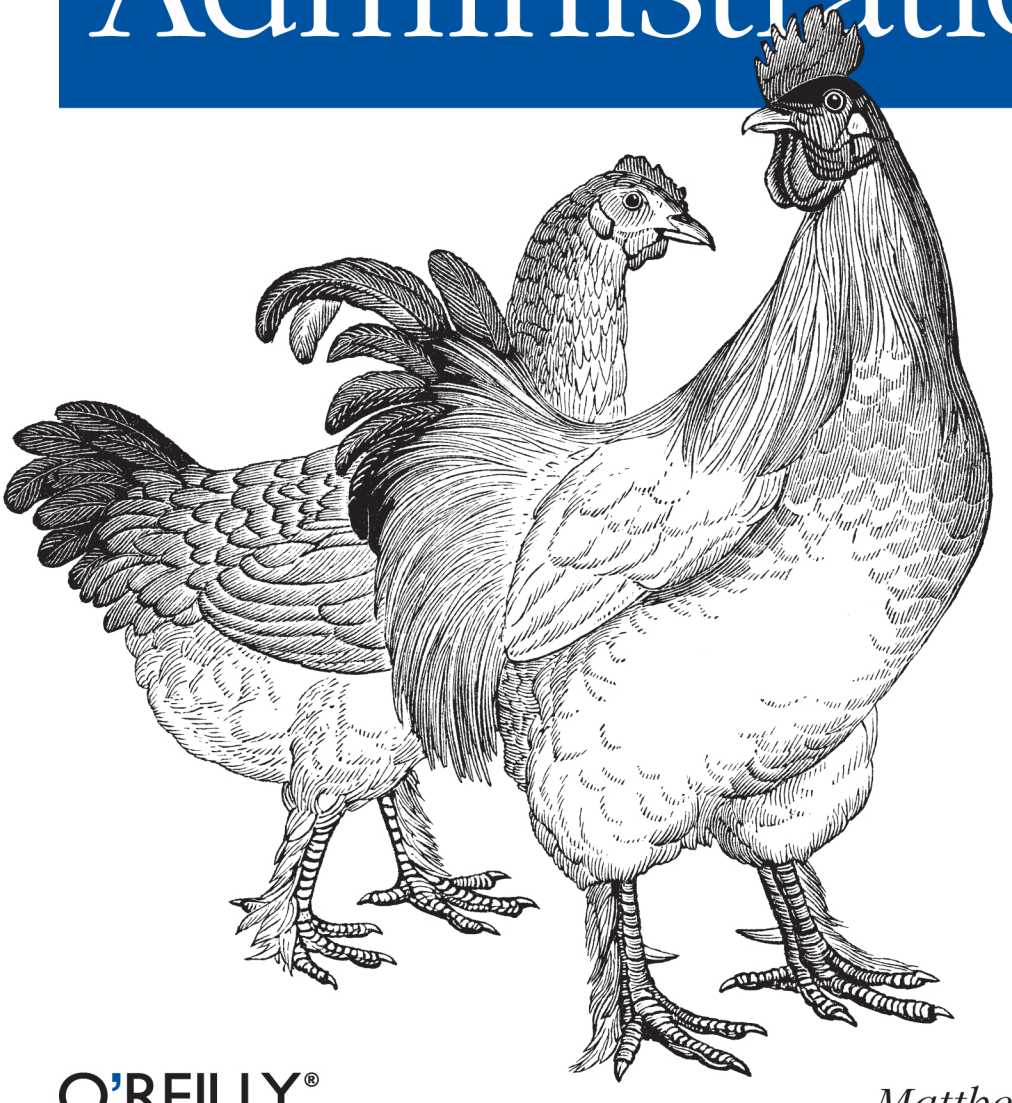
Using JIRA Effectively: Beyond the Documentation

Practical JIRA Administration



O'REILLY®

Matthew B. Doar

Practical JIRA Administration

If you're familiar with JIRA for issue tracking, bug tracking, and other uses, you know it can sometimes be tricky to set up and manage. In this concise book, software toolsmith Matt Doar clarifies some of the more confusing aspects by answering difficult and frequently asked questions about JIRA administration.

Practical JIRA Administration shows you how JIRA is intended to be used, making it an ideal supplement to the extensive documentation already available. The book's chapters are loosely connected, so you can go straight to the information that best serves your needs.

- Understand the difference between JIRA groups and JIRA project roles
- Discover what JIRA schemes do, and learn how to maintain them
- Use a consistent configuration approach to help you use JIRA as a platform
- Create a workflow from scratch
- Add, modify, and deactivate users
- Prepare for a JIRA upgrade, and troubleshoot if necessary
- Get remote access to JIRA via email, SQL, REST, and other methods

"Practical JIRA Administration is about using the power of JIRA to make your project a success!"

—Bryan Rollins
JIRA Product Manager

"Matt's experience with JIRA shows through in the quality of his book, which contains useful tips and overviews rather than a rehash of the documentation."

—Rob Castaneda
CEO, CustomWare

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 **CAN $22.99**
ISBN: 978-1-449-30541-3

9 781449 305413 51999

O'REILLY®
oreilly.com

If we had told people we were going to build a new bug tracker, they would have told us we were completely nuts. A little research into the market would tell you that there are scores, maybe hundreds, of potential competitors, from mega-expensive corporate systems and free open source projects, to on-demand software-as-a-service applications and homegrown tools purpose built to do one thing and do it well. And then there's Microsoft Excel, the all-in-one list builder and charting tool, which is still incredibly popular among small software teams.

Had we considered the massive competition out there, we may have never created JIRA. Fortunately for us, we had some naïveté in our favour, and no one told us *not* to do it. We built JIRA to help us track our own consulting business, which is what Atlassian was in 2001, and in 2002 it became a full-fledged product.

There's two reasons JIRA was successful: an unexpected business model and its flexible architecture. In 2002, Atlassian's sales model was unlike any other business-to-business software tools. It wasn't free like an open source project, but it wasn't expensive either like products from big corporations. It didn't require any professional services to use. And there were no sales people. It caused some confusion in the market. *Can you help us set up an evaluation?* Um, just download it and try it. *How can we make changes to the license agreement?* You can't. It's one size fits all. *How much for a support agreement?* It's included. Free. *Can I send you a purchase order?* Sure, or you can use your credit card. *A credit card? To purchase enterprise software?*

Of course, JIRA's popularity is more than a price point and business model. Most of the developers who started working on JIRA in 2003 are still at Atlassian today, building atop one of the most feature-rich and flexible issue trackers available. Depending on which company is using it, JIRA has been called a bug tracker, issue tracker, defect tracker, task tracker, project management system, or help desk system. It's used by waterfall and agile development teams. It's used by some of the largest corporations in the world to help build their biggest products, and some people use it to manage their personal cross country moves. The permissions system has allowed JIRA to work for both private and public-facing projects.

An ecosystem has been built up around JIRA. As of the time of writing this foreword, there are 273 commercial and open source plugins to JIRA on the Atlassian Plugin Exchange, and hundreds of other integrations built by companies for in-house use or by vendors who sell complementary products. We're extremely excited for Matt's book, too. Matt has been a terrific partner who has built custom integrations for JIRA, extending it far and beyond. In some ways, this book is another plugin to JIRA, helping customers to squeeze more value from the application. It's sure to provide assistance to all the aforementioned customers—the big companies and the small ones, the ones looking to configure it as a bug tracker, and those looking for project management tool.

The final word is about our customers who have pushed the product, our product and support teams, and our imaginations, further then we could have ever done by ourselves. It's been a lot of fun, and for that, we say *thanks, mate*.

Mike Cannon-Brookes and Scott Farquhar, Atlassian co-founders and CEOs

Practical JIRA Administration

Practical JIRA Administration

Matthew B. Doar

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Practical JIRA Administration
by Matthew B. Doar

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Kristen Borg
Proofreader: O'Reilly Production Services
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
June 2011: First Edition.

ISBN: 978-1-449-30541-3

[LSI]

1305913431

Table of Contents

Preface

What This Book Is About

This book is about *JIRA*, the popular issue tracker from *Atlassian*. An issue tracker lets people collaborate more effectively when there are things to be done. You can use an issue tracker for everything from tracking bugs in software to customer support requests, and beyond.

The book is intended for readers who administer a JIRA instance or design how JIRA is used locally. It assumes a basic familiarity with what JIRA can do and provides more information about how JIRA is *intended* to be used.

Each chapter should help clarify some confusing aspect of JIRA administration. The chapters are only loosely connected to each other, with the intention that they can be read in any order. Chapters 1 and 2 are "warm-up" chapters that deal with two specific aspects of JIRA administration. Chapters 3 through 5 cover more system-wide aspects. Chapters 6 through 9 cover other similarly focused areas.

The intention of this book is to supplement but not repeat the extensive JIRA documentation, freely available at *http://confluence.atlassian.com/display/JIRA/JIRA+Documentation*.

In selecting the different topics to cover in this book, I was conscious of the different questions that I, as a software toolsmith, am asked about JIRA every day. I chose the most frequently asked and difficult ones. If you can't find a particular topic and think it should be in a book such as this, then please do contact me with more details.

Some of the topics that are covered are expanded versions of entries already posted to my blog "Practical JIRA Development", at *http://jiradev.blogspot.com*. The chapters which are based on these entries are Chapters 1, 2, and 4.

JIRA Versions and System Details

This book refers to JIRA version *4.2.4 Standalone*, which was released in February 2011. Where there are differences between versions of JIRA (or for JIRA Studio or JIRA WAR/EAR), these are noted in the text.

The system used throughout this book is a Linux server with JDK 1.6 and MySQL. The main differences for other operating systems, deployment types, databases, and JVMs are the installation instructions and the names and paths of certain files. These details are described in the online JIRA documentation.

Development Environment

This book was written using OSX 10.6.6 on a Mac Mini 2.1, using DocBook 4.5, Emacs 22.1.50.1 and Subversion 1.5.2. The output files were generated using a custom remote toolchain provided by O'Reilly for their authors. Using a remote toolchain makes it easier to use DocBook and allows books to be updated more frequently.

Technical Reviewers

Stafford Vaughan
: Stafford started using JIRA in 2005 after completing a Software Engineering degree in Australia and joining CustomWare, Atlassian's leading services partner. He is a founding author of Atlassian's official JIRA training course materials, and has spent the past five years delivering training to hundreds of organizations worldwide. Stafford currently lives in San Francisco and works in Silicon Valley.

Bryan Rollins
: Bryan is the Product Manager for Atlassian JIRA.

Paul Slade
: Paul is a member of the Atlassian JIRA development team.

Matt Quail
: Matt is a member of the Atlassian JIRA development team.

Matt Silver
: Matt Silver has worked in the technical support field for 10 years and now works for O'Reilly. He's an avid rock drummer and lives in Northern California.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

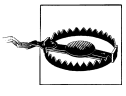
`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Administration→System→System Information
: Shows menu selections within JIRA, in this case the Administration menu item, the System menu item and then the System Information menu item.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Practical JIRA Administration* by Matthew B. Doar (O'Reilly). Copyright 2011 Matthew B. Doar, 978-1-449-30541-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites,

download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9781449305413

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

People at Atlassian who have been particularly helpful over the years include Jonathan Nolen, Sarah Maddox, and Jessie Curtner. The Atlassian founders Mike Cannon-Brookes and Scott Farquhar, and all of the JIRA team have always been responsive and encouraging.

Within the Atlassian community, Jamie Echlin, Nic Brough, Neal Applebaum, and Leonid Maslov stand out for the number of times they've answered questions from myself and others in the *JIRA Developer's Forum*. Other experts that I have benefited from discussions with include Andy Brook, Jonathan Doklovic, David Fischer, Jobin Kuruvilla, Bob Swift, Vincent Thoulé, and David Vittor. Many thanks to all of you, and see you at the next AtlasCamp!

My sincere thanks also go to all the clients of Consulting Toolsmiths for directly and indirectly providing me with the knowledge of which parts of JIRA confuse many JIRA administrators.

Behind all I do is my dearest wife Katherine and beloved children Lizi, Jacob, and Luke. Thank you all, and may the love of God rest and remain with you always.

CHAPTER 1

Groups Versus Roles

Overview

The difference between JIRA groups and JIRA project roles seems to confuse many JIRA administrators. This chapter explains the differences and what each one is good for.

Up until a few years ago, JIRA had users and groups of users, but no project roles. Groups were pretty powerful—wherever you could do something with a user, you could generally use a group instead.

For instance, if you wanted to allow a specific user `john.smith` to change the Reporter field in a project's issues, you could:

1. Create a new permission scheme named something like "`john.smith` can change Reporter".
2. Next, add the `john.smith` user to the appropriate Modify Reporter permission entry in the new permission scheme.
3. Change the appropriate project to use the new permission scheme.

You could also do the same thing with a group:

1. Define a new JIRA group named "Can Modify Reporters".
2. Add the user `john.smith` to the new group.
3. Create a new permission scheme named something like "Added an extra group of users that can change Reporter".
4. Add the *group* (instead of the user) to the appropriate Modify Reporter permission entry in the new permission scheme.
5. Just as before, change the appropriate project to use the new permission scheme.

Both of these approaches now allow `john.smith` to change the Reporter field. So far so good, but there are two main problems with using JIRA groups like this: scaling and updating.

Scaling

If you want `john.smith` to be able to edit the Reporter field in some projects, and also allow a different user, `jane.bloggs`, to do the same thing in other projects, then you have to create two permission schemes, one for each user being granted this permission. If you then decide that they are both allowed to edit the Reporter in some shared projects, then you need a *third* permission scheme. With lots of users, this leads to an explosion in the number of permission schemes (and any other JIRA scheme that supports groups). Keeping track of the difference between each of these schemes is tedious and error-prone, even with the scheme comparison tools (Administration→Scheme Tools).

Updating

As time passes, users may need to be part of different JIRA groups. A project lead usually knows which groups a user should currently be part of. However, only JIRA administrators can change the membership of JIRA groups, which means extra maintenance tasks for them that could be better handled by JIRA project leads.

Project Roles

What was needed to avoid these problems with JIRA groups was another level of indirection,* and that's exactly what JIRA project roles are. Figure 1-1 shows the basic idea.

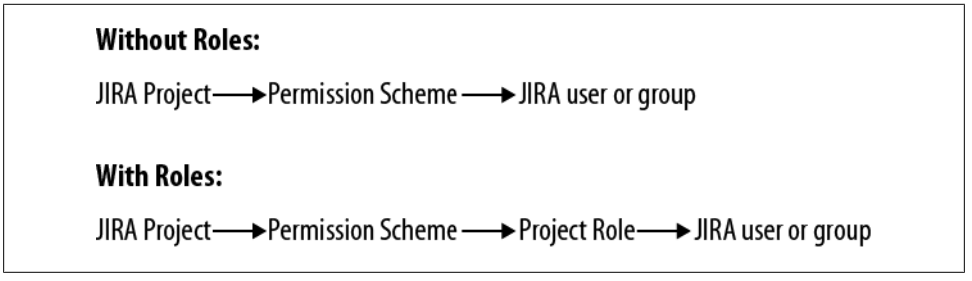


Figure 1-1. JIRA project roles

JIRA has three default project roles: *Administrators*, *Developers*, and *Users*. The current members of these roles for each project can be seen at Administration→Projects: click on the project name, then Project Roles, and then View Members.

* "All problems in computer science can be solved by another level of indirection." —David Wheeler, the inventor of the subroutine.

The number and names of the roles can be changed at Administration→Project Role Browser, but for now let's stick with the three default roles. Every JIRA project has the same set of project roles all the time. The default members of each role for new projects are shown in Figure 1-2.

Project Role Name	Users	Groups
Administrators	None	jira-administrators
Developers	None	jira-developers
Users	None	jira-users

Figure 1-2. JIRA default roles and their memberships

For each role in every project, you can define who plays that role by adding or removing a user or a group for the role.

For example, you could add an individual contractor using JIRA to the Users role for only the projects they need to work with.

Once you've chosen who plays each role for each project, you can use the roles in your schemes. For instance, when you look at the default permission scheme you'll see that all of the permissions are granted to project *roles*, not directly to users or groups. The significant thing about roles is that they can be changed *for each project* by people who are in the Administrators role but aren't JIRA administrators. These people can now create versions and components for their project without needing to change the underlying configuration of JIRA.

To put all that another way:

Who can change a project's versions and components?
: The users who have the Administer Projects permission.

Which users have the Administer Projects permission?
: People in the "Administrators" project role.

Which users have the Administrators project role?
: Members of the *jira-administrators* group and anyone else that you add to that role for a project.

Who can change other parts of JIRA's configuration?
: Only members of the *jira-administrators* group, not users who have the Administrators project role.

Creating a New Project Role

Another way to understand what's going on here is to create a new project role. Let's say that for some reason, we want to allow the technical publications ("Tech Pubs") user assigned to each project to modify the Reporter of an issue.

The default permission scheme already allows users with an Administrator role in a project to modify the Reporter of an issue. But we don't want to allow the Tech Pubs user to administer the whole project: we just want to give them that one specific permission.

We can create a new role *Documentation*, at Administration→Project Role Browser. We can also add our Tech Pubs lead `bobby.jones` as a default member in the "Users" column of the new project role so that he will be in the Documentation role for all new projects by default.

Now every JIRA project has this new role. When a new JIRA project is created, it will have the `bobby.jones` user in the Documentation role for the project. For existing projects, we can manually add the appropriate Tech Pubs user (or group) to the Documentation role for each project. Once the users for this role have been added, we can edit the appropriate permission schemes and add the Documentation role to the Modify Reporter permission entry. The permission scheme now checks which users are in the *role* for each project, rather than looking at a fixed list of users or groups of users.

If the Tech Pubs person changes for the project, then the people in the project Administrator role can change the members of the Documentation role for just that project. There is no need to ask the JIRA administrator to make the changes.

For more information about using project roles to control which users can view which projects, see "Hiding Projects from Users" on page 28.

From the other direction, you can also see which roles an individual user has in all the JIRA projects: go to Administration→User Browser, find the user, and click on Project Roles.

Summary

JIRA groups are made up of JIRA users and can only be changed by JIRA administrators. But JIRA project roles are made up of JIRA users and JIRA groups and can be changed per project by project administrators. Project administrators are all the users in the Administrators role for a JIRA project.

Should I use a group or a project role?

If you want to refer to the same set of more than six users across multiple projects, use a group. If you want to refer to a set of users that is potentially different per project, use a project role. Also, don't add new roles without considering whether the existing ones can be used in the permission scheme to accomplish what you are trying to do.

Further Reading

http://confluence.atlassian.com/display/JIRA/Managing+Groups discusses JIRA groups in general.

http://confluence.atlassian.com/display/JIRA/Managing+Project+Roles discusses JIRA Project Roles specifically.

Some of the background information to this chapter can be found at *http://confluence.atlassian.com/display/JIRA/Migrating+User+Groups+to+Project+Roles*, along with the documentation for the JIRA Scheme Comparison Tools. Unfortunately, the scheme tools only work with Permission and Notification Schemes.

CHAPTER 2

Resolved, Resolution, and Resolution Date

Overview

One thing that sometimes confuses both JIRA users and administrators is the difference between the Resolved status and the Resolution field. This chapter clears up some of the confusion between these very similar-sounding terms. The differences are summarized at the end of this chapter ("Summary" on page 10).

Getting this right is important, because many of the standard JIRA reporting gadgets expect the Resolution field to be set as expected—otherwise confusing results occur. For example, gadgets that refer to the date when issues were resolved use the Resolution Date field, which is in turn based on the Resolution field.

Resolved

JIRA comes with a default workflow (Administration→Global Settings→Workflows) named "jira", shown later in Figure 5-1 and summarized below in Figure 2-1. This workflow has the following statuses for an issue, shown in the order they commonly occur.

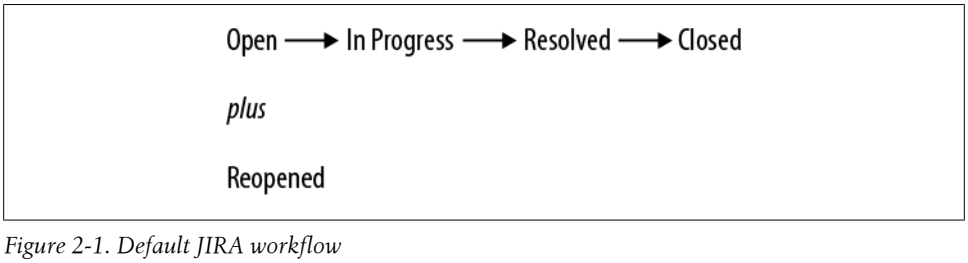
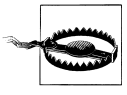


Figure 2-1. Default JIRA workflow

The idea is that an issue (such as a bug) is created with a status of *Open*, and is then moved to *In Progress* and then *Resolved* by the person who fixes it. The bug is then moved to either *Closed* or *Reopened* by someone who checks whether it really was fixed or not. So "Resolved" is just a name for an issue status. The status could just as well have been named "Believed Fixed" or "Ready for Testing".



If you create a workflow from scratch (see Chapter 5), the Resolved status is not connected in any way with the Resolution field

Resolution

It's generally a good idea to keep the number of statuses in your workflow as small as possible to make maintenance easier. It makes sense to avoid having lots of statuses with names like:

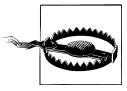
"Closed and Fixed"
"Closed and Won't Fix"
"Closed Because It's A Duplicate"
"Closed Since..."

The Resolution system field (Administration→Issue Settings→Resolutions) can be used to avoid having lots of similar statuses. The default values for Resolution of `Fixed`, `Won't Fix`, `Duplicate`, `Incomplete`, and `Cannot Reproduce` cover many of the reasons that a bug could be closed, and you can change this list if necessary.

The intended use of the Resolution field is that when a bug is created, the field is empty, with *no value at all*. This is displayed in an issue as a value of "Unresolved". When an issue is moved to a status such as Resolved or Closed, the Resolve Issue Screen is usually shown during the transition. This screen includes the Resolution field, ready for the user to set a value. A bug can have its Resolution set to "Fixed" while it is moving to the Resolved status, for example.

JIRA expects the Resolution field to be set in any status you would consider an issue resolved, and that the Resolution field should be cleared in all other statuses. It's up to you to make sure this happens, with a transition screen or post-functions when you create or modify workflows (see Chapter 5).

In the default JIRA workflow, the Resolve Issue Screen is the only screen where you can set the Resolution field for an issue, and this screen is only used for transitions to the Resolved and Closed statuses. In your own workflows you are responsible for making sure that the Resolution is set. Once the Resolution has been set, the issue is considered *resolved* by JIRA, even if the status of the issue is not the Resolved status but some other status defined by you (such as "Deployed").

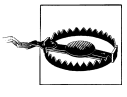


The only way to remove the resolution from an issue in a standard JIRA installation is to *Reopen* an issue. Adding the Resolution field to the Default Screen to make it easier to change is a bad idea, because it's a required field and will get set for issues that aren't yet resolved.

Another approach to clearing a Resolution field in an issue is to create transitions back to the same status that have a post-function to clear the Resolution. More details on this can be found at How to Clear the Resolution Field.

Resolution Date

The Resolution Date system field is the latest date that *any* value was set in the Resolution system field for that issue. Changing the issue status from Resolved to Closed also used to change the resolution date, but this was fixed in JIRA 4.3.1. The Resolution Date field will only be empty if there is no value at all in the Resolution field.



The Resolution Date is confusingly named *Resolved* in the list of Issue Navigator columns and some gadgets. This has nothing directly to do with the status named "Resolved".

Other Approaches

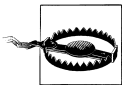
Many organizations using JIRA don't use the Resolution field for a number of reasons:

- It's hard to reset the value to empty for unresolved issues.
- It doesn't appear on standard screens, only during transitions.
- It's hard to make the Resolution field not required on custom transition screens.

Instead, they base their reporting only on the name of the statuses in the workflows. They may also create their own custom Select List field named something like *Our Resolution* or *Sub-status*, with its own values such as `Unresolved`, `Fixed`, `Won't Fix`, `Duplicate`, `Incomplete`, and `Cannot Reproduce`.

The drawback of not using the system Resolution field as JIRA intended is that many of the standard JIRA reporting gadgets are no longer useful. Making sure that the Resolution is set correctly in your workflow is a better approach in the long run (see the section "Implementing a Workflow" on page 38).

A helpful approach to consider in custom workflows is creating a transition for each desired resolution, with a post-function in each transition to set the resolution *without* using a screen. For example, there could be a transition from the current status to Closed named "Duplicate", which *automatically* sets the resolution to `Duplicate`.



Adding a resolution named "Unresolved" to the system Resolution field is a bad idea, because since the Resolution field now has a value, the issue will still be treated as resolved by the standard JIRA gadgets.

Summary

- "Resolved" is just the name of one of the issue statuses in the default JIRA workflow.
- Resolution is a system field with values of `Fixed`, `Not Fixed`, and so on. The Resolution field is usually set during transitions.
- An unresolved issue is one with *no value* in the Resolution field.
- The Resolution Date system field is the latest date that the Resolution field was changed to any value.

Further Reading

Some of the points in this chapter are mentioned briefly at *http://confluence.atlassian.com/display/JIRA/Configuring+Workflow#ConfiguringWorkflow-About%27open%27and%27closed%27issues*.

Clearing the Resolution field is covered at *http://confluence.atlassian.com//display/JIRA/How+to+clear+the+resolution+field+when+the+issue+is+reopened*.

The problem of the Resolution Date changing when an issue's status is updated is described in the JIRA bug JRA-20286.

The *Created vs Resolved Gadget* described at *http://confluence.atlassian.com/display/JIRA/Adding+the+Created+vs+Resolved+Gadget* is a good example of where confusion can occur about what "Resolved" actually means.

CHAPTER 3

Understanding Schemes

Overview

Schemes are a major part of configuring JIRA, but they are also one of the most confusing parts of JIRA. This chapter is intended to clear up some of that confusion. Chapter 4 has a worked example of how schemes can be used.

A JIRA *scheme* is a collection of configured values that can be used by more than one JIRA project. For example, a Notification scheme describes who receives what kinds of email when issues are changed. The same Notification scheme can be used by more than one JIRA project. In fact, the Default Notification scheme is used by all JIRA projects unless you configure a project differently.

The seven schemes that are being used for a particular JIRA project can be viewed and edited by editing the project (go to Administration→Projects and click on the project name, not Edit).

We'll cover the four schemes that are similar to the Notification scheme first and look at the remaining three (more complex) schemes later. The top-level page in the Atlassian documentation for all of this information is *http://confluence.atlassian.com/display/JIRA/Defining+a+Project*.

Project-Wide Schemes

The four schemes like the Notification scheme that apply to the whole project or to *all* issue types within a JIRA project are:

Notification Scheme
: Who receives what email when an issue changes

Permission Scheme
: Who can do what to an issue

Issue Security Scheme
: Who can even view an issue

Issue Type Scheme
: What issue types a JIRA project can use

First we'll take a quick detour to see how JIRA refers to users in such schemes.

Adding Users to Schemes

There are a dozen different ways (shown in Figure 3-1) that JIRA lets you specify a set of users, but happily the same ways can be used in both Notification and Permission Schemes. Issue Security schemes use a subset of these choices.

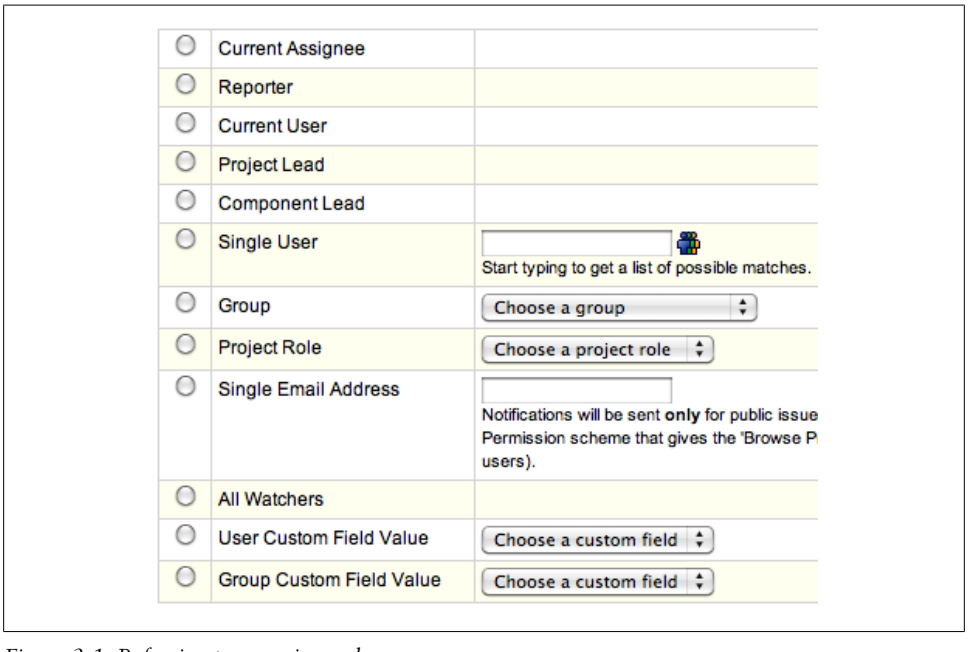


Figure 3-1. Referring to users in a scheme

The simplest of these are:

Current Assignee
: The user that the issue is currently assigned to.

Reporter
: The reporter of the issue.

Current User
: For permission schemes, the user who is logged in.

Project Lead
Component Lead
: The project lead is specified in each project's settings. Component leads are optionally configured for each component in a project.

Single Email Address
: A specific email address. This only works for issues that can be browsed anonymously.

All Watchers
: All users listed in the system Watchers field for an issue.

The not-so-obvious ways are:

Single User
: A username such as `john.smith`, *not* their full name such as "John Q. Smith".

Group
Project Role
: In general, use a Project Role instead of a Group, as discussed in Chapter 1.

User Custom Field Value
: Use the contents of a custom field of type User Picker or Multi User Picker. Such a field might be populated during a transition or by editing an issue.

Group Custom Field Value
: Use the contents of a custom field of type Group Picker or Multi Group Picker. In a notification scheme, all the members of these groups will receive email, so be careful about how many users are involved.

Notification Schemes

A notification scheme controls who receives what email about changes to JIRA issues. A reasonable default notification scheme comes with JIRA.

It's much easier to add changes to a notification scheme than to undo them. So always keep an unchanged copy of the default notification scheme as an easy way to undo any changes you make later on.

JIRA uses an *event-driven* model for its notifications. When something such as a comment or a status change happens to a JIRA issue, a specific kind of *Event* is sent within JIRA. Another part of JIRA listens for events and acts on them when they are received. For example, when the JIRA system Mail Listener (Administration→Listeners) receives an event from an issue, it uses the notification scheme for the issue's project to decide who should receive the email. This process is summarized in Figure 3-2. The actual set of users who are sent email for each different event can be defined in the various ways listed in "Adding Users to Schemes" on page 12.

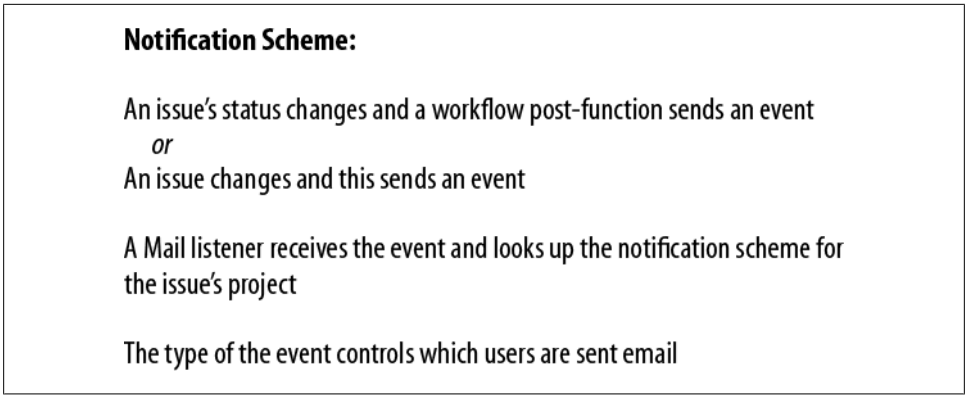


Figure 3-2. Notification Scheme

You can define your own custom events at Administration→General Configuration→Events, and then change the post-function in a workflow transition to make it send ("fire") the new event. The new event will appear as a new row in all the notification schemes, and you can then define who should receive email when that transition takes place.

Note that you cannot configure the type of event sent for non-workflow issue operations such as Assign or Comment.

It's important to avoid spamming your users with too much email, or they'll just filter it and miss useful information. Be careful how many users you add to a notification scheme.

For more information, see the documentation at *http://confluence.atlassian.com/display/JIRA/Creating+a+Notification+Scheme*.

Permission Schemes

A permission scheme is how you configure who is allowed to do what to a JIRA issue. There are a number of fine-grained permissions, such as "Create Issues" and "Edit Issues". Each of these permissions has a set of users that are granted that permission, as shown in Figure 3-3. Just like notification schemes, this set of users can be configured in the various ways described in "Adding Users to Schemes" on page 12.

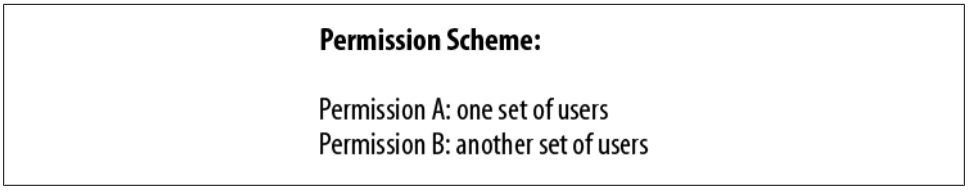


Figure 3-3. Permission Scheme

Just like other schemes, it's much easier to make changes to a permission scheme than to undo them. Keep an unchanged copy of the default permission scheme as an easy way to undo any changes you make later on.

Once defined, such permissions are used by JIRA in various ways. The most obvious permissions do what they say (e.g., "Link Issues" controls whether a user can link one issue to another). Other permissions such as Resolve Issue and Close Issue can be used in workflow conditions to control who can change an issue's status.

However, some of the permissions have effects that are not as obvious at first glance. For instance, when editing an issue, the Resolve Issue permission is needed to see the Fix Versions field, and the Schedule Issues permission is needed to see the Due Date field. If the user does not have those permissions, then these fields are hidden—not just grayed out—in an issue's edit screen.

For more information, see the documentation at *http://confluence.atlassian.com/display/JIRA/Managing+Project+Permissions*.

Issue Security Schemes

An issue security scheme controls who can view or edit a *specific issue*. In practice, most JIRA projects don't need to have an Issue Security scheme defined, which is why this scheme is set to "None" by default when you create a new project.

Within an issue security scheme, you can define one or more "security levels" as shown in Figure 3-4. There is actually no hierarchy involved in these levels; they're really just sets of users. Each security level can have users assigned to it in the same way described in "Adding Users to Schemes" on page 12. You can also choose one level to be a default security level for the scheme.

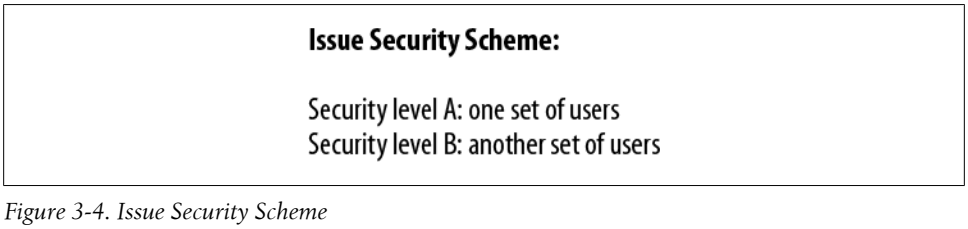


Figure 3-4. Issue Security Scheme

There is a system field in all issues named "Security Level", which contains a list of all the different levels configured in the issue security scheme that are active for that issue's project. Once a security level has been set in this field in an issue, then only users in that level can view or edit the issue. For other users, the issue is invisible and doesn't appear in searches—or recent issue lists either.

To be able to set the Issue Security field in an issue to anything but the default value, you need to have the Set Issue Security permission ("Permission Schemes" on page 14). The default permission scheme does not give any user this permission.

As an example of using an issue security scheme, the Atlassian support website for JIRA at *https://support.atlassian.com/browse/JSP* uses an issue security scheme that only allows the Reporter and Atlassian staff to see each issue. That way, confidential information in a support request from one customer is not seen by a competitor who also happens to be using JIRA.

For more information, please see the documentation at *http://confluence.atlassian.com/display/JIRA/Configuring+Issue+Level+Security*.

Issue Type Schemes

A JIRA project's issue type scheme controls which issue types are valid in that project, as shown in Figure 3-5. For instance, most JIRA projects that are not used by developers don't want to have the Bug issue type to be shown as a choice anywhere.

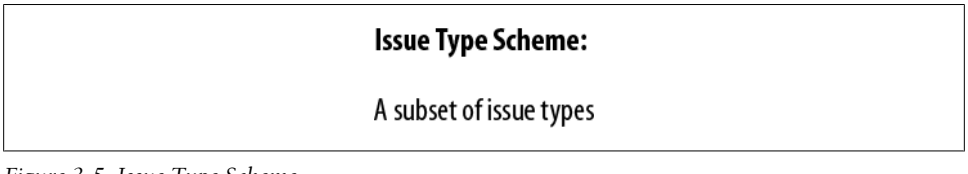


Figure 3-5. Issue Type Scheme

You can define an issue type scheme by going to Administration→Issue Settings→Issue Types and clicking on the tab *Issue Types Scheme* (not "Issue Type Schemes" as you might expect). This location is not as obvious as it might be. You can also set the default issue type that will be used when someone is creating an issue, and even change the order of issue types the user will see.

Changing any scheme for lots of projects is generally a long and repetitive task (see "Managing Custom Fields" on page 69), but not for issue type schemes. The *Associate* link in the list of issue type schemes allows you to select multiple projects to change at once.

For more information, see the documentation at *http://confluence.atlassian.com/display/JIRA/Associating+Issue+Types+with+Projects*.

Schemes That Use Issue Types

Every JIRA project has three other schemes whose behavior depends upon the issue type of an issue. For example, the fields for a Bug may be quite different from the fields for a Task—this is defined in a field configuration scheme.

These schemes are more complex than schemes that don't depend on the issue type, and they're the schemes that usually confuse JIRA administrators. One way to keep the two kinds of schemes separate is to remember that these three schemes are the last three of the seven schemes listed when you're editing a project's schemes.

The schemes that use an issue's issue type are:

Workflow Scheme
: Which workflow is used for each issue type

Field Configuration Scheme
: Which fields are part of each issue type

Issue Type Screen Scheme
: Where are the fields displayed in an issue

All of these schemes have the concept of a default. This is what is used if an issue type is not specifically mentioned in the scheme.

Different JIRA projects can all have totally different schemes, but to make maintenance easier you should always try to define schemes that can be reused by more than one project. This is discussed in "Working with Schemes" on page 21.

Workflow Schemes

JIRA is designed to support different workflows. A workflow scheme defines which workflow is used for each issue type, as shown in Figure 3-6. A default workflow can also be chosen in a workflow scheme, and this workflow will be used for all issue types that aren't specifically mentioned in the scheme.

The workflow scheme is an example of a common naming pattern for these three JIRA schemes: the Foo *Scheme* is the mapping from an issue type to a particular *Foo* thing.

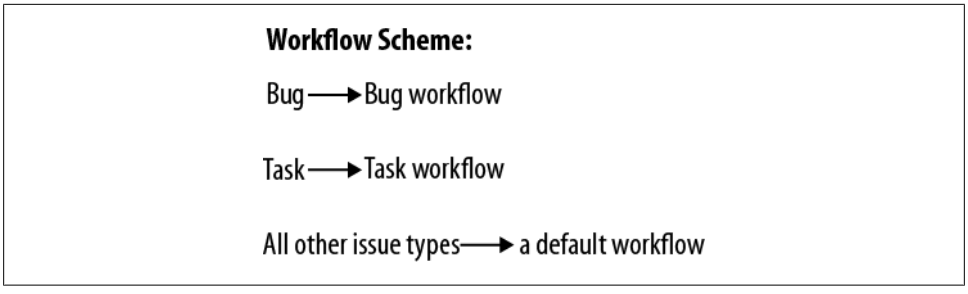


Figure 3-6. Workflow scheme

One common practice is to start with a single workflow for the default and to then add workflows for specific issue types as necessary. For example, begin with the "Default Workflow", then add a "Bug Workflow", a "Task Workflow", and so on.

For more information, please see the documentation at *http://confluence.atlassian.com/display/JIRA/Activating+Workflow*.

Field Configuration Schemes

A field configuration scheme defines which Field Configuration is used for each issue type, as shown in Figure 3-7. A default field configuration can also be set.

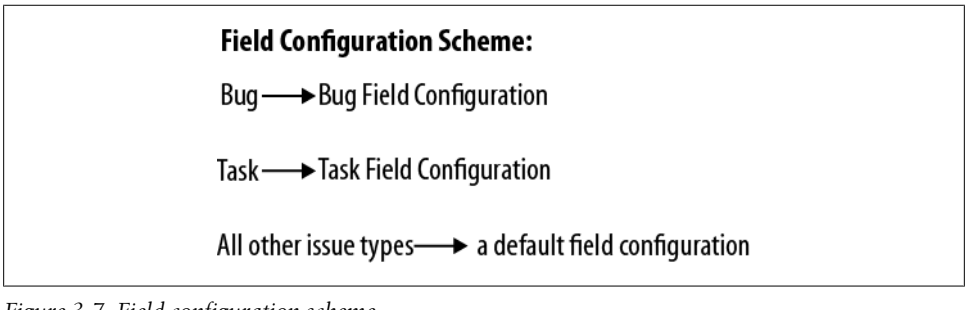


Figure 3-7. Field configuration scheme

A *Field Configuration* (which is not a scheme) is a list of all the possible fields in any issue, with further configuration so that each field is valid and therefore "shown", or invalid and "hidden". For instance, the Fix Versions field is useful for a Bug but maybe not for a Task, so it would be configured as hidden in the Task field configuration.

When you edit a field configuration, you'll notice that every possible field is listed, whether or not the field is restricted by issue type or project. This is the expected behavior.

Each field can also be marked as "Required", which means that it can't be left empty. JIRA will try to not allow you to make a field both required and hidden.

You can change a field's description in a field configuration, and also change the way that the contents of the field are displayed using various renderers. The renderers can show the contents of a field as raw text or can treat the text as wiki markup. If your users complain about JIRA mangling characters in the description, check which renderer is being used.

For more information, see the documentation at *http://confluence.atlassian.com/display/JIRA/Associating+Field+Behaviour+with+Issue+Types*.

Issue Type Screen Schemes (ITSS)

I've saved the most complex and confusing scheme for last. All the other JIRA schemes either control the behavior of an entire JIRA project (permission, notification, issue security, issue type schemes) or control the behavior differently per issue type. An Issue Type Screen Scheme (ITSS) controls how the fields are displayed for an issue using *two* levels of indirection instead of just one. This is shown in Figures 3-8 and 3-9.

Other things to note about this scheme are that an "Issue Type *Screen* Scheme" is entirely different from a "Issue Type Scheme", and also that the phrase "Screen Scheme" is a great tongue-twister. Try it!

Recall that a field configuration scheme and its field configurations define the fields that are valid for a particular project and issue type. Screens are what control how these fields are *viewed* by a user.

On the first level of indirection we have Screen Schemes, which are not used directly by a JIRA project. Each Screen Scheme refers to up to three screens—one for creating an issue, one for viewing an issue, and one for editing an issue. This is so that you can have the same set of fields displayed in a different order on each screen (or not show certain fields during issue creation if they don't make sense there).

I usually define a screen scheme for a single issue type. For example, you might have a Bug Screen Scheme that defines how to display the fields for a Bug during creation, viewing, and editing an issue, as shown in Figure 3-8.

At the second level of indirection, an ITSS defines which screen scheme should be used for each issue type. A default screen scheme can also be chosen in an ITSS.

For more information, see the online documentation at *http://confluence.atlassian.com/display/JIRA/Associating+Field+Behaviour+with+Issue+Types*.

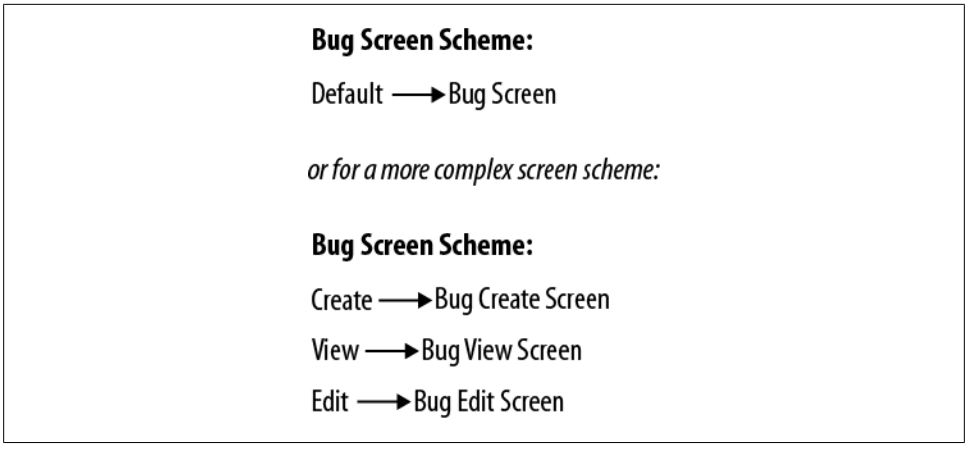


Figure 3-8. Screen scheme

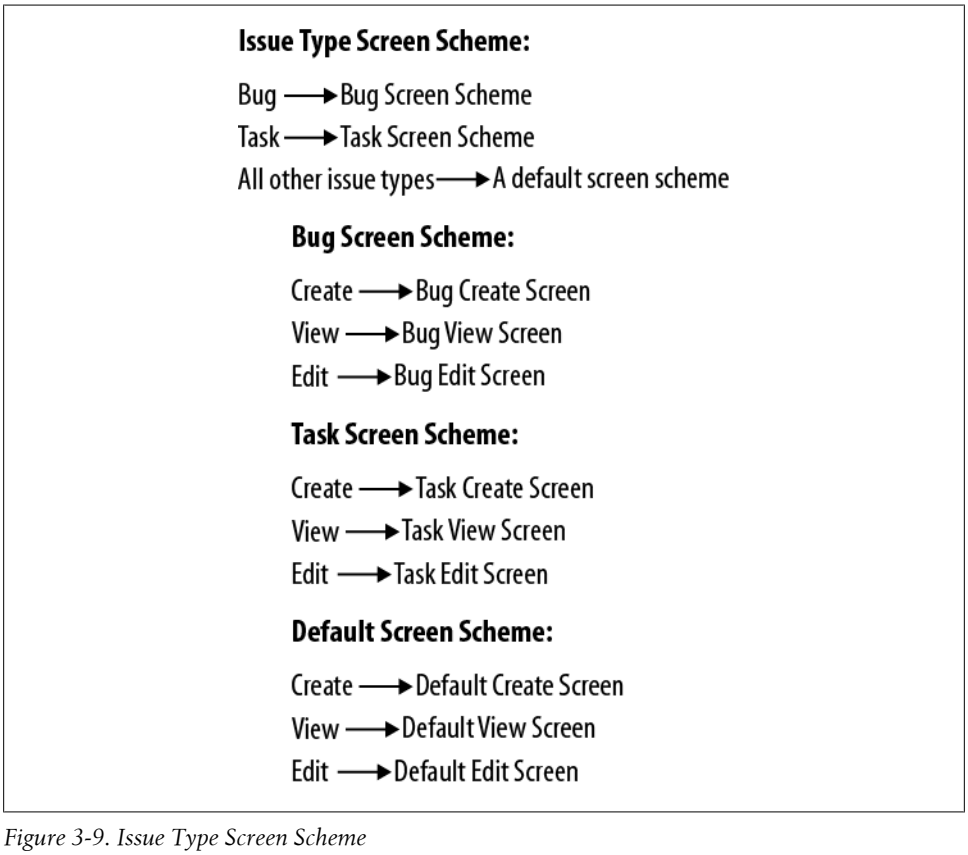


Figure 3-9. Issue Type Screen Scheme

Working with Schemes

Even once you've understood what the different schemes do in JIRA, they will still need to maintained. Every time that someone asks you to add a new field just for them, you will want to consider the effects of that change on everyone else's issues. Chapter 4 looks at how to make sure this happens in a controlled way, and the rest of this chapter covers some of the details of making this possible.

There is a necessary balance between the number of schemes and what they all do. I try to have only as many schemes as are needed by the different communities using a particular JIRA instance. You can configure JIRA so that every project has a complete set of schemes, but that's usually overkill and just makes for a lot of maintenance work later on. Also, if every group in an organization has a different process, then working together is going to be that much harder.

The most important thing to do before changing schemes is to *create a backup of your JIRA data*. If possible, test all scheme changes on a development server. Even better, do both.

Documenting Schemes

Each scheme has a name and an optional description. What are good names for schemes? Obviously it's a personal preference, but I usually name my schemes using a Project Category or Issue Type name, and then the scheme type.

For example, I might have a project category named "Customer Support" for all of the Customer Support department's JIRA projects. For an ITSS used by all those projects, I would use a name such as "Customer Support ITSS"—and then this scheme would refer to screen schemes with names like "Support Request Screen scheme" and "Training Screen scheme", using the issue types for the screen scheme names.

I also use the scheme's description field to record the latest change at the beginning, just after the summary of what a screen is for. The maximum length of the description varies according to the underlying database, but you can assume at least 4000 characters.

Sometimes I also add version numbers to a scheme's description or name, and update these when I change the scheme. This is useful, but I recommend adding the date as well, since you may end up with branched schemes. Only workflows record the date and user who last changed them.

If I'm really feeling paranoid or the JIRA configuration is particularly complicated, then I may even create a document describing the intended purpose of each scheme and the changes that have been made to it over time. This helps me when I'm trying to work out the effects of a proposed change to a particular scheme.

Debugging Schemes

Sometimes you have to try to understand how a JIRA instance has been configured by another person whose documentation and naming of schemes was perhaps minimal. This usually happens when someone asks you something like "why can't I edit this field anymore?"

The first thing to do is to see if the names of the various schemes bear any resemblance to what they are actually used for. Just because a scheme is called "Bug Workflow scheme" doesn't mean that it's only (or even) being used for bugs. The fastest way to get this information is to go to each scheme administration page and look at the projects that each scheme is assigned to. With luck, you'll spot an obvious pattern. If the scheme names don't end in "scheme", consider adding that to make it clear that it's a workflow *scheme*, not a workflow (for example).

You may also want to compare two schemes to see how they differ. There is a convenient tool for comparing permission and notification schemes available at Administration→Schemes→Scheme Tools. For other scheme types, I find that opening each scheme in a separate tab in my browser allows me to compare them reasonably well side-by-side.

If you decide to do a wholesale renaming of the schemes in your JIRA instance, then I recommend making a backup (of course) and then renaming just one collection of similar projects at a time (such as those with the same project category). Renaming old scheme names with an obvious prefix such as "OLD" can also help you spot the cases that you've missed. Since you've got a backup, you can also consider deleting inactive schemes.

Once you have a better understanding of which schemes exist and what they're used for, you can debug the original problem. My process for debugging most scheme problems is as follows:

1. Obtain a specific issue key where the problem can be reproduced.
2. If you can, get a sense of when the problem began, since it may be related to some other scheme change you've just made.
3. Note the project, issue type, and status of the issue, and also whether the problem occurs during creating, viewing, or editing the issue.
4. Go to Administration→Projects and click on the project name.
5. Note the names of the seven schemes that are currently being used by this project.
6. For each scheme in turn, view the details of the individual scheme and see what applies for the specific issue type. Note this information as well.
7. If the problem is about a field, then view the appropriate field configuration and the create, edit, or view screen. The field may be hidden, required, not present on the screen, or present but in another screen tab.

Some fields are only visible if the user has the correct permission, so you may not have the same problem as an administrator. Try creating a user account with the same groups and project roles as the user reporting the problem. Alternatively, log in as the user who reported the problem using the *JIRA SU (Switch User) Plugin*.

8. If the problem is about a workflow action, check the specific transition's conditions and validators first.
9. If the problem seems related to some other scheme, then drill down into that scheme's definition, bearing in mind the issue type and where the problem was seen.
10. Once you have identified the root cause and fixed it, revert the fix and confirm that the same problem reappears, then fix it again. This confirms that your analysis and changes are correct. You should also check other schemes where the same problem may exist.

Don't make any change to schemes before making a backup. Ideally, you should debug and fix the problem in a development instance of JIRA before touching the production JIRA.

The Future of Schemes

Schemes have always been a powerful but somewhat confusing part of JIRA. They're not going to go away, but Atlassian is actively investigating ways to make them easier to use. A sample of how JIRA might present all of the various schemes' information in the future can be seen in the JIRA Project Configuration Prototype plugin's home page (*https://plugins.atlassian.com/plugin/details/34407*).

This work is also typical of how Atlassian makes large feature changes to JIRA. First a plugin is developed that offers the functionality in parallel to the existing functionality, but in a harmless way. Feedback and fixes occur rapidly and then the plugin is bundled with the shipped JIRA package. Many of the core features of JIRA are in fact plugins when you look at their source code.

CHAPTER 4

JIRA as a Platform

Overview

A common request that JIRA administrators receive is to use JIRA for more than its current purpose. The typical case is that someone in one group tells a different group that "you can do that with JIRA, and it's already installed." It's true that JIRA can be used to track many different kinds of issues, and Chapter 3 described how to configure JIRA schemes to do just that.

This idea is, in effect, using JIRA as a *platform* for different web applications or "vertical solutions" for each group of users. There might be one such web application for Engineering JIRA projects, one for Customer Support JIRA projects, and so on. Using JIRA as a platform in this way is part of how it is designed to be used, but it does need a consistent configuration approach to be successful. This is particularly true if different groups don't want to see any part of other groups in the same JIRA instance.

However there isn't much documentation on how to do this in a *consistent* manner. This chapter describes one way to do this using a worked example, and then summarizes this in "Summary" on page 33.

What Can Be Configured

For each pair of a JIRA project and issue type, we can change the following:

- Which system and custom fields an issue can use, and whether they are required or not
- Whereabouts on an issue screen the system and custom fields appear
- The workflow for an issue, including the statuses available in an issue

On a per-project basis, we can also configure:

- The issue types used in the project
- The components and versions available for an issue
- The permissions for what a user can do with an issue, including even knowing the issue exists
- Who can access the whole project
- Who receives email about changes to issues

What Is Configured System-Wide

Some configurations for JIRA are system-wide and affect all the users of a JIRA instance. Such configurations are not part of using JIRA as a platform, but they may have a bearing in discussions between groups because changing them affects everyone. Some of the more common ones that I encounter when discussing this topic are:

- The logo and colors used by JIRA
- Names of system fields; any translation of a field name or status applies everywhere (Administration→Issue Settings→Statuses or Issue Types, Translate)
- Whether unassigned issues are allowed or not
- The maximum attachment size, which is set to 10MB by default
- Priorities; all issues use the same list of priorities in the system priority field
- Resolutions; all issues use the same list of resolutions in the Resolution field (this is most commonly seen during a workflow transition)

For the last two, there is a popular JIRA feature request (*JRA-3821*) to make priorities and resolutions configurable per project and issue type.

All the other system-wide configurations, such as enabling or disabling voting, can be found at Administration→Global Settings→General Configuration.

Worked Example: Configuring JIRA for a New Department

In this example, we are going to configure JIRA for use by an imaginary accounting department. The information stored is totally different from what appears in a Bug issue type, and includes an example custom field named "Amount". Only certain people can see the accounting information in JIRA, and some of this information is still further restricted. The accounting department also requested that they should see nothing about Engineering projects, since that was just unnecessary clutter on their screens.

The first thing to do is to take a backup of your JIRA data, do this work on a development JIRA instance, or both. The next things to do are:

1. Create a new Project Category for the accounting department, e.g., "Accounts". Some scheme names will use this word as a prefix, so make sure that the category name is obviously unique and meaningful.
2. Create a new issue type for that department's issues, for example, "Invoice". Add a description of what kind of information it contains. Other scheme names will also use this word as a prefix, so make it meaningful.
3. Create a test project with a project key such as `ACCTEST`. JIRA project keys should generally be as brief as possible since everyone types them frequently.* Once this project's configuration is complete, you can create more JIRA projects and configure all of them in the same way as this project.

Basic JIRA Project Setup

The next stage is to do the simplest part of the job first. Edit the project configuration with Administration→Projects, then click on the project name `ACCTEST` (not on Edit).

Project Lead

Set the Project Lead for the project. This user will be the default assignee for issues in the project. With the default notification scheme, email about new issues is sent to the assignee, reporter, and watchers, so the project lead should expect to receive email about issues assigned to them.

Project Category and Avatar

Set the Project Category for the project to the new Accounts category that we just created. Any other future accounting projects, such as `ACCMAIN` or `ACCSUB`, will also use this category.

It's also a nice touch to set an avatar (a small logo) for the project to make it easy for people to quickly distinguish it from other projects. You can upload your own images. One possible idea is to use the same image for all projects in the same category.

Notification Scheme

Open a new browser tab and create a new notification scheme named *Accounts Notification scheme*. Copying the Default Notification Scheme and modifying the copy is the most common way to do this. The name of this new scheme indicates what the

* You can also edit the *jira-application.properties* file to allow numbers in the project key, if that helps.

scheme is (a notification scheme) and which category of projects it is intended for (Accounts).

In your original browser tab, set the notification scheme for the project to the new notification scheme, *Accounts Notification scheme*.

Permission Scheme

Create a new permission scheme named *Accounts Permission scheme*. Again, copying the Default Permission Scheme and modifying the copy is the most common way to do this.

There should be no need to change any of the permissions except for one. The Set Issue Security permission controls who can change an issue's security level (see "Issue Security Schemes" on page 15). Add the Administrators role to this permission. As discussed in Chapter 1, we want to use a project role rather than a group here.

Set the permission scheme for the project to the new permission scheme, *Accounts Permission scheme*.

Groups and Roles

Define a new JIRA group named "Accounting" that contains all the users who should be able to see issues in the Accounts projects.

In the project configuration page, click on Project Roles: View Members, and then:

- Delete the `jira-users` group from the Users role and add the Accounting group.
- Delete the `jira-developers` group from the Developers role and add the Accounting group.
- Add the project lead as a user to the Administrators role.

Hiding Projects from Users

At this point only Accounting users can see the Accounting projects, which is as intended, but they can still see Engineering projects. This is because all of those projects are likely using the `jira-users` group in their Users role.

To change this, we need to step back and look at how we are defining the Users role for all of the JIRA projects. The Users role is what usually controls who can access a project. There is no way to explicitly block access to a project for a specific group in JIRA. So the approach I usually take is that all users should be members of the default `jira-users` group, but also members of a group that controls which projects they can see.

In this case I would define a group Engineering, add all the engineers to it, and then change all the Engineering projects' Users and Developers roles to use this group instead of `jira-users`.

Now the users in the Accounting group won't have access to the Engineering projects, and those projects won't clutter up JIRA for the Accounting users.

Issue Security Scheme

Create a new issue security scheme named *Accounts Issue Security scheme*:

- Add a security level named "All Accounting" and add the Accounts group to it. Make this level the default one.
- Add another security level named "Confidential Accounting" and add only the accounting users who are permitted to see the more confidential accounting information.

Set the issue security scheme for the project to the new issue security scheme, *Accounts Issue Security scheme*.

Advanced Project Setup

Now we need to define the more complex schemes and configure the `ACCTEST` project to use them. These schemes are:

- Issue Type scheme
- Workflow scheme
- Field Configuration scheme
- Issue Type Screen scheme, which uses at least one Screen scheme

Issue Type Scheme

An Issue Type Scheme controls which issue types can be used in a project.

Under Administration→Issue Setting→Issue Types, click on the Issue Types Scheme tab to create a new issue type scheme named *Accounts Issue Type scheme*. Then:

- Add the main accounting issue type *Invoice* as the default issue type.
- Add other issue types, such as *Task* and *Improvement*, only if they will be used by the new department. You can reorder them to change the order in which they appear when a user is creating an issue. The default issue type will be shown as selected at that time.

Now set the issue type scheme for the `ACCTEST` project to the new issue type scheme, *Accounts Issue Type scheme*.

Workflow Scheme

Create a new workflow for the Invoice issue type named *Invoice Workflow* and add the desired statuses and transitions to the new workflow. See Chapter 5 for more details on how to create a new workflow.

Create a workflow scheme named *Accounts Workflow scheme* and configure it to use the new workflow for Invoice issue types. For any other issue types that are allowed in the project, add their workflow mappings in *Accounts Issue Type scheme*.

Set the workflow scheme for the `ACCTEST` project to be the new workflow scheme, *Accounts Workflow scheme*.

Field Configuration Scheme

A field configuration controls which fields are part of an issue type, e.g., what data is part of an Invoice.

Create a new field configuration named *Invoice Field Configuration*. This is *not* a scheme. Don't hide any fields here yet since we'll use screens to do that later on. If a particular field is required in an Invoice, mark it as such here.

Create a new field configuration scheme named *Accounts Field Configuration scheme*, and configure this new field configuration scheme to use the *Invoice Field Configuration* for the Invoice issue type.

Now set the field configuration scheme for the `ACCTEST` project to the new field configuration scheme, *Accounts Field Configuration scheme*.

Screen Scheme

Screens control whether a field appears in an issue to a user, and also the order in which the fields appear. Screen Schemes choose which screen is used to create, edit, or view an issue.

Create a screen named *Invoice Screen*. This screen should have all the fields that are wanted in the Invoice issue type, including the custom field Amount (after it is defined in "Adding a Custom Field" on page 31). You can add more than one field at once, and then reorder them in one go using "Move to Position".

I recommend starting with just one screen and using it for all three of the screens (Create, View, and Edit). Later on, you can copy and edit the screen and change the screen scheme without having to change the project settings. A good reason to have different screens is that some fields may not be known when an issue is created, or there might be fields that are not directly editable by users.

Create a new screen scheme named *Invoice Screen scheme* and configure the Create, Edit, and View issue screens to all be the same screen for now. This can also be done by changing the default to use just one screen.

Issue Type Screen Scheme (ITSS)

An Issue Type Screen Scheme (ITSS) ensures that the right sets of screens are used for each issue type.

Create a new ITSS named *Accounts ITSS* and configure the default screen scheme to be the *Invoice Screen scheme* defined in "Screen Scheme" on page 30. If there are other issue types, then add mappings for each one to an appropriate screen scheme. For more information, see "Issue Type Screen Schemes (ITSS)" on page 19.

Now set the issue type screen scheme for the `ACCTEST` project to the new issue type screen scheme, *Accounts ITSS*.

Adding a Custom Field

Adding a custom field is the real test of all this work, since you'll probably do it more than once for all the projects in a category. The custom field for this example is named "Amount".

Define the new custom field with Administration→Issue Fields→Custom Fields and then the Add Custom Field link. Give the field a name and a description. Since the description appears just below the field in the issue screens, make it useful for people by describing what they are expected to enter, perhaps along with an example value. For example, *The dollar amount owed, with no dollar sign, e.g., "15.95"*.

Since the accounting department will want to be able to search on this field, make sure that the searcher template is not set to "None".

Now restrict the custom field to just the applicable *issue types* that uses it. For this example, that's just the Invoice issue type.

Don't restrict the custom field to a *project*, because then you'll have to come back and do that for every JIRA project that you add to the Accounts category. If you have lots of custom fields, that will take a long time to do manually.

Go to Administration→Issue Fields→Screens and add the new custom field to the *Invoice Screen* (or to the *Invoice Create*, *Invoice View*, and *Invoice Edit* screens if they were defined in "Screen Scheme" on page 30). To ensure that this new field doesn't interfere with other projects and their issues, don't add the new field to any other screens.

This is the end of the worked example. Note that when you're looking for the Amount field to use in a search in the Issue Navigator, you will have to choose a project and an issue type in order for that the custom field to appear as a choice.

Names Used in the Example

This section lists all the different names used in the example above in one convenient place:

Accounts
: A project category

ACCTEST, ACCMAIN, ACCSUB
: The keys of three JIRA projects in the Accounts project category

Accounting, Engineering
: Groups of JIRA users

Users, Developers
: The standard JIRA project roles

Invoice
: A new issue type

Amount
: A custom field in Invoice issues

The seven schemes and the things they control are:

Accounts Notification scheme
: The notification scheme for Accounts JIRA projects

Accounts Permission scheme
: The permission scheme for Accounts JIRA projects

Accounts Issue Type scheme
: The issue type scheme for Accounts JIRA projects

- Invoice—a new issue type used in the ACCTEST JIRA project
- Task, Improvement—existing issue types

Accounts Issue Security scheme
: The issue security scheme for Accounts JIRA projects

- All Accounting—a security level in the issue security scheme
- Confidential Accounting—another security level in the issue security scheme

Accounts Workflow scheme
: The workflow scheme for Accounts JIRA projects

- Invoice Workflow—a custom workflow for the Invoice issue type

Accounts Field Configuration scheme
: The field configuration scheme for Accounts JIRA projects

  - Invoice Field Configuration—the field configuration for the Invoice issue type

Accounts ITSS
: The issue type screen scheme for Accounts JIRA projects

  - Invoice Screen scheme—the screen scheme for the Invoice issue type
  - Invoice Screen—the screen used for the Invoice issue type by the Invoice Screen scheme

Summary

The key to using JIRA for many groups is to have a standard way of using JIRA schemes and issue types. The details of the approach used in the example in this chapter are:

- The project category is used as the common theme for related projects.
- The naming of schemes uses the category or issue type names.
- Field constraints are implemented using issue types not projects.
- Use project roles in preference to groups.
- Document what you do!

Of course there is a balance to be struck with any approach. Too few schemes, and every change will have unwanted consequences. Too many schemes, and you risk losing track of how they differ.

CHAPTER 5

Creating a Workflow from Scratch

Overview

Workflows are the different statuses that an issue can have, together with the transitions between the statuses. For instance, there could be a status named *Open*, with transitions leading to the *Resolved* and *Closed* statuses.

The word "status" is preferred over using "state" in the JIRA documentation (and also in this book), but in practice they seem to both be used interchangeably. One useful notion is that a "status" is a summary of "states". For example, someone's medical status could be summarized as "normal" based on the state of their heart, the state of their liver, and so on.

JIRA workflow transitions can also optionally have *conditions*, *validators*, and extra *post-functions*:

- Conditions restrict who can see that a transition exists.
- Validators check the values that were entered during a transition.
- Post-functions make changes after a transition has taken place and send events to say what just happened. JIRA automatically adds certain post-functions to every transition.

The statuses and transitions of the default JIRA workflow are shown in Figure 5-1, which is taken from *http://confluence.atlassian.com/display/JIRA/What+is+Workflow*. The expected sequence over time goes from the top left to the bottom right.

One of the major attractions of JIRA is the ability to customize workflows, including adding new statuses, new transitions, and making other things happen as part of transitions. This chapter describes how to create such a JIRA workflow from scratch.

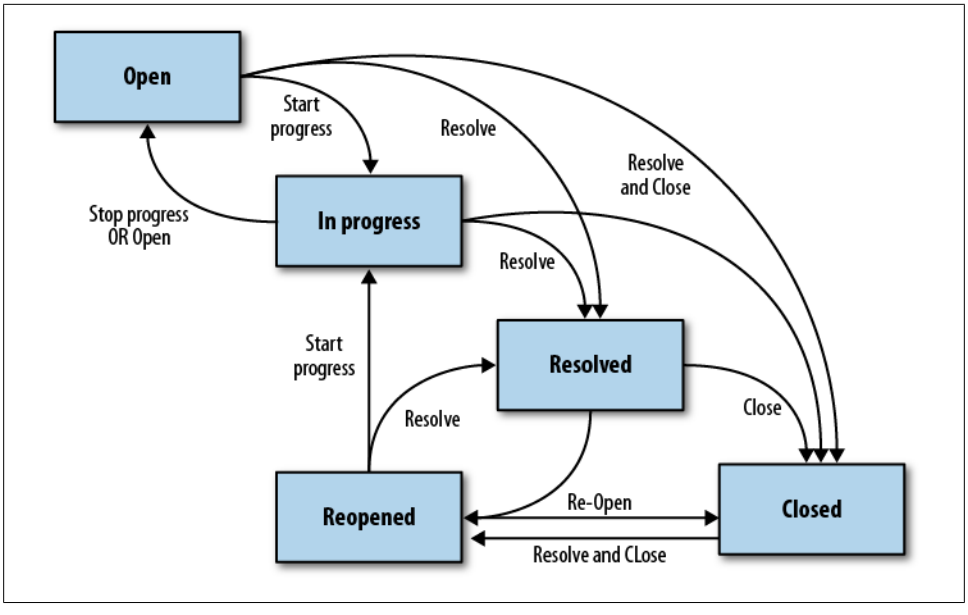


Figure 5-1. The default JIRA workflow

Why would you want to do that? JIRA comes with one default workflow. You could create your custom workflow by simply copying and modifying the default workflow. However, I've found that doing this often leads to maintenance problems including:

- The names of your statuses may not be the same as the ones in the default JIRA workflow, and renaming them doesn't completely hide the old names everywhere in JIRA.
- The JIRA *Closed* status, by default, has a property that does not allow issues in that status to be edited (`jira.issue.editable`). It's easy to forget that the property is there until you want to bulk change many issues, some of which may be in the *Closed* status.
- Some of the default JIRA transitions are *common* transitions, meaning that changing them in one place also changes them everywhere else they are used in a workflow. This isn't very clear from the standard JIRA workflow editor, so at least for simple workflows, it's better to not have any common transitions at all.

Designing a Workflow

The hardest part about designing workflows is getting everyone to agree on them in the first place. After all the smoke and noise is over, all that is really needed is:

- The issue types that are expected to use this workflow.
- A list of the status names and their descriptions.

- A list of the transitions with names and their descriptions.
- What, if any, information needs to be entered during each transition. Such information may be needed for restrictions of who can see or execute a transition. This information is used by the optional conditions and validators.
- Any changes to an issue other than status as part each transition. These changes are typically made using post-functions.

Both statuses and transition names should be as brief as possible.

Some general guidelines for designing workflows that I find helpful are:

Use the past tense of verbs for statuses
: For example "Closed", instead of "Closing" or "To Be Closed". The name of the status should describe what has *already happened* to an issue that has that status.

Use the imperative tense for transitions into a status
: For example "Close", instead of "Closing" or "To Be Closed". The name of the transition should describe you want to do to the issue.

Don't use transition names that are existing action names
: There is a standard issue action called "Assign" that assigns an issue to a different user. If you give a transition that same name, then your users will see two tabs on an issue, both named "Assign", that do very different things.

Fewer statuses is better
: Use the smallest number of statuses possible. More than about ten suggests you may have overcomplicated the workflow, which in turn means that other people will have a hard time using it. You may need to have more than one issue type and a workflow for each issue type.

  One way to decide whether you need a status is to consider what report would use it, and whether that same information can be provided by JIRA in a different way.

Clear descriptions
: Make sure that you have a brief description of the purpose of every status and transition, and enter it the JIRA workflow. This will appear as a floating tooltip to guide confused users.

Put statuses and transitions in their expected order
: Whenever you list the statuses during design or are adding them to a JIRA workflow, do so in the order that you expect them to appear most of the time. Similarly, add the most common transition first in any list of transitions from a status. An alternative way to change the order that transitions appear in using properties is described at *http://confluence.atlassian.com/display/JIRA/Configuring+Workflow*.

Workflow screens
: A transition can either happen immediately, or a transition screen with various fields on it can be shown during the transition. Transition screens also have a place to leave a comment about why the transition was made. JIRA provides two standard screens for this (Resolve Issue and Workflow), but if there are fields that you

want to allow to be edited during a transition, then you can define your own screens. Just don't use the Default screen as a transition screen, because when users change statuses they'll feel rather overwhelmed by seeing *all* the fields in the issue.

Which statuses are resolved?
: Decide which statuses should always have a value set in the system Resolution field (Chapter 2). Make sure that all transitions from a non-resolved status to a resolved status set the Resolution, and that all transitions from resolved to non-resolved statuses clear the resolution.

Allow for mistakes: no status is final
: Every status should have a transition to another status: otherwise, it becomes a final status. Final statuses are fine until someone accidentally moves an issue into one and then can't undo their mistake. You can always only allow administrators to execute a transition if you want to make it more difficult to change a status.

For many workflows, I find that thinking about the intended assignee for each status is helpful when designing the workflow. For instance, a bug might have been assigned to a default user for a project, then to a developer, then to QA for testing, and finally assigned to someone in Operations for deployment as part of a release. I try to consider what each person will want to do most frequently with the issue.

Implementing a Workflow

Once you have the names and descriptions of the statuses and transitions, you can create the new workflow at Administration→Global Settings→Workflows. For ideas on naming the workflow, see "Workflow Schemes" on page 17 and "Workflow Scheme" on page 30.

The *JIRA Workflow Designer* plugin is a graphical tool that makes implementing workflows easier. JIRA 4.0 to 4.3 are supported and the Early Access Program releases of JIRA 4.4 suggest that this will be included with JIRA as a new feature. While this is a great help, the underlying details of workflows are unchanged, including the limitations of draft workflows (see "Further Reading" on page 41).

First, create new statuses as necessary at Administration→Issue Settings→Statuses. The icons chosen for each status will be shown next to an issue's status wherever it appears in JIRA.

Now add the statuses to the workflow in the expected order of their most frequent use. A workflow is actually made up of *steps*, and each step has just one *status* associated with it. For simplicity, make the step names the same as the status names—otherwise, your users will see discontinuities in a few places in JIRA.

JIRA will have added a first step named "Open". After you add other steps, you will be able to make any one of them your initial status if you want to, and can then delete the original step that JIRA added for you. To change the initial status, click on the Open step name, then the Create Issue transition, then Edit, and finally change the Destination Step to the new initial status.

Next, add the transitions away from the first status, also in their expected order of use.

For each transition, after you've entered the name and description, check which conditions, validators, and post-functions are wanted, and add them.

You are allowed to have transitions back to the same status if you want to. This is one way to narrowly restrict what is changed in an issue, and is used in the section "Resolution" on page 8.

The default JIRA workflow has some conditions, validators, and post-functions that are worth knowing about:

- The initial Create Issue transition into Open has a validator to check that the user has the Create Issues permission.
- The Start Progress transition has a condition to check that the current user is the issue's assignee. Other users won't see this transition as a choice.
- The Closed status has the `jira.issue.editable` property set to `false` which means that issues with this status can't be edited.
- Many statuses and transitions have a property `jira.i18n.title` which is used to get the actual name. If you're having problems renaming something, look for this property, and either delete it or translate the status' name at Administration→Issue Types→Statuses.

There are five post-functions that are added by default to new transitions, but only one of these is editable: "Fire Generic Event". Events are discussed later in "Workflows and Events" on page 40.

Deploying and Testing a Workflow

When a workflow is created from scratch, there is of course no project or issue type that is using it, so it's *inactive*.

The first step towards making a workflow active is to create a workflow scheme to define which issue types use each workflow. For instance, tasks (issues with issue type *Task*) could have a different workflow from bugs, which have an issue type of *Bug*. See "Workflow Scheme" on page 30 for details on a recommended way to do this.

Once you have a workflow scheme that refers to the new workflow, you can edit a JIRA project to use the workflow scheme (go to Administration→Projects and click on the project name).

Now when you create a new issue of the specified type in that project, you should see that the status of the issue is the one that you chose as the initial status. The available workflow choices for the issue should be the transitions that you defined as possible from that status.

To test the workflow, execute the transitions between all the statuses, checking for usability errors as much as any actual failures or error messages. Check that any custom conditions, validators, or post-functions behave as expected. Manually testing all the different combinations of transitions and user permissions is only really possible for small to medium-sized workflows.

To make a change to a workflow once it's in use and active, you have to create a draft of the workflow, edit the draft, and finally publish the draft. The option of saving a copy of the original workflow is offered when the workflow is published, and can be useful if version numbers are added to the workflow name. However, I generally find it leads to too many copies of old workflows, so I don't use it.

One thing that's currently missing in JIRA is a way to compare two versions of the same workflow. When I really want to be sure of what has changed, I export the workflow's XML before and after the change and then compare the two files using a *diff* tool, preferably one that understands XML.

Workflows and Events

JIRA sends software "events" internally when issues are changed. Some of these events are hardcoded, such as the one sent when an issue's assignee changes. However, events sent during a transition are designed to be configurable. Many of the events listed at Administration→Global Settings→Events are really intended for use in workflows. For example, the "Work Started on Issue" event is intended to be sent ("fired") by a post-function on all transitions into the "In Progress" status.

The standard post-function "Fire Generic Event" can be edited to send a more appropriate event when a transition executes. The main reason that a JIRA administrator cares about what type of events are sent is because they are used by a project's Notification Scheme (see "Notification Schemes" on page 13), which controls who receives email when the status of an issue changes.

You can also add new types of events to JIRA at Administration→Global Settings→Events, as described in detail at *http://confluence.atlassian.com/display/JIRA/Adding+a+Custom+Event*.

The ability to create new events and have your workflow fire them off instead of the Generic event or some other standard event can be useful for trimming JIRA spam. For example, if you really want to fine-tune who receives email when an issue changes status, you can define a new event type for each transition, perhaps giving them highly-descriptive names such as *Task Workflow: Open to Resolved Event*. (The event names don't appear in email templates.) Then you can edit the transition from Open to Resolved, and change its post-function to fire the appropriate new event. In a custom notification scheme, you can then specify which users will receive email for precisely that one transition and no other transitions.

Further Reading

The documentation for configuring workflows can be found at *http://confluence.atlassian.com/display/JIRA/Configuring+Workflow*.

The process of changing a workflow so that issues in the Closed status can be edited is described at *http://confluence.atlassian.com/display/JIRA/Allow+editing+of+Closed+Issues*.

The limitations of how draft workflows can be changed are documented at at *http://confluence.atlassian.com/display/JIRA/Configuring+Workflow#ConfiguringWorkflow-Limitations*.

Details of adding a new event to JIRA are at *http://confluence.atlassian.com/display/JIRA/Adding+a+Custom+Event*.

CHAPTER 6

The User Lifecycle

Overview

One of the areas that a new JIRA administrator commonly feels uncertain about is adding, modifying, and deactivating users in JIRA. This chapter covers some of the different aspects of the lifecycle of a JIRA user.

Adding Users

Before someone can log into JIRA, a JIRA administrator has to create a user account for them. A JIRA administrator is someone in the `jira-administrators` or `jira-system-administrators` groups,* not someone in the project role *Administrators*.

JIRA has an internal directory service of user accounts, but there are also a number of other ways to define JIRA users.

The most common request is to have JIRA work with user accounts that have already been defined for other tools and network domains. For example, many organizations have a Microsoft Active Directory (AD) server where users and email group aliases are added. JIRA can use such directory services in a number of different ways:

Authentication
: The passwords for JIRA users are the same as their passwords in the directory service. This is usually the easiest one to set up.

Authorization
: The different groups that JIRA users belong to are defined in the directory service. This often means that changes to group membership have to be performed by the directory service administrators.

* The differences between these two groups are described at *http://confluence.atlassian.com/display/JIRA/Managing+Global+Permissions#ManagingGlobalPermissions-sysadmin*.

Provisioning of User Accounts
: The user accounts in JIRA are automatically created when they appear in the directory service. Automatic deactivation of accounts requires that the groups are also defined in the directory service.

Atlassian has a directory service product (Crowd, available at *http://www.atlassian.com/software/crowd*) that integrates with JIRA and all the other Atlassian tools. This allows you to manage all your Atlassian users in one place. Crowd can synchronize its list of users from another directory service such as AD.

As of JIRA 4.3, much of the user management aspect of Crowd has been merged into JIRA. Crowd is still required for single sign-on (SSO) or large numbers of users. Extracting a list of users from Crowd is not a straightforward process.

For more detailed information, please see the documentation at *http://confluence.atlassian.com/display/JIRA/Managing+Users*.

Modifying Users

The only constant is change, so you should expect JIRA user accounts to need updating. If a user is defined in the JIRA internal directory service, then changing a user's email address and full name is easy enough to do: go to Administration→User Browser, find the user, and click Edit.

JIRA doesn't check for duplicate email addresses for its users, so you can create two users with the same email address. However, both JIRA users may be sent email from within JIRA, which will result in receiving duplicate emails at the same email address. Also, if an email message is used to create a comment on an issue (see the section "Email" on page 57), JIRA assumes that the comment was sent by the first user whose email address matches the *From* address in the email.

As an aside, a user's full name can also be used to contain information about their affiliation (e.g., "John Smith (Example Company)"). The full name can be up to 255 characters.

Changing a Username

A name change after marriage is a common reason for a request to change a JIRA username, sometimes also referred to as the "userid" or "Username". The full name ("Jane Smith") of the user can be changed easily enough, but unfortunately changing the username (`jane.smith`) is harder to do cleanly. Leaving the username unchanged may not fit with corporate IT policies, or may just be seen as too confusing for other users.

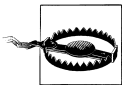
If you have simply created a user and made an error entering the username, deleting the user and starting again is the best approach. But once a JIRA user is active, then deleting a user is not allowed while there are issues assigned to the user or reported by the user. Deleting a user also removes useful historical information from issues.

If you really need to change an active username, then the following steps may help:

1. First, create a new JIRA user account with the correct username.
2. Make sure that the old and new user accounts are members of the same JIRA groups.
3. Delete any filter subscriptions for the old account and recreate them for the new account. You'll have to log in as the user to see how they are defined.
4. If the old account is used in any notification or permission schemes or workflows, then change those too.
5. If the user is a project lead for any projects, then change those projects to use the new account. JIRA also checks for this when deleting a user.
6. If the user might have been added to a project role directly, perhaps as a project administrator, change the role to use the new account. You can check this at Administration→User Browser: find the user, and click on Project Roles.

Once the new account is ready, then you can use a search to find all the issues assigned to the old account and assign them to the new account using Tools→Bulk Change. You'll also need to do the same thing for issues reported by the old account. Both of these changes require you to have the right permissions for all the issues involved, or the bulk edit action will be unavailable.

By now you're probably thinking "surely I can just do this with some SQL?" Selectively updating just some of the tables in the database is an easy way to corrupt your data—don't do it! However, there is another approach. The *Groovy script* for the *Script Runner* plugin will make the necessary changes directly in the database, more safely. Before using this script, confirm that it has been tested with your version of JIRA just in case new tables have been added.



Another really tedious approach is to take an XML backup, perform a global find and replace on the username string in this large file, and then reimport from the modified backup. At least try to do a *diff* to check that you didn't modify more than you expected. This really is a method of last resort.

A common question from frustrated administrators is "I know each user account has a unique numeric ID in the database `userbase` table, so *why* can't I just change the username?" The answer seems to be that JIRA was written to work with external directory services, and the unique user identifier for them was intended to be the username, not a numeric ID. The username was subsequently used in other database tables where you might have expected to see an ID used instead.

Deactivating Users

When a user should no longer have access to JIRA, they can be *deactivated*. This is preferable to *deleting* them, which removes useful historical information (and is also more work). The term is also less ambiguous than *disabling* someone. A deactivated user cannot log in and also does not appear in lists of users (e.g., for assigning issues).

The steps to deactivate a user are as follows:

1. Remove the user from all JIRA groups. If they aren't a member of the groups that have the JIRA Users permission (Administration→Global Settings→Global Permissions) then they can't log in to JIRA and issues can't be assigned to them. They also don't count towards the total number of users in licenses that only allow a limited number of users.

   Some JIRA administrators create a special group named something like *Deactivated Users*, and add deactivated users to that. So long as that group isn't referred to by any other part of JIRA, that's fine for unlimited licenses.
2. Add a prefix such as "zzz " to their full name. For example, "Roger Rabbit" becomes "zzz Roger Rabbit". This makes the user appear at the end of any list of users sorted by name. It also alerts other users that any issue currently assigned to the deactivated user is unlikely to see any progress in the future.†
3. Once a user leaves an organization, their email address will eventually become invalid and may cause "bounce" errors in the JIRA log files. To avoid this, change the top-level domain in the user's email address from `.org` or `.com` to `.invalid`, which is the official way to mark an email address as invalid (RFC 2606).

Reactivating a deactivated user is just a matter of reversing the above changes—add the user to some groups, and change their name and email back to what they were.

Users in JIRA Studio have an property *access level* with a value of *Developer*, *Collaborator*, or *No Access* that can be set by an administrator. Setting a user's access level to *No Access* is the equivalent of deactivating the user.

Monitoring Users

An often overlooked feature in JIRA is the ability to see which users are logged in, by using Administration→User Sessions. This feature can be useful during upgrades (see Chapter 7) for checking who needs to log out before the upgrade can begin. You can also send email to such users from within JIRA at Administration→Send E-mail.

† To do the opposite and make a user appear at the top of a list, use a prefix such as "(" or "[".

CHAPTER 7

Planning a JIRA Upgrade

Overview

Atlassian has released two new versions of JIRA each year for a few years now and may even increase this rate in the future. Recent releases and their dates are:

JIRA 4.0 - October 2009
JIRA 4.1 - April 2010
JIRA 4.2 - October 2010
JIRA 4.3 - March 2011
JIRA 4.4 - August 2011?

Major versions such as 4.2 and 4.3 are supported for "two years after the last minor iteration of that version is released." So if JIRA 4.2.4, which was released in February 2011, ends up being the final release that was based on JIRA 4.2, then all of the JIRA 4.2.x releases become unsupported from February 2013 onwards.

All of this means that upgrading JIRA on a yearly basis is not uncommon. The process of upgrading JIRA is complex enough to warrant having a chapter dedicated to it, especially since doing it once per year is just long enough to mislay your notes from the last time you did the same job. This chapter describes an upgrade procedure that can be used over and over, though of course it also depends on the specific release notes for each JIRA version, which are published at *http://confluence.atlassian.com/display/JIRA/Production+Releases*.

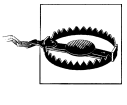
Preparing for an Upgrade

The very first thing to do is to work out which JIRA version you're allowed to upgrade to. JIRA licenses permit you to upgrade to any newer version released during a period when you are covered by a support contract. The first year of support comes bundled with the initial purchase. So if you buy JIRA on July 9th, 2011, then you are allowed to upgrade to any version released before July 9th, 2012. However, if the next version happens to be released in October 2012, then you'll have to renew your support contract to be able to upgrade to it. You can find your support details at Administration→System→License Details.

The next thing to do is to make sure that your backups are working. If you don't have a full *Bare Metal Recovery* procedure to test them with, then at least check that recent backups of the data and attachments are being created and contain some reasonable-looking data.

The hardest part of JIRA upgrades used to be working out whether all third-party plugins were compatible with the new version of JIRA. You had to look up each plugin listed in Administration→Plugins at the Atlassian Plugin Exchange (`plugins.atlassian.com`) by hand. The Universal Plugin Manager (UPM) now does this automatically for you. The UPM is what is seen at Administration→Plugins in JIRA 4.3 and later. It can also be installed as a plugin in earlier versions of JIRA, and in that case will appear at Administration→Universal Plugin Manager.

You may ask "what if I'm using a plugin that isn't compatible with the new version of JIRA?" Check if there is already an open issue at wherever that plugin tracks issues about compatibility (the plugin's issue tracker, not `jira.atlassian.com`), and create a support request if there isn't one already there. If the plugin is supported, you can also try contacting the plugin vendor. Some plugins take months after a new JIRA release to catch up, which is one reason why some people choose to wait for the first dot release (i.e., 4.3.1 instead of 4.3).



The upgrade procedure described below assumes that you have a development instance of JIRA, which is one configured just like your production instance and containing a recent snapshot of the data from the production instance—but which is only used by you. If this is not the case and you only have one JIRA instance (which is your production JIRA), then the upgrade job will take a little less time but is a bit more risky, so make doubly sure your backups are valid.

You may also want to start planning possible dates for the upgrade with your users and other groups. Most upgrades I've done recently seemed to require about two hours of JIRA downtime. In-place database upgrades (see the section "In-Place Database Upgrades" on page 55) and improved installers should reduce this time for large JIRA instances in the future.

Important JIRA Locations

There are at least four locations or services involved in every working JIRA instance. These are:

jira_app
: The *JIRA Install* directory; this is where JIRA was unpacked and is named something like *atlassian-jira-enterprise-4.2.4-standalone* or *atlassian-jira-4.3-standalone*. On Linux, I usually create a soft link named *jira_app* to the install directory from its parent directory to make future upgrades easier, like this:

```
lrwxr-xr-x jira jira 40 May  7 08:00 \
    jira_app@ -> atlassian-jira-enterprise-4.2-standalone
```

jira_data
: The *JIRA Home* directory; this is where JIRA stores any data that isn't in the database. These include files that are added as attachments to issues, files for plugins, and caches of the database data. This is the directory name that is configured in the *jira-applications.properties* file under the install directory.

  This directory can be named anything you wish, but I usually name it something like *jira_data_424* if it is being used for a JIRA 4.2.4 installation. On Linux, I usually create a soft link named *jira_data* to the JIRA home directory at the same level.

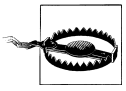
Database
: The database used by JIRA contains both the issues added by JIRA users and the JIRA configuration that is changed by JIRA administrators. The two major sets of files that are not kept in the database are attachments and plugins, which are kept in *jira_data*. Avatars (custom project icons) are also not in the database.

Directory Service
: The directory service is where JIRA keeps its list of users. By default, JIRA uses an internal directory service which is stored in the database. If Active Directory, Crowd, or some other separate directory service is used, then the upgraded JIRA needs to use that too—otherwise no one will be able to log in, including JIRA administrators (see "Adding Users" on page 43).

A General Upgrade Procedure

This section describes a general procedure for upgrades. Since no procedure can fit every upgrade precisely, please use it only as a basis for your own customized upgrade procedure. Briefly, the procedure looks like this:

- Install and configure the new, upgraded JIRA instance ready for data.
- Take a backup of the current JIRA instance and shut it down.
- Import the backup into the upgraded JIRA instance and test it.
- Declare the upgrade complete.



These steps are similar to the ones in the documentation at *http://confluence.atlassian.com/display/JIRA/Upgrading+JIRA*. Where they differ is that upgrading the existing database (see "In-Place Database Upgrades" on page 55) is the recommended approach for JIRA 4.3 and later releases. The approach used here is the original one and involves creating a new database. It's still valid, and also a little easier if you don't have access to your database's restore tools.

The four instances of JIRA referred to in this upgrade procedure are:

- The current production JIRA (e.g., version 4.2)
- The upgraded production JIRA (e.g., version 4.3), usually on the same server as the current production JIRA—upgrading JIRA and moving JIRA to a new server are two steps that are best done separately
- The current development JIRA (e.g., version 4.2)
- The upgraded development JIRA (e.g., version 4.3), usually on the same server as the current development JIRA

If you don't have a development instance of JIRA, then your production instance is effectively your development instance, and your upgrade is done after Step 15 on page 53.

The upgrade steps follow:

1. As usual before changing anything, create a backup. Then start taking your own notes about each of the upcoming steps. Record what you did and any unexpected results.
2. Unpack the new version of JIRA on the development server, making sure the files are owned by the user that will run JIRA. This location is the new *jira_app* directory for this instance. Any existing *jira_app* soft link will eventually refer to this directory when the upgrade is complete.
3. Create a new JIRA Home directory named *jira_data_43* for the new development JIRA instance. Any existing *jira_data* soft link will eventually refer to this directory.
4. Create a new, empty database for the new development JIRA instance. Make sure that the database uses the correct character set, collation order, table type, and so forth (or at least the same ones as the existing JIRA database).
5. Configure the new development JIRA files. Some of the files in *jira_app* that are likely to be changed include:

   atlassian-jira/WEB-INF/classes/jira-application.properties
   : Used to set *jira.home* to the absolute path of *jira_data*.

atlassian-jira/WEB-INF/classes/entityengine.xml
: Used to change the type of database being used, and then *conf/server.xml* to change the database `Resource` element to refer to the new database you just created.

  It's important to use the right database name here, or you will end up upgrading your existing database in place and abandoning an upgrade will become harder. See "In-Place Database Upgrades" on page 55 for more details about this.

atlassian-jira/WEB-INF/classes/osuser.xml
: Used if you have Active Directory or Crowd integrated with JIRA; JIRA 4.3 needs this file but then stores the information elsewhere.

bin/setenv.sh
: Used to change the JVM memory settings and to uncomment `DISABLE_NOTIFICATIONS` so that the development instance doesn't send or retrieve email.*

lib/.jar*
: You should check that the JDBC driver jar file for your database is available here. Recent versions of JIRA ship with the drivers for MySQL, PostgreSQL, and Oracle. Earlier versions of JIRA used a version of Tomcat that kept the driver files in *common/lib*.

JIRA 4.4 will likely see changes in how JIRA is configured after installation. When that time comes, use the appropriate installer to configure all these files.

6. Copy or modify any other files such as custom icons for issue types, priorities, resolutions and statuses, or a logo image file. You can find a list of files that have been changed in the current production JIRA under Administration→System Info→Modifications. If you added any custom events (see the section "Workflows and Events" on page 40) or changed your email templates, you'll also need to merge them over to the upgraded instance.

   If you have integrated JIRA with Crowd, make sure you copy or merge any relevant files for that as well.

It's about now that I usually start to wish I had used version control or a tools such as *puppet* to record the changes I made to the current JIRA when I last upgraded it. It's not too late to start, but few people do.

* I also tend to comment out the line with `jirabanner.txt` because I find ASCII art isn't so cool when it looks like line noise in a log file.

7. If you want to, you can install any updated plugin files that you've downloaded manually at this stage. Having the plugins in place will reduce the number of error log messages about missing plugins when you import data into the new instance.

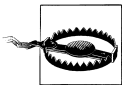
   Old-style version 1 plugin *.jar* files go into *atlassian-jira/WEB-INF/lib*. New-style version 2 plugin *.jar* files go in *jira_data/plugins/installed-plugins* (you'll have to create this directory). If a plugin's installation involves more than just copying a *.jar* file, I usually postpone installing it until after this step.

8. Export the data from the current production instance as a compressed XML backup file (Administration→Backup Data to XML).

   There is a short period here where JIRA is still active but the backup is running. Any changes made by users during this period will be lost when the upgrade is complete, so make sure you warn people not to use JIRA after a certain time.

   You may ask "Why can't I just make my current JIRA readonly?" You could do this, but then the port used by JIRA is still in use, and the new instance can't use it. Also, there's no really convenient way to allow users to log in to a read-only instance, which is necessary to for issue security.

9. Make sure any automated network monitoring systems are aware of the upcoming outage and then shut down the production JIRA instance. This is when the clock starts timing how long the upgrade inconveniences JIRA users.
10. Create a new directory called *jira_data/data/import* in the new development instance.
11. Copy the XML backup file into the *import* directory, or create a soft link to it there.
12. Start up the new JIRA development instance and browse to the start page. You shouldn't be prompted to log in since you're using a new database and a new *jira_data* directory.



If you are prompted to log in then you've probably just done an in-place database upgrade (see the section "In-Place Database Upgrades" on page 55) and you can skip the next step of importing the XML backup. However, if you stop the upgrade after this point, you will need to restore the database from a database backup.

13. Click on the link to import your existing data and give the absolute path to the XML backup file in the import directory. Then click *Restore*.

    If you need to update the license for this version of JIRA, you will be prompted for a new license key.

    You may also get a message about an import failing because the paths for the index directory or attachments directory wasn't found. This means that these directories were at locations in the current JIRA instance that don't exist on the upgraded instance. In that case, just click the "retry with default paths" link.

You can follow progress of the import in *jira_app/logs/catalina.out* or in *atlassian-jira.log*. It may take some minutes to parse the XML, store the generic entities to the new database, create the Lucene indexes, apply schema changes, and reindex for each schema change.

14. Copy the attachments over to the new *jira_data/data/attachments* directory. This can take some time if there are many large attachments. Check for other directories with any content in *jira_data/data* and copy them too.
15. Once JIRA is available again, you will want to test it as described in the next section, "Testing an Upgrade" on page 53.
16. After giving users some time to test the upgraded development instance, repeat all these steps for the production JIRA instance.

You can set a banner across the top of all of JIRA's pages using Administration→Options & Settings→Announcement Banner. This is a great way to warn users of an upcoming JIRA outage. For extra notice, you can use the existing CSS by surrounding your banner text with `div` elements:

```
<div class="infoBox">
Your announcement text goes here.
<div>
```

17. After the upgrade, you will need to change the name of the database that should be backed up. You may also need to change the JIRA Home directory location in the backup scripts. Any startup scripts to start JIRA after a server reboot should be checked too. If you have used soft links for *jira_data* and *jira_app*, then fewer things should have to change.

    Update any local Bare Metal Recovery documents and other administration notes, take a breath, and you're done!

Testing an Upgrade

The first and easiest test is whether user authentication is working—can you log in?

Start at the Administration→System Information page and look for errors or outdated settings. Does the Base URL need to change? Do the locations for Attachments, Indexing, and the Backup Service at Administration→Services look correct?

If the upgraded JIRA is a development instance, then you can compare it to the production JIRA directly in a browser. Compare the Administration→System Information pages for unexpected differences. Pick three issues at random and check that their data, including some attachments, is the same. You can also print all this out to use when testing the production JIRA after it is upgraded.

The most likely area for trouble with an upgrade is third-party plugins. Check the list of plugins for disabled plugins or plugin modules.

Another useful test is to count the number of custom fields in the development instance and in the production instance. Disabled custom field plugins may be responsible for any difference in numbers.

Check that all the custom fields that used to be searchable in the Issue Navigator are still searchable. If not, then a custom field searcher plugin may not be working.

If the upgraded JIRA is the production JIRA, then check that attachments can be downloaded from issues and that email can be sent and received.

The most likely place to notice that something is wrong is in the directory *jira_app/logs/catalina.out*. Searching in this file for `ERROR` and `WARN` will usually suggest things to investigate further. To me, a good upgrade is one that results in a minimal number of such warning messages in log files, and no log messages that repeat every minute and cause the log file to become bloated.

Troubleshooting an Upgrade

Most JIRA upgrades go just fine, but sometimes what was intended to be just a quick upgrade turns into a Bad Day. You've got frustrated users and only a few cryptic error messages to go on. What now?

If this is the upgraded production instance, then you should probably abandon the upgrade and let people use the current production JIRA instance again. To do that, simply stop the upgraded instance of JIRA, change any soft links back to where they used to point, and restart the current instance of JIRA.

Then check whether it was really a problem at all. Some messages in JIRA log files can appear alarming, but aren't in fact all that serious. For example, if you haven't installed a plugin that provides a custom field type then you will see an error like:

```
2011-04-16 21:54:23,847 main ERROR [jira.issue.managers.
DefaultCustomFieldManager] Could not load custom field type plugin with
key ... Is the plugin present and enabled?
```

JIRA preserves the data in any custom fields that use such custom field types, but doesn't display the custom field at Administration→Custom Fields. This means that you can't delete such fields and that you will see the error message every time you restart JIRA. Old services and listeners can generate similar error messages in log files.

More troublesome are messages that keep occurring, filling up a log file and making it harder to use when real problems occur. These are always worth tracking down—or as a last resort, reducing the log level for just that area in *log4j.properties*.

Other approaches to use when troubleshooting an upgrade are:

- Log into JIRA using the simplest URL possible. If you have forwarding set up with Apache, then disable that temporarily and use the default JIRA URL and port 8080.
- If you can't authenticate using Active Directory, then try logging in as some user that is defined in JIRA's internal directory service. The original JIRA administrator account may even still work for you.
- Check your `https` settings very carefully and only apply them after checking that everything else is working.

If all else fails, contact *Atlassian Support* or an *Atlassian Partner* to ask for help.

In-Place Database Upgrades

The upgrade procedure described in this chapter uses a new database for the new instance of JIRA. Leaving the database used by the current JIRA instance untouched makes stopping an upgrade easy. However, using a new database does mean that the data has to be migrated using an XML backup file, which can be very large and slow to generate for large JIRA instances.

As of JIRA 4.3, in-place database upgrades are officially supported and recommended. This means that you can reuse the same database for the new instance. The database schema, including the database indexes, will be modified by the new instance of JIRA when it starts up. Of course, you should back up at the database level before that occurs, and upgrading a development JIRA instance first is still a good idea. This also means that JIRA can lock users out during the upgrade, which reduces the period where a user's changes might be lost during an XML upgrade.

This affects Steps 4, 8, and 10 through 14 of the upgrade procedure.

New installers in future releases of JIRA are likely to reduce the number of manual changes required in Step 5.

Further Reading

The official Atlassian support policy for JIRA can be found at *http://confluence.atlassian.com/display/Support/Atlassian+Support+End+of+Life+Policy*.

Logging in JIRA is configured at Administration→System→Logging & Profiling. More detailed documentation about the underlying `log4j` logging framework can be found at *http://logging.apache.org/log4j/1.2/manual.html*.

There is an existing feature request to be able to make JIRA read-only at *http://jira.atlassian.com/browse/JRA-1924*.

CHAPTER 8

Remote Access to JIRA

Overview

"No man is an island," wrote John Donne, and this is doubly true of JIRA and almost every other application you may administer. Users want their data to appear in multiple places, administrators want to manage applications from a single place, and anyone may want to run some scripts to make lots of changes at once.

All of these require remote access to JIRA, where "remote access" is defined loosely as using JIRA without a browser.

This chapter covers a variety of remote access methods for JIRA. The quick summary is that SOAP is the way that most remote access to JIRA occurs in 2011. The future is likely to see more REST than SOAP access, but it will take a while.

Email

Email is one of the simplest ways to use JIRA remotely. Issues can be created and then comments added to them using the standard JIRA mail service and mail handlers. Email is a function commonly already found in many applications.* Balanced against the simplicity of email are its limitations:

- Email messages have limited structure. Only the *To*, *Cc*, *Subject*, *From* fields and any attachments are easily separated from the email body.
- There's no real guarantee about who most email is from, so authentication is hard.
- Email is asynchronous and unreliable, in the sense that retrying failed messages is slow and limited—so you don't know if your message reached JIRA. You may not even get any feedback about whether a JIRA server is currently active.

* *Zawinski's Law*: Every program attempts to expand until it can read mail.

Still, it's familiar and convenient and a fair number of JIRA users only interact with JIRA via email (see "Further Reading" on page 63). Some integrations between different systems use email, but it's not really a great idea because of the limitations listed above.

SQL

Accessing the underlying database of a JIRA instance is surprisingly common, perhaps because like email, many applications are already accessing their own database and adding one more is an obvious approach to try. Most access to JIRA's database is for creating reports using a separate report generating tool.

Reading data from the JIRA database using other systems is usually just fine (and is generally pretty fast) but does have the following strict limitations:

- *Access must be read-only*. This is because JIRA caches many of the values read from its database and may not update the caches until the next time it is restarted. If the data is changed in JIRA before that happens, then the updated values will be written back to the database, overwriting any changes that you made there.
- You have to carefully control who can view the different tables in the JIRA database. For instance, if some issues have issue security schemes (see "Issue Security Schemes" on page 15) defined so that only certain people can see the issues, the underlying confidential data could end up being unintentionally visible to the user running the SQL query.
- The JIRA database schema is deliberately only partially documented at *http://confluence.atlassian.com/display/JIRA/Database+Schema* because the schema does change between different versions of JIRA. JIRA upgrades handle such changes automatically, but Atlassian does not encourage (or support) direct database access. The actual file where the database schema is defined (and not documented) is *atlassian-jira/WEB-INF/classes/entitydefs/entitymodel.xml*.

If you do decide to take this approach, then one place to start is *http://confluence.atlassian.com/display/JIRACOM/Example+SQL+queries+for+JIRA*. This page has a number of SQL queries that other people have found useful. However, most of them don't have much explanation of why they work the way they do.

Another place to start is with the SQL queries that JIRA itself executes to retrieve data. While this is sometimes useful for low-level troubleshooting, the queries themselves may not be executed in quite the order you expect, since some data is prefetched and then cached for later use.

SOAP

SOAP (Simple Object Access Protocol) web service methods are currently the most common way of remotely accessing JIRA. The system plugin `rpc-jira-plugin` provides over a hundred different SOAP methods for accessing JIRA. Methods are defined within the JIRA server using the Java interface `JiraSoapService`, and are documented using `Javadoc`, just like any other Java method.

For example, the method to retrieve an issue looks like this:

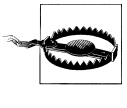
```
/**
 * Return a representation of a JIRA issue.
 *
 * @param token the SOAP authentication token
 * @param issueKey the key of the issue
 *
 * @return the issue fields in a RemoteIssue object
 */
 RemoteIssue getIssue(java.lang.String token, java.lang.String issueKey)
```

These SOAP methods are implemented in Java inside JIRA, but the choice of language to use for the client is much wider. There are SOAP libraries for C, C++, Perl, Python and Ruby, as well as Java. This means that using SOAP for scripted interactions with JIRA is possible using a language that you're probably already familiar with.

The JIRA documentation on using SOAP at *http://confluence.atlassian.com/display/JIRA/Creating+a+SOAP+Client* contains step-by-step examples of how to create a SOAP client in various languages. It also has nearly 300 comments from the last five years, which usually indicates a topic that many people have found relatively complex.

SOAP isn't a particularly fast way of remotely accessing any application since it uses XML and HTTP. It's worth bearing this in mind if you're designing scripts that have to iterate over all the issues in a project. For really long operations, it may be worth implementing your own custom SOAP method to do most of the work inside the JIRA server (see "Creating Custom SOAP Methods" on page 61) .

One other thing to note about SOAP is that although the endpoint URL does contain a version number (`v2`), this number will not have been changed when the API changed. Instead, all the changes so far have involved adding new methods and deprecating other methods (but leaving them in place).



There is another remote access method named *XML-RPC*. It's similar to SOAP, but is generally less well-supported in all versions of JIRA since 4.0, and thus is not commonly used anymore.

Debugging a SOAP Client

SOAP error messages are notoriously cryptic in any language. Some of the steps that I follow when I'm trying to get a buggy SOAP client to work are as follows:

1. The first thing to do is to double check that the *Accept Remote API Calls* setting is set to ON under Administration→General Configuration→Global Settings.
2. Next, check the URL you are using. The URL for Atlassian's own JIRA instance is *http://jira.atlassian.com/rpc/soap/jirasoapservice-v2?wsdl*. This should display lots of SOAP method details in your browser. Chrome users may need to install an extension such as *XML Tree* to see them.

   The WSDL file that is displayed is an XML document listing all the available SOAP methods (`wsdl:operation`) and data types (`complexType`). A command client tool such as *wget* can be useful for working out why you can't download the WSDL file.
3. Now try to download the WSDL file from your local JIRA instance. Your URL should look like some combination of one of these URLs:

   http://jira.example.com/rpc/soap/jirasoapservice-v2?wsdl
   https://jira.example.com/rpc/soap/jirasoapservice-v2?wsdl
   *http://jira.example.com****:8080****/rpc/soap/jirasoapservice-v2?wsdl*
   *http://my.example.com/****jira****/rpc/soap/jirasoapservice-v2?wsdl*

   The bold parts of the URLs indicate things to check if the URL is not working—scheme, port, context.
4. When you first connect to your JIRA instance at the correct URL, the *atlassian-jira.log* file should contain a message that looks like this:

   ```
   Publishing to jirasoapservice-v2 module class
   com.atlassian.jira.rpc.soap.JiraSoapServiceImpl with
   interface interface com.atlassian.jira.rpc.soap.JiraSoapService
   ```
5. The JIRA SOAP service expects you to use its `login` method with a valid user and password, and then to pass the authentication string `token` that is returned into every other JIRA SOAP method. This is so that JIRA can check that you have permission to execute each individual method. You can confirm that this authentication should succeed by logging into JIRA with the same username and password in a browser.

   If a specific user can't browse an issue in the browser, then that user won't be able to access the issue's details using SOAP.

   Some JIRA SOAP methods such as `getCustomFields` and all the methods for working with groups, schemes, and permissions require that the authenticated user is a member of the *jira-administrators* group.
6. If you have to use a compiled language such as Java or C++ for your client, try first using a dynamic language such as Perl or Python to create a small test script to

prove that any problem is not in the URL, authentication, or the values passed to the method. This can greatly reduce the time taken for each development iteration.

Creating Custom SOAP Methods

It's not easy to find the method you want in the long *list* of SOAP methods and some of the methods' documentation is pretty sparse. So how much work is it to create a new SOAP method for a specific need?

The answer is that it's not too hard to write most JIRA plugins. The SOAP endpoint URL used to access the new method will be different from the standard JIRA one, and you'll have to make sure you add your own authentication checking in the method. Another place to look for information about custom SOAP methods is at *http://confluence.atlassian.com/display/JIRA/RPC+Endpoint+Plugin+Module*.

REST

The future of remote access for JIRA is officially REST. REST methods are invoked using a URI, plus arguments passed in after the URI.† The resulting data from JIRA is in the JSON format. The returned data usually also contains further URIs to let you drill down into the data. This means that clients can "walk" the data and dynamically discover what is currently available.

A good place to start for information about JIRA and REST is the tutorial at *http://confluence.atlassian.com/display/JIRA/JIRA+REST+API+%28Alpha%29+Tutorial* .

The use of the word "Alpha" in the title is a strong clue that the JIRA REST API is still under development. As of JIRA 4.3, you can retrieve a fair amount of information about issues, but not modify them. General purpose REST resources for administering JIRA don't exist yet either.

If you want to see an example of what JIRA returns with REST for an issue, browse to *http://jira.atlassian.com/rest/api/latest/issue/JRA-9.json*. Different browsers will display the resulting JSON data differently. The *Pretty JSON* extension is helpful for Chrome users.

Alternatively, you can use a command line tool such as *curl* or *wget*. Replace `userid` with your JIRA username and `secret` with your password and try the following:

```
curl -u userid:secret \
  http://jira.atlassian.com/rest/api/latest/issue/JRA-9.json

wget --user=userid --password=secret \
  http://jira.atlassian.com/rest/api/latest/issue/JRA-9.json
```

† The more familiar URL (Uniform Resource Locator) is one kind of URI (Uniform Resource Identifier).

Over the past two years, JIRA has been changing internally to use REST for retrieving more of the information that is destined for the UI. The JIRA Dashboard gadgets all use REST, as have the Labels and Versions system fields since JIRA 4.2. This is all part of the general move in web applications toward Dynamic HTML (DHTML), where JavaScript running in the client's browser asynchronously populates the HTML and CSS that is used to produce the displayed web page.

The process of creating your own REST resources by writing a custom JIRA plugin is described at *http://confluence.atlassian.com/display/DEVNET/Plugin+Tutorial+-+Writing+REST+Services*.

XML and RSS

Retrieving an XML file with the details of just one issue is easy with a URL such as *http://jira.atlassian.com/si/jira.issueviews:issue-xml/JRA-9/JRA-9.xml*. The XML that's returned is formatted as an RSS feed item, but since it's structured data, it can be processed by clients for integrating other systems with JIRA.

You can also retrieve the results of a search as an XML file using the links in the Views menu in the Issue Navigator screen. The `XML` link is the one that is used for integrating JIRA with Confluence. JIRA provides similar RSS feeds for the activity of individual users and projects as well.

Authentication is probably the hardest part of using RSS feeds for integration. A username and password can be passed in the URL or a user can be prompted for a password, but neither method is particularly robust.

More information can be found in the documentation at *http://confluence.atlassian.com/display/JIRA/Displaying+Search+Results+in+XML*.

CLI (Command Line Interface)

Whether your underlying remote access method is SOAP or REST, you may not really care if all you want is a way to interact with JIRA from a command line or with a script. For example, you may want to have an automated build system add a new version in JIRA when a new release occurs. Or you might want to make it easier for an IT group to create new users in JIRA by providing a small script for them to run.

There are two CLIs available for JIRA—*JIRA CLI* by Bob Swift, and *Python CLI for JIRA* by myself. The first one is written entirely in Java, is invoked from a shell script, and is part of a suite of CLI tools for all of the Atlassian products. The second one is a Python script that could do with a rewrite. Both offer similar functionality and both use the standard JIRA SOAP methods.

So, which CLI to choose? I have to admit a certain fondness for my own creation but I tend to mostly use it as a starting point for other work these days. I create a lot of scripts

to automate JIRA administration tasks, along with custom plugins with new SOAP methods, and Python works well for me for this. Bob's CLI is well-maintained and very well tested, so if it does what you want, then try that one first and tell Bob I sent you!

Integrating with Other Applications

Whichever method is used for integrating other applications with JIRA, there are a few things are that worth bearing in mind.

Networking outages will inevitably occur, and your integration has to survive them without inconsistent or corrupted data. This means that any synchronous approach such as SOAP or REST has to be able to retry later on or warn users that something went wrong. Other than changing issues, most JIRA operations are not strictly transactional, which makes integrations harder.

JIRA provides *services*, custom tasks that are periodically executed as frequently as once per minute. This is one way to have an integration be able to recover from errors. Such services also run when JIRA is started, which may mean a large load on another system if JIRA has been down for a while. In this case, limiting the amount of work that is done in a single run of the service can help.

Synchronizing two systems in one direction only is much simpler than doing it in both directions. If you really do have to do it in both directions, consider very carefully how you're going to avoid infinite loops and which application will maintain the synchronization state (don't try storing it in both). JIRA issues have an `Updated` date field that can help with that if the systems' clocks are synchronized, and there are also the internal records of everything that has changed which can be used.

There is a useful book titled *Enterprise Integration Patterns*, by Gregor Hohpe and Bobby Woolf (Addison-Wesley) that covers these kinds of issues. And you haven't already read it, then *The Twelve Networking Truths* (RFC 1925, *http://www.faqs.org/rfcs/rfc1925.html*) is brief and applies to many integrations just as well as it does to networking design.

Further Reading

The process of using JIRA via email is documented at *http://confluence.atlassian.com/display/JIRA/Creating+Issues+and+Comments+from+Email*. The most commonly used plugin to do more with this is Andy Brook's *JEMH* (JIRA Extendable Mail Handler).

The main page for information about the JIRA database schema is *http://confluence.atlassian.com/display/JIRA/Database+Schema*. Enabling logging of all SQL queries is described at *http://confluence.atlassian.com/display/JIRA/Logging+JIRA+SQL+Queries*.

Good places to look for more general information about SOAP and REST are *http://en.wikipedia.org/wiki/SOAP* and *http://en.wikipedia.org/wiki/REST*, respectively. More information about JSON and an example of what that format looks like can be found at *http://en.wikipedia.org/wiki/JSON*.

Although neither CLI is technically a JIRA plugin, they are both available from `plugins.atlassian.com`. Bob Swift's *JIRA CLI* is at *https://plugins.atlassian.com/plugin/details/6398*, and my *Python CLI for JIRA* is at *https://plugins.atlassian.com/plugin/details/10751*.

Finally, the whole of the quote from John Donne:

> No man is an Iland, intire of it selfe; every man is a peece of the Continent, a part of the maine; if a Clod bee washed away by the Sea, Europe is the lesse, as well as if a Promontorie were, as well as if a Mannor of thy friends or of thine owne were; any mans death diminishes me, because I am involved in Mankinde; And therefore never send to know for whom the bell tolls; It tolls for thee.
>
> —Devotion XVII (Meditation XVII), John Donne, 1624

CHAPTER 9

Jiraargh! Frustrations

Overview

I once wrote an article with the provocative title *Bug Trackers: Do They Really All Suck?*. My conclusion was probably not, but that they are all annoying in some way. Why is this? My theory is that it's because tools like JIRA are used by multiple groups of people. This means that each group has different needs from the same tool, which in turn leads to no one being fully satisfied.

So what can be done? I think the first thing to remember is that every problem is at root a people problem (*Jerry Weinberg*). What that means for you and your JIRA is that decisions about using the tool need to be discussed and agreed upon by all the people who use the tool. Tools can only help to reduce the barriers for people and groups working together; there is no magic tool to make people *want* to work together.

That said, this chapter describes some of the more common frustrations with JIRA and possible ways to avoid some of them. These include both things that annoy users if not configured properly by administrators, and aspects of JIRA that annoy JIRA administrators.

Frustrations with Fields

One of the most common frustrations for users is having to enter data in fields that they don't understand or care about. This can happen in a number of ways:

- A field may be required by one group, but not another.

  The solution for this is to define a field configuration and screens for use by just that group's projects, along with the other schemes for their projects. Grouping projects by categories can help with maintaining such schemes, as described in Chapter 4.

- The field's description may be missing or misleading.

  Don't accept a request for a new custom field unless it comes with a succinct and clear description of what the field is intended for. Then use that as the field's description.
- The field is being used as a combination of other fields.

  This can happen if custom fields aren't added when needed, so users overload an existing field (such as the summary) to record two or more pieces of information. For example, I've seen summary fields that contain text like "Showstopper: customer can't log in". The "showstopper" part of that field should have been recorded in some other field, probably priority, to allow better reporting lafter on. It's also easy to forget the convention, or use it without really understanding it (perhaps using it just because some other issues did).
- Required fields are not on the default tab.

  Just don't do this when you're designing screens. Put all the required fields on the default tab, so you don't forget that they're required during issue creation.

Frustrations with Actions

Other frustrations for users center around issue actions and workflow transitions:

- Why can't I change an issue to a certain status?

  If it's just certain transitions, then it's probably due to conditions and validations that have been defined as part of the workflow for the issue. These conditions may or may not make any use of the Resolve Issues and Close Issues permissions, depending on how the workflow is configured (see Chapter 5).

  Adding the introduction gadget to the system dashboard and including links to documents that describe such expected restrictions can help.
- What permission do I need for a certain issue operation?

  Permissions are controlled with the permission scheme that an issue's project is using. To edit an issue, a user needs to have the Edit permission, which may be granted by project role or any of the other ways listed in the section "Adding Users to Schemes" on page 12.

  Some of the less obvious permissions are noted in the section "Permission Schemes" on page 14. Changing a permission usually involves adding a user to a specific project role, rather than modifying the permission scheme.
- Why can't I bulk change some issues?

  The default JIRA workflow doesn't let you edit issues in the Closed status. So if you want to change a field in a hundred issues using Bulk Change but just one of those issues is closed, then JIRA won't let you change any of them. Sometimes there's an explanation given, but more often it's a puzzle to users and their

administrators. Start with a smaller set of issues and see if you can bulk change those issues.

Not all of an issue's fields can always be altered using a Bulk Change. For example, the Resolution field can only be changed if the field configuration hasn't hidden that field.

More Information Needed!

Sometimes users need more guidance about where to create or not create issues, and what should go in each field. What is needed is a way to add this information to JIRA's issue screens.

For just this reason, the *Message* custom field types are part of the standard *JIRA Toolkit plugin* available from Atlassian. These field types allow you to display text on an issue's screens. For example, the *Message Custom Field (for edit)* field type allows you to insert text between other fields on an edit screen. Other custom field types from the same plugin allow you to insert HTML (or even use Velocity template files) to create the message. As shown in Figure 9-1, Atlassian uses these fields to provide additional guidance to users filing bugs with them.

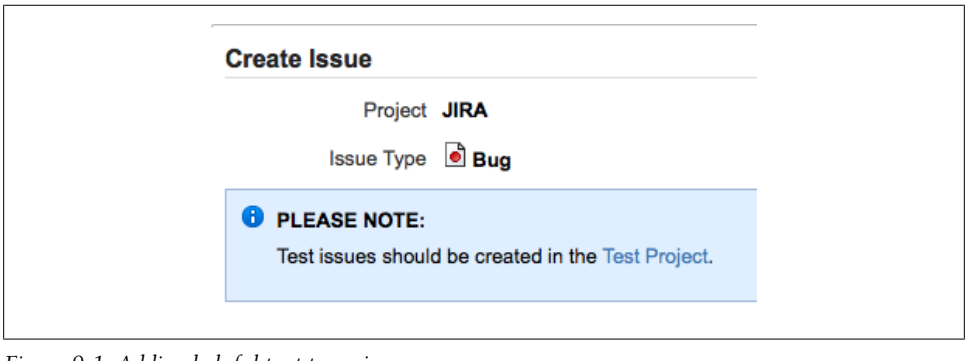


Figure 9-1. Adding helpful text to an issue screen

These message fields are defined just like any other custom field, and are added to screens as usual. They don't have any values and do not appear in an issue otherwise.

Frustrations with Email

How can you make sure that another user is notified of changes to a particular issue? That is what the Watchers system field is for. But even after another user has been added to the Watchers field, they won't receive email about the issue until the *next* change after that. It's like having a *Cc* on an email that only works for the next email.

Sometimes a user wants to just "poke" someone about an issue, but isn't sure who else will receive the email if they add a comment. This is like *Reply to All*, but without being

able to see the *To* field. What is really needed is the current list of email recipients for each action, just for when it is wanted. The *JIRA Email This Issue Plugin* can help here.

Learning JIRA Safely

The first thing that any new administrator usually does with JIRA is to create a test project and a few test issues. The next thing they might do is change the default priorities and then start in on some workflow changes. A few minutes later, they wish they had made a backup and hadn't been playing around with the production JIRA instance.

Now, playing about is a *powerful*[*] way to learn any tool as a new user or administrator, but do it somewhere that won't matter when you make mistakes. Having a development instance of JIRA is a great idea for this.

If you're concerned about sending email to other people as a result of your changes, then you can define an empty notification scheme for the JIRA project where you are playing with issues. You could also do the same thing by just removing the notification scheme from a project, but using an empty scheme makes it clearer in Administration→Notification Schemes as to which projects don't send email.

I also like to have a project named *SCRATCH* in production JIRA instances that is configured with the same schemes as one of my more heavily-used JIRA projects. I use that project to create test issues when checking my changes and avoid cluttering up any real projects.

Another thing I do is to maintain a separate project or component to track issues related to the local JIRA instance. This is using JIRA as a *meta-tracker* and it helps for all the same reasons you installed JIRA in the first place.

Too Many Administrators

Having too many JIRA administrators invariably leads to a JIRA instance with a confusing configuration. The temptation to experiment and hack until JIRA appears to do what you want seems to be irresistible. A main administrator, a couple of backup administrators, and maybe an IT-related administrator are usually all that should be needed. I recommend trimming the *jira-administrators* group regularly.

One oft-overlooked aspect of JIRA is that you can have both *jira-administrators* and *jira-system-administrators* groups. The differences between these two groups are described at *http://confluence.atlassian.com/display/JIRA/Managing+Global+Permissions#ManagingGlobalPermissions-sysadmin*. By default, the two groups are effectively the same, but one way to reduce what the existing JIRA administrators can do is to go to

[*] "Let my playing be my learning, and my learning be my playing." —*Homo Ludens*, Johan Huizinga, 1938

Administration→Global Settings→Global Permissions, and set the group used for JIRA System Administrators to a new *jira-system-administrators* group.

You may wonder "Why have I got so many administrators in my JIRA?" The most likely explanation is that earlier versions of JIRA allowed only JIRA administrators to add components and versions to a project. This implied that project leads had to also be JIRA administrators, which often led to many users in the *jira-administrators group*. The better approach nowadays is to use JIRA project roles (see the section "Project Roles" on page 2), which let you grant project leads the permissions they need for their projects without destabilizing the rest of your JIRA instance.

Debugging your Configuration

Working out why a user can't see or do something they expect can take a few frustrating minutes, even when you have a good procedure.

My approach, described in more detail in "Debugging Schemes" on page 22, is to note the issue's project and issue type, then go to the project administration screen. Note the names of all of the schemes that the project is using and then view the appropriate one. For example, for permissions, view the named permission scheme and see which project roles are needed for the permission in question. Then return to the project's administration screen and go to the project's roles to see which users have that role in the project.

For more complex schemes (such as the issue type screen scheme), view the scheme and use the issue type to work out which screen scheme is being applied. Then look at that screen scheme to work out which of the screens is involved. Finally, view that screen to see whether the field is shown or not, and where.

Managing Custom Fields

Managing large numbers of custom fields (more than a hundred) can be difficult because you have to scroll up and down to find the field you want. The "find" feature in your browser can help here. JIRA permits multiple custom fields with the same name and field type, so check carefully which issue types and projects each custom field is applied to. Even a single custom field can have multiple contexts (i.e., issue type and project pairs) so you may have to click *Configure* for a field before you can see all the ways that a field is configured.

Managing Projects

Changing the project settings for more than a few projects quickly becomes tedious. For example, if you've added a new issue type and have updated a copy of a workflow scheme to include the new issue type, you now have to manually update each project's workflow scheme to use the new scheme.

Opening each project in a new tab on a browser can help to make sure that you don't miss a project, but it's still awkward. Sometimes I create a new SOAP method (see "Creating Custom SOAP Methods" on page 61) to handle just this kind of case but it's a fiddly few hours of work that I'd rather not have to do.

Once a JIRA project has been configured as intended, a common request is to create another project configured in exactly the same way. The new project will likely be part of a number of projects with the same category. JIRA has no built-in way to use one project as a template for another—though look at "Further Reading" on page 71 for information about a plugin that can do this.

Managing Users

Adding one or two users to JIRA is straightforward enough, but adding a few dozen users soon becomes awkward. What is needed is a scripted way to do this.

One approach to automating this and JIRA administration in general is to use Jelly scripts. Jelly is a XML-based scripting language originally used by Maven 1.x, but now only used by JIRA (as far as I can tell). It has to be enabled explicitly for a JIRA instance, but after that you can go to Administration→Options & Settings→Jelly Runner, and paste in scripts that look like this:

```
<JiraJelly xmlns:jira="jelly:com.atlassian.jira.jelly.JiraTagLib">
  <jira:CreateUser username="john.smith"
                   password="secret"
                   confirm="secret"
                   fullname="John Smith"
                   email="jsmith@example.com"/>
</JiraJelly>
```

The *Python CLI for JIRA* (see "CLI (Command Line Interface)" on page 62) also has a function for creating users by reading from a text file of user details with one user per line.

Deactivating users is another task that can benefit from automated scripting, though the standard JIRA SOAP methods are not sufficient to do this. Custom SOAP methods can be created as described in "Creating Custom SOAP Methods" on page 61.

Further Reading

The same *Script Runner* plugin by Jamie Echlin that was used to change a username in "Changing a Username" on page 44 has scripts to copy the configuration from one project to another, and also to bulk modify resolutions.

The JIRA Email This Issue Plugin can be found at *https://plugins.atlassian.com/plugin/details/4977*.

The Jelly website is *http://commons.apache.org/jelly*, and the use of Jelly in JIRA is documented at at *http://confluence.atlassian.com/display/JIRA/Jelly+Tags*.

About the Author

Matt Doar is based in San Jose, CA where he runs a software tools consultancy. He has been helping other people with JIRA for over five years and is the author of a number of JIRA plugins. He is an Atlassian partner and is part of the wider Atlassian development community. He also wrote *Practical Development Environments* (O'Reilly), which described the basics of software tools—version control, build tools, testing, issue trackers, and automation. He has also held some sort of dubious record for the most bugs submitted about JIRA by a non-Atlassian. Before JIRA entered his world, Matt was a developer and then a software toolsmith at various networking companies. Before all that, he completed his B.A. and Ph.D. in Computer Networking at the University of Cambridge Computer Laboratory and St. John's College, Cambridge.

Colophon

The animals on the cover of *Practical JIRA Administration* are Cochin chickens (*Gallus domesticus*).

The cover image is from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed.