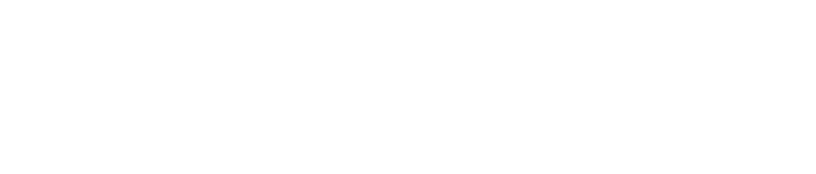
**Mini plan d'action du projet**



**I**

nstitut

**N**

ational de

**S**

tatistique

Et d’

**E**

conomie

**A**

ppliquée

**SUJET**

**Conception et réalisation d’un Telegram chatbot pour aider les processus CRM**

* FADOUL YOUSSEF
* Habib Allah Othmane
* Atraoui chaimaa
* Laabadia othmane

Année Universitaire

2021/2022

Préparé par

:

Dsdsd

dsd

Sous la direction de

:

**Mme**

**Milouda BOUICHOU**

1. **Introduction**

De nos jours, 90% des internautes utilisent au quotidien une messagerie instantanée. les social média constituent les premiers canaux de contacts. Parmi les 5 applications les plus téléchargées on trouve Telegram. C’est ce qui permet à ses utilisateurs d'échanger en toute confidentialité, à l'instar son concurrent WhatsApp, leurs messages -texte, audio, photos, vidéos- de manière chiffrée, c'est-à-dire illisible par un tiers. Parler à une personne est une tâche aisée pour les humains. En est-il de même pour une machine ? Ceci définit la problématique.

Dans le cadre de notre formation au sein de l’INSEA, nous avons travaillé sur un projet du module projet d’innovation afin de pratiquer les différents concepts de ce module. Notre projet a pour objectif la réalisation d’un chatbot sur Telegram pour aider les processus CRM dans notre entreprise et pour optimiser le traitement des clients dans nos front offices.

Dans ce rapport nous allons vous présenter la conception et les étapes de réalisation et déploiement de notre chatbot.

1. **Conception et réalisation du chatbot :**

Après avoir défini les problèmes et préciser les besoins auxquels notre projet doit répondre, nous arrivons à la phase de conception. Dans cette section, nous présentons quelques diagrammes de modélisation en explicitant leurs rôles. Nous allons principalement présenter les diagrammes qui définissent les principales fonctionnalités de notre système sans se préoccuper de comment il va le faire .

1. Introduction

Notre projet est de réaliser un système qui a pour objectif d’améliorer le service clientèle et d’aider les processus CRM dans notre entreprise afin optimiser le traitement des clients dans nos front offices.

Ce diagramme montre tous les éléments qui interagissent entre eux afin de garantir la mise en marche de notre système.

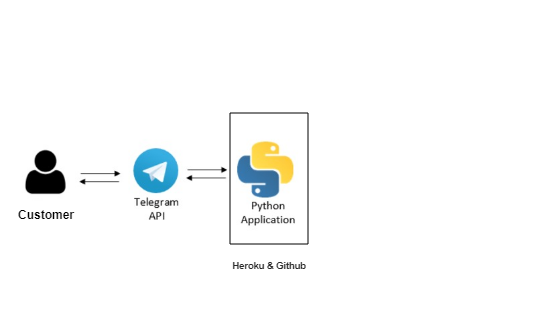


Figure 1- Diagramme de déploiement

1. conception du projet

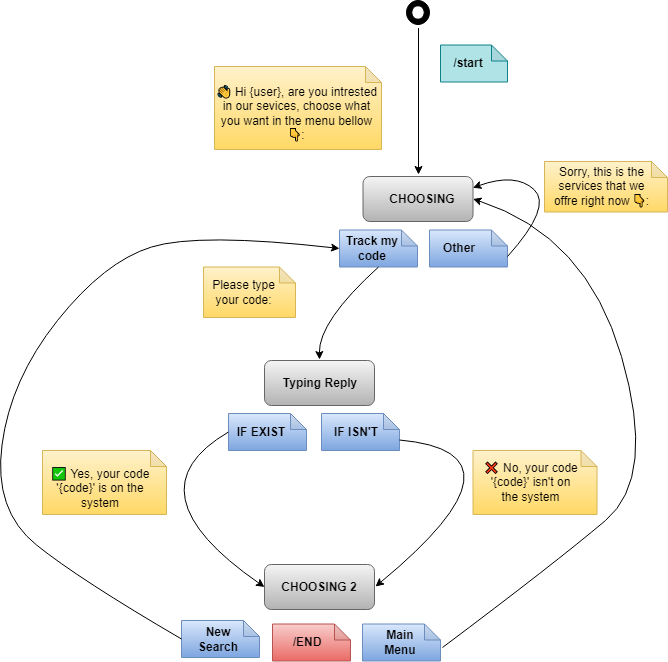
Pour l'usage courant, le chatbot sera une simple implémentation d'une conversation automatisée selon le schéma qui sera décrit ci-dessous.

Figure 2- Shema de la logique du chatbot

La sortie souhaitée est un chatbot Telegram qui fonctionne comme suit :

1. The Customer can access the chatbot via Telegram.
2. The chatbot has a name : Chatbot-DS-(a chosen code)
3. There is an automatic message displayed explaining the purpose of this chatbot.
4. The Customer initiates the conversation by sending  a greeting formula.
5. The chatbot replies by offering button options.
6. If Customer types something, the chatbot replies :”Sorry! Did you mean ?” with a button for the Main menu.
7. Options at the Main Menu are :Track my code, Other
8. If Customer chooses “Track my code”: the chatbot replies:”Please type your code”
9. If Customer types a code that exist in the list of code below then the chatbots replies : “Your code is on the system”, with 2 options : “End”, “Main Menu”, “New search”
10. If Costumer ends conversation then the chatbot is idle until triggered
11. If Customer chooses “New search” then go to step 8
12. If the code typed is not in the list then the chatbot replies :“Your code is on the system”, with 2 options : “End”, “Main Menu”, “New search”
13. **Réalisation du chatbot :**

Dans cette partie, nous allons enregistrer notre bot auprès de Telegram, configurer notre environnement de codage local et écrire le code du bot. Dans la deuxième partie, nous verrons comment héberger le bot sur Heroku, un service basé sur le cloud utile pour héberger de petites applications Web.

* 1. Enregistrer un nouveau bot avec Telegram :

Pour enregistrer un nouveau bot, accédez à l'application Telegram sur votre téléphone et suivez ces étapes :

1. Dans la barre de recherche, recherchez le "BotFather". "BotFather" est un bot qui vous assistera dans la création et la gestion de tous vos bots

2. Démarrez BotFather et tapez /help. Cela fait apparaître toutes les commandes possibles que ce bot peut gérer

3. Tapez /newbot ou cliquez sur cette commande dans cette liste

4. Suivez les instructions pour configurer un nouveau bot qui consiste essentiellement à choisir un nom et un nom d'utilisateur pour votre bot. [Astuce : Choisissez un nom d'utilisateur correspondant à ce que votre bot peut faire]

5. Une fois le bot créé avec succès, le BotFather renverra un jeton pour accéder à l'API HTTP. Enregistrez ce jeton en toute sécurité avec vous. Nous utiliserons ce jeton pour autoriser notre bot depuis le script python

Une image contenant texte, batterie, capture d’écran

Description générée automatiquement

Figure 3- creation d'un bot avec botfather

* 1. Mise en place d'un nouvel environnement virtuel Python :

Nous utiliserons **virtualenv** pour notre projet. Suivez ces étapes pour créer un environnement virtuel Python à l'aide de **virtualenv** :

$ mkdir telegram-bot && cd telegram-bot

Créer un nouvel environnement virtuel et l'activer :

$ virtualenv venv   
$ source venv/bin/activate

Nous utiliserons la bibliothèque **python-telegram-bot** pour ce projet. Cette bibliothèque fournit une interface Python pure pour **l'API Telegram Bot**. Pour installer cette bibliothèque, utilisez la commande suivante

$ pip install -U python-telegram-bot

* 1. Initialisation d'un répertoire Git:

Un répertoire git doit être initialisé dans le dossier du projet car **Heroku** utilise un référentiel git local pour déployer le code dans son service cloud. Alors que la télécommande Heroku est généralement ajoutée au référentiel git local pour pousser la validation, pour ce projet, nous ajouterons une télécommande de Github et connecterons davantage le référentiel Github à l'application Heroku.

Pour initialiser un dépôt git local, dans votre dossier de projet :

$ git init

Créez également un fichier **.gitignore**. Le fichier .gitignore specifié le nom des fichiers qui doivent être ignorés de tous les commits. Nous pouvons mettre à jour ce fichier plus tard

$ touch .gitignore

Nous devons maintenant connecter notre répertoire local à notre référentiel Github qu’on va créer.

$ git remote add origin URL-to-your-Github-repository $ git pull origin master

Le répertoire local est maintenant prêt à pousser les commits vers le répertoire Github

* 1. Création de code python :

Dans cette section, nous allons examiner les différentes parties du code et essayer de comprendre ce qui se passe :

import logging

from typing import Dict

import os

from dotenv import load\_dotenv

from telegram import ReplyKeyboardMarkup, Update, ReplyKeyboardRemove

from telegram.ext import (

    Updater,

    CommandHandler,

    MessageHandler,

    Filters,

    ConversationHandler,

    CallbackContext,

)

Tout d'abord, nous importons les classes nécessaires à partir de la bibliothèque python-telegram-bot. La première ligne importe les différents gestionnaires qui seront nécessaires pour ce programme. Les lignes qui suivent importe les modules associés à la création du clavier en ligne. Comme il s'agit d'un chatbot contextuel, seuls les claviers en ligne seront utilisés pour communiquer avec le bot.

load\_dotenv()

Pour importer les variables d’environnements comme Token, APPNAME, PORT.

# Enable logging

logging.basicConfig(

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s', level=logging.INFO

)

Activer la journalisation pour enregistrer les diffents utilisateurs.

# the list of code in our company system.

list\_code= ['A111',"A112","A113","A114",'A115',"A116","A117","A118"]

Cette liste représente la liste de code enregistrer dans le système de l’organisation.

logger = logging.getLogger(\_\_name\_\_)

# States

CHOOSING, TYPING\_REPLY, TYPING\_CHOICE, CHOOSING2= range(4)

"States" est une liste de toutes les étapes de conversation qui seront utilisées ultérieurement dans la configuration de ConversationHandler. "Données de rappel" est une liste de rappels qui sont renvoyés par le clavier en ligne lorsque vous appuyez dessus. Ils seront renvoyés par diverses fonctions qui formeront notre logique de conversation. Ici, nous initialisons simplement ces variables pour une utilisation ultérieure.

reply\_keyboard = [

    ['Track my code', 'Other'],

    ['End'],

]

markup = ReplyKeyboardMarkup(reply\_keyboard, one\_time\_keyboard=True)

reply\_keyboard2 = [

    ['Main menu', 'Search'],

    ['End'],

]

markup2 = ReplyKeyboardMarkup(reply\_keyboard2, one\_time\_keyboard=True)

Dans cette partie de code on a créé 2 reply\_keyboard qui constituent le menu principale et secondaire, dans le premier on a le bouton ‘**Track my code**’ qui donne à l’utilisateur la possibilité pour vérifier si leur code est déjà dans le system de l’organisme ou non, ‘**Other**’ pour voir les autres services et ‘**End**’. Dans le deuxième menu on a ‘Main menu’ pour revenir au menu principale, ‘**Search**’ qui donne à l’utilisateur la possibilité pour vérifier si leur code est déjà dans le system de l’organisme ou non et ‘**End**’ pour quitter la conversation.

def start(update: Update, context: CallbackContext) -> int:

    """Send message on `/start`."""

    # Get user that sent /start and log his name

    user = update.message.from\_user

    logger.info("User %s started the conversation.", user.first\_name)

    # Send message with text and appended InlineKeyboard

    update.message.reply\_text(f"👋 Hi {user.first\_name}, are you intrested in our sevices, choose what you want in the menu bellow 👇:", reply\_markup=markup)

    # Tell ConversationHandler that we're in state `FIRST` now

    return CHOOSING

la fonction start() est la première fonction appelée depuis le ConversationHandler que nous définirons plus tard. Il est activé en réponse à la commande /start. Une fois le ConversationHandler activé, il utilise les variables "State" renvoyées par chaque fonction pour contrôler le flux de conversation.

def start\_over(update: Update, context: CallbackContext) -> int:

    """Just to handel the case when the customers type sommeting"""

    # Send message with text and appended InlineKeyboard

    update.message.reply\_text("Sorry! Did you mean ? (Choose from buttons bellow 👇)", reply\_markup=markup)

    # Tell ConversationHandler that we're in state `FIRST` now

    return CHOOSING

la fonction start\_over() c’est la même chose que la fonction précède mais c’est juste pour gérer le cas où les clients tapent quelque chose.

def track\_mycode(update: Update, context: CallbackContext) -> int:

    """Ask the user for info about the code to track."""

    update.message.reply\_text(f'Please type your code:')

    return TYPING\_REPLY

la fonction track\_mycode() Demandez à l'utilisateur des informations sur le code à suivre.

def received\_information(update: Update, context: CallbackContext) -> int:

    """Store info provided by user and ask for the next category."""

    user\_data = context.user\_data

    text = update.message.text

    if text in list\_code:

        update.message.reply\_text(

            f"✅ Yes, your code '{text}' is on the system",

            reply\_markup=markup2,

        )

    else:

        update.message.reply\_text(

            f"❌ No, your code '{text}' isn't on the system",

            reply\_markup=markup2,

        )

    return CHOOSING2

la fonction received\_information() sert à stocker les informations fournies par l'utilisateur et demander la catégorie suivante : si le code est déjà enregistrer dans le system le message suivant va s’afficher « "✅ Yes, your code '{text}' is on the system », sinon le message suivant va s’afficher "❌ No, your code '{text}' isn't on the system".

def other(update: Update, context: CallbackContext) -> int:

    """Display the gathered info and end the conversation."""

    user\_data = context.user\_data

    update.message.reply\_text(

        "Sorry, this is the services that we offre right now 👇:",

        reply\_markup=markup

    )

    return CHOOSING

la fonction Other() va offrir les différents autres sévices qui existe, dans notre cas il n’y a aucun autre service donc le message suivant s’affiche : "Sorry, this is the services that we offre right now 👇:"

def end(update: Update, context: CallbackContext) -> int:

    """Display the gathered info and end the conversation."""

    user\_data = context.user\_data

    update.message.reply\_text(

        f"Bye! (if you want to talk to me just type '/start')"

    )

    return ConversationHandler.END

Ces fonctions sont suivies de la fonction end(). La fonction end est appelée lorsque le flux logique dans le ConversationHandler atteint la fin de la conversation. Il renvoie le signal ConversationHandler.END, qui met effectivement fin à la conversation.

def main() -> None:

    """Run the bot."""

    # setting to appropriate values

    TOKEN = os.getenv("TOKEN")

    APPNAME = os.getenv("APPNAME")

    # set PORT to be used with Heroku

    PORT = os.environ.get('PORT')

    # the Updater is the essentiel element to run the bot.

    updater = Updater(TOKEN, use\_context=True)

    # Get the dispatcher to register handlers

    dispatcher = updater.dispatcher

    # Add conversation handler with the states CHOOSING, TYPING\_CHOICE and TYPING\_REPLY

    conv\_handler = ConversationHandler(

        entry\_points=[CommandHandler('start', start)],

        states={

            CHOOSING: [

                MessageHandler(

                    Filters.regex('^(Track my code|Search)$'), track\_mycode

                ),

                MessageHandler(

                    Filters.regex('^(Hi|hi|Hello|hello)$'), start

                ),

                MessageHandler(

                    Filters.regex('^Other$'), other

                ),

                MessageHandler(

                    Filters.text, start\_over

                )

            ],

            TYPING\_REPLY: [

                MessageHandler(

                    Filters.text & ~(Filters.command | Filters.regex('^End$')),

                    received\_information,

                )

            ],

            CHOOSING2: [

                MessageHandler(

                    Filters.regex('^Main menu$'),

                    start,

                ),

                MessageHandler(

                    Filters.regex('^Search$'),

                    track\_mycode,

                )

            ],

            TYPING\_CHOICE: [

                MessageHandler(

                    Filters.regex('^Search$'),

                    track\_mycode,

                )

            ],

        },

        fallbacks=[MessageHandler(Filters.regex('^End$'), end)],

    )

    dispatcher.add\_handler(conv\_handler)

    # starting webhook and setting it up with heroku app

    #updater.start\_webhook(listen="0.0.0.0",

    #                        port=(PORT),

    #                        url\_path=TOKEN)

    #updater.bot.setWebhook(

    #  "https://{}.herokuapp.com/{}".format(APPNAME, TOKEN)

    #)

    # Start the Bot

    updater.start\_polling()

    # Run the bot until you press Ctrl-C or the process receives SIGINT,

    # SIGTERM or SIGABRT. This should be used most of the time, since

    # start\_polling() is non-blocking and will stop the bot gracefully.

    updater.idle()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

La classe Updater récupère en permanence les nouvelles mises à jour de Telegram et les transmet à la classe Dispatcher. Une fois l'objet Updater créé, il est utilisé pour créer un objet Dispatcher et ils sont ensuite liés ensemble dans une file d'attente. Différents types de gestionnaires peuvent être enregistrés auprès du répartiteur, qui trie ensuite toutes les mises à jour reçues de Telegram en fonction des gestionnaires enregistrés. Par exemple, nous ajoutons ici un ConversationHandler au répartiteur.

Le ConversationHandler gère quatre collections d'autres gestionnaires. Dans cet exemple, trois de ces collections sont utilisées, à savoir entry\_points, states et fallbacks.

La collection entry\_point est une liste utilisée pour initier la conversation. Dans cet exemple, une classe CommandHandler est utilisée qui répond à la commande "start".

La collection d'états est un dictionnaire contenant les différentes étapes de conversation et un ou plusieurs gestionnaires associés. Dans cet exemple, toutes les étapes de conversation sont associées à CallbackQueryHandler puisque toutes nos mises à jour de l'application se présentent sous la forme de rappels associés à l'appui sur un bouton du clavier en ligne.

1. **Simulation d’une conversation du chatbot :**