**Hungry Hippos: Restaurant Bot Documentation**

**Index**

**Abstract**

**Introduction**

Motivation

Objective

Background Research

**Design and Architecture**

**Requirements**

Functional Requirements

Usability Requirements

Non-Functional Requirements

Availability Requirements

**Architecture Overview**

System Design

Use Case Diagram

**Implementation Details**

-Initial Setup

-GCP Setup: Firebase, Cloud Functions

-Dialogflow Setup

Agent Creation

Intents Creation

Entity Creation

Training and Publishing the Model

-Components of Hungry Hippos

Search for Business from Yelp directory

Search for updates on Events happening

Reserving a table in a Restaurant

Ordering a Pizza

Asking details about configuration of Pizza

Item confirmation

Modification of Pizza order

Contents in Basket

Structure of Basket

Modifying the last item

Deleting the basket

Helping the User

Smalltalk

- Service Code Setup

- Deployment and Release

**Integrations**

* Text Based

Slack

* Telephony Based

Dialogflow Telephony Gateway

Voximplant

* Google Assistant

**Testing/Evaluation**

Webhook Testing

Client Testing

Validation - Validating Requests through Dialogflow console

**Project Planning**

Project Milestones and Deliverables

Development Iteration Breakdown

**Limitations**

**Exporting & packaging agent**

**Pricing of Dialogflow Services**

**Conclusion**

**Future Advancements**

Data Augmentation

**References**

**Appendix**

---------------------------------------------------------

**Installation steps**

Install Dialogflow Agent

Install Fulfilment Functions

Connect to GCP console/Firebase Database

**Sample Conversation with Agent**

**Architecture of Dialogflow Agent**

Components of Google’s Dialogflow

Intent Detection in Dialogflow

**Abstract**

**Introduction**

Conversational bots have many applications throughout the industry today as businesses are looking to take advantage of deploying such services on messaging platforms, voice assistant devices like Google Home, Alexa or customer service bots with the goal of improving user interaction, reducing time or freeing up time taken to service simple tasks.

Our goal was to explore building a task oriented chatbot into an initial minimum viable product that can aid in providing information about business (restaurants, grocery stores, retail stores, pharmacies, etc) from the Yelp directory, such as finding business location, operating hours or notify the user about any nearby events as well as automate some of the tasks that go into a reservation system or placing an order. We believe collecting feedback on this initial version and future development cycles would lead to a fully fleshed out agent that could be used by Yelp or Opentable for providing alternate ways to engage with the customer in lieu of the traditional website search.

Some key aspect that we were interested in focusing and learning about were:

1. Understand how all the Google Cloud Platform and Dialogflow's components work together during the building of such a service
2. How to make the user interaction fast and easy to use?
3. Understand the data needs for building such a service
4. Integrating with platforms and devices like Slack, Google Home

**Design and Architecture**

Certain requirements were considered while designing the agent architecture.

Some of the requirements that we kept in mind are mentioned below:

**1. Functional:**

The Dialogflow agent must respond when a conversation is triggered.

The agent must work with text based clients like Slack.

The agent must work with voice enabled devices like Google Home and other Google Assistant enabled devices.

**2. Usability :**

The chatbot is able to identify user requests to one of the intents defined within Dialogflow.

If a matching intent is not found the agent should fail gracefully by letting the user know that it could not process their request.

The architecture should be able to process multiple user requests being initiated from various end clients & devices processed simultaneously. This is possible due to the use of GCP Cloud Functions, where each user request can be processed in parallel and the service scales up to the number of requests being generated.

**3. Non-functional:**

Performance/ Response time requirement - The chatbot agent should reply to the user in a timely manner to keep the conversation flowing without interruption.

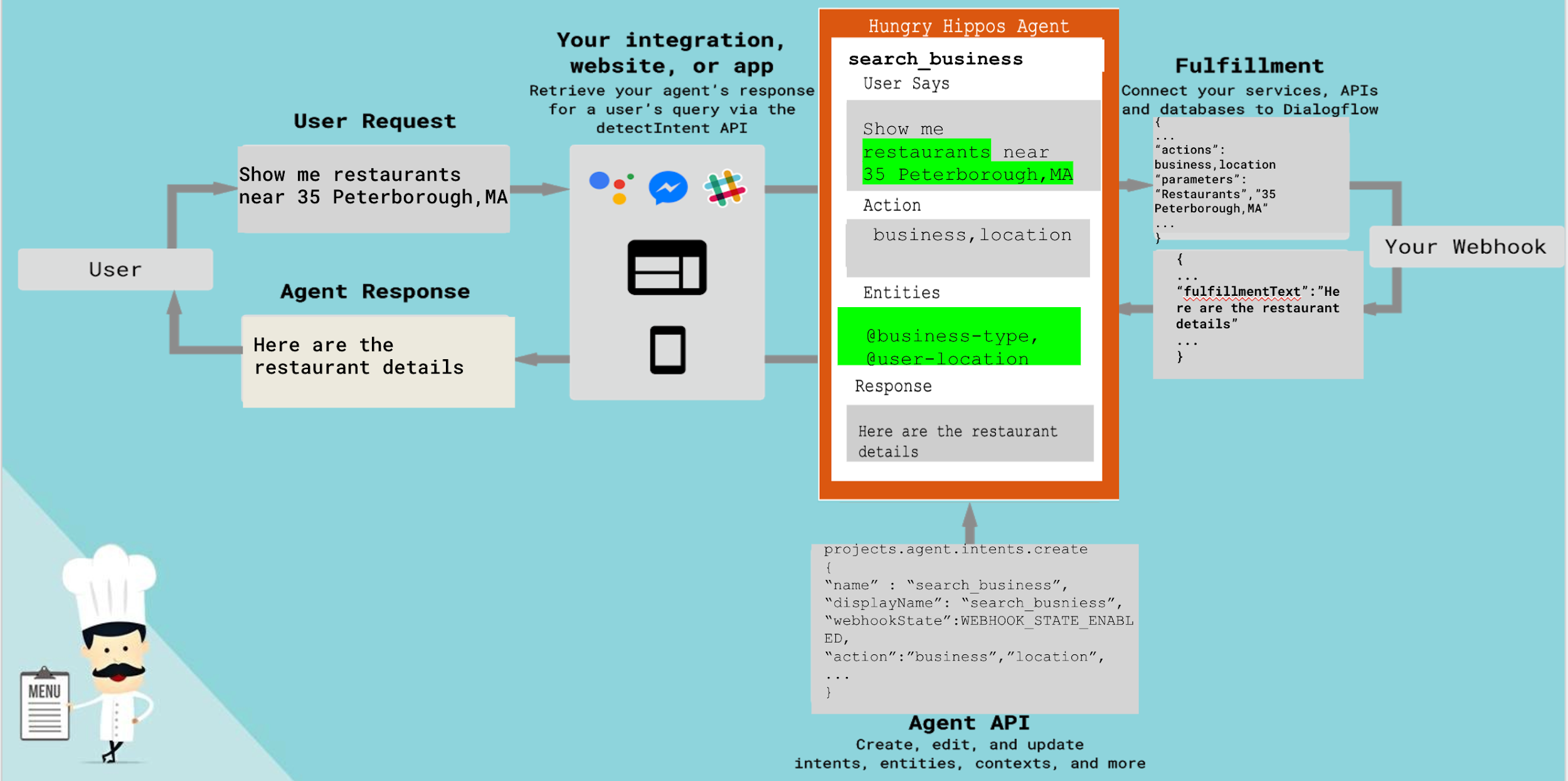
Reliability requirement- The chatbot agent must behave consistently, i.e. trigger the same intents and provide the same responses to user query each time.

**4. Availability:**

The agent must be available to chat at all times.

This is achieved by deploying the code on Google Cloud Platform services. This has a couple of benefits, any updates to the code will initiate a new deployment on the GCP server using serverless functions. This does not impact the end clients using the agent.

**Components of Dialogflow Agent**

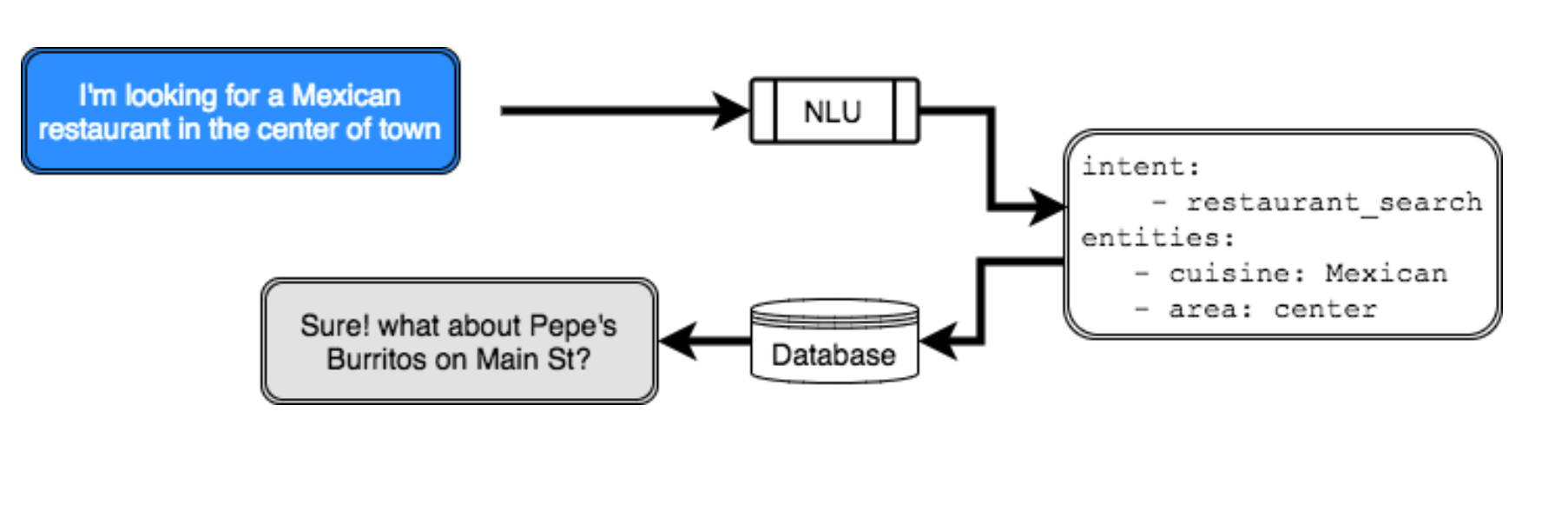
****

**Fig A: Architecture of Dialogflow**

**Agent**

Google’s Dialogflow is a natural language understanding platform that helps developers to design conversational interfaces for websites,mobile apps. A natural language understanding module helps in identifying intents and extracting entities from the user’s request.

NLU consists of a supervised intent classifier that is trained on a variety of sentences as inputs and intents as target and a Entity Recognition model which can be a pretrained model like Spacy or Stanford’s NLP Library to extract the entities present in the input.



**Fig B: NLU Architecture**

In Dialogflow, we can create Intents that map the user’s input to the responses. In each intent, you define examples of user statements that can trigger the intent, what to extract from the statement, and how to respond. Entities are extracted from user queries and used as parameters for processing that particular Intent. Entities are of two types namely- system entities and custom entities. System entities are predefined entities that can match the common types of data such as (Location, time etc.,). Custom entities are entities that can be created by us for matching custom data i.e, **@pizza-type** entity that can match the type of Pizza available for purchase in a restaurant.

In Fig A, we can see that the user wants or intends to search for restaurants near 35 Peterborough Street, MA. The intent ‘search\_business’ is triggered by the user query since the query has matched the format of the training phrases that we have used to train the agent for that intent label. We enter multiple training phrases for each intent to capture all possible user queries. The entities extracted are the **@business-type** and @**user-location.** The entities can be made either optional or mandatory for the user. If the entities are mandatory, then the bot would perform an **Action** by asking the user to enter more details to get parameter values.These parameters are stored in the backend or Firestore database. **Context** represents the current state of the user’s request and allows our agent to carry the information from one agent to another. It represents a necessary condition that has to exist before an intent can be triggered.

The Dialogflow can connect with external systems using the Fulfillment code,which is deployed as a webhook. During a conversation, the fulfillment lets you use the information extracted by Dialogflow's natural language processing to generate dynamic responses or trigger actions on your backend.

**Model Choices & Intent Detection in Dialogflow**

Dialogflow provides an option to use Hybrid or ML only models for intent detection process based on the number of training examples. If the training examples are limited, Dialogflow uses the Hybrid model which comprises Rule based approach and a traditional classification model. The ML only model is adopted when the number of examples in intents is less. It may use a state-of-the-art Deep learning approach for intent classification.

We chose to use the Hybrid Model due to the limited size of our training phrases, as we manually created the training phrases for each of our intents.

The classification model’s performance is evaluated by the confidence score. This confidence score indicates how certain the agent is that the intent has been assigned correctly to a given user query. The score can have a value between 0 and 1. A score is calculated for each intent for each user input and the one with the highest score is returned as the result. If the confidence score falls below the threshold, which is 0.3 by default but can be changed, the fallback intent is triggered. This indicates that our agent does not recognize the user input. The fallback intent re-prompts the user to provide necessary inputs for the action. If no fallback intent is defined, then we won't get any response.

**Implementation Process:**

**Dialogflow Console -** This is the entrypoint and the main page in Dialogflow for building a new chat agent.

This is where we start defining our Intents, Entities, add our training phrases, setup our integrations, and validate how our model is performing on new user queries sent from end clients apps like Slack, Google Home.

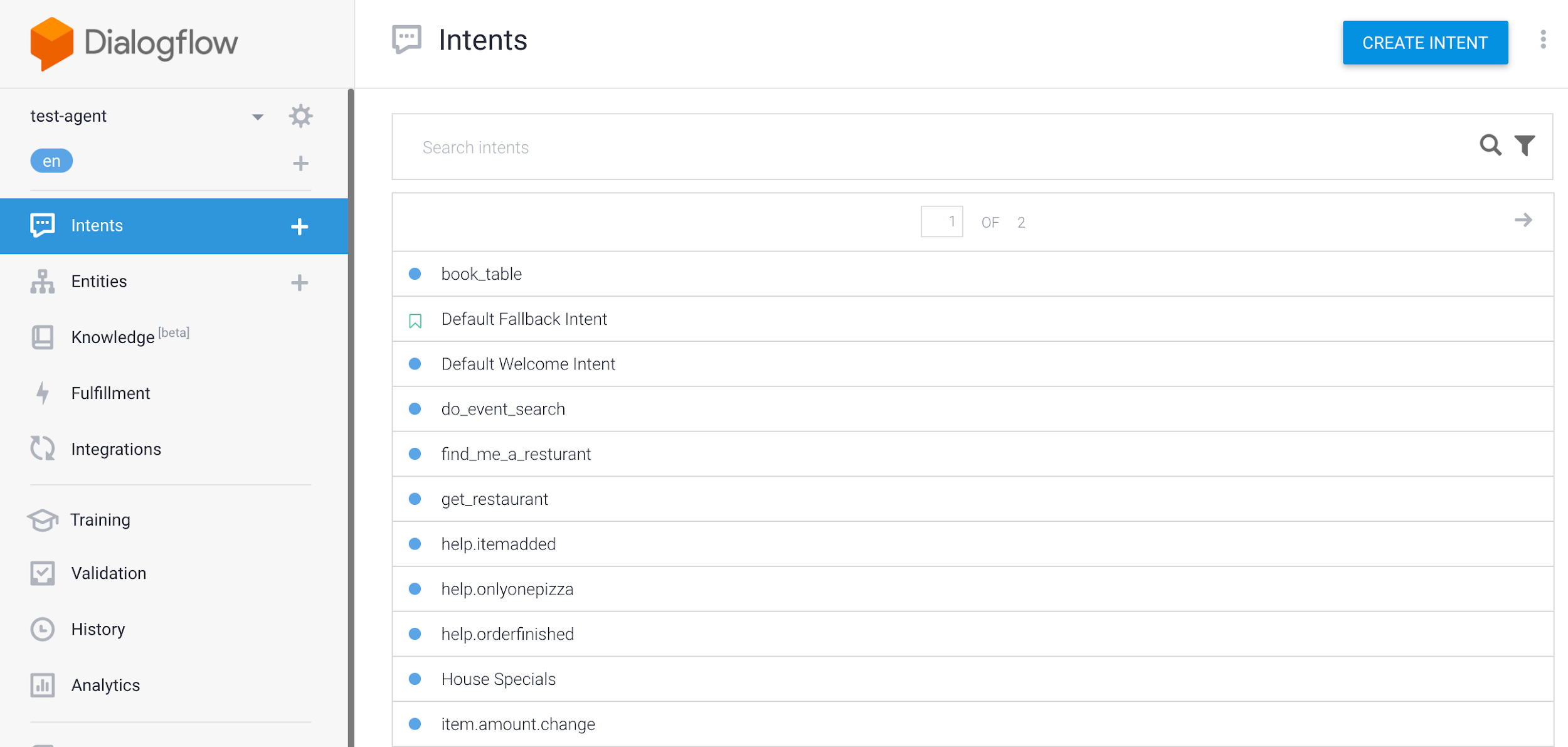
****

Fig. Dialogflow Console - Main Page

**Create New Agent**

First, we have to create a new Dialogflow agent and associate it with a new Google Cloud project which would allow us to enable Google Cloud Functions and Actions on Google SDK for this project, these would be required in order to service incoming requests from client applications.

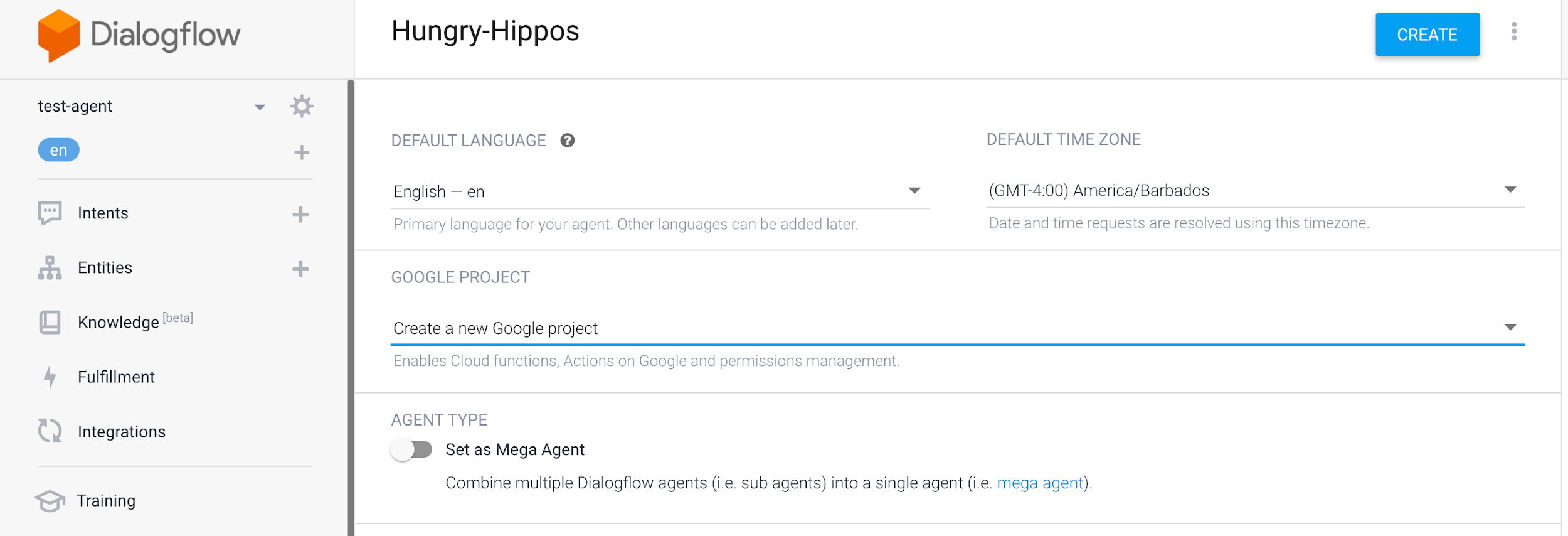
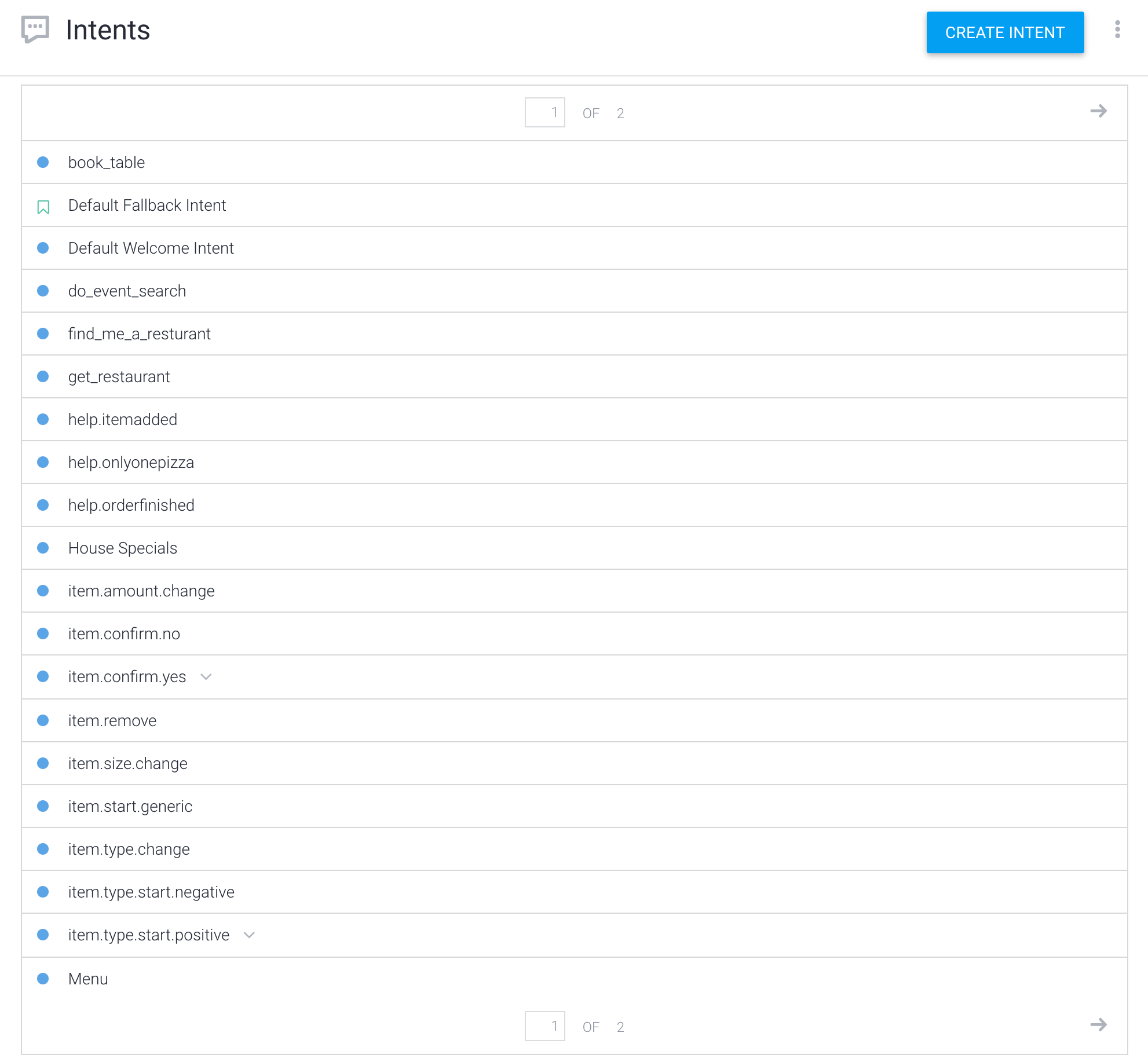


Fig. Creating New Test Agent

Note: For information on importing our exported agent files and code into Dialogflow see Appendix.

**Creating Intents**

Next we start adding the intents that we wanted our agents to recognize



We have a total of 29 intents for 6 Use Cases

**Creating Entities**

We created custom entities that could be used across all intents and would help in identifying business names, business types, cuisine types, business ratings, facility type, order type and thereby bootstrap the identification of such occurrences in conversational queries and somewhat circumvent the process of needing to tag too many entities in training phrases manually. (See Appendix. for additional info on these entity types)

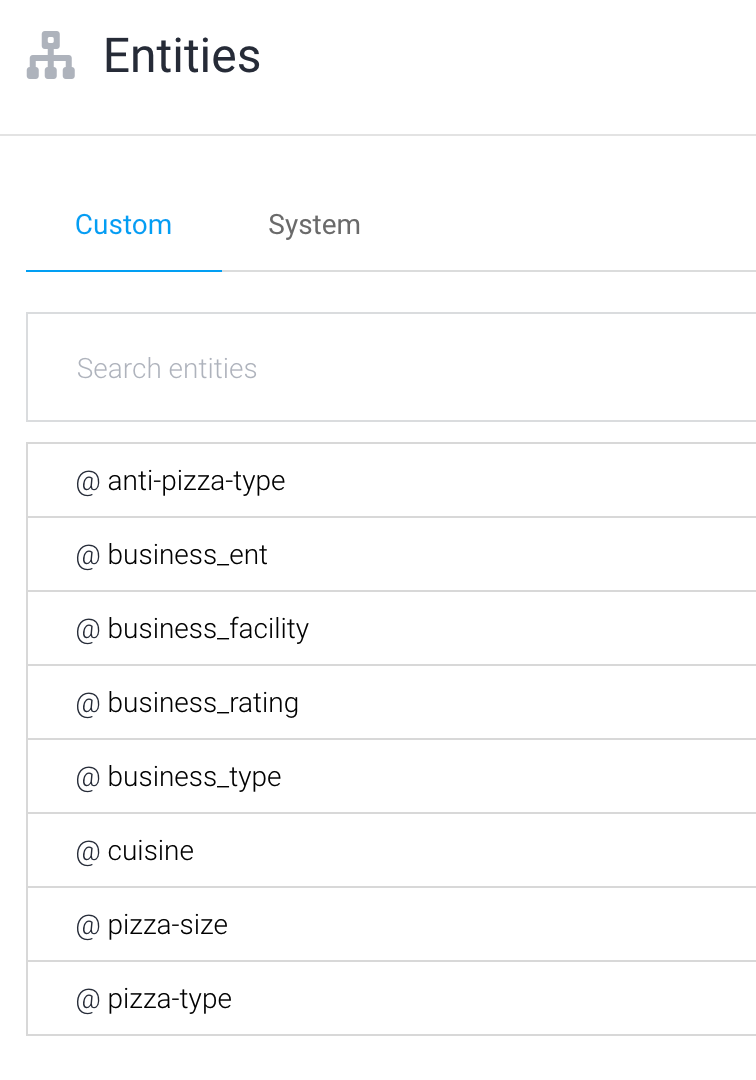


Fig. Creating custom entity types

**Adding Training Phrases**

For each intent we add our training phrases, and tag the entities with their corresponding entity type that we have defined. In the figure below, we show a sample of the training phrases defined for the business search intent.

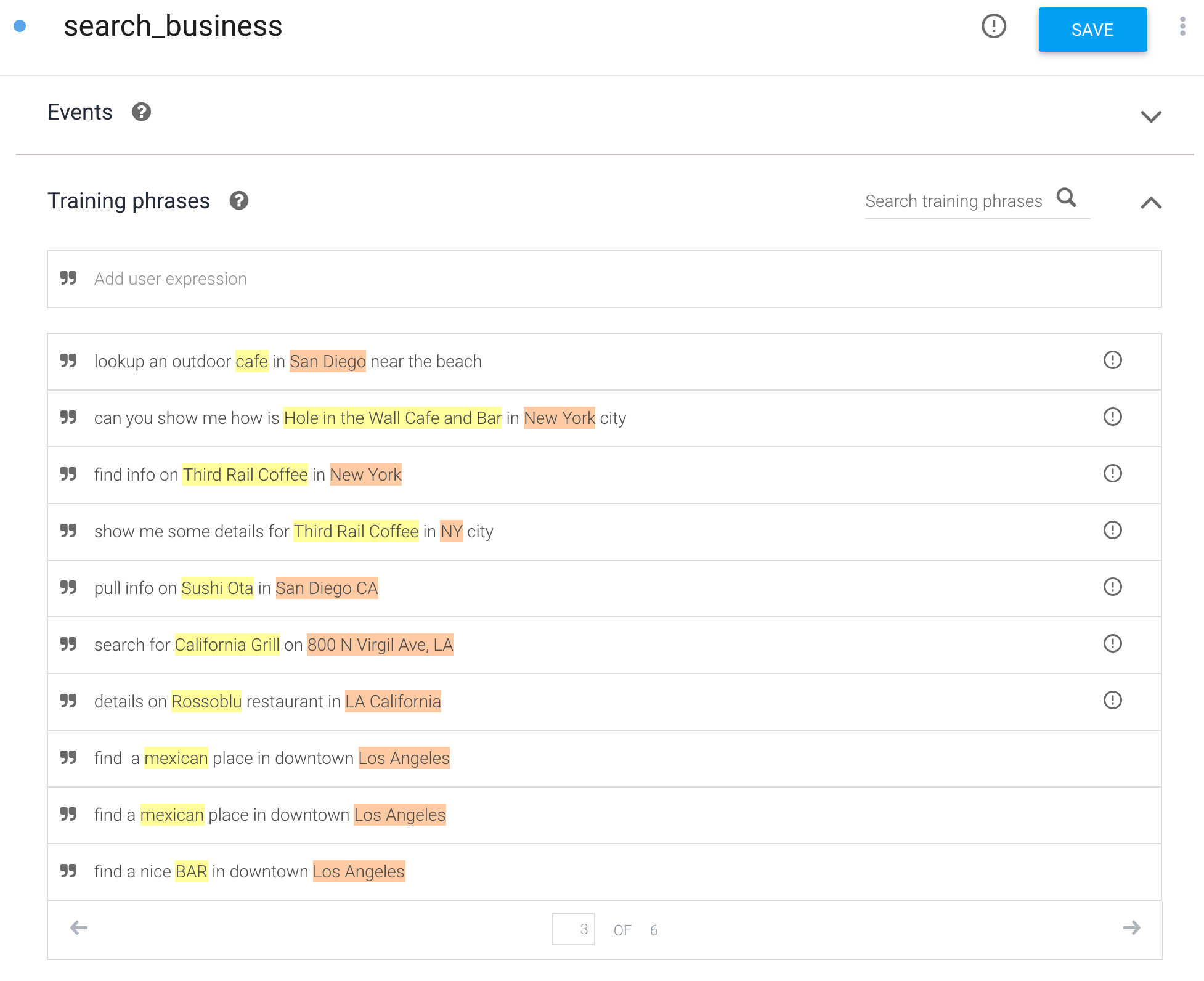
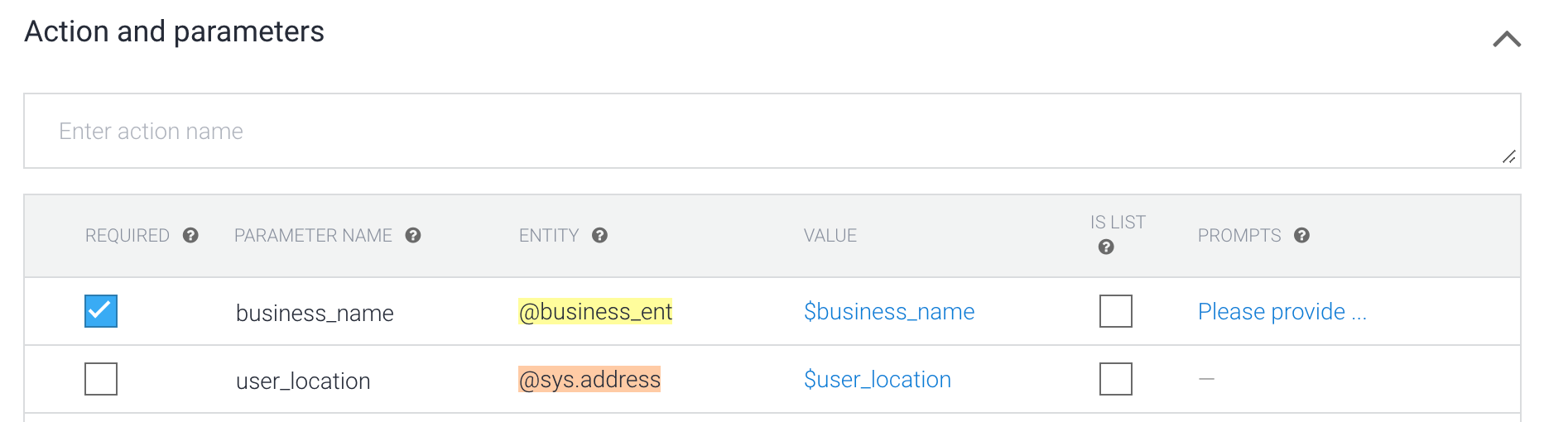


Fig. Sample of training phrases added for Yelp Business Search Intent in Dialogflow

The entities that are tagged in the training phrases include Business\_name and location entities which when extracted from the user request would be passed to the cloud function for fulfillment and fetch information pertaining to that business from the Yelp API.



Every addition of training phrases in an intent triggers a retraining and deployment of the model.

**Setting up the Google Cloud components for webhook calls and fullfillment:**

So far the steps we have listed above only train a model on the provided training phrases and detect/trigger an intent from the user query. This alone would not provide any dynamic response back to the user.

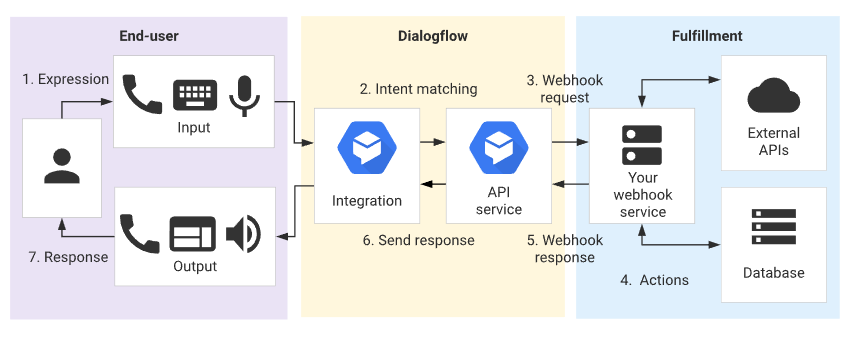
In order to do that we need to enable fulfillment for an intent. Each intent has a setting to enable fulfillment.

This allows us to respond to that intent by calling a service that we define. For example, if an end-user wants to reserve a table on Friday, your service can check your database and respond to the end-user with availability information for Friday.

When an intent with fulfillment enabled is matched, Dialogflow sends a request to our webhook service with information about the matched intent and the extracted entities and parameters.

The actions or responses that need to be executed for that intent are mapped with a corresponding cloud function which will then be executed. We have defined corresponding cloud functions for all of our intents, defining actions and responses appropriate for that use case. (See Appendix for a sample cloud function)

The following diagram shows the processing flow for fulfillment.



1. The end-user types or speaks an expression.
2. Dialogflow matches the end-user expression to an intent and extracts parameters.
3. Dialogflow sends a [webhook request](https://cloud.google.com/dialogflow/docs/fulfillment-webhook#webhook_request) message to your webhook service. This message contains information about the matched intent, the action, the parameters, and the response defined for the intent.
4. Your service performs actions as needed, like database queries or external API calls.
5. Your service sends a [webhook response](https://cloud.google.com/dialogflow/docs/fulfillment-webhook#webhook_response) message to Dialogflow. This message contains the response that should be sent to the end-user.
6. Dialogflow sends the response to the end-user.
7. The end-user sees or hears the response.

**Train and Publish the Model**

After adding the intents and entities, the model is trained and published. After the model is published, REST API can be accessed from the service to access the LUIS endpoint to get the intent and entity from the user’s query.

**Components of Hungry Hippos**

In this section, we will be discussing in depth about the different Use Cases and the intents defined within each use case and their respective workflows. **Note**: All the use cases and their respective intents have outputs defined for Slack and Google Home end clients.

**Use Case 1: Search for Business from Yelp**

This Use Case involves querying the Yelp database using the Fusion API endpoint to lookup the parameters extracted from the user query on matching one of the intents created for this use case. (see appendix for reference on the Yelp API endpoints used)

**Intent 1: Search for business info (search\_business)**

The end goal here is to bring back business information fields such as **address, no. of reviews, average rating, contact details, price, latest business image, business reviews.**

This intent will be triggered when the model is confident that the user query matches with the training phrases defined within this intent. The default confidence threshold for intent matching is 0.30.

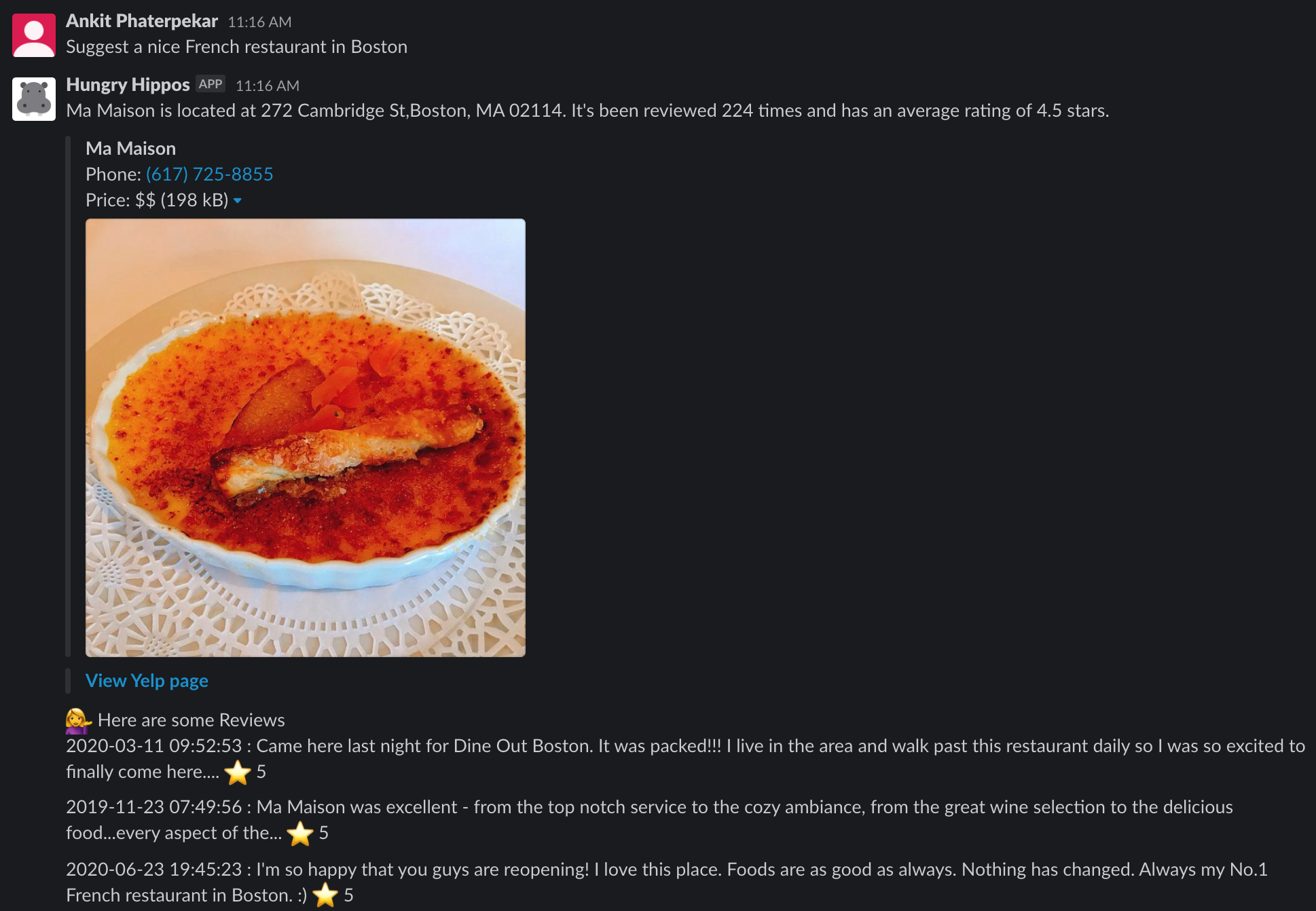
|  |  |  |
| --- | --- | --- |
| **Intent** | **Sample Training Phrases** | **Functional name in Dialogflow/ Cloud Function** |
| Search Business Info | show me some details for Rock On Pizza in Escondido, California | search\_business |
|  | lookup an outdoor cafe in San Diego |  |
|  | find me a pharmacy near me |  |

**For this intent we have set Business Name** as a required param. If not provided in the user query, the agent prompts the user to give one before proceeding.

**Location** is an optional param, if provided overwrites default user location. For the MVP we have set a default user location within the agent. Ideally we would want to pull this location metadata from the end device using the agent, but this required getting some additional permissions and steps that needed some more legwork, which we skipped for the purposes of this MVP. But we have designed the code keeping that extension in mind.

Once we have extracted the highlighted params, we next query the **Yelp Business Search endpoint** & **Yelp Review endpoint** with these parameters. This allows us to fetch information on our field of interest as mentioned earlier and build an appropriate response for the end client (Slack vs Google Home). All of these steps are coded and defined by us within the cloud function. (See Appendix for the Cloud function for this use case).

**Sample Slack Output Response:**



**Intent 2: Search for business details**

For this use case we wanted to answer user queries concerning the ‘operating hours’ for a store, or answer user queries regarding establishments open for food pickup or delivery.

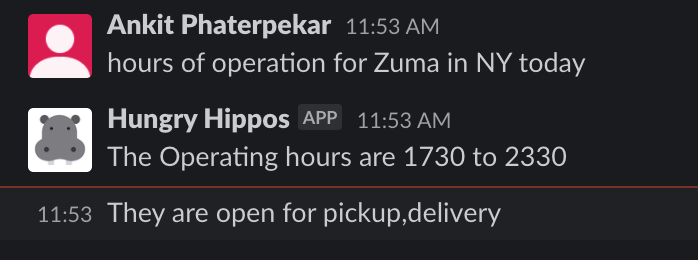
|  |  |  |
| --- | --- | --- |
| **Intent** | **Sample Training Phrases** | **Functional name in Dialogflow/ Cloud Function** |
| Search Business Details | When does Giant close tonight ? | search\_business.details |
|  | show me hours of operation for Rossoblu restaurant in LA California |  |
|  | Till what time is Laliguras open today ? |  |

**For this intent we have set Business Name** as a required param. If not provided in the user query, the agent prompts the user to give one before proceeding.

**Location** and Time are optional params, if provided overwrites defaults. The default for time is today

The process is similar to the one mentioned for the prior intent (business info intent), except that we query the Yelp’s **Business-Details endpoint** while execution. Yelp response provides as with a static list of weekly hours for the business, so we carry out some additional mappings for date-time objects to identify for which day of the week the user is requesting details regarding.

**Sample Slack Output**



**Use Case 2: Search for Events**

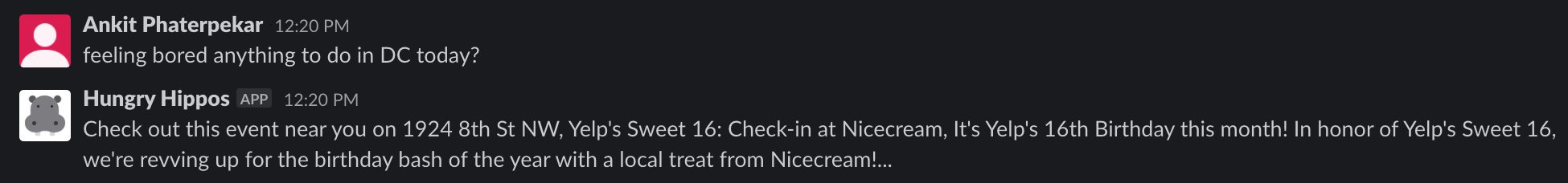
Here we wanted to design a functionality that would allow the user to search for nearby Events from the Yelp directory. For this feature we query the Yelp Events endpoint

Yelp has multiple categories to search from, the default category that we search for an event if none is provided by the user is 'food-and-drink' category.

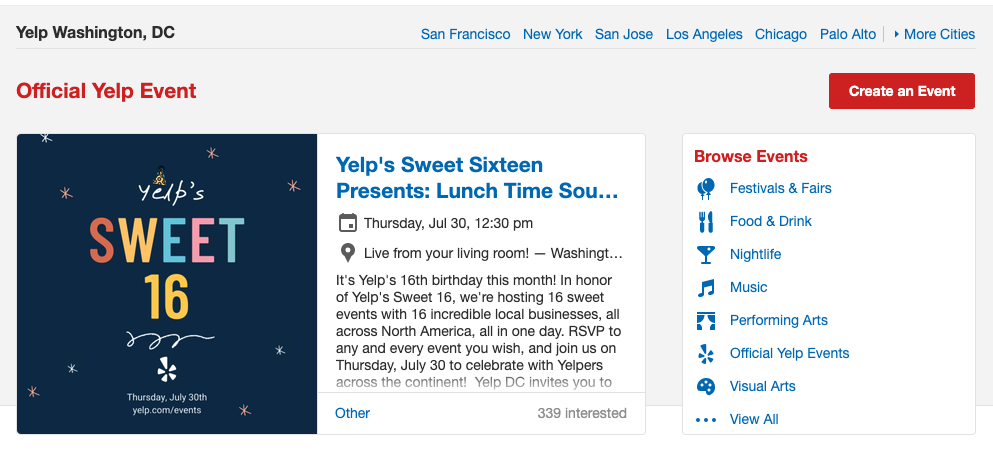
The events are sorted by popularity, other options available are "time-sort". Defacto, we search for events occurring within the next 24 hour time range. This can be extended to do an unbounded search as well for any time range.

We grab the first event, from the Yelp response and retrieve Event Name, Event Description, Event Location and build the agent response.

**Slack Sample Interaction:**



This pulled the correct information that we can validate from on the Events page.

****

**Use Case 3: Reserve a Table**

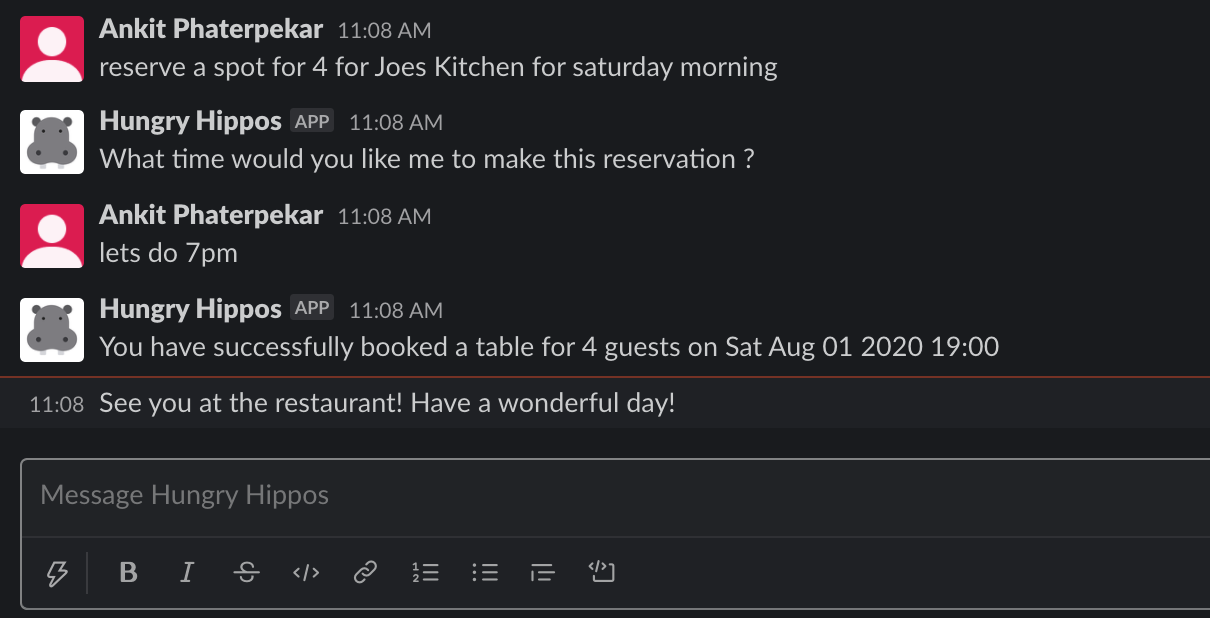
Here we wanted to design a functionality that would allow the user to initiate a conversation for reserving a table**. For this intent we have set all the entities within the training phrases i.e. guests, date, time as required parameters.** This means that agents will not close this particular conversation until all the required params are fetched from the user responses which is also called as slot filling.

When Slot filling is triggered the agent detects which parameter is missing from the user response, and comes back with the followup question for the user to request the missing details, before initiating the webhook call to the cloud function for further processing.

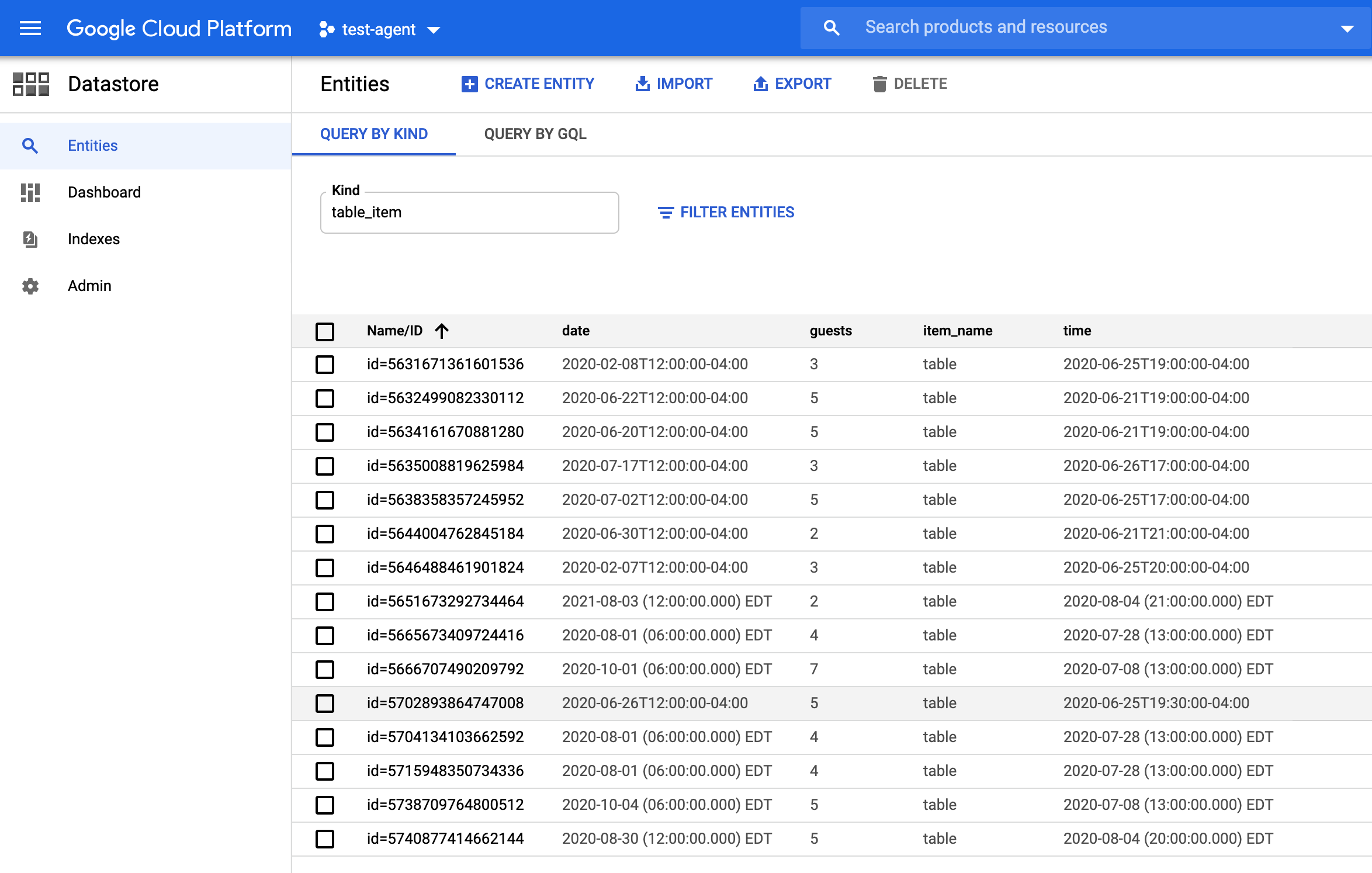
Slot Filling is desirable when we know in advance that a certain action, api request or code logic as defined in the Cloud function would fail due to a missing parameter/entity, and we would want to avoid our chatbot agent from crashing or returning unintended results.

|  |  |  |
| --- | --- | --- |
| **Intent** | **Sample Training Phrases** | **Functional name in Dialogflow/ Cloud Function** |
| Reserve a table | book a table for part of 10 for August 7th at 6pm | book\_table |
|  | reserve a spot for 4 for Joes Kitchen for saturday morning at 7 pm |  |
|  | plz make reservations for 5 at 9pm tomorrow |  |

**Sample Slack Interaction**



Once the request is successfully completed we store the reservation date in the Google Cloud Datastore table that we created for storing the reservation dates. The cloud Datastore is invoked within the Cloud Function defined for processing this intent.

****

**Use case: Placing an Order**

**Ordering a Pizza**

1. Asking details about configuration of Pizza

Some of the training phrases that could trigger the intents to start the order are:

|  |  |  |
| --- | --- | --- |
| **Operation** | **Training Phrases** | **Intent** |
| Start Order | “Hi” or “Order” | item.start.generic |
| Configure Pizza | “Can I have 4 large pepperoni?” | item.start.generic |
| Enquiring about a particular type of pizza | “Do you have Americanas” | item.type.start.positive |

Generally, while ordering a pizza in a restaurant, we generally enquire whether a particular type of Pizza is available and if yes, then we could start our order.

This action is mimicked by our bot via two custom entities namely @pizza-type and @pizza-type-no. @pizza-type consists of all the pizza types that can be offered whereas @pizza-type-no contains all the pizza types that are not offered by the restaurant.

When the customer asks the bot for a pizza type that is on the @pizza-type list, it will be matched to via *item.type.start.positiv*e,applying @pizza-type and the user is asked whether they want to order this type of pizza. The slot filling process is completed in the follow-up intent *item.type.start.positive* and the configuration of pizza is thus obtained.

In case, if a particular type of pizza is not available, this will be matched via *item.type.start.negative*, applying @pizza-type-no.

1. Confirmation of Item

When a pizza is configured, the bot will repeat the current configuration and confirms if the user is happy with his current order.

If the user confirms his order, then the order is added to the shopping cart/basket and the user is guided throughout the rest of the conversation by asking whether the user wants to place an additional order. If yes, then the bot can start configuring another pizza. If the user doesn’t want to add anything to the basket, the bot asks if they want to finish the order. If the user says yes, that is the end of the conversation and bot will reply “Hurray! you have completed your order”.

If the user is unhappy with the current order, before confirming it, he can change the configurations of the pizza.

**Modifying the Pizza Configuration**

The configurations of a pizza includes the type of pizza, amount of pizza/number of pizza to be ordered and the size of the pizzas. These configurations can be changed in a number of ways.

For example, if the user query to the bot is “ 4 small Pepperoni please”. Post confirming this configuration, the *item* context is set. While the *item* context is still intact, the configuration of the most recent order can be changed by the below mentioned intents.

|  |  |  |
| --- | --- | --- |
| **Operation** | **Training Phrases** | **Intent** |
| Change type of Pizza | “Can I have a Margherita instead?” | item.type.change |
| Change amount of Pizza | “I want 3” | item.amount.change |
| Change size of Pizza | “Can I have large instead” | item.size.change |

**Contents of the Basket**

In order to mimic a shopping cart, we have added a shopping basket functionality to our bot. Every time a user confirms the pizza configuration, the order is added to the basket.

An overview of the intents that help in modification of the basket is as shown below

|  |  |  |
| --- | --- | --- |
| **Operation** | **Training Phrases** | **Intent** |
| Finish the order | “I am done with my order” | order.finish |
| Remove an item from the basket | “Please remove pepperoni” | item.remove |
| Modify an item configuration | “Can I get a Mushroom instead” | item.type.change , item.size.change,  item.amount.change |
| Delete the basket | “Delete basket” | order.cancel |
| Start the order again post deletion | “Start order” | order.start |
| View items in basket | “Show basket” | order.showbasket |

1. Structure of Basket

The basket is a context that is programmatically set in the fulfillment. The items themselves are then passed along through the parameters of the context.

basketContext = {'name': 'basket', 'lifespan': 50, 'parameters': {}}

The items in the basket could just be an array of objects. However, the bot uses an object as a data structure, due to the fact that the Google Cloud products were designed for concurrency and removing or adding items from the array could result in inconsistent states.

"items": {

"900f2fb2-9ef7-47ea-bb96-3c2b6eb64131-0820055c": {

"amount": 2,

"type": "Margherita",

"size": "14''"

},

"e6b0da6a-1186-452d-b263-31ea401d080d-0820055c": {

"size": "10''",

"type": "Pepperoni",

"amount": 1

}

Since the basket is a list of items, we are simulating an array in that object. For the keys, ResponseID of Dialogflow conversation is used which serves as an unique ID.

1. Modifying the Last Item in Basket

The bot provides flexibility of changing the last item that was added . This is taken care of by the *confirmItem()* method in the fulfillment. If an item exists in the basket, then the record in basket is updated and if it doesn't exist, a new record of item is created in the basket. The confirmItem method adds the response ID to the item context, so that it can be used to get access to that item in the basket when the user wants to change its configuration later.

1. Removing a specific item

In case the user wants to remove a specific item from the basket, he can mention the type of Pizza that he wants to remove. For example, “Remove Pepperoni”. The particular item is deleted from the basket and the remaining items are retained. This is taken care of by the *removeItem()* method in Fulfillment. The method iterates through the items present in the basket and removes those items based on the type of pizza provided by the user.

1. Deleting the basket

The basket can be emptied and reset if the user wants to start over the order again thus triggering the intent *order.start*. The basketContext can be deleted by setting the lifespan to 0.

let basketContext = {'name': 'basket', 'lifespan': 0};

agent.context.set(basketContext)

**Helping the User**

There are no user interface elements in the bot that can guide the user on how it is intended to be used. So, at certain points in the conversation, the user should be able to ask for help and get information on the conversation flow.The bot fullfils this requirement of the user by matching the “help” query to different intents by using contexts.

|  |  |  |
| --- | --- | --- |
| **Point in conversation** | **Context** | **Intent** |
| After adding the item | *item* | help.itemadded |
| Post placing an order | *order-done* | help.orderfinished |

1. Fallback in Help

There is a special help intent: help.onlyonepizza

The users might try to order two pizzas in the same query.Currently, the bot has no way of dealing such a case.Thus, it outputs a simple message letting the users know that they can place only one order.

**Small Talk**

Smalltalk intents were added to make the bot more conversational. These smalltalk intents can be extended,depending on what conversational style we want our bot to achieve.

1. Context-Dependent Smalltalk

These intents are context dependent. The user gets different responses depending on where he is in the conversational flow. For example, after adding an item, the user says “Great!”, the *smalltalk.user.positive.itemdone* intent is matched through the *item*context.This is interpreted as a sign that the user is happy about having a pizza added to their basket.

1. Muti-Step Smalltalk

After placing the order,if the user is unhappy with it and types “no!”,the *smalltalk.user.negative.inorder* intent is triggered.The bot tries to pacify the user by telling him that he can always modify his order. Depending on what the user says next, the bot goes deep into the branch of the conversation, by going into different followup intents.

**Integrations**

Once the Dialogflow agent is ready, we can use the one-click integrations to connect to various platforms. The different types of integrations offered are:

Text based Integrations:

1. Slack:

We can create our own slack bot and integrate them with our Dialogflow Agent. Steps for integration can be found at : <https://cloud.google.com/dialogflow/docs/integrations/slack>

**Project planning**

1. Project Milestones and Deliverables

Project milestones can be used to track progress and identifying objectives which have been satisfied

|  |  |  |
| --- | --- | --- |
| Week | Milestone | Milestone Description |
| Week 1 | 1 | Analysis & requirement specifications |
| Week 2 | 2 | Design presentation |
| Week 3 | 3 | Iteration 1 |
| Week 4 | 4 | Iteration 2 |
| Week 5 | 5 | Iteration 3 & Status update 1 |
| Week 6 | 6 | Iteration 4-Final build & Status update 2 |
| Week 7 | 7 | Testing & Evaluation |
| Week 8 | 8 | Packaging & Documentation |

1. Development Iteration Breakdown

|  |  |
| --- | --- |
| Iteration | Functionality implemented |
| Iteration 1 | 1. Setup Dialogflow and connect to GCP 2. Add intents,entities, training phrases and enable webhook fulfillment 3. Integrate Places API and Yelp Fusion API 4. Store values obtained from API in Firebase |
| Iteration 2 | 1. Integrate the bot with Slack and test it 2. Implement additional use cases like reservation of tables and placing an order 3. Explore feasibility of open APIs to fetch Menu |
| Iteration 3 |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Limitations**

The goal of our project was to build a restaurant bot that could automate different restaurant services and could be integrated with different platforms. However, there are some roadblocks that we faced while building the agent..

1. Lack of flexibility to integrate Voximplant with Google Assistant: To initiate an outbound call, Voximplant or Dialogflow’s telephony could be used. However, these apps cannot be integrated with Google Assistant as such features are not yet available.
2. Manual work involved in Training : In many instances, Dialogflow makes it harder than it should be to automate processes and expand your conversational agent’s learning. This can get annoying because you have to input many things manually, especially when you consider the need to train your bot over time. Therefore, the ability to even make your chatbot better is hindered, which unfortunately defeats one of the platform’s main purposes
3. API server return error: We use the Yelp fusion API to fetch details about different businesses. Sometimes, the server might throw a ‘500 status return error’ when it encounters an unexpected condition which prevents it from fulfilling the request from a client. This could be triggered by different issues like error in scripting, failed disk or non-functional software module.
4. Bias in Training Phrases - Since the developer is defining the training phrases for each intents, there is a creator bias. Due to this, the training phrases are more tuned to the developer’s vocabulary and might fail to capture the heterogeneity in different users’ queries. This could be overcome by data augmentation where we can explore simulating more training phrases using synonyms and verb replacement strategies. Large selection of custom entities can be created to mitigate the limitations of less training phrases.

We expect that Alpha/Beta release will help us in capturing the failure due to lack of diversity in training phrases and make our agent more robust.

**Exporting and Packaging Agent**

1. The agent can be backed up or transferred to a different account via the Import and Export settings tab, found in Agent settings. This feature allows us to export the agent and its intents,entities into a zipped folder that can be uploaded on Github.
2. The cloud functions can be exported separately and reimported within the end user’s Dialogflow console.
3. A documentation regarding the workflow for each intent and the required parameters can be maintained along with the steps for integration for future reference.

**Pricing of Dialogflow Services**

1. The Dialogflow is available in two pricing plans: Essentials and Plus. Dialogflow is priced monthly based on the Edition and Pricing plan in addition to the number of requests made. The main difference between Essentials and Plus plans is that the Plus plan offers enterprise-ready quotas for knowledge connectors.
2. It is to be noted that a request is defined as any call to the Dialogflow service,whether direct via API usage or indirect with integration or console usage.Unless a feature is indicated as included, prices are cumulative for all features used by a request.

Refer appendix to get more information on price comparison for the two editions and pricing plans.

**Sample Conversations with Agent**

**Use Case: Reservation of Table**

**User:** Hi

***Agent:*** *Hi there! Welcome to Hungry Hippos, you can say things like "Where is the nearest Starbucks? ", "What are the operating hours for Giant?", "Reserve a table" or "Place an order". What would you like help with?*

**User:** Reserve a Table for 2

***Agent:*** *Which date would you like me to reserve?*

**User:** 31st July 2020

***Agent:*** *What time would you like me to make this reservation?*

**User:** 7pm

***Agent****: You can't make a reservation in the past. Please try again!*

**User:** Reserve a table for 5 on 20th August 2020 at 8pm

***Agent:*** *You have successfully booked a table for 5 guests on Sat Aug 20 2020 20:00*

*See you at the restaurant! Have a wonderful day!*

**Use Case: Order a Pizza**

**User:** Hi

***Agent:*** *Hi there! Welcome to Hungry Hippos, you can say things like "Where is the nearest Starbucks? ", "What are the operating hours for Giant?", "Reserve a table" or "Place an order". What would you like help with?*

**User:** Do you have Pepperoni?

***Agent:*** *Yes, we do offer Pepperoni. Would you like to order one?*

**User:** I want a large Pepperoni!

***Agent:*** *How many would you like*

**User:** 7

***Agent****: Fantastic! so that's 7 Pepperoni in 14''. Can you confirm?*

**User**: No! Make it 5

***Agent:*** *Sure. So that's 5 pizza Pepperoni in 14'' instead, all good?*

**User**: Yes

***Agent:*** *Confirming 5 pizza Pepperoni in 14”. Anything else?*

**User**: Nah

***Agent:*** *Alright, do you want to finish your order?*

**User**: Yeah

***Agent:*** *Hurray! you have completed your order!*

**APPENDIX**

**Importing an agent from exported files**

Install Dialogflow Agent

1. Go to the Google’s Dialogflow Console using your Google credentials : <https://dialogflow.cloud.google.com/>.
2. Click on ‘Create New Agent” from the agent selector dropdown.
3. Give it a name “Hungry Hippos” or “test-agent” and click “Create”.
4. Now go to Agent settings which is a small icon near the agent selector dropdown.
5. Select “Export and Import” tab.
6. Click on “RESTORE FROM ZIP” and upload the test-agent.zip file provided.

Install Fulfilments Functions

1. Click on the ***Enabled*** switch on the Inline Editor in the **Fulfillment** Tab of the Dialogflow Console.
2. Copy the contents of index.js and package.json into the Inline Editor window

Connect to Firebase Database/GCP Console

Enable Billing Account to link the Dialogflow Agent to access the services provided by Google Cloud.