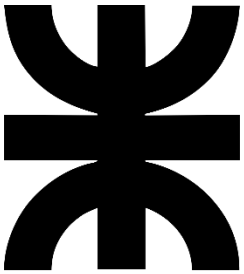
ZACARIAS, LAUTARO NAHUEL.



UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Reconquista

TECNICATURA UNIVERSITARIA EN PROGRAMACION

DISEÑO, PROGRAMACIÓN E IMPLEMENTACIÓN DE UN ASISTENTE VIRTUAL CON INTEGRACION WEB Y WHATSAPP

Reconquista, 2024

ZACARIAS, LAUTARO NAHUEL.

UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Reconquista

DISEÑO, PROGRAMACIÓN E IMPLEMENTACIÓN DE UN ASISTENTE VIRTUAL CON INTEGRACION WEB Y WHATSAPP

Docente:

Prof. Santiago Richard

Asesor:

Ing. Walter Ariel Soto

Reconquista, 2024

DEDICATORIA

Quiero dedicar este logro académico y profesional a mis padres Silvana Zacarias y Walter Ariel Soto, pilares fundamentales en mi crecimiento como profesional y persona, a mis abuelos Isabel Vico y Eduardo Zacarias, personas que jamás me soltaron la mano, guiaron y acompañaron.

A mi novia Sasha Bais quien siempre apoyo mis horas de dedicación.

A mis amigos, compañeros, y todos los que estuvieron y están de alguna manera.

AGRADECIMIENTO

A Dios por darme fortaleza y sabiduría para llegar a la meta.

A mis Padres y Abuelos por brindarme su apoyo y siempre confiar en mí.

A la UTN-FRRq. por ofrecerme la posibilidad de formarme como profesional.

Al Docente de catedra por construir el conocimiento necesario en sus estudiantes.

Al Docente Asesor por el asesoramiento requerido para la producción del proyecto.

RESUMEN

En el presente proyecto se realiza el diseño, programación e implementación de un asistente virtual desarrollado con el servicio de Google Cloud llamado Dialogflow Essentials, que es un framework de procesamiento de lenguaje natural. El mismo se integra de manera efectiva tanto en sitios web como en la plataforma de mensajería instantánea de WhatsApp con la utilización de la biblioteca Venom-Bot en un entorno de Node.Js.

Con el asistente implementado en el sitio web de la UTN-FRRq. y mediante un contacto de WhatsApp, usando palabras claves definidas, se puede establecer una conversación a base de pregunta/respuesta.

Palabras claves: Asistente virtual, WhatsApp, web, framework.

INDICE

1. INTRODUCCIÓN

A partir de octubre del año 2023 y hasta concluir el ciclo de cursado, se realizaron talleres de prácticas profesionales en la UTN-FFRq. para que los estudiantes que se encontrasen en fase de finalización de su carrera o con sus proyectos finales en redacción, puedan llegar a su conclusión. En el mismo se dio la posibilidad de desarrollar una aplicación, misma que se trabajó en una de las cátedras de la carrera. Para el caso los estudiantes involucrados en el presente proyecto durante el cursado de la catedra no tenían una aplicación, por lo que se aprovechó la propuesta de la Regional.

Siguiendo la consigna de desarrollar una "Pagina con menús desplegables, Preguntas Frecuentes, chatbot que responda automáticamente", se decidió centrar la mayor parte del trabajo en el desarrollo del asistente virtual con el objetivo de acompañar y guiar a los usuarios mientras navegaban por la página.

Bajo estas directrices, se desarrolló un bot simple utilizando Dialogflow, que se basó en interpretar el mensaje del remitente, buscar palabras clave y recorrer listas configuradas para proporcionar respuestas basadas en la información obtenida. Aunque el bot tenía capacidades avanzadas, como mantener conversaciones con contexto y leer base de datos, su función principal era guiar a los estudiantes y usuarios en sus "Preguntas Frecuentes".

Con una estructura básica y algo de información, se consultó a la Regional Reconquista para obtener lo que denominaban "Preguntas Frecuentes" y así poder construir un sistema más completo sin duplicar la información existente.

A pesar de que la universidad ya cuenta con su sitio web, se creó una versión simplificada del mismo, incorporando un menú adicional llamado "FAQs" o "Preguntas Frecuentes". Esta estructura de contenido referente a preguntas frecuentes se cargó en la web mediante Github, donde se pruebo la interacción con el agente virtual.

Después de completar la estructura básica del bot y construir un modelo de página en el que se ejecute, se exploraron las integraciones del agente fuera de la web. Dado que la mayoría de los estudiantes preferirían comunicarse por WhatsApp, se implementó el mismo agente virtual como contacto del sistema de mensajería mencionado.

Tras finalizar la integración, el bot pudo responder a las preguntas configuradas tanto en la web como en un contacto de WhatsApp.

En resumen, este proyecto se centra en la creación de un asistente virtual que permanece como un elemento adicional dentro de la página de la UTN-FRRq, constituyendo una sección nueva de la misma, además, se integra con un contacto de WhatsApp, el cual puede cambiar en cualquier momento, y se analiza el tipo de preguntas que se realizan a través del entorno trabajado.

1.1. Objetivos

1.1.1. Objetivo General

- Diseñar, programar e implementar en el sistema de la UTN-FRRq un asistente virtual desarrollado con el servicio de Google Cloud llamado Dialogflow Essentials, que es un framework de procesamiento de lenguaje natural. Que integre de manera efectiva tanto sitios web como en la plataforma de mensajería instantánea de WhatsApp con la utilización de la biblioteca Venom-Bot en un entorno Node.Js

1.1.2. Objetivos Específicos

- Optimizar la eficiencia y accesibilidad al abordar las preguntas frecuentes de estudiantes y usuarios, mejorando la experiencia al proporcionar un asistente virtual capaz de responder rápidamente, eliminando la necesidad de navegación extensa por la web, correos electrónicos o visitas presenciales.

- Controlar un entorno de entrenamiento del chatbot donde se registren y almacenen las consultas realizadas, capturando información relevante como el contenido de las preguntas, la frecuencia de consultas y el contexto de uso, evaluando así la efectividad que tendrá el sistema en términos de mejora de la precisión y relevancia de las respuestas proporcionadas por este.

- Reducir la carga administrativa del personal de la universidad al ofrecer un sistema con la capacidad de responder automáticamente las consultas de estudiantes, reduciendo los tiempos de respuestas y la resolución de problemas básicos.

- Orientar a nuevos usuarios que ingresen al sitio web proporcionando información sobre procesos de inscripción, servicios, cursos y otros aspectos relevantes acerca de la universidad.

- Mejorar la experiencia del estudiante al brindar respuestas rápidas y útiles frente a preguntas comunes, mejorando así la percepción y la interacción con la universidad.

- Satisfacer una necesidad de la universidad al cubrir una propuesta dada por la misma de manera flexible y escalable, que pueda adaptarse a diferentes escenarios de prueba en el futuro.

2. METODOLOGIA

- Se lleva a cabo una investigación sobre diversas herramientas disponibles para el desarrollo de un asistente virtual. El objetivo es identificar una solución que sea flexible, escalable y fácil de usar. En este proceso se pone especial atención en los siguientes aspectos clave:

  - Integración fácil con distintos tipos de servicios.

  - Interfaz intuitiva y sencilla.

  - Procesamiento de Lenguaje Natural Avanzado.

  - Desarrollo Multiplataforma.

  - Rápida implementación y amplia escalabilidad.

  - Entrenamiento continuo del modelo.

  - Plan gratuito con suficientes funcionalidades para iniciar.

- Tras realizar la investigación para tener una herramienta clara para el desarrollo, se busca en la documentación provista por la misma herramienta donde se enfoca en localizar fortalezas y limitaciones para un desarrollo y posterior integración. Se pone especial atención sobre:

  - Compatibilidad e integración con código fuente propio y con recursos externos.

  - Posibilidad de tomar valores preestablecidos y modificarlos en producción.

  - Personalización interna y externa del agente.

- Con los datos obtenidos de la investigación se procede con el diseño, armado y programación de plantillas básicas de preguntas/respuestas.
- Después, se inicia el agente donde se recopilan errores sobre el funcionamiento.

Se analizan los resultados, y para los casos que no sean los esperados se vuelve al paso 2, se consulta la documentación y con la nueva información recolectada se vuelve a realizar el armado con sus modificaciones necesarias.

3. CONTEXTUALIZACIÓN DEL PROBLEMA

La comunicación y resolución de dudas en el entorno universitario son parte de los problemas que atraviesan los estudiantes, por lo que hay que navegar, enviar correos, llamar o visitar personalmente la facultad puede causar pérdida de tiempo y energía para los estudiantes. Estas dificultades no solo afectan la experiencia del usuario, sino que también repercute directamente en la productividad académica en el marco de brindar información.

Para abordar eficazmente este problema, surge la necesidad de una solución innovadora que proporcione respuestas rápidas y precisas, mejorando así la experiencia del usuario y elevando la productividad en el entorno académico.

En este contexto, la implementación de un asistente virtual se presenta como una solución estratégica y eficiente. Un asistente virtual tiene el potencial de proporcionar respuestas inmediatas a preguntas frecuentes, eliminando la necesidad de procesos más tradicionales y, por lo tanto, optimizar el tiempo y los recursos de los estudiantes. Esta solución no solo mejora la eficiencia en la comunicación, sino que también contribuye a una experiencia universitaria más fluida y satisfactoria.

Al explorar el desarrollo de un asistente virtual, el objetivo principal es abordar las deficiencias actuales en la comunicación y la resolución de dudas. La implementación de esta solución busca ofrecer respuestas rápidas y precisas, mejorando la accesibilidad a la información necesaria.

Al iniciar el proyecto se investigaron herramientas disponibles para desarrollar agentes virtuales, para identificar una solución flexible escalable y fácil de usar. Durante este proceso, Dialogflow emergió como una opción destacada debido a su potente procesamiento de lenguaje natural, su integración con Google Cloud, su capacidad para desarrollar de manera efectiva en entornos web y mensajería instantánea y por último su costo y curva de aprendizaje.

La necesidad de una solución que no solo fuera efectiva en el sitio web de la universidad, sino también en la plataforma de mensajería WhatsApp, llevó a la elección de Dialogflow junto con la biblioteca Venom-Bot en un entorno Node.Js. Esta combinación permitiría una implementación cohesiva y eficiente tanto en la web como en WhatsApp.

El diseño y programación del bot con Dialogflow y la posterior integración en la página web de la UTN-FRRq se llevaron a cabo considerando las características específicas de la universidad y las necesidades de los estudiantes. Este enfoque permitiría proporcionar respuestas rápidas y precisas a preguntas frecuentes, mejorando así la experiencia general del usuario.

En conclusión, la implementación de un asistente virtual surge como respuesta a los desafíos de comunicación y resolución de dudas en el entorno universitario. Este enfoque promete mejorar la accesibilidad a la información y optimizando la experiencia

del usuario. La elección de Dialogflow se fundamenta en su potente procesamiento de lenguaje natural, integración versátil y costos accesibles. La combinación de estas herramientas en un entorno Node.Js permite una implementación cohesiva en WhatsApp, abordando así las distintas necesidades de los usuarios y mejorando la eficiencia global de la comunicación universitaria.

3.1. Introducción a Dialogflow

Dialogflow es un servicio de procesamiento de lenguaje natural (NLP) desarrollado por Google Cloud, utilizado para crear interfaces de usuario de conversación, como chatbots y asistentes virtuales capaz de interactuar con usuarios en diferentes canales de comunicación.

La elección de Dialogflow para el presente proyecto no solo se basa en su potente motor de NLP, sino también en su capacidad de adaptarse a los requisitos específicos de nuestro entorno universitario. A lo largo de esta sección, se explorarán las características clave de Dialogflow que hacen que esta herramienta sea la opción ideal, destacando su relevancia en el desarrollo de nuestro asistente virtual y las consideraciones que influyeron en esta elección.

3.2. Introducción a Características Principales de Dialogflow

En la creación de un asistente virtual con esta herramienta hay opciones para configurar su correcto uso, cada elemento es vital para crear una experiencia de usuario fluida y efectiva, mejorando la capacidad del asistente para comprender, contextualizar y responder con precisión las necesidades y preguntas de los usuarios.

- INTENTS (Intenciones): Son el componente fundamental de Dialogflow, representa una “intención” del usuario, cada intent se diseña para capturar la naturaleza de las preguntas o solicitudes del emisor, el entorno puede utilizar aprendizaje automático para asignar las expresiones del usuario a los intents creados, si bien esa facilidad pueda resultar en una especie de atajo a veces es necesario que manualmente direccionemos el “mensaje recibido” hacia una “intención”.

  Un intent se compone de 6 elementos, los cuales 2 son fundamentales para el correcto funcionamiento del mismo:

  El orden de importancia de los elementos que componen los INTENTS para este proyecto es:

Orden de importancia	
1	Training Phrases.
2	Responses.
3	Events.
4	Action and Parameters.
4	Fulfillment.

Tabla 3.1. Orden de importancia de Intents
Fuente: Elavoración propia

- Frases de Entrenamiento: frases que podemos esperar de los usuarios, las cuales ejecutarán el "intent". Se pueden definir todas las que se crean necesarias, pero se debe tener cuidado que mantengan cierta similitud o que sean sinónimos ya que, al aumentar la cantidad de intents, ergo, la cantidad de frases, se pueden llegar a repetir haciendo que el bot confunda su respesta.

  Gracias al motor NLP cuando un usuario dice algo similar a una frase de entrenamiento Dialogflow lo hace coincidir con el intent, haciendo que no sea necesario crear una lista exhaustiva, el entorno rellenará la lista con expresiones similares. Ejemplo de Intent de Bienvenida:

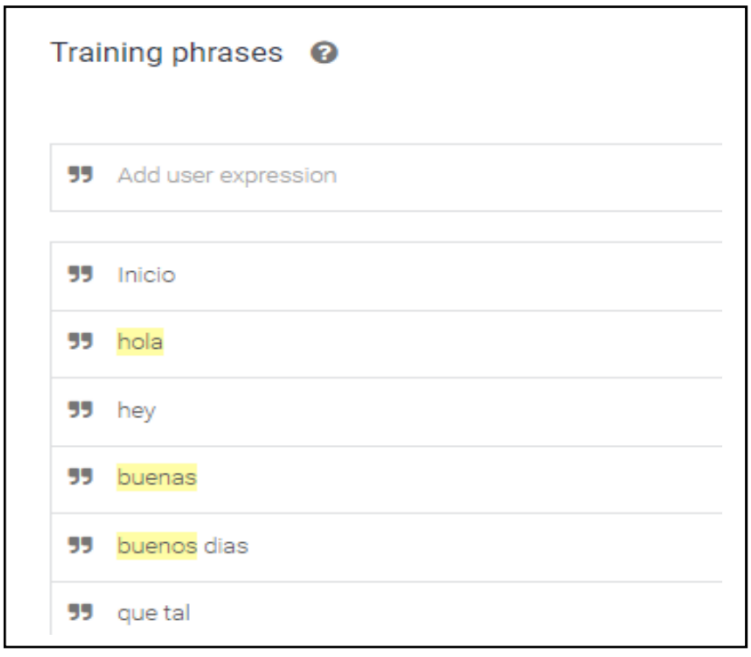


Imagen 3.1. Training phrases
Fuente: Elaboración propia

- Respuestas: Es la sección clave dentro de un Intent donde se definen las respuestas que el agente proporcionará cuando se detecta esa intención particular. Dentro de este apartado se usaron dos variaciones para responder los mensajes:

Text Response: Respuestas basadas únicamente en texto plano, que son especialmente útiles para capturarlas en otros entornos como lo sería Node.JS, ya que todas las plataformas pueden dar una respuesta de texto básico.

Custom Payload: Es una forma avanzada de proporcionar respuestas personalizadas a los usuarios, ya que permiten una mayor flexibilidad en la presentación de información, su estructura se basa en formato JSON, en el cual se puede organizar cada apartado de la misma respuesta, agregar imágenes, botones y eventos que desencadenen en otras respuestas.

Ejemplo Intent “7_DF_HorariosFechas”:

Usuario envía un mensaje que se emparejará con una de las siguientes expresiones definidas o un sinónimo de las mismas.

El bot responderá con las respuestas establecidas en orden ascendente.

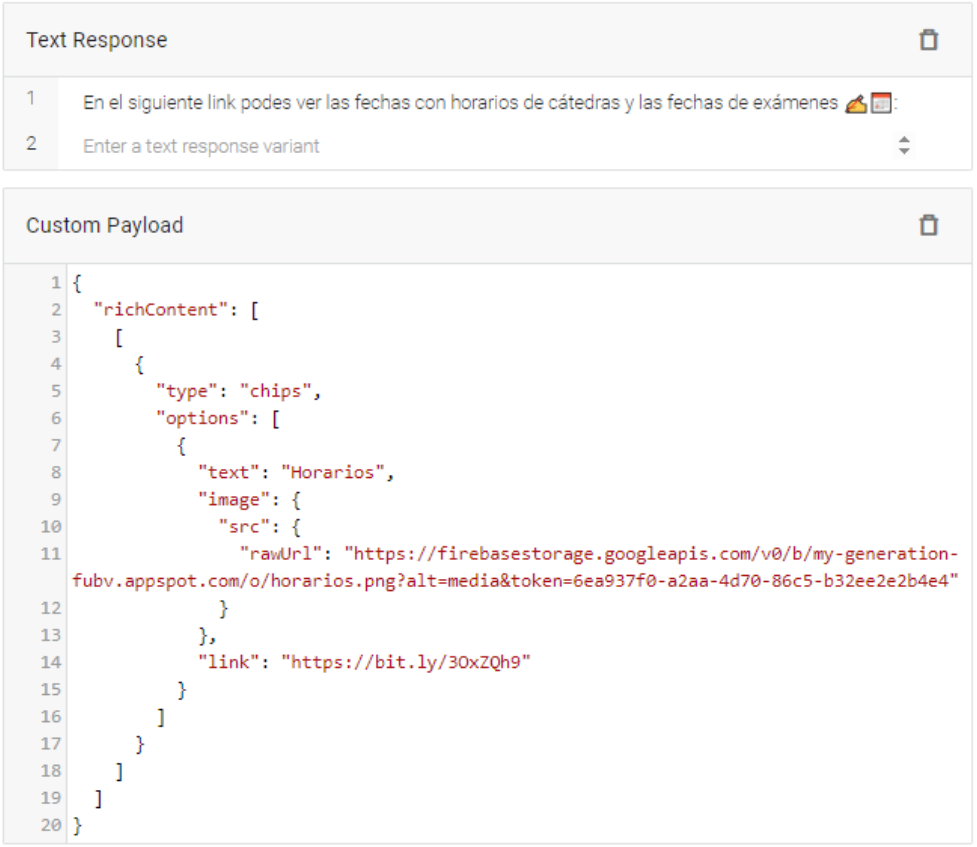


Imagen 3.2. Respuesta de texto y enriquecida
Fuente: Elaboración propia

Primero se enviará la respuesta basada en texto, posteriormente el mensaje con estructura JSON, que para él se mostrará lo siguiente:

RichContent: Una respuesta enriquecida del tipo “chips”, que es un botón en este entorno, y dentro del “chip” tendrá los atributos “img”, “text” y “link”, para ubicar una imagen, titulo y un enlace respectivamente.

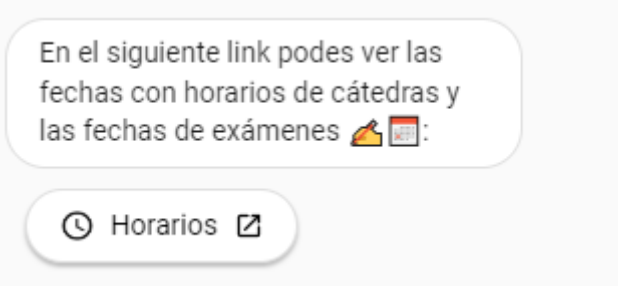


Imagen 3.3. Visualización de un mensaje enriquecido
Fuente: Elaboración propia

- Eventos: Son la forma de desencadenar un intent de manera programática en lugar de depender únicamente de la detección de una entrada del usuario. De esta manera se puede controlar el flujo de la conversación y dirigir específicamente a un intent sin que el usuario proporcione una entrada. Por ejemplo:

  El intent “Default Welcome” que es el inicio de conversación del bot devuelve un mensaje de texto plano y un mensaje de texto enriquecido en forma de lista que a su vez son botones, estos desencadenarán un evento que llamará a otro intent sin necesidad de que el usuario deba enviar una expresión para lograr el emparejamiento.

Custom Payload

```
{
  "richContent": [
    [
      {
        "event": {
          "parameters": {},
          "languageCode": "",
          "name": "carrerasEvent"
        },
        "subtitle": "Para saber sobre las carreras disponibles e información básica de las mismas",
        "title": "Carreras ",
        "type": "list"
      },
      {
        "type": "divider"
      },
      {
        "type": "list",
        "subtitle": "Infórmese sobre sus notas, consulte el estado de su cursado y más.",
        "event": {
          "name": "autogestionEvent",
          "languageCode": "",
          "parameters": {}
        },
        "title": "Autogestión "
      }
    ]
  ]
}
```

Imagen 3.4. Estructura de respuesta enriquecida con eventos asociados
Fuente: Elaboración propia

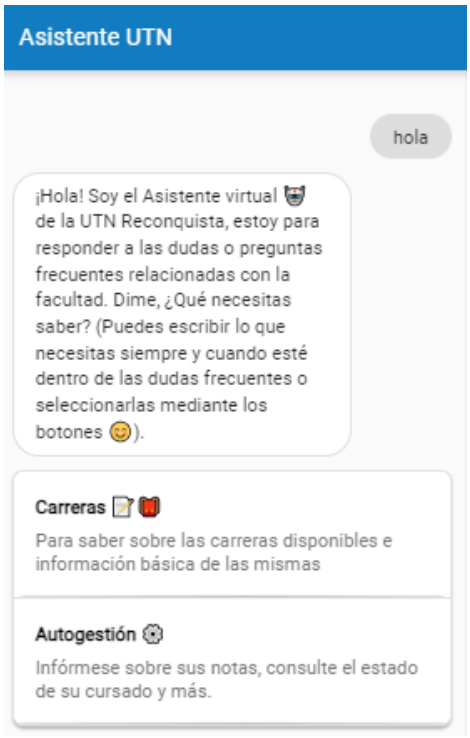


Imagen 3.5. Visualización en chat de imagen 3.4
Fuente: Elaboración propia

Se llama la intención de bienvenida y al hacer clic sobre uno de los botones se llama el evento definido dentro de alguna intención.

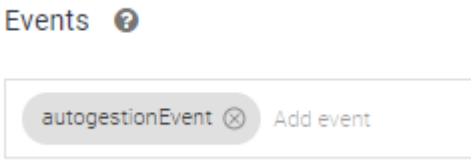


Imagen 3.6. Segundo evento de imagen 3.4
Fuente: Elaboración propia

Además de funcionar como un evento escuchador dentro de la integración web puede ser llamado, enviado y controlado directamente con programación resultando, que según las acciones del usuario dentro de la página web el bot pueda acompañar en todo momento con sugerencias.

4/5 - Acciones y Parámetros/ Cumplimientos: Son herramientas que pueden lograr que las interacciones con el chat bot sean más dinámicas, extrayendo datos de una intención, asignarles valores y guardarlos en un servidor externo propio del mismo entorno, especialmente útiles en sistemas de bots de reservas de mesas en un restaurante, o de pedidos.

6- Integrations (Integraciones): Es el apartado que refiere a la capacidad de integrar el agente de Dialogflow con diversas plataformas y canales de comunicación, entre ellos los más conocidos son Messenger de Facebook, Telegram, Skype, Twitter y páginas webs.

Para este proyecto las opciones que se evaluaron fueron las “basadas en texto” (WhatsApp no incluida entre ellas):

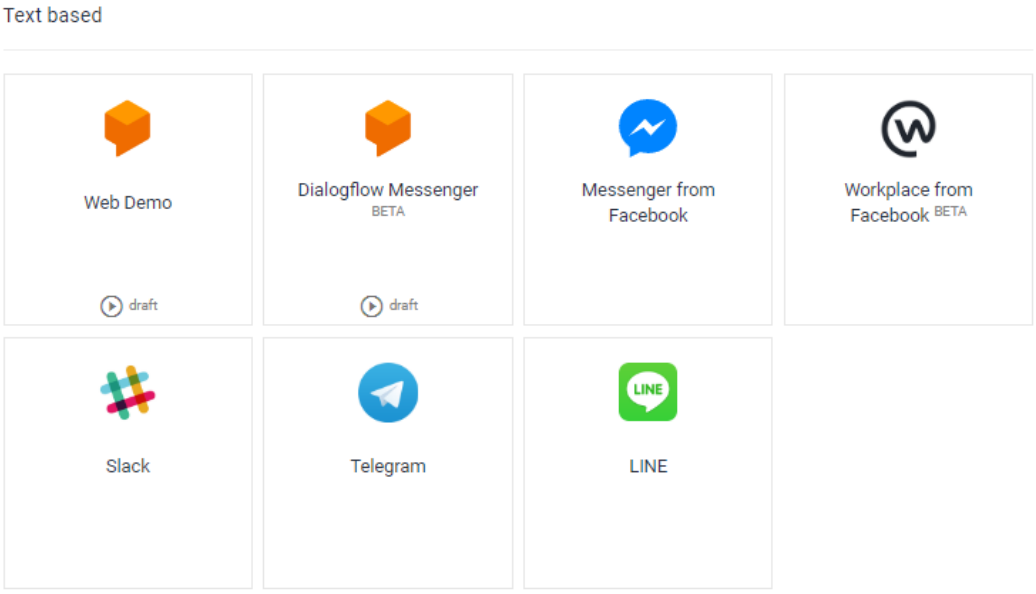


Imagen. 3.6. Integraciones oficiales de DialogFlow

Fuente: Elaboración propia

Ya que son las que nos permiten mantener cierta coherencia entre los contenidos que se entregarán.

7- History y Training (Historial y Entrenamiento): Secciones de administración del asistente virtual, en ambas se registran todas las interacciones entre los usuarios y el agente, mientras que en el historial se muestra la conversación general en la sección de entrenamiento se puede definir y refinar las intenciones y respuestas del agente, por ejemplo cuando el agente conteste con una respuesta que no es la correcta se puede redirigir a la intención correcta o agregar una nueva intención en base a lo que el usuario ha enviado.

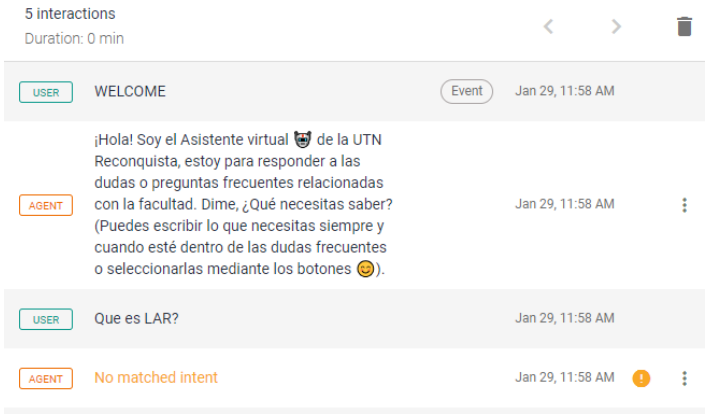


Imagen. 3.6. Ejemplo de historial “No Matched” 1

Fuente: Elaboración propia

En este ejemplo del historial se puede observar cómo es una conversación, donde se activa la intención “WELCOME” al entrar a la página web, responde con lo definido y la conversación sigue:

Usuario		Agente
“Que es LAR?”	=>	No matched (Respuesta genérica, ya que indica que no sabe cómo responder)

Tabla 3.2. Ejemplo de Default Fallback Intent
Fuente: Elaboración propia

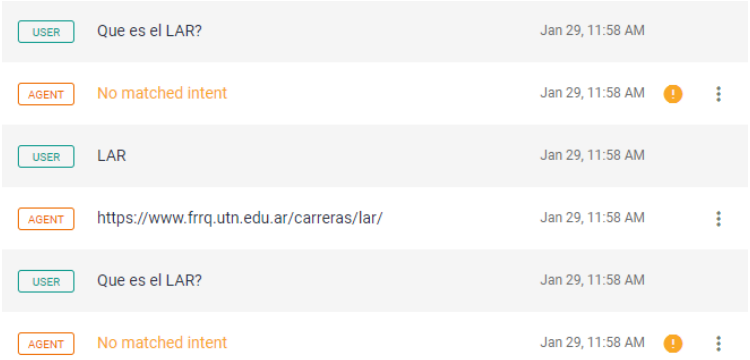


Imagen. 3.7. Ejemplo de historial “No Matched” 2

Fuente: Elaboración propia

Sin embargo, escribiendo únicamente “LAR” el agente sabe responder, en estos casos una vez se comprueba el historial descubriendo respuestas que no logren emparejar se debe utilizar la sección de entrenamiento, por ejemplo:

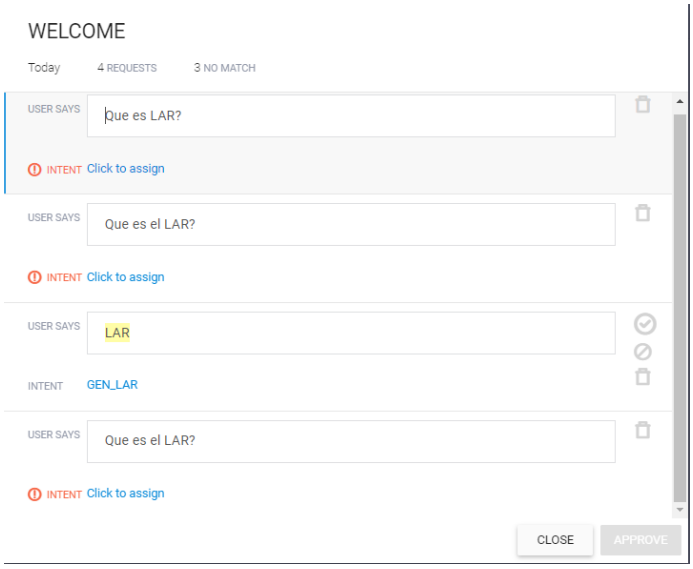


Imagen 3.8. Sección de entrenamiento
Fuente: Elaboración propia

Similar al historial, la sección de entrenamiento tiene una lista con las conversaciones que se realizaron, pero al desplegarlas, se aprecia de manera resumida lo que diga el usuario versus lo que responda el agente. En este caso el bot responde a la palabra clave “LAR” con el intent emparejado “GEN_LAR”, que contesta un texto plano y un enlace que redirige hacía la sección de la página web de la universidad donde se habla acerca de Lic. en Administración Rural, pero cuando se usa la palabra clave combinada con conectores o prefijos no es capaz de emparejar una intención.

Para solucionar esto lo único que se debe hacer es tocar sobre “Click to assign”, y se te desplegará el menú con todas las intenciones y se debe colocar la que corresponda o crearla en caso de ser necesario:

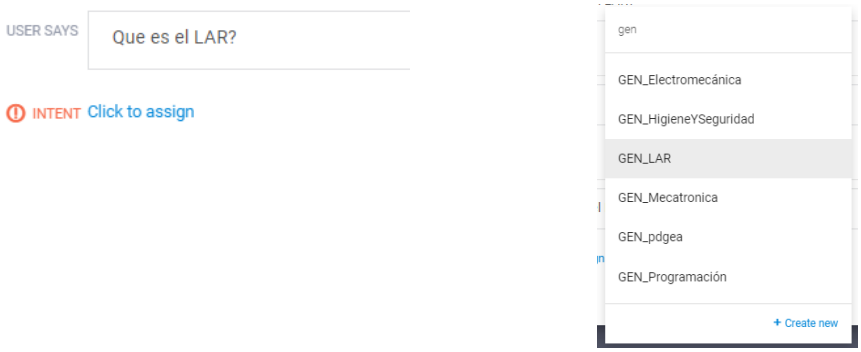


Imagen. 3.9. Asignación de intención
Fuente: Elaboración propia

Posteriormente solo hace falta escribirle un par de veces hasta que empareje correctamente:

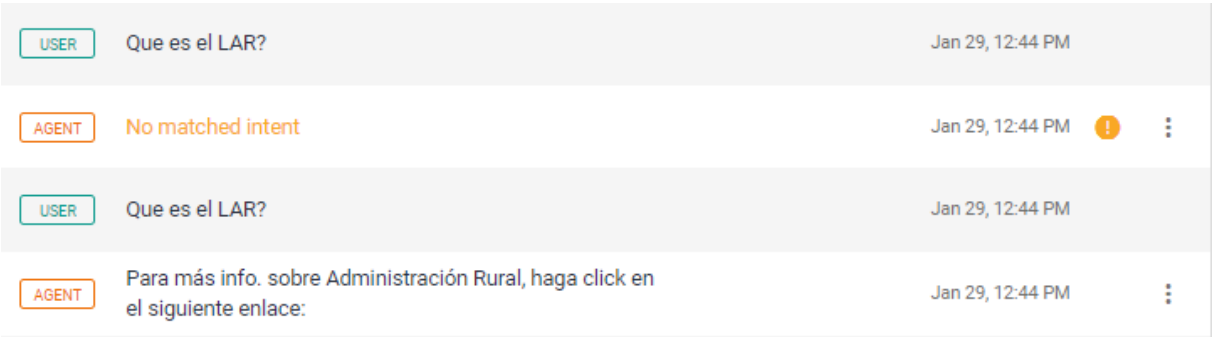


Imagen. 3.10. Ejemplo de Historial “No Matched” corregido

Fuente: Elaboración propia

4. INTEGRACIÓN DEL AGENTE EN UN SITIO WEB

Teniendo claro cómo se estructura la construcción del bot en el entorno Dialogflow, lo próximo es lograr una correcta integración con una página web, ya mencionado anteriormente, el entorno de Dialogflow nos ofrece una característica única de integrar el bot con un widget ya creado:

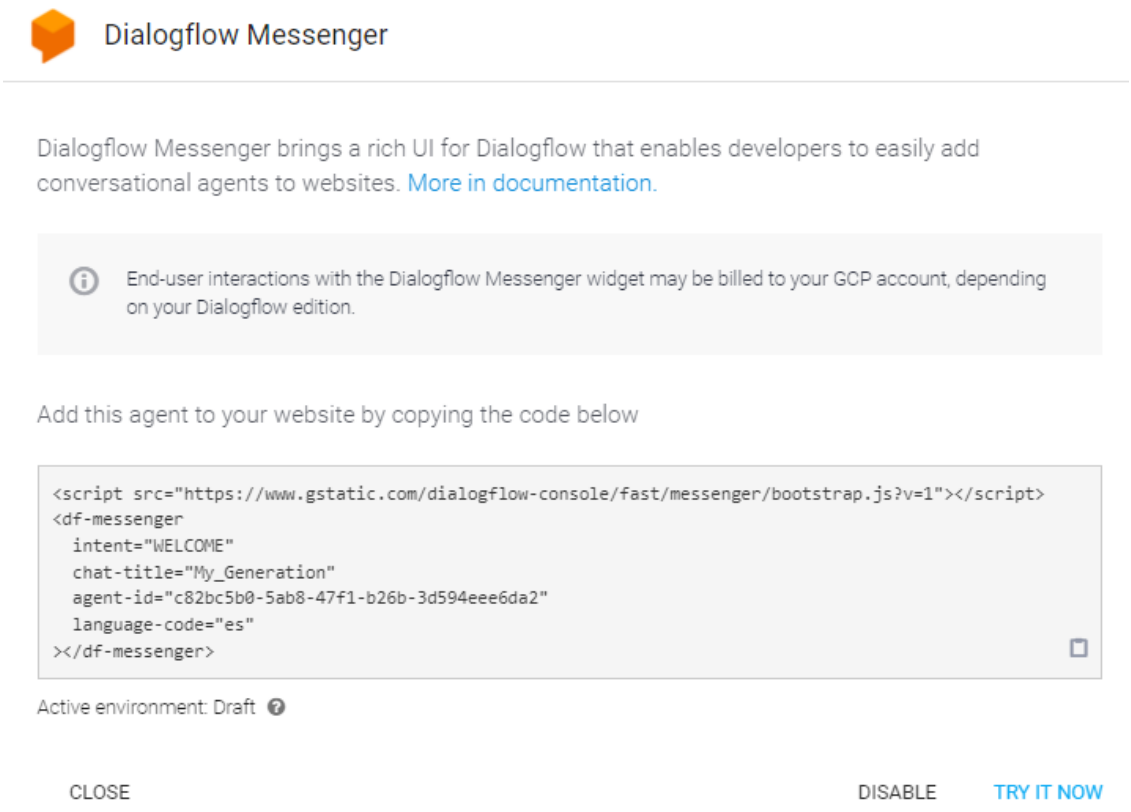


Imagen. 4.1. Etiqueta para integrar agente en una página web
Fuente: Elaboración propia

En la cual nos devuelve una etiqueta de código que se debe incorporar a un supuesto sitio web para que el bot se pueda usar. Además, con el botón “TRY IT NOW” se genera una vista previa que muestra cómo se vería en ejecución.

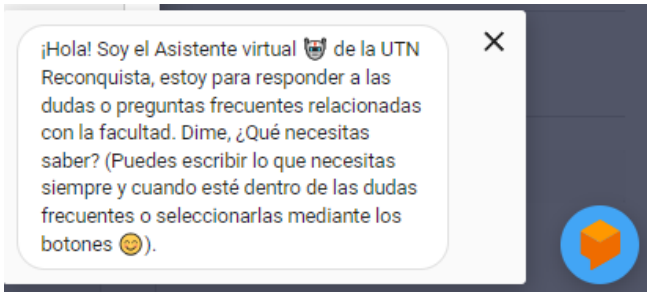


Imagen 4.2. Widget Predeterminado de Dialogflow 1
Fuente: Elaboración propia

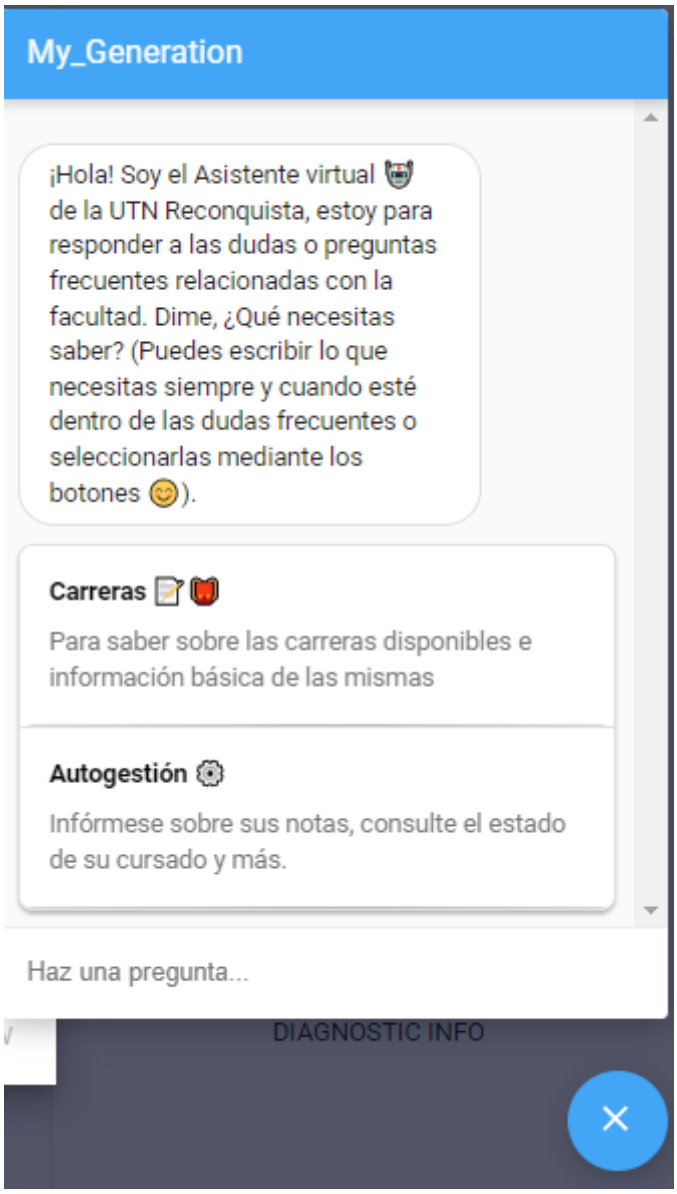


Imagen 4.3. Widget predeterminado de Dialogflow 2
Fuente: Elaboración propia

Para entender donde se debió colocar esa etiqueta de la sección de integraciones se debe ubicar el lenguaje de programación, si se trabaja con uno de compilado como lo es JAVA o C# se necesita un Entorno de Desarrollo Integrado (IDE), para aplicaciones, si es lenguaje interpretado por el navegador como es el caso, con un editor de código es suficiente.

5. INTRODUCIÓN A VISUAL STUDIO CODE

Visual Studio Code es un editor de código fuente de Microsoft que se utiliza como herramienta fundamental en la construcción de aplicaciones web.

Es el editor de código más famoso ya que tiene una amplia compatibilidad con lenguajes y frameworks web ya que ofrece un buen resaltado de sintaxis y funcionalidades de autocompletado que ayuda al desarrollo de código de manera más eficiente al sugerir automáticamente las palabras claves que se pueden usar, o no remarcarlas en caso de no tener el contexto de poder usarlas.

Visual Studio Code tiene integraciones muy potentes con los frameworks de desarrollo web, que en este caso se incorporó Node.js, esto proporciona a los desarrolladores trabajar de manera más eficiente al proveer de acceso directo a comandos y funcionalidades específicas de cada herramienta desde el propio editor.

Este editor de código para el desarrollo es tan simple como crear una carpeta en el directorio que se desee y arrastrarla al programa, o abrir la carpeta desde ese momento, cuando tengas tu carpeta de trabajo incluida se pueden crear archivos necesarios para el desarrollo.

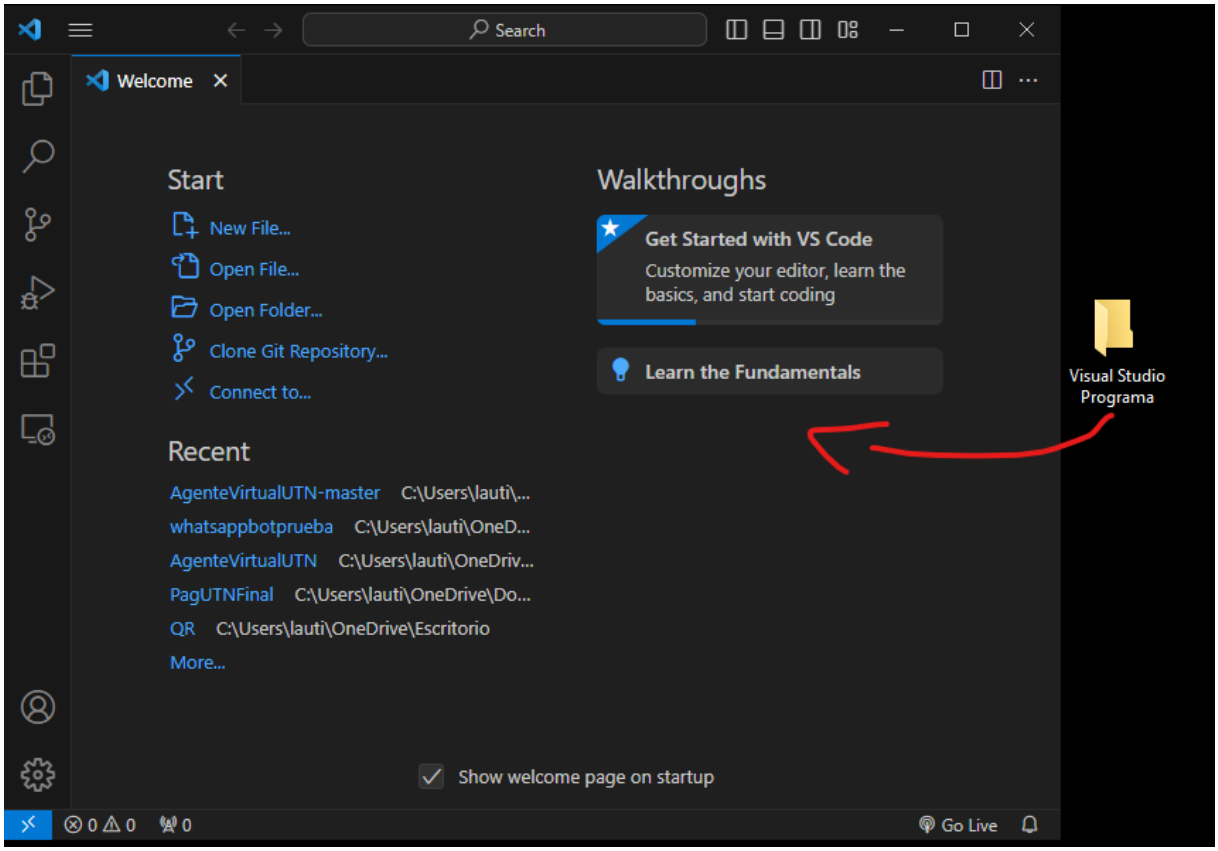


Imagen 5.1. Ejemplo de iniciar un proyecto en Visual Studio 1
Fuente: Elaboración propia

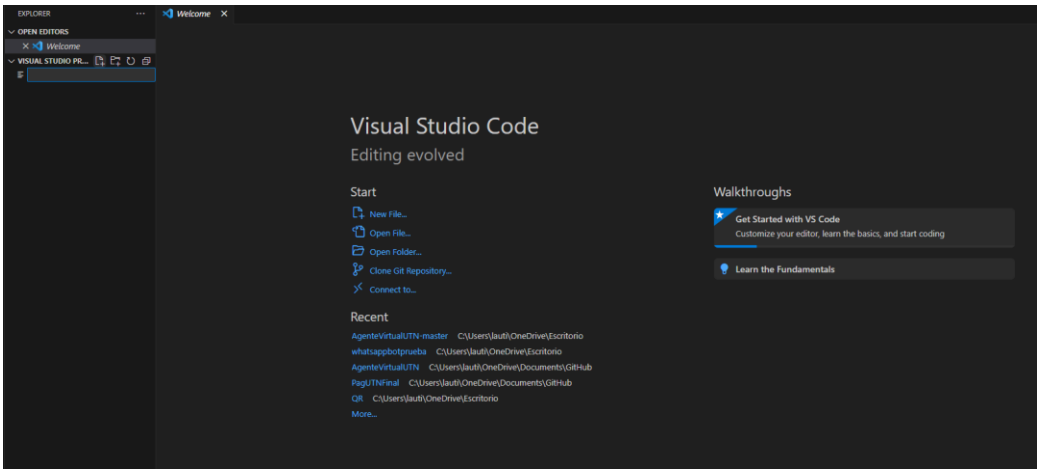


Imagen 5.2. Ejemplo de iniciar un proyecto en Visual Studio 2
Fuente: Elaboración propia

Luego de establecido la carpeta de trabajo solo se necesita crear un nuevo archivo (New File), colocar el nombre pertinente y la extensión sobre la que se quiera trabajar; .html para estructuras de contenido; .css para estilos; .js para funciones de programación.

5.1. Introducción a Desarrollo Frontend

Para comprender lo que es un desarrollo de una página web se debe saber que a esa práctica se le llama Programación Frontend que refiere a la puesta en marcha de:

a. HTML (HyperText Markup Language): Lenguaje de marcado utilizado para la estructuración y organización del contenido de una página web. Define los elementos y su disposición en la página, como los encabezados, párrafos, botones, enlaces, imágenes, etc. El archivo suele ser “nombre.html” y la forma de su uso se basa en usar los símbolos “< >” para las etiquetas que conformarán el contenido, se lo suele aprender como si fuese una especie de tetris, donde cada etiqueta <> es una caja donde puede haber cajas en su interior que se las puede modificar usando el lenguaje de estilos CSS.

   Ejemplo:

```
<body>
        <header></header>
        <nav></nav>
        <section>
          <article></article>
          <aside></aside>
          <aside></aside>
        </section>
        <section></section>
        <footer></footer>
</body>
```

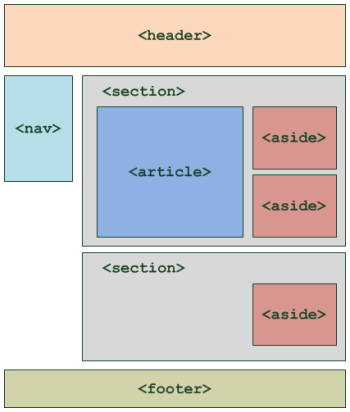


Imagen. 5.2. Estructura visual de etiquetas HTML
Fuente: Estructura de cajas de una página web. Desarrollo de Aplicaciones Web. Universidad de Murcia

b. CSS (Cascading Style Sheets): Es el lenguaje de diseño que se utiliza para dar estilos al contenido HTML, ya que permite controlar aspectos visuales como el color, tipografia, diseño general, espacio y otros aspectos de presentación de la página web. Para usar correctamente los estilos se debe vincular un archivo llámese "index.css" a tu respectivo .html: "<link rel="stylesheet" href="index.css">", luego simplemente citar la etiqueta que quieras modificar, en este caso el "<body>" que es el cuerpo de todo el contenido: body { font-family: Arial, sans-serif; margin: 0; padding: 0; background-color: white }, logrando así una fuente Arial quitando dejando "0" de espacio alrededor del contenedor con un color de fondo "White".
c. JS (Javascript): Es el lenguaje de programación de alto nivel que se utiliza para hacer las páginas web logrando que estas sean interactivas, dinámicas y se puedan agregar funcionalidades extras como animaciones, validaciones de formularios manipulación del DOM (), comunicación con servidores y bases de datos y muchísimas más interacciones de usuario.
Para mayor comprensión de las 3 herramientas en conjunto un ejemplo:
Estructura generada automáticamente de html con un botón en el cuerpo.

index.html | index.css | index.js

index.html > html > head > script

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7      <link rel="stylesheet" href="index.css">
8      <script src="index.js"></script>
9  </head>
10 <body>
11     <button id="boton">Verde</button>
12 </body>
13 </html>
```

Imagen. 7.3. Estructura genérica de código HTML
Fuente: Elaboración propia

Estilos para el botón basándose en “id=boton”, donde se le agregar “padding” espaciado desde el interior del bloque, tamaño de fuente, color de fondo y de texto, se le quita el borde y se lo redondea, y por último añade unas animaciones básicas de puntero y transición de color cuando se posa por encima (:hover).

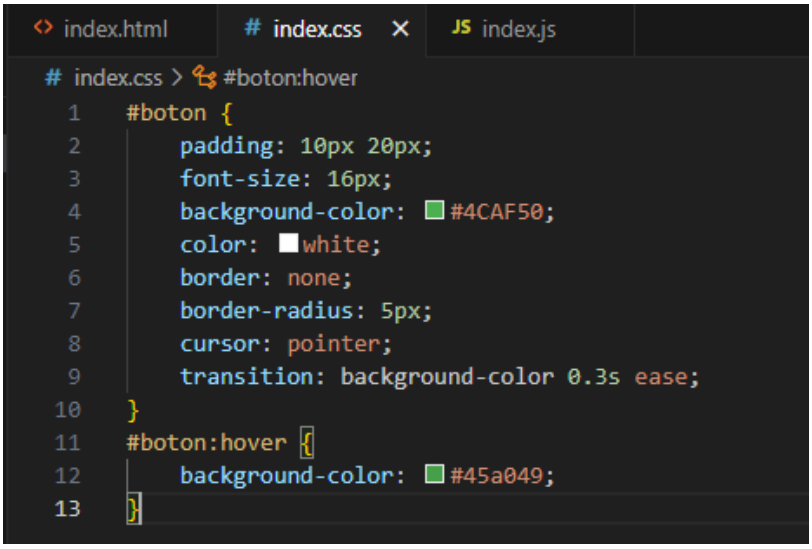


Imagen. 7.3. Estructura de código CSS
Fuente: Elaboración propia

Por último, el desarrollo de la programación con javascript

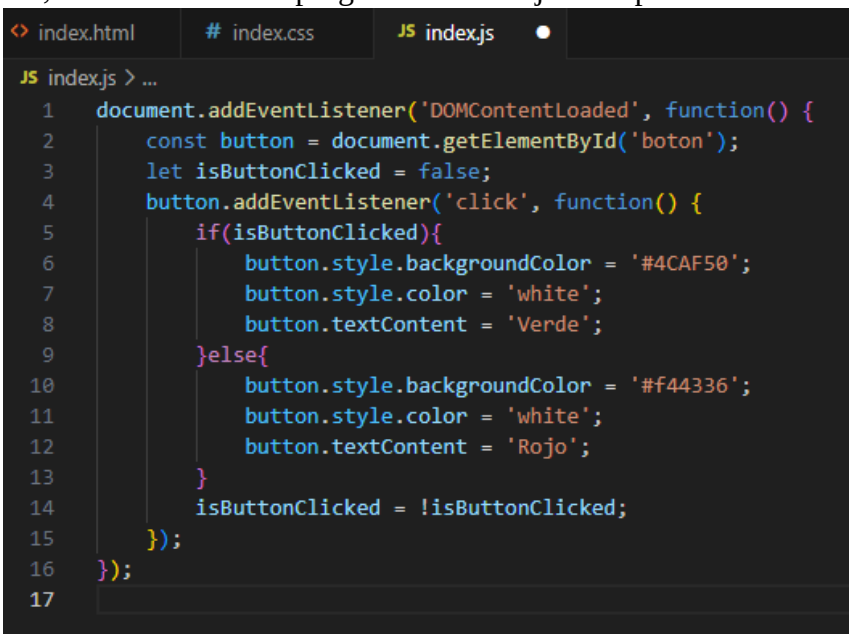


Imagen 5.2. Programación con javascript
Fuente: Elaboración propia

En este código lo podemos leer como un condicional dentro de una función que “escucha” una variable y todo dentro de otra función que vuelve a “escuchar” pero esta vez a todo el contenido de la página “DOM”.

Las partes relevantes para el posterior desarrollo y uso de Javascript son las que están de color amarillo, “addEventListener” como función que escucha algo y “getElementById” como método que obtiene un elemento del “DOM” para usarlo o asignarlo a una variable. En una lectura rápida de la ejecución de este “script” podríamos decir:

Línea	Descripción
1 a 3	Se escucha la carga completa del DOM, para saber cuándo se terminaron de cargar los recursos para poder interactuar y modificarlos. Se obtiene el elemento con el “id=’boton’” y se lo guarda en una variable “button”. Y por último se crea una variable “isButtonClicked = false” que se la podrá utilizar para revertir cambios.
4	Se escucha la variable “button” que a su vez tiene como valor la etiqueta “boton” del html cuando se ejecuta un evento “click” sobre ella.
5 a 13	Se establece un condicional basado en “Si ‘isButtonClicked’ es verdadero”, se establecerá un color de fondo Verde con el texto “Verde”, si es falso hará lo mismo, pero cambiando el color de fondo por Rojo y su texto por “Rojo”.
14	Se asigna un valor a la variable booleana por su contraria en ese momento para poder volver a ejecutar el script repetidamente.

Tabla. 5.2. Pasos de ejecución de código JavaScript
Fuente: Elaboración propia

Resultado del proceso:

Designación	Descripción
	Un botón verde que al pasar el mouse por encima cambiará ligeramente su tonalidad y que al hacer click pasará a verse rojo
	

Tabla. 5.2. Ejecución de código JavaScript de forma visual
Fuente: Elaboración propia

Con estos conceptos usamos HTML para crear contenedores que se apilaran con más contenedores dentro, CSS para cambiar tipografía, tamaño, color y lo que sean aspectos de diseño y JS para realizar acciones según eventos o interacciones del usuario.

5.2. Introducción a la Consola de Desarrollador del Navegador

Todos los navegadores web traen incorporada una herramienta que es fundamental para el desarrollo web, esta es la Consola de Desarrollador, que permite depurar, probar y analizar el comportamiento de una página web en tiempo real. Usando como ejemplo la página realizada en el punto anterior para acceder a la consola solo debemos presionar F12 en nuestro teclado o clic derecho y luego Inspeccionar:

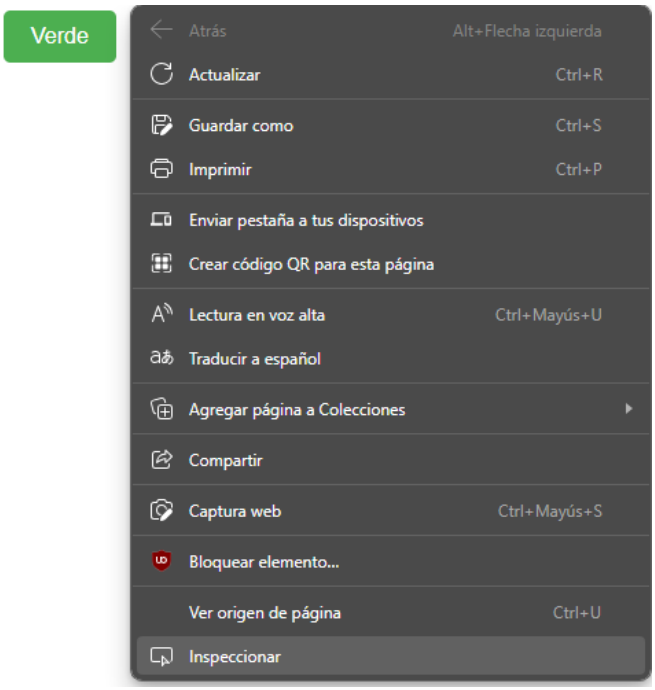


Imagen. 5.3. Ejemplo de inspeccionar elementos en navegador
Fuente: Elaboración propia

Luego el navegador nos desplegara una sección donde podremos acceder a todos los archivos públicos, estructura de la página web, diseños y a la consola donde se registrarán eventos o errores que sucedan durante la ejecución:

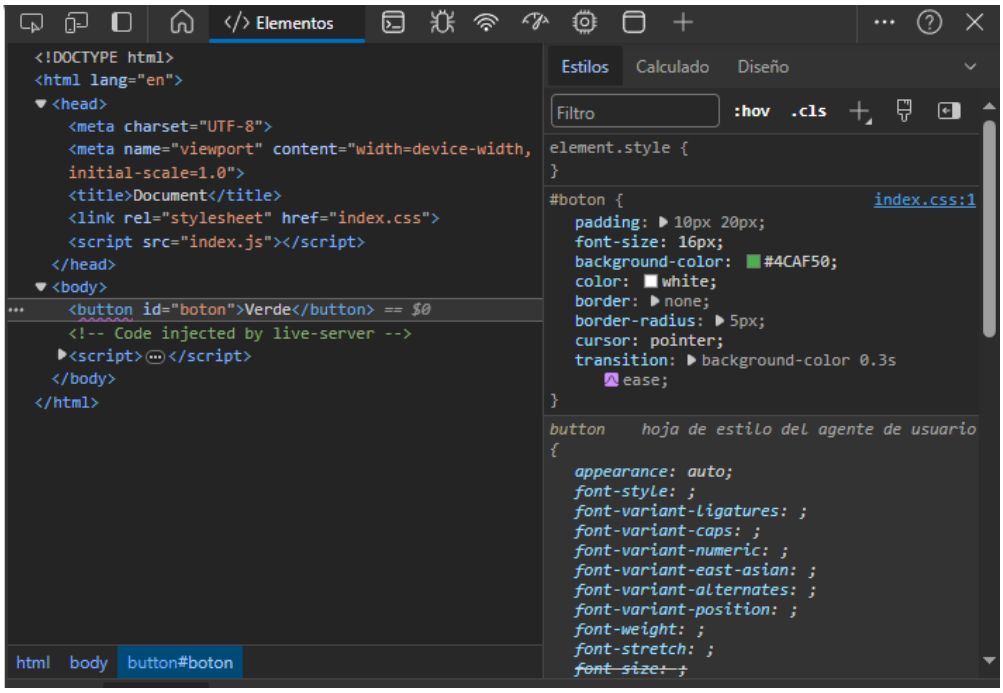


Imagen. 5.3. Consola de desarrollador del navegador
Fuente: Elaboración propia

Por el contrario del lenguajes de programación como C++, C# o Java que se los debe trabajar en un entorno de desarrollo completo donde antes de cada ejecución se debe compilar el código y no debe haber errores hasta lanzarse el programa, en desarrollo web no existe la compilación, ya que JavaScript es un lenguaje interpretado por el navegador, dando por resultado que cada vez que abras tu aplicación o página web haya errores o no se ejecutará igual, de esta manera la Consola de Desarrollador del Navegador es una herramienta clave a la hora de obtener errores, modificar acciones o estilos en tiempo real.

5.3. Introducción a Git y Github para desarrollo en onjunto

Estas herramientas son una parte importante del trabajo y del desarrollo en colaboración, además de llevar un control sobre los cambios.

Git es un sistema de control de versiones que permite rastrear los cambios en los archivos de un proyecto a lo largo del tiempo, pudiéndose controlar que cambios se hicieron, cuando y por qué. Mientras que GitHub por su parte es una plataforma para alojar el código que utiliza Git para controlar las versiones del desarrollo, permitiendo colaborar, alojar y revisar código.

A continuación, se nombran las características más importantes y las formas en que se usó para este proyecto Git y Github brevemente ya que estas herramientas dependen del usuario.

Para usar estas herramientas es necesario tener una cuenta registrada en github.com e instalar GIT en tu dispositivo, una vez hecho esos pasos es necesario crear un "repositorio" de código en tu cuenta de Github y darle los correspondientes permisos de lectura/escritura a tus colaboradores.

A partir de ese momento Github te generará un enlace que puedas vincular tu proyecto (en este caso en el editor Visual Studio Code) con tu repositorio:

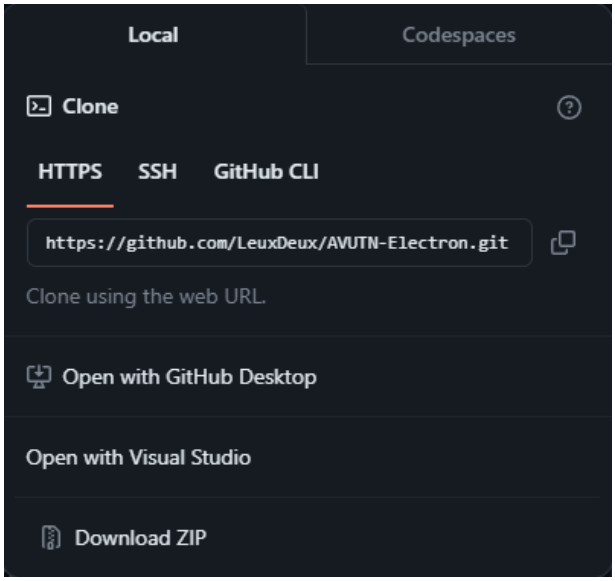


Imagen. 5.4. Apartado de vinculación de GitHub
Fuente: Elaboración propia

En este paso existen dos opciones:

a) El repositorio tiene código ya sea porque lo hizo otra persona, es de tu colaborador o se ha subido manualmente el código como un archivo comprimido.
b) El repositorio está vacío.

En el caso a) se debe "clonar" el repositorio de Github y luego inicializar Git en ese proyecto, que, en otras palabras, se traerán todos los archivos y código del repositorio a tu editor de código, luego al inicializar Git en el mismo directorio del proyecto se te creará una carpeta "oculta" llamada '.git' que contendrá todos los cambios que vayas realizando a medida que uses los comandos correspondientes.

En el caso b) que es el que nos halla, solo se deberá enlazar el proyecto con el enlace que proveerá Github:

En la "terminal/consola" de nuestro editor de código una vez instalado GIT en nuestro positivo deberemos:

- Inicializar GIT: 'git init'. Esto nos creará la carpeta .git donde se registrarán los cambios.
- Añadir el origen del repositorio y configurar una rama: 'git remote add origin "https://url..."', lo que nos vinculará nuestro repositorio local (.git) con el

repositorio remoto (GitHub) y ‘git branch -M main’ que se ocupará de que los cambios se realicen en una rama específica de tu repositorio, lo cual es útil cuando tienes un mismo programa con diferentes ramas ya sea porque es una aplicación que está en continua ejecución y deba haber cambios en tiempo real o por simple gestión de cambios.

- Opcionalmente se puede verificar que esté correctamente vinculado con: ‘git remote -v’. Lo que nos permitirá saber si hemos llevado a cabo la conexión.
- Una vez realizada la conexión correcta ya podemos empezar a realizar cambios en el código, donde nuestro editor nos marcará:

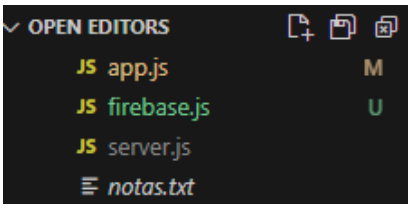
GitHub en el proyecto	
	Amarillo: Cambios que no se han guardado. Verde: Archivos nuevos que no están registrados. Gris: Archivos ignorados que adrede se han configurado para no guardarse y posteriormente subirse al repositorio remoto. Blanco: Archivos sin cambios realizados.

Table. 5.--. Ejemplo de cómo actúa GitHub en el proyecto
Fuente: Elaboración propia

- Opcionalmente se puede crear un archivo “.gitignore” e ir citando los archivos que se deben ignorar a la hora de controlarlos o subirlos al repositorio, esto suele ser útil cuando el repositorio es público (la mayoría de los casos) y se trabajan con credenciales privadas

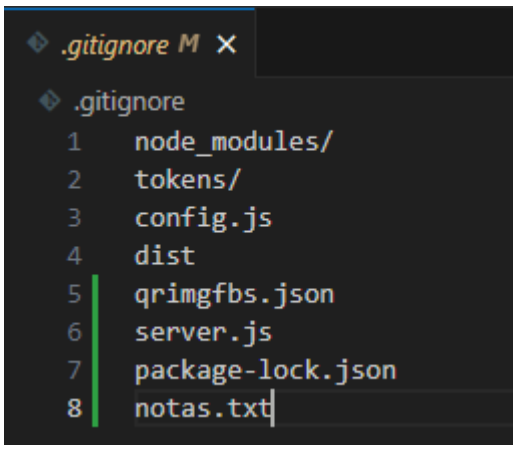


Imagen. 5.5. Archivos que serán ignorados por GitHub
Fuente: Elaboración propia

- Una vez realizados cambios para subirlos a tu repositorio local (Git) debes ejecutar:‘git add nombreArchivo’ en caso de que sea un solo archivo el que quieras guardar, o el más común ‘git add .’ el cual guardará todos los cambios de archivos salvo los guardados en tu “.gitignore”. Seguidamente debes ejecutar ‘git commit -m “He realizado estos cambios…”’, para llevar que en un futuro si debes volver pasos atrás veas los cambios que hiciste sumado a un mensaje con una descripción de lo que se cambió.
- Al lograr registrar tus cambios localmente el siguiente paso será ejecutar un “push”: ‘git push origin main”, siendo “main” el nombre de tu “branch” o rama principal que hayas configurado. Esto subirá todos los cambios que has hecho al repositorio remoto GitHub, modificando en el todo el codigo viejo con el nuevo y generandote apartados donde puedes volver “versiones” atrás para revisar cambios. Con estos pasos se logra utilizar Git y GitHub correctamente para un desarrollador y multiples lectores.
- En el caso de tener colaboradores o ser un desarrollo entre varios integrantes se debe utilizar el comando ‘git pull origin main’, que descargará todos los cambios del repositorio remoto y los fusionará automáticamente con tu codigo local.

5.4. Armado del Sitio Web

Teniendo en claro el cómo funciona el agente virtual de Dialogflow con su integración, las herramientas de desarrollo VS Code, la consola de desarrollador del navegador y nuestros lenguajes elegidos estamos preparados para simular una versión de la página de la UTN-FRRq con el bot ya integrado.

Siguiendo la consigna anteriormente establecida en la Introducción “Pagina con menús desplegables con Preguntas Frecuentes, y chatbot que responda automáticamente”, se optó por tomar parte de la estructura de diseño de la página original y replicarla como muestra de lo que debería ser la página, agregando un nuevo botón “desplegable” que te debería llevar a una sección donde se establecieron algunos botones con texto desplegable que deberían contestar a algunas dudas frecuentes sumado al botón del bot que acompañaría todo el recorrido de la exploración de la página.

Para no mostrar extensos pasos de código, estilos y métodos para organizar contenido se simplificará a mencionar lo que se ha hecho y se proveerá el código en el repositorio: https://github.com/JoaRome123/PagUTNFinal.

5.5. Pasos para la construcción de la sección “FAQs”

5.5.1. Obtención de colores y estructura de los navegadores

5.5.1.1. Obtención de código de colores.

Ingresamos a la url de la página https://www.frrq.utn.edu.ar/ y usaremos la consola de desarrollador para obtener el código de los colores predominantes de la página.

Imagen. 7.5. Visualización de la página de UTN-FRRq y su consola de desarrollador del navegador
Fuente: Elaboración propia

Se obtiene una vista de esta forma, donde en orden izquierda derecha veremos la página con una resolución menor para poder mostrar la consola con más detalle, la estructura de contenido .html y al seleccionar algún elemento en el último panel de la derecha podremos ver las características y estilos que conforman cada elemento.

Si bien puede parecer abrumador en un comienzo la cantidad de texto y propiedades, esto se debe a que este sitio web fue creado con un sistema llamado WordPress que genera páginas webs en base a plantillas reduciendo el código al mínimo, y este mismo entorno establece todas las clases y propiedades internamente.

Dentro de la consola hay un botón que asemeja a una flecha, podemos presionar ese botón o apretar la combinación “Control + Shift + C” y se nuestro cursor se convertirá en un selector de contenedores:

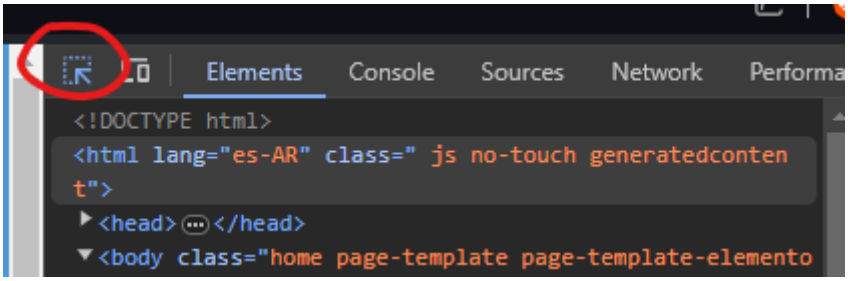


Imagen 5.6. Herramienta de inspección de elementos del navegador
Fuente: Elaboración propia

Con el selector activado donde posemos el mouse nos dará información sobre la etiqueta y propiedades básicas, mientras que en la consola nos llevará a la parte del código donde se encuentra dicha etiqueta a su vez mostrándonos todo lo que lleva consigo y sus propiedades heredadas y propias.

Imagen 5.7. Propiedades obtenidas al posar el mouse sobre un elemento
Fuente: Elaboración propia

Lo que nos muestra el cursor al usar el selector de etiquetas: ‘div.row.valign” con su tamaño y un par de propiedades básicas.

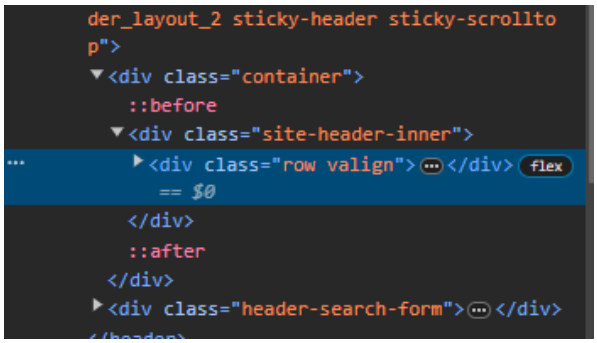


Imagen. 5.8. Elemento seleccionado para la inspección
Fuente: Elaboración propia

En la consola nos lleva a donde se definió esa etiqueta y podremos ir leyendo las propiedades hasta encontrar el color principal de la página web, si no está en ese elemento puede heredarse de un elemento padre.

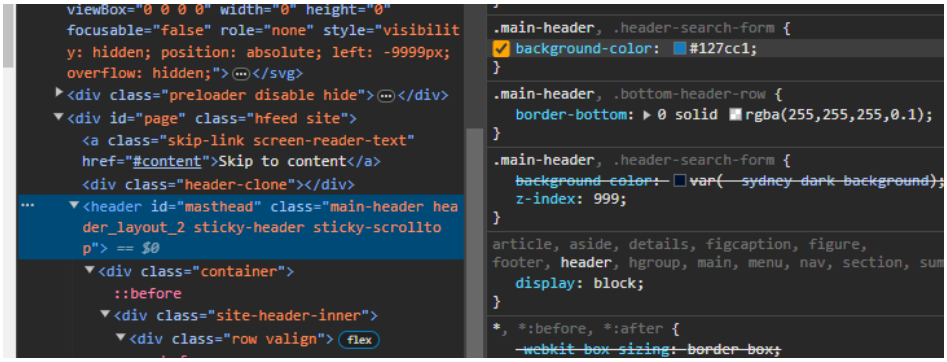


Imagen 5.9. Elemento inspeccionado y sus respectivas características en la consola de desarrollador de navegador
Fuente: Elaboración propia

En este caso se tuvo que subir 3 elementos padre hasta la etiqueta principal “<header>” que conforma toda la parte superior de la página para encontrar el color

"#127cc1" que es el principal, si bien existen más métodos para capturar el color de una manera más fácil de esta forma nos ayuda a comprender como está estructurado el contenido de la página.

5.5.1.2. Replica de la interfaz en página local.

Para este paso se creó una estructura similar a la página de la UTN-FRRq llamada FAQs con su sección de "Preguntas Frecuentes" donde habrá varios botones con un poco de texto a modo de "Lorem Ipsum" donde se contesten algunas dudas.

Imagen. 5.10. Menú de navegación adaptado
Fuente: Elaboración propia

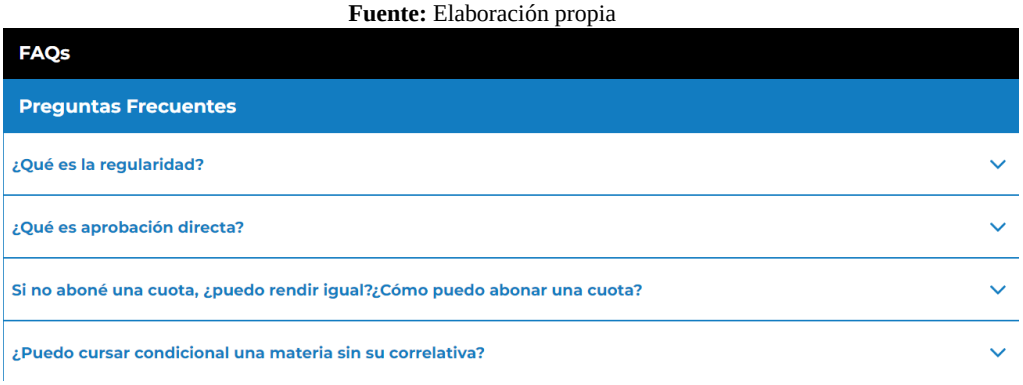


Imagen 5.11. Ejemplo del cuerpo de la página
Fuente: Elaboración propia

Para ver más a detalle visitar: FAQs (joarome123.github.io), donde se estableció la página usando las herramientas de GitHub.

5.5.2. Integración del Agente con estilos agregados

Para incorporar el bot correctamente como ya se mencionó, se debe colocar la etiqueta de Dialogflow al final del código .html:

```
    <div class="bot">
        <script src="https://www.gstatic.com/dialogflow-
console/fast/messenger/bootstrap.js?v=1"></script>
        <df-messenger
            intent="WELCOME"
            chat-title="Asistente UTN "
            agent-id="c82bc5b0-5ab8-47f1-b26b-3d594eee6da2"
            language-code="es"
            chat-icon="img/LogoUTN copy.png">
        </df-messenger>
    </div>
```

Imagen. 7.12. Colocar nombre de imagen
Fuente: Elaboración propia

Donde se agregó una imagen de la araña de UTN en “chat-icon”

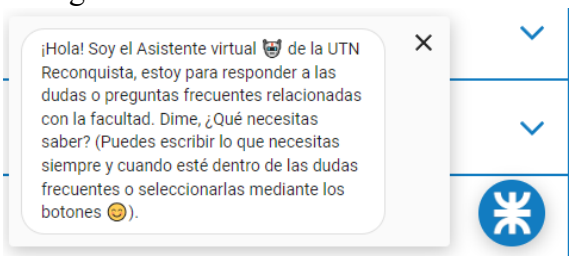


Imagen. 5.13. Colocar nombre de imagen
Fuente: Elaboración propia

Se utiliza el color “#127cc1” anteriormente obtenido para el color principal del bot:

```
<style type="text/css">
    df-messenger {
        --df-messenger-button-titlebar-color: #127cc1;
    }
</style>
```

Por último, se crea una función que detecta automáticamente la resolución del espectador y en base a ese valor se escala la interfaz del bot para que se vea bien en todas las resoluciones, que se puede observar en el código fuente de la página anteriormente mencionada del repositorio remoto de Github, por lo extenso de la misma: https://github.com/JoaRome123/PagUTNFinal

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
<script>
    $(document).ready(function() {
        window.addEventListener('dfMessengerLoaded', function (event) {
            var $r1 = document.querySelector("df-messenger");
            var $r2 = $r1.shadowRoot.querySelector("df-messenger-chat");
            var $r3 = $r2.shadowRoot.querySelector("df-messenger-user-input");
            adjustChatHeight();
            function adjustChatHeight() {
                var windowHeight = window.innerHeight;
                var chatHeight = calculateChatHeight(windowHeight);
                var sheet = new CSSStyleSheet;
                sheet.replaceSync(`div.chat-wrapper[opened="true"] { height: ${chatHeight}px }`);
                $r2.shadowRoot.adoptedStyleSheets = [sheet];
            }
            function calculateChatHeight(windowHeight) {
                if (windowHeight < 768) {
                    return 450;
                } else if (windowHeight < 1080) {
                    return 600;
                } else {
                    return 650;
                }
            }
            window.addEventListener('resize', function() {
                adjustChatHeight();
            });
        });
    });
</script>
```

Imagen 5.14. Colocar nombre de imagen
Fuente: Elaboración propia

5.6. Integración del Agente con Whatsapp

Una vez construido el bot con dialogflow e integrado a una web se decidió a su vez llevarlo a la red de mensajería más grande WhatsApp, ya que es el lugar ideal para tener un asistente virtual que conteste automáticamente las preguntas.

Para llevar a cabo esta acción se necesitó de NODE.JS como entorno de ejecución, librerías públicas para métodos más complejos como lo son "venom-bot", "dialogflow" y "mysql2" y por último el framework llamado ElectronJS para lograr llevar aplicaciones web al escritorio, entre otros módulos con menos relevancia que se irán presentando a medida que sean necesarios.

5.6.1. Introducción a NODE.JS

Toda la integración del asistente virtual se sostiene gracias al entorno de NodeJS, que es una plataforma de software de código abierto para construir aplicaciones web escalables y de alto rendimiento, nos ayuda con el desarrollo en tiempo real en base a eventos, en la integración de diferentes módulos que ahorraron código en todo el proyecto.

Generalmente cuando creamos un archivo '.js' donde se programan funciones de cualquier tipo para llevarlas a la ejecución debe ser leída e interpretada por el navegador en base a un archivo '.html', pero con NODE.JS tenemos la posibilidad de ejecutar independientemente cada archivo '.js' y lograr una devolución en nuestro propio editor de código. También es de importancia nombrar que la instalación de módulos es una parte fundamental del proyecto que solo se logra usando NODE, ejemplo:

```
"axios": "^1.6.7",
 "dialogflow": "^1.2.0",
 "express": "^4.18.2",
 "firebase-admin": "^12.0.0",
 "fs": "^0.0.1-security",
 "moment": "^2.30.1",
 "mysql2": "^3.6.5",
 "socket.io": "^3.0.4",
 "uuid": "^9.0.1",
 "venom-bot": "^5.0.21"
```

Dentro del directorio del programa existe el archivo "package.json", donde estarán todas las dependencias necesarias para la correcta ejecución de la aplicación, esta se va rellenando automáticamente en función de que agreguemos así también como la podemos ir modificando manualmente para reinstalar dependencias, básicamente funcionaria como un instalador de paquetes para que cierto programa funcione con normalidad. Por ejemplo, según estas dependencias podemos saber que el proyecto puede utilizar:

Dialogflow para manipular propiedades y eventos del mismo, FS o File System que nos permite manejar archivos del sistema operativo, Moment para capturar la fecha y hora de un evento, MySql2 para trabajar con base de datos, Venom-Bot para establecer

una conexión con WhatsApp entre otras. A veces puede ser que cierta versión no nos sirva por una u otra razón, en ese caso en vez de tener que ir hacía una página de terceros para instalar la correcta luego de desinstalar la actual usando NODE y el package.json solo se deberia cambiar, por ejemplo: “dialogflow: 1.2.0” por “dialogflow: 1.1.0” y volveríamos una versión atrás mientras que exista.

5.6.2. Importación de Módulos y Codigo Base

Para empezar a codificar nuestra aplicación debemos crear 3 archivos fundamentales para establecer las funciones y conexiones principales:

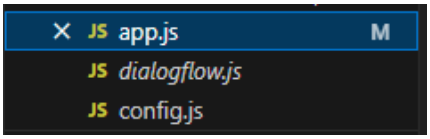


Imagen. 5.15. Archivos principales para la ejecución de la aplicación
Fuente: Elaboración propia

Designación	Descripción
config.js	Archivo privado donde las credenciales de conexión se establecieron como constantes.
dialogflow.js	En el cual se establecerá una conexión con el servicio del mismo nombre a través de Google Cloud además de verificar credenciales
app.js	Donde se ejecutarán las funciones principales directamente conectado con WhatsApp.

Tabla 5.--. Función de cada archivo en la aplicación
Fuente: Elaboración propia

Luego de creados los archivos principales, se deben importar los módulos pertinentes, en este caso “npm i venom-bot”, “npm i dialogflow”, “npm i uuid” y “npm i fs”, respectivamente: objeto para la conexión con whastapp, funciones para trabajar con dialogflow, generaciones de id únicos y funciones para interactuar con archivos del sistema.

Una vez creados los archivos principales e importados los módulos para empezar a trabajar sobre estas librerías se debe ir a su respectiva documentación que puede ser tanto en la página “npmjs.com” o en repositorios públicos de GitHub.

Para empezar, se debió ir al correo asociado al agente virtual en “console.cloud.google.com” y crear una nueva cuenta de servicio donde nos dará la opción de descargar sus nuevas credenciales a través de un archivo .json, el cual nos devolverá muchos valores importantes pero los que se usaron fueron:

Strings de credenciales
GOOGLE_CLIENT_EMAIL
GOOLE_PRIVATE_KEY
GOOGLE_PROJECT_ID
DF_LANGUAGE_CODE

Tabla. 5.--. Strings de credenciales
Fuente: Elaboración propia

Donde los primeros 2 valores corresponden a un correo y a una clave, mientras que los últimos dos corresponden al ID de nuestro proyecto donde está constituido el agente virtual y al lenguaje establecido del mismo.

Una vez establecidos estos valores se encerraron en un "module.exports" (anexo 1) que es una forma que tiene NODE, de exportar valores de constantes como Array para ser utilizados por otros archivos de nuestro proyecto. Y seguidamente se estableció el código en el que nos basaremos para establecer la conexión con Dialogflow al cual se le ha hecho modificaciones para trabajar en su desarrollo. Ver: Anexo 2.

Con los archivos "config.js" y "dialogflow.js" completos somos capaces de comunicarnos entre nuestro entorno de desarrollo, en este caso VS Code, y el asistente virtual. De esta forma a cualquier tipo de programa que pueda registrar un mensaje con JavaScript seremos capaces de obtener las respuestas de nuestro asistente y enviarlas por el método que sea, en ese punto las posibilidades son infinitas, pero se centró la atención en una conexión con WhatsApp. Para esto se conformó el archivo "app.js" que como su nombre lo indica es la aplicación principal del programa, la que se debe ejecutar y con ella todos los demás archivos.

Para la primera versión de "app.js" se usó una base de código proveída por diferentes usuarios de GitHub, y de la misma documentación de "Venom-Bot". En la cuál su funcionalidad se basa en la inicialización del objeto "venom" que creará la sesión virtual con WhatsApp luego de pasar por una verificación, una función "start(client)" que asocia el número del teléfono del remitente a una sesión única y envía los mensajes entrantes a Dialogflow para procesarlos, y por último se usa la función "sendMessageToWhatsapp()" para enviar el mensaje de respuesta obtenido hacia el remitente (anexo 3).

Con este código se logró que el programa sea capaz de establecer una conexión con WhatsApp, detectar cuando llega un mensaje y responder con lo establecido en el agente virtual, sin embargo, dentro de Dialogflow, existían respuestas enriquecidas donde se utilizaron imágenes y botones que no se pueden enviar hacía WhatsApp, por esto se desarrolló un nuevo archivo .js y una nueva función que empezaría a enriquecer la base del programa:

ResponseMappings.js, es un archivo donde se crea un mapeo a mano basado en el mensaje original que debería decir el agente virtual "original" por un mensaje adaptado para verse en WhatsApp "custom". Esta lógica (final anexo 3) ayudó en el caso de que se

emparejase un mensaje enriquecido del agente con el mensaje del remitente, ya que de ser el caso el bot no enviaría ningún mensaje por WhatsApp.

5.7. Conexión con Base de Datos MySQL y proceso de renderizado con ElectronJS

Al tener la aplicación funcionando y recibiendo mensajes de distintos usuarios se decide crear una base de datos con el gestor MySql WorkBench en dos tablas relacionadas:

Dato	Detalle

Contactos telefónicos

id_telefono	numero_telefono	cantidad_consultas
11	5493482617278	20
16	5493482530323	1
17	5493487676058	1
19	5493482543508	2
23	5493482207784	1
NULL	NULL	NULL

Table. 5.--. Tabla de Usuarios
Fuente: Elaboración propia

Entrada	Detalle

Consultas

id_consulta	id_telefo	numero_telefono	mensaje_entrante	respuesta_saliente	intent_emparejado	fecha	hora
00f13899-...	11	5493482617278	Donde puedo ano...	En el siguiente link pod...	7_DF_HorariosF...	2024-01-19	15:00:48
0187e555-...	17	5493487676058	Hola	¡Hola! Soy el Asistente ...	Default Welcome...	2024-01-19	15:12:22
37fbf02c-7...	11	5493482617278	Que es la regulari...	La regularidad es el tie...	1_DF_regularidad	2024-01-19	15:00:53
3fa8e0e2-...	11	5493482617278	Hola	¡Hola! Soy el Asistente ...	Default Welcome...	2024-01-19	19:48:43
891b6392-...	11	5493482617278	Hola	¡Hola! Soy el Asistente ...	Default Welcome...	2024-01-19	18:42:37

Table. 5.--. Tabla de Consultas
Fuente: Elaboración propia

Donde se consideró modificar el código de “app.js” para obtener los valores de las consultas y generar las funciones que envíen los valores a la base de datos. Ver: anexo 4

Una vez realizada la base de datos con su lógica se pensó en la manera más amigable que el usuario final pueda inicializar la aplicación, recibir datos y filtrarlos, para lograr esto se acopló el framework “Electron” al proyecto, que convierte toda la ejecución de los archivos en segundo plano en una aplicación de escritorio usando procesos de comunicación y renderizado.

Para trabajar con Electron se tuvieron que crear nuevos archivos para mantener la estructura y escalabilidad del proyecto.

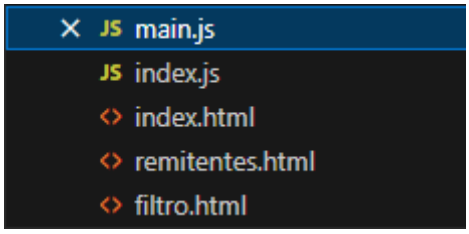


Imagen. 5.16. Archivos principales para la renderización
Fuente: Elaboración propia

Main.js: Configura la aplicación de Electron y controla su ciclo de vida, así como la comunicación entre procesos. Es el archivo que ejecuta todo el entorno visual, creando las ventanas principales de la aplicación, cargando los archivos HTML, definiendo manejadores de eventos para los mensajes enviados del proceso principal y lo más importante inicia un proceso secundario para ejecutar “app.js” manejando sus mensajes de salida y errores. Ver Anexo 5.

Index.js: Define las funciones “startApp()” y “stopApp()” que se encargan de enviar mensajes al proceso principal de Electron para iniciar y detener la aplicación, respectivamente. Esta capa es crucial para controlar la comunicación entre el proceso de renderización y el proceso principal de la aplicación, facilitando el inicio y la detención de esta, así como el manejo de eventos relacionados con la inicialización y los errores durante la ejecución.

Index.html: Es la ventana principal que se mostrará al iniciar la ejecución, en ella se muestra la imagen QR que se genera en “app.js”, la cual se debe escanear para poder iniciar la aplicación correctamente, funciona con un evento “setInterval(reloadImage, 15000);” que la recargará cada 15 segundos, adicional a eso contiene los botones “Iniciar” y “Detener” que ejecutarán el proceso secundario “app.js” o cerrarán toda ejecución, y un botón “Abrir Remitentes” que renderizará otro HTML. Ver Anexo 6 y 9.

Imagen. 5.17. Aplicación en ejecución
Fuente: Elaboración propia

Remitentes.html: Es una ventana secundaria que se ejecuta al presionar el botón “Abrir Remitentes” del “index.html” donde se define una estructura que muestra una lista de remitentes y controles para filtrarlos, por defecto al cargarse el “DOM” llama a la función “mostrarTodasLasConsultas()” almacenadas en la base de datos y las muestra en la tabla, ejecutada con un “setInterval()” con 10 segundos de demora que mantiene la lista actualizada. Al rellenar campos de filtrado y presionar el botón “Filtrar” se abrirá se ejecutará un evento que abrirá “filtro.html”. Ver Anexo 7.

Imagen. 5.18. Menú de filtrado

Fuente: Elaboración propia

Filtro.html: Es la ventana secundaria que se abre al ejecutar el evento en "remitentes.html" donde se mostrarán los datos según le filtro que se haya aplicado. Ver Anexo 8.

A cada una de las partes HTML que mostrarán el contenido se les codificó en su mismo archivo las funciones para separarlas de los bloques principales.

El siguiente diagrama presenta de forma simplificada las interacciones y conexiones entre los diversos componentes de la aplicación. Este gráfico tiene como objetivo proporcionar una representación visual de las relaciones y las comunicaciones entre los diferentes elementos que conforman el sistema, es fundamental para comprender la arquitectura y el flujo de datos que sustenta el funcionamiento de este.

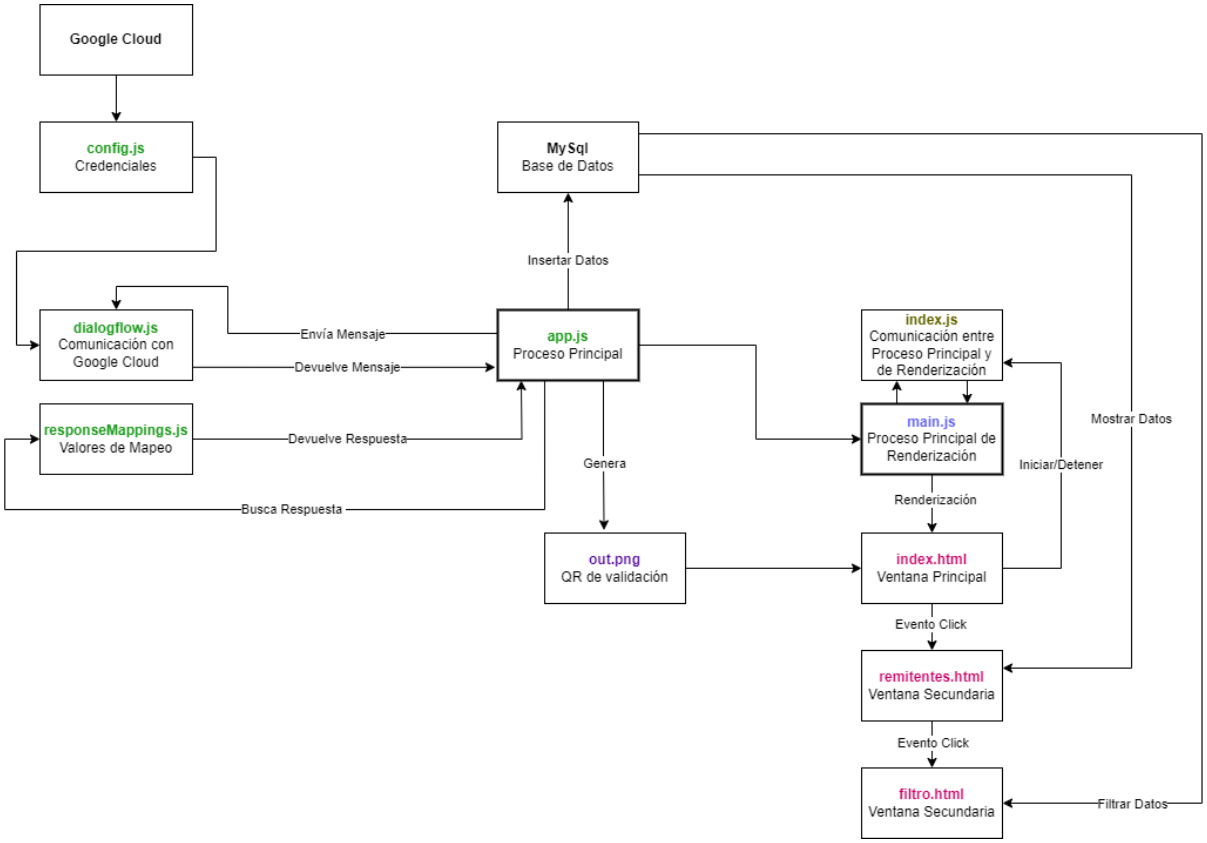


Diagrama 5.1. Interacciones y conexiones entre componentes de la aplicación
Fuente: Elaboración propia

6. RESULTADOS Y CONCLUSIONES

A nivel personal con la ejecución del proyecto se consiguió poner en práctica conceptos aprendidos tanto en el cursado de las diferentes cátedras de la carrera como los de capacitaciones extracurriculares.

A nivel profesional se consiguió desarrollar un agente virtual utilizando la plataforma Dialogflow que posea la capacidad de responder preguntas preconfiguradas y aprender de nuevas interacciones que podrá ser integrado en cualquier sitio web enriqueciendo la experiencia de navegación de los usuarios y se implementó una conexión con WhatsApp a través de tecnologías como Node.js, Electron.js y Venom-Bot que permitió recibir mensajes, responderlos con el agente y registrar los datos relevantes en una base de datos brindando una vision integral de las interacciones con los usuarios a través de este canal de comunicación. La UTN-FRRq podrá implementar esta tecnología como servicio complemento en la plataforma institucional.

La aplicación desarrollada cumple con lo propuesto en el plan de trabajo inicial y tiene la virtud de que puede adaptarse a diversos escenarios y cambios en su estructura y entorno gracias a su arquitectura flexible y a los frameworks en continuo desarrollo que sustentan su funcionamiento.

.

7. REFERENCIA BIBLIOGRÁFICA

Sitios web

Dialogflow ES. (2018). *Guías, ejemplos y referencias para compilar agentes, por Google for Developers.*

https://cloud.google.com/dialogflow/docs?hl=es-419

Electron JS. (2023). *Documentación para el desarrollo de aplicaciones de escritorio utilizando tecnologías web estándar, por Microsoft Corporation.*

https://www.electronjs.org/es/docs/latest/

Firebase Storage Cloud. (2018). Cloud Storage para Firebase para desarrolladores de apps, *por Google for Developers.*

https://firebase.google.com/docs/storage?hl=es-419

GitHub. (2023). Aprende como comenzar a crear, enviar y mantener software con GitHub, *por Microsoft Corporation.*

https://docs.github.com/es

Mdn. (14 de diciembre 2023). *Recursos para Desarrolladores, por desarrolladores.*

https://developer.mozilla.org/en-US/

NodeJS. (2023). Official API reference as runtime built on the V8 JavaScript engine, *desarrollado y sustentado por OpenJS Foundation.*

https://nodejs.org/docs/latest/api/documentation.html

NPMJS: Dialogflow. (7 de febrero de 2020). Dialogflow API: Node.js Client, *por Google Apis y contribuyentes.*

https://www.npmjs.com/package/dialogflow

NPMJS: Venom-Bot. (14 de octubre de 2023). High-performance system developed with JavaScript to create a bot for WhatsApp, *por Orkestral.io.*

https://www.npmjs.com/package/venom-bot
https://orkestral.io/

8. ANEXOS

Anexo 1	
Archivo config.js y su exportación:	**module.exports**= { **GOOGLE_CLIENT_EMAIL**:'botutnwhatsapp@my-generation-fubv.iam.gserviceaccount.com**',** **GOOGLE_PRIVATE_KEY**:'-----BEGIN PRIVATE KEY-----\n\n-----END PRIVATE KEY-----\n', **GOOGLE_PROJECT_ID**:'**my-generation-fubv',** **DF_LANGUAGE_CODE**:'es'**,**

Anexo 2

Dialoflow.js:

```
const dialogflow = require("dialogflow");
const config = require("./config");
const credentials = {
  client_email: config.GOOGLE_CLIENT_EMAIL,
  private_key: config.GOOGLE_PRIVATE_KEY,
};
const sessionClient = new dialogflow.SessionsClient({
  projectId: config.GOOGLE_PROJECT_ID,
  credentials,
});
async function sendToDialogFlow(msg, session, params) {
  let textToDialogFlow = msg;
  try {
    const sessionPath = sessionClient.sessionPath(
      config.GOOGLE_PROJECT_ID,
      session
    );
    const request = {
      session: sessionPath,
      queryInput: {
        text: {
          text: textToDialogFlow,
          languageCode: config.DF_LANGUAGE_CODE,
        },
      },


  queryParams: {
        payload: {
          data: params,
        },
      },
    };
    const responses = await sessionClient.detectIntent(request);
    const result = responses[0].queryResult;
```

```
    let defaultResponses = [];
    if (result.action !== "input.unknown") {
      result.fulfillmentMessages.forEach((element) => {
          defaultResponses.push(element);
      });
    }
    if (defaultResponses.length === 0) {
      result.fulfillmentMessages.forEach((element) => {
        if (element.platform === "PLATFORM_UNSPECIFIED") {
          defaultResponses.push(element);
        }
      });
    }
    result.fulfillmentMessages = defaultResponses;
    //console.log(JSON.stringify(result,null," "));
    //console.log(`\x1b[36m`,'\nIntent
Emparejado:',result.intent.displayName, `\x1b[36m`);
    //console.log(`\x1b[36m`,'\nTexto Respuesta:',result.fulfillmentText,
`\x1b[36m`);
    /*console.log(`\x1b[36m`,'\nMensaje Entrante:',result.queryText,
`\x1b[36m`);
    return result;
    //console.log("se enviara el resultado: ", result);
  } catch (e) {
    console.log("error");
    console.log(e);
  };
}
module.exports = {
  sendToDialogFlow,
};
```

Se optó por esta base de código para trabajar la conexión de DialogFlow, ya que es la que tiene mejor sintaxis haciendo que el programa se entienda de mejor manera.
En pocas palabras este código esta compuesto por 2 importaciones de módulos, 1 constante que obtiene las credenciales, 1 objeto instanciado y 1 función principal. Y lo que hace este código es:

a) Importa credenciales desde “config.js” y la biblioteca “dialogflow” desde NODE, y luego las guarda en una constante para validarlas “const credentials = {}”
b) Crea una instancia “SessionsClient()” para interactuar con Dialogflow donde se interactuará con sesiones de DialogFlow usando nuestro ID del proyecto de Google Cloud y las credenciales anteriormente obtenidas.
c) Se crea la estructura principal del archivo, la función “sendToDialogFlow()” de manera asincrónica que toma 3 valores (mensaje del usuario, sesión de dialogflow, parámetros adicionales) y guarda el mensaje del usuario en “let textToDialogFlow = msg;”

d) Ejecuta un bloque try-catch donde:
Crea una ruta de sesión y solicitud con los parámetros establecidos.
Define un objeto “request” que contenía la información de la solicitud que se enviará a Dialogflow.
Envía la solicitud a Dialogflow para detectar la intención del mensaje y espera “await” la respuesta.
Extrae el resultado de la primera respuesta y obtiene la información de la consulta resultante “const result = responses[0].queryResult;”.
Prepara opcionalmente un conjunto de respuestas predeterminadas que se utilizarán si la acción de la intención no es “input.unknown” o si no hay mensajes definidos.
Se establecen 5 “console.log()” para poder leer los valores en la consola en caso de errores.
Devuelve el objeto “result” con la información procesada de la respuesta de DialogFlow y captura errores cerrando el bloque “try-catch”.
Por último se exporta la función “sendToDialogFlow()” para poder usada por otros archivos.

Opcionalmente se creó la siguiente función donde se podrá mandar un mensaje local hacía DialogFlow para comprobar que funciona, no forma parte del código final ya que es una prueba:

Anexo 2

```
async function procesarMensajeLocal() {
  const mensaje = "Hola";
  const session = "id_de_sesion";
  const parametros = {};
  try {
     const respuesta = await sendToDialogFlow(mensaje, session, parametros);
     console.log("Respuesta de Dialogflow:", respuesta);
  } catch (error) {
     console.error("Error al procesar el mensaje:", error);
  }
}
procesarMensajeLocal();
```

Este código enviará el mensaje “const mensaje” para realizar pruebas locales sin aún la conexión con WhatsApp. Solo se debe ejecutar el archivo “dialogflow.js”.

Anexo 3

Base de app.js

```
const uuid = require("uuid");
const venom = require('venom-bot');
const dialogflow = require("./dialogflow");
const sessionIds = new Map();
venom
  .create({
    session: 'nombre de sesion'
  })
  .then((client) => start(client))
  .catch((erro) => {
    console.log(erro);
  });

function start(client) {
  client.onMessage(async (message) => {
    setSessionAndUser(message.from);
    let session = sessionIds.get(message.from);
    let payload = await dialogflow.sendToDialogFlow(message.body, session);
    let responses = payload.fulfillmentMessages;
    for (const response of responses) {
      await sendMessageToWhatsapp(client, message, response);
    }
  });
}
function sendMessageToWhatsapp(client, message, response) {
  return new Promise((resolve, reject) => {
    client
      .sendText(message.from, response.text.text[0])
      .then((result) => {
        console.log('Result: ', result);
        resolve(result);
      })
      .catch((erro) => {
        console.error('Error when sending: ', erro);
        reject(erro);
      });
  });
}
async function setSessionAndUser(senderId) {
  try {
    if (!sessionIds.has(senderId)) {
      sessionIds.set(senderId, uuid.v1());
    }
```

```
  } catch (error) {
    throw error;
  }
}
```

Este código es la base de la primera versión de “app.js”, entrando en más detalles se puede ver que se importan los módulos “uuid” para generar IDs únicos y “venom-bot” para la conexión con WhatsApp, además de eso se importa el archivo local que antes se ha programado “const dialogflow = require("./dialogflow");” que permitió comunicarse correctamente usando la función “sendToDialogFlow()” dentro de la función “start(client)” y un array de mapeo “const sessionIds = new Map();” que fue la base para combinar cada número con una sesión única dentro de una ejecución.
Para hablar correctamente de cada método se los separará en ítems:

a) Objeto “venom.create({})”: Es la parte del código que invoca el método “.create()” proporcionado por el mismo “venom”, el cual crea la instancia de WhatsApp, adicional a eso nos permite pasar un parámetro “session” que será el nombre de la sesión actual donde se registrarán las credenciales del número de WhatsApp que se haya vinculado. Luego ejecuta “.then((client) => start(client))” que seguidamente que se cree la sesión de WhatsApp con éxito se ejecutará la función “start(client)” proporcionada por el método “.then()”, que a su vez “client” es el cliente de WhatsApp recién creado pasado como argumento. E caso de que existan errores en la ejecución se ejecutará el bloque ”.catch()” que mostrará el error en la consola. En resumen, inicializa el objeto venom que se encarga de crear la sesión de WhatsApp y luego llama a la función “start()” para que empiece a manejar los mensajes para esa sesión, si ocurre un error se mostrará por consola. La forma en la que se estructura esta función proviene del creador de “Venom-Bot”, si quisiéramos escribirla de una forma más legible se debería ver:

Anexo 3

Objeto Venom

```
venom.create({
  session: 'nombre de sesion'
}).then(function(client) {
  return start(client);
}).catch(function(erro) {
  console.log(erro);
});
```

b) Función “start(client)”: Esta función recibe el parámetro “client”, que representa el cliente de WhatsApp creado por Venom, seguidamente se establece un evento “onMessage” haciendo que cada vez que la sesión de WhatsApp reciba un mensaje haga lo siguiente

1. Se ejecutará un bloque asíncrono que obligará a esperar en cierto punto del código.
2. Asocia el numero del remitente a una sesión, se envía el mensaje a DialogFlow para procesarlo se obtiene la respuesta.
3. Llama a la función “sendMessageToWhatsApp()” para enviar el mensaje.

c) Función “sendMessageToWhatsApp()”: Se encarga de enviar un mensaje de texto a través de WhatsApp. Toma 3 argumentos “client” que es el cliente de WhatsApp utilizado para enviar el mensaje “message” que contiene la información sobre el mensaje recibido y “response” que es la respuesta que se enviará al remitente. Esta

función devuelve una promesa que se resolverá si el mensaje se envía con éxito y se rechazará si ocurro algún error durante el proceso de envío. Internamente la función utiliza el método "sendText()" del cliente de WhatsApp para enviar el mensaje al remitente. Si el mensaje se envía correctamente, se imprime un mensaje de éxito en la consola, si no, se captura el error y se lo imprime.

d) Función asíncrona "setSessionAndUser()": Es la función más sencilla del bloque, ya que simplemente toma el parámetro "senderId" que es el número de teléfono del remitente e intenta comprobar si existe en un mapa creado anteriormente en los módulos, si no existe lo agregará. Esta función nos ayuda a lograr una mejor lectura en etapas de desarrollo al recibir varios mensajes de usuarios distintos.
Al crearse el nuevo archivo "responseMappings.js" y sus correspondientes valores se debió adaptar la lógica de la aplicación para su correcta funcionalidad. Para no dificultad la lectura se dará un ejemplo recortado de cómo se verá el archivo recortado por la longitud del mismo.

Anexo 3

responseMappings.js

```
ports = [
l: "Hola usuario de DialogFlow", custom: "Hola usuario de WhatsApp"},
l: "Buenas tardes usuario de DialogFlow", custom: "Buenas tardes usuario de WhatsApp"},
```

Dentro del archivo se guardaron cada una de las posibilidades afectadas como un módulo que se exportará en "app.js" como "const responseMappings = require('./responseMappings');"
Para usar el mapeo se creó una función básica de "if/else":

Anexo 3

Función findCustomResponse()

```
function findCustomResponse(originalText) {
  const mapping = responseMappings.find(mapping =>
originalText.startsWith(mapping.original));
  return mapping ? mapping.custom : originalText;
}
```

Busca un texto original dado "originalText" dentro del conjunto de mapeos de respuestas "responseMappings" y luego utiliza el método "find()" en el array anteriormente nombrado para buscar el primer elemento que cumpla con la condición proporcionada en la función de retorno, en este caso se uso el método "startsWith()" que verifica si un campo empieza con determinados caracteres, por último si se encuentra un mapeo que coincida la función devolverá el texto personalizado, sino, devolverá el texto original sin cambios.
Para llamar a esta función solo se debió agregar un pequeño bloque dentro del "for" de la función "start(client)":

Anexo 3

start(client)

```
for (const response of responses) {
      if (response.text && response.text.text.length > 0) {
        response.text.text[0] = findCustomResponse(response.text.text[0]);
      }
       await sendMessageToWhatsapp(client, message, response);
    }
```

Se usó la condicional "if" para verificar cada objeto "response" en el array "responses" si cumplen con la condición de tener una propiedad "text" y que su longitud sea mayor que cero, asegurando que haya texto para procesar. Por último sobreescribe "response.text.text[0]" con el texto personalizado de "findCustomResponse()" y lo envía a WhatsApp.

1) Base de Datos: Se creó la cadena de conexión con MySql, se modificó la función "start(client)" de "app.js" agregando una interación para capturar ciertos errores en la ejecución (en desarrollo) y se crearon variables que capturen datos relevantes del mensaje:

Anexo 4

Cadena de Conexión

```
const dbConfig = {
  host: 'localhost',
  user: 'root',
  password: 'intbotutn2023',
  database: 'nros_telefono',
};
```

Función "start(client)":

Anexo 4

start(client)

```
function start(client) {
  client.onMessage(async (message) => {
    setSessionAndUser(message.from);
    let session = sessionIds.get(message.from);
    for (let attempt = 0; attempt < 5; attempt++){
      if(!message.body || message.body.trim() === ''){
        if(attempt === 4){
            console.log('Mensaje vacio despues de 2 intentos no se pudo procesar');
            console.log('Puedes volver a repetir el mensaje');
            client.sendText(message.from, 'Puedes volver a enviar el mensaje?');
            return;
          }else{
            console.log('Mensaje vacio Reintentando');
            continue;
          }
      }
    }
    let payload=await dialogflow.sendToDialogFlow(message.body, session);
```

```
    let responses=payload.fulfillmentMessages;
    for (const response of responses) {
      if (response.text && response.text.text.length > 0) {
        response.text.text[0] = findCustomResponse(response.text.text[0]);
        let num = message.from.replace(/@c\.us$/, "");
        let msnin = message.body;
        let msnout = response.text.text[0];
        let intemp = payload.intent.displayName;
        const currentDateTime = moment();
        const currentDay = currentDateTime.format('YYYY-MM-DD');
        const currentHour = currentDateTime.format('HH:mm:ss');
        console.log('Numero de telefono:', num);
        console.log('Mensaje recibido: ', msnin);
        console.log('Respuesta enviada: ', msnout);
        console.log('Intent Emparejado: ', intemp);
        console.log('Dia de la consulta', currentDay);
        console.log('Hora de la consulta', currentHour);
        await insertarDatosEnTablas(num, msnin, msnout, intemp);
      }
       await sendMessageToWhatsapp(client, message, response);
    }
  });
}
```

Luego de capturados los datos: número de teléfono, mensaje entrante, mensaje de salida e intención emparejada se los envía como parámetros a la función “insertarDatosEnTablas()”y espera a que se ejecute correctamente antes de seguir con el proceso.

Anexo 4

insertarDatosEnTablas()":

```
async function insertarDatosEnTablas(num, msnin, msnout, intemp) {
  const connection = await mysql.createConnection(dbConfig);
  try {
    const [telefonosRows] = await connection.execute(
      'INSERT IGNORE INTO telefonos (numero_telefono, cantidad_consultas) VALUES
(?, 0)',
      [num]
    );

    const [idTelefonoRow] = await connection.execute(
      'SELECT id_telefono FROM telefonos WHERE numero_telefono = ?',
      [num]
    );
    const idTelefono = idTelefonoRow[0].id_telefono;

    const idConsulta = uuid.v4();

    const currentDateTime = moment();
    const fechaActual = currentDateTime.format('YYYY-MM-DD');
    const horaActual = currentDateTime.format('HH:mm:ss');

    await connection.execute(
      'INSERT INTO consultas (id_consulta, id_telefono, numero_telefono,
mensaje_entrante, respuesta_saliente, intent_emparejado, fecha, hora) VALUES (?, ?, ?, ?,
?, ?, ?, ?)',
      [idConsulta, idTelefono, num, msnin, msnout, intemp, fechaActual, horaActual]
    );

    await connection.execute(
      'UPDATE telefonos SET cantidad_consultas = cantidad_consultas + 1 WHERE
id_telefono = ?',
      [idTelefono]
    );

    console.log('Datos insertados con éxito en las tablas.');
  } catch (error) {
    console.error('Error al insertar datos en las tablas:', error);
  } finally {
    connection.end();
  }
}
```

Se establece una conexión con la base de datos, se intenta insertar el número de teléfono en la tabla "teléfonos" si no existe, con una cantidad inicial de consultas igual a cero. Obtiene el ID del número de teléfono recién insertado o recuperado. Genera un UUID

para la consulta. Obtiene la fecha y hora actuales al momento del mensaje. Inserta todos los datos de la consulta en la tabla “consultas”, incluyendo el ID de la consulta, ID del teléfono, número de teléfono, mensaje entrante, respuesta saliente, intención emparejada, fecha y hora. Por último, incrementa la cantidad de consultas en la tabla “teléfonos” para el numero correspondiente, llevada a cabo la ejecución muestra un mensaje de éxito o de error y cierra la conexión con la base de datos.

2) Codigo del bloque principal de renderización “main.js”, se divide en 5 partes:
   a) Importación de Módulos:

Anexo 5
Importación de Módulos

```
const { app, BrowserWindow, ipcMain } = require('electron');
const path = require('path');
const fs = require('fs');
const { spawn } =require('child_process');
```

Se importó 3 objetos para la manipulación de eventos con “Electron”, y 3 módulos adicionales “path”, “fs” y “spawn” para manipulación de archivos, manejo de rutas y creación de procesos secundarios respectivamente.

b) Creación de Ventana Principal:

Anexo 5
Creación de Ventana Principal

```
let mainWindow;
function createWindow() {
  mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true,
      contextIsolation: false,
    },
  });
  mainWindow.loadFile('index.html');
  mainWindow.on('closed', function () {
    mainWindow = null;
  });
}
```

Se definieron eventos relacionados con los estados de la aplicación, al estar lista “app.WhenReady()”, cuando todas las ventanas se cierran “window-all-closed”, y cuanto la aplicación se activa “activate”, en este caso se uso un condicional que llama a “createWindow()” cuando la ejecución esta lista y en caso de que todas las ventanas se cierran, se cierra la aplicación.

c) Eventos de Inicio y Detención y comunicación entre procesos “ipcMain”:

Anexo 5

ipcMain

```
ipcMain.on('start-app', () => {
  console.log('Evento Start-App recibido. Iniciando app...');
  const appPath = path.join(__dirname, 'app.js');
  //const appProcess = require('child_process').spawn('node', [appPath]);
  const appProcess = require('child_process').fork(appPath);

  appProcess.on('message', (message) => {
    if (message.type === 'stdout') {
      console.log(`[app.js] stdout: ${message.data}`);
    } else if (message.type === 'stderr') {
      console.error(`[app.js] stderr: ${message.data}`);
      mainWindow.webContents.send('app-error', message.data.toString());
    }
  });
  appProcess.on('exit', (code) => {
    console.log(`[app.js] child process exited with code ${code}`);
    if (code === 0) {
      mainWindow.webContents.send('app-started');
    }
  });
  /*appProcess.stdout.on('data', (data) => { //salida estandar
    console.log(`[app.js] stdout: ${data}`);
  });

  appProcess.stderr.on('data', (data) => { //salida de error
    console.error(`[app.js] stderr: ${data}`);
    mainWindow.webContents.send('app-error', data.toString());
  });

  appProcess.on('close', (code) => { //manejo de cierre del proceso hijo
    console.log(`[app.js] child process exited with code ${code}`);
    if (code === 0) {
      // Emitir evento 'app-started' al proceso de representación
      mainWindow.webContents.send('app-started');

    }
  });*///Comentado para pruebas de compatibilidad entre entorno de desarrollo y .exe
});
ipcMain.on('stop-app', () => {
  if (mainWindow) {
    mainWindow.close(); //cerrar ventana principal de electron
    console.log('Aplicación cerrada.');
  }
  const imagePath = path.join(__dirname, 'out.png'); //obtiene ruta de out.png
```

```
  if(fs.existsSync(imagePath)) {
    try {
    fs.unlinkSync(imagePath); //elimina out.png
    console.log('Archivo eliminado con exito.');
    }catch (err) {
      console.error('Error al eliminar el archivo', err);
    }
  }else{
    console.log('El archivo no existe. No re realizo ninguna accion');
  }
});
```

Se definió el evento de comunicación “ipcMain.on(‘start-app’) que captura el archivo “app.js” para ejecutarlo con 2 variaciones “.spawn()” que ejecutará nuestro archivo con comandos del sistema operativo permitiéndonos generar devoluciones “stdin, stdout y stderr” directamente en la consola de la aplicación renderizada y “.fork()” que se utiliza específicamente para ejecutar scripts de Node.Js en procesos secundarios, siendo mucho más optimo para la ejecución en diferentes entornos y directorios pero no permitiendo la devoluciones de errores en la consola de la misma manera que “.spawn()” (para este caso se debió desarrollar una lógica que genere los mensajes de errores y ejecución en un bloc de notas “log.txt” presente en el repositorio remoto de GitHub. Si se usa “.spawn()” se debe quitar los comentarios de “appProcess” para generar devoluciones).
También se creo el evento de comunicación “ipcMain.on(‘stop-app’) que cierra la ventana principal y verifica si existe la imagen “out.png” (QR) para eliminarla.

d) Creación de Ventanas Secundarias:

Anexo 5

Creación de Ventanas Secundarias

```
let remitentesWindow;
  function createRemitentesWindow() {
    remitentesWindow = new BrowserWindow({
      width: 800,
      height: 600,
      webPreferences: {
        nodeIntegration: true,
        contextIsolation: false,
      },
    });

    remitentesWindow.loadFile('remitentes.html');
    remitentesWindow.on('closed', function () {
      remitentesWindow = null;
    });
  }
  ipcMain.on('open-remitentes-window', () => {
    if (!remitentesWindow) {
      createRemitentesWindow();
    }
```

```
  });
  ipcMain.on('close-remitentes-window', () => {
    if (remitentesWindow) {
      remitentesWindow.close();
    }
  });
  let filtroWindow;
function createFiltroWindow(datosFiltrados) {
  filtroWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true,
      contextIsolation: false,
    },
  });
  filtroWindow.loadFile('filtro.html');
  filtroWindow.webContents.on('did-finish-load', () => {
    filtroWindow.webContents.send('datosFiltrados', datosFiltrados);
  });
  filtroWindow.on('closed', function () {
    filtroWindow = null;
  });
}
ipcMain.on('abrir-ventana-filtro', (event, datosFiltrados) => {
  if (!filtroWindow) {
    createFiltroWindow(datosFiltrados);
  }
});
```

Por último se agregó la lógica para crear las ventanas “remitentes.html” y “filtro.html” y manejar sus cierres.

3) Script de “index.html”:

Anexo 6

Script de “index.html”

```
<script>
  function reloadImage() {
  const timestamp = Date.now();
  let imagePath = "out.png";
  imagePath += `?timestamp=${timestamp}`;
  document.getElementById('qrCode').src = imagePath;
}
  setInterval(reloadImage, 15000);
  document.getElementById('openRemitentes').addEventListener('click', () => {
    ipcRenderer.send('open-remitentes-window');
  });
</script>
```

Se creó la función que recargará la imagen QR cada 15 segundos, usando el método "timestamp" para evitar el almacenamiento de caché. Luego se agregó un evento que llamará al proceso de renderización para mostrar "remitentes.html".

4) Script de "remitentes.html":

Anexo 7

Script de "remitentes.html"

```
<script>
  const mysql = require('mysql2/promise');
  const { remote } =require('electron');
  document.getElementById('volver').addEventListener('click', () => {
    ipcRenderer.send('close-remitentes-window');
  });
  document.addEventListener('DOMContentLoaded', async () => {
    await mostrarTodasLasConsultas();
  });

async function filtrarDatos() {
const numero = document.getElementById('numeroInput').value;
const fecha = document.getElementById('fechaInput').value;
const hora = document.getElementById('horaInput').value;
const intent = document.getElementById('intentInput').value;
const dbConfig = {
  host: 'localhost',
  user: 'root',
  password: 'intbotutn2023',
  database: 'nros_telefono',
};
let connection;
const filtros = {
  numero_telefono: numero,
  fecha: fecha,
  hora: hora,
  intent_emparejado: intent,
};
const condiciones = Object.entries(filtros)
  .filter(([_, valor]) => valor.trim() !== '')
  .map(([campo, valor]) => `${campo} = '${valor}'`)
  .join(' AND ');

if (!condiciones) {
  console.log('Debes proporcionar al menos un criterio de filtro.');
  return;
}
try {
  const connection = await mysql.createConnection(dbConfig);
```

```
  const [rows] = await connection.execute(`SELECT * FROM consultas WHERE
${condiciones}`);
  console.log('Datos filtrados con éxito.');

  ipcRenderer.send('abrir-ventana-filtro', rows);
} catch (error) {
  console.error('Error al filtrar datos:', error);
}if(connection){
  connection.end();
}
}

async function mostrarTodasLasConsultas() {
const dbConfig = {
  host: 'localhost',
  user: 'root',
  password: 'intbotutn2023',
  database: 'nros_telefono',
};
let connection;
try {
  connection = await mysql.createConnection(dbConfig);
  const [rows] = await connection.execute('SELECT * FROM consultas ORDER BY fecha
DESC, hora DESC');
  const tbody = document.getElementById('tbodyDatos');
  tbody.innerHTML = '';

  rows.forEach((row) => {
    const tr = document.createElement('tr');
    tr.innerHTML = `
      <td>${row.id_consulta}</td>
      <td>${row.id_telefono}</td>
      <td>${row.mensaje_entrante}</td>
      <td>${row.respuesta_saliente}</td>
      <td>${row.intent_emparejado}</td>
      <td>${row.fecha}</td>
      <td>${row.hora}</td>
    `;
    tbody.appendChild(tr);
  });

  console.log('Consultas mostradas con éxito.');
} catch (error) {
  console.error('Error al mostrar consultas:', error);
}finally{
  if(connection){
```

```
    console.log('Conexion cerrada dsps de consulta');
    connection.end();
  }
}
}

setInterval(mostrarTodasLasConsultas, 10000);

  const { ipcRenderer } = require("electron");
  const { extractFormattedSenderId } = require('./main.js');
</script>
<script src="main.js"></script>
```

Este script define las funciones “filtrarDatos()” y “mostrarTodasLasConsultas()” que recupera los valores de los inputs HTML establecen la conexión con la base de datos y filtra según los criterios proporcionados enviándolos a la ventana “filtro.html”, y ejecuta una consulta para obtener todas las consultas de la base de datos y las muestra en la tabla del mismo archivo llamando a la función en intervalos de 10 segundos para mantener actualizada la información.

1) Script de “filtro.html”:

Anexo 8

Script de “filtro.html

```
<script>
    const { ipcRenderer } = require('electron');

    ipcRenderer.on('datosFiltrados', (event, datosFiltrados) => {
      const tbody = document.getElementById('tbodyDatosFiltrados');
      tbody.innerHTML = '';

      datosFiltrados.forEach((row) => {
        const tr = document.createElement('tr');
        tr.innerHTML = `
          <td>${row.id_consulta}</td>
          <td>${row.id_telefono}</td>
          <td>${row.mensaje_entrante}</td>
          <td>${row.respuesta_saliente}</td>
          <td>${row.intent_emparejado}</td>
          <td>${row.fecha}</td>
          <td>${row.hora}</td>
        `;
        tbody.appendChild(tr);
      });
    });
  </script>
```

Este Script utiliza el método “ipcRenderer.on” para escuchar cuando se filtraron los datos en el “remitentes.html” y mostrar la tabla con los filtros realizados.

2) QR de validación para la ejecución de “app.js”: Para lograr implementar más controles en la función de Venom, se ha usado la documentación para ampliarla e integrarla con otros módulos:

Anexo 9

```
/*admin.initializeApp({
    credential: admin.credential.cert(serviceAccount),
    storageBucket: 'gs://qrimgbotutn.appspot.com'
  });
  const bucket = admin.storage().bucket();*/
////////////////////////////////////////////////////////////// Fin Bloque de Comunicación con Firebasestorage
////////////////////////////////////////////////////////////// Inicio Bloque principal de libreria Venom-bot
const admin = require('firebase-admin');
const serviceAccount = require('./qrimgfbs.json');
  venom
  .create(
    'Agente Virtual Whatsapp UTN',
    (base64Qr, asciiQR, attempts, urlCode) => {
      console.log(asciiQR);
      var matches = base64Qr.match(/^data:([A-Za-z-+/]+);base64,(.+)$/),
        response = {};

      if (matches.length !== 3) {
        return new Error('Invalid input string');
      }
      response.type = matches[1];
      response.data = new Buffer.from(matches[2], 'base64');
      var imageBuffer = response;
      require('fs').writeFile(
        'out.png',
        imageBuffer['data'],
        'binary',
        function (err) {
          if (err != null) {
            console.log(err);
          }
          ////////////////////////////////////////////////////////////// Inicio subir out.png a firebasestore y obtener url
          /*bucket.upload('out.png', {
            destination: 'path/to/upload/out.png' // La ruta en Firebase Storage donde deseas almacenar el archivo
          }).then(() => {
            console.log('Archivo subido exitosamente a Firebase Storage.');
            bucket.file('path/to/upload/out.png').getSignedUrl({
              action: 'read',
```

```
            expires: '01-01-2025' // Fecha de expiración de la URL firmada
          }).then(signedUrl => {
            console.log('URL de la imagen:', signedUrl);
          }).catch(error => {
            console.error('Error al obtener URL firmada:', error);
          });
          }).catch((error) => {
            console.error('Error al subir el archivo a Firebase Storage:', error);
          });*/
          ////////////////////////////////////////////////////////////// Fin subir out.png a firebasestore y
obtener url
        }

      );

    },
    undefined,
    { logQR: false }

  )
  .then((client) => {
    start(client);

  })
  .catch((erro) => {
    console.log(erro);
  });
```

Este bloque se divide en 4 partes donde:

a) Se importan los módulos “firebase-admin” para usar funciones y métodos de gestión de archivos en la nube (Google Firebase Storage) y “qrimgfbs.json” que es el nombre que se le dio a las credenciales de la nube de la misma manera que se hizo en Anexo 1.
b) Se amplía el código de “venom.create()”, donde sus funciones son las mismas pero ahora se muestran explícitamente en el código, logrando que se puedan modificar sus parámetros a detalle, en este caso obtener la variable que genera el QR y generar 3 intentos de inicio de sesión.
c) Se usó el módulo FS (File System) para generar la imagen QR como archivo en el directorio.
d) Se generó lógica para establecer una conexión con Firebase Storage, subir la imagen QR y devolver el enlace de la misma en la consola.