ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Московский институт электроники и математики им. А.Н. Тихонова

Документация разработчика

по проектной работе 1430 «Создание интерактивного вводного курса к экспозиции Музея криптографии»

Телеграмм бот написан на языке python 3 и представлен в виде скрипта .py.
Использованы следующие библиотеки:

- aiogram3 для работы с TelegramAPI
- модуль environs для работы с файлом .env с токеном

Бот инициализируется командой

bot: Bot = Bot(token=config.tg_bot.token, parse_mode='HTML'),

где config.tg_bot.token содержит уникальный токен, который выдает @BotFather при создании бота.

Связь между ботом и сервером Telegram осуществляется с помощью

await dp.start_polling(bot).

env и .env.example

В файле .env хранится токен бота и id админов, пример содержимого находится в файле .env.example.

requirements.txt

В файле requirements.txt содержатся необходимые библиотеки, которые используются в проекте.

bot.py

Данный файл является точкой входа в данном проекте.

book

Текст всех лекций хранится в папке book в файле book.txt

Для его заполнения необходимо добавлять тест по следующему правилу: лекции разделяются двумя пустыми строками, а страницы внутри одной лекции одной пустой строкой. Перед первой лекцией 3 пустые строки.

Функция для разделения текста на главы и страницы находится в файле **services/file_handling.**

prepare_book(path = BOOK_PATH)

заполняет пустой словарь book

Поддерживаемая разметка текста **HTML style.** Настройка разметки при инициализации бота.

bot: Bot = Bot(token=config.tg_bot.token, parse_mode='HTML')

В aiogram 3.0.0b7 поддерживаются следующие HTML-теги:

Жирный текст

Наклонный текст

<u>Подчеркнутый текст</u>

<s>Перечеркнутый текст</s>

Спойлер

Внешняя ссылка

Внутренняя ссылка

<pre>Предварительно отформатированный текст</pre>

images

Картинки к лекциям хранятся в папке images. Для удобства название состоит из номера лекции и страницы, на которой нужно отобразить эту картинку.например картинка для третьей лекции и второй страницы имеет название 3p2.png

После добавления изображений в папку, их нужно добавить в словарь с фотографиями photo_db (в папке database в файле database.py), где ключом является номер главы, а значением словарь с ключом - номером страницы и значением - названием файла с изображением.

config_data

Функция load_config из файла config_data.py возвращает заполненный Config, данные берутся из файла .env

```
Config(tg_bot=TgBot(token=env('BOT_TOKEN'),
                    admin_ids=list(map(int, env('ADMIN_IDS')))))
```

database

В файле database.py хранятся данные по используемым изображеним (photo_db) и по заданиям (task_db) в виде словарей.

lexicon

LEXICON - словарь, ключами которого являются сообщения от пользователя, команды и callback, а значениями текст, который отправляется пользователю, либо текст на инлайн кнопках. При добавлении команд или кнопок, эта информация добавляется сюда. При необходимости можно редактировать значения в словаре, не меняя ключей, так как ключи используются в обработчиках.

LEXICON_COMMANDS - словарь, ключами которого являются все команды,а значениями их описание в выпадающем меню.

LEXICON_CHAPTERS – словарь, ключами которого являются номера глав, а значениями их названия, которые появляются на инлайн кнопках при выборе глав.

keyboards

В данной папке содержатся файлы для создания разных библиотек.

Функция set_main_menu(bot: Bot) создает выпадающее меню с командами из LEXICON_COMMANDS.

Функция create_chapters_keyboard() -> InlineKeyboardMarkup создает клавиатуру с инлайн кнопками - названиями всех лекций из LEXICON_CHAPTERS.
Функция create_pagination_keyboard(*buttons: str, chapter: int=0) -> InlineKeyboardMarkup генерирует клавиатуру с номером текущей страницы, с заданиями к данной главе из LEXICON_TASKS, и кнопку LEXICON['main_return'] для возвращения к выбору глав.

handlers

@router.message(CommandStart())
async def process_start_command(message: Message, state: FSMContext)
Этот обработчик срабатывает на команду "/start" и добавляет пользователя в базу данных, если его там еще не было с помощь _fill_state(state, user_dict_template) и отправляет ему приветственное сообщение. Изменяет состояние state.set_state(FSMReadSolve.choose_action).

@router.message(StateFilter(default_state))
async def process_not_start_command(message: Message)
Этот обработчик срабатывает в состоянии по умолчанию, если бот получил что-то кроме команды /start, в этом случаем бот сообщает о том, что пользователю нужно сначала запустить бота.

@router.message(Command(commands='beginning'),
~StateFilter(FSMReadSolve.solve_task))
async def process_beginning_command(message: Message, state: FSMContext)
Этот обработчик срабатывает на команду "/beginning" и отправляет пользователю первую страницу первой лекции с кнопками пагинации.

@router.message(Command(commands='help'), ~StateFilter(default_state))
async def process_help_command(message: Message)
Этот обработчик срабатывает на команду "/help" и отправляет пользователю сообщение со списком доступных команд в боте, которые содержатся в LEXICON['/help'].

@router.message(Command(commands='chapters'), ~StateFilter(default_state))
async def process_help_cmd(message: Message)
Этот обработчик срабатывает на команду "/chapters" и отправляет пользователю клавиатуру с инлайн кнопками с доступными лекциями с помощью функции create_chapters_keyboard(), которая описана в keyboards/chapters_kb.py, и которая берет список лекций из списка значений LEXICON_CHAPTERS. При этом callback_data для каждой лекции будет соответствовать ключу из словаря LEXICON_CHAPTERS.

@router.message(Command(commands=['continue']),
~StateFilter(FSMReadSolve.solve_task))
async def process_continue_command(message: Message, state: FSMContext)
Этот обработчик срабатывает на команду "/'continue'" и возвращает пользователя на главу и страницу, на которой пользователь остановился.

@router.callback_query(IsChapterCallbackData(),
~StateFilter(FSMReadSolve.solve_task))
async def process_forward_press(callback: CallbackQuery, state: FSMContext)

Этот обработчик срабатывает на нажатие инлайн кнопки с главой и отправляет первую страницу выбранной лекции с клавиатурой reply_markup=create_pagination_keyboard. При этом данные пользователя в базе данных обновляются - глава и страница.

@router.callback_query(Text(text='forward'),
~StateFilter(FSMReadSolve.solve_task))
async def process_forward_press(callback: CallbackQuery, bot: Bot, state: FSMContext)
Этот обработчик будет срабатывать на нажатие инлайн-кнопки "вперед" во время взаимодействия пользователя с сообщением-книгой. При этом если это не последняя страница в главе
if users_db[callback.from_user.id]['page']
<len(book[users_db[callback.from_user.id]['chapter']]), то номер страницы увеличивается на 1
users_db[callback.from_user.id]['page'] += 1
и сообщение с книгой удаляется и заменяется на новое со следующей страницей.

@router.callback_query(Text(text='backward'),
~StateFilter(FSMReadSolve.solve_task))
async def process_backward_press(callback: CallbackQuery, state: FSMContext)
Этот обработчик будет срабатывать на нажатие инлайн-кнопки "назад" во время взаимодействия пользователя с сообщением-книгой. При этом если это не первая страница в главе
if users_db[callback.from_user.id]['page'] > 1, то
страница уменьшается на 1
users_db[callback.from_user.id]['page'] -= 1,

предыдущее сообщение удаляется и посылается новое с предыдущей страницей.

@router.callback_query(Text(text='more_try'))
async def process_more_task_press(callback: CallbackQuery)
Этот обработчик будет срабатывать на нажатие инлайн-кнопки во время выполнения задания 'решить еще одно задание'. При нажатии генерируется новое задание с клавиатурой reply_markup=create_tasks_keyboard('more_try', 'return').

@router.callback_query(Text(text='show_answer'))
async def process_showanswer_press(callback: CallbackQuery)
Этот обработчик будет срабатывать на нажатие инлайн-кнопки во время выполнения задания 'Показать ответ'. Он отправляет сообщение с правильным ответом и клавиатурой для возврата к чтению reply_markup=create_tasks_keyboard('return').

@router.callback_query(Text(text='return'))
async def process_edit_press(callback: CallbackQuery, state: FSMContext):
Этот обработчик будет срабатывать на нажатие инлайн-кнопки во время выполнения задания 'вернуться к чтению'. И открывает книгу на главе и странице, на которой остановился пользователь.

implementation_tests

@bot.message_handler(commands=["start"])
Данный обработчик команды "/start" необходим для того, чтобы бот мог реагировать на команду старта, которую пользователь отправляет в чат. Обычно команда "/start" используется для того, чтобы инициализировать

работу бота и получить первичную информацию о его функционале и возможностях.

В данном случае, обработчик команды "/start" позволяет боту отправить пользователю приветственное сообщение и предложить заполнить профиль, если это необходимо. Также этот обработчик может содержать в себе различные дополнительные функции, которые могут помочь пользователю понять, как использовать бота и какие команды ему доступны.

@bot.message_handler(content_types=['text'])

Данный обработчик сообщений необходим для того, чтобы бот мог реагировать на текстовые сообщения, которые отправляются пользователем в чат. В большинстве случаев, обработчик сообщений типа 'text' является основным для бота, так как именно текстовые сообщения позволяют пользователям общаться с ботом и получать от него ответы.

Обработчик может содержать в себе различные функции, которые позволяют боту обрабатывать и анализировать текстовые сообщения, например, распознавание ключевых слов и команд, обработку текста с использованием регулярных выражений, и т.д.

Обработчик сообщений типа 'text' позволяет боту быть более гибким и адаптивным к потребностям пользователей, так как они могут вводить любые текстовые сообщения, и бот должен быть готов к обработке их в режиме реального времени.

job_command_handlers

В этом блоке находятся функции для реализации интерактива.

def caesar(key, mes)

Описан алгоритм шифрования сообщения шифром Цезаря.

def caesar_decod(key, mes)

Описан алгоритм расшифрования сообщения шифром Цезаря

def atbash(mes)

Описан алгоритм шифрования/расшифрования шифром ATBASH.