Search documentation

Appian

Query Recipes

Appian 7.7

The recipes on this page show how to perform common data lookups using the `a!queryEntity()` expression function.

Although the examples on this page all deal with `a!queryEntity()`, the same patterns also apply to the `queryrecord()` function, which can be used to perform data lookups against entity-backed and process-backed records.

See also:

- Create Entity-Backed Records
- a!queryEntity()
- queryrecord()

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

Setup

Before we start with the recipes, we'll need a data store entity. For our examples, let's use the employee entity from the Records Tutorials.

See also: Records Tutorial

Next, create a constant called `EMPLOYEE_ENTITY` with Data Store Entity as the Type and the Employee entity as the Value.

See also: Constants

Retrieve the Data for a Single Field

Goal: Retrieve the data for a single field of an entity rather than all of the fields.

When you execute a query rule it pulls back the data for all of the fields of the entity. The more data you pull back from the database the longer the query rule takes to run. A common way to restrict the amount of data returned by a query rule is to create several different data store entities that reference the same database table, each of which only contains some of the fields. Instead, using `a!queryEntity()` to select specific fields as shown below restricts the amount of returned data, is faster to develop, and has the advantage that the field or fields can be selected at run-time rather than design-time.

In this example we are going to retrieve the phone number, stored in the field `phoneNumber`, for the employee whose id is `8`.

Expression

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    selection: a!querySelection(
      columns: {
        a!queryColumn(
          field: "phoneNumber"
        )
      }
    ),
    filter: a!queryFilter(
      field: "id",
      operator: "=",
      value: 8
    ),
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: 1
    )
  )
).data.phoneNumber[1]
```

This example should return the value `404-987-6543`.

Toggle Us

Tu

Re

Ot

To retrieve data for more than one field, you can add additional `a!queryColumn()`'s to the `columns` array.

Get the Distinct Values of a Field

Goal: Retrieve the unique list of values in a given field.

In order to calculate the list of unique values in a field using query rules one approach is to retrieve *all* of the values for the field and then use the `union(`
function to calculate the unique list. It will almost always be significantly faster to have the data source do the uniqueness calculation before returning th
data to Appian. This is especially true for large data sets.

In this example we are going to retrieve the list of departments that have employees.

Expression

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    aggregation: a!queryAggregation(
      aggregationColumns: {
        a!queryAggregationColumn(
          field: "department",
          isGrouping: true
        )
      }
    ),
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: -1,
      sort: a!sortInfo(
        field: "department",
        ascending: true
      )
    )
  )
).data.department
```

This example should return a list containing `"Engineering"`, `"Finance"`, `"HR"`, `"Professional Services"`, and `"Sales"`. Note that even though there is mor
than one employee in many of these departments, each department is only listed once in the result.

Aggregating on a Field

Goal: Perform an aggregation or computation on all values of field.

In this example we are going to count the number of employees in each department.

Expression

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    aggregation: a!queryAggregation(
      aggregationColumns: {
        a!queryAggregationColumn(
          field: "department",
          isGrouping: true
        ),
        a!queryAggregationColumn(
          field: "department",
          alias: "numberOfEmployees",
          aggregationFunction: "COUNT"
        )
      }
    ),
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: -1,
      sort: a!sortInfo(
        field: "department",
        ascending: true
      )
    )
  )
)
```

This example should return one dictionary for each department where the keys in the dictionary are `department` and `numberOfEmployees` and the values match the following table.

department	numberOfEmployees
Engineering	6
Finance	4
HR	2
Professional Services	4
Sales	4

Querying on Multiple Conditions

Goal: Retrieve data that meets at least one of two different conditions.

Using query rules, the only way to find entries that match at least one of two conditions is to run two different query rules and combine the results. Usir a logicalExpression inside the Query object we can execute the same logic in a single call to the data source, resulting in faster performance.

In this example we are going to retrieve the names of employees who **either** started within the last 2 years **or** have the word "Associate" in their title.

Expression

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    selection: a!querySelection(
      columns: {
        a!queryColumn(field: "firstName"),
        a!queryColumn(field: "lastName")
      }
    ),
    logicalExpression: a!queryLogicalExpression(
      operator: "OR",
      filters: {
        a!queryFilter(
          field: "startDate",
          operator: ">",
          value: date(year(now())-2, month(now()), day(now()))
        ),
        a!queryFilter(
          field: "title",
          operator: "includes",
          value: "Associate"
        )
      }
    ),
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: -1
    )
  )
)
```

The exact list of results that is returned will to depend on when you run the example:

- Before Jan 2, 2015 : John Smith will be the only employee returned because of the start date condition.
- On or after Jan 2, 2015 : no employees will be included in the results because of their start date.

Elizabeth Ward , Laura Bryant and Stephen Edwards will be included in the result regardless of when you run the example as they are included because their title contains the word Associate

Querying on Nested Conditions

Goal: Retrieve data based on complex or nested conditions.

In this example we are going to retrieve the names of the senior members of the Engineering department where "senior" is defined as either having a tit of "Director" or having a start date of more than 10 years ago.

Expression

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    selection: a!querySelection(
      columns: {
        a!queryColumn(field: "firstName"),
        a!queryColumn(field: "lastName")
      }
    ),
    logicalExpression: a!queryLogicalExpression(
      operator: "AND",
      filters: a!queryFilter(
        field: "department",
        operator: "=",
        value: "Engineering"
      ),
      logicalExpressions: {
        a!queryLogicalExpression(
          operator: "OR",
          filters: {
            a!queryFilter(
              field: "startDate",
              operator: "<",
              value: date(year(now())-10, month(now()), day(now()))
            ),
            a!queryFilter(
              field: "title",
              operator: "includes",
              value: "Director"
            )
          }
        )
      }
    ),
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: -1
    )
  )
)
```

This example should return `John Smith` and `Mary Reed`. `John Smith` is included because he is a Director and Mary Reed is included because her start date is more than 10 years ago. Both of them are in the Engineering department.

Filtering for Null Values

Goal: Find entries where a given field is null.

In this example we are going to find all employees who are missing either `firstName`, `lastName`, `department`, `title`, `phoneNumber`, or `startDate`.

Expression

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    selection: a!querySelection(
      columns: {
        a!queryColumn(field: "firstName"),
        a!queryColumn(field: "lastName")
      }
    ),
    logicalExpression: a!queryLogicalExpression(
      operator: "OR",
      filters: {
        a!queryFilter(field: "firstName", operator: "is null"),
        a!queryFilter(field: "lastName", operator: "is null"),
        a!queryFilter(field: "department", operator: "is null"),
        a!queryFilter(field: "title", operator: "is null"),
        a!queryFilter(field: "phoneNumber", operator: "is null"),
        a!queryFilter(field: "startDate", operator: "is null")
      }
    ),
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: -1
    )
  )
)
```

This example does not return any results because none of the employees in our sample data are missing any of the specified fields.

Searching on Multiple Fields

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

For an example on filtering for null values, see the recipe: Filtering for Null Values.

Expression

First, create an expression rule `ucSearchEmployees` with the following rule inputs:

- lastName (Text)
- title (Text)
- department (Text)
- pagingInfo (Any Type)

Enter the following definition for the rule:

```
a!queryEntity(
  entity: cons!EMPLOYEE_ENTITY,
  query: a!query(
    logicalExpression: if(
      and(
        isnull(ri!lastName),
        isnull(ri!title),
        isnull(ri!department)
      ),
      null,
      a!queryLogicalExpression(
        operator: "AND",
        filters: {
          if(
            isnull(ri!lastName),
            {},
            a!queryFilter(field: "lastName", operator: "includes", value: ri!lastName)
          ),
          if(
            isnull(ri!title),
            {},
            a!queryFilter(field: "title", operator: "includes", value: ri!title)
          ),
          if(
            isnull(ri!department),
            {},
            a!queryFilter(field: "department", operator: "=", value: ri!department)
          )
        }
      )
    ),
    pagingInfo: ri!pagingInfo
  )
)
```

Test it out

Unlike the recipes above, this one is a rule with inputs. So rather than just getting a single result let's take a look at several different results for different rule inputs.

First, let's try not specifying any fields except for pagingInfo:

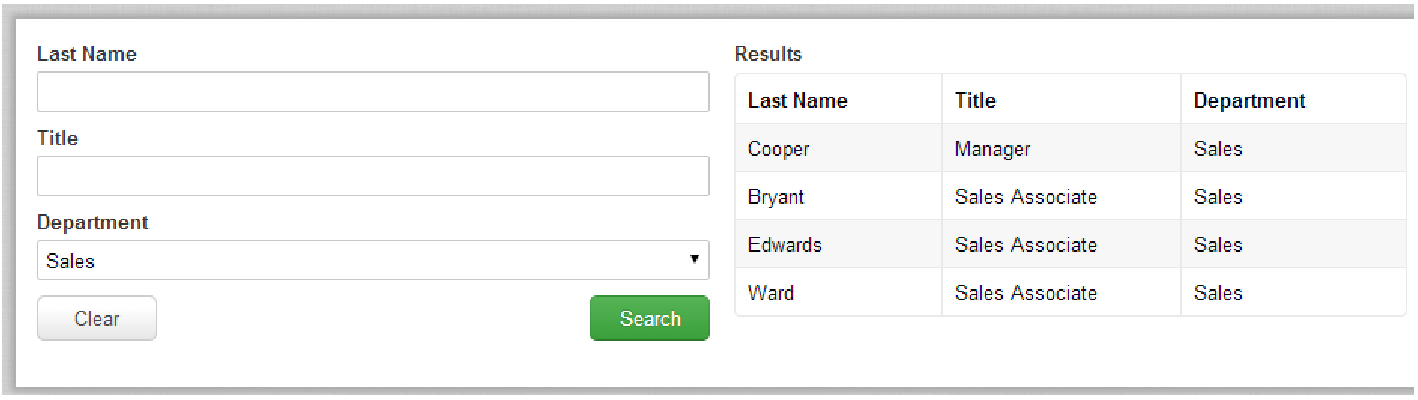
```
  rule!ucSearchEmployees(
    pagingInfo: a!pagingInfo(
      startIndex: 1,
      batchSize: 30,
      sort: a!sortInfo(
        field: "lastName",
        ascending: true
      )
    )
  )
```

This query will return the first 30 employees, sorted A-Z by last name.

Next let's try specifying a department in addition to the pagingInfo:

```
rule!ucSearchEmployees(
  department: "Sales",
  pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 30,
    sort: a!sortInfo(
      field: "lastName",
      ascending: true
    )
  )
)
```

This expression will return a list of employees in the Sales department, sorted A-Z by last name. In this example that is: Angela Cooper , Laura Bryant , Stephen Edwards , and Elizabeth Ward .



Last Name	
Title	
Department	Sales

Results

Last Name	Title	Department
Cooper	Manager	Sales
Bryant	Sales Associate	Sales
Edwards	Sales Associate	Sales
Ward	Sales Associate	Sales

We can also combine multiple filters together. Let's try searching by both last name and department:

```
rule!ucSearchEmployees(
  lastName: "Bryant",
  department: "Sales",
  pagingInfo: a!pagingInfo(
    startIndex: 1,
    batchSize: 30,
    sort: a!sortInfo(
      field: "lastName",
      ascending: true
    )
  )
)
```

This expression will return a list of employees that are in the Sales department and have a last name that contains Bryant . In this case that's a single employee: Laura Bryant .

To see an example of integrating this query into a SAIL interface, see the SAIL recipe: Searching on Multiple Fields