****

**NTT DATA Company**  
Technology - Digital experience

**Celíaco responde**

**Carla Brugulat Rica**

Tabla de contenido

[Tabla de contenido 2](#_Toc514315638)

[1. Introducción 3](#_Toc514315639)

[2. Esquema del proyecto 3](#_Toc514315640)

[3. Objetivos 5](#_Toc514315641)

[4. Desarrollo del proyecto 5](#_Toc514315642)

[4.1 DialogFlow 6](#_Toc514315643)

[4.1.1 Crear Agent 8](#_Toc514315644)

[4.1.2 Crear Entities 9](#_Toc514315645)

[4.1.3 Crear Intents 11](#_Toc514315646)

[4.1.4 Integración 19](#_Toc514315647)

[4.2 Content Experience Cloud 20](#_Toc514315648)

[4.2.1 Crear Site 21](#_Toc514315649)

[4.2.2 Definir una estructura de datos 27](#_Toc514315650)

[4.3 Node.js 30](#_Toc514315651)

[4.3.1 Crear aplicación de Node.js que se conecte con el chatbot 30](#_Toc514315652)

[4.4.3 Crear aplicación de Node.js que se conecte con el chatbot y el CEC 33](#_Toc514315653)

[4.5 Postman 38](#_Toc514315654)

[4.6 Heroku 40](#_Toc514315655)

[5. Bibliografía 46](#_Toc514315656)

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.

1. 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.

**NODE.JS**

Webhook

Cec-restapi

**HEROKU**

**CEC**

**Site**

Chatbot

**Content**

Item 1

Item …

Item 2

Ilustración : 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.

1. 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.

1. 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 celiaquía, 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.

* 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.

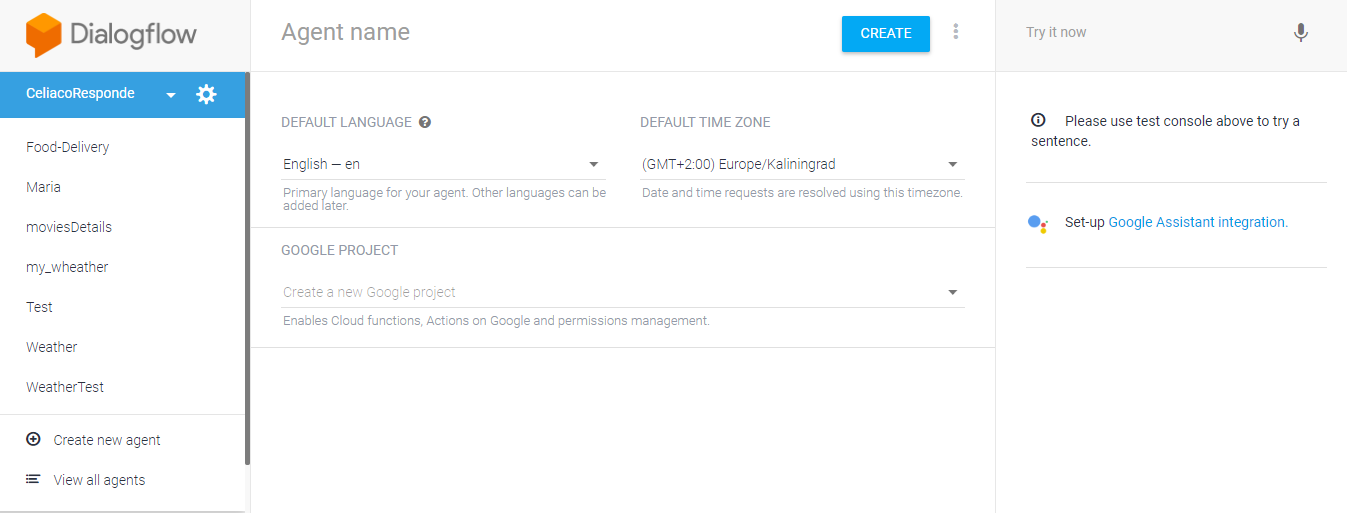
Sabiendo esto podemos empezar con la creación de un chatbot en *DialogFlow*; el primer paso es tener cuenta registrada en el portal o disponer de una cuenta de *Google*. Seguidamente, entrar en la Consola de *DialogFlow*, para poder empezar con la creación del *Agent*: 

Ilustración : 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:

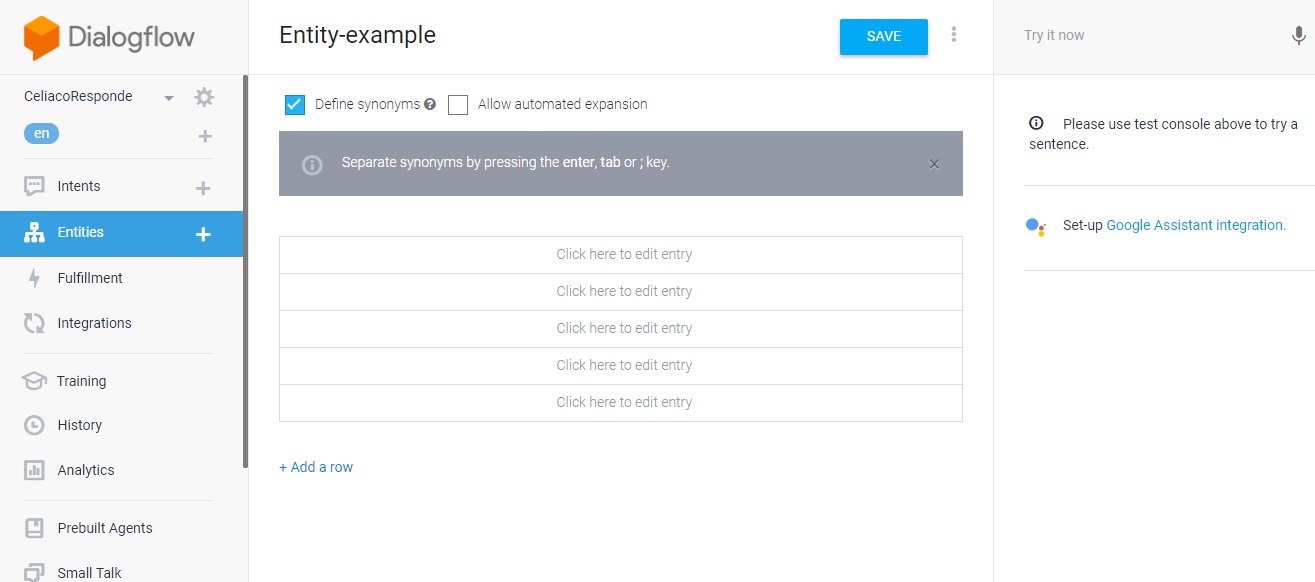


Ilustración : DIALOGFLOW - Pantalla de creación de Entities

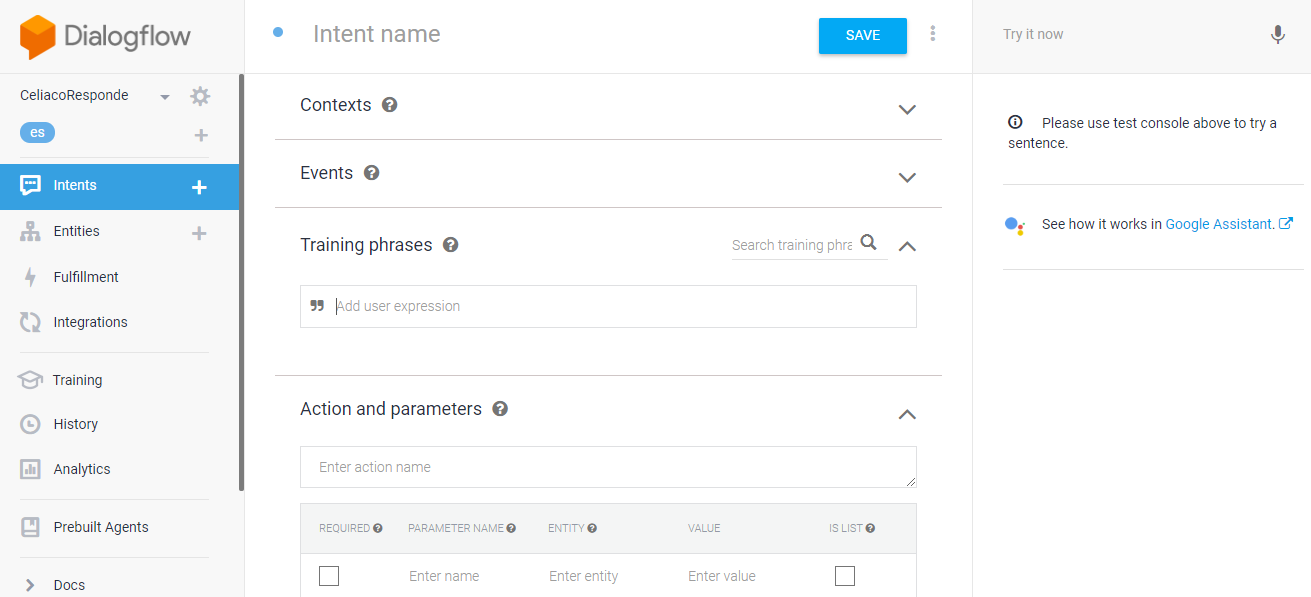


Ilustración : DIALOGFLOW – Pantalla de creación de Intents

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

Ahora se realizan los pasos anteriores con el ejemplo correspondiente:

* + 1. Crear Agent

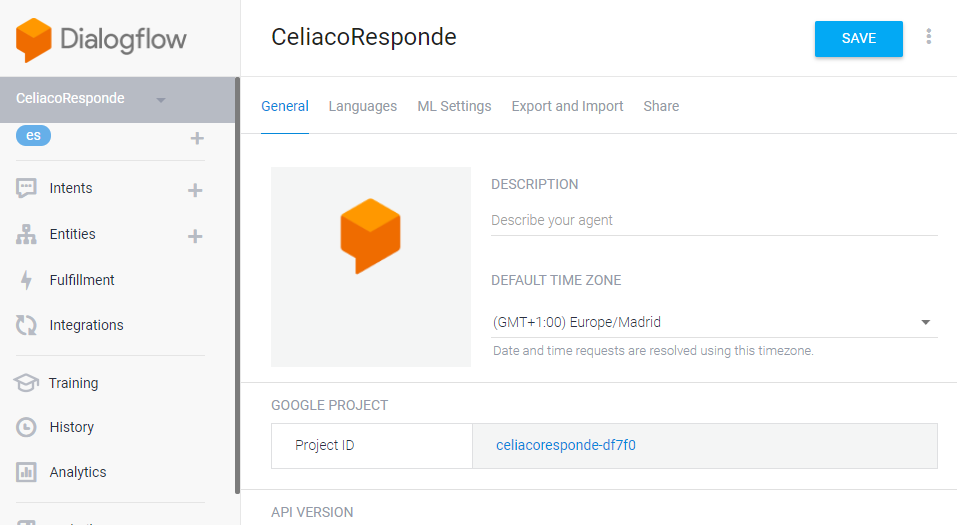


Ilustración : DIALOGFLOW - Creación Agent Ejemplo

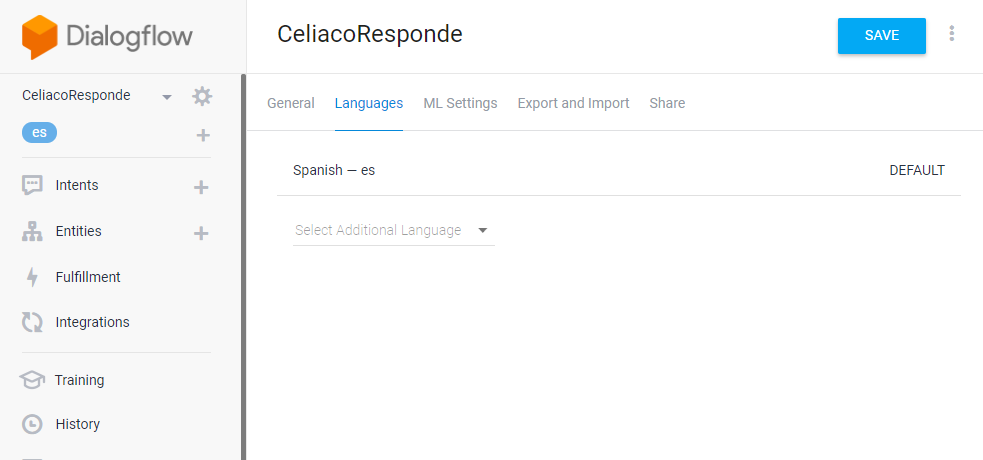


Ilustración : DIALOGFLOW - Creación Agent Ejemplo

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*.

* + 1. Crear Entities

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

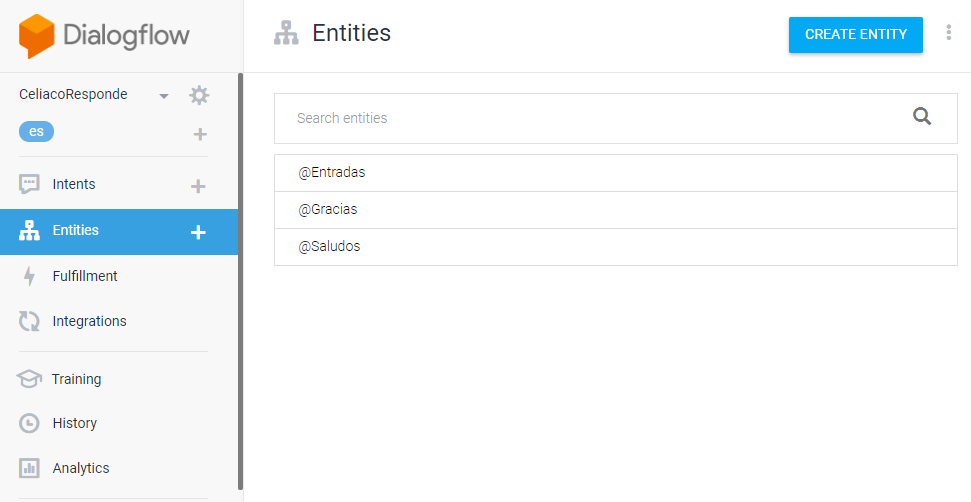


Ilustración : DIALOGFLOW – Conjunto de Entities creadas

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

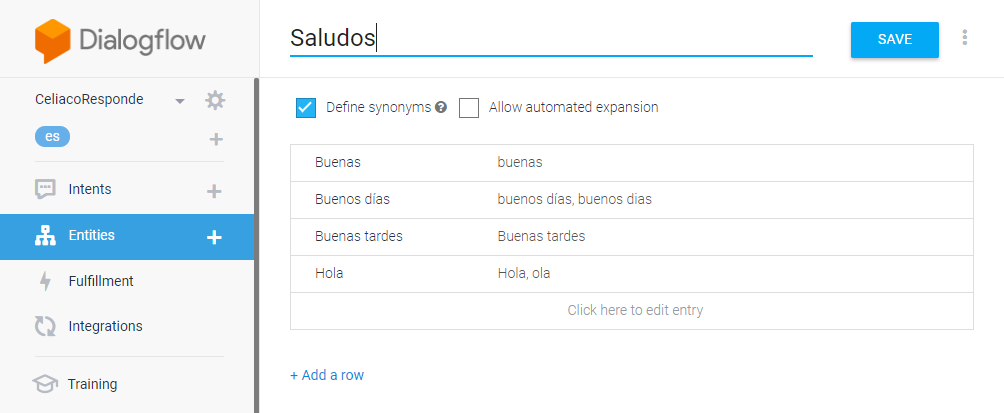


Ilustración : DIALOGFLOW - Entity Saludos

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

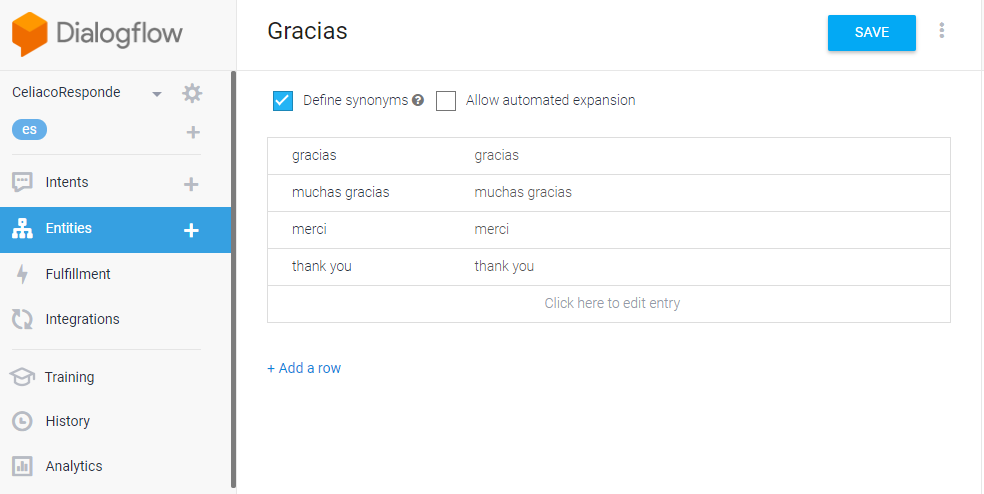


Ilustración : DIALOGFLOW – Entity Gracias

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

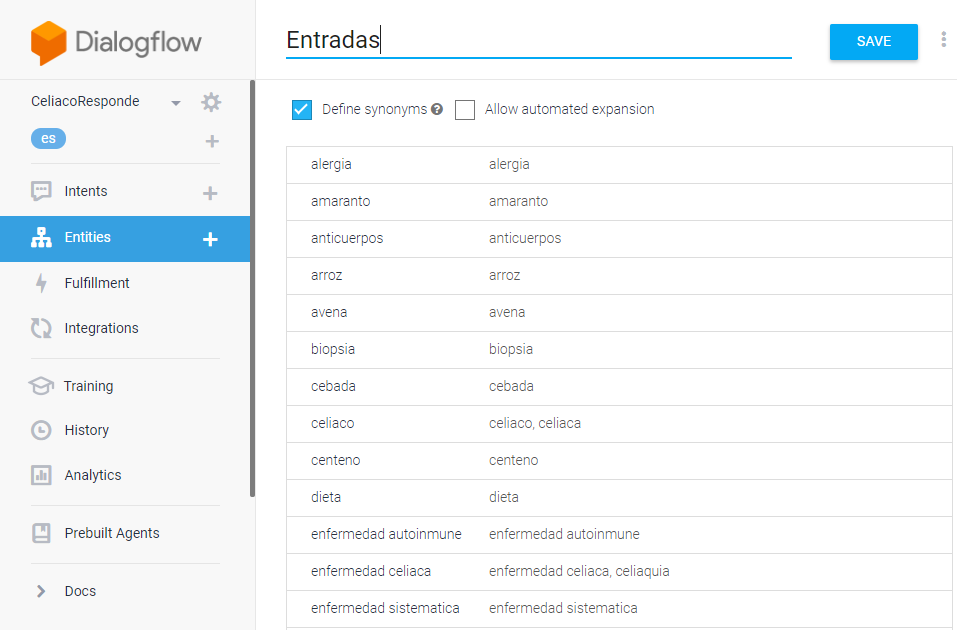


Ilustración : DIALOGFLOW – Entity Entradas

* + 1. Crear Intents

También vamos a crear tres *Intents*:

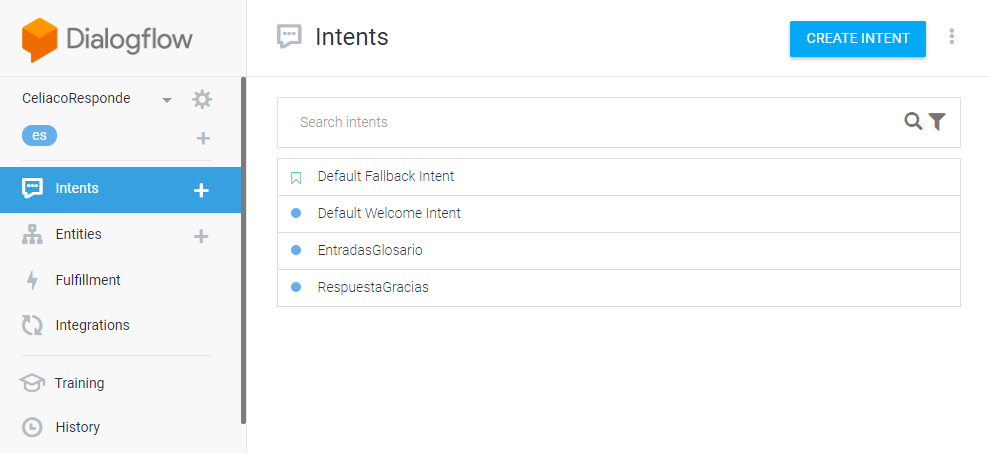


Ilustración : DIALOGFLOW – Conjunto de Intents

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

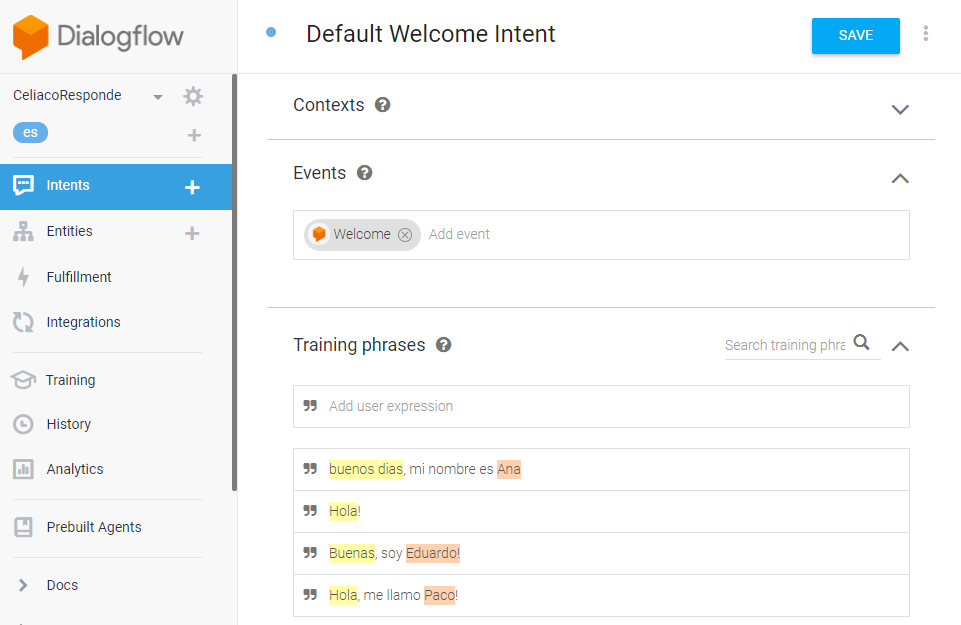


Ilustración 12: DIALOGFLOW – Default Welcome Intent, Training Phrases

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. Cómo 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*:

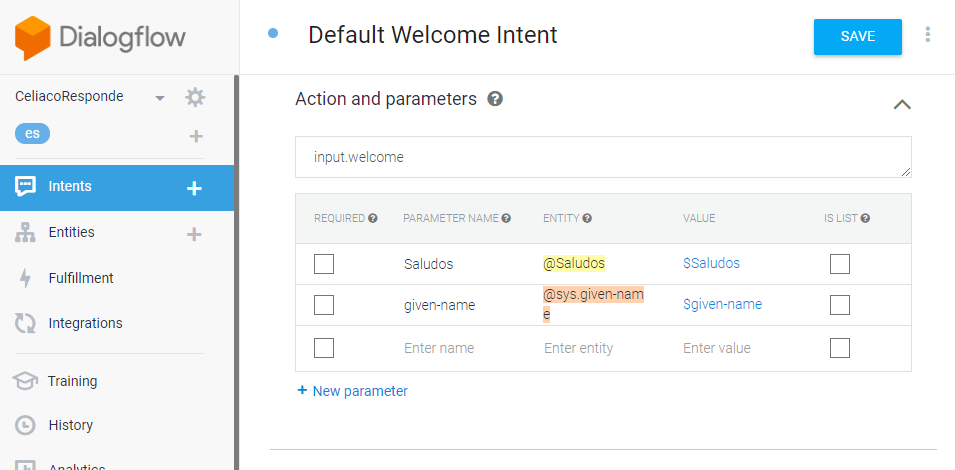


Ilustración 13: DIALOGFLOW – Default Welcome Intent, Parameters

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.

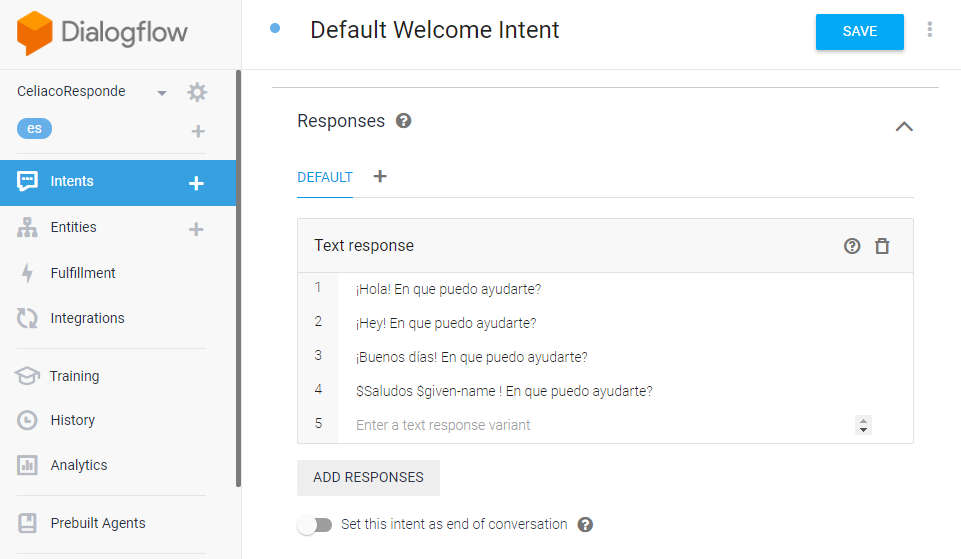


Ilustración : DIALOGFLOW – Default Welcome Intent, Responses

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*

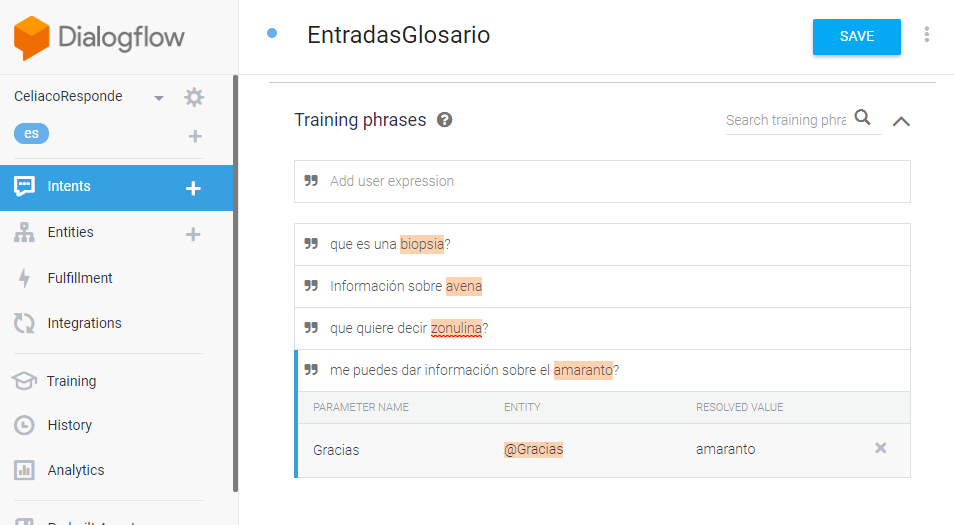


Ilustración : DIALOGFLOW – EntradasGlosario, Training phrases A

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”.

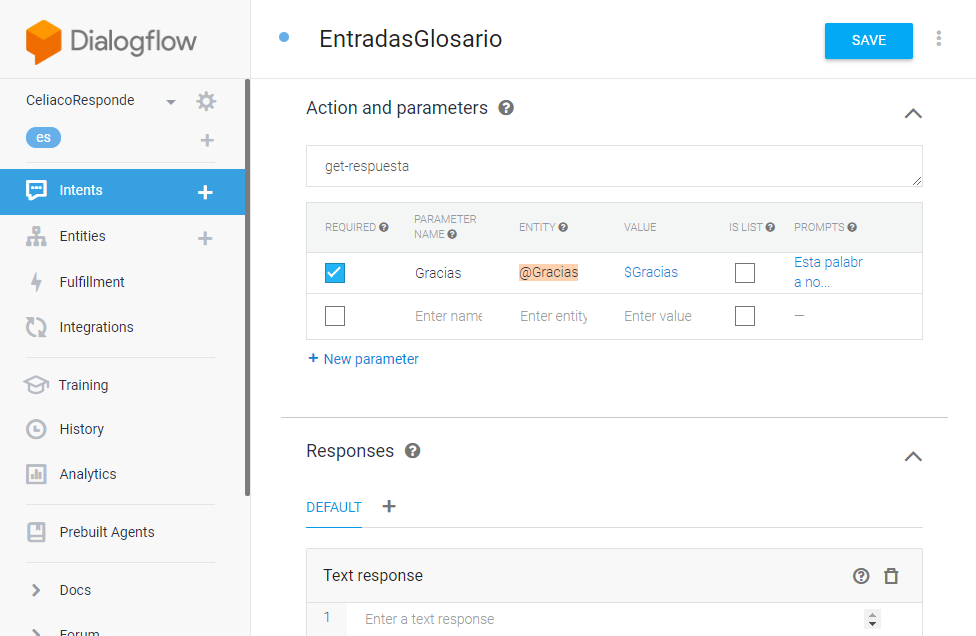


Ilustración : DIALOGFLOW – EntradasGlosario, Parameters A

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.

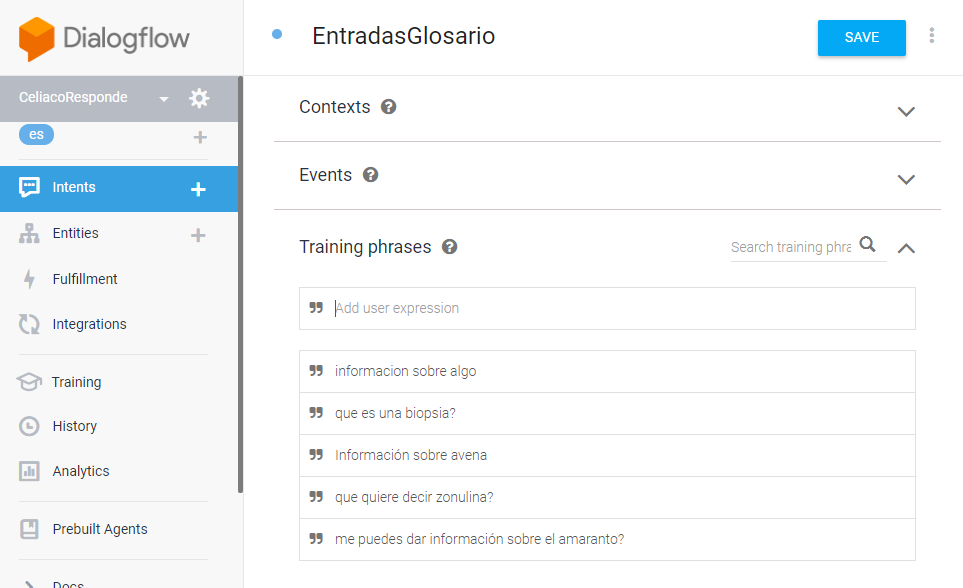


Ilustración : DIALOGFLOW – EntradasGlosario, Training phrases B

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.

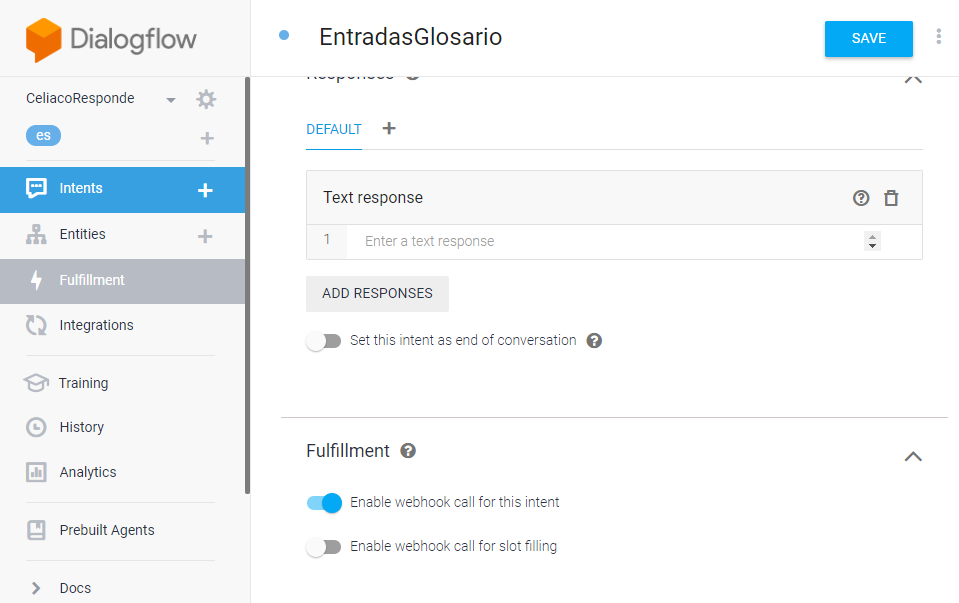


Ilustración : DIALOGFLOW – EntradasGlosario, Responses y Fulfillment

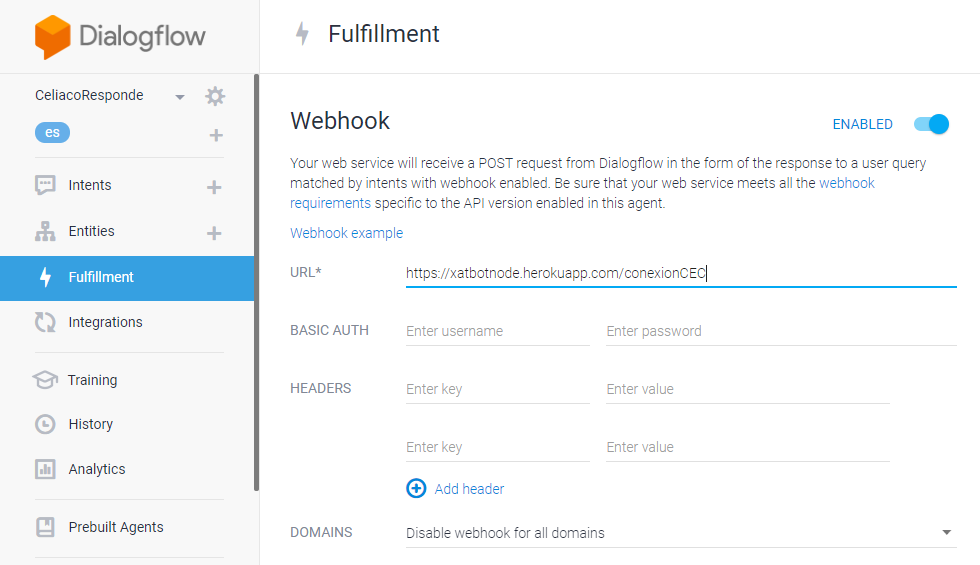


Ilustración : DIALOGFLOW - Webhook

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

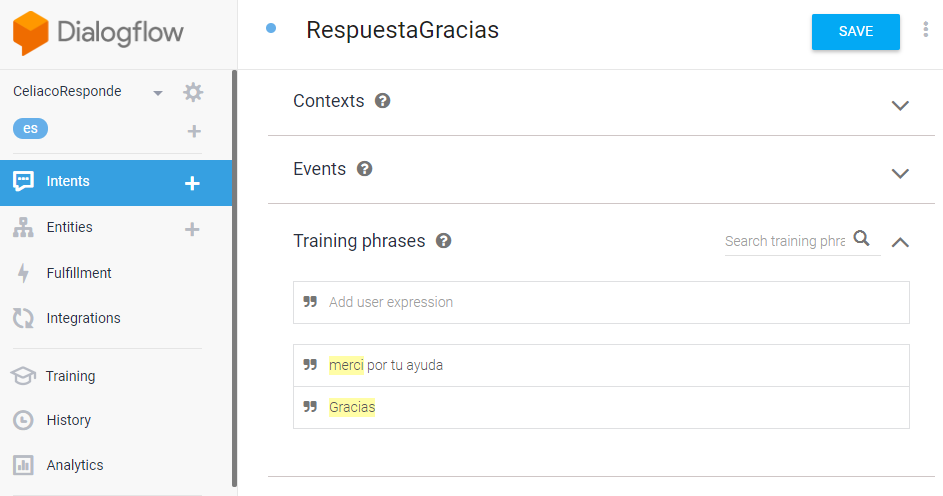


Ilustración : DIALOGFLOW – RespuestaGracias, Training phrases

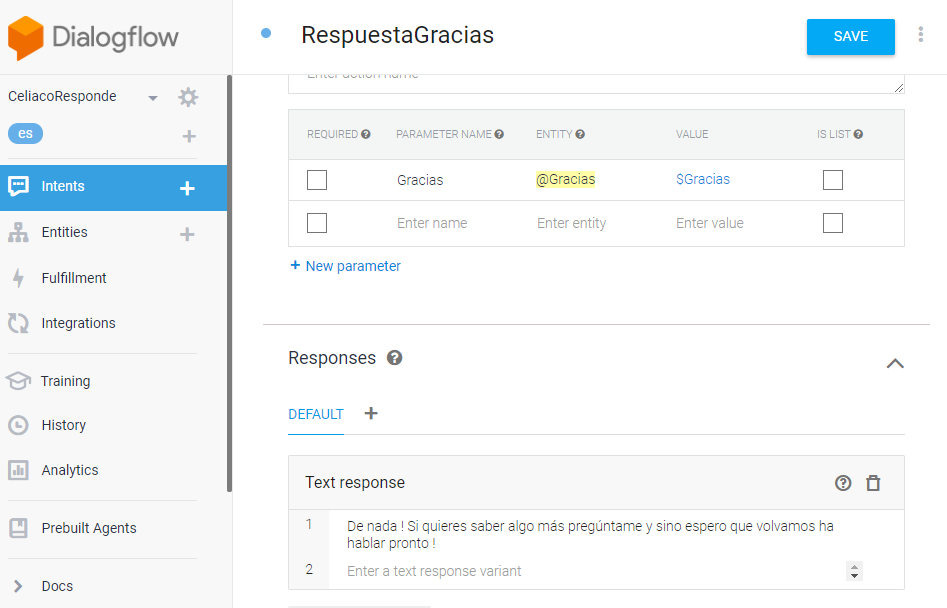


Ilustración : DIALOGFLOW – Parameters y Responses

* + 1. 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*.

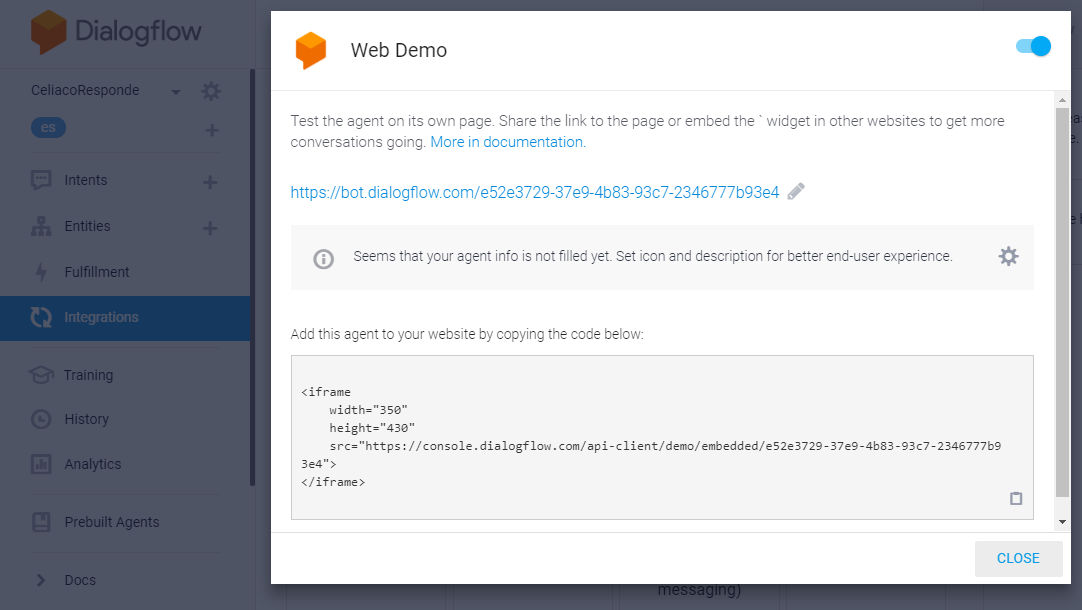


Ilustración : DIALOGFLOW – Integrations, Web Demo

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

* 1. 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.

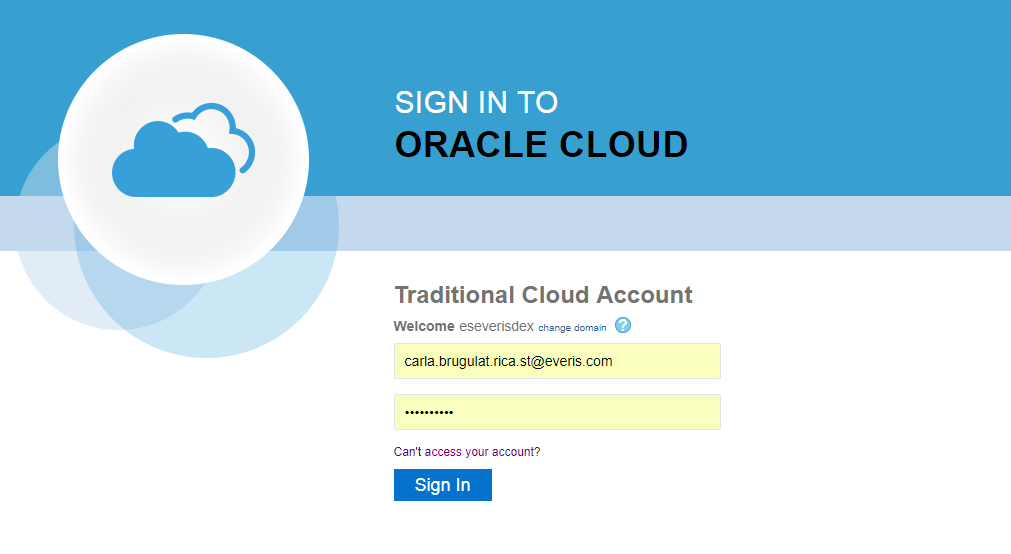


Ilustración 23: CEC – Sign in to Oracle Cloud



Ilustración : CEC - Content

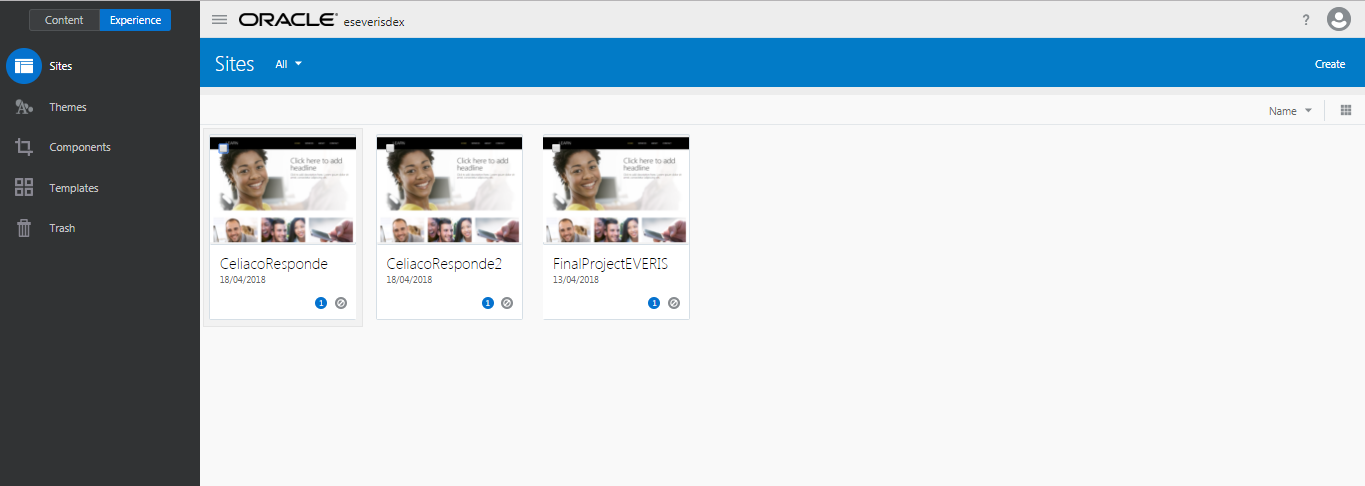


Ilustración : CEC - Experience

* + 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 implemention

// ----------------------------------------------

var SampleComponentImpl = function(args) {

this.SitesSDK = args.SitesSDK;

// Initialze 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: retrive new custom data and re-render the component

//

this.SitesSDK.subscribe(this.SitesSDK.MESSAGE\_TYPES.SETTINGS\_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:

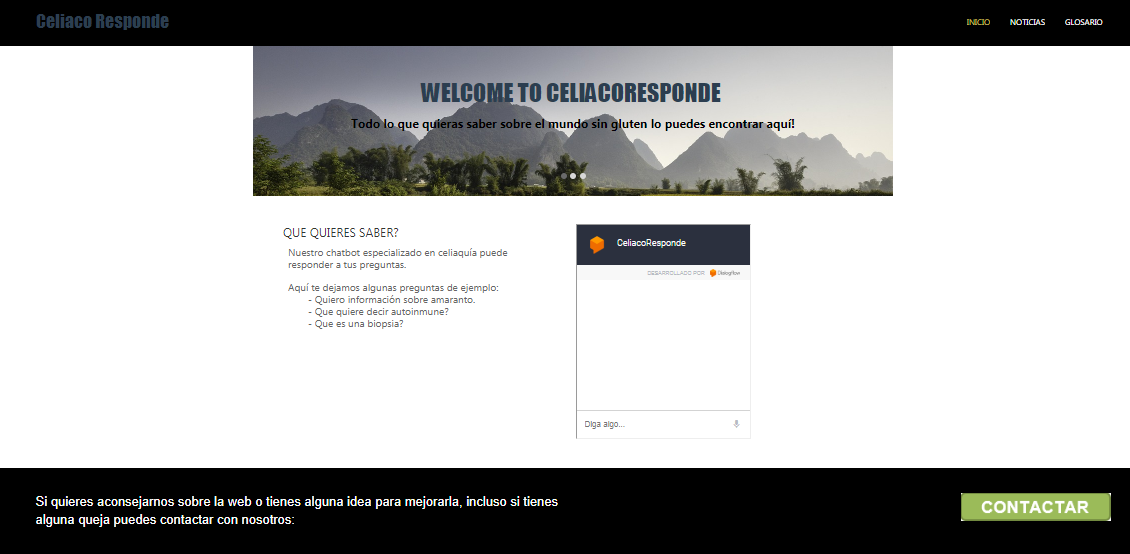
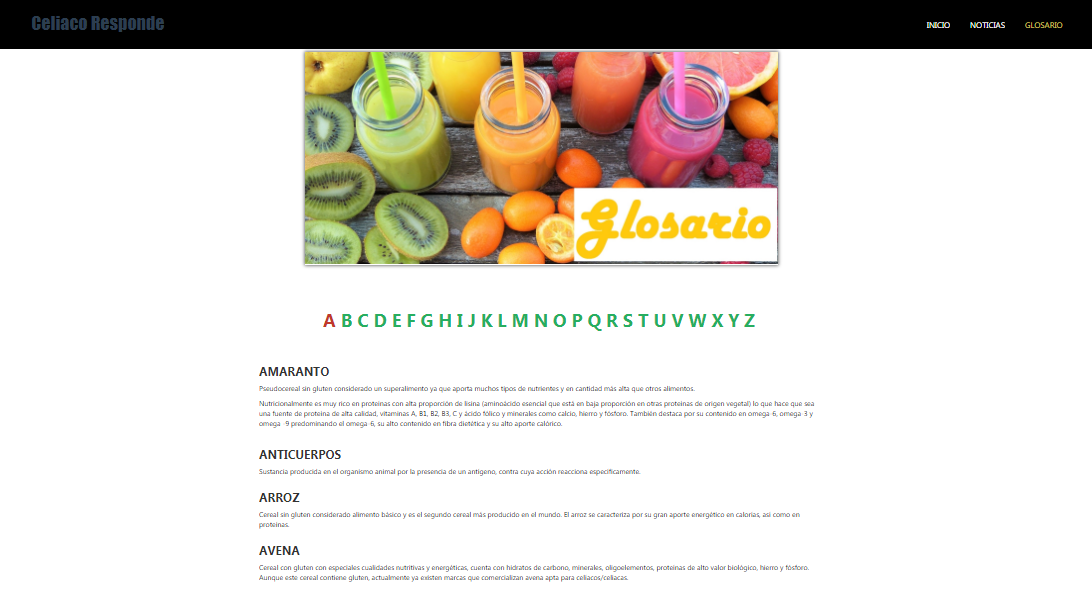


Ilustración : CEC – Site Página de Inicio



Ilustración : CEC – Site Página de Noticias



lustración : 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.

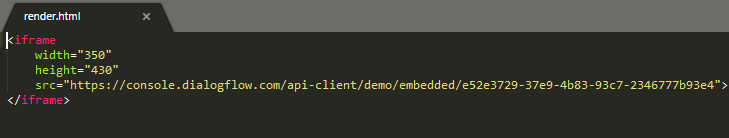


Ilustración : CEC – Componente HTML, contenido archivo render.html

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.

* + 1. 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.

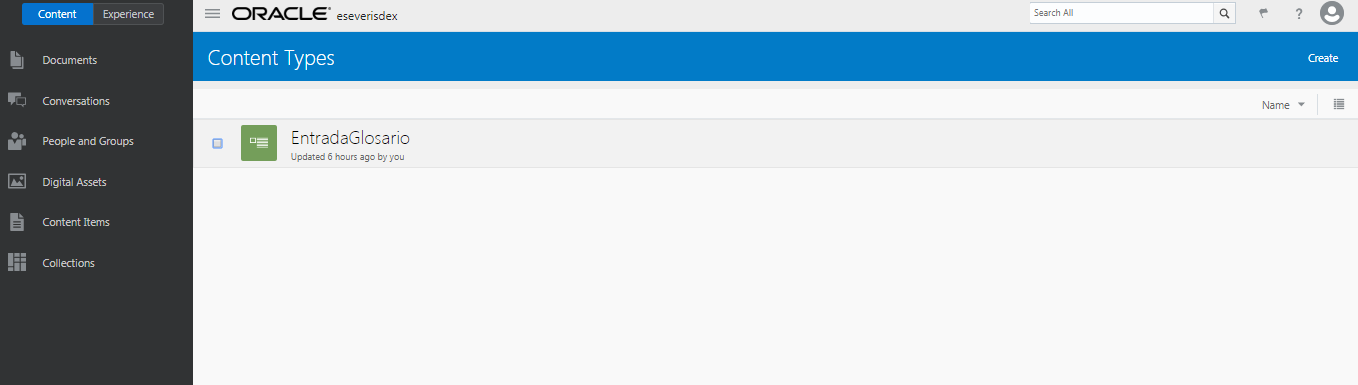
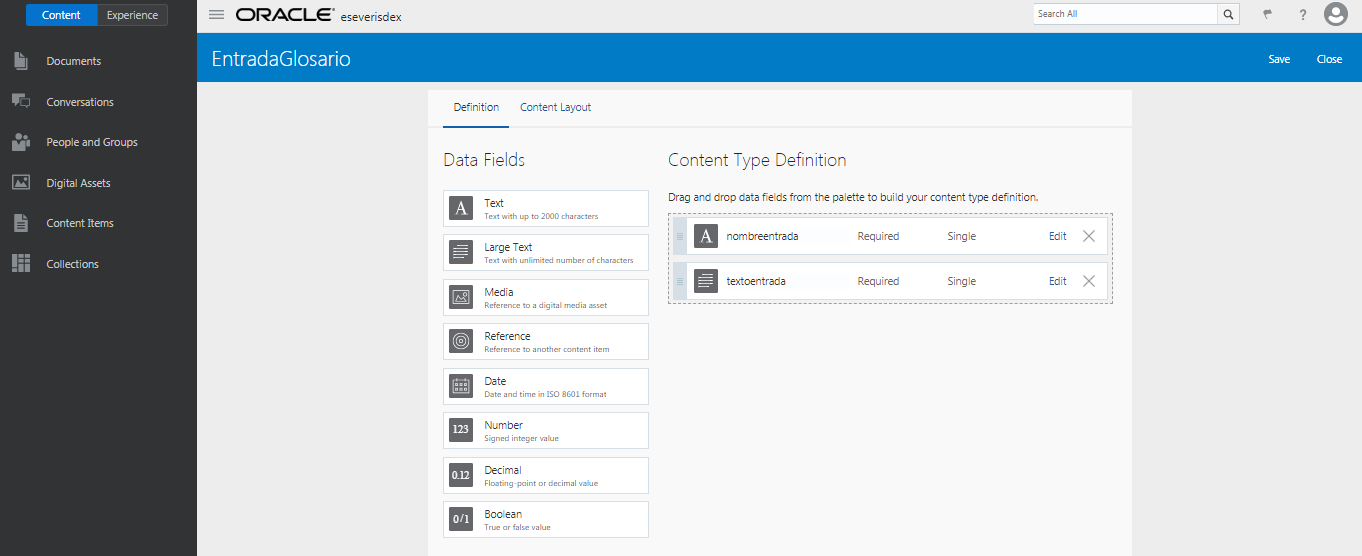
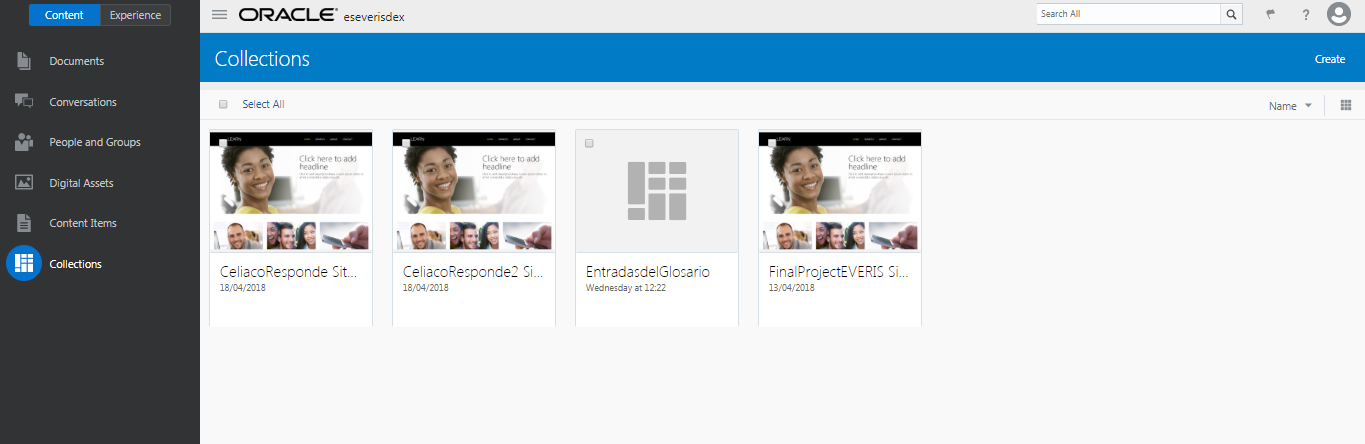


Ilustración : CEC - Collection EntradasdelGlosario

Ilustración 31: CEC – Content Type, Data Fields

Ilustración : CEC – Content Type, EntradaGlosario

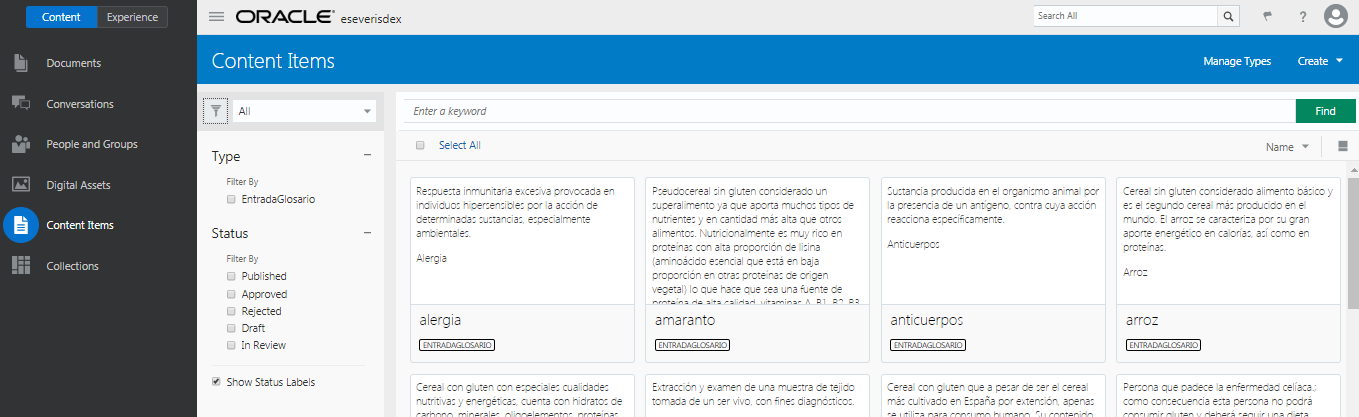


Ilustración : CEC – Content Items

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

* 1. 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.

* + 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.

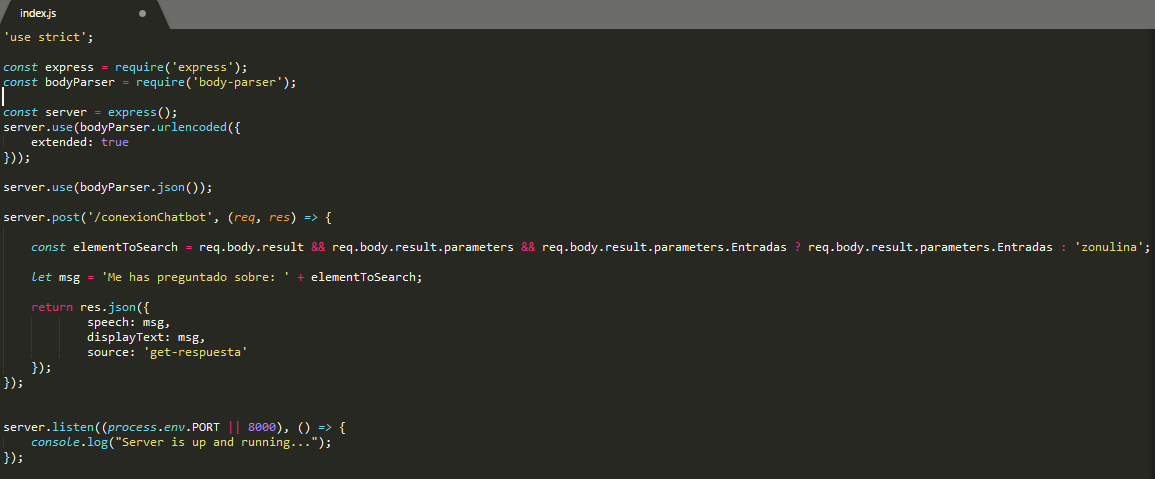
El primer paso será instalar *Node.js* a nuestro ordenador. Seguidamente, abriremos la consola de comandos e iremos a la ruta del proyecto para poder inicializar *Node.js* en la carpeta.

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*:



* + 1. Crear aplicación de Node.js que se conecte con el chatbot y con el CEC
  1. Heroku
     1. Cec-restapi
     2. Chatbot-oracle

Ilustración : 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 : 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.

* + 1. 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.

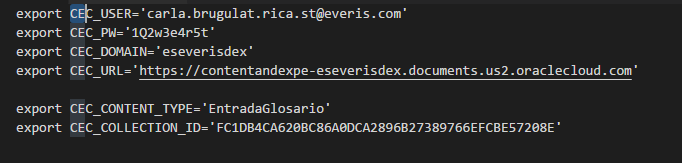
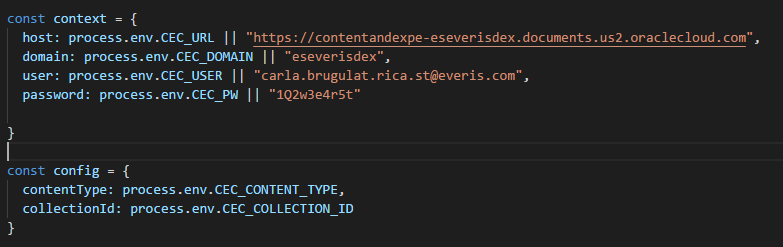
El *cec-restapi* es el código proporcionado por *Oracle* que nos habilita la conexión con el *Content Experience Cloud*, este código no debe ser modificado por nosotros prácticamente, solamente se deben actualizar las credenciales que nos permiten acceder a nuestro *CEC*. Los dos archivos que se deben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

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

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

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

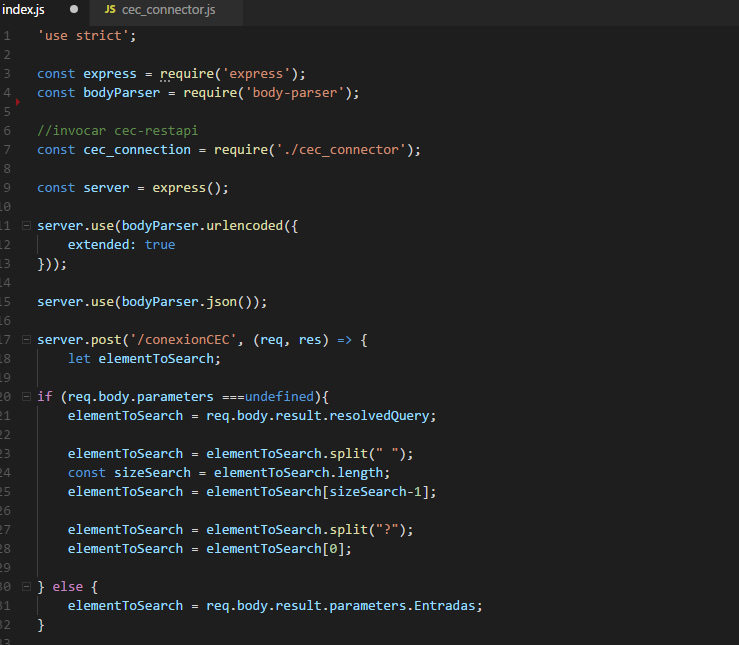
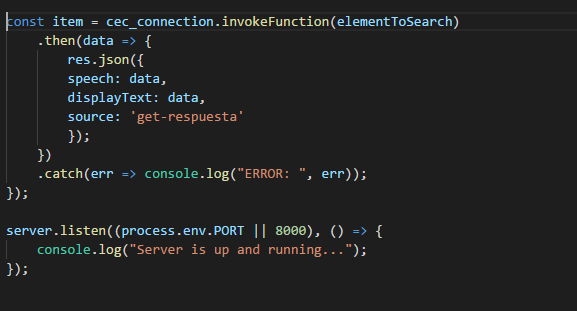


Ilustración : NODE:JS – segunda aplicación con conexión al cec-restapi

Para este segunda parte del proyecto la resolución esta echa 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:

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

"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"

}

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 cómo parámetro de la función.

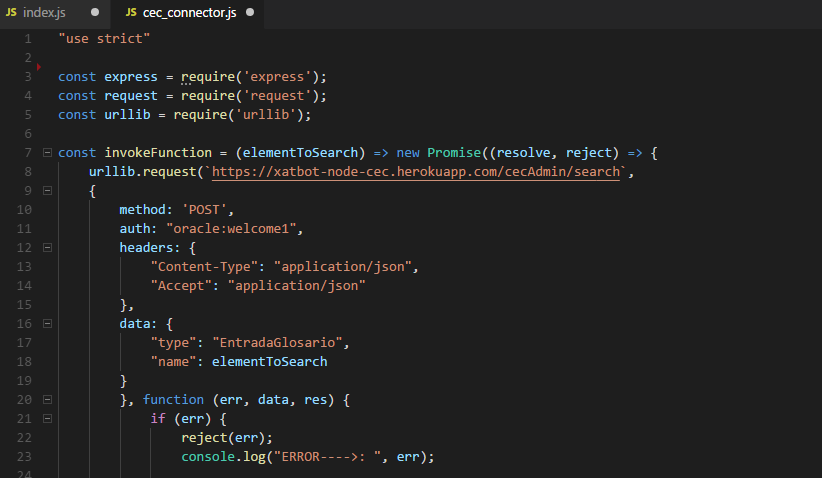


Ilustración : NODE.JS, cec\_connector.js 1

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*.

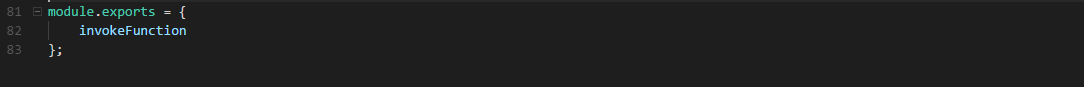
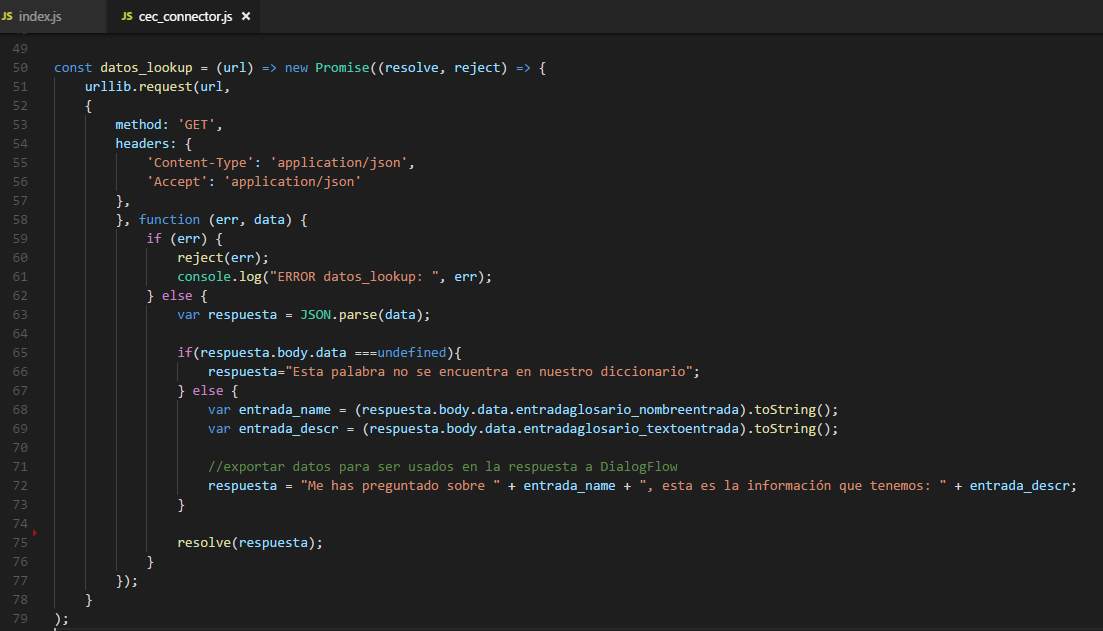


Ilustración : NODE.JS, cec\_connector.js 2

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.

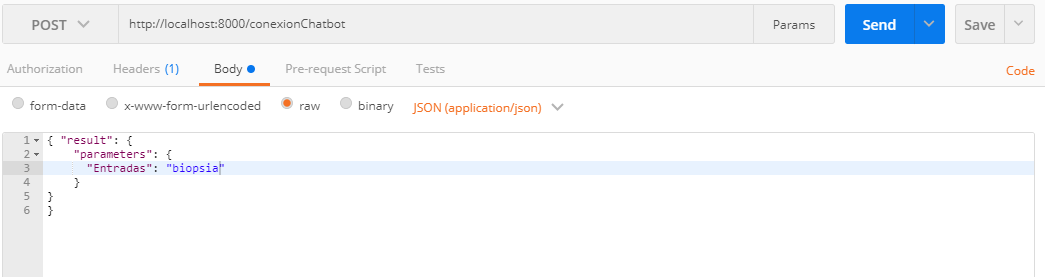
Todo el contenido creado para realizar esta parte del proyecto se puede encontrar en la carpeta principal del proyecto.

* 1. 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 : POSTMAN – Aplicación 1, pregunta



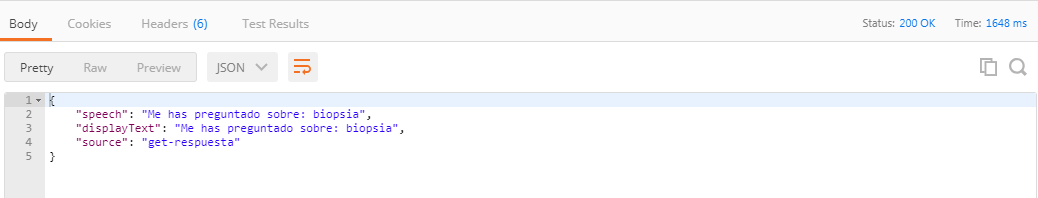
Al hacer clic en “*Send*” deberíamos recibir la siguiente respuesta:

Ilustración : 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.

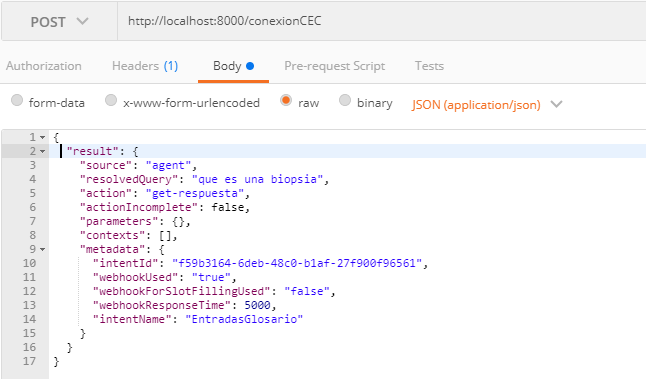


Ilustración : POSTMAN – Aplicación 2, pregunta

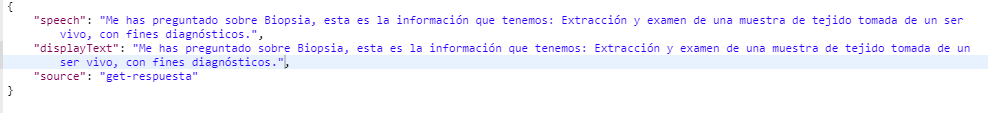
Al hacer clic en “*Send*” deberíamos recibir la siguiente respuesta:

Ilustración : 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.

* 1. 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:

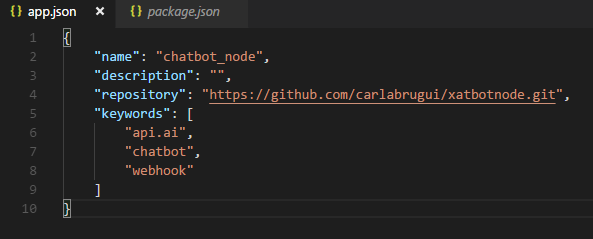


Ilustración : HEROKU – Archivo app.json aplicación entera

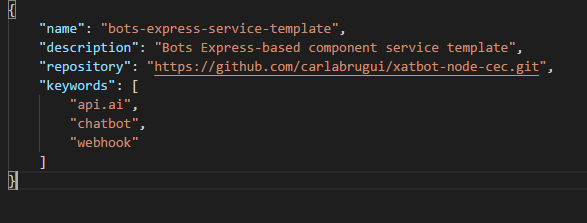


Ilustración :: HEROKU – Archivo app.json aplicación cec-restapi

1. Modificar el archivo *package.json*:

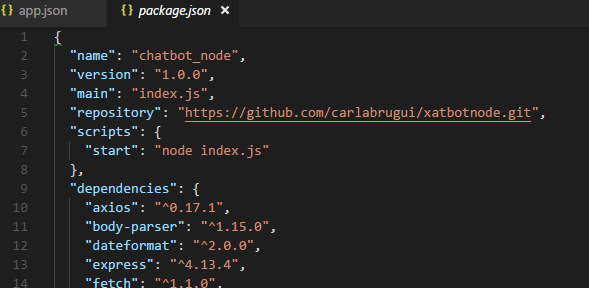


Ilustración : HEROKU – Archivo package.json aplicación entera

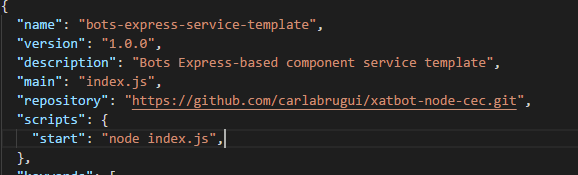


Ilustración : HEROKU – Archivo package.json aplicación cec-restapi

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

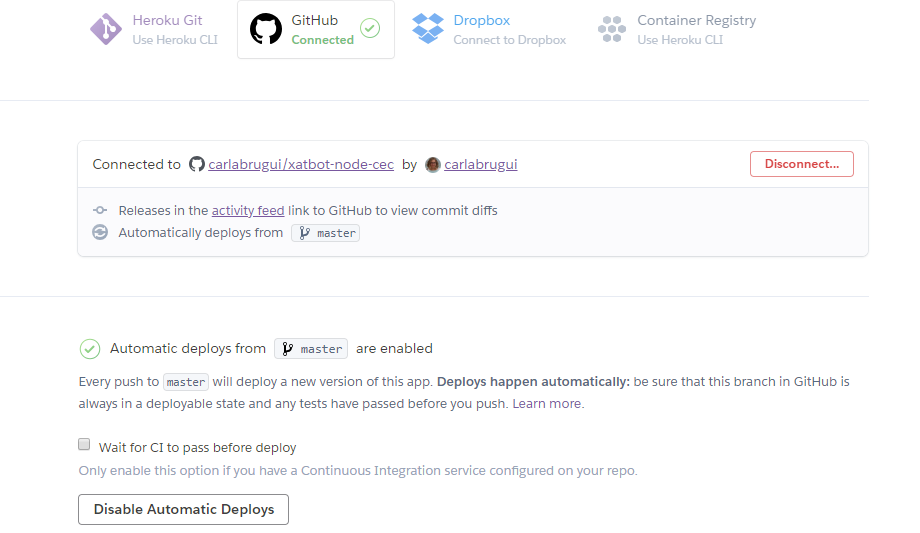
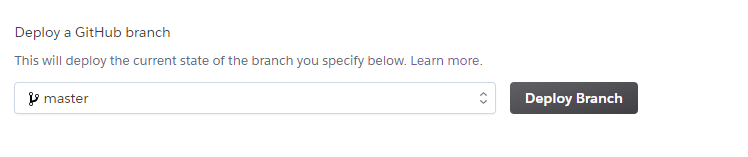
* Crear un repositorio en *GitHub*, tener en cuenta que el nombre debe coincidir con el del archivo *app.json* y *package.json.*
* Abrir *Git* *Bash.*
* Ir a la carpeta del proyecto: *cd “dominio”.*
* Clonar el repositorio que hemos creado en *GitHub*: *git clone “ruta repositorio”.*
* Ir a la carpeta 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.*

1. Comprobar que los cambios se muestran en *GitHub.*
2. Crear aplicación en *Heroku:*

* Una vez tenemos una aplicación creada debemos conectarla a *GitHub* para subir el código.

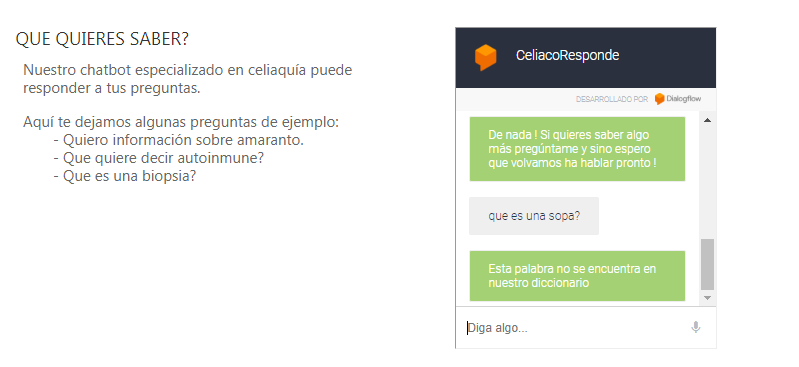
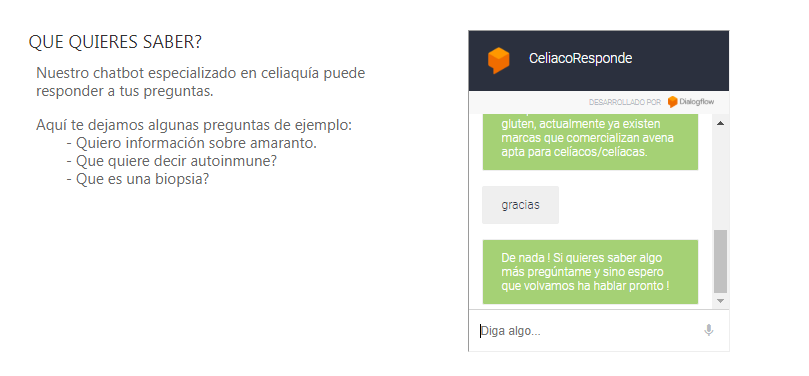
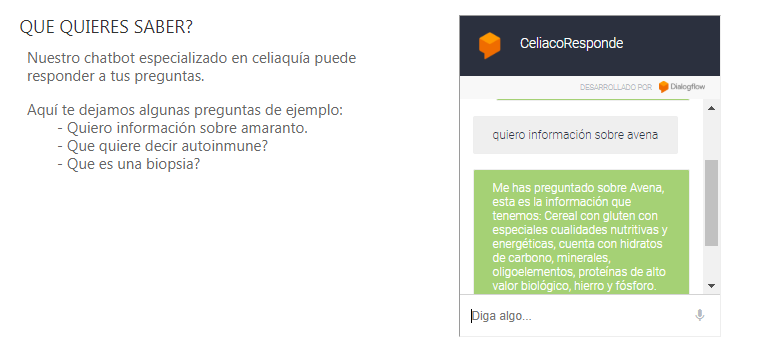
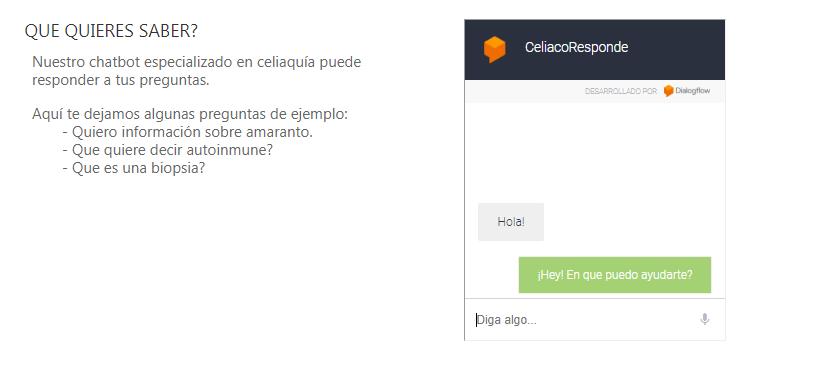
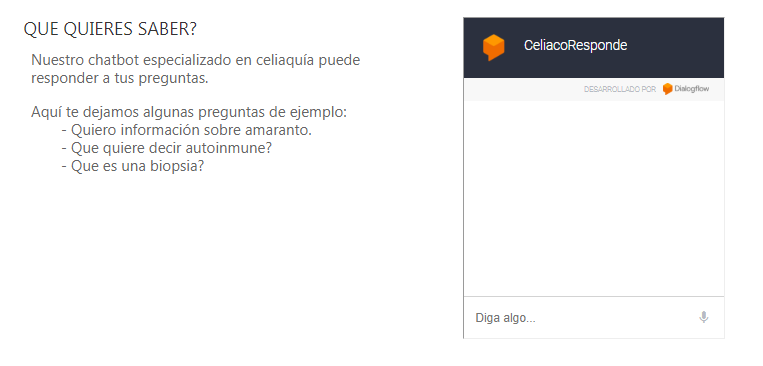
Finalmente, automatiza las subidas de código y haz un *Deploy Brach*.

Ilustración : HEROKU – Subir código de la aplicación des de GitHub



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

Ejemplo de conversación



1. 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>