UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

INTEGRACIÓN DE SOLUCIÓN DE CHATBOX PARA LA AUTOMATIZACIÓN DE LA COMUNICACIÓN CLIENTE-COMERCIO SOBRE FRAMEWORK DE DESARROLLO MOQUI

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

VICENTE ANDRÉS ROJAS CLERC

PROFESOR GUÍA:
ANDRÉS MUÑOZ ÓRDENES

MIEMBROS DE LA COMISIÓN:
FELIPE BRAVO MARQUEZ
CESAR GUERRERO SALDIVIA

SANTIAGO DE CHILE
2022

Resumen

En este informe se presenta el proyecto de integración de módulo de Chatbox dentro del *framework* Moqui, con el que se opta por al título de ingeniero civil en computación.

Para la realización de este proyecto se contó con el apoyo del equipo de Moit, principales desarrolladores de soluciones para empresas con el framework Moqui. Moit busca ampliar sus servicios a través de automatización de comunicación con sus clientes, para lo cual buscan añadir un módulo de Chatbox a Moqui. Un Chatbox consiste en un programa que captura preguntas de usuario y responde automáticamente, a través de aprendizaje de maquina con distintas formas de entrenamiento.

El procedimiento utilizado consistió en una fase de investigación, una fase de selección y finalmente una fase de desarrollo. Durante la fase de investigación hubo levantamiento de requisitos junto al equipo de Moit, se aprendió sobre Moqui y sus propiedades para facilitar la implementación posterior del módulo, y además para poder realizar una selección apropiada de alternativa de Chatbox. Se buscaron distintos servicios de Chatbox en esta fase para poder encontrar el más indicado para este trabajo, en base a los requisitos previamente acordados y los criterios que estos debían cumplir.

Luego se tiene una fase de selección, ya contando con conocimiento sobre Moqui y alternativas factibles a priori para este trabajo. Tras evaluar en detalle las encontradas, se llegó a elegir Rasa. Rasa consiste en un *framework* para trabajar con NLU (Natural Language Understanding), cuenta todo lo necesario para crear un Chatbox desde cero, personalizando las historias de uso y las consultas que este puede responder.

A gran escala, el diseño de la solución tiene al usuario interactuando directamente con Rasa Server a través del Chatbox, luego Rasa a través de su servidor de acciones personalizadas, el cual utiliza python con todas sus herramientas, manda solicitudes API a Moqui para obtener la información necesaria que resuelve la consulta del usuario.

Por lo tanto, la implementación se centró en, por un lado, desarrollar el motor del Chatbox sobre de Rasa, creando las acciones personalizadas, historias de uso, y data NLU entre otras cosas para automatizar la interacción, y por otro lado en desarrollar los servicios de Moqui para habilitar las solicitudes hacia la base de datos, la cual además fue modificada para crear el tipo de entidad correspondiente para los mensajes de Chatbox, usuarios de Chatbox, Tickets, y más.

Agradecimientos

Quiero agradecer en primer lugar al equipo de Moit, en particular a mi profesor guía Andrés Muñoz, por toda la ayuda, paciencia, grato ambiente de trabajo, y buena disposición por parte de todos.

También a mi familia, por siempre apoyarme e impulsarme a aprender más y llegar lo más lejos posible.

Y finalmente a todos mis amigos y amigas quienes fueron mi principal apoyo todos estos años de carrera, gracias por todo.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Antecedentes Generales

Moit (www.moit.cl) es una compañía que busca dar soporte a diferentes tipos de procesos de gestión empresarial, por ejemplo, procesos de comercio electronico o E-commerce. Provee soporte tecnológico a través del *Framework* Moqui para brindar soluciones de ERP (*Entrerprise Resource Management*), WMS (*Warehouse Management System*), CRM (*Customer Relationship Management*), u otros sistemas. Actualmente cuenta con servicios tales como Moit ERP, que integra y maneja los procesos de negocios asociados con las operaciones de producción y distribución [8]; Moit Cowork, que brinda una solución completa para la gestión de *Cowork*, incluyendo el control de acceso a las instalaciones y el uso de los recursos del mismo [8], y otras soluciones de tecnología.

Al momento de realizar este trabajo de memoria, Moit cuenta con un equipo de trabajo integrado por diez personas, de las cuales ocho son ingenieros civiles en computación.

Moqui (https://moqui.org/framework.html) es un *framework* de desarrollo basado en Java y cuenta con distintos módulos de apoyo para múltiples servicios. Cada uno de estos módulos busca brindar diferentes funcionalidades para Moqui que son la base para cualquier solución que se quiera entregar a una empresa. Los módulos proveen funcionalidades específicas, y cada uno de éstos está conectado con los demás, de tal forma que, si se implementa un servicio, este es accesible a los demás módulos para poder ser utilizados también.

Moqui destaca por construir aplicaciones de automatización de procesos en diferentes áreas. Este *framework* brinda herramientas para interacciones con bases de datos, lógica en servicios locales y web, seguridad, y más. En particular, permite el manejo fácil y cómodo de órdenes de compra, clientes, despachos, devoluciones, y otras operaciones comerciales, para una gestión más eficiente sobre éstas.

Moqui cuenta con distintos desarrolladores en todo el mundo, entre los que destaca la compañía Moit para Chile. La empresa utiliza este framework como la herramienta principal para todas sus soluciones, trabaja expandiéndolas a través de distintos módulos, incorporando nuevas funcionalidades como también mejorando las funcionalidades existentes.

Moit ha desarrollado sus propios módulos y servicios en Moqui para responder a las solicitudes de sus clientes dentro de un mismo framework, lo que le ha permitido hacer crecer su oferta de servicios y convertirse en un proveedor más integral. Algunos de sus desarrollos son implementaciones open source, por lo que no solo están disponibles para los clientes de Moqui en Chile, sino también a nivel global. En particular han realizado desarrollos regionales, tales como la traducción de toda la plataforma a español, soluciones para gestión de documentos tributarios con el SII, interoperabilidad con sistemas de Previred, y más.

Dentro de sus soluciones para E-Commerce, la empresa Moit buscaba generar una solución de Chatbox que estuviera disponible para cualquier desarrollador o usuario de Moqui y, por lo tanto, fuera flexible para cualquier servicio y compatible con el resto de sus módulos.

Para esto, Moit presentó la integración de un módulo de *Chatbox* como tema de memoria para alumnos de la Universidad de Chile, la que ha sido elegida para ser presentada en este documento.

Un *Chatbox* es una solución que es considerada como estándar para una solución web para un comercio. Un *Chatbox* consiste en un cuadro de diálogo donde un cliente puede interactuar para diferentes fines, como consultar por stock de productos, pedir reembolsos, buscar productos, entre otros. Se denomina *Chatbox* a la interfaz presente en la plataforma web o móvil y *Chatbot* a lo relacionado con la lógica de automatización de los mensajes presentes en el *Chatbox*, procesando estos a través de NLU incorporado en *back-end*. Este sistema de respuesta usualmente requiere de un proceso de entrenamiento o configuración del Chatbot, que se obtiene de la creación de historias de uso y especificaciones sobre cómo responder a mensajes tipo. También es posible su desarrollo a partir de algoritmos propios de cada programa de Chatbot, utilizando datos de prueba para que el sistema conteste adecuadamente.

Para simplificar la comprensión de este documento, se utilizará únicamente el concepto "Chatbox" a partir de este punto, que se referirá a la integración entre los servicios de Chatbox y Chatbot.

La principal ventaja de un *Chatbox* para este contexto es automatizar la interacción cliente-servicio, de manera que se puedan brindar soluciones para problemas estándares sin necesidad de interactuar directamente con un empleado. Si no se trata de un problema estándar, es posible además brindar un servicio de tickets que permita capturar la información del problema a abordar y que sea entregado a un *back-end*, para luego ser resuelto por un empleado.

1.2. Objetivos

1.2.1. Objetivo general

El objetivo de esta memoria es agregar un medio de comunicación de mensajería automatizada, directa e inmediata para clientes de un comercio electrónico, utilizando el *framework* Moqui e integrando soluciones existentes de *Chatbox*.

1.2.2. Objetivos específicos

1. Realizar un levantamiento de requisitos funcionales para un *Chatbox* en una solución de *E-commerce*.
2. Investigar alternativas disponibles de *Chatbox* y *Chatbots* existentes en el mercado.
3. Clasificar y caracterizar las soluciones de acuerdo con sus funcionalidades, capacidad de personalización y de integración con otros sistemas.
4. Conocer las características de uso y requerimientos técnicos para la integración de Moqui.
5. Filtrar las alternativas de solución investigadas para *Chatbox* con el objeto de determinar aquellas que son compatibles técnicamente con el *framework* Moqui.
6. Definir la mejor solución de *Chatbox* que cumpla de mejor manera con los criterios de evaluación previamente definidos.
7. Implementar en Moqui un componente para poder integrar el Chatbox.
8. Realizar una evaluación de la solución implementada utilizando herramientas tecnológicas y metodologías funcionales para determinar el cumplimiento de lo requisitos levantados y determinar los trabajos futuros que se podrían realizar para mejorar esta solución.

Capítulo 2

Preparación

En este capítulo se abordarán las definiciones iniciales de requisitos para esta memoria, la investigación respecto a las alternativas de *Chatbot* y las características del *framework* Moqui.

2.1. Definición de requerimientos

Para poder abordar apropiadamente el problema, en primera instancia se definió junto al equipo de Moit los requisitos iniciales que el módulo de Chatbox debía cumplir (Anexo G).

El primer requerimiento establecido fue con respecto a la adaptabilidad del Chtabox, que debía ser personalizable. Es decir, se debía configurarse la manera en que el Chatbox respondía frente a las consultas, definiendo tanto el input que recibe como el output que entrega. Además, debe tener manejo de contexto, para poder identificar posibles historias de uso. Esto además debe ser modificable por el equipo de Moit posterior al trabajo realizado, de tal forma que se debe documentar apropiadamente cómo se realizan estas modificaciones.

Otra funcionalidad crucial para Moit es llevar un registro sobre las conversaciones realizadas en el Chatbox, es importante que estas queden almacenadas en la base de datos de Moqui. Con esto se puede llevar registro de qué consultas se hacen más frecuentemente, y cuánto se ha utilizado el Chatbox en sí.

Respecto a las consultas a implementar, se abordarán dos casos que buscan evaluar el potencial del Chatbox: la solicitud de órdenes de compras, y de información de cliente. Estas dos solicitudes involucran utilizar información de usuarios para obtener información desde la base de datos de Moqui. Con ésto además queda en claro que el Chatbox debe ser capaz de comunicarse con Moqui.

Por otro lado, está la creación de Tickets. Un Ticket almacena la información que el usuario quiere solicitar, junto a los datos de dicho usuario. Para ello el Chatbox debe poder reconocer escenarios en que se deba crear un Ticket y almacenarlos en la base de datos de Moqui, para que desde *back-end* resuelvan la consulta al cliente.

Además, como se debe consultar información a cliente tales como e-mail, IDs de órdenes de compra y otros, el Chatbox debe ser capaz de identificar esta información, almacenarla, y realizar las consultas pertinentes con esta información.

Por último, se busca obtener soporte en línea a través de un agente, permitiendo al usuario tener soporte con un humano.

2.2. Investigación alternativas Chatbox

Tras tener una idea inicial de requerimientos, se buscaron alternativas para Chatbox a utilizar. Para esto, simplemente se buscaron ejemplos de Chatbox integrados en páginas webs, y también se buscó a través de Google.

Para cada alternativa elegida, se identificó sus características principales, comparando las opciones encontradas entre sí a través de distintos factores que se pueden categorizar como excluyentes u opcionales. Estas fueron:

- Historias de uso: La alternativa de Chatbox cuenta con la posibilidad de configurar historias de uso, que involucren reconocer una secuencia de mensajes de usuario, especificando que responder frente a cada uno. Esta característica es opcional, pues en caso de presentar una alternativa factible para ésto, se puede descartar esta funcionalidad.
- Soporte en linea: El Chatbox puede proveer servicio de comunicación en vivo con una persona. Esta caracterísitca es excluyente, pues se requiere que el Chatbox pueda cumplir con esta funcionalidad.
- Gratuito: No se requiere pagar para utilizar este Chatbox. Esto es opcional, pues en caso de ser necesario, Moit puede proveer financiamiento para el Chatbox.
- Tickets: El Chatbox permite crear tickets para ser guardados como consultas por clientes. Esto es excluyente, pues es la base de servicio a cliente por parte de Moit.
- 24/7: El Chatbox cuenta con disponibilidad de servicio 24/7 para responder consultas. Esto es opcional, pues en caso de contar con disponibilidad en horario laboral podría resultar suficiente.
- Lenguaje implementación: Se especifica en qué lenguaje de programación está implementado el Chatbox.
- Manejo de contexto: El Chatbox logra tener una idea de contexto para poder tener distintas respuestas frente a un mismo input, dependiendo de qué se había consultado previamente. Esto es excluyente, pues la principal ventaja de utilizar un Chatbox es poder manejar conversaciones reales con usuarios, para lo que se requiere manejo de contexto.
- Personalizable: Se puede cambiar el color, el tipo de letra, etc de este Chatbox. Esto es excluyente, pues se debe poder obtener un diseño apropiado para la solución brindada con Moqui.

- Documentación: Se especifica la calidad de documentación de esta alternativa, relativa a las demás.
- Compatibilidad PostgreSQL: Esta alternativa es compatible con el sistema de almacenamiento de bases de datos PostgreSQL. Esto es excluyente, pues en caso de no poder comunicarse con la base de datos de Moqui, el Chatbox no sería factible como solución.
- Multilenguaje: El Chatbox logra reconocer y trabajar con distintos lenguajes, tales como español, inglés, y otros. Esto es excluyente, pues se busca que esta solución pueda eventualmente ser utilizada a nivel global.
- Recolección estadísticas: Chatbox entrega distintas estadísticas sobre las conversaciones realizadas en este. Esto es opcional, mientras se permita manejar con *back-end* los inputs de usuario para recolección de data.

En total se encontraron 24 soluciones, de las cuales solo se analizaron diez, ya que el resto no contaba con alguna funcionalidad clave o por no contar con descarga gratuita, pues inicialmente se evaluó si sería suficiente contar con opciones *open source* para satisfacer los requerimientos de este trabajo.

2.3. Estudio Moqui Framework

Para poder realizar un trabajo apropiado dentro de Moqui, fue necesario investigar y familiarizarse con el funcionamiento del Framework. Moqui está construido en Java, sin embargo utiliza un meta-lenguaje basado en XML. Este es interpretado en tiempo de ejecución, siendo transformado a código Java. Además, dentro de Moqui se puede utilizar *scripts* en Groovy, el cual es un dialecto de Java no estructurado.

La estructura de un proyecto en Moqui se compone de las bibliotecas base para la ejecución del framework (moqui-framework y moqui-runtime), además de un conjunto de módulos que se cargan al ejecutar la plataforma. Estos módulos residen en directorios dentro del proyecto (runtime/components) y pueden variar dependiendo del tipo de solución que se desee construir.

Dentro de cada módulo se cuentan con directorios que cumplen roles específicos:

- data: En este directorio se realiza la carga de datos iniciales, y los datos de prueba. Con ésto se tiene los datos iniciales necesarios para levantar el módulo y para testear funcionamiento básico. El orden de ejecución de los archivos contenidos en esta carpeta es por orden alfabético, por lo que los archivos de carga de datos iniciales deben estar antes alfabéticamente que los archivos que se encargan de datos de prueba.
- entity: Contiene la definición de las entidades propias del módulo, estas se cargan durante el proceso de ejecución de Moqui.
- screen: Este directorio involucra todo lo relacionado con *front-end*.

- service: Involucra todo lo relacionado con *back-end*. Cada servicio utilizado aquí puede ser utilizado por módulos externos. En particular, aquí se pueden definir servicios API para permitir conexión con software externos.
- src: Contiene la implementación de las pruebas unitarias en Groovy y pruebas *end-to-end* en NodeJS.
- template: Este directorio puede contener definiciones estándares que se pueden utilizar en el *front-end*.

Se estudiaron distintos ejemplos de programas tipo servicios REST, entendiendo sus estructuras generales: siempre cuentan con un encabezado donde debe contar con un nombre descriptivo de lo que este servicio hace. Para ello se especifica un verbo y un sustantivo para nombrar el servicio. Un ejemplo es el siguiente:

```
<service verb="post" noun="ChatboxMessage"
authenticate="anonymous-all" allow-remote="true"
transaction-timeout="300000">
```

Como se puede ver, XML usa tags de manera similar a HTML, y aquí se indican el verbo, "post", y el sustantivo "ChatboxMessage", entendiendo que este servicio permite hacer una solicitud POST de un mensaje de Chatbox. Además, se especifican parámetros extras tales como la autenticación requerida, si se permite su uso en módulos remotos, y el tiempo de espera de ejecución del servicio antes de cancelarlo.

Posterior al encabezado, se debe especificar una breve descripción sobre lo que este servicio realiza. Luego, se debe indicar tanto los parámetros de entrada como los de salida. Como los nombres indican, los parámetros de entrada son valores que el servicio recibe y utiliza en sus acciones, y los parámetros de salida son los valores que el servicio retorna al ejecutarse.

Y finalmente se definen las acciones a realizar por el servicio. Aquí entra la programación tradicional con comandos tales como if, set para definir valores de variables, etc. Fue necesario revisar múltiples ejemplos de servicios en Moqui para tener un mejor entendimiento de la sintaxis de XML y como crear un servicio, para luego poder crear servicios propios en la fase de implementación.

Posteriormente, al ver que sería necesario tener servicios API para conectar el Chatbox a Moqui, se estudió más en detalle cómo se configuran estos servicios. Se explicará más sobre esto en la fase de implementación.

2.4. Selección final de alternativa de Chatbox

Una vez que se cuenta con un mejor manejo sobre Moqui y lo que se requeriría para un Chatbox, se comenzó a descartar las opciones previamente consideradas factibles. El problema más recurrente encontrado era que las opciones, en general, no eran *open source*, o no permitían interactuar con los mensajes obtenidos por usuarios (sea utilizar un mensaje

para luego realizar consultas, o almacenar los mensajes encontrados, o acceder a mensajes previos en el chat).

El proceso de descarte de alternativas consistió en proceder a descargar cada alternativa, y con eso tener un mejor entendimiento de sus propiedades únicas frente a los otros Chatbox. Durante este proceso se pudo ver que dos alternativas destacaban sobre el resto, pues las otras ocho presentaban problemas para cumplir con los requerimientos planteados con el equipo de Moit. Estas dos alternativas fueron Rasa y Botpress.

	Rasa	Tidio	Cliengo	User	Botsify	botpress	Gat-Blac	UNIQ	BindBot	databot
Historias de uso	Si	Si	Si	Si	Si	Si	-	-		
Soporte en linea	Si					Si	-	-		
Gratuito	open source	si pero limitad	Si pero limitada	si pero limitada	No	parcialmente, tiene	-	-		
Tickets							-	-		
Disponibilidad 24/7	Si	Si	N/A	N/A	N/A	Si	-	-	Si	
lenguaje implementación	python	flujo de acción	no code	flujo de acción	no code	no code	-	-		
manejo contexto	Si	Si	Si	Si	Si	Si	-	-		
personalizable	Si	Si	N/A	N/A	N/A	si	-	-		
Documentación	Buena (página +	decente (requ	Buena (blogs +	Buena (docs)	Mala (sin docs)	Buena	-	-		
Compatibilidad PostgreSQL	Si	Si	Si	No	No	Si	-	-		
Multilingual	Si	Si	Si	Si	Si	Si	-	-		
Recolección Estadisticas	Si	N/A	N/A	N/A	N/A	Pagado				
Observaciones	Levantar servidor	Versión gratui	Versión gratuita	-			Descartado	Descartado	Descartado	Descartado

Figura 2.1: Matriz de comparación para alternativas Chatbox.

Ambos parecían cumplir todos los requerimientos definidos, o al menos tenían el potencial para ello. Por lo tanto, para poder seleccionar una, se compararon entre sí viendo su usabilidad, flexibilidad y otros posibles factores que surjan tras utilizar en mayor detalle las dos opciones.

Al momento de descargar e instalar ambas alternativas, se pudo ver claramente que la usabilidad de Botpress era bastante inferior a Rasa. Una interfaz compleja con muchas opciones y documentación que dejaba bastante que desear. En cambio, Rasa tenía un *framework* amigable para un programador, excelente documentación, e incluso un canal de youtube con tutoriales para trabajar. Considerando que este trabajo está pensado para que alguien más lo continúe posterior a que el memorista finalice su trabajo, y considerando que Rasa contaba con un potencial y flexibilidad a la hora de implementar nuevas funcionalidades superior a Botpress, se optó por elegir Rasa como la alternativa de Chatbox a utilizar.

Capítulo 3

Diseño

En este capítulo se especificará el diseño realizado para la implementación del módulo de Chatbox. Se abordará sobre las funcionalidades esperadas para el servidor de Rasa y para Moqui.

3.1. Arquitectura

Tras tener un mejor entendimiento sobre Moqui y Rasa, se fue experimentando sobre como funcionaría la comunicación entre estos para poder brindar la solución esperada. Dado que Rasa cuenta con *custom actions* que permiten realizar consultas API, el servidor de acciones de Rasa puede conectarse de manera directa a Moqui para poder obtener objetos desde la base de datos de Moqui, y con ello otorgar las respuestas a las consultas realizadas por clientes.

Con esto, los actores relevantes para llevar a cabo el flujo de información sería el usuario, quien ingresa su consulta al Chatbox, Rasa, quien a través del servidor Rasa NLU maneja la solicitud e invoca a la acción correspondiente en Rasa Actions Server, y Moqui, quien habilita endpoints para consultas API para que Rasa Actions Server pueda obtener la información pertinente para la solicitud hecha inicialmente. Además, se cuenta con un bloque de código de *javascript* otorgado por Rasa, con el cual Rasa se encarga del front-end del Chatbox.

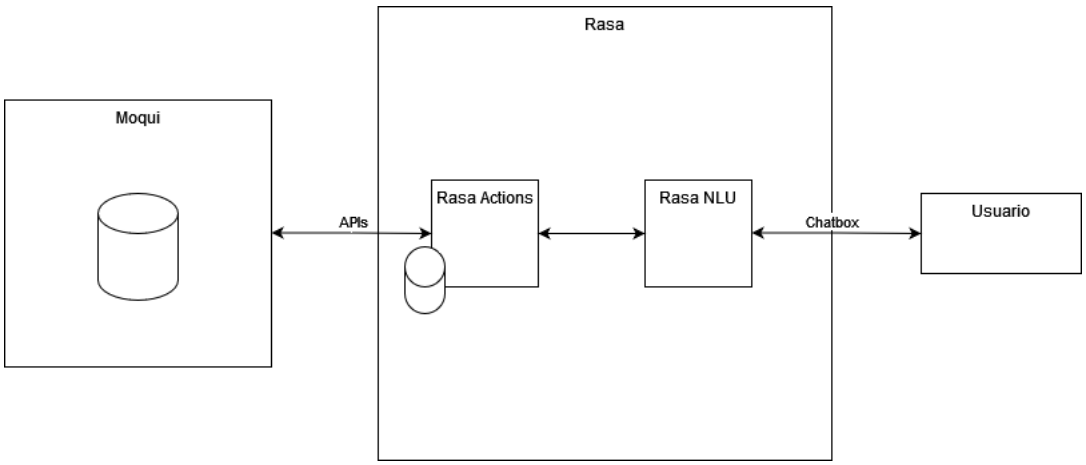

Figura 3.1: Diseño de Arquitectura de la solución.

3.2. Modificaciones a base de datos de Moqui

Con tal de integrar Rasa con Moqui, se analizó como abordar el modelamiento de las entidades involucradas en este trabajo. Afortunadamente, Moqui ya cuenta con un modelo de datos existente, por lo que el trabajo se realizó sobre esta mismo.

Para realizar el trabajo hubo dos entidades principales a utilizar: *workEfforts* y *communicationEvent*. Por un lado, *workEfforts* se utiliza como una entidad general dentro de Moqui, con la cual se almacena una variedad de tipos de datos para distintos propósitos. Por lo tanto, se sugirió utilizar esta entidad para la creación de *Tickets*.

Por otro lado, *communicationEvent*, como su nombre indica, almacena eventos de comunicación. Por ello fue natural utilizarlo para almacenamiento de los mensajes ingresados en el Chatbox.

Capítulo 4

Implementacion

4.1. Instalación inicial Moqui Framework

Para instalar Moqui se debía contar como prerequisito con JDK 11, IntelliJ IDEA y un cliente GIT, para poder utilizar GitLab. Todo lo anterior se instaló en una máquina virtual con Linux, para facilitar el trabajo de desarrollo.

Tras obtener los requisitos, se participó de una sesión de aprendizaje con el equipo de Moit, en donde se instruyó sobre el proceso de instalación: tras obtener una cuenta en Gitlab, se debe configurar una llave SSH para poder realizar la descarga, de tal forma que se tenga un sistema de autenticación a la hora de realizar descargas posteriores.

Para proceder con la instalación, primero se debe instanciar el repositorio Moqui-Framework, descargándolo a través de Git. En el repositorio de Gitlab se cuentan con distintas ramas, se trabaja sobre moit-devel, la cual tiene la última versión en desarrollo de Moqui en Moit. Luego se deben descargar los componentes de Moqui utilizando Gradlew, un sistema de automatización de construcción de código, con este se pueden descargar automáticamente los componentes de Moqui necesarios para el desarrollo. Cada uno de estos componentes son repositorios independientes. En particular en este proceso se obtiene el directorio runtime, sobre el cual se instalan el resto de los componentes.

Teniendo lo anterior, se procede a realizar el build, nuevamente utilizando Gradlew: en primera instancia se utiliza gradlew build, para construir todo lo necesario para obtener un archivo war, con el cual posteriormente se ejecuta Moqui. Luego con ***gradlew load*** se pueden cargar datos iniciales, considerando que a la hora de cargar datos estos van acompañados del tipo de semilla a cargar, y los datos iniciales funcionan como la base para poblar el modelo de datos. Finalmente, con ***gradlew run*** se levanta Moqui.

Cabe notar que Moit utiliza PostgreSQL para sus ambientes de aseguramiento de calidad (QA) y producción, sin embargo para ambiente de desarrollo, en caso de no utilizar un gestor de base de datos, Moqui crea una base de datos embedida, llamada H2.

4.2. Servidor de Rasa

Rasa consiste en un *framework* para trabajar con NLU (*Natural Language Understanding*). Brinda un servicio de Chatbot que permite manejo de contexto, pudiendo entender que se requiere distintas respuestas frente a una misma solicitud de usuario dependiendo del contexto. Previo a ésto, el estándar para Chatbox era determinísticamente contar con una respuesta frente a un input.

4.2.1. Implementación Rasa Server

Para utilizar Rasa, se instaló en la misma máquina virtual utilizada para el manejo de Moqui Framework. Cuenta con un *framework* que especifica dónde ingresar cada instrucción para el Chatbox. Rasa cuenta con *stories*, las cuales son historias de uso, en las cuales se especifican *intents* y *actions.* Rasa utiliza archivos YML para todo su *framework*, salvo las acciones personalizadas.

A continuación, se detallan las principales herramientas de Rasa:

Intents

Para realizar el trabajo con NLU, Rasa utiliza *NLU Data* para clasificar input de usuario dentro de *intents*. Los *intents* buscan descifrar que intención tiene el usuario al realizar una consulta al chatbox. Desde cosas básicas como decir "hola", que se puede clasificar dentro de un *intent* de "greeting", hasta "quiero revisar mi orden de compra", lo que se puede clasificar como una solicitud de orden de compra.

Para declarar un *intent*, se cuenta con el archivo ***nlu.yml***. Aquí se especifica el nombre de éste, y la NLU Data asociada a dicho *intent* para entrenar al Chatbox, con tal de que tenga ejemplos de posibles *inputs* para cada intención de usuario. Esto se puede apreciar en el siguiente ejemplo:

```
- intent: request_order
  examples: |
    - quiero revisar mi orden de compra
    - dame mi orden de compra
    - necesito ver mi orden de compra
    - dame el estado de mi orden de compra
    - requiero ver mi orden de compra
    - ordenes de compra
```

Al momento de entrenar el Chatbox, Rasa utiliza los ejemplos brindados para aprender qué tipo de mensajes entran en esta clasificación.

Luego, al momento de recibir el input, Rasa asocia una probabilidad de pertenencia a cada *intent* para lo ingresado por el usuario, y luego el *intent* con mayor probabilidad de

acierto es al cual se asocia la solicitud ingresada.

Stories

Rasa destaca por poder manejar contexto dentro de las conversaciones realizadas en el Chatbox, es la principal característica que le hace resaltar por sobre previas opciones de automatización de mensaje. Una forma de obtener manejo de contexto son las *Stories*. Una *Story* consiste en una secuencia de *intents* y *actions*. Ésta es la principal forma de saber que hacer frente a distintos *intents*.

Ejemplo:

```
- story: order
  steps:
  - intent: greet
  - action: welcome
  - intent: request_order
  - action: check_order
  - intent: give_order_id
  - action: order_set_id
```

Esta es la historia de uso para una solicitud de información de orden de compra. Al leer un *intent* presente en esta, Rasa procede a ejecutar la acción especificada a continuación del *intent*.

Cabe notar que no es necesario que el usuario pase por el primer paso para poder entrar a una historia. Es decir, si se obtiene el intent *request_order* y este no está presente en otras historias, Rasa entenderá que se entró a esta *story*.

Al momento de crear historias es crucial evitar historias que se contradigan entre sí, es decir, que frente a un mismo *intent* inicial se lleve a distintas acciones. Si puede ocurrir que dos historias distintas lean un mismo mensaje y con eso lleven a distintas acciones, siempre y cuando el *intent* del mensaje esté dentro de un contexto previo.

Ejemplo:

```
- story: historia 1
  steps:
  - intent: greet
  - action: welcome
  - intent: request_order
  - action: check_order
```

```
- story: historia 2
  steps:
  - intent: greet
  - action: welcome
  - intent: request_order
  - action: goodbye
```

Aquí se tiene un conflicto pues tras obtener un mensaje con intención de solicitar una *request*, Rasa no sabrá distinguir si ejecutar la acción *check_order* o *goodbye*, por un error de diseño de historias.

Con todo lo anterior, se diseñaron historias de uso para las solicitudes abordadas dentro de este trabajo (página 39).

Actions

Para poder responder las consultas ingresadas por usuario, Rasa utiliza acciones. Las acciones pueden ser desde algo simple como solo responder con texto simple, como algo más complejo como tomar un dato de usuario y luego realizar una consulta con ello.

Tras contar con un bot básico inicial, se procedió a experimentar con las acciones y las historias de uso, sin embargo, eventualmente se logra deducir que se requiere usar exclusivamente acciones personalizadas, ya que lo primero que debe hacer el Chatbox al recibir un mensaje, es ingresarlo a la base de datos de Moqui. Por lo tanto, solo se cuentan con acciones dentro de Rasa Actions Server, quien cuenta con distintas herramientas para trabajar, las cuales se explican más adelante.

Domains

El *Domain* involucra todo lo que el Chatbox deba contener en su memoria. Es decir, aquí se guarda una lista de todos los *intents*, las acciones, los *slots*, los *forms* (en caso de utilizarse), y las *responses*.

A la hora de invocar cualquiera de las opciones listadas, el Chatbox revisa dentro de su *Domain* por ella. Por ejemplo, cuando el Chatbox sigue una historia de uso y lee la acción que corresponde tomar, revisa dentro de la lista de acciones del dominio si la acción indicada en la historia existe, para luego invocarla.

En caso de querer utilizar más de un Chatbox dentro de un mismo servidor, cada Chatbox debe tener su propio *domain*. Esto facilita el desarrollo de varios Chatbox en paralelo para implementar en distintas soluciones.

Slots

Otra función crucial para el Chatbox es poder utilizar *slots*, que corresponden a la memoria a largo plazo del Chatbox. En éstos se puede guardar información de distinto tipo, sean valores numéricos, de tipo *bool*, *strings*, etc. El *tracker* logra acceder al valor de estos slots, utilizando el método *tracker.get_ slot(slot)*.

Los *slots* son declarados dentro del archivo domain.yml. Para ingresar un *slot*, tan solo es necesario ingresar su nombre y tipo. También se puede ingresar la opción de si éste debe influir en la conversación, es decir, que se tomen distintas rutas en las historias especificadas en caso de contar con este *slot*. Esta opción no se utilizó para este trabajo.

Rasa Actions Server

Para poder realizar acciones que requieran de trabajo con código, Rasa cuenta con un servidor de acciones personalizadas. Para el desarrollo de este servidor se utiliza Python,

pudiendo acceder a todas las herramientas que este otorga.

En particular, Rasa Action Server cuenta con herramientas propias para poder interactuar con la conversación del Chatbox.

Tracker

Para poder interactuar con el usuario, es importante poder recibir y utilizar los mensajes que este ingresa a la plataforma, para eso Rasa Action Server cuenta con un *Tracker*. El *Tracker* puede leer el último mensaje ingresado en el Chatbox, lo que es particularmente útil a la hora de solicitar información de usuario, y posteriormente realizar consultas a la base de datos de Moqui usando estos datos.

Dispatcher

Para poder especificar qué mensaje enviar al usuario, las acciones personalizadas cuentan con un *dispatcher*. El *dispatcher* puede utilizar el método *utter_message(text='texto')* para enviar mensajes al usuario. Además, se pueden especificar botones para que el usuario sepa que consultar al Chatbox. Para cada botón se debe especificar a que *intent* corresponde, y qué texto debería aparecer en el botón.

Acciones personalizadas

Contando con las herramientas mencionadas previamente, Rasa permite trabajar mensajes de usuarios a través de código Python.

Estas acciones están contenidas dentro de clases, que heredan la clase *Action* de Rasa. Dentro de cada clase se debe especificar dos métodos: *name*, donde solo se le asigna un nombre a esta clase para que Rasa lo reconozca (éste debe ser el nombre ingresado como acción dentro del dominio), y *run*, que es donde todo el trabajo de procesamiento ocurre.

Además, se pueden definir funciones dentro del servidor, para que éstas sean utilizadas dentro de las clases. Un ejemplo de función creada es la siguiente:

```
def generate_ticket(text):
    params = {"description":text}
    r = requests.post('http://127.0.0.1:8080/rest/s1/chatbox/ticket',
    auth=(xxxx, xxxx), params=params )
    if r.status_code == 200:
        rjson = r.json()
        return rjson["workEffortId"]
    else:
        return False
```

Esta función realiza una *request* al servidor de Moqui para poder crear un Ticket. Sola-

mente se ingresa el texto con la solicitud del usuario pues los datos adicionales se agregan desde Moqui.

Luego un ejemplo de clase creada en el servidor de acciones es la siguiente:

```
class TicketGenerate(Action):
    def name(self) -> Text:
        return "ticket_generate"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            text_received = tracker.latest_message["text"]
            post_message_user(text_received,
                              tracker.get_slot("conversationId"))
            workEffortId = generate_ticket(tracker.get_slot
                              ("ticket_content"))
            if workEffortId:
                text_to_user = "Ticket ingresado!"
                post_message_chatbox(text_to_user,
                tracker.get_slot("conversationId"))
                dispatcher.utter_message(text=text_to_user)
                return([SlotSet("ticket_content", None)])
            return []
```

Como se puede ver, en *name* solo se indica el nombre de la acción, y en *run* se utilizan las herramientas mencionadas previamente: con el *tracker* se obtiene el último mensaje escrito en el Chatbox, luego se guarda este mensaje en la base de datos de Moqui con la función *post_message_user*, se crea un *ticket* con *generate_ticket* y si se cumplió se indica al usuario que su *ticket* fue ingresado a través del *dispatcher*, además guardando el mensaje enviado por el Chatbox en la base de datos de Moqui a través de *post_message_chatbox*.

4.3. Servicios y Módulo en Moqui

Para poder brindar la información requerida en las consultas por los usuarios en el Chatbox, es necesario acceder a la base de datos de Moqui, ya que se debe consultar sobre órdenes de compra, información de usuarios, y más. Ésto lleva a la creación de un módulo de Chatbox desde Moqui.

En este módulo se tienen dos directorios con archivos claves: data y service.

Data

En data, se crea todo lo necesario para poder realizar los cambios pertinentes a la base de datos de Moqui. Al momento de levantar Moqui, este realiza una carga de datos, en la

cual se puede indicar que valores pueden tomar entidades específicas. En particular aquí se crea el *CommunicationEvent* con *communicationEventTypeId* de tipo "ChatboxMessage", indicando que ahora existen *CommunicationsEvents* que guardan los mensajes ingresados en el Chatbox. Para ello se utiliza el siguiente método:

```
<moqui.party.communication.CommunicationEvent
communicationEventTypeId="ChatboxMessage"
description="Chatbox Message"/>
```

Con esto al momento de cargar los datos, Moqui lee el archivo SetupData.xml dentro del directorio data para la carga de datos, y entiende que ahora pueden existir *CommunicationEvents* con *communicationEventTypeId* de tipo "ChatboxMessage".

De manera similar, se utiliza este archivo para crear las entidades que representan quien es el que envía el mensaje, el Chatbot, o un usuario. Se puede ver en el siguiente código:

```
<mantle.party.Party partyId="ChatboxBot" description="Chatbox Bot" />

<mantle.party.Party partyId="ChatboxUser" description="Chatbox User" />
```

Así, cuando se ingrese a la base de datos el objeto que represente el mensaje, se puede especificar por un lado el cuerpo del mensaje, y además quien está emitiendo este mensaje.

Finalmente, utilizando el mismo procedimiento, desde Data se especifica que existen WorkEfforts del tipo WetChatboxTicket (con Wet significando Work effort type).

Todo lo anterior se contiene dentro de tags indicando que esta información debe cargarse dentro de seed-initial, o sea, los datos iniciales de carga de Moqui, por lo que al momento de levantar el *framework* se actualizan las entidades indicadas.

Service

Desde *service* se crean los servicios propios del módulo de Chatbox. Puesto que Rasa maneja todo el trabajo de back-end del Chatbox, Moqui solo debe habilitar servicios API para acceso a la base de datos. Esto es, servicios que permitan solicitudes PUT y GET para brindar o ingresar las entidades correspondientes.

Para crear estos servicios se contó con dos archivos: ApiServices.xml y chatbox.rest.xml.

Desde chatbox.rest.xml se ingresan los servicios REST que existen dentro del módulo, indicando el nombre, una descripción, y qué servicio utiliza para realizar la solicitud. Ejemplo:

```
<resource name="customer" description="Customer Detail"
require-authentication="anonymous-view">
    <method type="get">
        <service name="moit.erp.ApiServices.get#Customer" />
    </method>
</resource>
```

Como se puede ver en el ejemplo, se ingresa dentro de un tag *resource* el nombre, la descripción y el nivel de autenticación requerido para realizar esta consulta. Para este trabajo se optó por utilizar vistas anónimas, quedando como trabajo a futuro buscar el nivel de autentificación ideal para acceder a estas consultas. Luego, se especifica que tipo de método se utiliza, PUT o GET, y finalmente el o los servicios que se utilizan desde ApiServices.xml para hacer esta consulta.

Por otro lado, desde ApiServices.xml se crean los servicios en sí, con todo lo que ésto involucra: el encabezado, los parámetros de entrada y salida, y qué acciones se realizan para obtener el resultado deseado. El procedimiento general fue el siguiente: se crea un arreglo donde se van insertando los valores deseados como parámetros de salida. Para obtener estos parámetros, se debe revisar inicialmente si se cuenta con los parámetros de entrada requeridos, esto a través de funciones *if*. Ejemplo:

```
<if condition="!orderId">
    <message error="true">
        Se requiere ID de la orden.
    </message>
</if>
```

Moqui permite regresar mensajes de errores como respuesta de salida. Esto es, si no se tiene la condición requerida por *if*, la ejecución del archivo se detiene y se obtiene un mensaje de error como respuesta. Además, al contar con un *error*, cualquier acción realizada en Moqui queda inválida, y si se hizo algún cambio en la base de datos, se procede a realizar el *rollback* respectivo.

Luego el procedimiento varía dependiendo de si es un servicio GET o POST. Para los servicios GET, Moqui cuenta con *entity-find* para la búsqueda de entidades a la base de datos, ingresando los datos para la solicitud a través de *econdition*. Ejemplo:

```
<entity-find entity-name="mantle.order.OrderItem" list="orderItemsList">
    <econdition field-name="orderId" from="orderId" />
</entity-find>
```

En este ejemplo simple se puede apreciar que, al indicar la entidad a buscar, en qué lista ingresar los resultados, y qué parámetro usar para filtrar, se puede realizar consultas a la base de datos. Luego tras asignación de valores y posibles iteraciones por listas (como, por ejemplo, al encontrar todas las partes de una orden de compra), se regresa a través de JSON los valores de salida.

4.4. Conectando Softwares y Resultado final

Como se pudo apreciar en las secciones previas, Moqui logra conectarse con Rasa Action Server a través de consultas API. Desde Moqui, los *end-points* se definen desde chatbox.rest.xml, quedando el *end-point* definido por los nombres ingresados de los *resources*, y desde Rasa se puede conectar gracias a Python utilizando *requests*. Es importante notar que

se requiere de autenticación por parte de Rasa a la hora de realizar la consulta API hacia Moqui. Ejemplo:

```
def init_conversation(bodyText, fromId, toId):
    params = {"body":bodyText,"fromPartyId":fromId, "toPartyId":toId}
    r = requests.post('http://127.0.0.1:8080/rest/s1/chatbox/initialize',
    auth=(xxxx, xxxx), params=params)
```

De esta forma Rasa envía la solicitud al *endpoint initialize* creada desde Moqui, gracias a la autenticación.

Por otro lado, para poder insertar el chatbox dentro de una página web, Rasa provee un bloque de código javascript para insertar en el *body* de ésta. En Moqui, se insertó este código dentro de una pantalla del módulo moit-erp, el cual corresponde al módulo desarrollado por Moit para los procesos de gestión empresarial. El código insertado se aprecia a continuación:

```
<text type="html,vuet,qvt">
    <![CDATA[
        <script>!(function () {
            let e = document.createElement("script"),
                t = document.head ||
                document.getElementsByTagName("head")[0];
            (e.src =
                "https://cdn.jsdelivr.net/npm/
                rasa-webchat@1.x.x/lib/index.js"),
                // Replace 1.x.x with the version that you want
                (e.async = !0),
                (e.onload = () => {
                    window.WebChat.default(
                        {
                            initPayload: 'init',
                            customData: { language: "es" },
                            socketUrl: "http://localhost:5005",
                            params: {
                                storage: "session",
                            }
                            // add other props here
                        },
                        null
                    );
                }),
                t.insertBefore(e, t.firstChild);
            })();
        </script>
    ]]>
</text>
```

Para la integración de este código en Moqui, se utiliza el tag *text* para insertar código

HTML de manera directa. Esto es posible pues se especifica que puede ser lenguaje HTML, VUET y QVT en *type*. Luego, en el contenido del tag, se utiliza la etiqueta *script* de HTML para ingresar el código Javascript de Rasa.

De aquí los parámetros relevantes a cambiar son *initPayLoad* y *socketUrl*. El primero simula un input inicial para que el Chatbox muestre lo que respondería frente a ello. Esta funcionalidad se aprovechó para ejecutar un método de inicialización del Chatbox, creando el ID de la conversación que se almacena junto a cada mensaje que se ingresa en el Chatbox. El segundo indica en qué servidor se está ejecutando Rasa Server, con tal de realizar la conexión a éste. En este caso, se utilizó el servidor local. Para efectos prácticos, esta URL se puede parametrizar por cada proyecto que la utilicen, dejando el módulo de Chatbox genérico.

Finalmente con todo lo anterior, se obtiene el Chatbox dentro de la interfaz de moit-erp:

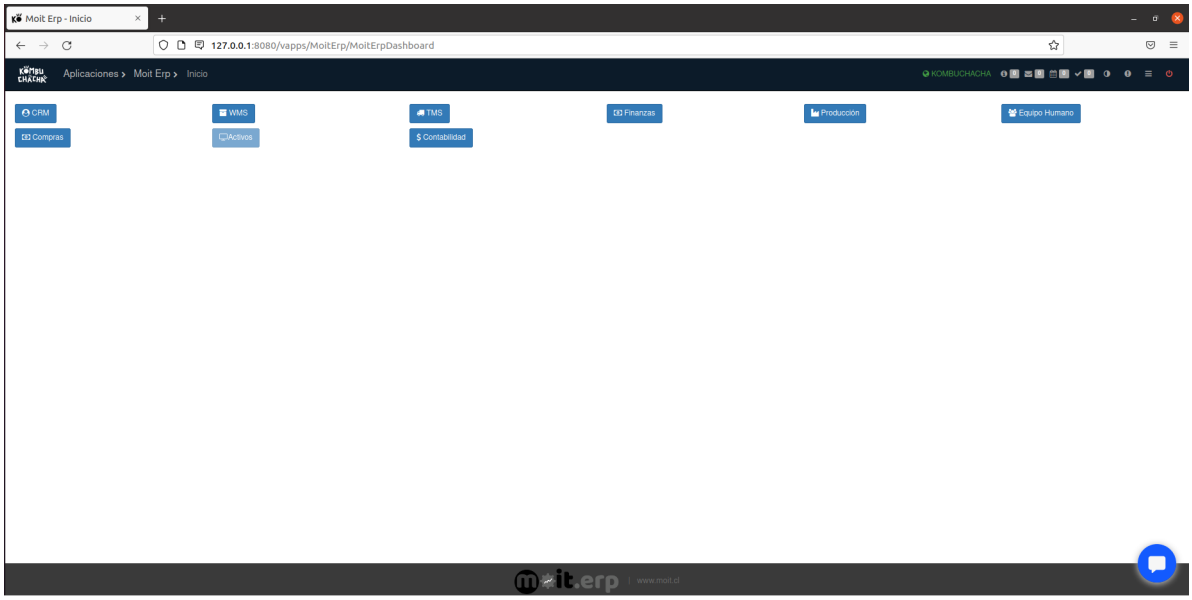

Figura 4.1: Icono de Chatbox dentro de pagina moit-erp.

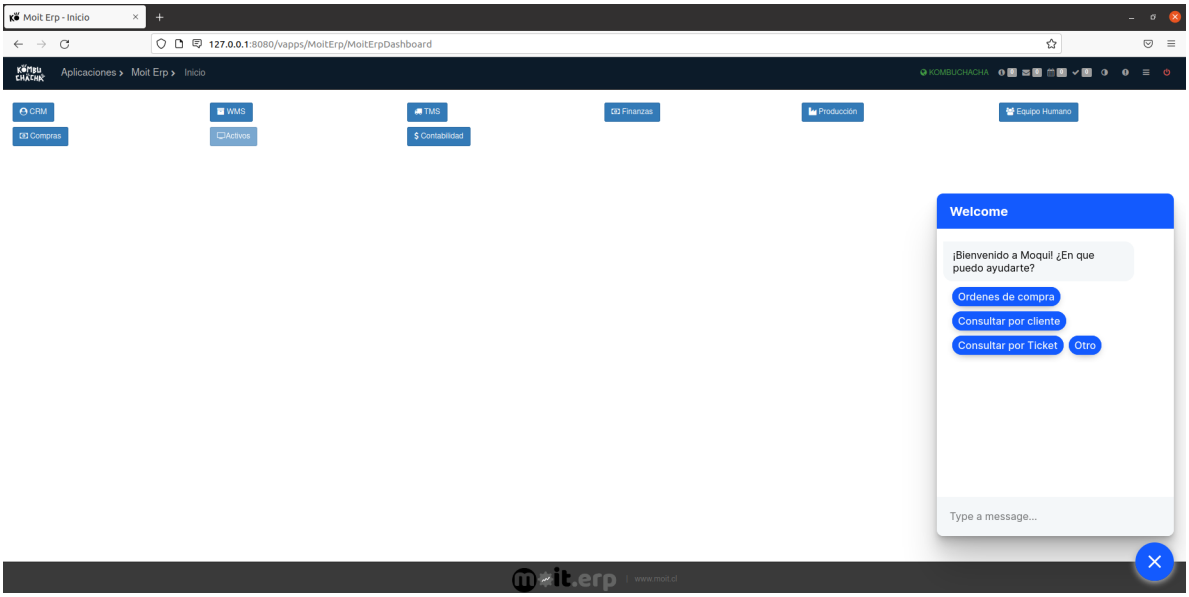

Figura 4.2: Chatbox abierto.

Luego se pueden seleccionar los botones o escribir manualmente las consultas a realizar. En el siguiente ejemplo se ve una solicitud:

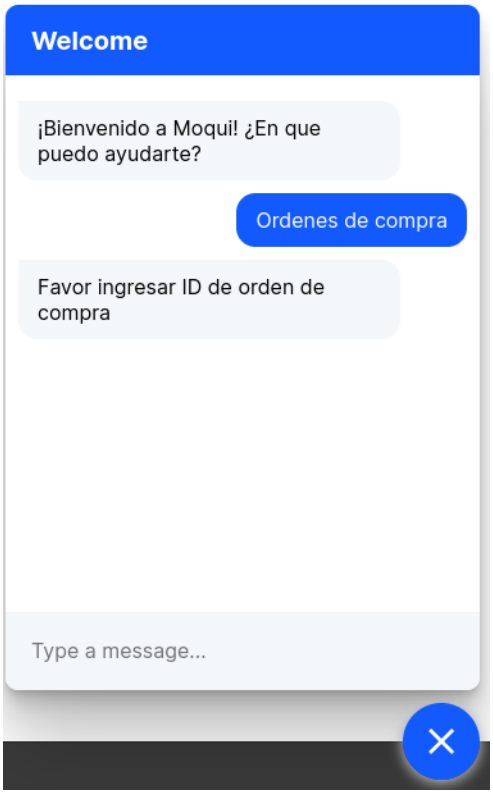

Figura 4.3: Solicitud de orden de compra.

Como se puede ver, el Chatbox solicita la ID de orden de compra. Tras obtenerla, presenta la información pertinente:

Figura 4.4: Respuesta a solicitud de orden de compra.

Aquí, tras recibir el ID por parte del usuario, Rasa lo utiliza para obtener la información solicitada.

Esto solo muestra la funcionalidad inicial con botones para guiar al usuario, pero además se espera que el Chatbox pueda recibir input de usuario. En la siguiente imagen se puede ver un ejemplo:

Figura 4.5: Solicitud información de cliente.

Rasa obtiene el input de usuario y solicita el RUT de cliente. Posterior a esto, al ingresar el RUT, Rasa muestra la información pertinente.

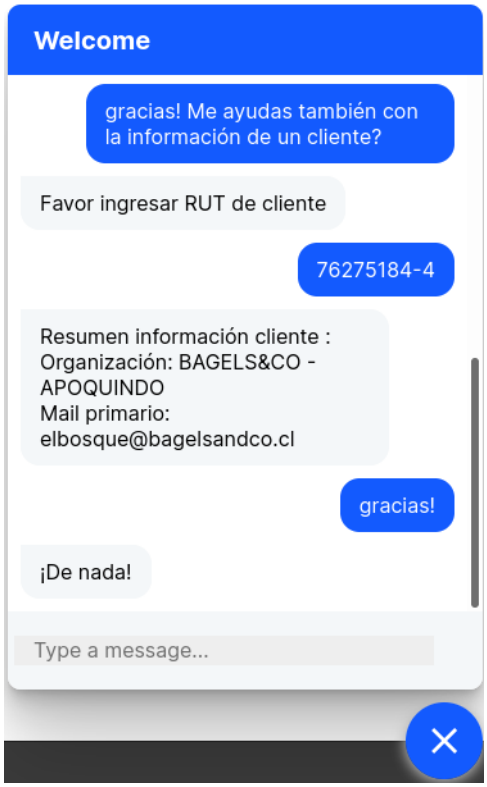

Figura 4.6: Respuesta frente a solicitud.

Capítulo 5

Conclusión

5.1. Retrospectiva

Este trabajo de memoria sirvió como una gran experiencia de aprendizaje para el estudiante. Abordar un tema de memoria sin conocimiento previo de éste ni de las herramientas por utilizar resultó ser un gran desafío, y este desconocimiento llevó a subestimar ciertas fases del plan de trabajo, en particular la fase de investigación. Moqui, a pesar de ser una herramienta potente con muchas facultades positivas, presenta grandes barreras de entradas para tener el conocimiento suficiente para realizar un trabajo de desarrollo en éste.

Pero, a pesar de las complicaciones, se pudo apreciar que es posible abordar y resolver problemáticas nuevas a través de investigación propia y apoyo de un equipo. Lograr unir dos softwares totalmente independientes entre sí gracias a una investigación previa y al avance tecnológico actual que permite esta conexión vía APIs fue una gran oportunidad de crecimiento personal como ingeniero.

Respecto a los objetivos planteados, se logra satisfacer los más importantes: se cuenta con una herramienta de automatización de mensajes, que logra resolver consultas especificadas por usuarios ingresando a la base de datos de Moqui. Esto gracias a la correspondiente investigación de alternativas, el trabajo en conjunto con Moit para la definición de requerimientos y trabajo con Moqui, pues cabe notar que la resolución de dudas brindada por el equipo fue notable, y al acierto frente a la opción elegida con *framework* de Chatbox.

5.2. Trabajo Futuro

El trabajo realizado, a pesar de cumplir el objetivo general de esta memoria, cuenta con varias posibles mejoras y funcionalidades extras que no fueron abordadas dentro de ésta.

En primer lugar, un punto a trabajar a futuro en caso de que Moit, o un futuro memorista aborde este proyecto, sería evaluar un sistema de autentificación para los usuarios que utilizan el Chatbox. Esto para poder filtrar quienes solicitan la información de la base de datos de

Moqui, pues solicitar información de usuario, por ejemplo, podría no ser algo que cualquier usuario debería poder hacer. Además, con eso quizás sería más fácil realizar las consultas mismas por parte del *back-end*.

Por otro lado, una funcionalidad deseada a comienzos de este trabajo era la posibilidad de contar con soporte en línea por parte de un ejecutivo para los usuarios. Debido a restricciones de tiempo, y la dificultad de esta implementación, se optó por omitir esto en el producto final.

Finalmente, cabe notar que en las etapas finales de desarrollo, Rasa anunció la versión 3.0. Pese a que aún no se cuenta con el detalle de que nuevas funcionalidades esta brindará, tiene el potencial de facilitar trabajo futuro con herramientas novedosas.

Bibliografía

[1] Bindbot. https://www.bindbot.cl/.

[2] Botpress. https://botpress.com/.

[3] Botsify. https://botsify.com/.

[4] Cliengo. https://www.cliengo.com/chatbot.

[5] Connect rasa to your website. https://rasa.com/docs/rasa/connectors/your-own-website/.

[6] Databot. https://databot.cl/.

[7] Gat-blanc. https://gat-blac.com/ES/.

[8] Moit, servicios. https://moit.cl/#serv.

[9] Moqui framework. https://www.moqui.org/framework.html.

[10] Rasa actions. https://rasa.com/docs/action-server/sdk-actions.

[11] Rasa custom actions. https://www.youtube.com/watch?v=rvHg7N8ux2I.

[12] Rasa dispatcher. https://rasa.com/docs/action-server/sdk-dispatcher.

[13] Rasa forums. https://forum.rasa.com/.

[14] Rasa open source. https://rasa.com/docs/.

[15] Rasa tracker. https://rasa.com/docs/action-server/sdk-tracker.

[16] Tidio. https://www.tidio.com/.

[17] Uniw. https://www.uniq.ai/.

[18] User. https://user.com/en/.

[19] Droid City. Rasa integration with a webpage. https://www.youtube.com/watch?v=eJMT2FovZsM. Accessed: 2022-02-28.

Anexos

Anexo A

Servicios REST en Moqui, archivo chatbox.rest.xml

```
<resource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/rest-api-2.1.xsd"
      name="chatbox" displayName="Chatbox REST API" version="1.0.0"
      description="Chatbox REST API"
      require-authentication="anonymous-view">
<resource name="customer" description="Customer Detail"
require-authentication="anonymous-view">
    <method type="get">
        <service name="moit.erp.ApiServices.get#Customer" />
    </method>
</resource>

<resource name="order" description="Order info"
require-authentication="anonymous-view">
    <method type="get">
        <service name="ApiServices.get#Order" />
    </method>
</resource>

<resource name="initialize" description="Chatbox Conversation Initializer"
require-authentication="anonymous-view">
    <method type="post">
        <service name="ApiServices.initiate#ChatboxConversation"/>
    </method>
</resource>

<resource name="message" description="Chatbox Message"
require-authentication="anonymous-view">
    <method type="post">
        <service name="ApiServices.post#ChatboxMessage"/>
```

```
        </method>
    </resource>

    <resource name="ticket" description="Ticket Generated from Chatbox"
    require-authentication="anonymous-view">
        <method type="post">
            <service name="ApiServices.post#ChatboxTicket"/>
        </method>
    </resource>

</resource>
```

Anexo B

Servicios API en Moqui, archivo ApiServices.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/service-definition-2.1.xsd">
    <service verb="get" noun="Customer" authenticate="anonymous-all"
    allow-remote="true" transaction-timeout="300000">
        <description>
            Servicio que obtiene datos de un cliente particular para trabajo con
            chatbox.
        </description>
        <in-parameters>
            <parameter name="organizationPartyId" />
            <parameter name="customerPartyId" />
            <parameter name="customerPartyIdentificationValue" />
        </in-parameters>
        <out-parameters>
            <parameter name="customer"/>
        </out-parameters>
        <actions>
            <set field="customer" from="[:]" />

            <if condition="!customerPartyId &amp;amp;&amp;amp;
            !customerPartyIdentificationValue">
                <message error="true">
                    Se requiere identificador de cliente o  rut.
                </message>
            </if>

            <if condition="!customerPartyId">
                <entity-find entity-name="mantle.party.PartyIdentification"
                list="partyIdentificationList">
```

```
        <econdition field-name="idValue"
        from="customerPartyIdentificationValue" />
    </entity-find>
    <if condition="partyIdentificationList.size()!=1">
        <message error="true">
            No se pudo identificar a cliente.
        </message>
    </if>

    <set field="customerPartyId"
    from="partyIdentificationList[0].partyId"/>
</if>

<set field="customer" from="customer+[partyId:customerPartyId]"/>

<entity-find-one entity-name="mantle.party.PartyDetail"
value-field="partyDetail">
    <field-map field-name="partyId" from="customerPartyId"/>
</entity-find-one>

<set field="customer"
from="customer+[organizationName:partyDetail.organizationName,
taxOrganizationName:partyDetail.taxOrganizationName,
firstName:partyDetail.firstName, lastName:partyDetail.lastName]"/>

<entity-find entity-name="mantle.party.contact.PartyContactMechInfo"
list="partyContactMechList">
    <econdition field-name="partyId" from="customerPartyId" />
    <econdition field-name="infoString" operator="is-not-null" />
</entity-find>

<set field="emailList" from="[]"/>
<iterate list="partyContactMechList" entry="partyContactMechItem" >
    <script>
        <![CDATA[
            emailList.add(
                contactMechPurposeId:
                partyContactMechItem.contactMechPurposeId,
                fromDate:partyContactMechItem.fromDate,
                thruDate:partyContactMechItem.thruDate,
                trustLevelEnumId:partyContactMechItem.trustLevelEnumId,
                infoString:partyContactMechItem.infoString)
        ]]>
    </script>
</iterate>

<set field="customer" from="customer+[emails:emailList]"/>
```

```
<entity-find
entity-name="mantle.party.contact.PartyContactMechPostalAddress"
list="partyContactMechPostalAddressList">
    <econdition field-name="partyId" from="customerPartyId" />

</entity-find>

<set field="addressList" from="[]"/>
<iterate list="partyContactMechPostalAddressList"
entry="partyContactMechAddressListItem" >
    <script>
        <![CDATA[
            addressList.add(
                contactMechPurposeId:
                partyContactMechAddressListItem.contactMechPurposeId,
                fromDate:partyContactMechAddressListItem.fromDate,
                thruDate:partyContactMechAddressListItem.thruDate,
                trustLevelEnumId:
                partyContactMechAddressListItem.trustLevelEnumId,
                infoString:partyContactMechAddressListItem.infoString,
                address1:partyContactMechAddressListItem.address1,
                address2:partyContactMechAddressListItem.address2)
        ]]>
    </script>
</iterate>

<set field="customer" from="customer+[address:addressList]"/>
```

```
<entity-find
entity-name="mantle.party.contact.PartyContactMechTelecomNumber"
list="partyContactMechTelecomNumberList">
    <econdition field-name="partyId" from="customerPartyId" />
</entity-find>

<set field="telecomList" from="[]"/>
<iterate list="partyContactMechTelecomNumberList"
entry="partyContactMechTelecomNumberListItem" >
    <script>
        <![CDATA[
            telecomList.add(
                contactMechPurposeId:
                partyContactMechTelecomNumberListItem.
                contactMechPurposeId,
```

```
                        fromDate:partyContactMechTelecomNumberListItem.fromDate,
                        thruDate:partyContactMechTelecomNumberListItem.thruDate,
                        trustLevelEnumId:
                        partyContactMechTelecomNumberListItem.trustLevelEnumId,
                        infoString:
                        partyContactMechTelecomNumberListItem.infoString,
                        contactNumber:
                        partyContactMechTelecomNumberListItem.contactNumber)
                ]]>
            </script>
        </iterate>

        <set field="customer" from="customer+[telecom:telecomList]"/>

    </actions>
</service>

<service verb="get" noun="Order" authenticate="anonymous-all"
allow-remote="true" transaction-timeout="300000">
    <description>
        Servicio que obtiene orden de compra para trabajo con chatbox.
    </description>
    <in-parameters>
        <parameter name="organizationPartyId" />
        <parameter name="customerPartyId" />
        <parameter name="customerPartyIdentificationValue" />
        <parameter name="orderId" />
        <parameter name="orderPartSeqId" />
    </in-parameters>
    <out-parameters>
        <parameter name="order" />
    </out-parameters>
    <actions>
        <set field="order" from="[:]" />
        <if condition="!orderId">
            <message error="true">
                Se requiere ID de la orden.
            </message>
        </if>

        <entity-find-one entity-name="mantle.order.OrderHeader"
        value-field="orderHeader">
            <field-map field-name="orderId" from="orderId" />
        </entity-find-one>
        <if condition="!orderHeader">
            <message error="true">
                No se pudo identificar orden de compra.
```

```
    </message>
</if>
<set field="orderHeader.approvedDate"
from="ec.l10n.format(orderHeader.approvedDate, 'dd-MM-yyyy')" />
<set field="orderHeader.lastUpdatedStamp"
from="ec.l10n.format(orderHeader.lastUpdatedStamp, 'dd-MM-yyyy')" />
<set field="orderHeader.entryDate"
from="ec.l10n.format(orderHeader.entryDate, 'dd-MM-yyyy')" />
<set field="order" from="order+[headerInfo:orderHeader]"/>

<if condition="!orderPartSeqId &amp;amp;&amp;amp; orderId">
    <entity-find entity-name="mantle.order.OrderPart"
    list="orderPartList">
        <econdition field-name="orderId"/>
        <select-field field-name="orderPartSeqId"/>
        <order-by field-name="orderPartSeqId"/>
    </entity-find>
    <set field="orderPartSeqId"
    from="orderPartList?.first?.orderPartSeqId"/>
</if>

<entity-find entity-name="mantle.order.OrderPart" list="orderPartsList">
    <econdition field-name="orderId" from="orderId" />
</entity-find>

<set field="orderParts" from="[]" />
<iterate list="orderPartsList" entry="orderPartsListItem" >
    <script>
        <![CDATA[
            orderParts.add(
                orderId:orderPartsListItem.orderId,
                orderPartSeqId:orderPartsListItem.orderPartSeqId,
                statusId:orderPartsListItem.statusId,
                vendorPartyId:orderPartsListItem.vendorPartyId,
                customerPartyId:orderPartsListItem.customerPartyId,
                )
        ]]>
    </script>
</iterate>

<set field="order" from="order+[orderParts:orderParts]"/>

<entity-find entity-name="mantle.order.OrderItem" list="orderItemsList">
    <econdition field-name="orderId" from="orderId" />

</entity-find>
```

```
        <set field="orderItems" from="[]" />
        <iterate list="orderItemsList" entry="orderItemsListItem" >
            <script>
                <![CDATA[
                    orderItems.add(
                        orderId:orderItemsListItem.orderId,
                        orderPartSeqId:orderItemsListItem.orderPartSeqId,
                        itemTypeEnumId:orderItemsListItem.itemTypeEnumId,
                        itemDescription:orderItemsListItem.itemDescription,
                        quantity:orderItemsListItem.quantity,
                        quantityCancelled:orderItemsListItem.quantityCancelled,

                        )
                ]]>
            </script>
        </iterate>

        <set field="order" from="order+[orderItems:orderItems]"/>

    </actions>
</service>


<service verb="post" noun="ChatboxMessage" authenticate="anonymous-all"
allow-remote="true" transaction-timeout="300000">
    <description>
        Servicio para obtención de mensajes de Chatbox.
    </description>
    <in-parameters>
        <parameter name="body" />
        <parameter name="fromPartyId" />
        <parameter name="toPartyId" />
        <parameter name="parentCommEventId" />
    </in-parameters>
    <out-parameters>
        <parameter name="communicationEventId" />
    </out-parameters>
    <actions>
        <!--<if condition="!body">
            <message error="true">
                Se requiere especificar contenido del mensaje.
            </message>
        </if> -->
        <if condition="!body">
            <return message="Se requiere especificar contenido del mensaje."/>
        </if>
        <set field="fromDate" from="ec.user.nowTimestamp"/>
```

```
        <set field="communicationEventTypeId" value="ChatboxMessage" />

        <service-call
        name="create#mantle.party.communication.CommunicationEvent"
        in-map="[body:body,fromPartyId:fromPartyId,toPartyId:toPartyId,par
        entCommEventId:parentCommEventId,
        fromDate:fromDate,communicationEventTypeId:communicationEventTypeI
        d]" out-map="context" />


    </actions>
</service>

<service verb="initiate" noun="ChatboxConversation"
authenticate="anonymous-all" allow-remote="true"
transaction-timeout="300000">
    <description>
        Servicio para inizialización de mensajes de Chatbox.
    </description>
    <in-parameters>
        <parameter name="body" />
        <parameter name="fromPartyId" />
        <parameter name="toPartyId" />
    </in-parameters>
    <out-parameters>
        <parameter name="communicationEventId" />
    </out-parameters>
    <actions>
        <if condition="!body">
            <return message="Se requiere especificar contenido del mensaje."/>
        </if>
        <set field="conversationDate" from="ec.user.nowTimestamp"/>
        <set field="communicationEventTypeId" value="ChatboxMessage" />

        <service-call
        name="create#mantle.party.communication.CommunicationEvent"
        in-map="[body:body,fromPartyId:fromPartyId,toPartyId:toPartyId,
        entryDate:conversationDate]" out-map="context" />


    </actions>
</service>

<service verb="post" noun="ChatboxTicket" authenticate="anonymous-all"
allow-remote="true" transaction-timeout="300000">
    <description>
        Servicio para ingresar Tickets de consultas a chatbox.
```

```
        </description>
        <in-parameters>
            <parameter name="description" />
        </in-parameters>
        <out-parameters>
            <parameter name="workEffortId" />
        </out-parameters>
        <actions>
            <if condition="!description">
                <return message="Se requiere especificar contenido del ticket." />
            </if>
            <set field="workEffortTypeEnumId" value="WetCustomerTicket" />
            <set field="description" from="description" />
            <service-call name="create#mantle.work.effort.WorkEffort"
            in-map="[workEffortTypeEnumId:workEffortTypeEnumId,
            description:description]" out-map="context" />
        </actions>
    </service>

</services>
```

Anexo C

Rasa Stories

En esta sección se presentan las historias utilizadas en Rasa. Notar que estas son tanto las utilizadas en el producto final, como las utilizadas previamente en la fase de experimentación.

```
    version: "2.0"

stories:

- story: happy path
  steps:
  - intent: greet
  - action: welcome
  - intent: mood_great
  - action: utter_happy

- story: sad path 1
  steps:
  - intent: greet
  - action: welcome
  - intent: mood_unhappy
  - action: utter_cheer_up
  - action: utter_did_that_help
  - intent: affirm
  - action: utter_happy

- story: sad path 2
  steps:
  - intent: greet
  - action: welcome
  - intent: mood_unhappy
  - action: utter_cheer_up
  - action: utter_did_that_help
  - intent: deny
```

```
  - action: goodbye


- story: help path
  steps:
  - intent: greet
  - action: welcome
  - intent: help_request
  - action: utter_help
  - intent: email
  - action: utter_mailaccepted

- story: help path no mail
  steps:
  - intent: greet
  - action: welcome
  - intent: help_request
  - action: utter_help
  - intent: email
  - action: utter_mailaccepted

- story: multiply by 5
  steps:
  - intent: multiply
  - action: utter_ask_number
  - intent: number_given
  - action: custom_multiply

- story: request
  steps:
  - intent: greet
  - action: welcome
  - intent: make_request
  - action: custom_request

- story: other_story
  steps:
  - intent: other
  - action: utter_other

- story: user thanks
  steps:
  - intent: thanks
  - action: utter_youre_welcome

- story: identify user
  steps:
```

```
  - intent: ask_phone
  - action: give_phone

- story: init
  steps:
  - intent: init
  - action: init

- story: order
  steps:
  - intent: greet
  - action: welcome
  - intent: request_order
  - action: check_order
  - intent: give_order_id
  - action: order_set_id

- story: client_info
  steps:
  - intent: greet
  - action: welcome
  - intent: request_user_info
  - action: check_client
  - intent: give_rut
  - action: client_set_rut

- story: out_of_scope
  steps:
  - intent: out_of_scope
  - action: check_out_of_scope
  - intent: affirm
  - action: ticket_redirect

- story: out_of_scope_negative
  steps:
  - intent: out_of_scope
  - action: check_out_of_scope
  - intent: deny
  - action: no_ticket

- story: ticket_request
  steps:
  - intent: ticket_request
  - action: ticket_request
  - action: ticket_form
  - active_loop: ticket_form
```

```
- story: ask_anything_else
  steps:
  - action: ask_need_anything_else
  - intent: deny
  - action: goodbye
```

Anexo D

Rasa Intents NLU Data

En esta sección se presentan las *intents* creadas junto con su data de entrenamiento en Rasa. Notar que estas son tanto las utilizadas en el producto final, como las utilizadas previamente en la fase de experimentación.

```
    version: "2.0"

nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi
    - hello there
    - good morning
    - good evening
    - moin
    - hey there
    - let's go
    - hey dude
    - goodmorning
    - goodevening
    - good afternoon
    - Hola
    - Buenas
    - ola
    - alo
    - Buenos dias
    - Buenas tardes
    - Buenas noches

- intent: goodbye
  examples: |
```

```
    - good afternoon
    - cu
    - good by
    - cee you later
    - good night
    - bye
    - goodbye
    - have a nice day
    - see you around
    - bye bye
    - see you later
    - adios
    - chao
    - hasta la vista

- intent: affirm
  examples: |
    - yes
    - y
    - indeed
    - of course
    - that sounds good
    - correct
    - si
    - claro
    - por supuesto

- intent: deny
  examples: |
    - no
    - n
    - never
    - I don't think so
    - don't like that
    - no way
    - not really


- intent: mood_great
  examples: |
    - perfect
    - great
    - amazing
    - feeling like a king
    - wonderful
    - I am feeling very good
    - I am great
```

```
    - I am amazing
    - I am going to save the world
    - super stoked
    - extremely good
    - so so perfect
    - so good
    - so perfect
    - bien
    - contento
    - perfecto
    - feliz
    - muy feliz
    - alegre
    - me siento bien
    - me siento alegre

- intent: mood_unhappy
  examples: |
    - my day was horrible
    - I am sad
    - I don't feel very well
    - I am disappointed
    - super sad
    - I'm so sad
    - sad
    - very sad
    - unhappy
    - not good
    - not very good
    - extremly sad
    - so saad
    - so sad
    - Tengo pena
    - Estoy triste
    - mi dia fue terrible
    - tuve un mal dia
    - estoy muy triste
    - ando con pena
    - triste
    - desanimado

- intent: bot_challenge
  examples: |
    - are you a bot?
    - are you a human?
    - am I talking to a bot?
    - am I talking to a human?
```

```
    - ¿Eres un robot?
    - Eres un bot?
    - Eres un robot?
    - Eres una persona?
    - Estoy hablando con un chatbox?
    - ¿Estoy hablando con un chatbox?

- intent: email
  examples: |
    - mail@gmail.com
    - abcdef@defg.cl
    - ejemplo@correo.com
    - email@email.com

- intent: help_request
  examples: |
    - necesito ayuda
    - ayudame
    - puedes ayudarme?
    - ayuda
    - no encuentro algo
    - auxilio

- intent: multiply
  examples: |
    - multiplica este valor
    - quiero multiplicar un numero
    - quiero multiplicar esto

- intent: number_given
  examples: |
    - 1
    - 2
    - 3
    - 4

- intent: make_request
  examples: |
    - necesito una request
    - dame esta request
    - requiero saber esto
    - dame este request

- intent: other
  examples: |
    - otro
    - Otro
```

```
- intent: thanks
  examples: |
    - gracias
    - gracias!
    - Gracias
    - Gracias!

- intent: ask_phone
  examples: |
    - Quiero revisar mi telefono
    - Telefono
    - requiero mi telefono
    - muestra mi telefono registrado
    - cual es mi telefono?

- intent: give_rut
  examples: |
    - 12345678-9
    - 97531642-5
    - 22365525-8
    - 8543256-k
    - 2344551-1

- intent: request_order
  examples: |
    - quiero revisar mi orden de compra
    - dame mi orden de compra
    - necesito ver mi orden de compra
    - dame el estado de mi orden de compra
    - requiero ver mi orden de compra
    - ordenes de compra

- intent: init
  examples: |
    - init

- intent: give_order_id
  examples: |
    - 100011
    - 100000
    - 123456
    - 100123

- intent: request_user_info
  examples: |
    - quiero la información de un cliente
```

```
    - quiero el telefono de juanito
    - dame la dirección de falabella
    - consultar por un cliente

- intent: ticket_request
  examples: |
    - quiero crear un ticket
    - envía un ticket por favor
    - requiero un ticket de ayuda
    - crea un ticket
    - crear ticket
    - ticket

- intent: out_of_scope
  examples: |
    - ¿Cuanto es 2+2?
    - Dame un dato de departamento
    - Has mi tesis
    - Quiero pedir comida
    - Quien es el presidente?
```

Anexo E

Rasa Domain

En esta sección se presenta el dominio del Chatbox desarrollado. Cabe notar que incorpora acciones y elementos utilizados durante la fase de investigación, que no afectan al producto final.

```
    version: "2.0"

intents:
  - affirm
  - ask_phone
  - bot_challenge
  - deny
  - email
  - give_order_id
  - give_rut
  - goodbye
  - greet
  - help_request
  - init
  - make_request
  - mood_great
  - mood_unhappy
  - multiply
  - number_given
  - order_get
  - other
  - out_of_scope
  - request_order
  - request_user_info
  - thanks
  - ticket_request

actions:
```

```
  - ask_need_anything_else
  - bot_reply
  - check_order
  - check_client
  - check_out_of_scope
  - client_get
  - client_set_rut
  - custom_multiply
  - custom_request
  - give_phone
  - goodbye
  - init
  - order_get
  - order_set_id
  - no_ticket
  - request_id
  - request_rut
  - ticket_generate
  - ticket_redirect
  - ticket_request
  - welcome

slots:
  conversationId:
    type: text
    influence_conversation: false
  orderId:
    type: text
    influence_conversation: false
  orderInfo:
    type: text
    influence_conversation: false
  rut:
    type: text
    influence_conversation: false
  scope_flag:
    type: bool
    influence_conversation: false
  ticket_content:
    type: text
    influence_conversation: false
  userdata:
    type: text
    influence_conversation: false


forms:
```

```
  ticket_form:
    required_slots:
      ticket_content:
      - type: from_text

responses:
  utter_greet:
  - text: "¡Bienvenido a Moqui! ¿En que puedo ayudarte?"
    buttons:
    - title: "Ordenes de compra"
      payload: "request_order"
    - title: "Revisar facturas"
      payload: "/make_request"
    - title: "Consultar por Ticket"
      payload: "/make_request"
    - title: "Información de cliente"
      payload: "/make_request"
    - title: "Otro"
      payload: "/other"

  utter_cheer_up:
  - text: "Aquí hay algo para animarte:"
    image: "https://i.imgur.com/nGF1K8f.jpg"

  utter_did_that_help:
  - text: "Eso te ayudó?"

  utter_happy:
  - text: "Excelente, sigamos!"

  utter_goodbye:
  - text: "Hasta luego!"

  utter_iamabot:
  - text: "Soy un bot, desarrollado por Rasa."

  utter_help:
  - text: "Ingresa tu correo electronico para poder ayudarte por favor"

  utter_maildenial:
  - text: "Sin tu correo la ayuda que puedo brindar es limitada, sin embargo puedo int

  utter_mailaccepted:
  - text: "Gracias, ahora cuentame, en que puedo ayudarte?"

  utter_ask_number:
  - text: "Por favor escribe el numero que quieres multiplicar"
```

```
  utter_other:
  - text: "Favor escribir solicitud"

  utter_youre_welcome:
  - text: "¡De nada!"

session_config:
  session_expiration_time: 60
  carry_over_slots_to_new_session: true
```

Anexo F

Rasa Custom Actions

En esta sección se presentan las acciones personalizadas incorporadas en Rasa. Notar que se presentan acciones incorporadas tanto en el producto final, como las utilizadas durante la fase de investigación.

```
    # This files contains your custom actions which can be used to run
# custom Python code.
#
# See this guide on how to implement these action:
# https://rasa.com/docs/rasa/custom-actions



from typing import Any, Text, Dict, List

from rasa_sdk import Action, Tracker, events
from rasa_sdk.executor import CollectingDispatcher
from rasa_sdk.events import SlotSet
import requests
import json

def init_conversation(bodyText, fromId, toId):
    params = {"body":bodyText,"fromPartyId":fromId, "toPartyId":toId}
    r = requests.post('http://127.0.0.1:8080/rest/s1/chatbox/initialize',
    auth=('info@kombuchacha.cl', 'demomoit'), params=params)
    if r.status_code == 200:
        rjson = r.json()
        print(r.status_code)
        return rjson["communicationEventId"]
    else:
        print("algo salió mal " + str(r.status_code()))
        return False
```

```
def post_message_user(bodyText, parentId):
    params = {"body":bodyText,"fromPartyId":"ChatboxUser",
    "toPartyId":"ChatboxBot", "parentId":parentId}
    r = requests.post('http://127.0.0.1:8080/rest/s1/chatbox/message',
    auth=('info@kombuchacha.cl', 'demomoit'), params=params)
    if r.status_code == 200:
        rjson = r.json()
        return rjson["communicationEventId"]
    else:
        return False

def post_message_chatbox(bodyText, parentId):
    params = {"body":bodyText,"fromPartyId":"ChatboxBot",
    "toPartyId":"ChatboxUser", "parentId":parentId}
    r = requests.post('http://127.0.0.1:8080/rest/s1/chatbox/message',
    auth=('info@kombuchacha.cl', 'demomoit'), params=params)
    if r.status_code == 200:
        rjson = r.json()
        return rjson["communicationEventId"]
    else:
        return False

def generate_ticket(text):
    params = {"description":text}
    r = requests.post('http://127.0.0.1:8080/rest/s1/chatbox/ticket',
    auth=('info@kombuchacha.cl', 'demomoit'), params=params )
    if r.status_code == 200:
        rjson = r.json()
        return rjson["workEffortId"]
    else:
        return False


class init(Action):
    def name(self) -> Text:
        return "init"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            conversation_id = init_conversation("init","ChatboxBot","ChatboxUser")
            if conversation_id:
                print("tengo el id")
                return [events.FollowupAction("welcome"),
                SlotSet("conversationId", conversation_id)]
            else:
```

```
                print("algo salio mal")
                return []



class Welcome(Action):
    def name(self) -> Text:
        return "welcome"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            buttons = []
            buttons.extend([{"title": 'Ordenes de compra' , "payload":
            '/request_order'},{"title": 'Consultar por cliente' , "payload":
            '/request_user_info'},{"title": 'Consultar por Ticket' ,
            "payload": '/make_request'},{"title": 'Otro' , "payload":
            '/other'}])
            output_text = "¡Bienvenido a Moqui! ¿En que puedo ayudarte?"
            dispatcher.utter_message(text= output_text , buttons=buttons)
            print(tracker.get_slot("orderId"))
            print(tracker.get_slot("conversationId"))
            post_message_chatbox(output_text, tracker.get_slot("conversationId"))
            return []



class ActionMultiply(Action):

    def name(self) -> Text:
        return "custom_multiply"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        number = int(tracker.latest_message['text'])
        result = str(number*5)

        dispatcher.utter_message(text = result)
        #post_message(result)

        return []

#class ActionRequest(Action):

#    def name(self) -> Text:
#        return "custom_request"
```

```
#    def run(self, dispatcher: CollectingDispatcher,
#            tracker: Tracker,
#            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
#            r =requests.get('http://127.0.0.1:8080/rest/s1/moqui/basic/enums
', auth=('info@kombuchacha.cl', 'demomoit'))
#            print(r.headers)
#            dispatcher.utter_message(text = r.text)
            #post_message(r.text)


class TelephoneGet(Action):
    def name(self) -> Text:
        return "give_phone"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            userdata = tracker.get_slot("userdata")
            if userdata:
                telefono = userdata["customer"]["telecom"][0]["contactNumber"]
                dispatcher.utter_message(text=telefono)
                return []

            dispatcher.utter_message(text="Favor ingresar rut")
            return []


class OrderGet(Action):
    def name(self) -> Text:
        return "order_get"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            orderId = tracker.get_slot("orderId")
            params = {'orderId':orderId}
            r =requests.get('http://127.0.0.1:8080/rest/s1/chatbox/order',
            auth=('info@kombuchacha.cl', 'demomoit'), params = params)
            rjson = r.json()
            if r.status_code == 200:
                print(tracker.get_slot("orderId"))
                header = rjson["order"]["headerInfo"]
                status = header["statusId"]
                if status is None:
                    status = "No especificado"
                total = str(header["grandTotal"])
```

```
                if total is None:
                    total = "No especificado"
                entryDate = header["entryDate"]
                if entryDate is None:
                    entryDate = "No especificado"
                resumen = f"Resumen orden de compra
                {tracker.get_slot('orderId')}: \nStatus: {status} \nCosto
                total: {total} \nFecha de solicitud: {entryDate}"

                dispatcher.utter_message(text = resumen)
                return [SlotSet("orderInfo", rjson)]

            dispatcher.utter_message(text="No se logró identificar orden de
            compra, favor revisar la orderId.")
            return []


class setOrderId(Action):
    def name(self) -> Text:
        return "order_set_id"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            orderId = tracker.latest_message['text']
            return[SlotSet("orderId",orderId),events.FollowupAction("order_get")]

class CheckOrder(Action):
    def name(self) -> Text:
        return "check_order"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            if tracker.get_slot("orderId"):
                print("cuento con orderID, procediendo a order_get")
                #aquí debería consultar si quiere la info de esa orden o de otra
                return [events.FollowupAction("order_get")]
            else:
                print("no cuento con orderID, procediendo a request_id")
                return[events.FollowupAction("request_id")]


class RequestId(Action):
    def name(self) -> Text:
        return "request_id"
```

```
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            msg = tracker.latest_message["text"]
            id = tracker.get_slot("conversationId")
            if not id:
                print("error al inicializar conversación - no se pudo
                encontrar conversastionId")
            print("procedo a obtener id")
            post_message_user(msg, id)
            msg_to_user = "Favor ingresar ID de orden de compra"
            post_message_chatbox(msg_to_user, id)
            dispatcher.utter_message(text = msg_to_user)
            return []

class BotChallenged(Action):
    def name(self) -> Text:
        return "bot_reply"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            msg = tracker.latest_message["text"]
            id = tracker.get_slot("conversationId")
            if id:
                post_message_user(msg,id)
                reply = "Soy un bot, desarrollado por Rasa."
                post_message_chatbox(reply,id)
                dispatcher.utter_message(reply)


class CheckClient(Action):
    def name(self) -> Text:
        return "check_client"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            if tracker.get_slot("rut"):
                return [events.FollowupAction("client_get")]
            else:
                return[events.FollowupAction("request_rut")]

class RequestRut(Action):
    def name(self) -> Text:
        return "request_rut"
```

```
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            msg = tracker.latest_message["text"]
            id = tracker.get_slot("conversationId")
            if id:
                post_message_user(msg, id)
                msg_to_user = "Favor ingresar RUT de cliente"
                post_message_chatbox(msg_to_user, id)
                dispatcher.utter_message(text = msg_to_user)
            return []

class ClientGet(Action):

    def name(self) -> Text:
        return "client_get"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

            rut = tracker.get_slot("rut")
            params = {'customerPartyIdentificationValue':rut}
            r =requests.get('http://127.0.0.1:8080/rest/s1/chatbox/customer',
            auth=('info@kombuchacha.cl', 'demomoit'), params = params)
            rjson = r.json()
            print(rjson)
            customer = rjson["customer"]
            if r.status_code == 200:
                customer = rjson["customer"]
                organization = customer["organizationName"]
                clientPrimaryEmail = ""
                for mail in customer["emails"]:
                    if mail["contactMechPurposeId"] == "EmailPrimary":
                        clientPrimaryEmail = mail["infoString"]
                        break

                info = f"Resumen información cliente : \nOrganización:
                {organization} \nMail primario: {clientPrimaryEmail}"
                dispatcher.utter_message(text = info)
                return [SlotSet("userdata", rjson)]

            dispatcher.utter_message(text="No se logró identificar usuario,
            favor revisar correo ingresado.")
            return []

class setClientRut(Action):
```

```
    def name(self) -> Text:
        return "client_set_rut"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            clientRut = tracker.latest_message['text']
            return[SlotSet("rut",clientRut),events.FollowupAction("client_get")]

#class requestTicketContent(Action):
#    def name(self) -> Text:
#        return "client_set_rut"
#
#    def run(self, dispatcher: CollectingDispatcher,
#            tracker: Tracker,
#            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
#            msg = tracker.latest_message["text"]
#            id = tracker.get_slot("conversationId")
#            post_message_user(msg,id)
#            msg_out = "Favor ingrese mensaje para Ticket"
#            post_message_chatbox(msg_out,id)


#class CheckOutOfScope(Action):
#    def name(self) -> Text:
#        return "check_out_scope"
#
#    def run(self, dispatcher: CollectingDispatcher,
#            tracker: Tracker,
#            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
#            text_received = tracker.latest_message["text"]
#            post_message_user(text_received,
tracker.get_slot("conversationId"))
#            if tracker.get_slot("scope_flag"):
#                workEffortId = generate_ticket(text_received)
#                if workEffortId:
#                    return([SlotSet("scope_flag",
False),events.FollowupAction("client_get")])
#                return []
#            else:
#                buttons = []
#                buttons.extend([{"title": 'Si' , "payload":
'/affirm'},{"title": 'No' , "payload": '/deny'}])
#                text_to_user = "Esta solicitud no está dentro de lo que
puedo manejar. Desea generar un Ticket?"
#                dispatcher.utter_message(text = text_to_user, buttons =
buttons)
```

```
#                return []

class CheckOutOfScope(Action):
    def name(self) -> Text:
        return "check_out_of_scope"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            text_received = tracker.latest_message["text"]
            post_message_user(text_received, tracker.get_slot("conversationId"))
            buttons = []
            buttons.extend([{"title": 'Si' , "payload": '/affirm'},{"title":
            'No' , "payload": '/deny'}])
            text_to_user = "Esta solicitud no está dentro de lo que puedo
            manejar. Desea generar un Ticket?"
            dispatcher.utter_message(text = text_to_user, buttons = buttons)
            return [SlotSet("ticket_content", text_received)]


class TicketRedirect(Action):
    def name(self) -> Text:
        return "ticket_redirect"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
                return [events.FollowupAction("ticket_request")]

class TicketRequest(Action):
    def name(self) -> Text:
        return "ticket_request"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
                text = tracker.latest_message["text"]
                post_message_user(text,tracker.get_slot("conversationId"))
                text_to_user = "Favor escribir solicitud para ticket"
                post_message_chatbox(text_to_user, tracker.get_slot("conversationId"))
                dispatcher.utter_message(text=text_to_user)
                return []

class TicketGenerate(Action):
    def name(self) -> Text:
        return "ticket_generate"
```

```
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            text_received = tracker.latest_message["text"]
            post_message_user(text_received, tracker.get_slot("conversationId"))
            workEffortId = generate_ticket(tracker.get_slot("ticket_content"))
            if workEffortId:
                text_to_user = "Ticket ingresado!"
                post_message_chatbox(text_to_user, tracker.get_slot("conversationId"))
                dispatcher.utter_message(text=text_to_user)
                return([SlotSet("ticket_content", None)])
            return []


class noTicket(Action):
    def name(self) -> Text:
        return "no_ticket"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
                text = tracker.latest_message["text"]
                post_message_user(text,tracker.get_slot("conversationId"))
                text_to_user = "Ok, puedo ayudarle en algo más?"
                post_message_chatbox(text_to_user)
                dispatcher.utter_message(text=text_to_user)
                return []

class AskNeedAnythingElse(Action):
    def name(self) -> Text:
        return "ask_need_anything_else"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            text_to_user = "Necesitas algo más?"
            post_message_chatbox(text_to_user,tracker.get_slot("conversationId"))
            dispatcher.utter_message(text= text_to_user)
            return []

class Goodbye(Action):
    def name(self) -> Text:
        return "goodbye"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
```

```
id = tracker.get_slot("conversationId")
post_message_user(tracker.latest_message["text"], id )
text_to_user = "Hasta pronto!"
post_message_chatbox(text_to_user, id)
dispatcher.utter_message(text= text_to_user)
return []
```

Anexo G

Documentación de Requisitos

En esta sección se presenta los requerimientos acordados a comienzo de proyecto junto a equipo de Moit.

Requerimientos

Configuración Chatbox para usuario administrador	Criterio aceptación
Como usuario administrador quiero poder configurar el Chatbot para poder personalizar las preguntas que un usuario realiza.	• Dado usuario administrador y consulta creada en configuraciones del Chatbot cuando el usuario administrador accede a configuraciones entonces se logra cambiar configuración asociada a consulta • Dado usuario administrador y consulta nueva a generar entonces se logra crear nueva pregunta en Chatbot • Dado usuario administrador y consulta existente entonces se logra acceder a configuración asociada a pregunta • Dado usuario administrador y consulta existente entonces se logra modificar la configuración asociada a pregunta • Dado usuario administrador y consulta existente entonces se logra eliminar la pregunta del Chatbot

Reconocimiento de solicitudes estándar	Criterio aceptación
Como usuario Cliente quiero poder realizar consulta sobre alguna entidad y obtener la información pertinente a esta para tener acceso a información de manera cómoda	• Dado un usuario y un producto con inventario disponible cuando el usuario desea conocer este inventario preguntando en el Chatbox, entonces el Chatbox responde con el inventario de este producto.
Como usuario Cliente quiero que Chatbox sea capaz de solicitar la información necesaria para cumplir tareas	• Dado un usuario y pregunta que requiere obtener información de usuario cuando el usuario realiza esta consulta en Chatbox, entonces Chatbox solicita y captura información necesaria.
Como usuario Cliente quiero poder ordenar acciones de manera directa a Chatbox para poder facilitar tareas de la plataforma	• Dado un usuario y una orden de compra nueva cuando usuario ingresa esta orden de compra en Chatbox, entonces Chatbox logra ingresar orden de compra al sistema.

Almacenamiento de Tickets en Backend	Criterio aceptación
Como usuario administrador quiero contar con la información de un Ticket para poder obtener la información de la interacción usuario - Chatbox	• Dado un usuario administrador y un Ticket cuando administrador solicita información de ticket a Backend entonces se obtiene la información en Ticket.
Como usuario Cliente quiero poder enviar un Ticket a Backend para que mi consulta sea recibida.	• Dada una consulta que requiere enviar Ticket a Backend cuando usuario ingresa esta consulta, entonces Chatbot almacena este Ticket en Backend.
Como usuario administrador quiero que Chatbox sepa reconocer el contexto sobre el cual debe enviar un Ticket para poder resolver lo que contenga este Ticket	• Dado un usuario y una pregunta no reconocida por Chatbox cuando usuario ingresa dicha pregunta en Chatbox entonces Chatbox genera Ticket y lo almacena en Backend. • Dado un usuario cuando este usuario solicita soporte en vivo en Chatbox entonces Chatbox almacena Ticket y redirige a soporte en línea.

Creación Backend del módulo	Criterio aceptación
Como usuario administrador quiero contar con registro de conversaciones realizadas en Chatbox para poder obtener información a partir de estas.	• Dada una interacción previa usuario-Chatbox cuando se solicita acceso a esta información entonces se logra acceder al historial de esta conversación.
Como usuario Administrador quiero contar con registro de la información de los usuarios que han utilizado el Chatbox	• Dado un usuario específico que interactuó con Chatbox cuando usuario administrador solicita usuarios que han interactuado con Chatbox entonces se cuenta con el usuario específico dentro de los usuarios solicitados.

Transición a chat en vivo tras no poder resolver consulta	Criterio aceptación
Como usuario quiero contar con soporte de chat en vivo con persona para concretar solicitudes	• Dada pregunta a Chatbox que este no sabe reconocer cuando usuario realiza esta pregunta entonces Chatbox redirige solicitud a soporte en línea.

Soporte servicios Moqui	Criterio aceptación
Como usuario quiero que el Chatbox me guie con el uso de Moqui	• Dado un usuario cuando consulta donde encontrar sus órdenes de compra en Chatbox, entonces Chatbox responde con url para ir a dicha sección. • Dado un usuario cuando solicita recibir atención con un ejecutivo, entonces Chatbox comunica a usuario con ejecutivo sujeto a disponibilidad. • Dado un usuario cuando solicita ingresar un reclamo en Chatbox, entonces Chatbox ingresa dicho reclamo en Backend.
Como usuario administrador quiero que Chatbox notifique a Backend cuando se requiere de soporte en línea	• Dado un usuario cuando este ha solicitado soporte en línea, entonces Chatbox notifica a Backend que se requiere un ejecutivo.
Como usuario ejecutivo quiero poder indicar que me encuentro disponible para resolver consultas en linea, para que usuarios puedan consultar por soporte.	• Dado usuario ejecutivo cuando este indica estar disponible para soporte en línea, entonces Chatbox habilita soporte en línea.

Autentificación de usuarios	Criterio de aceptación
Como usuario quiero que el sistema reconozca que estoy logueado en el sistema para que pueda obtener mi información	• Dado un usuario cuando este solicita información propia de este entonces Chatbox reconoce al usuario y regresa información correspondiente.
Como usuario anonimo quiero que el sistema pueda pedir información para autentificarme	• Dado un usuario no autentificado pero existente en el sistema cuando este pregunta algo que requiere autentificarse, entonces Chatbox solicita credenciales • Dado un usuario no autentificado pero existente en el sistema cuando este ingresa sus datos en Chatbox tras ser solicitados entonces Chatbox logra autentificar al usuario.

Definiciones no funcionales
Se debe contar con frontend para Chatbox, a partir de potencial bloque html-javascript.

Perfiles de usuarios:

- Cliente
- Usuario anónimo
- Administradores
- Ejecutivos