ChatGPT

Полное руководство для подготовки к экзамену: модель OSI, HTTP-методы, Client/Server, Web-scraping

1. Семь уровней модели OSI

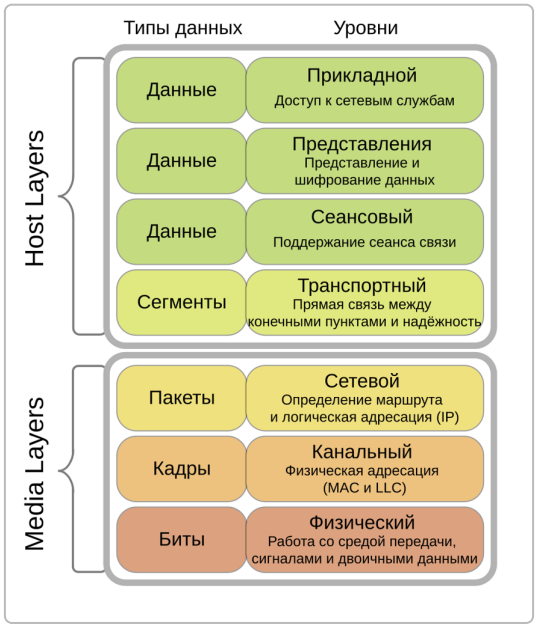

Рис. 1: Диаграмма 7 уровней модели OSI. Модель делится на две группы: нижние уровни относятся к среде передачи (Media Layers), верхние – к узлам (Host Layers). На каждом уровне данные представлены в определённой форме (биты, кадры, пакеты, сегменты или высокоуровневые данные).

Модель OSI (Open Systems Interconnection) – эталонная сетевая модель, разделяющая процесс сетевого взаимодействия на семь последовательных уровней. Модель была разработана в конце 1970-х годов для описания архитектуры и принципов работы сетей передачи данных [1]. Каждый уровень отвечает за свой аспект обработки информации и оперирует своими объектами данных. При передаче данные проходят через уровни 7→1 на стороне отправителя и 1→7 на стороне получателя. Ниже представлены все **семь уровней OSI** с краткими определениями для запоминания и пояснениями функций каждого уровня:

1. **Физический уровень (Physical, L1):** Аппаратный низший уровень, отвечающий за передачу битов по физической среде (кабели, радиоволны и т. п.). Здесь данные представлены в виде электрических/оптических сигналов (битов). Примеры: спецификации разъёмов, напряжений, частот, а также **физические среды передачи** (витая пара, оптика, Wi-Fi).
2. **Канальный уровень (Data Link, L2):** Обеспечивает надежную передачу кадров (frames) по физическому каналу между двумя узлами. Выполняет **управление доступом к среде** и обнаружение/исправление ошибок передачи. Здесь данные группируются в кадры,

добавляются MAC-адреса отправителя и получателя. Примеры протоколов: Ethernet (кадры Ethernet), Wi-Fi (802.11), PPP.

3. **Сетевой уровень (Network, L3):** Отвечает за логическую адресацию и маршрутизацию пакетов между узлами в разных сетях. Именно на этом уровне происходит выбор маршрута и **доставка пакетов** от отправителя к получателю через промежуточные узлы. Ключевой протокол – IP (Internet Protocol), также маршрутизаторы оперируют на этом уровне.
4. **Транспортный уровень (Transport, L4):** Обеспечивает надёжную передачу данных между приложениями на концах сети, разбивает данные на сегменты и контролирует их порядок и целостность. Реализует **управление потоком и контроль ошибок**: может гарантировать доставку без потерь или, наоборот, передавать без подтверждений для скорости. Примеры: TCP (с установлением соединения, гарантией доставки) и UDP (без соединения, возможно с потерями) [2].
5. **Сеансовый уровень (Session, L5):** Управляет сеансами связи между приложениями. Отвечает за установление, поддержание и завершение сеанса (диалога) между двумя приложениями, синхронизацию взаимодействия. В современных сетях функции сеансового уровня часто распределены между приложениями и транспортным уровнем, но концептуально сюда относятся механизмы возобновления прерванных сессий, проверки подлинности сеанса и т.п. Примеры: протоколы управления удалённым доступом или сеансами файловых сервисов (RPC, SQL Session, NetBIOS session service).
6. **Уровень представления (Presentation, L6):** Отвечает за преобразование данных в формат, понятный приложению, включая **синтаксическое представление, шифрование и сжатие данных**. Он обеспечивает, что данные, поступающие от приложений, могут быть корректно интерпретированы на другой стороне. Примеры: шифрование TLS/SSL, форматирование данных в виде JSON, XML, сериализация объектов, кодировки символов (UTF-8 vs. UTF-16).
7. **Прикладной уровень (Application, L7):** Самый верхний уровень, непосредственно взаимодействующий с прикладными процессами пользователя. Он обеспечивает доступ приложений к сетевым сервисам. Здесь работают **сетевые протоколы приложений**, определяющие формат и способ обмена данными конкретного типа. Примеры: HTTP (веб-приложения), SMTP (электронная почта), FTP (передача файлов), DNS (служба доменных имён).

Взаимодействие уровней и инкапсуляция данных: при отправке данные последовательно проходят через каждый уровень, обрастая служебной информацией (заголовками). Этот процесс называется *инкапсуляция* – преобразование пользовательских данных в пакеты низлежащих уровней [3]. На физическом уровне информация передается в виде битов. При получении происходит обратный процесс – *декапсуляция*: полученные на физическом уровне биты поднимаются наверх, каждый уровень удаляет свой заголовок и передает данные выше, вплоть до исходного вида на уровне приложения [3] [4]. Таким образом достигается совместимость: каждый уровень взаимодействует только с соседними, не завися напрямую от других.

Например, **передача файла**: пользователь отправляет файл по сети (уровень приложения, протокол FTP или HTTP). Данные файла на стороне отправителя проходят через транспортный уровень (TCP разбивает на сегменты), сетевой (IP определяет маршрут в виде пакетов), канальный (Ethernet формирует кадры для непосредственной отправки) и физический (биты по кабелю или Wi-Fi). На стороне получателя кадры принимаются сетевой картой, проходят проверку и раскадровку на канальном уровне, сборку пакетов и их маршрутизацию на сетевом, переупорядочение и контроль целостности на транспортном, и наконец передаются

соответствующему приложению [5]. Эти шаги иллюстрируют, как уровни совместно обеспечивают коммуникацию:

1. **Отправитель:** Прикладной уровень передает данные на транспортный; каждый следующий уровень добавляет заголовок и пересылает дальше. Так, транспортный добавляет порты, сетевой – IP-адреса, канальный – MAC-адреса, а физический преобразует в сигналы [5].
2. **Передача:** Данные в виде битов уходят по линии связи (кабелю, эфирному каналу). Возможны промежуточные устройства (маршрутизаторы на L3, коммутаторы на L2), которые обрабатывают соответствующие уровни.
3. **Получатель:** Физический уровень принимает биты, канальный формирует кадры и проверяет их целостность, сетевой собирает из пакетов сообщение и передает транспортному, который отслеживает последовательность и подтверждает получение. Наконец, сеансовый/представления/прикладной обеспечивают, что данные переданы нужному приложению в понятном формате [5].

Акценты для экзаменационной подготовки: важно запомнить названия всех 7 уровней в правильном порядке (от 1-го физического до 7-го прикладного или наоборот). Полезно понимать назначение каждого уровня и уметь кратко сформулировать его функции. Экзаменатор может спросить, например, на каком уровне работает определённый протокол или устройство (маршрутизатор – уровень 3, коммутатор – уровень 2, шлюз приложений – уровень 7), или в чем разница между, скажем, транспортным и сетевым уровнями. Для запоминания последовательности уровней можно использовать мнемонические фразы. Один из способов – запоминать по первой букве названия уровней. Например, на русском: *“Физика КАНет СЕТь ТРудно СЕгодня ПРеподавать ПРактично”* (Физический, Канальный, Сетевой, Транспортный, Сеансовый, Представления, Прикладной) – вы можете придумать свою фразу. Главное – чётко выучить, какие функции выполняет каждый уровень и какие данные (PDU) он обрабатывает. В ответе на экзамене целесообразно перечислить уровни по порядку, подчеркнуть ключевые роли (например, **L3=маршрутизация, L4=надежная доставка** и т.п.) и, при необходимости, привести 1–2 примера протоколов на каждом уровне для иллюстрации.

2. Семь HTTP-методов

HTTP-методы – это набор глаголов протокола HTTP, определяющих действие, которое клиент хочет выполнить над ресурсом на сервере. Базовые методы HTTP (по стандарту HTTP/1.1) включают следующие: **GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS**. Каждый из них служит для своей цели. Рассмотрим кратко все семь методов:

- **GET:** Запрашивает представление ресурса по указанному URI, **получение данных**. Не изменяет состояние ресурса на сервере (метод *только для чтения*). Например, загрузка страницы по URL – типичный GET-запрос. Поддерживает передачу параметров в строке запроса (query string). Обычно ответы на GET кешируются и могут быть повторены без побочных эффектов.
- **POST:** Отправляет данные на сервер для **создания нового ресурса или выполнения серверной операции**. Часто используется при отправке веб-форм, загрузке файлов. В теле POST-запроса передаются данные (например, заполненные пользователем поля формы). В ответ сервер обычно создает новый ресурс (или выполняет действие) и может вернуть его описание или результат. POST неидемпотентен (повторный одинаковый запрос может привести к дублированию).
- **PUT:** Загружает переданные данные по указанному URI, **замещая текущий ресурс**. Если ресурса не было – может создать. Отличается от POST тем, что PUT является

идемпотентным – повторный запрос приведёт к тому же результату, что и один (например, файл будет перезаписан, а не продублирован). Проще говоря, PUT используется для полного обновления ресурса: существующие данные заменяются на новые. Если ресурс уже есть на сервере, PUT его обновит, а не создаст дубликат [6].

- **PATCH:** Похож на PUT, но применяется для **частичного обновления ресурса**. В запросе передаются только изменения, которые нужно внести, а не весь ресурс целиком [7]. PATCH не гарантирует идемпотентности (двукратное применение одного патча может повторно изменить ресурс). Этот метод ввели, чтобы не пересылать лишние данные при небольших изменениях.
- **DELETE:** Удаляет ресурс по заданному URI. После успешного DELETE ресурса на указанном адресе больше нет (может возвращаться код 404 при повторном удалении). Идемпотентен (повторный DELETE обычно не приводит к ошибке). Используется для **удаления данных** на сервере.
- **HEAD:** Запрос идентичен GET, **но без тела ответа** – сервер возвращает только статус и заголовки. Используется, когда нужно узнать информацию о ресурсе (метаданные, наличие, размеры) без передачи самого ресурса. Например, HEAD позволяет проверить, доступен ли файл и каков его размер, не скачивая его полностью [8] [9]. Этот метод является безопасным и идемпотентным.
- **OPTIONS:** Запрашивает у сервера **доступные методы** для указанного ресурса или для всего сервера. В ответ сервер может вернуть заголовок **Allow** со списком поддерживаемых методов, а также другую информацию о возможностях взаимодействия. Часто используется для проверки, что можно делать с ресурсом, а также автоматически применяется браузерами при CORS-запросах (preflight) [10]. Например, клиент может послать OPTIONS-запрос к URI и получить ответ, что поддержаны GET, POST, DELETE и т.п. [8].

Обратите внимание, что методы GET и HEAD не предусматривают тела запроса (тело ответа у HEAD тоже отсутствует), а методы POST, PUT, PATCH обычно отправляют данные в теле. Методы GET, HEAD, OPTIONS считаются **безопасными** (не модифицируют данные на сервере), а методы GET, PUT, DELETE, HEAD и OPTIONS – **идемпотентными** (повторный такой же запрос дает эффект, эквивалентный одному запросу). Эти свойства могут иметь значение при оптимизации кэширования и обработке сбоев.

HTTP-методы vs REST и CRUD: Сами по себе методы HTTP не равны понятиям REST или CRUD – это разные уровни абстракции. REST (*Representational State Transfer*) – это архитектурный стиль для создания веб-сервисов, опирающийся на протокол HTTP и его методы. CRUD – акроним для основных операций с данными: Create (создать), Read (прочитать), Update (обновить), Delete (удалить). Отличие заключается в следующем: **CRUD определяет базовый набор операций с данными**, а **REST определяет принципы взаимодействия клиента и сервера через единообразный интерфейс HTTP** [11]. Проще говоря, CRUD – концепция в контексте хранения данных (например, в базе данных), а REST – архитектурные правила построения API поверх HTTP.

При проектировании RESTful API обычно проводится соответствие между CRUD-операциями и HTTP-методами:
Create → POST,
Read → GET (а также HEAD для получения мета-данных),
Update → PUT (для замены всего ресурса) или PATCH (для частичного изменения),
Delete → DELETE.

REST-сервис использует эти методы для реализации CRUD-действий над ресурсами [12], однако REST – более широкое понятие. В REST помимо выбора корректных HTTP-методов важно

соблюдение и других принципов (адресация ресурсов через URL, отсутствие состояния на сервере между запросами, кодовое описание связностей – HATEOAS, и др.). HTTP протокол же – это лишь транспорт и формат сообщений. Как сказал один из ответов на форуме: *“HTTP – это транспорт, а REST – архитектура для API. У них общего ничего – это как сравнивать тёплое и мягкое”* [13]. То есть HTTP предоставляет набор методов (как перечислено выше), а REST определяет, как грамотно использовать эти методы и URL для построения **API**. Что касается **CRUD**, то он может быть реализован не только через REST/HTTP – например, CRUD-операции есть в SQL для управления записями в таблице. Тем не менее, в контексте веб-разработки часто говорят о сопоставлении CRUD и HTTP-методов в RESTful сервисах. Экзаменационный вопрос может потребовать объяснить эти различия, поэтому помните: **HTTP** – протокол (набор методов и правил передачи данных), **REST** – стиль архитектуры (правила построения взаимодействия поверх протоколов, чаще всего HTTP), а **CRUD** – абстрактные операции с данными, которые могут быть реализованы через HTTP-методы, но не привязаны исключительно к HTTP.

3. Client Side и Server Side

Client Side (клиентская сторона) и **Server Side** (серверная сторона) – понятия, описывающие, где выполняется код или происходит обработка при работе приложения. В контексте веб-разработки **клиентская сторона** обычно означает код, выполняющийся в браузере пользователя (на устройстве пользователя), а **серверная сторона** – код, выполняющийся на сервере. Проще говоря, клиентская сторона – это то, что *видит и с чем взаимодействует пользователь* (например, HTML/CSS-разметка, выполнение JavaScript в браузере для динамики страницы), а серверная – это своеобразное *закулисье*, где происходят расчёты, обработка данных, доступ к базе данных и логика, невидимые напрямую пользователю [14].

На клиентской стороне работают технологии вроде **HTML/CSS/JavaScript**: они отвечают за отображение интерфейса, проверку введённых данных (например, валидация форм через JS), динамическое изменение содержимого страницы без перезагрузки (AJAX-запросы, работа с DOM) и т.д. Этот код загружается с сервера и исполняется в браузере, поэтому ограничен возможностями и безопасностью окружения браузера. Например, JavaScript на клиенте не может напрямую записывать файлы на диск пользователя без его участия и изолирован от доступа к серверным ресурсам (базам данных) – вместо этого он может отправлять запросы к серверу.

На серверной стороне работают **языки и фреймворки на сервере** (например, Python, PHP, Java, Node.js, Ruby и др.), и код там выполняется на удалённой машине (сервере). Серверная логика получает запросы от клиентов, обрабатывает их (взаимодействует с базой данных, файловой системой, внешними сервисами), и формирует ответ (например, HTML-страницу или JSON-данные) обратно клиенту. Пользователь не видит напрямую, что происходит на сервере – он получает только результат обработки. Таким образом, **серверная часть** отвечает за хранение данных, бизнес-логику приложения, безопасность (проверку прав доступа, шифрование данных), интеграцию с другими системами.

Обычно веб-приложение организовано по модели *“клиент-сервер”*: браузер (клиент) посылает HTTP-запросы (например, запрос страницы или отправка формы) на веб-сервер; сервер (backend) обрабатывает запрос, например, код на Python/PHP/Java обращается к базе данных, получает или изменяет нужные данные, формирует HTML или JSON и возвращает ответ; затем браузер отображает полученные данные пользователю. Такое разделение позволяет использовать возможности каждого из окружений: клиентское – для интерактивности и интерфейса, серверное – для надежного хранения и сложных вычислений.

Для наглядности реализуем **минимальный пример клиента и сервера на Python** с использованием библиотеки **aiohttp** (асинхронный фреймворк для веб-сервера и HTTP-клиента). Сервер будет при обращении возвращать строку `"Hello from server"`, а клиент – отправлять запрос к серверу и выводить ответ.

Простой сервер на aiohttp

Ниже приведён код простейшего веб-сервера. Он слушает входящие HTTP-запросы на указанном хосте/порту и отвечает фиксированной фразой. Код содержит комментарии, поясняющие каждый шаг:

```
# Импортируем необходимые модули
from aiohttp import web

# Определяем обработчик для входящих запросов
async def handle(request):
    # Функция вызывается при получении запроса
    # Возвращаем простой текстовый ответ
    return web.Response(text="Hello from server")

# Создаем приложение aiohttp
app = web.Application()
# Регистрируем маршрут: при GET-запросе к корню ("/") вызвать функцию handle
app.add_routes([web.get('/', handle)])

# Запускаем веб-сервер на локальном хосте (127.0.0.1) и порту 8080
if __name__ == "__main__":
    web.run_app(app, host="localhost", port=8080)
```

Пояснения к серверу:
– Мы использовали `aiohttp.web` для создания сервера. Определили асинхронную функцию `handle(request)`, которая возвращает объект `web.Response` с текстом. Эта функция привязана к маршруту `'/'` методом GET.
– `web.Application()` представляет приложение (веб-сервер). Мы добавили маршрут `app.add_routes([web.get('/', handle)])`, что означает: при GET-запросе к пути `/` вызывать нашу функцию `handle`.
– В конце `web.run_app(app, ...)` запускает сервер (блокирующая операция, запускает цикл обработки событий). Сервер начинает слушать порт 8080 на локальной машине. Если вы запустите этот скрипт, он будет работать до остановки (Ctrl+C), принимая запросы.

Простой HTTP-клиент на aiohttp

Теперь приведём код клиента, который отправляет запрос на наш сервер и получает ответ. Этот код можно выполнить в другом терминале пока запущен сервер:

```
import asyncio
import aiohttp

# Асинхронная функция для выполнения HTTP-запроса
```

```python
async def main():
    # Инициализируем сессию клиента
    async with aiohttp.ClientSession() as session:
        # Отправляем GET-запрос на локальный сервер (предполагаем, что он
слушает localhost:8080)
        async with session.get("http://localhost:8080/") as resp:
            # Читаем ответ как текст
            text = await resp.text()
            print("Ответ сервера:", text)

# Запускаем асинхронную функцию
asyncio.run(main())
```

Пояснения к клиенту:
– Здесь мы используем `aiohttp.ClientSession` для выполнения HTTP-запросов. В контексте `async with session.get(...) as resp` отправляется GET-запрос по указанному URL. В нашем случае это адрес локального сервера.
– `resp` – это объект ответа. Мы вызываем `await resp.text()`, чтобы получить тело ответа как строку (в нашем сервере это будет `"Hello from server"`). Затем выводим полученный текст.
– Код клиента запускается через `asyncio.run(main())`, поскольку `main()` определена как асинхронная. Внутри нее происходит всё взаимодействие.

Пошаговая инструкция для запуска примера

1. **Установить aiohttp:** Если не установлена библиотека aiohttp, сначала выполните установку командой `pip install aiohttp`. Убедитесь, что используете Python 3.7+ (поскольку asyncio).
2. **Сохранить код сервера:** Создайте файл, например `server.py`, и вставьте туда приведённый код сервера на aiohttp. Сохраните.
3. **Запустить сервер:** В терминале выполните `python server.py`. Должно появиться сообщение о запуске приложения (например, "Running on http://localhost:8080 ..."). Теперь сервер ждёт входящих соединений.
4. **Сохранить код клиента:** Создайте отдельный файл, например `client.py`, и вставьте код клиента. Сохраните.
5. **Запустить клиент:** Откройте второй терминал/окно и выполните `python client.py`. Клиентская программа должна подключиться к серверу, отправить GET-запрос и вывести ответ. Вы увидите вывод: `Ответ сервера: Hello from server`.
6. **Завершить работу:** Остановите сервер (`Ctrl+C` в окне, где он запущен), когда закончите эксперимент. Клиент завершится автоматически после получения ответа.

Этот пример демонстрирует принцип *client-side vs server-side* на минимальном уровне: серверная программа (server.py) постоянно работает и отвечает на запросы, а клиентская (client.py) разово запускается, делает запрос и завершается. В реальном веб-приложении роль **клиентской стороны** выполняет браузер пользователя (HTML/JS), а роль **серверной** – веб-сервер (код, генерирующий ответы). Наш пример с aiohttp иллюстрирует, как они общаются через HTTP: клиент отправил запрос – сервер обработал и вернул ответ.

4. Что такое web-scraping (веб-скрейпинг)

Web-scraping (веб-скрейпинг) – это технология автоматического извлечения данных с веб-сайтов [15]. Говоря простыми словами, программа-скрейпер отправляет HTTP-запросы к веб-страницам (обычно GET), получает HTML-код и вычленяет из него нужную информацию. Веб-скрейпинг может выполняться и вручную (человек копирует информацию с сайта), но обычно под этим термином понимают автоматизированный сбор данных с помощью кода [15].

Когда же **уместно использовать веб-скрейпинг?** Он применим, когда нужно собрать информацию со сторонних сайтов, которые не предоставляют удобного API или выгрузки данных. Например, веб-скрейпинг используют компании для мониторинга цен конкурентов (парсинг страниц интернет-магазинов на предмет цен и наличия товара), для агрегирования отзывов или объявлений с разных площадок, для научных исследований (сбор данных с новостных сайтов, соцсетей) или SEO-анализа (сбор информации о выданных поисковыми системами результатах). Веб-скрейпинг стал важным инструментом автоматизированного сбора информации в интернете [16] [17]. Скрейперы могут регулярно обходить веб-страницы, извлекать обновления (например, новые статьи, изменения на сайте) – т.е. выполнять мониторинг изменений на сайтах [17]. Ещё случаи: сбор контактных данных (имен, телефонов, email) из общедоступных источников, формирование собственных баз данных из открытой информации, интеграция данных с сайтов, которые официально не предоставляют API. По сути, **веб-скрейпинг — это "парсинг" сайтов**: программа получает HTML-разметку страницы и *парсит* (разбирает) её, выделяя определённые элементы (тексты, ссылки, таблицы и т.д.).

Конечно, применять веб-скрейпинг следует этично и осторожно. Перед парсингом стоит ознакомиться с **правилами использования сайтов**: некоторые прямо запрещают автоматическое извлечение данных в своих условиях или блокируют слишком частые запросы. Также полезно проверять файл `robots.txt` сайта – он указывает, какие разделы разрешены или запрещены для автоматических агентов (роботов). Ещё один аспект – нагрузка: плохо написанный скрейпер, штурмующий сайт сотнями запросов в секунду, может создать избыточную нагрузку. Поэтому рекомендуется соблюдать вежливость: не превышать разумную частоту запросов, кэшировать результаты, если возможно, и не обходить механизмы защиты от ботов. Если есть **официальное API**, зачастую лучше использовать его, чем парсить HTML, так как API дает данные в структурированном виде и с согласия владельцев. Веб-скрейпинг уместен, когда данные открыты для просмотра пользователями, но недоступны в виде выгрузки – в таких случаях программный парсинг автоматизирует то, что можно было бы сделать вручную через браузер.

Пример: простая программа веб-скрейпера с BeautifulSoup

Рассмотрим небольшой пример на Python, который берет веб-страницу и извлекает из нее заголовки (теги `<h1>`). Для парсинга HTML воспользуемся библиотекой **BeautifulSoup** (из пакета `beautifulsoup4`), а для выполнения HTTP-запросов – библиотекой **Requests**. (Перед запуском убедитесь, что установили эти библиотеки: `pip install requests beautifulsoup4`).

Допустим, мы хотим получить заголовок страницы с адресом `https://www.example.com`. Вот код скрипта и пояснения:

```
import requests
from bs4 import BeautifulSoup
```

```
url = "https://www.example.com"
# Выполняем GET-запрос к указанному URL
response = requests.get(url)

# Парсим HTML-код ответа с помощью BeautifulSoup
soup = BeautifulSoup(response.text, "html.parser")

# Находим все теги <h1> (заголовки первого уровня) на странице
titles = soup.find_all('h1')

# Выводим текст каждого найденного заголовка
for title in titles:
    print(title.text)
```

Что делает этот код:

- Импортируем библиотеку `requests` для HTTP-запросов и `BeautifulSoup` из `bs4` для парсинга HTML.
- Задаём переменную `url` со ссылкой на страницу. В примере используется `example.com` – это демонстрационный сайт, который всегда доступен и содержит простую страницу.
- Функция `requests.get(url)` выполняет HTTP GET-запрос к данному адресу и возвращает объект ответа. В `response.text` находится HTML-код полученной страницы.
- Создаем объект супа: `BeautifulSoup(response.text, "html.parser")`. Это инициализация парсера HTML – BeautifulSoup разбирает текст HTML для удобного поиска по элементам.
- Метод `soup.find_all('h1')` находит **список всех тегов** `<h1>` на странице. (Если нужно было бы собрать заголовки разных уровней, можно искать и `<h2>` и т.д., либо использовать более сложные селекторы. В данном простом случае ищем только первый уровень.)
- Далее перебираем найденные теги `h1` и выводим их содержимое (`title.text`). Свойство `.text` у объекта тега возвращает текст внутри этого элемента (без HTML-разметки). В случае `example.com` на странице обычно есть один `<h1>` с текстом "Example Domain", поэтому на консоль будет выведено именно **Example Domain**. Если бы заголовков было несколько, код вывел бы каждый с новой строки.

Запустив такой скрипт, вы получите в выводе все собранные заголовки. Таким образом, с помощью BeautifulSoup очень удобно извлекать нужные фрагменты из HTML по тегам, классам или другим признакам, а Requests упрощает получение страницы. Этот подход можно расширять: например, собрать все ссылки (`<a href>`), все элементы определённого класса CSS и т.д.

Итог: веб-скрейпинг – мощный инструмент для автоматизации сбора данных с веб-страниц. В ответе на экзамене стоит подчеркнуть, что это программный метод извлечения информации из HTML, объяснить, когда он применяется (сбор данных, мониторинг) и показать понимание базовых техник (HTTP-запрос, разбор HTML). Приведённый код иллюстрирует минимальный скрейпер: он делает запрос и парсит HTML с помощью библиотеки. В реальных задачах скрейпинг может включать обход нескольких страниц (краулинг), ожидание контента (если сайт динамически загружает данные через JS), обработку исключений (например, если сайт недоступен) и уважение ограничений сайта. Но основные принципы остаются: получить страницу, выделить нужные данные и использовать их (сохранить, проанализировать). Это следует понимать и уметь объяснить, приводя примеры, как в данном простом случае.

1 Сетевая модель OSI — Википедия
https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_OSI

2 5 Что такое модель OSI? – Объяснение 7 уровней OSI – AWS
https://aws.amazon.com/ru/what-is/osi-model/

3 4 Модель OSI. 7 уровней сетевой модели OSI с примерами
https://selectel.ru/blog/osi-for-beginners/

6 7 HTTP-запросы: структура, методы, заголовки
https://firstvds.ru/technology/metody-http-zaprosa

8 9 10 веб программирование - В чем отличие между HTTP методами HEAD и OPTIONS? - Stack Overflow на русском
https://ru.stackoverflow.com/questions/707649/%D0%92-%D1%87%D0%B5%D0%BC-%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B8%D0%B5-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-http-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B0%D0%BC%D0%B8-head-%D0%B8-options

11 12 Что такое CRUD? Его функции, преимущества и примеры
https://highload.tech/chto-takoe-crud-prostymi-slovami-funktsii-preimushhestva-i-primery/

13 веб программирование - В чем отличие REST API от HTTP? - Stack Overflow на русском
https://ru.stackoverflow.com/questions/1504286/%D0%92-%D1%87%D0%B5%D0%BC-%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B8%D0%B5-rest-api-%D0%BE%D1%82-http

14 Разница и взаимодействие client-side и server-side кода - Skypro
https://sky.pro/wiki/javascript/raznitsa-i-vzaimodeystvie-client-side-i-server-side-koda/

15 16 17 Веб-скрейпинг — Википедия
https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%BA%D1%80%D0%B5%D0%B9%D0%BF%D0%B8%D0%BD%D0%B3