Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Adaptive, Context-aware and Engaging AI conversational agent for IoT Service Portal

Supervisor:
Aalwahab Dhulfiqar, Ph.D
Assistant Professor

Author:
Aadarsh Mehdi
Computer Science BSc

Budapest, 2023

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Registration Form

Attention! The deadline for amendment is 1 February in case of final exam in June and 31 August in case of final exam in January.

Student's Data:

Student's Name: Mehdi Aadarsh

Student's Neptun code: I1E7UO

Course Data:

Student's Major: Computer Science BSc

I have an external supervisor

External Supervisor's Name: *Abrar hussain*

Supervisor's Home Institution: *Ericsson Magyarorszag*

Address of Supervisor's Home Institution: *Budapest, Magyar Tudósok Körútja 11, 1117*

Supervisor's Position and Degree: *Verification Engineer , MSc*

Supervisor's e-mail address: *abrar.hussain@ericsson.com*

Internal Supervisor's Name: *Dr. Dhulfiqar A Alwahab*

Supervisor's Home Institution: ***Department of Programming Languages and Compilers***

Address of Supervisor's Home Institution: ***1117, Budapest, Pázmány Péter sétány 1/C.***

Supervisor's Position and Degree: Assistant professor, PhD

Thesis Title: Adaptive, Context-aware and Engaging AI conversational agent for IoT Service Portal

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Problem Statement:

General Overview about the IoT Service Platform:

This platform enables enterprises to easily deploy, manage, and scale their global IoT business from a unified IoT platform. Delivered as a service through a global partner network, it provides complete visibility and control of all IoT devices throughout their entire lifecycle, no matter where in the world they are located.

About the IoT Service Portal:

The operator and enterprise customer use the IoT service portal to manage their subscriptions. This management comprises enabling/disabling the network connectivity, activating the billing state, fetching the analytics, onboarding new customers, ordering sims, changing the subscription plan, and accessing a number of other features.

Problem Objectives:

About Proposed AI Conversational Agent (Digital Assistant):

There is a need to enhance user experience and engage customers in the IoT Business space. To do this, a Digital Assistant is being developed to provide a single point of contact with an improved conversational experience. This assistant will be integrated with subscription databases and real-time analytics, enabling it to offer the following services:

• Context aware answers and guidance regarding portal configuration.

• Assisting with onboarding new customers.
• Offering recommendations based on a customer's profile, past behavior, and other relevant information.
• Enhancing service management experience by automating services/configurations in the portal.
• Reminding customers about deadlines, important modifications, and new features in the portal.
• Providing analytics related to customer subscriptions.

Problem Design:
The proposed digital assistant will be created using the RASA framework. This approach will enable the incorporation of our own models or the fine-tuning of pre-trained models. It will also include a Natural Language Understanding (NLU) component for context recognition, and dialogue managers to deal with the flow of conversation. Further details about the design will be shared in the thesis.

Budapest, 2023. 05. 23.

Contents

Chapter 1

Introduction

1.1 Motivation

The rapid emergence of enterprises in the Internet of Things (IoT) industry has led to an unprecedented surge in data generation, as devices and sensors constantly communicate and relay information. Companies like Ericsson, Aeris, Telenor, and Verizon are at the forefront of this revolution, innovating and developing tools to help users understand and manipulate data according to their needs. For example, Ericsson is investing in connected cars and automobiles, as well as cellular eSIM technology, broadening the scope of IoT applications beyond traditional hardware.

In today's digital era, the increasing scalability of connectivity in the business sphere has the potential to drive innovation in various domains. One such advancement is the IoT Service Portal, which aims to streamline operations, enhance customer engagement, and unlock valuable data-driven insights. However, as users continuously seek more convenient and user-friendly solutions, the need for a digital assistant integrated within the IoT Service Portal becomes apparent.

A digital assistant can address users' needs by providing quick answers to their queries, offering a conversational interface for executing tasks, and simplifying access to essential information without requiring users to navigate complex settings or sift through documentation. Ultimately, the introduction of a digital assistant can play a vital role in fostering business growth and customer satisfaction, as it caters to the ever-evolving demands of users in an increasingly interconnected world. This thesis, therefore, seeks to explore the design, implementation, and impact of a digital

assistant within the context of IoT Service Portals, setting the stage for enhanced user experiences and more efficient operations in the IoT industry.

1.2 Goal

As a result of the motivation outlined previously, my objective is to create a Digital Assistant within the context of IoT Service Portals using the RASA Framework. Additionally, deploying this service in a server using Microservices like architecture (Docker-Compose). This development aims to enhance user experience and make a significant impact on businesses operating in the IoT industry.

1.3 IoT Service Portal

The IoT Service Portal offers a web-based platform for operators, advanced resellers, and businesses, enabling them to manage subscription fleets, monitor traffic, and automate processes. It also integrates reporting, support, and administrative functions. Operators and advanced resellers can oversee subscription fleets for their enterprise clients and access additional capabilities like enterprise and price plan administration. This thesis showcases a version of the IoT Service Portal that highlights select features, primarily to demonstrate the functionality of the IoT Digital Assistant embedded within it.

1.4 IoT Digital Assistant

The IoT Digital Assistant is an interactive chatbot specifically designed for the IoT Service portal, assisting users with inquiries related to cellular IoT businesses. Developed using the RASA framework, this NLP-based chatbot excels in understanding user queries in natural language and generating accurate, contextually relevant responses.

At the heart of the IoT Digital Assistant is the Natural Language Understanding (NLU) system, which comprises two primary components: intents and entities. To better explain these concepts, let's consider an example. Imagine a user asks, "I want to book a hotel in Budapest." As humans, we can easily deduce the user's intention

is to make a booking. We can also identify that the user wants to book a "HOTEL" and that the desired location is "BUDAPEST." In NLU terms, "BOOKING" would be the intent, and "HOTEL" and "BUDAPEST" would be entities, which can be further classified as BOOKING-ENTITY and "CITY," respectively.

Once the NLU component maps user queries, the next step in the chatbot's pipeline is the Dialogue Manager. This powerful component is responsible for managing the flow of conversation and predicting suitable actions based on user queries. The Dialogue Manager also stores intents and entities from user queries, ensuring a seamless and contextually relevant conversation.

For instance, let's consider a scenario where a user requests to onboard a new enterprise. The IoT Digital Assistant, through its Dialogue Manager, would initiate an onboarding form and proceed to ask for required information in a step-by-step manner. These data points are stored in memory "slots," enabling users to pick up the conversation where they left off, even after interruptions or unrelated questions.

The Dialogue Manager is also connected to custom actions, facilitating interactions with databases, APIs, and managing the flow of conversations. Once the manager generates a response, it is sent directly to the user.

This thesis aims to explore a diverse range of use cases involving natural language queries, such as:

- Engaging in general chitchat, sharing jokes, and answering questions about the chatbot itself
- Providing information on specific topics upon request.
- Guiding users through portal configuration procedures or addressing IoT-related topics (using mocked data).
- Assisting users with onboarding new customers to the portal.
- Managing database fields for inventory, subscriptions, and customers (lock, unlock, or update).
- Offering users access to view the database.
- Presenting current headlines from around the world.

- Retrieving user data analytics upon request.

Although the chatbot is capable of understanding troubleshooting queries in context, it is currently unable to address them due to the lack of a running server engine. As a result, this functionality is limited to production environments and cannot be demonstrated, even in a mock server setting.

Chapter 2

User Documentation

In this chapter, we will provide a comprehensive guide on how to successfully set up and utilize the Digital Assistant for IoT Service Portals. We will begin by outlining the necessary steps for installing the required components and configuring the application to ensure optimal performance. This will enable users to harness the full capabilities of the Digital Assistant and unlock its potential for enhancing user experience and driving business growth in the IoT industry.

Throughout this chapter, we will cover the following topics:

- System requirements: We will discuss the hardware and software prerequisites for running the Digital Assistant application.
- Installation process: We will provide a detailed, step-by-step guide on how to install the required dependencies, libraries.
- Configuration and setup: We will explain how to properly configure the application settings to maximize compatibility and performance.
- User interface and navigation: We will offer a comprehensive walkthrough of the Digital Assistant's user interface, highlighting its features and demonstrating how to navigate through the application.
- Basic operations and commands: We will introduce the most common functions and commands supported by the Digital Assistant, enabling users to effectively interact with the IoT Service Portal.

2.1 Install Guide

2.1.1 Hardware and Software Requirements [1]

The IoT Service web application's minimum and recommended hardware requirements are included in the following table

Table 2.1: System Requirements

Hardware Requirements	**Software Requirements**
Processor: 2.0 GHz dual-core or higher Storage: At least 10 GB of free disk space RAM: 4 GB or higher	Operating System: Linux, macOS, Windows Docker (latest version) Docker Compose (latest version) Git (latest version)

Running model-driven apps on a computer which doesn't meet the minimum requirements may lead to poor performance. Furthermore, acceptable performance may be obtained using computers with alternative hardware configurations than those described here, such as a system with a recent quad-core processor, a lower clock speed, and more RAM.

2.1.2 Network Requirements [1].

Model-driven apps are optimized for networks that include the following features:

- Bandwidth greater than 50 KBps (400 kbps).
- Latency under 150 ms.

2.1.3 Supported web browsers [1].

Any of the following web browsers running on the mentioned operating systems can run the web application:

- Microsoft Edge (latest publicly-released version) running on Windows 11, Windows 10, Window 8.1.
  - Windows 8.1 extended support will end January 10, 2023.

  - Windows 8 extended support ended January 12, 2016.
  - Windows 7 extended support ended January 14, 2020.

- Mozilla Firefox (latest publicly-released version) running on Windows 11, Windows 10, Windows 8.1.
- Google Chrome.
  - Google Chrome (latest publicly-released version) running on Windows 11, Windows 10, Windows 8.1.
  - Google Chrome (latest publicly-released version) running on the two latest publicly-release Mac OS versions,
- Apple Safari (latest publicly-released version) running on the two latest publicly-release Mac OS versions.

Git [2]

Git is a widely used distributed version control system for tracking changes in source code during software development. In this guide, we'll cover how to install Git on different operating systems, including Windows, macOS, and Linux.[2]

Installing Git on Windows

To install Git on Windows, follow these steps:

1. Download the latest Git for Windows installer from the official website: `https://git-scm.com/download/win`
2. Run the installer and follow the installation wizard, choosing the default settings or customizing them according to your preferences.
3. After the installation is complete, open the Git Bash terminal or the Windows command prompt to start using Git.

Installing Git on macOS

To install Git on macOS, follow these steps:

1. Open the Terminal application.
2. Check if Git is already installed by typing `git -version`. If Git is not installed, a prompt will appear to install the Xcode Command Line Tools, which includes Git.
3. Alternatively, download the latest Git for macOS installer from the official website: `https://git-scm.com/download/mac` and follow the installation wizard.

Installing Git on Linux

To install Git on Linux, use the package manager for your distribution. Here are the commands for some popular distributions:

- Ubuntu, Debian, and their derivatives: `sudo apt-get install git`
- Fedora: `sudo dnf install git`
- CentOS and RHEL: `sudo yum install git`
- Arch Linux and Manjaro: `sudo pacman -S git`

Docker and Docker Compose [3]

Docker is an open platform for developing, shipping, and running applications using container technology. In this guide, we'll cover how to install Docker on different operating systems, including Windows with WSL2 daemon, macOS, and Linux.[**docker_installation_guide**]

Installing Docker on Windows with WSL2 daemon [https://docs.docker.com/desktop/windows/wsl/]

To install Docker on Windows using the WSL2 daemon, follow these steps:

1. Install WSL2 by following the official Microsoft documentation: `https://docs.microsoft.com/en-us/windows/wsl/install`

2. Download Docker Desktop for Windows from the official website: `https://www.docker.com/products/docker-desktop`

3. Run the installer and follow the installation wizard. Make sure to enable the WSL2 backend during the installation.

4. After the installation is complete, open the Docker Desktop settings and select the WSL2 distribution you want to use.

Installing Docker on macOS [4]

To install Docker on macOS, follow these steps:

1. Download Docker Desktop for Mac from the official website: `https://www.docker.com/products/docker-desktop`

2. Run the installer and follow the installation wizard.

3. After the installation is complete, launch Docker Desktop from the Applications folder.

Installing Docker on Linux [5]

To install Docker on Linux, use the package manager for your distribution. Here are the commands for some popular distributions:

- Ubuntu, Debian, and their derivatives:

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

- Fedora:

```
sudo dnf install docker-ce docker-ce-cli containerd.io
```

- CentOS and RHEL:

```
sudo yum install docker-ce docker-ce-cli containerd.io
```

Verifying the Installation

To check if all the installations have been completed. please run the following commands on the terminal:

1. For git, run:

```
$ git --version
```

After issuing this command, if you received response which is something like this :

```
git version 2.37.1.windows.1
```

Then It means that you have installed the git successfully.

1. For Docker, run:

```
$ docker --version
```

After issuing this command, if you received response which is something like this :

```
Docker version 20.10.24, build 297e128
```

Then It means that you have installed the Docker successfully.

Docker Configuration

To configure Docker after installation, follow these steps:

```
# Start the Docker service
sudo service docker start

# Add your user to the 'docker' group to run Docker commands
    without sudo
sudo usermod -aG docker your_username

# Verify Docker installation
docker --version

# Test Docker by running a simple container
docker run hello-world
```

Git Configuration

After installing Git, you can configure it with the following commands:

```
# Set your Git username and email
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"

# Verify the configuration
git config --global user.name
git config --global user.email

# Set your default text editor (optional)
git config --global core.editor "nano"

# Configure Git to use colorized output
git config --global color.ui true

# Check the Git version
git --version
```

2.2 Building Web App with required services

All the below steps are going to be focused only on Windows Operating system, there might be some different configuration needed for building this Application on other operating systems. so please consult for the documentation for issuing the commands in the corresponding Operating system.

2.2.1 Cloning the IoT-Digital Assistant repository

Windows

- Open Git Bash

- Open any Terminal you like and navigate to the directory you would like to clone the repository.

- then write :

```
    $ git clone https://github.com/amhkhowaja/IoT-Digital-Assistant
       .git
```

then click Enter.

You will see a similar output like shown following : -

```
$ git clone https://github.com/amhkhowaja/IoT-Digital-Assistant
   .git
  Cloning into 'IoT-Digital-Assistant'...
  remote: Enumerating objects: 2536, done.
  remote: Counting objects: 100% (1909/1909), done.
  remote: Compressing objects: 100% (1621/1621), done.
  remote: Total 2536 (delta 254), reused 1860 (delta 225), pack
     -reused 627Receiving objects: 100% (2536/2536), 1.71 GiB |
      19.72 MiB/s
  Receiving objects: 100% (2536/2536), 1.71 GiB | 18.93 MiB/s,
     done.
  Resolving deltas: 100% (569/569), done
```

Currently All of the UI part is stored in another branch called `with_UI`. We need to checkout to that branch For that we have to use following command:

```
$ git checkout with_UI
```

You will see following output

```
Updating files: 100% (1590/1590), done.
Switched to a new branch 'with_UI'
branch 'with_UI' set up to track 'origin/with_UI'
```

you can check the branch you are in with the following command :

```
$ git branch
  main
* with_UI
```

With * we know that we are in `with_UI` branch.

2.2.2 Running All the services

After Cloning the repository, We will have to start all the services that are required to run the Application as a whole.

Services which are used in this Application are as follow:

1. Rasa Server (Required for Digital assistant)
2. Rasa Action Server (Required for Digital assistant)

3. MongoDB (Required for Digital assistant and node server)

4. Node (Required for Databases on UI)

5. Nginx (Required for UI)

Fortunately for running the application, we do not need to run each service separately. we only need to run one command and it will start all of the services simultaneously.

First we need to navigate to the repository directory and in the repository directory we have to run the following command:

```
$ docker-compose up -d --build
```

Docker compose will build all the images for different services mentioned above and run all the container. Fortunately we can access it from local environment.

This will look roughly like following :

```
$ docker-compose up -d --build
[+] Building 0.4s (8/8) FINISHED
 => [internal] load build definition from Dockerfile.mongodb
                                                                0.0s
 => => transferring dockerfile: 103B
                                                                0.0s
 => [internal] load .dockerignore
                                                                0.0s
 => => transferring context: 2B
                                                                0.0s
 => [internal] load metadata for docker.io/library/mongo:latest
                                                                0.0s
 => [1/3] FROM docker.io/library/mongo:latest
                                                                0.0s
 => [internal] load build context
                                                                0.0s
 => => transferring context: 478.71kB
                                                                0.0s
 => CACHED [2/3] WORKDIR /data/db
                                                                0.0s
 => CACHED [3/3] COPY db_data /data/db
                                                                0.0s
 => exporting to image
                                                                0.3s
 => => exporting layers
                                                                0.0s
```

```
 => => writing image sha256:b2cba122389e6bf0c31a39fd491d6f52074f29661d3639b16b552d185f8ae571
                                                                                                0.0s
 => => naming to docker.io/library/iot-digital-assistant-mongo
                                                                                                0.0s
[+] Building 13.9s (8/10)
 => [internal] load build definition from Dockerfile.actions
                                                                                                0.1s
 => => transferring dockerfile: 1.14kB
                                                                                                0.0s
 => [internal] load .dockerignore
                                                                                                0.1s
 => => transferring context: 2B
                                                                                                0.0s
 => [internal] load metadata for docker.io/library/python:3.9.12
                                                                                                4.7s
[+] Building 13.9s (9/10)
 => [internal] load build definition from Dockerfile.node
                                                                                                0.1s
[+] Building 14.0s (8/10)
 => [internal] load build definition from Dockerfile.actions
                                                                                                0.1s
 => => transferring dockerfile: 1.14kB
                                                                                                0.0s
 => [internal] load .dockerignore
                                                                                                0.1s
 => => transferring context: 2B
                                                                                                0.0s
 => [internal] load metadata for docker.io/library/python:3.9.12
                                                                                                4.7s
 => [1/6] FROM docker.io/library/python:3.9.12@sha256:4200
    eda05642eabf85b70648e17e4b433f2d7608c0130ab5dcb56cce6bc35364
                                                                                                0.0s
 => [internal] load build context
                                                                                                0.0s
 => => transferring context: 32.27kB
                                                                                                0.0s
 => CACHED [2/6] WORKDIR /app
                                                                                                0.0s
 => [3/6] COPY /actions/ /app/
                                                                                                0.0s
[+] Building 14.2s (10/10) FINISHED
 => [internal] load build definition from Dockerfile.node
                                                                                                0.1s
 => => transferring dockerfile: 630B
                                                                                                0.1s
 => [internal] load .dockerignore
```

```
      0.1s
40 => => transferring context: 2B

      0.0s
41 => [internal] load metadata for docker.io/library/node:alpine

      5.3s
42 => [1/5] FROM docker.io/library/node:alpine@sha256:4559
      bc033338938e54d0a3c2f0d7c3ad7d1d13c28c4c405b85c6b3a26f4ce5f7
                                                                  4.6s
43 => => resolve docker.io/library/node:alpine@sha256:4559
      bc033338938e54d0a3c2f0d7c3ad7d1d13c28c4c405b85c6b3a26f4ce5f7
44
45 .......
46 [+] Running 5/6
47  Network iot-digital-assistant_rasa-network  Created

       0.0s
48  Container nginx                            Started

       2.3s
49  Container mongo                            Started

       1.1s
50  Container node                             Started

       1.6s
51  Container rasa_actions                     Started

       1.4s
52  Container rasa                             Started
```

2.2.3 Running the Application

We are all done with configuring the Application you will be able to see IoT Service Portal by navigating to *IoT-Digital-Assistant/Chatbot-Widget-main* directory and open index.html

Once you open the **index.html** page, You will see the following :

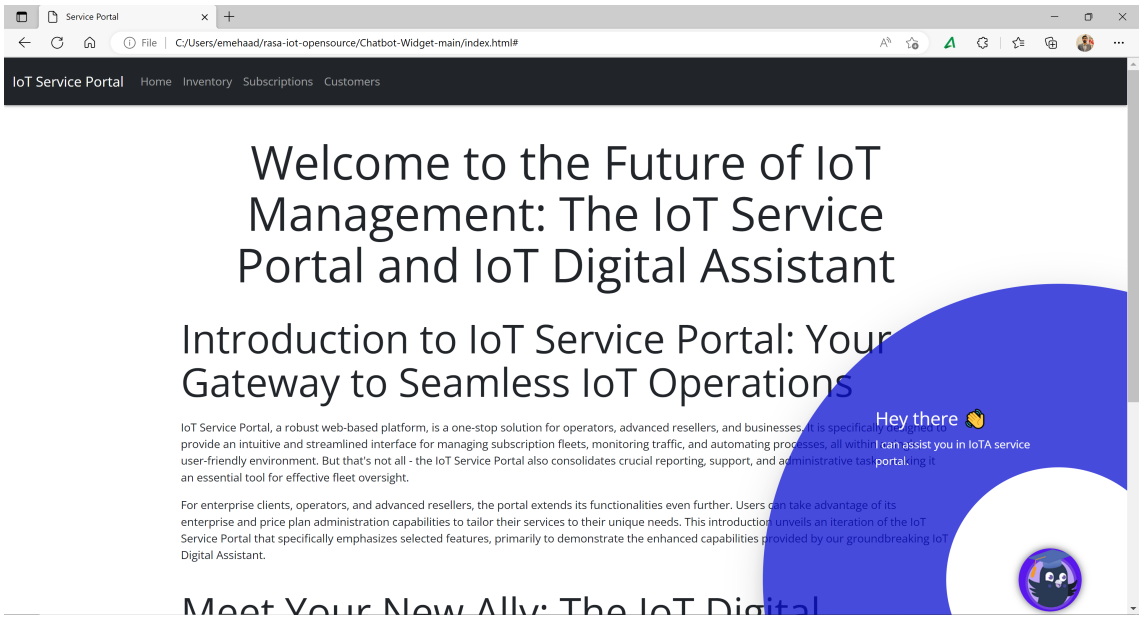


Figure 2.1: IoT Service Portal - Home Page

There are four pages in total ,

1. Home
2. Inventory
3. Subscriptions
4. Customers

Inventory, Customer, Subscription would be the data management pages whereas Home page would be the Information guide about IoT Service Portal and IoT Digital Assistant.

There is the Chatbot widget on the bottom right corner, Once clicked It would show something like that:

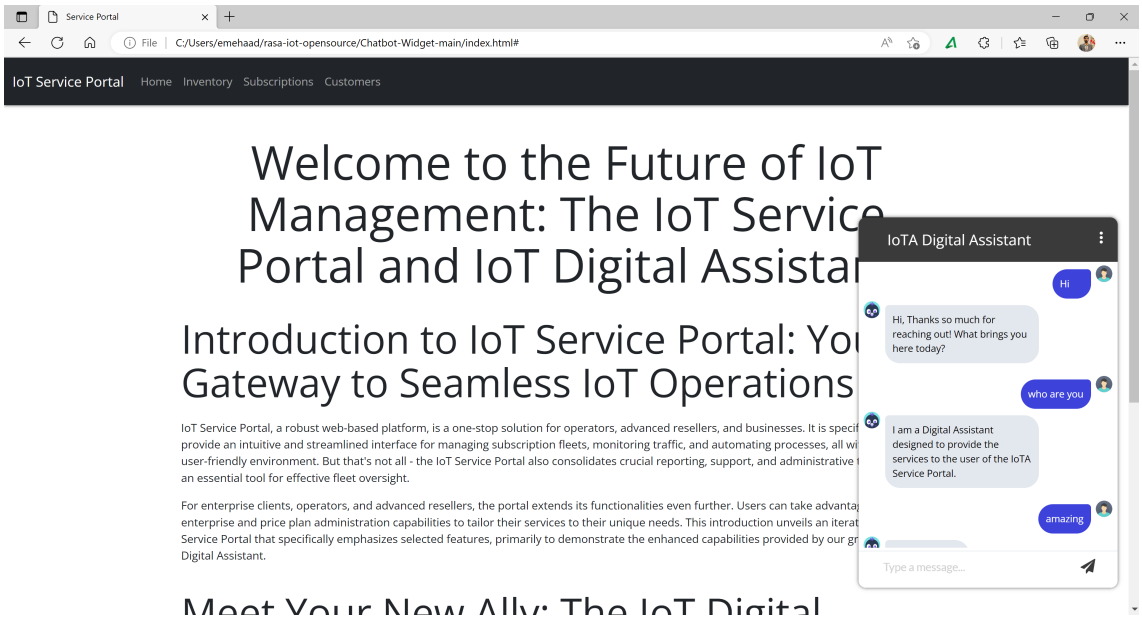


Figure 2.2: IoT Service Portal - Home Page

2.3 Features

Now We are going to look into some features that as a user you can test.

2.3.1 Chit-Chat

Please see the conversations features implemented in the series of following figures:

1. Greeting

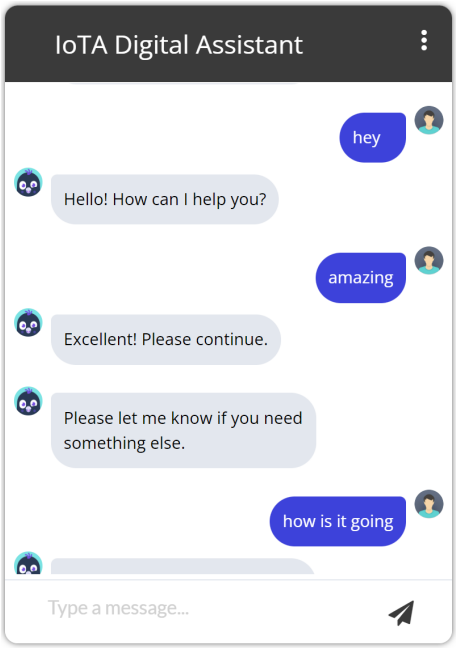


Figure 2.3: Greeting

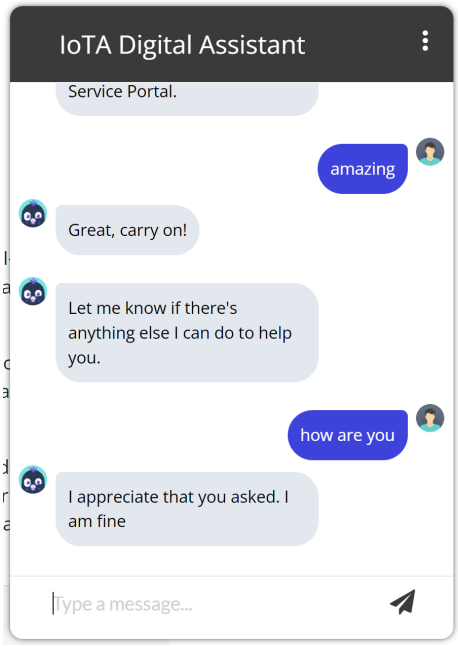


Figure 2.4: Greeting

2. About bot

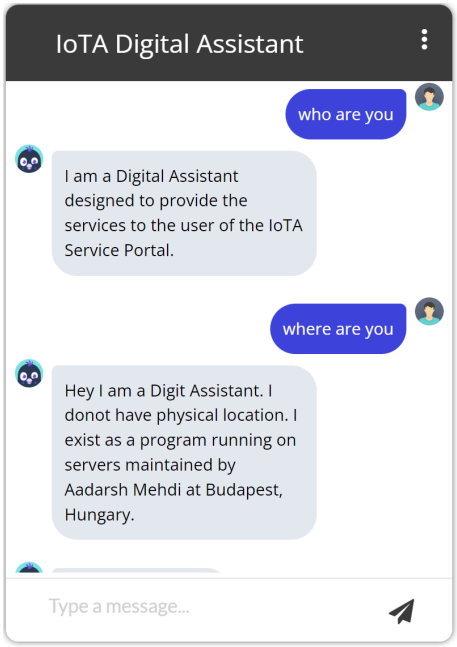


Figure 2.5: About bot

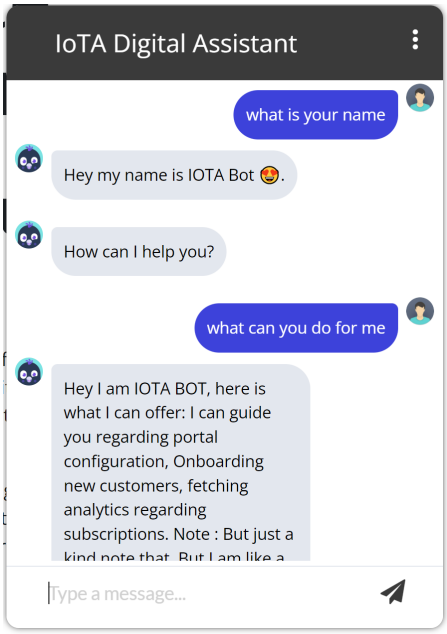


Figure 2.6: About bot

3. Telling Jokes

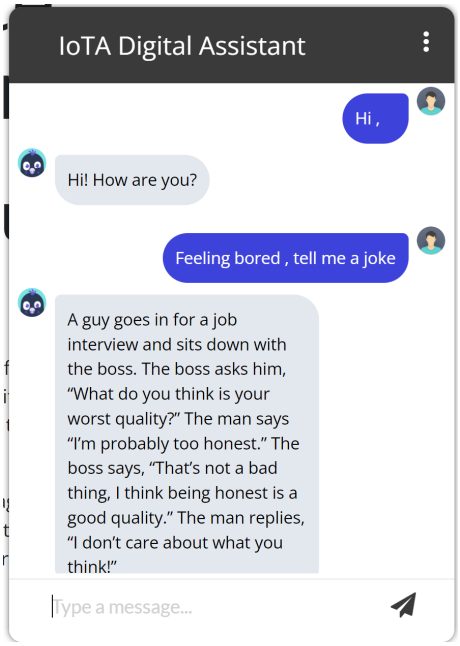


Figure 2.7: Telling joke

4. Farewell

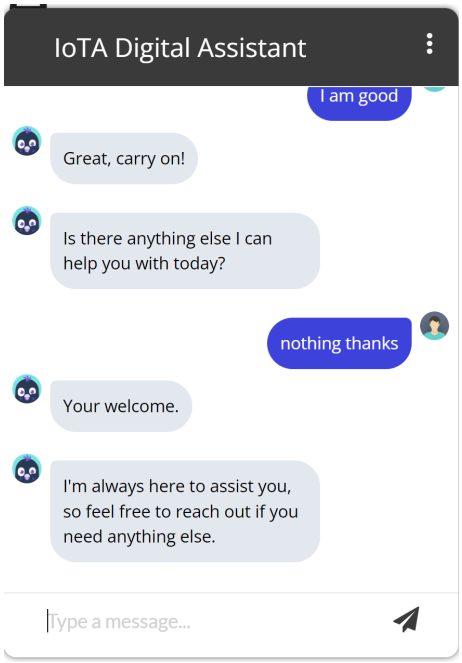


Figure 2.8: Farewell

2.3.2 Giving News Headlines

When asked from the bot about news related query. It gives back the latest headlines on BBC- News. After clicking on the link user can be redirected to the

BBC News website.

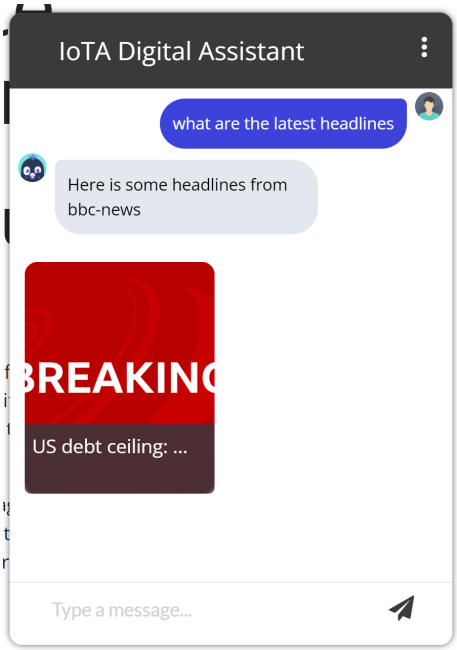


Figure 2.9: News Headline

2.3.3 Subscription Management

As you can see the subscription UI which has the table of user data.

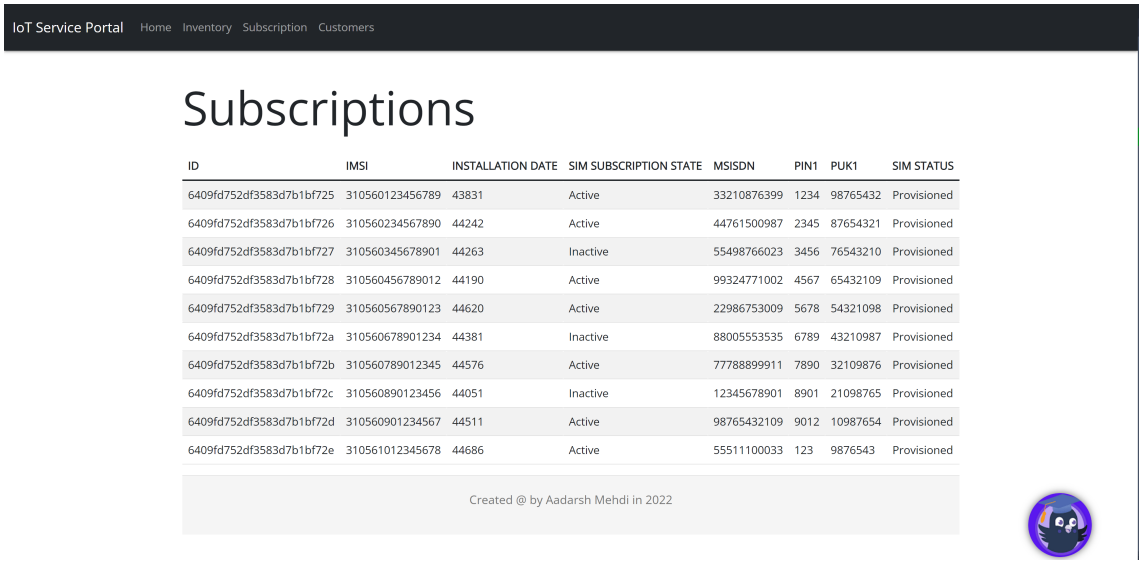


Figure 2.10: Subscriptions

Now We are going to fetch the Subscription details of the some IMSI.

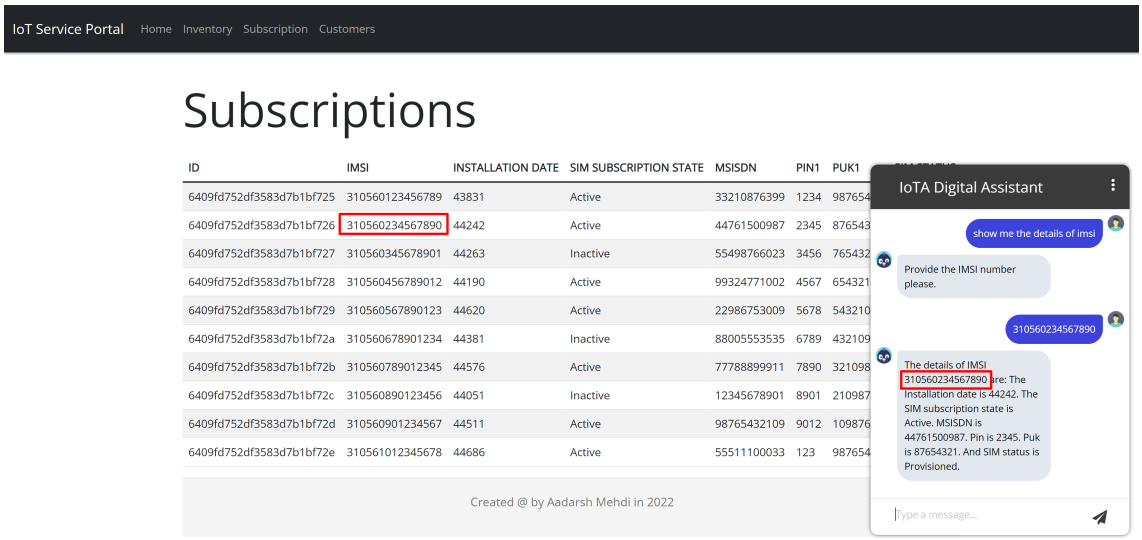


Figure 2.11: Subscriptions

2.3.4 Inventory Management

Updating the inventory

We can update the inventory with only one line of code without configuring it from the database.

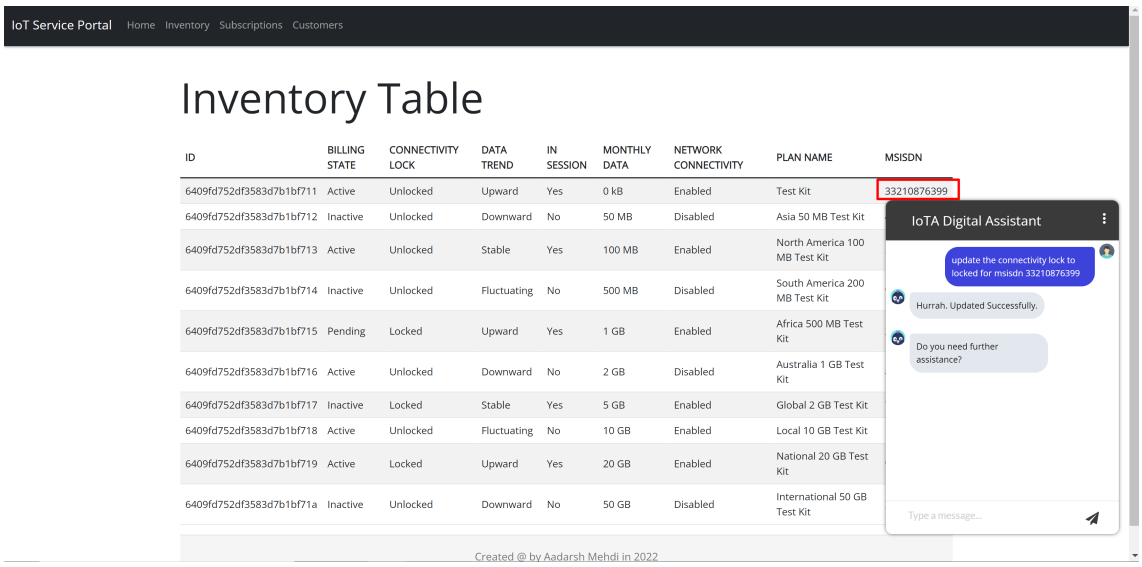


Figure 2.12: Inventory

And after reloading the page we can see that it has actually been updated.

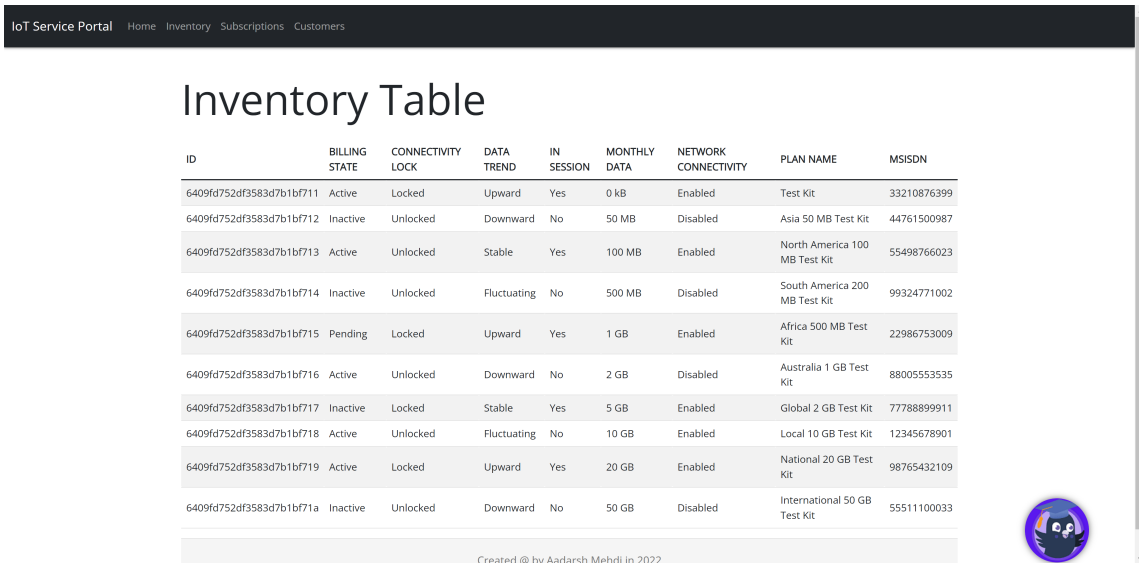


Figure 2.13: Inventory

We can do it for the other fields as well.

2.3.5 Customers Management

We can also onboard new customer in the portal We have two customer type currently:

1. Enterprise
2. advanced Reseller

Following will appear when we will load the Customer's Table.

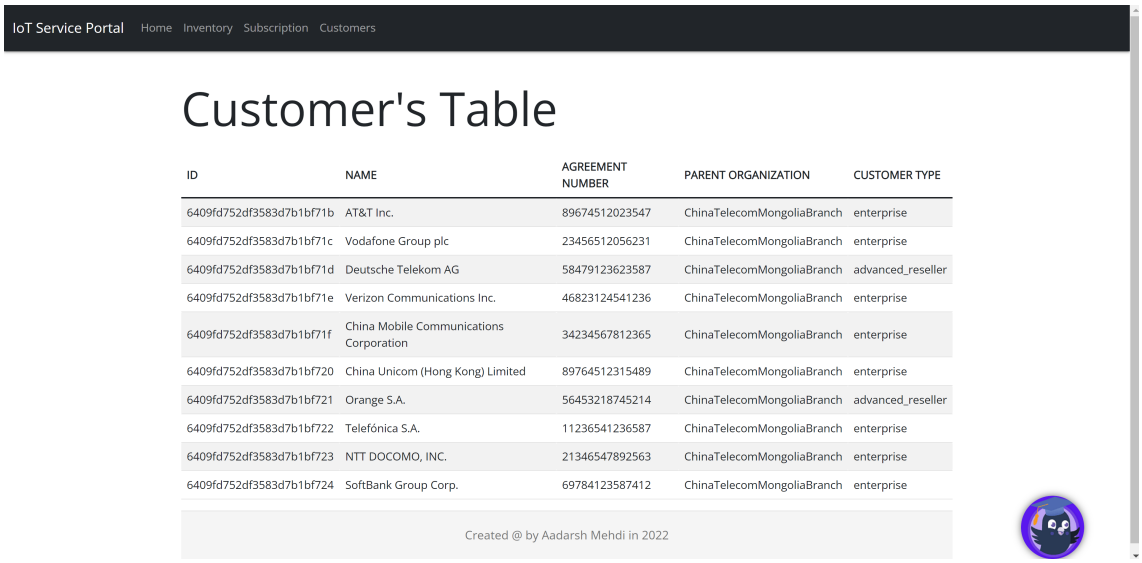


Figure 2.14: Customers

We can onboard new customer as follow:

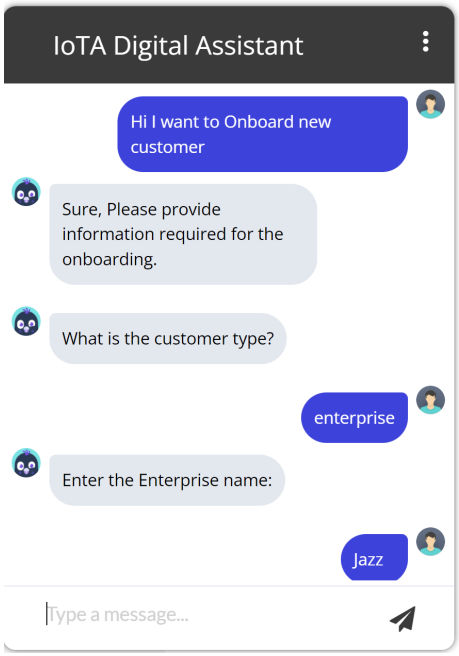


Figure 2.15: Customers

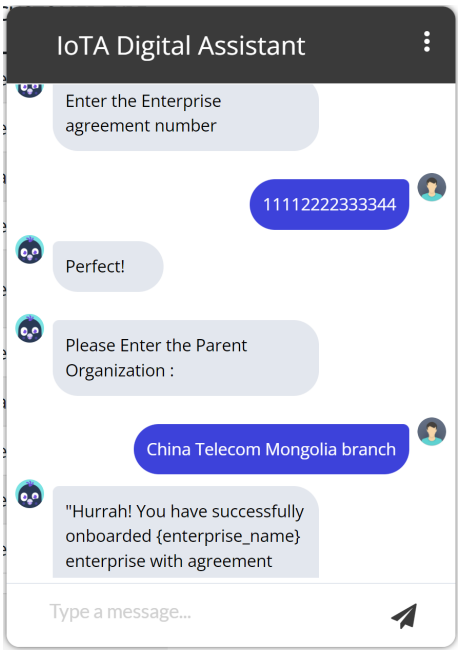


Figure 2.16: Customers

Chapter 3

Developer documentation

This chapter will cover the significant steps, the technology used, the implementation and the process of developing IoT Digital Assistant. The developer is assumed to have Intermediate knowledge of Python, RASA Platform, REST APIs, Databases (MongoDB), Data serialization documents such as JSON, Yaml, and Docker.

3.1 Cloning the Repository

For Cloning the repository Please follow the steps mentioned in Section 2.2.1

3.2 Prerequisites

3.2.1 Database

My Project of IoT Digital Assistant uses MongoDB to store the data.

A database in MongoDB is a collection of documents. On the MongoDB server, multiple databases can be created.[6]

MongoDB is a free and open source NoSQL database management system. NoSQL is a database technology that is used as an alternative to traditional relational databases. NoSQL databases are extremely useful when dealing with large amounts of distributed data. MongoDB is a tool for managing document-oriented data, as well as storing and retrieving data.[7]

More about Mongo DB : `https://www.mongodb.com/docs/`

3.2.2 Server

What happens in the server side? [8]

My IoT digital assistant relies heavily on a perpetually active server to fully utilize the project's features. At the moment, I'm employing a multitude of services, including a MongoDB instance, a RASA server, a RASA action server, and a node server. All of these services are interconnected within the same network for communication, configured through Docker-compose. Without these services running, users would face significant challenges in properly using the UI. While it is still possible to access the UI, certain features like populated database tables and a functioning chatbot won't be available. For instance, after each user message, the chatbot would display an error message. As a result, it's crucial to keep all these services running.

I will also elaborate on how these services can be started individually in this section.

3.2.3 RASA

RASA Server [9]

Rasa is a tool to help you build task oriented dialogue systems. When we say "task oriented" we mean that the user wants to accomplish something. When we say "dialogue system" we're talking about automated systems in a two way conversation. That means that we're talking about assistants that can talk back to the user, to help them achieve the task they're interested in.

The core of building a Rasa assistant is providing examples that your system learns from. That way, Rasa can attempt to generalise patterns in your data.

In this Project I have used RASA Open Source versions as follow:

```
$ rasa --version
   Rasa Version      :         3.4.4
   Minimum Compatible Version: 3.0.0
   Rasa SDK Version  :         3.4.0
   Python Version    :         3.9.12
   Operating System  :         Windows-10-10.0.19045-SP0
   Python Path       :         C:\Users\emehaad\Anaconda3\envs
      \rasa-iot\python.exe
```

You can learn the RASA documentation for the developers (Specific to RASA opensource) on this page : `https://rasa.com/docs/rasa/` `https://learning.rasa.com/conversational-ai-with-rasa/introduction-to-rasa/`

RASA Action Server [10]

A Rasa action server runs custom actions for a Rasa Open Source conversational assistant.

When your assistant predicts a custom action, the Rasa server sends a POST request to the action server with a json payload including the name of the predicted action, the conversation ID, the contents of the tracker and the contents of the domain.

When the action server finishes running a custom action, it returns a json payload of responses and events. See the API spec for details about the request and response payloads.

The Rasa server then returns the responses to the user and adds the events to the conversation tracker.

We will need to understand some concepts about Rasa action server to get started with getting our hands dirty in the code.

- **RASA SDK (Python) :**

Rasa SDK is a Python SDK for running custom actions. Besides implementing the required APIs, it offers methods for interacting with the conversation tracker and composing events and responses. If you don't yet have an action server and don't need it to be in a language other than Python, using the Rasa SDK will be the easiest way to get started.

3.2.4 Node [11]

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It is designed to build scalable network applications, and it is known for its efficiency and event-driven, non-blocking

I/O model, which makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices
n the framework of my project for the UI part, I am utilizing Node.js as the key element in running Mongoose for fetching the data from Mongo server running and provide the express app to be used in the client side for populating the data from MongoDB. It actually serves the purpose of bridge between Client Side UI and Mongo DB Server.

For installation of node you can refer to this link: `https://nodejs.dev/en/learn/how-to-install-nodejs/`

I have used two packages from npm(Node Package Manager) library for this project.

- **Express**
- **Mongoose**

Express [12]

Express is a popular unopinionated web framework, written in JavaScript and hosted within the Node.js runtime environment. This module explains some of the key benefits of the framework, how to set up your development environment and how to perform common web development and deployment tasks.

For Installation after installing node:

```
1    $ npm install express
```

Mongoose [13]

Mongoose is a Node.js-based Object Data Modeling (ODM) library for MongoDB. It is akin to an Object Relational Mapper (ORM) such as SQLAlchemy for traditional SQL databases. The problem that Mongoose aims to solve is allowing developers to enforce a specific schema at the application layer. In addition to enforcing a schema, Mongoose also offers a variety of hooks, model validation, and other features aimed at making it easier to work with MongoDB. For Installation after installing node:

```
1    $ npm install mongoose
```

3.2.5 MongoDB

The Digital Assistant utilizes MongoDB as a service, providing durable data storage integral to its custom actions based on the RASA framework. Interaction with the MongoDB server is initiated in numerous custom actions, in which data from the database is acquired, manipulated, and used to generate appropriate responses to the user's queries.

At present, our implementation operates using a single database, "IOTA", which contains four distinct collections. Further discussion and elaboration on these collections will be provided in subsequent sections of this document.

3.3 Build

3.3.1 Docker-Compose

I've employed docker-compose to assemble all services associated with the IoT Digital Assistant. Four key services have been configured in this system, and although they've been mentioned previously, I'll reiterate them here for clarity:

- RASA Server
- RASA Action Server
- MongoDB
- Node.js

These services operate within a shared network , mainly due to interdependencies among some of them.

Take, for example, Node.js; it requires MongoDB to be operational. Likewise, RASA leverages a tracker store to preserve conversations, which necessitates a functional MongoDB server, thereby creating a dependency. Similarly, the RASA Action Server, while it can be constructed and executed without an immediate reliance on MongoDB, will not be fully operational if an endpoint for a specific custom action - which depends on MongoDB - needs to be activated by the RASA server.

Therefore, even though the RASA Action Server could technically build and run without the MongoDB dependency, it would fall short in performance when a

custom action that depends on MongoDB is invoked. That's why I refer to it as an indirect dependency.

```
version: '3.8'
services:
# MONGO
  mongo:
    restart: always
    build:
      context: .
      dockerfile: Dockerfile.mongodb
    container_name: mongo
    ports:
      - "27017:27017"
    volumes:
      - ./init-mongo.js:/docker-entrypoint-initdb.d/init-mongo.js:
          ro
      - ./mongo-volume:/data/db
    networks:
    - rasa-network
# RASA Actions Server
  rasa_actions:
    build:
      context: .
      dockerfile: Dockerfile.actions
    container_name: rasa_actions
    ports:
      - 5055:5055
    volumes:
      - ./actions:/app/actions
    depends_on:
      - mongo
    environment:
      RASA_ACTION_ENDPOINT: http://localhost:5055/webhook
    command: ["rasa","run", "actions"]
    networks:
      - rasa-network
# NODE
  node:
```

```
    restart: always
    container_name: node
    working_dir: /usr/src/app
    build:
      context: .
      dockerfile: Dockerfile.node
    ports:
      - 3000:3000
    volumes:
      - ./Chatbot-Widget-main:/app/Chatbot-Widget-main/static/js
    depends_on:
      - mongo
    environment:
      MONGO_URI: mongodb://mongo:27017/IOTA
    networks:
      - rasa-network
# RASA Server
  rasa:
    restart: always
    build:
      context: .
      dockerfile: Dockerfile.rasa
    container_name: rasa
    ports:
      - 5005:5005
    command: ["rasa", "run", "--enable-api", "--cors", "*"]
    depends_on:
      - rasa_actions
      - mongo
    volumes:
      - ./data:/app/data
      - ./models:/app/models
      - ./config.yml:/app/config.yml
      - ./domain.yml:/app/domain.yml
      - ./endpoints_for_container.yml:/app/endpoints.yml
    networks:
      - rasa-network
volumes:
  db_data:
```

```
75
76 networks:
77   rasa-network:
```

Code 3.1: Docker Compose file of the Project

So As you can see in the configuration that every service is running on the configured port and each service has certain persistent volume. For more information on the Docker you can actually read the following article: `https://docs.docker.com/get-started/`

- RASA Action Server is running on Port : 5055
- RASA Server is running on Port : 5005
- Mongo Server is running on Port : 27017
- Node Server is running on Port : 3000

You can see that I have provided the *Dockerfile.rasa, Dockerfile.mongo, Dockerfile.actions, Dockerfile.node* for the corresponding services . It is used when The services are build : using the following command :

```
1     $ docker up -d --build
```

when we run docker up command it runs the services in the docker-compose.yaml file in that current directory and -d option is used to run it in the background whereas –build option actually restart the build from the Dockerfile.

3.3.2 Dockerfile

Each service referenced in the docker-compose.yml file is built utilizing individual Dockerfiles. In this process, the necessary dependencies are copied from the repository, installed, and subsequently, the services are activated.

The result is the creation of Docker containers, which are purposed to encapsulate and operate all services crucial to the IoT Chatbot. Docker technology brings immense utility and convenience to the user experience. It obviates the need for users to pre-install all dependencies, therefore simplifying the utilization process.

Not only does it streamline operations, but it also offers a significant degree of flexibility. Users can easily engage with the application without needing to navigate

through complex setup procedures, making Docker a profoundly effective solution for such tasks.

One of the example of Dockerfile I can share :

```
# Use the official Node.js image as the base image
FROM node:alpine

# Set the working directory inside the container
WORKDIR usr/src/app

# Copy the package.json and package-lock.json files to the
    container
COPY Chatbot-Widget-main/static/js/package*.json ./

# Install the application dependencies
RUN npm ci

# Copy the rest of the application code to the container
COPY Chatbot-Widget-main/static/js/ ./

# Start the application
CMD [ "npm", "start" ]
# Expose the port your application is listening on
EXPOSE 3000
```

Code 3.2: Dockerfile.node

3.4 External APIs Used

n the IoT Digital Assistant, I have integrated several external HTTP REST APIs into the custom actions to add a layer of interactivity and functionality to the Chatbot. Predominantly, the GET method is used in these APIs, enabling data retrieval based on user queries. The received responses are then processed and formatted to generate appropriate responses to user queries.

Here are the notable APIs that I have employed to enhance the Chatbot's capabilities:

1. **BBC API**: This News API brings the ability to pull the latest news headlines, offering users up-to-date information.

2. **Google's "People Also Ask" API**: This feature provides a wealth of commonly asked questions and answers, enriching the conversation by presenting frequently sought information related to a given topic.

Google's "People Also Ask" API: This feature provides a wealth of commonly asked questions and answers, enriching the conversation by presenting frequently sought information related to a given topic.

These APIs not only enrich the user's interaction but also extend the utility of the Chatbot, making it a more versatile tool for information retrieval.

3.4.1 BBC News API [14]

This is how you can Send a GET Request to the BBC News API:

```
        query_params = {
            "source": "bbc-news",
            "sortBy": "top",
            "apiKey": "4dbc17e007ab436fb66416009dfb59a8"
            }
        main_url = " https://newsapi.org/v1/articles"
        # fetching data in json format
        try:
            req = requests.get(main_url, params=query_params)
        except (requests.exceptions.RequestException, KeyError) as
           e:
            dispatcher.utter_message(text="Sorry! unable to connect
                BBC news. Please try again.")
```

Code 3.3: GET request BBC News API

So when the user asks the bot to know about the latest news or current affairs then the bot will trigger this API to get the latest news.

3.4.2 Google's People Also Asks API [15]

This is how you can Send a GET Request to the BBC News API: What you have to do is install the people also ask package from pip by running this code:

```
$ pip install people-also-ask
```

Code 3.4: Installing people also ask package from pip

And after installing it you can import that in the python

```
import people-also-ask as paa
```

Code 3.5: importing the package

and then you can directly use it

```
# Example Question
query = "What is MSISDN"
# Response will be saved in response variable.
response = paa.get_answer(query)
```

Code 3.6: GET request Google's People's Also Asks API

You can read the full capabilities in this URL : `https://pypi.org/project/people-also-ask/`

3.5 Software Architecture

In this segment, we will delve into the intricate details of the software design that underpins the IoT Digital Assistant. We will examine the RASA configuration specifically tailored for the chatbot and scrutinize the diagram showcasing the standard architecture of a RASA chatbot.

Additionally, we will investigate the structural augmentation implemented on top of the default chatbot. This includes examining the integration of external APIs and exploring how the database interacts with the custom action code. This comprehensive examination will provide a thorough understanding of the working principles and design aspects of the IoT Digital Assistant.

3.5.1 Business Logic for the IoT Digital Assistant

At the inception of the Digital Assistant Project, I meticulously strategized to formulate a business framework for the chatbot, particularly focusing on its interaction with user data and documentation.

As regards the IoT cellular industry, we possess a considerable amount of user data and data analytics stored within our database, all of which are displayed on the IoT Service Portal. Ordinarily, this portal offers a suite of tools for manipulating the database. However, for the purpose of this thesis, our focus is not directed towards the IoT service portal per se, but rather the manner in which the Service Portal interacts with the IoT digital assistant. The IoT Digital Assistant is designed to interact with the database, equipped with functionalities to update, retrieve, and insert data, but restricted to permitted intents only.

Our first step involves data preprocessing utilizing tools like NLTK and Spacy. The NLP Engine is tasked with understanding the Natural Language Understanding (NLU) component, which would then result in Intent Classification and Entity Recognition. Subsequently, the chatbot will cross-reference these parameters to determine the relevant documents matching the identified NLU. Upon locating an appropriate document, the chatbot will respond to the user by presenting the found document.

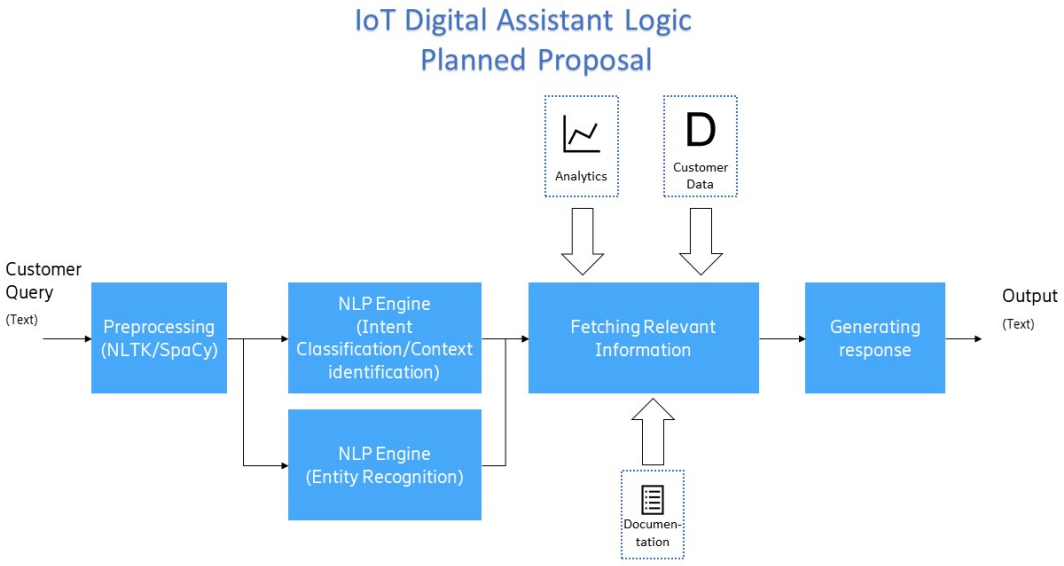


Figure 3.1: Planned Proposal of IoT Digital Assistant

3.5.2 RASA

Configuration

The RASA configuration serves as a systematic process, or pipeline, for training the RASA chatbot, focusing primarily on two key components: Natural Language Understanding (NLU) and the Dialogue Manager. This configuration sequence effectively tailors the chatbot to accurately interpret user input and manage conversations based on the data it has been trained on, thus fostering more sophisticated and efficient user interactions.

Following Figure will help you understand the training process.

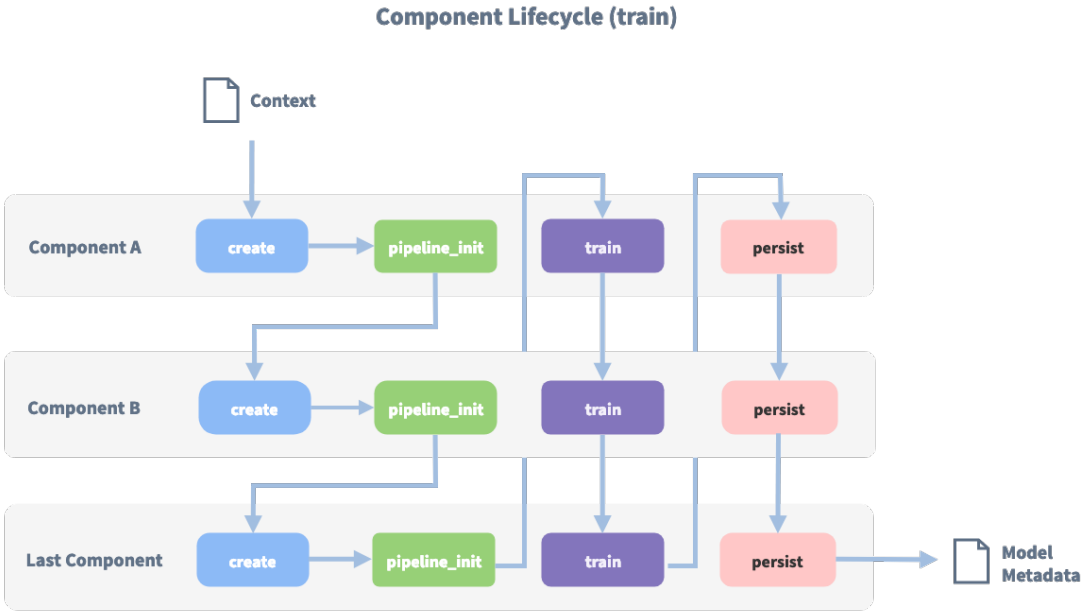


Figure 3.2: RASA Pipeline Training demonstration

It is usually stored in the yaml file. Here is the Config file for my chatbot :

```
  pipeline:
- name: SpacyNLP
  model: en_core_web_lg
  case_sensitive: false
- name: SpacyTokenizer
- name: SpacyFeaturizer
```

```
7   - name: RegexFeaturizer
8   - name: LexicalSyntacticFeaturizer
9   - name: CountVectorsFeaturizer
10  - name: CountVectorsFeaturizer
11    analyzer: char_wb
12    min_ngram: 1
13    max_ngram: 4
14  - name: DIETClassifier
15    epochs: 100
16  - name: EntitySynonymMapper
17  - name: ResponseSelector
18    epochs: 100
```

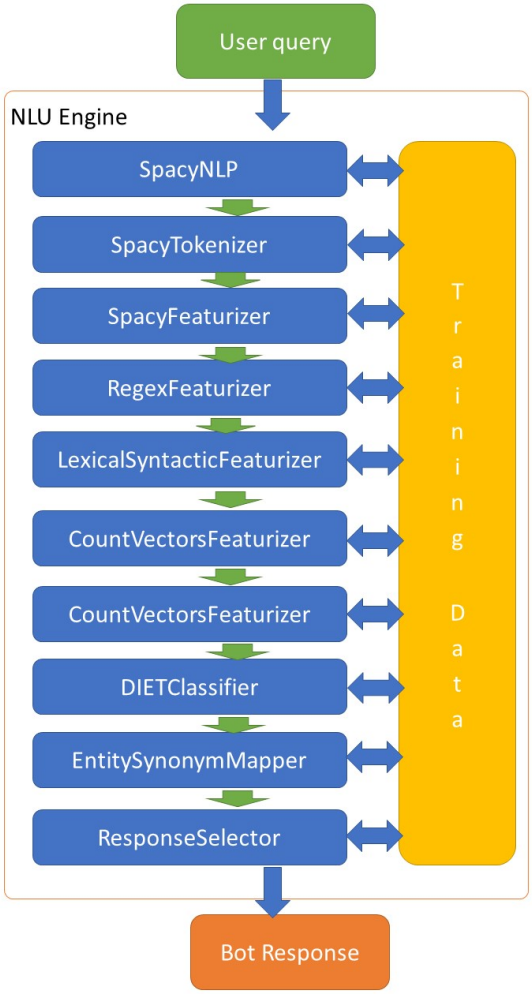


Figure 3.3: RASA Pipeline for IoT Digital Assistant

PIPELINE Explained:

The RASA configuration pipeline involves several components, each contributing to

the comprehensive understanding and processing of user messages.

Initially, we use SpacyNLP as our primary component, which employs the model specified within the pipeline, in this case, 'en-core-web-lg'. This is a large-scale Natural Language Processing (NLP) model for English, trained on an extensive dataset. It comprises numerous internal components such as a tokenizer, parser, named entity recognizer (NER), entity ruler, lemmatizer, part of speech tagger, and vectorizer. This pipe is critical for processing user messages into parameters that can be fed into machine learning algorithms.

Subsequently, we utilize the SpacyTokenizer, which breaks down the text into tokens using the SpacyNLP model. This is followed by the SpacyFeaturizer pipe, tasked with generating word embeddings for the text. In essence, it creates vector representations for each word to understand the contextual meaning of the word in different sentences.

Next in line is the RegexFeaturizer, a component in RASA's configuration that identifies entities based on regular expressions provided in the NLU data. This is followed by the LexicalSyntacticFeaturizer, which extracts entities based on sentence structure, such as case and word position.

Then comes the CountVectorFeaturizer, which forms a bag of words and n-grams to understand the typical combinations of words that appear together. We've defined minimum and maximum n-gram values, implying that words can be featurized independently or in combinations of up to four words.

Following this, the DietClassifier comes into play, a key element for Intent Classification and Entity Recognition. The preceding steps essentially prepared the data (or featurized it) for feeding into the NLU Model. The DietClassifier trains the NLU model with these prepared features. It's an apt choice for this chatbot due to its unique ability to recognize roles and groups within entities - a feature limited to a select few NLU classifier models. It is using 100 epochs for training which means that it is iterating training data for 100 cycles of different contextual parameters.

Following this, we employ the EntitySynonymMapper, which maps all synonyms defined in the nlu.yml file with the incoming user query.

Finally, we utilize a Response Selector pipe. This component retrieves a response from a set of candidate responses, along with predicted values and confidence scores,

and packages these into a dictionary format. This comprehensive pipeline enables our chatbot to effectively interpret and respond to user queries.

Understanding RASA architecture [16]

The diagram below provides an overview of the Rasa architecture. The two primary components are Natural Language Understanding (NLU) and dialogue management.

NLU is the part that handles intent classification, entity extraction, and response retrieval. It's shown below as the NLU Pipeline because it processes user utterances using an NLU model that is generated by the trained pipeline.

The dialogue management component decides the next action in a conversation based on the context. This is displayed as the Dialogue Policies in the diagram.

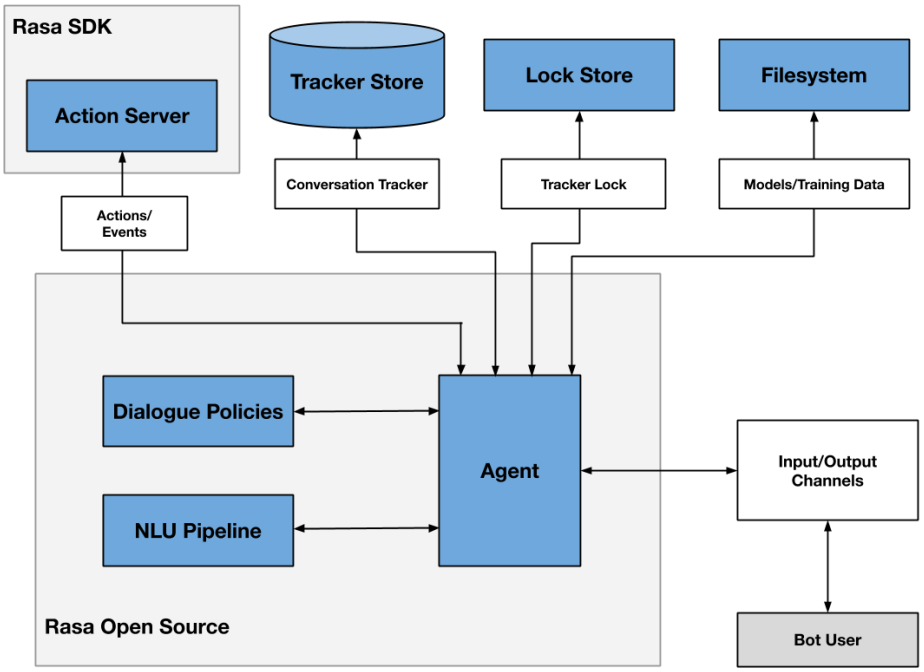


Figure 3.4: Rasa architecture

- We utilize a Tracker Store to keep a record of conversational history in memory. Moreover, this history from the Tracker Store is also preserved in the MongoDB collection of conversations.
- We are using Action Server for writing the custom programmed Actions using RASA SDK.

- After we train the Chatbot all of our models will be stored into models directory, which will be loaded by agent when loading the chatbot for the user.
- Tracker lock is out of scope from this Digital Assistant

If you want to learn more about RASA Architecture. Please go through this article: `https://rasa.com/docs/rasa/arch-overview/`

Rasa Project Directory Structure

It is specific to Rasa Directory Structure.

This is how the directories are organized for creating the conversational agent.

```
- IoT Digital Assistant/
    - data/
        - nlu.yml
        - stories.yml
        - rules.yml
    - tests/
        - conversation_tests.yml
    - models/
    - actions/
        - actions.py
    - config.yml
    - domain.yml
    - credentials.yml
    - endpoints.yml
```

Explanation of the files and folders:

1. **data/:** This directory contains all the training data for the Rasa models. This includes:
    - **nlu.yml** for NLU training data.
    - **stories.yml** for training stories which represent conversational flows.
    - **rules.yml** for defining rules that should always be followed by the chatbot.
2. **tests/:** This directory holds test stories which can be used to evaluate the performance of the trained model.

3. **models/:** This is where the trained model files are saved.

4. **actions/:** This directory contains the custom action code in ***actions.py***.

5. **config.yml:** This file contains the model configuration and defines the pipeline for processing and classifying the text.

6. **domain.yml:** This file defines the chatbot's universe, including intents, entities, slots, responses, and actions.

7. **credentials.yml:** This file contains necessary credentials for connecting with different channels like Facebook, Slack, or your custom application.

8. **endpoints.yml:** This file contains the different endpoints for your custom actions, model server, tracker store, and event broker(which I have not added in this project).

Training Data

You can read about the training data format on this URL : `https://rasa.com/docs/rasa/training-data-format` There is small example of the training data in a single file:

```
version: "3.1"

nlu:
- intent: greet
  examples: |
    - Hey
    - Hi
    - hey there [Sara](name)

- intent: faq/language
  examples: |
    - What language do you speak?
    - Do you only handle english?
```

```
14      - Please tell me some [packages]{"entity":"inventory"
            ,"value":"plan_name"} for the sim?
15
16  stories:
17  - story: greet and faq
18    steps:
19    - intent: greet
20    - action: utter_greet
21    - intent: faq
22    - action: utter_faq
23
24  rules:
25  - rule: Greet user
26    steps:
27    - intent: greet
28    - action: utter_greet
```

1. **nlu:** we have example of sentences categorized into certain intents . Each sentence may or may not have entities , if there are entities we can see it is denoted as [entity-value](entity-name).
   We need to understand few thing about formatting of entities in the sentence.

   - entities can be denoted as [entity-value](entity-name)
   - entities sometimes has some synonyms to be used for processing like plan name and package is the same thing and both are synonymous but the real entity value is inventory. so we can denote it as : [entity-detected]"entity":"[ENTITY NAME]","value":"[ENTITY SYNONYM]" so the entity synonym will be used to trigger the dialogue manager for the response.
   - It can have some other parameters as well like Roles and Groups: which is going to be denoted as [entity-detected]"entity":"[ENTITY NAME]","value":"[ENTITY SYNONYM]", "role": "[ENTITY ROLE]" . Now entity role in the entity object which gives more information about

the entity in a way that if there are more than two same entities then entity role will specify its meaning .

We will see in the example: For instance We have a sentence "I want to update the sim number from 123 to 223" now consider 123 and 223 are two entities and both entities are classified as mobile-number but without roles we would not know which one has to be changed to which number . Let say we have two roles: fetch-from and update-to in the 123 and 223 mobile-number entities respectively. now we would know which one will be modified to which one.

2. **stories: [17]** A story is a representation of a conversation between a user and an AI assistant, converted into a specific format where user inputs are expressed as intents (and entities when necessary), while the assistant's responses and actions are expressed as action names.

3. **rules: [18]** Rules are a type of training data used to train the assistant's dialogue management model. Rules describe short pieces of conversations that should always follow the same path.

Domain

The domain defines the universe in which your assistant operates. It specifies the intents, entities, slots, responses, forms, and actions your bot should know about. It also defines a configuration for conversation sessions.

Following is example of it but you can go through my domain file to understand the domain based domain.yml file. Also you can read the documentation : `https://rasa.com/docs/rasa/domain`

```
version: "3.1"

intents:
  - affirm
  - deny
  - greet
  - thankyou
  - goodbye
  - search_concerts
  - search_venues
  - compare_reviews
  - bot_challenge
  - nlu_fallback
  - how_to_get_started

```

```
entities:
  - name

slots:
  concerts:
    type: list
    influence_conversation: false
    mappings:
    - type: custom
  venues:
    type: list
    influence_conversation: false
    mappings:
    - type: custom
  likes_music:
    type: bool
    influence_conversation: true
    mappings:
    - type: custom

responses:
  utter_greet:
    - text: "Hey there!"
  utter_goodbye:
    - text: "Goodbye :("
  utter_default:
    - text: "Sorry, I didn't get that, can you rephrase?"
  utter_youarewelcome:
    - text: "You're very welcome."
  utter_iamabot:
    - text: "I am a bot, powered by Rasa."
  utter_get_started:
    - text: "I can help you find concerts and venues. Do you like music?"
  utter_awesome:
    - text: "Awesome! You can ask me things like \"Find me some concerts\" or \"What's a good
        venue\""

actions:
  - action_search_concerts
  - action_search_venues
  - action_show_concert_reviews
  - action_show_venue_reviews
  - action_set_music_preference

session_config:
  session_expiration_time: 60  # value in minutes
  carry_over_slots_to_new_session: true
```

3.5.3 Database Architecture

The MongoDB server, set up as a service within docker-compose, utilizes a persistent volume for storing the database found in the mongo-volume directory. At present, this project employs a single database called IOTA, which contains four collections: customers, inventory, subscription_details, and conversations. In the following section, I will outline the schema for these collections.

Refer to the subsequent illustrations for a clearer comprehension of the inventory schema, which also includes a few example entries.

- Database name : **IOTA**

  Collections:

  - inventory
  - subscription_details
  - customers
  - conversations

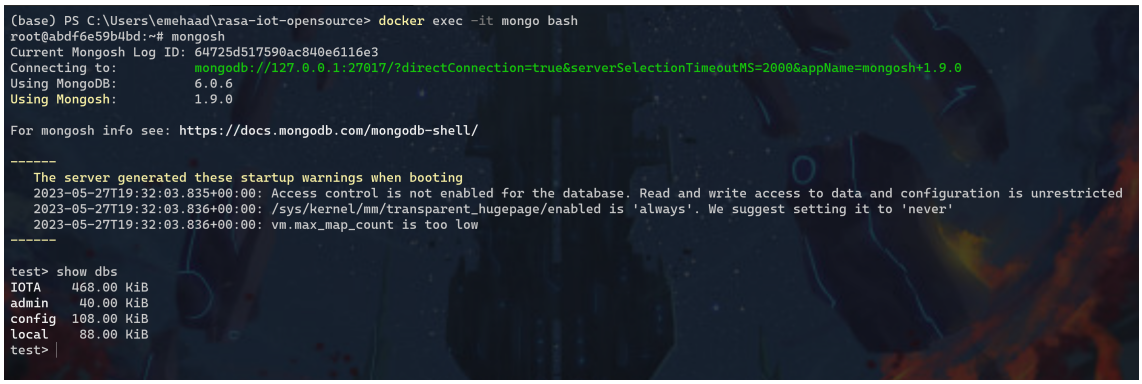


Figure 3.5: Mongosh Instance inside the docker container mongo

Inventory

Following is one of the data in the Inventory collection:

```
test> use IOTA
switched to db IOTA
IOTA> db.inventory.findOne()
{
  _id: ObjectId("6409fd752df3583d7b1bf711"),
  billing_state: 'Active',
  connectivity_lock: 'Locked',
  data_trend: 'Upward',
  in_session: 'Yes',
  monthly_data: '0 kB',
  network_connectivity: 'Enabled',
  plan_name: 'Europe 25 MB Test Kit',
  msisdn: Long("33210876399")
}
IOTA>
```

Figure 3.6: One document in the inventory collections

In this collection **msisdn, _id** are the only with the unique values.

Customers

Following is one of the data in the Customers collection:

```
IOTA> db.customers.findOne()
{
  _id: ObjectId("6409fd752df3583d7b1bf71b"),
  name: 'AT&T Inc.',
  agreement_number: Long("89674512023547"),
  parent_organization: 'ChinaTelecomMongoliaBranch',
  customer_type: 'enterprise'
}
```

Figure 3.7: One document in the customers collections

In this collection **_id** is the only with the unique values.

Subscription Details

Following is one of the data in the Customers collection:

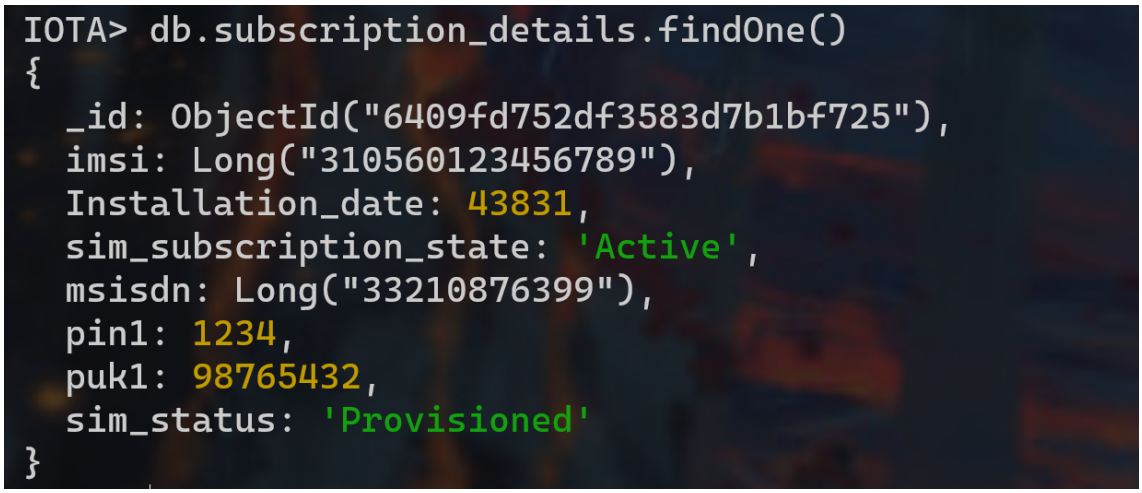


Figure 3.8: One document in the subscription_details collections

In this collection **_id** is the only with the unique values.

Conversations - From Tracker Store

Following is one of the data in the Customers collection:

```
  {
_id: ObjectId("6414a105fd849423480e38f8"),
sender_id: '05aa6d00d8d949939c1fd8f37b9768f1',
active_loop: {},
events: [
  {
    event: 'action',
    timestamp: 1679073541.9695835,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' },
    name: 'action_session_start',
    policy: null,
    confidence: 1,
    action_text: null,
    hide_rule_turn: false
  },
  {
    event: 'session_started',
    timestamp: 1679073541.9695835,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' }
  },
  {
    event: 'action',
    timestamp: 1679073541.9695835,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' },
    name: 'action_listen',
    policy: null,
    confidence: null,
    action_text: null,
    hide_rule_turn: false
  },
  {
    event: 'user',
    timestamp: 1679073542.2656803,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' },
    text: 'hey whats up',
    parse_data: {
```

```
      intent: { name: 'news_fetch', confidence: 0.9524834156036377 },
      entities: [],
      text: 'hey whats up',
      message_id: 'cc567e17eb9a496f99a60a6f37243c17',
      metadata: {},
      text_tokens: [ [ 0, 3 ], [ 4, 8 ], [ 8, 9 ], [ 10, 12 ] ],
      intent_ranking: [
        { name: 'news_fetch', confidence: 0.9524834156036377 },
        { name: 'ask_hru', confidence: 0.046181075274944305 },
        { name: 'greet', confidence: 0.00039312164881266654 },
        { name: 'bot_where', confidence: 0.0003473285469226539 },
        { name: 'thank', confidence: 0.00025358531274832785 },
        {
          name: 'ask_bot_question',
          confidence: 0.00008585600153310224
        },
        { name: 'fetch', confidence: 0.00006705214036628604 },
        { name: 'data_enter', confidence: 0.00004517769775702618 },
        { name: 'mood_unhappy', confidence: 0.00004127583815716207 },
        { name: 'unhappy_path', confidence: 0.00003275157723692246 }
      ],
      response_selector: {
        all_retrieval_intents: [],
        default: {
          response: {
            responses: null,
            confidence: 0,
            intent_response_key: null,
            utter_action: 'utter_None'
          },
          ranking: []
        }
      }
    },
    input_channel: 'cmdline',
    message_id: 'cc567e17eb9a496f99a60a6f37243c17'
  },
  {
    event: 'user_featurization',
    timestamp: 1679073546.7560768,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' },
    use_text_for_featurization: false
  },
  {
    event: 'action',
    timestamp: 1679073546.7560768,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' },
    name: 'action_news_fetch',
    policy: 'TEDPolicy',
    confidence: 0.9699466824531555,
    action_text: null,
    hide_rule_turn: false
  },
  {
    event: 'action',
    timestamp: 1679073546.7641299,
    metadata: { model_id: 'f6e41d52fe4846c1bb9c2c0622a32809' },
    name: 'action_listen',
    policy: 'TEDPolicy',
    confidence: 0.9995286464691162,
    action_text: null,
    hide_rule_turn: false
  }
],
followup_action: null,
latest_action: { action_name: 'action_listen' },
```

```
  latest_action_name: 'action_listen',
  latest_event_time: 1679073546.7641299,
  latest_input_channel: 'cmdline',
  latest_message: {
    intent: { name: 'news_fetch', confidence: 0.9524834156036377 },
    entities: [],
    text: 'hey whats up',
    message_id: 'cc567e17eb9a496f99a60a6f37243c17',
    metadata: {},
    text_tokens: [ [ 0, 3 ], [ 4, 8 ], [ 8, 9 ], [ 10, 12 ] ],
    intent_ranking: [
      { name: 'news_fetch', confidence: 0.9524834156036377 },
      { name: 'ask_hru', confidence: 0.046181075274944305 },
      { name: 'greet', confidence: 0.00039312164881266654 },
      { name: 'bot_where', confidence: 0.0003473285469226539 },
      { name: 'thank', confidence: 0.00025358531274832785 },
      { name: 'ask_bot_question', confidence: 0.00008585600153310224 },
      { name: 'fetch', confidence: 0.00006705214036628604 },
      { name: 'data_enter', confidence: 0.00004517769775702618 },
      { name: 'mood_unhappy', confidence: 0.00004127583815716207 },
      { name: 'unhappy_path', confidence: 0.00003275157723692246 }
    ],
    response_selector: {
      all_retrieval_intents: [],
      default: {
        response: {
          responses: null,
          confidence: 0,
          intent_response_key: null,
          utter_action: 'utter_None'
        },
        ranking: []
      }
    }
  },
  paused: false,
  slots: {
    IMSI_number: null,
    customer_type: null,
    enterprise_name: null,
    enterprise_agreement_number: null,
    parent_organization: null,
    plan_name: null,
    connectivity_lock: null,
    in_session: null,
    network_connectivity: null,
    monthly_data: null,
    data_trend: null,
    msisdn: null,
    requested_slot: null,
    session_started_metadata: null
  }
}
```


In this collection **_id, and sender_id** is the only with the unique values.

As You can see that only one document of the conversation is pretty long . That is because it has the list of events. In each event it stores the necessary information about the message it self.

In the Rasa tracker store, conversations are not limited to just user and bot messages and responses. They encompass a wide range of event categories, including

session initialization, slot updates, actions taken, user feature extraction, active loops for forms, rewinds, and rejected action executions. Each event category has its own corresponding fields.

Let's take the user event type as an example. It contains essential components such as the actual text message typed by the user, the predicted intent and entities extracted from that message, and the ranking of intents based on their confidence levels in machine learning. Additionally, it includes token positions and the unique message ID stored in MongoDB.

Similarly, in the action event type, we find the name of the predicted action and the policy applied to it. We can also determine the confidence level of the action prediction for the specific message.

3.5.4 Endpoints

RASA Endpoints

We have file named as the ***endpoints.yml*** available in the root directory which is responsible for managing the endpoints which can trigger rasa server or which can be triggered by rasa. Following are the endpoints used by rasa server

- **Action Endpoint:**

```yaml
action_endpoint:
  url: "http://rasa_actions:5055/webhook"
  token: ""
```

Figure 3.9: endpoints.yml

So basically Action Server (which has the actions.py file for the custom action will triggered)

For starting the action server we can use the following command in the root directory of the repository:

```
$ rasa run actions
```

Then all the custom action functions will be running and will be registered with the action endpoint *http://0.0.0.0:5055*

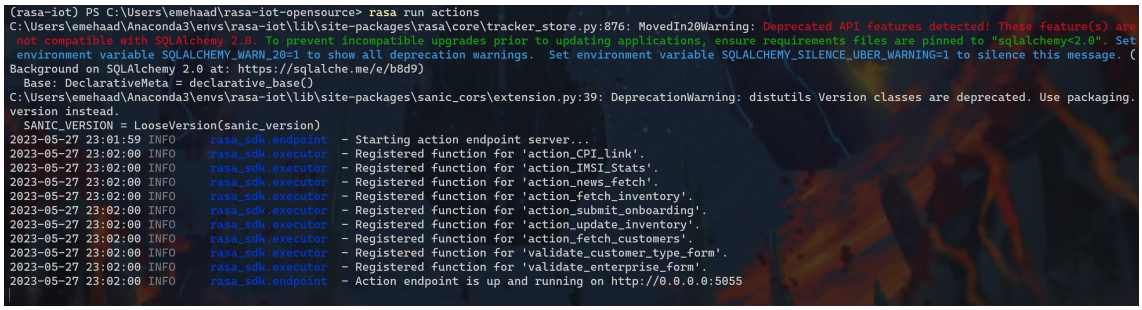


Figure 3.10: Running the rasa custom actions

- **Tracker Store:**

Figure 3.11: endpoints.yml

It will be automatically be triggered when the User will chat with the IoT Digital Assistant and all the events will be first in the local memory and will be stored in the mongoDB tracker store as well. Tracker store data content has been shown in the Database architecture section. As you can see that I have referred to the mongo container in the endpoint on on 27017 port number.

3.5.5 User Interface

The User Interface (UI) Consist of two main components:

1. IoT Service Portal (Mock) Pages
2. Chatbot Widget

The User Interface directory is available in the **Chatbot-Widget-main** in the root directory of the repository

IoT Service Portal (Mock)

We have currently four pages In the mock of Service Portal

- Home
- Inventory
- Customers
- Subscriptions

This is a simulated user interface (UI) for an IoT service portal, designed to resemble the actual service portal's interface. At present, it includes three pages for different business use cases: inventory, customer, and subscriptions. Each page features a table displaying data retrieved from a MongoDB server, albeit indirectly. The data retrieval process involves a Node server running Mongoose services inside express app, which in turn fetches the data from the MongoDB server.

Following Figure will help you to understand the architecture of how it has been implemented.

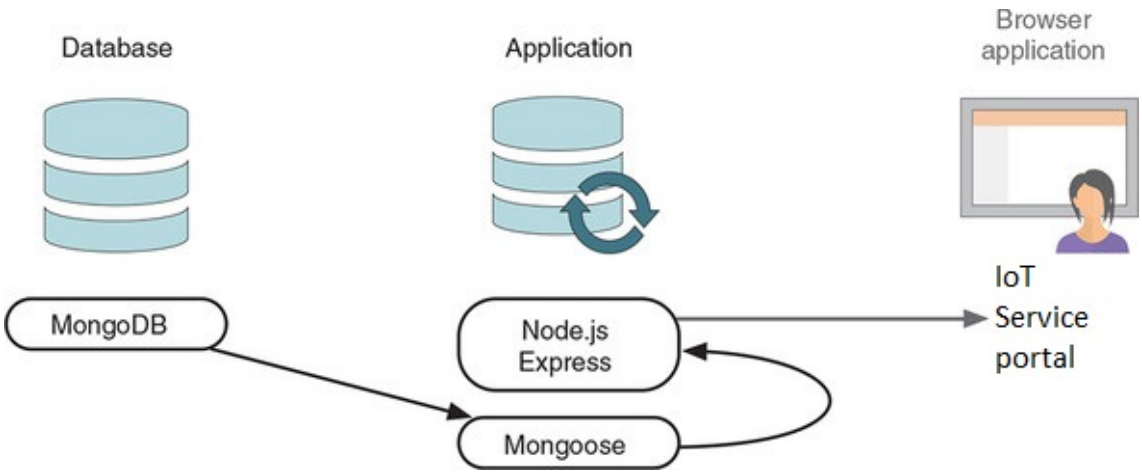


Figure 3.12: Architecture of Interaction of MongoDB and UI

The Node application has to be started as well for populating the data in the UI as shown below:

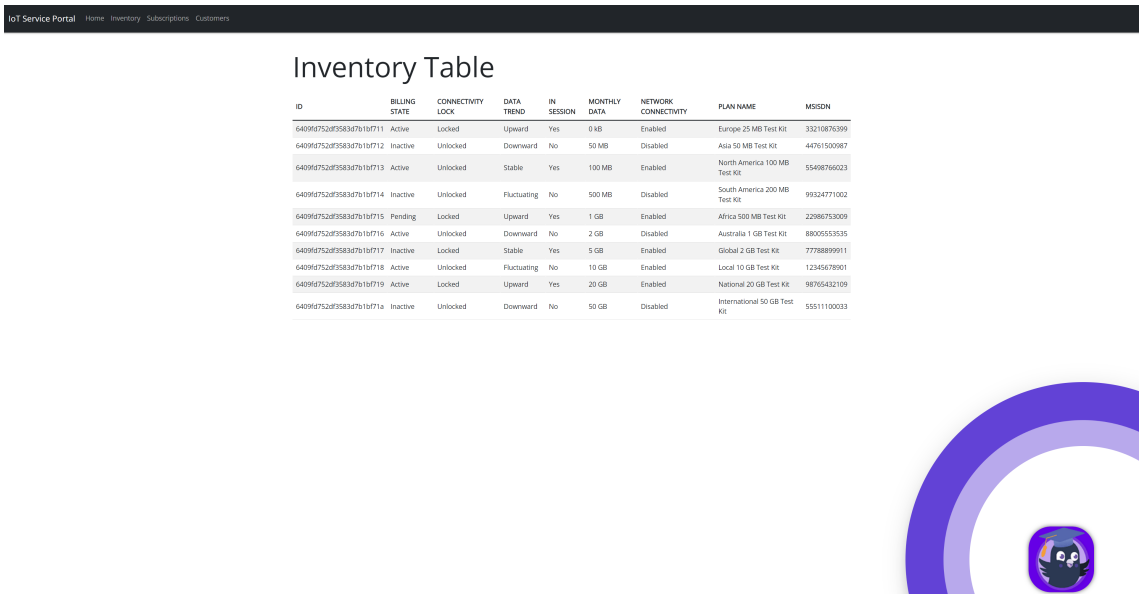


Figure 3.13: User Interface Inventory Page Populated with User Data

In the Express I am creating the four endpoints to send it to the MongoDB server: you can refer to the file : *Chatbot-Widget-main/static/js/server.js*

GET Requests

1. */inventory?filter=*
2. */inventory/*
3. */customers/*
4. */subscriptions/*

For running this service locally run the following command:

```
$ node server.js
```

One of the endpoint implementation is shown below :

```
const app = express();
app.use(cors());
// Define a route to retrieve inventory data and populate the HTML table
app.get('/inventory/', async (req, res) => {
  try {
    // Retrieve inventory data from the database
    const inventoryData = await Inventory.find();
    res.send(inventoryData);
  } catch (err) {
    console.error('Error retrieving inventory data:', err);
    res.status(500).send('Internal Server Error');
  }
  console.log("/inventory/ Endpoint is running");

});
```

Figure 3.14: server.js Endpoints

On the client side, I have used the following code :

in file : *Chatbot-Widget-main/static/js/inventory.js*

```
document.addEventListener('DOMContentLoaded', () => {
  fetch('http://localhost:3000/inventory')
    .then(response => {
      if (response.status !== 200) {
        throw new Error('Failed to fetch inventory data');
      }
      return response.json();
    })
    .then(inventoryData => {
      console.log(inventoryData);
      populateTable(inventoryData, "inventory-table");
    })
    .catch(error => {
      console.error('Error fetching inventory data:', error);
    });
});
```

Figure 3.15: client Side

populateTable has been implemented as follow: in file : *Chatbot-Widget-main/static/js/script.js*

```
1 function populateTable(data, tableId) {
2   const table = document.getElementById(tableId);
```

```
3
4   // Create the table header
5   const thead = document.createElement('thead');
6   const headerRow = document.createElement('tr');
7
8   // Extract the keys from the first object in the data
9   const keys = Object.keys(data[0]);
10
11  // Create table headings based on the keys
12  keys.forEach((key) => {
13    const formattedKey = key.replace(/_/g, ' '); // Remove
        underscores
14    const capitalizedHeading = formattedKey.toUpperCase();
15    const th = document.createElement('th');
16    th.textContent = capitalizedHeading;
17    headerRow.appendChild(th);
18  });
19  console.log(table);
20
21  thead.appendChild(headerRow);
22
23
24  // Create the table body
25  const tbody = document.createElement('tbody');
26
27  // Populate the table with data
28  data.forEach((obj) => {
29    const row = document.createElement('tr');
30
31    keys.forEach((key) => {
32      const cell = document.createElement('td');
33      cell.textContent = obj[key];
34      row.appendChild(cell);
35    });
36
37    tbody.appendChild(row);
38  });
39  console.log(tbody);
40  table.appendChild(thead);
```

```
41    table.appendChild(tbody);
42 }
```

Chatbot Widget [19]

In this thesis, I incorporated an existing Chatbot Widget, which I have referenced in this section. The Chatbot widget offers a plethora of features and functionalities. It is configured to run the Rasa server with CORS set to "*". The widget has been developed using JavaScript, HTML, and CSS, with the implementation of Materialize CSS. Noteworthy features of the Chatbot widget include:

1. Text responses
2. Markdown formatting
3. Button interactions
4. Image display
5. Video playback
6. PDF attachments
7. Dropdown menus
8. Quick reply options
9. Carousel-style card display
10. Charts for data visualization
11. Collapsible sections
12. Bot typing indicator
13. Location access capabilities.

These features collectively enhance the functionality and user experience of the Chatbot widget.

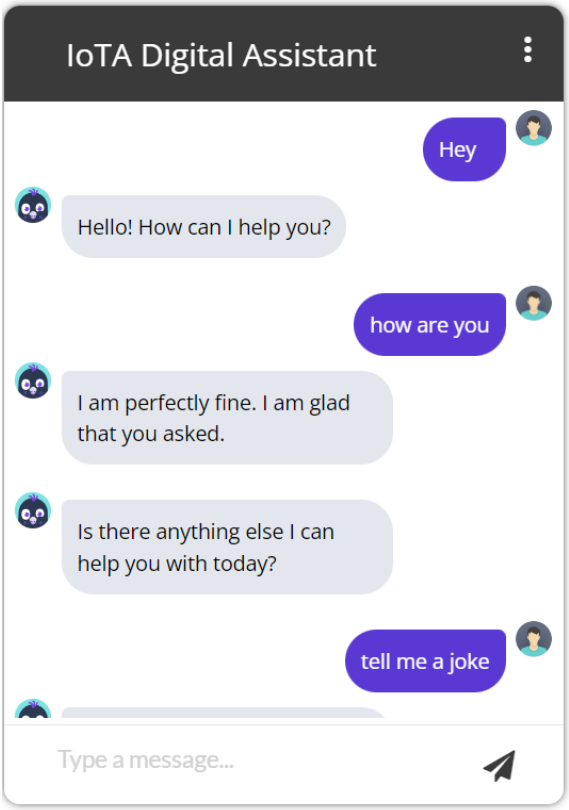


Figure 3.16: Chatbot User Interface Example

As You can see that UI is connected with the RASA Server that is why the responses are being shown onto the UI.

This is because we have configured endpoints into the the following files:

./Chatbot-Widget-main/static/js/constants.js

Chatbot-Widget-main > static > js > JS constants.js > ...

```
1  const action_name = "action_hello_world";
2  // const rasa_server_url = "http://10.44.62.84/webhooks/rest/webhook";
3  const rasa_server_url = "http://localhost:5005/webhooks/rest/webhook";
4  const sender_id = uuidv4();
5
```

Figure 3.17: ./Chatbot-Widget-main/static/js/constants.js

And that constant has been called here:

./Chatbot-Widget-main/static/js/components/chat.js

```
async function send(message) {
  await new Promise((r) => setTimeout(r, 2000));
  $.ajax({
    url: rasa_server_url,
    type: "POST",
    contentType: "application/json",
    data: JSON.stringify({ message, sender: sender_id }),
    success(botResponse, status) {
      console.log("Response from Rasa: ", botResponse, "\nStatus: ", status);

      // if user wants to restart the chat and clear the existing chat contents
      if (message.toLowerCase() === "/restart") {
        $("#userInput").prop("disabled", false);

        // if you want the bot to start the conversation after restart
        // customActionTrigger();
        return;
      }
      setBotResponse(botResponse);
    },
    error(xhr, textStatus) {
      if (message.toLowerCase() === "/restart") {
        $("#userInput").prop("disabled", false);
        // if you want the bot to start the conversation after the restart action.
        // actionTrigger();
        // return;
      }

      // if there is no response from rasa server, set error bot response
      setBotResponse("");
      console.log("Error from bot end: ", textStatus);
    },
  });
}
```

Figure 3.18: ./Chatbot-Widget-main/static/js/components/chat.js

We have also configured the Custom Actions endpoint in the UI in the same file:

Chatbot-Widget-main > static > js > components > JS chat.js > ...

```
310   * `Note: this method will only work in Rasa 2.x`
311   */
312  // eslint-disable-next-line no-unused-vars
313  function customActionTrigger() {
314    $.ajax({
315      url: "http://localhost:5055/webhook/",
316      type: "POST",
317      contentType: "application/json",
318      data: JSON.stringify({
319        next_action: action_name,
320        tracker: {
321          sender_id,
322        },
323      }),
324      success(botResponse, status) {
325        console.log("Response from Rasa: ", botResponse, "\nStatus: ", status);
326
327        if (Object.hasOwnProperty.call(botResponse, "responses")) {
328          setBotResponse(botResponse.responses);
329        }
330        $("#userInput").prop("disabled", false);
331      },
332      error(xhr, textStatus) {
333        // if there is no response from rasa server
334        setBotResponse("");
335        console.log("Error from bot end: ", textStatus);
336        $("#userInput").prop("disabled", false);
337      },
338    });
339  }
340
```

Figure 3.19: ./Chatbot-Widget-main/static/js/components/chat.js

3.5.6 Implementation Sketch and Integration

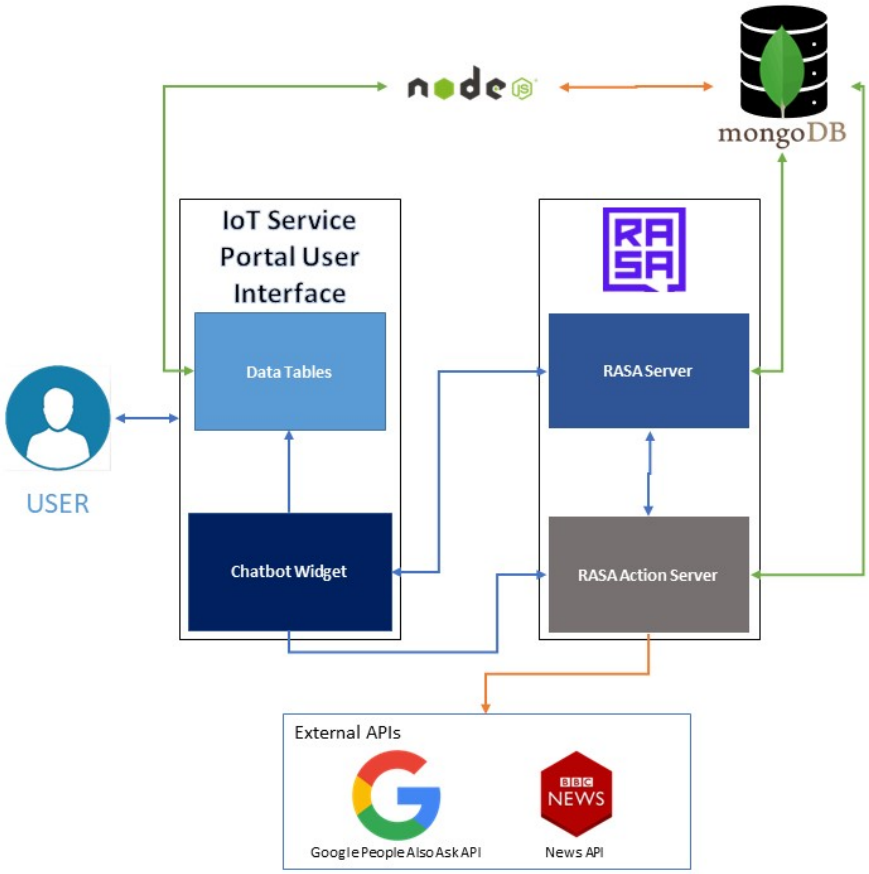


Figure 3.20: Implementation Sketch and Integration of all components

3.6 Implementation

I will go through all the steps that I have implemented to get the final product of the IoT Digital Assistant.

3.6.1 Domain of the Chatbot

I have Implemented the Domain of the IoT Digiral Assistant as follow with all the required Information.

Current Intents

Following are all classes of intents list currently implemented in the chatbot. They can be seen in the **domain.yml** file in the repo directory.

domain.yml

```
5   intents:
6   - Learn
7   - Troubleshoot
8   - affirm
9   - ask_bot_question
10  - ask_hru
11  - ask_joke
12  - bot_age
13  - bot_challenge
14  - bot_name
15  - bot_where
16  - customer_advanced_reseller
17  - customer_enterprise
18  - data_enter
19  - deny
20  - describe_bot
21  - enterprise_agreement_number_enter
22  - enterprise_name
23  - fetch
24  - goodbye
25  - greet
26  - information_Comparison
27  - information_seek
28  - mood_great
29  - mood_unhappy
30  - news_fetch
31  - onboard_new_customer
32  - parent_organization_china_telecom_mongolia_branch
33  - thank
34  - troubleshooting_time
35  - unhappy_path
36  - update_inventory
37  - weather_info
```

Figure 3.21: domain.yml

Current Entities

You can also check all classes of entities for the IoT Digital Assistant in the **domain.yml** file as well.

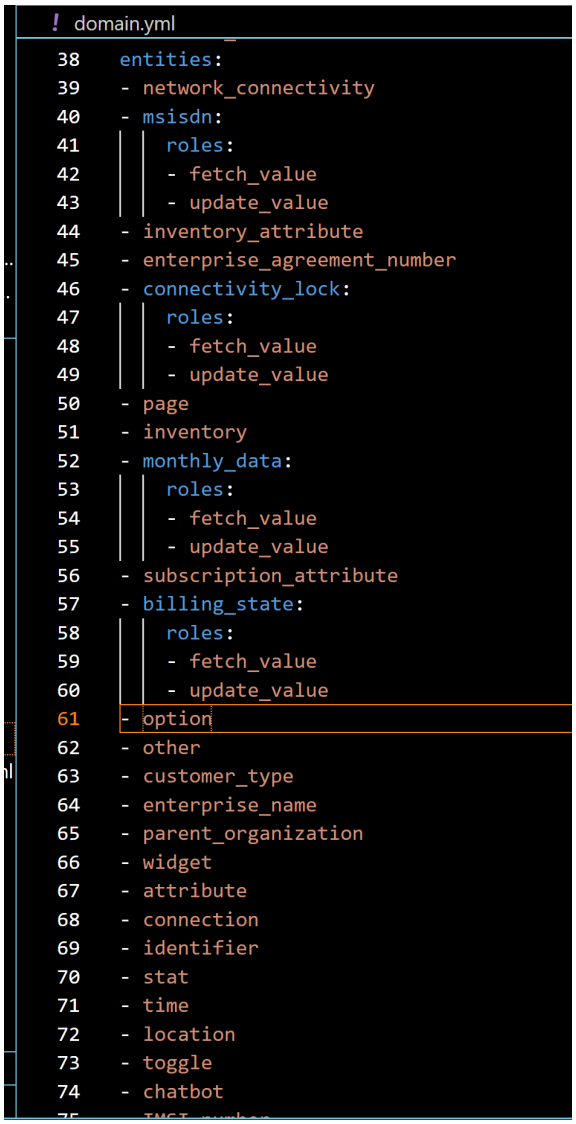


Figure 3.22: domain.yml

```
- chatbot
- IMSI_number
- news
- portal
- plan_name:
    roles:
    - fetch_value
    - update_value
- in_session:
    roles:
    - fetch_value
    - update_value
- data_trend:
    roles:
    - fetch_value
    - update_value
```

Figure 3.23: domain.yml

Current Slots

I have set up slot values for specific entities to capture and store user-provided data. Slots act like placeholders that hold certain types of information given by the user. For example, consider a user providing their MSISDN (Mobile Station International Subscriber Directory Number, more commonly known as a phone number). This constitutes a piece of personal data from the user. We need a slot to identify that its associated entity is a phone number. This information is stored in the chatbot's memory, enabling it to recall and reference this data when the user asks something related to their MSISDN. This way, the chatbot is able to provide a more personalized and relevant response.

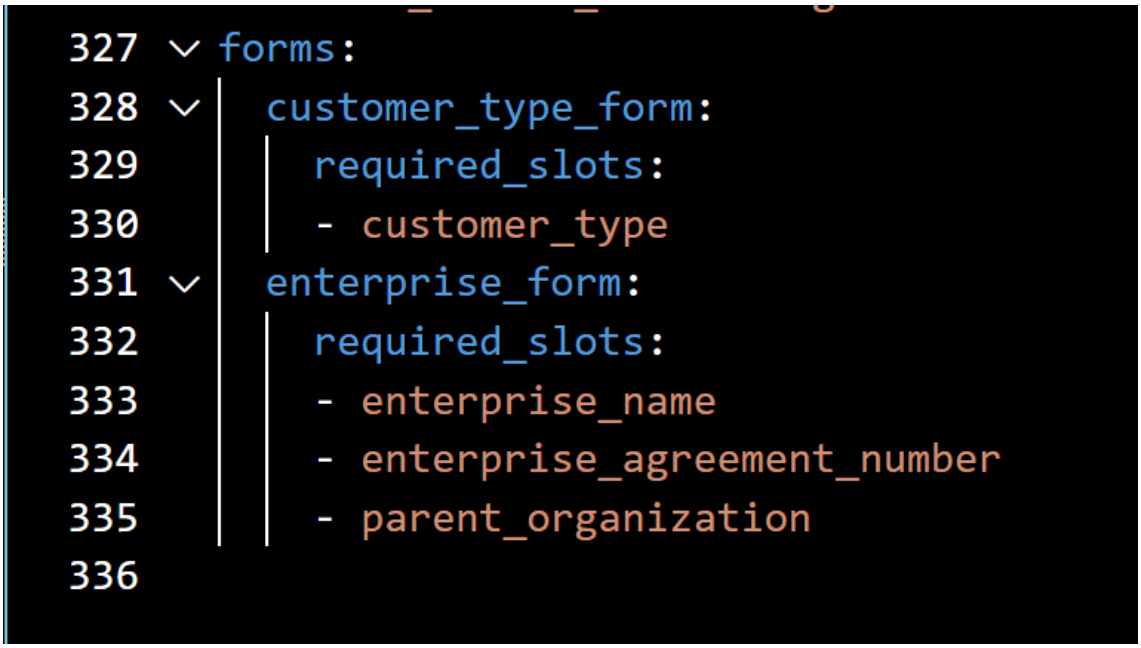
Following are all the classes of the slots i have configured:

```yaml
slots:
  IMSI_number:
    type: text
    mappings:
    - type: from_entity
      entity: IMSI_number
    influence_conversation: true
  customer_type:
    type: text
    mappings:
    - type: from_entity
      entity: customer_type
    influence_conversation: true
  enterprise_name:
    type: text
    mappings:
    - type: from_entity
      entity: enterprise_name
    influence_conversation: false
  enterprise_agreement_number:
    type: text
    mappings:
    - type: from_entity
      entity: enterprise_agreement_number
    influence_conversation: false
  parent_organization:
    type: text
    mappings:
    - type: from_entity
      entity: parent_organization
    influence_conversation: false
  plan_name:
    type: text
    mappings:
    - type: from_entity
      entity: plan_name
    influence_conversation: false
  connectivity_lock:
```

Figure 3.24: domain.yml

Current Forms

Subsequently, we have implemented forms to facilitate the customer onboarding process on the IoT Service Portal. Forms are a feature provided by Rasa that, once activated, prompts the bot to inquire about the necessary details required by the form. If the user gets sidetracked and starts discussing a different topic, the chatbot will respond appropriately to the new line of questioning. However, if the conversation swings back to the original context, the bot is designed to pick up from where the discussion was interrupted, ensuring a seamless conversational experience.



```
327 forms:
328   customer_type_form:
329     required_slots:
330     - customer_type
331   enterprise_form:
332     required_slots:
333     - enterprise_name
334     - enterprise_agreement_number
335     - parent_organization
336
```

Figure 3.25: domain.yml

Available actions

Currently the trained chatbot has the following actions which can be executed:

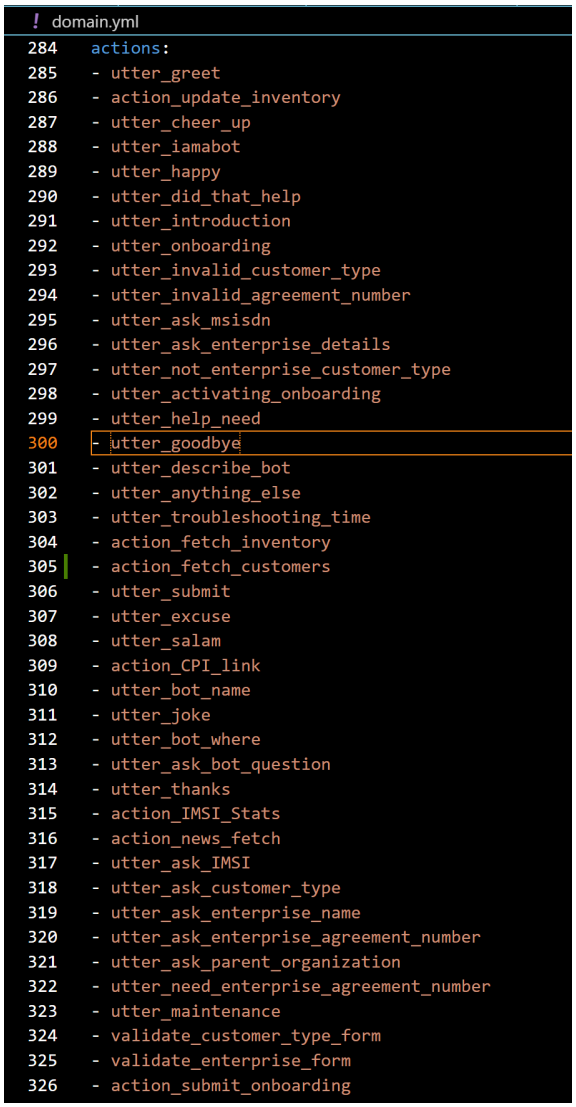


Figure 3.26: domain.yml

You can see that there are 4 kind of actions :

1. Those which starts with utter_[ACTION] are basically the simple text based actions which are configured with hard and fast rule.

2. Those which starts with action_[ACTION] are the custom actions coded in **actions/actions.py** file.

3. Those which starts with validate_[FORM NAME]_form are custom actions for the validation of the forms implemented in the **actions/actions.py** file.

4. Those which starts with utter_ask_[SLOT_NAME] are those actions which asks the user for the slot required by the form.

3.6.2 Training Data

It's well established that quality data is the lifeblood of effective machine learning. Without it, our ability to construct a high-performing machine learning model is significantly hindered, potentially leading to suboptimal accuracy. We also need to strike a balance in training our model to prevent overfitting, a scenario where a model performs impressively on the training data but falls short on the test data. Our goal is to ensure our model can adapt well to unseen data without being overfit.

In the context of our IoT Digital Assistant, we want to utilize user queries about IoT Service Portal as training data for the Natural Language Understanding (NLU) model. This model is vital for classifying the intent, i.e., deciphering the user's request to the digital assistant. Alongside this, we need a Named Entity Recognition (NER) model. This model's role is to classify the entities within a given sentence.

Another essential component is the Dialogue Manager, responsible for managing the conversation flow. For this component to function effectively, we need user narratives in the training data. These narratives or stories help to comprehend the progression of the conversation. For instance, a particular action should follow a certain intent, but it's crucial to remember that this isn't a rigid rule. By having a large number of stories associated with different intents and similar actions, the Dialogue Manager becomes equipped to handle varying edge cases and decide the appropriate action to execute.

There's also rule data, similar in nature to the stories, but with a critical difference. Rules are definitive and state that a specific intent should always be followed by a certain action. They cannot be overridden by stories. Ignoring this would lead to consistency errors and prevent the model from being trained.

Sources:

Initially, I had no user query data for the IoT Service Portal at my disposal. To overcome this, I began by gathering some Natural Language Understanding (NLU) data from existing IoT Service Portals. Despite the data being somewhat limited, I was able to generate additional examples using the available machine learning models at the time to rephrase the existing ones. To ensure a high recall value and a balanced dataset, I created at least 10 to 15 examples for each intent.

Once I had gathered a basic dataset sufficient for training, I trained the Intent

Classification model and the Named Entity Recognition (NER) model. I used Bidirectional Long Short-Term Memory (BI LSTM) alongside Natural Language Processing (NLP) libraries, NLTK for Intent Classification and Spacy for NER, respectively.

Subsequently, I utilized the Interactive Learning feature of RASA to train additional examples on the fly. Most of my dataset was accrued through this live training of my bot and the implementation of specific features.

Furthermore, I employed ChatGPT to generate numerous examples for each intent. An automation script was used to add the configured entities within the text, enhancing the depth and variety of my training dataset.

The training data is available in /**data**/ directory.

Natural Language Understanding NLU

For each intent, we've incorporated a wide array of examples into the nlu.yml file. These examples serve as training material for the Intent Classification model. Some sentences also include entities within them, aiding in the training of the Named Entity Recognition (NER) model. The NER model operates using BILUO tags, where:

- B stands for 'beginning',
- I signifies 'inside',
- L represents 'last',
- O denotes 'outside', and
- U indicates 'unit'.

These tags provide a structured way to label entities within sentences, thereby improving the model's accuracy in recognizing and classifying them.

Here is the example for adding the data for certain intent:

data > ! nlu.yml

```
853       - what is [msisdn](identifier)
854   - intent: fetch
855     examples: |
856       - List [billing states](option) please
857       - What are the [billing states](option)?
858       - What are some [billing states](option)?
859       - What [billing states](option) do you want to list?
860       - How do I list [billing states](option)?
861       - What [billing states](option) do you have?
862       - fetch the [traffic](inventory) generated last 30 [days](time).
863       - What is the [traffic](inventory) generated in last 30 [days](time)?
864       - How do I fetch [traffic](inventory) generated last 30 [days](time)?
865       - How can I fetch [traffic](inventory) generated last 30 [days](time)?
866       - How do I get [traffic](inventory) generated last 30 [days](time)?
867       - How can I get [traffic](inventory) generated last 30 [days](time)?
868       - fetch the [subscription](inventory) created in last 30 [days](time).
869       - How do I fetch the [subscription](inventory) created in last 30 [days](time)?
870       - How can I fetch the [subscription](inventory) created in last 30 [days](time)?
871       - How do I fetch [subscription](inventory) created in last 30 [days](time)?
872       - How do I fetch [subscriptions](inventory) created in last 30 [days](time)?
873       - How can I fetch my [subscription](inventory) created in last 30 [days](time)?
874       - Show me the [subscription](inventory) with the largest [traffic](inventory).
875       - Show me the [subscription](inventory) with largest [traffic](inventory).
876       - Show me the [subscription](inventory) with the biggest [traffic](inventory).
877       - show me the [subscriber](inventory) with the largest [traffic](inventory).
878       - Bring me [location](location) with largest [traffic](inventory).
879       - Where can I get the most [traffic](inventory)?
880       - What is the best [location](location) for [traffic](inventory)?
881       - What is the best [location](location) with the most [traffic](inventory)?
882       - Bring me [location](location) with the largest [traffic](inventory).
883       - Show me the [report](stat).
884       - Can you show me the [report](stat)?
885       - Give me the [report](stat).
886       - What is the [report](stat)?
887       - Could you show me the [report](stat)?
888       - How do I generate [invoices](stat)?
889       - How can I generate [invoices](stat)?
890       - How do you generate [invoices](stat)?
891       - How should I generate [invoices](stat)?
892       - How do I generate an [invoice](stat)?
893       - How does one generate [invoices](stat)?
894       - How do I get [usage data](stat)?
895       - Can you give me [usage data](stat)?
```

Figure 3.27: data/nlu.yml

Before NLU the examples are tokenized into words and featurized. You can read more about RASA NLU Pipeline and architecture on the section 3.5.2 .

Stories

I've strived to incorporate as many stories as possible to handle potential edge cases in the IoT Digital Assistant. However, it's important to note that this chatbot hasn't yet been deployed to end users for production testing. As a result, there may be instances where the conversation doesn't go as smoothly as anticipated.

The creation of stories for the stories.yml file involves documenting the corresponding sequence of events, such as the intent, slot_was_set, actions, or active_loop. These elements represent the progression of a conversation. While I've worked to make the conversation flow as natural and comprehensive as possible, the chatbot's performance will truly be determined once it starts interacting with real users in a production environment.

Here is an example of multiple stories for the same intent to distinguish between two conversations:

data > ! stories.yml

```yaml
236
237 - story: fetching inventory
238   steps:
239   - intent: greet
240   - action: utter_greet
241   - intent: fetch
242     entities:
243     - other: data
244     - inventory: inventory
245     - inventory_attribute: msisdn
246     - msisdn: '55511100033'
247   - slot_was_set:
248     - msisdn: '55511100033'
249   - action: action_fetch_inventory
250
251 - story: fetching_inventory2
252   steps:
253   - intent: fetch
254     entities:
255     - inventory_attribute: msisdn
256     - msisdn: '55511100033'
257   - slot_was_set:
258     - msisdn: '55511100033'
259   - action: action_fetch_inventory
260 - story: fetching_inventory3
261   steps:
262   - intent: fetch
263     entities:
264     - inventory_attribute: msisdn
265     - msisdn: '55511100033'
266   - slot_was_set:
267     - msisdn: '55511100033'
268   - action: action_fetch_inventory
269
270 - story: fetching_inventory3
271   steps:
272   - intent: fetch
273     entities:
274     - option: plans
275     - inventory: inventory
276     - inventory_attribute: billing_state
277     - billing_state: Inactive
278     - inventory_attribute: connectivity_lock
279     - connectivity_lock: Unlocked
280   - action: action_fetch_inventory
281
```

Figure 3.28: data/story.yml

As you can see there is difference of entities detected in the same intents which deeply means that the User is trying to ask to fetch the inventory but giving different parameter and with different way of talking.

For Instance :

```
1    User : show me the inventory in which the plans are
        unlocked.
2    Bot : ....
```

```
1    User : fetch the inventory in which msisdn is XYZ, the
        plans are locked .
2    Bot : ....
```

Rules

I've integrated a select number of rules into the Rasa framework. These rules are stringent and absolute, governing certain essential aspects of the chatbot's behavior. Here are the implemented rules for reference:

data > ! rules.yml

```yaml
version: "3.0"
rules:
- rule: Say goodbye anytime the user says goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
- rule: Say 'I am a bot' anytime the user challenges
  steps:
  - intent: bot_challenge
  - action: utter_iamabot
- rule: Fetch the CPI link
  steps:
  - or:
    - intent: Learn
    - intent: information_seek
  - action: action_CPI_link
  - action: utter_anything_else
- rule: thanking
  steps:
  - intent: thank
  - action: utter_thanks
  - action: utter_anything_else
- rule: Activating Customer Type Form
  steps:
  - intent: onboard_new_customer
  - action: utter_activating_onboarding
  - action: customer_type_form
  - active_loop: customer_type_form
- rule: Activating Enterprise Form
  condition:
  - slot_was_set:
    - customer_type: enterprise
  steps:
  - action: enterprise_form
  - active_loop: enterprise_form
- rule: Submitting Customer type form
  condition:
  - active_loop: customer_type_form
  steps:
  - action: customer_type_form
  - active_loop: null
  - slot_was_set:
    - requested_slot: null
  wait_for_user_input: false
- rule: Submitting Enterprise form
  condition:
  - active_loop: enterprise_form
  steps:
  - action: enterprise_form
  - active_loop: null
  - slot_was_set:
    - requested_slot: null
  - action: action_submit_onboarding
  - action: utter_submit

```

Figure 3.29: data/rules.yml

3.6.3 Actions

Once the user's query is effectively interpreted via Natural Language Understanding, a predefined story or rule comes into play. These, in turn, predict a specific action that needs to be performed to craft a response. This response is then communicated back to the user, ensuring a fluid and meaningful dialogue.

After the prediction of the action, the chatbot executes it. Actions can be categorized into various types, such as default actions, responses, custom actions, forms,

and slot validation actions. These action types play distinct roles within the chatbot's conversational architecture. Moreover, it's crucial to note that the names of these actions need to be properly cataloged in the domain.yml file to ensure accurate functioning and referencing.

For Reference : `https://rasa.com/docs/rasa/actions/`

Responses

In the domain.yml file we also define predefined responses and we refer those responses in the certain story in the stories.yml file.

The subsequent illustration provides a view of the **domain.yml** file, specifically highlighting the section related to 'responses'. This is where we define how the bot should respond to specific user intents or actions.

```yaml
responses:
  utter_troubleshooting_time:
  - text: I'm sorry, but I don't have any information about the maintenance schedule at the moment.
  utter_greet:
  - buttons:
    - payload: /mood_great
      title: Happy
    - payload: /mood_unhappy
      title: Sad
    text: Hi! How are you?
  - text: Hello! How can I help you?
  - text: Hi, Thanks so much for reaching out! What brings you here today?
  utter_cheer_up:
  - image: https://i.imgur.com/nGF1K8f.jpg
    text: 'Here is something to cheer you up:'
  utter_maintenance:
  - text: We are maintaining the system. We will look into your system for the possible solution
  - text: Unfortunately our system is under maintenance. We kindly ask for your patience as we are already working on it
  - text: Our team is currently carrying out maintenance on the system, and we will investigate the issue you reported as soon as possible.
  utter_need_enterprise_agreement_number:
  - text: Unfortunately, we cannot proceed onboarding your {customer_type} without "Enterprise agreement number" (Its a 14 digit number). You car
  - text: I'm sorry, but the agreement number is mandatory to complete the onboarding process of your {customer_type} which is a 14 digit numnber
  - text: We need your enterprise agreement number which is a 14 digit number for this procedure to proceed further. You can always find it in th
  utter_iamabot:
  - text: I am a bot, powered by IOTA.
  - text: I am a Digital Assistant designed to provide the services to the user of the IoTA Service Portal.
  utter_happy:
  - text: Great, carry on!
  - text: Excellent! Please continue.
  utter_ask_hru:
  - text: I am perfectly fine. I am glad that you asked.
  - text: I am spectacular. Thanks for asking.
  - text: I appreciate that you asked. I am fine
  utter_did_that_help:
  - text: Did that help you?
  - text: I hope it was helpful to you?
```

Figure 3.30: domain.yml

Custom Actions

I have implemented custom actions in python language for multiple use cases. Custom actions can be implemented in the RASA SDK. Rasa SDK provides the tool for implementing the custom actions and register them in the action server

endpoint. Rasa action server provides a webhook endpoint which accepts the POST Request from Rasa Server . See [20].

It provide payload input in the json format from Rasa Server and after running the certain action it sends the response payload to the rasa server.

A Few things to understand before going to the actual Implementation. I have provided the reference for reading.

1. **Action [21]**

   The Action class is the base class for any custom action. To define a custom action, create a subclass of the Action class and overwrite the two required methods, name and run. The action server will call an action according to the return value of its name method when it receives a request to run an action.

2. **Tracker [22]**

   The Tracker class represents a Rasa conversation tracker. It lets you access your bot's memory in your custom actions. You can get information about past events and the current state of the conversation through Tracker attributes and methods.

3. **Dispatcher [23]**

   A dispatcher is an instance of the CollectingDispatcher class used to generate responses to send back to the user.

   CollectingDispatcher CollectingDispatcher has one method, utter_message, and one attribute, messages. It is used in an action's run method to add responses to the payload returned to the Rasa server. The Rasa server will in turn add BotUttered events to the tracker for each response. Responses added using the dispatcher should therefore not be returned explicitly as events. For example, the following custom action returns no events explicitly but will return the response, "Hi, User!" to the user:

I have total 9 Custom Actions Implemented. They are enumerated in the following table:

Table 3.1: Custom Actions Base Classes

Custom Action Class	Base Class	Action name
ActionCPIlink	Action	"action_CPI_link"
ActionIMSIStats	Action	"action_IMSI_Stats"
ActionNewsFetch	Action	"action_news_fetch"
ValidateCustomerTypeForm	FormValidationAction	"validate_customer_type_form"
ValidateEnterpriseForm	FormValidationAction	"validate_enterprise_form"
ActionFetchInventory	Action	"action_fetch_inventory"
SubmitOnboardingForm	Action	"action_submit_onboarding"
ActionUpdateInventory	Action	"action_update_inventory"
ActionFetchCustomers	Action	"action_fetch_customers"

Following is the example of one of the custom action implementation for fetching the inventory when the user asks the bot to show the inventory.

```
1   class ActionFetchInventory(Action):
2   def name(self) -> Text:
3       return "action_fetch_inventory"
4
5   def run(self, dispatcher: CollectingDispatcher,
6           tracker: Tracker,
7           domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
8
9       prediction = tracker.latest_message
10      # slot_value = tracker.get_slot("IMSI_number")
11      attributes = ["msisdn","plan_name", "connectivity_lock", "network_connectivity", "
            in_session", "billing_state", "monthly_data", "data_trend"]
12
13      try:
14          client = MongoClient("mongodb://localhost:27017")
15          db = client["IOTA"]
16          inventory = db["inventory"]
17      except:
18          dispatcher.utter_message(text="Sorry! we can not build the connection with the
                database.")
19          return []
20      # Fetching all the data from the inventory database
21      all_entities = [entity["entity"] for entity in prediction["entities"]] #list
22      print(str(prediction))
23      all_entities = [entity for entity in all_entities if entity in attributes]
24
25      entities = {}
26      for i in all_entities:
27          print(str(i))
28          entities[i] = next(tracker.get_latest_entity_values(i), None)
29      filtered_entities = {k: int(v) if v.isnumeric() else {'$regex':'^'+v+'$', '$options':'i'}
            for k, v in entities.items() if v is not None}
30      # I am assuming that in most of queries the none entity will be main attribute which will
            be asked
31      none_entities = {k:v for k,v in entities.items() if v is None}
32
33      print("Entities"+ str(entities))
```

```
        print("Filtered_entities: "+str(filtered_entities))
        print("None Entities"+ str(none_entities))
        try:
            query_result = list(inventory.find(filtered_entities))
        except IndexError as e:
            dispatcher.utter_message(text="Sorry! there is no data on that one")
            return []
        msg = f"""Sure, Here is the data for AND[{entities}]=
        {query_result}
        """
        dispatcher.utter_message(text=msg)
        return []
```

As you can see in the Class of Custom Action, I have inherited Base Action Class from RASA SDK. and I am overriding two required method. ***name(self), run(self, dispatcher, tracker, domain)***

name() method will only return the name of the action name with *action_* as the prefix.

whereas, *run(self, dispatcher, tracker, domain)* method will actually execute the main logic of the action. It takes *dispatcher,tracker, domain* as its parameters. Like in tracker we can get the latest message information and the prediction from NLU and slots. Through Dispatcher we can actually send a response to the user with dispatcher.utter_message(text="Hello user") method. Finally domain parameter is the domain of the bot.

After Implementation, We have to train the model by running the following command :

```
$ rasa train
```

It will take take a bit long time to train. It all depends on the dataset you have. It trains sequence-wise as enlisted in the config.yml pipeline. After the model is ready it will be stored into the models/ directory in the repository.

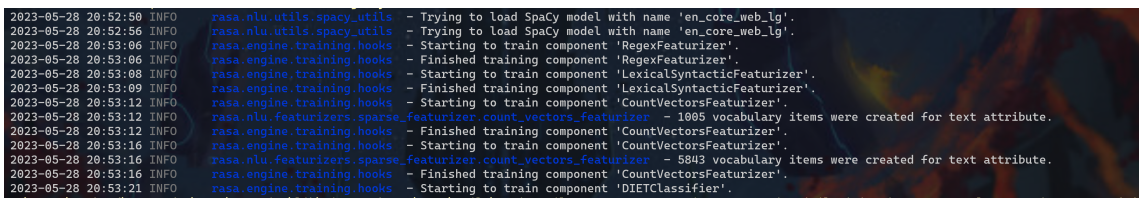


Figure 3.31: Training of the model part 1

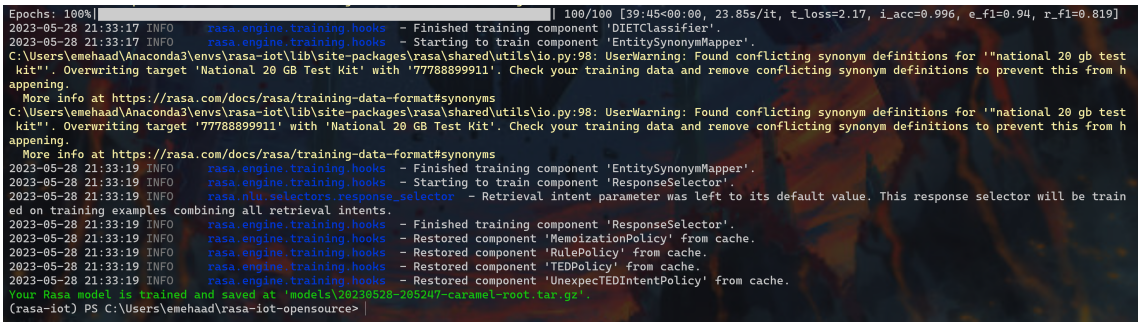


Figure 3.32: Training of the model part 2

You can also do the interactive training as well. In which Bot will start training after the conversation . This is the most suitable way to correctly train your chatbot after creating a feature . This way you will be able to correctly select the intent or create new intents and you will save alot of time instead of configuring it separately. For running the interactive training, you have to run the following command:

```
$ rasa interactive
```

rasa interactive command first takes the input from you for training the chatbot, saves the story, domain and nlu files accordingly and after that trains the model and model will be saved into the models directory.

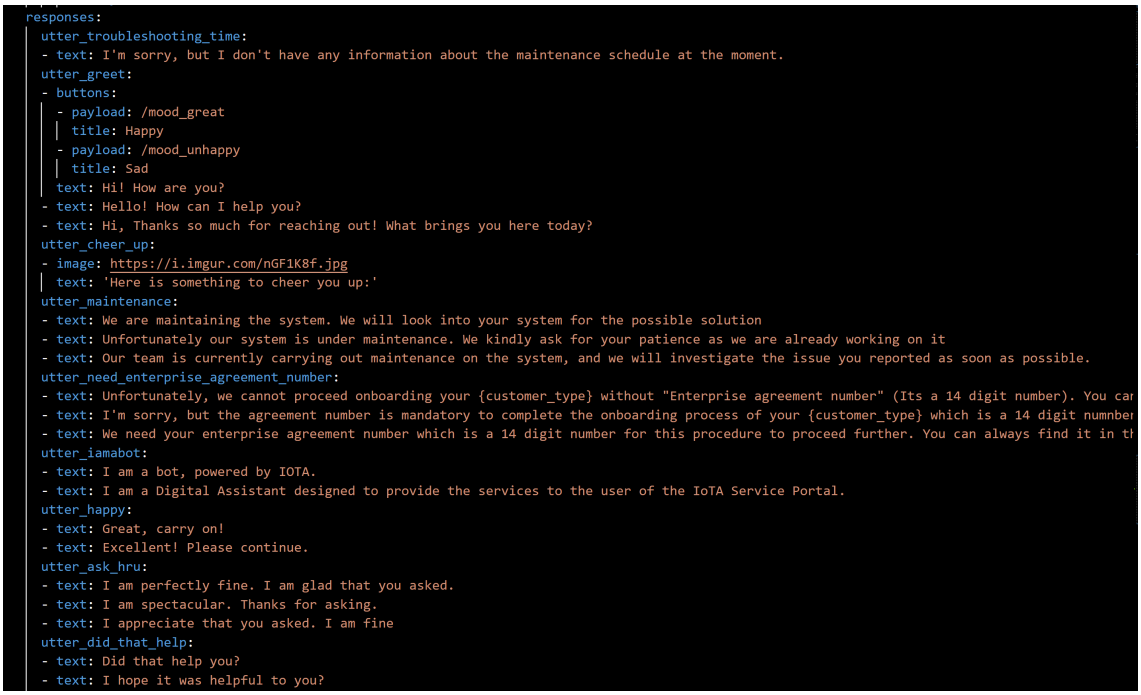


Figure 3.33: domain.yml

For running the IoT Digital Assistant on the shell terminal. You can use the following command :

```
$ rasa shell
```

And finally to make it run in the UI , you need to run the following command to enable the Rasa server endpoint with cors parameter as "*".

```
$ rasa run --enable-api --cors "*"
```

You can actually see all the RASA CLI commands on `https://rasa.com/docs/rasa/command-line-interface/`.

3.7 Testing

For Testing the IoT Digital Assistant, I am using rasa test capability to test the NLU, Dialogue manager , stories, rules , validation techniques.

3.7.1 Testing the NLU model

For Testing the NLU model, We would need to split some of the current NLU data into train and test data and then we will be able to classify the accuracy of Intent Classification and Named Entity Recognition. It is one of the **Black box testing**.

```
$ rasa test nlu
    --nlu data/nlu
    --cross-validation
    --folds 5
```

By default It will split the nlu.yml data into 80/20 shuffled data in training data and testing data : Then Rasa will train the model on only 80 percent splitted training data and will try to test it on the 20 percent of test data. I am using the cross validation technique and using 5 folds for specifying the test/train splits. My Training data is already really large so for each fold it will be really time consuming.

After Trainig the Data We have this output:

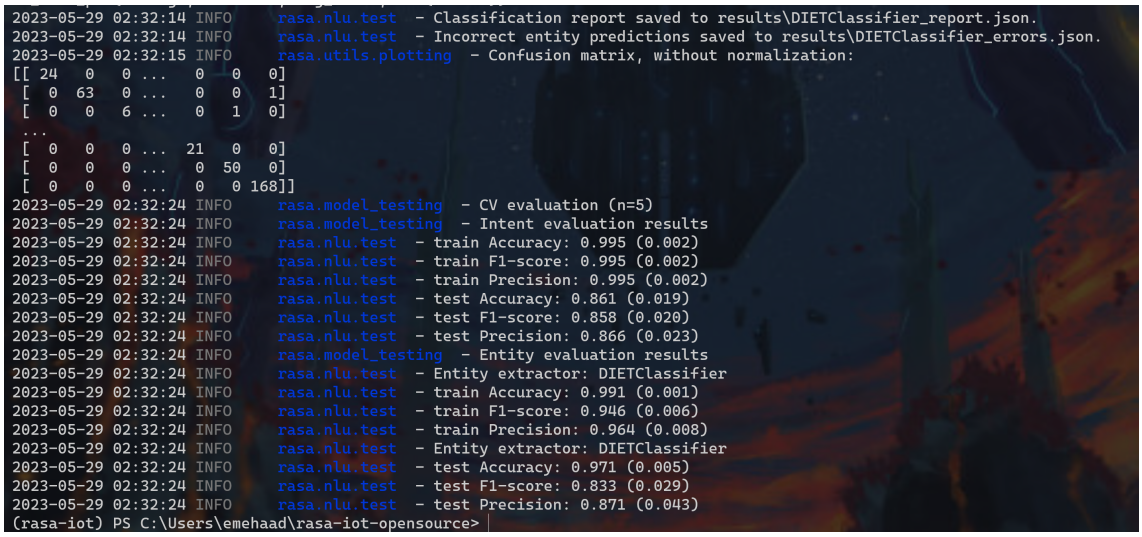


Figure 3.34: NLU Test Result

3.7.2 Interpretation of the Output

Following are some Plots for the insights on the Machine Learning models.

Intent Classification Confusion Matrix

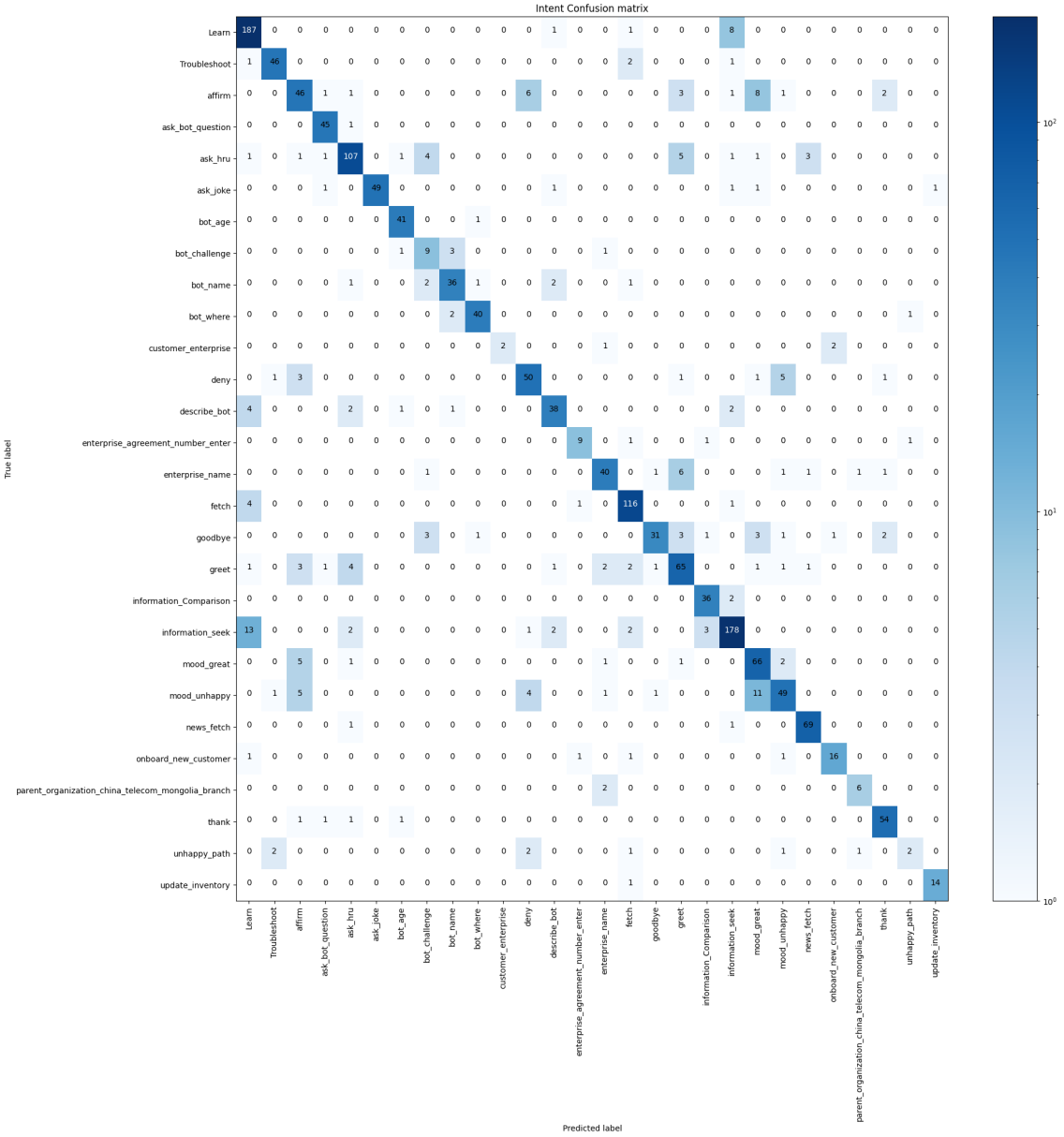


Figure 3.35: Intent Confusion Matrix

DIET Classification Entity Confusion Matrix

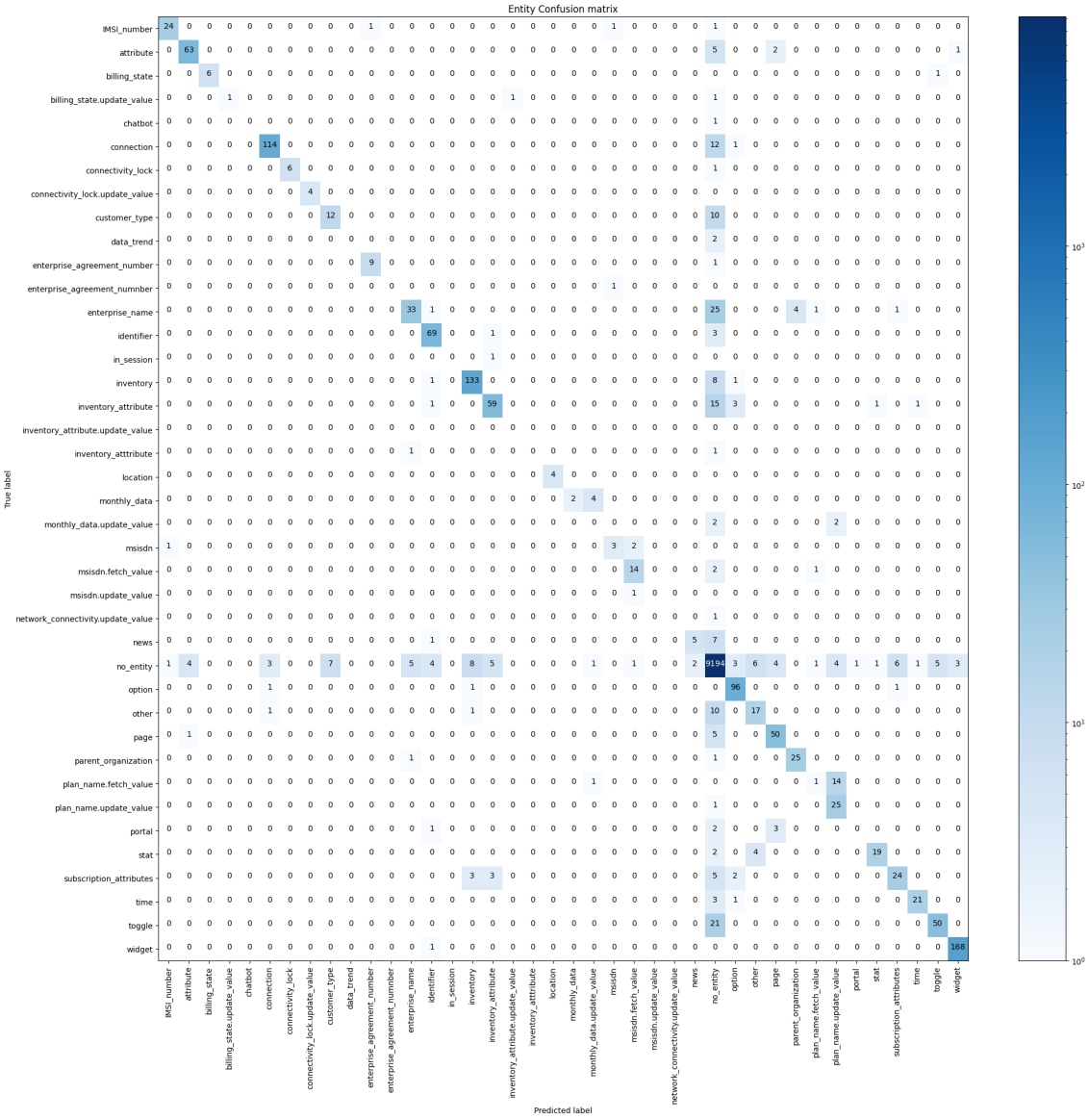


Figure 3.36: Intent Confusion Matrix

Chapter 4

Final Remarks and Prospective Developments:

The present thesis on the IoT Digital Assistant predominantly relies on mock data. For its maturation into a fully functional Digital Assistant, I am contemplating its deployment on the Google Cloud Platform and EdgeX, a widely used platform for IoT administration. This transition would necessitate the integration of more pertinent data to accommodate the use case of EdgeX, thereby enabling effective communication with actual device databases.

I have future plans to enhance its accuracy, expand its storyline repertoire, and make it available to test users to augment its performance. The models utilized currently yield modest accuracy, but leveraging more precise pre-trained models like GPT or other transformer-based models is expected to considerably amplify its precision.

Going forward, the Digital Assistant needs to be fortified with more sturdy action configurations and improved natural language generation abilities. Thus, it promises an exciting and dynamic path of advancements and refinements.

Bibliography

[1] *Microsoft.* URL: https://docs.microsoft.com/en-us/power-platform/admin/web-application-requirements.

[2] *Git Documentatin.* URL: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

[3] *Docker.* URL: https://www.docker.com/get-started/.

[4] *Docker Mac Install.* URL: https://docs.docker.com/desktop/install/mac-install/.

[5] *Docker Linux Install.* URL: https://docs.docker.com/desktop/install/linux-install/.

[6] *MongoDB Definition.* URL: https://www.geeksforgeeks.org/mongodb-database-collection-and-document/.

[7] *MongoDB Definition 2.* URL: https://searchdatamanagement.techtarget.com/definition/MongoDB.

[8] *Server side Definition.* URL: https://www.pluralsight.com/blog/software-development/front-end-vs-back-end.

[9] *RASA Introduction.* URL: https://learning.rasa.com/conversational-ai-with-rasa/introduction-to-rasa/.

[10] *RASA Actions Introduction.* URL: https://rasa.com/docs/action-server/.

[11] *Node Introduction.* URL: https://nodejs.org/en/about.

[12] *Express Introduction.* URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs.

[13] *Mongoose Introduction*. URL: https://www.mongodb.com/developer/languages/javascript/mongoose-versus-nodejs-driver/.

[14] *BBC API*. URL: https://newsapi.org/.

[15] *Google People Also Ask - API*. URL: https://pypi.org/project/people-also-ask/.

[16] *RASA Architecture*. URL: https://rasa.com/docs/rasa/arch-overview/.

[17] *RASA Stories*. URL: https://rasa.com/docs/rasa/stories.

[18] *RASA Rules*. URL: https://rasa.com/docs/rasa/rules.

[19] *Chatbot Widget*. URL: https://github.com/JiteshGaikwad/Chatbot-Widget.

[20] *RASA Action Server*. URL: https://rasa.com/docs/rasa/action-server.

[21] *RASA SDK Action*. URL: https://rasa.com/docs/rasa/action-server/sdk-actions.

[22] *RASA SDK Tracker*. URL: https://rasa.com/docs/rasa/action-server/sdk-tracker.

[23] *RASA SDK Dispatcher*. URL: https://rasa.com/docs/rasa/action-server/sdk-dispatcher.

List of Figures

List of Tables