Agent Interaction SDK Java Developer Guide

E-Mail Interactions

12/19/2025

Contents

E-Mail Interactions

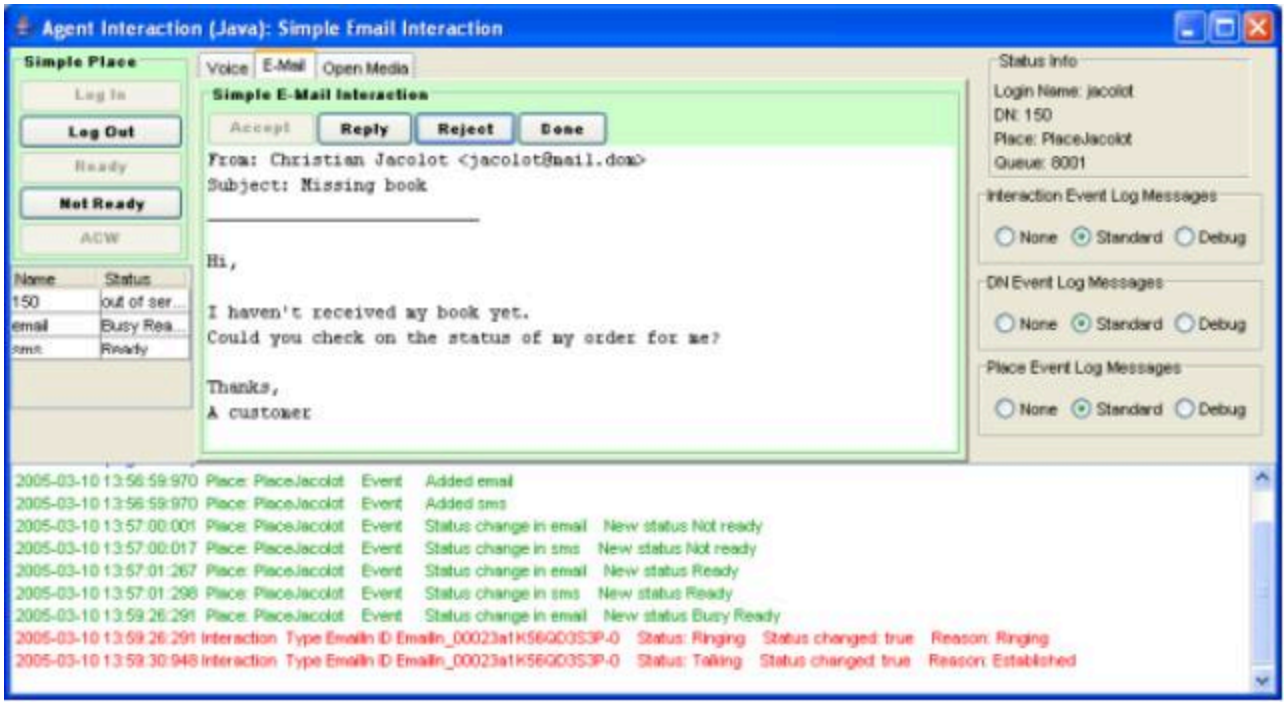
Multimedia interactions are interaction interfaces inheriting the `InteractionMultimedia` interface, that is, common e-mails, collaborative e-mails, chat interactions, and open media interactions. This chapter presents `SimpleEmailInteraction`, a code example that lets a user receive and answer e-mails. It also covers collaborative e-mails and workbin interactions.

SimpleEmailInteraction

This example is similar to `SimpleVoiceInteraction`, which was introduced earlier. It uses the same graphical user interface and the same internal structure, inheriting from `SimplePlace`.

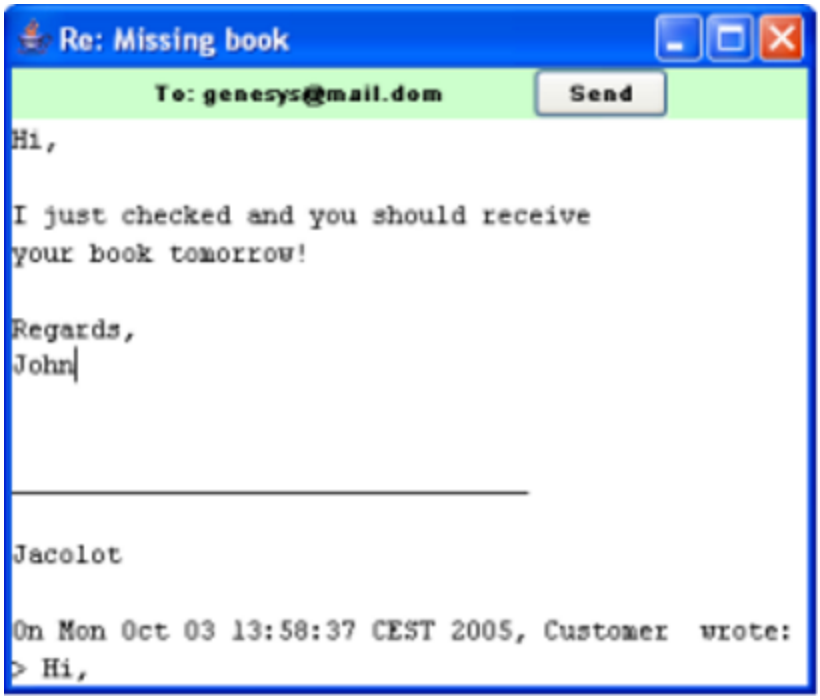
> **Important**
>
> For the sake of simplicity, this example is designed to handle one e-mail at a time. Set up a capacity rule limiting the agent to a single e-mail at a time in your routing strategy. For further details, see *Universal Routing 7.6 Documentation*.

When an e-mail arrives for this agent, the `Accept` button will be enabled. At this point, the agent can accept the e-mail. This displays the text of the incoming e-mail, and also enables the `Reply` button, as shown in Accepting an E-Mail.



Accepting an E-Mail

If the agent clicks the `Reply` button, a new window is created for the agent to type and send a response, as shown in The E-Mail Reply Window.



The E-Mail Reply Window

After the agent sends the e-mail, the processing of the inbound e-mail is finished and the example releases all the associated interactions.
As in the `SimplePlace` and `SimpleVoiceInteraction` examples, there are only six steps to follow in writing this application. As some of these steps have been done by `SimplePlace`, which is the superclass for this example, they will not be repeated here.

Set Up Button Actions

After calling the superclass method and setting up the tab, the `linkWidgetsToGui()` method is ready to set up the buttons. The `Accept` button is very simple. All it does is "answer the call" when an e-mail comes in:

```
sampleEmailIn.answerCall(null);
```

This changes the status of the incoming e-mail from `RINGING` to `TALKING`.
The `Reply` button issues a call to the `createReply()` method of `SimpleEmailInteraction`. This method first creates an outbound reply e-mail:

```
// Getting a queue to which to send the reply email.
String outboundQueue = getOutboundQueues();
sampleEmailReply = sampleEmailIn.reply(outboundQueue, false);
```

Then it makes call to an `agentInteractionGui` method, as shown here:

```
// Create a dialog box to enable the user to enter the reply text.
// The subject, the addresses, and part of the message are
// already created in the interaction.
agentInteractionGui.createReplyWindow( sampleEmailReply.getSubject(),
sampleEmailReply.getFromAddress().toString(),
sampleEmailReply.getMessageText());
```

This method creates a new window allowing the agent to enter a reply to the incoming e-mail in the widget and include a Send button. As for the`linkWidgetsToGui()` method, the `createReply()` method links widgets for managing the sending of the e-mail response.

```
// Linking widgets
sendButton = agentInteractionGui.sendButton;
replyWindow = agentInteractionGui.replyWindow;
responseTextArea = agentInteractionGui.responseTextArea;
```

When the agent has finished the reply, he or she can click the Send button in the reply window, which will do several things, as described here.
First, you will need to get the name of a queue. This is because you are about to send an e-mail interaction, and e-mail interactions must have a queue to go into. After getting a queue name, which will be explained in a moment, you must set the text of the outgoing e-mail based on the text entered by the agent:

```
//Get the queue
String outboundQueue = getOutboundQueues();
// Set the message text to the reply
sampleEmailReply.setMessageText(responseTextArea.getText());
```

Finally, you can send the response and close the reply window.

```
// Send the response
sampleEmailReply.send(outboundQueue);
replyWindow.dispose();
```

As mentioned above, you must supply a queue for the outgoing e-mail. There are several ways to do this. In this example, the available queues are read in and the first one is chosen, as shown in the next stretch of code. You may need to use more sophisticated means to perform this task.

```
String queueId = "";
Collection availableQueues = sampleEmailIn .getAvailableQueuesForChildInteraction();
Iterator iterator = availableQueues.iterator();
Queue queue;
while (iterator.hasNext()) {
   queue = (Queue) iterator.next();
   queueId = queue.getId();
   break;
}
return queueId;
```

That is a lot more work that you had to do in the previous examples, but there is not much left to do now.

Add Event-Handling Code

The `handleInteractionEvent()` method in this example is different from the corresponding method in `SimpleVoiceInteraction`. The significant difference is in how it processes two different types of e-

mail interactions: sampleEmailIn for an incoming e-mail and sampleEmailReply for the response.

> **Important**
>
> As explained in Threading section, you should write short and simple event handlers to avoid delaying the propagation of events.

In this purpose, the `SimpleEmailInteraction` uses `EmailInteractionEventThread` class to process `InteractionEvent` events. All the event processing is performed in the `run()` method of the thread.

```
public void handleInteractionEvent(InteractionEvent event) {
   super.handleInteractionEvent(event);
   EmailInteractionEventThread p = new EmailInteractionEventThread(event);
   p.start();
}
```

When an e-mail comes in, it will have a status of `RINGING`. At this point, if the `sampleEmailIn` interaction associated with the example is `null`, the example gets the event's e-mail interaction and displays a few details in the GUI, as shown here:

```
if (sampleEmailIn == null
   && event.getSource() instanceof InteractionMailIn
   && event.getStatus() == Interaction.Status.RINGING) {

   sampleEmailIn = (InteractionMailIn) event.getSource();

   //Displaying From and Subject fields
   String emailText = "From: "
      + sampleEmailIn.getFromAddress().toString()
      + "\nSubject: " + sampleEmailIn.getSubject();
   inboundEmailTextArea.setText(emailText);
}
```

At this point, if the received event provides notification of a change of status for the `sampleEmailIn` interaction, the `handleInteractionEvent()` method checks for two cases:

- If the e-mail is in `TALKING` status, the agent agreed to process it; `sampleEmailInteraction` reads information from the incoming e-mail and places it in the GUI by calling the displaySampleEmailIn() method.
- If the e-mail is in `IDLE` status, it marks it done (if necessary) and removes it from the example.

```
//The event involves the inbound email being processed
if(sampleEmailIn!= null
&& event.getSource().getId() == sampleEmailIn.getId())
{
   // The inbound email is in talking status, and can be displayed
   if (event.getStatus() == Interaction.Status.TALKING)
   {
      sampleEmailIn = (InteractionMailIn) event.getSource();
      displaySampleEmailIn();
   }
   // The interaction is processed,
```

```
   // the sample no longer needs to handle it
   else if (event.getStatus() == Interaction.Status.IDLE )
   {
      if(!sampleEmailIn.isDone())
      sampleEmailIn.markDone();
      sampleEmailIn = null;
   }
   setInteractionWidgetState();
}
```

Then, the event can also provide notification of a change of status for the sampleEmailReply interaction. In this case, the method checks whether or not this e-mail is released. If `true`, this interaction and its widgets are removed from the example, as shown here.

```
else if(sampleEmailReply!= null && event.getSource().getId() == sampleEmailReply.getId())
{
        if (event.getStatus() == Interaction.Status.IDLE )
        {
                if(!sampleEmailReply.isDone())
                        sampleEmailReply.markDone();
                sampleEmailReply = null;
                sendButton = null;
                replyWindow = null;
        }
        setInteractionWidgetState();
}
```

Synchronize the Widgets

As said previously, the standalone code examples use two similar methods to synchronize their user interface widgets with the application state: `setPlaceWidgetState()` and `setInteractionWidgetState()` .
`SimpleEmailInteraction` overwrites both methods:

- `setPlaceWidgetState()` to update login buttons in the `Simple Place` panel, according to the e-mail media's possible actions.

```
loginButton.setEnabled(sampleEmail.isPossible(Media.Action.LOGIN));
```

- `setInteractionWidgetState` to update the interaction buttons in the `SimpleE-Mail Interaction` panel, according to the actions available on the e-mail inbound interaction.

```
acceptButton.setEnabled(sampleEmailIn.isPossible(InteractionMailIn.Action.ANSWER_CALL));
```

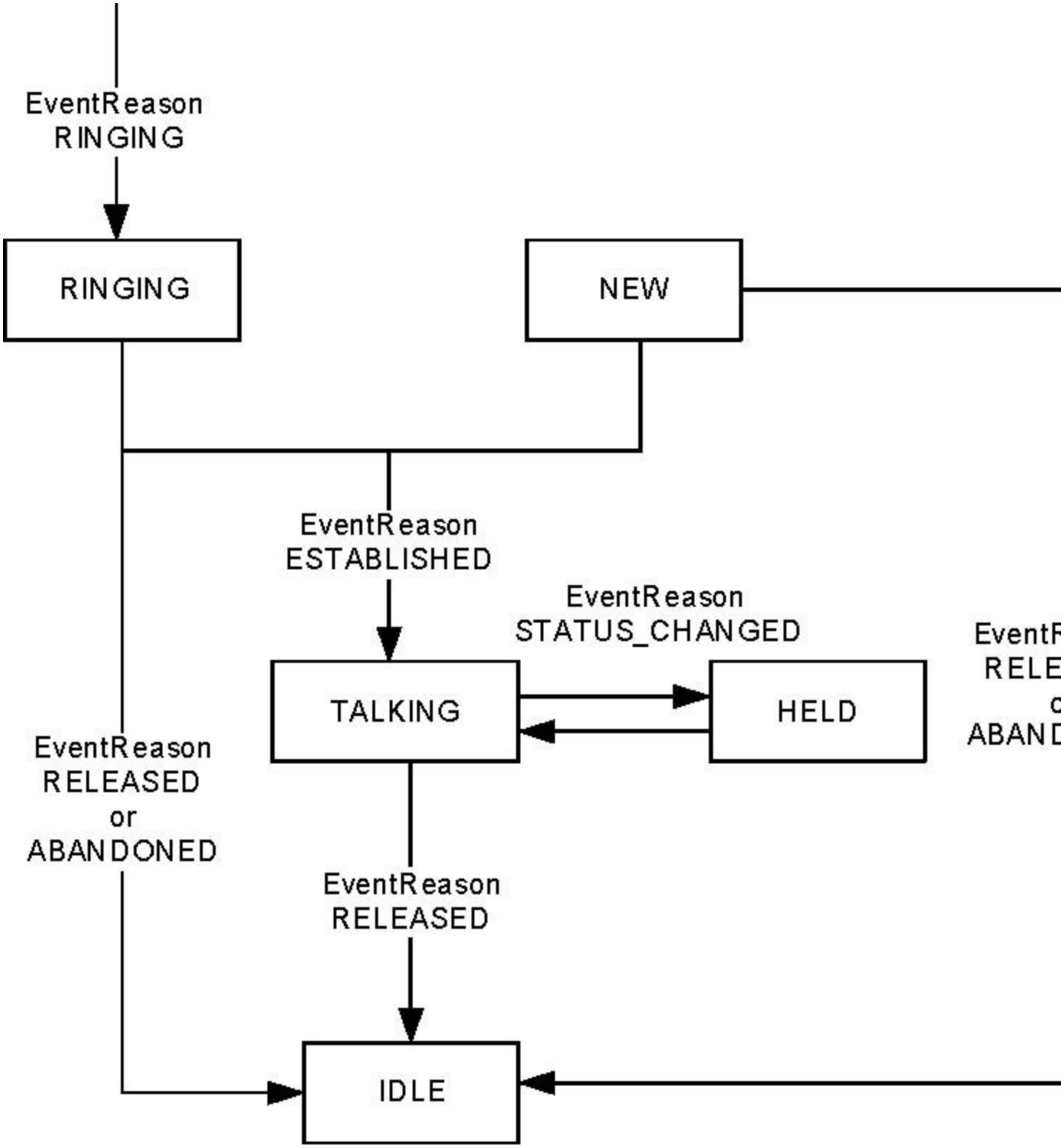
As you can see, the work you do to reply to an e-mail is not that different from what you do to handle a call. The following sections go into more detail about e-mail interactions.

Handling an E-Mail Interaction

To access e-mail interactions, an agent must log into the e-mail media of his or her place. Due to consolidation of the interactions, some method names and events applied to the manipulation of e-mail are drawn from telephony terminology.

E-Mail State

The e-mail state diagram below, E-Mail State, shows the life cycle of an e-mail interaction (inbound or outbound). The workflow is similar to that of a voice phone call, but without pure voice operations. States are `Interaction.Status` values and transitions are `InteractionEvent.EventReason` values.


EventReason
RINGING
RINGING
NEW
EventReason
ESTABLISHED
EventReason
STATUS_CHANGED
TALKING
HELD
EventReason
RELEASED
or
ABANDONED
EventReason
RELEASED
IDLE

E-Mail State

Sending an E-Mail

To send an e-mail, create an outgoing e-mail interaction, that is, an `InteractionMailOut` instance of type `Interaction.Type.EMAILOUT`.

Creating an e-mail is similar to creating a phone call. You first create an interaction with the `Agent.createInteraction()` (or `Place.createInteraction()`) method with type `EMAIL`, as shown in the following code snippet:

```
AilFactory factory = AilLoader.getAilFactory();
Agent agent = (Agent) factory.getPerson( AgentId );
Place place = (Place) sAF.getPlace(PlaceId);

HashSet myMedia = new HashSet();
myMedia.add(MediaType.EMAIL.toString());
agent.loginMultimedia(place, myMedia, ActionCode.Type.LOGIN.toString(),
"Login e-mail" );

//...

InteractionMailOut mailOut =
(InteractionMailOut) agent.createInteraction(MediaType.EMAIL, null, Queue);
```

Then, you immediately receive an `InteractionEvent` event, setting your interaction to the `Interaction.Status.TALKING` . You are now ready to fill in the e-mail with the methods of `InteractionMailOut` .
To set addresses, you must create and fill an `EmailAddress` object for each e-mail address, as shown here:

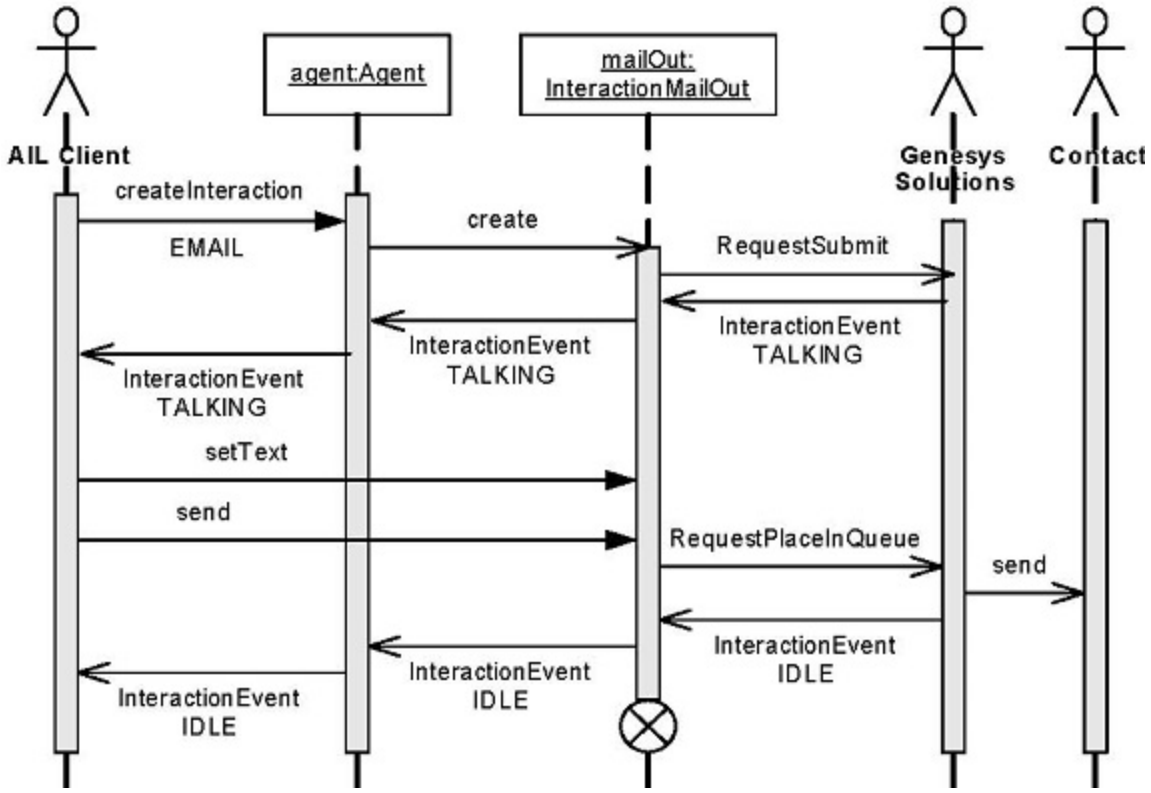
```
EmailAddress[] myAddresses = new EmailAddress[1];
myAddresses[0] = sAF.createEmailAddress("My Contact","myContact@company.com");


mailOut.setToAddresses( myAddresses );
mailOut.setSubject( "Your Subject" );
mailOut.setMessageText( "Your BodyText" );
```

Once finished, you call the InteractionMailOut.`send()` method to send your e-mail to the Genesys servers.

```
mailOut.send( Queue ); //you have to specify the queue
```

Invoking `send()` automatically releases your interaction. You receive an `InteractionEvent` event propagating the interaction status change to `Interaction.Status.IDLE`.
The following diagram presents event flow for sending an e-mail.



Sending an E-Mail

Receiving an E-Mail

When your agent receives an incoming e-mail, your application receives an `InteractionEvent` event, giving an interaction of type `Interaction.Type.EMAILIN` with the status `RINGING` . This interaction corresponds to the incoming e-mail and is available through the `InteractionMailIn` interface.

```
void handleInterationEvent( InteractionEvent event ) {
InteractionMailIn mailIn =
(InteractionMailIn) event.getSource();
// ...
}
```

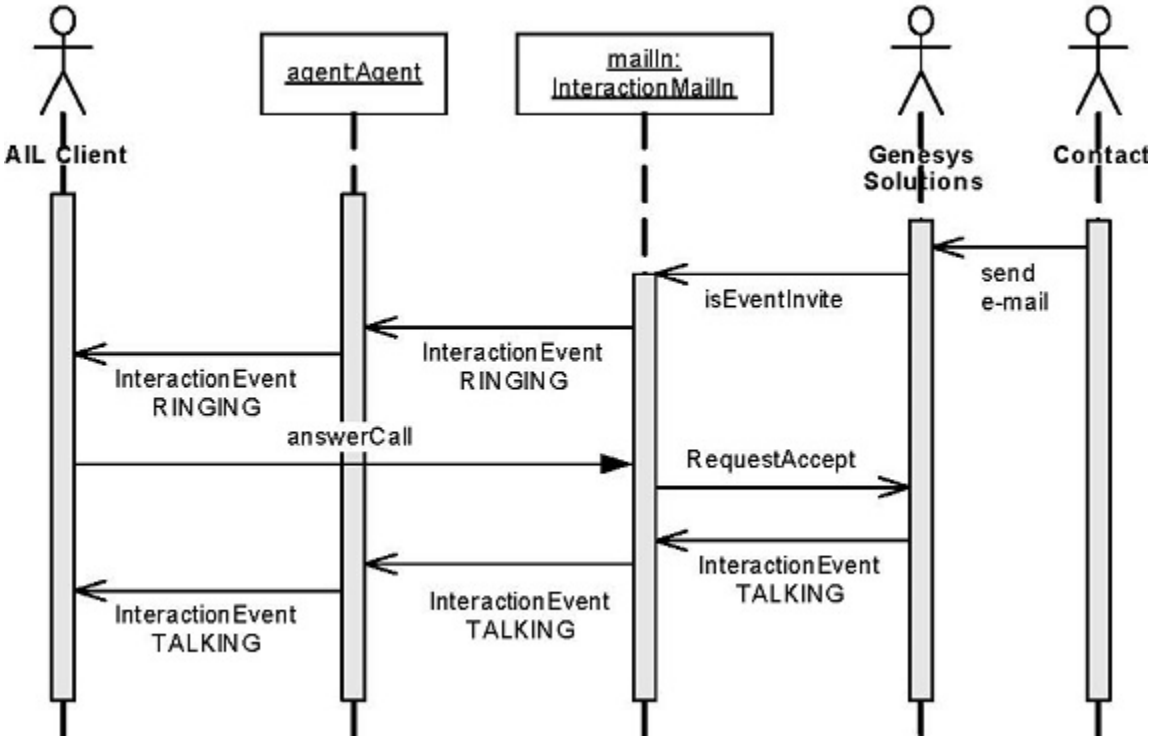
After invoking the `answerCall` () method to acknowledge the handling of the interaction, you receive another `InteractionEvent` event propagating the status of the interaction changed to `TALKING`.

```
mailIn.answerCall( null );
```

You can now access the incoming e-mail content.

```
System.out.println( "\nFrom: "+ mailIn.getFromAddress().toString() +"\nSubject: "+
mailIn.getSubject() +"\nText: "+ mailIn.getMessageText());
```

The following sequence diagram presents event flow for sending an e-mail.



Answering an E-Mail

Responding to an E-Mail

Responding to an e-mail is as straightforward as sending one. You can call the `InteractionMailIn.reply()` method to initiate a reply if the `InteractionMailIn.Action.REPLY` is available; you test this by invoking `isPossible()`. In the following code snippet, the Agent Interaction Java API creates a new interaction of type `Interaction.Type.EMAILOUT_REPLY` in `TALKING` status, and sends it to you through an event `InteractionEvent`, or as a result of the method.

```
//Create a reply, with auto mark done of the inbound e-mail
InteractionMailOut mailOutReply = mailIn.reply( Queue, false, true);
```

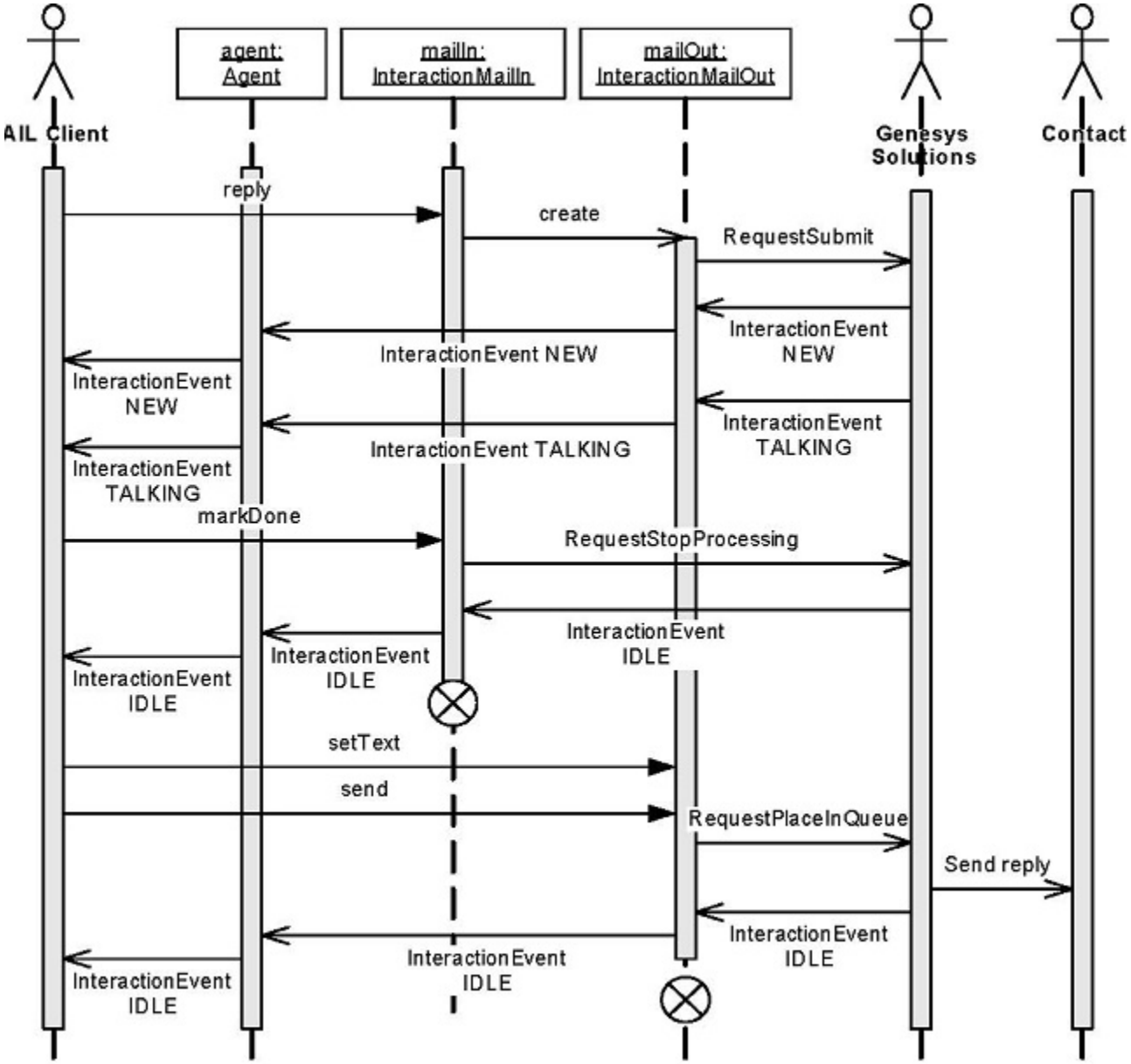
Above, the `InteractionMailIn` instance is automatically marked done, so it is no longer available. If you wish to keep alive the `InteractionMailIn` instance for making further replies, you set the `auto-mark-done` parameter to `false`. Later, to terminate this interaction, your application marks it done, as shown here:

```
//Create a reply, with no auto mark done of the e-mail in
InteractionMailOut mailOutReply = mailIn.reply( Queue, false, false);

//...

mailIn.markdone();
```

The `InteractionMailOut` reply is handled exactly in the same way as sending an e-mail. As soon as your application calls the `InteractionMailOut`.send() method, AIL places the request in the specified queue and terminates the `InteractionMailOut` reply. Your application does not need to mark done this interaction.



Responding to an E-Mail

Handling Collaborative E-Mail Interactions

When an agent is working on an outgoing e-mail, he or she can request the collaboration of other agents to elaborate the e-mail content.
A collaboration session involves several types of collaborative interactions. A collaborative interaction is an e-mail interaction that manages additional collaboration data.
During a collaboration session, your application can:

- Initiate a collaboration session on an outgoing e-mail.
- Participate in a collaboration session.

When an agent initiates the collaboration, he or she sends invitations to the participants. By periodically refreshing the status of the invitations, your application can monitor the participants' collaboration activity.
If the agent is a participant, your application manages `InteractionEvent` events for `Interaction.Status` changes and performs collaboration actions both on the invitation and on the reply that is sent as a result of the collaboration.
Classes and interfaces dedicated to the collaboration are part of the `com.genesyslab.com.ail.collaboration` package.

Types of Collaborative E-Mail Interactions

Types for collaborative e-mail interactions are accessed with the inherited `Interaction.getType()` method. The following table presents the types of collaborative e-mail interactions that your application can deal with.

Types of Collaborative E-Mail Interactions

Interactions	Interaction.Type	Interface	Description
Parent invitation	`COLLABORATION_INVIT_OUT`	`InteractionInvitationOut`	Interactions for sending invitations to participants.
Invitation seen by the parent	`COLLABORATION_INVIT_IN`	`InteractionInvitation ParentIn`	Interactions for invitations sent by the agent (or parent) who requested collaboration.
Incoming invitation	`COLLABORATION_INVIT_IN`	`InteractionInvitationIn`	Interactions for incoming invitations received by the participant (or child).
Reply to invitation	`COLLABORATION_REPLY_OUT`	`InteractionReplyOut`	Interactions for collaborative

Interactions	Interaction.Type	Interface	Description
			replies sent by participants (or children).

Outgoing Invitations

As mentioned in the Agent Interaction SDK (Java) API Reference, the collaborative `InteractionInvitationOut` interaction is used to set a list of participants in the collaboration session. This interaction creates and sends incoming invitation interactions to each participant on this list. You need a single `InteractionInvitationOut` interaction to send invitations to several participants.

> ### Warning
> As soon as invitations are sent, the outgoing invitation is no longer available. Nor is it stored in the database. Do not keep any reference on this object.

Invitation

Invitation interactions are inbound e-mails of type `COLLABORATION_INVIT_IN` . Depending on your application's role in the collaboration (child or parent), the application provides you with two interfaces to handle invitations:

- `InteractionInvitationIn` —child invitation:
  - Each participant in the collaboration session receives an incoming child invitation.
  - This interaction informs the participant of the collaboration request.
  - For information about child invitation management, see Participating in a Collaboration Session.
- `InteractionInvitationParentIn` —parent invitation:
  - For each invitation sent to a participant, the agent who initiates the collaboration can access the corresponding invitation interaction.
  - The agent uses parent invitations to monitor the collaboration and the participants' replies.
  - Each invitation is a child interaction of the initial outgoing e-mail interaction.
  - Use `InteractionMailOut.getSentInvitations()` to retrieve parent invitations.
  - For information about parent invitation management, see Handling a Collaboration Session.

Collaborative Reply

The collaborative reply is an outgoing e-mail replying to a child incoming invitation. It is created by calling the `InteractionInvitationIn.reply()` method. As for an outgoing replying e-mail, some e-mail fields are already filled in and you just have to set new text with

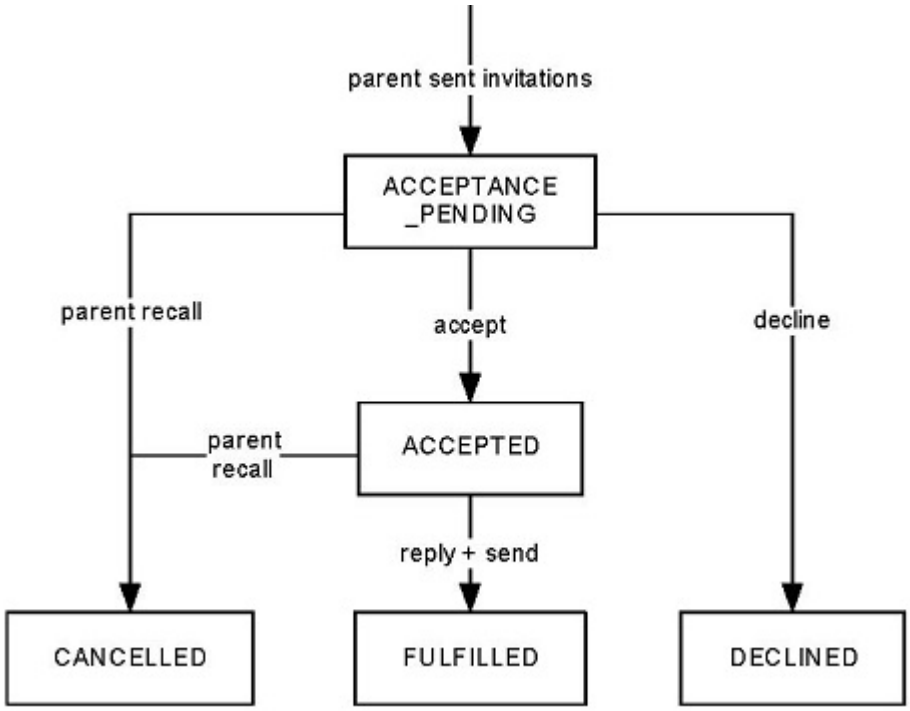
`theInteractionReplyOut.setMessageText()` method.
Use a collaborative reply in the same way that you would use a reply outgoing e-mail interaction.
For further details, see Participating in a Collaboration Session.

Collaboration Status

Collaborative interactions all have a collaboration status—described in the `InteractionInvitationIn.Status` inner class—available through the `getCollaborationStatus()` method.
Applications that initiate the collaboration have specific interests in the collaboration status of their parent invitations.When a parent invitation gets a `FULFILLED` collaboration status, the application can get the reply corresponding to the invitation by calling the `getCollaborativeReply()` method.

> **Important**
> Collaboration status changes do not launch additional `InteractionEvent` events. There is no notification of modifications through the API. Periodically read the collaboration status to check any status change.

The following state diagram shows non-exhaustive transitions that exist between collaboration statuses of an incoming invitation.



Transitions Between Statuses of an Incoming Invitation

> **Warning**
> This figure is provided as an informative example. It does not include all possible statuses and transitions.

You must take into account an invitation's collaboration status to perform collaboration actions.

Handling a Collaboration Session

From the parent's point of view, the collaboration feature has been designed as a set of invitations that are children of the outgoing e-mail interaction that requests a collaboration session.
This section discusses first the sending of invitations to participants, then the management of the parent invitations.

Sending Invitations

First, create a draft of the relevant outgoing e-mail interaction. You need a single InteractionInvitationOut interaction to send invitations to several participants.
Use the `InteractionMailOut.createCollaborationInvitation()` method as shown in the following code snippet:

```
InteractionInvitationOut myInvitationsToSend =
myInteractionMailOut.createCollaborationInvitation();
```

You have to specify the reason for this collaboration session. Use `setSubject()` and `setMessageText()` to add information.
You must add participants to the collaboration session. First, create each participant with the `InteractionInvitationOut.createParticipant()` method, then add those participants to the outgoing invitation with the `InteractionInvitationOut.addParticipant()` method, as shown in the following code snippet:

```
Participant myParticipant0 = myInvitationsToSend.createParticipant();
// Setting type and name of the participant that is agent 0.
myParticipant0.setType(QUEUE);
myParticipant0.setName("agent0");

// Adding agent0 to the list of participants
myInvitationsToSend.addParticipant(myParticipant0);
```

To determine if those participants have been correctly added to the outgoing invitation, you can implement a `ParticipantsListener` listener and add it with the `InteractionInvitationOut.addParticipantsListener()` method.
Once all participants have been added to the outgoing invitation, you can send them invitations. Depending on the method called to send invitations, your application activates a specific mode:

- `send()` —Sends the invitations to the participants in pull mode. The child invitations are available in workbins. See Putting Interactions in Workbins.
- `transfer()` —Transfers the invitations to the participants in push mode. Each participant receives the invitation as an incoming interaction.

The following code snippet uses the `transfer()` method:

```
// Use transfer in the push mode
myInvitationsToSend.transfer();
```

In push mode, on the child side, each participant receives a `RINGING InteractionEvent` for the corresponding `InteractionInvitationIn` . See Participating in a Collaboration Session.

Managing Parent Invitations

For each participant to the collaboration session, you manage a corresponding `InteractionInvitationParentIn` interaction. You can get the set of parent invitations by calling the `InteractionMailOut.getSentInvitations()` method of the outgoing e-mail interaction involved in the collaboration session.

Parent Actions

You can perform parent-specific actions on the invitations, that is, reminding participants about invitations or recalling invitations.
Those parent actions are available in the`InteractionInvitationParentIn.Action` inner class. To determine if the `RECALL` and `REMIND` actions are available, test the collaboration status of the invitation as shown in the following code snippet:

```
if((myInteractionInvitationParentIn.getCollaborationStatus() ==
InteractionInvitationIn.Status.ACCEPTED) ||
(myInteractionInvitationParentIn.getCollaborationStatus() ==
InteractionInvitationIn.Status.ACCEPTANCE_PENDING)
{
   // The parent can take the REMIND or RECALL action on the
   // interaction.
   myInteractionInvitationParentIn.remind(myPlace);
} else {
   // Collaboration status is DECLINED, FAILED, or FULFILLED
   // REMIND or RECALL are not available
}
```

Monitoring Participant Activity

`InteractionInvitationParentIn` interactions are not used as incoming e-mails; they should be used to monitor the participant activity on the invitation.
Test the collaboration status periodically to take changes into account. To get the collaboration status of an invitation, call the `InteractionInvitationParentIn.getCollaborationStatus()` method. When the collaboration status is `InteractionInvitationIn.Status.FULFILLED`, you can get the response of the corresponding participant by calling the `getCollaborativeReply()` method.

```
if(myInteractionInvitationParentIn.getStatus() == InteractionInvitationIn.Status.FULFILLED)
```

```
{
   InteractionReplyOut myCollabReply =
myInteractionInvitationParentIn.getCollaborativeReply();
   System.out.println("The reply is: "+myCollabReply.getMessageText() );
}
```

> **Important**
>
> You cannot start a collaboration session on a collaborative reply. You cannot reply to a collaborative reply.

Participating in a Collaboration Session

In push mode, from the participant (or child) point of view, the agent receives an `InteractionEvent` for a `RINGING` incoming e-mail interaction of type `COLLABORATION_INVIT_IN`. See Putting Interactions in Workbins for details about pull mode. The following code snippet implements the `handleInterationEvent()` method for an `AgentListener`:

```
void handleInterationEvent( InteractionEvent event ) {
Interaction myInteraction = (Interaction) event.getSource();

//...
if(myInteraction.getType() == Interaction.Type.COLLABORATION_INVIT_IN)
{
   //Management of the collaborative invitation
}
   //...
}
```

Cast the interaction associated with the event to `InteractionInvitationIn`, as shown in the following code snippet:

```
InteractionInvitationIn myInvitation = (InteractionInvitationIn) event.getSource();
```

You can manage this incoming interaction as a common e-mail by answering to it. See also Responding to an E-Mail.

Once the interaction is in `TALKING` status, the agent can either accept the collaboration by calling the `acceptInvitation()` method or decline the invitation by calling the `declineInvitation()` method.

If the agent accepts, he or she can reply to the invitation, as shown in the following code snippet:

```
//Getting the interaction for the reply
InteractionReplyOut myReply = (InteractionReplyOut) myInvitation.reply("Place0") ;

//Setting the collaboration message
myReply.setMessageText("My reply is...");

//Sending the message
```

```
myReply.send();
```

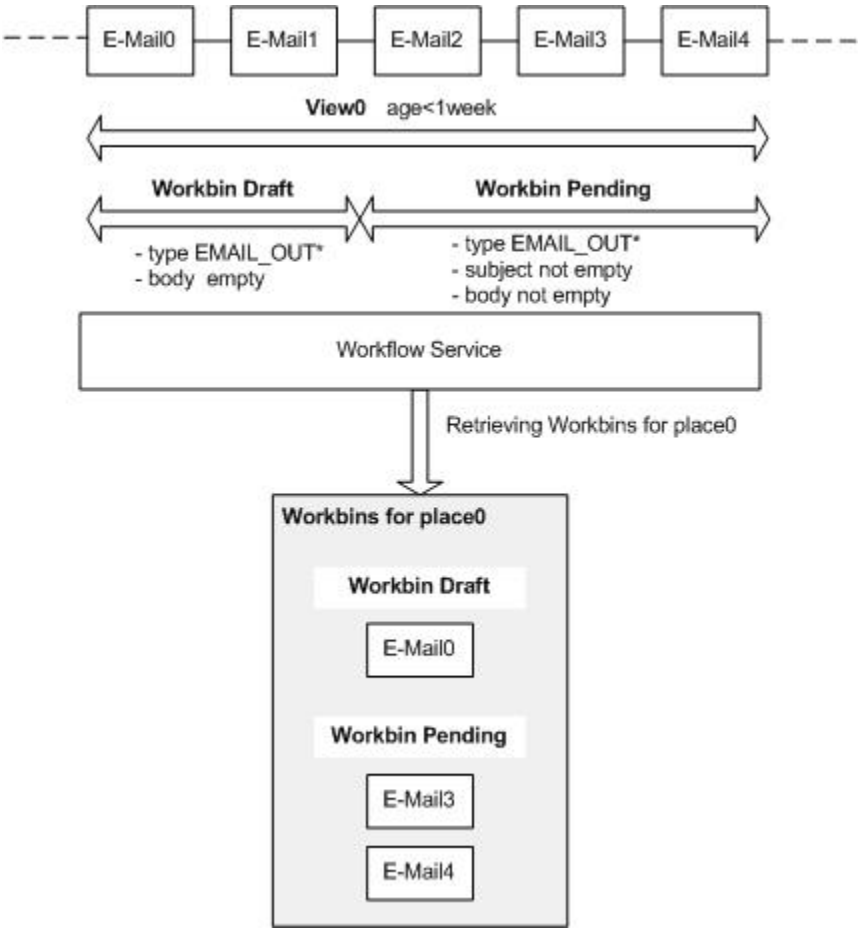
Once the reply is sent, the collaboration status of the corresponding invitation interaction becomes `FULFILLED` and the agent who initiates the collaboration can access the reply. Both the invitation and reply interactions are released.

Handling Workflow

Workflow management is provided with the `WorkbinManager` interface and classes of the`com.genesyslab.ail.workflow` package.

The `WorkbinManager` interface accesses the workbins of a place. From an agent's point of view, a workbin is a sort of interaction directory from which your application can pull, or into which it can put, interactions.

To define workbins more precisely: a queue contains interactions, and a view filters a queue's interactions according to a set of criteria. A workbin filters a view's interactions according to a further set of criteria. The following diagram shows an example of a view and some workbins defined for a queue.



Example for Workbins, Views, and Queues

The above figure shows a queue containing e-mail interactions. For this queue, `View0` lets your application see only e-mail interactions that are no older than a week.

In this view, two workbins coexist: one for draft e-mail interactions, and one for pending e-mail interactions. The `WorkbinManager` interface can use those filters to retrieve a set of interactions organized in workbins for a particular place.

Here, the `WorkbinManager` interface retrieves interactions no older than a week and available for `place0`. `E-Mail1` and `E-Mail2` are not retrieved, as they should not be processed in `place0`.

Use the Configuration Layer to define views and workbins. For further details, refer to your Configuration Layer documentation.

> **Important**
>
> Workbins can contain multimedia (non-voice) interactions only.

Workbins enable pull mode for multimedia interactions. Interaction status is `IDLE` in a workbin. You have to pull an interaction to change the interaction status and execute actions on the interaction. Use the `Workbin` and `WorkbinManager` interfaces to:

- Display workbins and their filtered interactions.
- Enable an agent to put an interaction in a workbin.

Getting the Workbin Manager

The workbin manager is available on the `Place` interface. Invoke `Place.getWorkbinManager()` on your agent's place to get the workbin manager.

```
WorkbinManager myWorkbinManager = place0.getWorkbinManager();
```

Use the `WorkbinManager` interface to get `Queue` and `Workbin` instances available for the agent. For example, the following code snippet gets the `Workbin` instance corresponding to Draft (see the diagram example for Workbins, Views, and Queues).

```
Workbin myDraftWorkbin = myWorkbinManager.getWorkbin("Draft");
```

Workbin Content

The `Workbin` interface is designed to manage the contents of a workbin as a set of `InteractionMultimediaSummary`. It provides the following methods:

- `getContent()`—retrieves interaction summaries for this particular workbin.
- `getContentForAll()`—retrieves interaction summaries for all agents or places defined for this workbin.

- `getSortedContentForAll()`—retrieves interaction summaries for all agents or places defined for this workbin, sorted by agent or place.

The `InteractionMultimediaSummary` interface is a summary description of an interaction available in a workbin. The corresponding interaction's status is `IDLE`. To execute actions on workbin interactions, you first have to pull those interactions. You cannot work with summaries or interactions retrieved from summaries.

The following code snippet displays the contents of the `draft` workbin for this place.

```
Collection myDrafts = myDraftWorkbin.getContent();
Iterator itDrafts = myDrafts.iterator();
while(itDrafts.hasNext())
{
   InteractionMultimediaSummary myDraft = (InteractionMultimediaSummary) itDrafts.next();
   System.out.println("Type: "+myDraft.getType().toString() + " Subject:"+myDraft.getSubject()
     + " Date:"+myDraft.getDateReceived().toString()+"\n");
}
```

The `Workbin` interface has methods to display information about the workbin itself—for instance, its name with `getName()`, the name of the associated queue with `getQueue()`, and its type with `getType()`.

Putting Interactions in Workbins

You can put an `InteractionMultimedia` into a workbin by calling the `Workbin.put()` method, as shown here:

```
/// Creating an outgoing e-mail interaction
InteractionMailOut mailOut = (InteractionMailOut) agent.createInteraction(MediaType.EMAIL,
null, Queue);
// Putting the interaction in the draft workbin
myDraftWorkbin.put(mailOut);
```

Two types of workbins coexist:

- Agent workbin—The `AgentWorkbin` interface allows your application to put interactions into the same workbin defined for other agents, or to get the contents of this workbin for another agent.
- Place workbin—The `PlaceWorkbin` interface allows your application to put interactions into same workbin defined for other places, or to get the contents of this workbin for another place.

You can cast the workbin according to its type, as shown here:

```
if( myDraftWorkbin.getType() == Workbin.Type.AGENT)
{
   AgentWorkbin myDraftAgentWorkbin = (AgentWorkbin) myWorkbinManager.getWorkbin("Draft");

   ///....
   // Creating an e-mail
   InteractionMailOut mailOut = (InteractionMailOut) agent.createInteraction(MediaType.EMAIL,
null, Queue);
   // Getting Agent2 interface
   Agent agent2 = (Agent) myAilFactory.getPerson("Agent2");
```

```
   // Putting the interaction in the draft workbin of agent2
   myDraftAgentWorkbin.put(mailOut, agent2);
}
```

During a collaboration session, the inviting agent can choose to use pull mode when sending his or her invitations. When setting the participant list of an `InteractionInvitationOut`, he or she can activate the pull mode. When the parent sends the outgoing invitation by calling `InteractionInvitationOut.send()`, participants of type `Participant.Type.AGENT` get their invitation in their workbin. They have to pull the invitation interaction to process it.

```
InteractionInvitationOut myInvitationsToSend =
myInteractionMailOut.createCollaborationInvitation();

// Setting type and name of the participant that is agent 0.
myParticipant0.setType(AGENT);
myParticipant0.setName("agent0");

// Adding agent0 to the list of participants
myInvitationsToSend.addParticipant(myParticipant0);

// Sending invitations: participant0 receives his or her invitation
// in his or her workbin
myInvitationsToSend.send();
```

Pulling Interactions

You can pull an interaction from its `InteractionMultimediaSummary` to a place by calling `pullInteractionMultimedia()`.

You can pull an interaction with the corresponding identifier by calling `Agent.openInteraction()` or `Place.openInteraction()`.

The `InteractionMultimedia` interaction is pulled on the agent or the place. Your application receives an `InteractionEvent` to update the `Interaction.Status` status which changes from `IDLE` in the workbin to `TALKING` once pulled.
Further processing of pulled interactions does not differ from the use cases described in the earlier sections.