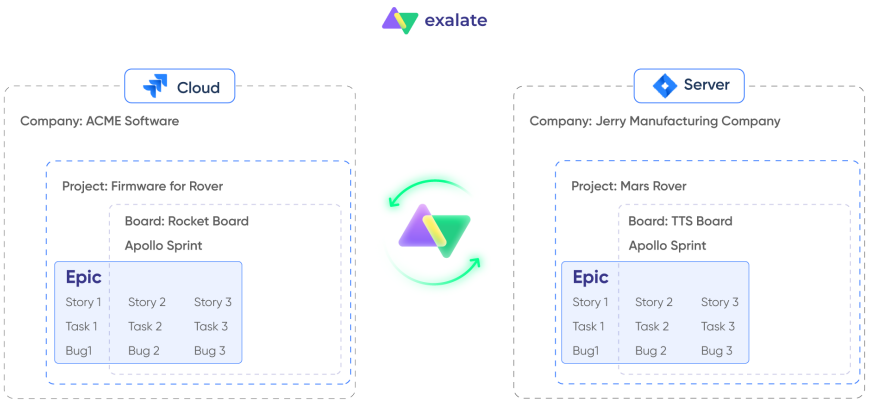
Jira Cloud Jira OnPrem: Full Agile Synchronization

This use case is very often encountered when companies need to collaborate on a certain project but would prefer to continue to use their own systems. Exalate can bridge the gap by syncing all the agile objects and maintain the relationships among items. All items are linked bi-directionally of course and the Jira boards also remain synced in real time.

We use an example of a Jira Cloud and Jira Server synchronization covering the following:

- Sprints are synced bi-directionally with all the meta data
- Epics and its child objects are synced across while ensuring that the relationships among the Epic and child tickets are maintained.
- Adding any tickets to a sprint should reflect on the remote board as well.
- Starting and completing a sprint should sync over and reflect on the remote board.
- Updating issues, changing swim lanes and creating new issues should all reflect on the remote board.
- Dynamically sync across versions and components if they are not present on the remote side.

The following graphic shows some of the objects and relationships we are syncing across:



There are many facets to this use case. Let us look at how each object is synced across:

Epics

We can easily sync Epics in Jira Cloud by including the following lines of script:

- Cloud Outgoing: *Epic.sendInAnyOrder()*
- Cloud Incoming: *Epic.receive()*

We can include the exact same lines on the Server side to sync Epics, but please ensure that you have placed the Epics.groovy script in the correct Jira directory.

Sprints

In order to send over the sprints from the Cloud side, we used the following script in the Outgoing side of Jira Cloud:

Cloud Outgoing

```
def boardIds = ["5"] //Boards which sprints will get synced
if(entityType == "sprint" && boardIds.find{it == sprint.originBoardId}){
    replica.name = sprint.name
    replica.goal = sprint.goal
    replica.state = sprint.state
    replica.startDate = sprint.startDate
    replica.endDate = sprint.endDate
    replica.originBoardId = sprint.originBoardId
}
```

***Note**: Please remember to configure the board id that you need to sync in the first line.*

On the Jira Server side, we included the following script to receive and process the sprint correctly:

Server Incoming

```
if(entityType == "sprint"){
    //Executed when receiving a sprint sync from the remote side
    def sprintMap = ["5":"37"] //[remoteBoardId: localBoardId]
    sprint.name = replica.name
    sprint.goal = replica.goal
    sprint.state = replica.state?.toUpperCase()
    sprint.startDate = replica.startDate
    sprint.endDate = replica.endDate
    def localBoardId = sprintMap[replica.originBoardId]

    if(localBoardId == null){
       throw new com.exalate.api.exception.IssueTrackerException("No board mapping for remote board id
"+replica.originBoardId)
    }
    sprint.originBoardId = localBoardId //Set the board ID where the sprint will be created
}
```

***Note:** The only things that you need to configure is the sprintMap to include the board mappings between the two systems.*

Also, please remember to create a trigger to sync the sprints!

The last piece of the puzzle is to assign the correct Sprint field value to the issues under sync. On the server side, you will need to include the following in the Incoming side:

Server Incoming

```
def sprintV = replica.customFields.Sprint.value?.id.collect{
    remoteSprintId ->
    return nodeHelper.getLocalIssueKeyFromRemoteId(remoteSprintId, "sprint")?.id?.toString()
    }.findAll{it != null}
issue.customFields."Sprint".value = sprintV
```

Versions

In order to dynamically sync Versions between Cloud and Server side, we include the following script on the Outgoing side of both Server and Cloud:

Outgoing

```
replica.fixVersions = issue.fixVersions
replica.affectedVersions = issue.affectedVersions
```

and the following on the Incoming script on Server and Cloud to create any missing sprints:

Incoming

```
issue.fixVersions = replica
  .fixVersions
  .collect { v -> nodeHelper.createVersion(issue, v.name, v.description) }
issue.affectedVersions = replica
  .affectedVersions
  .collect { v -> nodeHelper.createVersion(issue, v.name, v.description) }
```

Components

In order to dynamically sync Components between Cloud and Server side, we include the following script on the Outgoing side of both Server and Cloud:

Outgoing

```
replica.components       = issue.components
```

In order to dynamically create the components on the Cloud side, add the following script on the Incoming side of the Cloud:

Cloud Incoming

```
issue.components = replica.components.collect{
        nodeHelper.createComponent(
                issue,
                it.name,
                it.description,
                it.leadKey,
                it.assigneeType.name()
        )
}
```

In order to select the correct component on the Server side, please include the following script on the Incoming side of the Server:

Server Incoming

```
issue.components = replica.components
    .collect { remoteComponent ->
        nodeHelper.getComponent(
            remoteComponent.name,
            nodeHelper.getProject(issue.projectKey)
   )
}.findAll()
```

You can download the entire code used in this use case here.

A short video demonstration of the use case in action is as follows: