custom field_10101" instead of "10101". For those cases there is already a built-in translator that handles the overall case, both for system and custom fields:

```
translatorFactory.fromOption(TranslatorFactory.TranslateOption.FIELD_STRING_ID)
```

Now, this can be expressed in code as:

Implementing a `WorkflowTranslationPoint`

```
public WorkflowTranslationPoint getDateCompareConditionHookPoint() {

    ParamValueTranslator translator = translatorFactory.choiceOf(
        translatorFactory.fromOption(TranslatorFactory.TranslateOption.VOID_TRANSLATOR),
        translatorFactory.fromOption(TranslatorFactory.TranslateOption.CUSTOM_FIELD_ID),
        this::isSystemCustomFieldId, this::isSystemCustomFieldId);

    return new WorkflowTranslationPointImpl(
        translator,
        "//condition[arg[@name='class.name' and
(text()='de.codecentric.jira.condition.DateComparisonCondition')]]/arg[@name='FIELD_ID']/text()"
    );
}

private boolean isSystemCustomFieldId(String fieldId) {
    Field field = fieldManager.getField(fieldId);
    return (field != null) && !fieldManager.isCustomField(field);
}
```

Code Notes

- `WorkflowTranslationPointImpl` is a convenience class that facilitates creating implementations of `WorkflowTranslationPoint`.
- Method `TranslatorFactory.choiceOf(...)` takes as arguments the translators to use in each case and two predicates. The second predicate (*exportSelector*) will be applied to the internal string (that is the `FIELD_ID` arg shown in this example) when exporting, to decide in which of the two cases the internal string fits in. In the same way, the first predicate (*importSelector*) will be applied to a previously exported string, during the import, in order to decide which of the two translators should be used. It is a good idea to base these predicates on an *invariant*, something that does not change in all the export-import process. In this way you do not have to care about the differences between the internal and the external (exported) string and the same predicate will be valid for both import and export. In this example, as the references to system fields will not be changed or even subject to any particular processing, both predicates are the same one, based on checking if the string is the ID of a system field.

Congratulations!

You have created your first extension point for workflows. With it, any instance of the *Date Compare Condition* from *Workflow Essentials for Jira* can now be migrated in an easy and safe way to another Jira instance.

b. Assign specific user post-function

Let's look at the post-function, which will be the second extension to implement.

Analyze

This is an occurrence of this post-function inside a workflow descriptor:

“User in group validator” in the workflow descriptor

```
...  
  
    <function type="class">  
        <arg name="full.module.key">de.codecentric.jira.wesset-assignee-to-specific-user-function</arg>  
        <arg name="USERNAME_VALUE_FIELD">jsmith</arg>  
        <arg name="class.name">de.codecentric.jira.postfunction.SetAssigneeToSpecificUserPostFunction</arg>  
    </function>
```

You can see that the only reference to other entities in Jira is a username. Your first thoughts may be:

"Wait a minute, I expect the username to be the same at the source and target instances, so it is not necessary to change anything here. What's more, as Project Configurator's default behaviour is to migrate anything in the workflow descriptor as it is, I do not need to create an extension point for this post-function, right?"

It is absolutely true that nothing needs to be changed and that if you do not create an extension point for this post-function, the workflow will be migrated correctly in most cases. However, imagine user *jsmith* exists at the source instance but not at the target. In this case, the workflow will be migrated with a formally valid descriptor, but this validator would be referencing a user that does not exist. This is somewhat inconsistent, and it could mean that this post-function does not work as expected. It will likely fail when this transition takes place. If you instead create an extension point for this post-function, then Project Configurator will help the Jira admin to manage this situation:

- When a user displays PC's *Object Dependencies Report* it will show that this user is referenced in that workflow.
- The user will be exported whenever this workflow is exported.
- When importing this configuration, Project Configurator will ensure that the user is created before the workflow or report the problem otherwise.

To achieve these benefits, you need to create an implementation of `WorkflowReferencePoint`. This is similar to what we did with the Date Compare condition, so let's dive in.

Define location

As in any workflow extension, you have to specify the location of the reference string within the descriptor as an XPath:

XPath expression

```
//function[arg[@name='class.name' and  
(text()='de.codecentric.jira.postfunction.SetAssigneeToSpecificUserPostFunction')]]/arg[@name='USERNAME_VALUE_FIELD']/text()
```

Notice how the structure of this XPath location is similar to the one used for the previous condition.

Define the content of the reference

In this case, the reference consists of a single username. References to strings that do not have to change during the migration process are represented by instances of the `ReferenceMarker` interface. As with `ParamValueTranslator`, you do not have to implement instances of `ReferenceMarker`, but there is a `ReferenceMarkerFactory`, which is available in the Spring context, that should be used to create them.

Looking at `ReferenceMarkerFactory`, you can see it has method, `fromOption(ReferenceOption option)` to get one of a group of built-in `ReferenceMarker` instances.

Looking at the available ReferenceOption, you will find there is one to handle a username:

| |
|---|
| STATUS_NAME |
| A reference to a Status by its name |
| USERNAME |
| A reference to a user by its username (NOT the full name) like "jsmith" |

Figure 4

Combining both things in the extension code, you should add the following method to WES4JExtensionModule class:

Implementing a WorkflowTranslationPoint

```
public WorkflowReferencePoint getAssignSpecificUserHookPoint() {  
  
    return new WorkflowReferencePointImpl(  
        referencemarkerFactory.fromOption(ReferenceOption.USERNAME),  
        "///function[arg[@name='class.name' and  
(text()='de.codecentric.jira.postfunction.SetAssigneeToSpecificUserPostFunction')]]/arg[@name='USERNAME_VALUE_FIELD']/text()"  
    );  
}
```

Now test it!

Your second workflow extension is completed now. Congratulations again!

Refer to the tips at the end of this document for ideas on how to test this extension.

6 Dashboard Gadget Extensions

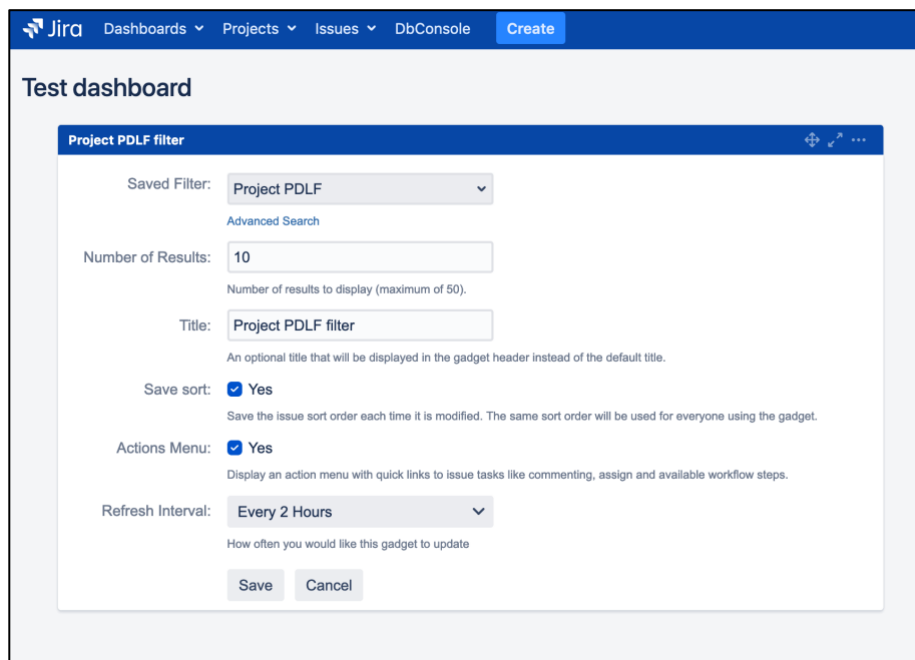
To explain how to create an extension for dashboard gadgets, we will base this on an example using the [Show Saved Filter with Columns for Jira](#) plugin, and its *Show Saved Filter with Columns* gadget.

Example source code

The complete source code for this example is available at <https://bitbucket.org/Adaptavist/example-gadget-extension/src/master/>.

Step 1: Create the gadget feature and export it with PC

First create an example of a gadget in a dashboard in Jira, configuring it as required. The screenshot below shows an example configuration.



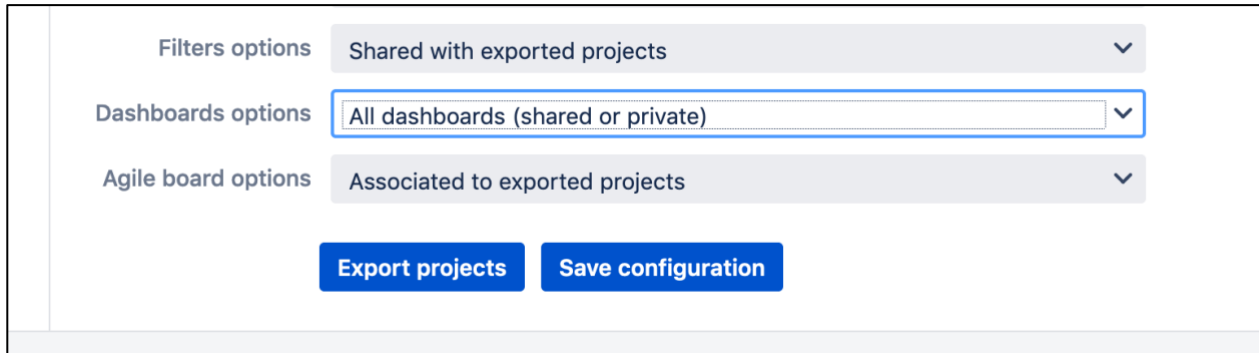
The screenshot shows the Jira interface with a 'Test dashboard' containing a 'Project PDLF filter' gadget. The configuration panel for the gadget is open, showing the following settings:

- Saved Filter:** Project PDLF (dropdown menu)
- Advanced Search** (link)
- Number of Results:** 10 (input field)
- Number of results to display (maximum of 50).** (text)
- Title:** Project PDLF filter (input field)
- An optional title that will be displayed in the gadget header instead of the default title.** (text)
- Save sort:** ☒ Yes
- Save the issue sort order each time it is modified. The same sort order will be used for everyone using the gadget.** (text)
- Actions Menu:** ☒ Yes
- Display an action menu with quick links to issue tasks like commenting, assign and available workflow steps.** (text)
- Refresh Interval:** Every 2 Hours (dropdown menu)
- How often you would like this gadget to update** (text)
- Save** and **Cancel** buttons

Figure 5

Next, export a configuration with PC that includes the dashboard. Bear in mind that Project Configurator does not export dashboards as a default, so you have to select the export option that exports the dashboard you created as a sample.

Go to the **Dashboards Options** drop-down menu in the *Export Projects* page and select an option that is appropriate for your case. In the example below, we have chosen to export all dashboards, which guarantees all dashboards in this instance will be exported (even private ones):



The screenshot shows the 'Export Projects' page with three dropdown menus: 'Filters options' set to 'Shared with exported projects', 'Dashboards options' set to 'All dashboards (shared or private)', and 'Agile board options' set to 'Associated to exported projects'. Below the dropdowns are two buttons: 'Export projects' and 'Save configuration'.

Figure 6

Once this configuration is exported, open the created XML file with an editor of your preference and locate the gadget description. There is a Dashboards element in the file that contains all exported dashboards, that contains several dashboard elements. Each of these will have several gadget items. It is easy to identify a dashboard by its name and owner, or as the default dashboard (system dashboard). You will see something similar to the following:

Section of XML file exported by Project Configurator

```
<dashboards>
  <dashboard>
    <default>false</default>
    <owner>admin</owner>
    <name>Test dashboard</name>
```

```
<description>Dashboard with sample gadget</description>
<layout>AA</layout>
<gadget>
  <type>rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml</type>
  <column>0</column>
  <row>0</row>
  <color>color1</color>
  <parameter>
    <key>columnNames</key>
    <value>issuetype@@issuekey@@summary@@priority</value>
  </parameter>
  <parameter>
    <key>filterId</key>
    <value>10000</value>
  </parameter>
  <parameter>
    <key>isConfigured</key>
    <value>true</value>
  </parameter>
  <parameter>
    <key>num</key>
```

```
        <value>10</value>
    </parameter>
    <parameter>
        <key>refresh</key>
        <value>120</value>
    </parameter>
    <parameter>
        <key>showActions</key>
        <value>true</value>
    </parameter>
    <parameter>
        <key>sortBy</key>
        <value></value>
    </parameter>
    <parameter>
        <key>sortByGlobal</key>
        <value>true</value>
    </parameter>
    <parameter>
        <key>title</key>
        <value>Project PDLF filter</value>
```

```
        </parameter>
    </gadget>
</dashboard>
```

Notice how each gadget has several parameters that represent how you configured the gadget at Jira user interface, setting preferences like the filter, the number of results to display, and the refresh period.

Step 2: Implement interface `HookPointCollection`

This step is the same as the example in Workflow Extensions. It is even possible to have workflow and gadget extensions within the same instance of `HookPointCollection`!

Step 3: Implement the gadget extension

Analyze

First check which of the parameter elements in your gadget contain references to other entities in Jira. As with workflows, you have to specify how these references can be found in the configuration for dashboards and what their content is. In this example, you can find two references:

- A parameter with key "columnNames" that is a list of field identifiers
- A parameter with key "filterId"

Parameter "columnNames" can be ignored. This appears in many gadgets and it is already handled by Project Configurator as standard.

Focus on parameter "filterId". If you check the filter set up during dashboard configuration, you will notice this parameter contains a string with the internal Jira ID for the selected filter. This means that this parameter will have to be translated to a different ID for a successful migration. This implies you will implement this extension as a `GadgetTranslationPoint`. If this parameter contained a reference to another entity that never requires a translation, then you would use a `GadgetReferencePoint`.

Define location

To identify a gadget extension point, you must specify two things:

- The type of gadget where it appears

- In which of the user preferences / parameters it appears for that gadget type

If you look at the `GadgetHookPoint` interface, which is the common ancestor of interfaces `GadgetTranslationPoint` and `GadgetReferencePoint`, you will notice it has these methods:

| |
|---|
| <code>com.atlassian.plugin.ModuleCompleteKey</code> <code>getModuleKey()</code> |
| <code>String</code> <code>getTypeURIStrng()</code> |
| <code>String</code> <code>getParamKey()</code> |

One of the first two methods will be used to identify the gadget type, in most cases the second one `getTypeURIStrng()`. In this example, looking at the XML file with the dashboard description, you will notice the gadget element has a child element `<type>`:

Gadget type

```
...  
    <gadget>  
        <type>rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml</type>  
        <column>0</column>  
    ...
```

This means the gadget type is identified by its URI (otherwise it would contain a child `<completeModuleKey>` element). Use the text of the `<type>` element as return value for the `getTypeURIStrng()` method of the associated `GadgetTranslationPoint`.

Define the content of the reference

This is described the same way as with workflow extensions. Navigate to the available predefined `TranslateOption` values that describe built in translators and identify one that handles a filter ID:

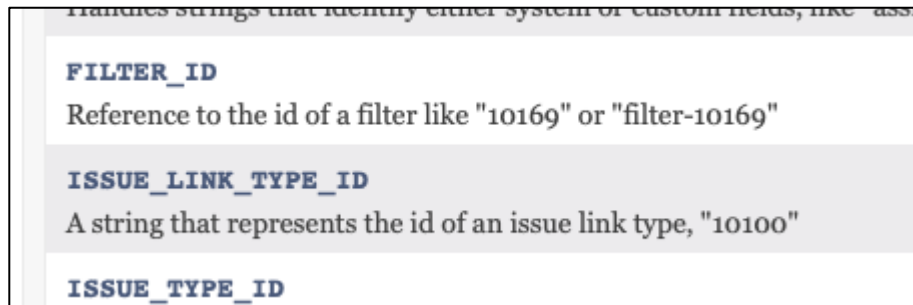


Figure 7

Combining the location and reference content, you define the `GadgetTranslationPoint`. It could similar to the following:

Implementation of `GadgetTranslationPoint`

```
public GadgetTranslationPoint getSSSCGadgetFilterHookPoint() {  
  
    return new GadgetTranslationPointImpl.Builder().  
        withTranslator(translatorFactory.fromOption(TranslatorFactory.TranslateOption.FILTER_ID)).  
        withUri("rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml").  
        withParamKey("filterId").build();  
}
```

Remember to add this as a method of the class you created in step 2, so you will have:

Complete implementation HookPointCollection

```
package com.adaptavist.projectconfigurator.ssscextension;

import com.atlassian.plugin.spring.scanner.annotation.Profile;
import com.atlassian.plugin.spring.scanner.annotation.imports.ComponentImport;
import com.awnaba.projectconfigurator.extensionpoints.common.HookPointCollection;
import com.awnaba.projectconfigurator.extensionpoints.extensionservices.ReferenceMarkerFactory;
import com.awnaba.projectconfigurator.extensionpoints.extensionservices.TranslatorFactory;
import com.awnaba.projectconfigurator.extensionpoints.gadget.GadgetTranslationPoint;
import com.awnaba.projectconfigurator.extensionpoints.gadget.GadgetTranslationPointImpl;
import org.springframework.stereotype.Component;

import javax.inject.Inject;

@Profile("pc4j-extensions")
@Component
public class SSSCEExtensionModule implements HookPointCollection {

    private TranslatorFactory translatorFactory;
    private ReferenceMarkerFactory referenceMarkerFactory;
```

```
@Inject
public SSSCEExtensionModule(@ComponentImport TranslatorFactory translatorFactory,
                           @ComponentImport ReferenceMarkerFactory referenceMarkerFactory) {
    this.translatorFactory = translatorFactory;
    this.referenceMarkerFactory = referenceMarkerFactory;
}

public GadgetTranslationPoint getSSSCGadgetFilterHookPoint() {

    return new GadgetTranslationPointImpl.Builder().
        withTranslator(translatorFactory.fromOption(TranslatorFactory.TranslateOption.FILTER_ID)).
        withUri("rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml").
        withParamKey("filterId").build();

}
```

And that's it!

You have now completed an integration of this gadget type with Project Configurator.

7 Custom Entity Extensions

We will use an example integration with the [Profields](#) app by Deiser, to help explain the concepts behind this type of extension. To limit the complexity of the example, we will assume some simplifications:

- The example will be focused on handling Profields belonging to 3 different types, layouts, and their relationship with projects.
- Working "as if" a layout could be associated to just one project.
- Using a simplified model of the inner structure of a layout. Actually, a layout has "sections". Each of these has up to three "columns", that can contain an indefinite number of containers (per column). Each container can have a number of "field items", that actually represent the occurrence of a field within the layout. In the simplified model of the example, a "section" will have "field items", thus ignoring "columns" and "containers".

Example source code

The complete source code for this example is available at <https://bitbucket.org/Adaptavist/example-custom-entities-extension/src/master>.

7.1 Have a Logical Model of the Information you Want to Support

Remember that implementing an extension to migrate a new kind of object that does not already exist in Jira before your app is installed, consists primarily of declaring the structure of those objects. In a second stage, it involves implementing the operations to create, update, and, in some cases, delete those objects.

Focusing on the structure of the data, you should be able to answer questions like these:

1. Which types of entities will be supported?
2. Which of these are parts of bigger entities and cannot exist independently?
3. Which properties should be migrated for each entity?
4. Which of these properties represent a reference to other entities?
5. A Jira admin using PC will move the configuration and/or data for one or several projects. In this situation, which items of your app should move with those projects?

The answer to each of these questions maps directly to concrete parts of the SPI. Do not be concerned if the precise answer to these questions is unclear now; we will revisit them in greater detail later in this document.

As a practical tip, before starting to create a custom entity extension, it is worthwhile creating a map of the entities to be extended with a content similar to a data model diagram. Include each entity, annotate it with its properties (maybe with a brief note of what each property means) and draw relations between entities whenever a property in an entity references another entity. Also include use relations to/from built-in entities in Jira (projects, custom fields, workflows, etc.) Using the Profields example, in a style similar to an E-R diagram will result in this model:

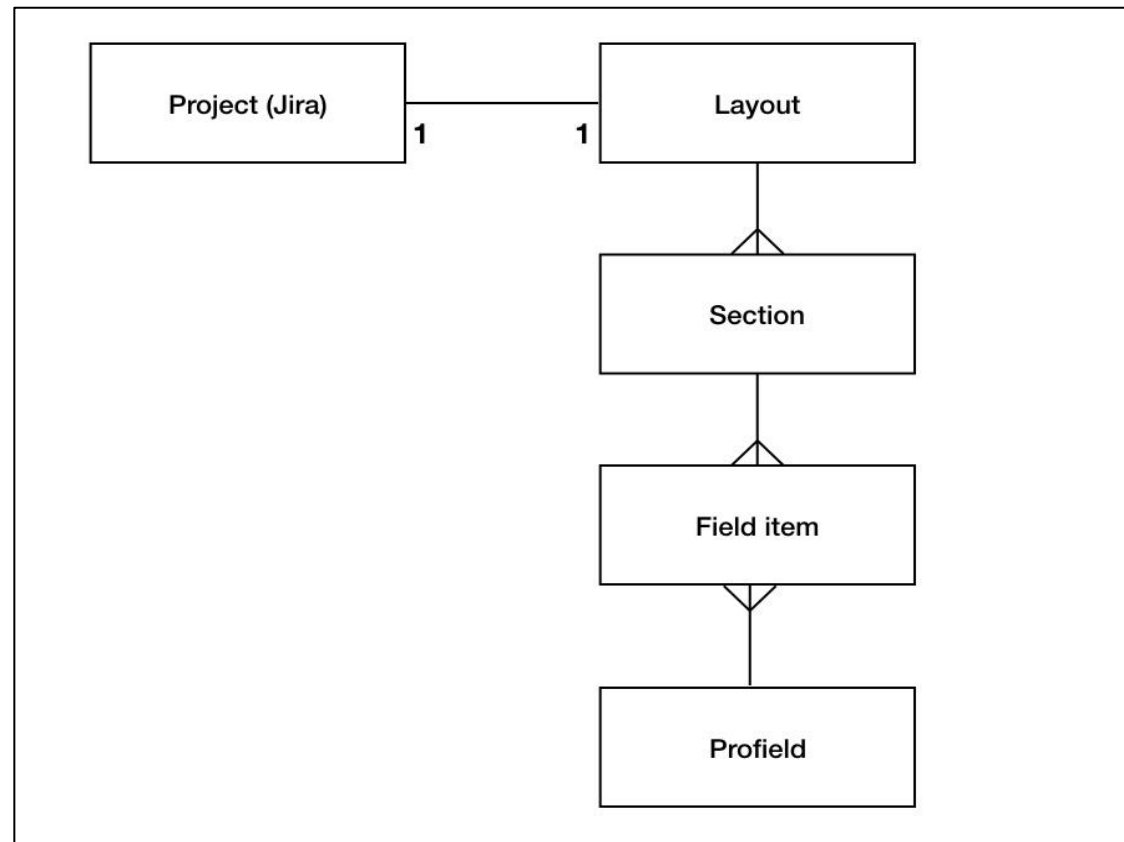


Figure 8

In some cases, it might be a matter of discussion for two related objects whether A uses B, or B uses A. We recommend following these patterns to make a decision:

- If for example, A contains a property, field, or attribute that is an identifier for B, but B does *not* contain a similar field/attribute with an identifier for A, we should say that "A uses B" or "A references B".
- If A requires object B to be properly configured, or to be useful in the normal operation of Jira, then we should consider that "A uses B". For example, a project must have a workflow scheme (even if it is the default one) in order to create issues, transition them, etc. On the other hand, a workflow scheme can exist even if no project is linked to it. Most people will consider that projects use workflow schemes, not the other way around.
- Think of how a typical Jira administrator would view the relationship. Try to model it in the way that will make most sense to this typical user. This is important, as it will make the user experience clearer and more intuitive.

CustomEntity is the main abstraction in any app extension. Each instance of CustomEntity represents the common properties of the set of objects that belong to the same type/entity defined by the Profields app. In this example, we can see that there are four entities to be represented:

- Layout
- Section
- Field item
- Profield

The next step is to decide for each entity if it represents objects that can exist on their own, or if they can only appear as part of other objects.

7.2 GlobalCustomEntity

Some objects in Jira exist independently of other objects. For example, a project may exist without being part of anything else in Jira. The same applies to the object types represented in a PC extension. If objects of a given type can exist on their own, then they are instances of a GlobalCustomEntity.

In this example, it is clear that both a layout and a "Profield" can be created without being inside any other object defined in the app, so their entities must be instances of `GlobalCustomEntity`:

Entity for Layouts

```
@Profile("pc4j-extensions")
@Component
public class LayoutEntity implements GlobalCustomEntity<Layout> {
    ....
}
```

Entity for Profields

```
@Profile("pc4j-extensions")
@Component
public class ProfieldEntity implements GlobalCustomEntity<Field> {
    ...
}
```

Note that `GlobalCustomEntity` has a type parameter that must be the Java class used to represent objects belonging to this entity, in these two cases `com.deiser.jira.profields.api.layout.Layout` and `com.deiser.jira.profields.api.field.Field`.

Names for the Custom Entity

Each `CustomEntity` has a name given by `getTypeName()`, which must be unique among all the `CustomEntity` objects belonging to the same app, and must not contain a colon.

Given these restrictions, concatenation of the app key, a colon, and the custom entity name will yield a name for the custom entity which is unique among all PC extensions that could be installed in a Jira instance. There is also a set of unique names for Jira built-in types, which are defined in `com.awnaba.projectconfigurator.operationsapi.ObjectAlias`, therefore there is a globally unique name for any entity whether built-in or part of a PC extension.

Creating New Objects

Any custom entity must define a method to create new objects of its type.

Creating New Layouts

```
@Override
public Layout createNew(ObjectDescriptor objectDescriptor) {
    Map<String, Object> properties = objectDescriptor.getProperties();
    LayoutBuilder builder = layoutService.getLayoutBuilder();
    builder.setName((String)properties.get(nameProperty.getPropertyName()));
    builder.setDescription((String)properties.get(descriptionProperty.getPropertyName()));
    return layoutService.create(builder);
}
```

This method receives an `ObjectDescriptor`. This is an object that contains all information relevant to the object about to be created. It has a map that contains internal values for all its properties keyed by the property names.

Configuration or data

Any `GlobalCustomEntity` has a boolean property that specifies if the object is part of the configuration or the data.

Consider two Jira instances. In one scenario, both are production instances and you want to move a project from one instance to the other (perhaps the line of business associated to that Jira project was sold to another company). In that case, the entire project, both its configuration (issue types, workflows, schemes, etc.) and its data (issues, attachments, comments) will be moved to the new instance.

In a different scenario, the destination instance is a production one and the source instance is for testing, where different changes to the project (perhaps to support new business processes) are being tested and validated by users. In this case, only the project configuration will be moved, as only those changes were made in testing and nobody would want to overwrite the live data in production.

Examining these scenarios will help you determine if a given object type must be moved in a configuration-only migration or not. PC supports both modes: migrating configuration only or configuration and data. If nothing is specified, as a default, global entities are considered to be configuration items (not data).

7.3 Properties

A property is any attribute of an object that users want to migrate to a different instance of Jira.

Any CustomEntity has a method `Collection<Property<T,?>> getProperties()`, that returns the properties of the entity that must be handled in the migration process. Usually, users will want to migrate persistent, visible properties of the objects. For example, they will not want to migrate IDs (not visible to users, not persistent across instances) or volatile information like cached results of a filter.

Returning to the Profields example, these properties can be associated to a Layout:

- Name
- Description
- The project it is associated to

So, a LayoutEntity defines this method:

Properties

```
@Override
public Collection<Property<Layout,?>> getProperties() {
    return Arrays.asList(nameProperty, descriptionProperty, layoutProjectsProperty);
}
```

A Property for the Description

Let us look into the property that represents the description.

Property is the top interface that represents any kind of property. It is generic with two types of parameters. The first represents the class of the owning object and the second is the type of *internal values* of this property. The internal value is found when getting the value of that property within Jira. It could be a variety of things: a string for properties like a description, a date, number, or even another object like an issue or a workflow. Imagine, for

example, that a relevant property of your entity is a filter it uses to work only on a specified set of issues; the internal value for that property might be the filter itself.

When it is exported, the internal value will be converted into a string (the "external string") and added to the exported contents. The external string must be instance-independent so that its meaning remains constant in any Jira instance. During the import, the reverse conversion will take place, from the external string to the internal value.

In the case of the layout description, looking at methods in the Profields Java API it is obvious that it can be read and set as a `java.lang.String`. So, this will be the type of internal values for this property. Moreover, the description of the layout is something that should be invariant in any instance of Jira where we want to move that layout. For simple properties that have strings as internal values, whose content is invariant across different instances of Jira and do not contain a reference to other Jira/app objects, there is a specific subinterface of `Property` called `StringProperty`:

Property for Name

```
private StringProperty<Layout> buildDescriptionProperty(){

    return new StringProperty<Layout>() {

        @Override
        public String getInternalValue(Layout layout) {
            // return null means "this property is empty"
            return layout.getDescription();
        }

        @Override
        public void setProperty(Layout layout, String s) {
            layout.setDescription(s);
            layoutService.update(layout);
        }
    };
}
```



```
    }

    @Override
    public String getPropertyName() {
        return "description";
    }

    @Override
    public boolean isSetInCreation() {
        return true;
    }
};
}
```

Additionally, a Property has a report name that identifies it to the end user (something like *description*, *applicable statuses*, or *default issue type*). See above method `getPropertyName()`.

There is also a method to set the property on a given object (see `setProperty()` method). This takes as arguments, the layout that will be modified and the new internal value of the description.

A Property also implements the method `boolean isSetInCreation()` that returns whether or not this property is set during object creation, as specified by its owning CustomEntity `createNew()` method. If this method returns true, PC will know that it does not have to set this property after creating a new object.

In this case, the method to create a layout (as seen before) sets its name and description, so the `isSetInCreation()` method for the description must return true.

A property for the associated Profield

ReferenceProperty

Often, a Property of an object consists of a reference to another object or to a collection of objects in Jira. These referred objects might be built-in Jira objects (issue types, statuses, filters, etc) or other objects which are supported by a PC extension.

This is relevant for the migration, so there is a specific sub-interface, `ReferenceProperty` that must be implemented for any Property that references other objects.

ReferenceProcessor

Every `ReferenceProperty` has a `ReferenceProcessor`. This is the object that is able to convert that reference to an external string and vice versa. It also resolves the reference and handles the cases where the reference is broken (i.e. the referred object does not exist in Jira).

`ReferenceProcessor(s)` will be supplied by some of the factory classes mentioned above. You should not create your own `ReferenceProcessor(s)`, except as a composition of `ReferenceProcessor(s)` provided by PC factory classes.

In the next section, we will review different types of `ReferenceProperty(s)` and their associated `ReferenceProcessor(s)`:

ExtendedObjectProperty / ObjectReferenceProcessor

A `ReferenceProperty` whose internal value is the referenced object itself.

ExtendedMultiObjectProperty / MultiObjectReferenceProcessor

A `ReferenceProperty` whose internal value is a collection of referenced objects.

TranslatableReference / ParamValueTranslator

A `ReferenceProperty` whose internal value is a string. It may need translation to be converted into an external string or not. For example, if it contains the ID string of the referred object, e.g. "11100", then it has to be translated during the migration to a different ID. The internal string may contain references to one or several objects.

MarkableReference / ReferenceMarker

Specialization of the above, for cases where translation into an instance-independent external string is not needed. For example, a collection of group names separated by commas, like "groupA, jira-developers, groupB".

When you have to create instances of `ParamValueTranslator` and `ReferenceMarker` you can use the techniques and examples described in the chapters about workflow and gadget extensions.

In the case of the field item, there is a property to represent its association with a `Profield`. We are going to use that as an example. Looking at the `Profields` Java API, you will find that there are methods to get and set the `com.deiser.jira.profields.api.field.Field` for a field item. Additionally, a field item cannot refer to more than one `Profield`, so it fits into an `ExtendedObjectProperty`:

Reference to Profield Property in `FieldViewItemEntity.java`

```
private ExtendedObjectProperty<FieldViewItemWithSection, Field> buildProfieldsRefProperty(){
    return new ExtendedObjectProperty<FieldViewItemWithSection, Field>(){
        @Override
        public ObjectReferenceProcessor<Field> getReferenceProcessor() {
            // This is an ObjectReferenceProcessor to one of the entities defined in this app
            return objectReferenceProcessorFactory.getObjectReferenceProcessor(profieldEntity);
        }

        @Override
        public Field getInternalValue(FieldViewItemWithSection fieldViewItemWithSection) {
            return fieldViewItemWithSection.getFieldViewItem().getField();
        }

        @Override
        public void setProperty(FieldViewItemWithSection fieldViewItemWithSection, Field field) {
```

```
        fieldViewItemWithSection.getFieldViewItem().setField(field);
        layoutService.update(fieldViewItemWithSection.getSectionViewParent().getParent());
    }

    @Override
    public String getPropertyName() {
        return "Proffield";
    }

    @Override
    public boolean isSetInCreation() {
        return true;
    }
};
}
```

Apart from the ReferenceProcessor, a ReferenceProperty is much like any other Property.

7.4 Identifiers

An Identifier is, for a given entity, a bidirectional mapping between String(s) and objects of that type. In other words, an Identifier permits finding the object of that type given a string or finding the string which identifies a particular object. For example, the relationship between statuses and their IDs is an Identifier. Given an ID string like "10110" it is possible to find the status with that ID, and from any given status it is easy to obtain its ID as a string.

There are two flavours of Identifiers. They can be InstanceIndependentIdentifier if the mapping would be the same in any instance of Jira. For example, given the key of a project in Jira, it is possible to find the corresponding project. We expect that the equivalent project in a different Jira instance will have the same key, so this mapping does not depend on a particular Jira instance. On the other hand, we can also map projects to their ID strings (like

"10202") and vice versa. However, it is most likely that equivalent projects in different Jira instances will have completely unrelated IDs. Therefore, this mapping changes whenever we look at a different instance. This would be an InstanceSpecificIdentifier.

Any Identifier has also a report name like "ID", "name", or "key", typically derived from the property they are based on.

Cross-instance Identifiers

Every CustomEntity must have at least one InstanceIndependentIdentifier, returned by the method `getCrossInstanceIdentifier()`. This identifier will be used to map equivalent objects in different instances.

For example, if this method returns an Identifier based on the object name, PC will treat entities of this type with the same name as equivalent objects in different instances. This means that during an import, if PC is bringing in an object with name "X", two situations may occur at the destination instance:

- No object of the same entity with name "X" exists previously: then PC will create a new object with that name with the same properties and children it had at the source instance.
- An object of the same entity with name "X" exists previously: then PC will modify that object so that it has the same properties and children it had at the source instance.

A CustomEntity may have an arbitrary number of additional Identifier(s), either InstanceIndependentIdentifier or InstanceSpecificIdentifier, returned by the methods `getOtherInstanceIndependentIdentifiers()` and `getInstanceSpecificIdentifiers()`. These are defined when they are used in ReferenceProperty from different entities as, for example, when objects of a different entity refer to objects of this entity by their ids. See methods such as this in TranslatorFactory:

[com.awnaba.projectconfigurator.extensionpoints.extensionservices.TranslatorFactory](#)

```
<T> ParamValueTranslator getFromNewObjectType(CustomEntity<T> customEntity,  
                                              InstanceSpecificIdentifier<T> internalRepresentation);
```

Creating identifiers from properties

Most often, identifiers will be based on one or several properties of an object. In this example, Profields will use a cross-instance identifier based on combining their type and name. There is a service, IdentifierFactory, that facilitates creating identifiers from object properties:

Creating an Identifier from Two Properties

```
private InstanceIndependentIdentifier<Field> buildNameTypeIdentifier(){
    return identifierFactory.identifierFromProperties(
        list -> findFieldByNameAndType(list.get(0).toString(), (FieldType)list.get(1)),
        nameProperty, fieldTypeProperty);
}

private Field findFieldByNameAndType(String name, FieldType type){
    List<Field> fields = fieldService.get(name);
    fields.removeIf(field -> !field.getType().equals(type));
    return fields.isEmpty() ? null : fields.get(0);
}
```

The method `identifierFromProperties()` takes as first argument a Function which receives a list with the internal values for the selected properties. The rest of arguments are the selected properties. Their order will be the same used to build the lists of internal values that will be received by the first Function.

If a property is part of the cross-instance identifier for an entity, it is not necessary to implement its `setProperty(...)` method, as it will never be called.

7.5 ChildCustomEntity

Some objects in Jira cannot exist outside a larger object (a parent object). A typical example would be the components in a project. No component can exist outside of a project. If the enclosing project is removed, then its components will also be removed.

If any entity in a PC extension exhibits this behaviour relative to other entity, then it should implement the ChildCustomEntity sub-interface. In the Profields example there are two entities that clearly meet this condition: section (which cannot exist outside a layout) and field item (that cannot exist outside its section). Then their corresponding entity classes will be as in the following:

Entity for Section

```
@Profile("pc4j-extensions")
@Component
public class SectionViewEntity implements ChildCustomEntity<SectionViewParent, Layout> {
    ...
}
```

Being a ChildCustomEntity has some implications:

- The ChildCustomEntity will inherit from its parent, CustomEntity, the status of being configuration or data.
- Subordinate objects may be removed from the Jira destination instance, giving their parent objects the same set of children as those in the configuration/data being imported. So, the ChildCustomEntity must implement the method `void delete(P parent, T childObject)` that removes a given child object.

delete() method

```
@Override
public void delete(Layout layout, SectionViewParent sectionViewParent) {
    layout.removeSection(sectionViewParent.getSectionViewItem());
    layoutService.update(layout);
}
```


- There must be a way to identify a child object from a String among the children of a given parent object. In the example above, the name can be used to identify a component among the components of a known Jira project. This mapping between String and child objects, restricted to a specified parent object, is called a `PartialIdentifier`. A `ChildCustomEntity` must have a `PartialIdentifier`. On the other hand, it does not have to declare a cross-instance identifier, as a default one will be automatically generated from the parent's cross-instance identifier and its `PartialIdentifier`. As in the case of the identifiers for `GlobalCustomEntity`, it is easy to create a `PartialIdentifier` from some properties of the `ChildCustomEntity`. In the following example, the name of a section is used to identify it within the layout.

Defining a `PartialIdentifier`

```
PartialIdentifier<SectionViewParent, Layout> buildPartialIdByName(){
    return identifierFactory.partialIdentifierFromProperties(
        (Layout layout, List<Object> singletonList) -> {
            String name = (String)singletonList.get(0);
            return layout.getSections().stream().
                filter(section -> ((SectionViewItem) section).getName().equals(name)).
                map(sectionViewItem -> new SectionViewParent(sectionViewItem,layout)).
                findFirst().orElse(null);
        },
        nameProperty);
}
```

Note that the first argument to method `partialIdentifierFromProperties()` is a `java.util.function.BiFunction` that receives as arguments the parent object and the list of internal values of the selected properties for the object to be looked up.

A method must be implemented to navigate to the parent object from one of its children: `P getParent(T childObject)`. In this example, given a section that method must return the layout it belongs to:

Navigate to the Parent Object

```
@Override
public Layout getParent(SectionViewParent sectionViewParent) {
    return sectionViewParent.getParent();
}
```

The object that represents sections in the Profields Java API, `SectionViewItem`, does not have a method that returns its parent layout. This is the reason why the example defines a bean class `SectionViewParent`, that combines a `SectionViewItem` and its parent layout. The example defines class `FieldViewItemWithSection` for the same reason.

7.6 Child Collections

Any `CustomEntity` with children must implement the method `Collection<ChildCollection<?,T>> getChildCollections()`.

Each `ChildCollection` returned by this method represents a collection of child entities of the same type. This collection has methods to:

- Identify the `ChildCustomEntity` of the children: `getChildCustomEntity()`
- Find the collection of child objects for a given parent: `getSubordinates(P parentObject)`

See this example in `LayoutEntity.java` specifying that a layout may contain sections:

Create a ChildCollection

```
private ChildCollection<SectionViewParent, Layout> buildSectionViewItemChildren(){

    return new ChildCollection<SectionViewParent, Layout>(){

        @Override
        public ChildCustomEntity<SectionViewParent, Layout> getChildCustomEntity() {
            return sectionViewEntity;
        }

        @Override
        public List<SectionViewParent> getSubordinates(Layout layout) {
            return layout.getSections().stream().
                map(section -> new SectionViewParent(section, layout)).
                collect(Collectors.toList());
        }

    };
}
```

A `ChildCustomEntity` can have its own children! See `SectionViewEntity` in this example.

7.7 ExportTriggerProperty

If an app defines new object types, it is likely that these objects will be used somewhere else in Jira. For example, imagine a test management app that creates new entities like *test case*, *test run*, or *test plan*. In this case, a project might use one or more test plans. This implies that, when writing a PC extension for this test management app, we need to specify this use relationship between projects and test plans. This relationship is relevant because we probably want to migrate a project's test plans when the project is migrated. In fact, this is the reason we started writing an extension for PC!

In the Profields example, a layout is associated to a project. It seems natural that when a project configuration is moved, if that project is using a layout, we will want that layout to be moved too as part of that project configuration. This is achieved with an `ExportTriggerProperty`.

Each instance of this interface represents one type of relationship between an entity not defined in this PC extension (like projects) to an entity defined within it (layouts).

`ExportTriggerProperty` is a subinterface of `ReferenceProperty`, so it will implement methods like:

LayoutProjectsProperty

```
public class LayoutProjectsProperty implements ExportTriggerProperty<Layout, Project, Project> {  
    ...  
    @Override  
    public ReferenceProcessor<Project> getReferenceProcessor() {  
        return objectReferenceProcessorFactory.getObjectReferenceProcessor(Project.class);  
    }  
    ...  
}
```

And it must implement two additional methods:

- `String getLinkedEntityName()` specifies the "entity name" of the entity (project) related to layouts. Typically, this will be one of the constants defined in class `ObjectAlias`.
- `Collection<T> getRelatedObjects(L linkedEntity)`, this method must return the collection of layouts that are linked to a given project.

[LayoutProjectsProperty](#)

```
public class LayoutProjectsProperty implements ExportTriggerProperty<Layout, Project, Project> {  
    ...  
    @Override  
    public String getLinkedEntityName() {  
        return ObjectAlias.PROJECT;  
    }  
  
    @Override  
    public Collection<Layout> getRelatedObjects(Project project) {  
        return Collections.singletonList(layoutService.getByProject(project));  
    }  
    ...  
}
```

Remember...

Functionally speaking, PC allows the migration of projects with all their configuration or data. This means that PC, when asked to export a group of projects, will include in the export file those projects plus all the objects that are directly or indirectly referenced by them. For example, a project might use a workflow scheme (direct reference) and the workflow scheme might use a workflow that in turn references a custom field (indirect references). All three (the workflow scheme, the workflow, and the custom field) will be exported alongside the project. On the other hand, a workflow that is not used by any of those projects will not be exported.

The same applies to the objects defined by PC extensions. If they are directly or indirectly part of the project configuration/data, they will be exported, otherwise they will be ignored. This means that, in order to be included in the migration, your app objects can be related directly to the project (as in the example) or indirectly through another object that is referenced by the project (like a custom field or a workflow which are used by that project).

8 Hints for Testing PC Extensions

8.1 Test the Actual Migration

Quite obviously, it is necessary to test that those configuration objects are actually moved to a different instance of Jira. At the end of the day, that is what 99% of the users will want the extension to do for them.

The following steps may help you to test this in a relatively fast way:

Create the configuration objects you want to move in your development instance

Going back to the gadget extension example, you would create a dashboard that contains a gadget of type *Show Saved Filter with Columns* and configure it with a valid filter.

Export a configuration containing those objects

Export these objects and check that the export completed without errors or warnings. If you look at the part of the XML file that corresponds to those objects you should notice that instance specific references (usually, all those that use IDs to refer to other objects) have been rewritten as instance independent strings. Continuing with the gadget example, you would see a description like:

Exported file

```
....  
  
    <name>Test dashboard</name>  
    <layout>AA</layout>  
    <gadget>  
        <type>rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml</type>  
        <column>0</column>  
        <row>0</row>  
        <color>color1</color>  
        <parameter>  
            <key>columnNames</key>  
            <value>issuetype@@issuekey@@summary@@priority</value>  
        </parameter>  
        <parameter>  
            <key>filterId</key>  
            <value>admin@@Due this week (RR)</value>
```

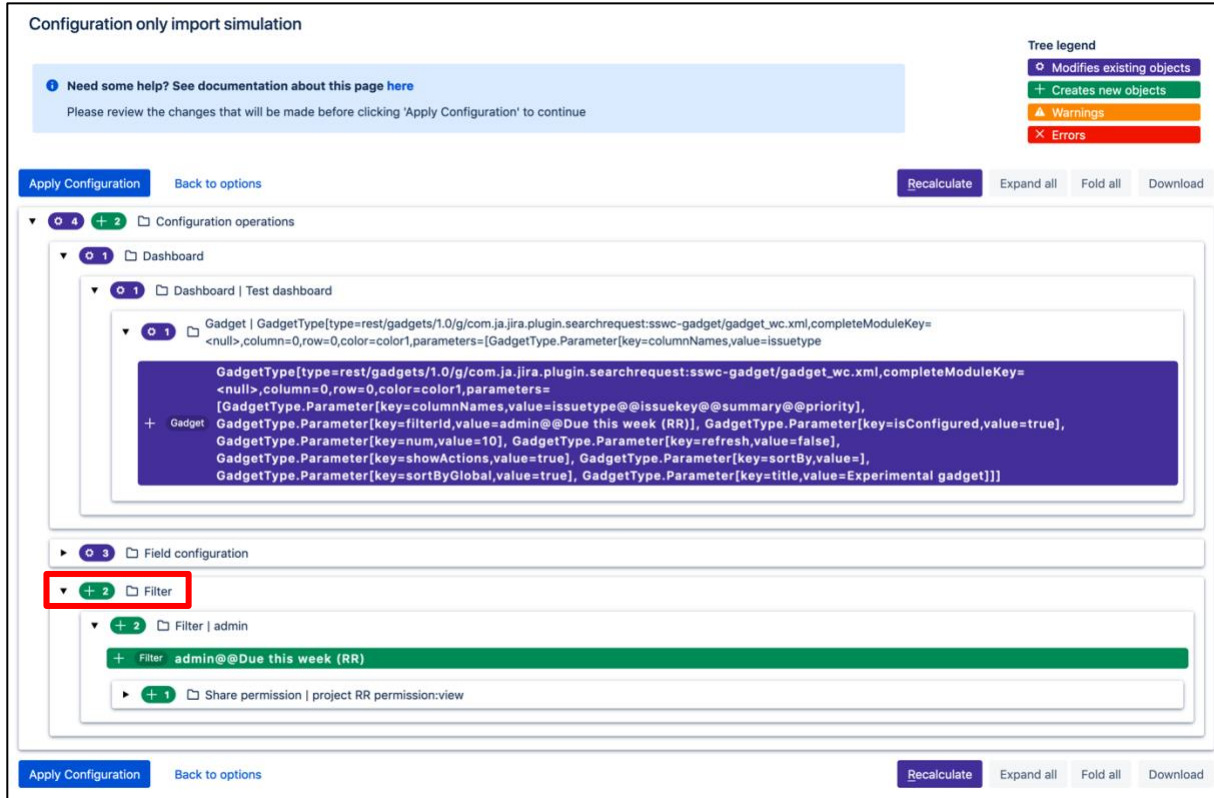
```
        </parameter>
        <parameter>
        . . . .
```

Remember that before the extension existed, this was exported with a filter ID, which would have been useless when moved into a different instance. Now that has been replaced by the filter owner and name, which will be used at the destination instance to rebuild the reference correctly.

Delete the configuration object and its dependencies and simulate the import of the exported file

To save the trouble of having two different Jira instances for testing, you could remove the created gadget and the filter it refers to, from the development instance and then import the exported file into the same instance again.

Run an import simulation first. You should see something like this (expanding the sections which are relevant to those objects):



Configuration only import simulation

Need some help? See documentation about this page here
Please review the changes that will be made before clicking 'Apply Configuration' to continue

Tree legend
 ○ Modifies existing objects
 + Creates new objects
 ⚠ Warnings
 ✖ Errors

Apply Configuration Back to options Recalculate Expand all Fold all Download

Configuration operations

Dashboard

Dashboard | Test dashboard

Gadget | GadgetType[type=rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml,completeModuleKey=<null>,column=0,row=0,color=color1,parameters=[GadgetType.Parameter[key=columnNames,value=issuetype],GadgetType.Parameter[key=filterid,value=admin@@Due this week (RR)],GadgetType.Parameter[key=isConfigured,value=true],GadgetType.Parameter[key=num,value=10],GadgetType.Parameter[key=refresh,value=false],GadgetType.Parameter[key=showActions,value=true],GadgetType.Parameter[key=sortBy,value=],GadgetType.Parameter[key=sortByGlobal,value=true],GadgetType.Parameter[key=title,value=Experimental gadget]]]

Field configuration

Filter

Filter | admin

Filter admin@@Due this week (RR)

Share permission | project RR permission:view

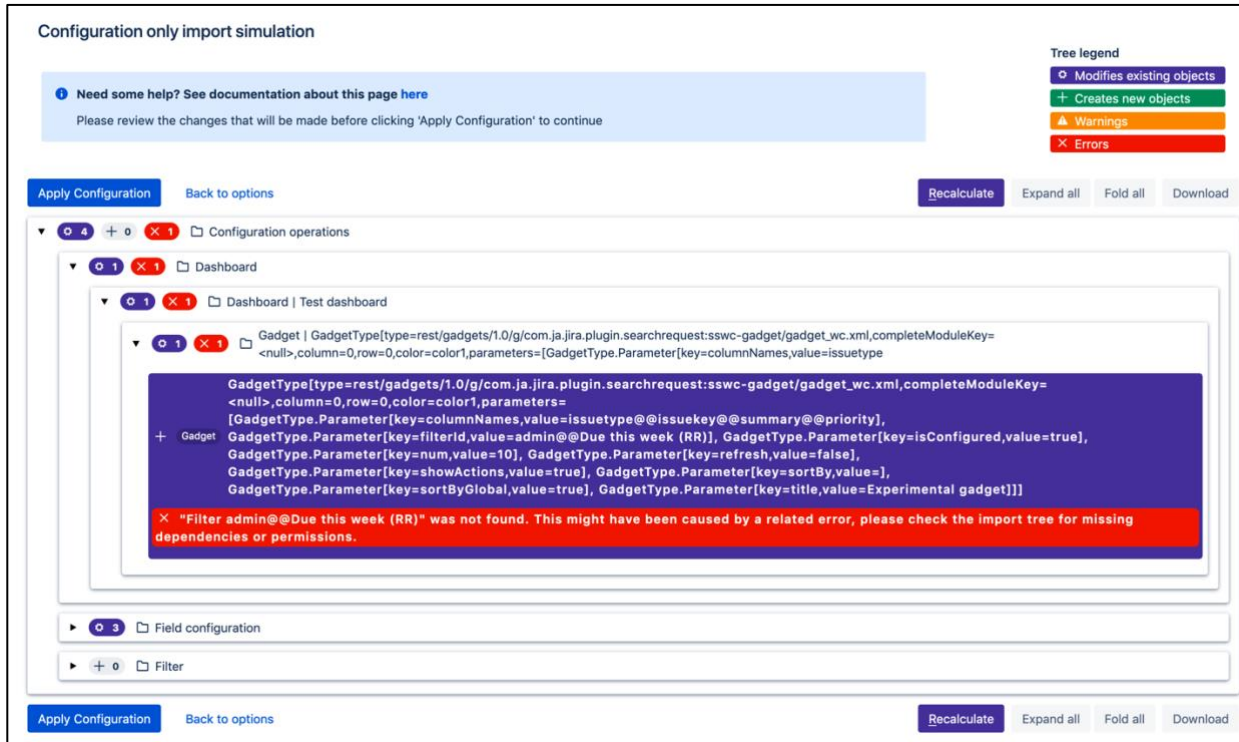
Apply Configuration Back to options Recalculate Expand all Fold all Download

Figure 9

Note that there are two changes planned to create both the missing gadget and the missing filter. Note also that the details for the gadget refer to the filter by its owner and name, not by its ID.

Disable creation of the referred object and recalculate

We know that the operations to create the gadget and the filter are logically related: if the filter were missing then it would not be possible to create the gadget, as it requires that filter to be properly configured. You can test this in the import simulation. Disable the creation of the filter by clicking the green action icon. It will be greyed out. Click the **Recalculate** button. The results appear similar to the following:



Configuration only import simulation

[Need some help? See documentation about this page here](#)

Please review the changes that will be made before clicking 'Apply Configuration' to continue

Tree legend

- Modifies existing objects
- Creates new objects
- Warnings
- Errors

Buttons: Apply Configuration, Back to options, Recalculate, Expand all, Fold all, Download

Configuration operations

- Dashboard**
 - Dashboard | Test dashboard**
 - Gadget**
 - Gadget | GadgetType[type=rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml,completeModuleKey=<null>,column=0,row=0,color=colort1,parameters=[GadgetType.Parameter[key=columnNames,value=issuetype
 - Gadget
 - GadgetType[type=rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml,completeModuleKey=<null>,column=0,row=0,color=colort1,parameters=[GadgetType.Parameter[key=columnNames,value=issuetype@@issuekey@@summary@@priority],GadgetType.Parameter[key=filterId,value=admin@@Due this week (RR)],GadgetType.Parameter[key=isConfigured,value=true],GadgetType.Parameter[key=num,value=10],GadgetType.Parameter[key=refresh,value=false],GadgetType.Parameter[key=showActions,value=true],GadgetType.Parameter[key=sortBy,value=],GadgetType.Parameter[key=sortByGlobal,value=true],GadgetType.Parameter[key=title,value=Experimental gadget]]]
 - Error:** "Filter admin@@Due this week (RR)" was not found. This might have been caused by a related error, please check the import tree for missing dependencies or permissions.

Field configuration

- Filter**

Buttons: Apply Configuration, Back to options, Recalculate, Expand all, Fold all, Download

Figure 10

Note that after disabling the changes on filters (they are greyed out), creation of the gadget has a red box (this means an error) that says a filter with the required owner and name cannot be found.

Enable operations and run the actual import

Until now nothing has changed in Jira. You are still working with a simulation; it is like a preview of the changes but none of them have actually occurred.

In the next step, re-enable changes to the filters and click **Apply Configuration**. This will apply the changes to your source Jira instance. The tree with the actual changes will be displayed:

Configuration only import results

Expand all Fold all Download

- Configuration operations
 - Dashboard
 - Dashboard | Test dashboard
 - Gadget | GadgetType[type=rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml,completeModuleKey=<null>,column=0,row=0,color=color1,parameters=[GadgetType.Parameter[key=columnNames,value=issuetype,


```
GadgetType[type=rest/gadgets/1.0/g/com.ja.jira.plugin.searchrequest:sswc-gadget/gadget_wc.xml,completeModuleKey=<null>,column=0,row=0,color=color1,parameters=[GadgetType.Parameter[key=columnNames,value=issuetype@issuekey@@summary@@priority],GadgetType.Parameter[key=filterid,value=admin@@Due this week (RR)],GadgetType.Parameter[key=isConfigured,value=true],GadgetType.Parameter[key=num,value=10],GadgetType.Parameter[key=refresh,value=false],GadgetType.Parameter[key=showActions,value=true],GadgetType.Parameter[key=sortBy,value=],GadgetType.Parameter[key=sortByGlobal,value=true],GadgetType.Parameter[key=title,value=Experimental gadget]]]
```
- Field configuration
 - Filter
 - Filter | admin
 - Filter admin@@Due this week (RR)
 - Share permission | project RR permission:view
 - Share permission project RR permission:view

Expand all Fold all Download

Figure 11

Verify that the changes displayed are what you expect. They should be the same as those previewed in the simulation. Finally, check the dashboard in Jira and verify that the new gadget has been created with the correct configuration. Make sure the gadget is configured for a new filter, also created by the import, called "Due this week (RR)" and owned by user "admin".

8.2 Use the Object Dependencies Report

The Object Dependencies report is a feature of Project Configurator that analyses dependencies between configuration objects in any instance of Jira.

In the case of the workflow examples discussed in this guide, you will notice the following "used by" relationships exist:

- User "jsmith" is used by the "New service workflow" (as it is used in a post-function of the workflow)
- Custom field "Release cut-off date" is used by the "New service workflow" (as it is used in a condition of the workflow)

Integrations with other apps are designed to work seamlessly with this feature. So, if you run the Object Dependencies report, you will see these relationships shown in the report, for example:

| Object dependencies report | |
|----------------------------|---|
| Custom field | Custom field |
| Event type | com.atlassian.jira.plugin.system.customfieldtypes:datepicker:Release cut-off date |
| Field configuration | Used by: |
| Group | Workflow "New service workflow" |
| Issue type | |
| Issue type scheme | |
| Issue type screen scheme | |
| Notification scheme | |

Figure 12

Note this can be quite useful for testing references like this one to a username, that are implemented as a `WorkflowReferencePoint` or a `GadgetReferencePoint`. In these cases, including the references in the integration of the app with Project Configurator is useful to the extent it makes Project Configurator aware of the dependency between the related objects. Verifying the "Object dependencies report" is a fast and simple way to check this is working as expected.

The "Object dependencies report" will include only those objects that are directly or indirectly used by any of the projects in the instance. This means that these will not appear: Objects that are exclusively used by dashboards, filters or Agile boards, and the relations where any object is used by any dashboard, filter or Agile board. In general, any object which is not used by any project (like an inactive workflow).

8.3 Pro Tip: Create the Extension in Two Stages

Creating the extension in two stages simplifies both the implementation and the testing.

When creating a custom entity extension, you can exclude from the initial implementation the methods that actually create or delete objects or set any of its properties. These methods will only be used during a real import, so the following operations should work correctly for the related custom entities, even when those methods are not yet implemented:

- Export
- Object dependencies report
- Import conflict detection
- Simulated import

This lets you implement partially the support for some custom entity, test that partial implementation and later, with the confidence that a substantial part of the extension is already correct, proceed to complete the "create / delete / update" methods.

8.4 Use the Jira Log if Necessary

It is possible to extract a lot of useful information from the Jira log file. If this is required, then set the log level to `DEBUG` for the package `com.awnaba.projectconfigurator` (see [Change Logging Levels in Jira Server](#) to learn how to temporarily change the log level for specific packages in Jira).

After setting the log level to DEBUG, run any operation with Project Configurator and you will see the increased content in the log file.

For example, in the case of an export with the workflow extension discussed in this guide, you will see entries like these:

```
...
2020-05-06 14:41:30,684+0200 http-nio-2990-exec-11 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.extensionpoints.management.ExtensionActivatorImpl] Starting extensions
2020-05-06 14:41:30,684+0200 http-nio-2990-exec-11 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.extensionpoints.management.ExtensionActivatorImpl] Enabling PC extensions in plugin: com.adaptavist.projectconfigurator.wes4jextension
2020-05-06 14:41:30,685+0200 http-nio-2990-exec-11 INFO admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.s.s.dynamic.contexts.GeminiOsgiBundleXmlApplicationContext] Refreshing
GeminiOsgiBundleXmlApplicationContext(bundle=com.adaptavist.projectconfigurator.wes4jextension, config=bundle://232.0:1/META-INF/pc4j-
extensions/spring/profile-pc4j-extensions.xml): startup date [Wed May 06 14:41:30 CEST 2020]; parent:
NonValidatingOsgiBundleXmlApplicationContext(bundle=com.adaptavist.projectconfigurator.wes4jextension, config=osgibundle:/META-
INF/spring/*.xml)
...
2020-05-06 14:41:30,739+0200 http-nio-2990-exec-11 WARN admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.projectconfigurator.wes4jextension.spring]
    Spring context started for bundle : com.adaptavist.projectconfigurator.wes4jextension id(233)
v(1.0.0.SNAPSHOT) file:/Users/pepemaranon/Workspaces/jira-project-config-plugin/project-configurator-plugin/amps-standalone-jira-
8.8.1/target/jira/home/plugins/installed-plugins/plugin_3339388906409322341_wes4jextension-1.0.0-SNAPSHOT.jar

    If you want to debug the Spring wiring of your code then set a DEBUG level log level as follows. [ This is a dev.mode only message. ]
log4j.logger.com.adaptavist.projectconfigurator.wes4jextension.spring = DEBUG, console, filelog
...
```

2020-05-06 14:41:31,176+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.extensionpoints.management.ExtensionActivatorImpl] Found 2 implementations of extension entities for plugin
com.adaptavist.projectconfigurator.wes4jextension

...

2020-05-06 14:41:32,514+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] XPath provided by extension: //function[arg[@name='class.name' and
(text()='de.codecentric.jira.postfunction.SetAssigneeToSpecificUserPostFunction')]]/arg[@name='USERNAME_VALUE_FIELD']/text()

2020-05-06 14:41:32,514+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] Node for this occurrence: 3 #text admin

2020-05-06 14:41:32,514+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] XPath found one occurrence: admin

2020-05-06 14:41:32,514+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] Found reference by string: admin

2020-05-06 14:41:32,514+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] Finished processing this extension

2020-05-06 14:41:32,514+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] XPath provided by extension: //condition[arg[@name='class.name' and
(text()='de.codecentric.jira.condition.DateComparisonCondition')]]/arg[@name='FIELD_ID']/text()

2020-05-06 14:41:32,515+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] Node for this occurrence: 3 #text 10001

2020-05-06 14:41:32,515+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] XPath found one occurrence: 10001

2020-05-06 14:41:32,515+0200 JiraTaskExecutionThread-5 DEBUG admin 881x49361x1 16cohu0 0:0:0:0:0:0:1 /secure/project-export!export.jspa
[c.a.p.adapters.workflow.WorkflowTranslator] Finished processing this extension

...

Note the following details:

- `/secure/project-export!export.jspa` is the relative URL where the export was launched from
- There are entries that mark when Project Configurator starts searching for extensions and a specific one for each extension
- There are also entries that show when the dynamic Spring context for each extension is started
- Other entry show that this workflow extension defines two extension points (one for the condition and another for the post-function)
- For each extension point, it is reported when its processing begins and ends and the number and content of occurrences found, for each workflow

Exploring the log files in this way can be useful when the extension appears not to work or even that it does not exist. This is often caused by a problem that prevents the starting of its dynamic Spring context. In these cases, Project Configurator will simply ignore the offending extension and continue with the rest of the operation.