Jira Cloud GitHub: Leveraging GitHub TaskLists to emulate the Epic-Child relationship from Jira

Jira allows you to create complex hierarchy between issues to better organize your project. However, if you are syncing such a Jira project to GitHub, chances are you would want to maintain some semblance of this hierarchy on the GitHub side as well. The only effective way we found to do this was by using Task Lists in GitHub. Once you get a hang of how these TaskLists work, you can build complex hierarchical structures even in GitHub!

Use Case

The use case being discussed here would be syncing Jira issues to GitHub issues with the following requirements:

- Ticket gets created in Jira
- Dynamically create the GitHub issue in the correct repository based on the user selection from a custom field in Jira.
- Represent the Epic-Child relationship from Jira to GitHub using TaskLists.
- Maintain the bi-directional synchronization between issues.
- When a GitHub issue is Closed, mark the Jira ticket as Done.

Solution

Step 1: Set up Trigger on Jira side.

- Create a Jira trigger based on the drop down custom field and the Jira project:

Jira Trigger

```
project=CM and "GitHub Repository" != null
```

Step 2: Determine the target repository in GitHub based on user selection from Jira side.

- From the Jira side, we need to ensure that the custom field value is being sent out. This can be done by adding the following line in the outgoing script of Jira:

Jira Outgoing

```
replica.customFields."GitHub Repository" = issue.customFields."GitHub Repository"
```

- Now on the GitHub side, we decide which repository to create the issue in. Add the following in the GitHub Incoming script:

GitHub Incoming

```
if(firstSync){
    if (replica.customFields."GitHub Repository"?.value?.value == "Project Mars")
        issue.repository   = "majid-org/project-mars"
    else if (replica.customFields."GitHub Repository"?.value?.value == "Demo Repo")
        issue.repository   = "majid-org/demo-repo"
    issue.summary      = replica.summary


}
```

Step 3: If Jira is sending an Epic, add the child issues to the replica.

- We use the JiraClient to run an API call to run a JQL filter and gather the results into the replica."child" data structure:

Jira Outgoing

```
replica."child" = []
if (issue.issueType.name == "Epic"){
    def js = new groovy.json.JsonSlurper()
    def jql = "cf[10014]=${issue.key}".toString()
    def localIssue = new JiraClient(httpClient).http("GET", "/rest/api/latest/search", ["jql":[jql]],
null, [:])  {
        response ->
        if (response.code >= 300 && response.code != 404)
            throw new com.exalate.api.exception.IssueTrackerException("Failed to perform the request
GET")
        if (response.code == 404)
            return null
        def txt = response.body as String
        txt = js.parseText(txt)
        txt.issues.each {
            it ->
            replica."child" += it.id
        }
    }
}
```

Step 4: Add TaskLists to the GitHub issues based on whether an Epic has arrived or a child issue.

- We extract the data from the replica."child" data structure and add the relevant task lists:

GitHub Incoming

```
issue.description  = replica.description
if (replica.issueType.name != "Task"){
    if (replica?."child".size != 0){
        issue.description += "\n\n______\nRelated Issues\n______"
        replica."child".each{
            it ->
            def localParent = syncHelper.getLocalKeyFromRemoteId(it, "issue")?.key
            issue.description += "\n- [ ] #${localParent}"
        }
    }
}


if (replica.issueType.name == "Task"){
    if (replica?.parentId){
        issue.description += "\n\n______\nParent Issue\n______"
            def localParent = syncHelper.getLocalKeyFromRemoteId(replica.parentId, "issue")?.key
            issue.description += "\n- [ ] #${localParent}"
        }
}
```

Step 5: Icing on the cake!

- When a child ticket in Jira is Exalated, this creates a sync event for the parent (Epic) before the child has been created in GitHub.
- The result is that the Task List contains a null in GitHub on firstSync.
- To get around this, we add store(issue) and syncBackAfterProcessing() to the firstSync block in GitHub. This ensures that the child sends a syncBack to Jira:

GitHub Incoming

```
if(firstSync){
    if (replica.customFields."GitHub Repository"?.value?.value == "Project Mars")
        issue.repository   = "majid-org/project-mars"
    else
        issue.repository   = "majid-org/demo-repo"
    issue.summary      = replica.summary
    issue.description  = replica.description
    store(issue)
    syncHelper.syncBackAfterProcessing()
}
```

- Then in Jira, we use the following to send a syncEvent for the parent (Epic):

Jira Incoming

```
if(issue.typeName == "Task"){
    def a = ""
    def matcher  = replica.description =~ /#(\d{1,3})$/
    if (matcher)
        a = replica.description.split("#")[1]

    def localIssue = syncHelper.getLocalIssueKeyFromRemoteId(issue.'SNOW Company'.toLong())
    def res = httpClient.get("/rest/api/3/issue/${localIssue.urn}")
    res = httpClient.get("/rest/api/3/issue/${res.fields.parent.key}")
    def localParent = syncHelper.getLocalIssueKeyFromRemoteId(res.fields.customfield_10094.toLong())

   syncHelper.eventSchedulerNotificationService.scheduleSyncEventNoChangeCheck(connection, localParent)
}
```

Step 6: Status sync from GitHub to Jira

- Add the following status mapping on the Jira side to ensure that if GitHub issues are closed, the corresponding Jira tickets get closed too:

Jira Incoming

```
def statusMapping = ["open":"To Do", "closed":"Done"]
def remoteStatusName = replica.status.name
issue.setStatus(statusMapping[remoteStatusName])
```

Demo and code

Here are the full code snippets used in this use case:

GitHub Incoming.groovy

GitHub Outgoing.groovy

Jira Incoming.groovy

Jira Outgoing.groovy

Watch the use case in action:

Happy Exalating!