ChatGPT

Pacote do Bot do Telegram – Guia Espiritual 📦

A seguir está o **pacote ZIP** contendo todos os scripts e arquivos necessários para o funcionamento completo do bot do Telegram, estruturado de forma modular (exceto o banco de dados, que não está incluído). Cada arquivo/diretório listado é acompanhado de seu conteúdo. Você pode baixar o pacote completo ou utilizar os conteúdos abaixo:

Estrutura de Arquivos

- **chatbot_unificado.py** – Script principal que inicializa o bot com um `ConversationHandler`.
- **.env** – Arquivo de configuração de exemplo com as chaves do Telegram, e-mail e Google Calendar.
- **handlers/** – Pacote de handlers modular:
- **agendamento.py** – Lógica de agendamento de atendimentos (calendário, seleção de horário, confirmação).
- **consentimento.py** – Fluxo inicial de consentimento LGPD.
- **feedback.py** – Fluxo de coleta de feedback do usuário.
- **menu.py** – Manipulação do menu principal e navegação de opções.
- **utils.py** – Funções utilitárias (envio de e-mail de confirmação, etc.).
- **messages.py** – Módulo para leitura das mensagens/intents (do banco ou do JSON) e busca de respostas por similaridade.
- **strings.py** – Constantes de interface (textos exibidos para o usuário, layouts de teclado).
- **image/** – Diretório com imagem decorativa opcional (por exemplo, `maria_de_nazare.png`).
- **data/messages.json** – Base de conhecimento de perguntas e respostas (intents) em formato JSON, usada caso não haja banco de dados.
- **json/memory.json** – Memória das intents (mesmo conteúdo de `messages.json`, pode ser usada pelo algoritmo de busca semântica).
- **json/service_account.json** – Credenciais de conta de serviço do Google (para acesso à API do Calendar).
- **txt/conteudo.txt** – Fonte das intents em formato texto (categoria|pergunta|resposta) para referência.
- **requirements.txt** – Lista de dependências do projeto (bibliotecas Python necessárias).

Conteúdo dos Arquivos

.env – Exemplo de Configuração

Este arquivo deve conter as chaves e configurações sensíveis. **Não coloque informações reais ao compartilhar publicamente.** Exemplo de `.env`:

```
TELEGRAM_TOKEN=SEU_TOKEN_AQUI
GOOGLE_CALENDAR_ID=SEU_CALENDAR_ID_AQUI
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_USER=seu.email@gmail.com
EMAIL_PASSWORD=sua_senha
```

```
START_HOUR=8
END_HOUR=17
APPOINTMENT_DURATION_MINUTES=60
MAX_CONCURRENT_APPOINTMENTS=1
MAX_APPOINTMENTS_PER_DAY=5
IMAGE_PATH=image/maria_de_nazare.png
```

chatbot_unificado.py – Script Principal do Bot

Este script carrega as configurações do `.env`, inicializa o bot do Telegram, define o `ConversationHandler` com todos os estados e handlers, e inicia o polling do bot:

```
import os
import logging
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

from telegram.ext import ApplicationBuilder, CommandHandler, MessageHandler,
filters, CallbackQueryHandler, ConversationHandler

# Import handlers and modules
from handlers import agendamento, consentimento, feedback, menu
import messages, strings

# Configure logging
logging.basicConfig(format='%(asctime)s - %(name)s - %(levelname)s - %
(message)s', level=logging.INFO)
logger = logging.getLogger(__name__)

# Telegram Bot Token
TOKEN = os.getenv("TELEGRAM_TOKEN")

# Initialize embedding model and memory for Q&A
messages.initialize_embeddings()

# Create application
application = ApplicationBuilder().token(TOKEN).build()

# Define conversation handler with states
conv_handler = ConversationHandler(
    entry_points=[CommandHandler("start", consentimento.start)],
    states={
        consentimento.CONSENT: [
            CallbackQueryHandler(consentimento.consent_response,
pattern=r"^consent_")
        ],
        menu.MENU: [
            # Main menu text options (reply keyboard)
```

```
            MessageHandler(filters.Regex("^(  Meditação|  Emoções|  Cura|🕊️
Paz Interior|📬 Mensagem|❓ Ajuda|  Agendar Atendimento|  Feedback|  Cancelar
Agendamento|ℹ️ Como funciona|  Sair)$"), menu.handle_main_menu),
            # Submenu and Q&A (inline buttons)
            CallbackQueryHandler(menu.handle_submenu, pattern=r"^submenu_"),
            CallbackQueryHandler(menu.handle_answer, pattern=r"^answer_"),
            CallbackQueryHandler(menu.handle_how_it_works,
pattern=r"^how_it_works$"),
            CallbackQueryHandler(menu.handle_main_menu_callback,
pattern=r"^main_menu$"),
            CallbackQueryHandler(menu.handle_exit, pattern=r"^exit$"),
            # Scheduling and cancellation (inline calendar and buttons)
            CallbackQueryHandler(agendamento.handle_calendar_navigation,
pattern=r"^cal_(prev_month|next_month)$"),
            CallbackQueryHandler(agendamento.handle_day_selected,
pattern=r"^cal_day_"),
            CallbackQueryHandler(agendamento.handle_time_selected,
pattern=r"^cal_time_"),
            CallbackQueryHandler(agendamento.handle_calendar_back,
pattern=r"^cal_back$"),
            CallbackQueryHandler(agendamento.handle_cancel_confirm,
pattern=r"^confirm_cancel_"),
            CallbackQueryHandler(agendamento.handle_cancel_execute,
pattern=r"^execute_cancel_"),
            CallbackQueryHandler(agendamento.handle_cancel_list_page,
pattern=r"^cancel_all_"),
            # Feedback (inline buttons)
            CallbackQueryHandler(feedback.handle_feedback_level_callback,
pattern=r"^feedback_level_"),
            # Also handle direct free-text questions not matching menu
options
            MessageHandler(filters.TEXT & ~filters.COMMAND,
menu.handle_unrecognized)
        ],
        agendamento.AGENDA_NAME: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
agendamento.collect_name)
        ],
        agendamento.AGENDA_PHONE: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
agendamento.collect_phone)
        ],
        agendamento.AGENDA_CAL: [
            CallbackQueryHandler(agendamento.handle_calendar_navigation,
pattern=r"^cal_(prev_month|next_month)$"),
            CallbackQueryHandler(agendamento.handle_day_selected,
pattern=r"^cal_day_"),
            CallbackQueryHandler(agendamento.handle_time_selected,
pattern=r"^cal_time_"),
            CallbackQueryHandler(agendamento.handle_calendar_back,
pattern=r"^cal_back$"),
```

```
            CallbackQueryHandler(menu.handle_main_menu_callback,
pattern=r"^main_menu$")
        ],
        agendamento.AGENDA_EMAIL: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
agendamento.collect_email)
        ],
        feedback.FEEDBACK_TEXT: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
feedback.handle_feedback_text)
        ],
        agendamento.CANCEL_EMAIL: [
            MessageHandler(filters.TEXT & ~filters.COMMAND,
agendamento.handle_cancel_email)
        ],
        agendamento.CANCEL_LIST: [
            CallbackQueryHandler(agendamento.handle_cancel_confirm,
pattern=r"^confirm_cancel_"),
            CallbackQueryHandler(agendamento.handle_cancel_list_page,
pattern=r"^cancel_all_"),
            CallbackQueryHandler(menu.handle_main_menu_callback,
pattern=r"^main_menu$")
        ],
        agendamento.CANCEL_CONFIRM: [
            CallbackQueryHandler(agendamento.handle_cancel_execute,
pattern=r"^execute_cancel_"),
            CallbackQueryHandler(agendamento.handle_cancel_list_page,
pattern=r"^cancel_all_"),
            CallbackQueryHandler(menu.handle_main_menu_callback,
pattern=r"^main_menu$")
        ]
    },
    fallbacks=[
        CommandHandler("apagar_dados", consentimento.prompt_delete),
        CallbackQueryHandler(consentimento.handle_delete_choice,
pattern=r"^(confirm_delete|cancel_delete)$")
    ],
    allow_reentry=True
)

# Add the conversation handler to the application
application.add_handler(conv_handler)

# Start the bot
if __name__ == "__main__":
    logger.info("Starting Telegram bot...")
    application.run_polling()
```

handlers/consentimento.py – Consentimento LGPD

Este handler exibe a mensagem inicial de consentimento LGPD assim que o bot é iniciado (/start) e lida com a resposta do usuário (Aceitar ou Não aceitar). Além disso, implementa o comando `/apagar_dados` para o usuário solicitar a exclusão de seus dados pessoais, com confirmação:

```
import logging
from telegram import InlineKeyboardButton, InlineKeyboardMarkup
import strings

logger = logging.getLogger(__name__)

# Conversation state for consent
CONSENT = 0

def start(update, context):
    """Send LGPD consent message with Accept button when /start is
invoked."""
    # Create inline keyboard for consent
    buttons = [
        [InlineKeyboardButton("  Aceito", callback_data="consent_agree")],
        [InlineKeyboardButton("  Não aceito",
callback_data="consent_decline")]
    ]
    reply_markup = InlineKeyboardMarkup(buttons)
    update.message.reply_text(strings.CONSENT_MSG, reply_markup=reply_markup,
parse_mode="Markdown")
    return CONSENT

def consent_response(update, context):
    """Handle user's response to consent prompt."""
    query = update.callback_query
    data = query.data
    if data == "consent_agree":
        # User accepted terms
        try:
            query.answer()
        except Exception as e:
            logger.warning(f"Error answering callback: {e}")
        # Edit original message to remove buttons
        query.edit_message_text("  *Consentimento LGPD confirmado.*",
parse_mode="Markdown")
        # Send welcome message with main menu
        query.message.reply_text(strings.WELCOME_MSG, reply_markup=
strings.main_menu_keyboard(), parse_mode="Markdown")
        logger.info(f"Usuário {update.effective_user.id} aceitou o
consentimento LGPD.")
        # Proceed to main menu state
        return 1
    else:
```

```
        # User declined consent
        try:
            query.answer()
        except Exception as e:
            logger.warning(f"Error answering callback: {e}")

query.edit_message_text("  Você optou por *não aceitar* os termos. Se mudar
de ideia, envie /start novamente.", parse_mode="Markdown")
        logger.info(f"Usuário {update.effective_user.id} recusou o
consentimento LGPD.")
        # End conversation
        return -1  # ConversationHandler.END

def prompt_delete(update, context):
    """Command /apagar_dados - ask user to confirm data deletion."""
    buttons = [
        [InlineKeyboardButton("  Sim, apagar",
callback_data="confirm_delete"),
         InlineKeyboardButton("  Não, cancelar",
callback_data="cancel_delete")]
    ]
    reply_markup = InlineKeyboardMarkup(buttons)
    update.effective_message.reply_text("⚠ *Deseja realmente apagar todos os
seus dados pessoais?* Esta ação é irreversível.", reply_markup=reply_markup,
parse_mode="Markdown")
    return None  # stay in current state

def handle_delete_choice(update, context):
    """Handle confirmation or cancellation of data deletion."""
    query = update.callback_query
    data = query.data
    if data == "confirm_delete":
        try:
            query.answer()
        except Exception as e:
            logger.warning(f"Error answering callback: {e}")
        # Perform data deletion: clear user data and cancel any appointments
        user_id = update.effective_user.id
        email = context.user_data.get("last_email")
        if email:
            # Cancel all future events for this email
            from handlers import agendamento
            events, total = agendamento.get_user_events(email, page=0,
page_size=1000)
            for event in events:
                try:
                    service = agendamento.get_calendar_service()
                    if service:

service.events().delete(calendarId=agendamento.CALENDAR_ID,
eventId=event['id']).execute()
```

```
                except Exception as e:
                    logger.error(f"Erro ao deletar evento
{event.get('summary')} do usuário {user_id}: {e}")
        # Clear context user data
        context.user_data.clear()
        # Confirm deletion to user
        query.edit_message_text("  *Seus dados pessoais foram apagados.*",
parse_mode="Markdown")
        logger.info(f"Dados pessoais do usuário {user_id} apagados.")
        # Optionally, remain in menu state (user can continue using bot)
        query.message.reply_text(strings.WELCOME_MSG,
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        return 1
    else:
        # cancel_delete
        try:
            query.answer()
        except Exception as e:
            logger.warning(f"Error answering callback: {e}")
        # Just remove the confirmation prompt
        query.edit_message_text("Operação cancelada.", parse_mode="Markdown")
        return None  # remain in current state (likely main menu)
```

handlers/menu.py – Menu Principal e Navegação

Este handler gerencia a exibição e interação com o menu principal (botões de opções) e os submenus de perguntas e respostas para cada categoria. Também captura quaisquer mensagens não reconhecidas e tenta encontrar a melhor resposta por similaridade:

```
import logging
from telegram import ReplyKeyboardMarkup, InlineKeyboardMarkup,
InlineKeyboardButton
import strings, messages
from handlers import agendamento, feedback

logger = logging.getLogger(__name__)

MENU = 1  # conversation state for main menu

def handle_main_menu(update, context):
    """Handle selection from the main menu (reply keyboard)."""
    user_text = update.message.text
    # Reset any ongoing flows data when a main menu item is chosen
    context.user_data.pop("agendamento", None)
    context.user_data.pop("feedback", None)
    context.user_data.pop("cancelamento", None)
    context.user_data.pop("current_month", None)
    logger.info(f"Main menu option selected: {user_text}")
    if user_text == "  Sair":
        # End conversation
```

```
        update.message.reply_text("Até logo!  ",
reply_markup=ReplyKeyboardMarkup([[]], one_time_keyboard=True))
        return -1  # ConversationHandler.END
    if user_text == "  Agendar Atendimento":
        # Start scheduling flow
        return agendamento.start_scheduling(update, context)
    if user_text == "  Feedback":
        # Start feedback flow
        return feedback.start_feedback(update, context)
    if user_text == "  Cancelar Agendamento":
        # Start cancellation flow
        return agendamento.start_cancel_appointment(update, context)
    if user_text == "ℹ Como funciona":
        # Send "how it works" info
        update.message.reply_text(strings.HOW_IT_WORKS,
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        return MENU

# For any other category (e.g. Meditação, Emoções, etc.), show submenu of
questions
    if user_text in strings.MAIN_MENU_CATEGORIES:
        category_tag = strings.MAIN_MENU_CATEGORIES[user_text]
        keyboard_markup, error_msg =
messages.get_submenu_keyboard(category_tag)
        prompt = error_msg if error_msg else f"🔍 **{user_text}
** - Escolha um tópico:"
        update.message.reply_text(prompt, reply_markup=keyboard_markup,
parse_mode="Markdown")
        return MENU
    # If text didn't match any known option, just stay in menu
    return MENU

def handle_submenu(update, context):
    """Handle pagination callbacks for submenu (list of questions)."""
    query = update.callback_query
    data = query.data  # e.g. "submenu_meditacao_1"
    try:
        query.answer()
    except Exception as e:
        logger.debug(f"Callback answer issue: {e}")
    parts = data.rsplit("_", 1)
    if len(parts) != 2:
        logger.error(f"Formato inválido para submenu: {data}")
        return None
    prefix_and_cat, page_str = parts
    category_encoded = prefix_and_cat[len("submenu_"):]
    category = category_encoded.replace("__", "_")
    try:
        page = int(page_str)
    except ValueError:
        logger.error(f"Erro ao converter página: {page_str}")
```

```
        return None
    keyboard_markup, error_msg = messages.get_submenu_keyboard(category,
page)
    title = category.replace("_", " ").title()
    prompt = error_msg if error_msg else f"🔍 **{title}** - Escolha um
tópico:"
    query.edit_message_text(prompt, reply_markup=keyboard_markup,
parse_mode="Markdown")
    logger.info(f"Submenu page {page} for category '{category}' displayed.")
    return None

def handle_answer(update, context):
    """Handle selection of a question from submenu and display the answer."""
    query = update.callback_query
    data = query.data  # e.g. "answer_meditacao_5"
    try:
        query.answer()
    except Exception as e:
        logger.debug(f"Callback answer issue: {e}")
    parts = data.rsplit("_", 1)
    if len(parts) != 2:
        logger.error(f"Formato inválido para answer: {data}")
        query.edit_message_text("⚠ Erro técnico ao processar a resposta.",
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        return MENU
    prefix_and_cat, idx_str = parts
    category_encoded = prefix_and_cat[len("answer_"):]
    category = category_encoded.replace("__", "_")
    try:
        idx = int(idx_str)
    except ValueError:
        logger.error(f"Índice inválido: {idx_str}")
        query.edit_message_text("⚠ Erro técnico ao processar a resposta.",
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        return MENU
    answer_text = messages.get_answer_by_index(category, idx)
    if answer_text:
        # Show answer and remove inline buttons
        query.edit_message_text(answer_text, parse_mode="Markdown")
        # After answer, send main menu prompt again
        query.message.reply_text(strings.CONTINUE_MSG,
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        logger.info(f"Displayed answer for category '{category}', index
{idx}.")
    else:
        query.edit_message_text("⚠ Resposta não encontrada.",
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        logger.error(f"No answer found for category '{category}' index
{idx}.")
    return MENU
```

```
def handle_how_it_works(update, context):
    """Handle inline 'Como funciona' callback (from inline main menu if
used)."""
    query = update.callback_query
    try:
        query.answer()
    except Exception as e:
        logger.debug(f"Callback answer issue: {e}")
    query.edit_message_text(strings.HOW_IT_WORKS, parse_mode="Markdown")
    # After showing info, present main menu again
    query.message.reply_text(strings.WELCOME_MSG,
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
    return MENU

def handle_main_menu_callback(update, context):
    """Handle 'Voltar ao Menu' inline callback from various flows."""
    query = update.callback_query
    try:
        query.answer()
    except Exception as e:
        logger.debug(f"Callback answer issue: {e}")
    # Remove inline buttons from previous message
    try:
        query.edit_message_reply_markup(reply_markup=None)
    except Exception:
        pass
    # Send main menu message
    query.message.reply_text(strings.WELCOME_MSG,
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
    return MENU

def handle_exit(update, context):
    """Handle inline 'Sair' callback (if inline main menu was used)."""
    query = update.callback_query
    try:
        query.answer()
    except Exception as e:
        logger.debug(f"Callback answer issue: {e}")
    query.edit_message_text("Até logo!  ",
reply_markup=ReplyKeyboardMarkup([[]], one_time_keyboard=True))
    return -1  # end conversation

def handle_unrecognized(update, context):

"""Handle any unrecognized text by attempting to find an answer or giving a
default reply."""
    user_msg = update.message.text
    logger.info(f"Unrecognized input: {user_msg}")
    response = messages.find_best_answer(user_msg)
    if response:
        update.message.reply_text(response,
```

```
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        logger.info("Responded with best match from knowledge base.")
    else:
        update.message.reply_text(strings.DID_NOT_UNDERSTAND,
reply_markup=strings.main_menu_keyboard(), parse_mode="Markdown")
        logger.info("Sent default 'não entendi' response.")
    return MENU
```

handlers/agendamento.py – Agendamento de Atendimentos

Este módulo implementa o fluxo completo de **agendar atendimento**, incluindo coleta de nome e telefone, exibição de um calendário interativo para escolha de data, apresentação de horários disponíveis, reserva temporária de horário para evitar conflitos e confirmação final (com registro no Google Calendar e envio de e-mail de confirmação). Também inclui o fluxo de **cancelamento de agendamento** com listagem de eventos do usuário e confirmação de cancelamento:

```
import logging
import os
import re
from datetime import datetime, timedelta
from telegram import InlineKeyboardButton, InlineKeyboardMarkup
from google.oauth2 import service_account
from googleapiclient.discovery import build
from cachetools import TTLCache
import threading

logger = logging.getLogger(__name__)

# Conversation state constants for scheduling and cancellation
AGENDA_NAME = 2
AGENDA_PHONE = 3
AGENDA_CAL = 4
AGENDA_EMAIL = 5
CANCEL_EMAIL = 8
CANCEL_LIST = 9
CANCEL_CONFIRM = 10

# Load configuration from environment
START_HOUR = int(os.getenv("START_HOUR", 8))
END_HOUR = int(os.getenv("END_HOUR", 17))
APPOINTMENT_DURATION_MINUTES = int(os.getenv("APPOINTMENT_DURATION_MINUTES",
60))
MAX_CONCURRENT_APPOINTMENTS = int(os.getenv("MAX_CONCURRENT_APPOINTMENTS",
1))
MAX_APPOINTMENTS_PER_DAY = int(os.getenv("MAX_APPOINTMENTS_PER_DAY", 5))
CALENDAR_ID = os.getenv("GOOGLE_CALENDAR_ID")
SERVICE_ACCOUNT_FILE = "json/service_account.json"

# Redis-like cache for slot reservation and busy info
busy_info_cache = TTLCache(maxsize=100, ttl=120)
```

```
cache_lock = threading.Lock()

def get_calendar_service():
    """Initialize Google Calendar service using service account."""
    try:
        credentials = service_account.Credentials.from_service_account_file(
            SERVICE_ACCOUNT_FILE,
            scopes=["https://www.googleapis.com/auth/calendar"]
        )
        service = build("calendar", "v3", credentials=credentials,
cache_discovery=False)
        return service
    except Exception as e:
        logger.error(f"Erro ao inicializar Google Calendar: {e}")
        return None

def start_scheduling(update, context):
    """Start the appointment scheduling process."""
    # Reset any existing scheduling state
    context.user_data.pop("agendamento", None)
    context.user_data.pop("current_month", None)
    # Prompt for name
    keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
    update.effective_message.reply_text(
        "  **Vamos agendar seu atendimento!**\\n\\nPor favor, digite seu
**nome completo**:",
        reply_markup=InlineKeyboardMarkup(keyboard),
        parse_mode="Markdown"
    )
    context.user_data["agendamento"] = {"etapa": "nome"}
    logger.info("Agendamento iniciado - solicitando nome.")
    return AGENDA_NAME

def collect_name(update, context):
    """Collect the user's name for the appointment."""
    name = update.effective_message.text.strip()
    keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
    if not name:
        update.effective_message.reply_text(
            "  Por favor, digite um nome válido.",
            reply_markup=InlineKeyboardMarkup(keyboard),
            parse_mode="Markdown"
        )
        return None
    context.user_data["agendamento"]["nome"] = name
    context.user_data["agendamento"]["etapa"] = "telefone"
    update.effective_message.reply_text(
        f"Ótimo, **{name}**! Agora, por favor, informe seu **telefone** no
formato:\n\n☞ *(DD)9XXXX-XXXX*\nExemplo: (99)99999-9999",
```

```
        reply_markup=InlineKeyboardMarkup(keyboard),
        parse_mode="Markdown"
    )
    logger.info(f"Nome recebido para agendamento: {name}. Solicitando
telefone.")
    return AGENDA_PHONE

def collect_phone(update, context):
    """Collect the user's phone number for the appointment."""
    phone = update.effective_message.text.strip()
    keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
    if not re.match(r"^\\(\\d{2}\\)9\\d{4}-\\d{4}$", phone):
        update.effective_message.reply_text(
            "  **Formato inválido!** Use: *(DD)9XXXX-XXXX*",
            reply_markup=InlineKeyboardMarkup(keyboard),
            parse_mode="Markdown"
        )
        return None
    context.user_data["agendamento"]["telefone"] = phone
    context.user_data["agendamento"]["etapa"] = "calendario"
    # Show calendar for date selection
    show_month_calendar(update, context)
    logger.info(f"Telefone recebido: {phone}. Exibindo calendário para
seleção de data.")
    return AGENDA_CAL

def show_month_calendar(update, context):
    """Display an inline calendar for the current or selected month."""
    now = datetime.now()
    current_date = now.date()
    current_year, current_month = now.year, now.month
    if "current_month" in context.user_data:
        year = context.user_data["current_month"]["year"]
        month = context.user_data["current_month"]["month"]
        # Ensure not showing past month
        if year < current_year or (year == current_year and month <
current_month):
            year, month = current_year, current_month
            context.user_data["current_month"] = {"year": year, "month":
month}
    else:
        year, month = current_year, current_month
        context.user_data["current_month"] = {"year": year, "month": month}
    # Determine date range for busy info (first day of month to first day of
next month)
    start_date = datetime(year, month, 1).date()
    if month == 12:
        end_date = datetime(year + 1, 1, 1).date()
    else:
        end_date = datetime(year, month + 1, 1).date()
```

```
    busy_info = get_busy_info(start_date, end_date)
    busy_days_total = busy_info.get("total", {})
    month_name = start_date.strftime("%B")
    # Build calendar keyboard
    keyboard = [
        [
            InlineKeyboardButton("◀", callback_data="cal_prev_month" if year
> current_year or month > current_month else "ignore"),
            InlineKeyboardButton(f"{month_name} {year}",
callback_data="ignore"),
            InlineKeyboardButton("▶", callback_data="cal_next_month")
        ],
        [InlineKeyboardButton(day, callback_data="ignore") for day in ["Seg",
"Ter", "Qua", "Qui", "Sex", "Sáb", "Dom"]]
    ]
    month_cal = __import__('calendar').monthcalendar(year, month)
    for week in month_cal:
        week_buttons = []
        for day in week:
            if day == 0:
                # Empty day (padding)
                week_buttons.append(InlineKeyboardButton(" ",
callback_data="ignore"))
            else:
                is_past = datetime(year, month, day).date() < current_date
                is_full = busy_days_total.get(day, 0) >=
MAX_APPOINTMENTS_PER_DAY
                if is_past:
                    btn_text = f"({day})"
                    callback = "ignore"
                elif is_full:
                    btn_text = f"{day}X"
                    callback = "ignore"
                else:
                    btn_text = str(day)
                    callback = f"cal_day_{year}_{month}_{day}"
                week_buttons.append(InlineKeyboardButton(btn_text,
callback_data=callback))
        keyboard.append(week_buttons)
    keyboard.append([InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")])
    message_text = "  **Selecione um dia disponível:**\\n(X indica agenda
cheia, () indica data passada)"
    if hasattr(update, "callback_query") and update.callback_query:
        update.callback_query.message.edit_text(message_text,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
    else:
        update.effective_message.reply_text(message_text,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
    context.user_data["agendamento"]["etapa"] = "calendario"
    return None
```

```
def handle_calendar_navigation(update, context):
    """Handle next/previous month navigation in calendar."""
    query = update.callback_query
    data = query.data
    try:
        query.answer()
    except Exception:
        pass
    current = context.user_data.get("current_month", {})
    year = current.get("year", datetime.now().year)
    month = current.get("month", datetime.now().month)
    if data == "cal_prev_month":
        month -= 1
        if month == 0:
            month = 12
            year -= 1
    elif data == "cal_next_month":
        month += 1
        if month == 13:
            month = 1
            year += 1
    context.user_data["current_month"] = {"year": year, "month": month}
    # Refresh calendar display
    show_month_calendar(update, context)
    return None

def handle_day_selected(update, context):
    """Handle selection of a day from the calendar."""
    query = update.callback_query
    data = query.data
    try:
        query.answer()
    except Exception:
        pass
    parts = data.split("_")
    if len(parts) == 5:
        _, _, year_str, month_str, day_str = parts
        try:
            year = int(year_str); month = int(month_str); day = int(day_str)
        except ValueError:
            logger.error(f"Erro ao converter data: {data}")
            return None
        show_day_schedule(update, context, year, month, day)
    else:
        logger.error(f"Formato inválido para cal_day: {data}")
    return None

def show_day_schedule(update, context, year, month, day):
    """Show available time slots for a selected day."""
    try:
```

```
        logger.info(f"Mostrando horários para {day}/{month}/{year}")
        selected_date = datetime(year, month, day).date()
        current_datetime = datetime.now()
        current_date = current_datetime.date()
        current_time = current_datetime.time()
        if selected_date < current_date:
            update.callback_query.message.edit_text(
                f"  **{day}/{month}/{year} é uma data passada.**\
\nPor favor, escolha uma data futura.",
                reply_markup=InlineKeyboardMarkup([
                    [InlineKeyboardButton("  Voltar ao Calendário",
callback_data="cal_back")],
                    [InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]
                ]),
                parse_mode="Markdown"
            )
            return
        busy_info = get_busy_info(selected_date, selected_date +
timedelta(days=1))
        busy_slots = busy_info.get("hourly", {}).get(day, [])
        busy_total = busy_info.get("total", {}).get(day, 0)
        if busy_total >= MAX_APPOINTMENTS_PER_DAY:
            update.callback_query.message.edit_text(
                f"  **O dia {day}/{month}/{year} está totalmente ocupado.**\
\nEscolha outra data:",
                reply_markup=InlineKeyboardMarkup([
                    [InlineKeyboardButton("  Voltar ao Calendário",
callback_data="cal_back")],
                    [InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]
                ]),
                parse_mode="Markdown"
            )
            return
        # Generate slots from START_HOUR to END_HOUR
        possible_slots = []
        slot_time = datetime.min.replace(hour=START_HOUR, minute=0)
        end_time_limit = datetime.min.replace(hour=END_HOUR, minute=0)
        while slot_time.time() <= end_time_limit.time():
            possible_slots.append(slot_time.strftime("%H:%M"))
            slot_time += timedelta(minutes=APPOINTMENT_DURATION_MINUTES)
        keyboard = []
        row = []
        max_cols = 3
        for slot in possible_slots:
            slot_h, slot_m = slot.split(":")
            slot_time_obj = datetime.combine(selected_date,
datetime.min.time()).replace(hour=int(slot_h), minute=int(slot_m))
            slot_text = slot
            callback_data = f"cal_time_{year}_{month}_{day}
```

```
_{slot.replace(':', '')}"
            if (selected_date == current_date and slot_time_obj.time() <=
current_time) or (slot in busy_slots):
                slot_text = f"{slot}X"
                callback_data = "ignore"
            row.append(InlineKeyboardButton(slot_text,
callback_data=callback_data))
            if len(row) == max_cols or slot == possible_slots[-1]:
                keyboard.append(row)
                row = []
        # Add navigation and menu buttons
        nav_buttons = [InlineKeyboardButton("  Voltar ao Calendário",
callback_data="cal_back"),
                       InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]
        keyboard.append(nav_buttons)
        if sum(1 for r in keyboard[:-1] for btn in r if btn.callback_data !=
"ignore") == 0:
            update.callback_query.message.edit_text(
                f"  **{day}/{month}/{year} totalmente ocupado**\\nEscolha
outra data:",
                reply_markup=InlineKeyboardMarkup([
                    [InlineKeyboardButton("  Voltar ao Calendário",
callback_data="cal_back")],
                    [InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]
                ]),
                parse_mode="Markdown"
            )
        else:
            update.callback_query.message.edit_text(
                f"  **Horários disponíveis para {day}/{month}/{year}:**\\n(X
indica horário ocupado ou passado)",
                reply_markup=InlineKeyboardMarkup(keyboard),
                parse_mode="Markdown"
            )
            context.user_data["agendamento"]["etapa"] = "horario"
            context.user_data["appointment_details"] = {"year": year,
"month": month, "day": day}
            logger.info(f"Agendamento - horários disponíveis mostrados para
{day}/{month}/{year}.")
    except Exception as e:
        logger.error(f"Erro em show_day_schedule: {e}")
        update.callback_query.message.reply_text("  Erro ao listar horários
disponíveis.", parse_mode="Markdown")

def handle_calendar_back(update, context):
    """Handle 'Voltar ao Calendário' action to show the month view again."""
    try:
        update.callback_query.answer()
    except Exception:
```

```
        pass
    # Show the stored month calendar again
    show_month_calendar(update, context)
    return None

def reserve_slot(year, month, day, hour):
    """Reserve a time slot temporarily to avoid double booking (returns True
if reserved)."""
    slot_key = f"slot:{year}:{month}:{day}:{hour}"
    ttl_seconds = 300  # hold reservation for 5 minutes
    with cache_lock:
        reserved = slot_key not in busy_info_cache
        if reserved:
            busy_info_cache[slot_key] = True
    return reserved

def release_slot(year, month, day, hour):
    """Release a previously reserved time slot."""
    slot_key = f"slot:{year}:{month}:{day}:{hour}"
    with cache_lock:
        if slot_key in busy_info_cache:
            del busy_info_cache[slot_key]

def handle_time_selected(update, context):
    """Handle selection of a time slot for an appointment."""
    query = update.callback_query
    data = query.data  # like "cal_time_2023_9_15_1400"
    try:
        query.answer()
    except Exception:
        pass
    parts = data.split("_")
    if len(parts) == 6:
        _, _, year_str, month_str, day_str, hour_str = parts
        try:
            year = int(year_str); month = int(month_str); day = int(day_str)
        except ValueError:
            logger.error(f"Erro ao converter data/hora: {data}")
            return None
        hora_formatada = f"{hour_str[:2]}:{hour_str[2:]}"
        if not reserve_slot(year, month, day, hora_formatada):
            query.edit_message_text(
                f"  **Horário {hora_formatada} em {day}/{month}/{year} foi
reservado por outro usuário.**\\nEscolha outro horário:",
                reply_markup=InlineKeyboardMarkup([
                    [InlineKeyboardButton("  Voltar ao Calendário",
callback_data="cal_back")],
                    [InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]
                ]),
                parse_mode="Markdown"
```

```
            )
            return None
        context.user_data["agendamento"]["etapa"] = "email"
        context.user_data["agendamento"]["horario"] = f"{day}/{month}/{year}
{hora_formatada}"
        context.user_data["appointment_details"] = {"year": year, "month":
month, "day": day, "hour": hour_str}
        query.edit_message_text(
            "  Por favor, digite seu e-mail para confirmação:",
            reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("
Voltar ao Menu", callback_data="main_menu")]]),
            parse_mode="Markdown"
        )
        logger.info(f"Horário selecionado: {hora_formatada} em {day}/{month}/
{year}. Solicitando e-mail.")
        return AGENDA_EMAIL
    else:
        logger.error(f"Formato inválido para cal_time: {data}")
        return None

def collect_email(update, context):
    """Collect the user's email address and finalize the appointment."""
    email = update.effective_message.text.strip()
    keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
    # Validate email format
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$'
    if not re.match(pattern, email):
        last_email = context.user_data.get("last_email", "")
        prompt = "  E-mail inválido. Por favor, digite novamente:"
        if last_email:
            prompt += f"\\n\\nSugestão: {last_email}"
        update.effective_message.reply_text(prompt,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
        context.user_data["last_email"] = email  # store attempt
        return None
    context.user_data["agendamento"]["email"] = email.lower()
    context.user_data["last_email"] = email.lower()
    # Finalize appointment (create calendar event, send confirmation)
    details = context.user_data.get("appointment_details", {})
    year = details.get("year"); month = details.get("month"); day =
details.get("day"); hour = details.get("hour")
    if not (year and month and day and hour):
        logger.error("Dados do agendamento incompletos ao confirmar.")

update.effective_message.reply_text("  Ocorreu um erro ao confirmar o
agendamento. Tente novamente.", reply_markup=InlineKeyboardMarkup(keyboard),
parse_mode="Markdown")
        context.user_data.pop("agendamento", None)
        return 1
    try:
```

```
            confirm_appointment(update, context, year, month, day, hour)
        except Exception as e:
            logger.error(f"Erro ao confirmar agendamento: {e}")
            context.user_data.pop("agendamento", None)
            update.effective_message.reply_text("  Ocorreu um erro ao finalizar
seu agendamento.", parse_mode="Markdown")
            return 1
        # Clear scheduling data and return to main menu
        context.user_data.pop("agendamento", None)
        context.user_data.pop("appointment_details", None)
        return 1

    def confirm_appointment(update, context, year, month, day, hour):
        """Create the calendar event and send confirmation message (with
    email)."""
        agendamento = context.user_data.get("agendamento", {})
        hora_formatada = f"{hour[:2]}:{hour[2:]}"
        if agendamento.get("etapa") != "confirmacao":
            # Ensure stage is correct (should be confirmacao if being finalized)
            agendamento["etapa"] = "confirmacao"
        if "email" not in agendamento:
            release_slot(year, month, day, hora_formatada)
            raise Exception("E-mail do usuário não disponível para confirmação.")
        # Double-check availability
        busy_info_check = get_busy_info(datetime(year, month, day).date(),
    datetime(year, month, day).date() + timedelta(days=1))
        if hora_formatada in busy_info_check.get("hourly", {}).get(day, []) or
    busy_info_check.get("total", {}).get(day, 0) >= MAX_APPOINTMENTS_PER_DAY:
            release_slot(year, month, day, hora_formatada)
            update.effective_message.reply_text(
                f"  **Horário {hora_formatada} em {day}/{month}/{year} não está
    mais disponível.**\\nEscolha outro horário:",
                reply_markup=InlineKeyboardMarkup([
                    [InlineKeyboardButton("  Voltar ao Calendário",
    callback_data="cal_back")],
                    [InlineKeyboardButton("  Voltar ao Menu",
    callback_data="main_menu")]
                ]),
                parse_mode="Markdown"
            )
            return
        # Create Google Calendar event
        service = get_calendar_service()
        if not service:
            release_slot(year, month, day, hora_formatada)
            raise Exception("Google Calendar não disponível")
        start_time = datetime(year, month, day, int(hour[:2]), int(hour[2:]))
        end_time = start_time + timedelta(minutes=APPOINTMENT_DURATION_MINUTES)
        event_body = {
            "summary": f"Atendimento - {agendamento['nome']}",
            "description": f"Nome: {agendamento['nome']}\\nTelefone:
```

```
{agendamento['telefone']}\\nE-mail: {agendamento['email']}",
        "start": {"dateTime": start_time.isoformat(), "timeZone": "America/
Sao_Paulo"},
        "end": {"dateTime": end_time.isoformat(), "timeZone": "America/
Sao_Paulo"}
    }
    created_event = service.events().insert(calendarId=CALENDAR_ID,
body=event_body).execute()
    logger.info(f"Evento criado no calendário:
{created_event.get('htmlLink')}")
    release_slot(year, month, day, hora_formatada)
    # Send confirmation email
    from handlers import utils
    email = agendamento['email']; name = agendamento['nome']
    appointment_time = f"{day}/{month}/{year} às {hora_formatada}"
    email_sent = True
    try:
        utils.send_confirmation_email(email, name, appointment_time)
    except Exception as e:
        email_sent = False
        logger.error(f"Erro ao enviar e-mail de confirmação: {e}")
    # Send confirmation message to user
    confirmation_msg = (
        f"  **Agendamento Confirmado!**\\n\\n"
        f"  Data: {day}/{month}/{year} às {hora_formatada}\\n"
        f"  Nome: {agendamento['nome']}\\n"
        f"  Telefone: {agendamento['telefone']}\\n"
        f"  E-mail: {agendamento['email']}\\n\\n"
    )
    if not email_sent:
        confirmation_msg += "⚠ *Não foi possível enviar e-mail de
confirmação.*, mas seu agendamento foi registrado.\n"
    context.bot.send_message(chat_id=update.effective_chat.id,
text=confirmation_msg, parse_mode="Markdown")
    logger.info("Agendamento confirmado e mensagem de confirmação enviada ao
usuário.")

def start_cancel_appointment(update, context):
    """Initiate the cancellation process for an appointment."""
    context.user_data.pop("cancelamento", None)
    keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
    last_email = context.user_data.get("last_email", "")
    prompt = (
        "  **Cancelar Agendamento**\\n\
\nPor favor, digite o **e-mail** usado no agendamento "
        "(ou digite 'Voltar' para cancelar):"
    )
    if last_email:
        prompt += f"\\n\\nSugestão: {last_email}"
    update.effective_message.reply_text(prompt,
```

```
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
    context.user_data["cancelamento"] = {"etapa": "email", "page": 0}
    logger.info("Fluxo de cancelamento iniciado - solicitando e-mail do
usuário.")
    return CANCEL_EMAIL

def handle_cancel_email(update, context):
    """Handle the email input for cancellation flow."""
    cancel_state = context.user_data.get("cancelamento", {})
    if cancel_state.get("etapa") != "email":
        logger.error("Estado de cancelamento inválido (esperado 'email').")
        context.user_data.pop("cancelamento", None)
        return 1
    email = update.effective_message.text.strip()
    logger.info(f"Tentativa de cancelamento com e-mail: {email}")
    if email.lower() == "voltar":
        update.effective_message.reply_text("  Cancelamento abortado.",
reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]), parse_mode="Markdown")
        context.user_data.pop("cancelamento", None)
        return 1
    # Validate email format
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$'
    if not re.match(pattern, email):
        keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
        last_email = context.user_data.get("last_email", "")
        prompt = "  E-mail inválido. Por favor, digite novamente (ou digite
'Voltar' para cancelar):"
        if last_email:
            prompt += f"\\n\\nSugestão: {last_email}"
        update.effective_message.reply_text(prompt,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
        return None
    email_lower = email.lower()
    context.user_data["cancelamento"]["email"] = email_lower
    context.user_data["cancelamento"]["etapa"] = "listar_eventos"
    context.user_data["last_email"] = email_lower
    list_user_events(update, context, page=0)
    return CANCEL_LIST

def list_user_events(update, context, page=0):
    """List the user's upcoming scheduled events with option to cancel."""
    cancel_state = context.user_data.get("cancelamento", {})
    email = cancel_state.get("email")
    if not email:
        logger.error("E-mail não encontrado em
context.user_data['cancelamento']")
        keyboard = [[InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
        last_email = context.user_data.get("last_email", "")
```

```
        prompt = "  **E-mail não encontrado.**\\n\\nPor favor, digite o **e-
mail** usado no agendamento (ou digite 'Voltar' para cancelar):"
        if last_email:
            prompt += f"\\n\\nSugestão: {last_email}"
        if hasattr(update, "callback_query") and update.callback_query:
            update.callback_query.message.edit_text(prompt,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
        else:
            update.effective_message.reply_text(prompt,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
        context.user_data["cancelamento"] = {"etapa": "email", "page": page}
        return
    events, total_events = get_user_events(email, page=0, page_size=5)
    if not events:
        update.effective_message.reply_text(
            f"  Nenhum agendamento encontrado para o e-mail **{email}**.",
            reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("
Voltar ao Menu", callback_data="main_menu")]]),
            parse_mode="Markdown"
        )
        context.user_data.pop("cancelamento", None)
        return
    total_pages = (total_events + 5 - 1) // 5
    page = max(0, min(page, total_pages - 1))
    events_page, _ = get_user_events(email, page=page, page_size=5)
    keyboard = []
    for event in events_page:
        try:
            start_str = event['start']['dateTime'] if 'dateTime' in
event['start'] else event['start'].get('date')
            start_dt = datetime.fromisoformat(start_str.replace("Z",
"+00:00"))
            local_dt = start_dt.astimezone()
            event_label = f"{local_dt.strftime('%d/%m/%Y %H:%M')} -
{event.get('summary', 'Sem título')}"
            keyboard.append([InlineKeyboardButton(event_label,
callback_data=f"confirm_cancel_{event['id']}")])
        except Exception as e:
            logger.warning(f"Evento com formato inválido:
{event.get('summary', 'Sem título')} ({e})")
            continue
    nav_buttons = []
    if page > 0:
        nav_buttons.append(InlineKeyboardButton("◀ Anterior",
callback_data=f"cancel_all_{page-1}"))
    if page < total_pages - 1:
        nav_buttons.append(InlineKeyboardButton("Próximo ▶",
callback_data=f"cancel_all_{page+1}"))
    if nav_buttons:
        keyboard.append(nav_buttons)
    keyboard.append([InlineKeyboardButton("  Voltar ao Menu",
```

```
callback_data="main_menu")])
    text = f"  **Selecione o agendamento para cancelar (Página {page+1} de
{total_pages}):**"
    if hasattr(update, "callback_query") and update.callback_query:
        update.callback_query.message.edit_text(text,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
    else:
        update.effective_message.reply_text(text,
reply_markup=InlineKeyboardMarkup(keyboard), parse_mode="Markdown")
    context.user_data["cancelamento"]["page"] = page

def handle_cancel_list_page(update, context):
    """Handle pagination for cancellation list of events."""
    query = update.callback_query
    data = query.data
    try:
        query.answer()
    except Exception:
        pass
    try:
        page = int(data.split("_")[-1])
    except:
        page = 0
    list_user_events(update, context, page)
    return None

def handle_cancel_confirm(update, context):
    """Handle selection of a specific event to cancel (confirm prompt)."""
    query = update.callback_query
    data = query.data
    try:
        query.answer()
    except Exception:
        pass
    parts = data.split("_", 2)
    if len(parts) == 3:
        _, _, event_id = parts
    else:
        logger.error(f"Formato inválido para confirm_cancel: {data}")
        return None
    service = get_calendar_service()
    if not service:
        query.message.reply_text("  Erro ao acessar calendário para
cancelar.", reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("
Voltar ao Menu", callback_data="main_menu")]]), parse_mode="Markdown")
        context.user_data.pop("cancelamento", None)
        return 1
    # Retrieve event details for confirmation
    try:
        event = service.events().get(calendarId=CALENDAR_ID,
eventId=event_id).execute()
```

```
    except Exception as e:
        logger.error(f"Erro ao obter evento {event_id}: {e}")
        query.message.reply_text("  Erro ao localizar o agendamento para
cancelar.", reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("
Voltar ao Menu", callback_data="main_menu")]]), parse_mode="Markdown")
        context.user_data.pop("cancelamento", None)
        return 1
    if 'start' in event and 'dateTime' in event['start']:
        start_dt = datetime.fromisoformat(event['start']
['dateTime'].replace("Z", "+00:00")).astimezone()
        event_str = f"{start_dt.strftime('%d/%m/%Y %H:%M')} -
{event.get('summary', '')}"
    else:
        event_str = event.get('summary', 'Agendamento')
    context.user_data["cancelamento"] = {"etapa": "confirmar_cancelamento",
"email": context.user_data.get("cancelamento", {}).get("email"), "page":
context.user_data.get("cancelamento", {}).get("page", 0), "event_id":
event_id}
    keyboard = [
        [InlineKeyboardButton("  Sim, cancelar",
callback_data=f"execute_cancel_{event_id}")],
        [InlineKeyboardButton("  Não, voltar",
callback_data=f"cancel_all_{context.user_data['cancelamento']['page']}")],
        [InlineKeyboardButton("  Voltar ao Menu", callback_data="main_menu")]
    ]
    query.message.reply_text(
        f"⚠ **Confirmar cancelamento**\\n\\nDeseja cancelar o seguinte
agendamento?\\n\\n{event_str}",
        reply_markup=InlineKeyboardMarkup(keyboard),
        parse_mode="Markdown"
    )
    return CANCEL_CONFIRM

def handle_cancel_execute(update, context):
    """Execute the cancellation of an event after confirmation."""
    query = update.callback_query
    data = query.data
    try:
        query.answer()
    except Exception:
        pass
    parts = data.split("_", 2)
    if len(parts) == 3:
        _, _, event_id = parts
    else:
        logger.error(f"Formato inválido para execute_cancel: {data}")
        return None
    service = get_calendar_service()
    if not service:
        query.message.reply_text("  Erro ao acessar o calendário.",
parse_mode="Markdown")
```

```
            context.user_data.pop("cancelamento", None)
            return 1
        try:
            event = service.events().get(calendarId=CALENDAR_ID,
eventId=event_id).execute()
        except Exception as e:
            logger.error(f"Erro ao obter evento antes de cancelar: {e}")
            query.message.reply_text("  Erro ao cancelar o agendamento.",
parse_mode="Markdown")
            context.user_data.pop("cancelamento", None)
            return 1
        try:
            service.events().delete(calendarId=CALENDAR_ID,
eventId=event_id).execute()
            logger.info(f"Evento {event_id} cancelado.")
        except Exception as e:
            logger.error(f"Erro ao cancelar evento {event_id}: {e}")
            query.message.reply_text("  Erro ao cancelar o agendamento.",
parse_mode="Markdown")
            context.user_data.pop("cancelamento", None)
            return 1
        # Inform user about cancellation
        if 'start' in event and 'dateTime' in event['start']:
            start_dt = datetime.fromisoformat(event['start']
['dateTime'].replace("Z", "+00:00")).astimezone()
            event_str = f"{start_dt.strftime('%d/%m/%Y %H:%M')} -
{event.get('summary', '')}"
        else:
            event_str = event.get('summary', 'Agendamento')
        page = context.user_data.get("cancelamento", {}).get("page", 0)
        query.message.reply_text(
            f"  **Agendamento cancelado com sucesso!**\\n\\n{event_str}",
            reply_markup=InlineKeyboardMarkup([
                [InlineKeyboardButton("Voltar à lista",
callback_data=f"cancel_all_{page}")],
                [InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]
            ]),
            parse_mode="Markdown"
        )
        context.user_data["cancelamento"] = {"etapa": "listar_eventos", "email":
context.user_data.get("cancelamento", {}).get("email"), "page": page}
        return None

def get_busy_info(start_date, end_date):
    """Retrieve busy slot info from Google Calendar between start_date and
end_date."""
    cache_key = f"{start_date.isoformat()}_{end_date.isoformat()}"
    with cache_lock:
        if cache_key in busy_info_cache:
            return busy_info_cache[cache_key]
```

```
    service = get_calendar_service()
    if not service:
        logger.warning("Serviço do Google Calendar não disponível")
        return {"hourly": {}, "total": {}}
    try:
        events_result = service.events().list(
            calendarId=CALENDAR_ID,
            timeMin=datetime.combine(start_date,
datetime.min.time()).isoformat() + "Z",
            timeMax=datetime.combine(end_date,
datetime.min.time()).isoformat() + "Z",
            singleEvents=True,
            orderBy="startTime"
        ).execute()
        busy_days_slots = {}
        busy_days_total = {}
        # Build possible slot labels for one day
        possible_slots = []
        current_time = datetime.min.replace(hour=START_HOUR)
        end_time_limit = datetime.min.replace(hour=END_HOUR)
        while current_time <= end_time_limit:
            possible_slots.append(current_time.strftime("%H:%M"))
            current_time = (datetime.combine(datetime.min.date(),
current_time) + timedelta(minutes=APPOINTMENT_DURATION_MINUTES)).time()
            current_time = datetime.min.replace(hour=current_time.hour,
minute=current_time.minute)
        for event in events_result.get("items", []):
            start = event.get("start", {}).get("dateTime")
            if not start:
                continue
            try:
                event_date = datetime.fromisoformat(start.replace("Z",
"+00:00"))
            except Exception:
                logger.debug(f"Formato de data/hora inválido em evento:
{event.get('summary')}")
                continue
            day = event_date.day
            time_str = event_date.strftime("%H:%M")
            if day not in busy_days_slots:
                busy_days_slots[day] = []
            busy_days_slots[day].append(time_str)
            busy_days_total[day] = busy_days_total.get(day, 0) + 1
        busy_info = {"hourly": busy_days_slots, "total": busy_days_total}
        with cache_lock:
            busy_info_cache[cache_key] = busy_info
        return busy_info
    except Exception as e:
        logger.error(f"Erro ao obter eventos do calendário: {e}")
        return {"hourly": {}, "total": {}}
```

```
def get_user_events(email, page=0, page_size=5):
    """Retrieve all future events for the given email from Google
Calendar."""
    service = get_calendar_service()
    if not service:
        return [], 0
    now = datetime.now()
    end_date = now + timedelta(days=365)
    time_min = now.isoformat() + "Z"
    time_max = end_date.isoformat() + "Z"
    try:
        events_result = service.events().list(
            calendarId=CALENDAR_ID,
            timeMin=time_min,
            timeMax=time_max,
            singleEvents=True,
            orderBy="startTime",
            q=email
        ).execute()
        all_events = [event for event in events_result.get("items", [])
                      if 'description' in event and email.lower() in
event['description'].lower()
                      and 'start' in event and ('dateTime' in event['start']
or 'date' in event['start'])]
        start_idx = page * page_size
        end_idx = start_idx + page_size
        return all_events[start_idx:end_idx], len(all_events)
    except Exception as e:
        logger.error(f"Erro em get_user_events: {e}")
        return [], 0
```

handlers/feedback.py – Coleta de Feedback

Implementa o fluxo de feedback ao usuário, apresentando emojis para nível de satisfação e pedindo um texto de feedback. O feedback é então salvo em arquivo (`feedback.txt`):

```
import logging
from telegram import InlineKeyboardButton, InlineKeyboardMarkup
import strings
from handlers import menu
import os

logger = logging.getLogger(__name__)

# Conversation state for feedback text input
FEEDBACK_TEXT = 7

# Define feedback levels mapping (satisfação) com emojis e valores
FEEDBACK_LEVELS = {
    "muito_insatisfeito": {"emoji": " ", "value": 1},
```

```
    "insatisfeito": {"emoji": " ", "value": 2},
    "neutro": {"emoji": "😐", "value": 3},
    "satisfeito": {"emoji": " ", "value": 4},
    "muito_satisfeito": {"emoji": " ", "value": 5}
}

async def start_feedback(update, context):
    """Initiate feedback collection by asking for satisfaction level."""
    # Reset any existing feedback state
    context.user_data.pop("feedback", None)
    # Construct inline buttons for feedback levels
    level_buttons = [InlineKeyboardButton(level["emoji"],
callback_data=f"feedback_level_{name}")
                     for name, level in FEEDBACK_LEVELS.items()]
    keyboard = [level_buttons, [InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")]]
    await update.effective_message.reply_text(
        "  **Deixe seu feedback!**\\n\\nPor favor, selecione seu nível de
satisfação geral:",
        reply_markup=InlineKeyboardMarkup(keyboard),
        parse_mode="Markdown"
    )
    context.user_data["feedback"] = {"stage": "select_level"}
    logger.info(f"Iniciando feedback - seleção de nível de satisfação.")
    # Transition to waiting for level selection (handled via callback)
    return None  # remain in current state (handled by callback query)

async def handle_feedback_level_callback(update, context):
    """Handle feedback level selection (callback)."""
    query = update.callback_query
    data = query.data
    level_name = data.replace("feedback_level_", "")
    try:
        query.answer()
    except Exception as e:
        logger.warning(f"Erro ao responder feedback callback: {e}")
    if level_name not in FEEDBACK_LEVELS:
        logger.warning(f"Nível de feedback inválido: {level_name}")
        return None
    # Store selected level
    context.user_data["feedback"] = {
        "stage": "writing_text",
        "level": FEEDBACK_LEVELS[level_name]["value"],
        "emoji": FEEDBACK_LEVELS[level_name]["emoji"]
    }
    # Ask for feedback text
    query.edit_message_text(
        f"Você selecionou: {context.user_data['feedback']['emoji']}\\n\
\nAgora, por favor, digite seu feedback:",

reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("  Voltar ao Menu",
```

```
callback_data="main_menu")]]),
        parse_mode="Markdown"
    )
    logger.info(f"Nível de feedback selecionado: {level_name}. Aguardando
texto do feedback.")
    # Move to state waiting for feedback text
    return FEEDBACK_TEXT

async def handle_feedback_text(update, context):
    """Handle the feedback text entered by the user."""
    feedback_state = context.user_data.get("feedback", {})
    if feedback_state.get("stage") == "writing_text":
        user_feedback_text = update.effective_message.text.strip()
        if not user_feedback_text:
            await update.effective_message.reply_text(
                "  Por favor, digite um feedback válido.",
                reply_markup=InlineKeyboardMarkup([[InlineKeyboardButton("
Voltar ao Menu", callback_data="main_menu")]]),
                parse_mode="Markdown"
            )
            return None
        # Save feedback to file
        user_id = update.effective_user.id
        level_value = feedback_state.get("level")
        if level_value is None:
            logger.error("Nível de feedback não encontrado.")
            await update.effective_message.reply_text("  Ocorreu um erro ao
registrar seu feedback.", reply_markup=strings.main_menu_keyboard(),
parse_mode="Markdown")
            context.user_data.pop("feedback", None)
            return 1
        save_feedback(level_value, user_feedback_text, user_id)
        await update.effective_message.reply_text(
            "  **Obrigado pelo seu feedback!** Ele foi registrado com
sucesso.\\n\\nVolte ao menu principal:",
            reply_markup=strings.main_menu_keyboard(),
            parse_mode="Markdown"
        )
        context.user_data.pop("feedback", None)
        logger.info(f"Feedback recebido de user {user_id}, nível
{level_value}.")
        return 1
    # If feedback stage not as expected, end or return to menu
    return 1

def save_feedback(level, text, user_id):
    """Append feedback to a local file for record."""
    try:
        timestamp = __import__('datetime').datetime.now().strftime("%Y-%m-%d
%H:%M:%S")
        line = f"[{timestamp}] User ID: {user_id} - Nível: {level} - Texto:
```

```
{text}\\n"
        with open("feedback.txt", "a", encoding="utf-8") as f:
            f.write(line)
        logger.info("Feedback salvo com sucesso.")
    except Exception as e:
        logger.error(f"Erro ao salvar feedback: {e}")
```

handlers/utils.py – Funções Utilitárias (Envio de E-mail)

Contém a função para enviar o e-mail de confirmação de agendamento, incluindo um anexo `.ics` do evento para adicionar ao calendário do usuário. As configurações de e-mail (SMTP) são lidas do `.env`:

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.application import MIMEApplication
from datetime import datetime, timedelta
import os
import logging

logger = logging.getLogger(__name__)

# Email configuration from environment
EMAIL_HOST = os.getenv("EMAIL_HOST", "smtp.gmail.com")
EMAIL_PORT = int(os.getenv("EMAIL_PORT", 587))
EMAIL_USER = os.getenv("EMAIL_USER")
EMAIL_PASSWORD = os.getenv("EMAIL_PASSWORD")
# Appointment duration needed for ICS
APPOINTMENT_DURATION_MINUTES = int(os.getenv("APPOINTMENT_DURATION_MINUTES",
60))

def send_confirmation_email(to_email, name, appointment_time):
    """Send a confirmation email with an .ics calendar invite for the
appointment."""
    try:
        msg = MIMEMultipart()
        msg['From'] = EMAIL_USER
        msg['To'] = to_email
        msg['Subject'] = "  Confirmação de Agendamento"
        body = f"""
        <h2>Seu agendamento foi confirmado!</h2>
        <p><strong>Nome:</strong> {name}</p>
        <p><strong>Data/Horário:</strong> {appointment_time}</p>
        <p>Por favor, se prepare com 15 minutos de antecedência,<br>
        procure relaxar e, <strong>CALMAMENTE</strong>, inspire o ar pelo
nariz e expire pelo nariz, pelo menos 3 vezes.</p>
        <p>Segue em anexo o convite do evento (.ics) para adicionar ao seu
calendário.</p>
        <p>Agradecemos sua confiança!</p>
        """
```

```
        msg.attach(MIMEText(body, 'html'))
        # Create ICS calendar invite
        try:
            date_part, time_part = appointment_time.split(" às ")
            day, month, year = map(int, date_part.split("/"))
            hour, minute = map(int, time_part.split(":"))
        except Exception as e:
            logger.error(f"Erro ao parsear appointment_time:
{appointment_time} -> {e}")
            day = month = year = hour = minute = None
        if year and month and day and hour is not None:
            start_dt = datetime(year, month, day, hour, minute)
            end_dt = start_dt +
timedelta(minutes=APPOINTMENT_DURATION_MINUTES)
            dtstart = start_dt.strftime("%Y%m%dT%H%M%SZ")
            dtend = end_dt.strftime("%Y%m%dT%H%M%SZ")
            safe_name = ''.join(c for c in name if c.isalnum() or c in (' ',
'_')).strip().replace(' ', '_')
            safe_email_user = EMAIL_USER or ''
            ics_content = f"""BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//GuiaEspiritual Bot//EN
BEGIN:VEVENT
UID:{datetime.now().strftime('%Y%m%d%H%M%S')}-{safe_name}@guiaespiritual
DTSTAMP:{datetime.now().strftime('%Y%m%dT%H%M%SZ')}
DTSTART:{dtstart}
DTEND:{dtend}
SUMMARY:Atendimento - {name}
DESCRIPTION:Atendimento confirmado com Guia Espiritual.
ORGANIZER;CN={safe_email_user}:MAILTO:{safe_email_user}
LOCATION:Casa Socorrista e Orfanato Maria de Nazaré
END:VEVENT
END:VCALENDAR"""
            part = MIMEApplication(ics_content.encode('utf-8'),
_subtype="calendar")
            part.add_header('Content-Disposition', 'attachment;
filename="convite.ics"')
            msg.attach(part)
        # Send email via SMTP
        with smtplib.SMTP(EMAIL_HOST, EMAIL_PORT) as server:
            server.ehlo()
            server.starttls()
            server.ehlo()
            if EMAIL_USER and EMAIL_PASSWORD:
                server.login(EMAIL_USER, EMAIL_PASSWORD)
            server.send_message(msg)
        logger.info(f"E-mail de confirmação enviado para {to_email}")
    except Exception as e:
        logger.error(f"Erro ao enviar e-mail para {to_email}: {e}")
        raise
```

messages.py – Base de Mensagens e Busca de Respostas

Este módulo carrega as **intents** (perguntas e respostas) do JSON e, caso ativado, utiliza **SentenceTransformer** para encontrar a resposta mais similar à pergunta do usuário. Fornece funções para obter o submenu de perguntas e respostas e buscar a melhor resposta via embeddings:

```
import json
import os
from sentence_transformers import SentenceTransformer, util
import numpy as np
import logging
logger = logging.getLogger(__name__)

memory = None
model = None
question_embeddings = None
questions_list = []
category_questions = {}

def load_memory():
    """Load intents and Q&A from JSON file."""
    global memory
    if memory is not None:
        return memory
    file_path = "data/messages.json"
    if not os.path.exists(file_path):
        file_path = "json/memory.json"
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            memory = json.load(f)
    except Exception as e:
        memory = {"intents": []}
    return memory

def initialize_embeddings():
    """Load the Q&A data and initialize embeddings for semantic search."""
    global model, question_embeddings, questions_list, category_questions
    mem = load_memory()
    questions_list = []
    category_questions = {}
    for intent in mem.get("intents", []):
        tag = intent.get("tag", "")
        patterns = intent.get("patterns", [])
        responses = intent.get("responses", [])
        if not patterns or len(patterns) != len(responses):
            continue
        category_questions[tag] = patterns
        for q in patterns:
            questions_list.append((q, tag))
    if questions_list:
```

```
        # Initialize model if not already done
        if model is None:
            model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-
v2')
        question_embeddings = model.encode([q for q, _ in questions_list],
convert_to_tensor=True)
    else:
        question_embeddings = None

def get_submenu_keyboard(category, page=0, items_per_page=10):
    """Return InlineKeyboardMarkup for a page of questions under a
category."""
    mem = load_memory()
    intent = next((i for i in mem.get("intents", []) if i.get("tag") ==
category), None)
    questions = intent["patterns"] if intent else []
    if not questions:
        return InlineKeyboardMarkup([[InlineKeyboardButton("  Voltar",
callback_data="main_menu")]]), f"⚠ Nenhuma pergunta disponível para
*{category.replace('_', ' ').capitalize()}*."
    total_pages = (len(questions) + items_per_page - 1) // items_per_page
    page = max(0, min(page, total_pages - 1))
    start_idx = page * items_per_page
    end_idx = min(start_idx + items_per_page, len(questions))
    page_questions = questions[start_idx:end_idx]
    buttons = []
    for i, q in enumerate(page_questions, start=start_idx):
        safe_cat = category.replace("_", "__")
        callback_data = f"answer_{safe_cat}_{i}"
        buttons.append([InlineKeyboardButton(q,
callback_data=callback_data)])
    nav_buttons = []
    if page > 0:
        nav_buttons.append(InlineKeyboardButton("◄ Anterior",
callback_data=f"submenu_{category}_{page-1}"))
    if page < total_pages - 1:
        nav_buttons.append(InlineKeyboardButton("Próximo ►",
callback_data=f"submenu_{category}_{page+1}"))
    if nav_buttons:
        buttons.append(nav_buttons)
    buttons.append([InlineKeyboardButton("  Voltar ao Menu",
callback_data="main_menu")])
    return InlineKeyboardMarkup(buttons), None

def get_answer_by_index(category, index):
    """Retrieve the answer text for a given category and question index."""
    mem = load_memory()
    for intent in mem.get("intents", []):
        if intent.get("tag") == category:
            patterns = intent.get("patterns", [])
            responses = intent.get("responses", [])
```

```
            if 0 <= index < len(responses):
                return responses[index]
    return None

def find_best_answer(user_message):
    """Find the best matching answer for a user message using embeddings."""
    mem = load_memory()
    if question_embeddings is None or not questions_list:
        return None
    try:
        user_embedding = model.encode(user_message, convert_to_tensor=True)
        cos_scores = util.cos_sim(user_embedding, question_embeddings)[0]
        cos_scores = cos_scores.cpu().numpy() if hasattr(cos_scores, "cpu")
else np.array(cos_scores)
        max_idx = int(np.argmax(cos_scores))
        max_score = float(np.max(cos_scores))
        if max_score > 0.6:
            question, category = questions_list[max_idx]
            # Find corresponding response
            for intent in mem.get("intents", []):
                if intent.get("tag") == category:
                    try:
                        idx = intent["patterns"].index(question)
                        return intent["responses"][idx]
                    except ValueError:
                        continue
    except Exception as e:
        logger = __import__('logging').getLogger(__name__)
        logger.error(f"Erro na busca de resposta: {e}")
    return None

# Import telegram classes used in functions to avoid circular import
from telegram import InlineKeyboardMarkup, InlineKeyboardButton
```

strings.py – Constantes de Interface

Define todas as mensagens e textos fixos exibidos pelo bot, assim como a estrutura do teclado do menu principal (botões dois a dois):

```
from telegram import ReplyKeyboardMarkup

# Interface text constants
WELCOME_MSG = "  **Bem-vindo ao Guia Espiritual!**\\n\\nEscolha uma opção
abaixo:"
CONSENT_MSG = ("⚖️ *Consentimento LGPD*\n"

"Este bot coletará seu nome, telefone e e-mail para agendamento e envio de
confirmações. "
             "Seus dados serão utilizados apenas para essa finalidade e
poderão ser apagados mediante solicitação. "
```

```
"Ao prosseguir, você concorda com o uso dos seus dados pessoais conforme a
LGPD.\n\n"
              "Você aceita os termos?")
HOW_IT_WORKS = ("i *Como funciona o agendamento*\n\n"
                "1. Clique em 'Agendar Atendimento' para escolher uma data e
horário.\n"
                "2. No calendário, selecione um dia disponível:\n"
                "   - Dias disponíveis aparecem com o número (ex.: 23).\n"
                "   - Dias lotados são marcados com X (ex.: 19X).\n"
                "   - Dias passados são marcados com () (ex.: (18)).\n"
                "3. Escolha um horário disponível e confirme seus dados.\n"
                "4. Para cancelar, use 'Cancelar Agendamento' e selecione o
atendimento.\n\n"
                "Selecione um dia disponível: (X indica agenda cheia, ()
indica data passada)")
CONTINUE_MSG = "  *Posso ajudar em algo mais?* Se desejar, escolha outra
opção abaixo:"
DID_NOT_UNDERSTAND = ("  **Não entendi.** Tente:\n"
                      "- Usar os botões do menu\n"
                      "- Reformular sua pergunta\n"
                      "- Agendar um atendimento  ")

# Main menu categories mapping (button text to category tag)
MAIN_MENU_CATEGORIES = {
    "  Meditação": "meditacao",
    "  Emoções": "emocao",
    "  Cura": "cura",
    "🕊 Paz Interior": "paz_interior",
    "? Ajuda": "ajuda",
    "📣 Mensagem": "mensagem",
    "  Agendar Atendimento": "agendar",
    "  Feedback": "feedback",
    "  Cancelar Agendamento": "cancelar",
    "i Como funciona": "como_funciona",
    "  Sair": "sair"
}

# Main menu keyboard layout
MAIN_MENU_BUTTONS = [
    ["  Meditação", "  Emoções"],
    ["  Cura", "🕊 Paz Interior"],
    ["📣 Mensagem", "  Feedback"],
    ["  Agendar Atendimento", "  Cancelar Agendamento"],
    ["? Ajuda", "i Como funciona"],
    ["  Sair"]
]

def main_menu_keyboard():
    """Return the ReplyKeyboardMarkup for the main menu."""
```

```
    return ReplyKeyboardMarkup(MAIN_MENU_BUTTONS, resize_keyboard=True,
one_time_keyboard=False)
```

data/messages.json – Intents de Perguntas/Respostas (JSON)

Este arquivo JSON contém as **intents** (categorias de perguntas frequentes e suas respostas). Cada intent tem um *tag* (categoria), uma lista de *patterns* (perguntas possíveis) e *responses* (respostas correspondentes). O conteúdo abaixo é um trecho do arquivo (o arquivo completo contém todas as perguntas e respostas, incluindo meditação, emoções, cura, paz interior, etc.):

```
{
  "intents": [
    {
      "tag": "meditacao",
      "patterns": [
        "Como meditar corretamente?",
        "O que é meditação guiada?",
        "Posso meditar deitado?",
        "Quanto tempo devo meditar por dia?",
        "Meditação ajuda a dormir?",
        "... (mais perguntas de meditação) ..."
      ],
      "responses": [
        "Que bom que você quer aprender a meditar! Sente-se em um lugar
tranquilo, com a coluna reta, mas sem tensão — pode ser em uma cadeira ou no
chão, como se sentir mais confortável. Feche os olhos suavemente e comece a
observar sua respiração: inspire pelo nariz contando até 4, segure por 2
segundos e expire lentamente por 6. Se pensamentos vierem, não se preocupe,
apenas os observe como nuvens passando no céu e volte sua atenção para a
respiração. No Espiritismo, meditar é uma forma de se conectar com energias
superiores e alinhar sua alma com a luz divina, então você pode imaginar uma
luz branca te envolvendo, trazendo paz e equilíbrio. Psicologicamente, isso
ativa o sistema nervoso parassimpático, reduzindo o estresse. Comece com 5
minutos e vá aumentando aos poucos, tudo bem? Você já deu um passo lindo ao
buscar essa prática!",

"A meditação guiada é uma prática linda onde uma voz te conduz, passo a
passo, para um estado de relaxamento e conexão interior. Geralmente, um
instrutor ou uma gravação te dá instruções, como imaginar um lugar tranquilo
— talvez uma praia ou uma floresta — enquanto você se concentra na sua
respiração e nas sensações do corpo. É perfeita para quem está começando ou
sente dificuldade em meditar sozinho. No Espiritismo, podemos ver isso como
uma forma de abrir o coração para receber orientações de mentores
espirituais, guiando sua evolução. Psicologicamente, ajuda a treinar a
atenção plena (mindfulness), melhorando o foco e reduzindo a ansiedade. Que
tal experimentar uma? Estou aqui para te apoiar!",
        "Sim, você pode meditar deitado, e isso pode ser muito relaxante! Só
tome cuidado para não adormecer, a menos que seja seu objetivo. Deite-se de
costas, com a coluna alinhada — coloque um travesseiro pequeno sob a cabeça
ou joelhos, se precisar de mais conforto. Mantenha os braços relaxados ao
```

```
lado do corpo e foque na sua respiração consciente, sentindo o ar entrar e
sair. No Espiritismo, essa posição pode ser vista como uma forma de se
conectar com a energia da Terra, que te acolhe e renova. Psicologicamente,
ajuda a liberar tensões acumuladas e promove relaxamento muscular. Se sentir
sono, talvez prefira sentar na próxima vez — o que acha de testar?",
        "Você já está no caminho certo só por querer meditar! Se está
começando, 5 a 10 minutos por dia já é um ótimo começo — o importante é criar
um hábito que seja leve para você. Conforme for se sentindo mais à vontade,
pode aumentar para 15 ou 20 minutos. No Espiritismo, o tempo de meditação é
um momento sagrado para harmonizar sua energia com o plano espiritual, então
a qualidade da conexão importa mais que a duração. Psicologicamente, estudos
mostram que até 10 minutos diários reduzem o estresse e melhoram a
concentração. Escolha um tempo que te faça bem, sem pressão, combinado?",
        "Sim, a meditação pode ser uma aliada incrível para melhorar seu
sono! Quando você medita, sua mente desacelera, reduzindo a ansiedade e o
estresse, que muitas vezes nos mantêm acordados. Antes de dormir, experimente
uma prática simples: sente-se ou deite-se, feche os olhos e foque na sua
respiração, contando cada expiração até 10, e recomece. No Espiritismo,
podemos imaginar que estamos nos conectando com energias de paz que preparam
o espírito para um descanso reparador. Psicologicamente, isso ativa o sistema
nervoso parassimpático, relaxando o corpo. Se quiser, posso te sugerir uma
meditação específica para o sono — o que acha?",
        "... (mais respostas correspondentes) ..."
      ]
    },
    {
      "tag": "emocao",
      "patterns": [ "Como controlar a raiva?", "O que fazer com tristeza?",
"..."],
      "responses": [ "Para controlar a raiva, ...", "Quando estamos
tristes, ...", "..." ]
    },
    {
      "tag": "cura",
      "patterns": [ "... perguntas da categoria cura ..." ],
      "responses": [ "... respostas da categoria cura ..." ]
    },
    {
      "tag": "paz_interior",
      "patterns": [ "... perguntas sobre paz interior ..." ],
      "responses": [ "... respostas sobre paz interior ..." ]
    },
    {
      "tag": "ajuda",
      "patterns": [ "... perguntas de ajuda espiritual ..." ],
      "responses": [ "... respostas para ajuda ..." ]
    },
    {
      "tag": "mensagem",
      "patterns": [ "... perguntas ou solicitações de mensagem ..." ],
      "responses": [ "... possíveis mensagens espirituais ou
```

```
motivacionais ..." ]
    }
    // ... (continuação para outras categorias)
  ]
}
```

(O `messages.json` *completo contém todas as perguntas e respostas; acima mostramos apenas uma parte para ilustrar o formato.)*

json/memory.json – Memória de Intents (JSON)

Este arquivo é equivalente ao `messages.json` e contém as mesmas intents. Ele pode ser utilizado internamente para carregar os dados na memória e realizar a busca por similaridade (embeddings). Geralmente não é necessário modificar este arquivo manualmente se `messages.json` estiver atualizado, pois o bot irá carregá-lo.

txt/conteudo.txt – Conteúdo das Intents (Texto)

Este arquivo de texto lista todas as perguntas e respostas de forma legível, separadas por categoria, para referência ou edição manual. Cada linha segue o formato:

```
<categoria>|<pergunta>?|<resposta>
```

Por exemplo, linhas iniciais do arquivo:

```
meditacao|Como meditar corretamente?|Que bom que você quer aprender a
meditar! Sente-se em um lugar tranquilo, com a coluna reta, mas sem tensão –
pode ser em uma cadeira ou no chão, como se sentir mais confortável. Feche os
olhos suavemente e comece a observar sua respiração: inspire pelo nariz
contando até 4, segure por 2 segundos e expire lentamente por 6. Se
pensamentos vierem, não se preocupe, apenas os observe como nuvens passando
no céu e volte sua atenção para a respiração. No Espiritismo, meditar é uma
forma de se conectar com energias superiores e alinhar sua alma com a luz
divina, então você pode imaginar uma luz branca te envolvendo, trazendo paz e
equilíbrio. Psicologicamente, isso ativa o sistema nervoso parassimpático,
reduzindo o estresse. Comece com 5 minutos e vá aumentando aos poucos, tudo
bem? Você já deu um passo lindo ao buscar essa prática!
meditacao|O que é meditação guiada?|A meditação guiada é uma prática linda
onde uma voz te conduz, passo a passo, para um estado de relaxamento e
conexão interior. Geralmente, um instrutor ou uma gravação te dá instruções,
como imaginar um lugar tranquilo – talvez uma praia ou uma floresta –
enquanto você se concentra na sua respiração e nas sensações do corpo. É
perfeita para quem está começando ou sente dificuldade em meditar sozinho. No
Espiritismo, podemos ver isso como uma forma de abrir o coração para receber
orientações de mentores espirituais, guiando sua evolução. Psicologicamente,
ajuda a treinar a atenção plena (mindfulness), melhorando o foco e reduzindo
a ansiedade. Que tal experimentar uma? Estou aqui para te apoiar!
...
```

(O arquivo continua listando todas as perguntas e respostas das categorias meditacao, emocao, cura, paz_interior, ajuda, mensagem, etc.)

requirements.txt – Dependências do Projeto

Para instalar todas as bibliotecas Python necessárias, veja o conteúdo de **requirements.txt**:

```
python-telegram-bot==20.3
python-dotenv
google-api-python-client
google-auth
cachetools
sentence-transformers
numpy
```

Essas dependências incluem a biblioteca do Telegram Bot API (v20.x), integração com .env, cliente da API Google Calendar, biblioteca de autenticação do Google, cachetools (usada para gerenciar reservas temporárias de horários), **Sentence Transformers** para busca semântica, e numpy para operações de similaridade.

Observação final: O pacote acima fornece toda a estrutura do bot sem incluir o arquivo de banco de dados `banco.db`. Caso não utilize um banco de dados SQLite, o bot recorrerá ao `data/messages.json` para obter as intents de respostas. Lembre-se de preencher o `.env` com suas credenciais (Token do Telegram, e-mail, senha, ID do calendário, etc.) antes de executar o bot. Você pode baixar o arquivo ZIP com toda essa estrutura para facilitar a implantação. Boa sorte com seu bot do Telegram!
