Agent Interaction SDK Java Developer Guide

Contact

12/14/2025

Contents

Contact

Contact Information

A contact is a customer with whom the agent may interact through a media type.
Each contact has an ID, which is a unique system reference used in the Genesys Framework. The Universal Contact Server (UCS) stores the contact data, which includes names, e-mail addresses, phone numbers, and other information. This server also stores the history of a contact, that is, processed interactions.
`ContactManager` is an interface that lets you:

- Get and set contact information.
- Search for contacts.
- Add new contacts and new contact information.

To get the `ContactManager`, call the `AilFactory.getContactManager()` method as shown in the following code snippet:

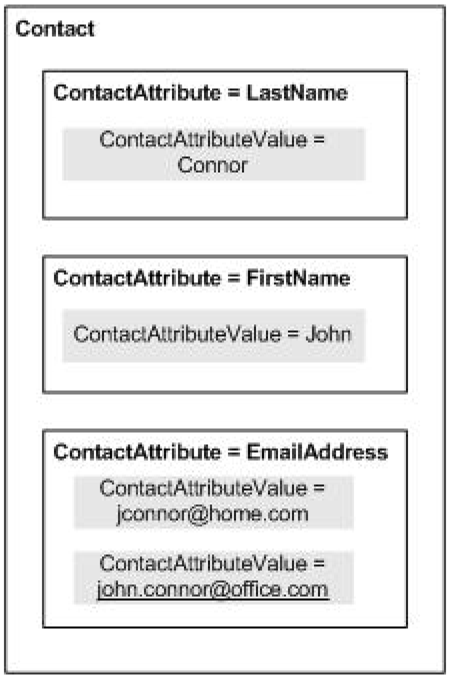
```
ContactManager myContactManager= myAilFactory.getContactManager();
```

Getting Contact Information

With the `ContactManager` , you can retrieve `Contact` objects. The `Contact` interface describes the data of a contact. To get a `Contact` instance describing a particular contact, you need the corresponding contact identifier. This identifier is available, for example, in interactions. Then, you can use the `ContactManager.getContact()` method to retrieve the contact. The following code snippet shows how to retrieve the `Contact` interface of an interaction:

```
public void handle(InteractionEvent event){
        //....
        Interaction myInteraction = event.getSource();
        String myContactID = myInteraction.getContactId();
        Contact myContact = myContactManager.getContact(myContactId();
        //....
}
```

The `Contact` interface offers access to contact attributes. A contact attribute is contact data that can have several values. For example, if a contact has one or several e-mail addresses, the e-mail address is an attribute and each e-mail address is contained in a `Contact`AttributeValue object as illustrated in Example of Contact Information.


Contact
ContactAttribute = LastName
ContactAttributeValue = Connor
ContactAttribute = FirstName
ContactAttributeValue = John
ContactAttribute = EmailAddress
ContactAttributeValue = jconnor@home.com
ContactAttributeValue = john.connor@office.com

Example of Contact Information

Default Attributes

First name, last name, phone number, e-mail address, and title are default attributes available for each contact. Those default attributes have `get` and `set` methods in the `Contact` interface for accessing directly their values. The following code snippet displays the last and first names of a contact:

```
System.out.println( myContact.getFirstName() +" "+myContact.getLastName());
```

Attributes and Metadata

The Universal Contact Server defines metadata for each type of attribute. (Attributes are defined in the Configuration Layer under `Business Attributes`.) For example, the last name is a type of contact attribute specified by metadata. For the last name attribute, the metadata specifies that the attribute name is `LastName`, the type of the attribute value is a string, the display name is `Last name`, and so on.

A single metadata is available for each type of attribute; it has a unique system identifier and a unique name. For example, a single metadata is available for all the existing last names' attribute values. The metadata is independent from the contact attributes' values.

The metadata interface is `ContactAttributeMetaData.` Call the `ContactManager.getContactAttributeMetaDataById()` or `ContactManager.getContactAttributeMetaDataByName()` to retrieve a

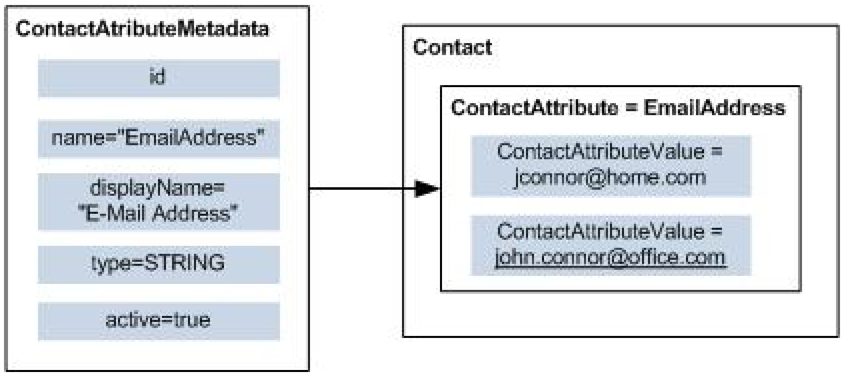
ContactAttributeMetadata interface. The following code snippet retrieves the metadata for e-mail addresses:

```
ContactAttributeMetaData myHomeAddressMetadata =
myContactManager.getContactAttributeMetaDataByName("HomeAddress");
```

For default attribute metadata, you can use a ContactManager.get*Attribute() that returns the attribute corresponding to*, as in the following example:

```
ContactAttributeMetaData myEmailAddressMetadata = myContactManager.getEmailAddressAttribute();
System.out.println(myEmailAddressMetadata.getDisplayName() );
```

For a particular contact, each ContactAttributeMetaData object is associated with a set of ContactAttributeValue objects that are available through the Contact interface. Each ContactAttributeValue object has a unique system identifier and contains a value of the contact attribute.



Attributes' Metadata and Values

Important

In the previous figure, the primary attribute value is underlined for each type of attribute.

Use the Contact.getAttributeValues() method to retrieve the contact attribute values of a contact.

```
// Getting all the contact e-mails
Collection myEmailAddresses = myContact.getAttributeValues(myEmailAddressMetadata, false);

// Displaying the string value for each ContactAttributeValue
Iterator itEMails = myEmailAddresses.iterator();
```

```
while(itEMails.hasNext())
{
   ContactAttributeValue _email = (ContactAttributeValue) itEMails.next();
   System.out.println(_email.getStrValue() );
}
```

Primary Attributes

A contact's primary attribute value is one of the attribute values marked as primary. For example, if a contact has several e-mail addresses, the e-mail address at work can be the primary e-mail attribute. For default attributes, the `Contact` interface provides you with `get/setPrimary*()` methods. For instance, the following code snippet displays the contact's primary e-mail address.

```
System.out.println(myContact.getPrimaryEmailAddress() );
```

The `ContactAttributeValue` interface includes an `isPrimary()` method that returns `true` if the value is a primary one.

Important

You cannot have two primary values for a given attribute. Your application must manage itself primary values.

Fast Contact Management

By calling the `ContactManager.findOrCreateContact()` method, you can specify key-value pairs for searching contacts, where the key is the string name of a `ContactAttribute` and the value is a string value. The method returns the IDs of matching contacts, or, if no such contact exists in the database, it creates a contact for these new parameters.

Through this method, your application can also take advantage of the Contact Server Custom Lookup algorithm (which accelerates the contact search) by adding a `MediaType` key with the `voice` or `email` value.

The following code snippet activates this algorithm and makes a fast search that creates a contact if the e-mail address is unknown.

```
// Creating the map of attribute values of the contact:
// tsmith@myCompany.com
Collection myFastContactSearch = new HashMap();

// Adding the last name attribute
myFastContactSearch.put("EmailAddress", "tsmith@myCompany.com");

// Adding the key-value pair that activates the CSCL algorithm
myFastContactSearch.put("MediaType", "email");

// Fast search
Collection result = ContactManager.findOrCreateContact(
   myFastContactSearch
);
```

Advanced Search Feature

The AIL library includes an advanced-search feature for contacts. With the `ContactManager` interface, your application can search contacts according to several attribute values and their associated metadata ID. The matching results can be sorted or truncated by sizing the returned array of results. This subsection details the steps to build an advanced search.

Creating a Contact Search Template

To search contacts, you first create a SearchContactTemplate object. Call the `ContactManager.createSearchContactTemplate()` method to get an empty instance:

```
SearchContactTemplate mySearchTemplate = myContactManager.createSearchContactTemplate();
```

Building a Filter Tree

Contact filter trees correspond to search requests that approximate SQL requests to the *Universal Contact Server* (UCS). Those filter trees are equivalent to assignment and logical expressions. For example,`(LastName="B*" and FirstName="A*")` searches for any contact whose first name begins with `A` and whose last name begins with `B`, and `(EMailAddress="ab*@company.com")` searches for any contact whose e-mail address begins with ab and finishes with `@company.com` .
Your application must build a `FilterTreeElement` object to fill the contact template. Without a filter tree, searching for a contact is not possible. A `FilterTreeElement` can be either a `FilterNode` or a `FilterLeaf` instance.
The following subsections detail the creation of filter leaves and nodes, then list the best practices for filling these elements.

Filter Leaves

A filter leaf contains a terminal expression that defines a search value for a contact attribute, such as `LastName=”B*”` or `primary Lastname=”B*”`. This means that your application can use the wildcards defined in the `FilterLeaf.LeafWildcard` enumerated type.
As the Agent Interaction Java API is able to find any occurrence, even if it includes the * character, your application creates a normalized string, as shown in the following code snippet:

```
//Creating the string B*
String leaf1Value = mySearchTemplate.normalizeSearchValue("B") ;
leaf1Value.concat(FilterLeaf.LeafWildcard.ANY.toString() );
```

For additional details about wildcards, see the Agent Interaction SDK 7.6 Java API Reference. Your application can create a filter leaf with an instance of the `FilterLeaf` class, which associates a metadata ID with a contact attribute value.
The following code snippet implements a `FilterLeaf` object for the `primary LastName=”B*”` expression:

```
// Getting the metadata
```

```
ContactAttributeMetaData myLastNameMetadata = myContactManager.getLastNameAttribute();

// Creating the leaf
FilterLeaf myLeaf = mySearchTemplate.createFilterLeaf();

// Setting the expression: "primary LastName=B*"
myLeaf.setPrimaryOnly(true)
myLeaf.setContactAttribute(myLastNameMetadata);
myLeaf.setOperator(FilterLeaf.LeafOperator.EQUAL);
myLeaf.setValue(leaf1Value);
```

To restrict the search to the primary value of the attribute, set the `primaryOnly` flag to `true` by calling the `FilterLeaf.setPrimaryOnly()` method. If your application calls the SearchContactTemplate.`setSearchPrimaryValueOnly(true)` method (see Filling the Search Template), the search does not take into account the `primaryOnly` flag of the `FilterLeaf` object.

Important

Before you set attributes for filter leaves, see details in Filling the Search Template.

Filter Nodes

A filter node contains a non-terminal expression that defines an operation for several non-terminal (nodes) or terminal (leaves) expressions. For example:

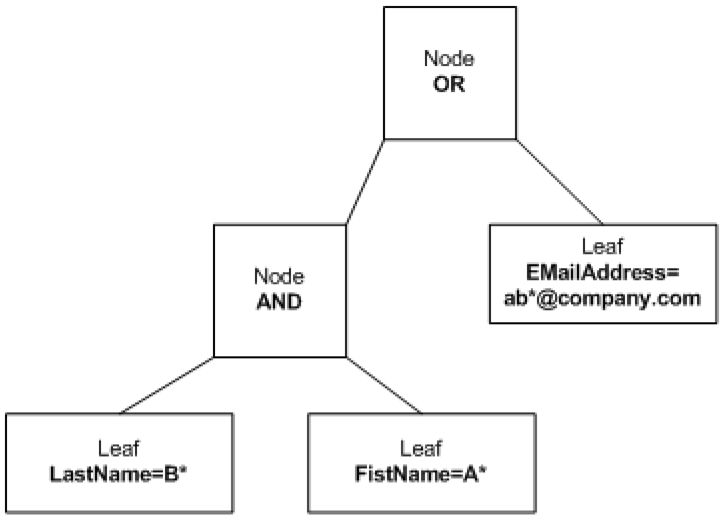
```
a or b or c

a and b and c

a or b

a and b
```

—where a, b, and c can be other filter nodes or leaves, and operators are `or` and and.

Your application can create a filter node with an instance of the `FilterNode` class, which associates an operator with a set of contact filter nodes or leaves.

The following figure presents a filter node containing the following expression:

```
(LastName="B*" and FirstName="A*") or (EMailAddress="ab*@company.com")
```



A Contact Filter Node

The following code snippet creates a `FilterNode` for the object for the `LastName="B*" AND FirstName="A*"` expression:

```
// Creating the leaf for FirstName="A*"
// Getting the metadata
ContactAttributeMetaData myFirstNameMetadata = myContactManager.getFirstNameAttribute();

// Creating the leaf
FilterLeaf myLeaf2 = mySearchTemplate.createFilterLeaf();

// Setting the expression: "FirstName=A*"
myLeaf2.setContactAttribute(myFirstNameMetadata);
myLeaf2.setOperator(FilterLeaf.LeafOperator.EQUAL);

//Creating the string A*
String leaf2Value = mySearchTemplate.normalizeSearchValue("A") ;
leaf2Value.concat(FilterLeaf.LeafWildcard.ANY.toString() );

myLeaf2.setValue(leaf2Value);
myLeaf2.setPrimaryOnly( true );

// Creating the Node
FilterNode myNode = mySearchTemplate.createFilterNode();

// Setting the operator AND
myNode.setOperator(FilterNode.NodeOperator.AND);

// Adding the leaf for "LastName=B*" (see above)
myNode.addFilterLeaf(myLeaf);

// Adding the second leaf
myNode.addFilterLeaf(myLeaf2);
```

Best Practices for Filling Your Filter Tree

The filter tree's definition determines the processing times of your search requests. There are several aspects to take into account to fit your application needs and fine-tune the building of your filter tree.

is-searchable

Genesys recommends that your application uses attributes marked as `is-searchable`. This ensures that you make calls to the appropriate UCS search algorithms and receive the most timely responses. To set up `is-searchable` attributes, open the targeted `Attribute Value` object in the `Contact Attributes` list of your Configuration Manager. In the Annex tab of the attribute object, open `settings` and set to `true` the `is-searchable` option.

In the Agent Interaction Java API, call the `isSearchable()` method of the `ContactAttributeMetadata` instance to determine whether the associated attribute values are searchable.

If your application searches for any attributes, regardless of whether they are marked as searchable, the process will be time consuming, and will slow down your application. In particular, if your application is a server, this method is inappropriate to ensure the most timely performance performances for your application and the system.

primary

If you set the `primaryOnly` flag to `true` by calling the `FilterLeaf.setPrimaryOnly()` method, you restrict the search to the attributes' primary values. The more you search for primary values, the less the SQL request is complex, and thus, UCS requires less time to return a result.

Also, for an even quicker search, you can set up the `SearchPrimaryValueOnly` flag to `true` by calling the SearchContactTemplate.`setSearchPrimaryValueOnly(true)` method. The search is restricted to attributes' primary values only, regardless of the definition of `FilterLeaf` objects in the filter tree.

Number of Attributes

The number of attributes used in the filter tree to refine your search impacts the request's processing time. The more attributes you set up, the finer your search is, but the longer the request takes.

Additionally, if you set up a wide search with few attributes or vague values, that returns a large collection of instances, and this increases the processing time as well, because it impacts the network activity. The problem is similar if you set up a great number of attributes to be returned with each matching instance: The more instances that match, the more data is returned, and this slows your response.

Filling the Search Template

A SearchContactTemplate object defines a contact search and the array of returned matching contacts. Once you have created the filter tree, fill the template by setting:

- The list of attributes to retrieve for each matching contact by passing their metadata.
- The list of attributes to use for sorting the matching contacts.
- The `SearchPrimaryValueOnly` flag.
- The filter tree.
- The number of returned results.
- The index of the first item in the matching results.

As explained in the Best Practices for Filling Your Filter Tree section, the filter tree definition impacts the time processing for receiving results, but the impact does not stop there. The list of attributes to use for sorting the matching contacts is important too. According to the number of matching results, if you set up a great number of attributes to be returned with each matching instance, you can face some network issues: the more instances that match, the more data is returned, and this slows your response.
The following code snippet fills the search template with the filter tree created in the previous section. It searches only for primary values.

```
// Results are sorted by names
mySearchTemplate.addSortAttribute(myLastNameMetadata, false);

// Contacts are returned with their last names, // their first names, and all their e-mails
mySearchTemplate.addRetrieveAttribute(myLastNameMetadata, false);
mySearchTemplate.addRetrieveAttribute(myFirstNameMetadata, false);
mySearchTemplate.addRetrieveAttribute(myEmailAddressMetadata, false);

// The results have to match the filter tree
mySearchTemplate.setFilter(myNode);

// Activating quick search
mySearchTemplate.
setSearchPrimaryValueOnly
(true);

// The first 10 contacts are returned
mySearchTemplate.setIndex(0);
mySearchTemplate.setLength(10);
```

Then, call the `ContactManager.` searchContact() method to request the contact search. It returns the matching contacts in a `Collection` . The following code snippet uses the above search template and displays the primary attributes of the matching contacts.

```
// Requesting a search for this template
Collection myMatchingContacts = myContactManager.searchContact(mySearchTemplate);
Iterator itContacts = myMatchingContacts.iterator();
while(itContacts.hasNext())
{
  Contact _Contact = (Contact) itContacts.next();
  System.out.println(_Contact.getFirstName()
   +" "+ _Contact.getLastName()
   +" "+ _Contact.getPrimaryEmailAddress());

}
```

Advanced Contact Management

With the `ContactManager` interface you can create contacts, and for each contact, the `Contact` interface allows you to add (or set) new values to attributes, remove some attribute values, merge information from another contact, or even remove the contact from the database.

Creating Contacts

You can create a contact with the `ContactManager.createContact()` method. If you only have default attributes to specify, you can create the contact in a simple call, as shown in the following code snippet:

```
Contact theCreatedContact = myContactManager.createContact("M.", "Terry", "Smith",
"tsmith@myCompany.com", "4153087723");
```

To create the contact with more attributes, you have to create `ContactAttributeValue` objects with the `ContactAttributeMetaData.createValue()` method for each attribute value of the created contact. The following code snippet shows how to proceed for the new contact `Terry Smith` .

```
// Creating the map of attribute values of the contact:
// Terry Smith, tsmith@myCompany.com
HashMap myNewContactAttributes = new HashMap();

// Creating the last name attribute
ContactAttributeValue myLastNameValue= myLastNameMetadata.createValue("Smith");
myLastNameValue.setPrimary(true);
myNewContactAttributes.put(myLastNameMetadata, myLastNameValue);

// Creating the first name attribute
ContactAttributeValue myFirstNameValue= myFirstNameMetadata.createValue("Terry");
myFirstNameValue.setPrimary(true);
myNewContactAttributes.put(myFirstNameMetadata, myFirstNameValue);

//...
//... Creating
// Creating the contact:
Contact theCreatedContact = myContactManager.createContact(myNewContactAttributes);
```

Adding Attribute Values

For any contact, you can add new values to an attribute (defined in the Configuration Layer) of the contact with the `Contact.setAttributeValues()` method. Create `ContactAttributeValue` objects with the `ContactAttributeMetaData.createValue()` method for each new attribute value. After a call to `Contact.setAttributeValues()` , propagate the contact changes in the database by calling the `Contact.save()` method, as shown below.

```
//Creating a collection of e-mail values
ArrayList myOtherEMails = new ArrayList();
ContactAttributeValue email1 =
myEmailAddressMetadata.createValue("terry.smith33@webmail.com");
myOtherEMails.add(email1);
ContactAttributeValue email2 = myEmailAddressMetadata.createValue("tsmith33@webmail.com");
myOtherEMails.add(email2);
```

```
// Setting the new values
theCreatedContact.setAttributeValues(myEmailAddressMetadata, myOtherEMails);
// Updating the database with changes
theCreatedContact.save();
```

If a `ContactAttributeValue` instance has a non-null identifier, the setAttributeValues() method updates the value corresponding to the identifier.

Contact History

The *Universal Contact Server* (UCS) stores contacts' data, including the contact history. The history manager gives access to a set of contact histories managed by the UCS. For each contact, its history contains a set of interactions involving the contact. Within the history manager, your application can retrieve summaries for a set of interactions.
To get an interface for the history manager, call the `AilFactory.getHistoryManager()` method as shown in the following code snippet:

```
HistoryManager myHistoryManager= myAilFactory.getHistoryManager();
```

For each contact, the `HistoryManager` can retrieve an `History` object. This `History` object contains a list of `History`Item s that are interaction summaries.
First, you create and fill a `SearchInteractionTemplate` object. The history manager uses this template to:

- Set the size of the `HistoryItem` list.
- Set the index of the first item retrieved with this list.
- Set the interaction attributes to retrieve.
- Set the interaction attributes to sort interactions.

The `HistoryManager` interface includes three default interaction attributes that are only used for `HistoryItem` sorting: the interaction subject, the owner identifier corresponding to an agent ID (or username), and the start date of the interaction. Those attribute values are always available in the get methods of a `HistoryItem` .
For each default interaction attribute, you can get the corresponding `InteractionAttributeMetaData` interface with three dedicated methods of the history manager:

```
HistoryManager.getInteractionAttributeMetaDataForSubject()
HistoryManager.getInteractionAttributeMetaDataForOwnerId()
HistoryManager.getInteractionAttributeMetaDataForStartDate()
```

You can also get other `InteractionAttributeMetaData` with the `InteractionManager.getInteractionAttributeMetaDataById()` or `InteractionManager.getInteractionAttributeMetaDataByName()` methods.
Those metadata correspond to the interaction attributes defined in the Configuration Layer.

Important

If you add those metadata in the list of retrieved attributes, they are available in the attached data of the item.

The following code snippet retrieves the first ten history items of a contact history, sorted by subject:

```
// Creating the search template
SearchInteractionTemplate myTemplate = myHistoryManager.createSearchInteractionTemplate();
// Setting list characteristics
// The ten first history items are retrieved
myTemplate.setLength(10);
myTemplate.setIndex(0);
// Getting the subject metadata for sorting interactions
InteractionAttributeMetaData mySubjectMetaData =
myHistoryManager.getInteractionAttributeMetaDataForSubject();

// The history items are sorted by subject
myTemplate.addSortAttribute(mySubjectMetaData,false);

// Getting the history containing the archived interactions
// and fulfilling the template
History myHistory = myHistoryManager.getHistory(myContact, myTemplate, true);

//Displaying the History Content
List myHistoryItems = myHistory.getHistoryItems();
Iterator itItems = myHistoryItems.iterator();
while(itItems.hasNext())
{
   HistoryItem myItem = (HistoryItem) itItems.next();
   System.out.println(myItem.getSubject());
   System.out.println(myItem.getDateCreated());
}
```