In the previous part of the tutorial, the implementation of the dataset, datastore and configuration class enabled the **provisioning of user input data**. Now these should be processed and the **desired output should be made to the user**. In the context of the Jira input component this means that the query of the Jira API is carried out and the received data is further processed. For this the **Partition Mapper** and the **Source** of the component must be implemented.

After this tutorial you can:

- implement a **simple partition mapper**.
- extract data with the source class **extract**, **edit** and **return as record to the user**.

1. Partition Mapper

As learned in chapter 3 of the tutorial, the Partition Mapper must implement the three methods *"Assessor", "Split "* and *"Emitter "* and has the task to split the data processing into parallel executable parts. Since the RESTful query made by the Jira input component cannot be parallelized, the implementation of our Partition Mapper proves to be very simple.

1.1 Assessor

The assessor should estimate the number of task parts. Since we do not divide the task, the method returns **1** as *"long "*.

```
@Assessor
public long estimateSize() {
     return 1L;
}
```

1.2 Split

The split method should return the list of subordinate, split InputMappers. In our case, this list only contains the partition mapper itself, which is why it is returned in conversion of the singleton pattern.

```
@Split
public List<OdiSysInputMapper> split(@PartitionSize final long bundles) {
    return singletonList(this);
}
```

1.3 Emitter

The emitter creates the source classes, which then do the tasks. With the Jira-Input component it is called only once and creates an object of the *"OdiSysInputSource "* with its own configuration, service and RecordBuilderFactory.

```
@Emitter
public OdiSysInputSource createWorker() {
    return new OdiSysInputSource(configuration, service, recordBuilderFactory);
}
```

1.4 Entire partition mapper class

In addition to the methods just implemented, the partition mapper class also contains the declaration of attributes and a constructor. The annotation *"@PartitionMapper "* indicates that the class is this one.

```
@Version(1)
@Icon(value= Icon.IconType.STAR)
@PartitionMapper(name = "OdiSysInput")
@Documentation("")
public class OdiSysInputMapper implements Serializable {
    private final OdiSysInputMapperConfiguration configuration;
    private final OdiSysService service;
    private final RecordBuilderFactory recordBuilderFactory;

    public OdiSysInputMapper(@Option("configuration") final OdiSysInputMapperConfiguration configuration,
                             final OdiSysService service,
                             final RecordBuilderFactory recordBuilderFactory) {
        this.configuration = configuration;
        this.service = service;
        this.recordBuilderFactory = recordBuilderFactory;
    }

    @Assessor
    public long estimateSize() {
        return 1L;
    }

    @Split
    public List<OdiSysInputMapper> split(@PartitionSize final long bundles) {
        return singletonList(this);
    }

    @Emitter
    public OdiSysInputSource createWorker() {
        return new OdiSysInputSource(configuration, service, recordBuilderFactory);
    }
}
```

2. Source

The method name in the Partition Mapper that creates the source object also reveals its task. *The method is called "CreateWorker() "* and the task of the object is the execution of the actual task.

Specifically for our Jira input component this means that the API query is made and the data is returned to the user.

2.1 API call in the first run of the producer

The first task is to query the API and store the results in a string. The query is made using the picture library *"Unirest "*. We extract the user's input from the configuration, i.e. the user name, password, URL and ID of the project.

To the URL we append the path *"/rest/api/2/search?jql=project="*, as well as the project ID, since this is the path where Jira has the second version of the RESTful API.

Finally, we extract the body from the result of the HTTP request and get the JSON response to our request.

```
String resString = "";
try {
   HttpResponse<String> res = Unirest.get(
           configuration.getDataset().getDatastore().getUrl()
                   + "/rest/api/2/search?jql=project="
                   + configuration.getDataset().getProjectId())
            .basicAuth(configuration.getDataset().getDatastore().getUsername(),
                   configuration.getDataset().getDatastore().getPassword())
            .asString();

    resString = res.getBody();
} catch (UnirestException e) {
    e.printStackTrace();
}
```

2.2 Returning a record

The producer method returns a record as the return value. A record can be created with the *"RecordBuilderFactory "* given in the constructor.

Data can be added to the record using the methods

- *withString()*
- *withBytes()*
- *withBoolean()*
- *with DateTime()*
- *with double()*
- *with int()*
- *withLong()*
- *with float()*
- *withArray()*
- *with Timestamp()*
- *withRecord()*

can be added. We add the received JSON as string and the key *"data"* to the record:

```
return builderFactory.newRecordBuilder().withString("data", resString).build();
```

2.3 Exit in the second pass of the producer

Now that the work is done and the user's data is in a record, the execution of the *"Producer Method "* must be stopped. Thanks to the *"@Producer "* annotation, the method is executed until it returns a *"zero "* value.

So we have to make sure that after one execution, in the second execution the value *"zero "* is returned to end the producer execution. This can be done with a boolean that is initialized to *"false "* and set to *"true "* in the first execution. The method then looks as follows:

```
private boolean executed = false;

@Producer
public Record next() {
    if (!executed) {
        String resString = "";
        try {
            HttpResponse<String> res = Unirest.get(
                    configuration.getDataset().getDatastore().getUrl()
                            + "/rest/api/2/search?jql=project="
                            + configuration.getDataset().getProjectId())
                    .basicAuth(configuration.getDataset().getDatastore().getUsername(),
                            configuration.getDataset().getDatastore().getPassword())
                    .asString();

            resString = res.getBody();
       } catch (UnirestException e) {
            e.printStackTrace();
      }

      executed = true;
      return builderFactory.newRecordBuilder().withString("data", resString).build();
   }
   return null;
}
```

2.4 Entire OdiSysInputSource Class

In addition to the method just implemented, the source class also contains the declaration of attributes and a constructor:

```
@Documentation("")
public class OdiSysInputSource implements Serializable {
    private final OdiSysInputMapperConfiguration configuration;
    private final OdiSysService service;
    private final RecordBuilderFactory builderFactory;
    private boolean executed = false;

    public OdiSysInputSource(@Option("configuration") final OdiSysInputMapperConfiguration configuration,
                             final OdiSysService service,
                             final RecordBuilderFactory builderFactory) {
        this.configuration = configuration;
        this.service = service;
        this.builderFactory = builderFactory;
    }

    @Producer
```

```
    public Record next() {
        if (!executed) {
            String resString = "";
            try {
                HttpResponse<String> res = Unirest.get(
                        configuration.getDataset().getDatastore().getUrl()
                                + "/rest/api/2/search?jql=project="
                                + configuration.getDataset().getProjectId())
                        .basicAuth(configuration.getDataset().getDatastore().getUsername(),
                                configuration.getDataset().getDatastore().getPassword())
                        .asString();
                ;
                resString = res.getBody();
            } catch (UnirestException e) {
                e.printStackTrace();
            }
            executed = true;

            return builderFactory.newRecordBuilder().withString("data", resString).build();
        }
        return null;
    }
}
```

4. Further tutorials

Feel free to send us a mail to talend@odisys.de for any questions or notes .

Source Code:

- https://github.com/neidigsi/jira_talend_component/

The following tutorials can be found here:

- Tutorial - Talend Component Kit #1: Setting Up the Development Environment
- Tutorial - Talend Component Kit #2: Testing your own component
- Tutorial - Talend Component Kit #3: Building an Input Component (Dataset, Datastore, Partition Mapper & Source))
- Tutorial - Talend Component Kit #4: Implementation of the Dataset and -store of a Jira-Input)
- Tutorial - Talend Component Kit #5: Implementation of the Partition Mapper and the Source of a Jira Input
- Tutorial - Talend Component Kit #6: Implementing "List" Input and further Output-Processing
- Tutorial - Talend Component Kit #7: Implement "Advanced Settings" of a Jira input component