A Model-Driven Approach To Creating Questionnaires In Oracle Digital Assistant Using Composite Bag Entities

Frank Nimphius, April 2020

Composite bag entities group related entities and items of type string, location and attachment to a single entity that represents a real world object like an order, a booking, a reservation, an incident report and many more. The value for each item in a composite bag entity can be resolved through natural language processing (nlp) or by the user when prompted to enter a value.

A quality of composite bag entities is that they can be resolved without developers to write dialog flow states for each of the contained bag items. Using the `System.CommonResponse` component or the `System.ResolveEntities` component, user prompts are automatically generated until the user cancels or completed providing required information.

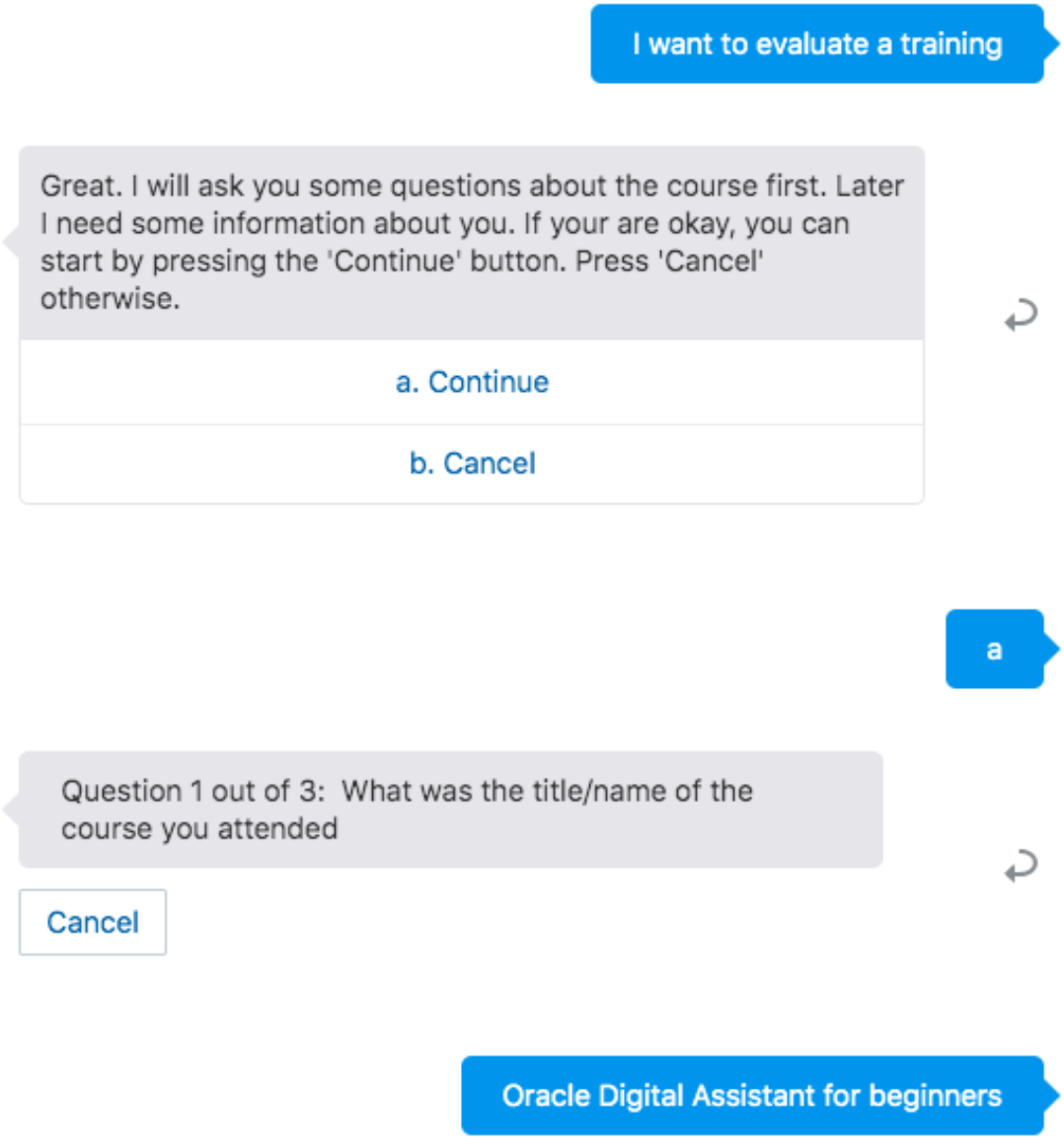
In the following, this article outlines an approach for building model-driven questionnaires using composite bag entities, the common response built-in component and resource bundles. The example use case is a course evaluation skill that students use at the end of a training class to provide feedback about the instructor and the training material and content.

Note: This article aims for advanced users and requires a good understanding of dialog flows in Oracle Digital Assistant and the System.CommonResponse component. You should also know about custom components in Oracle Digital Assistant to have an idea how the questionnaire could be processed to a backend.

Exploring The Sample

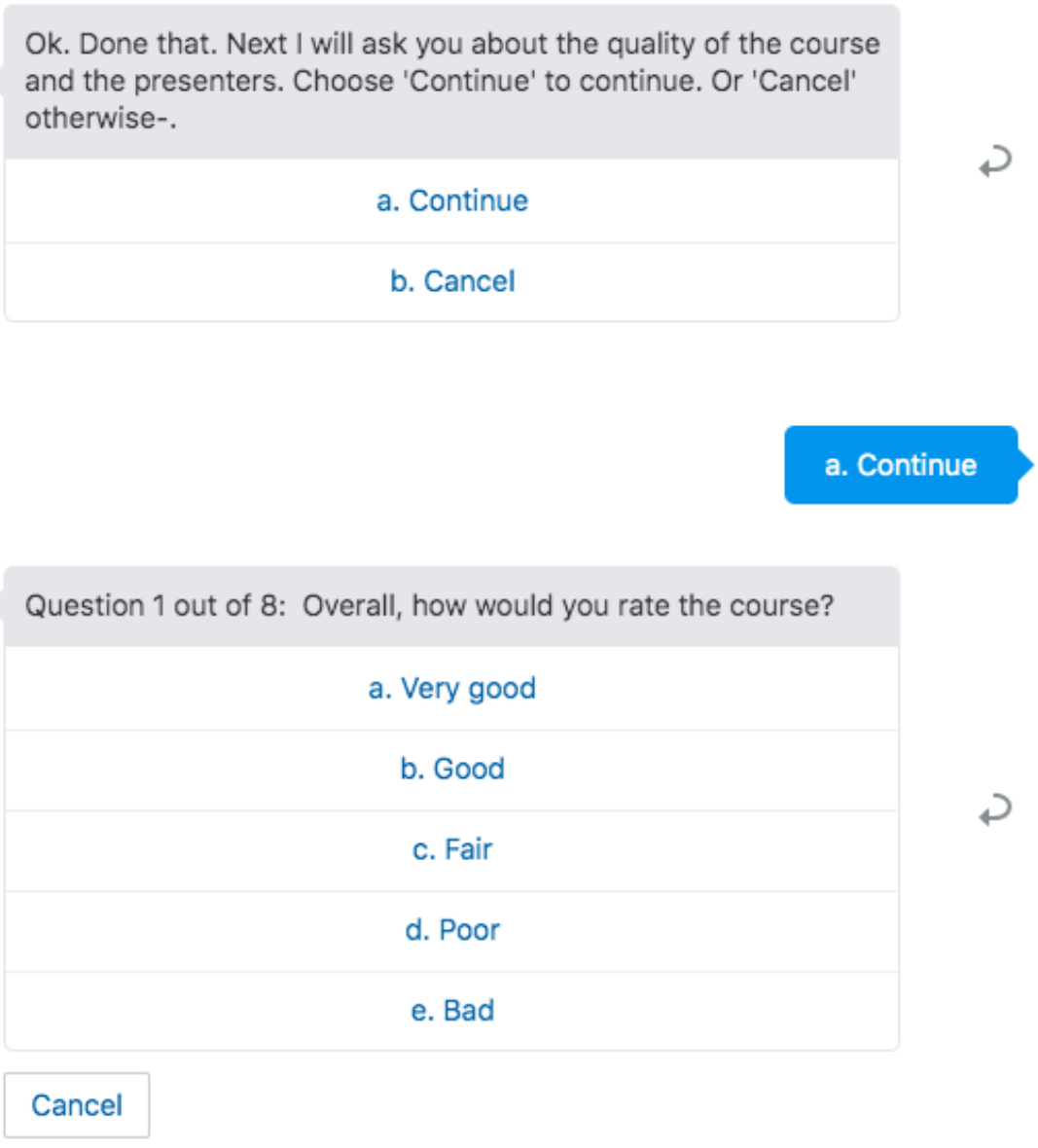
The following screen shots show the course evaluation questionnaire in action. The questionnaire has three sections: questions about the course to evaluate, questions about the course material and trainers and questions about the student rating the course. All three sections are modeled as a composite bag entity so that the user interaction is driven by the entity. For better a better user experience, the dialog flow shows dialogs in between that tell users about what they have achieved and what they are going to do. Each dialog and question offers users an exit out of the questionnaire.

Note: all labels, values and prompts are read from resource bundles that not only put strings into a single manageable place but also allow translation into multiple languages.

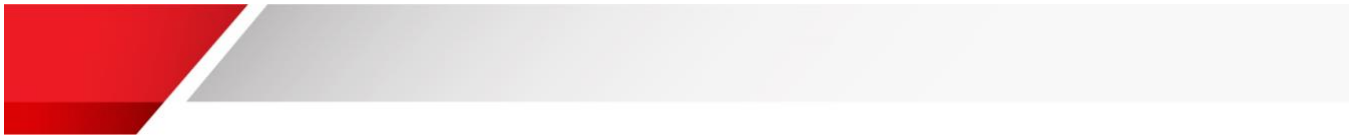


The image above shows questions with a free text answer. In the sample, text entries are not validated against a list of values or a pattern, though they could.

In the images below, questions are displayed with a re-defined list of answers. The answers are read from a value list entity and the answer provided by the user will be validated against the value list



Notice in above image how each question has a *cancel* button for the user to exit the questionnaire. You can decide to show the button all the time (as in this sample), or conditionally using expressions after a user provided an invalid user input. Also notice how the prompt of the question provides a hint for users about the questions answered so far and the questions to go.



At the end, the questionnaire lists the provided answers to then ask the user to submit the questionnaire or cancel it. Submitting a questionnaire would call a custom component (not provided in the sample) that calls a remote backend services to persists the user information. As you will see later, the summary is read from a data object that holds all the provided answers. This object can be passed to a custom component where it can be worked with as a JSON object.

Building the (Conversation) Model

For this sample, there are three types of questions to ask in the course evaluation: Questions about the course title, presenter and location, questions about the quality of the presenters, the material and the content, and personal information questions. For each section, this sample has a composite bag entity created. The composite bag items then become the questions that the user is prompted for to answer.

- CourseInformation_CBE
- StudentInformation_CBE
- CourseEvaluation_CBE

Of course, all questions could have gone into a single composite bag item. However, from a usability perspective, what would you prefer: 24 questions asked in a row or 8 questions in three blocks with a mental break between that summarized what has been achieve so far, what is coming next and that allows you to stop the questionnaire if you wanted? Hopefully you said yes, to the three blocks of questions. In the course evaluation sample skill that you can **download from here**, the three blocks don't have 8 questions each but 3 – 8 – 4.

Note: The "_CBE" postfix was chosen to visually distinguish composite bag entities from other custom entities in then Oracle skill's entity editor. You can adopt this naming convention, find your own convention or be without.

Alternative Solution

An alternative to this model-driven (or entity driven if you like) approach using composite bag entities is to encapsulate the questionnaire in a custom component. The custom component could read questions and, if applicable, a choice of pre-defined answers from a database table, which easily can be exposed through Oracle Rest Data Service (ORDS). You could also use VBCS to build an authoring system for administrative people to add and maintain questionnaires.

Advantage of a custom component approach

- Meta-data driven approach with all questions being managed in database
- Suitable for questionnaires that require different questions for different trainings
- Suitable for frequent changes in the questions
- Changes are applied to the database and not the bot
- Highly reusable because custom component operates as a "black box"

Advantage of a model-driven approach using composite bag entities

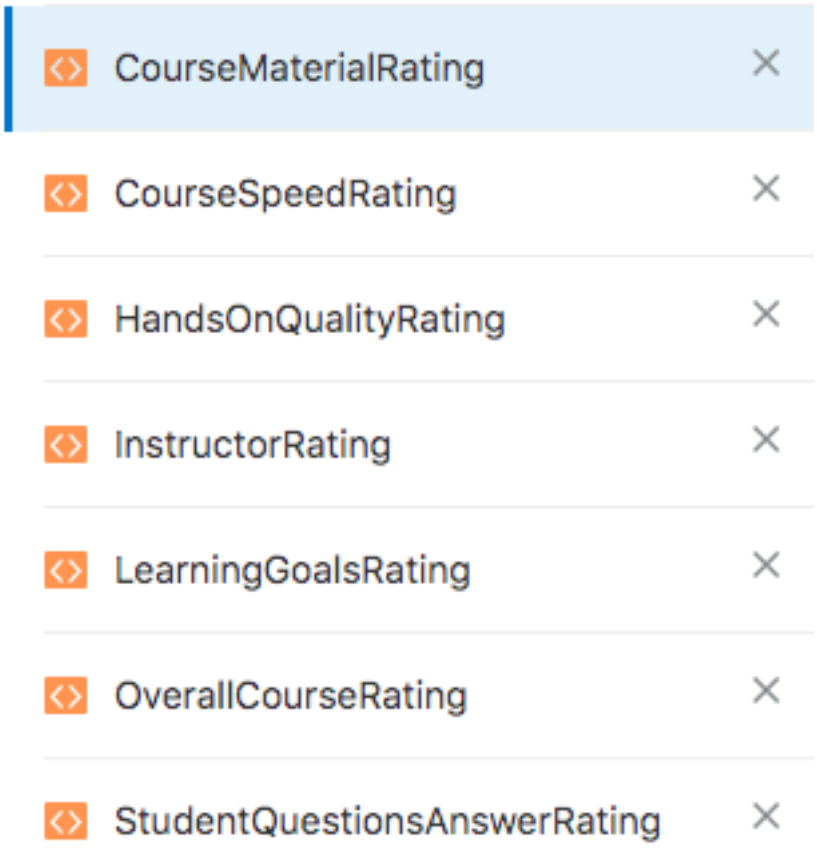
- Less code to write
- Reusable through creating a questionnaire skill that then is reused through digital assistants
- Access to skill resource bundles for multi-language support and good coding practices

- Advanced common response component features like optimization for text channels and multiple responses displayed in a single bubble
- Easy option to display intermediary dialogs or pass user input back to the intent engine

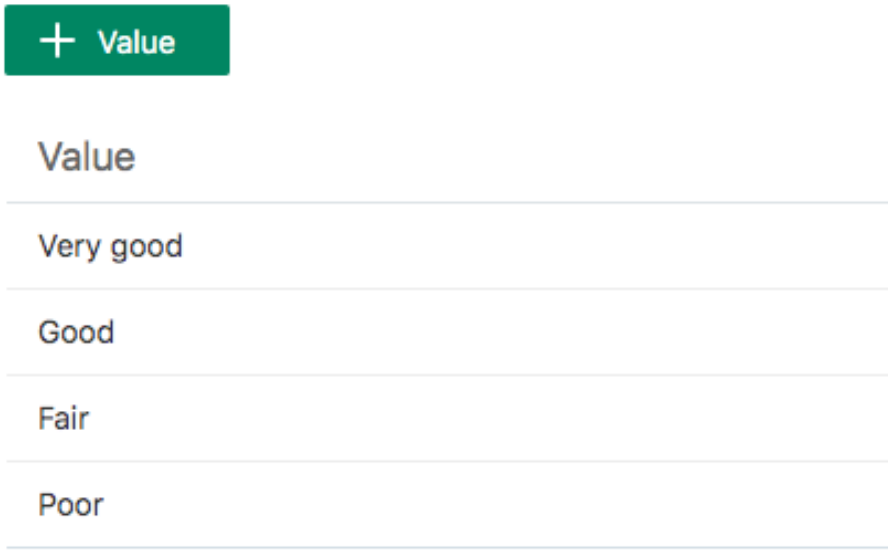
In summary, use case matters for whether you implement a metadata driven or model driven approach. This article follows the model driven approach.

The Composite Bag Entities

Each question in the model-driven questionnaire is designed as a bag item in a composite bag entity. For those questions that should be validated against a list of values or a pattern, or questions that should display a choice of answers to select from, you first need to create the entities.

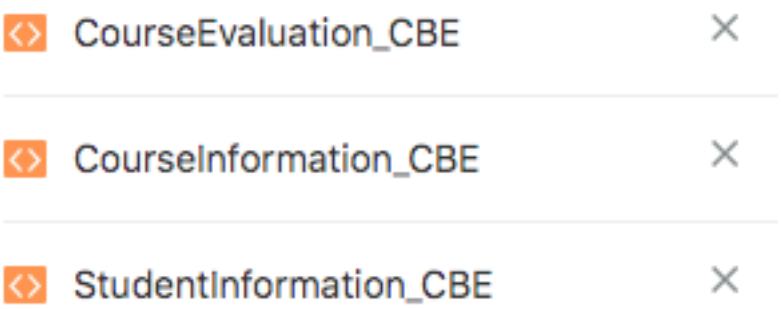


The **CourseMaterialRating** entity is a list value entity. Valid answers for this question are: *Very good*, *Good*, *Fair* and *Poor.*



Hint: To display the values in the user language (assuming a multi-language use of the questionnaire) then you need to make sure that you create resource bundle entries for each value. So your resource bundle would have a key name "Very good", "Good" etc.

With the entities in place, you can then start building the composite bag entities. The example uses 3 composite bag entities as shown in the image below:



As mentioned earlier, bag items of the composite bag entity define the questions to display in the questionnaire. The image below shows the "questions" defined for the **CourseEvaluation_CBE** composite bag entity. Notice that the questions are mostly entity based, except for the question about suggested course enhancements.

Name *

CourseEvaluation_CBE

Description

Configuration

Type *

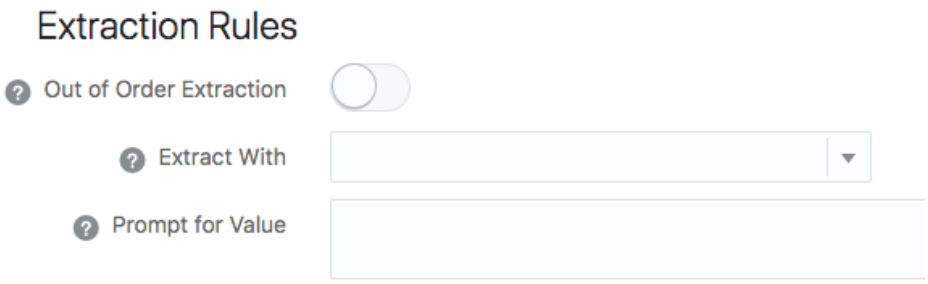
Composite Bag

Bag Items

+ Bag Item

Name	Type	Entity Name
overallRating	ENTITY	OverallCourseRating
courseMaterial	ENTITY	CourseMaterialRating
courseSpeed	ENTITY	CourseSpeedRating
handsOnMaterial	ENTITY	HandsOnQualityRating
instructor	ENTITY	InstructorRating
studentQuestions	ENTITY	StudentQuestionsAnswerRating
learningGoals	ENTITY	LearningGoalsRating
courseImprovements	STRING	

An important setting for each bag item is to disable out-of-order extraction:



Out-of-order extraction, when enabled, allows users to provide values for bag items that they are not prompted for, leading into prompts to be skipped. For the questionnaire, you want to force the exact sequence of bag items to be prompted for, thus you need to disable the out-of-order-extraction for each feature. Also, at least in this sample, many

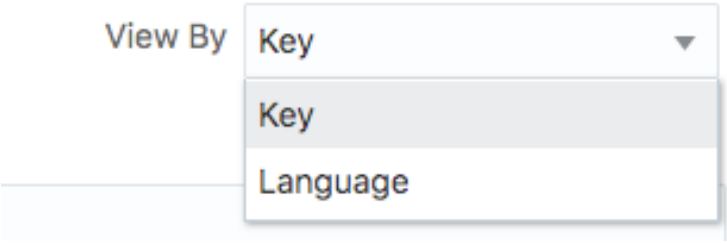
questions share the same answers "Good", "Poor", "Fair" etc. . Using out-of-order extraction would set those values on multiple questions at once, which is not what you want.

Finally, ensure that all bag items read their prompts from a message bundle. In the image below, a message bundle is used to read the prompt for a bag item (you can also decide to configure multiple prompts), as well as the index counter for the question.

Message Bundles

You use message bundles to prepare your skill's messages and prompts to display in multiple languages, but also to have all strings stored in a single place. The latter is convenient for maintenance of your strings. Imagine that after you built your bot, a decision was made to change the tone of it. In this case, using message bundles is so much easier to go over than to re-read the whole project. You access Message Bundles from the following icon: 

If you created message bundles, then you can view the strings by key name or language. To view strings by language makes it easier to translate message strings to a foreign language.



The languages you create are identified by the language's two character country code (de = German, fr = French, es = Spanish etc.). When creating keys in the message bundle, make sure the key are unique and bear context information. Key names like *promptQuestion1*, *promptQuestion2*, *promptQuestion3* etc. are hard to translate as they don't carry information.

The image below shows some of the keys used in the sample.

cancelActionLabel

companyNameQuestion

confirmSubmitMessage

continueActionLabel

courseImprovementComments

courseInstructorNameQuestion

courseLocationQuestion

courseMaterialRating

Note: The sample, as mentioned, uses message bundles to translate values too. So you will find keys named "Good", "Bad", "Fair", "Poor" etc. In this case translation is clear as well as the key name is equivalent to the value. In this case you translate the key name as the value.

To make resource bundles work, you need to define a variable of type "resourcebundle". The variable name then is used in the expressions to access the message key (see the next section for examples)

```
context:
  variables:
    iResult: "nlpresult"
    rb: "resourcebundle"
```

The Dialog Flow

The idea of a model driven approach is to write less code. Still however you need a dialog flow to get the user-bot conversation started and, in this sample at least, to add confirmation dialogs and states that handle help and the unresolved intent.

The image below shows the variable declaration. You see three variables that have a composite bag entity name associated as their type. These variables are used in a System.CommonResponse component to print the

questionnaire for a section. The *dataObject* variable could be defined a type "map" too. However, given the weak typing of JavaScript variables, you can use "string" as well.

```
context:
  variables:
    iResult: "nlpresult"
    rb: "resourcebundle"

    #Composite bag entity references. Composite bag
    courseInformation: "CourseInformation_CBE"
    courseQuality: "CourseEvaluation_CBE"
    personalInformation: "StudentInformation_CBE"

    #data object that takes the combined questions a
    #or passing to custom component
    dataObject: "string"
```

The image below prints the course related questions as text prompts. All bag items of the "courseInformation" variables are prompted for one after the other.

Notice the **${system.entityToResolve.value.prompt}** expression. The expression points to the prompt attribute of the current resolved bag item. Generic code like this makes it possible to use a single dialog flow state to prompt for all questions. The **gobalActions** section of the Sytem.CommonResponse component display a cancel button for the user to discontinue the questionnaire.

```
#free text questions about the course
  startCourseQuestions:
    component: "System.CommonResponse"
    properties:
      processUserMessage: true
      keepTurn: false
      variable: "courseInformation"
      nlpResultVariable:
      metadata:
        responseItems:
        - type: "text"
          text: "${system.entityToResolve.value.prompt}"
        globalActions:
        - label: "${rb.cancelActionLabel}"
          type: "postback"
          payload:
            action: "cancel"
    transitions:
      next: "startCourseQualityQuestions"
      actions:
        cancel: "exit"
```

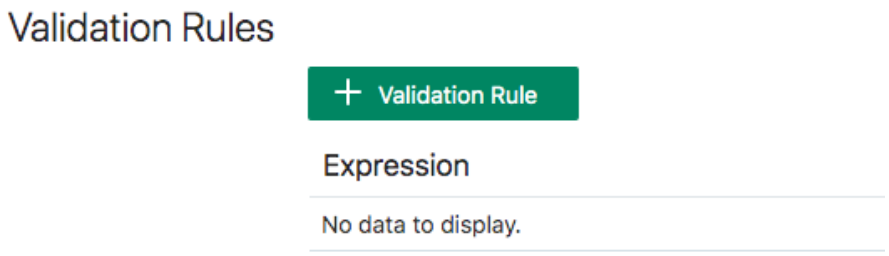
The image above navigates to the *startCouseQualityQuestoions* that displays a dialog informing the user for what she or he achieved so far and telling them what will come next. The next composite bag entity driven dialog is defined in the *courseQualityQuestions* state. For this state, notice the following

- Line 137 – referencing the *courseQuality* variable allow the System.CommonResponse component to resolve the bag items of the composite bag entity
- Line 144 – actions are stamped by the System.CommonResponse component to display a button for each possible answer to a question. The iterator for the answers is configured in line 147 and uses the following expression to read values from the bag item: ${system.entityToResolve.value.enumValues}.
- Line 148 – The keyword property on a common response component action allows you to define short cuts. In the sample, all answers have a leading alphabetical character as a short cut (a,b,c,d,…). Using the character displayed in front of an answer is a shortcut for selecting the answer. Notice, and you will need to look into the skill for the full expression, how the character is generated from the index position of the possible answer using Apache FreeMarker expressions.

```
132   courseQualityQuestions:
133     component: "System.CommonResponse"
134     properties:
135       processUserMessage: true
136       keepTurn: false
137       variable: "courseQuality"
138       transitionAfterMatch: "false"
139       nlpResultVariable:
140       metadata:
141         responseItems:
142         - type: "text"
143           text: "${system.entityToResolve.value.prompt}"
144           actions:
145           - label: "${((enumValue?index)+1)?lower_abc}. ${
146             type: "postback"
147             iteratorVariable: "system.entityToResolve.value
148             keyword: "${(enumValue?index)+1},${((enumValue
149             payload:
150               variables:
151                 courseQuality: "${enumValue}"
152         globalActions:
153         - label: "${rb.showMoreActionLabel}"
154           type: "postback"
155           visible:
156             expression: "${system.entityToResolve.value.nee
157           payload:
158             action: "system.showMore"
159             variables:
160               ${system.entityToResolve.value.rangeStartVar
161         - label: "${rb.cancelActionLabel}"
162           type: "postback"
163           payload:
164             action: "cancel"
165     transitions:
166       next: "startPersonalQuestions"
167       actions:
168         cancel: "exit"
```

The last dialog state that displays prompts from the composite bag entity is *personalQuestions*. This section uses free-text input. As mentioned though, still you could validate the user provided entries. All composite bag entity items

have a **Validation Rules** property that you can use to define custom validations using expressions. Alternatively, you can use the *PERSON* system entity to ensure that e.g. first name and last name are provided a valid names. The sample does not enforce validation on the name.



As you can see in the image below, there is nothing special about the *personalQuestion* dialog flow state.

```
198   personalQuestions:
199     component: "System.CommonResponse"
200     properties:
201       processUserMessage: true
202       keepTurn: false
203       variable: "personalInformation"
204       nlpResultVariable:
205       metadata:
206         responseItems:
207         - type: "text"
208           text: "${system.entityToResolve.val
209         globalActions:
210         - label: "${rb.cancelActionLabel}"
211           type: "postback"
212           payload:
213             action: "cancel"
214     transitions:
215       next: "createDataObject"
216       actions:
217         cancel: "exit"
```

Once the user answered all questions in the questionnaire, you want to read the relevant information into a single data object, which then could be sent to a custom component for further processing and to persist data using backend services.

The dialog flow state shown in the image below creates an array of objects. Each object has two properties: label and value. Notice how lines 243, 245 and others reference the resource bundle in their value section. As I mentioned, you can use defined answers as key names in a resource bundle. This way you can save the pre-defined answers selected by a user in the use language (assuming you translated the resource bundle). The answer summary and the data object would then contain content in the user language.

```
createDataObject:
  component: "System.SetVariable"
  properties:
    variable: "dataObject"
    value:
 #Course Information
    - label: "${rb.courseTitleQuestion}"
      value: "${courseInformation.value.courseTitle}"
    - label: "${rb.courseLocationQuestion}"
      value: "${courseInformation.value.location}"
    - label: "${rb.courseInstructorNameQuestion}"
      value: "${courseInformation.value.instructor}"
 #Quality Questions
    - label: "${rb.overallCourseRating}"
      value: "${rb(courseQuality.value.overallRating)}"
    - label: "${rb.instructorRating}"
      value: "${rb(courseQuality.value.instructor)}"
    - label: "${rb.courseMaterialRating}"
      value: "${rb(courseQuality.value.courseMaterial)}"
    - label: "${rb.studentQuestionsAnswerRating}"
      value: "${rb(courseQuality.value.studentQuestions)}"
    - label: "${rb.handsOnQualityRating}"
      value: "${rb(courseQuality.value.handsOnMaterial)}"
    - label: "${rb.courseSpeedRating}"
      value: "${rb(courseQuality.value.courseSpeed)}"
    - label: "${rb.learningGoalsRating}"
      value: "${rb(courseQuality.value.learningGoals)}"
    - label: "${rb.courseImprovementComments}"
      value: "${courseQuality.value.courseImprovements}"
  transitions:
    next: "showSummary"
```

The last dialog flow state of the sample is then to display a summary of questions and answers for the user to review and submit. Again the System.CommonResponse component is used to generate the overview. Notice **line 276** that uses a character **\u2022**, which is the utf8 code for a bullet point. **In line 277**, the data object array is referenced for the component to iterate over it and print a text message. The setting in **line 274** ensures that though messages are printed in their own stamp of the text response item, all messages appear in a single bubble.

```
showSummary:
  component: "System.CommonResponse"
  properties:
    processUserMessage: false
    metadata:
      responseItems:
      - type: "text"
        text: "${rb.thankYouAnsweringQuestionnaire}"
        footerText:
        iteratorVariable:
        separateBubbles: false
      - type: "text"
        text: "\u2022 ${dataObject.label} : ${dataObject.value}"
        iteratorVariable: "dataObject"
        separateBubbles: false
      - type: "text"
        text: "${rb('submitDataQuestion',rb.submitActionLabel,rb.cancelActionLabel)}"
        actions:
        - label: "a. ${rb.submitActionLabel}"
          type: "postback"
          keyword: "a,A,1"
          payload:
            action: "submit"
        - label: "b. ${rb.cancelActionLabel}"
          type: "postback"
          keyword: "b,B,2"
          payload:
            action: "cancel"
  transitions:
    actions:
      submit: "confirmSubmit"
      cancel: "exit"
```

An Alternative Data Object Structure

And there is one more option to construct the data object. In the structure show below, the response object is an object with each question to be saved as an attribute. Each attribute then represents an object "label" and "value". The structure below would make sense if you want to control the order in which a custom component works with the data passed on to it.

If you use this structure then my recommendation is to have the custom component displaying the information summary and displaying the submit/cancel button.

The image below shows how to construct this data object structure.

```
#Course Information
      courseTitle:
        label: "${rb.courseTitleQuestion}"
        value: "${courseInformation.value.courseTitle}"
      location:
        label: "${rb.courseLocationQuestion}"
        value: "${courseInformation.value.location}"
      instructorName:
        label: "${rb.courseInstructorNameQuestion}"
        value: "${courseInformation.value.instructor}"
#Quality Questions
      overallRating:
        label: "${rb.overallCourseRating}"
        value: "${rb(courseQuality.value.overallRating)}"
      instructorRating:
        label: "${rb.instructorRating}"
        value: "${rb(courseQuality.value.instructor)}"
      courseMaterial:
        label: "${rb.courseMaterialRating}"
        value: "${rb(courseQuality.value.courseMaterial)}"
      studentQuestions:
        label: "${rb.studentQuestionsAnswerRating}"
        value: "${rb(courseQuality.value.studentQuestions)}"
      handsonMaterial:
        label: "${rb.handsOnQualityRating}"
        value: "${rb(courseQuality.value.handsOnMaterial)}"
      courseSpeed:
        label: "${rb.courseSpeedRating}"
        value: "${rb(courseQuality.value.courseSpeed)}"
      learningGoals:
        label: "${rb.learningGoalsRating}"
        value: "${rb(courseQuality.value.learningGoals)}"
      courseImprovement:
        label: "${rb.courseImprovementComments}"
        value: "${courseQuality.value.courseImprovements}"
  transitions:
    next: "showSummary"
```