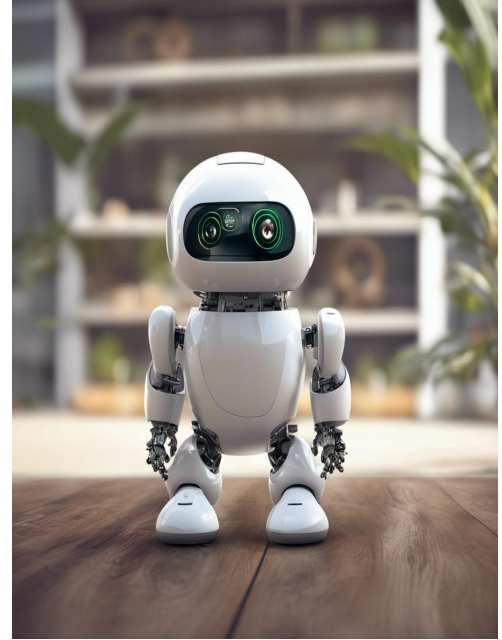


Universidad politécnica de Santa Rosa Jauregui

# Denisse un agente conversacional inteligente para mejorar la atención y servicios en la universidad politécnica de Santa Rosa Jauregui



20 de julio del 2023

Autor:

Kevin Arturo Mondragon Tapia, Luis Leonardo Mendoza Resendiz  
y Victor Alejandro Monroy Rodriguez

Tutor:

[Juan Manuel Peña Aguilar](#)

---

## ÍNDICE:

<b>ÍNDICE:</b>	<b>2</b>
<b>Resumen</b>	<b>5</b>
<b>Abstract</b>	<b>6</b>
<b>Marco Teórico</b>	<b>7</b>
¿Qué es un ChatBot?	7
Seguridad y Privacidad	8
<b>Objetivos</b>	<b>9</b>
objetivo general	9
objetivos específicos	9
<b>Metodología</b>	<b>9</b>
Metodología para el desarrollo	9
Planificación	10
Desarrollo Incremental	11
Público objetivo	12
<b>Descripción del rol de los actores del sistema</b>	<b>12</b>
Persona general (usuario)	12
Personal Administrativo (ChatBot)	13
<b>Requisitos no funcionales identificados</b>	<b>13</b>
<b>Diagrama de casos de uso UML</b>	<b>15</b>
<b>Diagrama de clases UML</b>	<b>16</b>
<b>Modelo de Datos</b>	<b>17</b>
<b>Modelo de Componentes</b>	<b>18</b>
• Plataforma de desarrollo de chatbots:	18
• Procesamiento del lenguaje natural (PLN):	18
• Base de conocimientos:	18
• Integración con sistemas existentes:	18
• Interfaz de usuario:	18
• Integración con redes sociales y sitio web:	20
• Seguridad y privacidad:	20
• Capacidades de aprendizaje automático:	20
• Métricas y análisis:	20
<b>Descripción de la Arquitectura</b>	<b>20</b>
¿Qué es la arquitectura orientada a servicios?	20
• Beneficios:	22
• Componentes:	23
• ¿Qué son los microservicios?	25
• ¿Qué es una API?	26
• Diferencias entre RPC y REST	27
<b>Modelo de Despliegue</b>	<b>28</b>

---

Git y Github	29
Git:	29
<b>Manual para programadores:</b>	<b>31</b>
¿Qué utilizamos para este desarrollo?	31
Requerimientos	31
¿Cómo saber que tengo el Node?	31
¿Cómo instalar Node?	31
¿Cómo saber que tengo Git?	32
¿Cómo instalar Git?	32
Instalación	32
Comenzamos	33
Pruébalo	33
Explicando código	35
Esenciales	36
Conceptos	36
Flow (Flujos)	36
Provider (Proveedor)	37
DataBase (Base de datos)	38
Flow	39
blackList	40
addKeyword()	40
addAnswer()	41
ctx	43
fallBack()	43
flowDynamic()	44
endFlow()	45
QR Portal Web	47
Proveedores	48
Twilio: Configuración	49
Meta: Configuración	49
DataBase (Base de datos)	49
<b>Migración</b>	<b>50</b>
Versión (legacy)	50
Versión 2 (0.2.X)	53
Flujos hijos	54
<b>Despliegue</b>	<b>58</b>
<b>Cómo realizar la instalación</b>	<b>58</b>
Entorno Local	58
<b>Entorno Docker</b>	<b>59</b>
Construir imagen	59
Iniciar contenedor	59
<b>Entorno Cloud</b>	<b>59</b>
Conectarse via SSH	61

---

Recuerda instalar Node 16 o superior	62
Implementar el bot	62
Firewall	63
Escanear QR	64
Ejecutar en Producción	65
Entrar a github	66
Dar click sobre el botón Code y copia el HTTPS	67
Instalarlo en una máquina de ubuntu	67
Ejecución en segundo plano en una máquina virtual Ubuntu	68
<b>Configuración de una VPN en Ubuntu 20.04 (Opcional)</b>	<b>70</b>

---

## Resumen

En el presente documento se describe la construcción de Denisse, un chatbot informativo de preguntas frecuentes sobre la clínica de fisioterapia de la universidad politécnica de Santa Rosa Jauregui y responder preguntas frecuentes (Frequently Asked Questions) sobre la misma universidad, FAQ por sus siglas en inglés; con diferentes herramientas y un framework de Javascript utilizando node.js como entorno de ejecución, que permiten el Procesamiento de palabras claves para el entendimiento de elección del usuario y emisión de respuesta con la información almacenada en una base de conocimiento. Utilizando una arquitectura Orientada a servicios, la cual nos permite utilizar la plataforma de whatsapp para que el usuario pueda tener una curva de aprendizaje sumamente corta, utilizando Sql para la recopilación de datos y opinión de los usuarios para futuras consultas y análisis de datos(En el caso de que la información sobre alguna pregunta ingresada no estuviera contemplada, se guardará en una base de datos para el docente encargado de administrar la base de conocimiento, que indica la necesidad de alimentación sobre la información no encontrada con el fin de mantenerla actualizada).

---

## **Abstract**

In the present document, the construction of Denisse is described, an informative chatbot for frequently asked questions about the physiotherapy clinic of Santa Rosa Jauregui Polytechnic University. It responds to Frequently Asked Questions (FAQ) about the university, using various tools and a JavaScript framework utilizing node.js as the execution environment. These tools enable keyword processing for user understanding and provide responses with information stored in a knowledge base.

An Oriented-to-Services architecture is used, allowing the utilization of the WhatsApp platform to ensure users have a very short learning curve. SQL is employed for data collection and user feedback for future inquiries and data analysis. In case the information for a particular question is not available, it will be stored in a database for the responsible instructor to manage the knowledge base. This indicates the need for updating the information not found to keep it up-to-date.

---

## Marco Teórico

### Introducción

En la era digital actual, la tecnología de inteligencia artificial ha revolucionado la forma en que interactuamos con las computadoras y los servicios en línea. Uno de los avances más notables es la creación de los chatbots, sistemas de conversación automatizados que utilizan el procesamiento del lenguaje natural (NLP) y el aprendizaje automático para interactuar con los usuarios de manera eficiente y efectiva.

Su capacidad para proporcionar respuestas rápidas, información precisa y servicios personalizados los ha convertido en una herramienta valiosa para empresas y organizaciones que buscan mejorar la experiencia del usuario y optimizar sus operaciones.

El objetivo de este proyecto es explorar el mundo de los chatbots, analizando su marco teórico, su arquitectura, los enfoques de desarrollo utilizados, así como las aplicaciones prácticas y beneficios que pueden aportar a diferentes industrias, en este caso la universidad. También se abordarán los desafíos y consideraciones éticas asociadas con el diseño y la implementación de estos sistemas, destacando la importancia de garantizar la privacidad y la seguridad de los datos del usuario.

### ¿Qué es un ChatBot?

Un chatbot es un programa de inteligencia artificial diseñado para interactuar con usuarios a través de conversaciones en lenguaje natural, ya sea escrito o hablado. Su objetivo principal es simular una conversación humana y proporcionar respuestas coherentes y relevantes a las preguntas o consultas de los usuarios.

Los chatbots utilizan tecnologías como el procesamiento del lenguaje natural (NLP) y el aprendizaje automático (Machine Learning) para entender las consultas de los usuarios, analizar el contexto de la conversación y generar respuestas apropiadas.

---

Algunos chatbots están basados en reglas, lo que significa que siguen un conjunto predefinido de instrucciones para responder a entradas específicas, mientras que otros utilizan modelos de lenguaje avanzados y entrenados con grandes cantidades de datos para generar respuestas más flexibles y contextual **Interacción y Experiencia del Usuario**

Diseñar una interacción conversacional amigable y natural. El chatbot debe ser capaz de entender preguntas en lenguaje cotidiano y proporcionar respuestas claras y precisas. Además, considerar la posibilidad de implementar elementos visuales, como botones o tarjetas, para facilitar la navegación y mejorar la experiencia del usuario.

### **Seguridad y Privacidad**

Garantizar la seguridad y privacidad de los datos del usuario. Asegurarse de cumplir con las regulaciones y políticas de protección de datos y ofrecer una opción clara para que los usuarios comprendan cómo se utilizarán sus datos.

En conclusión, un chatbot universitario bien diseñado puede ser una herramienta valiosa para mejorar la experiencia estudiantil, proporcionar información rápida y precisa, y automatizar ciertos procesos administrativos. Al seguir un marco teórico sólido, la universidad puede desarrollar un chatbot efectivo que beneficie a toda su comunidad estudiantil.



---

## Objetivos

### objetivo general

Crear un prototipo chatbot como agente conversacional para resolución de dudas frecuentes y procesos de la universidad politécnica de Santa Rosa Jauregui.

### objetivos específicos

- Investigar sobre la naturaleza de los chatbots y su utilización como asistentes virtuales para la resolución de preguntas frecuentes.
- Implementar árboles conversacionales que puedan orientar al usuario sobre los procesos que no quedan claros a través de la plataforma de la universidad
- Creación de una arquitectura orientada a servicios, la cual pueda utilizar la interfaz de whatsapp para mayor comodidad de los usuarios.
- Conexión a una base de datos sql para retroalimentación de los usuarios
- Realizar las pruebas necesarias para conocer el tiempo de respuesta y la carga necesaria.
- Posibilidad de agendado de citas desde el chatbot a través de archivos .ics , para gestión de la clínica
- Despliegue en un servidor

## Metodología

### Metodología para el desarrollo

Para el desarrollo del presente proyecto se utilizó el marco de trabajo Scrum ya que, a lo largo del proyecto, se realizaron entregas de productos incrementales tras adaptar los cambios con base en la retroalimentación obtenida por parte del tutor del product owner. Para ayudar en

el flujo de tareas se utilizó la herramienta en línea Miro, la cual es gratuita y permite registrar el movimiento de las tareas a través del tablero Scrum, como se puede ver en la figura 1.

Tablero Scrum en la herramienta Miro



Figura 1: Tablero Scrum de la herramienta Miro con las tareas realizadas.

Elaborada por: Kevin Mondragon

## Planificación

En la planificación se obtuvieron casi todos los requerimientos para el desarrollo del chatbot, sin embargo las estructuras cambiaron. Se establecieron las tareas a realizar y los Sprints a partir de las funcionalidades sobre las que Trabajó. Se establecieron los roles de Scrum de los involucrados en el proyecto, definiendo de esta manera:

- 
- Product owner: Se definieron como dueños del producto, [Juan Manuel Peña Aguilar](#) desempeña ese papel junto a Kevin Mondragon
  - Scrum master: Se designó como Scrum master, al tutor del presente desarrollo Kevin Mondragon, puesto que él se encargó de la gestión de reuniones y presentación de avances.
  - Equipo Scrum: El equipo Scrum, encargado de realizar los cambios y desarrollar los incrementos del proyecto, está conformado por Luis Leonardo Mendoza Resendiz, Victor Alejandro Monroy Rodriguez y Kevin Arturo Mondragon Tapia.
  - Usuarios: El rol de usuarios fue asignado tomando en cuenta que el chatbot será utilizado por tres tipos de usuarios: usuario administrador, usuario alumno y usuario visitante. Una vez definidos los roles, se procedió con el establecimiento de los backlogs.

## Desarrollo Incremental

Scrum se emplea como un marco de trabajo para el desarrollo de productos, con el propósito principal de generar productos terminados y utilizables que aporten mejoras significativas al producto final. Estos progresos se presentan como entregables al concluir cada Sprint.

Para el desarrollo del presente proyecto, se adoptó la técnica de prototipado, lo que facilitó una mejor organización de las etapas del proceso de desarrollo. Se elaboró un conjunto de especificaciones de diseño enfocadas en la arquitectura del chatbot. El prototipo se dividió en incrementos, los cuales se desarrollaron dentro de los Sprints, con el objetivo de lograr mejoras significativas en el producto.

Al finalizar cada Sprint, se entregó el avance incremental desarrollado para cada componente del prototipo, y estos fueron evaluados por el Scrum Master según su disposición.

1. Toma de requerimientos con usuarios de la clínica, alumnado y aspirantes
2. Análisis del contenido de la materia para la construcción del marco de conocimiento.
3. Análisis técnico del tipo de chatbot.
4. Motor del chatbot implementado en Node.js con las herramientas analizadas.
5. Creación de árboles conversacionales en el frameWork seleccionado
6. implementación de árboles conversacionales en el frameWork seleccionado
7. Interfaz de WhatsApp conectada al motor del chatbot por medio de una API

- 
8. Diferenciación de ambientes para los tres tipos de usuarios establecidos.
  9. Información de la materia almacenada en una base de datos relacional.
  10. Proceso de notificación vía ICS, indicando que la información en alguna plataforma

### **Público objetivo**

El ChatBot será implementado en el Web Site institucional para que estudiantes, personas fuera de la institución (personas que quieran agendar una cita, o quienes desean estudiar alguna carrera).

## **Descripción del rol de los actores del sistema**

### **Persona general (usuario)**

- **Consultar información académica:** En este caso cualquier persona podrá consultar cualquier información académica que desee consultar, como: Citas de fisioterapia, ubicación, consultar horarios de clases, servicios escolares, ayuda y preguntas frecuentes.
- **Agendar citas (Área de Fisioterapia):** El usuario (persona general) podrá agendar citas para el área de fisioterapia y así mismo consultarlas.
- **Solicitar información de contacto:** El usuario puede solicitar el contacto para las citas de fisioterapia.
- **Ubicación:** El usuario podrá consultar la ubicación de la universidad con especificaciones si es necesario.
- **Consultar horarios de clases:** El usuario podrá consultar los horarios de clases si así es necesario de una manera rápida y eficaz.
- **Ayuda y preguntas frecuentes:** El usuario podrá ver las preguntas más frecuentes hechas por otros usuarios, también podrá ver la palabra “ayuda”

---

donde se verá reflejado como se utiliza el chatbot y así hacer sus preguntas debidamente.

- **Servicios Escolares:** El usuario podrá consultar número y horarios sobre servicios escolares.

#### **Personal Administrativo (ChatBot)**

- **Agendar citas (Área de Fisioterapia):** El ChatBot podrá agendar citas conforme se vayan capturando los datos de la persona.
- **Solicitar información de contacto:** El ChatBot podrá solicitar la información del contacto como: número de teléfono y correo electrónico.
- **Ayuda y preguntas frecuentes:** El ChatBot podrá lanzar a la vista preguntas frecuentes hechas por otros usuarios, este también mandará la palabra “ayuda” por si el usuario necesita alguna orientación con el sistema.

### **Requisitos no funcionales identificados**

Los requisitos no funcionales son aquellos que definen las características y atributos del sistema que no están directamente relacionados con sus funcionalidades, sino con aspectos más amplios de su rendimiento, calidad, seguridad y otros aspectos no específicos de las tareas que realiza.

- **Rendimiento:** El chatbot deberá responder a las consultas de los usuarios en un tiempo de respuesta aceptable, generalmente inferior a 1 segundo, para proporcionar una experiencia fluida y satisfactoria.
- **Disponibilidad:** El chatbot debe estar disponible y accesible para los usuarios en todo momento, con un objetivo de tiempo de actividad cercano al 100% para evitar interrupciones prolongadas.
- **Seguridad:** Se deben implementar medidas de seguridad para proteger los datos y la privacidad de los usuarios, evitando accesos no autorizados y asegurando la

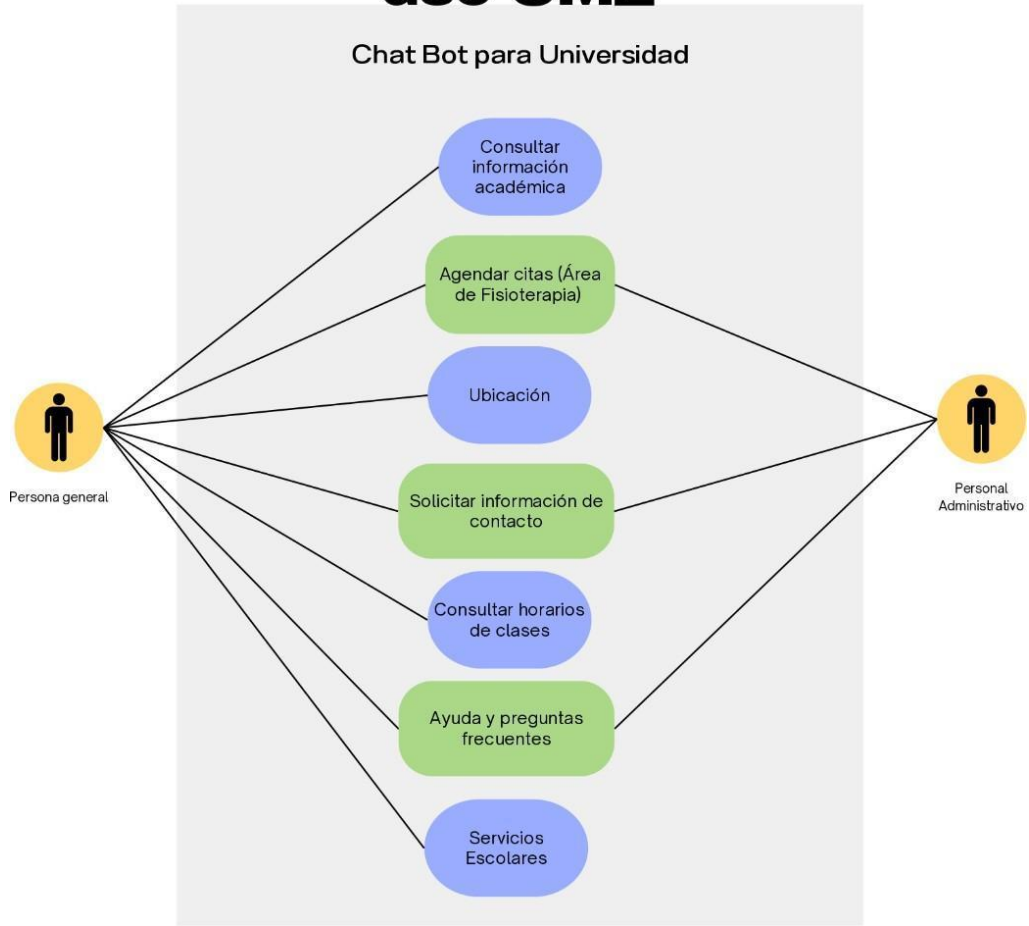
---

integridad de la información.

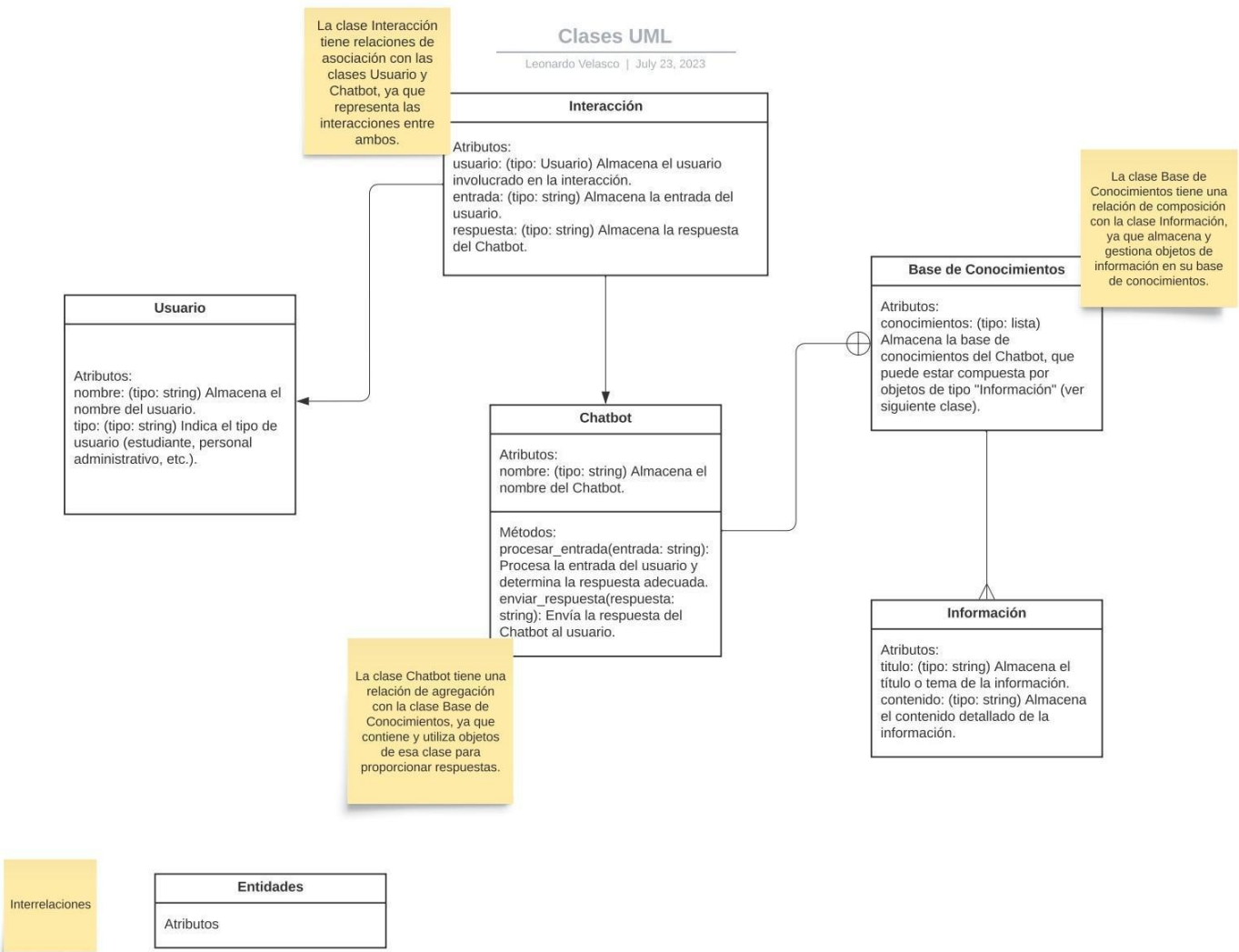
- **Usabilidad:** El chatbot debe ser fácil de usar y comprender para los usuarios, con una interfaz intuitiva que permita una interacción natural y sin complicaciones.
- **Mantenibilidad:** El código del chatbot debe estar bien estructurado y documentado para facilitar su mantenimiento y futuras actualizaciones.
- **Integración:** El chatbot debe ser capaz de integrarse con otros sistemas o servicios relevantes, como bases de datos, servicios web o plataformas de mensajería.

Diagrama de casos de uso UML

Diagrama de casos de uso UML

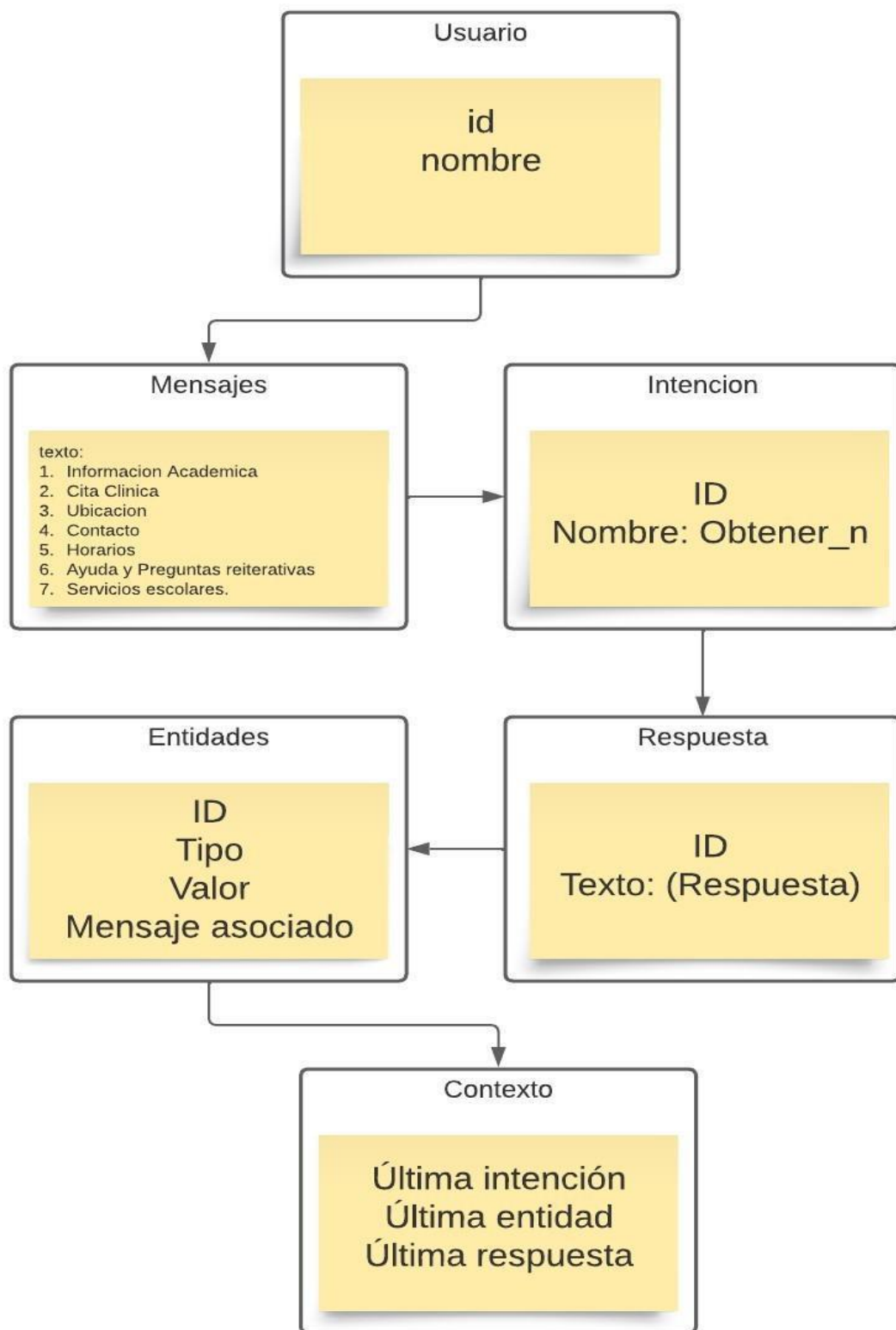


# Diagrama de clases UML





## Modelo de Datos



---

## Modelo de Componentes

Para construir el sistema de un chatbot para una universidad, se requiere una combinación de diferentes componentes de software para que el chatbot funcione de manera efectiva y brinde una experiencia de usuario óptima. Los principales componentes de software son:

- ***Plataforma de desarrollo de chatbots:***

Es el núcleo del sistema, proporcionando el entorno para crear, entrenar y desplegar el chatbot. Puede ser una plataforma específica de chatbots como Dialog Flow, Microsoft Net Framework, IBM Watson Assistant, Raza, etc.

- ***Procesamiento del lenguaje natural (PLN):***

El PLN es esencial para comprender y procesar el lenguaje humano. Incluye técnicas como reconocimiento de entidades, análisis de sentimiento, identificación de intenciones, entre otros. Estas técnicas permiten al chatbot entender y responder adecuadamente a las preguntas y solicitudes de los usuarios.

- ***Base de conocimientos:***

Un chatbot para una universidad debe tener acceso a una base de conocimientos actualizada que contenga información relevante sobre la institución, cursos, programas académicos, eventos, horarios, ubicaciones, entre otros datos importantes.

- ***Integración con sistemas existentes:***

El chatbot debe poder integrarse con sistemas preexistentes de la universidad, como bases de datos, sistemas de gestión de estudiantes, sistemas de información académica, etc. Esto permite que el chatbot acceda a información actualizada y brinde respuestas precisas.

- ***Interfaz de usuario:***

---

El chatbot necesita una interfaz de usuario amigable para que los estudiantes, profesores y personal de la universidad puedan interactuar con él fácilmente. Puede

---

implementarse como un chat en una página web, una aplicación móvil o incluso a través de plataformas de mensajería como Facebook Messenger o WhatsApp.

- ***Integración con redes sociales y sitio web:***

El chatbot puede ser implementado para responder preguntas y brindar asistencia en las redes sociales de la universidad y en su sitio web, lo que mejora la experiencia del usuario y la accesibilidad.

- ***Seguridad y privacidad:***

Dado que los chatbots pueden manejar información sensible, es importante implementar medidas de seguridad y privacidad para proteger los datos de los usuarios y garantizar el cumplimiento de las regulaciones de privacidad.

- ***Capacidades de aprendizaje automático:***

Un chatbot inteligente puede mejorar su rendimiento a lo largo del tiempo mediante técnicas de aprendizaje automático, lo que le permite adaptarse a patrones cambiantes de preguntas y respuestas y mejorar la precisión de sus respuestas.

- ***Métricas y análisis:***

Incorporar herramientas de análisis y seguimiento para evaluar el rendimiento del chatbot y realizar mejoras basadas en datos.

## **Descripción de la Arquitectura**

### **¿Qué es la arquitectura orientada a servicios?**

La **arquitectura orientada a servicios(SOA**, por sus siglas en inglés) es un método de desarrollo de software que utiliza componentes de software llamados servicios para crear aplicaciones empresariales. Cada uno de estos servicios brinda una capacidad

---

empresarial y, además, pueden comunicarse también con el resto de servicios mediante diferentes plataformas y lenguajes. Los desarrolladores usan SOA para

---

reutilizar servicios en diferentes sistemas o combinar varios servicios independientes para realizar tareas complejas.

- **Beneficios:**

Los desarrolladores reutilizan servicios en diferentes procesos empresariales para ahorrar tiempo y dinero. Pueden crear aplicaciones en menos tiempo con SOA en lugar de escribir código y llevar a cabo integraciones desde cero.

***Mantenimiento eficiente*** - Es más fácil crear, actualizar y corregir errores en servicios pequeños que en bloques grandes de código en aplicaciones monolíticas. La modificación de un servicio en SOA no afecta a la funcionalidad general del proceso empresarial.

***Excelente capacidad de adaptación*** - La SOA se adapta de mejor manera a los avances tecnológicos. Puede modernizar sus aplicaciones de forma eficiente y rentable. Por ejemplo, las organizaciones de atención médica pueden utilizar la funcionalidad de sistemas de registro de salud electrónico antiguos en aplicaciones basadas en la nube que son más recientes.

---

- **Componentes:**

**Servicio** - Los servicios son los componentes básicos de la SOA. Pueden ser privados (disponibles únicamente para los usuarios internos de una organización) o públicos (accesibles para todos en Internet). Cada servicio individual tiene tres características principales.

**Implementación de servicios** - La implementación de servicios es el código que crea la lógica para realizar la función de servicio específica, como la autenticación de usuarios o el cálculo de una factura.

**Contrato del servicio** - El contrato del servicio define la naturaleza del servicio y sus condiciones y términos asociados, como los prerequisites para utilizar el servicio, su costo y la calidad del servicio proporcionado.

**Interfaz del servicio** - En SOA, otros servicios o sistemas se comunican con un servicio a través de su interfaz. Esta interfaz define la manera en que se puede invocar al servicio para llevar a cabo actividades o intercambiar datos. Reduce las dependencias entre los servicios y quien los solicita. Por ejemplo, incluso los usuarios con poco o nulo entendimiento de la lógica de código subyacente pueden utilizar un servicio a través de su interfaz

---

**Proveedor de servicios** - El proveedor de servicios crea, mantiene y proporciona uno o más servicios que otros pueden utilizar. Las organizaciones pueden crear sus propios servicios o adquirirlos de proveedores de servicios externos.

**Consumidor de servicios** - El consumidor de servicios solicita al proveedor de estos poner en marcha un servicio específico. Puede ser un sistema completo, aplicación u otro servicio. El contrato de servicio especifica las reglas que el proveedor y el consumidor de servicios deben seguir al momento de interactuar entre sí. Los proveedores y consumidores de servicios pueden pertenecer a departamentos, organizaciones o incluso sectores diferentes.

**Registro de servicios** - Un registro de servicios, o repositorio de servicios, es un directorio de servicios disponibles accesible a través de redes. Almacena documentos descriptivos sobre el servicio que pertenecen a los proveedores de servicios. Los documentos descriptivos contienen información acerca del servicio y cómo comunicarse con él. Los consumidores de servicios pueden descubrir fácilmente los servicios que necesitan por medio de dicho registro.



---

- **¿Qué son los microservicios?**

La arquitectura de microservicios se compone de componentes de software muy pequeños y completamente independientes, llamados microservicios, que se especializan y se centran únicamente en una tarea. Los microservicios se comunican a través de las API, las cuales son reglas que los desarrolladores crean para permitir que otros sistemas de software se comuniquen con su microservicio.

El estilo arquitectónico de los microservicios se adapta de mejor manera a los entornos de computación en la nube modernos.

**Beneficios de los microservicios** - Los microservicios se escalan de forma independiente, son rápidos, portátiles y no dependen de una plataforma, características nativas de la nube. También están desacoplados, lo que significa que tienen poca a nula dependencia en otros microservicios. Para lograrlo, los microservicios tienen acceso local a todos los datos que necesitan en lugar de acceso remoto a datos centralizados a los que acceden y utilizan otros sistemas. Esto crea una duplicación de datos que los microservicios compensan con rendimiento y agilidad.

**SOA en comparación con los microservicios** - La arquitectura de microservicios es una evolución del estilo arquitectónico de SOA. Los microservicios tratan los defectos de SOA para hacer que el software sea más compatible con entornos empresariales modernos basados en la nube. Son detallados y favorecen la

---

duplicación de datos en contraste con el intercambio de datos. Por ello, son completamente independientes y cuentan con sus propios protocolos de comunicación que se exponen a través de API sencillas. Básicamente, es el trabajo de los consumidores utilizar el microservicio a través de su API, lo que elimina la necesidad de un ESB centralizado.

- **¿Qué es una API?**

Las API son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos. Por ejemplo, el sistema de software del instituto de meteorología contiene datos meteorológicos diarios. La aplicación meteorológica de su teléfono “habla” con este sistema a través de las API y le muestra las actualizaciones meteorológicas diarias en su teléfono.

**¿Cómo funcionan las API?** - La arquitectura de las API suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor.

**API de SOAP** - Estas API utilizan el protocolo simple de acceso a objetos. El cliente y el servidor intercambian mensajes mediante XML. Se trata de una API menos flexible que era más popular en el pasado.

**API de RPC** - Estas API se denominan llamadas a procedimientos remotos. El cliente completa una función (o procedimiento) en el servidor, y el servidor devuelve el resultado al cliente.

**API de WebSocket** - La API de WebSocket es otro desarrollo moderno de la API web que utiliza objetos JSON para transmitir datos. La API de WebSocket admite la comunicación bidireccional entre las aplicaciones cliente y el servidor. El servidor puede enviar mensajes de devolución de llamadas a los clientes conectados, por lo que es más eficiente que la API de REST.

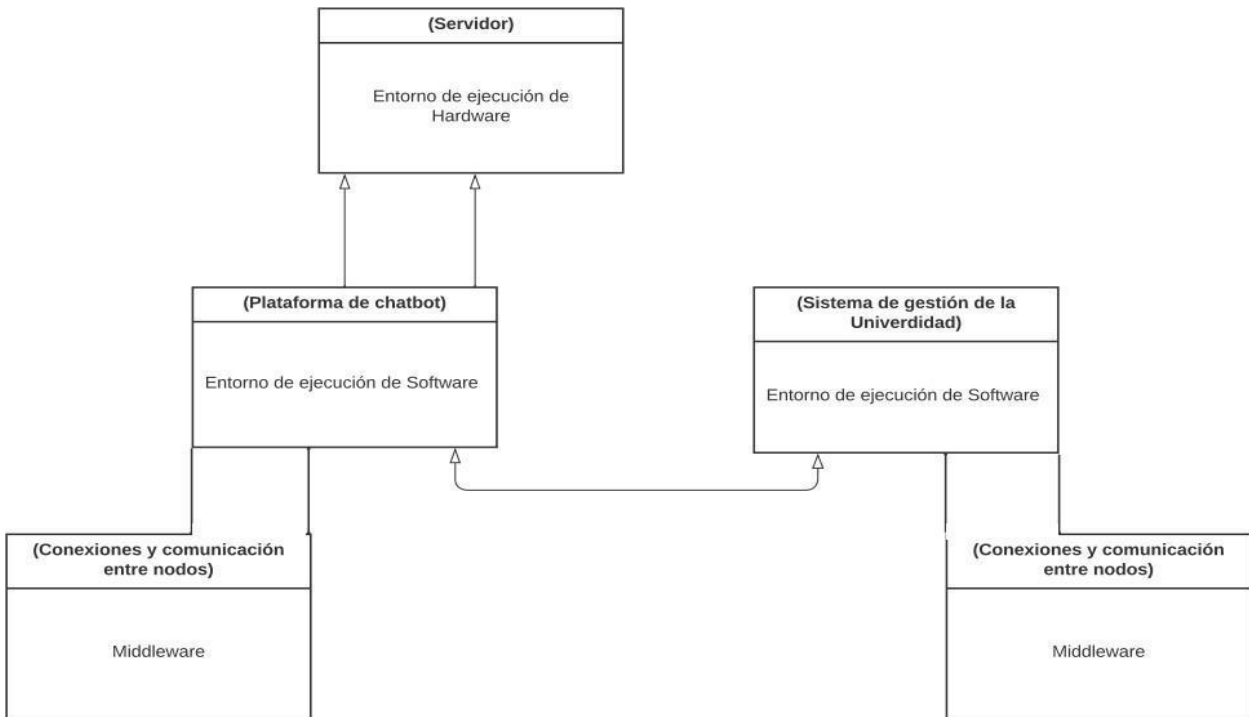
**API de REST** - Estas son las API más populares y flexibles que se encuentran en la web actualmente. El cliente envía las solicitudes al servidor como datos. El servidor utiliza esta entrada del cliente para iniciar funciones internas y devuelve los datos de salida al cliente. Veamos las API de REST con más detalle a continuación.

---

- **Diferencias entre RPC y REST**

	RPC	REST
¿Qué es?	Un sistema permite a un cliente remoto llamar a un procedimiento en un servidor como si fuera local.	Un conjunto de reglas que define el intercambio de datos estructurados entre un cliente y un servidor.
Se utiliza para lo siguiente:	Realizar acciones en un servidor remoto.	Realizar operaciones de creación, lectura, actualización y eliminación (CRUD) de objetos remotos.
Uso idóneo	Cuando se requieran cálculos complejos o desencadenar un proceso remoto en el servidor.	Cuando los datos del servidor y las estructuras de datos deban exponerse de manera uniforme.
Control de estado	Con o sin estado.	Sin estado.
Formato de transmisión de datos	En una estructura coherente definida por el servidor y aplicada en el cliente.	En una estructura determinada independientemente por el servidor. Se pueden transferir varios formatos diferentes dentro de la misma API.

## Modelo de Despliegue



- Los dos nodos superiores representan el entorno de ejecución de hardware y el entorno de ejecución de software para el chatbot. El entorno de ejecución de hardware puede ser un servidor donde se aloja el chatbot.
- El nodo de "Entorno de Ejecución de Software" representa la plataforma de chatbot, como Dialog Flow, Microsoft Net Framework, Raza, o cualquier otra plataforma de desarrollo de chatbot seleccionada.
- El nodo de "Entorno de Ejecución de Software" se conecta al nodo de "Entorno de Ejecución de Hardware" mediante un middleware, que representa las conexiones y la comunicación entre el servidor y la plataforma de chatbot.
- El tercer nodo inferior representa el "Entorno de Ejecución de Software" para el sistema de gestión de la universidad, que puede ser un sistema de gestión de estudiantes, bases de datos, sistemas de información académica, etc.
- El nodo de "Entorno de Ejecución de Software" para el sistema de gestión de la universidad se conecta al nodo de "Entorno de Ejecución de Hardware" mediante

---

otro middleware, que permite la comunicación entre el chatbot y el sistema de gestión de la universidad.

## Git y Github

Git y GitHub son herramientas esenciales en el mundo del desarrollo de software y la colaboración en proyectos de código abierto. Permíteme explicarte más detalladamente sobre cada uno de ellos.

### Git:

Git es un sistema de control de versiones distribuido desarrollado por Linus Torvalds en 2005. Su función principal es realizar un seguimiento de los cambios en el código fuente durante el desarrollo de software y permitir a los desarrolladores trabajar en equipo de manera eficiente. Algunos conceptos clave de Git son:

1. **Repositorio:** Es un almacén que contiene todos los archivos, historiales y metadatos de un proyecto.
2. **Commit:** Un commit es una instantánea del estado actual de los archivos en un momento específico. Cada comité tiene un mensaje descriptivo que explica qué cambios se realizaron.
3. **Rama (Branch):** Una rama es una línea independiente de desarrollo. Permite a los desarrolladores trabajar en características o correcciones por separado sin afectar la rama principal (generalmente llamada "master" o "main").
4. **Fusión (Merge):** Fusionar es el proceso de combinar los cambios de una rama en otra, lo que integra el trabajo de diferentes desarrolladores.

### GitHub:

GitHub es una plataforma de alojamiento de código fuente y colaboración en proyectos basada en Git. Permite a los desarrolladores almacenar, administrar y compartir sus repositorios de código. Algunos aspectos importantes de GitHub son:

1. **Repositorios Remotos:** Los repositorios de código se pueden alojar en GitHub como repositorios remotos, lo que permite a los equipos colaborar en el mismo proyecto desde diferentes ubicaciones geográficas.

- 
2. **Pull Request:** Una solicitud de extracción (pull request) es una característica de GitHub que permite a los desarrolladores proponer cambios a un repositorio. Otros desarrolladores pueden revisar estos cambios, comentar sobre ellos y, finalmente, fusionarlos si son adecuados.
  3. **Issues:** Los problemas (issues) son un medio para realizar un seguimiento de tareas, mejoras, errores y discusiones en un proyecto. Los desarrolladores y los colaboradores pueden colaborar para resolver problemas específicos.
  4. **Colaboración:** GitHub fomenta la colaboración al facilitar la comunicación entre desarrolladores, la revisión de código y la documentación de proyectos.
  5. **GitHub Pages:** Permite a los desarrolladores alojar sitios web estáticos directamente desde los repositorios de GitHub.

En resumen, Git es el sistema de control de versiones que permite el seguimiento de cambios y el control de versiones de un proyecto, mientras que GitHub es una plataforma que utiliza Git para alojar y colaborar en repositorios de código fuente. Juntas, estas herramientas han revolucionado la forma en que los equipos de desarrollo trabajan en proyectos y colaboran en todo el mundo.

---

# Manual para programadores:

## ¿Qué utilizamos para este desarrollo?

@bot-whatsapp es una librería que te permitirá crear chatbot para WhatsApp .A lo largo de esta documentación encontrarás ejemplos y material de ayuda.

Esta documentación te ayudará a instalar tu bot de WhatsApp en simples pasos; con el propósito de que tengas un chatbot funcional en solo minutos.

## Requerimientos

A continuación se describen los puntos técnicos que debes de tener en cuenta antes de trabajar con esta herramienta

- Node v16 o superior - [descargar node](#)
  - Git - [descargar Git](#)
- 

## ¿Cómo saber que tengo el Node?

Solo debes ejecutar el siguiente comando y esperar que la versión que te arroja sea superior a v16

```
$ node -v
```

```
v18.12.1
```

---

## ¿Cómo instalar Node?

- **Windows:** [Ver video](#). Si necesitas ayuda para instalar Node en Windows. A continuación te comparto un video en el minuto exacto donde explico como instalar.

- 
- **Ubuntu:** Te comparto un recurso de [Digital Ocean](#) donde explica como instalar node en Ubuntu.
  - **Mac-Os:** Te comparto un recurso de [Digital Ocean](#) donde explica como instalar node en Mac-Os.
- 

## ¿Cómo saber que tengo Git?

Solo debes ejecutar el siguiente comando y esperar que te mande la versión que tienes instalada, si te manda un error de comando no reconocido es que no lo tienes instalado.

```
$ git -v
```

```
git
```

---

## ¿Cómo instalar Git?

- Solo es necesario instalar Git si estás usando **Windows**, ya que Mac y Linux lo traen preinstalado.
- Lo puedes descargar desde esta [liga](#) .
- Descarga la versión necesaria para tu sistema operativo (32-bit o 64-bit).
- Una vez terminada la descarga, ejecuta el archivo descargado y dale "Siguiente" en todas las pantallas.
- Haz clic en el botón de "Finalizar".

## Instalación

Con esta librería, puedes construir flujos automatizados de conversación de manera agnóstica al proveedor de WhatsApp, configurar respuestas automatizadas para preguntas frecuentes, recibir y responder mensajes de manera automatizada, y hacer un seguimiento de las interacciones con los clientes. Además, puedes configurar fácilmente disparadores que te ayudarán a expandir las funcionalidades sin límites.



---

## Comenzamos

Para clonar un repositorio de GitHub en Windows, puedes utilizar el comando `git clone` en la línea de comandos. Aquí te muestro cómo hacerlo:

1. Abre el "Command Prompt" (Símbolo del sistema) en tu computadora. Puedes hacerlo buscando "cmd" en el menú de inicio.
2. Navega a la ubicación donde deseas clonar el repositorio. Puedes hacerlo utilizando el comando `cd` seguido de la ruta de la carpeta. Por ejemplo, si deseas clonar el repositorio en el escritorio, puedes usar el siguiente comando (asegúrate de modificar la ruta según tus preferencias):  
bash

```
cd C:\Users\TuUsuario\Desktop
```

Ahora, utiliza el comando `git clone` seguido de la URL del repositorio que deseas clonar. En tu caso, sería:

bash

```
git clone https://github.com/KirbyMondragon/BPFS_Chatbot.git
```

- 1.
2. Presiona Enter y Git comenzará a descargar los archivos del repositorio en la ubicación que especificaste.

Una vez que se complete el proceso de clonación, tendrás una copia local del repositorio en tu computadora.

## Pruébalo

Si copias y pegas este código y tu entorno de trabajo cumple con todos los requisitos te debe funcionar abajo explico muy por encima

```
const { createBot, createProvider, createFlow, addKeyword } = require('@bot-whatsapp/bot')
```

```
const WebWhatsappProvider = require('@bot-whatsapp/provider/web-whatsapp')
```

```
const MockAdapter = require('@bot-whatsapp/database/mock')
```

```
const flowPrincipal = addKeyword(['hola', 'ole', 'alo'])
```

```
  .addAnswer(['Hola, bienvenido a mi tienda', '¿Cómo puedo ayudarte?'])
```

```
  .addAnswer(['Tengo:', 'Zapatos', 'Bolsos', 'etc ...'])
```

```
/**
```

```
 * Esta es la funcion importante es la que realmente inicia
```

```
 * el chatbot.
```

```
*/
```

```
const main = async () => {
```

```
  const adapterDB = new MockAdapter()
```

```
  const adapterFlow = createFlow([flowPrincipal])
```

```
const adapterProvider = createProvider(WebWhatsappProvider)
```

```
  createBot({
```

```
    flow: adapterFlow,
```

```
    provider: adapterProvider,
```

```
    database: adapterDB,
```

```
  })
```

```
}
```

```
main()
```

---

## Explicando código

En esta parte solo estamos declarando las dependencias que vamos a utilizar. Si quieres saber a fondo cada una de las funciones te recomiendo pasarte por la sección de [conceptos](#)

```
const { createBot, createProvider, createFlow, addKeyword } = require('@bot-whatsapp/bot')

const WebWhatsappProvider = require('@bot-whatsapp/provider/web-whatsapp')

const MockAdapter = require('@bot-whatsapp/database/mock')
```

En la siguiente sección te declaramos las palabras claves que disparan un flujo de conversación.

### Ejemplo:

Si un usuario te escribe 🧑 hola ó 🧑 alo el bot responderá

🤖 **Hola, bienvenido a mi tienda, ¿Como puedo ayudarte?**

```
const flowPrincipal = addKeyword(['hola', 'alo'])

.addAnswer(['Hola, bienvenido a mi tienda', '¿Como puedo ayudarte?'])

.addAnswer(['Tengo:', 'Zapatos', 'Bolsos', 'etc ...'])
```

## Esenciales

### Conceptos

El desarrollo de la librería se base en tres (3) piezas claves para su correcto funcionamiento:

- **Flow:** Encargado de construir todo el contexto de la conversación, finalmente su objetivo es brindar una capa amigable al desarrollador.
- **Provider:** Como si de un [conector](#) se tratara el objetivo es poder cambiar facilmente de proveedor de Whatsapp en minutos sin el riesgo de dañar otras partes del bot
- **Database:** Siguiendo la línea de pensamiento de conectores, de igual manera que el provider nos brinda la capacidad de poder cambiar de capa de persistencia de datos (guardar datos) sin invertir tiempo en reescribir nuestro flujo.

---

### Flow (Flujos)

Los flujos hacen referencia al hecho de construir un flujo de conversión. Esto es un flow podemos observar que están presente dos métodos importantes **addKeyword** y el **addAnswer**.

Tan sencillo como decir **palabra/s clave** y **mensaje a responder**

Ambos métodos [addKeyword](#) y el [addAnswer](#) tienen una serie opciones disponibles

---

```
const { createBot, createProvider, createFlow, addKeyword } = require('@bot-whatsapp/bot')

const flowPrincipal = addKeyword(['hola', 'alo'])

  .addAnswer(['Hola, bienvenido a mi tienda', '¿Como puedo ayudarte?'])

  .addAnswer(['Tengo:', 'Zapatos', 'Bolsos', 'etc ...'])
```

---

## Provider (Proveedor)

⚡ Dependiendo del tipo de proveedor que utilices puede que necesites pasar algunas configuración adicional como **token, api, etc.** para esos casos te recomendamos guiarte de los [starters](#) o si gustas puedes editar esta documentación para ir agregando más info

Es la pieza que conecta tu flujo con Whatsapp. En este chatbot tenemos varios proveedores disponibles la mayoría gratis pero también tenemos integración la api oficial de whatsapp o twilio

```
const WhatsappProvider = require('@bot-whatsapp/provider/web-whatsapp')

....

const adapterProvider = createProvider(WhatsappProvider)
```

Los proveedores disponibles hasta el momento son los siguientes:

---

```
whatsapp-web.js require('@bot-whatsapp/provider/web-whatsapp')
```

```
Venom require('@bot-whatsapp/provider/venom')
```

```
Baileys require('@bot-whatsapp/provider/baileys')
```

```
Meta Oficial require('@bot-whatsapp/provider/meta')
```

```
Twilio Oficial require('@bot-whatsapp/provider/twilio')
```

---

## DataBase (Base de datos)

⚡ Dependiendo del tipo de conector que utilices puede que necesites pasar algunas configuración adicional como **user**, **host**, **password** para esos casos te recomendamos guiarte de los [starters](#) o si gustas puedes editar esta documentación para ir agregando más info

Es la pieza encargada de mantener el **"estado"** de una conversación, para mayor facilidad la librería te proporciona diferentes conectores que se dé adapten mejor a tu desarrollo

```
const MongoAdapter = require('@bot-whatsapp/database/mongo')

....

const adapterDB = new MongoAdapter({

  dbUri: 'mongodb://0.0.0.0:27017',

  dbName: 'db_bot',

})
```

---

```
require('@bot-whatsapp/database/json')
```

Los conectores disponibles hasta el momento son los siguientes:

```
require('@bot-whatsapp/database/mock')
```

```
require('@bot-whatsapp/database/mongo')
```

```
require('@bot-whatsapp/database/mysql')
```

## Flow

Los flujos hace referencia al hecho de construir un flujo de conversion. Esto es un flow podemos observar que estan presente dos metodos importantes **addKeyword** y el **addAnswer**.

Tan sencillo como decir **palabra/s clave** y **mensaje a responder**

Ambos metodos [addKeyword](#) y el [addAnswer](#) tienen una serie opciones disponibles

```
const { createBot, createProvider, createFlow, addKeyword } = require('@bot-whatsapp/bot')

const flowPrincipal = addKeyword(['hola', 'alo'])

  .addAnswer(['Hola, bienvenido a mi tienda', '¿Como puedo ayudarte?'])
```

---

```
.addAnswer(['Tengo:', 'Zapatos', 'Bolsos', 'etc ...'])
```

---

## blackList

Éste argumento se utiliza para **evitar que el bot se active** cuando los números de la lista se activan. Es importante que el número **vaya acompañado de su prefijo**, en el caso de México "52".

```
createBot(  
  {  
    flow: adapterFlow,  
    provider: adapterProvider,  
    database: adapterDB,  
  },  
  {  
    blackList: ['34XXXXXXXXXX', '34XXXXXXXXXX', '34XXXXXXXXXX', '34XXXXXXXXXX'],  
  }  
)
```

---

## addKeyword()



---

Esta función se utiliza para iniciar un flujo de conversión.

Recibe un **string** o un **array** de string [ 'hola', 'buenas' ].

### Opciones

- sensitive: Sensible a mayúsculas y minúsculas por defecto **false**

```
const { addKeyword } = require('@bot-whatsapp/bot')

const flowString = addKeyword('hola')

const flowArray = addKeyword(['hola', 'alo'])

const flowSensitive = addKeyword(['hola', 'alo'], {
  sensitive: true,
})
```

---

### addAnswer()

Esta función se utiliza para responder un mensaje despues del **addKeyword()**

### Opciones

- delay: 0 (milisegundos)
- media: url de imagen
- buttons: array [ {body:'Button1'}, {body:'Button2'}, {body:'Button3'} ]
- capture: true (para esperar respuesta)
- child: Objeto tipo flujo o arra de flujos hijos

---

Dato importante : los botones solo funcionan con Meta como proveedor

```
const { addKeyword } = require('@bot-whatsapp/bot')
```

```
const flowString = addKeyword('hola').addAnswer('Este mensaje se enviara 1 segundo despues', {  
  delay: 1000,  
})
```

```
const flowString = addKeyword('hola').addAnswer('Este mensaje envia una imagen', {  
  media: 'https://i.imgur.com/0HpzsEm.png',  
})
```

```
const flowString = addKeyword('hola').addAnswer('Este mensaje envía tres botones', {  
  buttons: [{ body: 'Boton 1' }, { body: 'Boton 2' }, { body: 'Boton 3' }],  
})
```

```
const flowString = addKeyword('hola').addAnswer('Este mensaje espera una respuesta del usuario', {  
  capture: true,  
})
```

---

---

ctx

Este argumento se utiliza para obtener el contexto de la conversación

```
const { addKeyword } = require('@bot-whatsapp/bot')

const flowString = addKeyword('hola').addAnswer('Indica cual es tu email', null, (ctx) => {
  console.log('👉 Informacion del contexto: ', ctx)
})
```

---

fallBack()

Esta función se utiliza para volver a enviar el último mensaje abajo un ejemplo. En el ejemplo de abajo esperamos que el usuario ingrese un mensaje que contenga @ sino contiene se repetirá el mensaje **Indica cual es tu email**

```
const { addKeyword } = require('@bot-whatsapp/bot')

const flowString = addKeyword('hola').addAnswer('Indica cual es tu email', null, (ctx, { fallBack })
=> {
```

```
if (!ctx.body.includes('@')) return fallBack()
})
```

---

## flowDynamic()

Esta función se utiliza para devolver mensajes dinámicos que pueden venir de una API o Base de datos. La función recibe un array que debe contener la siguiente estructura:

```
[{body: 'Mensaje'}, {body: 'Mensaje2'}]
```

```
const { addKeyword } = require('@bot-whatsapp/bot')

const flowString = addKeyword('hola')

.addAnswer('Indica cual es tu email', null, async (ctx, {flowDynamic}) => {

  const mensajesDB = () => {

    const categories = db.find(...)

    const mapDatos = categories.map((c) => ({body:c.name}))

    return mapDatos

  }

  await flowDynamic(mensajesDB())

})
```

---

---

endFlow()

Esta función se utiliza para **finalizar un flujo con dos** o más addAnswer. Un ejemplo de uso sería registrar 3 datos de un usuario en 3 preguntas distintas y que el usuario **pueda finalizar por él mismo el flujo**. Como podrás comprobar en el ejemplo siguiente, se puede vincular flowDynamic y todas sus funciones; como por ejemplo botones.

```
let nombre;  
  
let apellidos;  
  
let telefono;  
  
const flowFormulario = addKeyword(['Hola',  Volver al Inicio'])  
  
  .addAnswer(  
  
    ['Hola!','Para enviar el formulario necesito unos datos...' , 'Escriba su *Nombre*'],  
  
    { capture: true, buttons: [{ body: '✖ Cancelar solicitud' }] },  
  
    async (ctx, { flowDynamic, endFlow }) => {  
  
      if (ctx.body == '✖ Cancelar solicitud')  
  
        return endFlow({body: '✖ Su solicitud ha sido cancelada ✖', // Aquí terminamos el  
flow si la condicion se comple
```

```
        buttons:[{body:'🔙 Volver al Inicio' }] // Y además, añadimos un botón por
si necesitas derivarlo a otro flow
```

```
    ))

    nombre = ctx.body

    return flowDynamic(`Encantado *${nombre}*, continuamos...`)

  }

)

.addAnswer(

  ['También necesito tus dos apellidos'],

  { capture: true, buttons: [{ body: '❌ Cancelar solicitud' }] },

  async (ctx, { flowDynamic, endFlow }) => {

    if (ctx.body == '❌ Cancelar solicitud')

      return endFlow([body: '❌ Su solicitud ha sido cancelada ❌',

        buttons:[{body:'🔙 Volver al Inicio' }]

      ])

    apellidos = ctx.body

    return flowDynamic(`Perfecto *${nombre}*, por último...`)

  }

)
```

```
.addAnswer(

  ['Dejeme su número de teléfono y le llamaré lo antes posible.'],

  { capture: true, buttons: [{ body: '✖ Cancelar solicitud' }] },

  async (ctx, { flowDynamic, endFlow }) => {

    if (ctx.body == '✖ Cancelar solicitud')

      return endFlow({body: '✖ Su solicitud ha sido cancelada ✖',

        buttons:[{body:'⬅ Volver al Inicio' }]

      })

    telefono = ctx.body

    await delay(2000)

    return flowDynamic(`Estupendo *${nombre}*! te dejo el resumen de tu formulario

    \n- Nombre y apellidos: *${nombre} ${apellidos}*

    \n- Telefono: *${telefono}*)

  }

)
```

---

## QR Portal Web

Argumento para asignar nombre y puerto al BOT

```
const BOTNAME = 'bot'

QRPortalWeb({ name: BOTNAME, port: 3005 })
```

## Proveedores

⚡ Dependiendo del tipo de proveedor que utilices puede que necesites pasar algunas configuración adicional como **token, api, etc.** para esos casos te recomendamos guiarte de los [starters](#) o si gustas puedes editar esta documentación para ir agregando más info

Es la pieza que conecta tu flujo con Whatsapp. En este chatbot tenemos varios proveedores disponibles la mayoría gratis pero también tenemos integración la api oficial de whatsapp o twilio

```
const WhatsappProvider = require('@bot-whatsapp/provider/web-whatsapp')

....

const adapterProvider = createProvider(WhatsappProvider)
```

Los proveedores disponibles hasta el momento son los siguientes:

[whatsapp-web.js](#) `require('@bot-whatsapp/provider/web-whatsapp')`

[WPPConnect](#) `require('@bot-whatsapp/provider/wppconnect')`

[Venom](#) `require('@bot-whatsapp/provider/venom')`



---

[Baileys](#) `require('@bot-whatsapp/provider/baileys')`

[Meta Official](#) `require('@bot-whatsapp/provider/meta')`

[Twilio Official](#) `require('@bot-whatsapp/provider/twilio')`

---

## Twilio: Configuración

Estamos trabajando en el apartado de la documentación lo más claro posible. Puedes encontrar los [detalles aquí](#)

---

## Meta: Configuración

Estamos trabajando en el apartado de la documentación lo más claro posible. Puedes encontrar los [detalles aquí](#)

## DataBase (Base de datos)

⚡ Dependiendo del tipo de conector que utilices puede que necesites pasar algunas configuración adicional como **user**, **host**, **password** para esos casos te recomendamos guiarte de los [starters](#) o si gustas puedes editar esta documentación para ir agregando más info

Es la pieza encargada de mantener el "**estado**" de una conversación, para mayor facilidad la librería te proporciona diferentes conectores que se dé adapten mejor a tu desarrollo

```
const MongoAdapter = require('@bot-whatsapp/database/mongo')
```

```
....
```

```
const adapterDB = new MongoAdapter({  
  dbUri: 'mongodb://0.0.0.0:27017',  
  dbName: 'db_bot',  
})
```

Los conectores disponibles hasta el momento son los siguientes:

```
require('@bot-whatsapp/database/mock')
```

```
require('@bot-whatsapp/database/mongo')
```

```
require('@bot-whatsapp/database/mysql')
```

```
require('@bot-whatsapp/database/json')
```

## Migración

### Versión (legacy)

En la **versión (legacy)** se implementan los flujos de esta manera, en dos archivos independientes.

**initial.json** para establecer las palabras claves y el flujo a responder, por otro lado también se necesitaba implementar. **response.json** donde se escriben los mensajes a responder.

```
//initial.json
```

```
[
  {
    "keywords": ["hola", "ola", "alo"],
    "key": "hola"
  },
  {
    "keywords": ["productos", "info"],
    "key": "productos"
  },
  {
    "keywords": ["adios", "bye"],
    "key": "adios"
  },
  {
    "keywords": ["imagen", "foto"],
    "key": "catalogo"
  }
]
```

//response.json

```
{
  "hola": {
    "replyMessage": ["Gracias a ti! \n"],
    "media": null,
```

```
"trigger": null

},

"adios": {

  "replyMessage": ["Que te vaya bien!!"]

},

"productos": {

  "replyMessage": ["Más productos aquí"],

  "trigger": null,

  "actions": {

    "title": "¿Que te interesa ver?",

    "message": "Abajo unos botons",

    "footer": "",

    "buttons": [{ "body": "Telefonos" }, { "body": "Computadoras" }, { "body": "Otros" }]

  }

},

"catalogo": {

  "replyMessage": ["Te envio una imagen"],

  "media": "https://media2.giphy.com/media/VQJu0leULuAmCwf5SL/giphy.gif",

  "trigger": null

}

}
```

---

## Versión 2 (0.2.X)

En esta versión es mucho más sencillo, abajo encontrarás un ejemplo del mismo flujo anteriormente mostrado.

```
//app.js

const { createBot, createProvider, createFlow, addKeyword, addChild } =
require('@bot-whatsapp/bot')

const BaileysProvider = require('@bot-whatsapp/provider/baileys') //Provider
const MockAdapter = require('@bot-whatsapp/database/mock') //Base de datos

/**
 * Declarando flujos principales.
 */

const flowHola = addKeyword(['hola', 'ola', 'alo']).addAnswer('Bienvenido a tu tienda online!')

const flowAdios = addKeyword(['adios', 'bye']).addAnswer('Que te vaya bien!!').addAnswer('Hasta luego!')

const flowProductos = addKeyword(['productos', 'info']).addAnswer('Te envio una imagen', {
  buttons: [{ body: 'Telefonos' }, { body: 'Computadoras' }, { body: 'Otros' }],
})

const flowCatalogo = addKeyword(['imagen', 'foto']).addAnswer('Te envio una imagen', {
  media: 'https://media2.giphy.com/media/VQJu0leULuAmCwf5SL/giphy.gif',
})
```

```
const main = async () => {  
  
  const adapterDB = new MockAdapter()  
  
  const adapterFlow = createFlow([flowHola, flowAdios, flowProductos, flowCatalogo]) //Se  
  crean los flujos.  
  
  const adapterProvider = createProvider(BaileysProvider)  
  
  createBot({  
  
    flow: adapterFlow,  
  
    provider: adapterProvider,  
  
    database: adapterDB,  
  
  })  
}
```

## Flujos hijos

A continuación se muestra un ejemplo de flujos hijos, estos nos sirven para crear flujos que solo se disparan cuando el flujo anterior es el especificado, ejemplo:

Menú Principal (Escoge zapatos o bolsos)

- SubMenú 1 (Elegiste bolsos, ahora escoge piel o tela)
  - Submenu 1.1 (piel)
- Submenú 2 (Elegiste zapatos, ahora escoge piel o tela)
  - Submenu 2.1 (piel)

El **submenú 1** solo se va a disparar cuando el flujo anterior sea el **principal**, e igualmente el **submenu 1.1**, sólo cuando el flujo anterior sea el **submenú 1**, ejemplo:

```
/**
```

\* Aquí declaramos los flujos hijos, los flujos se declaran de atrás para adelante, es decir que si tienes un flujo de este tipo:

```
*
```

```
*     Menú Principal
```

```
*     - SubMenu 1
```

```
*         - Submenu 1.1
```

```
*         - Submenu 2
```

```
*         - Submenu 2.1
```

```
*
```

\* Primero declaras los submenus 1.1 y 2.1, luego el 1 y 2 y al final el principal.

```
*/
```

```
const flowBolsos2 = addKeyword(['bolsos2', '2'])
```

```
  .addAnswer('🤖 *MUCHOS* bolsos ...')
```

```
  .addAnswer('y mas bolsos... bla bla')
```

```
const flowZapatos2 = addKeyword(['zapatos2', '2'])
```

```
  .addAnswer('🤖 repito que tengo *MUCHOS* zapatos.')
```

```
  .addAnswer('y algunas otras cosas.')
```

```
const flowZapatos = addKeyword(['1', 'zapatos', 'ZAPATOS'])
```

```
  .addAnswer('🤖 Veo que elegiste zapatos')
```

```
  .addAnswer('Tengo muchos zapatos...bla bla')
```

```
.addAnswer(  
  ['Manda:', '*(2) Zapatos2*', 'para mas información'],  
  { capture: true },  
  (ctx) => {  
    console.log('Aquí puedes ver más info del usuario...')  
    console.log('Puedes enviar un mail, hook, etc..')  
    console.log(ctx)  
  },  
  [...addChild(flowZapatos2)]  
)
```

```
const flowBolsos = addKeyword(['2', 'bolsos', 'BOLSOS'])
```

```
.addAnswer('👏👏 Veo que elegiste bolsos')  
.addAnswer('Tengo muchos bolsos...bla bla')  
.addAnswer(  
  ['Manda:', '*(2) Bolsos2*', 'para mas información'],  
  { capture: true },  
  (ctx) => {  
    console.log('Aquí puedes ver más info del usuario...')  
    console.log('Puedes enviar un mail, hook, etc..')  
    console.log(ctx)  
  },  
  [...addChild(flowBolsos2)]  
)
```



```
/**
 * Declarando flujo principal
 */
const flowPrincipal = addKeyword(['hola', 'ole', 'alo'])

  .addAnswer(['Hola, bienvenido a mi tienda', '¿Como puedo ayudarte?'])

  .addAnswer(['Tengo:', 'Zapatos', 'Bolsos', 'etc ...'])

  .addAnswer(

    ['Para continuar escribe:', '*(1) Zapatos*', '*(2) Bolsos*'],

    { capture: true },

    (ctx) => {

      console.log('Aquí puedes ver más info del usuario...')

      console.log('Puedes enviar un mail, hook, etc..')

      console.log(ctx)

    },

    [...addChild(flowBolsos), ...addChild(flowZapatos)]

  )
```

---

# Despliegue

## Cómo realizar la instalación

### Entorno Local

Si deseas tener tu chatbot en ejecución en un servidor local (computadora personal, etc.) esta guía te ayudará. El servidor local deberá cumplir con los requisitos mínimos, puedes ver en [este enlace](#).

---

Si deseas instalar pm2 puedes ejecutar `npm install pm2 --global`

Debes ubicarte en el directorio donde tienes el código fuente de tu chatbot.

Independientemente de tu sistema operativo deberás ejecutar el chatbot con el comando a través de un sistema que mantenga el proceso en ejecución. Recomendamos [Pm2](#)

```
pm2 start app.js --name=bot1
```

id	name	mode	↻	status	cpu	memory
0	bot1	fork	0	online	0%	32.1mb

La consola de devolver un mensaje con una lista de procesos, en el ejemplo puedes observar, que tenemos un proceso llamado `bot1`

De esta manera ya puedes cerrar la terminal y tu bot seguirá en ejecución sin problema






---

# Entorno Docker

Previamente, necesitas tener instalado Docker en tu servidor dependiendo del sistema operativo, los procesos cambian, puedes encontrar toda la información oficial de docker en [este enlace](#).

---

Dependiendo del proveedor que hayas elegido necesitarás una implementación de Docker específica, pero no te preocupes, ya que viene implementada automáticamente en un archivo llamado **Dockerfile**, también puedes ver los otros Dockerfile en el apartado de [plantillas](#).

🔗 main ▾	bot-whatsapp / starters / apps / base-baileys-json /	Go to file	Add file ▾	...
	cheveguerra feat: mod de starters para habilitar portal ...	eceb170	5 days ago	🕒 History
..				
	Dockerfile	fix: 🔥 update qr package		5 days ago
	README.md	fix: 🔥 update qr package		5 days ago
	app.js	feat: mod de starters para habilitar portal		5 days ago
	package.json	fix: 🔥 update qr package		5 days ago

---

## Construir imagen

Solo es necesario construir la imagen del docker lo puedes hacer con el siguiente comando

```
docker build . -t botwhatsapp:latest
```

## Iniciar contenedor

Para iniciar el contenedor con la imagen previamente construida puedes realizarlo ejecutando el siguiente comando. Se utiliza el puerto **3001** solo con un ejemplo puedes usar el puerto que tu quieras

```
docker run -e PORT=3001 -p 3001:3001 botwhatsapp:latest
```

---

# Entorno Cloud

Si deseas tener tu chatbot en ejecución en un servidor en la nube esta guía te ayudará. El servidor deberá cumplir con los requisitos mínimos, puedes ver en [este enlace](#).

---

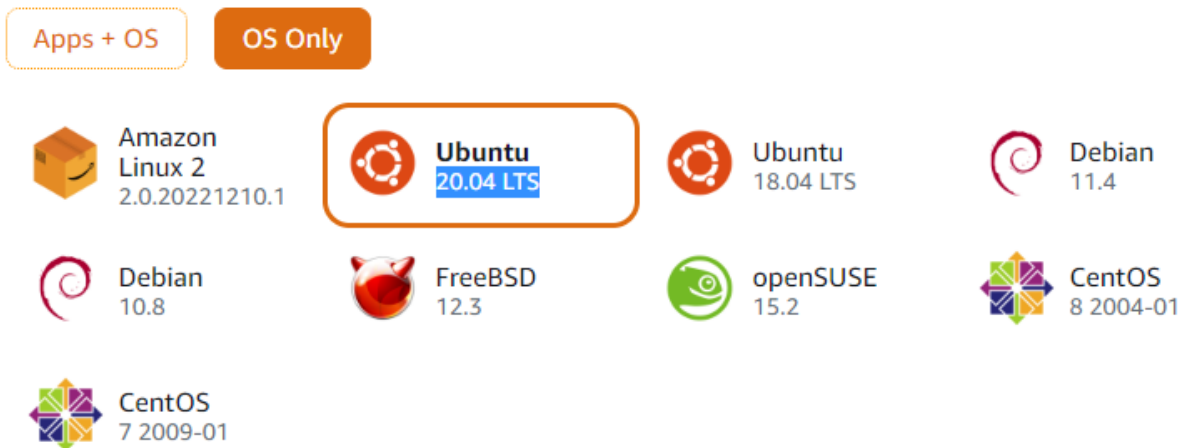
Dependiendo de tu proveedor de **servicio Cloud** debes crear una instancia (máquina virtual), este ejemplo iremos orientando en un entorno de AWS. En nuestro ejemplo creamos una máquina virtual con **Ubuntu 20.04**

Pick your instance image 

Select a platform



Select a blueprint



---

Posterior al proceso de crear la máquina esperamos unos minutos hasta que ya está operativo y tomamos nota del usuario y la IP pública para proceder a conectarnos vía SSH

---

CONNECT TO

**34.228.208.104**

IPv6: 2600:1f18:8e5:bb00:3ada:7fb4:97e7:42f9

USER NAME

**ubuntu**

SSH KEY

This instance was created with the personal SSH key named **api\_Ts**.

Manage your SSH keys from your [Account](#) page.

## Conectarse via SSH

Luego de obtener los datos necesarios para conectarnos a nuestra máquina, procedemos a hacerlo

```
ssh -i llaveBot.pem ubuntu@34.228.208.104
```

---

Luego puede aparecer un mensaje como el siguiente donde solo debes de responder **yes**

```
$ ssh -i llaveBot.pem ubuntu@34.228.208.104
The authenticity of host '34.228.208.104 (34.228.208.104)' can't be established.
ED25519 key fingerprint is SHA256:VvIHlDmgJY6UvitpXdGg6SrsVMshysYz0BgoCv6o34.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? |
```

---

Una vez conectado ya estás dentro de la máquina virtual, te aconsejamos la primera vez hacer una actualización de dependencias de Ubuntu con los siguientes comandos

```
sudo apt-get update
```

---

```
sudo apt-get upgrade
```

---

## Recuerda instalar Node 16 o superior

Puedes ver más a detalle los pasos de la instalación en este [blog](#)

```
curl -sL https://deb.nodesource.com/setup_18.x -o nodesource_setup.sh
```

```
sudo bash nodesource_setup.sh
```

```
sudo apt-get install -y nodejs
```

---

## Implementar el bot

Si tienes el código de tu chatbot en un repositorio, solo falta que clones el repo en el servidor y ejecutes **npm start**

Para escanear el **QR** puedes hacerlo vía WEB accediendo a la URL

**http://[TU\_IP\_PUBLICA]:3000** en este ejemplo sería **http://34.228.208.10:3000**

```
ubuntu@ip-172-26-14-26: ~/base-baileys-memory$ npm start

> base-bailey-memory@1.0.0 start
> node app.js

▶ ESCANEAR QR ▶
Existen varias maneras de escanear el QR code
- Tambien puedes visitar http://localhost:3000
- Se ha creado un archivo que finaliza qr.png

⚡ ⚡ ACCIÓN REQUERIDA ⚡ ⚡
Debes escanear el QR Code para iniciar bot.qr.png
Recuerda que el QR se actualiza cada minuto
Necesitas ayuda: https://link.codigoencasa.com/DISCORD
```

---

## Firewall

Si no te abre la página web asegúrate de tener el puerto abierto en tu firewall. Ejemplo permitir el puerto **3000**

# IPv4 Firewall ?

Create rules to open ports to the internet, or to a specific IPv4 address or range.

[Learn more about firewall rules](#)

+ Add rule

Specify a port and protocol to open. Specify a port range using a dash, such as 0 - 65535.

Application

Custom

Protocol

TCP

Port or range

3000

☐ Restrict to IP address

Cancel

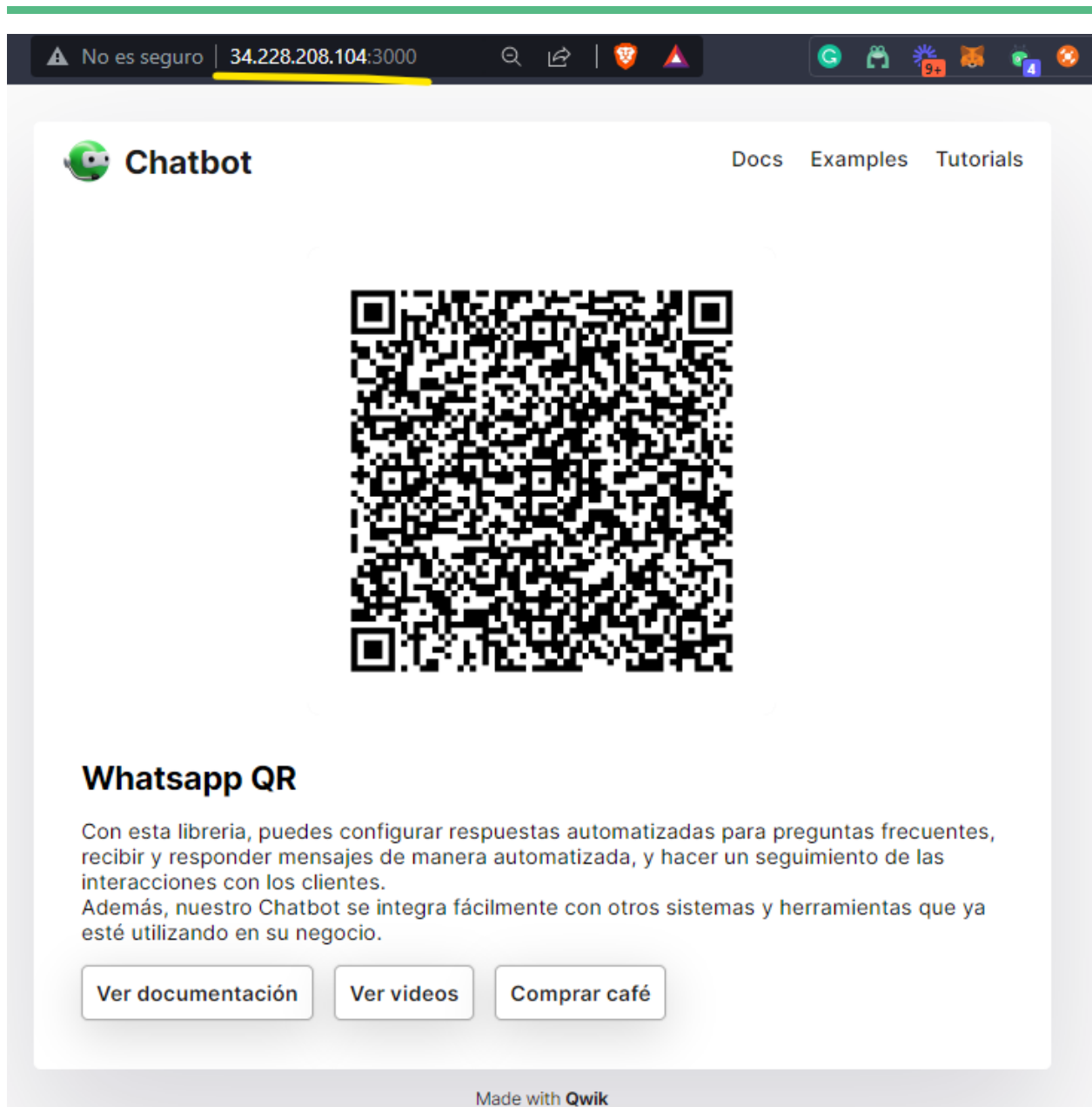
Create

☒ Duplicate rule for IPv6

Application	Protocol	Port or range / Code	Restricted to	
SSH	TCP	22	Any IPv4 address Lightsail browser SSH/RDP ?	<div><div></div><div></div></div>
HTTP	TCP	80	Any IPv4 address	<div><div></div><div></div></div>

## Escanear QR





---

## Ejecutar en Producción

Debes ubicarte en el directorio donde tienes el código fuente de tu chatbot.

Independientemente de tu sistema operativo deberás ejecutar el chatbot con el comando a través de un sistema que mantenga el proceso en ejecución. Recomendamos [Pm2](#)

```
pm2 start app.js --name=bot1
```

id	name	mode		status	cpu	memory
0	bot1	fork	0	online	0%	32.1mb

La consola de devolver un mensaje con una lista de procesos, en el ejemplo puedes observar, que tenemos un proceso llamado **bot1**

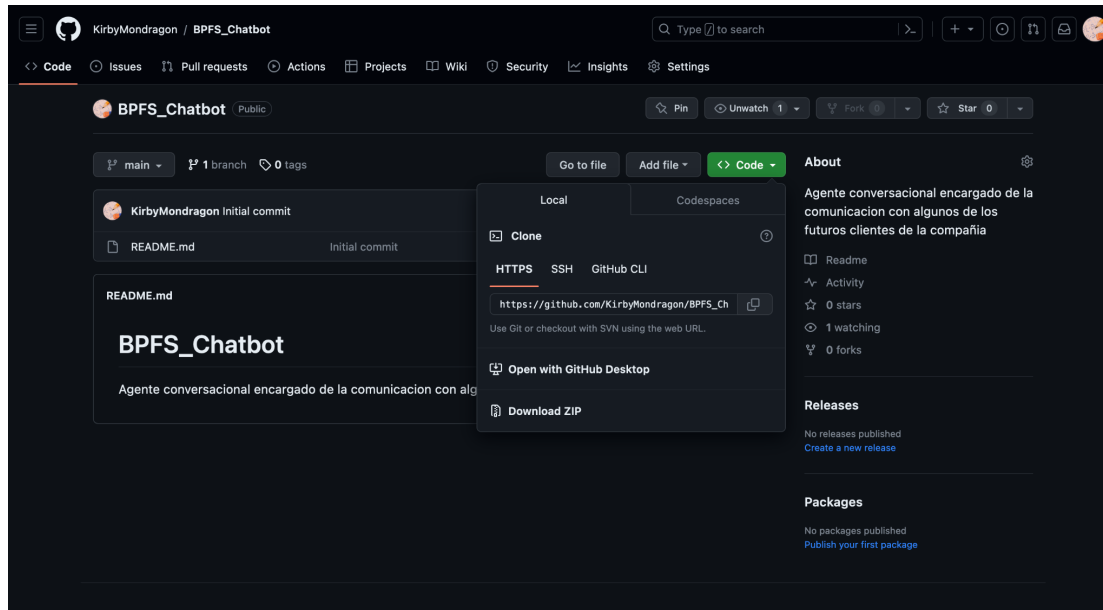
De esta manera ya puedes cerrar la terminal y tu bot seguirá en ejecución sin problema

## Entrar a github

Desde el siguiente link : [https://github.com/KirbyMondragon/BPFS\\_Chatbot](https://github.com/KirbyMondragon/BPFS_Chatbot)

The screenshot shows the GitHub repository page for **KirbyMondragon / BPFS\_Chatbot**. The repository is public and has 1 commit, 0 stars, 1 watching, and 0 forks. The README.md file is displayed, showing the project title **BPFS\_Chatbot** and a description: "Agente conversacional encargado de la comunicacion con algunos de los futuros clientes de la compañía". The right sidebar contains sections for "About", "Releases", and "Packages", all of which are currently empty or show no published content.

## Dar click sobre el botón Code y copia el HTTPS



## Instalarlo en una máquina de ubuntu

**Asegúrate de estar en la rama correcta:** Antes de actualizar, asegúrate de estar en la rama que deseas actualizar. Puedes verificar esto utilizando el comando `git branch`.

**Agrega el repositorio remoto original como referencia:** Si aún no has hecho esto, agrega el repositorio original como un remoto en tu repositorio local. Puedes hacerlo con el siguiente comando:

```
bash
```

```
git remote add upstream <URL_del_repositorio_original>
```

Sustituye `<URL_del_repositorio_original>` por la URL del repositorio original en GitHub.

**Obtén los cambios del repositorio original:** Para obtener los cambios más recientes del repositorio original, ejecuta el siguiente comando:

```
bash
```

```
git fetch upstream
```

---

**Actualiza tu rama local con los cambios remotos:** Una vez que hayas obtenido los cambios remotos, puedes combinar esos cambios con tu rama local. Por ejemplo, si estás en la rama principal (main o master):

bash

```
git checkout main # 0 "git checkout master" según la rama que estés usando
```

```
git merge upstream/main # 0 "git merge upstream/master" según la rama original
```

Esto fusionará los cambios del repositorio original en tu rama local.

**Resuelve conflictos si es necesario:** Si hay conflictos entre tus cambios locales y los cambios remotos, Git te indicará dónde se encuentran. Deberás resolver estos conflictos manualmente y luego hacer un commit para confirmar los cambios.

**Sube los cambios a tu repositorio en GitHub:** Una vez que hayas actualizado tu repositorio local, puedes subir los cambios a tu repositorio en GitHub con los siguientes comandos:

bash

```
git push origin main # 0 la rama que estés usando
```

¡Y eso es todo! Ahora tu repositorio local estará sincronizado con los cambios más recientes del repositorio original en GitHub. Recuerda que este proceso puede variar un poco dependiendo de tu flujo de trabajo y la configuración específica del repositorio

## Ejecución en segundo plano en una máquina virtual Ubuntu

El programa requiere el comando `npm start` para iniciarse, puedes seguir los pasos anteriores con algunas modificaciones para asegurarte de que el programa se ejecute correctamente en segundo plano.

1. Abre una terminal SSH en tu VPS con Ubuntu.
2. Navega al directorio donde se encuentra el programa `base-baileys-memory`.

Poner en la consola:

```
cd /ruta/del/programa
```

- 
3. Ejecuta el comando `nohup npm start` con `&` al final para desvincularlo de la terminal y ejecutarlo en segundo plano:

Poner en la consola:

```
nohup npm start &
```

4. La salida del programa se redirigirá al archivo `nohup.out` en el mismo directorio. Puedes verificar la salida del programa en cualquier momento mediante el comando:

Poner en la consola:

```
tail -f nohup.out
```

5. Puedes cerrar la ventana de la terminal sin detener el programa. El comando `npm start` se ejecutará en segundo plano en el VPS.
6. Si deseas verificar el estado del programa en algún momento posterior, puedes usar el comando `ps` para listar los procesos en ejecución y buscar el proceso correspondiente a `npm`:

Poner en la consola:

```
ps aux | grep npm
```

Igualmente, ten en cuenta que el proceso `grep` también aparecerá en los resultados, pero eso es normal. El proceso real del programa debería tener un PID diferente.

Recuerda que si en algún momento deseas detener el programa, puedes utilizar el comando `kill` con el PID del proceso relacionado con `npm`.

---

### Configuración de una VPN en Ubuntu 20.04 (Opcional)

Configurar una VPN en Ubuntu 20.04 puede hacerse utilizando el protocolo OpenVPN, que es una opción segura y ampliamente utilizada. A continuación, te guiaré a través de los pasos básicos para configurar un servidor OpenVPN en tu VPS con Ubuntu 20.04:

**Nota:** Antes de comenzar, asegúrate de tener acceso a tu servidor VPS mediante SSH y de tener privilegios de superusuario (sudo).

#### 1. Instala OpenVPN:

```
bash
```

```
sudo apt update
```

```
sudo apt install openvpn easy-rsa
```

#### Configura la Autoridad de Certificación (CA) y Genera Claves:

Crea un directorio para tus certificados:

```
bash
```

```
make-cadir ~/openvpn-ca
```

---

Sigue las instrucciones interactivas para generar las claves de la Autoridad de Certificación (CA) y las claves del servidor:

```
bash
```

```
cd ~/openvpn-ca
```

```
source vars
```

```
./clean-all
```

```
./build-ca
```

```
./build-key-server server
```

#### **Genera Claves para Clientes:**

```
bash
```

```
./build-key client1    # Cambia "client1" por el nombre del cliente
```

#### **Genera el Archivo de Configuración del Servidor:**

```
bash
```

```
cd ~/openvpn-ca
```

```
./build-dh
```

```
openvpn --genkey --secret keys/ta.key
```

#### **Configura el Servidor OpenVPN:**

Copia los archivos de configuración a la ubicación de OpenVPN:

```
bash
```

```
sudo cp
```

```
~/openvpn-ca/keys/{ca.crt,server.crt,server.key,dh2048.pem,ta.key}
```

```
/etc/openvpn
```

---

Crea un archivo de configuración para el servidor:

```
bash
```

```
sudo nano /etc/openvpn/server.conf
```

Agrega el siguiente contenido (modifica las direcciones IP según sea necesario):

```
plaintext
```

```
port 1194
```

```
proto udp
```

```
dev tun
```

```
ca ca.crt
```

```
cert server.crt
```

```
key server.key
```

```
dh dh2048.pem
```

```
tls-auth ta.key 0
```

```
topology subnet
```

```
server 10.8.0.0 255.255.255.0
```

```
push "redirect-gateway def1 bypass-dhcp"
```

```
push "dhcp-option DNS 208.67.222.222"
```

```
push "dhcp-option DNS 208.67.220.220"
```

```
user nobody
```

```
group nogroup
```

```
keepalive 10 120
```



---

```
persist-key
```

```
persist-tun
```

```
status /var/log/openvpn/status.log
```

```
verb 3
```

Guarda y cierra el archivo.

### **Habilita el IP Forwarding:**

Edita el archivo `/etc/sysctl.conf`:

```
bash
```

```
sudo nano /etc/sysctl.conf
```

Descomenta la línea que dice `net.ipv4.ip_forward=1`.

Luego, activa los cambios:

```
bash
```

```
sudo sysctl -p
```

### **Configura el Firewall:**

Si estás utilizando UFW, habilita el tráfico UDP en el puerto 1194:

```
bash
```

```
sudo ufw allow 1194/udp
```

Asegúrate de que el tráfico se enrutará correctamente:

```
bash
```

```
sudo nano /etc/ufw/before.rules
```

Agrega las siguientes líneas al principio del archivo:

---

plaintext

```
# NAT table rules

*nat

:POSTROUTING ACCEPT [0:0]

# Allow traffic from OpenVPN client to eth0

-A POSTROUTING -s 10.8.0.0/8 -o eth0 -j MASQUERADE

COMMIT
```

Guarda y cierra el archivo.

### Inicia y Habilita OpenVPN:

bash

```
sudo systemctl start openvpn@server

sudo systemctl enable openvpn@server
```

- 1.
2. **Configura los Clientes:**  
Copia los archivos de configuración de los clientes (certificados y claves) a sus máquinas locales.
3. **Conecta los Clientes a la VPN:**

Utiliza el cliente OpenVPN (puede ser **openvpn** en la línea de comandos o una interfaz gráfica) para conectar los clientes a la VPN utilizando los archivos de configuración generados.

Estos son los pasos básicos para configurar un servidor OpenVPN en Ubuntu 20.04. Ten en cuenta que esta es una guía simplificada y que la configuración puede variar según tus necesidades específicas y el entorno de red. Te recomiendo consultar la documentación oficial de OpenVPN y realizar ajustes de seguridad adicionales según sea necesario.