Search documentation Search

Appian

Create Service-Backed Records in Java

Appian 7.7

NOTE: This site does not include documentation on the latest release of Appian. Please upgrade to benefit from our newest features. For more information on the latest release of Appian, please see the Appian 7.8 documentation

The information on this page is provided by the Appian Center of Excellence

Toggle Na

This page is *not* supported by Appian Technical Support. For additional assistance, please contact your Appian Account Executive to engage with Appian Professional Services.

Background

A service-backed record is a record that sources data from an expression rule. Service-backed records are more flexible and dynamic than entity-backe and process-backed records because default filters and facets are defined with expression rules. In addition, it is possible to create dynamic facet option in a service-backed record.

Service-backed records are valuable when default filters, facet definitions, facets and/or facet options are likely to change over time.

Service-backed records can be created with existing expression rules and functions or by creating new expression functions in Java. This document explains how to use Java to implement expression rules to power a service-backed record. For information on using existing functions and rules to crea a service-backed record, see the Service-Backed Records section of the Records Tutorial.

When to write Java instead of using existing Rules and Functions for a Service Backed Record

Writing Java is necessary when there are no existing rules or functions that provide the capabilities needed to power the record. Specifically, a service-backed record type requires rules or functions to do the following:

1. Retrieve all records from the target source
2. Search and filter the list of records from the target source
3. Retrieve a specific record from a target source

For example, there are no out-of-the-box expression functions that allow a designer to query for all the groups/users in the environment. Therefore, in order to create a Record Type to represent Groups within Appian, it is necessary to write Java code to expose the Group and User information stored within the Appian engines.

Implementing Service-Backed Records in Java

Overview

There are five major steps in creating a service-backed record function in Java.

1. Create a Custom Data Type (CDT) definition in Java as a Java class. This is necessary as it provides the Java source function access to the data type fields. This new type can be added to the Appian environment by adding a `<datatype>` declaration in the plug-in's `appian-plugin.xml` file. If this type already exists in the Appian environment, there is no need to add the declaration.
2. Create a Java class that extends `DataSubset` and stores the records as objects of the data type created in step 1. Extending the `DataSubset` cla is necessary in order to provide paging and total count values for records containing more than 100 objects. This class functions as a CDT in the Appian environment, and is added via the `<datatype>` declaration in the plug-in's `appian-plugin.xml` file.
3. Create a function that returns a single instance of the object. The source expression in the **Edit Record Type** view will convert the single instanc to a DataSubset using `todatasubset()` . This function needs to be exposed in the plug-in's `appian-plugin.xml` file.
4. Create a function that returns a `DataSubset` containing a list of all the objects. This function needs to be exposed in the plug-in's `appian-plugin.xr` file.
5. Write the source expression in the **Edit Record Type** view. The source expression must alternate between the functions created in steps 3 and depending on the query.

The above list assumes that you are already experienced at creating Custom Function Plug-ins.

Walkthrough

The following sections explain how to implement the Java code for the different parts of your plug-in. An example of a service-backed record implemented in Java can be seen in the Groups Management application and plug-in available in Forum.

Creating Complex Data Types in Java

A Java class should be created to define the data type specified in the record type definition. Types must include the following XML annotation:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(namespace = <class_name>.NAMESPACE, name = <class_name>.LOCAL_PART)
@XmlType(namespace = <class_name>.NAMESPACE, name = <class_name>.LOCAL_PART)
```

`NAMESPACE` AND `LOCAL_PART` are Strings defined in the Java class:

```
public static final String NAMESPACE = "urn:appian:plugin:<class_name>:types";
public static final String LOCAL_PART = "<class_name>";
```

Each field in the Java class should also have the following annotation: `@XmlElement(name = "<field_name>")`

Code Example

See the example below on creating a CDT called `GroupProfile`:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(namespace = GroupProfile.NAMESPACE, name = GroupProfile.LOCAL_PART)
@XmlType(namespace = GroupProfile.NAMESPACE, name = GroupProfile.LOCAL_PART)
public class GroupProfile {
  public static final String NAMESPACE = "urn:appian:plugin:groupprofile:types";
  public static final String LOCAL_PART = "GroupProfile";
  public static final QName QNAME = new QName(NAMESPACE, LOCAL_PART);
  public static final String FIELD_NAME = "name";

  @XmlElement(name = FIELD_NAME)
  private String name;
  public GroupProfile(Group g) {
    this.name = g.getGroupName();
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Creating a DataSubset in Java

It is necessary to wrap the data objects into a `DataSubset` rather than returning the data itself for paging purposes. Create a class that extends `DataSubset` and use the `XmlSeeAlso` annotation to specify the list of types (classes) that the result page may hold at runtime. The subclass must defin both an `XmlType` and `XmlRootElement` annotations. See the product API for more information.

The extended `DataSubset` class should have two type parameters: `<T, I>`, where `I` refers to the identifiers.

Code Example

Follow the example below on how to create a `DataSubset` object containing data of the complex type `GroupProfile`.

```
@XmlRootElement(namespace = GroupProfileDataSubset.NAMESPACE, name = GroupProfileDataSubset.LOCAL_PART)
@XmlType(namespace = GroupProfileDataSubset.NAMESPACE, name = GroupProfileDataSubset.LOCAL_PART, propOrder = {"startIndex", "batchSize", "sort", "
totalCount", "data", "identifiers" })
@XmlSeeAlso({ GroupProfile.class })
public class GroupProfileDataSubset extends DataSubset<GroupProfile, Long> {
  private List<GroupProfile> data;
  private List<Long> identifiers;
  public static final String NAMESPACE = "urn:appian:plugin:groupprofile:types";
  public static final String LOCAL_PART = "GroupProfileDataSubset";
  private static final Logger LOG = Logger.getLogger(GroupProfile.class);
  public static final QName QNAME = new QName(NAMESPACE, LOCAL_PART);

  public GroupProfileDataSubset() {}

  public GroupProfileDataSubset(int startIndex, int batchSize,
      List<SortInfo> sort, int totalCount, List<GroupProfile> results,
      List<Long> identifiers) {
    super(startIndex, batchSize, sort, totalCount, results, identifiers);
```

```
  }

  @Override
  @XmlElement(type = Object.class, nillable = true, namespace = "")
  public List<GroupProfile> getData() {
    return data;
  }

  @Override
  @XmlElement(type = Object.class, nillable = true, namespace = "")
  public List<Long> getIdentifiers() {
    return identifiers;
  }

  @Override
  protected void setData(List<GroupProfile> data) {
    this.data = data;
  }

  @Override
  protected void setIdentifiers(List<Long> identifiers) {
    this.identifiers = identifiers;
  }
}
```

Creating the Record Source Functions

Service-backed record types call the expression defined in the Source field twice: once when loading the record list view and again when loading a single record. Note: Appian expects an array of objects when loading the record list view, but expects a single object when loading a single record (an array with a single object is not acceptable).

As a result, two distinct functions should be written in Java for handling the two cases. We will discuss each function below.

Single Record Function

To load a single record, the Java function should return an instance of the data type specified in the record type definition.

Code Example

Since the `GroupProfile` data type was created in Java, the function to load a single record should return one instance of the data type object. Follow the example below to create a single record load function.

```
@Function
public GroupProfile getGroupProfile(ServiceContext sc, final GroupService gs,
    @GroupDataType @Parameter Long id)
    throws InvalidGroupException, PrivilegeException {

  GroupProfile gp = new GroupProfile(gs.getGroup(id));
  return gp;
}
```

Record List View Function

To load the record list view, the Java function should return an instance of the extended `DataSubset` class with an array of data type objects created earlier as the data. When applying a search or facet to the record list, the framework will pass a `rsp!query` to the Source expression. Therefore the Java function should accept a `Query` object as a parameter.

Code Example

Follow the example below to create the function `GroupProfileRecordTypeSource` which takes a `Query` object as an input and which returns a `GroupProfileDataSubset` object.

The paging API method call that is used by the code generates a `ResultPage` object. The `ResultPage` object is converted into a `DataSubset`. Retrieving `ResultPage` is necessary because it allows the code to call `getAvailableItems()` to obtain the total number of records. The paging information passed to the paging API method is provided by the `Query` object through the function `getPagingInfo()`.

```
@Function
public GroupProfileDataSubset GroupProfileRecordTypeSource(
    final GroupService gs,
    ServiceContext sc,
    @Parameter @Type(namespace = Constants.TYPE_APPIAN_NAMESPACE,
    name = Query.LOCAL_PART) Query query)
    throws AppianException {

  PagingInfo pi = query.getPagingInfo();
  int numResults = pi.getBatchSize();
```

```
  List<GroupProfile> mGroupsList = new ArrayList<GroupProfile>(numResults);
  ArrayList<Long> groupIds = new ArrayList<Long>(numResults);

  GroupSearch gSearch;
  // update GroupSearch object with search and filter criteria

  int groupSortInfo;
  int groupSortOrder;
  // populate groupSortInfo using SortInfo object nested in PagingInfo object
  // populage groupSortOrder using SortInfo object nested in PagingInfo object

  ResultPage rp = gs.findGroupsPaging(gSearch, false, pi.getStartIndex() - 1, pi.getBatchSize(), groupSortInfo, groupSortOrder;

  // populate mGroupsList from the first 100 items in the ResultPage
  // populate the identifier list
  return new GroupProfileDataSubset(pi.getStartIndex(), numResults, pi.getSort(), (int) resultPage.getAvailableItems(), mGroupsList, groupIds);
}
```

Since the user can modify the record list view via changing the sort field, adding search terms, and/or selecting facets, those conditions should be applie while processing and populating the data objects list. For searching and filtering/faceting, apply the search condition before the facet conditions.

Sorting

The sort field and sort order specified by the designer in Record Types is stored in the `SortInfo` object, which is nested in the `PagingInfo` object, which i retrieved through the `Query` object. The `SortInfo` object is accessed through the function `getSort()` on the `PagingInfo` object. The sort field can be accessed by `get(0)` on the `SortInfo` object. The code example below demonstrates how to display the sort order:

```
private int getSortOrder(SortInfo selectedSort) {
  if (selectedSort.isAscending()) {
    return com.appiancorp.suiteapi.common.Constants.SORT_ORDER_ASCENDING;
  } else {
    return com.appiancorp.suiteapi.common.Constants.SORT_ORDER_DESCENDING;
  }
}
```

Searching

The `Query` object stores information on whether search terms were entered using the `hasSearch()` function. The terms are stored in a `Criteria` object that is accessed via the `getCriteria()` function. The `Criteria` object acts as a marker interface for creating a tree structure while combining filters, search and logical expressions. The search terms can be extracted with the following code snippet.

```
private String getSearchTermsIfAny(Criteria crit) {
  String searchTerms = "";
  if (crit instanceof Search) {
    Search search = (Search) crit;
    searchTerms = search.getSearchQuery();
  }

  if (crit instanceof LogicalExpression) {
    LogicalExpression le = (LogicalExpression) crit;
    if (le.getOperator() != LogicalOperator.AND) {
      throw new IllegalArgumentException("Only AND logical expressions are supported.");
    }

    List<Criteria> leCrits = le.getConditions();
    for (Criteria leCrit : leCrits) {
      String leSearch = getSearchTermsIfAny(leCrit);
      if (!StringUtils.isBlank(leSearch)) {
        searchTerms = leSearch;
        break;
      }
    }
  }
  return searchTerms;
}
```

Processing the search terms is dependent on the data. Ideally, the internal or external API used to retrieve the data should accept a search query. If no the code should iterate through the list of results and eliminate the results that do not include the search terms.

Filtering

Single filters (facets) are also stored in the `Criteria` object in the `Query` object. Multiple filters are stored as a `LogicalExpression` object that is nested within the `Criteria` object. **Multiple filters may only be combined with AND relationships**. The following code snippet demonstrates how to extrac filters into a `List<Filter>`.

```
private List<Filter> getFiltersIfAny(Criteria crit) {
  List<Filter> filters = new ArrayList<Filter>();
  if (crit instanceof Filter) {
    Filter filter = (Filter) crit;
    filters.add(filter);
  }

  if (crit instanceof LogicalExpression) {
    LogicalExpression le = (LogicalExpression) crit;
    if (le.getOperator() != LogicalOperator.AND) {
      throw new IllegalArgumentException("Only AND logical expressions are supported.");
    }

    List<Criteria> leCrits = le.getConditions();
    for (Criteria leCrit : leCrits) {
      filters.addAll(getFiltersIfAny(leCrit));
    }
  }
  return filters;
}
```

Processing the filters is dependent on the data. Ideally, the internal or external API used to retrieve the data should accept a list of filters to apply. If not, the code should iterate through the list of results and eliminate the results that do not match the filters.

Defining the Record Source Expression in Designer

The expression to define the data source of the record type in the **Edit Record Type** view of Designer should return two different `DataSubset`s depending on the query. This is determined based on the value at the index "field" of the query -- if the field contains an identifier, then the rule should return a `DataSubset` containing a single result identified by that identifier. Otherwise, the rule should return a `DataSubset` containing the record list view See the code example below for a record source expression.

```
= with(
  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
  ),

  if(isnull(local!requestedId),
    groupprofilerecordtypesource(ri!query),
    todatasubset(getgroupprofile(local!requestedId), ri!query.pagingInfo)
  )
)
```

User Guides by Role

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

Tutorials

- Records
- Interfaces
- Process

Release Information

- Release Notes
- Installation