Search documentation

Appian

SAIL Recipes

Appian 7.6

The recipes on this page show how to achieve common user interface design patterns using SAIL. Some recipes are applicable to both dashboards and forms, while most are more relevant to forms such as recipes around form validation and collecting user input for submission. As such, the setup and expressions are catered towards forms, but the same concepts apply to dashboards. We'll show you how to trar what you learn to dashboards.

Toggle Us

To use these recipes, you must have a basic understanding of SAIL concepts, specifically how to enable user interaction and the difference between `load()` and `with()`.

See also: Enable User Interaction in SAIL, and SAIL Tutorial

The Interface Designer can be used to see most of the recipes in action. When you use the SAIL interface in process, you need to know how to configure a SAIL task form, how to save the user's input into a node input (aka ACP) from a rule, and subsequently save it into a process variable.

See also: Process Modeling with SAIL Tutorial and Interface Designer

Tu

The recipes can be worked on in no particular order. However, make sure to read the first two sections to get yourself set up.

Re

Ot

- Aggregate Data from a Data Store Entity and Display in a Chart
- Aggregate Data from a Data Store Entity using a Filter and Display in a Chart
- Aggregate Data from a Data Store Entity by Multiple Fields and Display in a Chart
- Aggregate Data from a Data Store Entity and Conditionally Display in a Chart or Grid
- Configure a Chart Drilldown to a Grid
- Filter the Data in a Grid Using a Chart

How to Work Through the Recipes

Most of the recipes on this page define their own local variables using the `load()` function. This means that you can interact with your dynamic form as soon as you paste the expression into the Interface Designer. When you want to adapt a recipe to your own use case, and save the data captured on the SAIL Form into process, you can do the following:

1. Create an interface input of the right type for each local variable whose value needs to be saved into the process.
2. In the expression, find the places that use the local variable, and update them to use the interface input instead.
3. Delete the corresponding local variables from your rule definition.
4. Create your variables in process:
    - If you're using a SAIL task form, create node inputs (activity class parameters), and save the node inputs into process variables if you wan to use their values elsewhere in the process.
    - If you're using a SAIL start form, create process variables and make them parameters.
5. In your process model, update the SAIL Form definition to pass the variables that you created to your new interface inputs. Remember to use keywords for easier change management. For instance:
    - If you're using a SAIL task form, it may look like this: `=rule!sailRecipe(ruleInputName: ac!nodeInputName)`
    - If you're using a SAIL start form, it may look like this: `=rule!sailRecipe(ruleInputName: pv!processParamName)`
6. Save and publish your process model.
7. Try it out in Tempo.
8. Always test each change that you make incrementally. In the process details view, go to the variables tab to confirm that the user's values are saved.

If you are testing multiple recipes, you can continue using the same Interface Designer tab. Make sure you click Test after pasting in a new expression t ensure that any local variables used get updated correctly.

How to Adapt a Recipe for Use on a Dashboard

The recipes are catered for use in process forms, but the same concepts generally apply when used on a dashboard. To use the recipe on a record or report dashboard, you can do the following:

1. Update the expression to use `a!dashboardLayout()` instead of `a!formLayout()`.
2. Remove the `label` parameter and the `buttons` parameter configurations.
    - There will be no error if you do not remove these configurations, because these parameters will be ignored by a!dashboardLayout(). However, Appian recommends that you remove them to avoid confusion and to keep your expression more maintainable.
    - If you see an error such as "Invalid lexer symbol found", you most likely forgot to remove a comma when you removed the `buttons` parameter configuration.

Set the Default Value of an Input on a Task Form

Goal: Display a default value in some form inputs on a task form, and save the value to process when submitting.

Steps

1. Create an interface with one interface input called title (Text).
2. Enter the following definition for the interface, and save it as sailRecipe.
3. In your process model, add a node input to your user input task called caseTitle (Text), and set its value to `="My default text"`. Save it into a process variable.
4. On the Forms tab, check the "Use an existing form" checkbox, click on "SAIL Form", and configure the SAIL Form as `=rule!sailRecipe(title: ac!caseTitle)`
5. Save and publish the process model.
6. Start a new process.

Expression

```
=a!formLayout(
 label: "SAIL Example: Default Value",
 firstColumnContents: {
  a!textField(
   label: "Case Title",
   required: true,
   value: ri!title,
   saveInto: ri!title
  )
 },
 buttons: a!buttonLayout(
  primaryButtons: a!buttonWidgetSubmit(label: "Submit")
 )
```

```
)
```

Test it out

1. On the task form, don't modify the value of the text field, and click the Submit button. Check the value of the process variable, and notice that it has the default value.
2. Now modify the value of the text field, and click the Submit button. Check the value of the process variable, and notice that it has the value you entered.
3. To see what happens when the default value is incorrectly configured, remove the default value from the node input. Then update the interface definition and set the `value` parameter of the textField component to `="Default text from the component"`. View the form and submit without modifying the text field. Notice that the corresponding process variable does not have the default value.

Watch out! A common mistake is to use load() and to configure a local variable called ri!title as follows:

```
=load(
  ri!title: "Default text",
  ...
)
```

The expression doesn't result in the intended behavior because `load()` only creates local variables. In the above example, a local variable called ri!title is created, with this default value, but its value is not saved in process. Load variables must only be used for data that are not saved back into the proces

Set the Default Value of an Input on a Start Form

Goal: Display a default value in some form inputs on a start form, and save the value into the process when submitting.

Steps

1. Create an interface with one interface input called title (Text).
2. Enter the following definition for the interface, and save it as sailRecipe.
3. In your process model, add a process variable called caseTitle (Text), select the Parameter checkbox, and set its value to `="My default text"`.
    - The process variable must be a parameter. Otherwise, the user's input will not be saved when you start the process.
4. On the Start Form tab, check the "Use an existing form" checkbox, click "SAIL Form", and configure the SAIL Form as: `=rule!sailRecipe(title: pv!caseTitle)`
5. Save and publish the process model.
6. To view the start form in Tempo, add the process model to an application and configure it as an action. Don't forget to publish your application.

Expression

```
=a!formLayout(
  label: "SAIL Example: Default Value",
  firstColumnContents: {
    a!textField(
      label: "Case Title",
      required: true,
      value: ri!title,
      saveInto: ri!title
    )
  },
  buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
)
```

Test it out

1. On the start form, don't modify the value of the text field, and click the Submit button. Check the value of the process variable, and notice that i has the default value.
2. Now modify the value of the text field, and click the Submit button. Check the value of the process variable, and notice that it has the value you entered.
    - If you do not see the value you entered, make sure that you selected the Parameter checkbox when you created the process variable.
3. To see what happens when the default value is incorrectly configured, remove the default value from the process variable definition. Then update the expression and set the `value` parameter of the text component to `="Default text from the component"`. View the form and submit without modifying the text field. Notice that the corresponding process variable does not have the default value.

Configure a Button that Skips Validation (a Cancel Button)

Goal: Display a button that submits the form even if the form contains validation errors such as a blank required field or an invalid text.

Expression

```
=load(
  local!a,
  local!b: "abc@",
  local!cancel: false,
```

```
  a!formLayout(
    label: "SAIL Example: Cancel Button",
    firstColumnContents: {
      a!textField(
        label: "Text 1",
        required: true,
        value: local!a,
        saveInto: local!a
      ),
      a!textField(
        label: "Text 2",
        instructions: "@ is an invalid character",
        validations: if(find("@", local!b)=0, null, "Invalid character!"),
        value: local!b,
        saveInto: local!b
      )
    },
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(
        label: "Submit",
        style: "PRIMARY"
      ),
      secondaryButtons: a!buttonWidgetSubmit(
        label: "Cancel",
        confirmMessage: "Are you sure?",
        skipValidation: true,
        value: true,
        saveInto: local!cancel
      )
    )
  )
)
```

By using the `secondaryButtons` configuration, we've added a Cancel button to the left side of the form. On mobile devices, secondary buttons show up below the primary buttons. We've also configured the Cancel button to display a confirmation message in case the user clicks it by accident. Finally, we styled the Submit button as `"PRIMARY"`.

To see that the form goes away even when there are validation errors, we will test this recipe in process.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: a (Text), b (Text), cancel (Boolean)
3. Remove the `load()` function
4. Delete local variables: `local!a`, `local!b`, `local!cancel`
5. In your expression, replace:
    - `local!a` with `ri!a`
    - `local!b` with `ri!b`
    - `local!cancel` with `ri!cancel`
6. In your process model, create variables: aaa (Text) with no value, bbb (Text) with value `"abc@"`, cancel (Boolean) with value `false`
7. Add a user input task with a node input for each process variable.
8. On the Forms tab of the user input task, check the "Use an existing form" checkbox, click on "SAIL Form", and configure the SAIL Form as `=rule!sailRecipe(a: ac!aaa, b: ac!bbb, cancel: ac!cancel)`
9. Save and publish the process model.
10. Start a new process.

Test it out

1. On the task form, click the "Submit" button without entering or modifying any of the field values. Notice that the form doesn't submit due to the presence of the validation messages.
2. Click the "Cancel" button without entering or modifying any of the field values. Notice that the form submits despite the invalid fields.
3. Enter a value in the required field and remove the "@" character from the second field. Now click the "Submit" button.

Display a Placeholder Text in a Dropdown

Goal: Display a placeholder text as the default option in a dropdown. If the dropdown is marked as required, the user must select a different option before proceeding.

Expression

```
=load(
  local!a: 10, /* null is not a valid value when there's no placeholder */
  local!b,
  local!c,
  a!formLayout(
    label: "SAIL Example: Dropdown with Placeholder Text",
```

```
  firstColumnContents: {
    a!dropdownField(
      label: "No Placeholder",
      required: true,
      choiceLabels: {"Fruits", "Vegetables"},
      choiceValues: {10, 20},
      value: local!a,
      saveInto: local!a
    ),
    a!dropdownField(
      label: "With Placeholder (Optional)",
      choiceLabels: {"Fruits", "Vegetables"},
      choiceValues: {10, 20},
      placeholderLabel: "--- Select One ---",
      value: local!b,
      saveInto: local!b
    ),
    a!dropdownField(
      label: "With Placeholder (Required)",
      required: true,
      choiceLabels: {"Fruits", "Vegetables"},
      choiceValues: {10, 20},
      placeholderLabel: "--- Select One ---",
      value: local!c,
      saveInto: local!c
    )
  },
  buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Click "Submit" without changing the selected values. Notice that the validation message only appears under the last dropdown field.
2. Select a value in the last dropdown field, and then click "Submit". Notice that there are no validation messages.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: a (Number Integer), b (Number Integer), c (Number Integer)
3. Remove the `load()` function
4. Delete local variables: `local!a` , `local!b` , `local!c`
5. In your expression, replace:
    - `local!a` with `ri!a`
    - `local!b` with `ri!b`
    - `local!c` with `ri!c`
6. In your process model, create variables: foodTypeA (Number Integer) with value `10` , foodTypeB (Number Integer) with with value `=null` , foodTypeC (Number Integer) with with value `=null`
    - On a task form, create node inputs
    - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
    - On a task form: `=rule!sailRecipe(a: ac!foodTypeA, b: ac!foodTypeB, c: ac!foodTypeC)`
    - On a start form: `=rule!sailRecipe(a: pv!foodTypeA, b: pv!foodTypeB, c: pv!foodTypeC)`

Notable implementation details

- Process variables of type Number (Integer) default to 0 rather than null, so we have to explicitly set them to null instead of leaving the value blan

Configure a Dropdown Field to Save a CDT

Goal: When using a dropdown to select values from the database, or generally from an array of CDT values, configure it to save the entire CDT value rather than just a single field.

Expression

```
=load(
 local!foodTypes: {
   {id: 1, name: "Fruits"},
   {id: 2, name: "Vegetables"}
 },
 local!selectedFoodType,
 a!formLayout(
   label: "SAIL Example: Dropdown with CDT",
   firstColumnContents: {
```

```
    a!dropdownField(
     label: "Food Type",
     instructions: "Value saved: " & local!selectedFoodType,
     choiceLabels: index(local!foodTypes, "name", null),
     /* choiceValues gets the CDT/dictionary rather than the ids */
     choiceValues: local!foodTypes,
     placeholderLabel: "--- Select Food Type ---",
     value: local!selectedFoodType,
     saveInto: local!selectedFoodType
    )
   },
   buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(label: "Submit")
   )
  )
)
```

Test it out

1. Select the choices in the dropdown and notice that the instructions are updated to reflect the value of the variable that is saved, in this case the entire CDT.

Notable implementation details

- Saving the entire CDT saves you from having to store the id and query the entire object separately when you need to display attributes of the selected CDT elsewhere on the form.
- When you configure your dropdown, replace the value of `local!foodTypes` with a CDT array that is the result of `a!queryEntity()` or `queryrecord()` These functions allow you to retrieve only the fields that you need to configure your dropdown.
  - See also: SAIL Best Practices
- This technique is well suited for selecting lookup values for nested CDTs. Let's say you have a project CDT and each project can have zero, one, or many team members. Team members reference the employee CDT. Use this technique when displaying a form to the end user for selecting team members.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: cdt (Any Type)
3. Delete local variable: `local!selectedFoodType`
4. In your expression, replace:
   - `local!selectedFoodType` with `ri!cdt`
   - The dictionary array (value of `local!foodTypes`) with a CDT array
5. In your process model, create a variable called cdt that is of the same type as the CDT array with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(cdt: ac!cdt)`
   - On a start form: `=rule!sailRecipe(cdt: pv!cdt)`

Configure a Dropdown with an Extra Option for Other

Goal: Show a dropdown that has an "Other" option at the end of the list of choices. If the user selects "Other", show a required text field.

We describe two expressions: expression 1 populates the dropdown list with parallel arrays of data while expression 2 populates the list with an array of CDT values.

Expression 1: This expression shows a dropdown whose option labels and values come from parallel arrays.

```
=load(
 local!choiceLabels: {"Fruits", "Vegetables", "Other"},
 local!choiceValues: {10, 20, -1},
 local!selectedFoodType: tointeger(null),
 local!other,
 a!formLayout(
  label: "SAIL Example: Dropdown from parallel arrays with Other option",
  firstColumnContents: {
   a!dropdownField(
    label: "Food Type",
    instructions: "Value saved: " & local!selectedFoodType,
    choiceLabels: local!choiceLabels,
    choiceValues: local!choiceValues,
    placeholderLabel: "--- Select Food Type ---",
    value: local!selectedFoodType,
    saveInto: local!selectedFoodType
   ),
   if(
    local!selectedFoodType = -1,
```

```
      a!textField(
        label: "Other",
        required: true,
        value: local!other,
        saveInto: local!other
      ),
      {}
    )
  },
  buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(
      label: "Submit",
      value: null,
      saveInto: if(
        or(isnull(local!selectedFoodType), local!selectedFoodType = -1),
        local!selectedFoodType, /* Clear value if user selected Other */
        local!other             /* Clear value if user selected an available option */
      )
    )
  )
 )
)
```

Expression 2: This expression shows a dropdown whose options come from a CDT array, to which we append an extra entry for the "Other" option.

```
=load(
 local!foodTypes: {
   {id: 1, name: "Fruits"},
   {id: 2, name: "Vegetables"}
 },
 local!choices: append(local!foodTypes, {id: -1, name: "Other"}),
 local!selectedFoodType,
 local!other,
 a!formLayout(
   label: "SAIL Example: Dropdown from CDT array with Other option",
   firstColumnContents: {
     a!dropdownField(
       label: "Food Type",
       instructions: "Value saved: " & local!selectedFoodType,
       choiceLabels: index(local!choices, "name", {}),
       choiceValues: local!choices,
       placeholderLabel: "--- Select Food Type ---",
       value: local!selectedFoodType,
       saveInto: local!selectedFoodType
     ),
     if(
       and(not(isnull(local!selectedFoodType)), tointeger(local!selectedFoodType.id) = -1),
       a!textField(
         label: "Other",
         required: true,
         value: local!other,
         saveInto: local!other
       ),
       {}
     )
   },
   buttons: a!buttonLayout(
     primaryButtons: a!buttonWidgetSubmit(
       label: "Submit",
       value: null,
       saveInto: if(
         or(isnull(local!selectedFoodType), tointeger(local!selectedFoodType.id) = -1),
         local!selectedFoodType, /* Clear value if user selected Other */
         local!other             /* Clear value if user selected an available option */
       )
     )
   )
 )
)
```

Test it out

1. Select "Other" in the dropdown, enter a value and click on the Submit button.
2. Select "Fruits" in the dropdown and click on the Submit button.
3. Select "Other" in the dropdown, don't enter any value, and click on the Submit button. Notice that the form can't be submitted unless the user

enters a value in the text field.
4. When the text field is blank, select "Fruits" from the dropdown to hide the text field. You can successfully submit even though the local!other variable is null.

To write your data to process

Expression 1

1. Save your interface as sailRecipe
2. Create interface inputs: selectedFoodType (Number Integer), other (Text)
3. Delete local variables: `local!selectedFoodType`, `local!other`
4. In your expression, replace:
   - `local!selectedFoodType` with `ri!selectedFoodType`
   - `local!other` with `ri!other`
5. In your process model, create variables: selectedFoodType (Number Integer) with no value, other (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(selectedFoodType: ac!selectedFoodType, other: ac!other)`
   - On a start form: `=rule!sailRecipe(selectedFoodType: pv!selectedFoodType, other: pv!other)`

Expression 2

1. Save your interface as sailRecipe
2. Create your CDT with at least 2 fields, one for the selected id, and one for the label to show in the dropdown.
3. Create interface inputs: cdt (Any Type), other (Text)
4. Delete local variables: `local!selectedFoodType`, `local!other`
5. In your expression, replace:
   - `local!selectedFoodType` with `ri!cdt`
   - `tointeger(local!selectedFoodType.id)` with `ri!cdt.id`
   - `local!other` with `ri!other`
   - The value of `local!foodTypes` with a CDT array
6. If the id field in your CDT is not an integer, also replace `-1` with a value of the same type as your id field.
7. In your process model, create a variable called `cdt` that is of the same type as the CDT array and a variable called `other` of type Text
   - On a task form, create node inputs
   - On a start form, create process parameters
8. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(cdt: ac!cdt, other: ac!other)`
   - On a start form: `=rule!sailRecipe(cdt: pv!cdt, other: pv!other)`

Notable implementation details

- Notice that we cleared out the opposite variable upon submission so that only one variable gets updated. That is, if the user filled out the "Other" field and then switched the dropdown back to an available option, `local!other` would be set to null on submission of the form.
- When you configure your dropdown, replace the value of local!foodTypes with `a!queryEntity()` or `queryrecord()` to return your array of options.

Configure Cascading Dropdowns

Goal: Show different dropdown options depending on the user selection.

Expression

```
=load(
 /* Initialize local!make to a null integer because local variables have no type
  * and the comparison local!make=1 would fail.
  * When saving into a process variable or a node input, you don't need to initialize
  * it in this way because the variable will have the correct type. */
 local!make: tointeger(null),
 local!model,
 a!formLayout(
  label: "SAIL Example: Cascading Dropdowns",
  firstColumnContents: {
   a!dropdownField(
    label: "Make",
    choiceLabels: {"Subaru", "Toyota"},
    choiceValues: {1, 2},
    placeholderLabel: "--- Select Make ---",
    value: local!make,
    saveInto: {
     local!make,
     a!save(local!model, null)
    }
   ),
   with(
    local!cascadingChoices: if(local!make=1,
     {{id: 1,  label: "Forester"}, {id: 2,  label: "Legacy"}},
```

```
        {{id: 10, label: "Camry"   }, {id: 20, label: "Yaris" }}
       ),
       a!dropdownField(
        label: "Model",
        disabled: isnull(local!make),
        choiceLabels: local!cascadingChoices.label,
        choiceValues: local!cascadingChoices.id,
        placeholderLabel: "--- Select Model ---",
        value: local!model,
        saveInto: local!model
       )
      )
    },
    buttons: a!buttonLayout(
     primaryButtons: a!buttonWidgetSubmit(label: "Submit")
    )
   )
)
```

Test it out

1. Select "Subaru" in the first dropdown. Notice that the second dropdown is now enabled and shows the Subaru models. Select a model.
2. Next, change the first dropdown to "Toyota" and notice that the second dropdown now shows the placeholder label so that the user can select model applicable to "Toyota".

Notable implementation details

The value of the second dropdown is reset to null when the first dropdown's value changes to ensure that the value of the selected model always matches what the user sees in the UI.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: make (Number Integer), model (Number Integer)
3. Remove the `load()` function
4. Delete local variables: `local!make` , `local!model`
5. In your expression, replace:
   - `local!make` with `ri!make`
   - `local!model` with `ri!model`
6. In your process model, create variables: make (Number Integer), model (Number Integer)
   - On a task form, create node inputs
   - On a start form, create process parameters with value `=null`
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(make: ac!make, model: ac!model)`
   - On a start form: `=rule!sailRecipe(make: pv!make, model: pv!model)`

Configure a Boolean Checkbox

Goal: Configure a checkbox that saves a boolean (true/false) value, and validate that the user selects the checkbox before submitting a form.

Expression

```
=load(
 local!userAgreed,
 a!formLayout(
  label: "SAIL Example: Required Boolean Checkbox",
  firstColumnContents: {
   a!checkboxField(
    label: "Acknowledge",
    required: true,
    requiredMessage: "You must check this box!",
    choiceLabels: {"I agree to the terms and conditions."},
    choiceValues: {true},
    value: local!userAgreed,
    saveInto: local!userAgreed
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label:"Submit")
  )
 )
)
```

Test it out

1. Click "Submit" without selecting the checkbox. Notice that the custom message shows up. Appian recommends using a custom required messag

when you have a single required checkbox.
2. Select the checkbox and click "Submit".

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: userAgreed (Boolean)
3. Remove the `load()` function
4. Delete local variable: `local!userAgreed`
5. In your expression, replace:
   - `local!userAgreed` with `ri!userAgreed`
6. In your process model, create variable: userAgreed (Boolean) with no value
   - On a task form, create node input
   - On a start form, create process parameter
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(userAgreed: ac!userAgreed)`
   - On a start form: `=rule!sailRecipe(userAgreed: pv!userAgreed)`

Make a Component Required Based on a User Selection

Goal: Make a paragraph component conditionally required based on the user selection.

Expression

```
=load(
 local!isMinor,
 local!comments,
 a!formLayout(
  label: "SAIL Example: Conditionally Required Field",
  firstColumnContents: {
   a!checkboxField(
    label: "Is Minor",
    instructions: "Value saved: " & local!isMinor,
    choiceLabels: "Check the box if the patient is a minor",
    choiceValues: {true},
    value: local!isMinor,
    saveInto: local!isMinor
   ),
   a!paragraphField(
    label: "Comments",
    required: local!isMinor,
    value: local!comments,
    saveInto: local!comments
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Select the *Is Minor* checkbox. Notice that the *Comments* field is required. If the checkbox is selected but no comments are entered, the user cannot submit the form.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: isMinor (Boolean), comments (Text)
3. Remove the `load()` function
4. Delete local variables: `local!isMinor`, `local!comments`
5. In your expression, replace:
   - `local!isMinor` with `ri!isMinor`
   - `local!comments` with `ri!comments`
6. In your process model, create variables: isMinor (Boolean) with no value, comments (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(isMinor: ac!isMinor, comments: ac!comments)`
   - On a start form: `=rule!sailRecipe(isMinor: pv!isMinor, comments: pv!comments)`

Set the Default Value Based on a User Input

Goal: Set the default value of a variable based on what the user enters in another component.

Note: This example only applies when the default value is based on the user's input in another component. See Set the Default Value of an Input on a Task Form recipe when the default value must be set as soon as the form is displayed and without requiring the user to interact with the form.

Expression

```
=load(
 local!username,
 local!email,
 local!emailModified: false,
 a!formLayout(
  label: "SAIL Example: Default Value Based on User Input",
  firstColumnContents: {
   a!textField(
    label: "Username",
    instructions: "Value saved: " & local!username,
    refreshAfter: "KEYPRESS",
    value: local!username,
    saveInto: {
     local!username,
     if(local!emailModified, {}, a!save(local!email, append(save!value, "@example.com")))
    }
   ),
   a!textField(
    label: "Email",
    instructions: "Value saved: " & local!email,
    value: local!email,
    saveInto: {
     local!email,
     a!save(local!emailModified, true)
    }
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Type into the *Username* field and notice that the *Email* field is pre-populated.
2. Type into the *Username* field, then modify the *Email* value, and type into the *Username* field again. The *Email* field is no longer pre-populated.

Notice that the value of username as well as the email address field are updated as you type. That's because the username input is configured with `refreshAfter: "KEYPRESS"`

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: username (Text), email (Text)
3. Delete local variables: `local!username` , `local!email`
4. In your expression, replace:
   - `local!username` with `ri!username`
   - `local!email` with `ri!email`
5. In your process model, create variables: username (Text) with no value, email (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(username: ac!username, email: ac!email)`
   - On a start form: `=rule!sailRecipe(username: pv!username, email: pv!email)`

Set the Default Value of CDT Fields Based on a User Input

Goal: Set the value of a CDT field based on a user input.

Expression

```
=load(
 local!myCdt: type!LabelValue(),
 a!formLayout(
  label: "SAIL Example: Default Value Based on User Input",
  instructions: "local!myCdt: " & local!myCdt,
  firstColumnContents: {
   a!textField(
    label: "Label",
    instructions: "Value saved: " & local!myCdt.label,
```

```
      refreshAfter: "KEYPRESS",
      value: local!myCdt.label,
      saveInto: {
        local!myCdt.label,
        a!save(local!myCdt.value, append(save!value, "@example.com"))
      }
    ),
    a!textField(
      label: "Value",
      instructions: "Value saved: " & local!myCdt.value,
      refreshAfter: "KEYPRESS",
      value: local!myCdt.value,
      saveInto: local!myCdt.value
    )
  },
  buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Type into the first text field, and notice that the second text field is pre-populated. The instructions of the form show the value of the CDT variab

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: myCdt (Any Type)
3. Remove the `load()` function
4. Delete local variable: `local!myCdt`
5. In your expression, replace:
   - `local!myCdt` with `ri!myCdt`
6. In your process model, create variables: myCdt (LabelValue) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(myCdt: ac!myCdt)`
   - On a start form: `=rule!sailRecipe(myCdt: pv!myCdt)`

Format the User's Input

Goal: Format the user's input as a telephone number in the US and save the formatted value, not the user's input.

This expression uses the `text()` function to format the telephone number. You may choose to format using your own rule, so you would create the supporting rule first, and then create an interface with the main expression.

Expression

```
=load(
 local!telephone,
 a!formLayout(
  label: "SAIL Example: Format US Telephone Number",
  firstColumnContents: {
   a!textField(
    label: "Telephone",
    instructions: "Value saved: " & local!telephone,
    value: local!telephone,
    saveInto: a!save(local!telephone, text(save!value, "###-###-####;###-###-####"))
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Enter `1234567890` then click somewhere else on the form. Notice that the phone number is now formatted.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: telephone (Text)
3. Remove the `load()` function
4. Delete local variables: `local!telephone`

5. In your expression, replace:
   - `local!telephone` with `ri!telephone`
6. In your process model, create variables: telephone (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(telephone: ac!telephone)`
   - On a start form: `=rule!sailRecipe(telephone: ac!telephone)`

Gather Sensitive Data from a User and Encrypt It

Goal: Create a form that gathers sensitive information from a user and store it securely in an Encrypted Text variable.

Expression

```
load(
  local!firstName,
  local!lastName,
  local!ssn,
  local!ailment,
  a!formLayout(
    label: "New Patient Visit",
    firstColumnContents: {
      a!textField(
        label: "First Name",
        required: true,
        value: local!firstName,
        saveInto: local!firstName
      ),
      a!textField(
        label: "Last Name",
        required: true,
        value: local!lastName,
        saveInto: local!lastName
      ),
      a!encryptedTextField(
        label: "Social Security Number",
        required: true,
        instructions: "xxx-xx-xxxx",
        value: local!ssn,
        saveInto: local!ssn
      ),
      a!encryptedTextField(
        label: "Primary Ailment",
        value: local!ailment,
        saveInto: local!ailment
      )
    },
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(
        label: "Submit"
      )
    )
  )
)
```

Test it out

1. Enter values for the fields on the start form, and click Submit to see that there are no errors.
2. The values entered in the EncryptedTextField components are encrypted before being stored. To see the encrypted values, test this form in process.

Note

1. Values generated from an encryptedTextField component will cause an error if you try to display them in a textField component, but can be displayed in clear text using an encryptedTextField component, as in this example.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: firstName (Text), lastName (Text), ssn (Encrypted Text), and ailment (Encrypted Text)
3. Delete local variables: `local!firstName`, `local!lastName`, `local!ssn`, and `local!ailment`
4. In your expression, replace:
   - `local!firstName` with `ri!firstName`
   - `local!lastName` with `ri!lastName`
   - `local!ssn` with `ri!ssn`

    - local!ailment with ri!ailment
5. In your process model, create variables: firstName (Text), lastName (Text), ssn (Encrypted Text), and ailment (Encrypted Text), all with no value
    - On a task form, create node inputs
    - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
    - On a task form: =rule!sailRecipe(firstName: ac!firstName, lastName: ac!lastName, ssn: ac!ssn, ailment: ac!ailment)
    - On a start form: =rule!sailRecipe(firstName: pv!firstName, lastName: pv!lastName, ssn: pv!ssn, ailment: pv!ailment)
7. If you interrogate your process variables after starting a process, entering values, and submitting the form, you will see that the variable contains an encrypted value.

Add Custom Validation Rules

Goal: Enforce that the user enters no more than a certain number of characters in their text field, e.g. to match the size constraint on a database column.

Expression

```
=load(
 local!a,
 local!b,
 a!formLayout(
  label: "SAIL Example: Custom Validation Rules",
  firstColumnContents:{
   a!textField(
    label: "5 Characters",
    instructions: "Character count: " & len(local!a) & "/5",
    refreshAfter: "KEYPRESS",
    validations: if(len(local!a)<=5, null, "Enter no more than 5 characters"),
    value: local!a,
    saveInto: local!a
   ),
   a!textField(
    label: "Text",
    validations: "Any text you enter is invalid",
    value: local!b,
    saveInto: local!b
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Type more than 5 characters in the first text field to see the validation message. The form cannot be submitted while the validation message is displayed.
    - Notice that the second text component doesn't show the validation message until the user types into it. This is because the text value isn't considered invalid until it has a value. See also: Validating User Inputs

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: a (Text), b (Text)
3. Remove the load() function
4. Delete local variables: local!a , local!b

5. In your expression, replace:
   - `local!a` with `ri!a`
   - `local!b` with `ri!b`
6. In your process model, create variables: aaa (Text) with no value, bbb (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(a: ac!aaa, b: ac!bbb)`
   - On a start form: `=rule!sailRecipe(a: pv!aaa, b: pv!bbb)`

Add Multiple Validation Rules to One Component

Goal: Enforce that the user enters at least a certain number of characters in their text field, and also enforce that it contains the "@" character.

Expression

```
=load(
 local!a,
 a!formLayout(
  label: "SAIL Example: Multiple Validation Rules on One Component",
  firstColumnContents:{
   a!textField(
    label: "Text",
    instructions: "Enter at least 5 characters, and include the @ character",
    validations: {
     if(len(local!a)>5, null, "Enter at least 5 characters"),
     if(isnull(local!a), null, if(find("@", local!a)<>0, null, "You need an @ character!"))
    },
    value: local!a,
    saveInto: local!a
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Type fewer than 5 characters and click "Submit".
2. Type more than 5 characters but no "@" and click "Submit".
3. Type more than 5 characters and include "@" and click "Submit".

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: a (Text)
3. Remove the `load()` function
4. Delete local variable: `local!a`
5. In your expression, replace:
   - `local!a` with `ri!a`
6. In your process model, create variable: aaa (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(a: ac!aaa)`
   - On a start form: `=rule!sailRecipe(a: pv!aaa)`

Define a Simple Currency Component

Goal: Show a text field that allows the user to enter dollar amounts including the dollar symbol and thousand separators, but save the value as a decim rather than text. Additionally, always show the dollar amount with the dollar symbol.

Expression

```
=load(
 local!amount,
 a!formLayout(
  label: "SAIL Example: Simple Currency Component",
  firstColumnContents:{
   a!textField(
    label:"Amount in Text",
    instructions: "Type of local!amount: " & typename(typeof(local!amount)),
    value: if(isnull(local!amount), "", dollar(local!amount)),
```

```
      saveInto: a!save(local!amount, todecimal(save!value))
    ),
    if(isnull(local!amount), {},
     a!textField(
       label: "Divided by 10",
       value: local!amount/10,
       readOnly: true
     )
    )
   },
   buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(label: "Submit")
   )
  )
)
```

Test it out

1. Enter `$12345` and click away from the field. Notice that the text box shows $12,345.00 and that the saved value is a decimal.
2. Enter `$12,345.23` and click away from the field.
3. Enter `a1b2c3` and click away. Notice that the text box removes the non-numeric characters and treats the remaining as a decimal value. A true currency component would catch this as an error case, hence why this is called a simple currency example.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: amount (Number Decimal)
3. Remove the `load()` function
4. Delete local variables: `local!amount`
5. In your expression, replace:
   - `local!amount` with `ri!amount`
6. In your process model, create variables: amount (Number Decimal) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(amount: ac!amount)`
   - On a start form: `=rule!sailRecipe(amount: pv!amount)`

Note: If you want to save this example as a reusable component, see also: Creating Reusable Custom Components.

Add Multiple Text Components Dynamically

Goal: Show a dynamic number of text components to simulate a multi-text input box. A new text box is shown as soon as the user starts typing into the last input box.

The main expression uses two supporting rules, so let's create them first.

- `ucDynamicFieldsUpdateArray` : Updates the guest array at the specified index, and appends a null item at the end of the array. If the last item in th array is already null, no new item is added.
- `ucDynamicFieldEach` : Returns a text field populated with the guest value at the given index.

Create expression rule `ucDynamicFieldsUpdateArray` with the following rule inputs:

- index (Number Integer)
- guests (Text Array)
- newValue (Text)

Enter the following definition for the rule:

```
=with(
 local!newGuestList: updatearray(ri!guests, ri!index, ri!newValue),
 if(isnull(local!newGuestList[count(local!newGuestList)]),
  local!newGuestList,
  append(local!newGuestList, "")
 )
)
```

Create expression rule `ucDynamicFieldEach` with the following rule inputs:

- index (Number Integer)
- guests (Text Array)

Enter the following definition for the rule:

```
=a!textField(
 label: if(ri!index=1, "Guest Names", ""),
 refreshAfter: "KEYPRESS",
 value: ri!guests[ri!index],
```

```
  saveInto: a!save(ri!guests, rule!ucDynamicFieldsUpdateArray(ri!index, ri!guests, save!value))
)
```

Now that we've created the two supporting rules, let's move on to the main expression.

Expression

```
=load(
  local!guests: {""},
  a!formLayout(
    label:"SAIL Example: Add Text Components Dynamically",
    firstColumnContents: {
      /* The guests array is passed to the rule directly, creating a partial function */
      a!applyComponents(
        function: rule!ucDynamicFieldEach(index: _, guests: local!guests),
        array: 1+enumerate(count(local!guests))
      )
    },
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(label: "Submit")
    )
  )
)
```

Test it out

1. Type into the text field and notice that an empty one is appended.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: guests (Text Array)
3. Remove the `load()` function
4. Delete local variable: `local!guests`
5. In your expression, replace:
    - `local!guests` with `ri!guests`
6. In your process model, create variables: guests (Text Array) with value `""`
    - On a task form, create node inputs
    - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
    - On a task form: `=rule!sailRecipe(guests: ac!guests)`
    - On a start form: `=rule!sailRecipe(guests: pv!guests)`

Add Multiple File Upload Components Dynamically

Goal: Show a dynamic number of file upload components. After a file is uploaded, add a new file upload component for another file to be uploaded shou it be necessary. If an uploaded file is removed, remove its component so the set of components always has exactly one empty spot.

The main expression uses two supporting rules, so let's create them first.

- `ucMultiFileUploadResizeArray` : Updates an array with a value at an index, growing the array if the index is past the end and shrinking the array if the value is null.
- `ucMultiFileUploadRenderField` : Returns a file upload component populated with the value at the given index.

Create expression rule `ucMultiFileUploadResizeArray` with the following rule inputs:

- array (Document Array)
- index (Integer)
- value (Document)

Enter the following definition for the rule:

```
=if(                          /* Do we need to add?                          */
  and(
    not(isnull(ri!value)),    /* Yes, if the value isn't null                */
    ri!index = 1 + length(ri!array) /* and we're at the end of the list       */
  ),
  append(
    ri!array,
    ri!value                  /* Add: Append the new value to the end of the list */
  ),
  if(                         /* Do we need to remove?                       */
    isnull(ri!value),         /* Only if value is now null (value was removed) */
    remove(ri!array, ri!index), /* Remove: remove the list item for this field */
    insert(                   /* Neither add nor remove, so insert at index  */
      ri!array,
      ri!value,
```

```
    ri!index
   )
  )
)
```

Create expression rule `ucMultiFileUploadRenderField` with the following rule inputs:

- label (Text)
- files (Document Array)
- target (Document or Folder)
- index (Integer)

Enter the following definition for the rule:

```
=with(
 local!paddedArray: append(ri!files, null),
 a!fileUploadField(
  label: if(ri!index = 1, ri!label, ""),
  target: ri!target,
  value: local!paddedArray[ri!index],
  saveInto: {
   a!save(ri!files, ucMultiFileUploadResizeArray(ri!files, ri!index, save!value))
  }
 )
)
```

Now that we've created the two supporting rules, let's move on to the main expression.

Expression:

```
=load(
 local!files: {  },
 local!target: tofolder(-1), /* Must be updated to a real folder to actually persist the files! */
 a!formLayout(
  label: "SAIL Example: Multiple File Upload",
  firstColumnContents: {
   a!applyComponents(
    function: rule!ucMultiFileUploadRenderField(label: "Upload Files", files: local!files, target: local!target, index: _),
    array: 1 + fn!enumerate(1 + length(local!files))
   ),
   a!textField(label: "local!files value", readOnly: true, value: local!files)
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out:

1. Upload a file and notice that an empty one is appended.
2. Upload several more files, then try removing one from the beginning or middle of the list. Notice the empty component is removed.

To write your data to process:

1. Save your interface as sailRecipe
2. Create or choose a destination folder for the uploaded files
3. Create interface input: files (Document Array), target (Document or Folder)
4. Remove the `load()` function
5. Delete local variable: `local!files`, `local!target`
6. In your expression, replace:
   - `local!files` with `ri!files`
   - `tofolder(-1)` with `ri!target`
7. In your process model, create variables: files (Document Array) with no value, target (Document or Folder) with the value of a system folder.
   - On a task form, create node inputs
   - On a start form, create process parameters
8. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(files: ac!files, target: ac!target)`
   - On a start form: `=rule!sailRecipe(files: pv!files, target: pv!target)`

Notable implementation details

- The process variable `pv!target` can be used in one of two ways. You can set it to a constant of type Document or Folder that points to a system folder, if you want to hardcode the folder in your process. If you're also creating the folder in process, you can pass that folder into `pv!target` as well.

Add and Populate Sections Dynamically

Goal: Add and populate a dynamic number of sections, one for each item in a CDT array. Each section contains an input for each field of the CDT. A ne entry is added to the CDT array as the user is editing the last section to allow the user to quickly add new entries without extra clicks. Sections can be independently removed by clicking on a "Remove" button. In the example below, attempting to remove the last section simply blanks out the inputs. Your own use case may involve removing the last section altogether.

The main expression uses a few supporting rules, so let's create them first.

- `ucDynamicSectionAddOne` : Takes an array of records, and adds a null record of the same type if the given index is for the last item in the array.
- `ucDynamicSectionRemoveFromArray` : Removes the item at the given index from the records array. If removing the last item in the array, replaces the last item with a null record of the same type.
- `ucDynamicSectionEach` : Returns a section with its components populated with the value of the record at the specified index.

Create expression rule `ucDynamicSectionAddOne` with the following rule inputs:

- array (Any Type)
- index (Number Integer)

Enter the following definition for the rule:

```
=if(ri!index <> count(ri!array),
 ri!array,
 append(ri!array, cast(typeof(ri!array), null))
)
```

Create expression rule `ucDynamicSectionRemoveFromArray` with the following rule inputs:

- index (Number Integer)
- array (Any Type)

Enter the following definition for the rule:

```
=if(count(ri!array)=1,
 {cast(typeof(ri!array), null)},
 remove(ri!array, ri!index)
)
```

Create interface `ucDynamicSectionEach` with the following interface inputs:

- index (Number Integer)
- records (Any Type)
- recordTokens (Any Type)

Enter the following definition for the interface:

```
=a!sectionLayout(
 label: "Section " & ri!index,
 firstColumnContents:{
  a!textField(
   label: "Label",
   refreshAfter: "KEYPRESS",
   value: ri!records[ri!index].label,
   saveInto: {
    ri!records[ri!index].label,
    a!save(ri!records, rule!ucDynamicSectionAddOne(ri!records, ri!index)),
    a!save(ri!recordTokens, rule!ucDynamicSectionAddOne(ri!recordTokens, ri!index))
   }
  ),
  a!textField(
   label: "Value",
   refreshAfter: "KEYPRESS",
   value: ri!records[ri!index].value,
   saveInto: ri!records[ri!index].value
  ),
  if(
   count(ri!records) > 1,
   a!buttonArrayLayout(
    a!buttonWidget(
     label: "Remove",
     value: ri!index,
     saveInto: {
      a!save(ri!records, rule!ucDynamicSectionRemoveFromArray(save!value, ri!records)),
      a!save(ri!recordTokens, rule!ucDynamicSectionRemoveFromArray(save!value,  ri!recordTokens))
     }
    )
   ),
```

```
    {}
   )
  }
)
```

Now that we've created the supporting rules, let's move on to the main expression.

Expression

```
=load(
 local!records: {type!LabelValue()},
 local!recordTokens,
 a!formLayout(
  label: "SAIL Example: Add Sections Dynamically",
  instructions: "local!records: " & local!records,
  firstColumnContents: {
   a!applyComponents(
    function: rule!ucDynamicSectionEach(index: _, records: local!records, recordTokens: local!recordTokens),
    array: 1+enumerate(count(local!records)),
    arrayVariable: local!recordTokens
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
```

Test it out

1. Fill in the first field and notice that a new section is added as you're typing.
2. Add a few sections and click on the Remove button to remove items from the array.

Notable implementation details

- `local!recordTokens` must be declared as a `load()` local variable as shown above for adding and removing of sections to work correctly. The local variable is then passed to the looping function `a!applyComponents` as its third parameter. a!applyComponents will create an array in this variable that is the same length as the record array. Changes to the record array such as adding, removing, or swapping must also be made to the recordTokens array.
- When dynamically adding and generating SAIL components in this way, always use the `a!xxxComponents()` looping functions. See also: New Looping Functions for Components

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: records (Any Type)
3. Delete local variable: `local!records`
4. In your expression, replace:
   - `local!records` with `ri!records`
5. In your process model, create variables: records (LabelValue array) as an array with a single null value, such as `{type!LabelValue()}`
   - On a task form, create node inputs
   - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(records: ac!records)`
   - On a start form: `=rule!sailRecipe(records: pv!records)`

Display an Array of Images Stored in Document Management

Goal: Display a set of images stored in Appian's document management system.

First, upload a few images into Appian. Then, create a constant of type Document and supporting rule.

- `UC_IMAGE_DOCS` : An array of images in Document Management.
- `ucDocumentImageEach` : Returns a DocumentImage for use by the image field component. The document name is used as the caption.

Create constant `UC_IMAGE_DOCS` of type Document and select Multiple. Select the images that you uploaded as the value.

Create expression rule `ucDocumentImageEach` with the following rule input:

- document (Document)

Enter the following definition for the rule:

```
=a!documentImage(
 document: ri!document,
 caption: document(ri!document, "name")
)
```

Now that we've created the supporting constant and rule, let's move on to the main expression.

Expression

```
=a!imageField(
 label: "Images",
 size: "THUMBNAIL",
 images: apply(rule!ucDocumentImageEach, cons!UC_IMAGE_DOCS)
)
```

Test it out 1. Click on an image and navigate the array using the mouse or cursor keys.

Display Images in a Grid

Goal: Display a set of images in a read-only paging grid. The images are stored in Appian's document management system.

First, upload a few images into Appian. Then, create a constant of type document named `UC_IMAGE_DOCS` and select the Multiple checkbox. Select the images that you uploaded as the value.

Now that we've created the supporting constant, let's move on to the main expression.

```
=a!gridField(
 label: "SAIL Example: Grid with Images",
 columns: {
  a!gridTextColumn(
   label: "Document Name",
   data: apply(fn!document, cons!UC_IMAGE_DOCS, "name")
  ),
  a!gridImageColumn(
   label: "Image Thumbnail Column",
   size: "THUMBNAIL",
   data: apply(a!documentImage(document: _), cons!UC_IMAGE_DOCS)
  )
 },
 value: a!pagingInfo(
  startIndex: 1,
  batchSize: count(cons!UC_IMAGE_DOCS)
 ),
 totalCount: count(cons!UC_IMAGE_DOCS)
)
```

Notable implementation details

- Because we did not configure any paging or sorting, clicking on the column headers will not sort the columns.
  - See also: Filter the Data in a Grid
- If your images are appropriate to show at a 20 x 20px size you can use `size: "ICON"` instead of `size: "THUMBNAIL"`.

Add a Custom Required Message

Goal: Instead of the product message that shows up when a required field has no value, show a custom message.

Expression

```
=load(
 local!a,
 local!b,
 a!formLayout(
  label: "SAIL Example: Custom Required Message",
  firstColumnContents:{
   a!textField(
    label: "Custom Message",
    required: true,
    requiredMessage: "You must enter a value!",
    value: local!a,
    saveInto: local!a
   ),
   a!textField(
    label: "Product Message",
    required: true,
    value: local!b,
    saveInto: local!b
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
```

```
  )
 )
)
```

Test it out

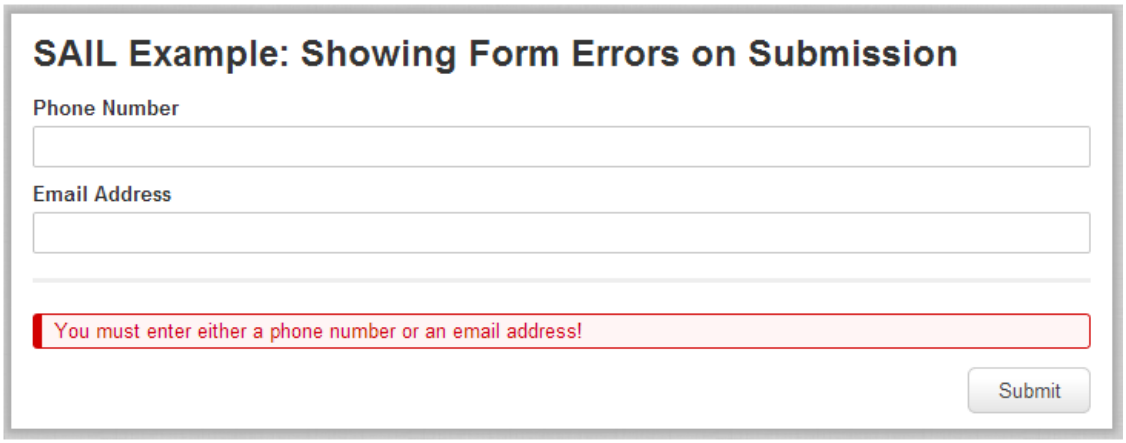
1. Leave both text fields blank and click "Submit".

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: a (Text), b (Text)
3. Remove the `load()` function
4. Delete local variables: `local!a` , `local!b`
5. In your expression, replace:
   - `local!a` with `ri!a`
   - `local!b` with `ri!b`
6. In your process model, create variables: aaa (Text) with no value, bbb (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(a: ac!aaa, b: ac!bbb)`
   - On a start form: `=rule!sailRecipe(a: pv!aaa, b: pv!bbb)`

Showing Validation Errors that Aren't Specific to One Component

Goal: Alert the user about form problems that aren't specific to one component, showing the message only when the user clicks "Submit". In this case, there are two fields and although neither are required, at least one of them must be filled out to submit the form.



Expression

```
=load(
 local!phone,
 local!email,
 a!formLayout(
  label: "SAIL Example: Showing Form Errors on Submission",
  firstColumnContents:{
   a!textField(
    label: "Phone Number",
    value: local!phone,
    saveInto: local!phone
   ),
   a!textField(
    label: "Email Address",
    value: local!email,
    saveInto: local!email
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  ),
  validations: {
   if(
    and(isnull(local!phone), isnull(local!email)),
    a!validationMessage(
     message: "You must enter either a phone number or an email address!",
     validateAfter: "SUBMIT"
    ),
    {}
```

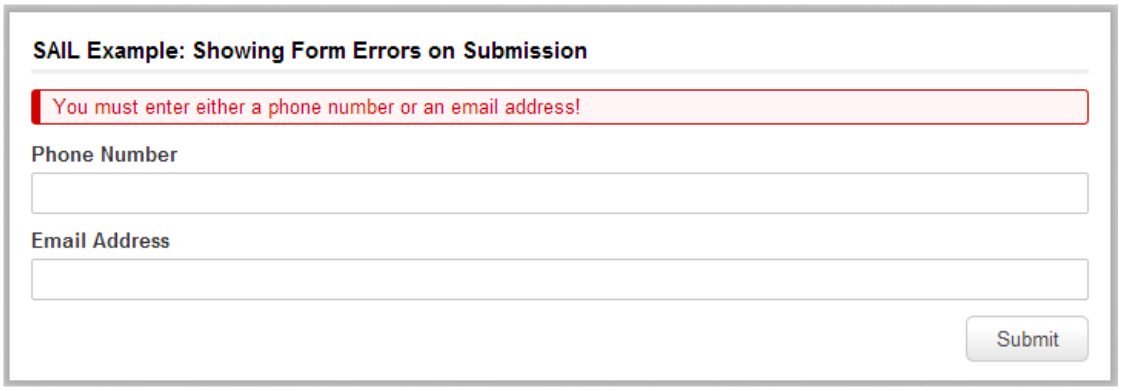
```
    )
   }
  )
)
```

Test it out

1. Leave both text fields blank and click "Submit".

Notable implementation details

- The system function a!validationMessage() allows us to specify whether the validation message is shown right away (`REFRESH`) or when the user submits the form (`SUBMIT`). If the validation message should always be shown right away, we could just pass the message to `a!formLayout()` 's `validations` parameter as Text. To show multiple messages, we can pass a list of Text, a list of `a!validationMessage()` , or a mix of the two.
- You can also configure `a!sectionLayout()` to show validation messages:



To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: phone (Text), email (Text)
3. Remove the `load()` function
4. Delete local variables: `local!phone` , `local!email`
5. In your expression, replace:
   - `local!phone` with `ri!phone`
   - `local!email` with `ri!email`
6. In your process model, create variables: phone (Text) with no value, email (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(phone: ac!phone, email: ac!email)`
   - On a start form: `=rule!sailRecipe(phone: pv!phone, email: pv!email)`

Approve/Reject Buttons with Conditional Requiredness

Goal: Present two buttons to the end user called "Approve" and "Reject" and only make the comments field required if the user clicks "Reject".

Note: `validationGroup` can have any string that you define. See also: Using Validation Groups

Expression

```
=load(
 local!comments,
 local!hasApproved,
 a!formLayout(
  label: "SAIL Example: Approve Reject Buttons with Conditional Requiredness",
  firstColumnContents: {
   a!paragraphField(
    label: "Comments",
    instructions: "This also shows an example of a custom required message",
    required: true,
    requiredMessage: "You must enter comments when you reject",
    validationGroup: "reject",
    value: local!comments,
    saveInto: local!comments
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: {
    a!buttonWidgetSubmit(
     label: "Approve",
     value: true,
     saveInto: local!hasApproved
```

```
      ),
      a!buttonWidgetSubmit(
        label: "Reject",
        validationGroup: "reject",
        value: false,
        saveInto: local!hasApproved
      )
    }
  )
 )
)
```

Test it out

1. Click "Reject" without entering any comments. Notice that the custom required message that we configured using the `requiredMessage` parameter shows up rather than the generic product message.
2. Click Approve without entering any comments.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: comments (Text), hasApproved (Boolean)
3. Remove the `load()` function
4. Delete local variables: `local!comments` , `local!hasApproved`
5. In your expression, replace:
   - `local!comments` with `ri!comments`
   - `local!hasApproved` with `ri!hasApproved`
6. In your process model, create variables: comments (Text) with no value, hasApproved (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(comments: ac!comments, hasApproved: ac!hasApproved)`
   - On a start form: `=rule!sailRecipe(comments: ac!comments, hasApproved: ac!hasApproved)`

Approve/Reject Buttons with Multiple Validation Rules

Goal: Present two buttons to the end user called "Approve" and "Reject". Also display a comments field. The number of characters in the comments fi must not exceed 100 characters, regardless of the button clicked. Additionally, the user must enter comments if she clicks "Reject" (comments are required in this case).

Note: We recommend that you go through the Approve/Reject Buttons with Conditional Requiredness recipe before working on this one.

Expression

```
=load(
 local!comment,
 local!hasApproved,
 a!formLayout(
  label: "SAIL Example: Approve Reject Buttons with Multiple Validation Rules",
  firstColumnContents: {
   with(
    local!commentIsValid: if(len(local!comment)<=100, true, false),
    a!paragraphField(
     label: "Comments",
     instructions: "Comments must have no more than 100 characters, regardless of the button clicked. Comments are required if rejecting.",
     validations: if(local!commentIsValid, "", "Comments must have no more than 100 characters"),
     required: local!commentIsValid,
     requiredMessage: "You must enter comments when you reject",
     validationGroup: if(local!commentIsValid, "reject", ""),
     value: local!comment,
     saveInto: local!comment
    )
   )
  },
  buttons: a!buttonLayout(
   primaryButtons: {
    a!buttonWidgetSubmit(
     label: "Approve",
     value: true,
     saveInto: local!hasApproved
    ),
    a!buttonWidgetSubmit(
     label: "Reject",
     validationGroup: "reject",
     value: false,
     saveInto: local!hasApproved
```

```
    )
   }
  )
 )
)
```

Test it out

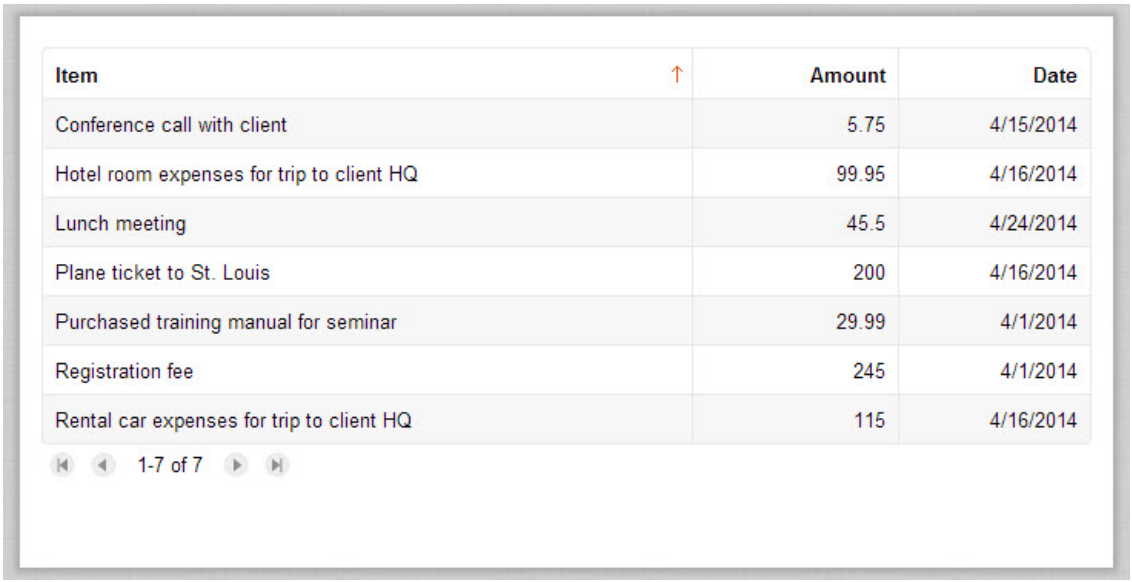
1. Click "Reject" without entering any comments. Notice that the custom required message that we configured using the `requiredMessage` parameter shows up rather than the generic product message.
2. Enter more than 100 characters as comments, and click "Reject". Click "Approve'. You shouldn't be able to submit in either case.
3. Click "Approve" without entering any comments.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface inputs: comments (Text), hasApproved (Boolean)
3. Remove the `load()` function
4. Delete local variables: `local!comments` , `local!hasApproved`
5. In your expression, replace:
   - `local!comments` with `ri!comments`
   - `local!hasApproved` with `ri!hasApproved`
6. In your process model, create variables: comments (Text) with no value, hasApproved (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(comments: ac!comments, hasApproved: ac!hasApproved)`
   - On a start form: `=rule!sailRecipe(comments: ac!comments, hasApproved: ac!hasApproved)`

Display Data from a Record in a Grid

Goal: Display data from a record type in a read-only paging grid.



Item	Amount	Date
Conference call with client	5.75	4/15/2014
Hotel room expenses for trip to client HQ	99.95	4/16/2014
Lunch meeting	45.5	4/24/2014
Plane ticket to St. Louis	200	4/16/2014
Purchased training manual for seminar	29.99	4/1/2014
Registration fee	245	4/1/2014
Rental car expenses for trip to client HQ	115	4/16/2014

1-7 of 7

This scenario demonstrates:

- How to use the report builder to generate a grid to display data from a record type.
- How to modify the generated expression to change the grid column alignment.

For this recipe, you'll need a record. Let's use the process-backed record from the Records Tutorial. If you haven't already created the Expense Report record type, do so now by completing the first five steps of the "Create Process-Backed Records" tutorial and starting at least one instance of the process, then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EXPENSE_REPORT_RECORD` with Record Type as the type and `Expense Report` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EXPENSE_REPORT_RECORD` constant.
4. In the *Add a field...* dropdown, select `expenseDate` and click **Add Field**.
5. Set the display name for each of the three columns as `Item` , `Amount` , and `Date` , respectively.
6. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "expenseItem",
```

```
      ascending: true
    )
  ),
  with(
    local!datasubset: queryrecord(
      cons!EXPENSE_REPORT_RECORD,
      a!query(
        selection: a!querySelection(columns: {
          a!queryColumn(field: "expenseItem"),
          a!queryColumn(field: "expenseAmount"),
          a!queryColumn(field: "expenseDate"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!gridField(
      totalCount: local!datasubset.totalCount,
      columns: {
        a!gridTextColumn(
          label: "Item",
          field: "expenseItem",
          data: index(local!datasubset.data, "expenseItem", null)
        ),
        a!gridTextColumn(
          label: "Amount",
          field: "expenseAmount",
          data: index(local!datasubset.data, "expenseAmount", null)
        ),
        a!gridTextColumn(
          label: "Date",
          field: "expenseDate",
          data: index(local!datasubset.data, "expenseDate", null)
        ),
      },
      value: local!pagingInfo,
      saveInto: local!pagingInfo
    )
  )
)
```

7. Right align the "Amount" and "Date" columns by modifying the expression as shown below:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 20,
    sort: a!sortInfo(
      field: "expenseItem",
      ascending: true
    )
  ),
  with(
    local!datasubset: queryrecord(
      cons!EXPENSE_REPORT_RECORD,
      a!query(
        selection: a!querySelection(columns: {
          a!queryColumn(field: "expenseItem"),
          a!queryColumn(field: "expenseAmount"),
          a!queryColumn(field: "expenseDate"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!gridField(
      totalCount: local!datasubset.totalCount,
      columns: {
        a!gridTextColumn(
          label: "Item",
          field: "expenseItem",
          data: index(local!datasubset.data, "expenseItem", null)
        ),
        a!gridTextColumn(
          label: "Amount",
          field: "expenseAmount",
          data: index(local!datasubset.data, "expenseAmount", null),
```

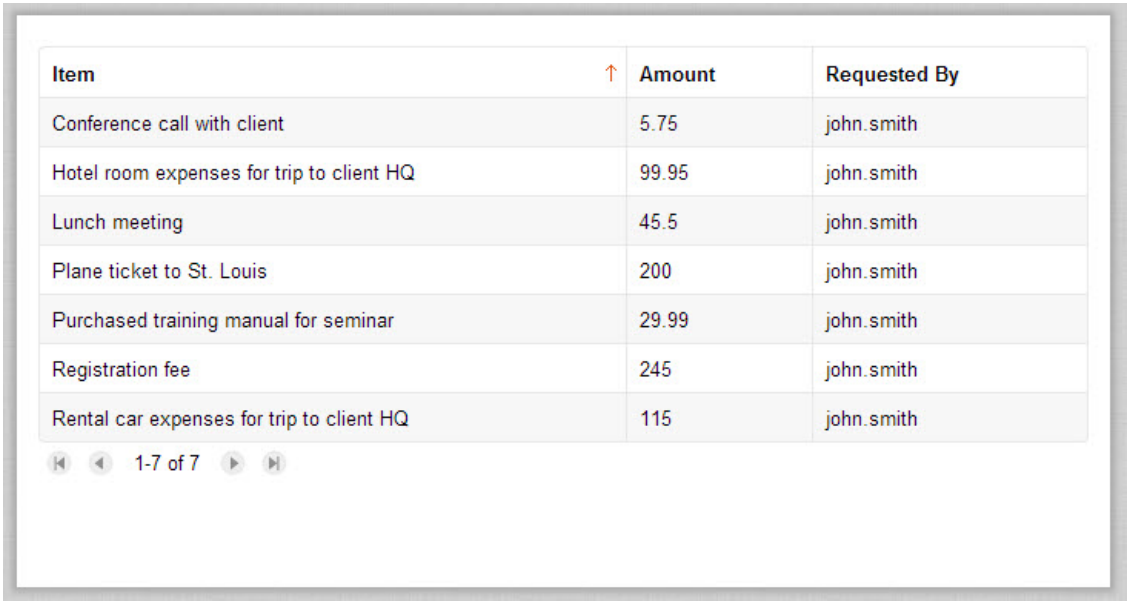
```
        alignment: "RIGHT"
      ),
      a!gridTextColumn(
        label: "Date",
        field: "expenseDate",
        data: index(local!datasubset.data, "expenseDate", null),
        alignment: "RIGHT"
      ),
    },
    value: local!pagingInfo,
    saveInto: local!pagingInfo
  )
 )
)
```

Notable implementation details

- The grid generated by the report builder is already configured to page and sort.
- The query that populates this grid will return all data for the record type in batches of 20. To filter the data returned by the query, see also: Filter Data from a Record in a Grid

Display Data with CDT Fields from a Record in a Grid

Goal: Display data that contains CDT fields from a record type in a read-only paging grid.



Item ↑	Amount	Requested By
Conference call with client	5.75	john.smith
Hotel room expenses for trip to client HQ	99.95	john.smith
Lunch meeting	45.5	john.smith
Plane ticket to St. Louis	200	john.smith
Purchased training manual for seminar	29.99	john.smith
Registration fee	245	john.smith
Rental car expenses for trip to client HQ	115	john.smith

1-7 of 7

This scenario demonstrates:

- How to use the report builder to generate a grid to display data from a record type.
- How to modify the generated expression to show data from a nested field in the grid.

For this recipe, you'll need a record. Let's use the process-backed record from the Records Tutorial. If you haven't already created the Expense Report record type, do so now by completing the first five steps of the "Create Process-Backed Records" tutorial and starting at least one instance of the process, then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EXPENSE_REPORT_RECORD` with Record Type as the type and `Expense Report` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EXPENSE_REPORT_RECORD` constant.
4. Set the display name for each of the columns as `Item` and `Amount`, respectively.
5. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "expenseItem",
   ascending: true
  )
 ),
 with(
  local!datasubset: queryrecord(
   cons!EXPENSE_REPORT_RECORD,
   a!query(
    selection: a!querySelection(columns: {
```

```
        a!queryColumn(field: "expenseItem"),
        a!queryColumn(field: "expenseAmount"),
       }),
       pagingInfo: local!pagingInfo
      )
    ),
    a!gridField(
     totalCount: local!datasubset.totalCount,
     columns: {
      a!gridTextColumn(
       label: "Item",
       field: "expenseItem",
       data: index(local!datasubset.data, "expenseItem", null)
      ),
      a!gridTextColumn(
       label: "Amount",
       field: "expenseAmount",
       data: index(local!datasubset.data, "expenseAmount", null)
      ),
     },
     value: local!pagingInfo,
     saveInto: local!pagingInfo
    )
   )
)
```

6. Add a new query column for `pp.initiator` and a corresponding column in the grid by modifying the expression as shown below:

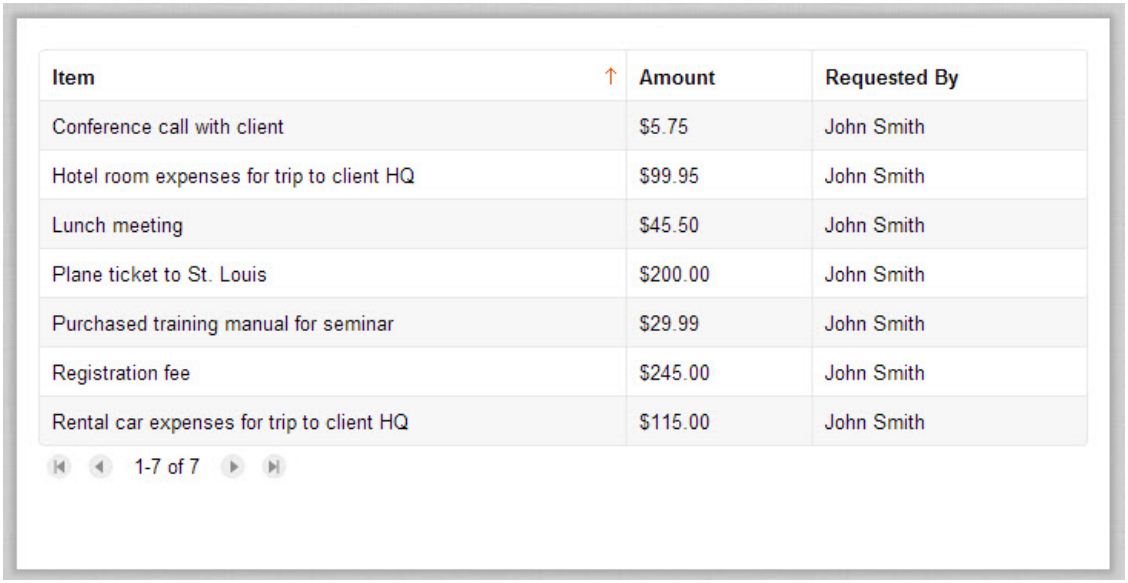
```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "expenseItem",
   ascending: true
  )
 ),
 with(
  local!datasubset: queryrecord(
   cons!EXPENSE_REPORT_RECORD,
   a!query(
    selection: a!querySelection(columns: {
     a!queryColumn(field: "expenseItem"),
     a!queryColumn(field: "expenseAmount"),
     a!queryColumn(field: "pp.initiator", alias: "initiator")
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  a!gridField(
   totalCount: local!datasubset.totalCount,
   columns: {
    a!gridTextColumn(
     label: "Item",
     field: "expenseItem",
     data: index(local!datasubset.data, "expenseItem", null)
    ),
    a!gridTextColumn(
     label: "Amount",
     field: "expenseAmount",
     data: index(local!datasubset.data, "expenseAmount", null)
    ),
    a!gridTextColumn(
     label: "Requested By",
     field: "initiator",
     data: index(local!datasubset.data, "initiator", null)
    )
   },
   value: local!pagingInfo,
   saveInto: local!pagingInfo
  )
 )
)
```

Notable implementation details

- The grid generated by the report builder is already configured to page and sort.
- Nested fields cannot be added through the report builder, so they must be added after the expression is generated.
- The query that populates this grid will return all data for the record type (in pages). If you want to return only a subset of data, add a default filte to the query. See also: Filter Data from a Record in a Grid

Format Data from a Record in a Grid

Goal: Format the data from a record type to display in a read-only paging grid, specifically a decimal number as a dollar amount and a username as a user's display name.



Item	Amount	Requested By
Conference call with client	$5.75	John Smith
Hotel room expenses for trip to client HQ	$99.95	John Smith
Lunch meeting	$45.50	John Smith
Plane ticket to St. Louis	$200.00	John Smith
Purchased training manual for seminar	$29.99	John Smith
Registration fee	$245.00	John Smith
Rental car expenses for trip to client HQ	$115.00	John Smith

1-7 of 7

This scenario demonstrates:

- How to use the report builder to generate a grid to display data from a record type.
- How to modify the generated expression to show data from a nested field in the grid.
- How to format the data that is returned from the query to display in the grid.

For this recipe, you'll need a record. Let's use the process-backed record from the Records Tutorial. If you haven't already created the Expense Report record type, do so now by completing the first five steps of the "Create Process-Backed Records" tutorial and starting at least one instance of the process, then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EXPENSE_REPORT_RECORD` with Record Type as the type and `Expense Report` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EXPENSE_REPORT_RECORD` constant.
4. Set the display name for each of the columns as `Item` and `Amount`, respectively.
5. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 20,
    sort: a!sortInfo(
      field: "expenseItem",
      ascending: true
    )
  ),
  with(
    local!datasubset: queryrecord(
      cons!EXPENSE_REPORT_RECORD,
      a!query(
        selection: a!querySelection(columns: {
          a!queryColumn(field: "expenseItem"),
          a!queryColumn(field: "expenseAmount"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!gridField(
      totalCount: local!datasubset.totalCount,
      columns: {
        a!gridTextColumn(
          label: "Item",
          field: "expenseItem",
          data: index(local!datasubset.data, "expenseItem", null)
        ),
```

```
      a!gridTextColumn(
        label: "Amount",
        field: "expenseAmount",
        data: index(local!datasubset.data, "expenseAmount", null)
      ),
    },
    value: local!pagingInfo,
    saveInto: local!pagingInfo
  )
 )
)
```

6. Add a new query column for `pp.initiator` and a corresponding column in the grid by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "expenseItem",
   ascending: true
  )
 ),
 with(
  local!datasubset: queryrecord(
   cons!EXPENSE_REPORT_RECORD,
   a!query(
    selection: a!querySelection(columns: {
     a!queryColumn(field: "expenseItem"),
     a!queryColumn(field: "expenseAmount"),
     a!queryColumn(field: "pp.initiator", alias: "initiator")
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  a!gridField(
   totalCount: local!datasubset.totalCount,
   columns: {
    a!gridTextColumn(
     label: "Item",
     field: "expenseItem",
     data: index(local!datasubset.data, "expenseItem", null)
    ),
    a!gridTextColumn(
     label: "Amount",
     field: "expenseAmount",
     data: index(local!datasubset.data, "expenseAmount", null)
    ),
    a!gridTextColumn(
     label: "Requested By",
     field: "initiator",
     data: index(local!datasubset.data, "initiator", null)
    )
   },
   value: local!pagingInfo,
   saveInto: local!pagingInfo
  )
 )
)
```

7. Create an expression rule `ucUserDisplayName` with a single input `user` of type User and the following definition:

```
=user(ri!user, "firstName") & " " & user(ri!user, "lastName")
```

8. Format the "Amount" column using the `dollar` function to display the value in dollars and the "Requested By" column using `rule!ucUserDisplayName` to display the user's first and last name by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "expenseItem",
   ascending: true
  )
```

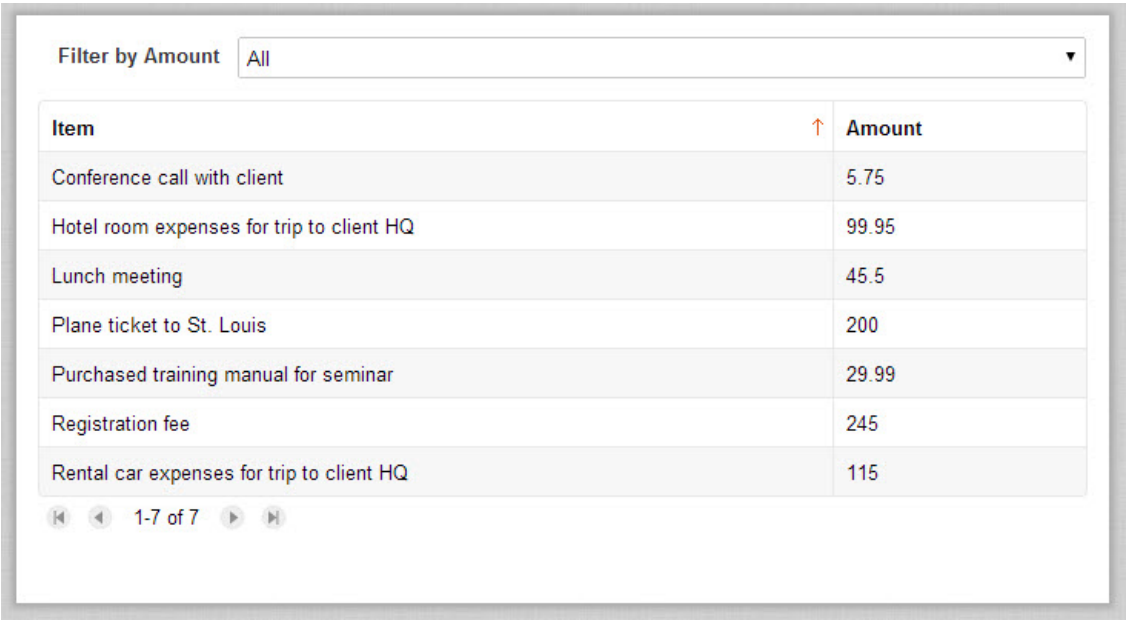
```
  ),
  with(
   local!datasubset: queryrecord(
    cons!EXPENSE_REPORT_RECORD,
    a!query(
     selection: a!querySelection(columns: {
      a!queryColumn(field: "expenseItem"),
      a!queryColumn(field: "expenseAmount"),
      a!queryColumn(field: "pp.initiator", alias: "initiator")
     }),
     pagingInfo: local!pagingInfo
    )
   ),
   a!gridField(
    totalCount: local!datasubset.totalCount,
    columns: {
     a!gridTextColumn(
      label: "Item",
      field: "expenseItem",
      data: index(local!datasubset.data, "expenseItem", null)
     ),
     a!gridTextColumn(
      label: "Amount",
      field: "expenseAmount",
      data: dollar(index(local!datasubset.data, "expenseAmount", {}))
     ),
     a!gridTextColumn(
      label: "Requested By",
      field: "initiator",
      data: apply(rule!ucUserDisplayName, index(local!datasubset.data, "initiator", {}))
     )
    },
    value: local!pagingInfo,
    saveInto: local!pagingInfo
   )
  )
)
```

Notable implementation details

- The grid generated by the report builder is already configured to page and sort.
- Nested fields cannot be added through the report builder, so they must be added after the expression is generated.
- The grid displays the text representation of all types, including Appian Objects such as a user, so we applied our own formatting.
- The query that populates this grid will return all data for the record type (in pages). If you want to return only a subset of data, add a default filte to the query. See also: Filter Data from a Record in a Grid

Filter Data from a Record in a Grid

Goal: Display data from a record type in a read-only paging grid and a dropdown to allow the user to filter the data that is displayed.



This scenario demonstrates:

- How to use the report builder to generate a grid to display data from a record type.
- How to modify the generated expression to add a filter for the data.

For this recipe, you'll need a record. Let's use the process-backed record from the Records Tutorial. If you haven't already created the Expense Report record type, do so now by completing the first five steps of the "Create Process-Backed Records" tutorial and starting at least one instance of the process, then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EXPENSE_REPORT_RECORD` with Record Type as the type and `Expense Report` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EXPENSE_REPORT_RECORD` constant.
4. Set the display name for each of the columns as `Item` and `Amount`, respectively.
5. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 20,
    sort: a!sortInfo(
      field: "expenseItem",
      ascending: true
    )
  ),
  with(
    local!datasubset: queryrecord(
      cons!EXPENSE_REPORT_RECORD,
      a!query(
        selection: a!querySelection(columns: {
          a!queryColumn(field: "expenseItem"),
          a!queryColumn(field: "expenseAmount"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!gridField(
      totalCount: local!datasubset.totalCount,
      columns: {
        a!gridTextColumn(
          label: "Item",
          field: "expenseItem",
          data: index(local!datasubset.data, "expenseItem", null)
        ),
        a!gridTextColumn(
          label: "Amount",
          field: "expenseAmount",
          data: index(local!datasubset.data, "expenseAmount", null)
        ),
      },
      value: local!pagingInfo,
      saveInto: local!pagingInfo
    )
  )
)
```

6. Add a filter to the query and a dropdown so that the user can select what data set to display by modifying the expression as shown below:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 20,
    sort: a!sortInfo(
      field: "expenseItem",
      ascending: true
    )
  ),
  local!priceRange,
  with(
    local!datasubset: queryrecord(
      cons!EXPENSE_REPORT_RECORD,
      a!query(
        selection: a!querySelection(columns: {
          a!queryColumn(field: "expenseItem"),
          a!queryColumn(field: "expenseAmount"),
        }),
        filter: if(
          isnull(local!priceRange),
```

```
        null,
        a!queryFilter(
          field: "expenseAmount",
          operator: choose(local!priceRange, "<", "between", ">"),
          value: choose(local!priceRange, 100, {100, 200}, 200)
        )
      ),
      pagingInfo: local!pagingInfo
    )
  ),
  {
    a!dropdownField(
      label: "Filter by Amount",
      labelPosition: "ADJACENT",
      choiceLabels: {"Less than $100", "$100 - $200", "Greater than $200"},
      choiceValues: {1, 2, 3},
      placeholderLabel: "All",
      value: local!priceRange,
      saveInto: {
        local!priceRange,
        /* We need to reset the paging info so that it goes back to the first
         * page of the grid when the user changes the filter. Otherwise, the grid
         * errors out */
        a!save(local!pagingInfo.startIndex, 1)
      }
    ),
    a!gridField(
      totalCount: local!datasubset.totalCount,
      columns: {
        a!gridTextColumn(
          label: "Item",
          field: "expenseItem",
          data: index(local!datasubset.data, "expenseItem", null)
        ),
        a!gridTextColumn(
          label: "Amount",
          field: "expenseAmount",
          data: index(local!datasubset.data, "expenseAmount", null)
        ),
      },
      value: local!pagingInfo,
      saveInto: local!pagingInfo
    )
  }
 )
)
```

Test it out

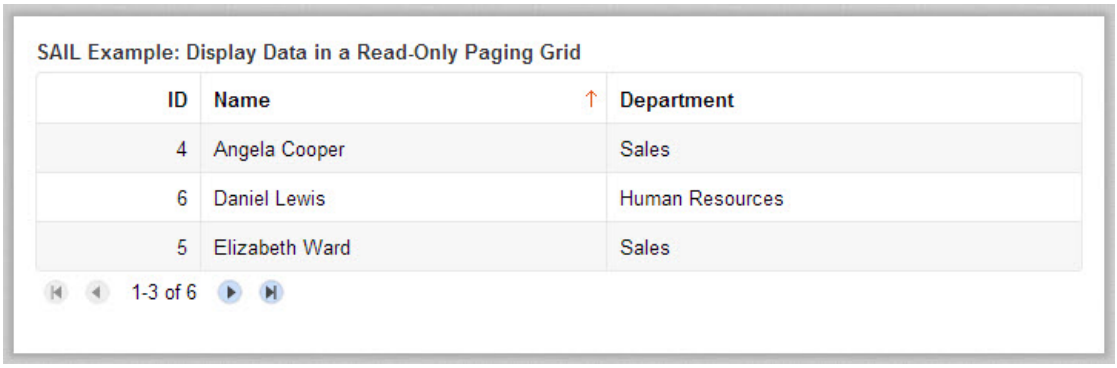
1. Select the "Less than $100" option from the filter dropdown. Notice that only items where the amount is less than $100 are displayed in the grid.

Notable implementation details

- The grid generated by the report builder is already configured to page and sort.
- Notice that when the user makes a selection from the dropdown, we're always resetting the value of `local!pagingInfo` so that the user always see the first page of results for the selected filter. This is necessary regardless of what the user has selected, so we ignore the value returned by the component (in this case, the value of the dropdown selection) and instead insert our own value.

Display Array of Data in a Grid

Goal: Display an array of CDT data in a read-only paging grid.



This scenario demonstrates:

- How to display an array of data in a read-only paging grid.
- How to configure paging and sorting in a read-only grid.

Expression

```
=load(
 local!data: {
  {id: 1, name: "John Smith",      department: "Engineering"},
  {id: 2, name: "Michael Johnson", department: "Finance"},
  {id: 3, name: "Mary Reed",       department: "Engineering"},
  {id: 4, name: "Angela Cooper",   department: "Sales"},
  {id: 5, name: "Elizabeth Ward",  department: "Sales"},
  {id: 6, name: "Daniel Lewis",    department: "Human Resources"}
 },
 /* batchSize is 3 to show more than 1 page of data in this recipe. Increase it as needed. `*/
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 3, sort: a!sortInfo(field: "name", ascending: true)),
 with(
  local!datasubset: todatasubset(local!data, local!pagingInfo),
  a!gridField(
   label: "SAIL Example: Display Data in a Read-Only Paging Grid",
   totalCount: local!datasubset.totalCount,
   columns: {
    a!gridTextColumn(
     label: "ID",
     field: "id",
     data: index(local!datasubset.data, "id", {}),
     alignment: "RIGHT"
    ),
    a!gridTextColumn(
     label: "Name",
     field: "name",
     data: index(local!datasubset.data, "name", {})
    ),
    a!gridTextColumn(
     label: "Department",
     field: "department",
     data: index(local!datasubset.data, "department", {})
    )
   },
   value: local!pagingInfo,
   saveInto: local!pagingInfo
  )
 )
)
```

Test it out

1. Go to the second page of the grid. Notice that grid data changes to show the next batch of data.
2. Sort the grid by clicking on one of the column headers. Notice that the grid goes back to the first page and displays the data in the correct order

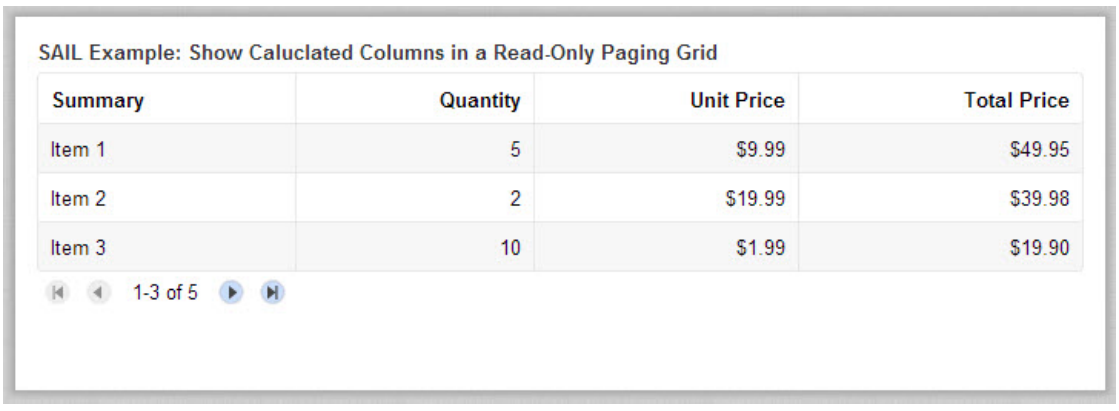
Notable implementation details

- Notice that `local!pagingInfo` is a `load()` variable and `local!datasubset` is a `with()` variable. This allows us to save a new value into `local!pagingInfo` when the user interacts with the grid and then use that value to recalculate `local!datasubset`

See also: load(), with(), Paging and Sorting Configurations

Show Calculated Columns in a Grid

Goal: Display and sort on an additional column for total price that is not a field in the CDT, but calculated based on the unit price and quantity fields. To do this, we'll store the calculated data in a dictionary that contains the relevant CDT fields as well as the additional calculate field.



The main expression uses a supporting rule, so let's create it first.

- `ucGridData` : Calculates additional data to be displayed in the grid

Create expression rule `ucGridData` with the following rule inputs:

- data (Any Type)

Enter the following definition for the rule:

```
={
 summary: tostring(ri!data.summary),
 /* Explicit casting allows us to sort using todatasubset(). This is only *
  * necessary because the underlying data is a dictionary.                */
 qty: tointeger(ri!data.qty),
 unitPrice: todecimal(ri!data.unitPrice),
 totalPrice: tointeger(ri!data.qty) * todecimal(ri!data.unitPrice)
}
```

Now that we've created the supporting rule, let's move on to the main expression.

Expression

```
=load(
 local!data: {
  {summary: "Item 1", qty: 5, unitPrice: 9.99},
  {summary: "Item 2", qty: 2, unitPrice: 19.99},
  {summary: "Item 3", qty: 10, unitPrice: 1.99},
  {summary: "Item 4", qty: 4, unitPrice: 14.99},
  {summary: "Item 5", qty: 7, unitPrice: 3.99}
 },
 local!gridData: apply(rule!ucGridData, local!data),
 /* batchSize is 3 to show more than 1 page of data in this recipe. Increase it as needed. */
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 3),
 with(
  local!datasubset: todatasubset(local!gridData, local!pagingInfo),
  a!gridField(
   label: "SAIL Example: Show Caluclated Columns in a Read-Only Paging Grid",
   totalCount: local!datasubset.totalCount,
   columns: {
    a!gridTextColumn(
     label: "Summary",
     field: "summary",
     data: index(local!datasubset.data, "summary", {})
    ),
    a!gridTextColumn(
     label: "Quantity",
     field: "qty",
     data: index(local!datasubset.data, "qty", {}),
     alignment: "RIGHT"
    ),
    a!gridTextColumn(
     label: "Unit Price",
     field: "unitPrice",
     data: dollar(index(local!datasubset.data, "unitPrice", {})),
     alignment: "RIGHT"
    ),
    a!gridTextColumn(
     label: "Total Price",
     field: "totalPrice",
     data: dollar(index(local!datasubset.data, "totalPrice", {})),
     alignment: "RIGHT"
    )
   },
   value: local!pagingInfo,
   saveInto: local!pagingInfo
  )
 )
)
```

Test it out

1. Sort the grid by the "Total Price" column. Notice that the data sorts appropriately even across all pages of data.

Notable implementation details

- In order to correctly sort on the calculated column, we queried the entire data set, calculated the new value for each row, then paged and sorted on the result. If you were to only calculate the column for the current page, sorting on the calculated column would not work correctly across

pages.
- Since you must query all data and then loop over each item to calculate the additional data, this technique should not be used for a large amoun of data, as the query and calculation may become slow. To further optimize this grid, only perform the calculation on every row when the user sorts by the calculated column, otherwise, simply calculate the data for the current page.

Conditionally Hide a Column in a Grid

Goal: Conditionally hide a column in a read-only paging grid when all data for that column is a specific value. In this case, only display the "Active?" column if there is at least one inactive employee.

Expression

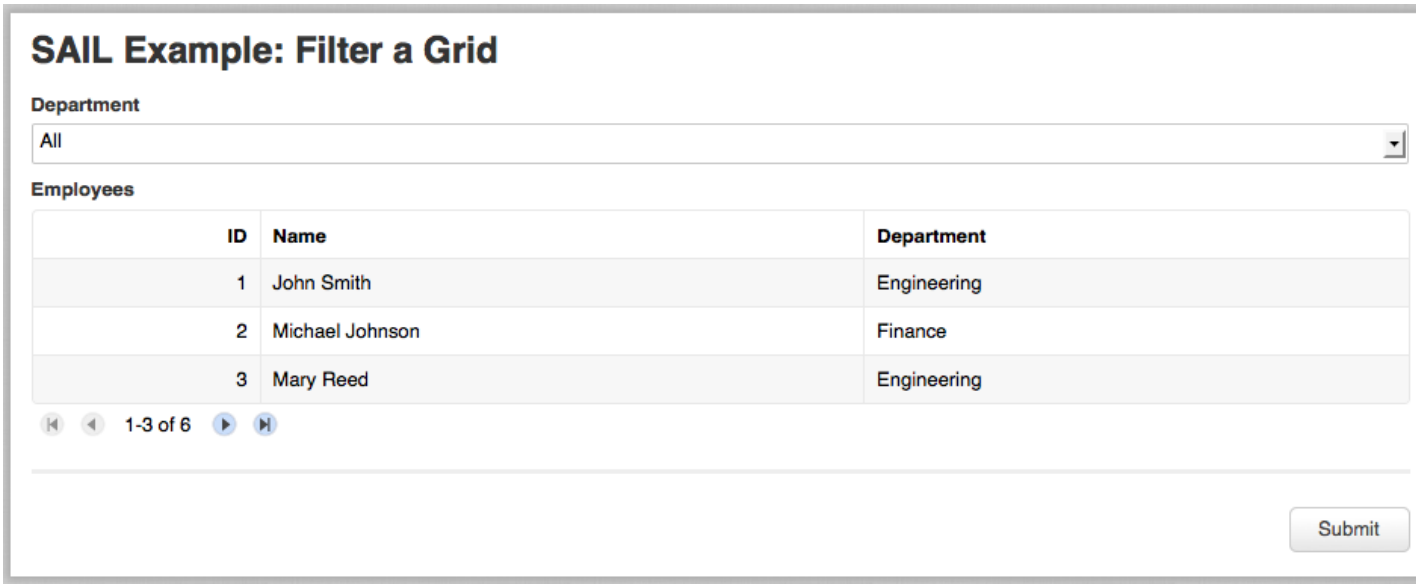
```
=load(
 local!data: {
  {id: 1, name: "John Smith",      department: "Engineering",     active: true},
  {id: 2, name: "Michael Johnson", department: "Finance",         active: true},
  {id: 3, name: "Mary Reed",       department: "Engineering",     active: true},
  {id: 4, name: "Angela Cooper",   department: "Sales",           active: true},
  {id: 5, name: "Elizabeth Ward",  department: "Sales",           active: true},
  {id: 6, name: "Daniel Lewis",    department: "Human Resources", active: true}
 },
 /* batchSize is 3 to show more than 1 page of data in this recipe. Increase it as needed. */
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 3),
 with(
  local!datasubset: todatasubset(local!data, local!pagingInfo),
  a!gridField(
   label: "SAIL Example: Display Data in a Read-Only Paging Grid",
   totalCount: local!datasubset.totalCount,
   columns: {
    a!gridTextColumn(
     label: "ID",
     field: "id",
     data: index(local!datasubset.data, "id", {}),
     alignment: "RIGHT"
    ),
    a!gridTextColumn(
     label: "Name",
     field: "name",
     data: index(local!datasubset.data, "name", {})
    ),
    a!gridTextColumn(
     label: "Department",
     field: "department",
     data: index(local!datasubset.data, "department", {})
    ),
    if(
     and(local!data.active),
     {},
     a!gridTextColumn(
      label: "Active?",
      field: "active",
      data: if(index(local!datasubset.data, "active", {}), "Yes", "No")
     )
    )
   },
   value: local!pagingInfo,
   saveInto: local!pagingInfo
  )
 )
)
```

Test it out

1. Change one of the *active* fields to `false` and click the **Test** button. Notice that the column is no longer hidden.

Filter the Data in a Grid

Goal: Filter the data in a read-only paging grid using a dropdown. When the user selects a value to filter by, update the grid to show the result. The result displays on page 1 even if the user was previously on a different page number.



This scenario demonstrates:

- How to change the value of a variable when the user updates another variable.
- How to ignore the value returned by the component and update a variable with a literal value.

Expression

```
=load(
 local!data: {
  {id: 1, name: "John Smith",      department: "Engineering"},
  {id: 2, name: "Michael Johnson", department: "Finance"},
  {id: 3, name: "Mary Reed",       department: "Engineering"},
  {id: 4, name: "Angela Cooper",   department: "Sales"},
  {id: 5, name: "Elizabeth Ward",  department: "Sales"},
  {id: 6, name: "Daniel Lewis",    department: "Human Resources"}
 },
 /* batchSize is 3 to show more than 1 page of data in this recipe. Increase it as needed. */
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 3),
 local!department,
 with(
  /* Set the value of local!filteredData based on your filtering logic */
  local!filteredData: if(isnull(local!department), local!data, index(local!data, where(search(local!department, local!data.department)), {})),
  local!datasubset: todatasubset(local!filteredData, local!pagingInfo),
  a!formLayout(
   label: "SAIL Example: Filter a Grid",
   firstColumnContents: {
    a!dropdownField(
     label: "Department",
     choiceLabels: {"Engineering", "Finance", "Sales", "Human Resources"},
     choiceValues: {"Engineering", "Finance", "Sales", "Human Resources"},
     placeholderLabel: "All",
     value: local!department,
     saveInto: {
      local!department,
      /* We need to reset the paging info so that it goes back to the first
       * page of the grid when the user changes the filter. Otherwise, the grid
       * errors out */
      a!save(local!pagingInfo.startIndex, 1)
     }
    ),
    a!gridField(
     label: "Employees",
     totalCount: local!datasubset.totalCount,
     columns: {
      a!gridTextColumn(
       label: "ID",
       field: "id",
       data: index(local!datasubset.data, "id", {}),
       alignment: "RIGHT"
      ),
      a!gridTextColumn(
       label: "Name",
       field: "name",
       data: index(local!datasubset.data, "name", {})
```

```
          ),
          a!gridTextColumn(
            label: "Department",
            field: "department",
            data: index(local!datasubset.data, "department", {})
          )
        },
        value: local!pagingInfo,
        saveInto: local!pagingInfo
      )
    },
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(label: "Submit")
    )
  )
 )
)
```

Test it out

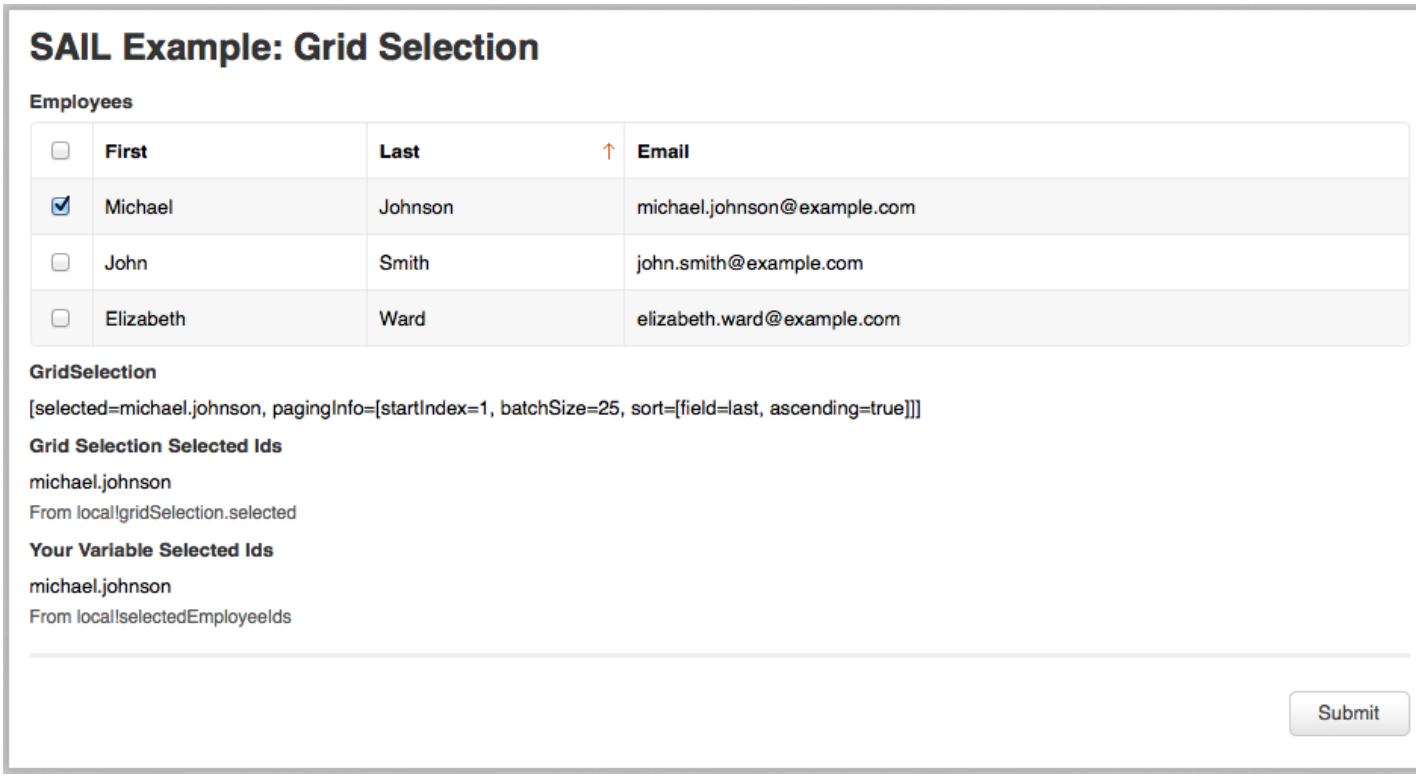
1. Select a department from the dropdown to filter the grid.
2. Select "All" from the dropdown. Go to the second page of the grid, then select a department. Notice that you go back to the first page of return items.

Notable implementation details

- Notice that when the user makes a selection from the dropdown, we're always resetting the value of `local!pagingInfo` so that the user always see the first page of results for the selected filter. This is necessary regardless of what the user has selected, so we ignore the value returned by the component (in this case, the value of the dropdown selection) and instead insert our own value.
- When you configure your grid, replace the value of `local!data` with the result of your `a!queryEntity()` or `queryrecord()`. These functions allow yo to retrieve only the fields that you need to configure your dropdown. See also: SAIL Design Best Practices
- When using `a!queryEntity()` or `queryrecord()`, you can remove the filtering step and simply use a filter within `a!queryEntity()` or `queryrecord()` itself. For examples, see Query Recipes.

Select Rows in a Grid

Goal: Display checkboxes on a read-only paging grid to allow users to select multiple rows of data.



Expression

```
=load(
 local!selectedEmployeeIds,
 local!employeeData: {
  {id: "john.smith",      first: "John",      last: "Smith",   email: "john.smith@example.com"},
  {id: "michael.johnson", first: "Michael",   last: "Johnson", email: "michael.johnson@example.com"},
  {id: "elizabeth.ward",  first: "Elizabeth", last: "Ward",    email: "elizabeth.ward@example.com"}
 },
 /* Set the default paging and sorting config */
```

```
  local!gridSelection: a!gridSelection(
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: 25,
      sort: a!sortInfo(
        field: "last",
        ascending: true
      )
    )
  ),
  with(
    /* Replace the value of local!datasubset with a!queryEntity(), or queryrecord(),
     * or use your own CDT array in todatasubset() */
    local!datasubset: todatasubset(local!employeeData, local!gridSelection.pagingInfo),
    a!formLayout(
      label: "SAIL Example: Grid Selection",
      firstColumnContents:{
        a!gridField(
          label: "Employees",
          totalCount: local!datasubset.totalCount,
          selection: true,
          identifiers: index(local!datasubset.data, "id" , {}),
          columns: {
            a!gridTextColumn(label: "First", field: "first", data: index(local!datasubset.data, "first" , {})),
            a!gridTextColumn(label: "Last",  field: "last",  data: index(local!datasubset.data, "last" , {})),
            a!gridTextColumn(label: "Email", field: "email", data: index(local!datasubset.data, "email" , {}))
          },
          value: local!gridSelection,
          saveInto: {
            local!gridSelection,
            a!save(local!selectedEmployeeIds, index(save!value, "selected", null))
          }
        ),
        a!textField(
          label: "GridSelection",
          value: local!gridSelection,
          readOnly: true
        ),
        a!textField(
          label: "Grid Selection Selected Ids",
          instructions: "From local!gridSelection.selected",
          readOnly: true,
          value: local!gridSelection.selected
        ),
        a!textField(
          label: "Your Variable Selected Ids",
          instructions: "From local!selectedEmployeeIds",
          readOnly: true,
          value: local!selectedEmployeeIds
        )
      },
      buttons: a!buttonLayout(
        primaryButtons: a!buttonWidgetSubmit(label: "Submit")
      )
    )
  )
)
```

Test it out

1. Select a checkbox or two and observe the values of the text fields.
    - The *GridSelection* text field displays the value of the entire GridSelection object returned by the grid's `saveInto` parameter.
    - The *Selected Ids* text field displays just the value of the GridSelection's `selected` attribute.
    - The *Your Variable Selected Ids* text field displays the value of the local variable that only has the selected ids, which is the value that you would ultimately use.

Notable implementation details

- The `value` of the grid is GridSelection rather than PagingInfo, and the value that the grid returns when the user interacts with it is also a GridSelection. The PagingInfo information is embedded in GridSelection. GridSelection must be used when a grid is configured for selection.

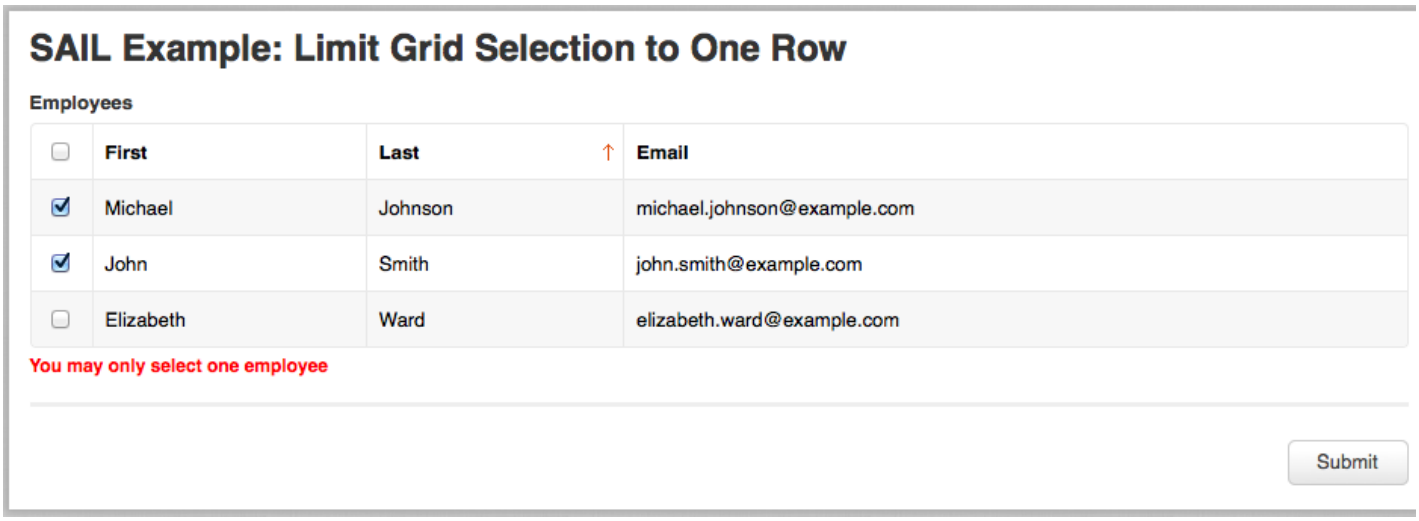
To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: selectedEmployeeIds (Text)
3. Delete local variable: `local!selectedEmployeeIds`
4. In your expression, replace:

    - `local!selectedEmployeeIds` with `ri!selectedEmployeeIds`
5. In your process model, create variables: selectedEmployeeIds (Text) with no value
    - On a task form, create node inputs
    - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
    - On a task form: `=rule!sailRecipe(selectedEmployeeIds: ac!selectedEmployeeIds)`
    - On a start form: `=rule!sailRecipe(selectedEmployeeIds: pv!selectedEmployeeIds)`

Limit the Number of Rows in a Grid That Can Be Selected

Goal: If you want to limit how many rows users can select in a read-only paging grid, you can constrain the number of selections by using validation. Ir this example, we limit the number of rows that the user can select to one.



Expression

```
=load(
 local!selectedEmployeeId,
 local!employeeData: {
  {id: "john.smith",      first: "John",      last: "Smith",   email: "john.smith@example.com"},
  {id: "michael.johnson", first: "Michael",   last: "Johnson", email: "michael.johnson@example.com"},
  {id: "elizabeth.ward",  first: "Elizabeth", last: "Ward",    email: "elizabeth.ward@example.com"}
 },
 local!gridSelection: a!gridSelection(
  pagingInfo: a!pagingInfo(
   startIndex: 1,
   batchSize: 25,
   sort: a!sortInfo(
    field: "last",
    ascending: true
   )
  )
 ),
 with(
  /* Replace the value of local!datasubset with a!queryEntity(), or queryrecord(),
   * or use your own CDT array in todatasubset() */
  local!datasubset: todatasubset(local!employeeData, local!gridSelection.pagingInfo),
  a!formLayout(
   label: "SAIL Example: Limit Grid Selection to One Row",
   firstColumnContents:{
    a!gridField(
     label: "Employees",
     totalCount: local!datasubset.totalCount,
     selection: true,
     identifiers: index(local!datasubset.data, "id", {}),
     columns: {
      a!gridTextColumn(label: "First", field: "first", data: index(local!datasubset.data, "first", {})),
      a!gridTextColumn(label: "Last",  field: "last",  data: index(local!datasubset.data, "last", {})),
      a!gridTextColumn(label: "Email", field: "email", data: index(local!datasubset.data, "email", {}))
     },
     validations: if(count(local!gridSelection.selected)>1, "You may only select one employee", null),
     value: local!gridSelection,
     saveInto: {
      local!gridSelection,
      if(
       count(local!gridSelection.selected)>1,
       {}, /* Does not update the value if the user is attempting to select more than one id */
```

```
            a!save(local!selectedEmployeeId, index(save!value, "selected", null)) /* Shows how to store the selected id only */
          )
        }
      )
    },
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(label: "Submit")
    )
  )
 )
)
```

Test it out

1. Select one of the rows in the grid and note that there is no validation message.
2. Select a second row and note that a validation message appears.

Notable implementation details

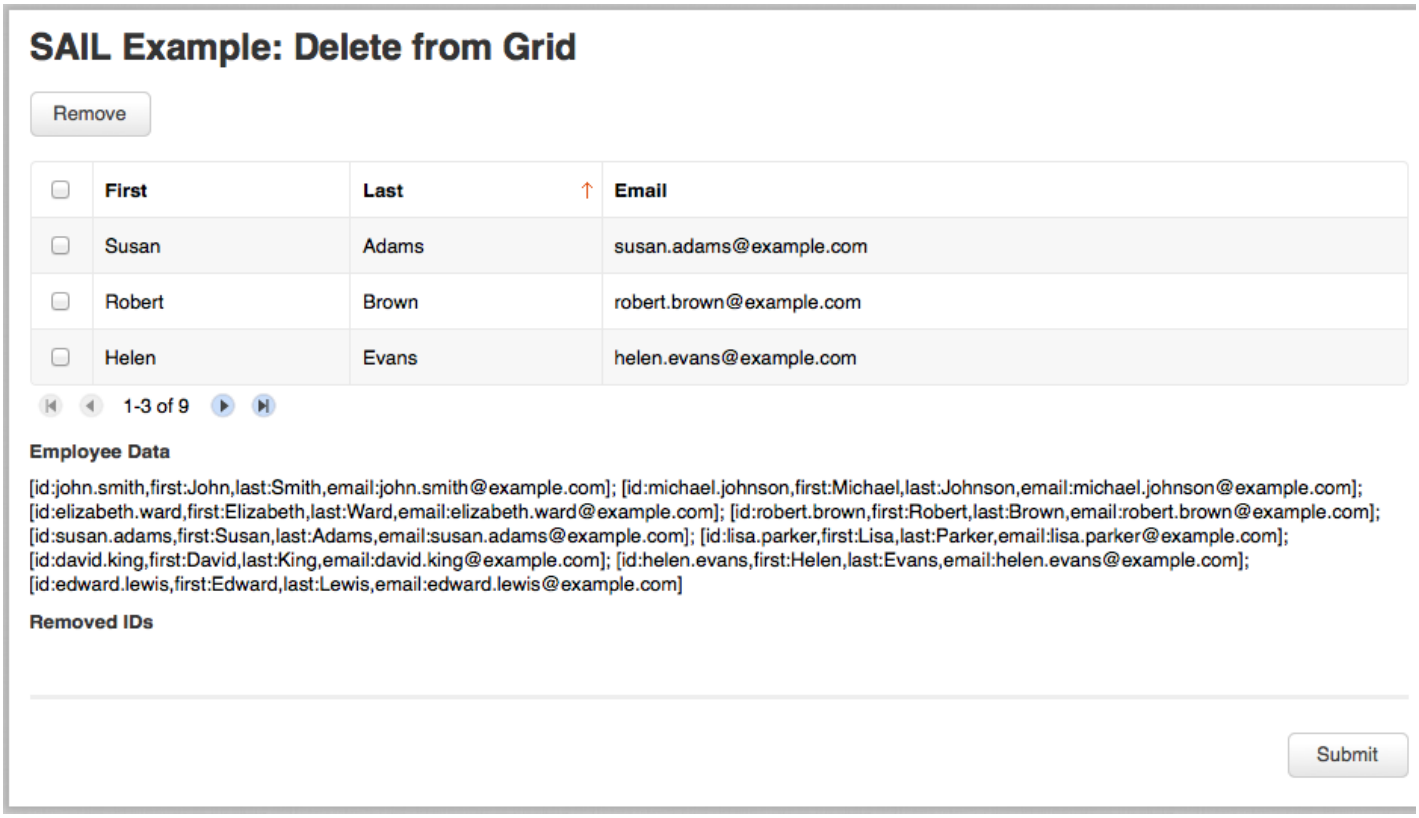
- The `value` of the grid is GridSelection rather than PagingInfo, and the value that the grid returns when the user interacts with it is also a GridSelection. The PagingInfo information is embedded in GridSelection. GridSelection must be used when a grid is configured for selection.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: selectedEmployeeId (Text)
3. Delete local variable: `local!selectedEmployeeId`
4. In your expression, replace:
   - `local!selectedEmployeeId` with `ri!selectedEmployeeId`
5. In your process model, create variables: selectedEmployeeId (Text) with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(selectedEmployeeId: ac!selectedEmployeeId)`
   - On a start form: `=rule!sailRecipe(selectedEmployeeId: pv!selectedEmployeeId)`

Delete Rows in a Grid

Goal: Delete one or more rows of data in a read-only paging grid.



The main expression uses a supporting rule, so let's create it first.

- `ucReturnNewStartIndex` : Returns a start index for the paging grid. This rule makes sure that the grid does not break when all the entries on the la page are deleted, resetting the view to the new last page of results.

Create expression rule `ucReturnNewStartIndex` with the following rule inputs:

- pagingInfo (Any Type)

- length (Number (Integer))

Enter the following definition for the rule:

```
if(ri!pagingInfo.startIndex < ri!length,
/* If at least as many items as the previous start index exist, use it */
  ri!pagingInfo.startIndex,
  /* Otherwise, create a new start index to avoid an error */
  if(ri!length<1,
    /* If the last item in the grid was deleted, use 1 */
    1,
    /* If there is more than one item remaining, adjust the start index so that the last page of items is shown */
    (ri!length+1)-ri!pagingInfo.batchSize
  )
)
```

Now that we've created the supporting rule, let's move on to the main expression.

Expression

```
load(
 local!removeFailure: false,
 local!removedIDs: {},
 local!employeeData: {
  {id: "john.smith",      first: "John",      last: "Smith",   email: "john.smith@example.com"},
  {id: "michael.johnson", first: "Michael",   last: "Johnson", email: "michael.johnson@example.com"},
  {id: "elizabeth.ward",  first: "Elizabeth", last: "Ward",    email: "elizabeth.ward@example.com"},
  {id: "robert.brown",    first: "Robert",    last: "Brown",   email: "robert.brown@example.com"},
  {id: "susan.adams",     first: "Susan",     last: "Adams",   email: "susan.adams@example.com"},
  {id: "lisa.parker",     first: "Lisa",      last: "Parker",  email: "lisa.parker@example.com"},
  {id: "david.king",      first: "David",     last: "King",    email: "david.king@example.com"},
  {id: "helen.evans",     first: "Helen",     last: "Evans",   email: "helen.evans@example.com"},
  {id: "edward.lewis",    first: "Edward",    last: "Lewis",   email: "edward.lewis@example.com"}
 },
 /* Set the default paging and sorting config */
 local!gridSelection: a!gridSelection(
  pagingInfo: a!pagingInfo(
   startIndex: 1,
   batchSize: 3,
   sort: a!sortInfo(
    field: "last",
    ascending: true
   )
  )
 ),
 with(
  /* Replace the value of local!datasubset witha!queryEntity(), orqueryrecord(),
   * or use your own CDT array in todatasubset() */
  local!datasubset: todatasubset(local!employeeData, local!gridSelection.pagingInfo),
  a!formLayout(
   label: "SAIL Example: Delete from Grid",
   firstColumnContents:{
    a!buttonLayout(
     secondaryButtons: {
      a!buttonWidget(
       label: "Remove",
       value: true,
       saveInto: {
        if(or(isnull(local!gridSelection.selected), length(local!gridSelection.selected)<1),
         local!removeFailure,
         {}
        ),
        a!save(local!employeeData, remove(local!employeeData, wherecontains(local!gridSelection.selected, local!employeeData.id))),
        if(local!removeFailure,
         {},
         a!save(local!removedIDs, append(local!removedIDs, local!gridSelection.selected))
        ),
        /*  This sets the start index back by one page when  *
         *  all of the results on the final page are deleted.  */
        a!save(
         local!gridSelection.pagingInfo.startIndex,
         rule!ucReturnNewStartIndex(pagingInfo: local!gridSelection.pagingInfo, length: length(local!employeeData))
        ),
        a!save(local!gridSelection.selected, null)
       }
      )
```

```
        }
      ),
      a!gridField(
        instructions: if(local!removeFailure, "Select one or more items to remove.", ""),
        totalCount: local!datasubset.totalCount,
        selection: true,
        identifiers: index(local!datasubset.data, "id" , {}),
        columns: {
          a!gridTextColumn(label: "First", field: "first", data: index(local!datasubset.data, "first" , {})),
          a!gridTextColumn(label: "Last",  field: "last",  data: index(local!datasubset.data, "last" , {})),
          a!gridTextColumn(label: "Email", field: "email", data: index(local!datasubset.data, "email" , {}))
        },
        value: local!gridSelection,
        saveInto: {
          local!gridSelection,
          a!save(
            local!removeFailure,
            if(
              or(isnull(local!gridSelection.selected), length(local!gridSelection.selected) < 1),
              local!removeFailure,
              false
            )
          )
        }
      ),
      a!textField(
        label: "Employee Data",
        value: local!employeeData,
        readOnly: true
      ),
      a!textField(
        label: "Removed IDs",
        readOnly: true,
        value: local!removedIDs
      )
    },
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(label: "Submit")
    )
  )
 )
)
```

Test it out

1. Click Remove and note that instructions appear above the grid because you have not selected any rows.
2. Select a checkbox or two, click Remove, and note that the rows are removed from the grid.
    - The *Employee Data* text field displays the value of the entire test data object.
    - The *Removed IDs* text field tracks the IDs of rows deleted from the grid. This allows the user to verify that the data has been modified by the delete action.

Notable implementation details

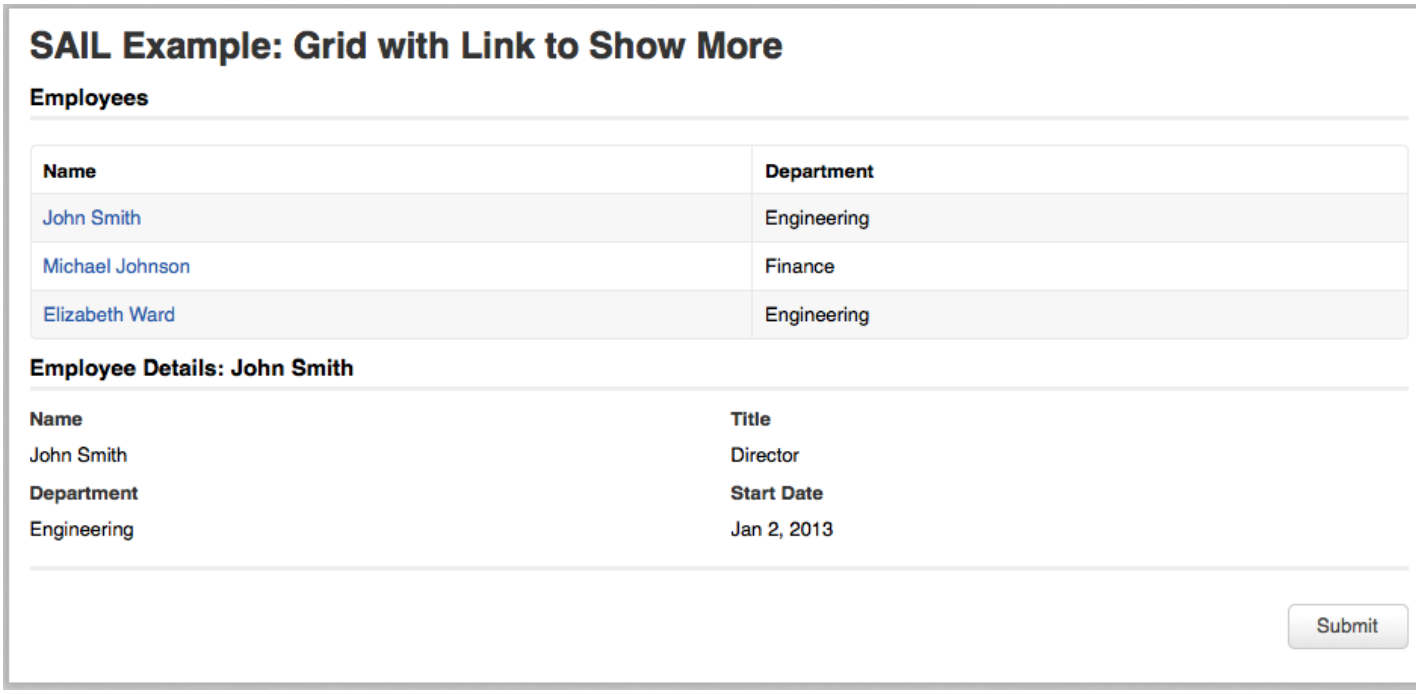
- The `value` of the grid is GridSelection rather than PagingInfo, and the value that the grid returns when the user interacts with it is also a GridSelection. The PagingInfo information is embedded in GridSelection. GridSelection must be used when a grid is configured for selection.
- When deleting rows in a paging grid, we need to tell the grid what to display when all the results on a page are deleted. `rule!ucReturnNewStartInde` makes sure that the grid does not break when all the entries on the last page are deleted, resetting the view to the new last page of results. Since we're now manually setting our start index when the last page is deleted, we now have to catch when the last result overall is deleted, to correctl set the index to 1.
- In this example, we stored the ids of the removed items. This is useful when you want to remove items from an external source, such as a database or another process. If the data on the form is the authoritative version, then you may wish to return the remaining items, rather than what was removed. For that, follow the process below for `local!employeeData` instead of `local!removedIDs`.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: removedIDs (List of Text)
3. Delete local variables: `local!removedIDs`
4. In your expression, replace:
    - `local!removedIDs` with `ri!removedIDs`
5. In your process model, create variables: removedIDs (List of Text) with no value
    - On a task form, create node inputs
    - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
    - On a task form: `=rule!sailRecipe(removedIDs: ac!removedIDs)`
    - On a start form: `=rule!sailRecipe(removedIDs: pv!removedIDs)`

Use Links in a Grid to Show More Details About an Object

Goal: Display more of an object's data than can fit in a single row in a read-only paging grid. Rather than placing all the data in the grid, have the user select a row by clicking on a link, and show the row details in a separate section of the form.



To achieve this scenario, we use `a!dynamicLink` to modify a variable, which can then be used to populate other areas of the SAIL interface.

Expression

```
=load(
 local!selectedEmployeeId,
 local!employees: {
  {id: 1, name: "John Smith",      department:"Engineering", title: "Director",          startDate: date(2013, 1, 2)},
  {id: 2, name: "Michael Johnson", department:"Finance",     title: "Analyst",           startDate: date(2012, 6, 5)},
  {id: 3, name: "Elizabeth Ward",  department:"Engineering", title: "Software Engineer", startDate: date(2001, 1, 2)}
 },
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 25),
 with(
  local!employeeDataSubset: todatasubset(local!employees, local!pagingInfo),
  local!data: index(local!employeeDataSubset, "data", null),
  a!formLayout(
   label: "SAIL Example: Grid with Link to Show More",
   firstColumnContents: {
    a!sectionLayout(
     label: "Employees",
     firstColumnContents: {
      a!gridField(
       totalCount: local!employeeDataSubset.totalCount,
       columns: {
        a!gridTextColumn(
         label: "Name",
         data: index(local!data, "name", {}),
         /* Creates a dynamic link for every item in local!data */
         links: apply(a!dynamicLink(value: _, saveInto: local!selectedEmployeeId), index(local!data, "id", {}))
        ),
        a!gridTextColumn(
         label: "Department",
         data: index(local!data, "department", null)
        )
       },
       value: local!pagingInfo,
       saveInto: local!pagingInfo
      )
     }
    ),
    if(isnull(local!selectedEmployeeId), {},
     with(
      /* Replace with your rule to get the employee details */
      local!selectedEmployee: displayvalue(local!selectedEmployeeId, local!employees.id, local!employees, {}),
      a!sectionLayout(
```

```
          label: "Employee Details: " & local!selectedEmployee.name,
          firstColumnContents: {
            a!textField(
              label: "Name",
              readOnly: true,
              value: local!selectedEmployee.name
            ),
            a!textField(
              label: "Department",
              readOnly: true,
              value: local!selectedEmployee.department
            )
          },
          secondColumnContents: {
            a!textField(
              label: "Title",
              readOnly: true,
              value: local!selectedEmployee.title
            ),
            a!dateField(
              label: "Start Date",
              readOnly: true,
              value: local!selectedEmployee.startDate
            )
          }
        )
      )
    )
  },
  buttons: a!buttonLayout(
    primaryButtons: a!buttonWidgetSubmit(label: "Submit")
  )
 )
)
)
```

Test it out

1. Click on the names in the grid in the left column. Notice how the value in the fields in the right column change to reflect the name you most recently clicked on.

Use Links in a Grid to Show More Details and Edit Data

Goal: Allow end users to click a link in a read-only paging grid to view the details for the row, and make changes to the data. The data available for editir may include more fields than are displayed in the grid.

To achieve this goal, we use `a!dynamicLink` to store the selected employee data into a variable, and then allow editable text fields to update the individu fields of the employee. Finally, we associate a `saveInto` with the submit button to update the array of employees that populates the grid with the updated employee.

Expression

```
=load(
 local!employees: {
  {id: 1, name: "John Smith",      department:"Engineering", title: "Director",          startDate: date(2013, 1, 2)},
  {id: 2, name: "Michael Johnson", department:"Finance",     title: "Analyst",           startDate: date(2012, 6, 5)},
  {id: 3, name: "Elizabeth Ward",  department:"Engineering", title: "Software Engineer", startDate: date(2001, 1, 2)}
 },
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 25),
 local!employeeToUpdate,
 with(
  local!employeeDataSubset: todatasubset(local!employees, local!pagingInfo),
  a!formLayout(
   label: "SAIL Example: Grid with Link to Show More and Edit Data",
   firstColumnContents: {
    a!sectionLayout(
     label: "Employees",
     firstColumnContents: {
      a!gridField(
       totalCount: local!employeeDataSubset.totalCount,
       columns: {
        a!gridTextColumn(
         label: "Name",
         data: index(local!employeeDataSubset.data, "name", {}),
         /* Creates a dynamic link for every item in local!data */
         links: apply(a!dynamicLink(value: _, saveInto: local!employeeToUpdate), local!employeeDataSubset.data)
```

```
            ),
            a!gridTextColumn(
              label: "Department",
              data: index(local!employeeDataSubset.data, "department", null)
            )
          },
          value: local!pagingInfo,
          saveInto: local!pagingInfo
        )
      }
    ),
    if(isnull(local!employeeToUpdate), {},
      a!sectionLayout(
        label: "Employee Details: " & local!employeeToUpdate.name,
        firstColumnContents: {
          a!textField(
            label: "Name",
            value: local!employeeToUpdate.name,
            saveInto: local!employeeToUpdate.name
          ),
          a!textField(
            label: "Department",
            value: local!employeeToUpdate.department,
            saveInto: local!employeeToUpdate.department
          ),
          a!textField(
            label: "Title",
            value: local!employeeToUpdate.title,
            saveInto: local!employeeToUpdate.title
          ),
          a!dateField(
            label: "Start Date",
            value: local!employeeToUpdate.startDate,
            saveInto: local!employeeToUpdate.startDate
          ),
          a!buttonLayout(
            primaryButtons: a!buttonWidget(
              label: "Update",
              value: local!employeeToUpdate.id,
              saveInto: a!save(
                local!employees,
                updatearray(
                  local!employees,
                  wherecontains(save!value, local!employees.id),
                  local!employeeToUpdate
                )
              )
            )
          )
        }
      )
    ),
    buttons: a!buttonLayout(
      primaryButtons: a!buttonWidgetSubmit(label: "Submit")
    )
  )
 )
)
```

Test it out

1. Click on an employee's name in the grid's left column.
2. Change the employee's details and click the "Update" button.

Use Links in a Grid to Show More Details and Edit Data in External System

Goal: Allow end users to click a link in a read-only paging grid to view the details for the row, and make changes to the data. The changes are immediately persisted to the external system where the data resides. Sorting and paging the grid shows the latest data from the external system.

Unlike the previous grid examples, this recipe retrieves the values for the `local!employees` array from an external system via a web service. It does so by using the `bind()` function, which associates two rules or functions with a `load()` local variable: one that gets the data from the external system, and one that updates the data in the external system. When the variable is referenced, the getter rule or function is called and value is populated with the result.

The definition of the relevant `getEmployees` rule is given below. This supporting rule uses the `webservicequery` function to call a sample web service created for this recipe which returns the same employee information shown in previous examples.

```
= webservicequery(
 a!wsConfig(
  wsdlUrl: "http://localhost:8080/axis2/services/CorporateDirectoryService?wsdl",
  service: "{http://bindrecipe.services.ws.appiancorp.com}CorporateDirectoryService",
  port: "CorporateDirectoryServiceHttpSoap12Endpoint",
  operation: "{http://bindrecipe.services.ws.appiancorp.com}getEmployees"
 ),
 {
  getEmployeesRequest: {}
 }
).returnValue.getEmployeesResponse.return
```

The corresponding `setEmployees` rule uses the `webservicewrite` function. The `setEmployees` rule is never executed because the bound local variable `local!employees` is never saved into in this example. Although it isn't called as part of the example, definition of `setEmployees` is presented here for completeness.

```
= webservicewrite(
 a!wsConfig(
  wsdlUrl: "http://localhost:8080/axis2/services/CorporateDirectoryService?wsdl",
  service: "{http://bindrecipe.services.ws.appiancorp.com}CorporateDirectoryService",
  port: "CorporateDirectoryServiceHttpSoap12Endpoint",
  operation: "{http://bindrecipe.services.ws.appiancorp.com}setEmployees"
 ),
 {
  setEmployeesRequest: {
   ri!employees
  }
 }
)
```

This recipe writes the updated value of a single employee back to the external system by using another bound variable, `local!updatedEmployee`. The `local!updatedEmployee` variable is the `saveInto` target associated with the "Update" button. When the user clicks the button and the variable is saved into, the setter rule or function given as the second parameter to the `bind` function is executed. The setter rule or function must return a Writer, which is a special type of value that can be used to write data when a SAIL interface saves into a bound variable.

The definition of the relevant `setSingleEmployee` rule, which uses the `webservicewrite` function to write data to the external system through a web service call, is as follows:

```
= webservicewrite(
 a!wsConfig(
  wsdlUrl: "http://localhost:8080/axis2/services/CorporateDirectoryService?wsdl",
  service: "{http://bindrecipe.services.ws.appiancorp.com}CorporateDirectoryService",
  port: "CorporateDirectoryServiceHttpSoap12Endpoint",
  operation: "{http://bindrecipe.services.ws.appiancorp.com}setSingleEmployee"
 ),
 {
  setSingleEmployeeRequest: {
   employee: ri!employee
  }
 }
)
```

The corresponding `getSingleEmployee` reader rule is defined as follows:

```
= webservicequery(
 a!wsConfig(
  wsdlUrl: "http://localhost:8080/axis2/services/CorporateDirectoryService?wsdl",
  service: "{http://bindrecipe.services.ws.appiancorp.com}CorporateDirectoryService",
  port: "CorporateDirectoryServiceHttpSoap12Endpoint",
  operation: "{http://bindrecipe.services.ws.appiancorp.com}getSingleEmployee"
 ),
 {
  getSingleEmployeeRequest: {
   id: ri!id
  }
 }
).returnValue.getSingleEmployeeResponse.return
```

To adapt this example to work with your own web service:

1. Update the getter and setter rules defined in the `bind` functions in the example to call your web service operations that get and set the objects
2. Use the configured `webservicequery` and `webservicewrite` functions in the examples above as a guide to determine the corresponding paramete to set for your web services. Refer to the a!wsConfig() documentation to determine which values are appropriate for your web service.
3. Update the expression below to access and display the fields that are relevant for your web service. For instance, if your web service returns a

product instead of an employee, you could replace `local!employeeToUpdate.department` with `local!productToUpdate.description`, etc.

See Also: webservicequery(), webservicewrite(), bind

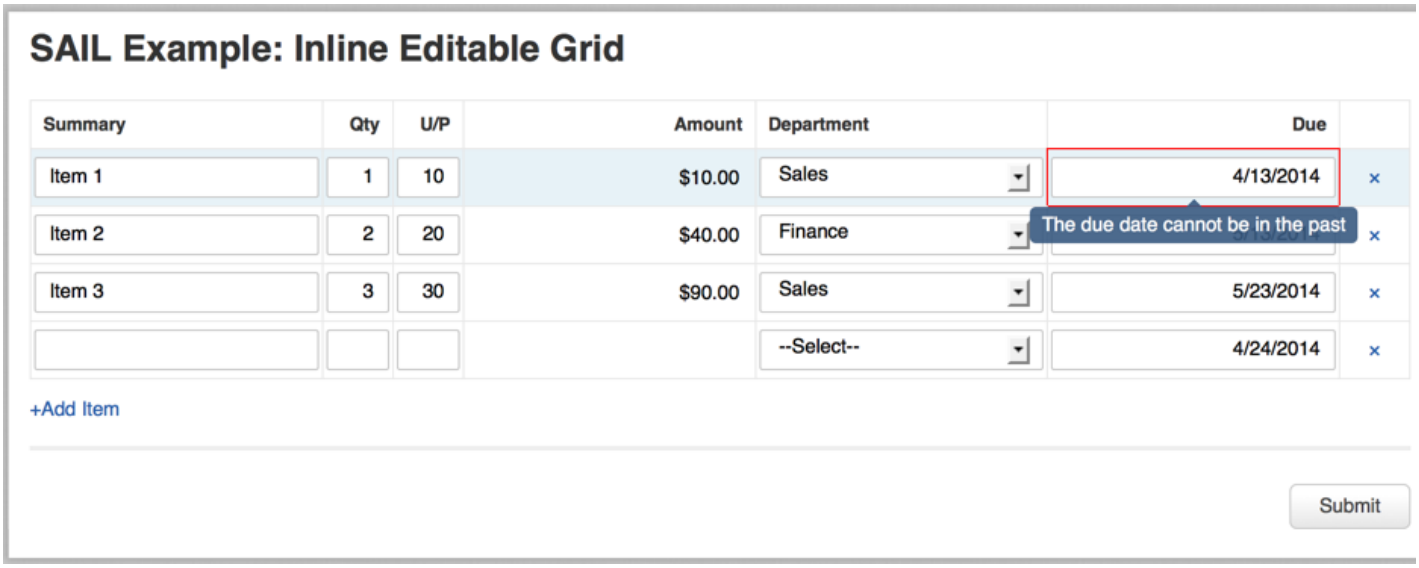
Expression:

```
=load(
 local!employees: bind(rule!getEmployees(), rule!setEmployees( _ )),
 local!pagingInfo: a!pagingInfo(startIndex: 1, batchSize: 25),
 local!employeeToUpdate,
 with(
  local!employeeDataSubset: todatasubset(local!employees, local!pagingInfo),
  local!data: index(local!employeeDataSubset, "data", null),
  a!formLayout(
   label: "SAIL Example: Grid with Link to Show More and Edit Data in an External System",
   firstColumnContents: {
    a!sectionLayout(
     label: "Employees",
     firstColumnContents: {
      a!gridField(
       totalCount: local!employeeDataSubset.totalCount,
       columns: {
        a!gridTextColumn(
         label: "Name",
         data: index(local!data, "name", {}),
         /* Creates a dynamic link for every item in local!data */
         links: apply(
          a!dynamicLink(
           value: _,
           saveInto: local!employeeToUpdate
          ),
          local!data
         )
        ),
        a!gridTextColumn(
         label: "Department",
         data: index(local!data, "department", null)
        )
       },
       value: local!pagingInfo,
       saveInto: local!pagingInfo
      )
     }
    ),
    if(isnull(local!employeeToUpdate), {},
     load(
      local!updatedEmployee: bind(
       rule!getSingleEmployee(employeeToUpdate.id),
       rule!setSingleEmployee( _ )
      ),
      a!sectionLayout(
       label: "Employee Details: " & local!employeeToUpdate.name,
       firstColumnContents: {
        a!textField(
         label: "Name",
         value: local!employeeToUpdate.name,
         saveInto: local!employeeToUpdate.name
        ),
        a!textField(
         label: "Department",
         value: local!employeeToUpdate.department,
         saveInto: local!employeeToUpdate.department
        ),
        a!textField(
         label: "Title",
         value: local!employeeToUpdate.title,
         saveInto: local!employeeToUpdate.title
        ),
        a!dateField(
         label: "Start Date",
         value: local!employeeToUpdate.startDate,
         saveInto: local!employeeToUpdate.startDate
        ),
        a!buttonLayout(
         primaryButtons: a!buttonWidget(
          label: "Update",
```

```
              value: local!employeeToUpdate,
              saveInto: local!updatedEmployee
            )
          )
        },
        buttons: a!buttonLayout(
          primaryButtons: a!buttonWidgetSubmit(
            label: "Done",
            style: "PRIMARY"
          )
        )
      )
    )
  )
 )
)
```

Add, Edit, and Remove Data in an Inline Editable Grid

Goal: Allow the user to add, edit, and remove data directly in a grid. Validate that the user's input is correct, and automatically calculate values as the user changes the data.



This scenario demonstrates:

- How to use the grid layout component to build an inline editable grid
- How to allow the user to add and remove rows in the grid
- How to validate user input in each cell
- How to mark individual cells as required
- How to configure a calculated column

The main expression uses a supporting rule, so let's create it first.

- `ucItemRowEach` : Returns one row of items to display in a grid.

Create interface `ucItemRowEach` with the following rule inputs:

- items (Any Type)
- index (Number (Integer))
- itemsToken (Any Type)

Enter the following definition for the interface:

```
=a!gridRowLayout(
 id: ri!index,
 contents: {
  a!textField(
   /* Labels in the row contents are for debugging purposes only */
   label: "summary " & ri!index,
   required: true,
   value: ri!items[ri!index].summary,
   saveInto: ri!items[ri!index].summary
  ),
  a!integerField(
   label: "qty " & ri!index,
```

```
    align: "RIGHT",
    validations: if(tointeger(ri!items[ri!index].qty) < 1, "Quantity must be greater than 0", null),
    value: ri!items[ri!index].qty,
    saveInto: ri!items[ri!index].qty
   ),
   a!floatingPointField(
    label: "unitPrice " & ri!index,
    align: "RIGHT",
    validations: if(todecimal(ri!items[ri!index].unitPrice) < 1, "Unit price must be greater than 0", null),
    value: ri!items[ri!index].unitPrice,
    saveInto: ri!items[ri!index].unitPrice
   ),
   a!textField(
    label: "amount " & ri!index,
    align: "RIGHT",
    readOnly: true,
    value: if(or(isnull(ri!items[ri!index].qty), isnull(ri!items[ri!index].unitPrice)),
            null,
            dollar(tointeger(ri!items[ri!index].qty) * todecimal(ri!items[ri!index].unitPrice))
           )
   ),
   a!dropdownField(
    label: "dept " & ri!index,
    choiceLabels: {"Finance", "Sales"},
    choiceValues: {"Finance", "Sales"},
    placeholderLabel: "--Select-- ",
    value: ri!items[ri!index].dept,
    saveInto: ri!items[ri!index].dept
   ),
   a!dateField(
    label: "due " & ri!index,
    align: "RIGHT",
    validations: if(todate(ri!items[ri!index].due) < today(), "The due date cannot be in the past", null),
    value: ri!items[ri!index].due,
    saveInto: ri!items[ri!index].due
   ),
   a!linkField(
    label: "delete " & ri!index,
    align: "CENTER",
    links: a!dynamicLink(
     label: char(10005),
     value: ri!index,
     saveInto: {
      a!save(ri!items, remove(ri!items, save!value)),
      /*
       * When modifying the size of the array used in a!applyComponents,
       * make the same change in the "token" array variable
       */
      a!save(ri!itemsToken, remove(ri!itemsToken, save!value))
     }
    )
   )
  }
)
```

Now that we've created the supporting rule, let's move on to the main expression.

Expression

```
=load(
 local!items: {
  {id: null, summary: "Item 1", qty: 1, unitPrice: 10, dept: "Sales",   due: today() - 10},
  {id: null, summary: "Item 2", qty: 2, unitPrice: 20, dept: "Finance", due: today() + 20},
  {id: null, summary: "Item 3", qty: 3, unitPrice: 30, dept: "Sales",   due: today() + 30}
 },
 /* Needed when adding or removing items via a!applyComponents */
 local!itemsToken,
 a!formLayout(
  label: "SAIL Example: Inline Editable Grid",
  firstColumnContents: {
   a!gridLayout(
    headerCells: {
     a!gridLayoutHeaderCell(label: "Summary"),
     a!gridLayoutHeaderCell(label: "Qty", align: "RIGHT"),
     a!gridLayoutHeaderCell(label: "U/P", align: "RIGHT"),
     a!gridLayoutHeaderCell(label: "Amount", align: "RIGHT"),
```

```
        a!gridLayoutHeaderCell(label: "Department"),
        a!gridLayoutHeaderCell(label: "Due", align: "RIGHT"),
        /* For the "Remove" column */
        a!gridLayoutHeaderCell(label: "")
       },
       /* Only needed when some columns need to be narrow */
       columnConfigs: {
        a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
        a!gridLayoutColumnConfig(width: "NARROW"),
        a!gridLayoutColumnConfig(width: "NARROW"),
        a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
        a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
        a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
        a!gridLayoutColumnConfig(width: "NARROW")
       },
       rows: a!applyComponents(
        function: rule!ucItemRowEach(
         items: local!items,
         index: _,
         itemsToken: local!itemsToken
        ),
        array: if(or(isnull(local!items), count(local!items) < 1), {}, 1+enumerate(count(local!items))),
        arrayVariable: local!itemsToken
       )
      ),
      a!linkField(
       label: "Add Link",
       labelPosition: "COLLAPSED",
       links: a!dynamicLink(
        label: "+Add Item",
        /*
         * For your use case, set the value to a blank instance of your CDT using
         * the type constructor, e.g. type!PurchaseRequestItem(). Only specify the field
         * if you want to give it a default value e.g. due: today()+1.
         */
        value: {due: today() + 1},
        saveInto: {
         a!save(local!items, append(local!items, save!value)),
         /*
          * When modifying the size of the array used in a!applyComponents,
          * make the same change in the "token" array variable
          */
         a!save(local!itemsToken, append(local!itemsToken, save!value))
        }
       )
      )
     },
     buttons: a!buttonLayout(primaryButtons: a!buttonWidgetSubmit(label: "Submit"))
    )
)
```

Test it out

1. Change a quantity cell and notice that the corresponding amount is updated.
2. Change the invalid due date to a current or future date and notice that the validation message is removed.
3. Attempt to enter a negative quantity or unit price and notice that a validation message is displayed.
4. Clear the value of a summary cell (which is a required cell) and attempt to submit.

Notable implementation details

- Notice that the `ucItemRowEach` rule has access to the items across all rows, as well as all the fields of the item within its own row. This is how you' able to dynamically calculate values across columns, and if necessary, across rows.
- The component in each cell has a label, but the label isn't displayed. Labels and instructions aren't rendered within a grid cell. It is useful to enter a label for expression readability. It also helps identify which cell has an expression error since the label is displayed in the error message.
- The index of the item in the array is used rather than the id of the CDT to make saving back into the main items array easier, i.e. `saveInto: ri!index[index].fieldName`. This works when the entire array of data is available in the form, and the data is not being updated outside th form.
- The conversion to `tointeger()`, `todate()`, and `todecimal()` in various places are not necessary when working with custom data types. They are needed here because the example uses an ad-hoc dictionary structure.
- To keep track of which items are removed so that you can execute the Delete from Data Store Entity smart service for the removed items, see the Track Adds and Deletes in an Inline Editable Grid recipe.

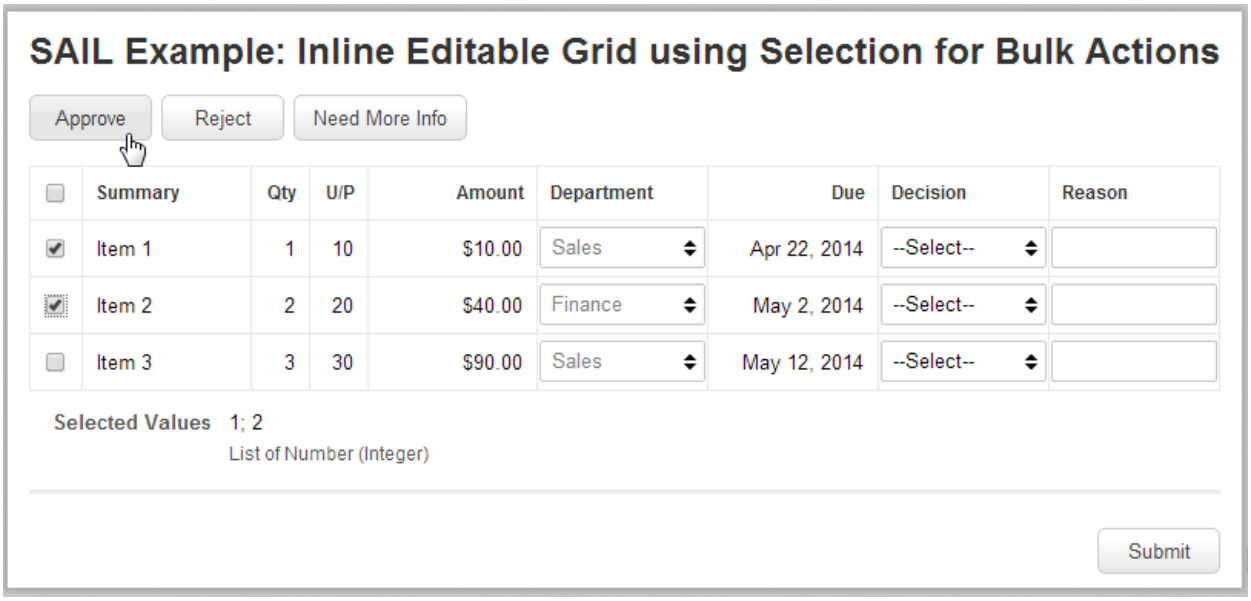
To write your data to process

1. Save your interface as sailRecipe
2. Create your custom data type for items with the same fields as in the example

3. Create interface input: items (Any Type)
4. Delete local variable: `local!items`
5. In your expression, replace:
   - `local!items` with `ri!items`
6. In your process model, create a variable called items that is of the same type as the CDT array with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(items: ac!items)`
   - On a start form: `=rule!sailRecipe(items: pv!items)`

Use Selection For Bulk Actions in an Inline Editable Grid

Goal: Allow the user to edit data inline in a grid one field at a time, or in bulk using selection.



This scenario demonstrates:

- How to use the grid layout component to build an inline editable grid
- How to use selection to enable bulk actions
- How to make a cell conditionally required based on the value in another cell

Let's start by creating the supporting rule:

- `ucItemRowEach` : Returns one row of items to display in a grid.

Create interface `ucItemRowEach` with the following rule inputs:

- items (Any Type)
- index (Number (Integer))

Enter the following definition for the interface:

```
=a!gridRowLayout(
 id: ri!index,
 contents: {
  a!textField(
   /* Labels in the row contents are for debugging purposes only */
   label: "summary " & ri!index,
   readOnly: true,
   value: ri!items[ri!index].summary
  ),
  a!integerField(
   label: "qty " & ri!index,
   align: "RIGHT",
   readOnly: true,
   value: ri!items[ri!index].qty
  ),
  a!floatingPointField(
   label: "unitPrice " & ri!index,
   align: "RIGHT",
   readOnly: true,
   value: ri!items[ri!index].unitPrice
  ),
  a!textField(
   label: "amount " & ri!index,
```

```
    align: "RIGHT",
    readOnly: true,
    value: if(or(isnull(ri!items[ri!index].qty), isnull(ri!items[ri!index].unitPrice)),
            null,
            dollar(ri!items[ri!index].qty * ri!items[ri!index].unitPrice)
           )
   ),
   a!dropdownField(
    label: "dept " & ri!index,
    choiceLabels: {"Finance", "Sales"},
    choiceValues: {"Finance", "Sales"},
    placeholderLabel: "--Select-- ",
    disabled: true,
    value: ri!items[ri!index].dept
   ),
   a!dateField(
    label: "due " & ri!index,
    align: "RIGHT",
    readOnly: true,
    value: ri!items[ri!index].due
   ),
   a!dropdownField(
    label: "decision " & ri!index,
    choiceLabels: {"Approve", "Reject", "Need More Info"},
    choiceValues: {"Approve", "Reject", "Need More Info"},
    placeholderLabel: "--Select-- ",
    required: true,
    value: ri!items[ri!index].decision,
    saveInto: ri!items[ri!index].decision
   ),
   a!textField(
    label: "reason" & ri!index,
    required: and(not(isnull(ri!items[ri!index].decision)), ri!items[ri!index].decision <> "Approve"),
    requiredMessage: "A reason is required for items that are not approved",
    value: ri!items[ri!index].reason,
    saveInto: ri!items[ri!index].reason
   )
  }
)
```

This example makes use of a custom data type. Create a `PrItem` custom data type with the following fields:

- id (Number (Integer))
- summary (Text)
- qty (Number (Integer))
- unitPrice (Number (Decimal))
- dept (Text)
- due (Date)
- decision (Text)
- reason (Text)

Note: We use Text as the data type for `dept` and `decision` for simplicity. Typically, these fields would reference a lookup table.

Now that we've created the supporting rules and data type, let's move on to the main expression.

Expression

```
=load(
 local!items: {
  type!PrItem(id: 1, summary: "Item 1", qty: 1, unitPrice: 10, dept: "Sales",   due: today() + 10),
  type!PrItem(id: 2, summary: "Item 2", qty: 2, unitPrice: 20, dept: "Finance", due: today() + 20),
  type!PrItem(id: 3, summary: "Item 3", qty: 3, unitPrice: 30, dept: "Sales",   due: today() + 30)
 },
 local!selectedIndices: tointeger({}),
 a!formLayout(
  label: "SAIL Example: Inline Editable Grid using Selection for Bulk Actions",
  firstColumnContents: {
   a!buttonArrayLayout(buttons: {
    a!buttonWidget(
     label: "Approve",
     disabled: count(local!selectedIndices) = 0,
     value: "Approve",
     /* You can save into a field at many indices at a time */
     saveInto: {
      local!items[local!selectedIndices].decision,
      /* Clear the selected indices after a decision is made */
```

```
          a!save(local!selectedIndices, tointeger({}))
        }
      ),
      a!buttonWidget(
        label: "Reject",
        disabled: count(local!selectedIndices) = 0,
        value: "Reject",
        saveInto: {
          local!items[local!selectedIndices].decision,
          /* Clear the selected indices after a decision is made */
          a!save(local!selectedIndices, tointeger({}))
        }
      ),
      a!buttonWidget(
        label: "Need More Info",
        disabled: count(local!selectedIndices) = 0,
        value: "Need More Info",
        saveInto: {
          local!items[local!selectedIndices].decision,
          /* Clear the selected indices after a decision is made */
          a!save(local!selectedIndices, tointeger({}))
        }
      )
    }),
    a!gridLayout(
      selectable: true,
      selectionValue: local!selectedIndices,
      selectionSaveInto: local!selectedIndices,
      headerCells: {
        a!gridLayoutHeaderCell(label: "Summary"),
        a!gridLayoutHeaderCell(label: "Qty", align: "RIGHT"),
        a!gridLayoutHeaderCell(label: "U/P", align: "RIGHT"),
        a!gridLayoutHeaderCell(label: "Amount", align: "RIGHT"),
        a!gridLayoutHeaderCell(label: "Department"),
        a!gridLayoutHeaderCell(label: "Due", align: "RIGHT"),
        a!gridLayoutHeaderCell(label: "Decision"),
        a!gridLayoutHeaderCell(label: "Reason")
      },
      /* Only needed when some columns need to be narrow */
      columnConfigs: {
        a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
        a!gridLayoutColumnConfig(width: "NARROW"),
        a!gridLayoutColumnConfig(width: "NARROW")
      },
      rows: a!applyComponents(
        function: rule!ucItemRowEach(
          items: local!items,
          index: _
        ),
        array: if(or(isnull(local!items), count(local!items) < 1), {}, 1+enumerate(count(local!items)))
      )
    ),
    a!textField(
      label: "Selected Values",
      instructions: typename(typeof(local!selectedIndices)),
      labelPosition: "ADJACENT",
      readOnly: true,
      value: local!selectedIndices
    )
  },
  buttons: a!buttonLayout(primaryButtons: a!buttonWidgetSubmit(label: "Submit"))
 )
)
```

Test it out

1. Select a decision inline in the grid.
2. Select a couple rows using the selection checkboxes and notice that the buttons are enabled above the grid. Click on a button and notice that th decision field is updated for the selected rows.
3. Select "Reject" or "Need More Info", leave the corresponding reason blank, and attempt to submit to see the required message.

Notable implementation details

- You can save into a field in an array of custom data type values directly without having to use looping functions or rules i.e. `saveInto: ri!cdtArray[{1,2,3}].fieldName`. Use this design pattern when the form has access to the entire array of data, and the data is not being modified outside the form. Otherwise, create a rule to find the items to update by the item id.

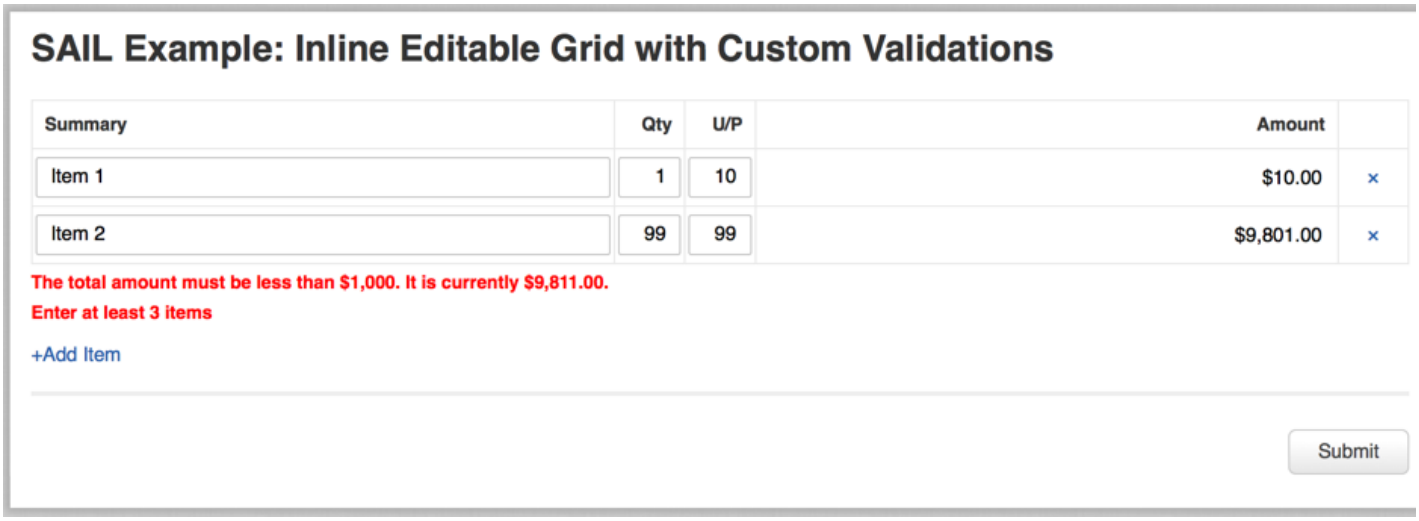
- Notice that the `ucItemRowEach` rule has access to the items across all rows, as well as all the fields of the item within its own row. This is how you' able to dynamically calculate values across columns, and if necessary, across rows.
- The component in each cell has a label, but the label isn't displayed. Labels and instructions aren't rendered within a grid cell. It is useful to enter a label for expression readability. It also helps identify which cell has an expression error since the label is displayed in the error message.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: items (Any Type)
3. Delete local variable: `local!items`
4. In your expression, replace:
   - `local!items` with `ri!items`
5. In your process model, create a variable called items that is of the same type as the CDT array with no value
   - On a task form, create node inputs
   - On a start form, create process parameters
6. In your process model, enter your SAIL Form definition:
   - On a task form: `=rule!sailRecipe(items: ac!items)`
   - On a start form: `=rule!sailRecipe(items: pv!items)`

Add Validation to an Inline Editable Grid

Goal: Allow the user to add, edit, and remove data directly in a grid. Validate that the user adds at least three rows, and that the total amount is less tha $1,000.



This scenario demonstrates:

- How to configure validation messages that only shows up when a submit button is pressed.
- How to configure validation messages that as soon as a condition is satisfied.

The main expression uses a supporting rule, so let's create it first.

- `ucItemRowEach` : Returns one row of items to display in a grid.

Create interface `rule!ucItemRowEach` with the following rule inputs:

- items (Any Type)
- index (Number (Integer))
- itemsToken (Any Type)

Enter the following definition for the interface:

```
=a!gridRowLayout(
 id: ri!index,
 contents: {
  a!textField(
   /* Labels in the row contents are for debugging purposes only */
   label: "summary " & ri!index,
   required: true,
   value: ri!items[ri!index].summary,
   saveInto: ri!items[ri!index].summary
  ),
  a!integerField(
   label: "qty " & ri!index,
   align: "RIGHT",
   validations: if(tointeger(ri!items[ri!index].qty) < 1, "Quantity must be greater than 0", null),
   value: ri!items[ri!index].qty,
   saveInto: ri!items[ri!index].qty
  ),
```

```
    a!floatingPointField(
      label: "unitPrice " & ri!index,
      align: "RIGHT",
      validations: if(todecimal(ri!items[ri!index].unitPrice) < 1, "Unit price must be greater than 0", null),
      value: ri!items[ri!index].unitPrice,
      saveInto: ri!items[ri!index].unitPrice
    ),
    a!textField(
      label: "amount " & ri!index,
      align: "RIGHT",
      readOnly: true,
      value: if(or(isnull(ri!items[ri!index].qty), isnull(ri!items[ri!index].unitPrice)),
              null,
              dollar(tointeger(ri!items[ri!index].qty) * todecimal(ri!items[ri!index].unitPrice))
             )
    ),
    a!linkField(
      label: "delete " & ri!index,
      align: "CENTER",
      links: a!dynamicLink(
        label: char(10005),
        value: ri!index,
        saveInto: {
          a!save(ri!items, remove(ri!items, save!value)),
          /*
           * When modifying the size of the array used in a!applyComponents,
           * make the same change in the "token" array variable
           */
          a!save(ri!itemsToken, remove(ri!itemsToken, save!value))
        }
      )
    )
  }
)
```

Now that we've created the supporting rule, let's move on to the main expression.

Expression

```
=load(
  local!items: {
    {id: null, summary: "Item 1", qty: 1, unitPrice: 10},
    {id: null, summary: "Item 2", qty: 2, unitPrice: 99}
  },
  /* Needed when adding or removing items via a!applyComponents */
  local!itemsToken,
  a!formLayout(
    label: "SAIL Example: Inline Editable Grid with Custom Validations",
    firstColumnContents: {
      a!gridLayout(
        validations: if(or(isnull(local!items), count(local!items) < 1), null,
          {
            /* Check the total amount and show a message immediately */
            with(
              local!total: sum(tointeger(local!items.qty)*todecimal(local!items.unitPrice)),
              if(local!total &amp;lt;= 1000,
                null,
                "The total amount must be less than $1,000. It is currently " & dollar(local!total) & "."
              )
            ),
            /* Check the # items and show a message on submit */
            if(count(local!items) > 3,
              null,
              a!validationMessage(validateAfter: "SUBMIT", message: "Enter at least 3 items")
            )
          }
        ),
        headerCells: {
          a!gridLayoutHeaderCell(label: "Summary"),
          a!gridLayoutHeaderCell(label: "Qty", align: "RIGHT"),
          a!gridLayoutHeaderCell(label: "U/P", align: "RIGHT"),
          a!gridLayoutHeaderCell(label: "Amount", align: "RIGHT"),
          /* For the "Remove" column */
          a!gridLayoutHeaderCell(label: "")
        },
        /* Only needed when some columns need to be narrow */
```

```
      columnConfigs: {
       a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
       a!gridLayoutColumnConfig(width: "NARROW"),
       a!gridLayoutColumnConfig(width: "NARROW"),
       a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
       a!gridLayoutColumnConfig(width: "NARROW")
      },
      rows: a!applyComponents(
       function: rule!ucItemRowEach(
        items: local!items,
        index: _,
        itemsToken: local!itemsToken
       ),
       array: if(or(isnull(local!items), count(local!items) < 1), {}, 1+enumerate(count(local!items))),
       arrayVariable: local!itemsToken
      )
     ),
     a!linkField(
      label: "Add Link",
      labelPosition: "COLLAPSED",
      links: a!dynamicLink(
       label: "+Add Item",
       /*
        * For your use case, set the value to a blank instance of your CDT using
        * the type constructor, e.g. type!PurchaseRequestItem(). Only specify the field
        * if you want to give it a default value e.g. due: today()+1.
        */
       value: {id: null},
       saveInto: {
        a!save(local!items, append(local!items, save!value)),
        /*
         * When modifying the size of the array used in a!applyComponents,
         * make the same change in the "token" array variable
         */
        a!save(local!itemsToken, append(local!itemsToken, save!value))
       }
      )
     )
    },
    buttons: a!buttonLayout(primaryButtons: a!buttonWidgetSubmit(label: "Submit"))
   )
)
```

Test it out

1. Change the quantity to 99 and the unit price to 99. Notice that the validation message shows up when you move away from the field.
2. Presss the Submit button. Notice that the second validation message only shows up when attempting to submit.

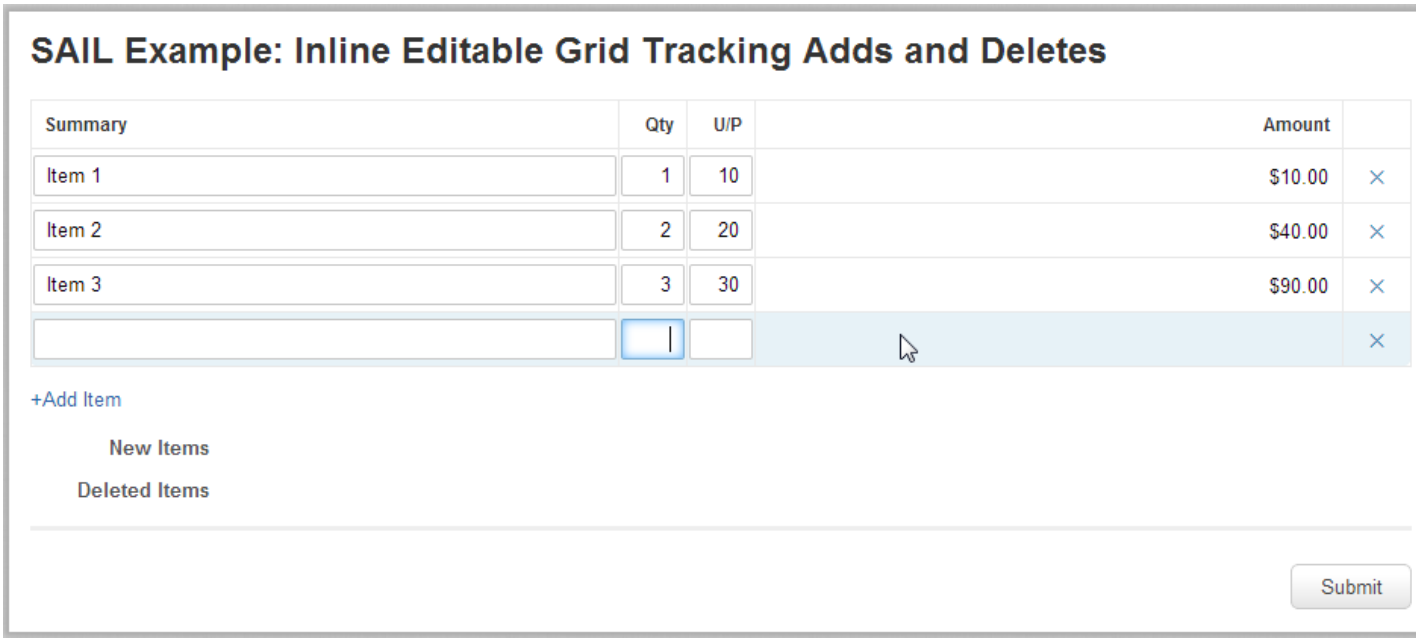
Notable implementation details

- The array of validations can contain either text or a validation message created with `a!validationMessage()`. Use the latter when you want the validation message to show up when the user clicks a submit button, or a validating button.

To write your data to process

1. Save your interface as sailRecipe
2. Create your custom data type for items with the same fields as in the example
3. Create interface input: items (Any Type)
4. Delete local variable: `local!items`
5. In your expression, replace:
    - `local!items` with `ri!items`
6. In your process model, create a variable called items that is of the same type as the CDT array with no value
    - On a task form, create node inputs
    - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
    - On a task form: `=rule!sailRecipe(items: ac!items)`
    - On a start form: `=rule!sailRecipe(items: pv!items)`

Track Adds and Deletes in Inline Editable Grid



Goal: In an inline editable grid, track the items that are added for further processing in the next process steps. Also track the ids of items that are delete for use in the Delete from Data Store Entities smart service after the user submits the form.

See also: Delete from Data Store Entities Smart Service

This scenario demonstrates:

- How to store data into multiple variables when the user interacts with the components in a grid layout.

The main expression uses a supporting rule, so let's create it first.

- `ucItemRowEach` : Returns one row of items to display in a grid.

Create interface `rule!ucItemRowEach` with the following rule inputs:

- items (Any Type)
- index (Number (Integer))
- itemsToken (Any Type)
- deletedItemIds (Number (Integer) Array)

Enter the following definition for the interface:

```
=a!gridRowLayout(
 id: ri!index,
 contents: {
  a!textField(
   /* Labels in the row contents are for debugging purposes only */
   label: "summary " & ri!index,
   value: ri!items[ri!index].summary,
   saveInto: ri!items[ri!index].summary
  ),
  a!integerField(
   label: "qty " & ri!index,
   align: "RIGHT",
   value: ri!items[ri!index].qty,
   saveInto: ri!items[ri!index].qty
  ),
  a!floatingPointField(
   label: "unitPrice " & ri!index,
   align: "RIGHT",
   value: ri!items[ri!index].unitPrice,
   saveInto: ri!items[ri!index].unitPrice
  ),
  a!textField(
   label: "amount " & ri!index,
   align: "RIGHT",
   readOnly: true,
   value: if(or(isnull(ri!items[ri!index].qty), isnull(ri!items[ri!index].unitPrice)),
           null,
           dollar(tointeger(ri!items[ri!index].qty) * todecimal(ri!items[ri!index].unitPrice))
          )
  ),
  a!linkField(
   label: "delete " & ri!index,
   align: "CENTER",
```

```
    links: a!dynamicLink(
      label: char(10005),
      value: ri!index,
      saveInto: {
        /*
         * Capture the deleted item's id before removing it from the ri!items array.
         * No need to filter out null ids from new items since the Delete from Data Store
         * Entities smart service ignores null identifiers.
         */
        a!save(ri!deletedItemIds, append(ri!deletedItemIds, index(ri!items.id, save!value, null)))
        a!save(ri!items, remove(ri!items, save!value)),
        /*
         * When modifying the size of the array used in a!applyComponents,
         * make the same change in the "token" array variable
         */
        a!save(ri!itemsToken, remove(ri!itemsToken, save!value))
      }
    )
  )
 }
)
```

Now that we've created the supporting rule, let's move on to the main expression.

Expression

```
=load(
 local!items: {
  {id: 1, summary: "Item 1", qty: 1, unitPrice: 10},
  {id: 2, summary: "Item 2", qty: 2, unitPrice: 20},
  {id: 3, summary: "Item 3", qty: 3, unitPrice: 30}
 },
 /* Needed when adding or removing items via a!applyComponents */
 local!itemsToken,
 local!newItems,
 local!deletedItemIds,
 a!formLayout(
  label: "SAIL Example: Inline Editable Grid Tracking Adds and Deletes",
  firstColumnContents: {
   a!gridLayout(
    headerCells: {
     a!gridLayoutHeaderCell(label: "Summary"),
     a!gridLayoutHeaderCell(label: "Qty", align: "RIGHT"),
     a!gridLayoutHeaderCell(label: "U/P", align: "RIGHT"),
     a!gridLayoutHeaderCell(label: "Amount", align: "RIGHT"),
     a!gridLayoutHeaderCell(label: "") /*For the "Remove" column*/
    },
    /* Only needed when some columns need to be narrow */
    columnConfigs: {
     a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
     a!gridLayoutColumnConfig(width: "NARROW"),
     a!gridLayoutColumnConfig(width: "NARROW"),
     a!gridLayoutColumnConfig(width: "DISTRIBUTE"),
     a!gridLayoutColumnConfig(width: "NARROW")
    },
    rows: a!applyComponents(
     function: rule!ucItemRowEach(
      items: local!items,
      index: _,
      itemsToken: local!itemsToken,
      deletedItemIds: local!deletedItemIds
     ),
     array: if(or(isnull(local!items), count(local!items) < 1), {}, 1+enumerate(count(local!items))),
     arrayVariable: local!itemsToken
    )
   ),
   a!linkField(
    label: "Add Link",
    labelPosition: "COLLAPSED",
    links: a!dynamicLink(
     label: "+Add Item",
     /*
      * For your use case, set the value to a blank instance of your CDT using
      * the type constructor, e.g. type!PurchaseRequestItem(). Only specify the field
      * if you want to give it a default value e.g. due: today()+1.
```

```
        */
        value: {id: null},
        saveInto: {
          a!save(local!items, append(local!items, save!value)),
          /*
           * When modifying the size of the array used in a!applyComponents,
           * make the same change in the "token" array variable
           */
          a!save(local!itemsToken, append(local!itemsToken, save!value))
        }
      )
    ),
    a!textField(
      label: "New Items",
      readOnly: true,
      value: local!newItems
    ),
    a!textField(
      label: "Deleted Items",
      readOnly: true,
      value: local!deletedItemIds
    )
  },
  buttons: a!buttonLayout(
    /*
     * Using a!buttonWidget() for the updated value of local!newItems to be displayed
     * in the UI for testing purposes. Use a!buttonWidgetSubmit() in your form.
     */
    primaryButtons: a!buttonWidget(
      label: "Submit",
      /* This null should have the same type as your id field */
      value: tointeger(null),
      /*
       * The tointeger() conversion in wherecontains() is not needed when
       * you swap the value of local!items with your CDT array
       */
      saveInto: a!save(
        local!newItems,
        index(local!items, wherecontains(save!value, tointeger(local!items.id)), null)
      )
    )
  )
 )
)
```

Test it out

1. Remove a row from the pre-loaded data set by clicking the "X" link. Notice that the item id is added to the array of deleted item ids.
2. Add a row, enter values into the blank fields, and click the "Submit" button. Notice that the new item is added to the array of added items.

Notable implementation details

- The array of added items is captured when the submit button is clicked. This allows the expression to be evaluated once and works when you on need the new items array to be submitted to process. If you need the new items right away in your form, save the values as the user clicks "Add Item", but also remove the appropriate item when the user removes the newly added item.
- If you intend to immediately write the new and edited items using the Write to Data Store Entities smart service, you don't need to capture the array of added items separately from your items array. This is because the Write to Data Store Entity smart service can do updates and inserts a the same time. See also: Write to Data Store Entity Smart Service
- The array of deleted item ids may contain null values corresponding to newly added items. You don't have to remove the nulls if you are planning on passing the ids to the Delete from Data Store Entities smart service. This is because the smart service ignores the null values. See also: Delete from Data Store Entities Smart Service

To write your data to process

1. Save your interface as sailRecipe
2. Create your custom data type for items with the same fields as in the example
3. Create interface inputs: items (Any Type), newItems (Any Type), deletedItemIds (Integer (Number) array)
4. Delete local variable: `local!items`, `local!newItems`, `local!deletedItemIds`
5. In your expression, replace:
   - `local!items` with `ri!items`
   - `local!newItems` with `ri!newItems`
   - `local!deletedItemIds` with `ri!deletedItemIds`
6. In your process model, create variables `items` and `newItems` for your custom data type), and `deletedItemIds` (Integer (Number) Array)
   - On a task form, create node inputs
   - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:

- On a task form: `=rule!sailRecipe(items: ac!items, newItems: ac!newItems, deletedItemIds: ac!deletedItemIds)`
- On a start form: `=rule!sailRecipe(items: pv!items, newItems: pv!newItems, deletedItemIds: pv!deletedItemIds)`

Build a Wizard in SAIL

Goal: Divide a big form into sections presented one step at a time. All the fields must be valid before the user is allowed to move to the next steps. However, the user is allowed to move to a previous step even if the fields in the current step aren't valid. The last step in the wizard is a confirmation screen.

Note: SAIL should be used to break up a large form into smaller steps rather than activity-chaining. Users can step back and forth within a SAIL wizard without losing data in between steps.

In the first example, we save the entire expression as one rule so that you can more easily read it and follow the logic of how the buttons are configured to go forward and back. In the second example, we show you how to break up the expression into multiple rules so that we can test each step independently, and keep the main rule easier to read and maintain. This will make it easier for you to add more fields to each step of the wizard.

Expression 1:

```
=load(
 local!cdt: type!DataSubset(),
 local!currentStep: 1,
 choose(
  local!currentStep,
  a!formLayout(
   label: "SAIL Example: Wizard Step 1",
   firstColumnContents: {
    a!integerField(
     label: "Start Index",
     required: true,
     value: local!cdt.startIndex,
     saveInto: local!cdt.startIndex
    )
   },
   buttons: a!buttonLayout(
    primaryButtons: {
     a!buttonWidget(
      label: "Next",
      style: "PRIMARY",
      validate: true,
      value: 2,
      saveInto: local!currentStep
     )
    }
   )
  ),
  a!formLayout(
   label: "SAIL Example: Wizard Step 2",
   firstColumnContents: {
    a!integerField(
     label: "Batch Size",
     required: true,
     value: local!cdt.batchSize,
     saveInto: local!cdt.batchSize
    )
   },
   buttons: a!buttonLayout(
    primaryButtons: {
     a!buttonWidget(
      label: "Next",
      style: "PRIMARY",
      validate: true,
      value: 3,
      saveInto: local!currentStep
     )
    },
    secondaryButtons: {
     a!buttonWidget(
      label: "Previous",
      value: 1,
      saveInto: local!currentStep
     )
    }
   )
  ),
  a!formLayout(
   label: "SAIL Example: Wizard Confirmation",
```

```
    firstColumnContents: {
     a!integerField(
      label: "Start Index",
      readOnly: true,
      value: local!cdt.startIndex,
      saveInto: local!cdt.startIndex
     ),
     a!integerField(
      label: "Batch Size",
      readOnly: true,
      value: local!cdt.batchSize,
      saveInto: local!cdt.batchSize
     )
    },
    buttons: a!buttonLayout(
     primaryButtons: {
      a!buttonWidgetSubmit(
       label: "Submit",
       style: "PRIMARY"
      )
     },
     secondaryButtons: {
      a!buttonWidget(
       label: "Previous",
       value: 2,
       saveInto: local!currentStep
      )
     }
    )
   )
  )
)
```

Test it out

1. Click "Next" without entering a value for *Start Index*. The user stays on step 1 until a valid integer is entered.
2. Enter a valid number in step 1, then click "Next" to go to step 2. Enter a valid number, then click "Previous". Click "Next" again, and notice that the number on step 2 is preserved.
    - If you use activity-chaining, the user's input on step 2 would be lost at this point, which is why Appian recommends using SAIL when designing interface wizards.
3. Navigate back and forth through the wizard to understand its mechanics.

Notable implementation details

- We use the `choose()` function to show/hide the wizard steps instead of nested `if()` functions. The `choose()` function only evaluates the expression at the index given as its first parameter.
- Each "Next" button is configured with `validation: true`. This ensures that the fields on the current step must be valid before `local!currentStep` is updated to the next step index.
- Each "Previous" button is not configured to enforce validation. This allows the user to go to a previous step even if the current step has null required fields and other invalid fields.

Now that you understand how the simple case works, let's break up the large expression into more manageable chunks to make it easier to add more fields to each step. We'll create a rule that is responsible for the layout and buttons for every step. We then create one rule for the fields of each step.

Let's start by creating the supporting rules.

- `ucWizardStep` : Configures a form layout and the buttons to be used for displaying every step of the wizard.
- `ucWizardFieldsFirst` : Returns the contents of the first step in the wizard.
- `ucWizardFieldsSecond` : Returns the contents of the first step in the wizard.

Create interface `ucWizardStep` with the following rule inputs:

- stepLabel (Text)
- currentStepVariable (Number Integer)
- numSteps (Number Integer)
- stepContents (Any Type)

Enter the following definition for the interface:

```
=a!formLayout(
 label: ri!stepLabel,
 firstColumnContents: ri!stepContents,
 buttons: a!buttonLayout(
  primaryButtons: a!buttonWidget(
   /* On the last step, show "Submit", and configure the button to submit */
   label: if(ri!currentStepVariable=ri!numSteps, "Submit", "Next"),
   submit: ri!currentStepVariable=ri!numSteps,
   style: "PRIMARY",
```

```
    validate: true,
    value: ri!currentStepVariable+1,
    saveInto: ri!currentStepVariable
   ),
   secondaryButtons: {
    if(ri!currentStepVariable=1, {},
     a!buttonWidget(
      label: "Previous",
      value: ri!currentStepVariable-1,
      saveInto: ri!currentStepVariable
     )
    )
   }
  )
)
```

Now create interface `ucWizardFieldsFirst` with the following rule inputs:

- cdt (Any Type)
- readOnly (Boolean)

Enter the following definition for the interface:

```
={
 a!integerField(
  label: "Start Index",
  readOnly: ri!readOnly,
  required: if(ri!readOnly, false, true),
  value: ri!cdt.startIndex,
  saveInto: ri!cdt.startIndex
 )
}
```

Now create interface `ucWizardFieldsSecond` with the following rule inputs:

- cdt (Any Type)
- readOnly (Boolean)

Enter the following definition for the interface:

```
={
 a!integerField(
  label: "Batch Size",
  readOnly: ri!readOnly,
  required: if(ri!readOnly, false, true),
  value: ri!cdt.batchSize,
  saveInto: ri!cdt.batchSize
 )
}
```

Now that we've created all the supporting rules, let's move on to the main expression:

Expression 2:

```
=load(
 local!cdt: type!DataSubset(),
 local!currentStep: 1,
 choose(
  local!currentStep,
  /* step 1 */
  rule!ucWizardStep(
   stepLabel: "SAIL Example: Wizard Step 1",
   currentStepVariable: local!currentStep,
   numSteps: 3,
   stepContents: rule!ucWizardFieldsFirst(cdt: local!cdt)
  ),
  /* step 2 */
  rule!ucWizardStep(
   stepLabel: "SAIL Example: Wizard Step 2",
   currentStepVariable: local!currentStep,
   numSteps: 3,
   stepContents: rule!ucWizardFieldsSecond(cdt: local!cdt)
  ),
  /* confirmation step */
  rule!ucWizardStep(
   stepLabel: "SAIL Example: Wizard Confirmation",
```

```
    currentStepVariable: local!currentStep,
    numSteps: 3,
    stepContents: {
      rule!ucWizardFieldsFirst(cdt: local!cdt, readOnly: true),
      rule!ucWizardFieldsSecond(cdt: local!cdt, readOnly: true)
    }
  )
 )
)
```

Test it out

1. The behavior should be functionally the same as for expression 1.

The way that the expression is set up makes it much easier to add more fields to each step and to read the overall flow of the steps in the main expression. The fields in the wizard steps are reused in the confirmation step, with a flag to show each field in read-only mode.

To write your data to process

1. Save your interface as sailRecipe
2. Create interface input: cdt (Any Type)
3. Remove the `load()` function
4. Delete local variables: `local!cdt`
5. In your expression, replace:
    - `local!cdt` with `ri!cdt`
6. In your process model, create variables: cdt (Datasubset) with no value
    - On a task form, create node inputs
    - On a start form, create process parameters
7. In your process model, enter your SAIL Form definition:
    - On a task form: `=rule!sailRecipe(cdt: ac!cdt)`
    - On a start form: `=rule!sailRecipe(cdt: pv!cdt)`

Configure an Array Picker

Goal: Allow users to choose from a long text array using an autocompleting picker. Also, allow users to work with user-friendly long labels but submit machine-friendly abbreviations.

The main expression uses two supporting rules, so let's create them first.

- `ucArrayPickerFilter` : Scans labels that match the text entered by the user and returns a DataSubset for use in the SAIL picker component.
- `ucArrayPickerGetLabelForIdentifier` : For an identifier, finds its index in the identifiers array and then returns the corresponding label.

Create expression rule `ucArrayPickerFilter` with the following rule inputs:

- filter (Text)
- labels (Text Array)
- identifiers (Text Array)

Enter the following definition for the rule:

```
=with(
 local!matches: where(apply(search(ri!filter, _), ri!labels)),
 type!DataSubset(data: index(ri!labels, local!matches), identifiers: index(ri!identifiers, local!matches))
)
```

Create expression rule `ucArrayPickerGetLabelForIdentifier` with the following rule inputs:

- identifier (Text)
- labels (Text Array)
- identifiers (Text Array)

Enter the following definition for the rule:

```
=index(ri!labels, wherecontains(ri!identifier, ri!identifiers))
```

Now that we've created the two supporting rules, let's move on to the main expression.

Expression

```
=load(
 local!pickedStates,
 local!stateLabels: { "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "Florida", "Georgia", "Hawaii", "Idaho", "Il
inois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana"
, "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio", "Oklahoma", "Oregon", "Pennsyl
ania", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyo
ming" },
 local!stateAbbreviations: { "AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "FL", "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MD", "MA", "MI", "MN",
"MS", "MO", "MT", "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC", "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI", "W
```

```
Y" },
 a!formLayout(firstColumnContents: {
  a!pickerFieldCustom(
   label: "States",
   instructions: "Select up to three U.S. states",
   maxSelections: 3,
   suggestFunction: rule!ucArrayPickerFilter(filter: _, labels: local!stateLabels, identifiers: local!stateAbbreviations),
   selectedLabels: if(
    or(isnull(local!pickedStates), count(local!pickedStates) = 0),
    null,
    apply(rule!ucArrayPickerGetLabelForIdentifier(identifier: _, labels: local!stateLabels, identifiers: local!stateAbbreviations), local!pickedStates)
   ),
   value: local!pickedStates,
   saveInto: local!pickedStates
  ),
  a!textField(
   label: "Selected Data",
   readOnly: true,
   value: local!pickedStates
  )
 })
)
```

Test it out

1. Type in the picker field. Even if you don't know for sure how to spell the state you want or what its abbreviation is, the picker constrains your choices to valid states.
2. Select 3 states. Notice that because of the `maxSelections` configuration, once you select three states you must remove at least one of them before selecting another one.
3. Remove a selection by clicking on the blue oval. Now you can add another one.

Configure an Array Picker that Ignores Duplicates

Goal: Allow users to choose from a long text array using an autocompleting picker and prevent any choice from being picked multiple times by ignoring duplicate user selections.

The main expression uses a few supporting rules, so let's create them first.

- `ucArrayPickerFilter` : Scans labels that match the text entered by the user and returns a DataSubset for use in the SAIL picker component.
- `ucArrayPickerGetLabelForIdentifier` : For an identifier, finds its index in the identifiers array and then returns the corresponding label.
- `ucArrayPickerRemoveDuplicates` : Remove any duplicates from a text array.

Create expression rule `ucArrayPickerFilter` with the following rule inputs:

- filter (Text)
- labels (Text Array)
- identifiers (Text Array)

Enter the following definition for the rule:

```
=with(
 local!matches: where(apply(search(ri!filter, _), ri!labels)),
 type!DataSubset(data: index(ri!labels, local!matches), identifiers: index(ri!identifiers, local!matches))
)
```

Next, create expression rule `ucArrayPickerGetLabelForIdentifier` with the following rule inputs:

- identifier (Text)
- labels (Text Array)
- identifiers (Text Array)

Enter the following definition for the rule:

```
=index(ri!labels, wherecontains(ri!identifier, ri!identifiers))
```

Next, create expression rule `ucArrayPickerRemoveDuplicates` with the following rule input:

- array (Text Array)

Enter the following definition for the rule:

```
=union(ri!array, cast(typeof(ri!array), {}))
```

Now that we've created the supporting rules, let's move on to the main expression.

Expression

```
=load(
```

```
  local!pickedStates: {},
  local!stateLabels: { "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "Florida", "Georgia", "Hawaii", "Idaho", "Il
inois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana"
, "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio", "Oklahoma", "Oregon", "Pennsylv
ania", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyo
ming" },
  local!stateAbbreviations: { "AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "FL", "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MD", "MA", "MI", "MN",
"MS", "MO", "MT", "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC", "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI", "W
Y" },
  a!formLayout(firstColumnContents: {
    a!pickerFieldCustom(
      label: "States",
      instructions: "Select up to three U.S. states",
      maxSelections: 3,
      suggestFunction: rule!ucArrayPickerFilter(filter: _, labels: local!stateLabels, identifiers: local!stateAbbreviations),
      selectedLabels: if(
        count(local!pickedStates) = 0,
        null,
        apply(rule!ucArrayPickerGetLabelForIdentifier(identifier: _, labels: local!stateLabels, identifiers: local!stateAbbreviations), local!pickedStates)
      ),
      value: local!pickedStates,
      saveInto: a!save(local!pickedStates, rule!ucArrayPickerRemoveDuplicates(save!value))
    ),
    a!textField(
      label: "Selected Data",
      readOnly: true,
      value: local!pickedStates
    )
  })
)
```

Test it out

1. Type in the picker field. Even if you don't know for sure how to spell the state you want or what its abbreviation is, the picker constrains your choices to valid states.
2. After selecting a state, start typing its name again. Notice that it appears in the suggestions, but cannot be re-selected.
3. Select 3 states. Notice that because of the `maxSelections` configuration, once you select three states you must remove at least one of them before selecting another one.
4. Remove a selection by clicking on the blue oval. Now you can add another state.

Configure an Array Picker with a Show All Option

Goal: Allow users to choose from a long text array using an autocompleting picker, but also allow them to see the entire choice set using a dropdown. Dropdowns with many choices can be a little unwieldy, but when there is doubt about even the first letters of an option they can be very useful.

The main expression uses two supporting rules, so let's create them first.

- `ucArrayPickerFilter` : Scans labels that match the text entered by the user and returns a DataSubset for use in the SAIL picker component.
- `ucArrayPickerGetLabelForIdentifier` : For an identifier, finds its index in the identifiers array and then returns the corresponding label.

Create expression rule `ucArrayPickerFilter` with the following rule inputs:

- filter (Text)
- labels (Text Array)
- identifiers (Text Array)

Enter the following definition for the rule:

```
=with(
  local!matches: where(apply(search(ri!filter, _), ri!labels)),
  type!DataSubset(data: index(ri!labels, local!matches), identifiers: index(ri!identifiers, local!matches))
)
```

Next, create `ucArrayPickerGetLabelForIdentifier` with the following rule inputs:

- identifier (Text)
- labels (Text Array)
- identifiers (Text Array)

Enter the following definition for the rule:

```
=index(ri!labels, wherecontains(ri!identifier, ri!identifiers))
```

Now that we've created the two supporting rules, let's move on to the main expression.

Expression

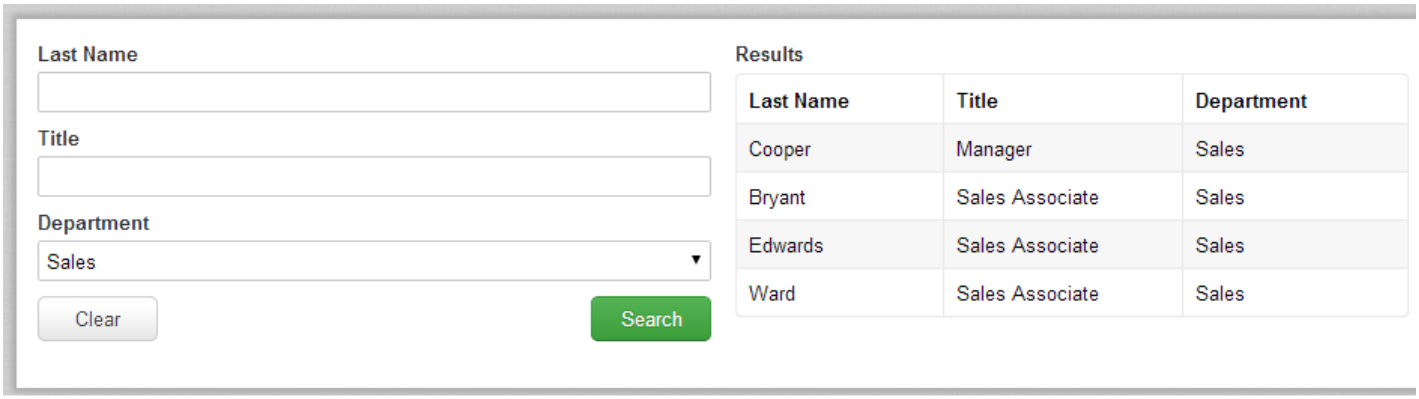
```
=load(
  local!pickedState,
  local!showPicker: true,
  local!stateLabels: { "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "Florida", "Georgia", "Hawaii", "Idaho", "Il
inois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana"
, "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio", "Oklahoma", "Oregon", "Pennsylv
ania", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyo
ming" },
  local!stateAbbreviations: { "AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "FL", "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MD", "MA", "MI", "MN",
"MS", "MO", "MT", "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC", "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI", "W
Y" },
  a!formLayout(firstColumnContents: {
    if(
      local!showPicker,
      {
        a!pickerFieldCustom(
          label: "States",
          instructions: "Select a U.S. state",
          maxSelections: 1,
          suggestFunction: rule!ucArrayPickerFilter(filter: _, labels: local!stateLabels, identifiers: local!stateAbbreviations),
          selectedLabels: if(
            or(isnull(local!pickedState), count(local!pickedState) = 0),
            null,
            apply(rule!ucArrayPickerGetLabelForIdentifier(label: _, labels: local!stateLabels, identifiers: local!stateAbbreviations), local!pickedState)
          ),
          value: local!pickedState,
          saveInto: local!pickedState
        ),
        a!linkField(
          links: a!dynamicLink(
            label: "Show All",
            value: false,
            saveInto: local!showPicker
          )
        )
      },
      a!dropdownField(
        label: "States",
        placeholderLabel: "Select a US state",
        choiceLabels: local!stateLabels,
        choiceValues: local!stateAbbreviations,
        value: local!pickedState,
        saveInto: local!pickedState
      )
    ),
    a!textField(
      label: "Selected Data",
      readOnly: true,
      value: local!pickedState
    )
  })
)
```

Test it out

1. Click the "Show All" link and see how the picker changes to a dropdown. In this example there is no link to switch back to a picker, but one could easily be added.
2. Now make a selection with the picker. Then click "Show All". Notice how, since they use the same variable as a value, the dropdown is set to the value previously selected with the picker.

Searching on Multiple Fields

Goal: Display a grid populated based on search criteria specified by end users e.g. when looking for employees by last name, title, or department. Searc criteria that are left blank are not included in the query.



For this recipe, you'll need both a Data Store Entity that is populated with data and a helper rule that will query that entity. To create the Data Store Entity, let's use the Employee entity from the Records Tutorials.

See also: Records Tutorial

For the rule that will retrieve the data from that entity, let's use the one from the Searching on Multiple Fields recipe on the Query Recipes page.

See also: Searching on Multiple Fields

Expression

```
load(
  local!allDepartments: {"Engineering", "Finance", "HR", "Professional Services", "Sales"},
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 20
  ),
  local!lastName,
  local!title,
  local!department,
  local!searchResults,
  a!dashboardLayout(
    firstColumnContents: {
      a!textField(
        label: "Last Name",
        value: local!lastName,
        saveInto: local!lastName
      ),
      a!textField(
        label: "Title",
        value: local!title,
        saveInto: local!title
      ),
      a!dropdownField(
        label: "Department",
        placeholderLabel: "All Departments",
        choiceLabels: local!allDepartments,
        choiceValues: local!allDepartments,
        value: local!department,
        saveInto: local!department
      ),
      a!buttonLayout(
        primaryButtons: {
          a!buttonWidget(
            label: "Search",
            saveInto: a!save(
              local!searchResults,
              rule!ucSearchEmployees(
                lastName: local!lastName,
                title: local!title,
                department: local!department,
                pagingInfo: local!pagingInfo
              )
            )
          )
        },
        secondaryButtons:{
          a!buttonWidget(
            label: "Clear",
            saveInto: {
              a!save(local!lastName, null),
              a!save(local!title, null),
```

```
            a!save(local!department, null),
            a!save(local!searchResults, null)
          }
        )
      }
    )
  },
  secondColumnContents: {
    a!gridField(
      label: "Results",
      columns: {
        a!gridTextColumn(
          label: "Last Name",
          field: "lastName",
          data: index(local!searchResults.data, "lastName", {})
        ),
        a!gridTextColumn(
          label: "Title",
          field: "title",
          data: index(local!searchResults.data, "title", {})
        ),
        a!gridTextColumn(
          label: "Department",
          field: "department",
          data: index(local!searchResults.data, "department", {})
        )
      },
      totalCount: if(
        isnull(local!searchResults),
        0,
        local!searchResults.totalCount
      ),
      value: local!pagingInfo,
      saveInto: local!pagingInfo
    )
  }
 )
)
```

Test it out

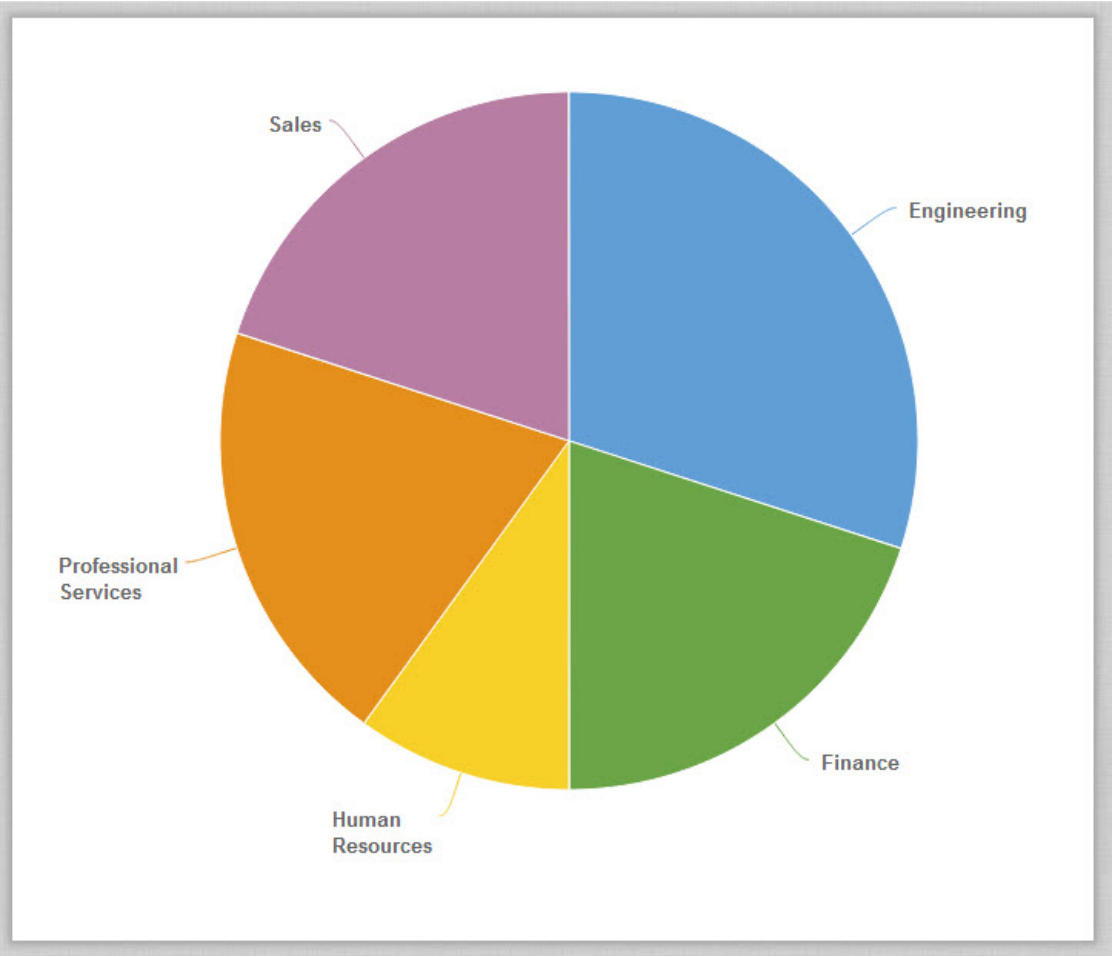
1. Click the `Search` button. Notice that an unfiltered list of employees appears in the grid.
2. Select "Sales" in the department dropdown and click on the `Search` button. The list of employees in the grid is now limited to those in the Sales department.
3. Set the department dropdown back to "All Departments", enter "Johnson" into the last name field, and click the `Search` button. The grid now contains only employees whose last name contains "Johnson".

Notable implementation details

- Because `rule!ucSearchEmployees()` is called as part of the `a!save()` function in the `saveInto` parameter of the button, the query only executes when a user clicks the Search button. Had `rule!ucSearchEmployees()` been called outside the `saveInto` parameter and the results stored in a `with()` variable, the query would be executed every time the user interacts with the interface.

Aggregate Data from a Data Store Entity and Display in a Chart

Goal: Aggregate data from a data store entity, specifically the total number of employees in a given department, to display in a pie chart.



This scenario demonstrates:

- How to use the report builder to aggregate data from a data store entity and display in a pie chart.
- How to modify the generated expression to display data labels.

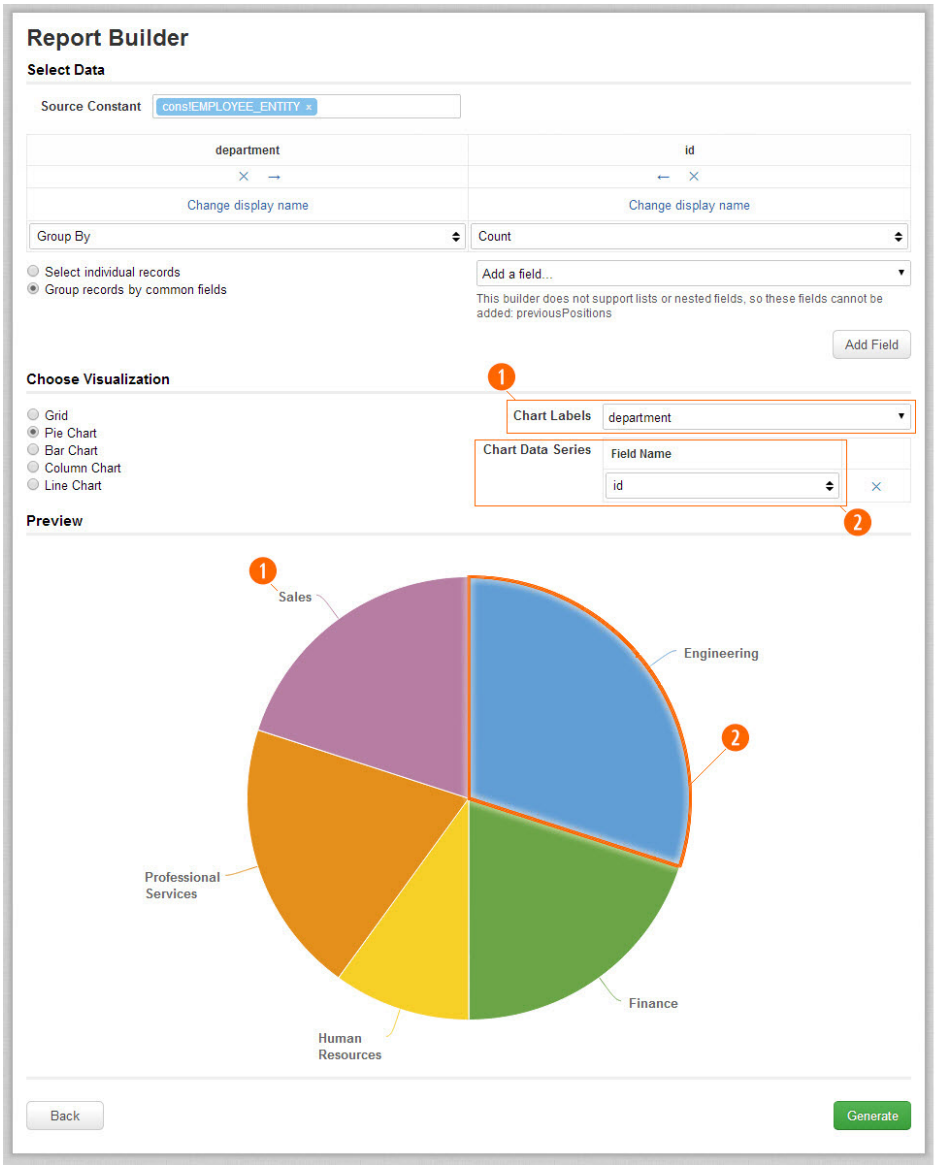
For this recipe, you'll need a data store entity. Let's use the entity that we created for the entity-backed record from the Records Tutorial. If you haven already created the Employee data store entity, do so now by completing the first five steps of the "Create Entity-Backed Records" tutorial and then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the type and `Employee` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EMPLOYEE_ENTITY` constant.
4. Delete the `firstName` and `lastName` columns so that only the `department` and `id` columns remain.
5. Select the *Group records by common fields* option and select `Count` for the `id` column.
6. Select *Pie Chart* as the visualization.
7. In the *Chart Labels* dropdown, select `department`.
8. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  a!pieChartField(
   series: {
    apply(
```

```
        a!chartSeries(label: _, data: _),
        merge(
          index(local!datasubset.data, "department", null),
          index(local!datasubset.data, "id", null)
        )
      )
    }
  )
 )
)
```

Let's take a look at the expression to see how it relates to what we configured.




```
a!pieChartField(
  series: {
    apply(
      a!chartSeries(label: _, data: _),
      merge(
        index(local!datasubset.data, "department", null),
        index(local!datasubset.data, "id", null)
      )
    )
  }
)
```

The field selected in the "Chart Labels" configuration (1) determines the labels that are displayed with each pie slice. This is the same as the *label* configuration in `a!chartSeries()`. The field selected in the "Chart Data Series" configuration (2) determines the data used to create the pie chart slices. Tl is the same as the *data* parameter in `a!chartSeries()`.

Next, let's configure the chart to display the data value next to the label for each slice.

9. Configure the chart to display data labels by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
   startIndex: 1,
   batchSize: -1,
   sort: a!sortInfo(
     field: "department",
     ascending: true
   )
 ),
 with(
   local!datasubset: a!queryEntity(
     entity: cons!EMPLOYEE_ENTITY,
     query: a!query(
```

```
      aggregation: a!queryAggregation(aggregationColumns: {
        a!queryAggregationColumn(field: "department", isGrouping: true),
        a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
      }),
      pagingInfo: local!pagingInfo
    )
  ),
  a!pieChartField(
    series: {
      apply(
        a!chartSeries(label: _, data: _),
        merge(
          index(local!datasubset.data, "department", null),
          index(local!datasubset.data, "id", null)
        )
      )
    },
    showDataLabels: true
  )
 )
)
```

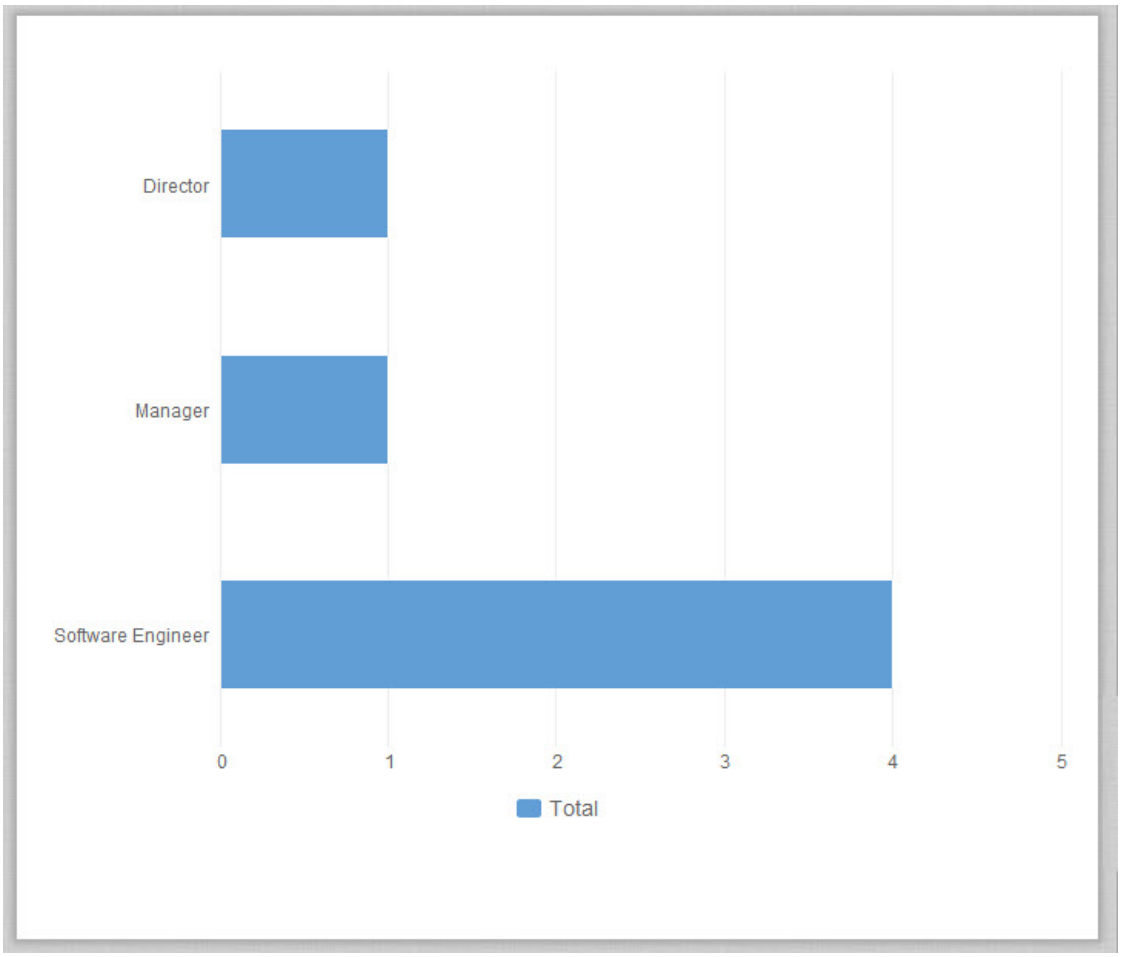
Notable implementation details

- The query that populates this chart will aggregate on the entire data set. To filter the data returned by the query before aggregating, see also: Aggregate Data from a Data Store Entity using a Default Filter and Display in a Chart

See also: Query Recipes

Aggregate Data from a Data Store Entity using a Filter and Display in a Chart

Goal: Aggregate data from a data store entity, specifically the total number of employees for each title in the Engineering department, to display in a ba chart.



This scenario demonstrates:

- How to use the report builder to aggregate data from a data store entity and display in a bar chart.
- How to modify the generated expression to add a default filter to the query.

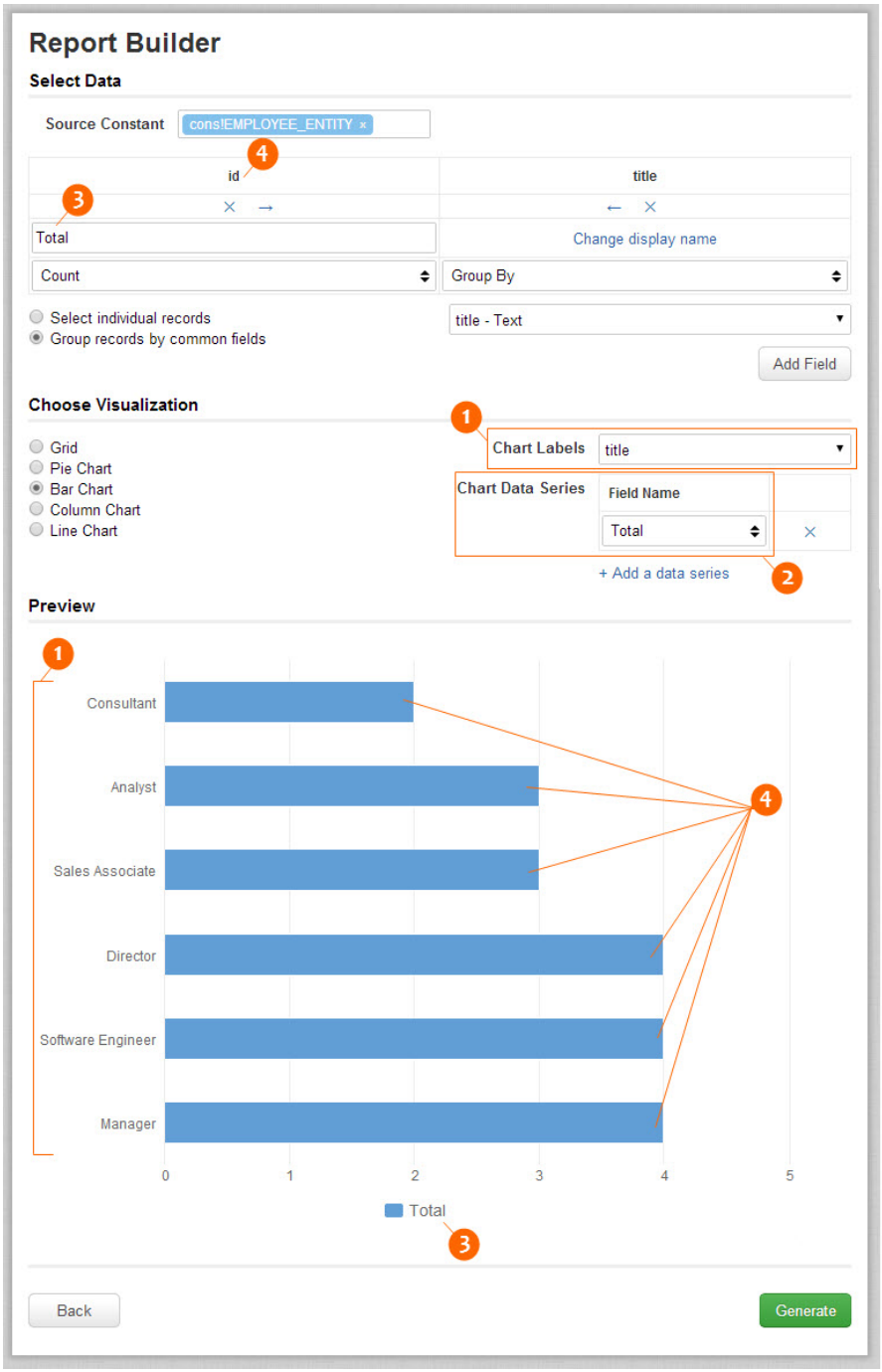
For this recipe, you'll need a data store entity. Let's use the entity that we created for the entity-backed record from the Records Tutorial. If you haven already created the Employee data store entity, do so now by completing the first five steps of the "Create Entity-Backed Records" tutorial and then

follow the steps below to generate a grid using the report builder.

1. Create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the type and `Employee` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EMPLOYEE_ENTITY` constant.
4. Delete the `firstName`, `lastName`, and `department` columns so that only the `id` column remains.
5. In the *Add a field...* dropdown, select `title` and click **Add Field**.
6. Select the *Group records by common fields* option and select `Count` for the `id` column.
7. Set the display name for the id column as `Total`.
8. Select *Bar Chart* as the visualization.
9. In the *Chart Labels* dropdown, select `title`.
10. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "id",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
          a!queryAggregationColumn(field: "title", isGrouping: true),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!barChartField(
      categories: {
        index(local!datasubset.data, "title", null)
      },
      series: {
        a!chartSeries(
          label: "Total",
          data: index(local!datasubset.data, "id", null)
        ),
      }
    )
  )
)
```

Let's take a look at the expression to see how it relates to what we configured.




```
a!barChartField(
  categories: {
    index(local!datasubset.data, "title", null)
  },
  series: {
    a!chartSeries(
      label: "Total",
      data: index(local!datasubset.data, "id", null)
    )
  }
)
```

The field selected in the "Chart Labels" configuration (1) determines the labels that are displayed on the left of each bar. This is the same as the *categori* configuration in `a!barChartField()`. The field selected in the "Chart Data Series" configuration (2) determines the data used to create the bars. The displa name for the field (3) is used in the chart legend. This is the same as the *label* configuration in `a!chartSeries()`. The data from the field itself (4) is used to create each bar. This is the same as the *data* configuration in `a!chartSeries()`.

11. Add a filter so that only employees in the Engineering department are included in the aggregation by modifying the expression with the following:

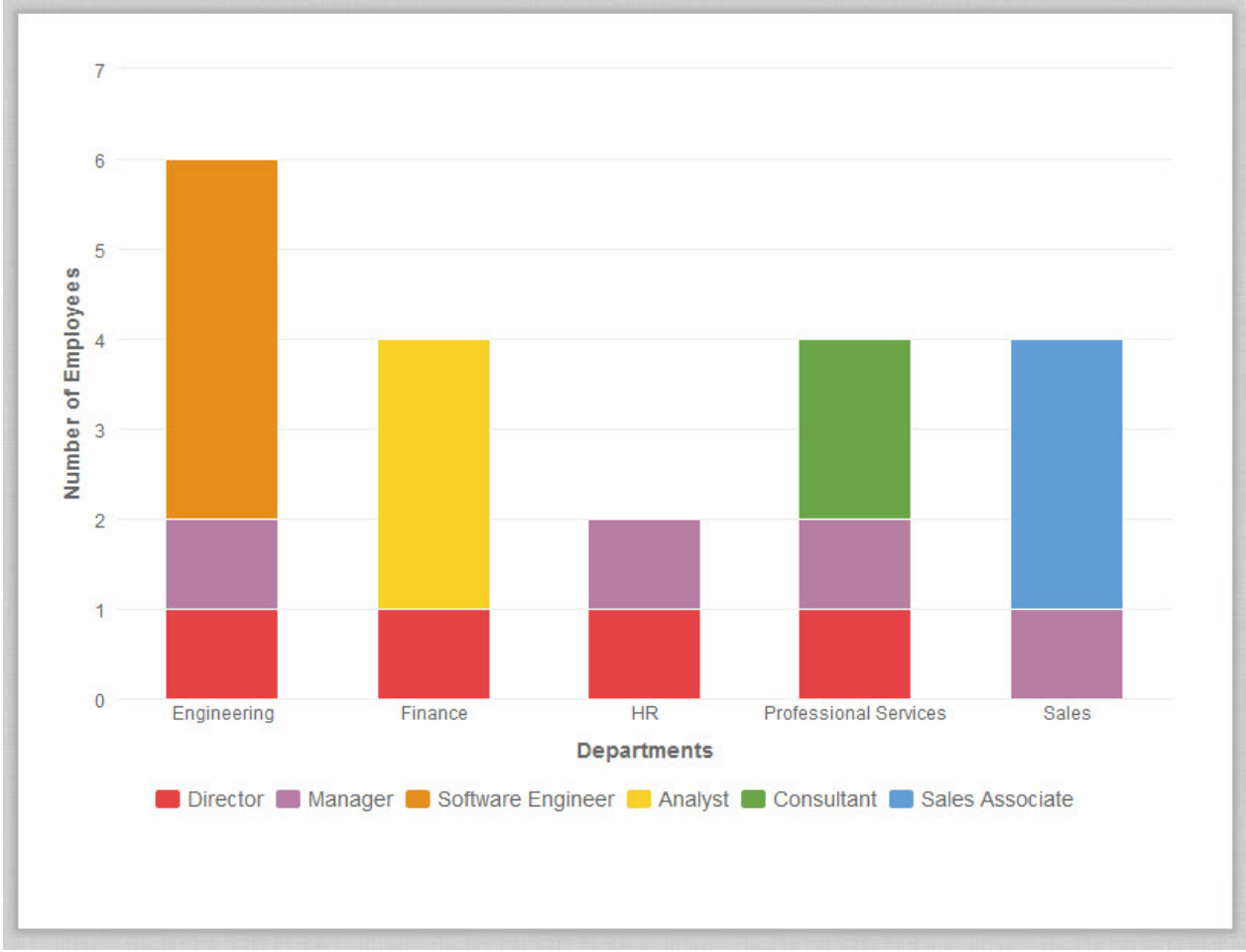
```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "id",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
          a!queryAggregationColumn(field: "title", isGrouping: true),
        }),
        filter: a!queryFilter(field: "department", operator: "=", value: "Engineering"),
```

```
      pagingInfo: local!pagingInfo
    )
  ),
  a!barChartField(
    categories: {
      index(local!datasubset.data, "title", null)
    },
    series: {
      a!chartSeries(
        label: "Total",
        data: index(local!datasubset.data, "id", null)
      ),
    }
  )
 )
)
```

See also: Query Recipes

Aggregate Data from a Data Store Entity by Multiple Fields and Display in a Chart

Goal: Aggregate data from a data store entity by multiple fields, specifically the total number of employees for each title in each department, to display a stacked column chart. This implementation can be used more generally for any chart with multiple series.



This scenario demonstrates:

- How to use the report builder to aggregate data from a data store entity by multiple fields and display in a column chart.
- How to modify the generated expression to group multiple data points together in one chart series.

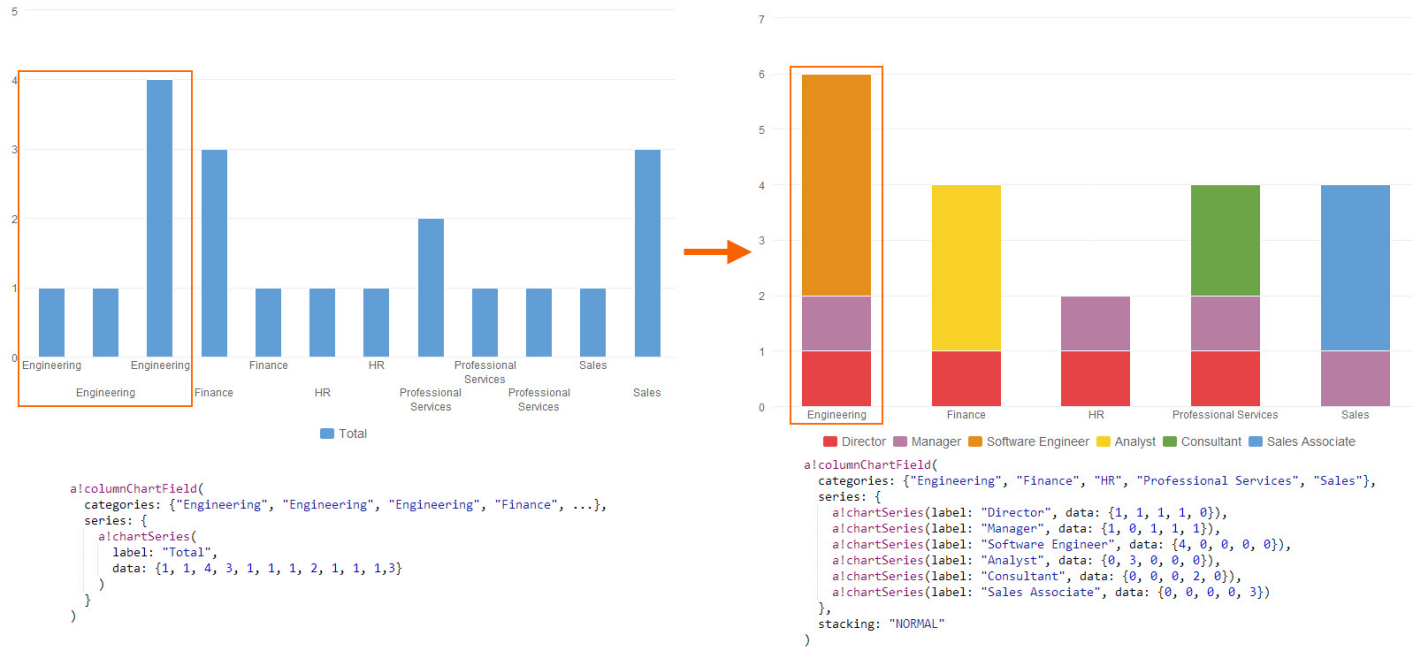
For this recipe, you'll need a data store entity. Let's use the entity that we created for the entity-backed record from the Records Tutorial. If you haven already created the Employee data store entity, do so now by completing the first five steps of the "Create Entity-Backed Records" tutorial and then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the type and `Employee` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EMPLOYEE_ENTITY` constant.
4. Delete the `firstName` and `lastName` columns so that only the `department` and `id` columns remain.
5. In the *Add a field...* dropdown, select `title` and click **Add Field**.
6. Select the *Group records by common fields* option and select `Count` for the `id` column.

7. Set the display name for the `id` column as `Total`.
8. Select *Column Chart* as the visualization.
9. In the *Chart Labels* dropdown, select `department`.
10. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
          a!queryAggregationColumn(field: "title", isGrouping: true),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!columnChartField(
      categories: {
        index(local!datasubset.data, "department", null)
      },
      series: {
        a!chartSeries(
          label: "Total",
          data: index(local!datasubset.data, "id", null)
        ),
      }
    )
  )
)
```

Notice that each department has multiple columns, each one representing the total number of employees for each title within that department. This is because all the data points are included in a single series. Instead, let's change the expression so that each department has a single column that stacks a the totals for each title, as shown below:



To do this, we need to create a separate chart series for each title that contains all data points for that title across all departments. First, we'll need to ge the unique set departments and titles to loop over in `rule!generateMultipleChartSeries`. Then, we'll need to loop over the list of titles to create a single series for each using `rule!generateChartSeries`. Finally, we'll need to loop over each department to find the corresponding data point using `rule!findDatapoints`.

11. Create an expression rule called `findDatapoints` with the following rule inputs:

- data (Any Type)
- categoryField (Text)
- labelField (Text)
- datapointField (Text)
- category (Any Type)
- label (Any Type)

Enter the following definition for the rule:

```
=with(
 local!categories: index(ri!data, ri!categoryField, {}),
 local!labels: index(ri!data, ri!labelField, {}),
 /* Find all datapoints that match both the category and chart series label   */
 local!intersection: intersection(
   where(local!categories=cast(typeof(local!categories), ri!category), 0),
   where(local!labels=cast(typeof(local!labels), ri!label), 0)
 ),
 if(
   length(local!intersection)=0,
   /* If there is no datapoint for this category-label pair, return 0 so that *
    * all subsequent points are in the correct order with the categories      */
   0,
   index(index(ri!data, ri!datapointField, {}), local!intersection, 0)
 )
)
```

12. Create an expression rule called `generateChartSeries` with the following rule inputs:

- data (Any Type)
- categoryField (Text)
- labelField (Text)
- datapointField (Text)
- categories (Any Type)
- label (Any Type)

Enter the following definition for the rule:

```
=a!chartSeries(
 label: ri!label,
 /* Loop over list of categories to find each datapoint that matches both   *
  * the series label and the category. This will ensure that the datapoints *
  * are in the correct order to display in the chart.                       */
 data: apply(
   rule!findDataPoints(
     data: ri!data,
     categoryField: ri!categoryField,
     labelField: ri!labelField,
     datapointField: ri!datapointField,
     category: _,
     label: ri!label
   ),
   ri!categories
 )
)
```

13. Create an expression rule called `generateMultipleChartSeries` with the following rule inputs:

- data (Any Type)
- categoryField (Text)
- labelField (Text)
- datapointField (Text)

Enter the following definition for the rule:

```
=load(
 local!categories: index(ri!data, ri!categoryField, {}),
 local!labels: index(ri!data, ri!labelField, {}),
 /* Loop over the list of series labels to find the datapoints *
  * for each series                                            */
 apply(
   rule!generateChartSeries(
     data: ri!data,
     categoryField: ri!categoryField,
     labelField: ri!labelField,
     datapointField: ri!datapointField,
     /* Remove duplicates from the list of categories         */
```

```
    categories: union(
      local!categories, cast(
        typeof(local!categories),
        {}
      )
    ),
    label: _
  ),
  /* Remove duplicates from the list of series labels       */
  union(local!labels, cast(typeof(local!labels), {}))
 )
)
```

14. Create the new chart series values using `generateMultipleChartSeries` by modifying the expression with the following:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
     a!queryAggregationColumn(field: "title", isGrouping: true),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  local!departments: union(local!datasubset.data.department, cast(typeof(local!datasubset.data.department), {})),
  local!series: rule!generateMultipleChartSeries(
   data: local!datasubset.data,
   categoryField: "department",
   labelField: "title",
   datapointField: "id"
  ),
  a!columnChartField(
   categories: local!departments,
   series: local!series,
   xAxisTitle: "Departments",
   yAxisTitle: "Number of Employees",
   stacking: "NORMAL"
  )
 )
)
```

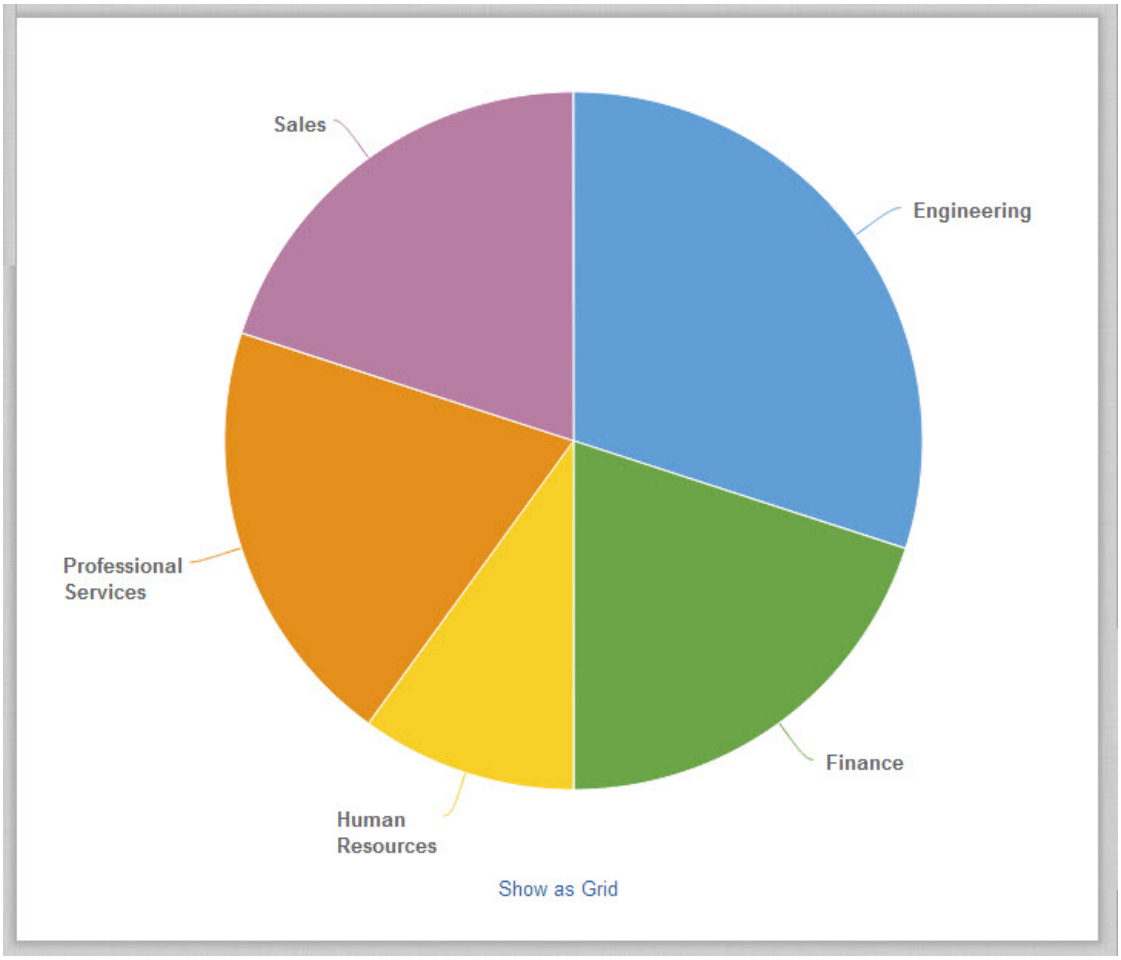
Notable implementation details

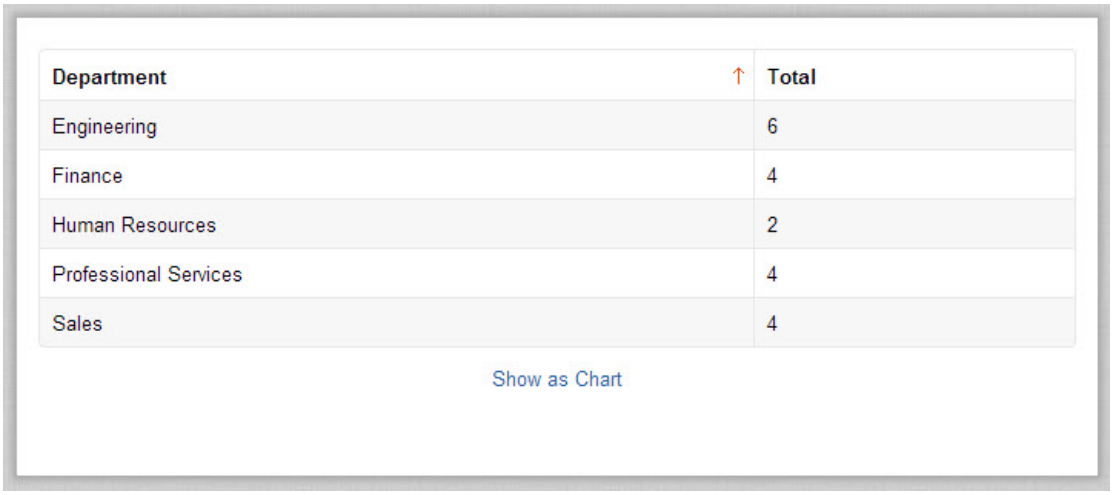
- The three rules that are used to group the data into multiple chart series can be used more generally than this example. They can be used to generate series for bar, column, or line charts and with different data sets, though they would need to be modified to work with CDT fields.
- The query that populates this chart will aggregate on the entire data set. To filter the data returned by the query before aggregating, see also: Aggregate Data from a Data Store Entity using a Filter and Display in a Chart

See also: Query Recipes

Aggregate Data from a Data Store Entity and Conditionally Display in a Chart or Grid

Goal: Aggregate data from a data store entity, specifically the total number of employees in each department, to display in a pie chart. Give the user th option to view the same data as a grid.





Department	Total
Engineering	6
Finance	4
Human Resources	2
Professional Services	4
Sales	4

Show as Chart

This scenario demonstrates:

- How to use the report builder to aggregate data from a data store entity and display in a pie chart.
- How to use the report builder to aggregate data from a data store entity and display in a grid.
- How to modify the generated expression to show the data in either a chart or a grid.

For this recipe, you'll need a data store entity. Let's use the entity that we created for the entity-backed record from the Records Tutorial. If you haven already created the Employee data store entity, do so now by completing the first five steps of the "Create Entity-Backed Records" tutorial and then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the type and `Employee` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EMPLOYEE_ENTITY` constant.
4. Delete the `firstName` and `lastName` columns so that only the `department` and `id` columns remain.
5. Select the *Group records by common fields* option and select `Count` for the `id` column.
6. Select *Pie Chart* as the visualization.
7. In the *Chart Labels* dropdown, select `department`.
8. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
```

```
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!pieChartField(
      series: {
        apply(
          a!chartSeries(label: _, data: _),
          merge(
            index(local!datasubset.data, "department", null),
            index(local!datasubset.data, "id", null)
          )
        )
      }
    )
  )
)
```

9. Copy this expression and paste it into a separate Interface Designer window for later use.
10. Clear the expression out of the design pane on the left-hand side and select **Report Builder** from the list of templates. You will see that the previous configuration is preserved.
11. Set the display name for the columns as `Department` and `Total`, respectively.
12. Select *Grid* as the visualization.
13. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 20,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!gridField(
      totalCount: local!datasubset.totalCount,
      columns: {
        a!gridTextColumn(
          label: "Department",
          field: "department",
          data: index(local!datasubset.data, "department", null)
        ),
        a!gridTextColumn(
          label: "Total",
          field: "id",
          data: index(local!datasubset.data, "id", null)
        ),
      },
      value: local!pagingInfo,
      saveInto: local!pagingInfo
    )
  )
```

```
)
```

14. Using the previously generated expression, merge the two expressions so that both the chart and the grid are displayed as shown below:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    {
      a!pieChartField(
        series: {
          apply(
            a!chartSeries(label: _, data: _),
            merge(
              index(local!datasubset.data, "department", null),
              index(local!datasubset.data, "id", null)
            )
          )
        }
      ),
      a!gridField(
        totalCount: local!datasubset.totalCount,
        columns: {
          a!gridTextColumn(
            label: "Department",
            field: "department",
            data: index(local!datasubset.data, "department", null)
          ),
          a!gridTextColumn(
            label: "Total",
            field: "id",
            data: index(local!datasubset.data, "id", null)
          ),
        },
        value: local!pagingInfo,
        saveInto: local!pagingInfo
      )
    }
  )
)
```

15. Add a link below the chart to allow the user to control whether the data is displayed in a grid or a chart:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  local!displayAsChart: true,
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
```

```
        a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
       }),
       pagingInfo: local!pagingInfo
      )
     ),
     {
      if(
       local!displayAsChart,
       a!pieChartField(
        series: {
         apply(
          a!chartSeries(label: _, data: _),
          merge(
           index(local!datasubset.data, "department", null),
           index(local!datasubset.data, "id", null)
          )
         )
        }
       ),
       a!gridField(
        totalCount: local!datasubset.totalCount,
        columns: {
         a!gridTextColumn(
          label: "Department",
          field: "department",
          data: index(local!datasubset.data, "department", null)
         ),
         a!gridTextColumn(
          label: "Total",
          field: "id",
          data: index(local!datasubset.data, "id", null)
         ),
        },
        value: local!pagingInfo,
        saveInto: local!pagingInfo
       )
      ),
      a!linkField(
       labelPosition: "COLLAPSED",
       align: "CENTER",
       links: a!dynamicLink(
        label: "Show as " & if(local!displayAsChart, "Grid", "Chart"),
        value: not(local!displayAsChart),
        saveInto: local!displayAsChart
       )
      )
     }
    )
)
```

Test it out

1. Click the "Show as grid" link. The chart will be replaced with a grid that displays the same data.
2. Click the "Show as chart" link. The grid will be replaced with the original chart.

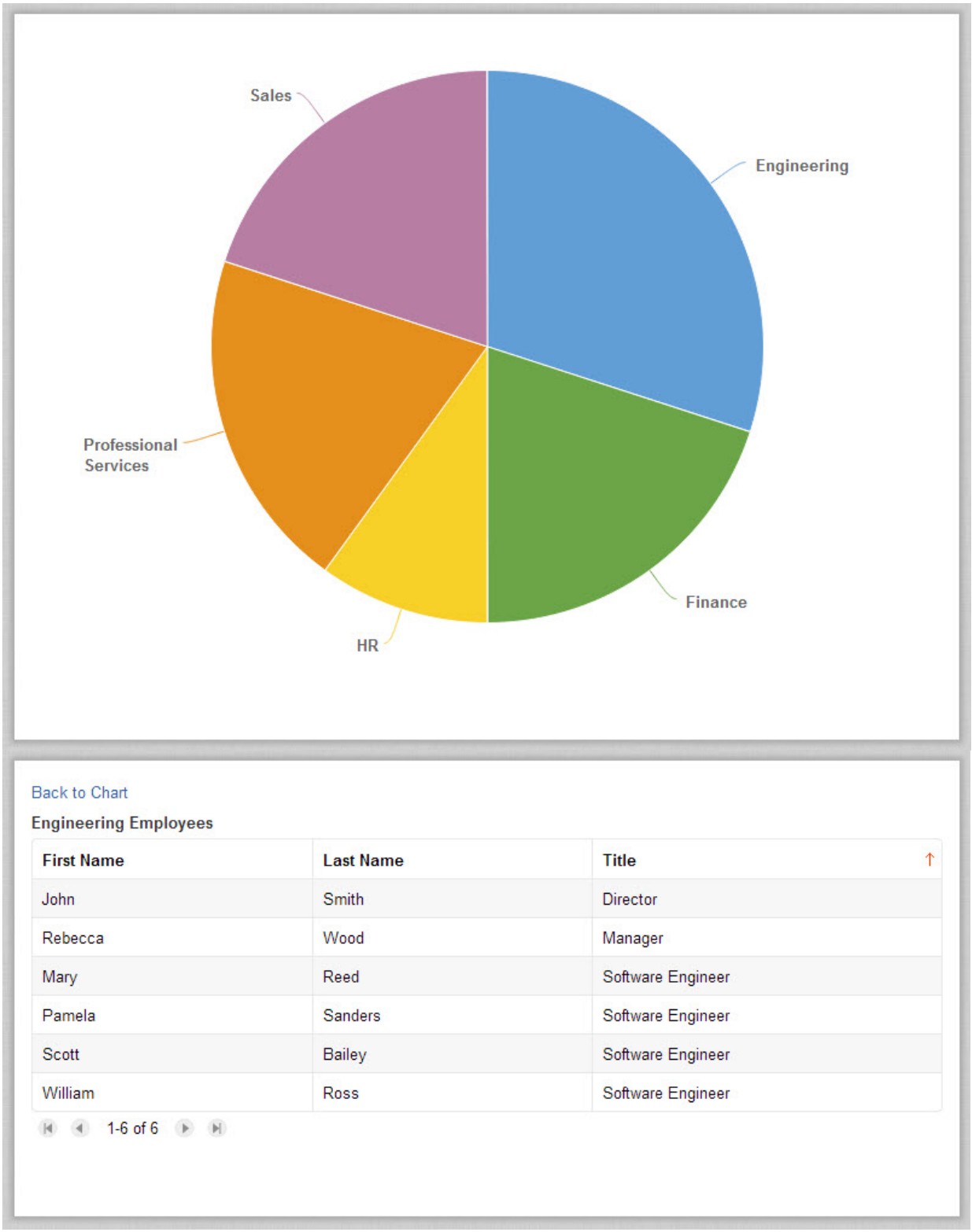
Notable implementation details

- The grid and the chart use the same paging info because they are displaying the same data. For larger data sets or charts that use multiple series you may consider using a separate paging info for the grid so that you can display the data in batches.

See also: Query Recipes

Configure a Chart Drilldown to a Grid

Goal: Display a chart with aggregate data from a data store entity, specifically the total number of employees for each department. Then add links to tl chart slices so that when a user clicks on a department, the chart is replaced by a grid that displays all employees for that department with a link to retu to the chart.



This scenario demonstrates:

- How to use the report builder to aggregate data from a data store entity and display in a pie chart.
- How to modify the generated expression to add links to each slice of the pie chart.
- How to modify the generate expression to display a grid when the user clicks on a slice of the pie chart.

For this recipe, you'll need a data store entity. Let's use the entity that we created for the entity-backed record from the Records Tutorial. If you haven already created the Employee data store entity, do so now by completing the first five steps of the "Create Entity-Backed Records" tutorial and then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the type and `Employee` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EMPLOYEE_ENTITY` constant.
4. Delete the `firstName` and `lastName` columns so that only the `department` and `id` columns remain.
5. Select the *Group records by common fields* option and select `Count` for the `id` column.
6. Select *Pie Chart* as the visualization.
7. In the *Chart Labels* dropdown, select `department`.
8. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!pieChartField(
      series: {
        apply(
          a!chartSeries(label: _, data: _),
          merge(
            index(local!datasubset.data, "department", null),
            index(local!datasubset.data, "id", null)
          )
        )
      }
    )
  )
)
```

9. Add a dynamic link to each chart so that the department is saved when a slice is clicked by modifying the expression as shown below:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  local!selectedDepartment,
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    a!pieChartField(
      series: {
        apply(
          a!chartSeries(label: _, data: _, links: _),
          merge(
            index(local!datasubset.data, "department", null),
            index(local!datasubset.data, "id", null),
            apply(
              a!dynamicLink(value: _, saveInto: local!selectedDepartment),
              index(local!datasubset.data, "department", {})
            )
          )
        )
      }
    )
  )
)
```

10. Add a grid that will show all employees for the selected department when a user clicks on a slice of the chart by modifying the expression as show below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 local!selectedDepartment,
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  if(
   isnull(local!selectedDepartment),
   a!pieChartField(
    series: {
     apply(
      a!chartSeries(label: _, data: _, links: _),
      merge(
       index(local!datasubset.data, "department", null),
       index(local!datasubset.data, "id", null),
       apply(a!dynamicLink(value: _, saveInto: local!selectedDepartment), index(local!datasubset.data, "department", null))
      )
     )
    }
   ),
   load(
    local!gridPagingInfo: a!pagingInfo(
     startIndex: 1,
     batchSize: 20,
     sort: a!sortInfo(
      field: "title",
      ascending: true
     )
    ),
    with(
     local!gridDatasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
       selection: a!querySelection(columns: {
        a!queryColumn(field: "firstName"),
        a!queryColumn(field: "lastName"),
        a!queryColumn(field: "title")
       }),
       filter: a!queryFilter(field: "department", operator: "=", value: local!selectedDepartment),
       pagingInfo: local!gridPagingInfo
      )
     ),
     a!gridField(
      label: local!selectedDepartment & " Employees",
      columns: {
       a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
       a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
       a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
      },
      value: local!gridPagingInfo,
      saveInto: local!gridPagingInfo,
      totalCount: local!gridDatasubset.totalCount
     )
    )
   )
  )
 )
```

```
)
```

11. Add a link to go back to the chart from the grid by modifying the expression as shown below:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  local!selectedDepartment,
  with(
    local!datasubset: a!queryEntity(
      entity: cons!EMPLOYEE_ENTITY,
      query: a!query(
        aggregation: a!queryAggregation(aggregationColumns: {
          a!queryAggregationColumn(field: "department", isGrouping: true),
          a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
        }),
        pagingInfo: local!pagingInfo
      )
    ),
    if(
      isnull(local!selectedDepartment),
      a!pieChartField(
        series: {
          apply(
            a!chartSeries(label: _, data: _, links: _),
            merge(
              index(local!datasubset.data, "department", null),
              index(local!datasubset.data, "id", null),
              apply(a!dynamicLink(value: _, saveInto: local!selectedDepartment), index(local!datasubset.data, "department", null))
            )
          )
        }
      ),
      load(
        local!gridPagingInfo: a!pagingInfo(
          startIndex: 1,
          batchSize: 20,
          sort: a!sortInfo(
            field: "title",
            ascending: true
          )
        ),
        with(
          local!gridDatasubset: a!queryEntity(
            entity: cons!EMPLOYEE_ENTITY,
            query: a!query(
              selection: a!querySelection(columns: {
                a!queryColumn(field: "firstName"),
                a!queryColumn(field: "lastName"),
                a!queryColumn(field: "title")
              }),
              filter: a!queryFilter(field: "department", operator: "=", value: local!selectedDepartment),
              pagingInfo: local!gridPagingInfo
            )
          ),
          {
            a!linkField(
              labelPosition: "COLLAPSED",
              links: a!dynamicLink(
                label: "Back to Chart",
                value: null,
                saveInto: {
                  local!selectedDepartment,
                  /* Need to reset the paging info back to the first page when the user *
                   * changes the filter. Otherwise, the grid could error out.          */
                  a!save(local!gridPagingInfo.startIndex, 1)
                }
              )
            ),
```

```
        a!gridField(
          label: local!selectedDepartment & " Employees",
          columns: {
            a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
            a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
            a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
          },
          value: local!gridPagingInfo,
          saveInto: local!gridPagingInfo,
          totalCount: local!gridDatasubset.totalCount
        )
      }
    )
  )
 )
)
```

Test it out

1. Click a slice of the chart. The chart will be replaced with a grid that displays all employees for that department.
2. Click the "Back to Chart" link. The grid will be replaced with the original chart.

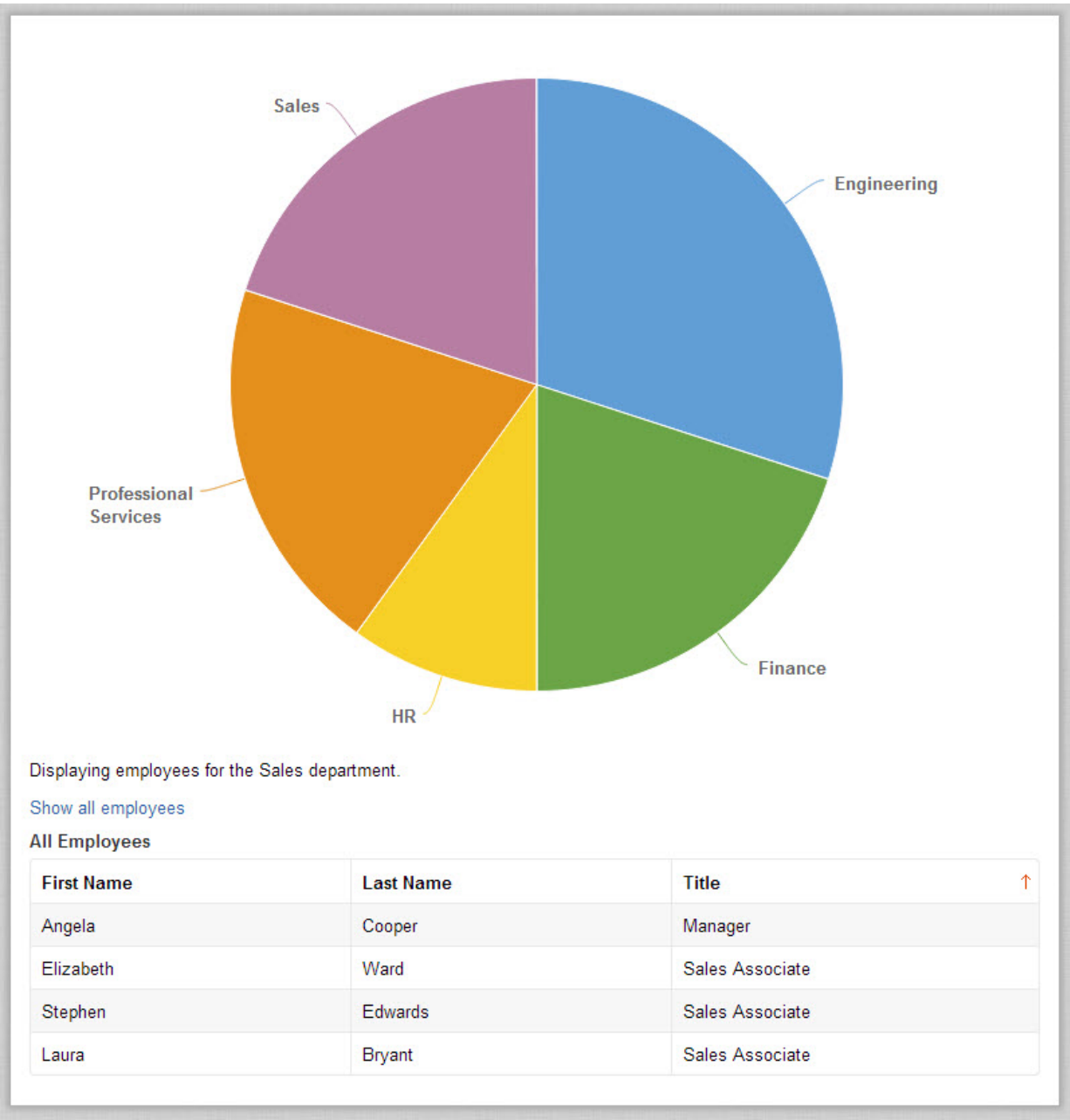
Notable implementation details

- The grid uses a separate paging info from the chart so that it can be sorted and paged independently from the chart.
- Notice that when the user goes back to the chart to select a new filter, we're always resetting the value of `local!gridPagingInfo`, ensuring that the user will see the first page for the new filter. This is necessary regardless of what the user has selected, so we ignore the value returned by the component and instead insert our own value.
- For this example, the value of the `department` field is the department name, so it can be used both for display and as the value of the filter. If usi a lookup instead, you would need to use the name as a display but save the id instead.

See also: Query Recipes

Filter the Data in a Grid Using a Chart

Goal: Display a chart with aggregate data from a data store entity, specifically the total number of employees for each department, and a grid to displa all data, specifically all employees. Then add links to the chart slices so that when a user clicks on a department, the grid will be filtered to show only employees for that department.



This scenario demonstrates:

- How to use the report builder to aggregate data from a data store entity and display in a pie chart.
- How to modify the generated expression to add links to each slice of the pie chart.
- How to modify the generate expression to display a grid with additional information below the chart when the user clicks on a slice of the pie char

For this recipe, you'll need a data store entity. Let's use the entity that we created for the entity-backed record from the Records Tutorial. If you haven already created the Employee data store entity, do so now by completing the first five steps of the "Create Entity-Backed Records" tutorial and then follow the steps below to generate a grid using the report builder.

1. Create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the type and `Employee` as the Value.
2. Open the Interface Designer and select **Report Builder** from the list of templates.
3. In the *Source Constant* field, select the `EMPLOYEE_ENTITY` constant.
4. Delete the `firstName` and `lastName` columns so that only the `department` and `id` columns remain.
5. Select the *Group records by common fields* option and select `Count` for the `id` column.
6. Select *Pie Chart* as the visualization.
7. In the *Chart Labels* dropdown, select `department`.
8. Click **Generate**. You should see the following expression in the design pane on the left-hand side:

```
load(
  local!pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: -1,
    sort: a!sortInfo(
      field: "department",
      ascending: true
    )
  ),
  with(
```

```
  local!datasubset: a!queryEntity(
    entity: cons!EMPLOYEE_ENTITY,
    query: a!query(
      aggregation: a!queryAggregation(aggregationColumns: {
        a!queryAggregationColumn(field: "department", isGrouping: true),
        a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
      }),
      pagingInfo: local!pagingInfo
    )
  ),
  a!pieChartField(
    series: {
      apply(
        a!chartSeries(label: _, data: _),
        merge(
          index(local!datasubset.data, "department", null),
          index(local!datasubset.data, "id", null)
        )
      )
    }
  )
 )
)
```

9. Add a grid below the chart to show the employees by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 local!gridPagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "title",
   ascending: true
  )
 ),
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  local!gridDatasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    selection: a!querySelection(columns: {
     a!queryColumn(field: "firstName"),
     a!queryColumn(field: "lastName"),
     a!queryColumn(field: "department"),
     a!queryColumn(field: "title")
    }),
    pagingInfo: local!gridPagingInfo
   )
  ),
  {
   a!pieChartField(
    series: {
     apply(
      a!chartSeries(label: _, data: _),
      merge(
       index(local!datasubset.data, "department", null),
       index(local!datasubset.data, "id", null)
      )
```

```
        )
       }
      ),
      a!gridField(
       label: "All Employees",
       columns: {
        a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
        a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
        a!gridTextColumn(label: "Department", field: "department", data: index(local!gridDatasubset.data, "department", {})),
        a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
       },
       value: local!gridPagingInfo,
       saveInto: local!gridPagingInfo,
       totalCount: local!gridDatasubset.totalCount
      )
     }
    )
)
```

10. Next we need to create the rule that will generate the dynamic links for each pie chart slice. Create an expression rule called `ucGenerateDynamicLinkForChartSeries` with the following inputs:

- departmentValue (Text)
- selectedDepartment: (Text)
- pagingInfo (Any Type)

Enter the following definition for the rule:

```
=a!dynamicLink(
 value: ri!departmentValue,
 saveInto: {
  ri!selectedDepartment,
  a!save(ri!pagingInfo.startIndex, 1)
 }
)
```

11. Add a dynamic link to each chart slice so that the department is saved when a slice is clicked by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 local!gridPagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "title",
   ascending: true
  )
 ),
 local!selectedDepartment,
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  local!gridDatasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    selection: a!querySelection(columns: {
     a!queryColumn(field: "firstName"),
     a!queryColumn(field: "lastName"),
     a!queryColumn(field: "department"),
     a!queryColumn(field: "title")
```

```
      }),
      pagingInfo: local!gridPagingInfo
    )
  ),
  {
    a!pieChartField(
      series: {
        apply(
          a!chartSeries(label: _, data: _, links: _),
          merge(
            index(local!datasubset.data, "department", null),
            index(local!datasubset.data, "id", null),
            apply(
              rule!ucGenerateDynamicLinkForChartSeries(
                departmentValue: _,
                selectedDepartment: local!selectedDepartment,
                pagingInfo: local!gridPagingInfo
              ),
              index(local!datasubset.data, "department", null)
            )
          )
        )
      }
    ),
    a!gridField(
      label: "All Employees",
      columns: {
        a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
        a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
        a!gridTextColumn(label: "Department", field: "department", data: index(local!gridDatasubset.data, "department", {})),
        a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
      },
      value: local!gridPagingInfo,
      saveInto: local!gridPagingInfo,
      totalCount: local!gridDatasubset.totalCount
    )
  }
 )
)
```

12. Add a filter to the grid expression so that the grid only displays employees for the selected department by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 local!gridPagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "title",
   ascending: true
  )
 ),
 local!selectedDepartment,
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  local!gridDatasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
```

```
      selection: a!querySelection(columns: {
        a!queryColumn(field: "firstName"),
        a!queryColumn(field: "lastName"),
        a!queryColumn(field: "department"),
        a!queryColumn(field: "title")
      }),
      filter: if(
        isnull(local!selectedDepartment),
        null,
        a!queryFilter(field: "department", operator: "=", value: local!selectedDepartment)
      ),
      pagingInfo: local!gridPagingInfo
    )
  ),
  {
    a!pieChartField(
      series: {
        apply(
          a!chartSeries(label: _, data: _, links: _),
          merge(
            index(local!datasubset.data, "department", null),
            index(local!datasubset.data, "id", null),
            apply(
              rule!ucGenerateDynamicLinkForChartSeries(
                departmentValue: _,
                selectedDepartment: local!selectedDepartment,
                pagingInfo: local!gridPagingInfo
              ),
              index(local!datasubset.data, "department", {})
            )
          )
        )
      }
    ),
    a!gridField(
      label: "All Employees",
      columns: {
        a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
        a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
        a!gridTextColumn(label: "Department", field: "department", data: index(local!gridDatasubset.data, "department", {})),
        a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
      },
      value: local!gridPagingInfo,
      saveInto: local!gridPagingInfo,
      totalCount: local!gridDatasubset.totalCount
    )
  }
 )
)
```

13. Add a text field to indicate how the grid is filtered and link to clear the filter between the grid and the chart by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 local!gridPagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "title",
   ascending: true
  )
 ),
 local!selectedDepartment,
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
```

```
    aggregation: a!queryAggregation(aggregationColumns: {
      a!queryAggregationColumn(field: "department", isGrouping: true),
      a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
  )
),
local!gridDatasubset: a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    selection: a!querySelection(columns: {
      a!queryColumn(field: "firstName"),
      a!queryColumn(field: "lastName"),
      a!queryColumn(field: "department"),
      a!queryColumn(field: "title")
    }),
    filter: if(
      isnull(local!selectedDepartment),
      null,
      a!queryFilter(field: "department", operator: "=", value: local!selectedDepartment)
    ),
    pagingInfo: local!gridPagingInfo
  )
),
{
  a!pieChartField(
    series: {
      apply(
        a!chartSeries(label: _, data: _, links: _),
        merge(
          index(local!datasubset.data, "department", null),
          index(local!datasubset.data, "id", null),
          apply(
            rule!ucGenerateDynamicLinkForChartSeries(
              departmentValue: _,
              selectedDepartment: local!selectedDepartment,
              pagingInfo: local!gridPagingInfo
            ),
            index(local!datasubset.data, "department", {})
          )
        )
      )
    }
  ),
  if(
    isnull(local!selectedDepartment),
    {},
    {
      a!textField(
        labelPosition: "COLLAPSED",
        readOnly: true,
        value: "Displaying employees for the " & local!selectedDepartment & " department."
      ),
      a!linkField(
        labelPosition: "COLLAPSED",
        links: a!dynamicLink(
          label: "Show all employees",
          value: null,
          saveInto: {
            local!selectedDepartment,
            /* Need to reset the paging info back to the first page when the user *
             * changes the filter. Otherwise, the grid could error out.          */
            a!save(local!gridPagingInfo.startIndex, 1)
          }
        )
      )
    }
  ),
  a!gridField(
    label: "All Employees",
    columns: {
      a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
      a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
      a!gridTextColumn(label: "Department", field: "department", data: index(local!gridDatasubset.data, "department", {})),
      a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
```

```
      },
      value: local!gridPagingInfo,
      saveInto: local!gridPagingInfo,
      totalCount: local!gridDatasubset.totalCount
    )
  }
 )
)
```

14. Hide the "Department" column in the grid when a filter is applied by modifying the expression as shown below:

```
load(
 local!pagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: -1,
  sort: a!sortInfo(
   field: "department",
   ascending: true
  )
 ),
 local!gridPagingInfo: a!pagingInfo(
  startIndex: 1,
  batchSize: 20,
  sort: a!sortInfo(
   field: "title",
   ascending: true
  )
 ),
 local!selectedDepartment,
 with(
  local!datasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    aggregation: a!queryAggregation(aggregationColumns: {
     a!queryAggregationColumn(field: "department", isGrouping: true),
     a!queryAggregationColumn(field: "id", aggregationFunction: "COUNT"),
    }),
    pagingInfo: local!pagingInfo
   )
  ),
  local!gridDatasubset: a!queryEntity(
   entity: cons!EMPLOYEE_ENTITY,
   query: a!query(
    selection: a!querySelection(columns: {
     a!queryColumn(field: "firstName"),
     a!queryColumn(field: "lastName"),
     a!queryColumn(field: "department"),
     a!queryColumn(field: "title")
    }),
    filter: if(
     isnull(local!selectedDepartment),
     null,
     a!queryFilter(field: "department", operator: "=", value: local!selectedDepartment)
    ),
    pagingInfo: local!gridPagingInfo
   )
  ),
  {
   a!pieChartField(
    series: {
     apply(
      a!chartSeries(label: _, data: _, links: _),
      merge(
       index(local!datasubset.data, "department", null),
       index(local!datasubset.data, "id", null),
       apply(
        rule!ucGenerateDynamicLinkForChartSeries(
         departmentValue: _,
         selectedDepartment: local!selectedDepartment,
         pagingInfo: local!gridPagingInfo
        ),
        index(local!datasubset.data, "department", {})
       )
      )
     )
```

```
      }
    ),
    if(
      isnull(local!selectedDepartment),
      {},
      {
        a!textField(
          labelPosition: "COLLAPSED",
          readOnly: true,
          value: "Displaying employees for the " & local!selectedDepartment & " department."
        ),
        a!linkField(
          labelPosition: "COLLAPSED",
          links: a!dynamicLink(
            label: "Show all employees",
            value: null,
            saveInto: {
              local!selectedDepartment,
              /* Need to reset the paging info back to the first page when the user *
               * changes the filter. Otherwise, the grid could error out.          */
              a!save(local!gridPagingInfo.startIndex, 1)
            }
          )
        )
      }
    ),
    a!gridField(
      label: "All Employees",
      columns: {
        a!gridTextColumn(label: "First Name", field: "firstName", data: index(local!gridDatasubset.data, "firstName", {})),
        a!gridTextColumn(label: "Last Name", field: "lastName", data: index(local!gridDatasubset.data, "lastName", {})),
        if(
          isnull(local!selectedDepartment),
          a!gridTextColumn(label: "Department", field: "department", data: index(local!gridDatasubset.data, "department", {})),
          {}
        ),
        a!gridTextColumn(label: "Title", field: "title", data: index(local!gridDatasubset.data, "title", {}))
      },
      value: local!gridPagingInfo,
      saveInto: local!gridPagingInfo,
      totalCount: local!gridDatasubset.totalCount
    )
  }
 )
)
```

Test it out

1. Click a slice of the chart. The grid below the chart will be filtered to display only employees for that department.
2. Click the "Show all employees" link. The grid will display all employees.

Notable implementation details

- Notice that when the grid is filtered, we are not querying the department field. This allows us to only query the data that we plan on displaying in the grid.
- Notice that when the user goes back to the chart to select a new filter, we're always resetting the value of `local!gridPagingInfo`, ensuring that the user will see the first page for the new filter. This is necessary regardless of what the user has selected, so we ignore the value returned by the component and instead insert our own value.

See also: Query Recipes