



Taking Your Chatbot to Production



Welcome to the module Taking Your Chatbot to Production.

Agenda

Operationalizing Your Agent

Deploying a WebHook for Fulfillment

Building a Custom Chatbot User Interface

Securing the Webhook

Integrations



So far, we looked at how to use the **building blocks** of Dialogflow to build your agent, and we also tested it out using the web demo. We are now going to look at how you can **operationalize your agent**...in other words what you need to do to take your chatbot into production.

Operationalizing your agent

Automation

Leverage existing sources like playbook, FAQs, etc. to build your agent.

Backend services

Connect to backend systems and services.

Branding

Customize your frontend to include branding, logos, etc.

Security

Secure your webhook to allow authenticated calls only.



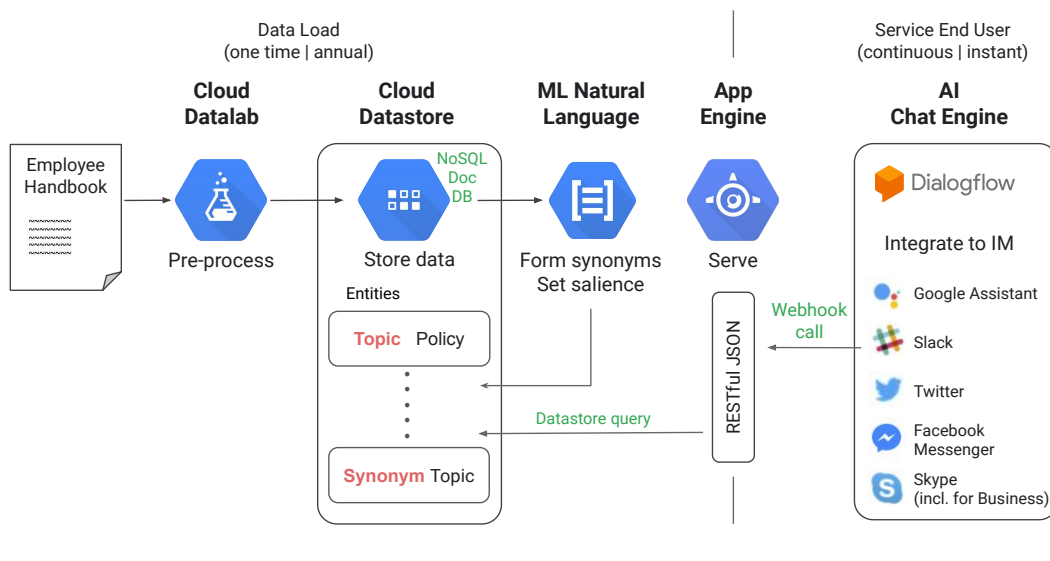
For the pizza example, we built our agent step by step starting with creating intents, followed by entities and then contexts, all done within the Dialogflow UI. Sometimes, you may already have existing sources that you can extract entities from, for instance a call center playbook or a faq document. So we'll look at ways to leverage this data when building our agent.

Sometimes, all you need is a conversation interface to surface answers from the existing backend systems and capabilities. and so we'll look at deploying your backend code as a webhook on appengine.

Next, we will be looking at how to customize your agent's UI so that you can have your own branding. This is where we deploy a custom frontend on AppEngine.

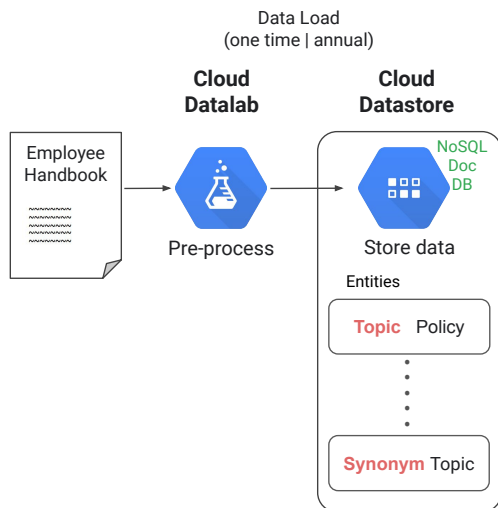
We also want to secure our webhook so it only makes authenticated calls to the backend services.

HR chatbot architecture



To that end, we will use a different example and create a brand new HR chatbot that can answer human resources-related questions. In the architecture, you can see how various Google Cloud products come together to productionize your conversational agent. For example, AppEngine will host the frontend UI and the webhook code, and Cloud Datastore serves as the backend, where the knowledge base is stored, and provides answers to user queries by responding to the webhook calls. So let's go through this architecture step by step....starting with building the knowledge base in Cloud Datastore.

Leveraging existing sources to derive entities



And this is one big difference with the pizza ordering example....where the backend was meant to store pizza orders....so you are mainly writing into the database. While in the HR example, the backend is a knowledge base, where a query will lookup for definitions and sent it back to userso you're mainly reading from the database. So, how do we build this knowledge base in the first place? well we have the employee handbook...which is an HR manual document with topic keywords and definitions. So all we need to do is leverage that to build our knowledge base. This kind of exercise is typically a one time activity. In the hands-on labs, which I will demonstrate, you will use Cloud Datalab notebooks to quickly run Python scripts to extract topics from the sample HR Manual, then push them into a Datastore entity using the Cloud Datastore API. And now is a good time to talk a little about Cloud datastore.

Cloud Datastore is a good place to store and retrieve your chatbot's data with minimal fuss!



Cloud Datastore is a serverless global document database.

Database as a service

Features

Planet-scale

7 years+

15 trillion requests/month

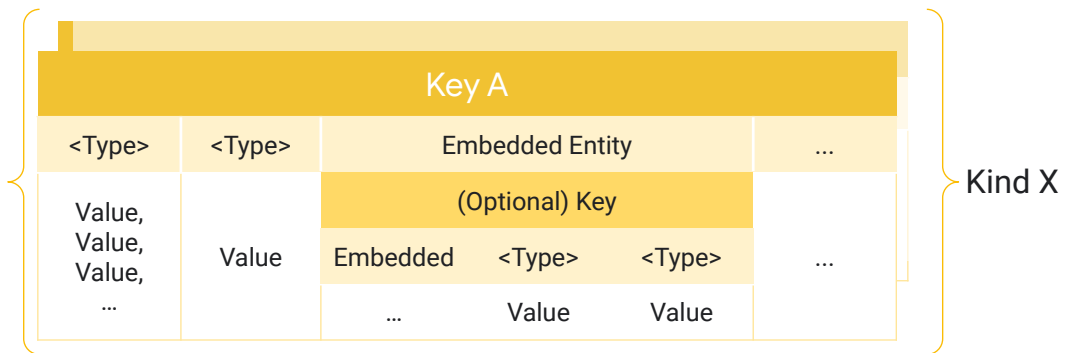
Fully managed service



Datastore is Google Cloud's highly scalable NoSQL database for web and mobile applications. It pairs very well with App Engine and serves as a great backend for App Engine applications, scaling seamlessly and automatically with your data and traffic.

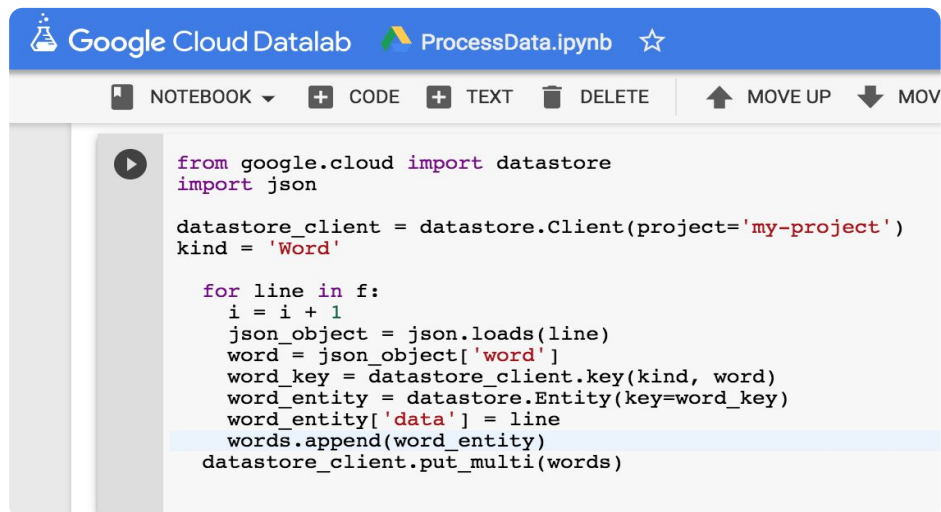
In the hands-on lab, after you store the keywords and definitions into Datastore, we use that as the data source that populates your Dialogflow agent's entities.

Cloud Datastore data model



Let's take a deeper look at Datastore's data model. Since Datastore is a non-relational database, it doesn't fit the relational database constructs of objects and relationships. Instead, you can think of paths and entities, similar to a filesystem. Entities in datastore have a particular Kind. In the case of this lab, the Kind is "Topic" - the keywords that the HR Manual contains. Queries in Datastore are eventually consistent, meaning that there can be a delay from when data is added or modified and when it is returned in a query. To enable strong consistency in Datastore, more similarly to a relational database, you can organize your data with Keys. This will create an entity group of that key, in which queries are strongly consistent.

Deploying to Cloud Datastore



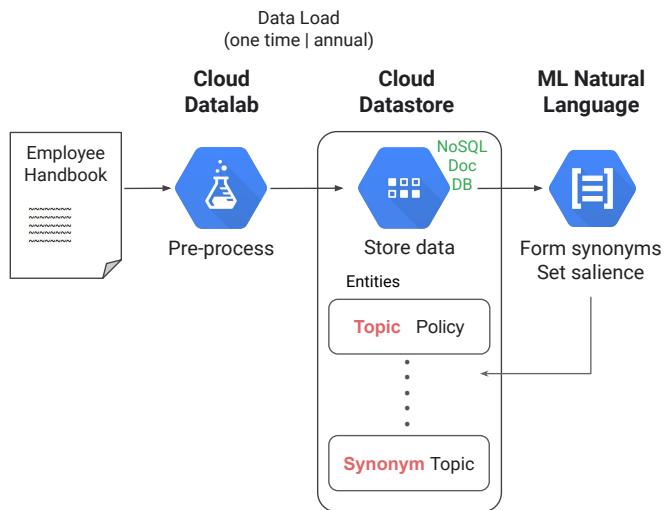
```
from google.cloud import datastore
import json

datastore_client = datastore.Client(project='my-project')
kind = 'Word'

for line in f:
    i = i + 1
    json_object = json.loads(line)
    word = json_object['word']
    word_key = datastore_client.key(kind, word)
    word_entity = datastore.Entity(key=word_key)
    word_entity['data'] = line
    words.append(word_entity)
datastore_client.put_multi(words)
```

Here is a code snippet from one of the Cloud Datalab notebooks which shows how the code can extract the keywords from the HR Manual and push them to Datastore in your GCP project. In this snippet, the code iterates over each line in the HR Manual and populates the 'Word' entity in Datastore with each keyword entity that it finds in the document.

Adding ML to augment your agent



Next we use Google's Natural Language API to form synonyms of these topics and add them to Datastore.

The diagram illustrates the data pipeline for building a chatbot. It starts with a document icon labeled "Employee Handbook" under the heading "Data Load (one time | annual)". An arrow points to a flask icon labeled "Pre-process" under "Cloud Datalab". Another arrow points to a database icon labeled "Store data" under "Cloud Datastore". The "Cloud Datastore" is also labeled "NoSQL Doc DB" and contains a list of "Entities" including "Topic Policy" and "Synonym Topic". An arrow points from the "Store data" icon to a document icon labeled "Form synonyms Set salience" under "ML Natural Language". Finally, an arrow points from the "Form synonyms Set salience" icon to the "Dialogflow" logo.

Finally, we call the Dialogflow API to then populate your Dialogflow agent's entity list with these topics from the Datastore entity. The demo that follows will walk you through the steps covered so far.

Demo

Operationalizing Your Agent

In this demo, we'll use Cloud Datalab notebooks to quickly run Python scripts to extract topics from the sample HR Manual, then push them into a Datastore entity using the Cloud Datastore API. We also leverage the Natural Language API to generate synonyms for the HR topics. Finally, we use the Dialogflow API to populate entities into Dialogflow.



Demo will cover the first 3 python notebooks: processhandbook, processsynonyms, dialogflow

In this demo, we'll use Cloud Datalab notebooks to quickly run Python scripts to extract topics from the sample HR Manual, then push them into a Datastore entity using the Cloud Datastore API. We also leverage Google Natural Language API to generate synonyms for the HR topics. Finally, we use the Dialogflow API to populate entities into Dialogflow.

- Enable Dialogflow API
- initialize App Engine and Datastore (paired together)
- Set up Datalab:
 - set Compute zone
 - create datalab instance [takes a while]
 - hit Y for SSH key if needed, hit enter twice for passphrase (no passphrase) for RSA Key
 - Web Preview > Port 8081 to see Datalab notebooks
 - Create new untitled notebook just to run code to copy from Cloud Storage bucket
 - View HR Manual sample txt file
 - Run ProcessHandbook notebook (if any errors, reset session and re-run)
 - should see the extracted text.... check Datastore and should see the entity 'Topic' with the entries listed
 - ProcessSynonyms notebook - do the same thing [Note - may have to

- refresh browser to see Synonyms entity show up in Datastore]
- Create Dialogflow agent at Dialogflow.com
 - Sign In with Google > choose Qwiklabs account
 - Create Agent - import existing project - Qwiklabs project
 - create Topic entity - uncheck 'define synonyms', check 'allow automated expansion'
- Back to Datalab - run Dialogflow notebook
 - This will push the Topic entities from Datastore into Dialogflow entity list in the Dialogflow console
- In Dialogflow console, create Intent called Topic
 - create Lookup action
 - Create training phrases
 - Save - dialogflow will train your model

Agenda

Operationalizing Your Agent

Deploying a WebHook for Fulfillment

Building a Custom Chatbot User Interface

Securing the Webhook

Integrations



In the pizza example, we used cloud functions to do fulfillment and store pizza orders into a database.

The other option is to create a webhook.

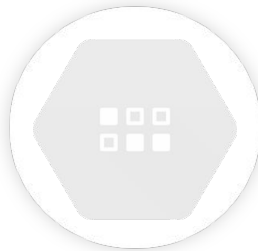
A webhook is a web server endpoint that you create and host which can be a backend for your Dialogflow agent. While you could create a working agent completely through the Dialogflow console, writing some code in a webhook will give you additional options and customization capabilities.

For the HR chatbot example, we will use a webhook that is written in Python.

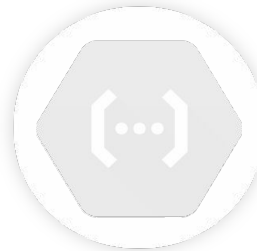
Understand the serverless options for your webhook



App Engine



Cloud
Datastore



Cloud
Functions



For production, you will want your webhook to always be on and ideally able to scale up if your chatbot is popular. You also want your webhook to be available via HTTPS (to protect your data and credentials in transit). So let's take a look at some serverless, scalable options.

App Engine is a great place to deploy your webhook code....which provides a scalable and secure “always-on” presence for our webhook.

App Engine is a microservices platform for multiple programming languages



Event-oriented
architectures/Webhook

Platform as a service

Triggers

HTTP

Language support



App Engine is Google's fully managed, serverless, platform as a service product that allows you to build highly scalable applications. The underlying infrastructure is taken care of for you. With support for multiple programming languages, App Engine enables developers to focus on developing their applications without needing to worry about infrastructure concerns.

Code using popular languages, frameworks, and tools

Popular languages

Python
Java
PHP
Go
Node.js
C#
.NET

Popular frameworks

Django
Flask
Spring
webapp2
web2py

Popular tools

Eclipse
IntelliJ
PyCharm
Jenkins



One great feature of App Engine is that you can quickly build and deploy applications using popular languages, including Java, PHP, Node.js, Python, C#, .NET, Ruby and Go, and frameworks, including Django, Flask and Spring. You can also bring your own language runtime and frameworks in a custom environment. App Engine provides many services and APIs, such as memcache, Logging API, Users API for authentication, asynchronous task queues, and more. You can easily connect your application to Cloud Datastore, our NoSQL database, Cloud SQL, or Cloud storage for object storage such as images and even code files. App Engine also handles scaling and load balancing, allowing you to autoscale and load balance between different instances of your application. There is also easy app versioning, which can enable A/B testing and traffic splitting.

Deploying to App Engine

In production

`gcloud app deploy`

In development

`dev_appserver.py app.yaml` (Python)

`dev_appserver.sh` (Java)

... 0

```
application: guest-book-123
version: 1
runtime: python27
api_version: 1

handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /.*
  script: main2.app

libraries:
- name: webapp2
  version: "2.5.2"
- name: jinja2
  version: "latest"
```

app.yaml



You can deploy your webhook to App Engine by running the “gcloud app deploy” command. App Engine will find your app.yaml file for your service, read any additional parameters you set such as version number, runtime, and service name, and deploy your application.

To run your app locally before pushing it to production, you can run the dev_appserver.py command to test your Python app from Cloud Shell.

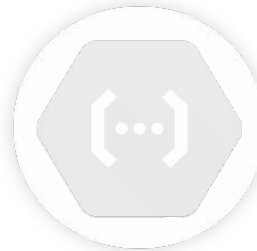
Understand the serverless options for your webhook



App Engine



Cloud
Datastore



Cloud
Functions



Besides code, your webhook will typically need to interact with a database, to store data into or read data from. Cloud Datastore is a natural database choice when using App Engine. Do take note though that Datastore does not store the webhook code, but it can store the data that the code is supposed to retrieve or persist. Both App Engine and Cloud Datastore, which you will use later in the hands-on labs, are serverless.

Understand the serverless options for your webhook



App Engine



Cloud
Datastore

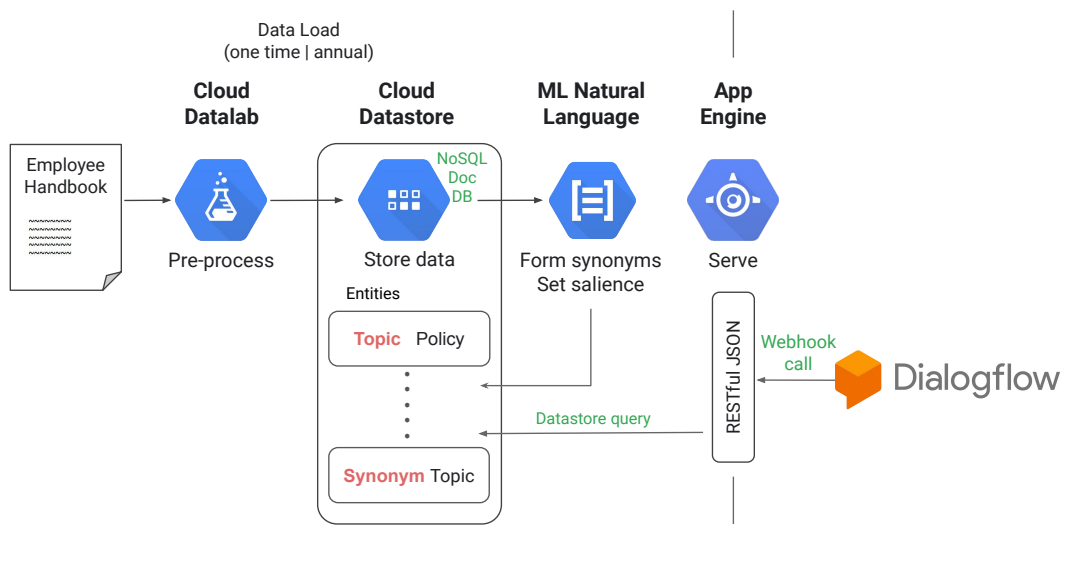


Cloud
Functions



Another option to host your Dialogflow webhook code is Cloud Functions, another serverless offering from Google Cloud. In an earlier module, we talked about how you can create your fulfillment flow by writing code in cloud functions within Dialogflow console using the inline editor.

Deploying a webhook on App Engine



In our HR chatbot example, the webhook is called by the DialogFlow agent's fulfillment component to get answers from Datastore once a topic entity is matched. The webhook code uses information from the HTTP POST request sent by your agent to look up the keyword and match it with the Topic entity in Cloud Datastore. The webhook will then return a response back to the user which contains the definition of the keyword passed in the request. Let's take a quick look at the webhook code.

Use NDB to query Cloud Datastore synonyms

```
def getSynonym(query_text):  
    synonym_key = ndb.Key('Synonym', query_text)  
    synonyms = Synonym.query_synonym(synonym_key).fetch(1)  
  
    synonym_text = ""  
    for synonym in synonyms:  
        synonym_text = synonym.synonym  
        break  
  
    return synonym_text
```



The getSynonym method uses Google's NDB Datastore library to allow your App Engine Python application to connect to Cloud Datastore. This will generate synonyms for the topic keywords in the HR manual and store them as a new Synonym entity in Datastore.

Use NDB to look up Topic Key in Cloud Datastore

```
def getActionText(synonym_text):
    synonym_text = synonym_text.strip()
    topic_key = ndb.Key('Topic', synonym_text)
    topics = Topic.query_topic(topic_key).fetch(1)

    action_text = ""
    for topic in topics:
        action_text = topic.action_text

    if action_text == None or action_text == "":
        return ""

    return action_text
```



This method `getActionText` also uses the NDB library to query your Datastore entities for the topic parameter in order to look up the definition of the keyword.

Map data returned from Datastore into Python classes

```
class Topic(ndb.Model):
    action_text = ndb.StringProperty()

    @classmethod
    def query_topic(cls, ancestor_key):
        return cls.query(ancestor=ancestor_key)

class Synonym(ndb.Model):
    synonym = ndb.StringProperty()

    @classmethod
    def query_synonym(cls, ancestor_key):
        return cls.query(ancestor=ancestor_key)
```



This part of the code shows how the action text stored in Datastore, whether it is the Topic or the Synonym, is mapped into Python classes for Topic and Synonym, respectively. You query the ancestor key to maintain the entity's root parent in the Datastore hierarchy.

app.yaml for backend service

```
runtime: python27
api_version: 1
threadsafe: true
service: dialogflow

# [START handlers]
handlers:
- url: /static
  static_dir: static
- url: /.*
  script: main.app
# [END handlers]
```



Next, you will deploy a new service for the webhook backend in App Engine.

We will leave the default service to point to the custom UI that we will build in the next section.

After you deploy, App Engine creates a new separate service called dialogflow that points to your webhook backend.

Agenda

Operationalizing Your Agent

Deploying a WebHook for Fulfillment

Building a Custom Chatbot User Interface

Securing the Webhook

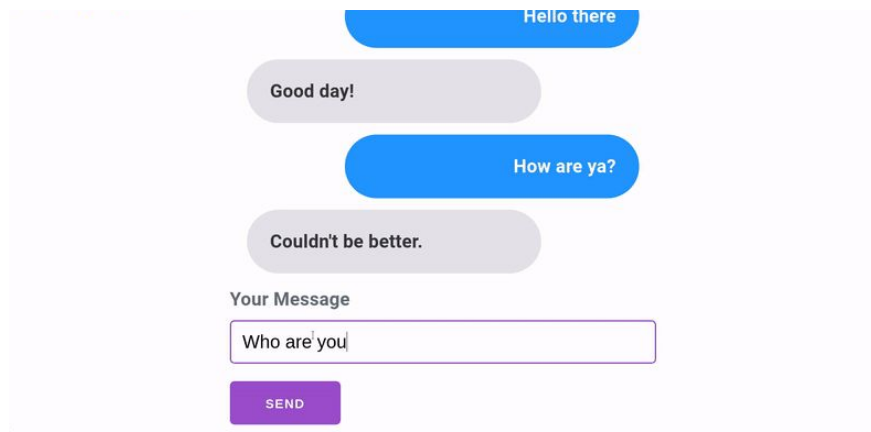
Integrations



The next theme we will look at is customization, including building a custom frontend.

Later in the lab, you will deploy a custom UI for your chatbot on App Engine, which allows you to wrap your agent in a customizable user interface. You could also tweak the UI to incorporate your own branding or to integrate your chatbot into an existing website.

Building a custom chatbot user interface



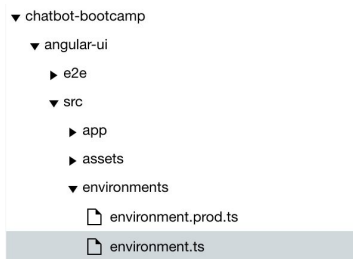
<https://github.com/AngularFirestore/59-angular-chatbot-dialogflow>



Since its a custom UI....you could write one of your own...or there are plenty of examples online. The one we chose for the purpose of the HR chatbot uses angular code and is written by Jeff Delaney and licensed under the MIT license.

<https://github.com/AngularFirestore/59-angular-chatbot-dialogflow>

Building a custom chatbot user interface



```
export class ApiAiClient {
  constructor(options) {
    if (!options || !options.accessToken) {
      throw new ApiAiClientConfigurationError("Access token is required for new ApiAi.Client instance");
    }
    this.accessToken = options.accessToken;
    this.apiLang = options.lang || ApiAiConstants.DEFAULT_CLIENT_LANG;
    this.apiVersion = options.version || ApiAiConstants.DEFAULT_API_VERSION;
    this.apiBaseUrl = options.baseUrl || ApiAiConstants.DEFAULT_BASE_URL;
    this.sessionId = options.sessionId || this.guid();
  }
}
```

Dialogflow Javascript Client - ApiAiClient class



The Angular code for the custom UI works by using your agent's client access token to populate the UI with your agent's data. You will set your client access token in the environment file, which is then read by the UI code. This code snippet shows the definition of the ApiAiClient object which takes in your agent's access token as a parameter.

Include attribution for Angular code:

<https://github.com/AngularFirestore/59-angular-chatbot-dialogflow>

The ChatService class: chat.service.ts

```
@Injectable()
export class ChatService {

  readonly token = environment.dialogflow.angularBot;
  readonly client = new ApiAiClient({ accessToken: this.token });
```



The ChatService class reads the token that you set in the environment file and passes it in as a parameter to the ApiAiClient class. Dialogflow was formerly named API.AI. In Angular, you can define reusable tasks in a service, which can be injected into any other component. This chat service will handle populating our UI with our chatbot.

app.yaml to deploy custom UI as default service

```
runtime: python27
api_version: 1
threadsafe: true

skip_files:
- ^(!dist) # Skip any files not in the dist folder

handlers:
# Routing for bundles to serve directly
- url: /((?:inline|main|polyfills|styles|vendor)\.[a-z0-9]+\.(bundle|.js))
  secure: always
  redirect_http_response_code: 301
  static_files: dist/\1
  upload: dist/.*
```



Here is the app.yaml file that you will use to deploy the Angular UI as the default service in App Engine. The runtime is Python 2.7 since it will connect with your Python backend webhook service. It handles all of the Angular files that are generated when you run the 'ng build' command to package them into the 'dist' folder and ensure that the Angular styling is applied.

Demo

Deploying Webhook and Frontend on App Engine

In this demo, we'll deploy a custom UI and the webhook code on App Engine. At the end, we will be able to test our agent.



Demo: deploy webhook and frontend on AppEngine

In this demo, we will deploy a custom UI and the webhook code on AppEngine. And at the end, we will be able to test our agent.

For the custom UI, we will use the **angular code that was** written by Jeff Delaney and licensed under the MIT license.

- Clone the AngularFire repo
 - create an environments subdirectory with environment file and environment.prod file; add client access token
 - Run npm install; ng build
 - Create app.yaml file
- Deploy to App Engine as default service

Lab

Lab 2,Part 1, 2



Agenda

Operationalizing Your Agent

Deploying a WebHook for Fulfillment

Building a Custom Chatbot User Interface

Securing the Webhook

Integrations



The webhook code deployed on App Engine works well and your agent can call it to do a lookup. But so can everyone else if they had the url. So in this section, you will learn how to secure your webhook using HTTP basic authentication to lock it down with a username and password, and where to add these credentials inside your agent.

Require basic HTTP authentication to access

```
def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username,
auth.password):
            return authenticate()
        return f(*args, **kwargs)
    return decorated
```



You will add HTTP basic authentication to your webhook to prevent unauthorized access. This ensures that unauthorized callers cannot call your webhook.

In the webhook code, you will implement Flask's security framework for HTTP Basic Authentication:

The **requires_auth()** method requires authentication to access the app.

Validate submitted username and password

```
def check_auth(username, password):  
    """This function is called to check if a username / password  
    combination is valid.  
    """  
    uname="myuser"  
    pwd="mypassword"  
  
    return username == uname and password == pwd
```



The **check_auth()** method validates the submitted username and password. Note the values for username and password (you will use these later).

Handling authentication failure

```
def authenticate():
    """Sends a 401 response that enables basic auth"""
    logging.info("inside authenticate")
    return Response(
        'Could not verify your access level for that URL.\n'
        'You have to login with proper credentials', 401,
        {'WWW-Authenticate': 'Basic realm="Login Required"'})

@app.route('/webhook/', methods=['POST'])
@requires_auth
#def handle():
```



The **authenticate()** method sends a 401 response if authentication fails. You will also add the `@requires_auth` decorator to the `handle` method to ensure that the method validates the authorization credentials.

Demo

Adding Basic Authentication and Configuring Credentials

In this demo, we'll add basic authentication to our webhook and configure the credentials in the Dialogflow console.



Lets walk through where to add the code snippets that were just discussed. I will also show you where in the Dialogflow UI to add the username and password credentials so the agent knows to use these when calling the webhook.

In this demo, we are going to add basic authentication to our webhook and configure the credentials in the dialgflow console.

- Secure the webhook - add HTTP basic auth to the webhook notebook
 - run the notebook - runs locally
- Port the webhook to App Engine - copy the code from Cloud Storage bucket
 - look at main.py
 - run pip install
 - review app.yaml file which deploys new dialogflow service
- go back to look at appspot url - should have working chatbot with secured backend

Lab

Lab 2,Part 3



Agenda

Operationalizing Your Agent

Deploying a WebHook for Fulfillment

Building a Custom Chatbot User Interface

Securing the Webhook

Integrations



Besides having a customized frontend, Dialogflow agents can be enabled in multiple channels and surfaces, from text to voice to phone.

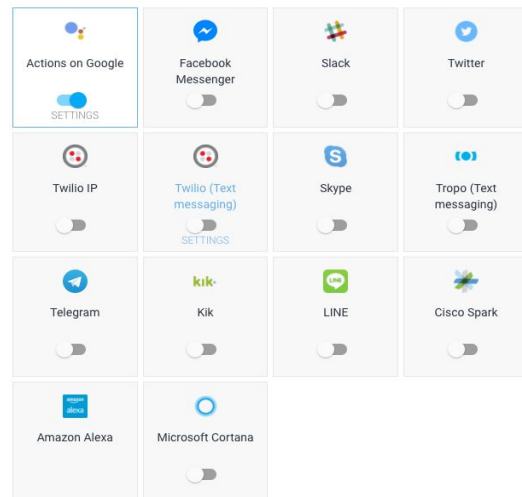
Dialogflow agents can be enabled in multiple channels and surfaces

Actions on Google

Google Home, Pixel, and more to come

External integrations

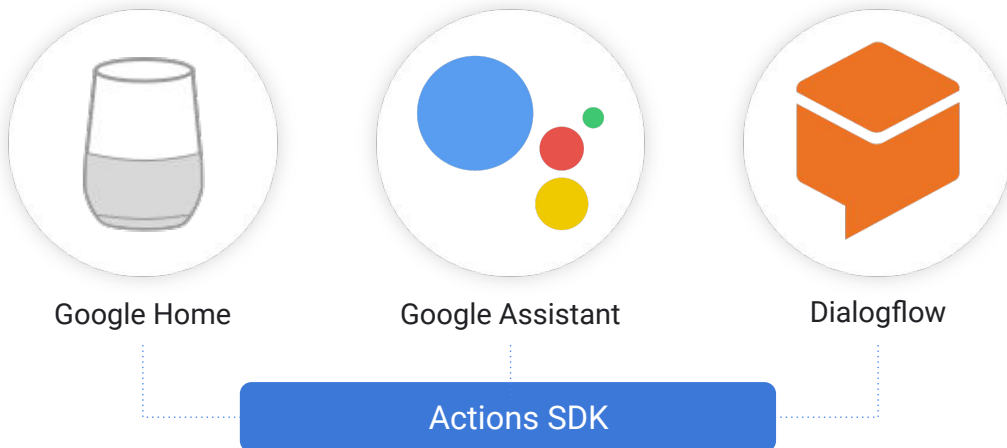
Slack, Facebook Messenger, Twitter, Twilio, Skype, Tropo, Telegram, Kik, LINE, Cisco Spark, Alexa, Cortana



Besides having a customized frontend, Dialogflow agents can be enabled in multiple channels and surfaces, from text to voice to phone.

Beyond the web chat core feature, you can use the built-in feature for deploying your agent onto any Google Assistant-enabled device, such as the Assistant mobile app or a Google Home device. You can also easily integrate your Dialogflow agent into third-party apps such as Facebook Messenger, Twitter and Slack. There are also import and export capabilities for easily importing your Dialogflow agent into Amazon Alexa or Microsoft Cortana-compatible files.

Integrate seamlessly between surfaces with the Actions SDK



While Dialogflow gives you several surface options, from the Dialogflow console web interface to Google Assistant to Google Home devices, these all rely on the Actions SDK for creating the actions that from your agent. This allows you to seamlessly move between surfaces for the same agent, no matter which platform or functionality you need.

Demo

Building Voice Chat with Actions on Google

In this demo, we'll integrate the Dialogflow agent into an Actions on Google project to enable voice chat functionality on any Google Assistant device.



Let's look at how you can easily integrate your Dialogflow agent into an Actions on Google project to enable voice chat functionality on any Google Assistant device, such as a Google Home. We will use Dialogflow's built-in integration with Google Assistant to make this happen.

- Dialogflow menu - Intents
 - Default Welcome Intent
 - Add Google Assistant Welcome under Events
- Click Integrations > Google Assistant > Integration settings
 - Explicit invocation = Default Welcome Intent
 - Implicit Invocation = Topic
 - Click Test
- Invocation > Display Name, i.e. HR Manual
- Test the assistant

<https://pixabay.com/en/sound-wave-waveform-aural-audio-1781569/>

Demo

Integrating with Slack

In this demo, we'll integrate the Dialogflow agent within the Slack org to create a Dialogflow-powered Slackbot.



Next, let's see how you can integrate your Dialogflow agent within your Slack org to create a Dialogflow-powered Slackbot. You will need a Slack account and a Slack team, and you will use your app credentials from Slack to connect your Dialogflow agent to your Slackbot.

- Go to Slack Developer Console
- Create an App - name the app, select the Team
- Add a Bot User
- Basic Information > look at App Credentials
- Go to Dialogflow > Integrations > Slack - enter those App Credentials
- Copy OAuth URL from Dialogflow Slack page
- Slack Developer Console - click on OAuth & Permissions, add new redirect URL: paste the OAuth URL
- Copy Event Request URL from Dialogflow's Slack page
- Slack Developer Console - click Event Subscriptions
- Enable Events: On, paste Event Subscription URL into Request URL - it will then be verified
- Event Subscriptions > enable > Subscribe to Bot Events > Add Bot User Event
 - Examples: message.im, message.groups, message.channels - this will bring in the bot to any of these events
- Add Slackbot to a Team:
 - Manage Distribution > Share your app with your team > Add to Slack > Authorize
- Test in Slack app

Summary

- 1 Set up a knowledge base on Cloud Datastore
- 2 Automate parts of your agent building process
- 3 Build a custom UI and set up a webhook on App Engine
- 4 Secure the webhook



In this module, we talked about taking your chatbot to production and how GCP products like App Engine, Datastore, Natural Language API, and Cloud Functions can make that happen.

For the new HR chatbot that we created, we set up a knowledge base on Cloud Datastore by extracting entities from the HR manual and used the Natural Language API to generate synonyms, which allowed us to automate parts of your agent building process.

We then deployed a custom UI (frontend) and a python webhook (backend) App Engine.

Finally, we added http basic authentication to secure your backend webhook.

cloud.google.com

