1. Что такое Telegram-бот и каковы его основные возможности? Опишите функциональные возможности Telegram-бота и приведите примеры его применения.

Telegram-бот — это специальный аккаунт в мессенджере Telegram, который управляется программным кодом и может взаимодействовать с пользователями автоматически. Боты создаются с помощью Telegram Bot API и регистрируются через специального бота BotFather. Они не являются полноценными пользователями, а представляют собой автоматизированные сервисы, способные принимать сообщения, команды и запросы от пользователей и отвечать на них.

Основные возможности Telegram-ботов:

- **Обработка сообщений и команд**: Боты могут реагировать на текстовые сообщения, команды (начинающиеся с "/"), кнопки и inline-запросы.
- **Отправка контента**: Текст, фото, видео, аудио, документы, стикеры, опросы и даже игры.
- **Интеграция с клавиатурами**: Обычные клавиатуры (reply keyboard) для быстрого ввода и inline-клавиатуры для динамических взаимодействий.
- **Вебхуки и polling**: Для получения обновлений от Telegram-сервера.
- **Платежи и игры**: Поддержка встроенных платежей и HTML5-игр.
- **Групповые и канальные функции**: Боты могут быть добавлены в группы или каналы для модерации, уведомлений и т.д.
- **Inline-режим**: Боты могут предоставлять контент в других чатах (например, GIF или статьи).

Функциональные возможности:

- Автоматизация задач: Рассылка уведомлений, поиск информации, обработка форм.
- Интеграция с внешними сервисами: API для погоды, новостей, баз данных.
- Персонализация: Хранение данных о пользователях для контекстных ответов.

Примеры применения:

- **Уведомления**: Бот для мониторинга акций на бирже (например, @StockBot отправляет обновления цен).
- **Поиск и сервисы**: @weatherbot для прогноза погоды.
- **Игры и развлечения**: @TriviaBot для викторин.
- **Бизнес**: Боты для онлайн-магазинов, где пользователи могут выбирать товары через кнопки и оплачивать.

- **Образование**: Боты для изучения языков, где они отправляют уроки и проверяют ответы.

2. Как создать простейший Telegram-бот с использованием библиотеки python-telegram-bot? Расскажите последовательность действий от регистрации бота в BotFather до написания минимального кода.

Последовательность действий для создания простейшего Telegram-бота с библиотекой python-telegram-bot (PTB):

1. **Регистрация бота в BotFather**:
    - Откройте Telegram и найдите бота @BotFather.
    - Отправьте команду /newbot.
    - Укажите имя бота (например, "MyFirstBot").
    - Укажите username бота (должен заканчиваться на "bot", например, "myfirst_bot").
    - BotFather выдаст API-токен (например, "123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11"). Сохраните его — это ключ для доступа к API.
2. **Установка библиотеки**:
    - Установите PTB через pip: pip install python-telegram-bot.
3. **Написание минимального кода**:
    - Создайте файл, например, bot.py.
    - Импортируйте необходимые модули.
    - Создайте updater с токеном.
    - Добавьте обработчик для сообщений.
    - Запустите polling для получения обновлений.

Минимальный код (эхо-бот, повторяющий сообщения пользователя):

Python

```
from telegram import Update
from telegram.ext import ApplicationBuilder, MessageHandler, filters


# Токен от BotFather
TOKEN = 'YOUR_TOKEN_HERE'
```

```
async def echo(update: Update, context):
    await update.message.reply_text(update.message.text)

app = ApplicationBuilder().token(TOKEN).build()
app.add_handler(MessageHandler(filters.TEXT & ~filters.COMMAND, echo))
app.run_polling()
```

4. **Запуск бота**:
    - Запустите скрипт: python bot.py.
    - Найдите бота в Telegram по username и отправьте сообщение — он должен ответить эхом.

Это базовая настройка. Для продвинутых функций добавляйте больше обработчиков.

3. Как обработать команду /start в Telegram-боте? Опишите механизм регистрации команд и вызова соответствующих обработчиков.

Обработка команды /start в Telegram-боте с PTB:

Механизм регистрации команд:

- В PTB команды регистрируются через CommandHandler, который привязывается к функции-обработчику.
- Обработчик вызывается, когда Telegram отправляет update с типом "message" и текстом, начинающимся с "/command".
- Регистрация происходит в Application (ранее Updater), где добавляются хендлеры в порядке приоритета.

Пример обработки /start:

Python

```
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler

async def start(update: Update, context):
    await update.message.reply_text('Привет! Я бот. Напиши что-нибудь.')

app = ApplicationBuilder().token('TOKEN').build()
app.add_handler(CommandHandler('start', start))  # Регистрация команды
```

app.run_polling()

- Когда пользователь отправляет /start, Telegram отправляет update.
- PTB проверяет хендлеры: если текст — команда /start, вызывается функция start.
- В функции можно получить данные о пользователе (update.message.from_user) и отправить ответ.

Преимущества: Легко добавлять аргументы (например, /start deep_link), фильтры и контекст для хранения данных.

4. Какие типы обновлений (updates) предоставляет Telegram Bot API? Перечислите основные типы и опишите их назначение.

Telegram Bot API предоставляет обновления (updates) — события от пользователей или системы. Основные типы:

- **Message**: Сообщение от пользователя (текст, фото, видео, документ, стикер, опрос и т.д.). Назначение: Обработка входящего контента.
- **EditedMessage**: Отредактированное сообщение. Назначение: Реакция на изменения в чате.
- **ChannelPost**: Пост в канале. Назначение: Для ботов в каналах, мониторинг публикаций.
- **EditedChannelPost**: Отредактированный пост в канале.
- **InlineQuery**: Inline-запрос от пользователя (бот предоставляет контент в другом чате). Назначение: Динамический поиск, как @gif для GIF.
- **ChosenInlineResult**: Выбор результата inline-запроса. Назначение: Логирование выбора.
- **CallbackQuery**: Нажатие на inline-кнопку. Назначение: Обработка взаимодействий с клавиатурой.
- **ShippingQuery**: Запрос на доставку (для платежей). Назначение: Интеграция с магазинами.
- **PreCheckoutQuery**: Предварительная проверка платежа.
- **Poll**: Опрос. Назначение: Создание и обработка голосований.
- **PollAnswer**: Ответ на опрос.
- **MyChatMember**: Изменение статуса бота в чате (добавлен/удален).
- **ChatMember**: Изменение статуса пользователя в чате.
- **ChatJoinRequest**: Запрос на присоединение к чату.

Updates получаются через getUpdates (polling) или webhook. Каждый update имеет ID для предотвращения дубликатов.

5. Как реализовать inline-кнопки в сообщениях Telegram бота? Объясните, что такое inline-клавиатура и как с ней взаимодействовать.

Inline-клавиатура (InlineKeyboardMarkup) — это набор кнопок, прикрепляемых к сообщению, которые не занимают место в клавиатуре пользователя, а отображаются под сообщением. Каждая кнопка может иметь callback_data для обработки нажатий или URL для перехода.

Реализация в PTB:

1. Импортируйте InlineKeyboardButton и InlineKeyboardMarkup.
2. Создайте список кнопок.
3. Прикрепите клавиатуру к сообщению через reply_markup.

Пример:

Python

```
from telegram import InlineKeyboardButton, InlineKeyboardMarkup, Update
from telegram.ext import ApplicationBuilder, CommandHandler


async def start(update: Update, context):
  keyboard = [
    [InlineKeyboardButton("Кнопка 1", callback_data='button1')],
    [InlineKeyboardButton("Кнопка 2", url='https://example.com')]
  ]
  reply_markup = InlineKeyboardMarkup(keyboard)
  await update.message.reply_text('Выбери кнопку:', reply_markup=reply_markup)


app = ApplicationBuilder().token('TOKEN').build()
app.add_handler(CommandHandler('start', start))
app.run_polling()
```

Взаимодействие:

- Пользователь нажимает кнопку — генерируется CallbackQuery.
- Обработайте его отдельным хендлером (CallbackQueryHandler).

- Inline-клавиатура не исчезает после нажатия, в отличие от reply-клавиатуры.

6. Что такое CallbackQuery и как его обрабатывать? Расскажите, в каких случаях возникает CallbackQuery, и приведите пример его обработки.

CallbackQuery — это update, генерируемый при нажатии на inline-кнопку с callback_data. Он содержит данные кнопки, ID сообщения и пользователя.

Случаи возникновения: Когда пользователь кликает на кнопку в inline-клавиатуре (не URL-кнопки, они открывают ссылку).

Обработка в PTB: Используйте CallbackQueryHandler.

Пример:

Python

```
from telegram import Update
from telegram.ext import ApplicationBuilder, CallbackQueryHandler


async def button(update: Update, context):
    query = update.callback_query
    await query.answer()  # Подтверждение нажатия (опционально с текстом)
    await query.edit_message_text(text=f"Нажата: {query.data}")


app = ApplicationBuilder().token('TOKEN').build()
app.add_handler(CallbackQueryHandler(button))  # Обработчик для всех callback
app.run_polling()
```

- query.data — данные кнопки.
- Можно редактировать сообщение (edit_message_text) или отправлять новое.
- Подтверждение answer() предотвращает "загрузку" кнопки.

7. Каким образом можно реализовать многофункциональные команды в Telegram-боте? Объясните особенности обработки разных команд и аргументов.

Для многофункциональных команд:

- Регистрируйте несколько CommandHandler для разных команд.
- Для одной команды с аргументами: Парсите update.message.text.split() после "/command".
- Используйте filters для условий (например, filters.TEXT для сообщений).

Особенности:

- Команды чувствительны к регистру? Нет, но username бота добавляется для приватности (/start@mybot).
- Аргументы: Передаются после команды, разделенные пробелами.
- Контекст: Храните состояние в context.user_data.

Пример с несколькими командами и аргументами:

Python

```
async def greet(update: Update, context):
  args = context.args  # Аргументы как список
  name = args[0] if args else 'пользователь'
  await update.message.reply_text(f'Привет, {name}!')


async def help_cmd(update: Update, context):
  await update.message.reply_text('Список команд: /greet [имя]')


app.add_handler(CommandHandler('greet', greet))
app.add_handler(CommandHandler('help', help_cmd))
```

- Для сложных: Используйте ConversationHandler для состояний.

8. Как организовать хранение данных о пользователях или сессиях в Telegram-боте? Обсудите подходы, от in-memory хранилищ до использования баз данных.

Подходы к хранению данных:

1. **In-memory (в памяти)**:
    - Используйте context.user_data (словарь в PTB) для сессий.
    - Преимущества: Быстро, просто.
    - Недостатки: Данные теряются при перезапуске бота.
    - Пример: context.user_data['name'] = 'User'.
2. **Файловое хранение**:
    - JSON или pickle для сохранения данных на диск.
    - Преимущества: Персистентность без БД.
    - Недостатки: Не для высоконагруженных систем (блокировки файлов).

3. **Базы данных**:
    - **SQLite**: Локальная, простая (библиотека sqlite3). Для малого трафика.
    - **PostgreSQL/MySQL**: Для масштаба, с ORM как SQLAlchemy.
    - **NoSQL (MongoDB, Redis)**: Для сессий (Redis для кэша, Mongo для документов).
    - Преимущества: Масштабируемость, запросы.
    - Интеграция: В обработчиках читайте/пишите в БД.

Пример с SQLite:

- Создайте таблицу users (id, data).
- В коде: conn = sqlite3.connect('db.sqlite'); cursor.execute('INSERT ...').

Выберите по нагрузке: In-memory для тестов, БД для продакшена.

9. Что такое объект Context в библиотеке python-telegram-bot и как его использовать? Опишите преимущества передачи контекста между обработчиками.

Объект Context (CallbackContext) в PTB — контейнер для данных, связанных с обновлением. Он передается в обработчики вместе с Update.

Использование:

- **context.bot**: Для отправки сообщений (bot.send_message).
- **context.user_data**: Словарь для данных пользователя (персистентен в сессии).
- **context.chat_data**: Для чата/группы.
- **context.args**: Аргументы команды.
- **context.job_queue**: Для планирования задач.

Преимущества:

- Передача состояния между хендлерами без глобальных переменных.
- Безопасность: Изоляция данных по пользователям/чатам.
- Удобство: Автоматическая передача в асинхронных функциях.

Пример:

Python

```
async def start(update: Update, context):
  context.user_data['count'] = 0
```

```
    await update.message.reply_text('Счетчик: 0')


async def increment(update: Update, context):
    context.user_data['count'] += 1
    await update.message.reply_text(f'Счетчик: {context.user_data["count"]}')
```

10. Как настроить вебхук для Telegram-бота? Опишите отличия между опросом (polling) и вебхуками, а также порядок настройки вебхука.

Отличия:

- **Polling**: Бот периодически запрашивает updates у Telegram (getUpdates). Просто, но тратит ресурсы на пустые запросы. Подходит для разработки.
- **Webhook**: Telegram отправляет updates на URL бота (HTTPS). Эффективнее, реал-тайм, но требует сервера с HTTPS.

Порядок настройки вебхука:

1. Разверните бота на сервере (Heroku, VPS) с HTTPS (используйте Let's Encrypt).
2. В коде PTB: app.run_webhook(listen='0.0.0.0', port=PORT, url_path=TOKEN, webhook_url=f'https://yourdomain.com/{TOKEN}').
3. Установите вебхук через API: requests.post(f'https://api.telegram.org/bot{TOKEN}/setWebhook?url=https://yourdomain.com/{TOKEN}').
4. Обработайте POST-запросы от Telegram в веб-сервере (PTB использует aiohttp или similar).

Пример в коде: Замените run_polling() на run_webhook().

11. Какие методы существуют для отправки мультимедийного контента (фото, аудио, видео) ботом? Перечислите методы API и особенности их использования.

Методы Telegram Bot API для мультимедиа (в PTB: bot.send_*):

- **sendPhoto**: Отправка фото. Параметры: chat_id, photo (file_id, URL или файл). Особенности: Поддержка caption, reply_markup.
- **sendAudio**: Аудио. Параметры: audio, duration, performer, title. Для музыки/голоса.
- **sendVideo**: Видео. Параметры: video, duration, width, height. Поддержка streaming.
- **sendAnimation**: GIF/анимация.
- **sendVoice**: Голосовое сообщение (OGG).

- **sendVideoNote**: Круглое видео (квадратное).
- **sendDocument**: Файлы любого типа (PDF, ZIP и т.д.).
- **sendMediaGroup**: Группа медиа (альбом фото/видео).

Особенности:

- Файлы до 50MB (20MB для фото).
- Используйте file_id для повторной отправки без загрузки.
- В PTB: await context.bot.send_photo(chat_id=chat_id, photo=open('file.jpg', 'rb')).

12. Как интегрировать Telegram-бот с внешними REST API или базами данных? Объясните, как можно осуществлять запросы к внешним сервисам и обрабатывать полученные данные.

Интеграция:

1. **REST API**: Используйте requests для запросов в обработчиках.
2. **Базы данных**: Подключите SQLAlchemy или pymongo для чтения/записи.

Процесс:

- В обработчике: Получите данные от пользователя.
- Сделайте запрос: response = requests.get('https://api.example.com/data').
- Обработайте: data = response.json(); сохраните в БД или отправьте пользователю.
- Асинхронно: Используйте aiohttp для async.

Пример с погодой:

Python

```
import requests


async def weather(update: Update, context):
  city = context.args[0]
  resp = requests.get(f'https://api.openweathermap.org/data/2.5/weather?q={city}&appid=API_KEY')
  data = resp.json()
  await update.message.reply_text(f'Температура в {city}: {data["main"]["temp"]}°C')
```

Для БД: В обработчике выполняйте db.query().

13. Что такое web parsing и какие задачи он позволяет решать? Дайте определение web scraping/parsing и обсудите его применение.

Web parsing (или web scraping) — процесс извлечения данных из веб-страниц автоматически. Parsing фокусируется на анализе HTML/XML, scraping — на сборе данных.

Определение: Web scraping — автоматизированный сбор структурированных данных с сайтов via HTTP-запросов и парсинга HTML. Parsing — разбор содержимого для извлечения элементов.

Задачи:

- Сбор цен с e-commerce (мониторинг цен).
- Агрегация новостей/статей.
- Анализ конкурентов (отзывы, данные).
- Исследования: Сбор данных для ML (например, тексты для NLP).

Применение: Боты для уведомлений о скидках, SEO-анализ, научные данные. Учитывайте robots.txt, законность (GDPR, ToS) и этику — не перегружайте серверы.

14. Какой модуль Python наиболее популярен для отправки HTTP-запросов и почему? Обсудите библиотеку requests и её преимущества.

Наиболее популярный — requests.

Почему: Простота, читаемость, поддержка всех HTTP-методов, обработка JSON, сессий, аутентификации.

Преимущества:

- Человеческий API: requests.get(url) вместо urllib.
- Автоматическая обработка: Кодировки, редиректы, ошибки.
- Расширяемость: Сессии для cookies, hooks.
- Интеграция: С другими библиотеками (BeautifulSoup).

Пример: response = requests.post('https://api.com', json={'key': 'value'}); data = response.json().

Альтернативы: aiohttp для async, но requests — стандарт для sync.

15. В чем разница между использованием BeautifulSoup и lxml при парсинге HTML? Сравните их по производительности, удобству и функциональности.

Разница:

- **BeautifulSoup (BS)**: Библиотека для парсинга HTML/XML, использует парсеры как lxml или html.parser.

- **lxml**: Парсер на C, может использоваться standalone или как backend для BS.

Сравнение:

- **Производительность**: lxml быстрее (C-based), BS медленнее без lxml.
- **Удобство**: BS проще для новичков (soup.find_all('a')), lxml требует XPath/CSS (tree.xpath('//a')).
- **Функциональность**: BS — навигация по тегам, поиск, модификация; lxml — быстрый парсинг, XPath, XML-поддержка, но менее интуитивен.

Рекомендация: BS с lxml-backend (BeautifulSoup(html, 'lxml')) для баланса.

16. Приведите пример кода для извлечения всех ссылок с веб-страницы с использованием BeautifulSoup. Продемонстрируйте базовый пример кода с комментариями.

Пример:

Python

```
import requests
from bs4 import BeautifulSoup

# Получаем HTML страницы
url = 'https://example.com'
response = requests.get(url)
html = response.text

# Парсим HTML
soup = BeautifulSoup(html, 'html.parser')  # Или 'lxml' для скорости

# Извлекаем все теги <a> с атрибутом href
links = soup.find_all('a')
for link in links:
    href = link.get('href')  # Получаем значение href
    if href:  # Проверяем, что href существует
        print(href)  # Выводим ссылку
```

Комментарии: Это извлекает все ссылки. Для фильтра: soup.find_all('a', class_='specific').

17. Как с помощью Selenium можно обрабатывать динамические веб-страницы? Опишите особенности работы с динамическим контентом и приведите примеры инициализации браузера.

Selenium — для автоматизации браузера, идеален для JS-динамики (AJAX, React).

Особенности:

- Эмулирует пользователя: Клик, ввод, скролл.
- Обрабатывает динамику: Ждет загрузки элементов.
- Поддержка headless (без окна).

Инициализация:

Python

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# Настройка опций (headless)
options = Options()
options.headless = True

# Инициализация браузера
driver = webdriver.Chrome(options=options)  # Нужно скачать chromedriver
driver.get('https://example.com')

# Пример: Клик по кнопке, которая загружает контент
button = driver.find_element_by_id('load-more')
button.click()

# Закрытие
driver.quit()
```

Для динамики: Используйте waits (WebDriverWait) для ожидания элементов.

18. Что такое XPath и каким образом он используется для поиска элементов на странице? Объясните синтаксис XPath и разницу с CSS-селекторами.

XPath — язык запросов для навигации по XML/HTML-дереву.

Использование: В Selenium (find_element_by_xpath), lxml (tree.xpath).

Синтаксис:

- / — корень.
- // — любой потомок.
- @attr — атрибут.
- Пример: //div[@class='content']//a — все в div с class='content'.
- [1] — индекс.

Разница с CSS:

- CSS: .class #id — проще для стилей, быстрее.
- XPath: Мощнее (родители, текст: //a[contains(text(),'Link')]), но медленнее, сложнее.
- CSS удобнее для фронтенда, XPath — для сложных структур.

19. Как организовать ожидание загрузки элементов в Selenium? Расскажите про методы «явного» и «неявного» ожидания и их применение.

Ожидание нужно для динамики.

- **Неявное (implicit wait)**: Глобальное ожидание для всех find_element. driver.implicitly_wait(10) — ждет 10 сек перед ошибкой.
    - Применение: Для простых случаев, но может замедлить.
- **Явное (explicit wait)**: WebDriverWait для конкретных условий.
    - Применение: Ждать visibility, clickability.

Пример явного:

Python

```
from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions as EC


wait = WebDriverWait(driver, 10)
```

element = wait.until(EC.presence_of_element_located((By.ID, 'myElement')))

- Sleep — статическое, избегайте.

20. Какие проблемы могут возникнуть при разработке парсеров для сайтов с защитой от ботов? Перечислите типичные меры защиты и возможные способы обхода (с учетом законности и этических аспектов).

Проблемы: Блокировка IP, CAPTCHA, ошибки.

Меры защиты:

- CAPTCHA (reCAPTCHA).
- Rate limiting (ограничение запросов).
- JS-челленджи (Cloudflare).
- User-agent проверка.
- Honeypots (скрытые поля).

Способы обхода (этично, законно: Только с разрешения, не для коммерции):

- Прокси/ротация IP.
- Изменение user-agent (requests.headers).
- Задержки между запросами (time.sleep).
- Headless браузеры для JS.
- API, если доступно.

Этика: Соблюдайте ToS, не перегружайте, используйте для исследований.

21. Как можно предотвратить блокировку IP при интенсивном парсинге веб-страниц? Обсудите использование прокси-серверов и изменение user-agent.

Предотвращение:

- **Прокси**: Ротация IP через прокси (бесплатные/платные как Bright Data).
    - В requests: proxies={'http': 'http://proxy:port'}.
    - Ротация: Список прокси, меняйте рандомно.
- **Изменение user-agent**: Имитируйте браузер.
    - headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...'}.

Другие: Задержки (sleep(1-5)), лимит запросов/мин, распределение по времени.

22. Чем отличаются CSS селекторы от XPath в контексте поиска элементов на странице? Сравните удобство и возможности обоих подходов.

Отличия:

- **CSS**: Основан на стилях (.class, #id, tag > child). Быстрее, проще.
- **XPath**: Путь по дереву (//tag[@attr='value']). Мощнее для текста, индексов, родителей.

Удобство: CSS интуитивен для веб-разработчиков, XPath — для сложных запросов. Возможности: CSS не поддерживает текст поиска (contains), XPath да. CSS быстрее в браузерах.

Пример: CSS 'div.content a'; XPath '//div[@class="content"]/a'.

23. Как можно использовать регулярные выражения для извлечения данных из текстовых блоков? Приведите пример задачи и решение с применением модуля re.

Регулярные выражения (regex) — для поиска шаблонов в тексте.

Задача: Извлечь email из строки.

Решение:

Python

```
import re


text = 'Контакт: user@example.com, support@site.ru'
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)
print(emails)  # ['user@example.com', 'support@site.ru']
```

- re.findall — все совпадения.
- Шаблон: \b — граница слова, + — один или больше.

24. Что такое AJAX и как он влияет на процесс парсинга веб-страниц? Объясните, почему AJAX-запросы могут усложнять сбор данных и как с этим работать.

AJAX (Asynchronous JavaScript and XML) — техника загрузки данных без перезагрузки страницы (via XMLHttpRequest или fetch).

Влияние: HTML не содержит все данные сразу — они загружаются динамически.

Усложнения: Статика (requests) видит только начальный HTML, без JS-контента.

Работа:

- Используйте Selenium для рендеринга JS.
- Инспектируйте Network в браузере, имитируйте AJAX-запросы (requests.get(api_url)).

- Библиотеки: Splash или Puppeteer для headless.

25. Как создать базовое REST API приложение с использованием FastAPI? Приведите пример минимального кода и объясните его структуру.

FastAPI — фреймворк для API на Python, с async и валидацией.

Минимальный код:

Python

```
from fastapi import FastAPI


app = FastAPI()  # Инициализация приложения


@app.get("/")  # Маршрут для GET /
async def root():
  return {"message": "Hello World"}
```

Структура:

- app = FastAPI(): Создает приложение.
- @app.get("/"): Декоратор для маршрута, метод GET.
- Функция: Возвращает JSON.
- Запуск: uvicorn main:app --reload.

26. В чем особенности асинхронных обработчиков в FastAPI по сравнению с синхронными? Обсудите, почему FastAPI оптимизирован под async и как это влияет на производительность.

Особенности:

- Асинхронные: def -> async def, await для IO.
- Синхронные: Обычные def, блокирующие.

FastAPI на Starlette/ASGI, оптимизирован для async: Поддержка coroutines, лучше для IO-bound (запросы, БД).

Производительность: Async обрабатывает больше запросов параллельно (не ждет IO), выше throughput. Для CPU-bound — используйте workers.

Пример async: async def read(): await asyncio.sleep(1)

27. Как настраивается маршрутизация (routing) в FastAPI? Объясните принцип работы маршрутов, использование path-параметров и query-параметров.

Маршрутизация: Через декораторы @app.method(path).

Принцип: Path матчит URL, вызывает функцию.

- Path-параметры: В {} , типизированные. /users/{user_id:int}
- Query-параметры: В функции как аргументы с default. def get(q: str = None)

Пример:

Python

```
@app.get("/items/{item_id}")
async def read_item(item_id: int, q: str | None = None):
  return {"item_id": item_id, "q": q}
```

- /items/5?q=hello -> {"item_id": 5, "q": "hello"}

28. Как используется Pydantic для валидации данных в FastAPI? Приведите пример описания модели данных и валидации входящих запросов.

Pydantic — для моделей данных с валидацией.

В FastAPI: Авто-валидация для запросов.

Пример:

Python

```
from pydantic import BaseModel
from fastapi import FastAPI

app = FastAPI()

class Item(BaseModel):  # Модель
  name: str
  price: float
  is_offer: bool = None

@app.post("/items/")
async def create_item(item: Item):  # Валидация
  return item
```

- POST {"name": "Foo", "price": 42.0} — OK.
- Неверный тип — 422 ошибка.

29. Каким образом FastAPI генерирует документацию для API? Опишите интеграцию Swagger UI и ReDoc, а также их возможности по настройке.

Генерация: Автоматическая на основе кода (OpenAPI schema).

- Swagger UI: /docs — интерактивный UI для тестов.
- ReDoc: /redoc — красивая документация.

Настройка:

- app = FastAPI(title='My API', description='Desc', openapi_tags=[...])
- Кастом: Переопределите /docs с HTML.

Возможности: Тест запросов, схемы моделей, авторизация.

31. Как следует обрабатывать исключения в FastAPI? Расскажите о создании обработчиков ошибок (exception handlers) и их регистрации.

Обработка: Через exception_handlers.

Пример:

Python

```
from fastapi import FastAPI, HTTPException, Request
from fastapi.responses import JSONResponse

app = FastAPI()

class CustomError(Exception):
  pass

@app.exception_handler(CustomError)
async def custom_handler(request: Request, exc: CustomError):
  return JSONResponse(status_code=418, content={"message": "Custom error"})

@app.get("/")
async def root():
```

raise CustomError("Oops")

- HTTPException: Встроенная для статусов.
- Регистрация: @app.exception_handler(Type).

32. Что такое Background Tasks в FastAPI и как они могут применяться? Объясните механизм выполнения фоновых задач и приведите пример.

Background Tasks: Задачи, выполняемые после ответа, не блокируя.

Механизм: Добавляются в BackgroundTasks, FastAPI запускает после return.

Пример:

Python

```
from fastapi import FastAPI, BackgroundTasks


app = FastAPI()


def write_log(message: str):
  with open("log.txt", "a") as log:
    log.write(message)


@app.get("/send-log")
async def send_log(background_tasks: BackgroundTasks):
  background_tasks.add_task(write_log, "Logged!")
  return {"message": "Task added"}
```

Применение: Отправка email, обработка файлов.

34. Приведите пример интеграции FastAPI с внешним источником данных.

Пример с БД (SQLite via SQLAlchemy):

Python

```
from fastapi import FastAPI
from sqlalchemy import create_engine, Column, Integer, String, MetaData, Table
from databases import Database
```

```
DATABASE_URL = "sqlite:///./test.db"
database = Database(DATABASE_URL)
metadata = MetaData()

users = Table(
  "users",
  metadata,
  Column("id", Integer, primary_key=True),
  Column("name", String),
)

engine = create_engine(DATABASE_URL)
metadata.create_all(engine)

app = FastAPI()

@app.on_event("startup")
async def startup():
  await database.connect()

@app.on_event("shutdown")
async def shutdown():
  await database.disconnect()

@app.get("/users/{user_id}")
async def read_user(user_id: int):
  query = users.select().where(users.c.id == user_id)
  return await database.fetch_one(query)
```

- Интеграция: Подключение на startup, запросы в маршрутах.

35. В чем разница между методами POST и PUT, и как их использовать в FastAPI? Объясните особенности каждого метода в контексте создания и обновления ресурсов.

Разница:

- **POST**: Создание нового ресурса. Сервер генерит ID. Неидемпотентен (множественные вызовы — множественные ресурсы).
- **PUT**: Обновление/создание ресурса по ID. Идемпотентен (повтор — тот же результат).

В FastAPI:

- @app.post("/items/") для создания.
- @app.put("/items/{item_id}") для обновления.

Особенности: POST для коллекций, PUT для конкретных (если не существует — создать).

36. Как создать главное окно приложения с помощью Tkinter? Опишите основные шаги создания окна и инициализации основного цикла приложения.

Шаги:

1. Импортируйте tkinter.
2. Создайте root = tk.Tk().
3. Настройте: root.title('App'), root.geometry('300x200').
4. Добавьте виджеты.
5. Запустите root.mainloop().

Пример:

Python

```
import tkinter as tk


root = tk.Tk()  # Главное окно

root.title('My App')

root.mainloop()  # Цикл событий
```

Mainloop: Обрабатывает события, держит окно открытым.

37. Какие способы размещения виджетов в Tkinter вы знаете (pack, grid, place)? Объясните особенности каждого метода и приведите примеры их применения.

Способы:

- **pack**: Авто-упаковка (top, bottom, left, right). Простой, но менее контролируемый.
    - Пример: label.pack(side='top')
- **grid**: Сетка (row, column). Гибкий для форм.
    - Пример: label.grid(row=0, column=0)
- **place**: Абсолютное позиционирование (x, y). Точный, но не responsive.
    - Пример: label.place(x=10, y=20)

Применение: Grid для таблиц, pack для простых, place для overlay.

38. Как добавить кнопку (Button) на форму и связать её с функцией-обработчиком события? Приведите пример кода с созданием кнопки и назначением команды.

Пример:

Python

```
import tkinter as tk


def on_click():
  print('Кнопка нажата!')


root = tk.Tk()
button = tk.Button(root, text='Нажми', command=on_click)
button.pack()
root.mainloop()
```

- command: Вызывает функцию без аргументов при клике.

39. Как создать иерархию меню в Tkinter? Опишите создание главного меню, выпадающих списков и подключение команд к пунктам меню.

Создание:

- menubar = tk.Menu(root)
- root.config(menu=menubar)
- filemenu = tk.Menu(menubar, tearoff=0)
- menubar.add_cascade(label='File', menu=filemenu)

- filemenu.add_command(label='Open', command=open_func)

Пример: Иерархия с подменю.

40. Каким образом можно обновить текст или свойства виджета Label в Tkinter? Расскажите, как использовать методы для изменения отображаемых данных.

Обновление:

- label.config(text='New text')
- Или StringVar: var = tk.StringVar(); label = tk.Label(textvariable=var); var.set('New')

Пример:

Python

var = tk.StringVar(value='Old')

label = tk.Label(root, textvariable=var)

label.pack()

var.set('New') # Обновит label

41. Что такое Canvas и как его использовать для рисования графических элементов? Обсудите возможности Canvas и приведите пример рисования фигур.

Canvas — виджет для графики.

Возможности: Линии, формы, изображения, текст, события.

Пример:

Python

canvas = tk.Canvas(root, width=200, height=200)

canvas.pack()

canvas.create_rectangle(10, 10, 100, 100, fill='blue') # Прямоугольник

canvas.create_line(0, 0, 200, 200) # Линия

42. Как реализовать ввод данных пользователем через виджет Entry? Объясните метод получения значения из текстового поля и его последующей обработки.

Entry — для ввода текста.

Пример:

Python

entry = tk.Entry(root)

```
entry.pack()


def get_value():
  value = entry.get()  # Получить текст
  print(value)


button = tk.Button(root, text='Get', command=get_value)
button.pack()
```

- Обработка: В функции command.

43. Как использовать Listbox для отображения списка элементов и обработки выбора пользователя? Приведите пример создания Listbox и работы с его элементами.

Пример:

Python

```
listbox = tk.Listbox(root)
listbox.insert(tk.END, 'Item1')
listbox.insert(tk.END, 'Item2')
listbox.pack()


def on_select(event):
  selected = listbox.curselection()  # Индекс
  print(listbox.get(selected))


listbox.bind('<<ListboxSelect>>', on_select)
```

- curselection(): Выбранные индексы.
- get(idx): Значение.

44. Как создать диалоговое окно для отображения ошибок или информационных сообщений в Tkinter? Опишите использование модулей, таких как messagebox, для создания окна сообщений.

Используйте tkinter.messagebox.

Пример:

Python

```
from tkinter import messagebox


messagebox.showinfo('Info', 'Message')  # Инфо
messagebox.showerror('Error', 'Oops')  # Ошибка
messagebox.askyesno('Question', 'Yes/No?')  # Вопрос
```

- Блокирует до ответа.

45. Как настроить обработку событий клавиатуры в Tkinter? Приведите примеры связывания событий нажатия клавиш с функциями.

События: <Key>, <KeyPress-A>.

Пример:

Python

```
def on_key(event):
  print(f'Нажата: {event.char}')


root.bind('<Key>', on_key)  # Для всего окна
```

- event.keysym для спецклавиш.

46. Какие особенности нужно учитывать при разработке многооконных приложений в Tkinter? Обсудите использование дополнительных окон (Toplevel) и их управление.

Особенности:

- Главное — Tk(), дополнительные — Toplevel().
- Управление: toplevel.grab_set() для фокуса, destroy() для закрытия.
- Коммуникация: Через глобальные или классы.

Пример:

Python

```
top = tk.Toplevel(root)
top.title('Второе окно')
```

- Учитывайте: Mainloop только один, события общие.

47. Каким образом можно интегрировать Tkinter с задачами, выполняемыми в отдельных потоках (multithreading)? Расскажите об особенностях работы GUI в многопоточных приложениях и способах безопасного обновления интерфейса.

Особенности: Tkinter не thread-safe, обновления только из main thread.

Интеграция:

- threading.Thread для задач.
- Обновление: root.after(100, func) или queue.

Пример:

Python

```
import threading


def long_task():
  # Долгая задача
  root.after(0, update_label)  # Безопасно обновить


thread = threading.Thread(target=long_task)
thread.start()
```

- Избегайте прямых вызовов из threads.

48. Что такое метод mainloop() в Tkinter и почему он является необходимым элементом приложения? Объясните его роль в поддержании жизненного цикла окна.

mainloop(): Запускает цикл событий, обрабатывает клики, клавиши, redraw.

Необходим: Без него окно появляется и исчезает, приложение не реагирует.

Роль: Поддерживает GUI, ждет событий, обновляет.

49. Как использовать модуль ttk для создания современных и стилизованных виджетов в Tkinter? Опишите преимущества ttk и для каких целей его целесообразно применять.

ttk: Темированные виджеты.

Использование: from tkinter import ttk; button = ttk.Button(...)

Преимущества: Кросс-платформенные стили, темы (clam, alt), лучше выглядят.

Цели: Современный UI, кнопки, комбо, прогрессбары.

Пример: style = ttk.Style(); style.theme_use('clam')

50. Опишите архитектуру приложения, в котором интегрированы Telegram-бот, парсинг/анализ данных, FastAPI и Tkinter. Расскажите, какие задачи может решать такое комплексное приложение, как распределить функциональность между компонентами и как обеспечить взаимодействие между ними.

Архитектура: Модульная, с центральным сервером.

Компоненты:

- **Telegram-бот**: Фронтенд для пользователей (обработка команд, отправка данных). Использует PTB.
- **Парсинг/анализ**: Модуль для сбора данных (requests, BS, Selenium). Анализирует сайты, сохраняет в БД.
- **FastAPI**: Бэкенд API для интеграции (эндпоинты для данных, аутентификации). Обслуживает запросы от бота/GUI.
- **Tkinter**: Десктопный GUI для админа (мониторинг, настройки).

Распределение:

- Бот: Взаимодействие с пользователями, запросы к FastAPI.
- Парсинг: Фоновые задачи в FastAPI (background tasks) для сбора данных.
- FastAPI: Центральный хаб, БД (SQLite/PostgreSQL), API для бота/GUI.
- Tkinter: Админ-панель, запросы к FastAPI для просмотра данных.

Взаимодействие:

- Бот -> FastAPI: requests.post к API для данных.
- Парсинг: Вызывается из FastAPI или бота via queues (RabbitMQ для async).
- Tkinter -> FastAPI: requests для обновлений.
- Общее: БД для хранения, logging.

Задачи: Мониторинг сайтов (парсинг цен), уведомления в Telegram, админ в GUI. Например, бот для трекинга новостей: Парсит сайты, анализирует, отправляет в чат; админ управляет через Tkinter.