DATAGROUP

Atomic KIs

It's vital to keep KIs as atomic as possible, meaning we should break down the steps into smaller, manageable tasks. Let’s explore an example:

```
ki
  name: "New incident, proceed with adding member"
  description: "New incident, proceed with adding member"
  version: "1@cln8sd29t04zz0186ataggjya_clnek6ibg05wx0106o0xre4iv"

on
ogit/name == "default_issue_node"

when
ProcessIssue
ticketId =~ "IN"
description =~ "Nutzer"
description =~ "Verteiler"
description =~ "Aktion: aufnehmen"
category == "DG-KIT-Exchange Änderung E-Mail Verteiler"
serviceName == "DG-ZIT-Mail"
assignmentGroup == "DG-DGSD-HIRO"
not worklog_createpersons =~ "TASKEXECUTOR-\d+@DG"


do

result: userName = javascript("
  let description = `${description}`;
  description.match(/.*Nutzer:(.*).*/)[1].split(',').map((s) => s.trim().replace(/\s/g,'.'));
  ")

result: distributionList = javascript("
  let description = `${description}`;
  description.match(/.*Verteiler:(.*).*/)[1].split(',').map((s) => s.trim().replace(/ /g,'.'));
  ")


AcceptTicket = "true"
CheckDistributionListExists = "true"
CheckUserExists = userName
```

This initial KI contains too many different steps that can be split into more atomic steps for easier reuse. In this KI, we are currently making decisions about:

1. Issue Type
2. Action Type
3. Assignment Group
4. Requestor

In the Do block, we are:

- Extracting Data
- Executing next course of Actions

These checks can be separated into different KIs, allowing them to be easily reused and making it possible to add new ones when more validation steps are required.

Important Considerations:

1. **Classification of Issue Type, assignment group or requestor:** This type of KI doesn't require a specific ToDo but you need to ensure that the conditions in the When block are appropriate for the execution of these KIs.
2. **Action Type:** This will classify the type of ticket to be resolved. It's important to define the Parent ToDo, allowing the default "ProcessIssue" ToDo to be deleted and replaced with a more representative variable that leads the task resolution.
3. **Proper usage of "If-Else" Structures:** If you find yourself using many "if-else" statements in the Do block, it's a sign that the KI can be split into smaller atomic steps. "If-else" structures must only be used for interpreting results from external calls.
4. **Data Extraction:** When extracting data from free text, ensure that each extraction is handled in different KIs so that validation errors can be considered separately. This type of KI doesn't require a ToDo. In case of errors, they will be stored in validation error list.
5. **Validation Checkpoints:** Before proceeding to execution steps, it's important to have a validation checkpoint for all initial checks. This can be done with a decision KI.
6. **Action KIs:** These remain independent and atomic, each with its own ToDo and information variables that provide the context for the execution of these steps.

Breaking down task into atomic steps

Now Let´s see how we would break down this initial KI into atomic ones:

CLASSIFICATION OF ISSUE TYPE

1. KI: identify Issue Type

```
when
category =~ "DG-KIT-Exchange"
not ogit/Automation/issueType

do

ogit/Automation/issueType = "Exchange"
```

In the ontology we have the following atribute: "**ogit/Automatio/issueType**“ defined and it is tied to the Task. We can use this variable to identify the Issue Type of a ticket.

IDENTIFICATION OF ACTION TYPE

2. Identify action Type

```
when
ProcessIssue
description =~ "Nutzer"
description =~ "Verteiler"
description =~ "Aktion: aufnehmen"
ogit/Automation/issueType == "Exchange"

do

AddMemberToList = "true"  # Parent to do for this ticket
delete(ProcessIssue)
```

At the moment we identify the action type, we can then delete the general „ProcessIssue“ and find a specific parent ToDo for this action. In this case “AddMemberToList“.

In the WHEN block we use all the conditions, in this case coming from free text that informs us about the task at hand.

DATA EXTRACTION

3. Retrieve User information

```
when
ogit/Automation/issueType == "Exchange"
not userName
description =~ "Nutzer:(.*)"

do

result: LOCAL::user = javascript("
  let description = `${description}`;
  description.match(/.*Nutzer:(.*).*/)[1].split(',').map((s) => s.trim().replace(/\s/g,'.'));
  ")

  if LOCAL::user != "" then
    username = LOCAL::user
  else
    log.error("It was not possible to retrieve userName")
    add(validationErrors = "userName couldnt be retrieved")
  end
```

4. Retrieve distribution list information from issue

```
when
ogit/Automation/issueType == "Exchange"
not distributionList
description =~ "Verteiler:(.*)"

do

result: LOCAL::DL = javascript("
  let description = `${description}`;
  description.match(/.*Verteiler:(.*).*/)[1].split(',').map((s) => s.trim().replace(/ /g,'.'));
  ")

if LOCAL::DL != "" then
    distributionList = LOCAL::DL
else
    log.error("It was not possible to retrieve distribution List")
    add(validationErrors = "distribution list couldnt be retrieved")
end
```

In KI 3 and KI 4 we purely extract data that will be using during the resolution of the tasks.

If it was not possible to extract the parameters we need, we will be reporting these as validations errors to be checked in the decision KI

ERROR VALIDATION KIS

In this KI, which is a decision-making KI, we wait for a "maxValidationTime" in seconds to ensure that no additional KIs are needed to identify issue, action type or extract information. Once this validation step passes, we can proceed with executing the steps necessary to resolve the task at hand.

1. Validating request before proceding with issue

```
when
 AddMemberToList
 ? maxValidationTime := 5
 ExecIdle > maxValidationTime
 not validationErrors
 list userName
 distributionList
 not AcceptTicket
 not CheckUserExists
 not CheckDistributionListExists


do
AcceptTicket = "true"
CheckUserExists = userName
CheckDistributionListExists = distributionList
```

Please note that "AddMemberToList" is now the parent ToDo we use to navigate the task until its resolution.

Whenever validation errors appear, a new teaching session will be created so you can then create a KI for these cases.

2. KI: There are validations errors, please inform agent

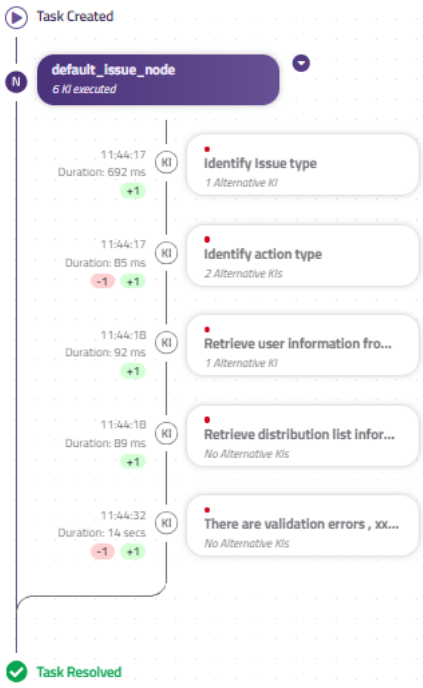
```
when
 ? maxValidationTime := 5
 ExecIdle > maxValidationTime
 list validationErrors
 not AddComment

do

# Return ticket to HUMAN with validationError
AddComment = validationErrors
```

Note: AddComment will execute a KI: “Update ticket“ with all the validations Errors collected during the process.

This is how a task with validation errors would be resolved in the Task inspector.



IT’s that simple.

ACTION KIS

Action Kis: CheckUserExists, CheckDistributionListExists, AddMember and AddComment KIs remain the same and will be triggered to resolve the task.

1. KI: Check Distribution List exists

```
when
 CheckDistributionListExists
 distributionList


do

bodyPSProxy = '{
"Module": "getdistributiongroup",
"Identity": "${distributionList}"
}'

body: LOCAL::BODY,
code: LOCAL::CODE,
headers: LOCAL::HEADERS,
exit: LOCAL::SYSTEMRC = action(capability: "PSProxy",
body: bodyPSProxy
)

# result interpretation
if LOCAL::CODE == "200" then
        if "${LOCAL::BODY#RecipientTypeDetails}" =~ "MailUniversalDistributionGroup"  then
            distributionListID = "${LOCAL::BODY#SamAccountName}"
            distributionListExists = "yes"
            AddComment = " Distribution list ${distributionList} exists"
        else
            distributionListExists = "no"
            AddComment = "Distribution list ${distributionList} does NOT exist"
        end
        delete(CheckDistributionListExists)

else
        AddComment = "${distributionList}: Failed to check distribution List existance"
end
```

2. KI: Check User Exists

```
when
CheckUserExists
userName
? list userStatus

do

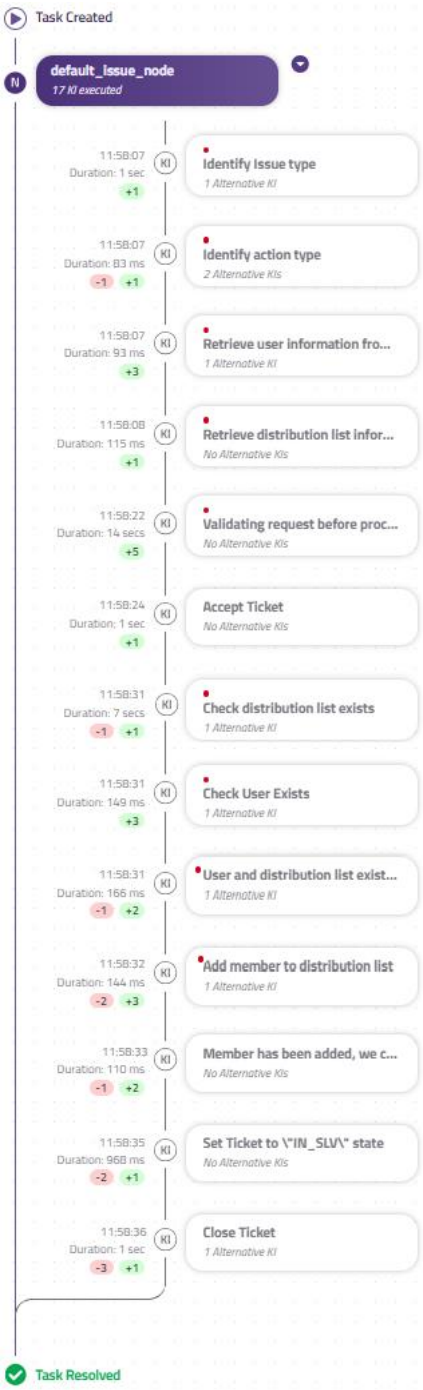
bodyPSProxy = '{
"Module": "FindRecipient",
"Identity": "${userName}"
}'

body: LOCAL::BODY,
code: LOCAL::CODE,
headers: LOCAL::HEADERS,
exit: LOCAL::SYSTEMRC = action(capability: "PSProxy",
body: bodyPSProxy
)

# result interpretation
if LOCAL::CODE == "200" then
   if "${LOCAL::BODY#Status}" == "SUCCESS" and "${LOCAL::BODY#ObjectClass}" == "user" then
      userStatus[userName] = "active"
      AddComment = " User ${userName} exists"
   else
      userStatus[userName] = "inactive"
      AddComment = "${userName}: Failed to check user existance or user doesnt exist"
   end
   delete(CheckUserExists)
else
   userStatus[userName] = "inactive"
   AddComment = "${#userName}: Failed to check user existance or user doesn‘t exist"
end
```


IT’s that simple.

This is how the task would be successfully resolved in the Task Inspector.



IT’s that simple.

Adding additional steps for Task Resolution

Tickets for a specific topic have being automatically resolved. However, after six months, a new policy was introduced, changing the resolution process for these tickets. As a result, additional steps now need to be added.

In this case, we will implement a task approval step that requires management´s approval before executing the resquested action.

We aim to implement these KIs seamlessly so they can be triggered for this specific task. Since this is part of the new policy resolution, it serves as a good example of where we can use policy tier feature within a KI.

1. KI: Approval is required in the process

```
ki
  name: "Approval is required in the process"
  description: "Approval required before adding member"
  version: "1 "
  tier: "policy"
on
   ogit/name == "default_issue_node"

when
   ogit/Automation/issueType == "Exchange"
   AddMemberToList
   not approve
   distributionListID
   distributionListExists == "yes"
   userName
   userStatus[userName] == "active"

do
   GetApproval = "choice"
   approvalTitle = "HIRO needs your review"
   approvalBody = "Please confirm your approval to add ${userName} to distribution list ${distributionListID}"
   emailAddress = "beatriz.mora@datagroup.de"
   user = userName
```

2. KI: Get Manager approval

```
ki
  name: "Get Manager approval"
  description: "Get human approval before executing an action"
  version: "1 "
  tier: "policy"
on
ogit/name == "default_issue_node"

when
GetApproval == "choice"
ogit/_id # ogit/_id from the task
approvalTitle
approvalBody
emailAddress

do

 LOCAL::model = '{
  "action":"approve",
  "task":"${approvalTitle}",
  "template":"${approvalBody}",
  "controls":[
     {
       "approve":{"/approve":"true"},
       "reject":{"/approve":"false"}
     }
   ]
  }'

  exec: LOCAL::RESULT,
  stdout: LOCAL::OUTPUT,
  stderr: LOCAL::ERROR,
  exit: LOCAL::SYSTEMRC = action(
     capability: "AskApprove",
```


IT’s that simple.

```
    user_email: emailAddress,
    issue_id: ISSUE::ogit/_id,
    question_model: model
    )


if LOCAL::SYSTEMRC == 0 then
  log("SUCCESS: ${LOCAL::RESULT}")
else
  log.error("ERROR ${LOCAL::ERROR}")
end

eject_issue(message: "Wait for approve")
```

3. KI: Request Approved

```
ki
  name: "Request approved"
  description: "Request approved"
  version: "1 "
  tier: "policy"
on
   ogit/name == "default_issue_node"

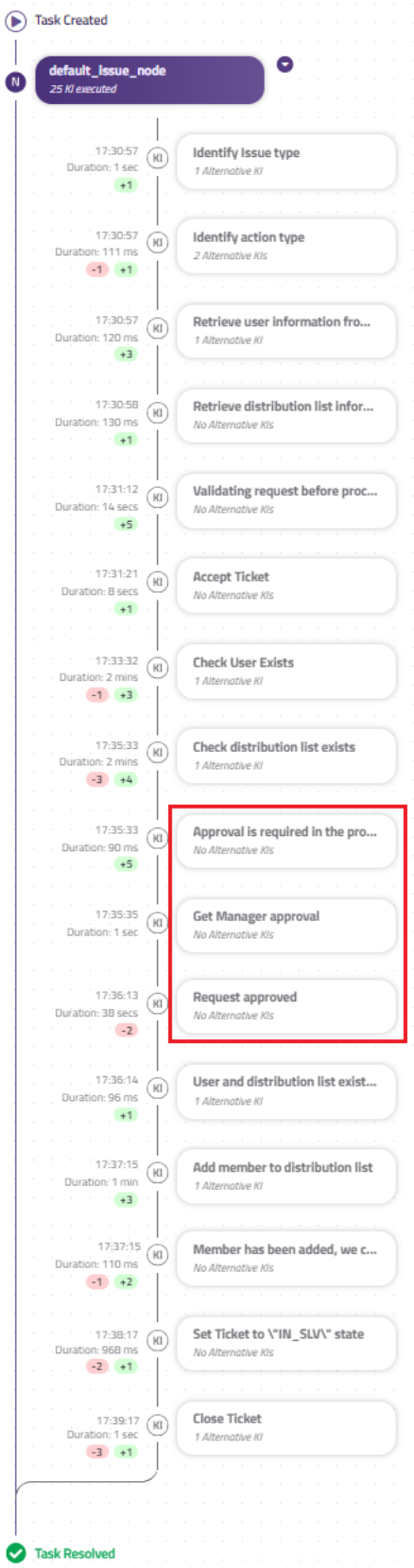
when
   GetApproval
   approve == "true"
   userName
   userStatus

do
   delete(approve)
   delete(GetApproval)
```


IT’s that simple.

The policy tier will ensure that these KIS are executed first as soon as all conditions from ON and WHEN block are met.

Go **here** to learn more about tiers.



IT's that simple.

Exponential teaching

This case illustrates how exponential teaching occurs. Initially, we need to guide HIRO through several steps to handle the first ticket. As a result, when similar tickets come into HIRO in the future, it can solve them automatically. For example, one day, a ticket arrived with missing information, causing one of the validations KIs to fail and collect Validation Errors. This required a second teaching session to address just that single step—Teaching Session 2 focused on teaching HIRO how to deal with Validation errors.

Six months later, a new company policy is introduced for management approval step before executing any actions. This required another teaching session to introduce the approval process step. Subsequently, an additional teaching session was needed to handle cases where approval was rejected.

You can have a look at the teaching sessions created in the bootcamp for this ticket.

1. Teaching Session 1: " **Example Teaching Session Vertailer**"

You can check for this Task ID: cln8tg3pa0fpl01868wmrpcjl_cls1ivltzvi6h0150prn4vjpw in haas004 (bootcamp)

- ✓ Identify Issue type
- ✓ Identify action type
- ✓ Retrieve user information from issue
- ✓ Retrieve distribution list exists
- ✓ Check user exists
- ✓ Walidating request before proceding with issue
- ✓ User and distribution list exist, Add member
- ✓ Add member to distribution list

2. Teaching Session 2: "**Example Teaching Session Vertailer - There are validation Errors"** (taskID: cln8tg3pa0fpl01868wmrpcjl_clnlfwkp4ebck0123h5l1k3c2)
    - ✓ There are validation errors, please inform agent

3. Teaching Session 3: "**Example Teaching Session Vertailer – Adding approval process"** (taskID: cln8tg3pa0fpl01868wmrpcjl_clnlfxh7bedy4012329nautap)
    - ✓ Approval is required in the process
    - ✓ Get Manager approval
    - ✓ Request approved

4. Teaching Session 4: “ **Example Teaching Session Vertailer – Approval Rejected**“ (taskID: cln8tg3pa0fpl01868wmrpcjl_clnn8qte9a5wl0106kxawqgvi)
    - ✓ Request rejected

From now on, Kis from these four teaching sessions will be recombined to resolve this type of tickets. As you can see the first teaching session required more efforts as to explain all the steps, while the next ones added just only a few steps each

Exponential Teaching with HIRO involves incrementally transferring actionable knowledge from SMEs to the HIRO engine. Initially, HIRO™ understands the environment but lacks the detailed operational knowledge necessary to perform specific tasks. Through teaching sessions, HIRO learns atomic steps, known as Knowledge Items, which it can later recombine to tackle real-world tasks effectively. This approach mirrors how a new colleague learns from seasoned team members, gradually building expertise and connecting knowledge across different domains. The power of Exponential Teaching lies in its ability to leverage and reuse existing knowledge, achieving high levels of automation and efficiency across teams and departments.

KEY PRINCIPLES

- Knowledge capturing (and KI writing) is done task by task
- It is backward looking (“How have I solved a specific problem?”) instead of forward looking (“How would I solve a specific problem?”)
- It is based on “real tasks”, that have been ejected by HIRO
- Knowledge Items are deployed in a timely manner after solving a given task
- Knowledge is captured directly from SME (either via UI or KE) and translated into KIs (either via UI or KE)