flask_dialogflow

Release v2.0.0

Georg Molau

Sep 19, 2019

CONTENTS

flask_dialogflow is a Flask extension to build Dialogflow agents. It aims to shine through the following features:

- A familiar Flask extension structure that handles the mundane stuff behind the scenes
- Robust JSON serialization and deserialization of the entire Dialogflow and Actions on Google API to native Python classes
- A simple API for high-level Google Assistant features
- Special template features for voice assistants
- Support for multi-platform agents and extensibility to new platforms
- Integration with the Flask CLI and shell
- Helpers to test an agent
- A comprehensive test suite

Here is a quick example:

```
from flask import Flask
from flask_dialogflow.agent import DialogflowAgent

app = Flask(__name__)
agent = DialogflowAgent(app)

@agent.handle(intent='HelloWorld')
def hello_world(conv):
   conv.ask('Hello world!')
   return conv
```

For more information, check out the *Tutorial* and the *API documentation*.

CHAPTER

ONE

TUTORIAL

This tutorial aims to give an overview over the core features. For details on particular interfaces see the *API documentation*.

1.1 Installation and setup

Install the library from Pip. As usual, it is recommended to install into a virtualenv:

```
cd my_project
virtualenv venv
source venv/bin/activate
pip install flask_dialogflow
```

A Flask app can be initialized with a *DialogflowAgent* in two ways. One way is to pass the Flask instance directly to the init method:

```
app = Flask(__name__)
agent = DialogflowAgent(app)
```

The other way is to defer initialization until later and the calling *DialogflowAgent.init_app()* manually:

```
app = Flask(__name__)
agent = DialogflowAgent()

agent.init_app(app)
```

The latter works with application factories and is the recommended approach. In both cases the Flask app gets:

- A new route that accepts the webhook requests from Dialogflow.
- A second Jinja loader to be able to load the agents responses from a YAML file.
- A reference to the agent in the `Flask.extensions` dictionary.
- A reference to the agent in a Flask shell.

The only change that the agent makes to the Flask app is that it sets `Flask.jinja_env.auto_reload` to True. This is necessary to enable template randomization. See *Templating* for details.

The URL endpoint defaults to `/` and the templates file to `templates.yaml`. Both can be configured in the init method:

```
agent = DialogflowAgent(
    route='/agent', templates_file='agent/templates.yaml'
)
```

flask_dialogflow currently supports two versions of the Dialogflow API: `v2` and `v2beta1`. The latter is, despite its name, a superset of the former and therefore set as the default version. This means that the conversation objects will be of type *`V2beta1DialogflowConversation`* and that API objects such as Cards and Images should be imported from the `flask_dialogflow.google_apis.dialogflow_v2beta1` module.

While it is possible to change the version to `v2` there is not much point to this as the difference is minuscule. This option exists mostly to make the library forward compatible with future Dialogflow versions.

The DialogflowAgent has a debug mode that can be activate via the `debug` init param or the `flask_dialogflow_DEBUG` environment variable. It causes all webhook requests and responses to be logged to the console (prettified).

1.2 Google APIs and serialization

This library uses marshmallow to serialize and deserialize the Dialogflow and Actions on Google API objects, but this is completely abstracted. The objects are implemented as dataclasses and each have a corresponding marshmallow schema. Each class and schema are linked in such a way the entire de-/serialization process is hidden behind `from_json`/`to_json` methods on the classes. These classes implement the entire Dialogflow (v2, v2beta1) and Actions on Google API in three modules:

- `flask_dialogflow.google_apis.actions_on_google_v2`
- `flask_dialogflow.google_apis.dialogflow_v2`
- `flask_dialogflow.google_apis.dialogflow_v2beta1`

Here is an example of how it works:

```
from flask_dialogflow.google_apis.dialogflow_v2beta1 import Image

# Deserialization from JSON
Image.from_json(
    {'imageUri': 'https://image.png', 'accessibilityText': 'Image'}
)
# Image(image_uri='https://image.png', accessibility_text='Image')

# Serialization to JSON
Image(image_uri='https://image.png', accessibility_text='Image').to_json()
# {'imageUri': 'https://image.png', 'accessibilityText': 'Image'}
```

Note: By JSON, we always mean plain Python data structures that can be handled by `json.dumps()`/`json.loads()`, i.e. usually dictionaries. Pythons type system does unfortunately not allow recursive types, which is why we type JSON as `MutableMapping[str, Any]`.

This system powers the entire library and can also be used by users. See the *API documentations* section on JSON handling for details. Note also that users will sometimes have to import classes from the API modules directly, such as when using rich response items like cards or carousels.

The API classes are not documented because they map API interfaces into native Python classes. Because of that, users will have to consult the original Google documentations:

- The authoritative source for the Dialogflow API is the Dialogflow Discovery document.
- A web version of this is available on the Google Cloud Dialogflow page.
- The Actions on Google API is documented on the Actions on Google website.

Since the conversion from API objects (Protobuf messages) to Python classes is not an exact science, here are conversion rules that we have applied:

- Every API object becomes a Python dataclass.
- CamelCase attribute names are converted to snake_case.
- Names are kept as they are, except for a small number of cases were a class name is not unique across the API. In these cases the name is usually prepended with the enclosing messages name.
- All fields are optional unless a field is explicitly documented as required. In these cases we have set them as required here as well to avoid some `x is None` checks.
- Optional fields always default to None, except for lists and dictionaries. They default to empty collections to again avoid some None checks.
- Oneof fields are implemented as individual, optional attributes.
- Enums become Python enums.
- Structs become `Dict[str, Any]`.
- Numbers are typed as int when either the Discovery document or the comments in the web documentation clearly state them as such, even though the web documentation knows only numbers. Otherwise they are floats.

The marshmallow schemas are only used to map the attributes from API objects to classes. They perform no validation or type conversion, this, if at all, must be done by the Python classes.

1.3 Conversations and handlers

Conversation objects are the core idea of this library. They represent one turn of the conversation with the user and expose the request attributes as well as methods to build the response. *V2beta1DialogflowConversation* is the specific type that the conversation will be of under the default settings. It is initialized from the `WebhookRequest` behind the scenes and handed over to the appropriate handler function. After the handler has done its job it is supposed to hand it back to the library, which will render it to a `WebhookResponse`, serialize it to JSON and send it back to Dialogflow.

Conversations expose the request attributes as properties, e.g.:

```
conv.intent      # The intent name
conv.parameters  # The requests parameters
conv.session     # The session id
```

They also offer methods to build responses:

```
# A simple text response
conv.tell('Hello world!')

# Rendering a response from a template
from flask import render_template
conv.tell(render_template('hello'))

# Showing a card
from flask_dialogflow.google_apis.dialogflow_v2beta1 import Card
card = Card(title='Beautiful image', image_uri='image.png')
conv.show_card(card)
```

Conversations also give access to a requests contexts, for that see the Contexts section.

Conversation handlers implement the core business logic of the agent. They are functions that accept the conversation object, inspect its request attributes, perform necessary business logic, build the response and return the conversation object again. Handlers can be arbitrarily complex as long as they accept the conversation as their first argument and return it again.

Handlers can of course pass the conversation on to sub handlers. This makes the data flow easier to understand and test. Here is an example of a slightly more complex handler setup:

```
@agent.handle('SelectDate)
def choose_date_handler(conv):
    # Entry point for conversations for the SelectDate intent
    date = parse(conv.parameters['selected_date'])
    if date >= datetime.datetime.now():
        conv = valid_date(conv)
    else:
        conv = invalid_date(conv)
    return conv

def valid_date(conv):
    ... # Business logic
    conv.tell('Date was chosen!')
    return conv

def invalid_date(conv):
    ... # Business logic
    conv.tell('Date is invalid:(')
    return conv
```

The general idea is always that a handler gets a conversation, examines the request attributes, passes the conversation on to where the specific conversation state is best handled, builds the response and eventually hands the conversation back to the library, which will take care of rendering it correctly and sending it back.

Conversations are not meant to be inspected, i.e. one should never 'check' if a certain response was already set and then try to do something based on the result. Responses should be set once where it is appropriate and then not be touched anymore.

Dialogflow has some constraints on what kind of and how many responses go together (e.g. only two speech bubbles, one card etc.), but these are not enforced by the conversation object as they are not always clearly documented would make the API quite brittle. Users are expected to be familiar with the Dialogflow API and watch the Dialogflow logs for errors.

1.4 Templating

flask_dialogflow uses the Jinja2 templating library just like Flask itself, but adds two features to make it work better for voice assistants.

The first one is that we expect all templates to be assembled into a single YAML file. Each key of the file is its own template an can be rendered independently. They are of course full Jinja templates and can use all features of the Jinja templating language:

```
# A plain string template
welcome: Hi, welcome to SomeAgent!

# A template with a variable and a filter
confirm_delivery: Ok, your delivery will arrive by {{ date|format('%A') }}.
```

These two would be rendered like any normal Flask template and passed to the conversations response methods. Since we render templates a lot we typically alias the `render_template` function:

```
from flask import render_template as __

conv.tell(__('welcome'))
conv.tell(__('confirm_delivery', date=datetime.datetime.now()))
```

The second feature that we add is randomization. For voice assistants it is typically desirable to vary each speech response somewhat so as not to sound robotic. flask_dialogflow makes this simple by supporting randomization out of the box. It can be used by using arrays of different formulations for one template in the templates file:

```
welcome:
  - Hi, welcome to SomeAgent!
  - Hi there, SomeAgent here.
  - Hello, here is SomeAgent!
```

This template is rendered as usual (`render_template('welcome')`), but one of the three variations will be chosen at random.

It is also possible to weigh the options by specifying them as two-element arrays, where the second element is the weight. The weight is optional and defaults to 1:

```
welcome:
  - ['Hi, welcome to SomeAgent!', 2]
  - Hi there, SomeAgent here.
  - ['Yo, wazzup? SomeAgent here for you.', 0.5]
```

In this case the first variant has a probability of ~57% (=2/3.5), the second of ~29% (=1/3.5) and the third of ~14% (0.5/3.5). When using this option care has to be taken to properly quote the strings so as to not accidentally malform the array.

1.5 Contexts

Contexts are essential to realize complex, multi-turn dialogs. Conversations expose a requests contexts via the `V2beta1DialogflowConversation.contexts` attribute, which returns a *`ContextManager`* that has methods to get, set, check and delete a context.

Checking if a context is present:

```
conv.contexts.has('some_ctx')

# Or shorter:
'some_ctx' in conv.contexts
```

Getting a context, returning a *`flask_dialogflow.context.Context`* instance:

```
conv.contexts.get('some_ctx')

# Or shorter via attribute access:
conv.contexts.some_ctx
```

Setting a context:

```
# Setting an empty context with the default lifespan:
conv.contexts.set('some_ctx')

# Customizing the lifespan:
conv.contexts.set('some_ctx', lifespan_count=3)

# Including context parameters:
conv.contexts.set('some_ctx', lifespan_count=3, some_param='some_value')

# Initializing a complex context up front and setting it:
from flask_dialogflow.context import Context
ctx = Context(
    'some_ctx',
    lifespan_count=3,
    parameters={'foo': 'bar'}
)
conv.contexts.set(ctx)
```

Deleting a context still sends it back in the next response, but with a lifespan of 0 to ensure that it gets deleted in Dialogflow:

```
conv.contexts.delete('some_ctx')

# Or shorter:
del conv.contexts.some_ctx
```

Often one would like to have guarantees about the state of certain contexts. It is therefore possible to register contexts on the agent via *DialogflowAgent.register_context()*.

Keeping a context around: This ensures that it never expires by resetting its lifespan to a high value on each request. This happens before the conversation is passed to the handler, so the handler can still delete the context manually:

```
agent.register_context('some_ctx', keep_around=True)
```

This does not create a context when it doesn't exist. For that use a default factory, that initialized a context with the results of this factory as the parameters when it is not part of the request:

```
# This context will be initialized with an empty parameters dict
agent.register_context('some_ctx', default_factory=dict)

# This context has some parameters already set
agent.register_context(
    'some_other_ctx', default_factory=lambda: {'foo': 'bar'}
)
```

Setting both `keep_around` and `default_factory` ensures that a context is always present and `conv.contexts.some_ctx` never raises an AttributeError.

For complex contexts it is desirable to have the parameters attribute not be a dictionary, but rather a class instances. This requires that the instance can be serialized to JSON. Context can therefore be register with a serializer and deserializer function. The result of the deserializer will be bound to the parameters attribute when the conversation is initialized. After handling the serializer will be used to convert the instance back to JSON. This makes it possible to use arbitrary Python classes as contexts and hence attach business logic to them.

To make this even easier there is an `DialogflowAgent.context` decorator that can be used on *JSONType* subclasses. It will set the serializer, deserializer and default_factory automatically (should the default_factory not be needed it can be set to None). Here is an example of how this can be used to implement a GameState context for a quiz game:

```
# Implement the game state class and schema
from marshmallow.fields import Int, Str
from flask_dialogflow.json import JSONType, JSONTypeSchema

class _GameStateSchema(JSONTypeSchema):
    questions_answered = Int()
    last_answer = Str()

@agent.context('game_state', keep_around=True)
@dataclass
class GameState(JSONType, schema=_GameStateSchema):
    questions_answered: int = 0
    last_answer: Optional[str] = None
```

This ensures that:

- The `game_state` context will always be present.
- It will be correctly initialized if necessary.
- Its lifespan never expires.
- The `Context.parameters` are an instance of the GameState class, not a dict.

In a handler this context could be used like this:

```
@agent.handle('CorrectAnswer')
def handle_correct_answer(conv):
    conv.contexts.game_state.parameters.questions_answered += 1
    conv.contexts.game_state.parameters.last_answer = ...
    return conv
```

1.6 Integrations

Dialogflow is a generic Google Cloud API that can be integrated with a large number of different platforms. The most well-known of the is Actions on Google (i.e. the Google Assistant), others are Slack, Facebook Messenger and Telegram. It is also possible to integrate Dialogflow with custom platforms such as proprietary chat platforms or third party smart speakers.

flask_dialogflow supports all of these use cases. There is extensive support for Actions on Google (see below), basic support for the other integrations and tools to build helpers for custom integrations.

Integrations can send platform-specific data in the webhook request and receive platform-specific responses in the webhook response, they essentially piggyback on the Dialogflow webhook protocol. Because of this we give them each its own conversation object that is accessible via the overall DialogflowConversation object.

All integration conversations must subclass the *AbstractIntegrationConversation*, which ensures that they can be initialized from a request and rendered to a response. The default implementation of this interface is *GenericIntegrationConversation*, which behaves like a dict. This class is used for all integrations except Actions on Google, which has a more elaborate class.

Dialogflow's default integrations are set up in the conversation by default. This means that platform-specific responses can be included without further setup, enabling multi-platform agents out of the box:

```
conv.facebook['foo'] = 'bar'  # Response only for Facebook
conv.slack['bar'] = 'baz'     # This is for Slack
```

What kind of responses the platforms accept depends on them and has to be looked up in their documentation.

It is also possible to register new integrations via `DialogflowAgent.register_integration`. This is useful when the Dialogflow API is used from a custom system that has additional features. An example of this would be a custom smart speaker that has a blinking light that can be controlled via parameters in the response payload. This would be a case were it is useful to implement a custom conversation class to abstract this functionality and to register it on the agent.

```
from flask_dialogflow.integrations import GenericIntegrationConversation

class BlinkingLightSpeakerConv(GenericIntegrationConversation):
    # Subclass the generic conv to get the usual dict behavior

    def blink(times=1):
        # Build the JSON payload that makes the light blink
        self['blink'] = times

agent.register_integration(
    source='blink_speaker',
    integration_conv_cls=BlinkingLightSpeakerConv
)
```

Now, every DialogflowConversation passed to a handler will have an instance of this special conversation object that can be used to make the light blink:

```
@agent.handle('BlinkTwice')
def blink_twice_handler(conv):
    conv.blink_speaker.blink(times=2)
    # ... other response parts as usual
    return conv
```

Should the speaker carry data when calling Dialogflow (via the `OriginaDetectIntentRequest.payload`), it can be made available via the conversation class just like any other request attributes. Let's assume the speaker would tell the webhook whether the light is currently on or off by sending `{'light_on': True}` in the payload. The conversation class could then make this info available like this:

```
from flask_dialogflow.integrations import GenericIntegrationConversation

@agent.integration('blink_speaker')
class BlinkingLightSpeakerConv(GenericIntegrationConversation):

    @property
    def light_on(self) -> bool:
        # The GenericIntegrationConversation is already a dict, we
        # simply expose this attribute as a property for
        # convenience
        return self['light_on']

    def turn_light_off(self):
        # Method to turn the light off (assuming the speaker
        # handles this)
        self['light_on'] = False
```

This can now be used in handler functions as well:

```
@agent.handle('TurnLightOff')
def turn_light_off_handler(conv):
    if conv.blink_speaker.light_on:
```

(continues on next page)

(continued from previous page)

```
        conv.blink_speaker.turn_light_off()
    return conv
```

1.7 Actions on Google

Actions on Google (AoG) is the most important integration of Dialogflow, many agents will probably never use another one. Because of this AoG has a fairly elaborate conversation class that is available via `conv.google`: *`V2ActionsOnGoogleDialogflowConversation`*. This class should always be used for AoG in favor of Dialogflow's generic responses, and when an agent is only targeted for the Google Assistant it is perfectly fine to use it exclusively.

Because it works just like the normal conversation, we only highlight the most important features here, see the *API docs* for a full reference.

AoG by default sends all responses as SSML. This means that templates can contain SSML tags and just work:

```
welcome: Hi there! <audio src="https://some_jingle.mp3"/>
```

```
conv.google.tell(__('welcome'))  # Plays the jingle
```

AoG supports system intents that take over the conversation for a brief period of time and obtain standardized information from the user. System intents are implemented as methods on the AoG conversation object and are typically named `ask_for_*`. for example:

```
# Ask for permission to get the users name
conv.google.ask_for_permission('To greet you by name', 'NAME')

# Ask for a confirmation
conv.google.ask_for_confirmation('Do you really want to do this?')

# Ask the user to link a third-party OAuth account
conv.google.ask_for_sign_in('To access your Tinder account')

# Ask for a selection from a list
from flask_dialogflow.google_apis.actions_on_google_v2 import ListSelect
list_select = ListSelect(...)  # Build the ListSelect
conv.google.ask_for_list_selection(list_select)
```

The response to a system intent is usually included in the `conv.google.inputs` array of the next request. The precise format varies and has to be looked up in the AoG docs.

AoG has a user_storage field that makes it possible to persist user information server side across sessions (thereby differing from Dialogflow contexts, which are always bound to a session). This field is available under `conv.google.user.user_storage` and makes use of the same serialization system as the contexts. It is by default treated as a dict and de-/serialized with json.loads/dumps, which means that all of its attributes must be JSON-serializable.

Should a more elaborate system be needed, such as a custom user storage class, it can be configured via the DialogflowAgents init params (`aog_user_storage_deserializer`, `aog_user_storage_serializer`, `aog_user_storage_default_factory`). The behavior is the same as for the contexts.

1.8 Testing

The DialogflowAgent has a special *DialogflowAgent.test_request()* method that can be used to quickly construct webhook requests and route them trough the agent. The response will be a special `WebhookResponse` subclass that makes it easy to make assertions about the response. For example:

```
# Call the Welcome intent
resp = agent.test_request('Welcome')

# Assert a text response
assert 'Hi, welcome to SomeAgent!' in resp.text_responses()

# Assert that a certain context is present
assert resp.has_context('some_ctx)

# Get the context to inspect it in more detail
resp.context('some_ctx')
```

Note that the helper currently only support the generic Dialogflow responses, the AoG response have to be inspected manually (`resp.payload['google']`).

1.9 Flask CLI and shell

The agent adds a `agent` sub command to the Flask CLI that can be used to quickly get information about the agent. It supports the following commands:

```
$ flask agent intents
# Prints a table with the registered intents and handlers

$ flask agent contexts
# Prints a table with the registered contexts

$ flask agent integrations
# Prints a table with the registered integration conversation classes
```

The agent is also available in a `flask shell` under the `agent` name. This in combination with *DialogflowAgent.test_request()* is the quickest way to test the agent during development.

CHAPTER

TWO

API REFERENCE

This part of the documentation covers all the interfaces of flask_dialogflow.

2.1 Agent object

The agent is the core object of this library.

class flask_dialogflow.agent.**DialogflowAgent**(*app: Optional[flask.app.Flask] = None, version: Optional[str] = 'v2beta1', route: Optional[str] = '/', templates_file: Optional[str] = 'templates.yaml', debug: Optional[bool] = False, aog_user_storage_default_factory: Optional[Callable[[], T]] = <class 'dict'>, aog_user_storage_deserializer: Optional[Callable[[str], T]] = <function loads>, aog_user_storage_serializer: Optional[Callable[[T], str]] = <function dumps>, aog_text_to_speech_as_ssml: Optional[bool] = True*)

Dialogflow agent.

This is the central object that represents the Dialogflow agent and integrates this library with Flask. It keeps track of registered intent handlers, contexts and integrations and handles requests behind the scenes. It initializes a `DialogflowConversation` for each requests and hands it over to the corresponding handler, which does the actual business logic needed to fulfill the request.

Parameters

- **app** – The Flask app to initialize this agent with.
- **version** – The version of the Dialogflow API to use. Defaults to v2beta1, which despite its name appears to be a superset of v2 (i.e. is completely compatible with it).
- **route** – The URL endpoint under which this agent will be served. Will be registered on the Flask app to accept POST requests.
- **templates_file** – A single YAML file with the Jinja templates. See *templating* for details. The path must be relative to the Flask apps root_path.
- **debug** – Debug mode for the agent. If on, every request and response will be logged as prettified JSON. Can be set via the flask_dialogflow_DEBUG environment variable.
- **aog_flask_dialogflow_default_factory** – The default factory to use for the user_storage of the AoG integration.

- **aog_user_storage_deserializer** – The function to deserialize the user_storage of the AoG integration.
- **aog_user_storage_serializer** – The function to serialize the user_storage of the AoG integration.
- **aog_text_to_speech_as_ssml** – Whether to send text responses for Actions on Google as SSML by default. This makes it possible to use SSML directives in templates without additional setup.

init_app(*app: flask.app.Flask*, *route: Optional[str] = None*, *templates_file: Optional[str] = None*) → None

Initialize a Flask app.

This can be used to manually initialize a Flask app when it wasn't passed to init. Adds the route, the template loader and a shell context processor. Sets auto_reload to True on the Jinja env.

Parameters

- **app** – The Flask app to initialize with this agent.
- **route** – The URL endpoint for this agent. If None, defaults to the agents route.
- **templates_file** – The YAML templates file. If None, defaults to the agents template file.

Returns None

register_handler(*intent: str, handler: Callable[[Union[flask_dialogflow.conversation.V2DialogflowConversation, flask_dialogflow.conversation.V2beta1DialogflowConversation]], Union[flask_dialogflow.conversation.V2DialogflowConversation, flask_dialogflow.conversation.V2beta1DialogflowConversation]]*) → None

Register a conversation handler.

Takes the name of an intent (the display_name, i.e. the name it was given in the Dialogflow console) and registers a handler function for it. All requests to this intent will then be routed to this handler.

Parameters

- **intent** – The intent to register the handler for.
- **handler** – The conversation handler for this intent.

Returns None

handle(*intent: str*)

Decorator to register conversation handlers.

Example:

```
@agent.handle('HelloWorld')
def hello_world_handler(conv):
    # This handler will be called for requests to
    # the HelloWorld intent
    conv.ask('Hello world!')
    return conv
```

Parameters **intent** – The intent to register the handler for.

Returns The decorator to be applied to a conversation handler.

register_integration(*source: str, integration_conv_cls: Type[AbstractIntegrationConversation], version: Optional[str] = None, integration_conv_cls_kwargs: Optional[Mapping] = None*) → None

Register an integration conversation class.

This can be used to register conversation classes for custom integrations, i.e. subclasses of *AbstractIntegrationConversation*. The class will then be available via the standard DialogflowConversation. Should the webhook request from this integration carry custom payload it too will be available via conversation object.

Example:

Assume you want to integrate your Dialogflow agent with a custom speaker that has a blinking light that can be controlled via the webhook. You could then write a custom conversation class that abstracts this functionality:

```
from flask_dialogflow.integrations import GenericIntegrationConversation

class BlinkingLightSpeakerConv(GenericIntegrationConversation):
    # Subclass the generic conv to get the usual dict behavior

    def blink(times=1):
        # Build the JSON payload that makes the light blink
        self['blink'] = times

agent.register_integration(
    source='blink_speaker',
    integration_conv_cls=BlinkingLightSpeakerConv
)
```

Now, every DialogflowConversation passed to a handler will have an instance of this special conversation object that can be used to make the light blink:

```
@agent.handle('BlinkTwice')
def blink_twice_handler(conv):
    conv.blink_speaker.blink(times=2)
    # ... other response parts as usual
    return conv
```

The speaker could carry data when calling Dialogflow (via the `OriginaDetectIntentRequest.payload`), which can be made available via the conversation class. Let's assume the speaker would tell the webhook whether the light is currently on or off by sending `{'light_on': True}` in the payload. The conversation class could then make this info available like this:

```
from flask_dialogflow.integrations import GenericIntegrationConversation

class BlinkingLightSpeakerConv(GenericIntegrationConversation):

    @property
    def light_on(self) -> bool:
        # The GenericIntegrationConversation is already a dict, we
        # simply expose this attribute as a property for
        # convenience
        return self['light_on']

    def turn_light_off(self):
        # Method to turn the light off (assuming the speaker
        # handles this)
        self['light_on'] = False
```

This can now be used in handler functions as well:

```
@agent.handle('TurnLightOff')
def turn_light_off_handler(conv):
    if conv.blink_speaker.light_on:
        conv.blink_speaker.turn_light_off()
    return conv
```

Parameters

- **source** – The integration platform to use this conversation for.
- **integration_conv_cls** – The conversation class to use for this integration.
- **version** – Optional version qualifier for the source.
- **integration_conv_cls_kwargs** – Kwargs to pass to the conversations from_webhook_request_payload method.

Returns None

integration(*source: str*, *version: Optional[str] = None*, ***kwargs*)
Decorator version of *register_integration()*.

Parameters

- **source** – The integration platform to use this conversation for.
- **version** – Optional version qualifier for the source.
- ****kwargs** – Kwargs to pass to the conversations from_webhook_request_payload method.

register_context(*display_name: str, keep_around: Optional[bool] = False, default_factory: Optional[Callable[[], CtxT]] = None, deserializer: Optional[Callable[[MutableMapping[str, Any]], CtxT]] = None, serializer: Optional[Callable[[CtxT], MutableMapping[str, Any]]] = None*) → None
Register a context.

Registering a context abstracts certain parts of context handling, making them easier to work with. Most importantly, it makes it possible to represent the parameters of a context as a class instead of a plain dictionary and have de-/serialization handled behind the scenes.

Parameters

- **display_name** – The display name of the context to register.
- **keep_around** – Ensure that this context never expires by resetting its lifespan to a high value on each request. This happens before the handler is called, meaning the context can still be expired manually. This does not create a context when it doesn't already exist, use default_factory for that.
- **default_factory** – A factory to initialize a context when it is not present in a request. This function must only return the context parameters (either a dict or a class instance), it will be wrapped in a *Context* object automatically. Setting this in combination with keep_around ensures that a context will always be present, i.e. that `conv.contexts.some_ctx` never raises an AttributeError.
- **deserializer** – Function to deserialize the context parameters with. Context params will be deserialized with json.load, this function can be used to deserialize them further into a class. This makes it possible to work with context params as class instances instead of dicts and to implement custom context classes with additional business logic. Care has

to be taken though because Dialogflow adds its own fields to contexts, the deserializer has to be able to ignore them.

- **serializer** – Function with which the context params will be serialized to JSON.

Returns None

context(*display_name: str*, ***kwargs*) → Callable[[Type[CtxT]], Type[CtxT]]

Decorator version of register_context.

This decorator can be applied to *JSONType* classes which have de-/serialization built in and set the correct deserializer/serializer functions automatically. For details on how the JSONTypes work see the section on *JSON handling*. Here is an example how one could realize a game state context with this:

```
# Implement the game state class and schema
from marshmallow.fields import Int, Str
from flask_dialogflow.json import JSONType, JSONTypeSchema

class _GameStateSchema(JSONTypeSchema):
    questions_answered = Int()
    last_answer = Str()

@agent.context('game_state', keep_around=True)
@dataclass
class GameState(JSONType, schema=_GameStateSchema):
    questions_answered: int = 0
    last_answer: Optional[str] = None
```

This ensures that:

- The `game_state` context will always be present.
- It will be correctly initialized if necessary.
- Its lifespan never expires.
- The `Context.parameters` are an instance of the GameState class, not a dict.

In a handler this context could be used like this:

```
@agent.handle('CorrectAnswer')
def handle_correct_answer(conv):
    conv.contexts.game_state.parameters.questions_answered += 1
    conv.contexts.game_state.parameters.last_answer = ...
    return conv
```

Applying this decorator to a non-JSONType class requires that the deserializer and serializer are provided manually, which is the same as calling *register_context()* directly.

Parameters

- **display_name** – The display name of the context to register.
- ****kwargs** – The same kwargs that *register_context()* takes.

Returns A class decorator for JSONType subclasses.

list_handler() → Iterable[Tuple[str, Callable[[Union[flask_dialogflow.conversation.V2DialogflowConversation, flask_dialogflow.conversation.V2beta1DialogflowConversation]], Union[flask_dialogflow.conversation.V2DialogflowConversation, flask_dialogflow.conversation.V2beta1DialogflowConversation]]]]

List all registered handlers.

Yields Tuples of (intent name, handler function).

list_integrations() → Iterable[Tuple[str, Optional[str], flask_dialogflow.integrations.AbstractIntegrationConversation, Optional[Mapping]]]
List all registered integrations.

Yields Tuples of (source, integration conv class, version, kwargs).

list_contexts() → Iterable[flask_dialogflow.context.ContextRegistryEntry]
List all registered contexts.

Yields ContextRegistryEntry objects that contain information about the contexts.

test_request(**args*, ***kwargs*) → flask_dialogflow.agent.TestWebhookResponse
Make a test request.

This builds a `WebhookRequest` from the passed parameters and processes it like a normal request. Everything that happens between the deserialization of a requests POST payload and the serialization of the handlers response will also happen during this test request. It can thus be used to quickly test the agents request handling end-to-end.

Example:

```
resp = agent.test_request('HelloWorld')
# Builds a request for the 'HelloWorld' intent and passes it
# through the agent. resp is now the webhook response that would be
# send back to Dialogflow.
```

This does not involve Flask and does thus also not need an active app or request context.

Parameters **kwargs** (*args,*) – The arguments are the same that *build_webhook_request()* takes. The first one is the intent name.

Returns An instance of *TestWebhookResponse*, a `WebhookRequest` subclass that offers some additional methods to make assertions about the response.

2.2 Conversation objects

Conversation classes are the core abstraction of this library. They come in two versions for the two supported Dialogflow version, but are, except for some additional features in v2beta1, completely identical.

class flask_dialogflow.conversation.**V2DialogflowConversation**(*webhook_request: Optional[flask_dialogfl = None, context_manager: Optional[ContextManager] = None, integration_convs: Optional[Mapping[str, flask_dialogflow.integrations.AbstractIntegr = None*)

The core Dialogflow Conversation object.

This object is the heart of this library. It represents a single turn in a Dialogflow conversation and is the interface to both the incomint request data as well as to the response construction methods. This object is instantiated by flask_dialogflow automatically and then passed to the handler function matched to this request. The handler function will usually inspect the request data in more detail, perform some business logic, maybe update the

server-side state (contexts, user storage) and then build a response before returning the conversation object back to the library. It will then be rendered into a webhook response and serialized to JSON behind the scenes.

This class is specific to v2 of the Dialogfow API. There is a corresponding *V2beta1DialogflowConversation* for v2beta1. These two are currently the only supported Dialogflow versions. (v2beta1 appears, despite its name, to be a superset of v2, there is thus no harm in always using it, which is why it is the default conversation class.)

The DialogflowConversation does also carry integration-specific conversation classes to implement features specific to individual integrations. The most important of them is `V2ActionOnGoogleDialogflowConversation` for the Actions on Google integration. It is registered on the agent by default and always available under the `google` attribute. See *Integrations* for details.

Note that the response methods on this class refer to the generic Dialogflow responses. Some integrations, particularly Actions on Google, have their own set of much more elaborate responses. The methods here should thus only be used when cross-platform compatibility is desired. For agents that are only used with Actions on Google one should always use the `V2DialogflowConversation.google` methods exclusively. The other integration convs are currently `GenericIntegrationConversations`, which behave like dicts. Users can implement their own conversation classes and register them on the agent to support custom features.

property webhook_request
The `WebhookRequest` that this conversation represents.

It is usually not necessary and not recommended to interact with this directly, it is offered as a fallback option to give access to the raw request data. Modyfing this is highly discouraged and may lead to unexpected results.

property session
This requests session id.

property response_id
This requests response id.

property query_text
This requests query text (i.e. the text spoken by the user).

property language_code
This requests language code.

property intent
This requests intent (display name).

property action
This requests action.

property contexts
This requests incoming contexts.

This returns a special *ContextManager* object that provides a simple API to manage the conversations context state. See its documentation for details.

property parameters
This requests parameters.

property all_required_params_present
Whether all required parameters for this intent are present.

property fallback_level
This requests fallback level.

Default is 0, the first fallback intent gets level 1. If this is immediately followed by another fallback intent (i.e. the user was still not understood) the level is 2 and so on. The next non-fallback intent resets the level to 0.

It is good design practice to handle the levels differently, see the Design guidelines for details.

property diagnostic_info
This requests diagnostic info.

property intent_detection_confidence
This requests intent detection confidence.

property speech_recognition_confidence
This requests speech recognition confidence.

property sentiment
This requests sentiment.

property source
This requests source (i.e. the integration platform).

property version
This requests source version (usually only set for AoG).

property payload
This requests integration payload.

This platform-specific payload will be used to initialize the integration convs. Users should typically access these directly (via `V2DialogflowConversation.google` etc.), the raw data is only included as a fallback option. Modifying it is highly discouraged.

property integrations
The dictionary of integration convs.

The default integrations (AoG, Facebook, Slack ectc) have their own properties and do not need to access their convs via this dictionary, but custom integration platforms will. It is a default dict that returns a *GenericIntegrationConversation* by default, which means that new platforms can be used without additional setup.

This class implements a __getattr__ method that looks up attributes in the integrations mapping. These two lines are therefore equivalent:

```
conv.integrations['foobar']
conv.foobar  # Same thing
```

ask(**texts*) → None
Ask the user something.

The v2 has no endInteraction field, which probably implies that the session can not be closed manually. v2beta1 has a separate tell() function that does end the interaction.

Parameters **texts** – The texts to speak.

show_quick_replies(**quick_replies*, *title: Optional[str] = None*) → None
Show quick replies.

Parameters

- **quick_replies** – The replies to suggest.
- **title** – The title of the replies collection.

show_card(*card: flask_dialogflow.google_apis.dialogflow_v2.Card*) → None
Show a card.

Parameters **card** – The card to show.

show_image(*image: flask_dialogflow.google_apis.dialogflow_v2.Image*) → None
Show an image.

Parameters **image** – The image to show.

property google
The Actions on Google conversation object.

This objects abstracts all AoG-specific features. When AoG is the only integration where an agent is used it is perfectly fine to use this exclusively.

property facebook
The Facbook integration conv.

property slack
The Slack integration conv.

property telegram
The Telegram integration conv.

property kik
The Kik integration conv.

property skype
The Skype integration conv.

property twilio
The Twilio integration conv.

property twilio_ip
The TwilioIP integration conv.

property line
The Line integration conv.

property spark
The Spark integration conv.

property tropo
The Tropo integration conv.

property viber
The Viber integration conv.

to_webhook_response() → flask_dialogflow.google_apis.dialogflow_v2.WebhookResponse
Render the `WebhookResponse` for this conversation.

This is the last step during conversation handling and is usually done automatically by the framework. Modifying the conversation after the response has been rendered may lead to unexpected results.

Returns A complete Dialogflow WebhookResponse that can be serialized to JSON.

class flask_dialogflow.conversation.**V2beta1DialogflowConversation**(*webhook_request: Optional[flask_dialogflow.google_apis.* = *None*, *context_manager: Optional[ContextManager]* = *None*, *integration_convs: Optional[Mapping[str, flask_dialogflow.integrations.Abstrac* = *None*)

The v2beta1 version of the DialogflowConversation.

This has a few additional features, but is otherwise completely identical to the *V2DialogflowConversation*.

tell(**texts*) → None
Like ask, but the interaction is ended after it.

property alternative_query_results
Alternative QueryResults from knowledge connectors.

2.3 Contexts and Context Manager

Contexts are essential to manage (Dialogflow) server side state. These tools help in doing that accurately.

class flask_dialogflow.context.**Context**(*name: Optional[str] = None*, *lifespan_count: Optional[int] = None*, *parameters: CtxT = None*)

A wrapper around the API context.

Adds a display_name property and is parametrizable to give accurate type hints when using anything else but a dict for the parameters attribute (i.e. when registering this display name with a context class). Otherwise exactly the same as the API Context object.

property display_name
Get the contexts display name, i.e. without the session id.

classmethod from_context(*ctx: flask_dialogflow.google_apis.dialogflow_v2.Context*)
Initialize this class from an API context.

class flask_dialogflow.context.**ContextManager**(*contexts: Optional[Iterable[flask_dialogflow.context.Context]]* = *None*, *session: Optional[str]* = *''*, *context_registry: Optional[flask_dialogflow.context.ContextRegistry]* = *None*)

Interface to the collections of contexts on a conversation.

Contexts are server-side state that have to be managed from the client, i.e. this agent. This class represents the collection of contexts of a Dialogflow conversation and presents a set of methods to manage them in a predictable way.

This object is what is returned by V2DialogflowConversation.contexts.

Parameters

- **contexts** – An iterable of incoming contexts.
- **session** – This requests session id. Required to build full context names.
- **context_registry** – A reference to the context_registry of an agent.

get(*display_name: str*) → flask_dialogflow.context.Context
Get a context by its display name.

A shorter way to do this is via attribute access:

```
# These two are equivalent:
conv.contexts.get('foo_context')
conv.contexts.foo_context
```

Returns The complete context object, if present.

Raises **KeyError** – When the context is not present.

set(*display_name_or_ctx_instance: Union[str, flask_dialogflow.context.Context], lifespan_count: Optional[int] = None, **parameters*) → None
Set a context.

Parameters

- **display_name_or_ctx_instance** – Either the display name of the new context (will be concatenated with the session id) or a complete *Context* instance.
- **lifespan_count** – The lifespan of the new context. None defaults to Dialogflows default, currently 5.
- **parameters** – Params for this context, i.e. the context data.

Returns None

Raises **ValueError** – If either a context instance was given and the a separate lifespan or params set or the display name is invalid.

delete(*display_name: str*, *keep_in_registry: Optional[bool] = True*) → None
Delete a context.

Deleting a context means settings its lifespan to zero, which will cause Dialogflow to delete them server side. This is why deleted contexts will still be included in the next webhook response (with lifespan 0).

This too works via attribute access:

```
# These two are equivalent:
conv.contexts.delete('foo_context')
del conv.contexts.foo_context
```

Parameters

- **display_name** – The display_name of the context to delete.
- **keep_in_registry** – Keep the context in the agents context registry, should it have been in there.

Returns None

Raises **KeyError** – If a context with this name doesn't exist.

has(*display_name: str*) → bool
Check whether a context is present.

This does not include deleted contexts, even though they will still be included in the next webhook response (to set their lifespan to 0).

A shorter version of this is the `in` operator:

```
# These two are equivalent:
conv.contexts.has('foo_context')
'foo_context' in conv.contexts
```

Returns True if it is present, false if not.

as_list() → List[flask_dialogflow.context.Context]
Render the current collection of contexts as a list.

This will we called automatically to add the contexts to the WebhookResponse. Contexts should not be modified after this has been called.

Returns A list of context objects.

2.4 Integration Conversation objects

Dialogflow integrations get their own conversation objects, which work like the standard Dialogflow conversation object. They make it possible to include platform-specific responses, even for new or completely custom platforms. The default integration conversation is *GenericIntegrationConversation*, which works like a dict. It is used for all integrations that do not have a special conversation class registered. Actions-on-Google has a custom conversation object that supports AoG's special features. It is registered for AoG requests by default.

class flask_dialogflow.integrations.**AbstractIntegrationConversation**
Interface for integration-specific conversation objects.

This interface mandates methods to initialize a conversation from a webhook request payload and to render it to JSON for the webhook response. All custom integration convs must implement this interface.

abstract classmethod from_webhook_request_payload(*payload: Optional[MutableMapping[str, Any]] = None*, ***kwargs*) → flask_dialogflow.integrations.AbstractIntegrationConv
Initialize this conversation from the webhook request payload.

Webhook requests contain platform-specific payload. This payload should be exposed by the conversation class. This method mandates that the conv can be instantiated from the payload (optionally with additional kwargs). The payload may be None or an empty dict, implementations should be able to handle this.

Parameters

- **payload** – The webhook request payload for this integration.
- ****kwargs** – Additional kwargs, which can be set when registering this integration with an agent.

Returns An instance of this conversation, which will be available via the `DialogflowConversation`.

abstract to_webhook_response_payload() → MutableMapping[str, Any]
Render this conversation back to JSON.

This method must render the handled conversation back to JSON to be included as the integration payload in the webhook response.

Returns The fully processed conversation.

class flask_dialogflow.integrations.**GenericIntegrationConversation**(*data: Optional[MutableMapping[str, Any]] = None*)

Generic integration conversation.

This is the default conversation used for all integrations that don't have a custom conversation registered. It implements the MutableMapping ABC, which means it can be treated as a dict.

2.5 Actions on Google Conversation object

Actions on Google is currently the only integration platform that has a custom conversation class. It supports advanced AoG features such as additional rich responses, system intents, permissions and user storage.

class flask_dialogflow.integrations.actions_on_google.**V2ActionsOnGoogleDialogflowConversati**

Conversation class for the Actions on Google integration.

This class implements all AoG specific features. It is registered as the integration class for AoG by default and available via the `DialogflowConversation.google` attribute. It exposes AoG specific request attributes

and offers methods to build AoG responses. It also handles the de-/serialization of the AoG user storage.

When a Dialogflow agent is only meant to be used via Actions on Google, all responses can simply be set on this class. It is however perfectly possible to use this next to another integration class to realize agents for multiple platforms.

classmethod from_webhook_request_payload(*payload: Optional[MutableMapping[str, Any]] = None*, ***kwargs*) → flask_dialogflow.integrations.actions_on_google.V2ActionsOnGoog

Initialize this conversation from a webhook request payload.

Parses the payload to an `AppRequest` and initializes the conv.

Parameters

- **payload** – The `OriginalDetectIntentRequest.payload` of a webhook request from AoG.
- ****kwargs** – Kwargs that init takes.

property app_request

The underlying `AppRequest`.

Should usually not be needed, but might be useful to access the raw request data.

property user

The `User` of this AppRequest.

This returns a special *`UserFacade`* object, which wraps the original User object and adds some more features.

property inputs

The sequence of `Inputs` of this request.

property surface

The surface `Capabilities` as a sequence of strings.

I.e. something like `('actions.capability.SCREEN_OUTPUT', 'actions.capability.AUDIO_OUTPUT')`. See Surface capabilities for more details.

property has_screen

Whether this request has the SCREEN_OUTPUT capability.

property available_surfaces

The available surface capabilities that can be handed off to.

Returns a sequence of the capabilities name, just like `surface`.

property is_in_sandbox

Whether this is a sanbox request.

ask(**texts*) → None

Ask the user something.

This implies that the session is kept open. Multiple texts will be concatenated wit a space and end up in one speech bubble. Call this method multiple times to produce multiple bubbles, but beware that there is currently a limit of two bubbles.

Parameters **texts** – The texts to speak.

ask_ssml(**texts*) → None

Explicitly ask something in SSML.

This can be used to force SSML when the ssml-by-default option is turned off. Wraps the text in <speak> tags automatically.

Parameters **texts** – The texts to speak.

tell(**texts*) → None
Tell the user something.

This implies that the session will be closed. Other behavior is the same as for *ask()*.

Parameters **texts** – The texts to speak.

tell_ssml(**texts*) → None
Explicitly tell something in SSML.

Equivalent of *ask_ssml()*, session will be closed.

Parameters **texts** – The texts to speak.

display(**texts*)
Set a separate display text on the last text response.

The spoken and the displayed text should normally not diverge too much, but there might be cases were the spoken text is very colloquical and a separate display text is desired. This adds a separate display text to the last text response. Can be used after ask, ask_ssml, tell and tell_ssml.

Parameters **texts** – The texts to display.

Raises **ValueError** – If no text response has been set yet.

suggest(**suggestions*) → None
Display suggestion chips.

Can be called once with multiple suggestions or multiple times in a row or both. Suggestions are kept in the order they are set, but are not de-duplicated.

Parameters **suggestions** – The suggestions to display.

show_basic_card(*basic_card: flask_dialogflow.google_apis.actions_on_google_v2.BasicCard*) → None
Show a `BasicCard`.

Parameters **basic_card** – The card to show.

show_image(*url: str*, *accessibility_text: str*, *height: Optional[float] = None*, *width: Optional[float] = None*, *image_display_options: Optional[flask_dialogflow.google_apis.actions_on_google_v2.ImageDisplayOptions] = None*)
Show an image.

A plain image can be show as a basic card without title or description. This is therefore simply a wrapper around *show_basic_card()*.

Parameters

- **url** – The images URL. Must be HTTPS.
- **accessibility_text** – The images accessibilitiy text.
- **height** – The images height.
- **width** – The images width.
- **image_display_options** – More detailes `ImageDisplayOptions`.

show_table_card(*table_card: flask_dialogflow.google_apis.actions_on_google_v2.TableCard*) → None
Show a `TableCard`.

Parameters **table_card** – The card to show.

play_media_response(*media_response: flask_dialogflow.google_apis.actions_on_google_v2.MediaResponse*) → None
Play a `MediaResponse`.

Parameters **media_response** – The media response to play.

show_carousel_browse(*carousel_browse: flask_dialogflow.google_apis.actions_on_google_v2.CarouselBrowse*) → None
Show a `CarouselBrowse`.

Parameters **carousel_browse** – The carousel to show.

show_order_update(*order_update: flask_dialogflow.google_apis.actions_on_google_v2.OrderUpdate*) → None
Show an `OrderUpdate`.

Parameters **order_update** – The order update to show.

suggest_link_out(*destination_name: str*, *url: str*, *url_type_hint: Optional[UrlTypeHint] = None*) → None
Suggest a (web or Android app) link.

Parameters

- **destination_name** – The title to show on the button.
- **url** – The URL.
- **url_type_hint** – Optional hint for the URL, to be used when it is an Android link.

ask_for_permission(*reason: str*, **permissions*)
Ask for permissions.

Parameters

- **reason** – The reason for the request.
- **permissions** – The permissions to request.

ask_for_confirmation(*request_confirmation_text: str*) → None
Ask for a confirmation.

Parameters **request_confirmation_text** – The text to confirm.

ask_for_sign_in(*reason: str*) → None
Ask for sign in to link an OAuth account.

Parameters **reason** – The reason for the request.

ask_for_datetime(*request_text: str*) → None
ASk for a datetime.

Parameters **request_text** – The request text.

ask_for_date(*request_text: str*) → None
ASk for a date.

Parameters **request_text** – The request text.

ask_for_time(*request_text: str*) → None
ASk for a time.

Parameters **request_text** – The request text.

ask_for_screen_surface(*context: str*, *notification_title: str*) → None
Ask to hand the conversation over to a screen surface.

This wraps *ask_for_new_surface()* for screen surfaces.

Parameters

- **context** – The context that will be picked up on the new surface.
- **notification_title** – The title of the notification on the new device.

ask_for_new_surface(*capabilities: MutableSequence[str], context: str, notification_title: str*)
Ask to hand the conversation over to a specific surface.

Use *ask_for_screen_surface()* if you want to hand off to a screen.

Parameters

- **capabilities** – Capabilities that the new surface must have.
- **context** – The context that will be picked up on the new surface.
- **notification_title** – The title of the notification on the new device.

ask_for_link(*open_url_action: flask_dialogflow.google_apis.actions_on_google_v2.OpenUrlAction, dialog_spec: Optional[flask_dialogflow.google_apis.actions_on_google_v2.DialogSpec] = None*) → None
Ask for a link.

Unclear what this is for.

Parameters

- **open_url_action** – The URL action to perform.
- **dialog_spec** – The dialog spec to use (unspecified).

ask_for_simple_selection(*simple_select: flask_dialogflow.google_apis.actions_on_google_v2.SimpleSelect*)
Ask for a simple selection.

Parameters **simple_select** – The selection options.

ask_for_list_selection(*list_select: flask_dialogflow.google_apis.actions_on_google_v2.ListSelect*)
Ask for a selection from a list.

Parameters **list_select** – The list with the selection options.

ask_for_carousel_selection(*carousel_select: flask_dialogflow.google_apis.actions_on_google_v2.CarouselSelect*)
Ask for a selection from a carousel.

Parameters **carousel_select** – The carousel with the selection options.

ask_for_collection_selection(*collection_select: flask_dialogflow.google_apis.actions_on_google_v2.CollectionSele*
→ None
Ask for a selection from a collection.

Parameters **collection_select** – The collection with the selection options.

ask_for_delivery_address(*reason: str*) → None
Ask for a delivery address.

Parameters **reason** – The reason for this request.

ask_for_transaction_requirements_check(*order_options: flask_dialogflow.google_apis.actions_on_google_v2.OrderOptions, payment_options: flask_dialogflow.google_apis.actions_on_google_v2.PaymentOptions*) → None
Ask for the transactions requirements check.

Parameters

- **order_options** – The order options to check.

- **payment_options** – The payment options to check.

ask_for_transaction_decision(*proposed_order: flask_dialogflow.google_apis.actions_on_google_v2.ProposedOrder*
order_options: flask_dialogflow.google_apis.actions_on_google_v2.OrderOptions,
payment_options: flask_dialogflow.google_apis.actions_on_google_v2.PaymentOptio
presentation_options: flask_dialogflow.google_apis.actions_on_google_v2.Presentati
→ None

Ask for a transaction decision.

Parameters

- **proposed_order** – The order to propose.
- **order_options** – The order options to propose.
- **payment_options** – The payment options to propose.
- **presentation_options** – The presentation options to propose.

to_webhook_response_payload() → MutableMapping[str, Any]

Render this conversation to the webhook response payload.

The response payload is not a `AppResponse`, but a custom, Dialogflow-specific format.

Returns A dict with the necessary response data.

class flask_dialogflow.integrations.actions_on_google.**UserFacade**(*user: Optional[flask_dialogflow.google_apis.ac = None, user_storage_default_factory: Callable[[], T] = <class 'dict'>, user_storage_deserializer: Optional[Callable[[str], T]] = <function loads>, user_storage_serializer: Optional[Callable[[T], str]] = <function dumps>*)

A facade to the user object.

This wraps the `User` object and adds some additional features, most notably the handling of the user storage de-/serialization. This class is what is returned by `V2ActionsOnGoogleDialogflowConversation.user`.

property user_id

The `User.user_id`.

property id_token

The `User.id_token`.

property profile

The `UserProfile`.

property access_token

The `User.access_token`.

property permissions
The list of `Permissions`.

property locale
The `User.locale`.

property last_seen
When this user was last seen as a datetime object.

property last_seen_before
The amount of time since the user was last seen as a timedelta.

property package_entitlements
The list of `PackageEntitlements` of this user.

property user_storage
The deserialized `User.user_storage`.

Deleting the user_storage resets it to the default factory. To ensure that we don't send and empty dictionary back to Google the user_storage is set to None when it evaluates to False during serialization. Before the next request it will then again be initialized with the default factory, thus keeping the type consistent.

2.6 JSON handling

Helpers for JSON de/serialization. This module, together with marshmallow powers the serialization and deserialization of the Google API objects to native, idiomatic Python classes. This system is part of the public API and can be used by users to implement custom context classes.

Note: By JSON, we always mean plain Python data structures that can be handled by `json.dumps()`/`json.loads()`, i.e. usually dictionaries. Pythons type system does unfortunately not allow recursive types, which is why we type JSON as `MutableMapping[str, Any]`.

---

class `flask_dialogflow.json.`**JSONType**
Mixin class that provides to_json and from_json methods.

Custom classes can inherit from this class and specify their marshmallow schema in the class definition. The schema must be a subclass of *JSONTypeSchema*. This class will then make sure that class and schema are linked and add to_json and from_json methods to the class. These methods completely abstract the schema and the marshmallow processing and allow it convert instances of this class to and from JSON in the simplest possible way.

Classes are defined as normal classes (optionally dataclasses), schemas as normal marshmallow schemas.

Example:

```
from marshmallow import fields

class _CustomClassSchema(JSONTypeSchema):
    foo = fields.Str()
    bar = fields.Int()

@dataclass
class CustomClass(JSONType, schema=_CustomClassSchema):
    foo: str
    bar: int
```

`CustomClass` is now linked with its schema and has *`to_json()`* and *`from_json()`* methods. They abstract the de-/serialization, the user does not have to care about marshmallow or the schema anymore:

```
>>> CustomClass.from_json({'foo': 'baz', 'bar': 42})
CustomClass(foo='baz', bar=42)
>>> CustomClass(foo='baz', bar=42).to_json()
{'foo': 'baz', 'bar': 42}
```

This works with all marshmallow features and can thus be used to quickly de-/serialize complex class hierarchies (such as `WebhookRequests`). The only caveat is that the result of `Schema.load()` will be spread into the classes init method, i.e. the params must map to each other. It is therefore recommend to use plain dataclasses as JSONTypes. However, both the to_json and from_json accept schema and dump/load kwargs, should further customization be desired.

Raises

- **`AttributeError`** – When this class was subclassed without specifying a schema and a schema could also not be found in a super class.
- **`TypeError`** – When the specified schema is not a JSONTypeSchema subclass.

`classmethod from_json`(*data: MutableMapping[str, Any]*, *schema_kwargs=None*, *load_kwargs=None*)

Instantiate this class from JSON.

Parameters

- **`data`** – The data to load.
- **`schema_kwargs`** – Kwargs to pass through to this classes schemas init method. See `marshmallow.Schema` for details.
- **`load_kwargs`** – Kwargs to pass through to this classes schemas load method. See `marshmallow.Schema.load()` for details.

`to_json`(*schema_kwargs=None*, *dump_kwargs=None*)

Dump an instance of this class to JSON.

Parameters

- **`schema_kwargs`** – Kwargs to pass through to this classes schemas init method. See `marshmallow.Schema` for details.
- **`dump_kwargs`** – Kwargs to pass through to this classes schemas dump method. See `marshmallow.Schema.dump()` for details.

class `flask_dialogflow.json.`**`JSONTypeSchema`**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Base class for schemas for JSONTypes.

This class mixes in a make_obj method that is registered as a marshmallow post_load hook. This ensures that the data will be loaded into a JSONType class instance, not just a dict. The hook is registered here but the actual object class is only accessed at runtime, after the schema has been linked with its JSONType.

This class is set to exclude unknown fields by default. This is because one of the core use cases of this class are custom context classes, and Dialogflow adds its own fields there all the time. This can of course always be overridden in subclasses.

class `flask_dialogflow.json.`**`ModuleLocalNested`**(*nested: Union[str, marshmallow.schema.Schema]*, *module_name: Optional[str] = None*, **args*, ***kwargs*)

Nested field subclass that can be parametrized with the modules FQN.

Nested marshmallow fields get the name of schema of the nested class as a string. This string can be just the schemas name itself, but it is usually more robust to give the fully qualified name with the module path. As our schemas typically reside in the same module as the object classes we would like to have the fully qualified name used automatically. This `fields.Nested` subclass makes this possible by accepting the module name as a string and then building the full name for every field where it is used.

Example:

```
# In some_api_module.py: Parametrize the nested field with the
# module name by using a partial
from functoools import partial
Nested = partial(ModuleLocalNested, module_name=__name__)

class _SomeSchema(Schema):
    ...

class _SomeOtherSchema(Schema):
    some_nested_field = Nested('_SomeSchema')

# The nested schema is now stored as some_api_module._SomeSchema
```

2.7 Templating

This library uses the same Jinja templating library as Flask, but with a custom loader to support YAML files with many individual templates (since speech responses tend to be very short). The loader also supports randomization to add greater variability to speech responses.

class flask_dialogflow.templating.**YamlLoaderWithRandomization**(*path: str*)

A simple template loader for YAML files that supports randomization.

This template loader loads all templates from a single, flat YAML file. The file is loaded once during initialization and then only reloaded when a change is detected.

It supports randomization in that if the template is an array it selects one of the arrays elements at random. This can be used to add variability to templates within the same context. The array elements can also be 2-element arrays themselves, were the second element is a number. This number will be used to weigh the random choice. Elements without weight default to 1.

Examples:

```
simple_template: Hello world!

# A list template: All variants have equal probability (i.e. 50% here)
random_template:
  - Hi there, this is the first variant!
  - And this is the second variant.

# Template with weights, default value is 1
weighted_template:
  - ['This is the first variant.', 0.5]     # ~14% prob. (0.5/3.5)
  - This is the second variant.             # ~29% prob. (1/3.5)
  - ['And this is the third variant.', 2]   # ~57% prob. (2/3.5)

# NOT allowed: Nested templates
outer_template:
  inner_template:
    - This would be the actual text (if it were allowed)!
```

Note that for randomization to work `auto_reload` has to be enabled on the Jinja environment. Otherwise the env will cache the templates internally and not call this loader. The loader itself will select a random version of each template every time it is called, see *`get_source()`* for details.

Parameters **`path`** – The path to the templates YAML file.

`get_source`(*environment: jinja2.environment.Environment*, *name: str*) → Tuple[str, str, Callable[[], bool]]
Get a template source.

Expects the template to be a key from the YAML file and looks it up in the cached mapping, reloading it beforehand if the file was modified. The uptodate callback returned as the third param always returns false to force the environment to call this function every time and thus trigger the random selection again. This in combination with the auto_reload setting on the Jinja environment is necessary to make randomization work.

Parameters

- **`environment`** – The `Environment` to load the template from.
- **`name`** – A key from the YAML templates file.

Returns The same (source, filename, uptodate) tuple as the BaseLoader.

Raises

- **`TemplateNotFound`** – When the template name is not in the file.
- **`TemplateError`** – When the template files format is invalid (usually because of misquoted strings).

`list_templates`() → Iterable[str]
List the templates of this loader.

Returns An iterable of template names.

2.8 CLI interface

A special command group for Flask's CLI interface. Adds an `agent` sub command to the `flask` command which gives access to certain information about the current Dialogflow agent. See also `flask agent --help`. The agent itself is also available in a `flask shell` as `agent`.

`flask_dialogflow.cli.`**`intents`**(**args*, ***kwargs*)
List the registered intent handlers.

Prints a table with the registered intent names and their handler functions.

`flask_dialogflow.cli.`**`contexts`**(**args*, ***kwargs*)
List the registered contexts.

Prints a table with the registered context names, their default factories and whether they should be kept around.

`flask_dialogflow.cli.`**`integrations`**(**args*, ***kwargs*)
List the registered integration conversation classes.

Prints a table with the registered integrations (source and version), the corresponding conversation class and its init kwargs.

2.9 Test helper

A few tools to make testing Dialogflow agents easier. The recommended way to test Agents built with this library is the *DialogflowAgent.test_request()* method, which simulates an end-to-end request through the agent. See also Testing Flask Applications for more tips.

flask_dialogflow.agent.**build_webhook_request**(*intent: Optional[str] = 'Default Welcome Intent', action: Optional[str] = None, source: Optional[str] = None, session: Optional[str] = 'projects/foo/agent/sessions/bar', parameters: Optional[Dict[str, Any]] = None, contexts: Optional[Iterable[flask_dialogflow.context.Context]] = None, payload: Optional[Dict[str, Any]] = None, is_fallback: Optional[bool] = False, dialogflow_version: Optional[str] = 'v2beta1'*) → Union[flask_dialogflow.google_apis.dialogflow_v2.WebhookReque flask_dialogflow.google_apis.dialogflow_v2beta1.WebhookRequest]

Factory function to build a `WebhookRequest`.

Params not explicitly given are set to sensible defaults, allowing for request construction with minimal effort. This functions will rarely be used explicitly, but powers other test helpers under the hood, especially *DialogflowAgent.test_request()*, which accepts the same kwargs as this function.

Examples:

```
# This builds a valid request to the FooIntent
build_webhook_request('FooIntent')

# A slighly more complex request with params and context
from flask_dialogflow.google_apis.dialogflow_v2 import Context

build_webhook_request(
    intent='FooIntent',
    parameters={'some-date': '2018-10-02T19:30:26Z'},
    contexts=[
        Context('foo_context', parameters={'foo': 'bar'})
    ]
)
```

Parameters

- **intent** – The requests intents display name.
- **action** – The requests action.
- **source** – The source from where this request was send to Dialogflow.
- **session** – The requests session. Must conform to the session str format.
- **parameters** – The dict of params parsed from the input text.
- **contexts** – An iterable of *Context*. Defaults to an empty list when not given.
- **payload** – The platform-specific request payload.
- **is_fallback** – Whether this intent is a fallback intent.

- **dialogflow_version** – The Dialogflow version to use. Defaults to v2beta1, which has all features.

class flask_dialogflow.agent.**TestWebhookResponse**(*followup_event_input: Optional[EventInput] = None*, *output_contexts: List[Context] = <factory>*, *fulfillment_text: Optional[str] = None*, *fulfillment_messages: List[Message] = <factory>*, *payload: Dict[str, Any] = <factory>*, *source: Optional[str] = None*, *end_interaction: Optional[bool] = None*)

Response class returned from *DialogflowAgent.test_request()*.

This is a subclass of WebhookRequest with a few extra methods that help in making assertions against the response.

classmethod from_webhook_response(*webhook_response: Union[flask_dialogflow.google_apis.dialogflow_v2.Webho flask_dialogflow.google_apis.dialogflow_v2beta1.WebhookResponse]*)

Classmethod to instantiate this from a normal WebhookResponse.

Used internally, should not be used by users.

Parameters **webhook_response** – The normal WebhookResponse that should be converted to this class.

text_responses() → Iterable[str]

Get an iterable of all individual text responses.

Note that this yields only the generic Dialogflow responses, i.e. responses set via conv.ask, not conv.google.ask.

Yields The individual text responses.

has_context(*display_name: str*) → bool

Check whether the response has a certain context set.

This does not check the lifespan of the context because None is a valid lifespan that defaults to Dialogflows default lifespan.

Parameters **display_name** – The display name to check for (i.e. the context name without the session id).

context(*display_name: str*) → flask_dialogflow.context.Context

Get a context by its display name.

Returns the *Context* object when it is part of this response. Throws a ValueError when it is not.

Parameters **display_name** – The display name to get (i.e. the context name without the session id).

Raises **ValueError** – When the context is not part of the response.

CHAPTER

THREE

CHANGELOG

3.1 Version 0.9.0

- Original implementation as developed by ONSEI internally

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

M

P

Q

R

S

T

U

V

W

Y