Jira Service Management to Jira Software

Last Modified on 03/25/2024 11:05 am EDT

Warning: Despite our best efforts, code can change without notice due to a variety of factors. If you encounter an issue in any of the code shown here and find that a specific block of code is not correct, or is causing errors, please check with the Community to find an updated version.

This use case describes a synchronization between Jira Service Management and Jira Software.

With the Exalate app for Jira, you can connect 2 or more projects and sync issues in different ways. It is possible to configure synchronization locally within one instance or externally between multiple Jira instances.

The app supports a secure connection and can be integrated even if one of the Jira instances is behind the firewall and is not accessible from the outside network. Find more details about how the traffic is different between a public and a private side.

Data exchange between the Service Management and Jira Software can be handled in a uni- or bi-directional manner.

Below you find the configuration description of the use case, where you need to handle data flow from the customer-facing Service Management to the software development project.

These projects might be inside the same Jira instance or in different Jira instances. For example, the Service Management could be a public external instance, and the DEV project in the internal network, which is behind the firewall.

Configuration Requirements

When the customer submits a new issue via the Customer Portal, a twin issue should be created automatically in the Jira Software development project.

Basic fields such as summary, description, and attachments should be kept in bi-directional sync.

Only public comments from the DEV project are synced to the SD project. All comments on the SD project issue should be synced to the DEV project issue.

When certain transitions occur in the issue in the DEV project, they should trigger some transitions in the SD project issue.

Additionally, there may also be some data added to a custom field during the transition, this needs to be synced to the SD issue as well.

Mapping Configuration

- **Issue types:**

  if SD request type is *service request* -> create a *task* issue type in DEV;

if SD request type is *feature request* -> create an *improvement* in DEV;

if SD request type is *bug* -> create a *bug* in DEV;

Configuration Example 1

Receiving Side(Jira Software Project)

Incoming Sync

```
  // Set type name according to the mapping, if not found set a default to Task
      def issueTypeMapping = [
  // "Remote issue type" : "Local issue type"
          "Service Request" : "Task",
          "Feature Request" : "Improvement"
          "Bug" : "Bug"
]

issue.typeName = issueTypeMapping[replica.type?.name] ?: "Task"
}
```

- **Project Mapping:**

  You can specify the source project and the destination project during the connection set up if both SD and DEV projects are in the same Jira instance.

  If the projects are in different Jira instances, you need to specify the project for Incoming sync during the connection set up on each side separately.

- **Issue Fields Mapping:**

  All fields in both projects might be mapped in between.
  You can specify which issue fields should be synced, including any possible mapping combination.

- **Ticket Reporter:**

  Set a reporter in the DEV project based on the issue assignee from the Service Management side.

Configuration Example 2

Receiving Side(Jira Software Project)

Incoming Sync

Set local issue reporter based on the remote issue assignee, if the user is not found set a default user.

```
issue.reporter = nodeHelper.getUserByEmail(replica.assignee?.email ?: "admin@admin.com");
```

- **Workflow Mapping:**

  create SD ticket -> create DEV ticket;
  update status on SD/DEV -> update status on the other side;
  close DEV -> close SD;
  specify issue status mapping

Status Mapping Configuration Example

Incoming Sync

Set the local status based on the received status value from the remote side.

```
def statusMap = [

      // "remote status name": "local status name"
        "To Do" : "New",
        "In Progress" : "In Progres",
        "Done" : "Canceled"
   ]
def remoteStatusName = replica.status.name
issue.setStatus(statusMap[remoteStatusName] ?: remoteStatusName)
```

- **Comment Handling:**
  Sync original SD comments without any changes. You can also display the original author of the synced comment.

Configuration Example 3

Receiving Side(Jira Software Project)

  - Use the original comment author.
  - If the author does not exist in the local Jira, use the proxy user as the comment author.

Incoming Sync

```
replica.addedComments.each { it.executor = nodeHelper.getUserByEmail(it.author?.email) }
replica.changedComments.each { it.executor = nodeHelper.getUserByEmail(it.updateAuthor?.email) }
issue.comments = commentHelper.mergeComments(issue, replica, { it })
```

You might keep some of the comments internally. For example, if the received comment from the SD project is internal, make it visible only to a certain group of users on the DEV side.

Configuration Example 4

Receiving Side(Jira Software Project)

Incoming Sync

```
issue.comments = commentHelper.mergeComments(issue, replica, {
     comment ->
     if (comment.internal) {
         // if the remote comment is internal make it visible to only users with role "team"
         comment.roleLevel = "team"
     } else {
     // remove all restrictions
         comment.roleLevel = null
         comment.groupLevel = null
     }
     comment
}
)
```

- **Attachment Handling:**

You can configure different behaviors according to your needs:

- Sync only a specific file type of the attachments, for example:

  Send only .PDF files

  Outgoing Sync (Sending side)

  ```
  replica.attachments = issue.attachments.findAll { attachment -> attachment.filename.endsWith(".pdf") }
  ```

- Sync only the attachments up to a certain file size.

  Send only attachments < 500 bytes

  Outgoing Sync (Sending side)

  ```
  replica.attachments = issue.attachments.findAll { attachment -> attachment.filesize < 500}
  ```

Conclusion

The Exalate app provides a wide range of possibilities and keeps its flexibility at all times. You can define in the Sync Rules which data gets synced and which remains unsynced. With the Exalate distributed architecture, your configuration is not affect the other side. It gives autonomy to you and your partner.

Exalate uses its own transactional synchronization engine, meaning that every issue event is transformed into an outgoing or incoming sync event and is synced in the same order as it was performed in the original issue. The possibility to pause and resume any ongoing synchronization allows controlling all the data flow between instances.

Have more questions? Ask the community