Toggle navigation

Records Tutorial

Appian 7.6

Toggle nav

The walkthroughs on this page will help you create your first records. They are split up by source type and increase in complexity as you progress.

Use the data provided to understand how the configurations work. Then, try it with your own data. Keep in mind, the final configurations will need to change if your data has different field names.

The content below assumes a basic familiarity with SAIL interfaces and focuses more on the specifics of selecting a data source and modifying the list view and related actions for a record. Consider going through the SAIL Tutorial and taking a look at the Record Design page before proceeding.

See also: Record Design and SAIL Tutorial

Create Entity-Backed Records

The steps below show how to create a simple entity-backed record type for employee information saved to a data store entity.

Before we begin, let's create the data store entity and its data.

1. Create a custom data type called `Employee` with the following fields in the associated data types:
   - id (Integer)
   - firstName (Text)
   - lastName (Text)
   - department (Text)
   - title (Text)
   - phoneNumber (Text)
   - startDate (Date)

   See also: Defining a Custom Data Type
2. Designate the id field as the primary key and set to generate value. Only entities that have a defined primary key (and not an automatically generated one) can be a record source.
   - See also: Indexes and Primary Keys
3. Save and publish the CDT.
4. Create a data store entity that references the `Employee` CDT and publish its data store.
   - See also: Data Stores
5. Insert the following values into the table:

id	firstName	lastName	department	title	phoneNumber	startDate
1	John	Smith	Engineering	Director	800-123-4567	2013-01-02
2	Michael	Johnson	Finance	Analyst	866-987-6543	2012-06-15
3	Mary	Reed	Engineering	Software Engineer	789-456-0123	2001-01-02
4	Angela	Cooper	Sales	Manager	404-123-4567	2005-10-15
5	Elizabeth	Ward	Sales	Sales Associate	570-987-6543	2010-01-02
6	Daniel	Lewis	HR	Manager	866-876-5432	2010-01-02
7	Paul	Martin	Finance	Analyst	703-609-3691	2009-12-01
8	Jessica	Peterson	Finance	Analyst	404-987-6543	2004-11-01
9	Mark	Hall	Professional Services	Director	789-012-3456	2009-06-01
10	Rebecca	Wood	Engineering	Manager	570-210-3456	2008-07-27
11	Pamela	Sanders	Engineering	Software Engineer	570-123-4567	2005-04-01

12	Christopher	Morris	Professional Services	Consultant	321-456-7890	2011-03-29
13	Kevin	Stewart	Professional Services	Manager	800-345-6789	2008-02-29
14	Stephen	Edwards	Sales	Sales Associate	866-765-4321	2006-01-02
15	Janet	Coleman	Finance	Director	789-654-3210	1999-12-30
16	Scott	Bailey	Engineering	Software Engineer	404-678-1235	2005-03-15
17	Andrew	Nelson	Professional Services	Consultant	321-789-4560	2005-03-15
18	Michelle	Foster	HR	Director	404-345-6789	2005-03-15
19	Laura	Bryant	Sales	Sales Associate	800-987-6543	2004-11-01
20	William	Ross	Engineering	Software Engineer	866-123-4567	2008-07-27

Now that we have the data, we can create the record type object which will generate a record for each of the rows in our Employees data store entity.

1. From the left navigation of the System view, select **System Administrator Home > Data Management > Data Management**.
2. Select the **Record Types** tab.
3. Click **Create Record Type**.
4. Enter the following values for the associated fields:
   - Name: Employee Directory
   - Plural Name: Employees
   - Description: List of current employees
   - URL Stub: employees
   - Source Type: Data Store Entity
   - Source: "Employee"
   - List View Item Template:

     ```
     =a!listViewItem(
     title: rf!firstName & " " & rf!lastName
     )
     ```

   - Sort Field: lastName
   - Sort Order: Ascending
   - SAIL Summary Dashboard:

     ```
     =a!dashboardLayout(
     firstColumnContents: {
     a!textField(
     label: "Name",
     labelPosition: "ADJACENT",
     readOnly: true,
     value: rf!firstName & " " & rf!lastName
     ),
     a!textField(
     label: "Department",
     labelPosition: "ADJACENT",
     readOnly: true,
     value: rf!department
     ),
     a!textField(
     label: "Title",
     labelPosition: "ADJACENT",
     readOnly: true,
     value: rf!title
     ),
     a!textField(
     label: "Contact Number",
     labelPosition: "ADJACENT",
     readOnly: true,
     value: rf!phoneNumber
     ),
     a!textField(
     label: "Start Date",
     labelPosition: "ADJACENT",
     readOnly: true,
     value: rf!startDate
     )
     }
     )
     ```

Your form should look like the following:

5. Click **Create Record Type**.

You have just created a record for each of the employees listed in the Employees data store entity.

To view your records, open Tempo in a new browser, select the Records tab, and click Employees. You should see the following:

To view a record, select a name from the list. For Andrew Nelson you should see the following:

The design of the summary dashboard is based on the expression we added to the SAIL Summary Dashboard field of the record type. Looking back at the expression, you'll notice we used the `rf!` domain to access our data. This is required for entity-backed records. Other than that, you can change this record dashboard design as you would any other SAIL interface.

See also: Configure the SAIL Summary Dashboard and SAIL Tutorial

You can also configure additional record dashboards for users to view more information on the record.

See also: Add Additional Dashboards

Since this record type is not backed by a process model with quick tasks and we have not configured a related action yet, selecting the Related Actions view displays an empty list. Later on, we'll add to this list, but for now, let's focus on providing users with more data.

While working through the sections below, keep Tempo open with your record in one browser and the Designer interface with your record type object open in another. This way, when we make changes to the record type object, you can just refresh Tempo to see the changes automatically.

Add More Information to the List View

Currently, our record list view is pretty bare. Looking at the syntax for our List View Item Template value, you can see why. It only includes the name field:

```
=a!listViewItem(
  title: rf!firstName & " " & rf!lastName
)
```

By adding to this value, we can add more information to the record list view. Let's create a description to appear under the record titles that includes the Department and Title for each of our records.

Navigate to the Record Types tab of the Data Management page and select the Employee Directory record.

Modify the List View Item Template with the following:

```
=a!listViewItem(
  title: rf!firstName & " " & rf!lastName,
  details: rf!title & ", " & rf!department
)
```

Save the record type and refresh Tempo.

You should see the following:

Other options for the record list view include configuring a unique image or adding a timestamp.

See also: Configure the Record List View

Add A Facet

Now that we have Employee Directory records, let's add a facet so users can filter the records by Department.

1. Navigate to the **Record Types** tab of the Data Management page and select the Employee Directory record.
   - The Edit Record Type page displays.
2. Scroll down to the Facets section.
3. Click **Create a New Facet**.
4. Enter the following values for the associated fields exactly as shown (including quotes where present):
   - Facet Name: Department
   - Order Index: 1
   - Label: "Department"
   - Record Field: department
   - Initial Option Label: "Engineering"
   - Operator: =
   - Value 1: "Engineering"
5. Click **Create Facet**.
   - This creates the Department facet and the first facet option of Engineering. Now we need to add facet options for the remaining departments.
6. In the same Facets section, click Department from the Facets list.
   - The Update Facet section displays.
7. Click **Create a New Facet Option**.
8. Enter the following values for the related Finance option:
   - Option Label: "Finance"
   - Operator: =
   - Value 1: "Finance"
9. Click **Create New Facet Option**.
   - Finance is added to the list of facet options.
10. Click **Create a New Facet Option**.
11. Enter the following values for the Human Resources option:
    - Option Label: "Human Resources"
    - Operator: =
    - Value 1: "HR"
12. Click **Create New Facet Option**.
    - Human Resources is added to the list of facet options.

13. Click **Create a New Facet Option**.
14. Enter the following values for the Professional Services option:
    - Option Label: "Professional Services"
    - Operator: =
    - Value 1: "Professional Services"
15. Click **Create New Facet Option**.
    - Professional Services is added to the list of facet options.
16. Click **Create a New Facet Option**.
    - Enter the following values for the Sales option:
    - Option Label: "Sales"
    - Operator: =
    - Value 1: "Sales"
17. Click **Create New Facet Option**.
    - Sales is added to the list of facet options. You should see five items in your Facet Options list.
18. Click **Update Facet**.

You have just created a Department facet with the five departments as options.

Refresh the Employees record in Tempo. You should see the following:

To see the full list of options, click **More...** Users can select any one of these options to filter the records down to just those in the selected department(s).

See also: Facets

Add a Related Action

Remember how the Related Actions view for each record was empty? Now let's add an action to it that allows users to update the phone number for the record they are currently viewing.

1. Create a query rule called `getEmployeeById` that takes an integer as a parameter and returns the Employee with that id.
2. Create a process model called Update Phone Number with a process variable named `employeeId` of type Number (Integer) marked as a parameter and no start form.
3. Set up an activity chain from the model's start node into a User Input Activity.
4. For the User Input activity, assign the task to "=pp!initiator", and create a form titled "Update Phone Number" with one text field called "New Phone Number" that has a default value of `rule!getEmployeeById(pv!employeeId).phoneNumber` and saves into `ac!newPhoneNumber`. `ac!newPhoneNumber` should then be saved into a process variable so that it can be accessed by other nodes later in the process.
5. Use the Write to Data Store Entity Smart Service to update the data store entity that has an id that matches the `employeeId` process variable.
6. Publish the process model.
7. Navigate to the Record Types tab of the Data Management page and select the Employee Directory record.
8. Scroll down to the Related Actions section, and click **Create a New Related Action**.
9. For Target Process Model, select the process model you configured in step 2.
10. For Context Expression, enter: `{employeeId: rp!id}`

The Update Phone Number process model is now a related action for your Employee Directory records where users can update the phone number of the record they're viewing.

Refresh the Employees record in Tempo and select an employee from the list. Click **Related Actions**. You should see the following:

Click **Update Phone Number**. You should see the following:

Enter a new number, click **Submit**, and refresh Daniel Lewis' record. You should see the updated number.

See also: Related Actions

Give Users Access to View the Records

Since we didn't add any values to the Viewer, Editor, Auditor, or Administrator Group fields, only system administrators can view the records in Tempo. By adding a group to these fields, you can open up the records to other users.

1. Create a group that contains all of the users who should be able to see Employee records. See Creating Groups
2. Navigate to the Record Types tab of the Data Management page and select the Employee Directory record.
    - The Edit Record Type page displays.
3. In the Viewers field, select the group you created in step 1.
4. Click Update Record Type

All members of the group from step 1 can now see the Employee record. You can repeat these steps for the other roles.

See also: Record Type Security

Create Process-Backed Records

Now that we've created an entity-backed record, let's move on to process-backed records.

Before we start, though, we'll need a process model. For our example, let's use the process model from the Process Model Tutorial. If you haven't already created the Submit Expenses process model, do so now by completing the first five sections of the tutorial, then follow the steps below to create your record type.

See also: Process Model Tutorial

1. From the left navigation of the System view, select System Administrator Home > Data Management > Data Management.
2. Select the Record Types tab.
3. Click **Create Record Type**.
4. Enter the following values for the associated fields:
   - Name: Expense Report
   - Plural Name: Expense Reports
   - Description: List of expense reports
   - URL Stub: expenses
   - Source Type: Process Model
   - Source: "Submit Expenses"
   - List View Item Template:

```
=a!listViewItem(
title: rf!expenseItem
)
```

   - Sort Field: expenseDate
   - Sort Order: Descending
   - SAIL Summary Dashboard:

```
=a!dashboardLayout(
firstColumnContents: {
a!textField(
label: "Requester",
labelPosition: "ADJACENT",
readOnly: true,
value: user(touser(rf!pp.initiator), "firstName") & " " & user(touser(rf!pp.initiator), "lastName")
),
a!textField(
label: "Amount",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!expenseAmount
),
a!textField(
label: "Date",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!expenseDate
)
}
)
```

Your form should look like the following:

5. Click **Create Record Type**.

You have just created a record for each of the processes of the Submit Expenses process model.

If you go the record list view in Tempo, however, you might not see any expense reports listed. This is because you don't have any instances of the process. Let's add one and take a look at the result.

Start an instance of the Submit Expenses process with the following values:

- Expense Item: Plane ticket to St. Louis
- Expense Amount: 200
- Expense Date: 4/16/2013

In Tempo, select the **Records** tab, and click **Expense Reports**.

You should see the following:

Select the "Plane ticket to St. Louis" record.

You should see the following:

Just like the entity-backed records, the design of the record dashboard is based on the expression we added to the SAIL Summary Dashboard field of the record type. You'll notice we used the `rf!` domain again, but this time it was to access process variables of the process model. This is required for process-backed records. Along with the process variables, you can also access certain process and process model properties using the same domain.

Only certain process and process model properties are available for use in SAIL interfaces.

For the full list, see also: Configure the Record List View

Other than that, you can change this record dashboard design as you would any other SAIL interface.

Similarity to Entity-backed Records

Notice that both the list view and summary dashboard for process-backed records look the same as for entity-backed records. The same advanced configurations you did in the entity-backed record example are also applicable to process-backed records. The process for implementing these configurations is exactly the same as well. See the sections in the entity-backed records example for the step-by-step process.

Add Related Actions for Quick Tasks

The only new concept for process-backed records that we did not cover with entity-backed records is exposing quick tasks as related actions. This is because entities can't have quick tasks; only processes can. However, exposing your processes quick tasks as related actions on the record is easy.

If the process model you used as the source for the record had any quick tasks (the example we used did not), the quick tasks that are available to the user would automatically appear in the list of related actions for those records. You don't have to take any further action to get them to appear in the list. You are also able to configure other related actions following the same example we saw for entity-backed records.

Create Service-Backed Records

Service-backed records have an expression as the source of data. The expression must return a value of type DataSubset. Appian recommends saving the source expression as a rule for version control and testing purposes and then calling the rule as the Source value.

See also: DataSubset and a!dataSubset.

Create the Source Data Type

Let's begin with the same Employee Directory example that we used for entity-based records above, but this time we will create it as a service-backed record type.

To read about the different types of records and understand when to use which, see also: Record Source

We will start by setting up the same data as we used previously, so you can skip these steps if you already have the data set up from the entity-backed record type example.

1. Create a custom data type called `Employee` with the following fields in the associated data types:
   - id (Integer)
   - firstName (Text)
   - lastName (Text)
   - department (Text)
   - title (Text)
   - phoneNumber (Text)
   - startDate (Date)

   See also: Defining a Custom Data Type
2. Designate the id field as the primary key.
   - See also: Indexes and Primary Keys
3. Save and publish the CDT.
4. Create a data store entity that references the `Employee` CDT and publish its data store.
5. Insert the following values into the table:

id	firstName	lastName	department	title	phoneNumber	startDate
1	John	Smith	Engineering	Director	800-123-4567	2013-01-02
2	Michael	Johnson	Finance	Analyst	866-987-6543	2012-06-15
3	Mary	Reed	Engineering	Software Engineer	789-456-0123	2001-01-02
4	Angela	Cooper	Sales	Manager	404-123-4567	2005-10-15
5	Elizabeth	Ward	Sales	Sales Associate	570-987-6543	2010-01-02
6	Daniel	Lewis	HR	Manager	866-876-5432	2010-01-02
7	Paul	Martin	Finance	Analyst	703-609-3691	2009-12-01
8	Jessica	Peterson	Finance	Analyst	404-987-6543	2004-11-01
9	Mark	Hall	Professional Services	Director	789-012-3456	2009-06-01
10	Rebecca	Wood	Engineering	Manager	570-210-3456	2008-07-27
11	Pamela	Sanders	Engineering	Software Engineer	570-123-4567	2005-04-01
12	Christopher	Morris	Professional Services	Consultant	321-456-7890	2011-03-29
13	Kevin	Stewart	Professional Services	Manager	800-345-6789	2008-02-29
14	Stephen	Edwards	Sales	Sales Associate	866-765-4321	2006-01-02
15	Janet	Coleman	Finance	Director	789-654-3210	1999-12-30
16	Scott	Bailey	Engineering	Software Engineer	404-678-1235	2005-03-15
17	Andrew	Nelson	Professional Services	Consultant	321-789-4560	2005-03-15
18	Michelle	Foster	HR	Director	404-345-6789	2005-03-15
19	Laura	Bryant	Sales	Sales Associate	800-987-6543	2004-11-01
20	William	Ross	Engineering	Software Engineer	866-123-4567	2008-07-27

Create the Source Rule

Now that we have the data in place, create a query rule called `getAllEmployees` that queries the Employee entity for all of its data. We will use `getAllEmployees` to define our Source expression for the Employee record type.

See also: Query Rules

Create the Record Type

Let's create the record type reusing the same List View Item Template and SAIL Summary Dashboard expressions that are used in the entity-backed records example above.

1. From the left navigation of the System view, select System Administrator Home > Data Management > Data Management.
2. Select the Record Types tab.
3. Click **Create Record Type**.
4. Enter the following values for the associated fields:
   - Name: Employee Directory
   - Plural Name: Employees
   - Description: List of current employees
   - Source Type: Expression
   - Source:

```
=with(
local!employees: rule!getAllEmployees(),
a!dataSubset(
data: local!employees,
identifiers: local!employees.id,
startIndex: 1,
batchSize: -1,
totalCount: count(local!employees)
)
)
```

   - Source Data Type: "Employee"
   - List View Item Template:

```
=a!listViewItem(
title: rf!firstName & " " & rf!lastName,
details: rf!title & ", " & rf!department
)
```

   - Sort Field: lastName
   - Sort Order: Ascending
   - SAIL Summary Dashboard:

```
=a!dashboardLayout(
firstColumnContents: {
a!textField(
label: "Name",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!firstName & " " & rf!lastName
),
a!textField(
label: "Department",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!department
),
a!textField(
label: "Title",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!title
),
a!textField(
label: "Contact Number",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!phoneNumber
),
a!textField(
label: "Start Date",
labelPosition: "ADJACENT",
readOnly: true,
value: rf!startDate
)
}
)
```

The Source expression uses the `getAllEmployees` query rule to extract the source data and defines a `DataSubset` using that data. Notice that we used the "record field" (`rf!`) domain to access our data for the record type definition. With service-backed records, all fields of the Source Data Type are available for use with the `rf!` domain.

The above Source expression illustrates the use of `a!dataSubset` to create a `DataSubset`. Alternatively, we could have our

`getAllEmployees` query rule return a `DataSubset` directly by passing it a paging parameter. In that case, the Source expression would simplify to look like:

```
=rule!getAllEmployees(topaginginfo(1,-1))
```

Update your Source expression with the simplified one above. To view the record type that you just created, go to Tempo and select the Employees record from the Records tab. You should see the following:

Notice that the record list is not sorted by `lastName` even though we specified `lastName` as the Sort Field in the record type definition. That's because the field specified as the Sort Field is stored in a `query` object and we need to reference that `query` object in our Source expression in order to apply the sort.

Modify the record source definition by replacing the current Source expression with the following:

```
=rule!getAllEmployees(rsp!query.pagingInfo)
```

For service-backed records, we can access the `query` object by using the "record source parameter" (`rsp!`) domain, i.e. `rsp!query`. The `pagingInfo` field of the `query` object contains the sort info that we need to apply to the record list, so the above expression passes the `pagingInfo` field as a paging parameter into the `getAllEmployees` query rule.

See also: Query Rule Paging Parameter

Go to Tempo again to view the resulting sorted record list:

Given that we are going to continue to modify our record type definition expressions, let's put them in expression rules so that each version gets saved as we modify them.

See also: Rules

Create an expression rule called `employeeRecordSource` with an input named `query` of Any Type. This is the rule that we will call in the record type Source expression. Define the rule with the following:

```
=rule!getAllEmployees(ri!query.pagingInfo)
```

Create an expression rule called `employeeRecordLVI` with inputs `firstName`, `lastName`, `title`, and `department`, all of type Text. This is the rule that we will call in the record type List View Item Template expression. Define the rule with the following:

```
=a!listViewItem(
  title: ri!firstName & " " & ri!lastName,
  details: ri!title & ", " & ri!department
)
```

Create the SAIL Summary Dashboard

You can create the record dashboard as an interface using the Interface Designer. From the Rules tab of the Designer interface, click **Create an Interface**. Here you can quickly create and modify your interface with the ability to immediately see and test the result. See also: Creating an Interface

Name your interface `employeeRecordSummaryDashboard` with give it inputs `firstName`, `lastName`, `department`, `title`, `phoneNumber` of type Text and `startDate` of type Date.

Enter the following in the design pane on the left-hand side:

```
=a!dashboardLayout(
  firstColumnContents: {
    a!textField(
      label: "Name",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!firstName & " " & ri!lastName
    ),
    a!textField(
      label: "Department",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!department
    ),
    a!textField(
      label: "Title",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!title
    ),
    a!textField(
      label: "Contact Number",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!phoneNumber
    ),
    a!textField(
      label: "Start Date",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!startDate
```

```
    )
  }
)
```

You should see the following:

Now that we have all of our expressions in rules, let's update our record type definition to call those rules:

1. Navigate to the Record Types tab of the Data Management page and select the Employee Directory record type.
2. Enter into the Source field:

   ```
   =rule!employeeRecordSource(rsp!query)
   ```

3. Enter into the List View Item Template field:

   ```
   =rule!employeeRecordLVI(rf!firstName, rf!lastName, rf!title, rf!department)
   ```

4. Enter into the SAIL Summary Dashboard field:

   ```
   =rule!employeeRecordSummaryDashboard(rf!firstName, rf!lastName, rf!department, rf!title, rf!phoneNumber, rf!startDate)
   ```

Refresh the Employees record in Tempo to ensure that the record list still displays as before and that nothing has changed.

Now that our list view looks the way we want it to, let's move on to the Summary dashboard. If you click into a few of the Employees records in Tempo to view their Summary dashboards, you'll notice that they all display information for the *first* Employee record in the record list, rather than for the *correct* Employee record. In order for each dashboard to display the information for the correct record, we need to modify our Source expression so that the information of which record the user clicked on is passed along to it.

Before we do that, let's create a query rule called `getEmployeeById` with a rule input `id` of type Integer. Set the rule to query the Employee entity with a query condition that sets the field `id` equal to the rule input `id`.

See also: Query Rules

Now open your `employeeRecordSource` rule and modify the definition with the following:

```
=with(
  local!pagingInfo: ri!query.pagingInfo,
  local!queryCondition: index(ri!query, "logicalExpression|filter|search", null),
  local!requestedId: if(
    and(index(local!queryCondition, "field", null)="rp!id", index(local!queryCondition, "operator", null)="="),
    index(local!queryCondition, "value", null),
    null
  ),
  local!employees: if(
    not(isnull(local!requestedId)),
    rule!getEmployeeById(local!requestedId, local!pagingInfo),
    rule!getAllEmployees(local!pagingInfo)
  ),
  local!employees
)
```

When a user clicks on a service-backed record in a record list to navigate to its Summary dashboard, the information about which record the user clicked on is populated in the `query` rule input of the Source rule, specifically in `query.logicalExpression|filter|search`, where `query.logicalExpression|filter|search.field` is set to `rp!id`, `query.logicalExpression|filter|search.operator` is set to `=`, and `query.logicalExpression|filter|search.value` is set to the primary key of the record (in this case the `id` field).

In the above Source expression, we store the `query.logicalExpression|filter|search` information in a local variable (`local!queryCondition`) so that we can easily reuse it throughout the expression. We then extract the `query.logicalExpression|filter|search.value` value into another local variable (`local!requestedId`) when the `field` and `operator` fields indicate that a record has been selected (as explained in the previous paragraph). This is then used to query appropriate record(s) in `local!employees`. So when the user first clicks on the Employees record type to get to the record list, `query.logicalExpression|filter|search` is null, which results in calling the `getAllEmployees` query rule to return all records. When a user clicks on a specific record, `query.logicalExpression|filter|search` is populated, which results in calling the `getEmployeeById` query rule to return the relevant record. This results in the record field values relevant to the appropriate record being shown on its Summary dashboard.

Refresh the Employees record in Tempo and select a record from the list. You should see the corresponding data for the record. For Andrew Nelson, you should see the following:

Enable Search

Users can automatically perform searches on entity-backed and process-backed records.

For service-backed records, we need to enable it by capturing the search term from the `query` object in a local variable and then using the value to configure a filter on the data.

Let's set up the Employees record type to allow searching by last name. We'll start by creating a query rule called `searchEmployeesByLastName` with a rule input `searchTerm` of type Text. Set the rule to query the Employee entity with a query condition that is met when the field `lastName` includes the rule input `searchTerm`.

Now open your `employeeRecordSource` rule and modify the definition with the following (the grey text indicates what was a part of the

previous expression so you can easily see what we added):

```
=with(
  local!pagingInfo: ri!query.pagingInfo,
  local!queryCondition: index(ri!query, "logicalExpression|filter|search", null),
  local!requestedId: if(
    and(index(local!queryCondition, "field", null)="rp!id", index(local!queryCondition, "operator", null)="="),
    index(local!queryCondition, "value", null),
    null
  ),
  local!searchTerm: index(local!queryCondition, "searchQuery", null),
  local!employees: if(
    not(isnull(local!requestedId)),
    rule!getEmployeeById(local!requestedId, local!pagingInfo),
    if(
      not(isnull(local!searchTerm)),
      rule!searchEmployeesByLastName(local!searchTerm, local!pagingInfo),
      rule!getAllEmployees(local!pagingInfo)
    )
  ),
  local!employees
)
```

We added to the expression another local variable to capture the value of `query.search.searchQuery`, which contains the search string provided by the user when they perform a search on the record list. When the user performs a search, the local variable is populated, which results in calling the `searchEmployeesByLastName` query rule to return the matching records.

Add A Facet

As seen in the examples of entity-backed and process-backed records, facets allow the list view of records to be filtered by certain criteria. A facet can have multiple facet options that represent filtering by different values of the same field on the source data. A list view can have multiple facets that represent sets of filters on different fields.

Adding facets for service-backed records is a bit different from adding facets for process-backed and entity-backed records. The Facets field for service-backed record accepts an expression.

Each employee belongs to a department, which is a field on the record, so let's add a facet with options for each of the departments.

Create a rule called `employeeRecordFacets` and define it with the following:

```
=a!facet(
  name: "Department",
  options: {
    a!facetOption(
      id: 1,
      name: "Engineering",
      filter: a!queryFilter(
        field: "department",
        operator: "=",
        value: "Engineering"
      )
    ),
    a!facetOption(
      id: 2,
      name: "Finance",
      filter: a!queryFilter(
        field: "department",
        operator: "=",
        value: "Finance"
      )
    ),
    a!facetOption(
      id: 3,
      name: "Human Resources",
      filter: a!queryFilter(
        field: "department",
        operator: "=",
        value: "HR"
      )
    ),
    a!facetOption(
      id: 4,
      name: "Professional Services",
      filter: a!queryFilter(
        field: "department",
        operator: "=",
        value: "Professional Services"
      )
    ),
    a!facetOption(
      id: 5,
      name: "Sales",
      filter: a!queryFilter(
        field: "department",
        operator: "=",
        value: "Sales"
      )
```

```
    )
  }
)
```

We created a single Facet named "Department" using the `a!facet` system function and gave it a list of options using the `a!facetOption` system function. The FacetOptions each were given a unique id and a name, which will appear in the list view as the facet option name that the user can select.

Each facet option has a filter field of type QueryFilter, which we used the `a!queryFilter` system function to define. A QueryFilter allows specifying a field, operator, and value to apply as a filter when the facet option is selected. *field* is set to `department`, *operator* is set to `=`, and *value* is set to each of the unique values in the *department* field of the record.

An advantage of service-backed records is that the Facet field accepts an expression. This means that instead of hardcoding the list of facet options, you could dynamically generate them by creating a rule that returns facet options by using the `apply()` function. This would allow you to do things like only show facet options that have one or more matching records.

You can configure multiple facets by defining the Facets expression to return an array of Facets.

See also: a!facet, a!facetOptions, and a!queryFilter

After saving the rule, you need to update the record type to use it:

1. Navigate to the Record Types tab of the Data Management page and select the Employee Directory record type.
    - The Edit Record Type page displays.
2. Scroll down to the Facets field.
3. Add the following to the field:

```
=rule!employeeRecordFacets()
```

After saving the record type, navigate back to the Employees records list in Tempo and refresh. You should see that your list view now has a Department facet with the three facet options that we added:

If you click on one of the facet options, it will be selected, but the refreshed list view still shows the full list of employees. Similar to how we needed to modify the Source expression to apply the specified sort and search, we also need to modify it to apply the facet selection.

First, let's create a query rule called `getEmployeesByDepartment` with a rule input `department` of type Text. Set the rule to query the Employee entity with a query condition that is met when the field `department` equals the rule input `department`.

Now open your `employeeRecordSource` rule and modify the definition with the following:

```
=with(
  local!pagingInfo: ri!query.pagingInfo,
  local!queryCondition: index(ri!query, "logicalExpression|filter|search", null),
  local!requestedId: if(
    and(index(local!queryCondition, "field", null)="rp!id", index(local!queryCondition, "operator", null)="="),
    index(local!queryCondition, "value", null),
    null
  ),
  local!searchTerm: index(local!queryCondition, "searchQuery", null),
  local!department: if(
    and(index(local!queryCondition, "field", null)="department", index(local!queryCondition, "operator", null)="="),
    index(local!queryCondition, "value", null),
    null
  ),
  local!employees: if(
    not(isnull(local!requestedId)),
    rule!getEmployeeById(local!requestedId, local!pagingInfo),
    if(
      not(isnull(local!searchTerm)),
      rule!searchEmployeesByLastName(local!searchTerm, local!pagingInfo),
      if(
        not(isnull(local!department)),
        rule!getEmployeesByDepartment(local!department, local!pagingInfo),
        rule!getAllEmployees(local!pagingInfo)
      )
    )
  ),
  local!employees
)
```

Similar to what we did for search, we added to the expression another local variable (`local!department`) to capture the value of the value of `query.logicalExpression|filter|search`, which contains the `QueryFilter` value set when the user selected a facet option. When the user selects a facet option, the local variable is populated, which results in calling the `getEmployeesByDepartment` query rule to return the matching records.

If you go to the record list in Tempo and click on one of the facet options, it will be selected and the list will now refresh to show only the qualifying records.

An Example Using External Data

The previous example serves to introduce the basic concepts of service-backed records. However, the Employees record type example could have been implemented as an entity-backed record type. As the name implies, service-backed records are a powerful

means to bring data from external services into Appian as records.

For this example we'll build a service-backed record type that retrieves data from a publicly available RESTful web service. New York City offers a RESTful API that provides content from their online 311 public service. We will use the API to retrieve data to display as records in the Tempo interface.

First, start by creating the data type to be used for the record type.

1. Create a custom data type called `NYC311Service` with the following fields:
   - name (Text)
   - summary (Text)
   - description (Text)
   - category (Text)
   - linkUrls (Text) - configure the field to allow multiple values
   - linkLabels (Text) - configure the field to allow multiple values
2. Save and publish the CDT.

The fields of this data type are now available for use under the rf expression context in the record type definition.

Now let's define the rule that will provide the source.

Create a new expression rule called `NYC311ServicesSource` with an input named `query` of Any Type:

```
=with(
  local!filterValue: if(
    isnull(ri!query.'logicalExpression|filter|search'),
    "finder",
    ri!query.'logicalExpression|filter|search'.value
  ),
  local!services:xpathsnippet(
    "http://www.nyc.gov/portal/apps/311_contentapi/services/" & local!filterValue & ".xml",
    "/services/service"),
    a!dataSubset(
      startIndex: 1,
      batchSize: count(local!services),
      totalCount: count(local!services),
      data: apply(
        rule!NYC311ServicesCreator,
        local!services
      ),
      identifiers: apply(
        rule!NYC311ServicesExtractID,
        local!services
      )
    )
 )
```

The `filtervalue` variable is set to the value of the `value` field of the query that is sent to the rule based on the user's facet option selection. When the list view first loads, there are no facet options selections by the user to filter the list, so the *logicalExpression|filter|search* field will be null. The updated expression handles that by returning the value `finder`, which corresponds to the default URL to retrieve the data for the records list view.

The source rule retrieves the data from the `nyc.gov` web service by using the `xpathsnippet()` function.

The *data* field calls the `apply()` function using a new rule called `NYC311ServicesCreator` and the *identifiers* field calls the `apply()` function using a new rule called `NYC311ServicesExtractID`. Let's define these rules.

Create an expression rule called `NYC311ServicesCreator` with an input named `service` of type Text. This rule instantiates the `NYC311Service` data type that we created earlier.

Define the rule with the following:

```
=type!NYC311Service(
  name: xpathsnippet(ri!service,"/service/service_name/text()"),
  summary: xpathsnippet(ri!service,"/service//brief_description/text()"),
  description: xpathsnippet(ri!service,"/service/description/text()"),
  category: xpathsnippet(ri!service,"/service/categories/category_name/text()")
)
```

Create an expression rule called `NYC311ServicesExtractID` with an input named `service` of type Text. This rule extracts the ID.

Define the rule with the following:

```
=xpathsnippet(ri!service,"/service/id/text()")[1]
```

In the definition of this rule, we use the index syntax to take the first item from the list of items returned by `xpathsnippet()`. Each service only has a single id, but the `xpathsnippet()` function always returns a list, so indexing in this way ensures we get a scalar value for each identifier. We populate the identifiers based on the values given in the element of the XML from the service.

Now let's create the expression rule that defines the List View Item Template. Create an expression rule called `NYC311ServicesLVI` with inputs `title` and `details`, both of type Text.

Define the rule with the following:

```
=a!listViewItem(
  title: ri!title,
  details: ri!details
)
```

Now that we have the basic rules that we need, let's create a record type that uses them.

1. From the left navigation of the System view, select System Administrator Home > Data Management > Data Management.
2. Select the **Record Types** tab.
3. Click **Create Record Type**.
4. Enter the following values for the associated fields:
   - Name: NYC 311 Service
   - Plural Name: NYC 311 Services
   - Description: A directory of New York City's 311 services
   - URL Stub: nyc311
   - Source Type: Expression
   - Source:

     ```
     =rule!NYC311ServicesSource(rsp!query)
     ```

   - Source Data Type: "NYC311Service"
   - List View Item Template:

     ```
     =rule!NYC311ServicesLVI(rf!name, rf!summary)
     ```

   - Sort Field: name
   - Sort Order: Ascending
   - SAIL Summary Dashboard: `{}`
5. Click **Create Record Type**.

You have just created a record based on external data from NYC's online 311 public service.

To view your records, open Tempo in a new browser, select the **Records** tab, and click NYC 311 Services. You should see the following:

Now we need a dashboard to display data about the service when the user clicks on one.

Using the Interface Designer, create an interface called "NYC311ServicesDashboardUI" with inputs named `category`, `summary`, and `description`, all of type Text.

Enter the following in the design pane on the left-hand side:

```
=a!dashboardLayout(
  firstColumnContents: {
    a!textField(
      label: "Category",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!category
    ),
    a!textField(
      label: "Summary",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!summary
    ),
    a!textField(
      label: "Description",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!description
    )
  }
)
```

You should see the following:

After saving the interface, let's set it as the SAIL Summary Dashboard for the record type:

1. Navigate to the Record Types tab of the Data Management page and select the NYC 311 Service record type.
   - The Edit Record Type page displays.
2. Scroll down to the SAIL Summary Dashboard section.
3. Enter `=rule!NYC311ServicesDashboardUI(rf!category, rf!summary, rf!description)` in the input, which will call the rule we just created and pass the appropriate record fields as the rule inputs.

Refresh the NYC 311 Services record list in Tempo and select Library Information. You should see the following:

That looks pretty good, but the REST API offers some more content than just the basic info that we could display on the dashboard. It also offers some links to other sites that can provide more info on the public service. Let's add some links.

Open your `NYC311ServicesCreator` rule. Modify the definition with the following (the grey text indicates what was a part of the previous expression so you can easily see what we added):

```
=type!NYC311Service(
  name: xpathsnippet(ri!service,"/service/service_name/text()"),
  summary: xpathsnippet(ri!service,"/service//brief_description/text()"),
  description: xpathsnippet(ri!service,"/service/description/text()"),
  category: xpathsnippet(ri!service,"/service/categories/category_name/text()"),
  linkUrls: apply(
    rule!NYC311ServicesLinkUrl,
    xpathsnippet(ri!service,"service/web_actions/web_action/url/text()/ancestor::web_action")),
  linkLabels: apply(
    rule!NYC311ServicesLinkLabel,
    xpathsnippet(ri!service,"service/web_actions/web_action/url/text()/ancestor::web_action"))
)
```

The new portion of the expression applies the new rules `NYC311ServicesLinkUrl` and `NYC311ServicesLinkLabel` to each `web_action` element in the XML that has a text body for the child `url` element.

Create a rule called `NYC311ServicesLinkUrl` with an input named `link` of type Text.

Define it with the following:

```
=xpathsnippet(ri!link,"/web_action/url/text()")
```

Create a rule called `NYC311ServicesLinkLabel` with an input named `link` of type Text.

Define it with the following:

```
=xpathsnippet(ri!link,"/web_action/label/text()")
```

Now open your `NYC311ServicesDashboardUI` interface. Add two inputs named `linkUrl` and `linkLabel`, both of type Text and set to allow multiple values. Modify the interface definition with the following:

```
=a!dashboardLayout(
  firstColumnContents: {
    a!textField(
      label: "Category",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!category
    ),
    a!textField(
      label: "Summary",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!summary
    ),
    a!textField(
      label: "Description",
      labelPosition: "ADJACENT",
      readOnly: true,
      value: ri!description
    ),
    if(count(ri!linkUrl) > 0,
      a!linkField(
        label: "Links",
        labelPosition: "ADJACENT",
        instructions: "These links provide more information regarding this service.",
        links:
          apply(
            a!safeLink(uri: _, label: _),
            merge(ri!linkUrl,ri!linkLabel)
          )
      ),
      {}
    )
  }
)
```

You will see an error displayed saying that "Only list arguments are allowed". This is because your inputs `linkUrl` and `linkLabel` allow multiple values but are currently null. You can make sure that the interface actually works by giving these inputs some non-null values.

To create the list of links the new portion of the expression uses the LinkField component.

Since we updated the Summary dashboard rule to accept two new inputs, we will need to update the field in the record type to pass in those new inputs:

1. Navigate to the Record Types tab of the Data Management page and select the NYC 311 Service record type.
    - The Edit Record Type page displays.
2. Scroll down to the SAIL Summary Dashboard field.
3. Modify the field with the following:

```
=rule!NYC311ServicesDashboardUI(rf!category, rf!summary, rf!description, rf!linkUrls, rf!linkLabels)
```

Now navigate to the list of NYC 311 Services in the Records tab in Tempo, and choose a record. If you choose a record that has links, you should see a dashboard like this:

By following the walkthrough, you've built a service-backed record that retrieves and displays data from an external service as records

in the Appian interface. You've given each record a summary dashboard.

We've only talked about this new record type in terms of the web interface, but if you pull up the Appian app on your mobile device and log into the site you've been using, you'll see that what you've built is also automatically mobile enabled and available to you wherever you go. Even on the mean streets of New York!

User Guides by Role

Designer Developer Web Admin Server Admin (On-Premise Only)

Tutorials

Records Interfaces Process

Release Information

Release Notes Installation Migration System Requirements Hotfixes Release History

Other

STAR Methodology Best Practices Glossary APIs

© Appian Corporation 2002-2014. All Rights Reserved. • Privacy Policy • Disclaimer