dialogflow-fulfillment

Gabriel Farias Caccáos

Jan 19, 2023

OVERVIEW

1 Installation **1**
- 1.1 Version support . . . 1
- 1.2 Installing *dialogflow-fulfillment* . . . 1

2 Examples **3**
- 2.1 Dialogflow fulfillment webhook server with **Flask** . . . 3
- 2.2 Dialogflow fulfillment webhook server with **Django** . . . 4

3 Fulfillment overview **5**
- 3.1 What is fulfillment? . . . 5
- 3.2 A detailed example . . . 5

4 Webhook client **7**

5 Contexts **11**

6 Rich responses **13**
- 6.1 Card . . . 13
- 6.2 Image . . . 15
- 6.3 Payload . . . 16
- 6.4 Quick Replies . . . 17
- 6.5 Rich Response . . . 18
- 6.6 Text . . . 19

7 Contributing guide **21**
- 7.1 How can I contribute to *dialogflow-fulfillment*? . . . 21

8 Change log **23**
- 8.1 1.0.0 - 2022-04-17 . . . 23
- 8.2 0.4.4 - 2021-08-09 . . . 23
- 8.3 0.4.3 - 2021-06-29 . . . 24
- 8.4 0.4.2 - 2020-11-29 . . . 24
- 8.5 0.4.1 - 2020-10-11 . . . 24
- 8.6 0.4.0 - 2020-09-12 . . . 24
- 8.7 0.3.0 - 2020-07-29 . . . 25
- 8.8 0.2.0 - 2020-07-17 . . . 25
- 8.9 0.1.5 - 2020-07-17 . . . 26
- 8.10 0.1.4 - 2020-07-17 . . . 26
- 8.11 0.1.3 - 2020-07-17 . . . 26
- 8.12 0.1.2 - 2020-03-27 . . . 26

9 A simple example 27

10 Installation 29

11 Features 31

12 Limitations 33

Python Module Index 35

Index 37

CHAPTER

ONE

INSTALLATION

1.1 Version support

Note: *dialogflow-fulfillment* requires Python 3 or later.

1.2 Installing *dialogflow-fulfillment*

The preferred way to install *dialogflow-fulfillment* is from PyPI with pip, but it can be installed from source also.

1.2.1 From PyPI

To download *dialogflow-fulfillment* from PyPI with pip, simply run

```
$ pip install dialogflow-fulfillment
```

1.2.2 From source

In order to install *dialogflow-fulfillment* from the source code, you must clone the repository from GitHub:

```
$ git clone https://github.com/gcaccaos/dialogflow-fulfillment.git
```

Then, install *dialogflow-fulfillment* in editable (`-e`) mode with pip:

```
$ cd dialogflow-fulfillment
$ pip install -e .
```

CHAPTER

TWO

EXAMPLES

2.1 Dialogflow fulfillment webhook server with Flask

Listing 1: app.py

```
from logging import INFO
from typing import Dict

from dialogflow_fulfillment import WebhookClient
from flask import Flask, request
from flask.logging import create_logger

# Create Flask app and enable info level logging
app = Flask(__name__)
logger = create_logger(app)
logger.setLevel(INFO)


def handler(agent: WebhookClient) -> None:
    """Handle the webhook request."""


@app.route('/', methods=['POST'])
def webhook() -> Dict:
    """Handle webhook requests from Dialogflow."""
    # Get WebhookRequest object
    request_ = request.get_json(force=True)

    # Log request headers and body
    logger.info(f'Request headers: {dict(request.headers)}')
    logger.info(f'Request body: {request_}')

    # Handle request
    agent = WebhookClient(request_)
    agent.handle_request(handler)

    # Log WebhookResponse object
    logger.info(f'Response body: {agent.response}')

    return agent.response
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    app.run(debug=True)
```

2.2 Dialogflow fulfillment webhook server with Django

Listing 2: views.py

```
from json import loads
from logging import getLogger

from dialogflow_fulfillment import WebhookClient
from django.http import HttpRequest, HttpResponse, JsonResponse
from django.views.decorators.csrf import csrf_exempt

logger = getLogger('django.server.webhook')


def handler(agent: WebhookClient) -> None:
    """Handle the webhook request."""


@csrf_exempt
def webhook(request: HttpRequest) -> HttpResponse:
    """Handle webhook requests from Dialogflow."""
    if request.method == 'POST':
        # Get WebhookRequest object
        request_ = loads(request.body)

        # Log request headers and body
        logger.info(f'Request headers: {dict(request.headers)}')
        logger.info(f'Request body: {request_}')

        # Handle request
        agent = WebhookClient(request_)
        agent.handle_request(handler)

        # Log WebhookResponse object
        logger.info(f'Response body: {agent.response}')

        return JsonResponse(agent.response)

    return HttpResponse()
```

CHAPTER

THREE

FULFILLMENT OVERVIEW

3.1 What is fulfillment?

Dialogflow's console allows to create simple and static responses for user's intents in conversations. In order to create more dynamic and complex responses, such as retrieving information from other services, the intent's **fulfillment setting** must be enabled and a webhook service must be provided:

> When an intent with fulfillment enabled is matched, Dialogflow sends a request to your webhook service with information about the matched intent. Your system can perform any required actions and respond to Dialogflow with information for how to proceed.
>
> —Source: Fulfillment.

3.2 A detailed example

Fig. 1: A representation of how data flows in a conversation between a user and a Dialogflow agent.

The above diagram is a simplified representation of how data flows in a conversation between a user and a Dialogflow agent through an user interface. In this example, the user's intent is fulfilled by the agent with the help of a webhook service, allowing to handle more dynamic responses, like calling an external API to fetch some information.

The flow of data in a conversation with fulfillment enabled can be described as follows:

1. The user types a text into the application's front-end in order to send a query to the agent.
2. The input is captured by the application's back-end, which calls Dialogflow API's *detectIntent* ` resource, either via the official client or via HTTPS request in the form of a JSON. The request's body contain a `QueryInput` object, which holds the user's query (along with other information).
3. Dialogflow detects the intent that corresponds to the user's query and, since the intent in this example has the fulfillment setting enabled, posts a `WebhookRequest` object to the external webhook service via HTTPS in the form of a JSON. This object has a `QueryResult` object, which also holds the user's query and information about the detected intent, such as the corresponding action, detected entities and input or output contexts.
4. The webhook service uses information from the `QueryResult` object (along with other data from the `WebhookRequest` object) in order to determine how the conversation must go. For example, it could trigger some event by setting an `EventInput`, change the value of a parameter in a `Context` or generate `Message` objects using data from external services, such as APIs or databases.
5. In this example, the webhook service calls an external API in order to fulfill the user's query.
6. Then, a `WebhookResponse` object with the generated response data is returned to Dialogflow.

7. Dialogflow validates the response, checking for present keys and value types, and returns a `DetectIntentResponse` object to the interface application.
8. Finally, the application's front-end displays the resulting response message(s) to the user.

CHAPTER

FOUR

WEBHOOK CLIENT

class WebhookClient(*request*)

Bases: object

A client class for handling webhook requests from Dialogflow.

This class allows to dinamically manipulate contexts and create responses to be sent back to Dialogflow (which will validate the response and send it back to the end-user).

Parameters **request** (*dict*) – The webhook request object (WebhookRequest) from Dialogflow.

Raises TypeError – If the request is not a dictionary.

Return type None

See also:

For more information about the webhook request object, see the WebhookRequest section in Dialogflow's API reference.

query

The original query sent by the end-user.

Type str

intent

The intent triggered by Dialogflow.

Type str

action

The action defined for the intent.

Type str

context

An API class for handling input and output contexts.

Type *Context*

contexts

The array of input contexts.

Type list(dict)

parameters

The intent parameters extracted by Dialogflow.

Type dict

console_messages

The response messages defined for the intent.

Type list(*RichResponse*)

original_request

The original request object from *detectIntent/query*.

Type str

request_source

The source of the request.

Type str

locale

The language code or locale of the original request.

Type str

session

The session id of the conversation.

Type str

add(*responses*)

Add response messages to be sent back to Dialogflow.

Examples

Adding a simple text response as a string:

```
>>> agent.add('Hi! How can I help you?')
```

Adding multiple rich responses one at a time:

```
>>> agent.add(Text('How are you feeling today?'))
>>> agent.add(QuickReplies(quick_replies=['Happy :)', 'Sad :(']))
```

Adding multiple rich responses at once:

```
>>> responses = [
...     Text('How are you feeling today?'),
...     QuickReplies(quick_replies=['Happy :)', 'Sad :('])
... ]
>>> agent.add(responses)
```

Parameters **responses** (*str*, RichResponse, *list*(*str*, RichResponse)) – A single response message or a list of response messages.

Return type None

property followup_event: Optional[Dict[str, Any]]

The followup event to be triggered by the response.

Examples

Accessing the *followup_event* attribute:

```
>>> agent.followup_event
None
```

Assigning an event name to the *followup_event* attribute:

```
>>> agent.followup_event = 'WELCOME'
>>> agent.followup_event
{'name': 'WELCOME', 'languageCode': 'en-US'}
```

Assigning an event dictionary to the *followup_event* attribute:

```
>>> agent.followup_event = {'name': 'GOODBYE', 'languageCode': 'en-US'}
>>> agent.followup_event
{'name': 'GOODBYE', 'languageCode': 'en-US'}
```

Raises TypeError – If the event is not a string or a dictionary.

Type dict, optional

handle_request(*handler*)

Handle the webhook request using a handler or a mapping of handlers.

In order to manipulate the conversation programatically, the handler function must receive an instance of *WebhookClient* as a parameter. Then, inside the function, *WebhookClient*'s attributes and methods can be used to access and manipulate the webhook request attributes and generate the webhook response.

Alternatively, this method can receive a mapping of handler functions for each intent.

Note: If a mapping of handler functions is provided, the name of the corresponding intent must be written exactly as it is in Dialogflow.

Finally, once the request has been handled, the generated webhook response can be accessed via the *response* attribute.

Examples

Creating a simple handler function that sends a text and a collection of quick reply buttons to the end-user (the response is independent of the triggered intent):

```
>>> def handler(agent: WebhookClient) -> None:
...     agent.add('How are you feeling today?')
...     agent.add(QuickReplies(quick_replies=['Happy :)', 'Sad :(']))
```

Creating a mapping of handler functions for different intents:

```
>>> def welcome_handler(agent):
...     agent.add('Hi!')
...     agent.add('How can I help you?')
...
```

(continues on next page)

(continued from previous page)

```
>>> def fallback_handler(agent):
...     agent.add('Sorry, I missed what you said.')
...     agent.add('Can you say that again?')
...
>>> handler = {
...     'Default Welcome Intent': welcome_handler,
...     'Default Fallback Intent': fallback_handler,
... }
```

Parameters **handler** (*callable*, *dict(str, callable)*) – The handler function or a mapping of intents to handler functions.

Raises TypeError – If the handler is not a function or a map of functions.

Returns The output from the handler function (if any).

Return type any, optional

property response: Dict[str, Any]

The generated webhook response object (WebhookResponse).

See also:

For more information about the webhook response object, see the WebhookResponse section in Dialogflow's API reference.

Type dict

CHAPTER

FIVE

CONTEXTS

class Context(*input_contexts*, *session*)

Bases: object

A client class for accessing and manipulating input and output contexts.

This class provides an API that allows to create, edit or delete contexts during conversations.

Parameters

- **input_contexts** (*list(dict)*) – The contexts that were active in the conversation when the intent was triggered by Dialogflow.
- **session** (*str*) – The session of the conversation.

Return type None

input_contexts

The contexts that were active in the conversation when the intent was triggered by Dialogflow.

Type list(dict)

session

The session of the conversation.

Type str

contexts

A mapping of context names to context objects (dictionaries).

Type dict(str, dict)

delete(*name*)

Deactivate an output context by setting its lifespan to 0.

Parameters **name** (*str*) – The name of the context.

Return type None

get(*name*)

Get the context object (if exists).

Parameters **name** (*str*) – The name of the context.

Returns The context object (dictionary) if exists.

Return type dict, optional

`get_output_contexts_array()`

Get the output contexts as an array.

Returns The output contexts (dictionaries).

Return type list(dict)

`set(name, lifespan_count=None, parameters=None)`

Set a new context or update an existing context.

Sets the lifepan and parameters of a context (if the context exists) or creates a new output context (if the context doesn't exist).

Parameters

- **name** (*str*) – The name of the context.
- **lifespan_count** (*int, optional*) – The lifespan duration of the context (in minutes).
- **parameters** (*dict, optional*) – The parameters of the context.

Raises **TypeError** – If the name is not a string.

Return type None

CHAPTER

SIX

RICH RESPONSES

6.1 Card

class Card(*title=None*, *subtitle=None*, *image_url=None*, *buttons=None*)

Bases: *dialogflow_fulfillment.rich_responses.base.RichResponse*

Send a card response to the end-user.

Examples

Constructing a *Card* response:

```
>>> card = Card(
...     title='What is your favorite color?',
...     subtitle='Choose a color',
...     buttons=[{'text': 'Red'}, {'text': 'Green'}, {'text': 'Blue'}]
... )
```

Parameters

- **title** (*str, optional*) – The title of the card response.
- **subtitle** (*str, optional*) – The subtitle of the card response. Defaults
- **image_url** (*str, optional*) – The URL of the card response's image.
- **buttons** (*list(dict(str, str)), optional*) – The buttons of the card response.

Return type None

See also:

For more information about the *Card* response, see the Card responses section in Dialogflow's documentation.

property buttons: Optional[List[Dict[str, str]]]

The buttons of the card response.

Examples

Accessing the *buttons* attribute:

```
>>> card.buttons
[{'text': 'Red'}, {'text': 'Green'}, {'text': 'Blue'}]
```

Assigning value to the *buttons* attribute:

```
>>> card.buttons = [{'text': 'Cyan'}, {'text': 'Magenta'}]
>>> card.buttons
[{'text': 'Cyan'}, {'text': 'Magenta'}]
```

Raises TypeError – If the value to be assigned is not a list of buttons.

Type list(dict(str, str)), optional

property image_url: Optional[str]

The URL of the card response's image.

Examples

Accessing the *image_url* attribute:

```
>>> card.image_url
None
```

Assigning value to the *image_url* attribute:

```
>>> card.image_url = 'https://picsum.photos/200/300.jpg'
>>> card.image_url
'https://picsum.photos/200/300.jpg'
```

Raises TypeError – If the value to be assigned is not a string.

Type str, optional

property subtitle: Optional[str]

The subtitle of the card response.

Examples

Accessing the *subtitle* attribute:

```
>>> card.subtitle
'Choose a color'
```

Assigning value to the *subtitle* attribute:

```
>>> card.subtitle = 'Select a color below'
>>> card.subtitle
'Select a color below'
```

Raises TypeError – If the value to be assigned is not a string.

Type str, optional

property title: Optional[str]

The title of the card response.

Examples

Accessing the *title* attribute:

```
>>> card.title
'What is your favorite color?'
```

Assigning value to the *title* attribute:

```
>>> card.title = 'Which color do you like?'
>>> card.title
'Which color do you like?'
```

Raises TypeError – If the value to be assigned is not a string.

Type str, optional

6.2 Image

class Image(*image_url=None*)

Bases: *dialogflow_fulfillment.rich_responses.base.RichResponse*

Send an image response to the end-user.

Examples

Constructing an image response:

```
>>> image = Image('https://picsum.photos/200/300.jpg')
```

Parameters **image_url** (*str, optional*) – The URL of the image response.

Return type None

See also:

For more information about the *Image* response, see the Image responses section in Dialogflow's documentation.

property image_url: Optional[str]

The URL of the image response.

Examples

Accessing the *image_url* attribute:

```
>>> image.image_url
'https://picsum.photos/200/300.jpg'
```

Assigning a value to the *image_url* attribute:

```
>>> image.image_url = 'https://picsum.photos/200/300?blur.jpg'
>>> image.image_url
'https://picsum.photos/200/300?blur.jpg'
```

Raises TypeError – If the value to be assigned is not a string.

Type str, optional

6.3 Payload

class Payload(*payload=None*)

Bases: *dialogflow_fulfillment.rich_responses.base.RichResponse*

Send a custom payload response to the end-user.

This type of rich response allows to create advanced, custom, responses.

Examples

Constructing a custom *Payload* response for file attachments:

```
>>> payload_data = {
...     'attachment': 'https://example.com/files/some_file.pdf',
...     'type': 'application/pdf'
... }
>>> payload = Payload(payload_data)
```

Parameters **payload** (*dict, optional*) – The content of the custom payload response.

Return type None

See also:

For more information about the *Payload* response, see the Custom payload responses section in Dialogflow's documentation.

property payload: Optional[Dict[Any, Any]]

The content of the custom payload response.

Examples

Accessing the *payload* attribute:

```
>>> payload.payload
{'attachment': 'https://example.com/files/some_file.pdf', 'type': 'application/
↪pdf'}
```

Assigning a value to the *payload* attribute:

```
>>> payload.payload = {
...     'attachment': 'https://example.com/files/another_file.zip',
...     'type': 'application/zip'
... }
>>> payload.payload
{'attachment': 'https://example.com/files/another_file.zip', 'type':
↪'application/zip'}
```

Raises TypeError – If the value to be assigned is not a dictionary.

Type dict, optional

6.4 Quick Replies

class QuickReplies(*title=None*, *quick_replies=None*)

Bases: *dialogflow_fulfillment.rich_responses.base.RichResponse*

Send a collection of quick replies to the end-user.

When a quick reply button is clicked, the corresponding reply text is sent back to Dialogflow as if the user had typed it.

Examples

Constructing a *QuickReplies* response:

```
>>> quick_replies = QuickReplies('Choose an answer', ['Yes', 'No'])
```

Parameters

- **title** (*str, optional*) – The title of the quick reply buttons.
- **quick_replies** (*list, tuple(str), optional*) – The texts for the quick reply buttons.

Return type None

See also:

For more information about the *QuickReplies* response, see the Quick reply responses section in Dialogflow's documentation.

property quick_replies: Optional[Union[List[str], Tuple[str]]]

The texts for the quick reply buttons.

Examples

Accessing the *quick_replies* attribute:

```
>>> quick_replies.quick_replies
['Yes', 'No']
```

Assigning a value to the *quick_replies* attribute:

```
>>> quick_replies.quick_replies = ['Yes', 'No', 'Maybe']
>>> quick_replies.quick_replies
['Yes', 'No', 'Maybe']
```

Raises TypeError – if the value to be assigned is not a list or tuple of strings.

Type list, tuple(str), optional

property title: Optional[str]

The title of the quick reply buttons.

Examples

Accessing the *title* attribute:

```
>>> quick_replies.title
'Choose an answer'
```

Assigning a value to the *title* attribute:

```
>>> quick_replies.title = 'Select yes or no'
>>> quick_replies.title
'Select yes or no'
```

Raises TypeError – If the value to be assigned is not a string.

Type str, optional

6.5 Rich Response

class RichResponse

Bases: object

The base (abstract) class for the different types of rich responses.

See also:

For more information about the *RichResponse*, see the Rich response messages section in Dialogflow's documentation.

6.6 Text

class Text(*text=None*)

Bases: *dialogflow_fulfillment.rich_responses.base.RichResponse*

Send a basic (static) text response to the end-user.

Examples

Constructing a *Text* response:

```
>>> text = Text('this is a text response')
```

Parameters **text** (*str, optional*) – The content of the text response.

Return type None

See also:

For more information about the *Text* response, see the Text responses section in Dialogflow's documentation.

property text: Optional[str]

The content of the text response.

Examples

Accessing the *text* attribute:

```
>>> text.text
'this is a text response'
```

Assigning a value to the *text* attribute:

```
>>> text.text = 'this is a new text response'
>>> text.text
'this is a new text response'
```

Raises **TypeError** – If the value to be assigned is not a string.

Type str, optional

CHAPTER

SEVEN

CONTRIBUTING GUIDE

7.1 How can I contribute to *dialogflow-fulfillment*?

If you want to **request an enhancement**, **report a bug**, or simply **have a question** that has not been answered by the documentation, you are always welcome to create an issue on GitHub.

CHAPTER

EIGHT

CHANGE LOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

8.1 1.0.0 - 2022-04-17

8.1.1 Removed

- RichResponse's set_* methods (use property attributes instead).
- WebhookClient's set_followup_event method (use property attribute instead).

8.1.2 Dependencies

- Change sphinx's template to *furo*.
- Upgrade many development dependencies.

8.2 0.4.4 - 2021-08-09

8.2.1 Fixed

- Bug when the webhook request is an empty JSON.

8.2.2 Dependencies

- Bump dependencies in requirements and setup files

8.3 0.4.3 - 2021-06-29

8.3.1 Added

- Code of Conduct file.

8.3.2 Dependencies

- Django example dependencies versions.

8.4 0.4.2 - 2020-11-29

8.4.1 Fixed

- Bug when parsing *WebhookRequest* object in Django example (#1).
- Bug when calling *response* in *WebhookClient* multiple times (#2).

8.5 0.4.1 - 2020-10-11

8.5.1 Added

- Continuous integration and continuous deployment with Github Actions.

8.5.2 Improved

- Health of the source code.
- Documentation.

8.6 0.4.0 - 2020-09-12

8.6.1 Added

- Getters and setters for RichResponse's attributes (and deprecation warnings to set_*() methods).
- Getter and setter for WebhookClient's followup_event attribute (and deprecation warning to set_followup_event() method).
- Docs: Examples to WebhookClient's methods docstrings.
- Docs: Examples to RichResponse's attributes docstrings.
- Docs: "See also" sections in RichResponse's docstrings.
- Docs: Type hints to WebhookClient's handle_request() method's docstring.
- Docs: "Detailed example" section in "Fulfillment overview" page.

8.6.2 Improved

- Typing annotations coverage.

8.7 0.3.0 - 2020-07-29

8.7.1 Added

- Docs: Change log and contributing guide pages.
- set_text() method for the Text response.
- set_subtitle(), set_image() and set_buttons() methods for the Card response.
- set_title() and set_quick_replies() to the QuickReplies response.

8.7.2 Fixed

- Fix missing fields in Card and QuickReply responses.
- Fix optional parameters for all rich responses.
- Fix parsing of Image and Card responses from requests.
- Fix RichResponse instantiation (shouldn't be able to instantiate an abstract base class).

8.7.3 Improved

- Docs: improve classes and methods docstrings.

8.7.4 Changed

- Docs: Change theme to Read the Docs' theme.

8.8 0.2.0 - 2020-07-17

8.8.1 Added

- Tests for Context and WebhookClient.

8.8.2 Changed

- Rewrite tests using pytest.

8.9 0.1.5 - 2020-07-17

8.9.1 Fixed

- Fix a key access error in WebhookClient's request processing.

8.10 0.1.4 - 2020-07-17

8.10.1 Added

- Type hints for WebhookClient methods.
- Type hints for Context methods.
- Type hints for RichResponse methods.

8.11 0.1.3 - 2020-07-17

8.11.1 Added

- Public API of the package.

8.12 0.1.2 - 2020-03-27

- Initial release.

dialogflow-fulfillment is a package for Python that helps developers to create webhook services for Dialogflow.

The package provides an API for creating and manipulating response messages, output contexts and follow-up events in conversations.

See also:

For more information about fulfillment and how it works, see *Fulfillment overview*.

CHAPTER

NINE

A SIMPLE EXAMPLE

Working with *dialogflow-fulfillment* is as simple as passing a webhook request object from Dialogflow (a.k.a. `WebhookRequest`) to an instance of a *`WebhookClient`* and using a handler function (or a mapping of functions for each intent) via the *`handle_request()`* method:

Listing 1: simple_example.py

```
from dialogflow_fulfillment import QuickReplies, WebhookClient


# Define a custom handler function
def handler(agent: WebhookClient) -> None:
    """
    Handle the webhook request.

    This handler sends a text message along with a quick replies
    message back to Dialogflow, which uses the messages to build
    the final response to the user.
    """
    agent.add('How are you feeling today?')
    agent.add(QuickReplies(quick_replies=['Happy :)', 'Sad :(']))


# Create an instance of the WebhookClient
agent = WebhookClient(request)  # noqa: F821

# Handle the request using the handler function
agent.handle_request(handler)
```

The above code produces the resulting response object (a.k.a. `WebhookResponse`), which can be accessed via the *`response`* attribute:

```
{
   'fulfillmentMessages': [
      {
         'text': {
            'text': [
               'How are you feeling today?'
            ]
         }
      },
      {
```

(continues on next page)

(continued from previous page)

```
        'quickReplies': {
          'quickReplies': [
            'Happy :)',
            'Sad :('
          ]
        }
      }
    ]
}
```

CHAPTER

TEN

INSTALLATION

The preferred way to install *dialogflow-fulfillment* is from PyPI with pip:

```
$ pip install dialogflow-fulfillment
```

See also:

For further details about the installation, see *Installation*.

CHAPTER

ELEVEN

FEATURES

dialogflow-fulfillment's key features are:

- **Webhook Client**: handle webhook requests using a custom handler function or a map of handlers for each intent
- **Contexts**: process input contexts and add, set or delete output contexts in conversations
- **Events**: trigger follow-up events with optional parameters
- **Rich Responses**: create and send the following types of rich response messages:
  - Text
  - Image
  - Card
  - Quick Replies
  - Payload

CHAPTER

TWELVE

LIMITATIONS

Currently, *dialogflow-fulfillment* has some drawbacks, which will be addressed in the future:

- No support for platform-specific responses

PYTHON MODULE INDEX

d

dialogflow_fulfillment.contexts, 11
dialogflow_fulfillment.webhook_client, 7

INDEX

A

action (*WebhookClient attribute*), 7
add() (*WebhookClient method*), 8

B

buttons (*Card property*), 13

C

Card (*class in dialogflow_fulfillment.rich_responses*), 13
console_messages (*WebhookClient attribute*), 7
Context (*class in dialogflow_fulfillment.contexts*), 11
context (*WebhookClient attribute*), 7
contexts (*Context attribute*), 11
contexts (*WebhookClient attribute*), 7

D

delete() (*Context method*), 11
dialogflow_fulfillment.contexts
    module, 11
dialogflow_fulfillment.webhook_client
    module, 7

F

followup_event (*WebhookClient property*), 8

G

get() (*Context method*), 11
get_output_contexts_array() (*Context method*), 11

H

handle_request() (*WebhookClient method*), 9

I

Image (*class in dialogflow_fulfillment.rich_responses*), 15
image_url (*Card property*), 14
image_url (*Image property*), 15
input_contexts (*Context attribute*), 11
intent (*WebhookClient attribute*), 7

L

locale (*WebhookClient attribute*), 8

M

module
    dialogflow_fulfillment.contexts, 11
    dialogflow_fulfillment.webhook_client, 7

O

original_request (*WebhookClient attribute*), 8

P

parameters (*WebhookClient attribute*), 7
Payload (*class in dialogflow_fulfillment.rich_responses*), 16
payload (*Payload property*), 16

Q

query (*WebhookClient attribute*), 7
quick_replies (*QuickReplies property*), 17
QuickReplies (*class in dialogflow_fulfillment.rich_responses*), 17

R

request_source (*WebhookClient attribute*), 8
response (*WebhookClient property*), 10
RichResponse (*class in dialogflow_fulfillment.rich_responses.base*), 18

S

session (*Context attribute*), 11
session (*WebhookClient attribute*), 8
set() (*Context method*), 12
subtitle (*Card property*), 14

T

Text (*class in dialogflow_fulfillment.rich_responses*), 19
text (*Text property*), 19
title (*Card property*), 15
title (*QuickReplies property*), 18

W

WebhookClient (*class in dialogflow_fulfillment.webhook_client*), 7