



NTT DATA Company

Technology - Digital experience

Celiaco responde

Carla Brugulat Rica

Tabla de contenido

Tabla de contenido.....	2
1. Introducción	3
2. Esquema del proyecto.....	3
3. Objetivos	5
4. Desarrollo del proyecto.....	5
4.1 DialogFlow	6
4.1.1 Crear Agent	8
4.1.2 Crear Entities	9
4.1.3 Crear Intents.....	11
4.1.4 Integración	19
4.2 Content Experience Cloud.....	20
4.2.1 Crear Site	21
4.2.2 Definir una estructura de datos	27
4.3 Node.js	30
4.3.1 Crear aplicación de Node.js que se conecte con el chatbot	30
4.4.3 Crear aplicación de Node.js que se conecte con el chatbot y el CEC.....	33
4.5 Postman	38
4.6 Heroku.....	40
5. Bibliografía	46

1. Introducción

A rasgos generales, en este proyecto se pretende realizar una conversación con un chatbot capaz de recuperar contenido del *Content Experience Cloud (CEC)*.

Oracle Content and Experience Cloud es un *hub* de contenido basado en la nube que permite activar la gestión de contenido omnicanal y agilizar la experiencia del usuario.

El chatbot se quiere crear y gestionar con *DialogFlow* y la aplicación que permita la comunicación entre ellos se desarrollará con *Node.js*. Para ello, Oracle nos ha proporcionado una *Rest API* que implementa la comunicación con el *CEC*.

En los siguientes apartados se definen los objetivos principales y los procesos que se han realizado en cada etapa del proyecto para llegar al resultado final deseado.

2. Esquema del proyecto

En el siguiente esquema se muestra la idea global del proyecto que nos permitirá a continuación analizar gráficamente las comunicaciones que se deben establecer para obtener el resultado esperado.

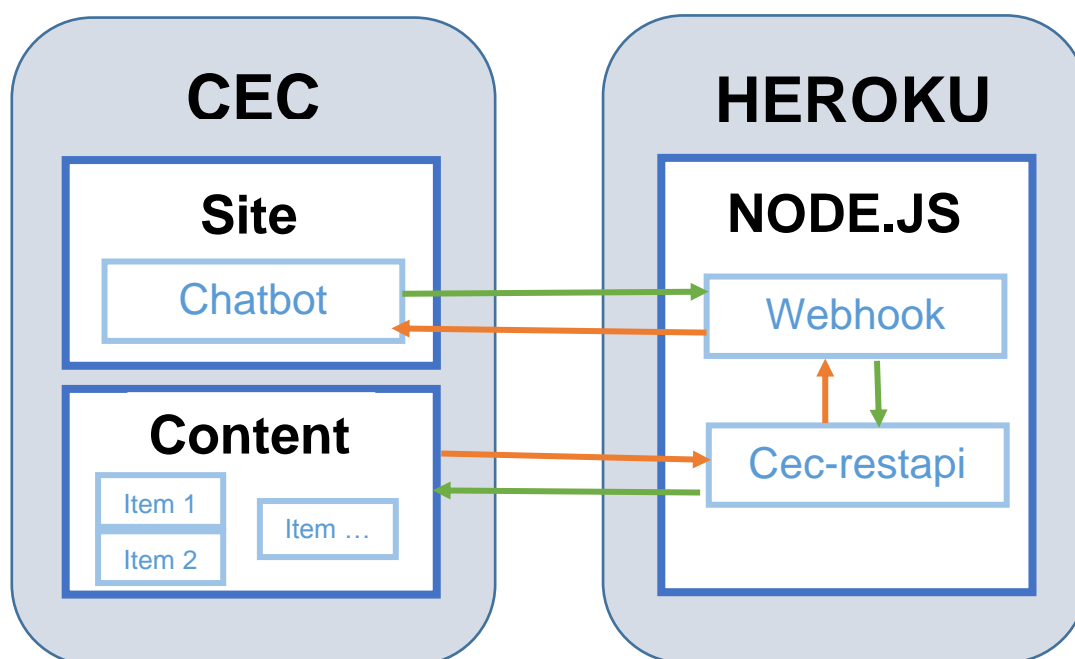


Ilustración 1: Esquema del proyecto

Para desarrollar el proyecto vamos a crear un site y una estructura de datos dentro del *CEC*, el site contendrá un *frame* con el chatbot que crearemos en *DialogFlow*.

La idea general será recuperar la información que contiene la estructura de datos a través de una conversación en el chatbot.

Para poder recuperar el contenido, el primer paso es habilitar la opción de *Webhook* en el chatbot e implementar una aplicación con *Node.js* que sea capaz de recuperar el código *json* de la conversación del chatbot que ha activado el *Webhook*.

Una vez somos capaces de recuperar la parte de conversación que nos indica el contenido que se requiere de la estructura de datos del *CEC*, tenemos que pasar dicha información al *cec-restapi* para que establezca las comunicaciones pertinentes con el *CEC* y recupere el *content type* o *content item* necesario.

A partir de aquí el proceso es el mismo pero a la inversa:

Cuando el *CEC* recibe la petición de *cec-restapi* le pasa el contenido deseado y este lo devuelve a la aplicación de *Node.js* que hace de puente con el chatbot.

El *Webhook* devolverá en formato de conversación (*json*) la información recuperada y se mostrará en el *frame* del chatbot dentro del site.

Para que las dos aplicaciones de *Node.js*: *Webhook* y *cec-restapi* se puedan utilizar las vamos a subir a *Heroku* que es una plataforma como servicio, es decir, un entorno de desarrollo completo que se hospeda en la nube y permite que los desarrolladores de aplicaciones creen aplicaciones de forma rápida y sencilla.

3. Objetivos

A continuación se listan los principales objetivos del proyecto que se deben cumplir:

- Crear un chatbot básico de pregunta/respuesta con *DialogFlow* con *Webhook* habilitado.
- Crear Site en el *CEC* a partir de una *template* por defecto.
- Definir una estructura de datos y añadir elementos a esta.
- Crear un componente “customizado” que integre el chatbot al site.
- Conectar el chatbot con una aplicación *Node.js* que devuelva una respuesta.
- Recuperar contenido del *CEC* para que a través de una aplicación *Node.js* devuelva esta información al chatbot.

4. Desarrollo del proyecto

En este apartado se muestra paso a paso cómo se ha desarrollado el proyecto completo, para ello se divide la información en cinco apartados, uno para cada plataforma utilizada:

1. *DialogFlow*
2. *Content Experience Cloud*
3. *Node.js*
4. *Postman*
5. *Heroku*

En cada apartado se realiza primero una explicación teórica y después una parte práctica cómo ejemplo.

El ejemplo que vamos a seguir es el siguiente: el site que se crea es una página web sobre la enfermedad celiaca, en ella se encontrará el chatbot al cual el usuario de la web le podrá preguntar sobre palabras clave del mundo sin gluten, cómo celiacuría, biopsia, gluten, zonulina, etc. Las definiciones de estas palabras

estarán contenidas dentro de la estructura de datos del *CEC*; así que cuando el usuario pregunte (por ejemplo) : qué es una biopsia? Se recuperará el contenido de la estructura de datos correspondiente a la palabra biopsia.

4.1 DialogFlow

En este apartado se detalla cómo realizar un chatbot extremadamente simple a través de *DialogFlow* y como se activa la opción de *Webhook*.

Para crear un chatbot primero necesitamos tener tres conceptos básicos claros:

Que es un *Agent*?

Los *Agents* se pueden describir cómo módulos de entendimiento de lenguaje natural. .

Que es un *Entity*?

Las *Entities* son herramientas potentes que se utilizan para extraer valores de parámetros de entradas de lenguaje natural. Cualquier información importante que se desee obtener de la solicitud de un usuario tendrá una entidad correspondiente.

Que es un *Intent*?

Los *Intents* representan un mapeo entre lo que dice el usuario y que acción debe tomar tu software para responder adecuadamente al usuario.

Sabiendo esto podemos empezar con la creación de un chatbot en *DialogFlow*; el primer paso es tener cuenta registrada en el portal o disponer de una cuenta de *Google*. Seguidamente, entrar en la Consola de *DialogFlow*, para poder empezar con la creación del *Agent*:

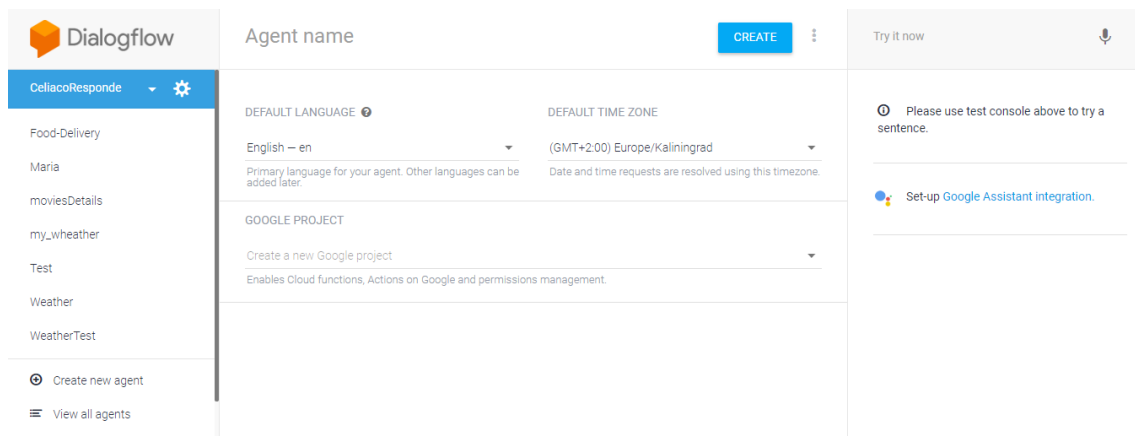


Ilustración 2: DIALOGFLOW - Pantalla de creación de un Agent

Seguidamente podremos empezar a crear *Entities* e *Intents*, en las siguientes imágenes vemos las pestañas correspondientes a la creación de estos:

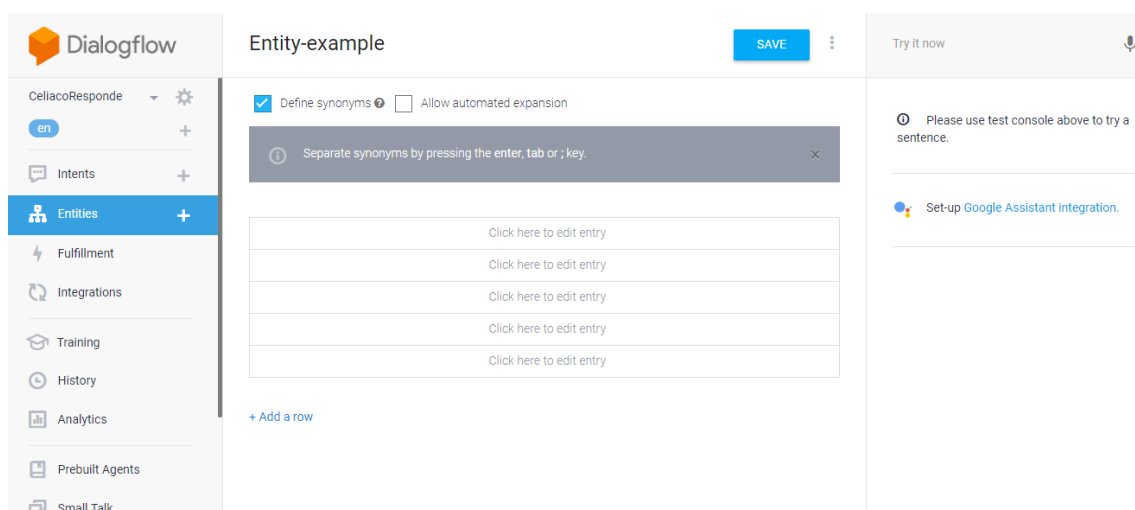


Ilustración 3: DIALOGFLOW - Pantalla de creación de Entities

Dialogflow

CeliacoResponde

Intents

Entities

Fulfillment

Integrations

Training

History

Analytics

Prebuilt Agents

Docs

Intent name

SAVE

Contexts

Events

Training phrases

Search training phrase

Add user expression

Action and parameters

REQUIRED	PARAMETER NAME	ENTITY	VALUE	IS LIST
<input type="checkbox"/>	Enter name	Enter entity	Enter value	<input type="checkbox"/>

Try it now

Please use test console above to try a sentence.

See how it works in Google Assistant.

Ilustración 4: DIALOGFLOW – Pantalla de creación de Intents

Finalmente, sólo nos queda habilitar el *Webhook* y la integración.

Ahora se realizan los pasos anteriores con el ejemplo correspondiente:

4.1.1 Crear Agent

Dialogflow

CeliacoResponde

SAVE

General Languages ML Settings Export and Import Share

DESCRIPTION

Describe your agent

DEFAULT TIME ZONE

(GMT+1:00) Europe/Madrid

Date and time requests are resolved using this timezone.

GOOGLE PROJECT

Project ID

celiacoresponde-df7f0

API VERSION

Ilustración 5: DIALOGFLOW - Creación Agent Ejemplo

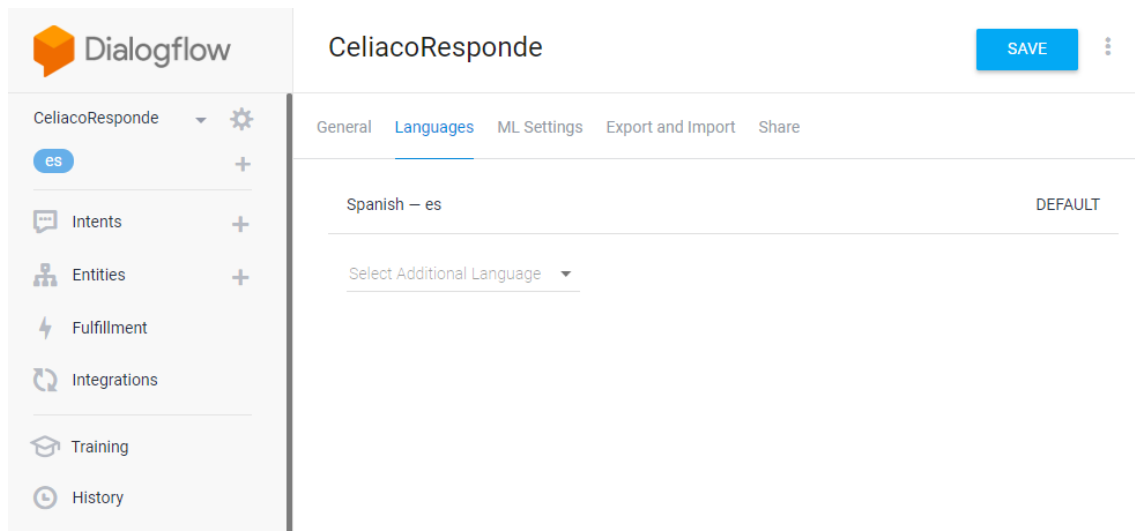


Ilustración 6 : DIALOGFLOW - Creación Agent Ejemplo

Los parámetros que deberemos modificar para este ejemplo son el nombre que va a tener el *Agente*, en este caso, *CeliacoResponde*, la zona horaria y el lenguaje que va a usar nuestro chatbot. Cómo queremos que la conversación se realice en Castellano, escogemos *Spanish – es*.

4.1.2 Crear Entities

Para este ejemplo vamos a crear tres *Entities* diferentes:

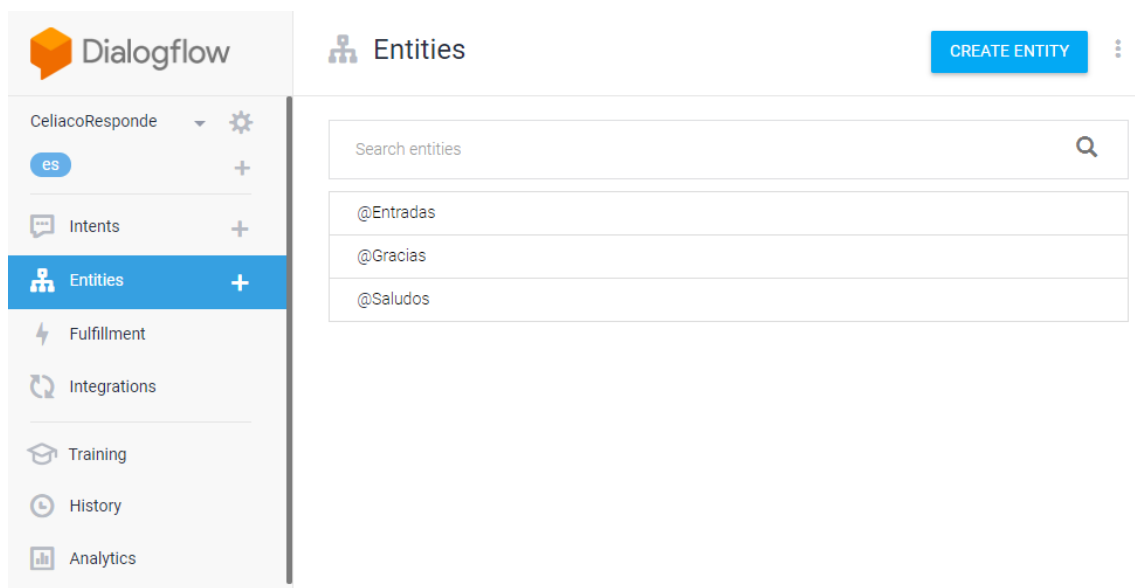
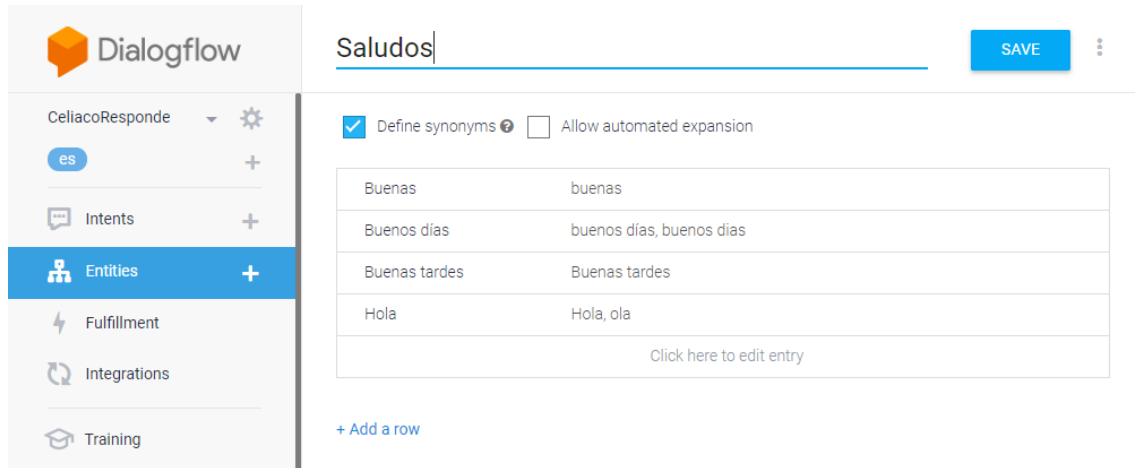


Ilustración 7: DIALOGFLOW – Conjunto de Entities creadas

La primera entidad que vamos a crear reúne el conjunto de Saludos que el chatbot va a identificar por parte del usuario:



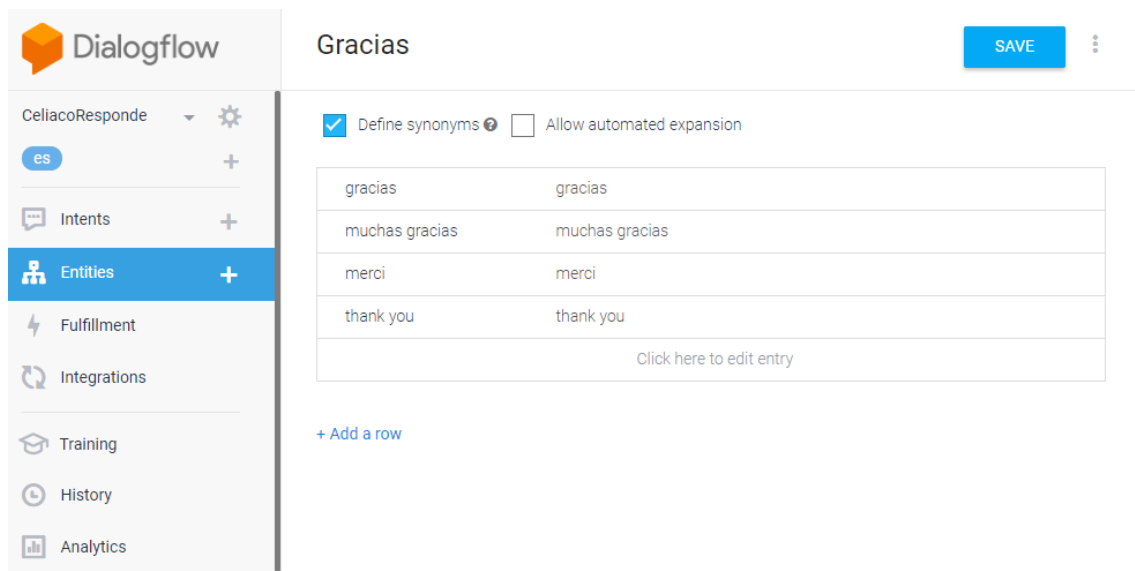
The screenshot shows the Dialogflow console interface. On the left, a sidebar contains navigation options: 'CeliacoResponde' (selected), 'es', 'Intents', 'Entities' (highlighted in blue), 'Fulfillment', 'Integrations', and 'Training'. The main area is titled 'Saludos' and includes a 'SAVE' button. Below the title, there are two checkboxes: 'Define synonyms' (checked) and 'Allow automated expansion' (unchecked). A table lists four entities with their synonyms:

Buenas	buenas
Buenos días	buenos días, buenos días
Buenas tardes	Buenas tardes
Hola	Hola, ola

Below the table, there is a link 'Click here to edit entry' and a '+ Add a row' button.

Ilustración 8: DIALOGFLOW - Entity Saludos

Seguidamente, la Entity que agrupa el conjunto de formas de dar las gracias:



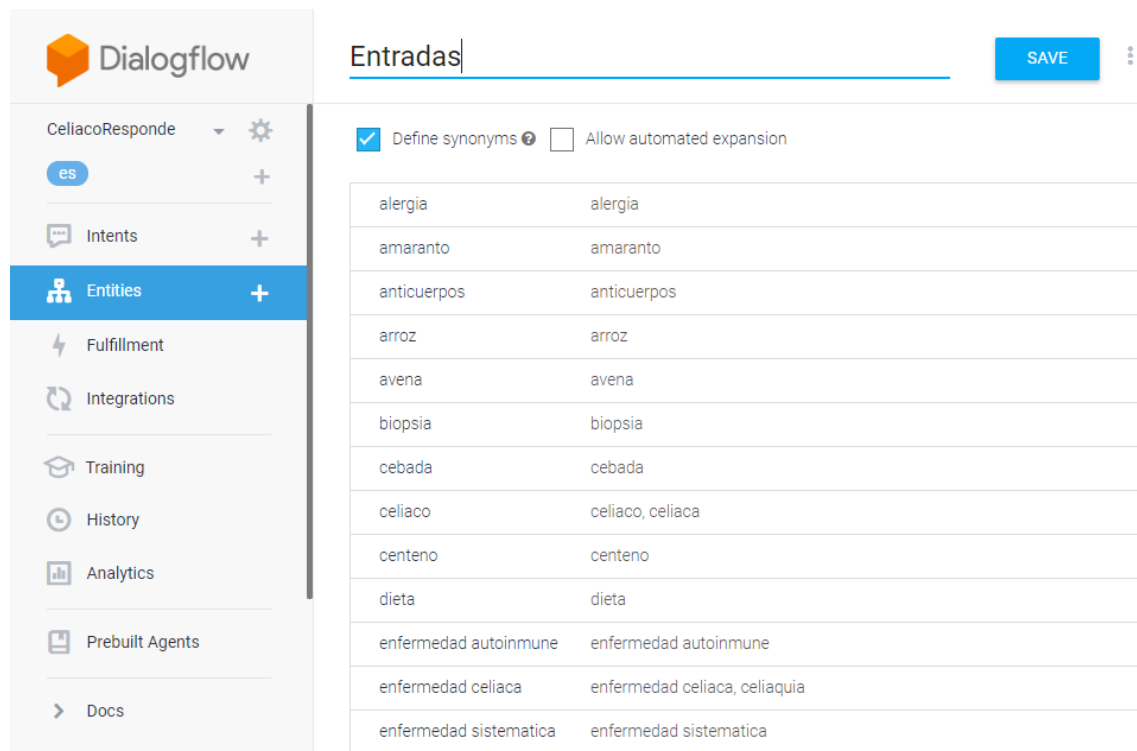
The screenshot shows the Dialogflow console interface for the 'Gracias' entity. The sidebar is identical to the previous screenshot, with 'Entities' highlighted. The main area is titled 'Gracias' and includes a 'SAVE' button. Below the title, there are two checkboxes: 'Define synonyms' (checked) and 'Allow automated expansion' (unchecked). A table lists four entities with their synonyms:

gracias	gracias
muchas gracias	muchas gracias
merci	merci
thank you	thank you

Below the table, there is a link 'Click here to edit entry' and a '+ Add a row' button.

Ilustración 9: DIALOGFLOW – Entity Gracias

La última Entity que vamos a crear es el conjunto de palabras de las cuáles vamos a poder recuperar información del *CEC*:



Dialogflow

CeliacoResponde es +

Intents +

Entities +

Fulfillment

Integrations

Training

History

Analytics

Prebuilt Agents

> Docs

Entradas SAVE ⋮

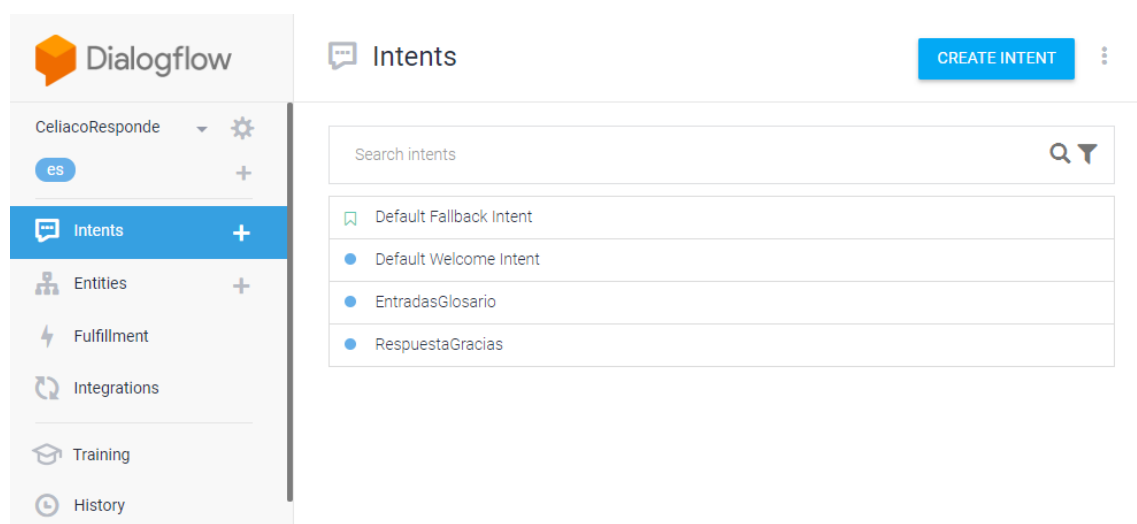
☒ Define synonyms ⓘ ☐ Allow automated expansion

alergia	alergia
amaranto	amaranto
anticuerpos	anticuerpos
arroz	arroz
avena	avena
biopsia	biopsia
cebada	cebada
celiaco	celiaco, celiaca
centeno	centeno
dieta	dieta
enfermedad autoinmune	enfermedad autoinmune
enfermedad celiaca	enfermedad celiaca, celiarquia
enfermedad sistematica	enfermedad sistematica

Ilustración 10: DIALOGFLOW – Entity Entradas

4.1.3 Crear Intents

También vamos a crear tres *Intents*:



Dialogflow

CeliacoResponde es +

Intents +

Fulfillment

Integrations

Training

History

Intents CREATE INTENT ⋮

Search intents 🔍 🔼

- Default Fallback Intent
- Default Welcome Intent
- EntradasGlosario
- RespuestaGracias

Ilustración 11: DIALOGFLOW – Conjunto de Intents

El primero que vamos a crear es el de inicio de la conversación, los saludo que se pueden intercambiar el chatbot con el usuario:

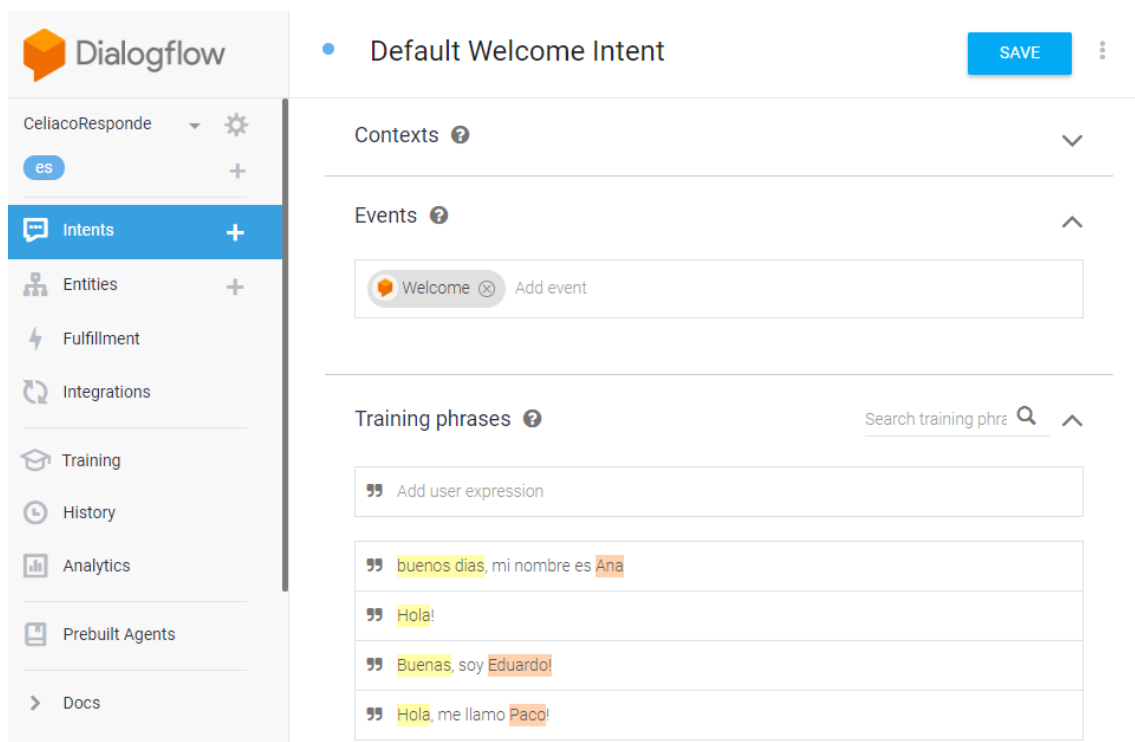


Ilustración 12: DIALOGFLOW – Default Welcome Intent, Training Phrases

En la sección de *Training phrases* se añaden las frases con las que el usuario creemos que puede empezar la conversación, y por lo tanto vamos a entrenar el chatbot para que las entienda. Como se ve en la imagen algunas palabras quedan remarcadas en color; las de color amarillo corresponden al conjunto de expresiones que forman la *Entity* de Saludos que hemos descrito con anterioridad. Las que se marcan en rojo provienen de una base de datos interna que tiene la plataforma que reúne un gran número de nombres en español. Eso permitirá que las opciones de identificación de las frases introducidas por el usuario sean mayores, ya que el chatbot tanto va a entender:

Buenos días, mi nombre es Ana! cómo *Hola, mi nombre es María.*

Esto se puede ver indicado en la siguiente captura del apartado de *Action and parameters*:

The screenshot shows the Dialogflow console interface. On the left is a sidebar with navigation options: Intents, Entities, Fulfillment, Integrations, Training, History, and Analytics. The main area is titled 'Default Welcome Intent' and contains a 'SAVE' button. Below the title is the 'Action and parameters' section, which includes a text input field with 'input.welcome' and a table of parameters.

REQUIRED	PARAMETER NAME	ENTITY	VALUE	IS LIST
<input type="checkbox"/>	Saludos	@Saludos	\$Saludos	<input type="checkbox"/>
<input type="checkbox"/>	given-name	@sys.given-name	\$given-name	<input type="checkbox"/>
<input type="checkbox"/>	Enter name	Enter entity	Enter value	<input type="checkbox"/>

Below the table is a '+ New parameter' link.

Ilustración 13: DIALOGFLOW – Default Welcome Intent, Parameters

Finalmente, en el apartado *Responses* añadimos algunas frases que servirán de respuesta por parte del chatbot al usuario cuando este empieza la conversación de las formas descritas en el apartado anterior.

The screenshot shows the Dialogflow console interface for the 'Default Welcome Intent'. The main area is titled 'Responses' and contains a 'SAVE' button. Below the title is the 'Responses' section, which includes a 'DEFAULT' tab and a list of text responses.

Text response

- ¡Hola! En que puedo ayudarte?
- ¡Hey! En que puedo ayudarte?
- ¡Buenos días! En que puedo ayudarte?
- \$Saludos \$given-name ! En que puedo ayudarte?
- Enter a text response variant

Below the list is an 'ADD RESPONSES' button and a toggle switch for 'Set this intent as end of conversation'.

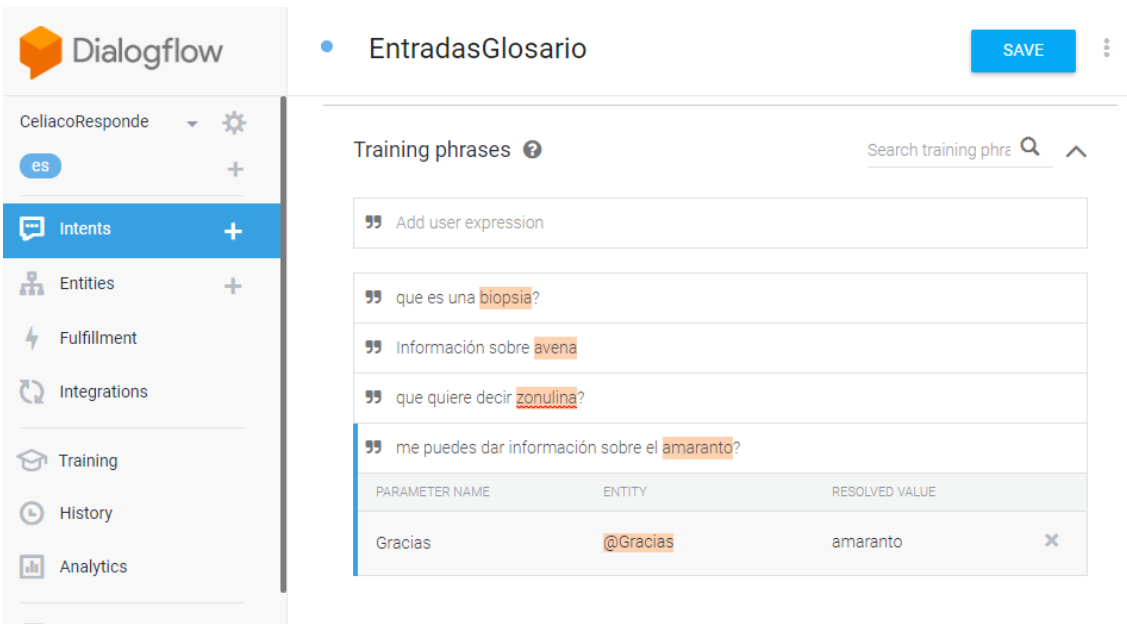
Ilustración 14: DIALOGFLOW – Default Welcome Intent, Responses

El siguiente *Intent* que vamos a generar corresponde al principal de este chatbot, es el que se encargará de activar el *Webhook* y recuperar la información deseada del *CEC*. Para este *Intent* vamos a considerar dos procedimientos a seguir.

En esta parte de la conversación el usuario nos especificará la palabra de la cual quiere saber su significado, y necesitaremos averiguar si esta palabra se encuentra dentro de nuestro contenido en el *CEC* o no para saber si podemos devolver la información solicitada.

La primera opción consiste en solucionar este concepto en el propio chatbot y la segunda hacerlo en el código de la aplicación *Node.js*.

Detección palabra dentro del glosario en *DialogFlow*



PARAMETER NAME	ENTITY	RESOLVED VALUE
Gracias	@Gracias	amaranto

Ilustración 15: DIALOGFLOW – EntradasGlosario, Training phrases A

Para esta opción remarcamos las palabras que forman parte de la *Entity* de Entradas, ya que nos permitirá, cómo se ve en la siguiente captura, marcar obligatoriedad en que las frases siempre deben tener una palabra de esa *Entity*, en caso contrario el chatbot responderá con la frase que se incluye como *Prompt*. Por ejemplo, “Esta palabra no forma parte del glosario”.

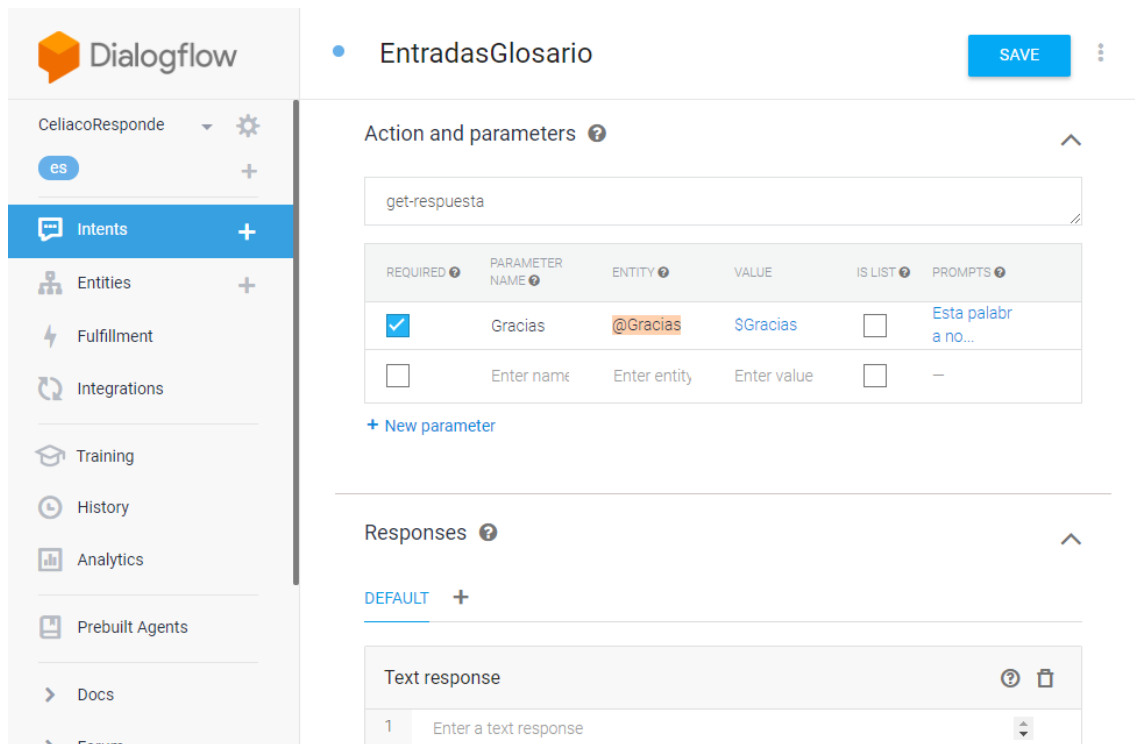


Ilustración 16: DIALOGFLOW – EntradasGlosario, Parameters A

Detección palabra dentro del glosario en Node.js

Esta segunda opción consiste en no utilizar la lista de Entradas que se conforma cómo *Entity* y por lo tanto no remarcar ninguna palabra en *Training phrases*, de esta forma siempre se activará el *Webhook* i será la aplicación *Node.js* la que se encargará de resolver la duda.

Si comparamos estas dos opciones, cada una aporta cosas buenas o malas, para el primer caso el proceso de detección es más fácil de realizar pero implicará que la lista de palabras incluidas en la *Entity* coincida con la lista de palabras incluidas en el *CEC* para que no nos salte ningún error; la segunda opción, el mayor problema que nos puede dar es que el chatbot no reconozca la frase del usuario ya que la palabra clave podrá variar de la usada en el *training*, para solventar este tema será muy importante entrenar correctamente el chatbot en la pestaña de *Training* también.

Otra opción sería, mezclar la dos anteriores, nos permitiría garantizar al máximo la búsqueda correcta, pero podría pasar que la palabra requerida estuviera en el *CEC* y no referida en la *Entity* y por lo tanto que el chatbot respondiera con un

“no reconoce esta palabra”, cuando el *CEC* sí que tendría la información necesaria para responder.

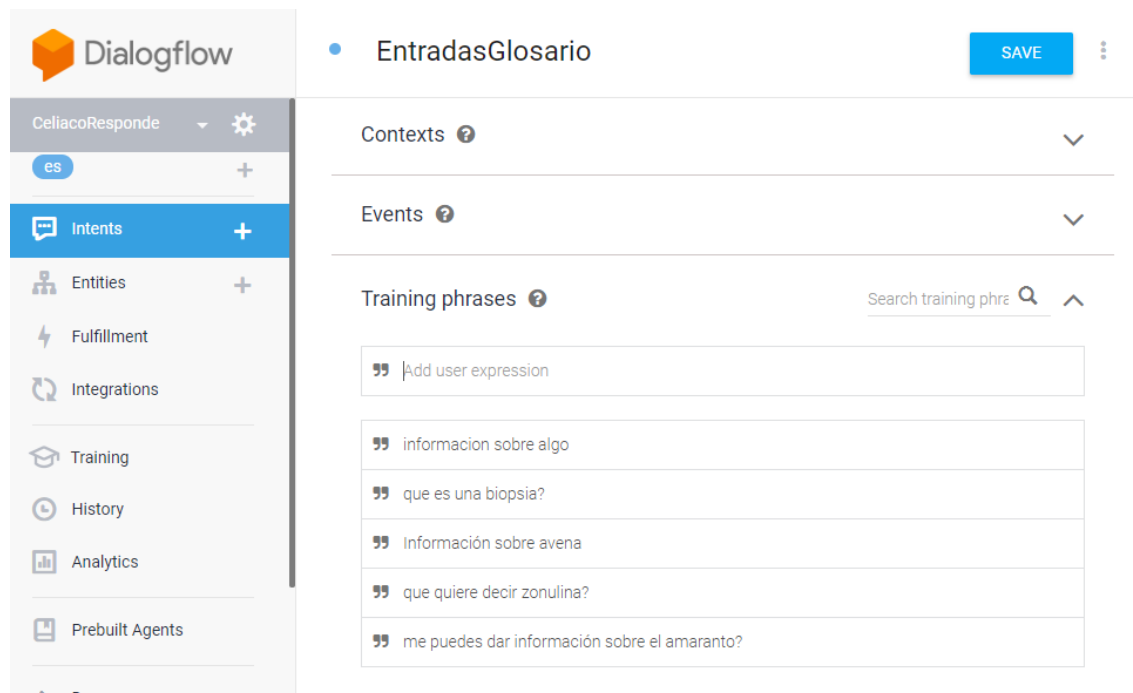


Ilustración 17: DIALOGFLOW – EntradasGlosario, Training phrases B

Para la segunda opción, el apartado de *Action and parameters* no tendrá ninguna información.

Finalmente, sea cual sea la opción escogida, no se van a utilizar *Responses* ya que va a ser el *Webhook* quién devuelva la respuesta a nuestro usuario. Para acabar la creación de este *Intent* nos falta activar la opción de *Webhook* que aparece en la zona de *Fulfillment* cómo se ve en la siguiente captura.

Esto nos permitirá ir a la pestaña de *Fulfillment* del menú principal, para incluir los datos necesarios para que cuando durante una conversación se produzca una de las situaciones entrenadas del *Intent* este llame a la aplicación *Node.js* que deberá responder con el contenido adecuado del *CEC*.

En la pestaña de *Fulfillment* simplemente deberemos indicar, la *URL* que llama a nuestra aplicación ya subida en *Heroku*, todo este procedimiento se explica más adelante en la sección final de desarrollo del proyecto.

Dialogflow

CeliacoResponde

es

Intents

Entities

Fulfillment

Integrations

Training

History

Analytics

Prebuilt Agents

Docs

EntradasGlosario

SAVE

DEFAULT

Text response

1 Enter a text response

ADD RESPONSES

Set this intent as end of conversation

Fulfillment

Enable webhook call for this intent

Enable webhook call for slot filling

Ilustración 18: DIALOGFLOW – EntradasGlosario, Responses y Fulfillment

Dialogflow

CeliacoResponde

es

Intents

Entities

Fulfillment

Integrations

Training

History

Analytics

Prebuilt Agents

Fulfillment

Webhook

ENABLED

Your web service will receive a POST request from Dialogflow in the form of the response to a user query matched by intents with webhook enabled. Be sure that your web service meets all the [webhook requirements](#) specific to the API version enabled in this agent.

Webhook example

URL* <https://xatbotnode.herokuapp.com/conexionCEC>

BASIC AUTH Enter username Enter password

HEADERS Enter key Enter value

Enter key Enter value

+ Add header

DOMAINS Disable webhook for all domains

Ilustración 19: DIALOGFLOW - Webhook

El último *Intent* que vamos a crear es básico como el primero de *Saludos*, los pasos seguidos se muestran a continuación:

Dialogflow

CeliacoResponde

es

Intents

Entities

Fulfillment

Integrations

Training

History

Analytics

RespuestaGracias

SAVE

Contexts

Events

Training phrases

Search training phrase

Add user expression

merci por tu ayuda

Gracias

Ilustración 20: DIALOGFLOW – RespuestaGracias, Training phrases

Dialogflow

CeliacoResponde

es

Intents

Entities

Fulfillment

Integrations

Training

History

Analytics

Prebuilt Agents

Docs

RespuestaGracias

SAVE

Enter entity name

REQUIRED	PARAMETER NAME	ENTITY	VALUE	IS LIST
<input type="checkbox"/>	Gracias	@Gracias	\$Gracias	<input type="checkbox"/>
<input type="checkbox"/>	Enter name	Enter entity	Enter value	<input type="checkbox"/>

+ New parameter

Responses

DEFAULT

Text response

1 De nada ! Si quieres saber algo más pregúntame y sino espero que volvamos ha hablar pronto !

2 Enter a text response variant

Ilustración 21: DIALOGFLOW – Parameters y Responses

4.1.4 Integración

El último paso consiste en decidir cómo vamos a incluir el chatbot en nuestra aplicación, cómo lo que vamos a crear es un site, escogemos la integración web en la pestaña *Integrations*, cuando activemos esta opción nos aparecerá un código de ejemplo que vamos a usar más adelante en el desarrollo del site en el CEC.

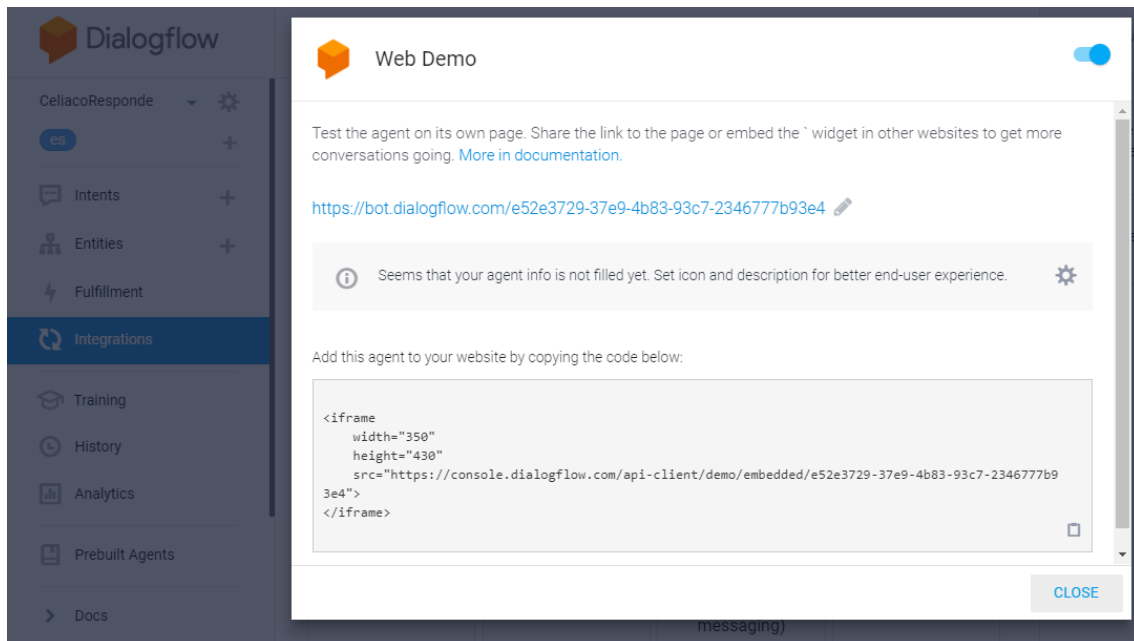


Ilustración 22: DIALOGFLOW – Integrations, Web Demo

Todo el contenido creado para realizar esta parte del proyecto se puede encontrar en la carpeta del proyecto dentro de Docs/DIALOGFLOW.

4.2 Content Experience Cloud

En las siguientes capturas se muestra el aspecto de la plataforma, la primera imagen es una captura de la sección de *Sign In*, la segunda es de la parte de gestión de contenido y la tercera es de la sección de experiencia, dónde se puede interactuar con los *sites* y las plantillas.

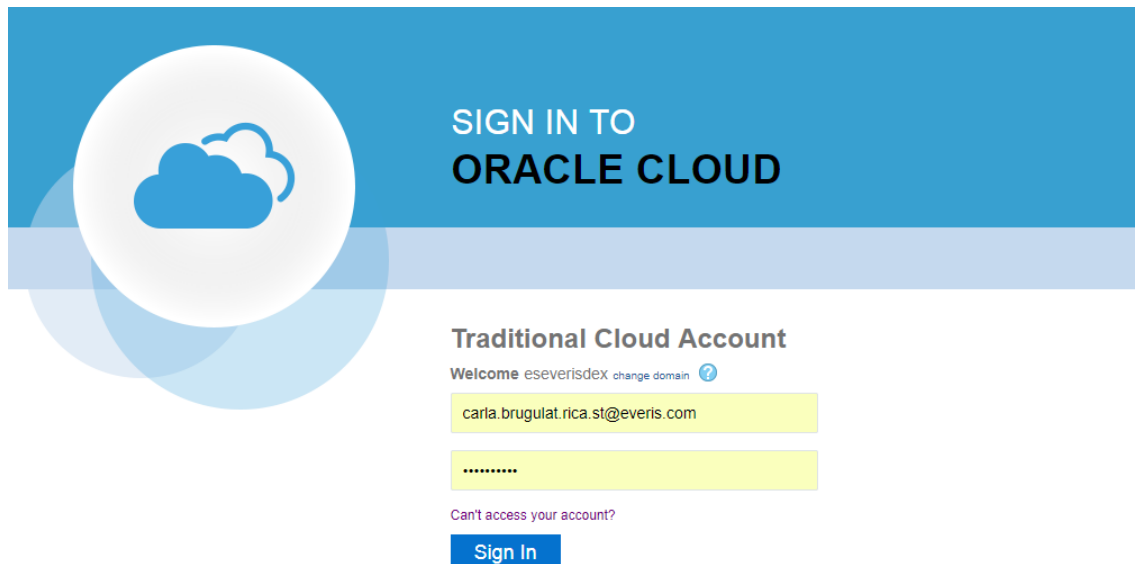


Ilustración 23: CEC – Sign in to Oracle Cloud



Ilustración 24: CEC - Content

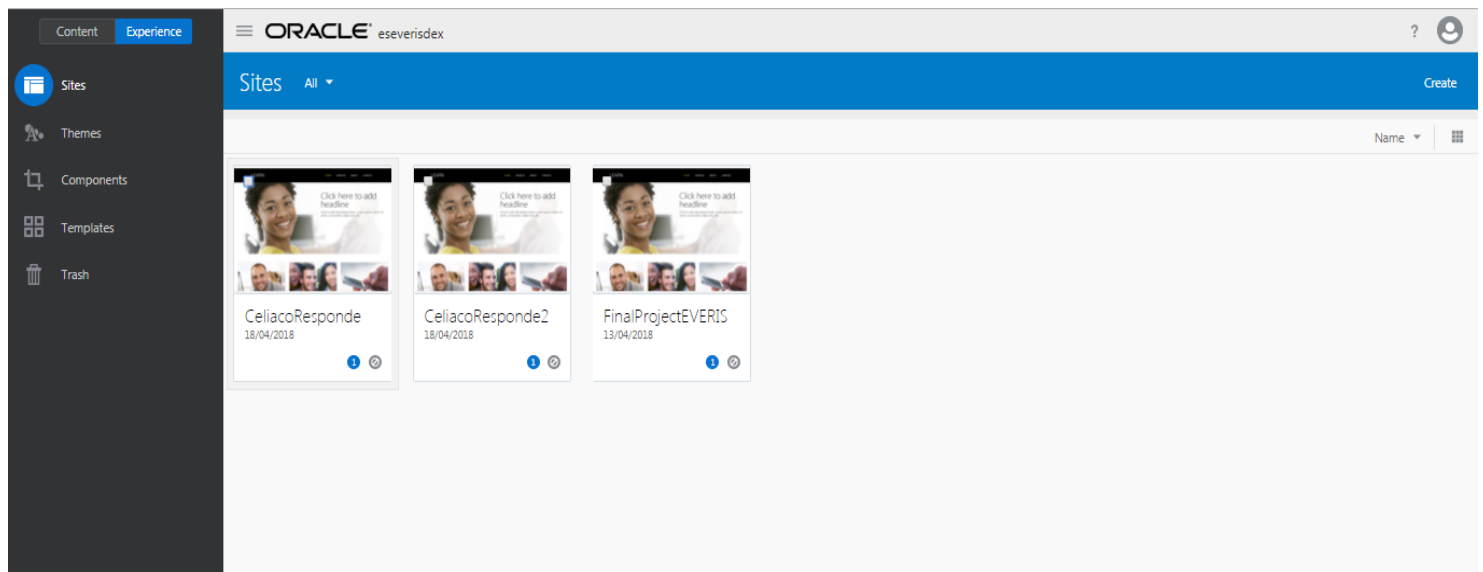


Ilustración 25: CEC - Experience

4.2.1 Crear Site

En este apartado se exponen los pasos seguidos para crear un *Site* en el CEC. Para poder entrar al CEC debemos disponer de un dominio, un usuario y una contraseña que se nos será proporcionado por nuestro Administrador.

Los pasos a seguir para crear un site son los siguientes:

1. Ir a la pestaña de “*Experience*”.
2. Acceder a la sección de “*Sites*”.
3. Pulsar en “*Create*”, para poder realizar esta opción es necesario disponer de alguna template.
4. Rellenar los campos del nuevo *Site*, sólo el nombre es obligatorio, la descripción es opcional y pulsar “*Create*”.

Para poder modificar un *Site*, se debe seleccionar el elegido y pulsar “*Edit*”; si es la primera vez que editas el site tendrás que escoger un nombre y una posible descripción para esta modificación. En caso contrario, simplemente escoge la modificación que quieres seguir editando.

Dentro del *site* necesitaremos tener un componente que me renderize el chatbot en el cual el usuario va a preguntar por sus palabras clave del glosario, la teoría para realizar este componente se muestra a continuación:

Crear un componente “customizado” que integre el chatbot

En esta segunda etapa queremos crear un componente que me permita visualizar el chatbot creado en el *DialogFlow*, para ello debemos crear lo que se denomina un “*HTML Component*”.

Para crearlo seguimos los siguientes pasos:

1. Crear un nuevo *Local Component*

- Ir a la pestaña de “*Experience*”
- Entrar en “*Sites*”
- Pulsar en “Create” seguido de “*Create Local Component*”
- Introducir un nombre (requerido) y una descripción (opcional).

Dentro del componente creado, en la carpeta “*assets*” incluir el archivo “*mustache.min.js*” que se puede descargar de *GitHub* directamente.

Finalmente, en la misma carpeta crear *render.html* con el código *HTML* necesario para tu componente. Si quieres añadirle estilos, en la misma carpeta crear *design.css* con todos los estilos que le quieras aplicar al *HTML Component*.

Finalmente, dentro de la misma carpeta se encuentra el archivo *render.js* cuyo contenido se debe modificar completamente por el que se muestra a continuación, no importa el contenido que haya introducido anteriormente cómo *HTML* o *CSS*, este código siempre es el mismo y no se debe modificar:

```
/* globals define */
define(['jquery', './mustache.min',
'text!./render.html', 'css!./design.css'],
function($, Mustache, template, css) {

    'use strict';

    // -----
    // Create a Mustache-based component implementation
    // -----
    var SampleComponentImpl = function(args) {
```

```

        this.SitesSDK = args.SitesSDK;

        // Initialize the custom component
        this.createTemplate(args);

        this.setupCallbacks();
    };
    // create the template based on the initial values
    SampleComponentImpl.prototype.createTemplate =
function(args) {
    // create a unique ID for the div to add, this
will be passed to the callback
    this.contentId = args.id + '_content_' +
args.viewMode;
    // create a hidden custom component template that
can be added to the DOM
    this.template = '<div id="' + this.contentid +
'">' +
        template +
        '</div>';
    };
    SampleComponentImpl.prototype.updateSettings =
function(settings) {
    if (settings.property === 'customSettingsData') {
        this.update(settings.value);
    }
    };
    SampleComponentImpl.prototype.update =
function(data) {
        this.data = data;

this.container.html(Mustache.to_html(this.template,
this.data));
    };
    //
    // SDK Callbacks

```

```

    // setup the callbacks expected by the SDK API
    //
    SampleComponentImpl.prototype.setupCallbacks =
function() {
    //
    // callback - render: add the component into the
page
    //
    this.render = $.proxy(function(container) {
        this.container = $(container);

this.SitesSDK.getProperty('customSettingsData',
$.proxy(this.update, this));
        }, this);
    //
    // callback - SETTINGS_UPDATED: retrieve new custom
data and re-render the component
    //

this.SitesSDK.subscribe(this.SitesSDK.MESSAGE_TYPES.S
ETTINGS_UPDATED, $.proxy(this.updateSettings, this));
    //
    // callback - dispose: cleanup after component
when it is removed from the page
    //
    this.dispose = $.proxy(function() {
        // nothing required
    }, this);
};
// -----
// Create the factory object for your component
// -----
var sampleComponentFactory = {
    createComponent: function(args, callback) {
        // return a new instance of the component

```



```

        return callback(new SampleComponentImpl(args));
    }
};
return sampleComponentFactory;
});

```

Ahora el componente ya estaría creado y a punto para ser añadido en el *site* a través de la zona de editar.

Para el desarrollo del ejemplo la template que se va a usar para crear el site se llama *Learn_2*, se editará para que tenga un aspecto parecido al de las siguientes imágenes:

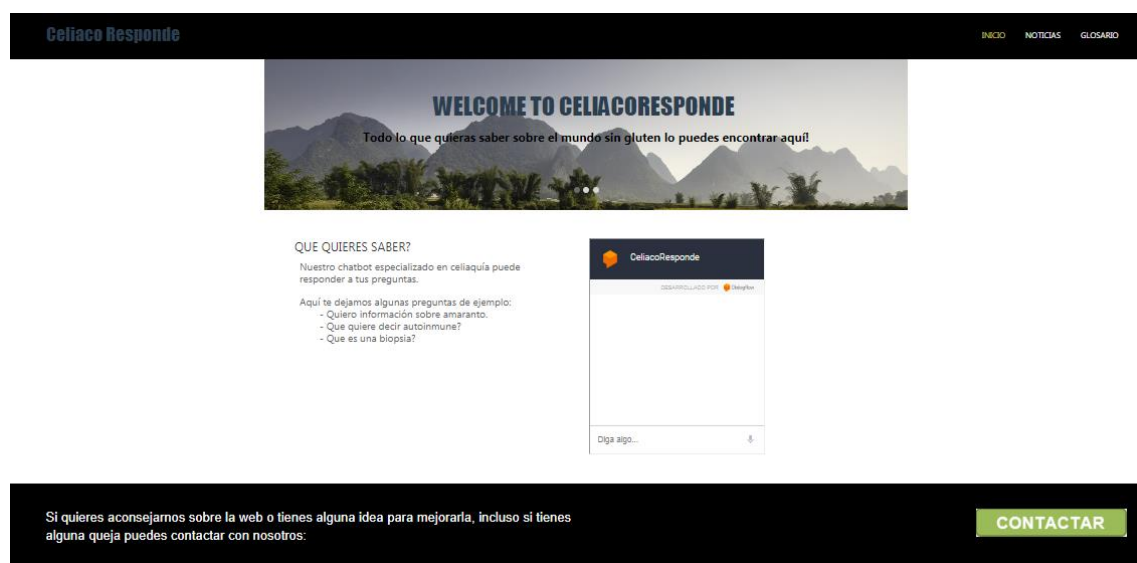


Ilustración 26: CEC – Site Página de Inicio



Ilustración 27: CEC – Site Página de Noticias

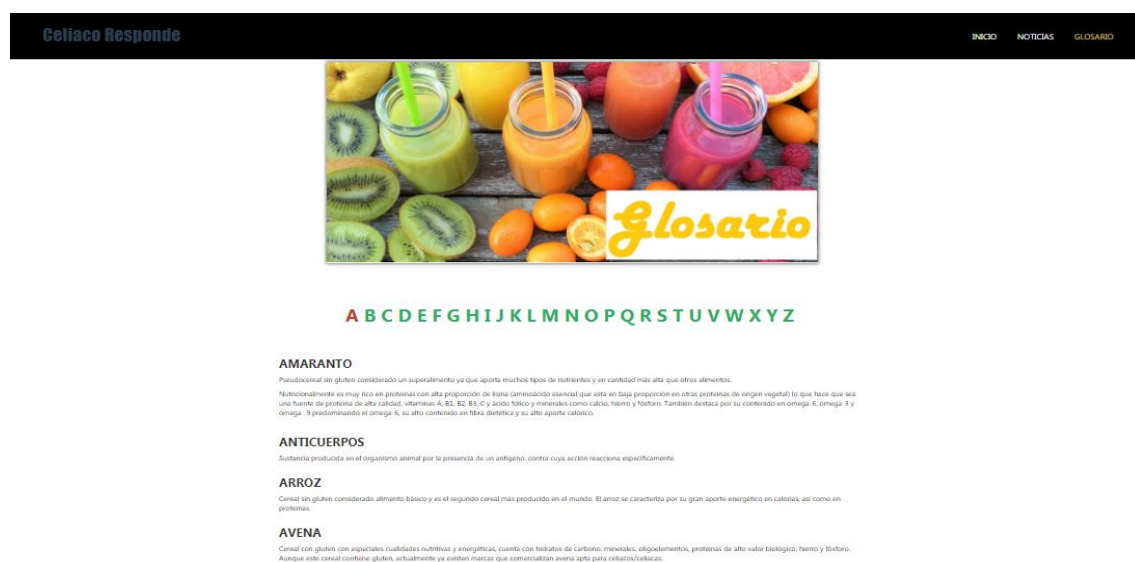
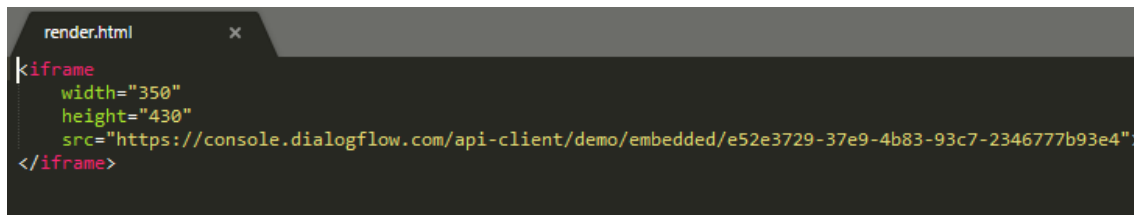


Ilustración 28: CEC – Página de Glosario

Todas las modificaciones que se aprecian en el site son meramente de aspecto, no tienen ninguna funcionalidad, para modificar el aspecto se usa el menú lateral izquierdo de la zona de edición de *sites*.

Cómo se aprecia en la captura del Inicio, se ha incluido un *frame* con el chatbot que hemos creado anteriormente en *DialogFlow*, para añadir este elemento es necesario crear un *Componente* en *HTML* siguiendo los pasos que se indicaron antes.

El código del chatbot que se debe introducir en *render.html* es el mismo que nos daba de ejemplo la propia aplicación de *DialogFlow* en el apartado de integración. Para este ejemplo no necesitaremos crear un archivo para código CSS ya que no necesitamos modificar el aspecto del componente.



```
render.html
<iframe
  width="350"
  height="430"
  src="https://console.dialogflow.com/api-client/demo/embedded/e52e3729-37e9-4b83-93c7-2346777b93e4">
</iframe>
```

Ilustración 29: CEC – Componente HTML, contenido archivo *render.html*

Para estar seguro de que se ha realizado el componente correctamente se pueden realizar dos comprobaciones; comprobar que en la carpeta *assets* se incluyen los siguientes archivos:

- *Mustache.min.js*
- *Render.html*
- *Render.js*
- *Settings.html*

Y la segunda es añadir el contenido al *site* creado y ver si el resultado final es el esperado.

4.2.2 Definir una estructura de datos

El siguiente objetivo de este apartado es crear una estructura de datos, es decir un *content type*, para poder hacerlo es necesario ser Administrador de CEC. Una vez comprobado que eres Administrador puedes proseguir con la creación de contenido.

Para crear un *content type* se deben seguir los siguientes pasos:

1. Ir a la pestaña de “*Content*”
2. Entrar en “*Content Items*”
3. Sólo si eres Administrador te aparecerá la opción *Manage Types*, púlsala.

4. Pulsar “*Create*” y escoger un nombre, una posible descripción y un icono para el *content type*.
5. Añadir campos al *content type*, para cada uno se requiere un nombre y saber si es un campo obligatorio de rellenar o no.

Ahora puedes empezar a añadir contenido al *content type*, cada contenido se denomina *content item*. A continuación, los pasos para crear un *content item*:

1. Ir a la pestaña de “*Content*”
2. Entrar en “*Content Items*”
3. Pulsar en “*Create Item*”.
4. Escoger nombre, descripción opcional y una colección*.

(*)Cosas a tener en cuenta sobre la elección de la colección:

Es un campo obligatorio que no podrá ser modificado una vez se ha creado el *content item*.

Si la colección escogida está asociada a un *site*, debes tener en cuenta que ese *site* dejara de ser exportado como *template* ya que estará relacionado con un *content item* y los *sites* que contienen *content items* o *digital assets* no pueden ser exportados.

Una vez creado debemos añadirle contenido, es decir, rellenar los campos que tiene según el *content type* al que va asociado.

Para el desarrollo del ejemplo vamos a crear un *Content Type* denominado EntradaGlosario y le vamos a añadir dos definiciones, un *Text: nombreentrada* y un *Large Text: textoentrada*.

Para completar el *content type* le vamos a añadir un conjunto de *content items*, que serán las palabras glosario que los usuarios del *site* podrán requerir su contenido.

Cada *content item* se debe asociar a una *Collection*, por eso vamos a crear primero una colección denominada EntradasGlosario, esta colección no estará ligada a ningún *site* y de esta manera garantizamos que el *site* sea exportable a *template* en todo momento.

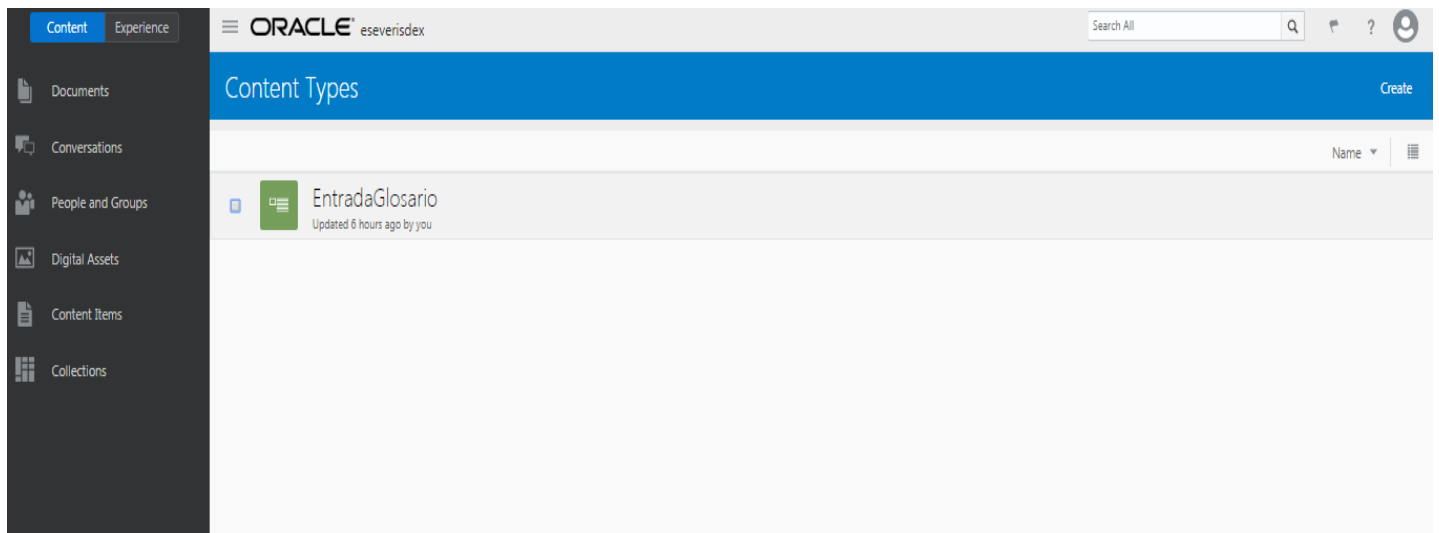


Ilustración 32: CEC – Content Type, EntradaGlosario

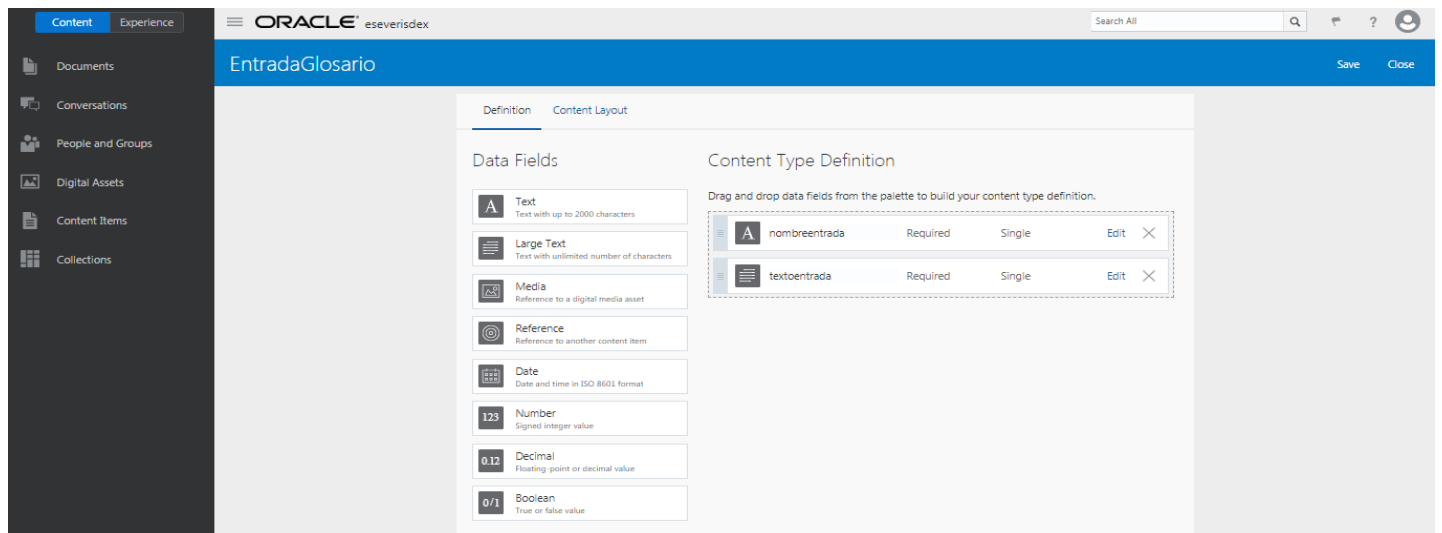


Ilustración 31: CEC – Content Type, Data Fields

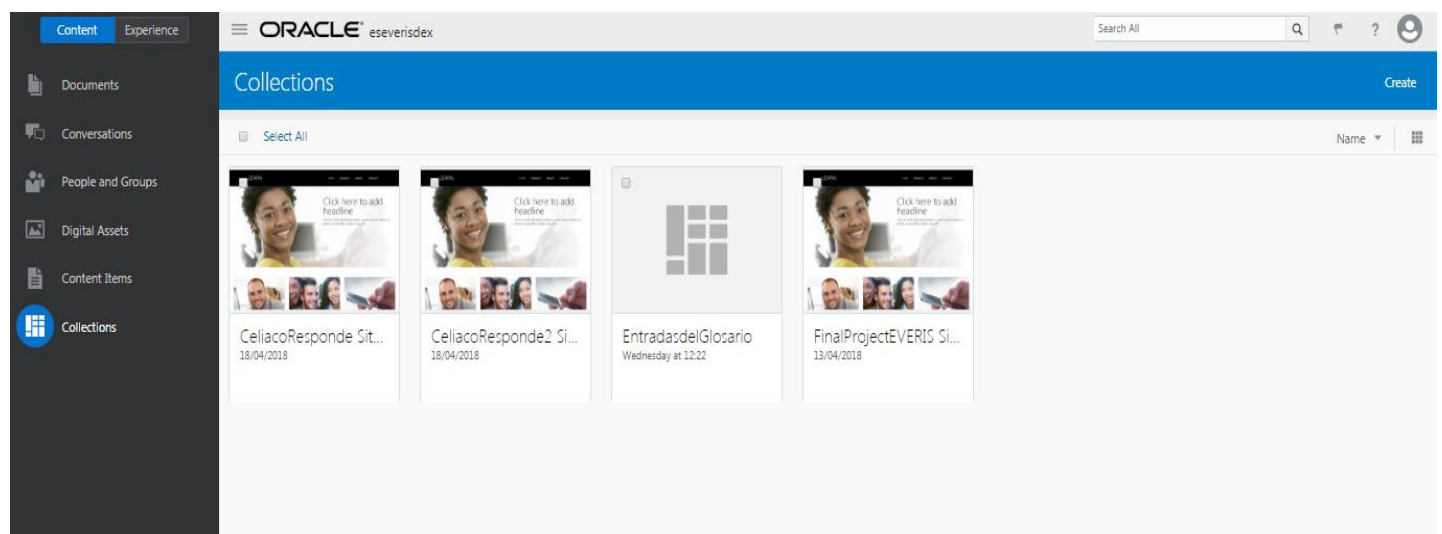


Ilustración 30: CEC - Collection EntradasdelGlosario

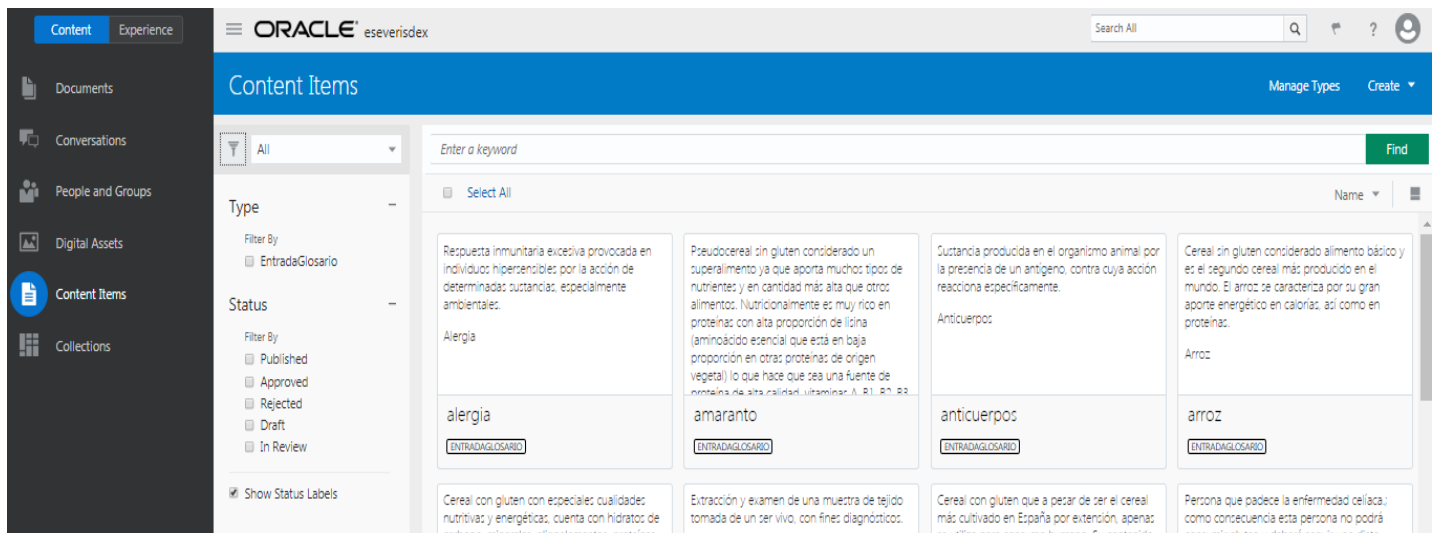


Ilustración 33: CEC – Content Items

Todo el contenido creado para realizar esta parte del proyecto se puede encontrar en la carpeta del proyecto dentro de Docs/ CEC.

4.3 Node.js

El siguiente paso consiste en crear la aplicación *Node.js* que va a dar funcionalidad al *Webhook*, para ello vamos a separar este apartado en dos secciones; la primera sección es para el caso más fácil en el que la aplicación solamente se comunica con el chatbot y el segundo caso dónde ya hay también comunicación con el *CEC*.

En este apartado la teoría se va entrelazar con el ejemplo para que así todo se entienda mejor.

4.3.1 Crear aplicación de Node.js que se conecte con el chatbot

Vamos a crear una aplicación muy simple que solamente nos devuelva el mismo valor que tiene la palabra clave que el usuario del chatbot pide información.

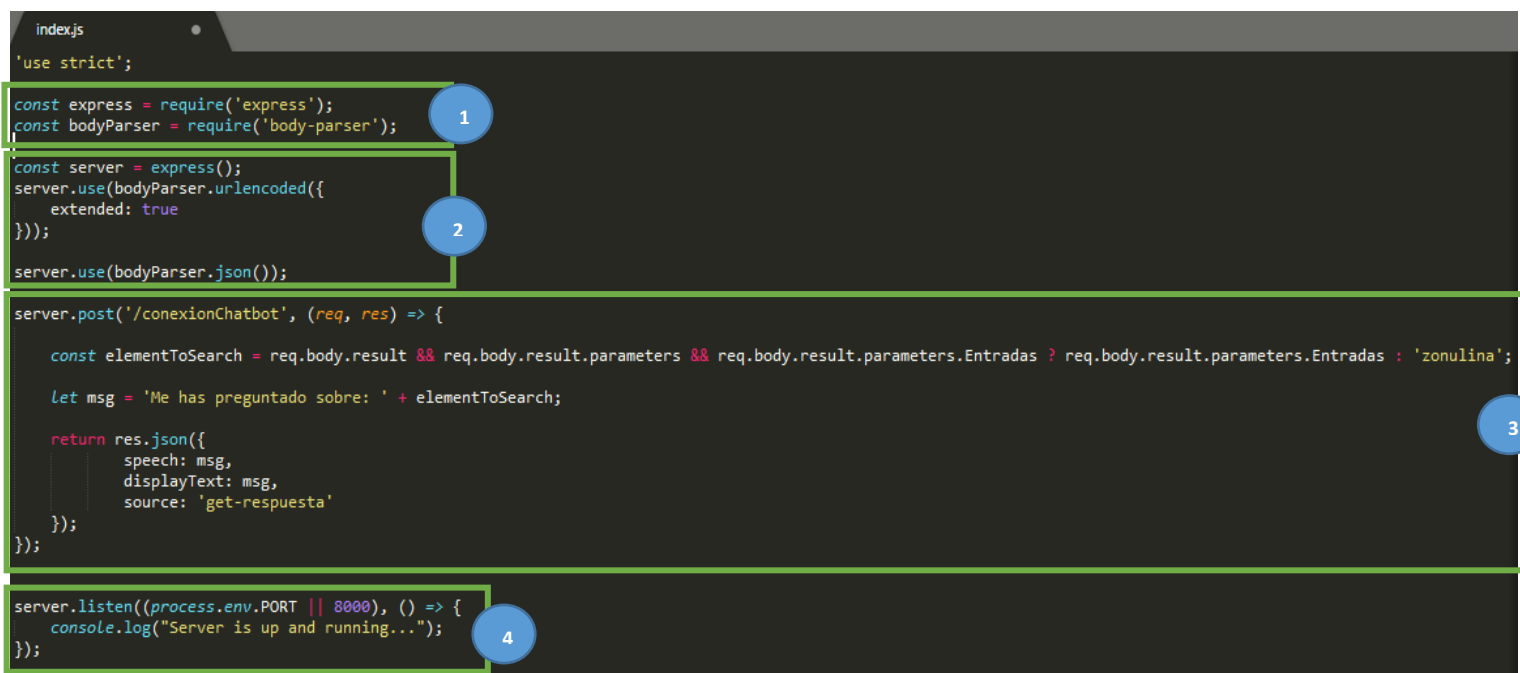
El primer paso será instalar *Node.js* a nuestro ordenador. Seguidamente, abriremos la consola de comandos e iremos a la ruta del proyecto para poder inicializar *Node.js* en la carpeta.

El comando que inicializa un nuevo módulo de *Node.js* es: *npm init*. Esto va a generar un documento en la carpeta del proyecto denominado: *package.json*.

A continuación vamos a instalar un seguido de paquetes que vamos a necesitar para el desarrollo de nuestra aplicación: *npm install express body-parser*.

Si comprobamos la carpeta del proyecto veremos que el archivo *package.json* incluye ahora las dependencias de *express* y *body-parser* y que en la carpeta hay una carpeta nueva denominada: *node_modules*.

La aplicación la vamos a desarrollar en un archivo que vamos a llamar *index.js*:



```
index.js
'use strict';

const express = require('express');
const bodyParser = require('body-parser');

const server = express();
server.use(bodyParser.urlencoded({
  extended: true
}));
server.use(bodyParser.json());

server.post('/conexionChatbot', (req, res) => {

  const elementToSearch = req.body.result && req.body.result.parameters && req.body.result.parameters.Entradas ? req.body.result.parameters.Entradas : 'zonulina';

  let msg = 'Me has preguntado sobre: ' + elementToSearch;

  return res.json({
    speech: msg,
    displayText: msg,
    source: 'get-respuesta'
  });
});

server.listen((process.env.PORT || 8000), () => {
  console.log("Server is up and running...");
});
```

Ilustración 34: NODE.JS – index.js, código de la aplicación

En la sección 1 se importan las dependencias a paquetes necesarios para desarrollar la aplicación:

- *Express*: es el *framework* web más popular de *Node.js*, y es la librería subyacente para un gran número de otros *frameworks* web de *Node.js* populares. En este proyecto nos interesa porque proporciona escritura de manejadores de peticiones con diferentes verbos *HTTP* en diferentes caminos *URL* (rutas).
- *Body Parser*: se necesita para el correcto funcionamiento de *express* al realizar el *http post request*. Este módulo “parsea” el *JSON*, *buffer*,

string y los datos cifrados de la *URL* que se pasan con la *HTTP POST request*.

Seguidamente, en la sección dos, se pone el módulo *Express* en funcionamiento para poder hacer un *POST HTTP* en la sección tres y así poder empezar con la recuperación de contenido del chatbot.

La sección 3 conforma la principal funcionalidad de la aplicación, el post nos permite responder a el trozo de conversación del chatbot que ha activado el *Webhook* y por lo tanto la aplicación que estamos creando en *Node.js*. Este apartado se desarrolla considerando la opción A en la creación del *Intent* EntradasGlosario.

El código de la conversación proveniente del chatbot seguirá la siguiente estructura (el lenguaje de programación es JSON):

```
"result": {
  "source": "agent",
  "resolvedQuery": "que es una biopsia?",
  "action": "",
  "actionIncomplete": false,
  "parameters": {
    "Entradas": "biopsia"
  },
  "contexts": [],
  "metadata": {
    "webhookResponseTime": 1361,
    "intentName": "EntradasGlosario",
    "intentId": "f59b3164-6deb-48c0-b1af-27f900f96561",
    "webhookUsed": "true",
    "webhookForSlotFillingUsed": "false"
  }
}
```

Ilustración 35: NODE.JS – Conversación que ejecuta la aplicación con entity

Nos interesa recuperar el valor de “biopsia” es decir:

result → parameters → Entradas → biopsia

El valor de *elementToSearch* tiene que acabar siendo biopsia para ello comprobamos que todos los campos anteriores existan y para no recibir errores indicamos que en caso de no especificar ninguna entrada se devuelva el valor de zonulina. Esto lo hacemos, básicamente, porque antes de subir la aplicación a *Heroku* y así poder usarla en *DialogFlow*, vamos a hacer pruebas con *Postman*.

El formato indicado dentro del return es el que necesita el chatbot para ser capaz de entender y pintar la respuesta.

Finalmente, en la sección 4 el servidor se pone en escucha esperando la petición de funcionamiento procedente, en nuestro caso, del chatbot en *DialogFlow*. Para poder subir el código a *Heroku* es muy importante que este paso se haga siguiendo el mismo formato.

4.4.3 Crear aplicación de Node.js que se conecte con el chatbot y el CEC

Ahora ya vamos a tratar el ejemplo al completo, vamos a aprovechar la aplicación creada antes con algunos cambios y junto a ella el *cec-restapi* que nos permitirá conectarnos al *CEC* para recuperar el contenido deseado.

El *cec-restapi* es el código proporcionado por *Oracle* que nos habilita la conexión con el *Content Experience Cloud*, este código no debe ser modificado por nosotros prácticamente, solamente se deben actualizar las credenciales que nos permiten acceder a nuestro *CEC*. Los dos archivos que se deben modificar son: *cec-restapi/pam/cec/.env* y *RestApi.js*.

```
export CEC_USER='carla.brugulat.rica.st@everis.com'  
export CEC_PW='1Q2w3e4r5t'  
export CEC_DOMAIN='eseverisindex'  
export CEC_URL='https://contentandexpe-eseverisindex.documents.us2.oraclecloud.com'  
  
export CEC_CONTENT_TYPE='EntradaGlosario'  
export CEC_COLLECTION_ID='FC1DB4CA620BC86A0DCA2896B27389766EFCBE57208E'
```

Ilustración 36: NODE:JS, cec-restapi, archivo .env

```
const context = {
  host: process.env.CEC_URL || "https://contentandexpe-eseverisdex.documents.us2.oraclecloud.com",
  domain: process.env.CEC_DOMAIN || "eseverisdex",
  user: process.env.CEC_USER || "carla.brugulat.rica.st@everis.com",
  password: process.env.CEC_PW || "1Q2w3e4r5t"
}

const config = {
  contentType: process.env.CEC_CONTENT_TYPE,
  collectionId: process.env.CEC_COLLECTION_ID
}
```

Ilustración 37: NODE:JS, cec-restapi, archivo .env

Ahora vamos a modificar la aplicación inicial creada para que pueda usarse con el cec-restapi:



```
index.js • JS cec_connector.js
1 'use strict';
2
3 const express = require('express');
4 const bodyParser = require('body-parser');
5
6 //invocar cec-restapi
7 const cec_connection = require('./cec_connector');
8
9 const server = express();
10
11 server.use(bodyParser.urlencoded({
12   extended: true
13 }));
14
15 server.use(bodyParser.json());
16
17 server.post('/conexionCEC', (req, res) => {
18   let elementToSearch;
19
20   if (req.body.parameters === undefined) {
21     elementToSearch = req.body.result.resolvedQuery;
22
23     elementToSearch = elementToSearch.split(" ");
24     const sizeSearch = elementToSearch.length;
25     elementToSearch = elementToSearch[sizeSearch-1];
26
27     elementToSearch = elementToSearch.split("?");
28     elementToSearch = elementToSearch[0];
29   } else {
30     elementToSearch = req.body.result.parameters.Entradas;
31   }
32
33   const item = cec_connection.invokeFunction(elementToSearch)
34     .then(data => {
35       res.json({
36         speech: data,
37         displayText: data,
38         source: 'get-respuesta'
39       });
40     })
41     .catch(err => console.log("ERROR: ", err));
42
43   server.listen((process.env.PORT || 8000), () => {
44     console.log("Server is up and running...");
45   });
46 }
```

Ilustración 38: NODE:JS – segunda aplicación con conexión al cec-restapi

Para esta segunda parte del proyecto la resolución está hecha para la opción B del *Intent* EntradasGlosario, por lo tanto deberemos gestionar dentro de la aplicación la posibilidad de que la palabra que requiere el usuario no forme parte de nuestro glosario.

Ahora el código *json* de la conversación entrante es diferente ya que no hay ninguna *Entity* asociada a las frases que escribe el usuario, su estructura actual será la siguiente:

```
"result": {
  "source": "agent",
  "resolvedQuery": "que es una biopsia?",
  "action": "get-respuesta",
  "actionIncomplete": false,
  "parameters": {},
  "contexts": [],
  "metadata": {
    "intentId": "f59b3164-6deb-48c0-b1af-27f900f96561",
    "webhookUsed": "true",
    "webhookForSlotFillingUsed": "false",
    "webhookResponseTime": 5000,
    "intentName": "EntradasGlosario"
  }
}
```

Ilustración 39: NODE.JS – Conversación que ejecuta la aplicación sin entity

En la captura anterior se puede apreciar que la sección de parámetros está vacía y por lo tanto tendremos que recuperar la información de *resolvedQuery*. Eso es lo que se realiza en la sección 2 remarcada del código de la nueva aplicación.

Después de esto, en la sección 2 se invoca a la función que hace de nodo con el *cec-restapi*, esta función se describe en el archivo: *cec_connector.js*.

Se encargará de hacer la petición de contenido del *CEC* relacionada directamente con el valor de *elementToSearch*, por esta razón este valor se pasa como parámetro de la función.

```
JS index.js • JS cec_connector.js •
1  "use strict"
2
3  const express = require('express');
4  const request = require('request');
5  const urlLib = require('urlLib');
6
7  const invokeFunction = (elementToSearch) => new Promise((resolve, reject) => {
8    urlLib.request(`https://xatbot-node-cec.herokuapp.com/cecAdmin/search`,
9    {
10      method: 'POST',
11      auth: "oracle:welcome1",
12      headers: {
13        "Content-Type": "application/json",
14        "Accept": "application/json"
15      },
16      data: {
17        "type": "EntradaGlosario",
18        "name": elementToSearch
19      }
20    }, function (err, data, res) {
21      if (err) {
22        reject(err);
23        console.log("ERROR---->: ", err);
24      }
25    }
26  });
27
28  } else {
29    var buffer = Buffer.from(new Uint8Array(data));
30
31    var resp = JSON.parse(buffer.toString());
32
33    var id_CEC = resp.body.items[0];
34
35    if(id_CEC===undefined){
36      id_CEC="";
37    }
38
39    id_CEC = id_CEC.id;
40    console.log("id_CEC----->", id_CEC);
41    var url = `https://xatbot-node-cec.herokuapp.com/cecAdmin/wp/${id_CEC}`;
42
43    datos_lookup(url)
44      .then(data => {resolve(data)})
45      .catch(err => console.log("ERROR: ", err));
46  }
47
48  });
49
50  }
51
52  );
```

Ilustración 40: NODE.JS, cec_connector.js 1

En la primera sección se realiza la llamada al *cec-restapi* con el formato requerido, se le pasa el *elementToSearch* ya que es la palabra clave que nos permitirá conseguir el *content item* deseado. La *URL* que se aprecia es la que se asocia al *cec-restapi* una vez modificado y subido a *Heroku*, esto se explica en el último apartado del desarrollo.

En la sección 2 lo primero que comprobamos es si el *content item* que lleva el mismo nombre que el *elementToSearch* realmente existe en el *CEC*, en caso negativo se pasa un *id_CEC* vacío para acabar devolviendo la respuesta correcta al chatbot. En caso de obtener, un *content item* con el nombre coincidente al *elementToSearch* recuperamos su *id* para poder recuperar en la siguiente sección su contenido.

Al invocar *datos_look* lo que vamos a hacer es llamar a la parte del *cec-restapi* que nos permite recuperar el contenido del *content item* que tiene el id referido en la *URL*.

```
49
50 const datos_lookup = (url) => new Promise((resolve, reject) => {
51   urlLib.request(url,
52     {
53       method: 'GET',
54       headers: {
55         'Content-Type': 'application/json',
56         'Accept': 'application/json'
57       },
58     }, function (err, data) {
59       if (err) {
60         reject(err);
61         console.log("ERROR datos_lookup: ", err);
62       } else {
63         var respuesta = JSON.parse(data);
64
65         if(respuesta.body.data ===undefined){
66           respuesta="Esta palabra no se encuentra en nuestro diccionario";
67         } else {
68           var entrada_name = (respuesta.body.data.entradaglosario_nombreentrada).toString();
69           var entrada_descr = (respuesta.body.data.entradaglosario_textoentrada).toString();
70
71           //exportar datos para ser usados en la respuesta a DialogFlow
72           respuesta = "Me has preguntado sobre " + entrada_name + ", esta es la información que tenemos: " + entrada_descr;
73         }
74
75         resolve(respuesta);
76       }
77     });
78   });
79
80 module.exports = {
81   invokeFunction
82 };
83
```

Ilustración 41: NODE.JS, *cec_connector.js* 2

En este caso la segunda invocación en el *cec-restapi* es un *GET* ya que queremos información proveniente del *CEC*. En la sección 3 si el *id_CEC* que hemos enviado era nulo preparamos una respuesta para indicar al usuario que la información sobre la palabra pedida no está en nuestro glosario. En caso de que el *id_CEC* si tuviera un valor se devolverá la información correspondiente a *nombreentrada* y *textoentrada* que hemos visto cómo se creaban unos apartados atrás.

Finalmente en la sección 5, simplemente exportamos la primera función para que pueda utilizarse en el *index.js*. La importación se realiza en la sección 1 remarcada de la ilustración 38.

Todo el contenido creado para realizar esta parte del proyecto se puede encontrar en la carpeta principal del proyecto.

4.5 Postman

En este apartado vamos a comprobar si las dos aplicaciones realizadas en el apartado anterior funcionan en local antes de subirlas a *Heroku*.

Primero vamos a comprobar la aplicación de conexión solamente con el chatbot: para ello abrimos un terminal, vamos a la ruta del proyecto y ejecutamos *node index.js* para poner en funcionamiento el servidor. Seguidamente abrimos *Postman* y ejecutamos la siguiente *URL*:

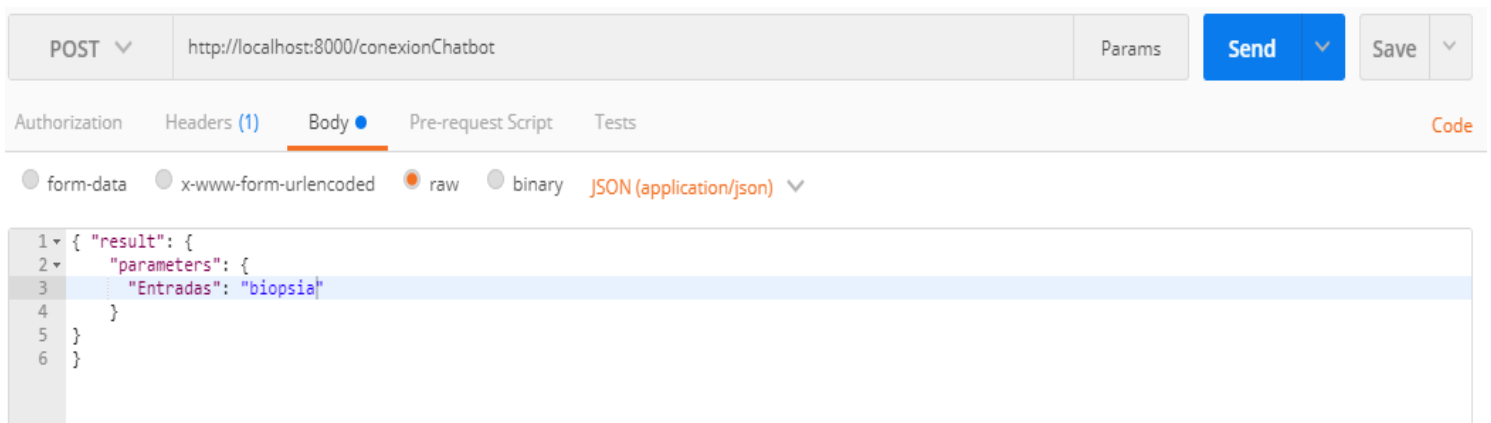


Ilustración 42: POSTMAN – Aplicación 1, pregunta

Al hacer clic en “Send” deberíamos recibir la siguiente respuesta:



Ilustración 43: POSTMAN – Aplicación 1, respuesta

Ahora vamos a comprobar la segunda aplicación, para este caso necesitaremos dos terminales, el primero para el *cec-restapi* y el segundo para la aplicación que conecta al chatbot. En los dos terminales ejecutaremos *node index.js* y seguidamente abriremos *Postman*, en él comprobaremos que la aplicación entera funcione.

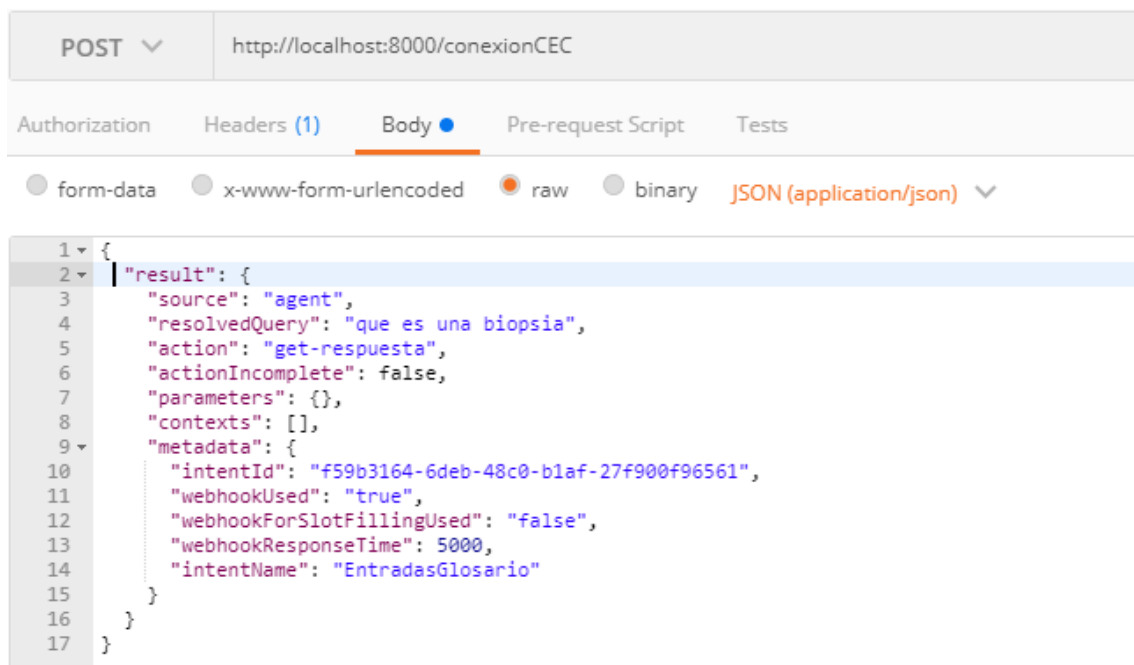


Ilustración 44: POSTMAN – Aplicación 2, pregunta

Al hacer clic en “Send” deberíamos recibir la siguiente respuesta:

```
{
  "speech": "Me has preguntado sobre Biopsia, esta es la información que tenemos: Extracción y examen de una muestra de tejido tomada de un ser vivo, con fines diagnósticos.",
  "displayText": "Me has preguntado sobre Biopsia, esta es la información que tenemos: Extracción y examen de una muestra de tejido tomada de un ser vivo, con fines diagnósticos.",
  "source": "get-respuesta"
}
```

Ilustración 45: POSTMAN – Aplicación 2, respuesta

Ahora que hemos comprobado que la aplicación entera funciona correctamente en nuestro ordenador, vamos a subirla a Heroku y así poder usarla en el site directamente.

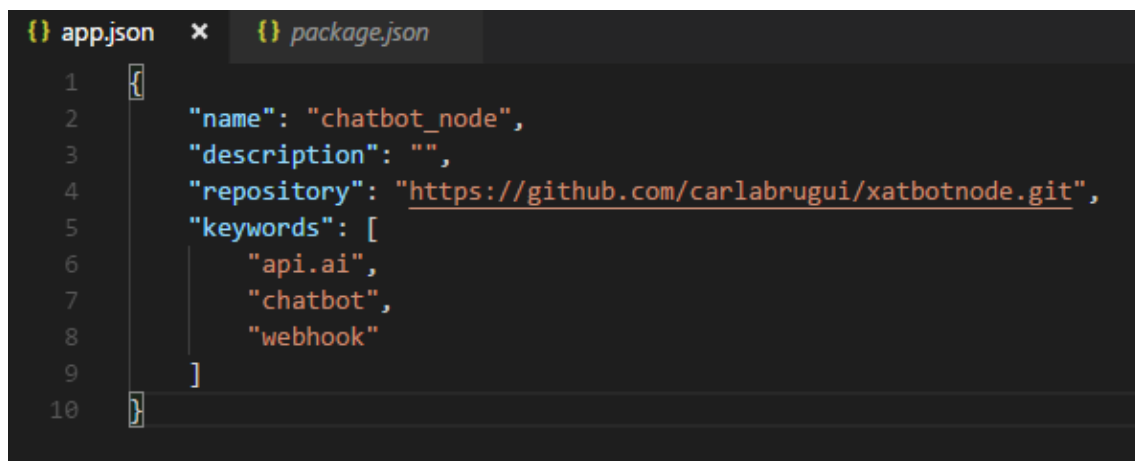
Para hacer estas pruebas en local seguramente se necesitará configurar el móvil o usar una red de internet diferente a la de la empresa.

Todo el contenido creado para realizar esta parte del proyecto se puede encontrar en la carpeta del proyecto dentro de Docs/POSTMAN.

4.6 Heroku

Para subir una aplicación a *Heroku* primero tendremos que hacer algunas modificaciones. Tendremos que hacer todos estos pasos tanto para el *cec-restapi* cómo para la aplicación entera.

1. Crear archivo *app.json* en la carpeta principal del proyecto:

A screenshot of a code editor with two tabs: 'app.json' and 'package.json'. The 'app.json' tab is active, showing a JSON object with the following content:

```
{
  "name": "chatbot_node",
  "description": "",
  "repository": "https://github.com/carlabrugui/xatbotnode.git",
  "keywords": [
    "api.ai",
    "chatbot",
    "webhook"
  ]
}
```

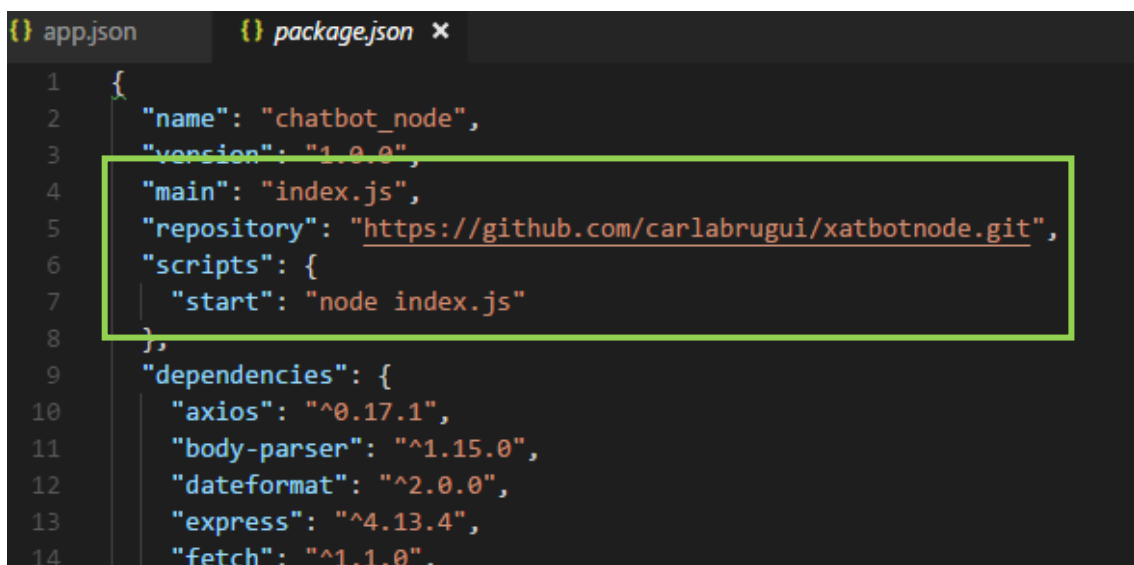
The code is syntax-highlighted, with strings in orange and keywords in blue. Line numbers 1 through 10 are visible on the left side of the editor.

Ilustración 46: HEROKU – Archivo *app.json* aplicación entera


```
{
  "name": "bots-express-service-template",
  "description": "Bots Express-based component service template",
  "repository": "https://github.com/carlabrugui/xatbot-node-cec.git",
  "keywords": [
    "api.ai",
    "chatbot",
    "webhook"
  ]
}
```

Ilustración 47:: HEROKU – Archivo `app.json` aplicación `cec-restapi`

2. Modificar el archivo `package.json`:



```
{ app.json package.json x
1  {
2    "name": "chatbot_node",
3    "version": "1.0.0",
4    "main": "index.js",
5    "repository": "https://github.com/carlabrugui/xatbotnode.git",
6    "scripts": {
7      "start": "node index.js"
8    },
9    "dependencies": {
10     "axios": "^0.17.1",
11     "body-parser": "^1.15.0",
12     "dateformat": "^2.0.0",
13     "express": "^4.13.4",
14     "fetch": "^1.1.0",
```

Ilustración 48: HEROKU – Archivo `package.json` aplicación entera

```
{
  "name": "bots-express-service-template",
  "version": "1.0.0",
  "description": "Bots Express-based component service template",
  "main": "index.js",
  "repository": "https://github.com/carlabrugui/xatbot-node-cec.git",
  "scripts": {
    "start": "node index.js",
  },
  "dependencies": {
```

Ilustración 49: HEROKU – Archivo `package.json` aplicación `cec-restapi`

3. Subir proyectos a *GitHub* con *Git Bash*:

- Crear un repositorio en *GitHub*, tener en cuenta que el nombre debe coincidir con el del archivo `app.json` y `package.json`.
- Abrir *Git Bash*.
- Ir a la carpeta del proyecto: `cd "dominio"`.
- Clonar el repositorio que hemos creado en *GitHub*: `git clone "ruta repositorio"`.
- Ir a la carpeta del repositorio: `cd "dominio repositorio"`.
- Comprobar estado del repositorio: `git status`.
- Añadir los cambios: `git add -A`.
- Reafirmar los cambios: `git commit -m '...'`.
- Subir los cambios a *GitHub*: `git push origin master`.

4. Comprobar que los cambios se muestran en *GitHub*.

5. Crear aplicación en *Heroku*:

- Una vez tenemos una aplicación creada debemos conectarla a *GitHub* para subir el código.
- Finalmente, automatiza las subidas de código y haz un *Deploy Brach*.

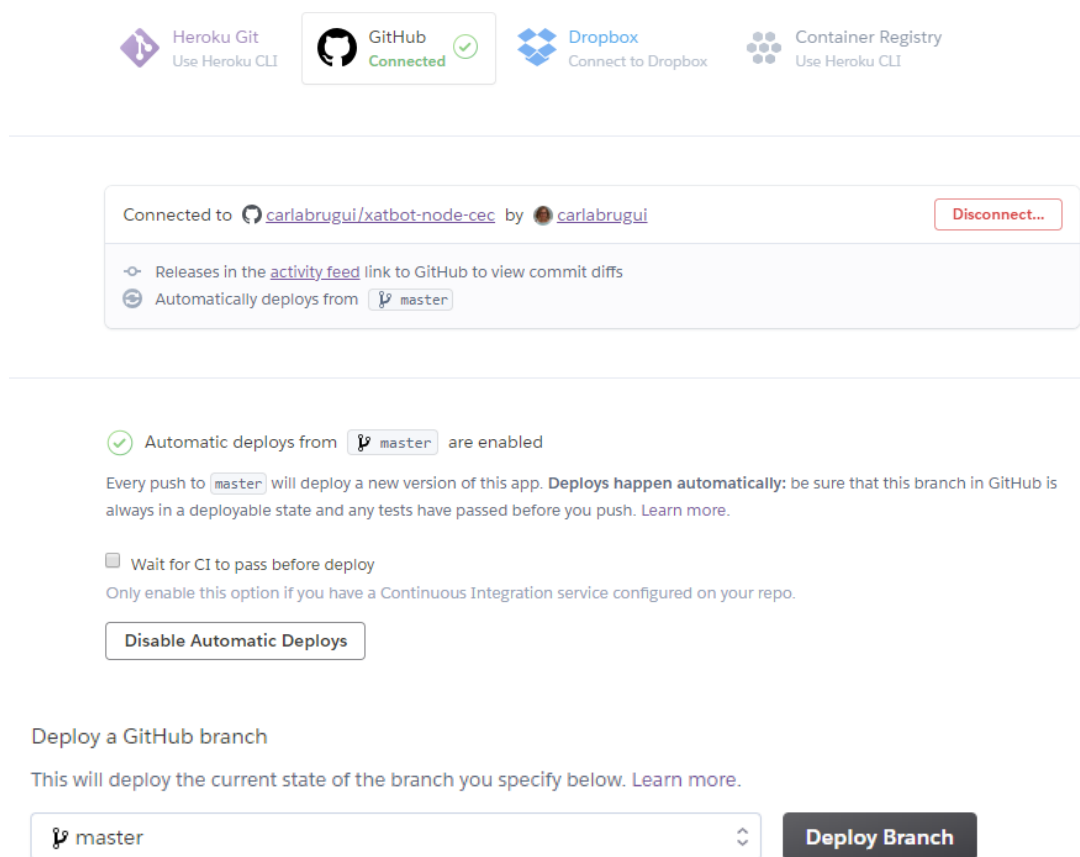


Ilustración 50: HEROKU – Subir código de la aplicación desde GitHub

Ahora si abrimos el site en el CEC (sección editar) ya podremos realizar la conversación deseada en el chatbot:

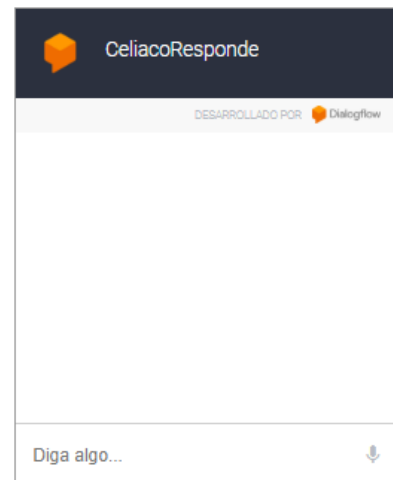
Ejemplo de conversación

QUE QUIERES SABER?

Nuestro chatbot especializado en celiaquía puede responder a tus preguntas.

Aquí te dejamos algunas preguntas de ejemplo:

- Quiero información sobre amaranto.
- Que quiere decir autoinmune?
- Que es una biopsia?

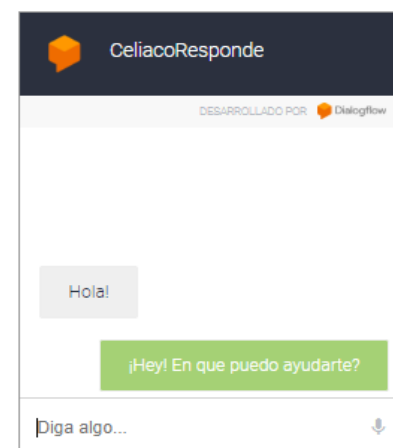


QUE QUIERES SABER?

Nuestro chatbot especializado en celiaquía puede responder a tus preguntas.

Aquí te dejamos algunas preguntas de ejemplo:

- Quiero información sobre amaranto.
- Que quiere decir autoinmune?
- Que es una biopsia?



QUE QUIERES SABER?

Nuestro chatbot especializado en celiaquía puede responder a tus preguntas.

Aquí te dejamos algunas preguntas de ejemplo:

- Quiero información sobre amaranto.
- Que quiere decir autoinmune?
- Que es una biopsia?

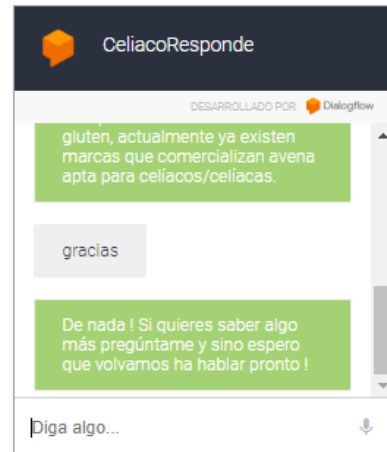


QUE QUIERES SABER?

Nuestro chatbot especializado en celiacía puede responder a tus preguntas.

Aquí te dejamos algunas preguntas de ejemplo:

- Quiero información sobre amaranto.
- Que quiere decir autoinmune?
- Que es una biopsia?

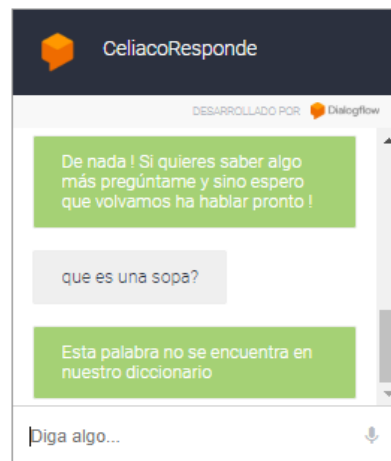


QUE QUIERES SABER?

Nuestro chatbot especializado en celiacía puede responder a tus preguntas.

Aquí te dejamos algunas preguntas de ejemplo:

- Quiero información sobre amaranto.
- Que quiere decir autoinmune?
- Que es una biopsia?



5. Bibliografía

[1] DialogFlow(formerly API.AI): Let's create a Movie ChatBot in minutes →

<https://chatbotslife.com/api-ai-lets-create-a-movie-chatbot-in-minutes-f68d8bb568f9>

[2] How do I create a site? →

<https://docs.oracle.com/en/cloud/paas/content-cloud/user/how-do-i-create-site.html>

[3] How do I edit a site? →

<https://docs.oracle.com/en/cloud/paas/content-cloud/user/how-do-i-edit-site.html>

[4] What is structured content?

<https://docs.oracle.com/en/cloud/paas/content-cloud/user/what-is-structured-content.html>

[5] How do I create structured content? →

<https://docs.oracle.com/en/cloud/paas/content-cloud/user/how-do-i-create-structured-content.html>

[6] How do I create and share content types? →

<https://docs.oracle.com/en/cloud/paas/content-cloud/user/how-do-i-create-content-type.html>

[7] How do I create content items? →

<https://docs.oracle.com/en/cloud/paas/content-cloud/user/how-do-i-create-content-items.html>

[8] Create the HTML Component →

<https://docs.oracle.com/en/cloud/paas/content-cloud/developer/create-html-component.html>

[9] Node.js Tutorial →

<https://www.w3schools.com/nodejs/default.asp>

[10] Creating a Node.js based Webhook for Intelligent Bots →

<https://chatbotslife.com/creating-a-nodejs-based-webhook-for-intelligent-bots-a91ecbe33402>

[11] Let's Create a Live News Chatbot in 10 Minutes (Upload chatbot to Heroku)

→ <https://chatbotsmagazine.com/lets-create-a-live-news-api-for-our-chatfuel-bot-in-minutes-7ed67fc67ae6>