

# **Práctica de SOA**

---

**Servicios REST usando JEE (Jersey + Jetty),  
lanzados con Heroku, usando NLP via API.ai e  
integrados con Facebook Messenger**

---

**Arquitecturas y Tecnologías del Software**

## **Autores**

**David Lozano Jarque (NIU 1359958)**

**Carlos González Cebrecos (NIU 1212586)**

**Adrián Soria Bonilla (NIU 1360940)**

Escuela de Ingeniería, Universidad Autónoma de Barcelona

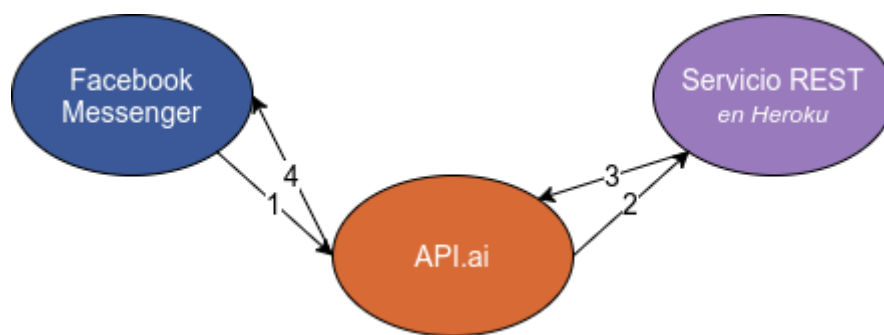
Curso 2016-2017

# Introducción

---

En esta práctica, desarrollaremos un *chatbot* en la plataforma *Facebook*, usando su producto de chat *Facebook Messenger*. Para ello, usaremos *API.ai* con el objetivo de reconocer el lenguaje natural *NLP* e interpretarlo para ofrecer respuestas coherentes.

Finalmente, usaremos un servicio web *REST* que implementaremos usando *Jersey*, servido usando *Jetty* y lo publicaremos mediante *Heroku* para que sea accesible públicamente. Éste servicio web REST lo usaremos para que *API.ai* pueda obtener mediante nuestra API datos relevantes, en éste caso, le proveeremos del estado meteorológico dada una ciudad que *API.ai* nos interpretará a partir de los mensajes recibidos en *Facebook Messenger*.



## Creación del *chatbot* en *Facebook Messenger*


---

En primer lugar, para poder empezar a unir todos estos servicios tenemos que estar registrados en todas las plataformas. Empezaremos realizando el registro en *Facebook Developer* (que requiere previo registro en la red social *Facebook*) para poder desarrollar aplicaciones sobre *Facebook*, en nuestro caso el *chatbot*. También deberemos crear una página de Facebook para vincular nuestro *chatbot* con ésta.

Una vez en la plataforma de desarrolladores y habiendo creado una página en la red social, crearemos una aplicación.



**Panel**



## ATS Bot ○

Esta aplicación está en modo de desarrollo y solo la pueden usar los administradores, desarrolladores y evaluadores [\[?\]](#)

Versión de la API [\[?\]](#)

Identificador de la aplicación

v2.9

307586166342231

Clave secreta de la aplicación

●●●●●●●●

Mostrar

Una vez creada la aplicación en *Facebook* ya podremos conectar un *chatbot* con la plataforma *API.ai*

## Conexión con *API.ai*


### Registro

Comenzaremos por registrarnos en [api.ai](#) y crear un nuevo [agente](#). Los agentes son la interfície del *chatbot* que convertirá las conversaciones en lenguaje natural en acciones a realizar para proporcionar respuestas coherentes.

Una vez creado nuestro agente, podemos configurar algunos aspectos importantes como el idioma o la zona horaria.

**ATSBOT** SAVE ⋮

[General](#) [ML Settings](#) [Export and Import](#) [Share](#)



DESCRIPTION

Describe your agent

LANGUAGE [?](#)


Español




DEFAULT TIME ZONE


(GMT+1:00) Europe/Madrid

GOOGLE PROJECT

[Create or link existing Google project id](#)

 API keys

|                        |                                  |   |
|------------------------|----------------------------------|---|
| Client access token    | 561fd49588b64387a206cbddcbe4efaa |   |
| Developer access token | cc219d93a164413caa7d97fe71decb47 |    |

 DANGER ZONE

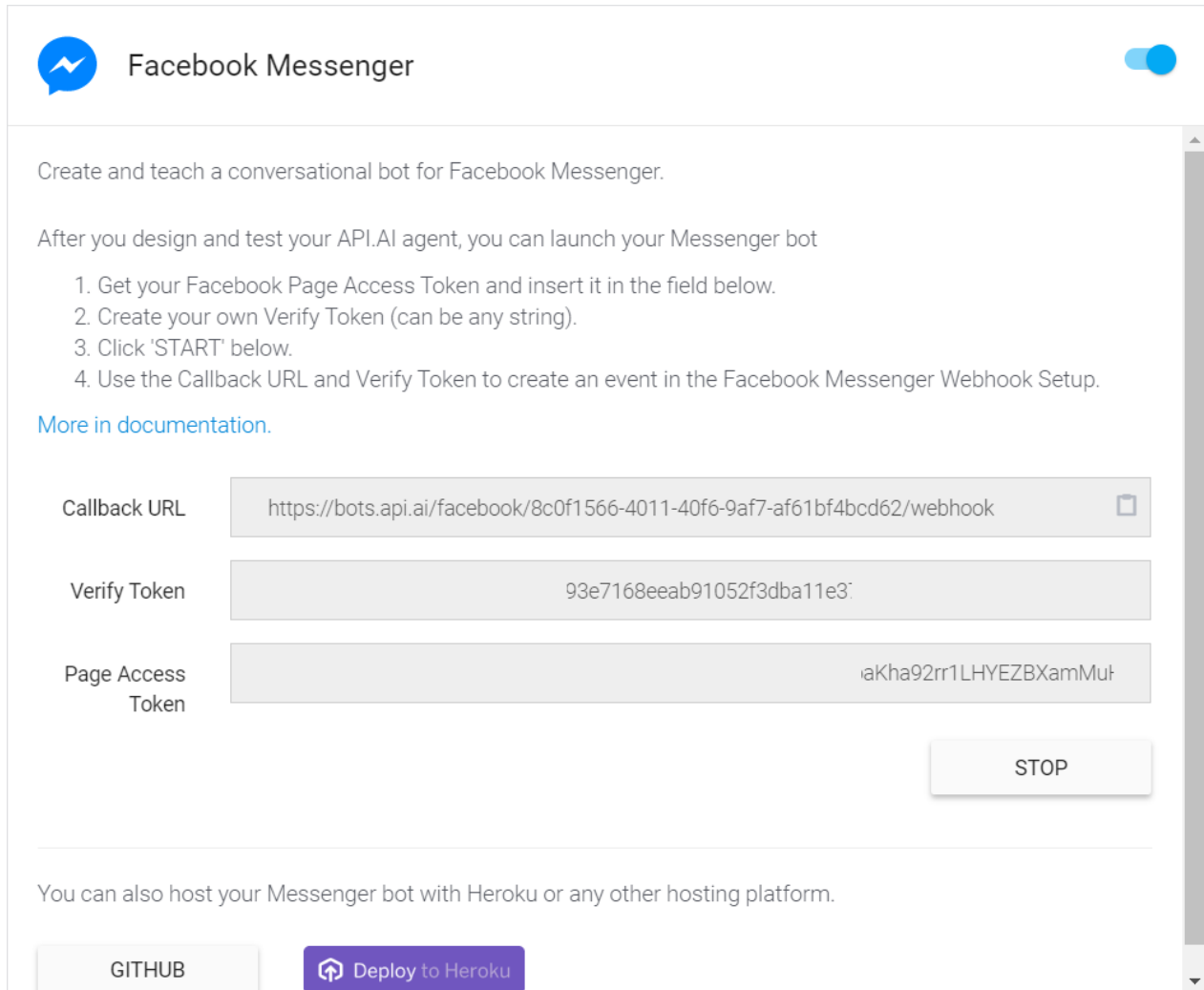
Delete Agent

Are you sure you want to delete agent ATSBOT? This will destroy the agent with all corresponding data and cannot be undone!

DELETE THIS AGENT

## Integración con *Facebook Messenger*

Una vez hecho esto, deberemos vincularlo con nuestra aplicación de *Facebook*, yendo al apartado *integrations* de la plataforma *API.ai*, y seleccionaremos *Facebook Messenger*. Una vez elegido, se nos desplegará un menú como el siguiente, donde deberemos introducir los datos que nos solicite.



The screenshot shows the 'Facebook Messenger' integration setup page. At the top, there's a header with the Facebook Messenger logo and a toggle switch. Below the header, the main content area contains instructions: 'Create and teach a conversational bot for Facebook Messenger.' and 'After you design and test your API.AI agent, you can launch your Messenger bot'. A numbered list follows: 1. Get your Facebook Page Access Token and insert it in the field below. 2. Create your own Verify Token (can be any string). 3. Click 'START' below. 4. Use the Callback URL and Verify Token to create an event in the Facebook Messenger Webhook Setup. A link 'More in documentation.' is provided. Below the instructions are three input fields: 'Callback URL' with the value 'https://bots.api.ai/facebook/8c0f1566-4011-40f6-9af7-af61bf4bcd62/webhook', 'Verify Token' with the value '93e7168eeab91052f3dba11e3', and 'Page Access Token' with the value 'aKha92rr1LHYEZBXamMu'. A 'STOP' button is located to the right of the input fields. At the bottom, there's a note: 'You can also host your Messenger bot with Heroku or any other hosting platform.' and two buttons: 'GITHUB' and 'Deploy to Heroku'.

El significado de los campos es el siguiente:

- **Callback URL**

URL que debemos configurar en la plataforma de desarrollo de *Facebook*, en el apartado de *Messenger*, para que cuando recibamos un mensaje del chat, haga una petición *POST* hacia *API.ai* para que éste interprete el mensaje y devuelva una respuesta. Como vemos, la URL apunta a los servidores de *API.ai* que usando nuestro agente responderán al chat de forma automática.

- **Verify Token**

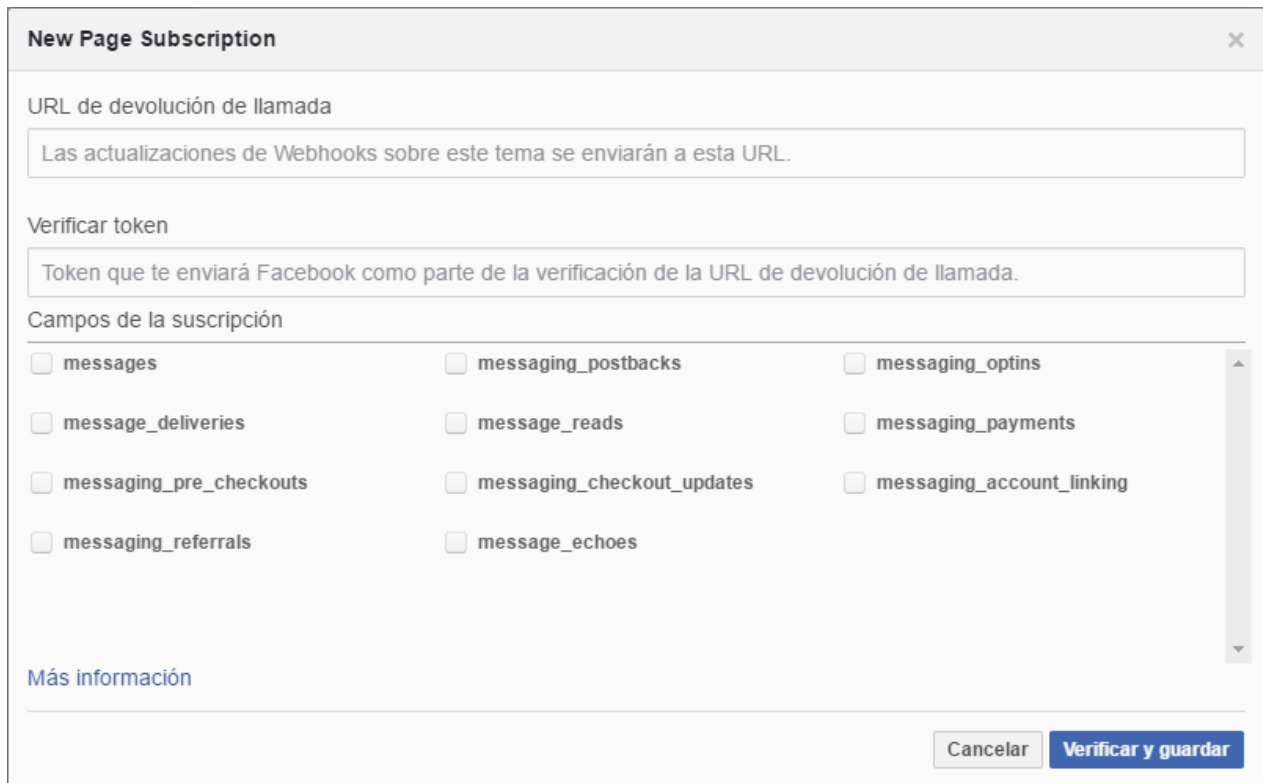
Cadena de caracteres que **debemos crear (preferiblemente de forma aleatoria)** como clave para que sólo el que la posea pueda usar nuestro agente. Hemos de escoger bien la clave, puesto que si escogemos una clave o token débil, cualquiera

que conozca nuestra *Callback URL* puede usar el *chatbot* directamente sin pasar por *Facebook* de forma no autorizada

- **Page Access Token**

Token que autentica a *API.ai* frente a *Facebook* para que pueda enviar respuestas a los mensajes.

Ahora debemos entrar en la plataforma de desarrollo de Facebook, e introducir un nuevo *webhook* en el apartado de *Messenger*, con la *Callback URL* indicada y *Verify Token*.



The screenshot shows the 'New Page Subscription' form in the Facebook Developer console. It includes a title bar with a close button, a 'URL de devolución de llamada' field with a placeholder text, a 'Verificar token' field with a placeholder text, and a 'Campos de la suscripción' section with a grid of checkboxes for various events like 'messages', 'message\_deliveries', 'messaging\_pre\_checkouts', 'messaging\_referrals', 'messaging\_postbacks', 'message\_reads', 'messaging\_checkout\_updates', 'message\_echoes', 'messaging\_optins', 'messaging\_payments', and 'messaging\_account\_linking'. At the bottom, there is a 'Más información' link and two buttons: 'Cancelar' and 'Verificar y guardar'.

Finalmente, nos dirigimos al apartado de *Generación de token* en la misma página de *Messenger* de la aplicación de Facebook en la plataforma de Facebook Developers y copiamos el *Page Access Token* en el formulario de *API.ai*.



The screenshot shows the 'Generación de token' form in the Facebook Developer console. It has a title bar, a paragraph of explanatory text about the token, and a section with two labels: 'Página' and 'Token de acceso a la página'. The 'Página' label has a dropdown menu showing 'ATS Bot'. The 'Token de acceso a la página' label has a text input field containing a long alphanumeric string. Below the input field is a link that says 'Crear una página'.

## Desarrollo del *chatbot*

Ahora debemos programar nuestro *chatbot*. Por ello el siguiente paso es crear unos cuantos *intents*, para que nuestro *chatbot* reconozca intenciones del usuario y empiece a dar respuestas válidas, como por ejemplo el saludo en una conversación, algunas frases de cortesía y demás.

Por defecto, se encuentra el *Default Fallback intent*, que contestará al usuario frases para que el usuario vuelva a repetir la pregunta o texto de forma diferente puesto que se ejecutará cuando no se haya detectado ninguna otra intención.

Lo más básico es crear un *intent* para reconocer saludos y dar respuesta. Dado un mensaje concreto del usuario (que el *chatbot* intentará relacionar si es un mensaje similar), deberemos definir las posibles respuestas que dará nuestro agente.

Nosotros también crearemos un *intent* para que nuestro *chatbot* conteste a la pregunta:

Qué tiempo hace en <Ciudad>?

Y llamaremos a dicho *intent* `weather`. También podemos introducir otras frases para que reconozca dicho *intent*, como

```
Clima en <Ciudad>?  
Hace frío en <Ciudad>?
```

Para que reconozca la ciudad, debemos escribir las anteriores frases con una ciudad de ejemplo y marcar la ciudad como una **entidad**. Las entidades definen conceptos y permiten relacionar las acciones o *intents* con conceptos físicos o abstractos. Podemos definir entidades personalizadas o usar las que existen por defecto. En nuestro caso, usaremos una por defecto, llamada `@sys-geocity` que identifica ciudades.

Finalmente, debemos proporcionar una respuesta a la pregunta. Para ello, desarrollaremos un servicio web con API REST para que dada la ciudad que envíe *API.ai* una vez interpretado el mensaje del usuario, nos devuelva el tiempo de dicha ciudad.

Debemos desarrollar un servicio web propio puesto que debe interpretar los mensajes que enviará *API.ai* que vienen dado un **formato específico**, por lo que no podemos usar directamente una API REST para obtener el tiempo. Nuestro servicio web actuará de intermediario.

## Webhooks en *API.ai*

Para que *API.ai* envíe la solicitud del tiempo indicando la ciudad a nuestro servicio web, debemos de nuevo configurar un *webhook* para que *API.ai* solicite a nuestro servicio web la respuesta del tiempo.

Nos dirigimos al apartado de *fulfillment* y creamos un nuevo *webhook* del cual indicamos su URL. También nos permite usar autenticación para comprobar que *API.ai* está autorizado a usar el webservice (igual que *Facebook Messenger* con *API.ai* en la integración realizada previamente), pero nosotros no lo configuraremos, simplemente introduciremos nuestra URL del *webservice*.

## Webhook

Your web service will receive a POST request from API.AI 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](#).

[Webhook example](#)

ENABLED ☒

URL\*

BASIC AUTH

HEADERS

[+ Add header](#)

DOMAINS

La URL del webservice la suponemos puesto que vamos a crear una aplicación en Heroku con nombre personalizado. En el caso de desconocerla, podemos realizar este paso de integración con *webhooks* una vez creado el webservice y por lo tanto cuando conozcamos dicha URL

En último lugar, nos dirigimos al intent de *Weather* y seleccionamos la casilla de *webhook* para en el caso de recibir un mensaje con solicitud del estado meteorológico, solicitar al servicio web nuestro con el *webhook* configurado la respuesta.

**i** If no response is defined in an integration tab, responses below will be used.

Text response

1

2

ADD MESSAGE CONTENT

Fulfillment

☒ Use webhook ☐ Use webhook for slot-filling

# Creación de *webservice* REST usando *Jersey*, *Jetty* y publicado en *Heroku*

## Creación del *webservice*

*Jersey* nos ofrece un arquetipo de *Maven* preparado para desarrollar un servicio web de forma rápida, y además, con código extra (archivo *Profile*) para poder una vez desarrollado localmente y probado, enviarlo para su publicación online a *Heroku*.



En la página de [Getting started](#) de *Jersey* nos indican el comando a realizar para crear el proyecto usando dicho arquetipo de *Maven*:

```
mvn archetype:generate
  -DarchetypeArtifactId=jersey-heroku-webapp \
  -DarchetypeGroupId=org.glassfish.jersey.archetypes \
  -DinteractiveMode=false \
  -DgroupId=com.example \
  -DartifactId=simple-heroku-webapp \
  -Dpackage=com.example \
  -DarchetypeVersion=2.25.1
```

Una vez ejecutamos el comando, podemos empezar a desarrollar el servicio web, importando el proyecto en nuestro IDE favorito, como *Eclipse*.

## Desarrollo del *webservice*

Podemos ver que en `com.domain.heroku.Main.java` tenemos un fichero que contiene código para ejecutar el servidor de aplicaciones JEE con nuestro *webservice* ya publicado.

Pero el fichero interesante es `com.domain.resources.ApiResource.java`, dónde se encuentra un ejemplo de `GET` y `POST` de nuestro *webservice*

```
@Path("api")
public class ApiResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getIt() {
        return Response.status(200).entity("{}").build();
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response postIt(String req) {
        return Response.status(200).entity(req).build();
    }
}
```

El código indica que cuando realicemos una petición `GET` a nuestro servidor JEE en la URL `/api`, nos devolverá el *string* `{}`, que al ser devuelto con tipo *JSON*, interpretará como un objeto *JSON* vacío. En el caso del `POST`, devolverá los datos que reciba.

Con este ejemplo, ya podemos lanzar nuestro *webservice* usando el comando

```
mvn clean install
```

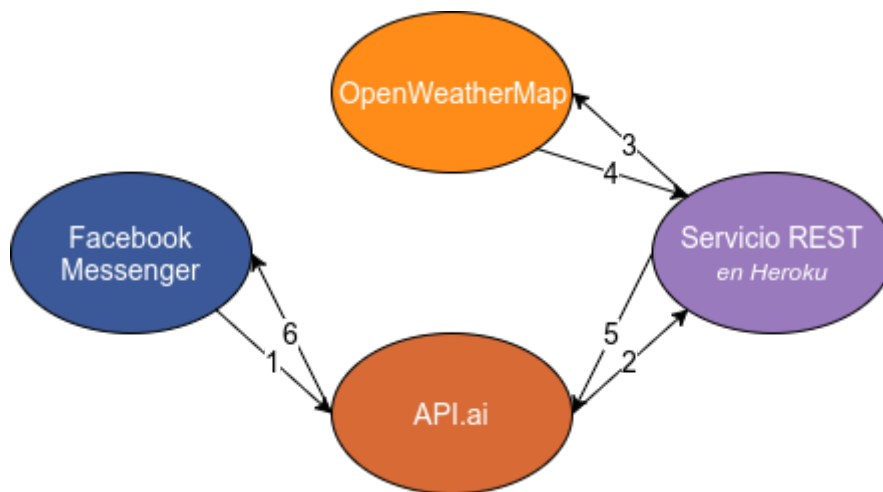
Para compilar y

```
mvn jetty:run
```

Para ejecutar el servidor *Jetty* con nuestro *webservice* lanzado.

## Consumición del servicio web de meteorología conectado a *API.ai*

Tal y como hemos mencionado antes, nuestro *webservice* ha de interpretar los mensajes que *API.ai* nos envíe mediante el *webhook* que configuramos para responder el tiempo de la ciudad solicitada. El diagrama del sistema entonces se ampliaría al siguiente:



### *OpenWeatherMap*

El servicio REST que consumiremos para obtener el tiempo de una ciudad es [OpenWeatherMap](#), que según su [documentación](#) requiere que para obtener el tiempo de una ciudad, realicemos un `GET` a la siguiente URL

```
https://api.openweathermap.org/data/2.5/weather?q=London
```

Donde *London* es la ciudad a consultar. También necesitamos añadir a estos *query parameters* de la URL una *API key* (parámetro `appid`) que podemos obtener registrándonos en *OpenWeatherMap* y opcionalmente añadir el idioma de la respuesta mediante el parámetro `lang`, de manera que la URL a la que realizar el `GET` es

```
https://api.openweathermap.org/data/2.5/weather?q=London&lang=es&appid=<token>
```

### Código para consumir el servicio

Para consumir el servicio, crearemos una nueva clase en el proyecto dónde con un método *main* podremos comprobar localmente si podemos obtener correctamente el tiempo de una ciudad pasada por parámetro.

Usaremos la librería *JSON* de `javax`, en su implementación de `Glassfish` para analizar la [respuesta proporcionada por OpenWeatherMap](#), convirtiendo el *String* de respuesta en un objeto Java más fácil de manipular y acceder a los campos del objeto *JSON*. El código resultante es el siguiente:

```

public class WeatherConsumer {
    public static void main(String[] args) {
        System.out.println(weather(args[0]));
    }
    public static String weather(String city) {
        // Cliente REST
        Client client = ClientBuilder.newClient();
        // Generación de URL
        String api_key = "<api_key>"; // esto no lo revelamos :P
        String URL = "http://api.openweathermap.org/data/2.5/weather";
        try {
            URL += "?q=" + URLEncoder.encode(city, "UTF-8");
            URL += "&lang=" + URLEncoder.encode("es", "UTF-8");
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        URL += "&appid=" + api_key;

        // Petición GET
        WebTarget resource = client.target(URL);
        Builder request = resource.request();
        request.accept(MediaType.APPLICATION_JSON);
        Response response = request.get();

        // Análisis de la respuesta
        if (response.getStatusInfo().getFamily() == Family.SUCCESSFUL) {
            // Conversión String <-> JSON
            JsonReader jsonReader = Json.createReader(new StringReader(
                response.readEntity(String.class)));
            JsonObject json = jsonReader.readObject();
            jsonReader.close();
            // Obtención del campo description, dentro del primer obje
to
            // del array contenido en el campo "weather"
            JsonArray weatherArray = json.getJsonArray("weather");
            JsonObject weatherFirstObject = weatherArray.getJsonObject(0
);
            String weatherDescription = weatherFirstObject.getString("des
cription");
            // Devolución del resultado
            return weatherDescription;
        } else {
            System.out.println("ERROR: " + response.getStatus());
            System.out.println(response.getEntity());
        }
        return "No se ha encontrado respuesta";
    }
}

```

Probamos el código localmente y comprobamos que efectivamente, devuelve el tiempo en dicha ciudad, en castellano.

## Respondiendo a *API.ai*

El último paso es responder al `POST` de *API.ai* con el tiempo, obteniendo primero la ciudad dados los datos proporcionados por *API.ai* y creando una respuesta interpretable por *API.ai*

### Interpretando los datos de *API.ai*

Usando la [documentación de API.ai](#), podemos ver que los datos en formato *JSON* que nos enviará *API.ai* tienen un objeto `parameters` dentro del campo `result`, en el cual aparecen las entidades identificadas en el mensaje, en éste caso, la ciudad, del tipo `sys-geocity`.

Para obtener la ciudad, pues, debemos convertir los datos que recibamos de *API.ai* en un objeto Java para acceder al campo deseado y llamar a nuestro cliente consumidor del servicio web de *OpenWeatherMap* para obtener el tiempo.

Modificamos así el código del `POST` de ejemplo del proyecto y introducimos el siguiente código:

```
@POST
@Produces(MediaType.APPLICATION_JSON)
public Response post(String req) {
    // Read JSON
    JsonReader jsonReader = Json.createReader(new StringReader(req));
    JsonObject json = jsonReader.readObject();
    jsonReader.close();
    String city = json
        .getJsonObject("result")
        .getJsonObject("parameters")
        .getString("geo-city");
    String weather = WeatherConsumer.weather(city);
    // [...]
}
```

De manera que ahora en la variable `city` tendremos la ciudad a devolver el tiempo y mediante la llamada a nuestro consumidor del servicio web, obtendremos el resultado en la variable `weather`.

### Devolviendo el resultado a *API.ai*

Siguiendo la documentación de *API.ai* debemos devolver un objeto *JSON* como respuesta a la petición que contenga el texto a devolver al usuario en el campo `speech` y `displayText`.

Continuamos el desarrollo de la función anterior para elaborar la respuesta.

```

@POST
@Produces(MediaType.APPLICATION_JSON)
public Response post(String req) {
    // [...]
    // Elaboramos respuesta
    JsonBuilderFactory responseFactory = Json.createBuilderFactory(nul
1);
    JsonObject response = responseFactory.createObjectBuilder()
        .add("speech", weather)
        .add("displayText", weather)
        .build();
    // Convertimos respuesta a String
    String responseString = response.toString();
    // Enviamos respuesta
    return Response.status(200).entity(responseString).build();
}

```

Ahora sólo nos queda publicar el servicio web en *Heroku* para que *API.ai* pueda usar el *webhook*

## Desplegando el servicio en *Heroku*

Para no tener el servidor con el *webservice* en nuestra máquina local, usaremos [Heroku](#) que nos permitirá tener de forma gratuita nuestro servicio web alojado en una máquina virtual de [Amazon Web Services](#).

Para ello, primero debemos instalar las [Heroku CLI Tools](#) para poder ejecutar comandos que nos permitan comunicarnos con *Heroku* para lanzar nuestras aplicaciones.

En primer lugar, debemos iniciar sesión desde el terminal (debemos habernos registrado previamente)

```
heroku login
```

Una vez iniciada la sesión, creamos una nueva aplicación donde lanzaremos nuestro proyecto. Para ello, nos situamos en el directorio del proyecto y ejecutamos el siguiente comando

```
heroku create ats-webservice
```

Esto nos creará una aplicación disponible en la URL

```
https://ats-webservice.herokuapp.com
```

Para enviar nuestro proyecto allí, debemos realizar un commit con los cambios y subirlo al remoto de *Heroku*


```
git add . && git commit && git push heroku master
```


También podemos vincular a través del panel de administración web de Heroku, un repositorio de *GitHub* para que cuando se realice un `push` a una determinada *branch*, *Heroku* automáticamente actualice el proyecto con el último *commit* de la rama.

---

App connected to GitHub

Code diffs, manual and auto deploys are available for this app.


Connected to  [uab-projects/ats-webservice](#) Disconnect...

- ✓ Releases in the [activity feed](#) link to GitHub to view commit diffs
- ✓ Automatically deploys from  `master`

---

Automatic deploys

Enables a chosen branch to be automatically deployed to this app.

✓ Automatic deploys from  `master` are enabled

Every push to `master` will deploy a new version of this app. Deploys happen automatically: be sure that this branch in GitHub is always in a deployable state and any tests have passed before you push. [Learn more](#).

☐ Wait for CI to pass before deploy

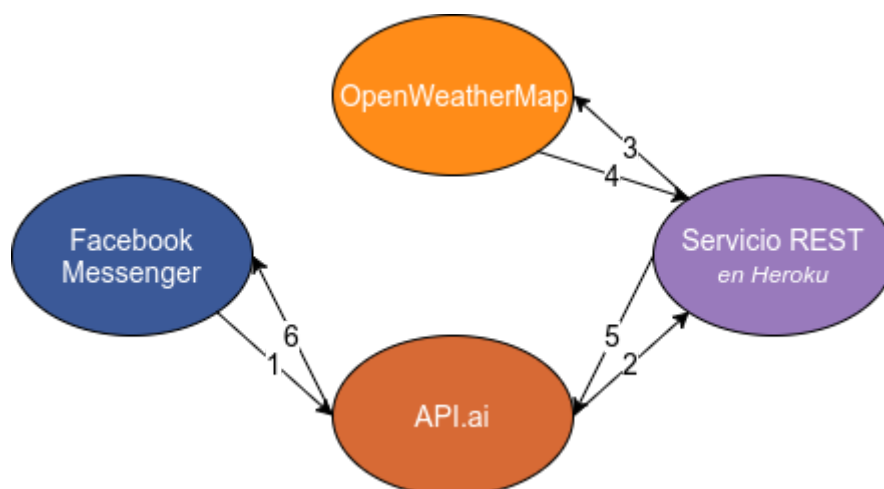
Only enable this option if you have a Continuous Integration service configured on your repo.

[Disable Automatic Deploys](#)

---

## Detalle de comunicaciones

Revisemos de nuevo las comunicaciones con APIs del tipo REST en formato JSON que ocurren en nuestro sistema:



Recordamos que las primeras comunicaciones se realizan desde *Facebook Messenger* hacia *API.ai* para que éste interprete el lenguaje natural. A pesar de que la integración es automática, los mensajes que se intercambian también son una API en formato JSON usando webservices (recordemos que configuramos un *webhook*).

En segundo lugar, *API.ai* si detecta que la conversación solicita el tiempo, se comunicará con nuestro servicio web en *Heroku* para obtener el tiempo.

Y finalmente nuestro servicio web usa una API REST en formato JSON de *OpenWeatherMap* para obtener el estado meteorológico

## API de *Facebook Messenger*

### Notificación de nuevos mensajes

En cuanto a *Facebook Messenger*, cuando un usuario introduce un mensaje, tenemos un *webhook* enlazado al evento de *message*. De ésta forma, cuando un usuario envía un mensaje, *Facebook Messenger*, lanza el evento *message* y sus consecuentes *webhooks*, enviando en este caso a *API.ai* un documento *JSON* usando un `POST` HTTP a nuestra *Callback URL* de *API.ai* que configuramos previamente con los datos del mensaje.

El formato de dicho documento es el siguiente:

```
{
  "sender": {
    "id": "USER_ID"
  },
  "recipient": {
    "id": "PAGE_ID"
  },
  "timestamp": 1458692752478,
  "message": {
    "mid": "mid.1457764197618:41d102a3e1ae206a38",
    "text": "hello, world!",
    "quick_reply": {
      "payload": "DEVELOPER_DEFINED_PAYLOAD"
    },
    "attachments": [{
      "type": "image",
      "payload": {
        "url": "IMAGE_URL"
      }
    }]
  }
}
```

Dónde `sender` nos identifica el usuario que solicita el mensaje (con una ID de *Facebook* que identifica su perfil), `recipient` el destinatario (seáse la página de *Facebook* dónde envió el mensaje), `timestamp` el tiempo en segundos desde la época UNIX, y `message` un subdocumento *JSON* que contiene no sólo el texto que envió el



usuario (campo `text` ), sino también el identificador del mensaje (campo `mid` ) además de otros campos no relevantes.

En el caso de tener archivos adjuntos, como imágenes o audio, un campo llamado `attachments` dentro del subdocumento `message` contendría estos en forma de `array` . Cada `attachment` , contendría el tipo ( `video` , `image` , `audio` ...) en el campo `type` , y su ubicación en el subdocument `payload` , normalmente, la `URL` para acceder a éste

## Envío de respuestas

Para responder a los mensajes recibidos, no debemos responder al `POST` con datos, sino realizar nosotros un `POST` a la siguiente `URL` :

```
https://graph.facebook.com/v2.6/me/messages?access_token=<PAGE_ACCESS_TOKEN>
```

Con la respuesta o mensaje a enviar, en un documento *JSON* del siguiente formato:

```
{
  "recipient": {
    "id": "USER_ID"
  },
  "message": {
    "text": "hello, world!"
  }
}
```

Dónde `recipient.id` es el destinatario (lo identificamos por el `sender.id` del document *JSON* para recibir mensajes) y `message.text` es el texto a enviar como respuesta.

Todo este proceso lo realiza de forma automática *API.ai* para gestionar la integración con *Facebook*, pero en el caso de querer características adicionales, deberíamos de integrar esta *API* en nuestro *webservice* para poder comunicarnos con *Facebook Messenger*. Luego podríamos seguir usando *API.ai* para procesamiento de lenguaje natural usando su *API* de comunicaciones.

## API de API.ai

El siguiente paso es la comunicación mediante *webhook* de *API.ai* con nuestro *webservice* para proporcionar una respuesta válida a la pregunta sobre el estado meteorológico. Para ello, *API.ai* realizará una petición *POST* a nuestro *webservice* en

*Heroku* solicitando el estado meteorológico de una ciudad indicada, que esperará como respuesta a dicho *POST* la respuesta a enviar al usuario finalmente.

## **Notificación de *API.ai* a nuestro *webservice***

El formato de documento que usa *API.ai* para notificarnos una nueva solicitud del estado climatológico es:

```

{
  "originalRequest": {
    "source": "facebook",
    "data": {
      "sender": {
        "id": "1489813491089329"
      },
      "recipient": {
        "id": "1279462395495348"
      },
      "message": {
        "mid": "mid.$cAASLqmo2AKxiFRbtuVb49GlQTWWv",
        "text": "Que tiempo hace en Mollet?",
        "seq": 125729
      },
      "timestamp": 1.494175819193E12
    }
  },
  "id": "40a543cf-5543-4b3c-a942-dbb6a47173ad",
  "timestamp": "2017-05-07T16:50:19.621Z",
  "lang": "es",
  "result": {
    "source": "agent",
    "resolvedQuery": "Que tiempo hace en Mollet?",
    "speech": "",
    "action": "",
    "actionIncomplete": false,
    "parameters": {
      "geo-city": "Mollet"
    }
  },
  "contexts": [{
    "name": "generic",
    "parameters": {
      "geo-city": "Mollet",
      "geo-city.original": "Mollet",
      "facebook_sender_id": "1489813491089329"
    }
  },
  "lifespan": 3
}],
  "metadata": {
    "intentId": "163a1f8a-aacd-4ce9-a81f-13d4d04dcc28",
    "webhookUsed": "true",
    "webhookForSlotFillingUsed": "false",
    "intentName": "Tiempo"
  },
  "fulfillment": {
    "speech": "Déjame consultar ...",
    "messages": [{
      "type": 0,
      "speech": "Déjame consultar ..."
    }]
  },
  "score": 0.92
},

```

```

    "status": {
      "code": 200,
      "errorType": "success"
    },
    "sessionId": "a639035c-ce83-42fb-be16-e52237cd45f6"
  }
}

```

Vemos que en el documento que nos envía *API.ai* podemos obtener la *request* original de *Facebook*, usando su API, en el campo `originalRequest`. En el campo `lang` se nos indica el código del idioma de la petición, y lo más importante se encuentra el el subdocumento del campo `result`, en el que aparece el resultado del procesado del lenguaje natural. Allí vemos que podemos obtener la ciudad que el usuario ha indicado en `parameters.geo-city` y que el *intent* es el de obtener el tiempo en `metadata.intentName`.

## Respuesta a *API.ai*

Como respuesta a dicho mensaje, debemos responder un documento *JSON* con el formato especificado en la [API de webhooks de API.ai](#). La respuesta debe producirse en menos de 5 segundos para poder tenerse en cuenta, de lo contrario, se responderá un texto (o varios) por defecto configurable en *API.ai*

```

{
  "speech": "Algo de nubes",
  "displayText": "Algo de nubes"
}

```

La respuesta debe contener dos campos, `speech` y `displayText` con el texto a devolver al usuario. La diferencia entre estos dos campos es que el `speech` es el texto a pronunciar en el caso que se vaya a devolver un texto hablado y `displayText` el texto escrito a mostrar. **Ambos deben estar especificados en el documento, aunque sean iguales** para que *API.ai* pueda procesar la respuesta.

## API de OpenWeatherMap

La última *API* es la que usa nuestro *webservice* para comunicarse con *OpenWeatherMap* y obtener el tiempo de una ciudad.

### Solicitud de tiempo

Para solicitar el estado meteorológico de una ciudad, debemos hacer una petición `GET` HTTP a la URL

```
https://api.openweathermap.org/data/2.5/weather?q=London&lang=es&appid=  
=<token>
```

En la URL se indican los parámetros como la ciudad, la *API key* para autorizarnos como solicitantes lícitos (y restringirnos el servicio si realizamos más peticiones de la cuenta) o el idioma.

Es importante no publicar la llave API en un repositorio público, por lo que podemos hacer para proteger dicha llave es guardar la llave en una variable de entorno que [configuraremos en Heroku](#) y codificar en *Java* que la llave API la obtenga leyendo dicha variable de entorno

## Respuesta a la solicitud del tiempo

La respuesta a dicha petición es un documento *JSON* con el siguiente formato:

```

{
  "coord": {
    "lon": -0.13,
    "lat": 51.51
  },
  "weather": [{
    "id": 300,
    "main": "Drizzle",
    "description": "light intensity drizzle",
    "icon": "09d"
  }],
  "base": "stations",
  "main": {
    "temp": 280.32,
    "pressure": 1012,
    "humidity": 81,
    "temp_min": 279.15,
    "temp_max": 281.15
  },
  "visibility": 10000,
  "wind": {
    "speed": 4.1,
    "deg": 80
  },
  "clouds": {
    "all": 90
  },
  "dt": 1485789600,
  "sys": {
    "type": 1,
    "id": 5091,
    "message": 0.0103,
    "country": "GB",
    "sunrise": 1485762037,
    "sunset": 1485794875
  },
  "id": 2643743,
  "name": "London",
  "cod": 200
}

```

Dónde el campo que aprovecharemos será el `weather.main` que contiene una descripción del tiempo en la ciudad en forma de texto en el idioma seleccionado

## Test del *chatbot*

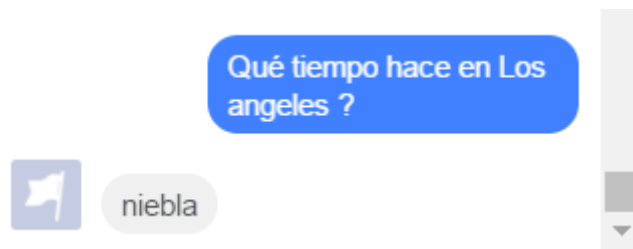
---

### Enlace del *chatbot*

El *chatbot* puede usarse desde el siguiente enlace:

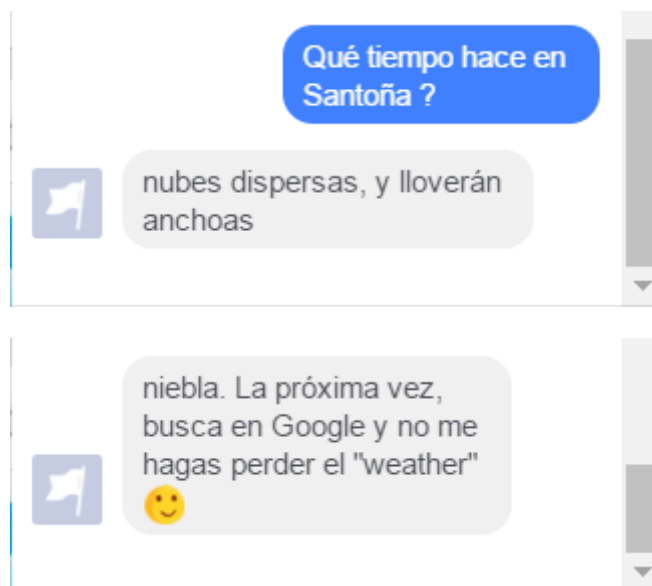
<https://www.messenger.com/t/1279462395495348>

Finalmente, probamos que nuestro *chatbot* ya está listo y operativo



## Continuación del desarrollo

A partir de aquí, podemos comenzar a desarrollar para ofrecer una experiencia de usuario más cómoda, cercana y divertida



## Repositorio

---

El código de nuestro *webservice* se encuentra en *GitHub* en el siguiente repositorio:

<https://github.com/uab-projects/ats-webservice>

Esta misma documentación se encuentra allí accesible en formato *Markdown*, *PDF* y *HTML* en la rama `gh-pages` y es visible en:

<https://uab.codes/ats-webservice/>