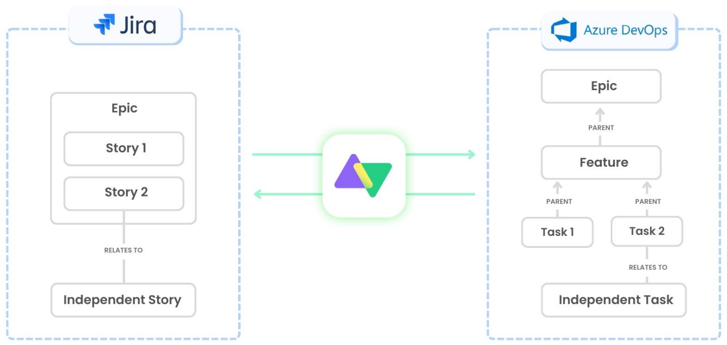
Jira Cloud Azure DevOps: Bi-directional hierarchy sync

Recently a client approached us with a very specific requirement of syncing their Epic-Story relationship from Jira Cloud to

the Feature-Task relationship in Azure DevOps i.e.

- Epic (Jira) becomes a Feature (ADO).
- Story (Jira) becomes a Task (ADO).
- The hierarchy must be maintained bi-directionally.
- Another requirement was to map all Features coming from Jira to be placed under a specific Epic in ADO.
- The issueLInks (specifically the relates to link) must also be maintained bi-directionally.
- A custom status mapping must be maintained from Jira to ADO side only.

The following graphic depicts what the end goal of this synchronization is envisaged to be:



Let us start by looking at each direction separately here:

Jira Cloud to Azure DevOps:

In order to achieve this use case, Jira needs to be ensure that it is sending out the requisite info i.e. parent info and issueLinks.

This can be achieved by adding the following to the Jira Outgoing script (in addition to the default scripts):

Jira Outgoing

```
replica.linkedIssues = issue.issueLinks
replica.parentId       = issue.parentId
```

Once ADO receives this information as part of the replica, it will have several tasks. The first is to map the issue types.

This can be achieved by adding the following to the ADO Incoming script:

ADO Incoming

```
if(firstSync){
   // Set type name from source entity, if not found set a default
   workItem.projectKey  =  "Majids Development"
      def typeMap = [
       "Epic" : "Feature",
       "Story" : "Task"
       ]
    workItem.typeName = nodeHelper.getIssueType(typeMap[replica.type?.name],workItem.projectKey)?.name ?: "Task"
    workItem.summary      = replica.summary
    if(replica.issueType.name=="Epic")
        workItem.parentId = "9785"
    store(issue)
}
```

***Note**: Here 9785 is the workItem number for the static Epic that each Feature must be created under.*

Also note the use of the store() function here to commit the changes.

The next task on the ADO Incoming side is to deal with the Epic-Story hierarchy arriving from the Jira side.

This can be done by using the **parentId** information from the replica. Please add the following to the ADO Incoming script:

ADO Incoming

```
if (replica.parentId) {
   def localParent = syncHelper.getLocalIssueKeyFromRemoteId(replica.parentId.toLong())
   if (localParent) {
      workItem.parentId = localParent.id
   }
}
```

***Note**: The getLocalIssueKeyFromRemoteId() method of syncHelper is immensely useful here to fetch the correct parent locally.*

The last piece of the puzzle would be to remove any existing links on the ADO side, and repopulate them according to the latest information

contained in the replica. We can do that by adding the following to the ADO Incoming script:

ADO Incoming

```
def res =httpClient.get("/Majids%20Development/_apis/wit/workItems/${workItem.id}/revisions",true)
def await = { f -> scala.concurrent.Await$.MODULE$.result(f, scala.concurrent.duration.Duration.apply(1, java.
util.concurrent.TimeUnit.MINUTES)) }
def creds = await(httpClient.azureClient.getCredentials())
def token = creds.accessToken()
def baseUrl = creds.issueTrackerUrl()
def project = workItem.projectKey
def localUrl = baseUrl + '/_apis/wit/workItems/' + workItem.id
int x =0
res.value.relations.each{
    revision ->
        def createIterationBody1 = [
            [
                op: "test",
                path: "/rev",
                value: (int) res.value.size()
            ],
            [
                op:"remove",
                path:"/relations/${++x}"
            ]
        ]

}

def linkTypeMapping = [
    "relates to": "System.LinkTypes.Related"
]
def linkedIssues = replica.linkedIssues
if (linkedIssues) {
    replica.linkedIssues.each{
       def localParent = syncHelper.getLocalIssueKeyFromRemoteId(it.otherIssueId.toLong())
       if (!localParent?.id) { return; }
       localUrl = baseUrl + '/_apis/wit/workItems/' + localParent.id
     def createIterationBody = [
            [
                op: "test",
                path: "/rev",
                value: (int) res.value.size()
            ],
            [
                op:"add",
                path:"/relations/-",
                value: [
                    rel:linkTypeMapping[it.linkName],
                    url:localUrl,
                    attributes: [
                        comment:""
                    ]
                ]
            ]
        ]

def createIterationBodyStr = groovy.json.JsonOutput.toJson(createIterationBody)
        converter = scala.collection.JavaConverters;
        arrForScala = [new scala.Tuple2("Content-Type","application/json-patch+json")]
        scalaSeq = converter.asScalaIteratorConverter(arrForScala.iterator()).asScala().toSeq();
        createIterationBodyStr = groovy.json.JsonOutput.toJson(createIterationBody)
        def result = await(httpClient.azureClient.ws
            .url(baseUrl+"/${project}/_apis/wit/workitems/${workItem.id}?api-version=6.0")
            .addHttpHeaders(scalaSeq)
            .withAuth(token, token, play.api.libs.ws.WSAuthScheme$BASIC$.MODULE$)
            .withBody(play.api.libs.json.Json.parse(createIterationBodyStr), play.api.libs.ws.
JsonBodyWritables$.MODULE$.writeableOf_JsValue)
            .withMethod("PATCH")
            .execute())

    }
}
```

The ADO script can close with a custom status mapping:

ADO Incoming

```
def statusMapping = ["To Do":"New", "In Progress":"Active", "Done" : "Closed"]
def remoteStatusName = replica.status.name
workItem.setStatus(statusMapping[remoteStatusName])
```

Azure DevOps to Jira Cloud:

Similarly to the above, this time ADO needs to ensure that the requisite data is being send to the Jira side. ADO send out the parentId for the parent child type

of relationships and then runs an API call to get all other links.

This can be done by adding the following code snippets to the ADO Outgoing script:

ADO Outgoing

```
replica.parentId = workItem.parentId

def res = httpClient.get("/_apis/wit/workitems/${workItem.key}?\$expand=relations&api-version=6.0",false)
if (res.relations != null)
    replica.relations = res.relations
```

Once Jira receives the data from ADO, it needs to firstly map the issue types:

Jira Incomng

```
if(firstSync){
   issue.projectKey   = "CM"
   // Set type name from source issue, if not found set a default
    def typeMap = [
       "Feature":"Epic",
       "Task":"Story"
       ]
    issue.typeName     = nodeHelper.getIssueType(typeMap[replica.type?.name], issue.projectKey)?.name //?:
"Task"
    issue.summary      = replica.summary

    if (replica.typeName=="Feature") {
        issue.customFields."Epic Name".value = replica.summary
    }
    store(issue)
}
```

The next task is for Jira to create the correct hierarchy. The Epic-Story relationship can be easily created by using the Epic Link field:

Jira Incomng

```
if (replica.parentId) {
   def localParent = syncHelper.getLocalIssueKeyFromRemoteId(replica.parentId.toLong())
   if (localParent) {
      issue.customFields."Epic Link".value = localParent.urn
   }
}
```

The last thing left to do is to ensure that the issueLInks are created appropriately as well. This can be done by a custom API call

by using the **relations** data being sent by ADO (see above):

Jira Incomng

```
replica.relations.each {
    relation ->
    if (relation.attributes.name == "Related"){
            def a = syncHelper.getLocalIssueKeyFromRemoteId(relation.url.tokenize('/')[7])//?.urn
            if (issue.issueLinks[0].otherIssueId != a.id){
                def res = httpClient.put("/rest/api/2/issue/${issue.key}", """
                {
                   "update":{
                      "issuelinks":[
                         {
                            "add":{
                               "type":{
                                  "name":"Relates"
                               },
                               "outwardIssue":{
                                  "key":"${a.urn}"
                               }
                            }
                         }
                      ]
                   }
                }
                """)
            }
        }
}
```

The entire code snippets for this example are included here:

ADO Incoming.groovy

ADO Outgoing.groovy

Jira Incoming.groovy

Jira Outgoing.groovy

A video demonstration of this use case in action can be seen here:

Happy Exalating!