Genesys Intelligent Automation Bots Integration Guide

Guidelines on Integrating IA with Microsoft Bot Framework

12/16/2025

Contents

Guidelines on Integrating IA with Microsoft Bot Framework

As the Microsoft Bot Framework is a toolkit for app developers, the following guidelines and assumptions are provided to ensure an optimal integration between Intelligent automation and the Microsoft Bot Framework.

An Activity is the primary means of information exchange between the Bot Framework and IA. The Activity schema is explained in detail. We will use the **value** field in the Activity schema to demonstrate how this field can be used in IA.

The following sections describe what IA expects from a bot in additional to some code samples (Node.js scripts)

Passing Context Variables

Context variables collected in a call-flow can be passed to the bot using the **Natural Language Settings** menu. The context parameters will be available in the *slots* sub-field in the *value* field of the activity.

```
Activity
{
    …
    ‘value’:
    {
        …,
        ‘slots’:
        {
            ‘var1’: 32
            ‘var2’: “Some string”
            ‘var3’: true
        }
    }
}
```

If you wish to leverage this information, it will be incumbent for the bot to extract out this information and make it persistent throughout the interaction You can now use extract this information and make it available to your bot to take action. The following code sample shows an additional step added when processing the first message of the conversation. It shows how incoming context variables can easily be read from the incoming message and stored in a persistent state.

```
/**
     * Pulls context variables out of the incoming message. These would have been
        * included in the message incoming from IA
     */
    async contextVarStep(stepContext) {
        let incomingActivity = stepContext.context['_activity'];
        let state = stateManager.fetchState(incomingActivity.conversation.id);
        if (incomingActivity.value && incomingActivity.value.slots) {
            state.contextVariables = incomingActivity.value.slots;
        }

        return await stepContext.next();
    }
```

The following sample shows how to persist the extracted information (the origin and the destination airports) so that additional follow-up questions are not required by the bot.

```
async actStep(stepContext) {
    …
    let state = stateManager.fetchState(stepContext.context["_activity"].conversation.id);

    let bookingDeails = state.contextVariables ? state.contextVariables : {};
    …
    // Call LUIS and gather any potential booking details. (Note the TurnContext has the
    //response to the prompt)
    const luisResult = await
                   this.luisRecognizer.executeLuisQuery(stepContext.context);
    switch (LuisRecognizer.topIntent(luisResult)) {
        case ‘BookFlight’:
            …
            // Extract the values for the composite entities from the LUIS result.
            const fromEntities = this.luisRecognizer.getFromEntities(luisResult);
            const toEntities = this.luisRecognizer.getToEntities(luisResult);
            …
            // Don't overwrite what we get from IA if we get nothing
            if (toEntities.airport) {
                bookingDetails.destination = toEntities.airport;
            }

            if (fromEntities.airport) {
                bookingDetails.origin = fromEntities.airport;
            }
            …
    }
    …
}
```

Intent Recognition

For intent recognition to work, the recognized intent has to be passed on to Intelligent Automation along with the conversation data.

The following example assumes that the intent received from LUIS is stored and persisted. It also includes the intent in a format that can be understood by IA.

```
async finalStep(stepContext) {
   …
   const currentIntent = stateManager.fetchSate(
        stepContext.context[“_activity”].conversation.id)
        .currentIntent

   …
   // This is an activity sent at the end of the conversation.
   // will be detailed below
   let endActivty = {
        type: “endOfConversation”,
        value: {
            ‘currentIntent’: currentIntent
        }
   }
}
```

As long as the intent is included in the *endOfConversation* activity, it will be recognized and handled by IA. This allows the call-flow to follow any paths set in the **Intent Lists** menu of the relevant sub-modules.

If the intent is not included, IA will not display an error for the unrecognized intent and the call flow will branch off the call-flow defined for errors. It is also possible to return the intent before the the *endOfConversation* activity.

End of Conversation

As shown in the previous example, each conversation should include an activity of type *endOfConversation* along with any final messages. Again, this *endOfConversation* message must include the intent parsed by LUIS.

The following example demonstrates how to send the *endOfConversation* activity along with a final message. It would be placed within the *finalStep* method.

```
return await stepContext.context.sendActivities([msgActivity, endActivity]);
```

Introduction Messages

Due to an inconsistent timing behaviour in Bot Framework, currently IA will make the first move in terms of sending a query to the bot. As a result, we ask that messages not be sent from Bot Framework to IA until the user’s first query.

To provide an introduction message, use the **Initial Question Prompt** available within the **Other Prompts** section of the **Prompt List** menu within the appropriate sub-module.