User Guide: Advanced Jira to Spira Migration Tool

Introduction

Jira and SpiraPlan are both software tools that help teams plan, manage, and deliver projects. Jira is a product of Atlassian, and SpiraPlan is a product of Inflectra. This tool migrates and converts Jira issues to SpiraPlan artifacts along with metadata and attachments.

The tool serves as an alternative to SpiraPlan's native migration app, Jira Importer, but is designed to offer more flexibility and control over the migration process because it allows you to customize the mapping between the two systems.

This means you need to do some manual configuration in SpiraPlan before running the migration. For example, create custom fields, set workflows, and associations.

In this document, you will find instructions on how to configure and run the migration tool. It operates via the command line and the intended user is a developer, a DevOps engineer or technical system administrator.

However, the user guide could also be of interest to any stakeholder of the migration process and who wants to get a deeper understanding of how the tool is built and how it works.

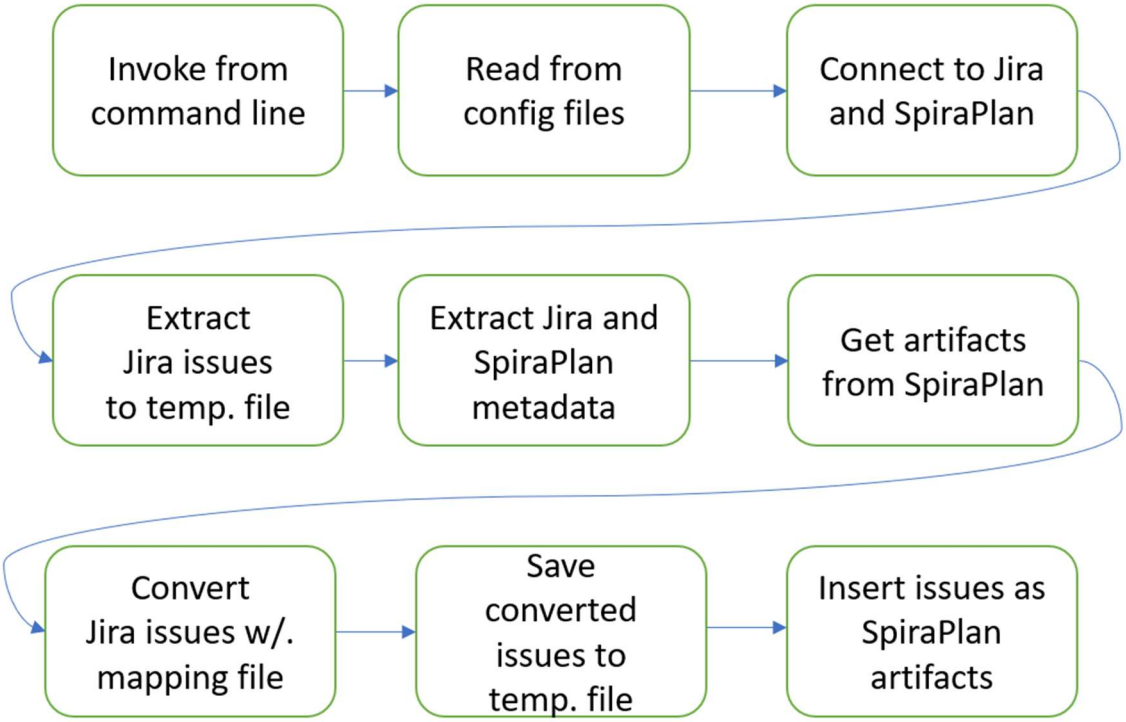
For information on how to navigate and operate in SpiraPlan or Jira, please use their respective instruction manual.

High Level Description

The tool is an independent Python program that interacts with Jira- and SpiraPlan instances through their respective REST APIs. It operates from the command line and uses a configurable mapping file, that defines a set of rules for the data transfer between the two systems.

Data Flow Chart

The migration starts with the user entering a given command. An example of the data flow when migrating issues is visualized in the following diagram.



Features the tool can handle.

Program Level:

- Import custom lists for use at the system level.
- Import versions to project milestones.
- Import issues as capability artifacts and establish associations with milestones.
- Remove existing capability artifacts and milestones.

Product Level:

- Import custom lists at product template level.
- Import versions into a structured hierarchy of releases.
- Import components.
- Import issues as requirements and create a hierarchical structure while linking them to capabilities, releases, and components.
- Import issues as incidents and associate them with corresponding requirements.
- Import issues as tasks and set association with relevant requirements.
- Map standard and custom properties from Jira issues
- Import issue comments to the corresponding artifact at product level.
- Set associations between artifacts within the same Spira product.
- Import documents to a designated folder and link them to correct artifacts.

- In a separate command, set link between documents and correct artifact within the same Spira product
- Assign appropriate statuses to artifacts in Spira based on a provided mapping template.
- Assign priorities to artifacts in Spira as defined in the programs mapping template.
- Remove all artifacts, components, and releases with a separate command, excluding documents.
- Remove all documents at the product level using a separate command.

Prerequisites

System Requirements

Operating System

The tool is designed to run in both Linux and MacOS environments. For Windows, a WSL Linux environment is suggested to make it work.

Disk Space

Ensure there is sufficient disk space for data storage and temporary files. Generated during the migration.

Network Connectivity

Ensure the system has internet access and firewall rules allow outbound connections.

The tool requires an IDE or a terminal window.

Access

Ensure writeable access to the tools code base.

Ensure that the accounts that will be used in connection files have correct access in Jira and in SpiraPlan. That means that the account can retrieve all necessary information in Jira and can both retrieve and create

Software Requirements

Python >=3.10

Ensure that python is installed, and that the version is 3.10 or later.

This can be checked by typing **python –version** or **python3 –version** in the terminal.

‘pip’ Package Manager:

Ensure that pip is installed.

This can be checked by typing **pip –version** or **pip3 –version** in the terminal.

pip is a package manager for Python, used to install and manage software packages written in Python. It simplifies the process of downloading, installing, and managing libraries and dependencies for Python projects. Installing packages with pip can require internet access depending on the package version

Getting started

Downloading the tool

Download the source code from …

Installing dependencies

Necessary dependencies are stored in the requirement.txt file inside the project.

To install all dependencies run: **pip install -r requirements.txt** or **pip3 install -r requirements.txt.**

Configuring the Migration

Configure SpiraPlan

Users

Create all users in SpiraPlan. Users are mapped over the e-mail address.

Program and Capabilities

Create a program. For Capabilities, set priority, status, and type. These are the values you will use in the mapping template.

Create custom properties included in the mapping template. You also need to create a custom property called Jira Id of type text.

Products and Requirements

Create a product and prepare artifacts at template level for the requirements. Set importance, status and type. These will be used in the mapping template.

Create custom properties included in the mapping template, included the property called Jira Id of type text.

Incidents

Set priority, status and type at template level. These will be used in the mapping template.

Create custom properties included in the mapping template, included the property called Jira Id of type text.

Tasks

Set priority and type at template level. These will be used in the mapping template.

Create custom properties included in the mapping template, included the property called Jira Id of type text.

Documents

Create custom properties called Jira Id of type text.

Configure Connection files

There is one ".env" connection file for both Jira and SpiraPlan. A ".env.template" is provided in the script root that must be copied and renamed to ".env" in the same directory.

Fill in the now copied ".env" file with the base URL, username, and API for both Jira and SpiraPlan. For SpiraPlan, include the curly brackets on each side of the API key.

Configure mapping template yaml file

The tool uses a mapping file to determine how to map the meta data from Jira to SpiraPlan. The mapping file is written in YAML format. In the mapping file, each Jira value is a key (left) and each Spira value is the mapped value for the key (right).

```
statuses:
    capabilities:
        Open: To Do
```

Example of YAML-code in the mapping config file

The example above says: Mapping statuses for issues that will be inserted as capabilities and carries the status Open in Jira, it will be set to status To Do in SpiraPlan.

The mapping file maps the following attributes: Priority, Status, Issue Properties, and Type of Issue.

The Type of Jira Issue determines what type of artifact the issue will be in SpiraPlan.

The mapping file also sets the order of when the artifacts will be inserted in Spira. This is important because Spira sets relations in a hierarchical structure.

All values and keys are case sensitive.

Running the Migration Tool

The tool operates from the command line and has a fixed set of commands with additional flags attached to them. It is very important that some of the commands run in a specific set order, because of how the connections and dependencies are created in SpiraPlan. The commands are presented below in the recommended order.

Commands, arguments, and flags.

Arguments

jql

Jql stands for Jira Query Language and is a flexible way to search and filter for jira issues.

Example: "**issue = PROJ-12**" to search for a specific issue or "**issuetype = Epic**" to receive all Epics

list of projects

This argument specifies the Jira projects from which you want to migrate your issue data. The list should be a space-separated sequence of project names without quotation marks.

Example: **PROJ1 PROJ2 PROJ3**

list of template ids

This argument specifies SpiraPlan templates to which you want to migrate your issue data to. The list should be a space-separated sequence of template names or ids without quotation marks.

Example: 12 13 **My_spiraplan_project_template**

program identifier

The SpiraPlan program name or only program id.

Example: **My_spiraplan_program** or **4**

project identifier

The SpiraPlan project name or only project id.

Example: **My_spiraplan_project** or **25**

Flags

Most of these flags are optional and exist on most of the commands, but see specific command for help with relevant flags.

nossl

Boolean flag, specify if you want to disable the ssl check. Disabling SSL opens the script for man-in-the-middle-attacks but might be required if there is no valid HTTPS cert available.

Example: **-nossl** or **--skip-ssl-check**

m

This flag specifies a file from which to take the mapping configuration. If this flag is not set the default value, config/mapping_template.yaml, will be used

Example: **-m my_mapping_template.yaml** or **--mapping my_mapping_template.yaml**

jo

This flag specifies a file from to where the extracted jira data is going to. If this flag is not set the default value, jira_output.json, will be used.

Example: **-jo my_jira_output_file.json** or **--jira-to-json-output my_jira_output_file.json**

system

A Boolean flag that is set when migrating custom lists at system level. This will only be needed to done once. The system flag will override the template flag.

Example: **-system** or **–system-level**

Template

A flag set when migrating custom lists at template level. The flag specifies a list of template names or ids.

Example: **-template or –spira-templates**

Commands

All commands start with python main.py or python3 main.py. This is necessary for the operating system to use the python interpreter and the main.py argument specifies that the tool should start executing from the main.py file.

Clean program

This command removes all milestones and all capabilities.

python3 main.py clean_program <program identifier>

Clean product

This command removes all releases, components, requirement, incidents, and tasks

python3 main.py clean_product <product identifier>

Clean documents

This command removes all documents.

python3 main.py clean_product_documents <product identifier>

Migrate custom lists system level

This command populates the custom lists needed to create custom properties of type select list or multiselect list. It needs to be run first.

python3 main.py migrate_customlists <PROJ1 PROJ2 PROJ3> -system

Migrate custom lists product template level

This uses the same command as creating custom lists at system level, with the difference of using another flag. Creating custom lists at system level and at project template level, can be done in any order, but needs to be done before creating any list custom properties.

python3 main.py migrate_customlists <PROJ1 PROJ2 PROJ3> -template

Migrate versions to milestones

This command operates at a program level and migrates Jira versions to SpiraPlan milestones. It is important to execute this command before migrating issues to capabilities, because when creating a capability, it sets the association to a milestone.

python3 main.py migrate_milestones <program identifier> <jql> <PROJ1 PROJ2 PROJ3 >

Migrate issues to capabilities

This command operates at program level and migrates Jira issues to SpiraPlan artifacts of type capabilities. If the jql is set to empty, i.e., “”, it will take all Epics and Initiative in Jira migrate them to capabilities. This command needs to be run before migrating issues. This is because requirements at project level get connected to a capability when they are created.

python3 main.py migrate_capabilities <program identifier> <jql>

Migrate to components

This command migrates all components in your Jira projects to components in SpiraPlan

This needs to be run before migrating issues, since issues associate to a component when they are created

python3 main.py migrate_components <product identifier> <PROJ1 PROJ2 PROJ3>

Migrate versions to releases

This command migrates all versions in your Jira project to releases. In Jira an issue can have a list of versions, however SpiraPlan can only have one. The tool will choose the first one in the list.

This needs to be run before migrating issues, since issues associate to a release when they are created

python3 main.py migrate_releases <product identifier> <PROJ1 PROJ2 PROJ3>

Migrate issues to requirements, incidents, and tasks artifacts

This command migrates all issues in the set jql and inserts them into SpiraPlan with the correct artifact type. The command only handles one product to migrate to at a time. It is important to insert the artifacts in spira in correct order. This is because there are dependencies between some of them. For example, a task artifact can be connected to a requirement artifact, hence the requirement needs to be inserted before the task. If all issues are moved in one migration the migrating tool will automatically set the correct order. However, if you decide to divide the migration to one product in several jql-queries, make sure that requirements are moved first and tasks last. It is important that this command is run after migrating capabilities, releases and components because of their internal relations. The order can be customized in the mapping file under the “artifact_type_order” key.

python3 main.py migrate_issues <product identifier> <jql>

Migrate issue links to artifact associations

This command sets the association between two artifacts. In Jira the association is named as “linked to” or “blocked by” etc. However, in SpiraPlan the type of association between two artifacts is not that detailed. The information about what type of association was migrated from Jira is shown in the association comment instead.

The command extracts the issues and its metadata selected by the jql and from that info which and what type of association is needed to be created in SpiraPlan. All issues must be migrated to the project before executing this command.

python3 main.py migrate_associations <product identifier> <jql>

Migrate issue comments to artifact comments

This command migrates all comments associated with an issue. All comment features in Jira cannot be applied in SpiraPlan. For example, in Jira you can link to an issue in the comment, but in SpiraPlan it will be shown as the Jira id with square brackets. All issues must be migrated to the project before executing this command.

python3 main.py migrate_comments <product identifier> <jql>

Migrate issue attachments to document artifact with artifact association

Unlike Jira, where documents only exist as attachments to specific issues, SpiraPlan has a separate artifact type for documents and allows them to be linked with other artifacts through associations. This command migrates the attachments and sets the same associations in SpiraPlan as it was in Jira. The command works by retrieving all product artifacts from SpiraPlan and matching them with the Jira id of the migrated attachments. The association is created when the attachment is uploaded as a document artifact to SpiraPlan. All issues must be migrated to the project before executing this command.

python3 main.py migrate_documents <product identifier> <jql>

Add document artifact association without migrating attachments

This command establishes the association between a document and the linked requirement, incident, or task by verifying a custom property labeled 'Jira id.' The 'Jira id' represents the Jira issue previously linked to the document within Jira. The associations are only set within the SpiraPlan product. This is necessary when artifacts have been cleansed and re-migrated without the removing of associated documents.

python3 main.py add_document_associations <product identifier>

Executing the Full Migration from scratch

This is a checklist to execute a full migration from Jira to one SpiraPlan product.

1. In SpiraPlan:
    a. Create a program
    b. Create a product
    c. Create all users
    d. Create API key for user that will be used in the migration tool and make sure it has all necessary access.
2. In Jira: Check that the user that will be used in the migration tool has all necessary access.
3. Migrate custom lists both at system level
4. Migrate custom lists at template- level
5. In SpiraPlan:
    a. For program capabilities, set priority, status, and type.
    b. For program capabilities, create all custom properties including Jira Id of type text.
    c. For product template requirement, set importance, status, and type.

    d. For product template requirement, create all custom properties including Jira Id of type text.
    e. For product template incidents, set priority, status, and type.
    f. For product template incidents, create all custom properties including Jira Id of type text.
    g. For product template tasks, set priority and type.
    h. For product template tasks, create all custom properties including Jira Id of type text.
    i. For product template documents, create all custom property Jira Id of type text.
6. In config files in migration tool:
    a. Fill in mapping yaml-file.
    b. Copy the ".env.template" file to a new ".env" file.
    c. Set Jira and SpiraPlan connection data in the ".env" file.
7. Migrate milestones
8. Migrate capabilities
9. Migrate releases
10. Migrate components
11. Migrate issues
12. Migrate associations
13. Migrate comments
14. Migrate documents

Monitoring and Logging

Understanding Log Files

TBA

Verifying Progress and Success

TBA

Handling Errors and Exceptions

TBA

Post-Migration Steps

Data Validation and Verification

TBA

Post-Migration Checklist

TBA