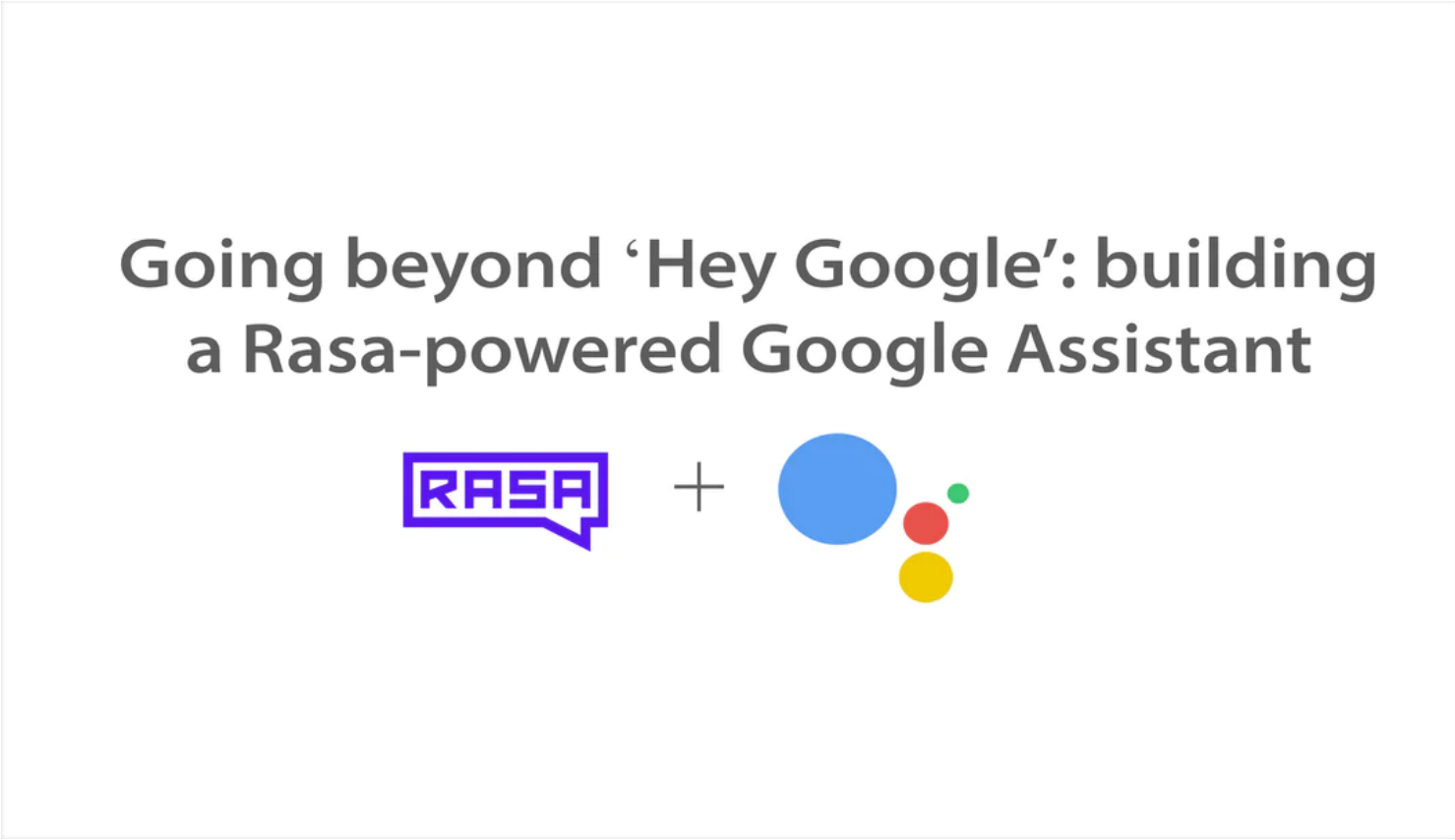
blog / Tutorials



September 20th, 2018

Going beyond ‘Hey Google’: building a Rasa-powered Google Assistant

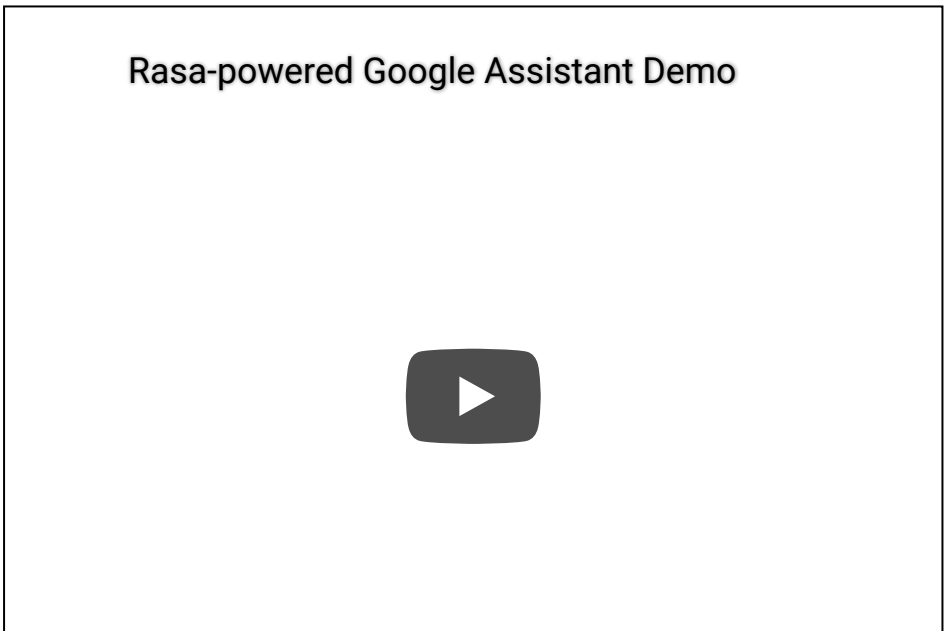
Justina Petraitytė

Since the launch in 2016, Google Assistant has rapidly became a familiar voice at people's homes. From playing your favourite song to controlling the lights - Google Assistant handles it all through simple voice interactions. But what about handling more complex conversations? If you used Google Assistant you have probably noticed that every request has to start with the magic phrase 'Hey Google' and that the majority of conversations with the assistant don't go beyond a single request.

In this post, we are going to show you how you can enable your Google Assistant to handle deeper and more natural conversations by integrating it with Open Source Rasa. There are a

- It's open source, which allows full customization of the framework and ensures that you have full ownership of your app's data.
- It uses machine learning to steer the dialogue - it learns the patterns from the real conversational data. It scales a lot better in production than the pre-defined rules or state machines.
- You can run your Rasa AI assistant across multiple channels which means that your assistant can be accessible across different messaging and voice platforms.

By combining Rasa Open Source with Google Assistant's powerful speech-to-text capabilities and a widely used interface, you can take your AI assistant to the whole new level. Rasa integration with voice interfaces like Google Assistant is a highly requested feature from our **Rasa Community**, so keep reading and learn how to do the integration yourself! Here is an example of what you can achieve by following this tutorial:



Outline

1. The Rasa AI assistant
2. Initializing the Google Assistant custom skill
3. Creating the custom Google Assistant action
4. Creating the Rasa Connector for Google Assistant
5. Putting all the pieces together

Here are the tools and software which you will need to create a Rasa-powered Google Assistant:

- Google Assistant smart speaker or a Google Assistant app - a tool to test the custom Google Assistant skill
- Google Actions console - the platform for initializing custom Google Assistant actions
- Google Actions SDK - a software pack designed to build custom Google Assistant skills
- Rasa - an open source conversational AI software stack
- Ngrok - a tunnel to expose a locally running application to the outside world

Step 1: The Rasa AI assistant

The main goal of this tutorial is to show you how you can connect a Rasa-powered agent to Google Assistant and make it more conversational. There are two options of how you can follow this tutorial - you can use your custom built Rasa assistant or you can grab our prebuilt places search assistant called Place Finder and use it to make the integration.

If you choose the first option, you can jump to step two of this tutorial. If you choose the second one, now is the best time to grab the Place Finder by cloning this repository.

Place Finder AI assistant

Place Finder is a Rasa-powered AI assistant capable of suggesting places like restaurants, shops, banks within the user specified radius and providing additional details about the returned place - the address, rating and opening hours. The assistant gets all these details by making a call to the Google Places API.

The assistant consists of the following files:

- **data/nlu_data.md** is a file which contains the training data for Rasa NLU model. This data consists of example user queries alongside the corresponding intents and entities.
- **config.yml** file contains the configuration of the NLU training pipeline as well as policy configuration to train the dialogue model.
- **data/stories.md** is a file which contains training data for the Rasa Core model. This data consists of stories - actual conversations between a user and an assistant written in a Rasa for-

- **domain.yml** is a file which contains the configuration of the assistant's domain. It consists of the following pieces:

5.1. **intents** and **entities** that are defined in Rasa NLU training data examples;

5.2. **slots** which work like placeholders for saving the details that the assistant has to remember throughout the conversation;

5.3. **templates** which define the text responses that an assistant should return when the corresponding utterances are predicted

5.4. **actions** which the assistant should be able to predict and execute.

- **actions.py** is a file which contains the code of the custom action where the assistant makes an api call to retrieve the details about the user's current location and search for the specified place within the user requested radius.The assistant then uses the returned details to generate the response and stores some of the retrieved details as slots for the assistant to use in the later stages of the conversation.
- **endpoints.yml** contains the configuration of custom actions webhook.
- **ga_credentials.yml** - a file to store your Google Places API key.
- **credentials.yml** - a file to configure your custom connector.

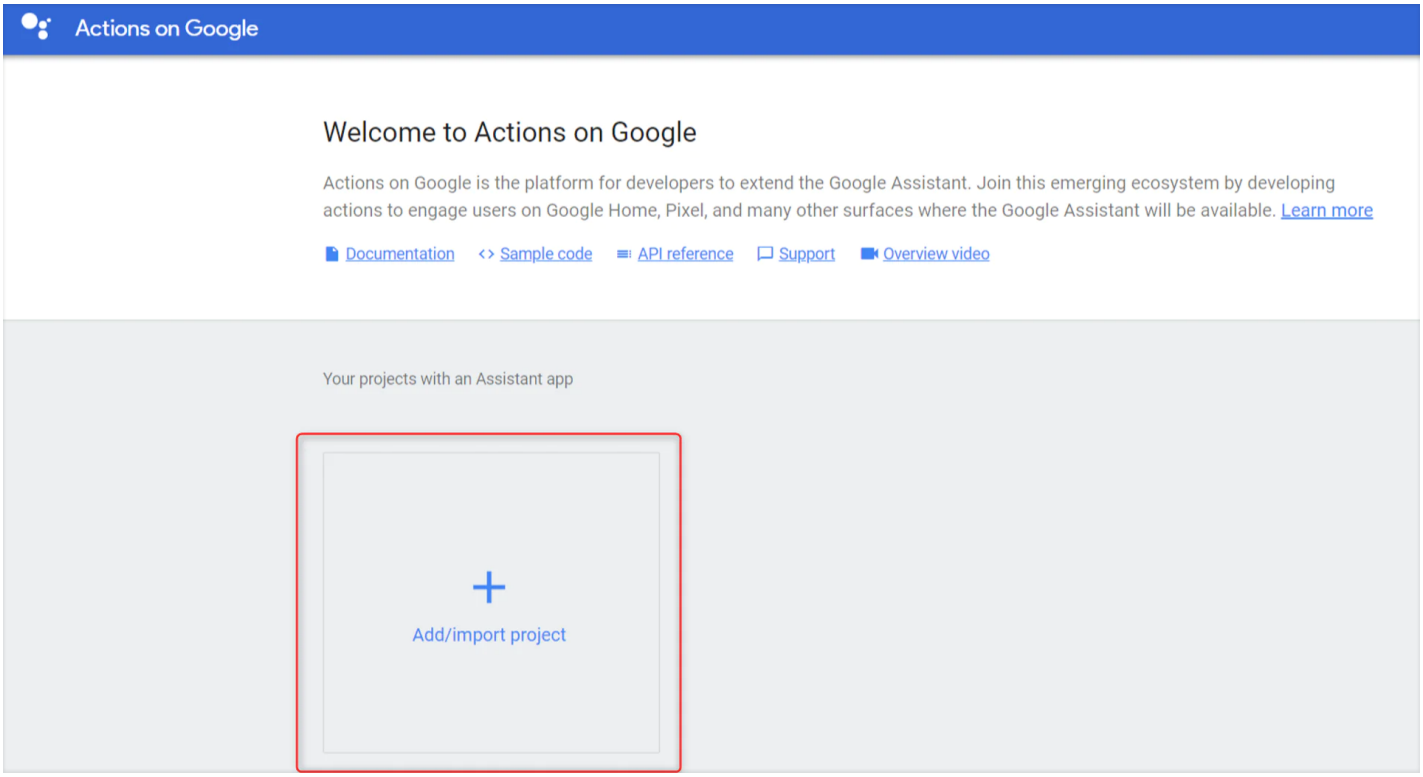
All these files are ready for you to use, all you have to do is provide your Google Places API key to ga_credentials.yml file and train the models. You can do that by running the command below which will train both NLU and dialogue models and save them as a compressed file in a models directory:

```
rasa train
```

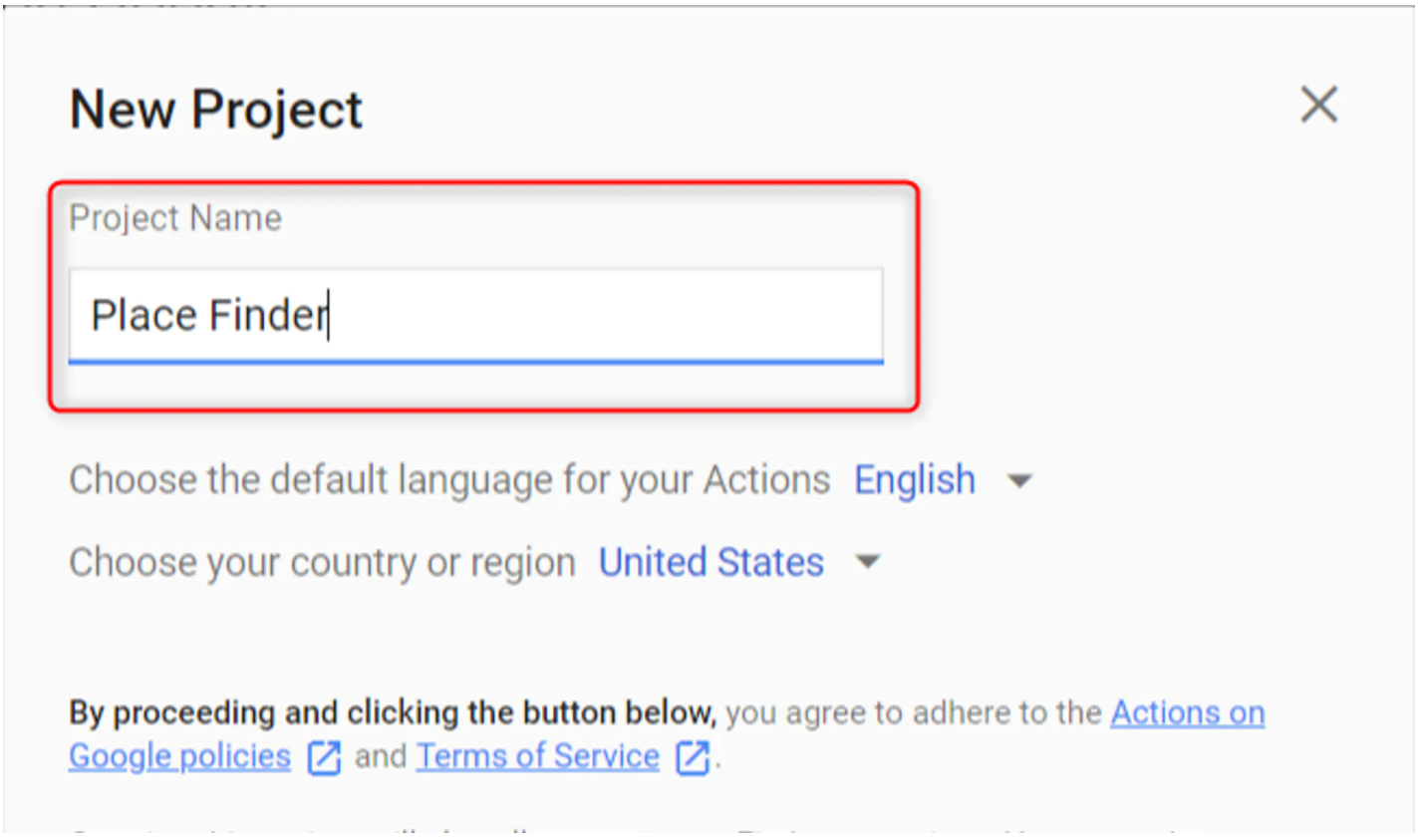
By training NLU and dialogue models you create the brain of your AI assistant. These two models will be responsible for understanding what the user says and deciding how an assistant should respond. Now it's time to create the ear and the mouth of the assistant using the Google Assistant.

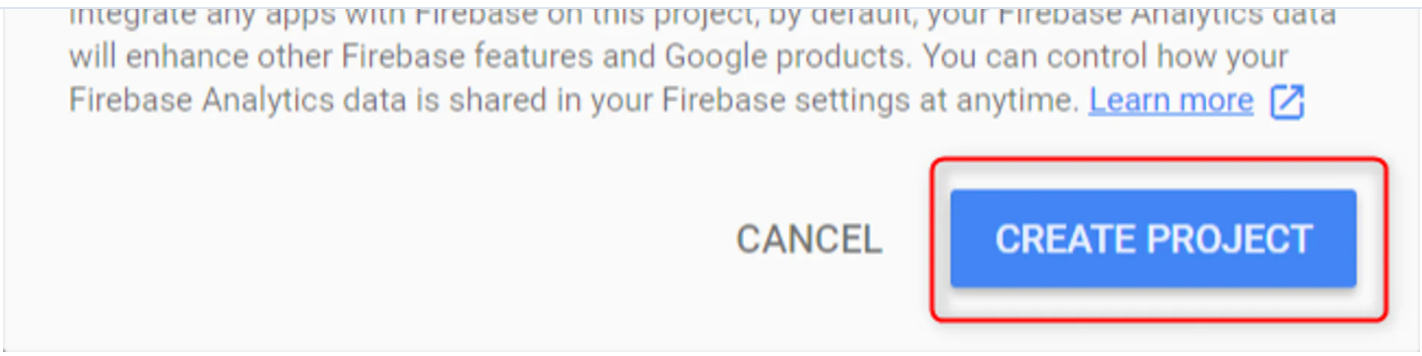
Step 2: Initializing the Google Assistant custom action

The custom Google Assistant action will package your Rasa-powered agent as a custom skill and will enable you to use Google Assistant's speech-to-text service to enable the voice communication with your assistant. First, you have to initialize the Google Assistant custom action. Here is how you can do it:

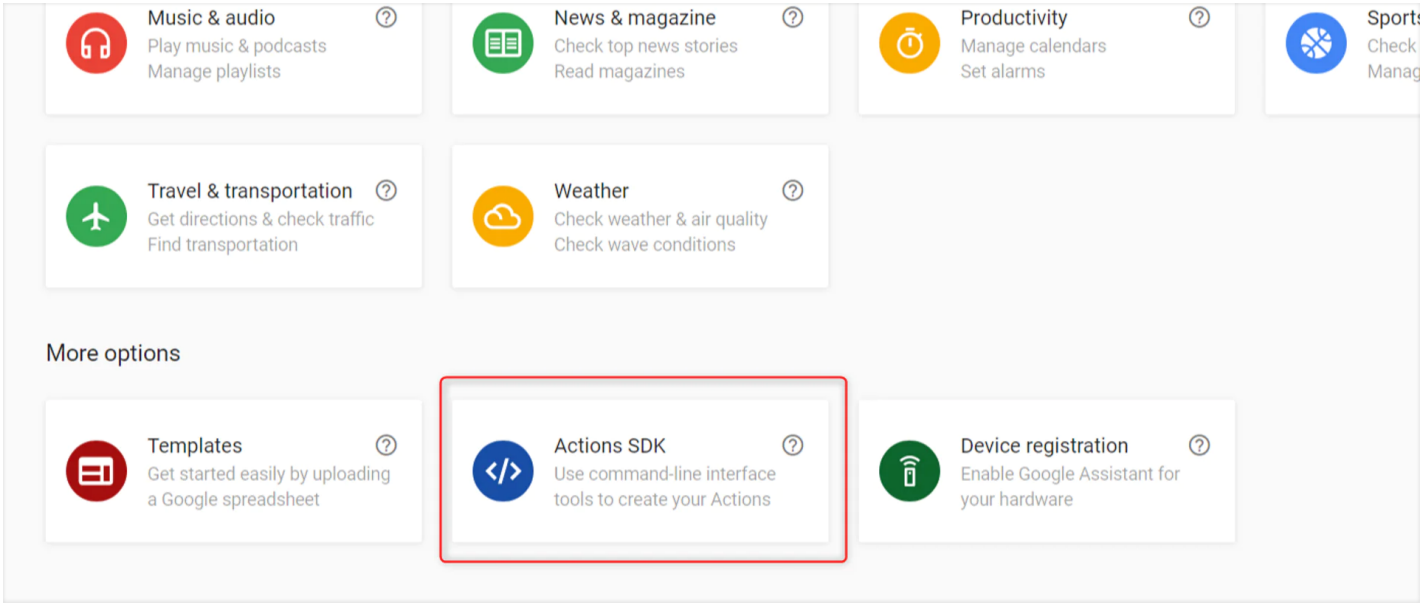


1. Set the name of your project and pick the language you want your assistant to use. Click 'Create project'.

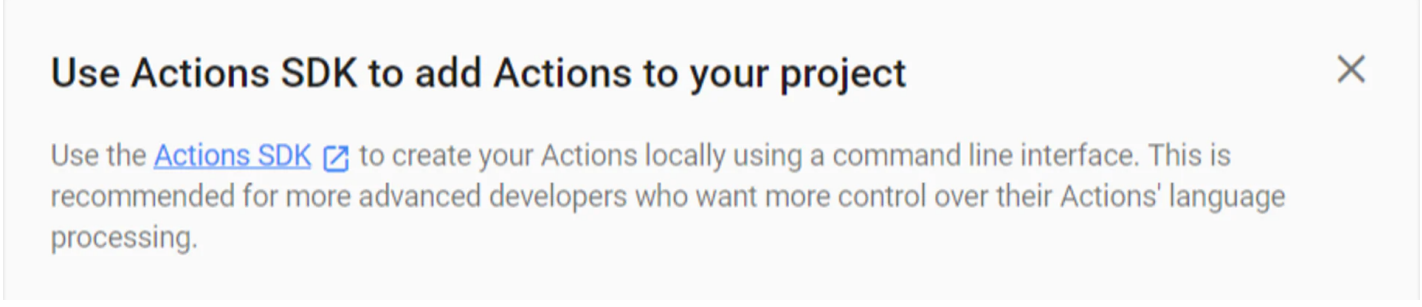


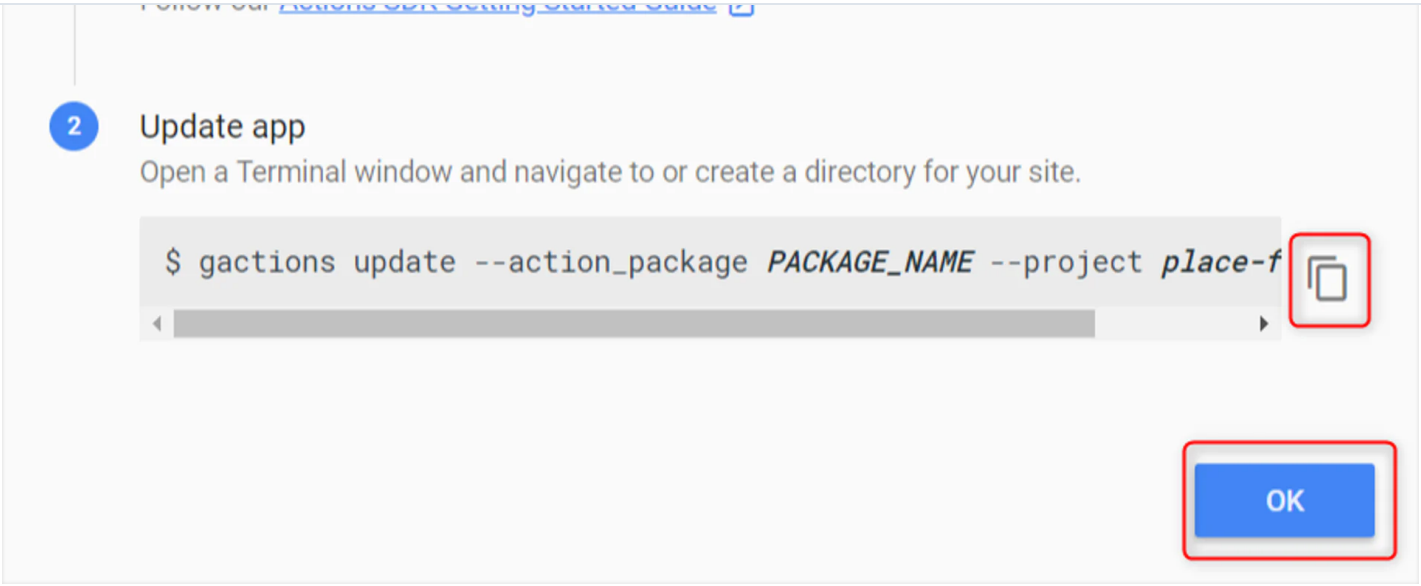


1. Once the project is created, you will be prompted to a page where you can choose the category of your project. Scroll down to the bottom of the page and choose 'Actions SDK' option. By doing that, you tell your Google Assistant that you will implement your own custom action with a custom NLP and dialogue management system.

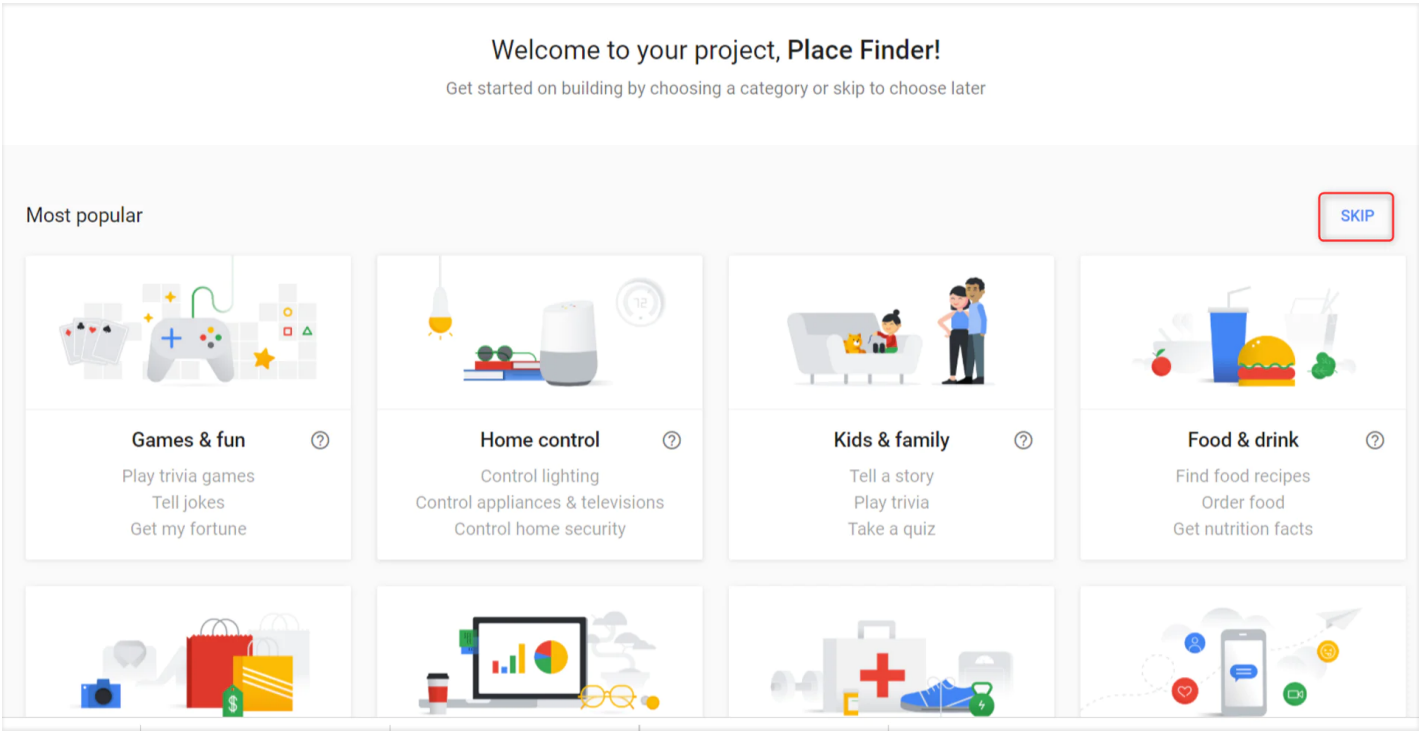


1. Inside the window which was opened, after selecting Actions SDK you will find Google Actions SDK guide and a gactions function which you will use later to deploy the custom action. Copy the provided function or take note of the --project parameter value and click OK.

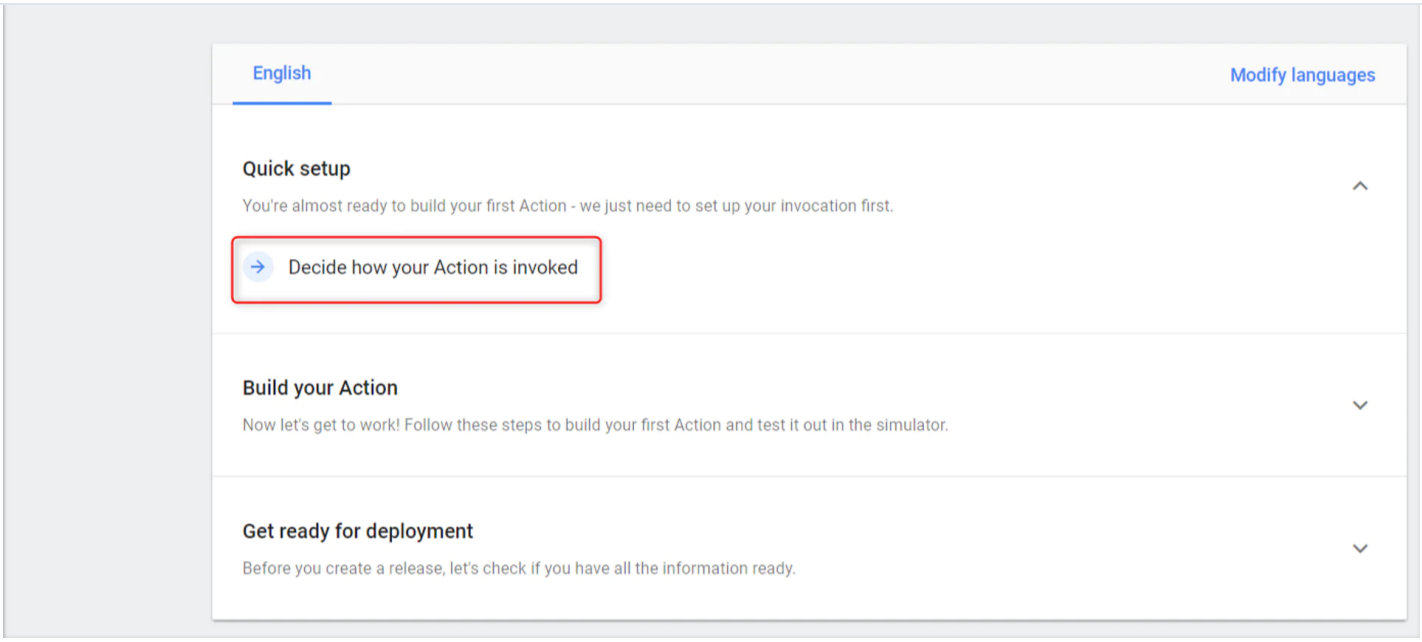




1. Now you can pick a project category or skip this step by selecting 'Skip' at the top of the page.



1. In the project configurations page you can specify how you want your custom skill to be invoked. You can do it by clicking on 'Decide how your Action is invoked' on a 'Quick setup' pane.



1. Here you can select the Display name - the name which Google Assistant looks for to start your custom skill. For example, if you set the display name to 'Place Finder', then to start this custom skill you will have to say something like 'Hey Google, talk to a Place Finder'. You can also choose the voice that you would like your Google Assistant to use and once you are happy with the configurations select 'Save'.

do when the user starts using the skill.

Step 3: Creating a custom Google Assistant action

Since the open source Rasa will take care of natural language understanding and dialogue management, custom Google Assistant action will be responsible for only two things - understanding that a user invoked the custom action and passing all user inputs to Rasa after the skill is invoked. To achieve this you will have to define intents for these two behaviours and fulfillments - things that the Google Assistant will have to do once these intents are matched after performing speech-to-text on user voice inputs. Here is how to do it:

1. Download Google gactions CLI and place it inside your project directory.
2. Inside that directory, initialize Google Assistant custom action configuration file by running the following command:
   `gactions init`

- Tip: If you are mac user you will likely need to convert gactions to executable file. You can do that by running `chmod +x gactions`. After that you can use the gactions as follows: `./gactions init`.*

1. Open the actions.json file which was initialized after running the command above. This file has the bare bones configuration of the custom skill invocation action. It consists of three main parts:

- **name** - it defines the type of the intent
- **intent** - corresponds to what the action is about
- **fulfillment** - a webhook which the Google Assistant should send the request to once the corresponding intent is matched

```json
{
  "actions": [
    {
      "description": "Default Welcome Intent",
      "name": "MAIN",
      "fulfillment": {
        "conversationName": "<INSERT YOUR CONVERSATION NAME HERE>"
      },
```

```
        "trigger": {
          "queryPatterns": [
            "talk to <INSERT YOUR NAME HERE>"
          ]
        }
      }
    }
  ],
  "conversations": {
    "<INSERT YOUR CONVERSATION NAME HERE>": {
      "name": "<INSERT YOUR CONVERSATION NAME HERE>",
      "url": "<INSERT YOUR FULLFILLMENT URL HERE>"
    }
  },
  "locale": "en"
}
```

All custom Google Assistant skills need a invocation action, so all you have to do is to provide the necessary details to the configuration above:

- *conversationName* which will work as a connector between the intent and a corresponding fulfillment. For the skill invocation action a conversationName could be something like 'welcome'
- queryPatterns - an example phrase that a user might say. For the skill invocation action a query pattern could be something like 'Talk to Place Finder'.

Once the invocation action is configured, all you have to add is one another action which will pass all user inputs to Rasa Stack after the custom skill is invoked. For this, you will have to define a new intent with the name 'TEXT' which will capture all user inputs after speech-to-text service is performed and pass it to Rasa Stack using the webhook which you will define in the next step. Below is the full configuration of the custom Google Assistant action:

```json
{
  "actions": [
    {
      "description": "Default Welcome Intent",
      "name": "MAIN",
```

```
        },
        "intent": {
          "name": "actions.intent.MAIN",
          "trigger": {
            "queryPatterns":["talk to Place Finder"]
          }
        }
      },
          {
        "description": "Rasa Intent",
        "name": "TEXT",
        "fulfillment": {
          "conversationName": "rasa_intent"
        },
        "intent": {
          "name": "actions.intent.TEXT",
          "trigger": {
            "queryPatterns":[]
          }
        }
      }],
    "conversations": {
      "welcome": {
        "name": "welcome",
        "url": "...",
        "fulfillmentApiVersion": 2
    },
      "rasa_intent": {
        "name": "rasa_intent",
        "url": "...",
        "fulfillmentApiVersion": 2
    }
  }
}
```

In order to pass user inputs to Rasa, Google Assistant needs a webhook url which will be used to establish the connection between the two. In the next step you will write a custom connector to connect Rasa to Google Assistant.

By default, Rasa comes with a bunch of **prebuilt connectors** to the most popular messaging platforms, but since Google Assistant connector is not yet included, you will learn how to create one yourself! Here are the steps how to do it:

1. In you project directory create a new file called 'ga_connector.py'. In this file you will write the code.
2. Open the ga_connector.py and start by importing necessary libraries which include:

- logging (for debugging)
- sanic (for creating the webhook)
- some Rasa Core classes (for creating the custom connector)
- json

```python
import logging
import json
from sanic import Blueprint, response
from sanic.request import Request
from typing import Text, Optional, List, Dict, Any

from rasa.core.channels.channel import UserMessage, OutputChannel
from rasa.core.channels.channel import InputChannel
from rasa.core.channels.channel import CollectingOutputChannel

logger = logging.getLogger(__name__)
```

3.Create a class called GoogleAssistant which inherits rasa core InputChannel class. This class will consist of two functions: name and blueprint:

```python
class GoogleAssistant(InputChannel):

  @classmethod
  def name(cls):

  def blueprint(self, on_new_message):
```

```python
@classmethod
def name(cls)
  return 'google_assistant'
```

5.Now let's fill in the blueprint function. This function will define the webhook that Google Assistant will use to pass the user inputs to Rasa Core, collect the responses and send them back to Google Assistant. Start by initialising the webhook name:

```python
def blueprint(self, on_new_message):
        google_webhook = Blueprint('google_webhook', __name__)
```

Next, you have to define the routes of your webhook. You should include at least two routes: health route for the endpoint '/' and a receive route for the endpoint '/webhook'. Below is the implementation of a health route - it will receive GET requests sent by Google Assistant and will return 200 OK message confirming that the connection works well.

```python
@google_webhook.route("/", methods=['GET'])
async def health(request):
            return response.json({"status": "ok"})
```

The receive route will receive POST requests from Google Assistant and pass the received payload to the Rasa agent. The code block below contains the implementation of this route, here is what it does:

- Once the POST request is received, the connector takes the payload of the request.
- The payload contains the details about the user's input sent by Google Assistant. These details include the user id, the intent (is it a launch request or a regular message) and the actual message in a text format. All these details are extracted from the payload and assigned to the corresponding variables.
- If the intent is a skill launch request, the Assistant will respond with introductory message "Welcome to Rasa-powered Google Assistant skill. You can start by saying hi."

and pass it to Rasa alongside the user input and sender id. Rasa Core will make prediction on how the assistant should respond and will allow you to retrieve the response message from the output channel.

- Finally, the produced message is incorporated into a json response which is sent back to Google Assistant.

```python
@google_webhook.route("/webhook", methods=['POST'])
async def receive(request):
    payload = request.json
    intent = payload['inputs'][0]['intent']
    text = payload['inputs'][0]['rawInputs'][0]['query']

    if intent == 'actions.intent.MAIN':
        message = "Hello! Welcome to the Rasa-powered Google Assistant skill.
    else:
        out = CollectingOutputChannel()
        await on_new_message(UserMessage(text, out))
        responses = [m["text"] for m in out.messages]
        message = responses[0]
    r = {
      "expectUserResponse": 'true',
      "expectedInputs": [
        {
          "possibleIntents": [
            {
              "intent": "actions.intent.TEXT"
             }
          ],
          "inputPrompt": {
            "richInitialPrompt": {
              "items": [
                {
                  "simpleResponse": {
                    "textToSpeech": message,
                    "displayText": message
                    }
                }
              ]
            }
```

```
            ]
        }

    return response.json(r)
```

Below is the full code of Google Assistant connector class:

Python

```python
import logging
import json
from sanic import Blueprint, response
from sanic.request import Request
from typing import Text, Optional, List, Dict, Any

from rasa.core.channels.channel import UserMessage, OutputChannel
from rasa.core.channels.channel import InputChannel
from rasa.core.channels.channel import CollectingOutputChannel

logger = logging.getLogger(__name__)

class GoogleConnector(InputChannel):
    """A custom http input channel.
    This implementation is the basis for a custom implementation of a chat
    frontend. You can customize this to send messages to Rasa Core and
    retrieve responses from the agent."""

    @classmethod
    def name(cls):
        return "google_assistant"

    def blueprint(self, on_new_message):

        google_webhook = Blueprint('google_webhook', __name__)

        @google_webhook.route("/", methods=['GET'])
        async def health(request):
            return response.json({"status": "ok"})
```

```
        payload = request.json
        intent = payload['inputs'][0]['intent']
        text = payload['inputs'][0]['rawInputs'][0]['query']

        if intent == 'actions.intent.MAIN':
            message = "Hello! Welcome to the Rasa-powered Google Assistant
        else:
            out = CollectingOutputChannel()
            await on_new_message(UserMessage(text, out))
            responses = [m["text"] for m in out.messages]
            message = responses[0]
        r = {
              "expectUserResponse": 'true',
              "expectedInputs": [
                {
                  "possibleIntents": [
                    {
                      "intent": "actions.intent.TEXT"
                    }
                  ],
                  "inputPrompt": {
                    "richInitialPrompt": {
                      "items": [
                        {
                          "simpleResponse": {
                            "textToSpeech": message,
                            "displayText": message
                            }
                          }
                        ]
                      }
                    }
                  }
              ]
            }

        return response.json(r)

    return google_webhook
```

Step 5: Putting all the pieces together

All that is left to do is to start the Rasa server which will use custom connector to connect to Google Assistant. You can do it by a command below which will load the Rasa assistant and will star the webhook which will listen for incoming user inputs:

```shell
rasa run --enable-api -p 5004
```

Tip: If you get an error that gaconnector.py module doesn't exist, add your project to the python path by running: `export PYTHONPATH=/pathtoplacefinderproject/:$PYTHONPATH`.

Since your Rasa assistant runs locally and Google Assistant runs on the cloud, you will need a tunneling service to expose your locally running Rasa assistant to the outside world. Ngrok is a good choice for that, so start ngrok on the same port as the Rasa agent is running on - 5004. Now you can connect it to Google Assistant by passing your webhook URL to fulfillments of your Google Assistant custom action. To do that, copy the url generated by ngrok (it will be different than the one we got for the example below), attach the '/webhooks/google_assistant/webhook' suffix to it, go back to action.json file and provide the url to the conversations part of the configuration file:

```json
{
   "actions": [
      {
        "description": "Default Welcome Intent",
        "name": "MAIN",
        "fulfillment": {
          "conversationName": "welcome"
        },
        "intent": {
          "name": "actions.intent.MAIN",
          "trigger": {
            "queryPatterns":["talk to Place Finder"]
          }
        }
```

```
          "description": "Rasa Intent",
          "name": "TEXT",
          "fulfillment": {
            "conversationName": "rasa_intent"
          },
          "intent": {
            "name": "actions.intent.TEXT",
            "trigger": {
              "queryPatterns":[]
            }
          }
        }],
      "conversations": {
        "welcome": {
          "name": "welcome",
          "url": "https://145f1bf4.ngrok.io/webhooks/google_assistant/webhook",
          "fulfillmentApiVersion": 2
      },
        "rasa_intent": {
          "name": "rasa_intent",
          "url": "https://145f1bf4.ngrok.io/webhooks/google_assistant/webhook",
          "fulfillmentApiVersion": 2
      }
    }
  }
```

If your Rasa assistant includes custom actions (just like the Place Finder does), make sure to start the actions server by running the following command:

```
rasa run actions
```

And one last thing - if you have DucklingHTTPExtractor in your NLP pipeline (just like the Place Finder does) make sure to start a local server for it as well. You can do it by running a command below:

```
docker run -p 8000:8000 rasa/duckling
```

Now, all that is left to do is to deploy your custom Google Assistant skill and enable testing. You can deploy the custom Google Assistant skill by running the command below. Here,

```
gactions update --action_package action.json --project PROJECT_ID
```

Tip: At this point you may be asked to allow gactions command to access your Google account. This is necessary to make sure that gactions command on your command line can access your custom action on Actions Console. To do the authorisation follow the instructions on your command line and make sure that you use the same Google account you used to set up your action.

Once the action is uploaded, enable testing of the action by running the following command:

```
gactions test --action_package action.json --project PROJECT_ID
```

Tip: if you forgot to take note of your project id, you can find it inside the projects settings on Google Actions console:

Congratulations! You have just built a custom Google Assistant skill with Rasa Stack!

Now you can go ahead and test it on your Google Home smart speaker or your Google Assistant app (make sure to be logged in with the same Google account you used to develop the custom skill on Google Actions console)! Alternatively, you can also test it on Google Actions console.

Let us know how you are getting on!

Resources

- Full code of this tutorial
- Rasa website
- Rasa Community Forum
- Rasa Stack docs
- Rasa GitHub repositories

RASA

Docs

Developer Portal
Rasa Open Source
Rasa Action Server
Rasa X

Education

Case Studies
Developer Portal
Conversation-driven Development
Rasa Blog
Rasa on YouTube
Rasa on Udemy

Community

Join the Community
How to Contribute
Community Showcase
Forum

Company

About Us
Careers We're hiring!
Our Mission
How we make money
Research
Company Blog
Contact Us

Social

Youtube
Twitter
Github
Stack Overflow
Linkedin
Angellist