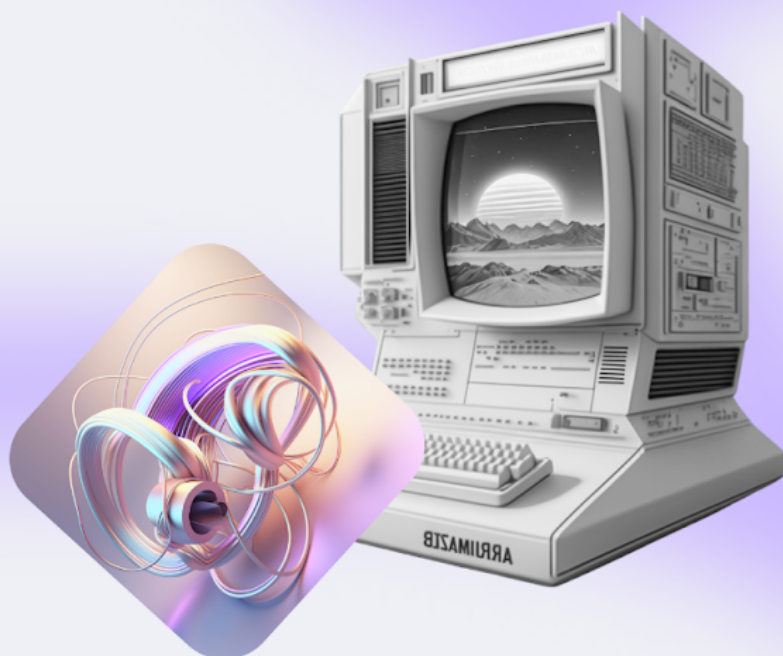


ОСНОВЫ КЛИЕНТ-СЕРВЕРНОГО ВЗАИМОДЕЙСТВИЯ. ПАРСИНГ API

Сбор и разметка данных



Оглавление

Словарь терминов	3
Введение	4
Сбор и разметка данных	4
Парсинг данных	6
Data-специалисты	6
Взаимодействие клиента и сервера	7
Введение в Web APIs	9
Протоколы прикладного уровня (Application) в OSI-модели	14
Representational State Transfer (REST)	19
Клиент-серверный поток HTTP	21
Создание HTTP-запросов в Postman	24
Создание HTTP-запросов в Python	28
Создание датафрейма в Jupyter-ноутбуке	30
Заключение	32
Что можно почитать еще?	33

Словарь терминов

API (Application Programming Interface) — описание способов взаимодействия одной компьютерной программы с другими.

Мэшап (mashup) — это веб-приложение, объединяющее данные из нескольких источников в один инструмент. Например, можно объединить картографические данные Google Maps с данными о недвижимости от Craigslist и получить новый веб-сервис, который изначально не предлагал ни один из источников данных.

Клиент-сервер (client-server) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг (серверами) и заказчиками услуг (клиентами).

Протокол — набор правил и действий (очередности действий), который позволяет осуществлять соединение и обмен данными между двумя и более включенными в сеть устройствами.

OSI (The Open Systems Interconnection model) — сетевая модель стека (магазина) сетевых протоколов OSI/ISO. С помощью этой модели разные сетевые устройства могут взаимодействовать друг с другом.

HTTP (Hypertext Transfer Protocol) — протокол прикладного уровня передачи данных. Изначально — в виде гипертекстовых документов в формате HTML. Сейчас — для передачи произвольных данных.

SOAP (Simple Object Access Protocol) — протокол обмена структурированными сообщениями в распределенной вычислительной среде. Используется для обмена произвольными сообщениями в формате XML.

REST (Representational State Transfer) — набор правил о том, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

JSON (JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript.

URI (Uniform Resource Identifier) — унифицированный (единообразный) идентификатор ресурса. Последовательность символов, которая идентифицирует какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и так далее.

Введение

Добро пожаловать на курс «Сбор и разметка данных»!

Меня зовут Петр Рубин, я врач и дата-сайентист. Моя работа на протяжении более пятнадцати лет была связана с научными исследованиями в области медицины. Я хочу поделиться с вами своими знаниями и надеюсь, что наша работа будет продуктивной и полезной.

За последнее десятилетие работа с данными кардинально трансформировалась. В первую очередь, это обусловлено быстрым ростом объема данных и растущим спросом на высококачественные датасеты.

Если 15–20 лет назад сбор данных часто заключался в ручном заполнении документов и в ручном переносе их в электронный вид, то сегодня, с развитием машинного обучения и искусственного интеллекта, произошла революция в способах сбора и разметки данных. Автоматизированные инструменты и библиотеки упростили извлечение больших объемов данных с веб-сайтов. Кроме того, появились инструменты на основе искусственного интеллекта, которые позволяют упростить процесс разметки и уменьшить потребность в человеческих усилиях.

В нашем курсе 9 лекций и много практики. Мы разберем основы клиент-серверного взаимодействия, принцип сбора информации из разных интернет-источников с помощью Python, работу с NoSQL-системами, управление базами данных, парсинг API и HTML, поговорим о работе с данными и их разметке.

На первой лекции мы рассмотрим основы взаимодействия «клиент-сервер» и то, как API используют для облегчения этого взаимодействия, а также займемся парсингом API. Вы узнаете о различных форматах ответов и о том, как с ними работать.

Чтобы погрузиться в тему было проще, разберем несколько вводных понятий.

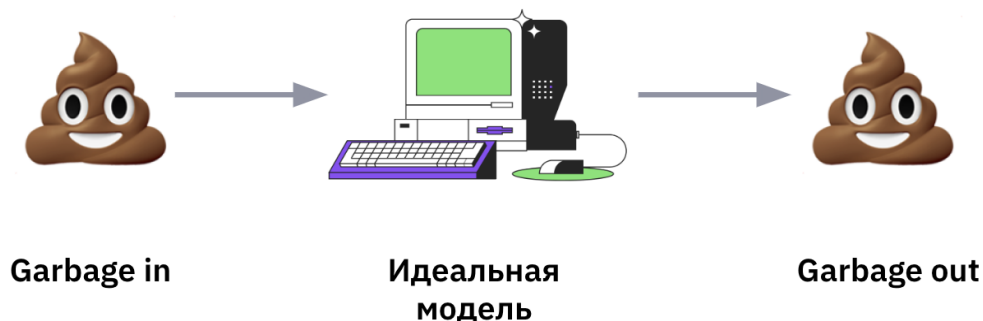
Сбор и разметка данных

В рамках курса мы рассмотрим основные концепции, связанные со сбором и разметкой данных. Вы узнаете, как извлекать их из разных источников, и как правильно их размечать (аннотировать), то есть присваивать метки, теги или категории с целью сделать их осмысленными и удобными для использования.

Надежность выводов, полученных в результате анализа, полностью зависит от качества сбора и разметки данных. Поэтому важно помнить:



Garbage in, garbage out (мусор на входе — мусор на выходе)



Garbage in, garbage out (мусор на входе — мусор на выходе) — это концепция, согласно которой неполноценные или бессмысленные (мусорные) входные данные приводят к бессмысленному выводу, даже если сам по себе алгоритм анализа правильный.

Garbage in — это неструктурированные и некачественные входные данные. Они неточные, неполные, непоследовательные, в них может дублироваться информация. Этими проблемами страдают все данные, которые собираются в необработанном виде. Например, данные соцсетей часто неструктурированные, их нужно обработать, прежде чем использовать для анализа.

Примеры плохих данных:

- маленькая выборка (недостаточно данных),
- нерепрезентативная выборка,
- несбалансированная выборка,
- отсутствующие или пропущенные данные,
- устаревшая информация,
- данные введены в неправильное поле,
- дублирование записей,
- ошибки, опечатки и орфографические вариации.

Проблемы кажутся несущественными, но они — причина плохих данных и серьезное узкое место в случае, когда эти данные нужно перенести в платформу бизнес-аналитики или использовать для анализа.

Согласно исследованию Gartner¹, компании оценивают, что плохие данные обходятся им почти в 13 миллионов долларов в год. Влияние плохих данных на отделы продаж и маркетинга может варьироваться от неточного таргетинга (препятствие для поиска клиентов) до слабых продаж.

Парсинг данных

Парсинг данных — это процесс получения данных в одном формате и преобразования их в другой формат. Парсеры используются повсеместно, например, в компиляторах, когда из компьютерного кода генерируется машинный код. Парсеры есть в SQL-движках: SQL-движки парсят SQL-запрос, выполняют его и возвращают результаты.

В случае веб-скрейпинга парсинг обычно выполняется после того, как данные были извлечены из веб-страницы.

Веб-скрейпинг (скрепинг, скрапинг) — это технология получения данных путем извлечения их со страниц веб-ресурсов. После того как вы собрали данные из интернета, следующий шаг — сделать их более читабельными и пригодными для анализа.

Хороший парсер данных не ограничивается определенными форматами. У вас должна быть возможность вводить данные любого типа и получать данные другого типа: например, преобразовывать необработанный HTML в JSON-объект.

Парсеры широко используются в веб-скрейпинге, потому что сырой HTML, который мы получаем, нелегко прочитать. Нужно, чтобы данные были преобразованы в формат, который сможет понять человек. Например, можно генерировать отчеты из HTML или создавать таблицы для отображения важной информации.

Data-специалисты

В сборе и разметке данных могут участвовать разные специалисты, например:

Data Collector — сборщики данных. Отвечают за получение данных из разных источников: API, баз данных, путем веб-скрейпинга. Используют инструменты и

¹ Gartner — исследовательская и консалтинговая компания США. Специализируется на рынках информационных технологий.

методы для извлечения, очистки, предварительной обработки данных и подготовки к разметке.

Data Labeler — специалисты по разметке данных. Отвечают за разметку — добавляют к данным метки, теги или категории, чтобы упорядочить их и облегчить поиск. Помогают обучать алгоритмы машинного обучения.

Data Engineer — дата-инженеры. Разрабатывают и внедряют инфраструктуру сбора и разметки данных — инструменты и рабочие процессы. Могут тесно сотрудничать со сборщиками и разметчиками данных, чтобы обеспечить эффективность и результативность процесса сбора и разметки.

Data Scientist — специалисты по анализу данных. Используют размеченные данные для построения и обучения моделей машинного обучения, а также для анализа данных с целью получить информацию и составить прогнозы. Дата-сайентисты могут тесно работать со специалистами, перечисленными выше, а могут самостоятельно заниматься сбором или разметкой данных.

Взаимодействие клиента и сервера

В контексте взаимодействия «клиент-сервер» клиент и сервер — это две отдельные сущности, которые взаимодействуют друг с другом для обмена данными и для выполнения задач.

Клиент — это устройство или приложение, которое запрашивает данные или услуги у сервера. Клиентами могут быть настольные компьютеры, смартфоны или любые другие устройства, способные отправлять запросы на сервер.

Сервер — это устройство или приложение, которое предоставляет данные или услуги клиентам. Серверы могут быть физическими компьютерами, виртуальными машинами или облачными сервисами. Они обрабатывают входящие запросы от клиентов и отвечают на них, предоставляя запрошенные данные или услуги.

Отношения «клиент-сервер» основаны на цикле «запрос-ответ»: клиент отправляет запрос на сервер, а сервер отвечает запрошенными данными или информацией. Такое разделение обязанностей позволяет создавать масштабируемые и гибкие системы, поскольку клиент и сервер могут разрабатываться и внедряться независимо друг от друга.

Цикл «запрос-ответ» можно разбить на этапы:

1. **Клиент посылает запрос на сервер.** Запрос обычно отправляется по определенному протоколу, например HTTP или HTTPS, и содержит информацию о данных или услугах, которые запрашивает клиент.
2. **Сервер получает запрос и обрабатывает его.** Использует информацию из запроса, чтобы определить, какие данные или услуги запрашиваются и как ответить на запрос.
3. **Сервер отправляет ответ клиенту.** Ответ содержит запрошенные данные или информацию, а также коды состояния и другую информацию о запросе.
4. **Клиент получает ответ и обрабатывает информацию.** Использует информацию в ответе, чтобы отобразить запрошенные данные или выполнить задачу.

Цикл «запрос-ответ» может происходить несколько раз в рамках одного взаимодействия клиента и сервера. Например, клиент может отправить несколько запросов на сервер, а сервер ответит на каждый запрос соответствующей информацией.

Взаимодействие «клиент-сервер» может принимать разные формы в зависимости от требований приложения и типа данных, которыми обмениваются.

Рассмотрим распространенные типы взаимодействия «клиент-сервер».

- **Простой запрос-ответ.** Базовая форма взаимодействия: клиент отправляет на сервер один запрос, сервер отвечает одним ответом. Этот тип взаимодействия используется для простых задач: например, для получения веб-страницы или отправки формы.
- **Stateful-взаимодействие.** Сервер сохраняет информацию о состоянии взаимодействия между клиентом и сервером. Этот тип взаимодействия позволяет выполнять более сложные взаимодействия, где серверу нужно отслеживать сессию клиента и данные, которыми он обменивается: например, онлайн-покупки или банковские операции.

Например, когда вы заходите на сайт интернет-магазина и начинаете просматривать товары, сервер отслеживает просмотренные товары, корзину и другие сведения о вашей сессии. Эта информация хранится на сервере, и сервер использует ее, чтобы подход к вам был персонализированным.

Противоположность — **stateless-взаимодействие**. Сервер не отслеживает состояние взаимодействия, каждый запрос рассматривается как независимая транзакция.

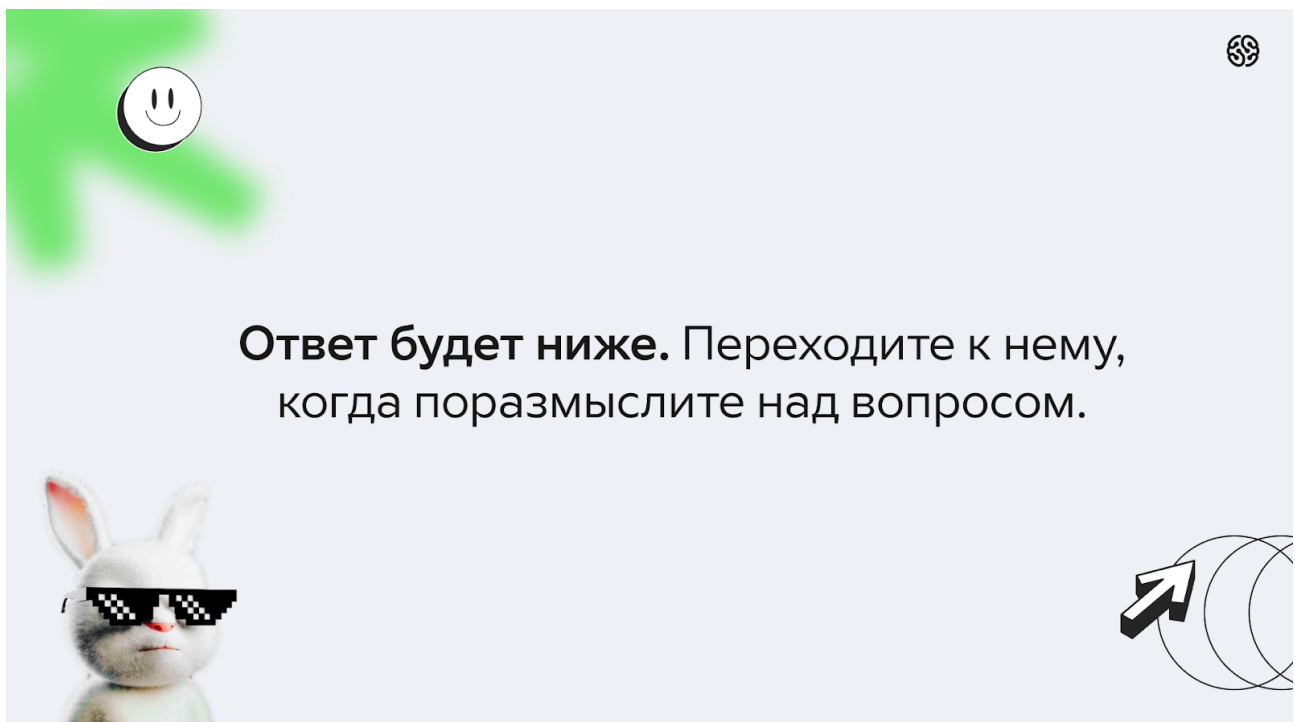
- **Взаимодействие в реальном времени.** Сервер и клиент обмениваются данными в режиме реального времени, причем обмен данными происходит по мере их поступления. Этот тип взаимодействия используется в онлайн-играх или видеоконференциях, где решающую роль играют низкая задержка и высокая скорость связи.

У каждого типа взаимодействия «клиент-сервер» есть набор требований. Какой выбрать — зависит от потребностей приложения.

Введение в Web APIs

API — это основа программной инфраструктуры, которая обеспечивает согласованную работу разных программ. Чтобы разобраться было проще, приведем аналогию: способ коммуникации между людьми — это речь, а между программами — API.

Как вы думаете, зачем нужна коммуникация между программами?



Одна из задач коммуникации — получение или передача информации. Разберем на примере владельца кофейни. Ему нужно предоставлять услугу (варить кофе), мыть посуду, давать рекламу, вести бухучет и выполнять еще много задач. Он может изучить нужную информацию и выполнять их самостоятельно. А может с помощью речи договориться с другими специалистами, у которых есть нужные знания и навыки, и нанять их на работу.

Так и с программами. Если вы создаете интернет-магазин, теоретически, можете написать с нуля все его компоненты: каталог, систему приема платежей, CRM и так далее. Но удобнее использовать уже готовые компоненты. А взаимодействие с этими компонентами происходит через API.

API (Application Programming Interface) — это описание способа коммуникации между двумя единицами кода. Иными словами, способ связи между двумя программами.

Например, когда вы копируете текст из браузера и вставляете его в редактор, выполнить эту операцию позволяет API внутри компьютера — Clipboard API, часть операционной системы, которая предоставляет приложениям общий интерфейс для доступа к системному буферу обмена.

Когда вы копируете текст из браузера, он сохраняется в системном буфере обмена — временной области хранения информации, которая вырезается, копируется или вставляется. API буфера обмена позволяет текстовому редактору получить доступ к информации, хранящейся в буфере обмена, и вставить ее.

Если вы слушаете музыку на компьютере или в смартфоне, приложение-плеер связывается с API вашей операционной системы. При этом API дает возможность использовать закрытый код (closed-source applications). Некоторые компании, например, Microsoft или Apple, позволяют использовать функции своих приложений, не давая доступ к коду.

Есть несколько типов API. У каждого своя специфика:

- **Открытые API** (внешние или публичные) доступны для разработчиков и пользователей за пределами организации, создавшей API. Позволяют получить доступ к данным, услугам или функциональности, которые можно использовать другими приложениями.
- **Внутренние API.** Используются внутри организации, чтобы приложения и службы могли взаимодействовать друг с другом. Подходят для обмена данными, услугами или функциональностью внутри организации. Не предназначены для внешнего использования.
- **Партнерские API.** Используются между организациями для обмена данными и услугами. Подходят для обмена данными, услугами или функциональностью между двумя организациями. Не предназначены для внешнего использования.

API позволяет создать новое приложение из частей кода, расположенных на разных компьютерах (mashups). Например, из одного сервера можно получить информацию о трафике, а из другого — картографические данные.

Почему это важно? Например, наш новый веб-сервис может использовать эквайринг одного из банков, Яндекс Карты, сервис чат-бота, мейл-автоответчик и так далее. Образуется сеть, в которой участники используют сервисы друг друга и получают выгоду для бизнеса.

Другая важная функция API — получение большого количества накопленных данных для аналитики. Например, вы можете разработать сервис, который дает рекомендацию бизнесу о том, где лучше открыть торговую точку. Для этого можно получить данные по API Google Maps и создать рекомендательную модель машинного обучения.

Может возникнуть вопрос: как мы можем быть уверены, что нам удастся объединить в одной программе сервисы разных компаний, которые используют разные операционные системы, спецификации и структуры данных? Как мы можем быть уверены, что все эти устройства смогут коммуницировать друг с другом?

Чтобы машины говорили на одном языке, нужно установить правила — **протоколы**.

Продолжим аналогию с коммуникацией между людьми. Чтобы понимать друг друга, мы должны установить правила. Первое — договориться, на каком языке будем общаться. Второе — понять, что речь — это инструмент высокого уровня, говоря компьютерными терминами. То есть, используя речь, мы не задумываемся о том, как она функционирует, но на самом деле понимаем, что есть речевой аппарат, который функционирует благодаря сложной нервной регуляции и в итоге вызывает колебания воздуха.

Так и API — это инструмент высокого уровня. Чтобы немного заглянуть под капот, разберемся, что такое модель OSI.

Open System Interconnection (OSI) model — абстрактная модель, которая обеспечивает коммуникацию компьютерных систем, невзирая на их системные характеристики.

Оригинальная версия состоит из 7 уровней. Каждый уровень обслуживает уровень выше и обслуживается уровнем ниже. В каждом уровне может быть множество протоколов, но у всех должна быть возможность получать и передавать данные в соседние слои.

Open System Interconnection (OSI) model



Представьте, что вы отправляете электронное письмо со своего компьютера на компьютер друга. Письмо написано и готово к отправке. Чтобы отправить его, нужно выполнить несколько шагов. Каждый из них соответствует уровню модели OSI:

1. **Application Layer.** Прикладной уровень. Программа-клиент электронной почты на вашем компьютере отправляет письмо на почтовый сервер, используя простой протокол передачи почты (SMTP).
2. **Presentation Layer** отвечает за представление данных, включая их сжатие, шифрование и форматирование. При отправке электронной почты этот уровень может преобразовывать данные в определенный формат, например ASCII или UTF-8, чтобы гарантировать, что данные могут быть правильно интерпретированы получателем. Presentation Layer также может шифровать данные, чтобы обеспечить конфиденциальность электронной почты.
3. **Session Layer** отвечает за создание, поддержание и завершение сеансов между приложениями. При отправке электронной почты Session Layer устанавливает сеанс между программой клиента электронной почты на компьютере отправителя и почтовым сервером. Session Layer поддерживает сеанс во время передачи электронного письма и завершает его, когда письмо успешно доставлено на почтовый сервер.
4. **Transport Layer.** Программа-клиент электронной почты делит электронное письмо на мелкие пакеты данных и добавляет информацию об источнике и получателе каждого пакета. Затем транспортный уровень отправляет пакеты на сетевой уровень.

5. **Network Layer.** Сетевой уровень использует информацию, содержащуюся в каждом пакете, чтобы определить лучший маршрут, по которому пакет достигнет почтового сервера. Сетевой уровень также добавляет информацию об источнике и пункте назначения каждого пакета, например, IP-адреса компьютеров, участвующих в коммуникации.
6. **Data Link Layer.** Этот уровень добавляет информацию к каждому пакету, например, коды коррекции ошибок, чтобы обеспечить надежную передачу пакета. Уровень Data Link также разделяет пакеты на мелкие единицы данных (frames) и отправляет их на физический уровень.
7. **Physical Layer.** Физический уровень отвечает за физическую передачу данных. Преобразует данные в электрические сигналы, которые передаются по сети на почтовый сервер.

На почтовом сервере происходит обратный процесс, и электронное письмо принимается и сохраняется в папке входящих сообщений клиентской программы электронной почты вашего друга.

При создании приложений, использующих API, разработчики работают на уровне Application. Операционные системы, как правило, выполняют все нижележащие задачи.

Подробнее про уровни: [OSI model - Wikipedia](#).

Вернемся к нашей основной задаче на сегодня — сбору и парсингу данных с помощью API. Допустим, мы выяснили, что какой-то ресурс предоставляет нужные нам данные (например, Google Maps предоставляет картографические данные). Мы хотим эти данные получить.

Сперва нужно «попросить» удаленный сервер предоставить данные, то есть сделать запрос с помощью API. API позволяет связать наш компьютер с удаленным компьютером независимо от того, какие операционные системы на них установлены. Значит, разные компьютеры должны говорить на одном языке, то есть использовать общие протоколы. Давайте разберемся с этим.

Протоколы прикладного уровня (Application) в OSI-модели

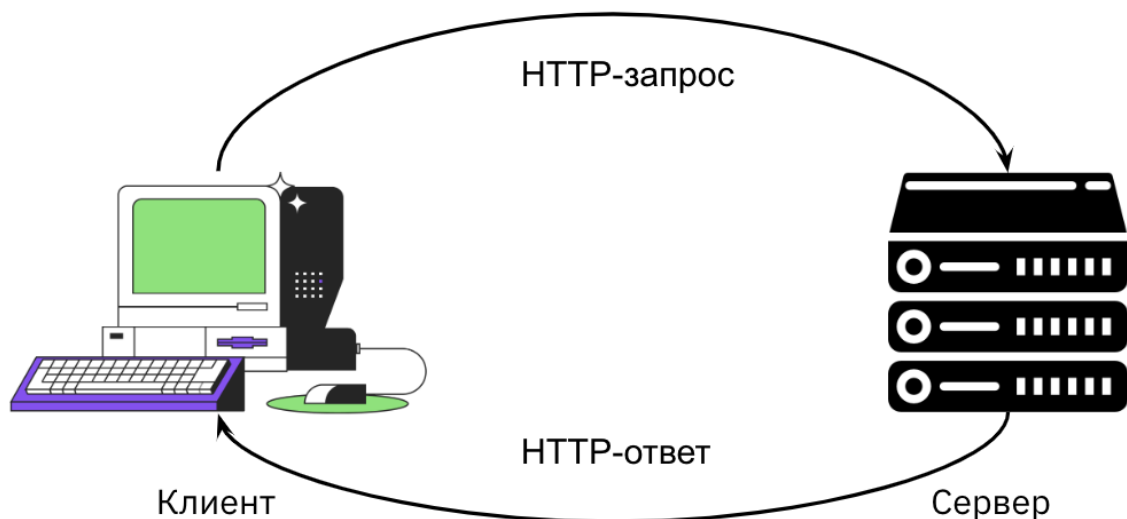
На уровне Application находятся протоколы, с помощью которых приложения обмениваются данными. Это может быть передача файлов, электронная почта и просмотр веб-страниц. Прикладной уровень использует протоколы для выполнения своих функций.

Протоколы прикладного уровня — это стандарты связи, которые определяют правила и форматы обмена данными между приложениями.

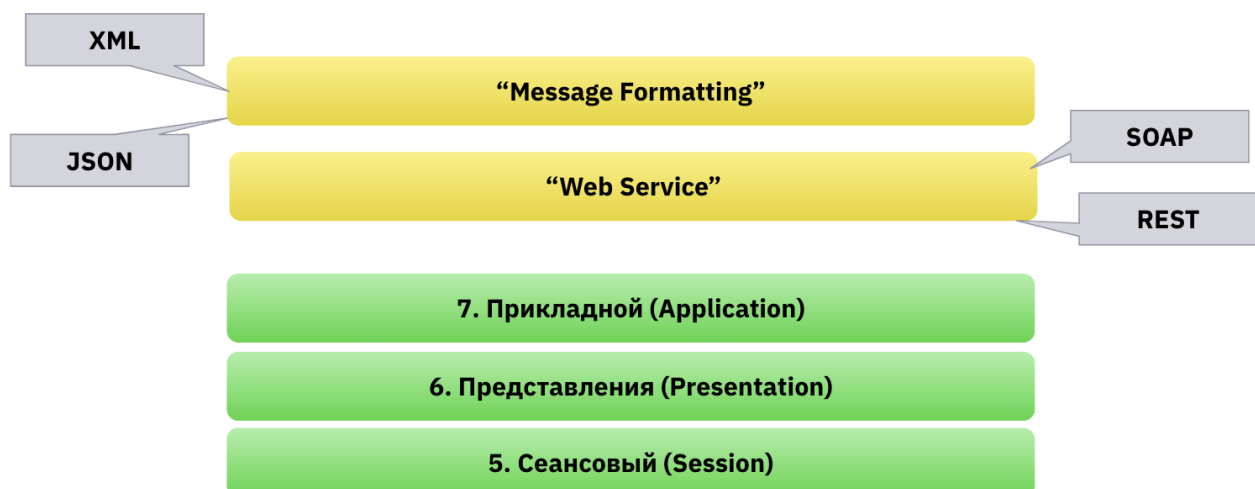
Распространенные протоколы прикладного уровня:

- **HTTP** (протокол передачи гипертекста) — основной протокол для передачи данных в интернете. Используется для отправки и получения веб-страниц, а также других типов данных. Основа интернета.
- **FTP** (протокол передачи файлов) — протокол для передачи файлов между компьютерами в сети. Рассчитан на загрузку и выгрузку файлов. Часто используется для передачи больших файлов: музыки, видео и ПО.
- **SMTP** (простой протокол передачи почты) — основной протокол для отправки и получения электронной почты. Используется для передачи email-сообщений с одного почтового сервера на другой. Основа системы электронной почты.
- **DNS** (система доменных имен) — протокол для преобразования доменных имен в IP-адреса. Используется для преобразования читабельных доменных имен (www.example.com) в машиночитаемые IP-адреса (192.0.2.1).
- **Telnet** — протокол для удаленных терминальных соединений. Позволяет пользователям входить на удаленный компьютер и получать доступ к его ресурсам (например, файлам и приложениям), как если бы они были физически подключены к компьютеру.

Это лишь некоторые из множества протоколов прикладного уровня. Самый популярный из них — **HTTP**, который работает посредством серии запросов и ответов между клиентом и сервером.



Давайте разберемся в технологиях передачи информации в рамках «клиент-сервер» на уровне Application. Если представить модель OSI как общий механизм общения между компьютерами, то уровень Application в нем содержит правила и формат данных для передачи информации. Чтобы лучше понять, о каких правилах и форматах идет речь, добавим еще два условных уровня: Web Service и Message Formatting, которые отражают правила (протоколы) и форматы данных.



Уровень Web Service находится выше уровня Application и определяет протокол отправки и получения данных с помощью API. Один из популярных протоколов на этом уровне — **SOAP** (Simple Object Access Protocol). SOAP использует **XML** (eXtensible markup language) для передачи порций информации и функционирует с помощью HTTP или SMTP (Simple Mail Transfer Protocol).

Другая возможная опция на этом уровне — **REST** (Representational State Transfer). Сейчас REST считается лучшей практикой в API-разработке. Строго говоря, REST — это не протокол, а набор стилей и правил, которые используются при проектировании распределенных систем. При разработке веб-служб используется термин RESTful.

Форматы данных уровня Message Formatting определяют структуры данных, которые мы хотим передать или получить.

Есть несколько вариантов реализации, но более распространены два формата:

- **XML** (eXtensible markup language) — расширяемый язык разметки. Похож на HTML, его могут читать и люди, и компьютеры.
- **JSON** (JavaScript Object Notation) — объектная нотация JavaScript. Напоминает код JavaScript и представляет собой пары «атрибут-значение», из-за чего его тоже легко читать.

Итак, у нас есть 4 технологии реализации веб-интерфейсов API: архитектуры — SOAP и REST, форматы данных — JSON и XML. Давайте сравним их.

	SOAP	REST
Суть	Протокол	Архитектурный стиль
Состояние	Stateful, Stateless	Stateless
Формат	XML	XML, JSON, HTML, текст
Протокол передачи	HTTP, FTP, TCP, SMTP	HTTP
Скорость	Медленный	Быстрый
Кривая обучения	Легко	Сложно

SOAP и REST зарекомендовали себя как два надежных варианта реализации веб-сервисов. SOAP был разработан Microsoft в 1998 году.

В SOAP мы должны использовать XML, а в качестве протокола передачи данных можем выбрать HTTP или другой базовый протокол. REST — скорее описательный проект, чем проект разработки. Аббревиатура впервые появилась в работе Роя Филдинга в 2000 году.

REST использует HTTP для доступа к ресурсам и манипулирования ими, но может использовать любой тип протокола для структурирования передаваемых данных (XML, JSON и других). Чтобы приложение было действительно RESTful, оно должно придерживаться ряда ограничений, которые Филдинг определил в своей работе.

XML и JSON — два эффективных формата данных, поэтому сравним их.

XML vs. JSON

<pre><?xml version="1.0" encoding="UTF-8" ?> <dataset> <record> <id>1</id> <first_name>Kyle</first_name> <last_name>Danzey</last_name> <email>kdanzey0@dedecms.com</email> </record> <record> <id>2</id> <first_name>Stanly</first_name> <last_name>Chaise</last_name> <email>schaise1@php.net</email> </record> <record> <id>3</id> <first_name>Valentine</first_name> <last_name>Vasler</last_name> <email>vvasler2@ifeng.com</email> </record> <record> <id>4</id> <first_name>Herve</first_name> <last_name>Tollet</last_name></pre>	<pre>{ "dataset": { "record": [{ "id": "1", "first_name": "Kyle", "last_name": "Danzey", "email": "kdanzey0@dedecms.com" }, { "id": "2", "first_name": "Stanly", "last_name": "Chaise", "email": "schaise1@php.net" }, { "id": "3", "first_name": "Valentine", "last_name": "Vasler", "email": "vvasler2@ifeng.com" }, { "id": "4", "first_name": "Herve", "last_name": "Tollet", "email": "htollet3@chronoengine.com"</pre>
---	---

Источник: johnmichaelross.com

	XML	JSON
Читабельность	Сложнее (язык разметки)	Очень легко
Компактность кода	Больше кода	Меньше кода
Скорость парсинга	Медленнее	Быстрее
Простота синтаксиса	Требует знания тегов	Легко

Гибкость	У данных нет типа	Работает с ограниченным количеством типов данных
Поддержка массивов	Нет	Да

XML разработан в 1997 году. Он использует язык разметки с помощью тегов, как в HTML, и обеспечивает жесткий способ структурирования данных.

JSON разработан в 2001 году и является производным от JavaScript. Как и HTML, его легко прочитать человеку, однако JSON имеет более краткий формат за счет меньшего количества символов, что делает его очень легким.

Какой формат лучше? Поскольку API созданы, чтобы облегчить жизнь разработчика, мы должны использовать формат данных, более удобный для конечного пользователя. Когда-то XML был самым популярным форматом обмена данными, но JSON быстро набрал обороты и сейчас считается более распространенным форматом обмена данными в API.

Опираясь на популярность JavaScript, JSON легко приняли в сообществе веб-разработчиков как наиболее распространенный способ структурирования данных, передаваемых через RESTful API.

Почему разработчики предпочитают REST, а не SOAP? Есть ряд причин:

- **Простота.** REST легко понять и реализовать. Он использует стандартные методы HTTP (например, GET, POST, PUT и DELETE) для выполнения операций и возвращает данные в формате, который легко парсить, например, JSON или XML.
- **Гибкость.** REST позволяет разработчикам использовать любой формат для передачи данных: например, JSON, XML или обычный текст. Это делает REST более гибким, чем SOAP, который требует использовать XML.
- **Производительность.** REST быстрее и эффективнее SOAP, поскольку использует меньше ресурсов и требует меньше вычислительной мощности. Подходит для приложений, требующих высокой производительности и масштабируемости.
- **Интероперабельность.** REST основан на стандарте HTTP, который широко поддерживается веб-серверами и прокси-серверами. Это делает REST более простым в реализации и совместимым с широким спектром систем, по сравнению с SOAP, который требует специфической поддержки протокола SOAP.

- **Скорость разработки.** REST проще и быстрее разрабатывать, чем SOAP, поскольку у него простая структура и нужно меньше шаблонного кода. REST подходит для приложений, которые нужно разработать быстро, например, прототипов или пробных концепций.

Representational State Transfer (REST)

Подведем небольшой итог из того, что мы узнали про REST.

REST — это архитектурный стиль для создания веб-служб, которые должны быть масштабируемыми, гибкими и простыми в использовании. REST предназначен для обеспечения простого и последовательного способа взаимодействия с ресурсами в интернете и основан на принципах протокола HTTP.

Для практического применения REST веб-сервис разрабатывается таким образом, чтобы предоставлять ресурсы, такие как товары, через набор конечных точек API. Клиенты могут взаимодействовать с этими ресурсами, посылая HTTP-запросы в соответствующие конечные точки и получая HTTP-ответы. Методы HTTP (например, GET, POST, PUT и DELETE) используются для указания типа операции, которую запрашивает клиент: например, получение данных или создание нового ресурса.

REST помогает в сборе данных, предоставляя способ доступа и взаимодействия с ресурсами в интернете. Например, клиент может использовать метод GET, чтобы получить список книг из конечной точки API, а затем метод POST, чтобы добавить новую книгу в список. Это позволяет легко и эффективно собирать данные из веб-сервисов и управлять ими.

REST — скорее набор рекомендаций, чем протокол, поскольку в его основе лежит протокол HTTP, к которому применяется ряд ограничений для работы с коммуникацией и управлением ресурсами.

Разберем эти ограничения, чтобы лучше понять особенности REST.

Разделение клиента и сервера. REST требует четкого разделения задач между клиентом и сервером. Клиент отвечает за выполнение запросов и отображение данных, а сервер — за хранение и управление данными. Такое разделение задач позволяет клиенту и серверу развиваться независимо друг от друга и уменьшает связь между ними.

Концепция Stateless. Мы уже обсуждали, что Stateful, в отличие от Stateless, запоминает действия клиента между запросами.

Когда вы пользуетесь интернет-магазином, где есть корзина покупок или сессия входа в систему, может казаться, что сервер запоминает вашу активность в течение всего времени пребывания на сайте. Однако RESTful-архитектура не позволяет сохранять информацию о состоянии другой машины во время процесса коммуникации.

Каждый запрос от клиента к серверу должен рассматриваться так, будто это первый запрос, который сервер когда-либо видел от этого клиента. Сервер не должен запоминать клиентов и соответствующим образом корректировать свое состояние. Сервер может передавать клиенту актуальную информацию о своем состоянии и позволять вносить изменения, только если он уполномочен на это.

Cacheable. REST требует, чтобы ресурсы были кэшируемыми, то есть могли храниться в кэше для будущего использования. Это позволяет клиентам сократить количество запросов к серверу и повысить производительность.

Например, у нас есть RESTful API для интернет-магазина, который позволяет получить список книг. Если API разработан с возможностью кэширования, то клиент может сохранить список книг в кэше после первого запроса. В следующий раз, когда клиенту понадобится список книг, он сможет получить его из кэша, а не делать еще один запрос к серверу. Это может значительно повысить производительность и скорость реакции клиента, а также снизить нагрузку на сервер.

Единый интерфейс архитектуры RESTful должен быть между всеми клиентами и серверами. Например, сервер не должен требовать разных способов доступа к данным, если у клиента ноутбук с Windows, iPhone или Unix-сервер. Получение доступа к этим конечным точкам одинаково для любой машины.

Многослойная система — клиент может получить доступ к конечной точке, которая опирается на другие конечные точки, без необходимости разбираться во всех базовых реализациях. Если клиент А хочет связаться с сервером В, а тот обращается к Google или другой базе данных, чтобы получить ответ, клиенту А не нужно использовать другие технологии, кроме доступа к точке сервера В. Многослойность позволяет выполнять сложные задачи без необходимости понимать все базовые сложности, которые требуются для ответа.

Код по требованию — необязательное ограничение для RESTful-приложений. Означает, что сервер может отправлять исполняемый код клиенту для выполнения. Позволяет повысить гибкость и функциональность клиента.

Дополнительная информация об ограничениях RESTful: [Fielding Dissertation: CHAPTER 5: Representational State Transfer \(REST\)](#).

Клиент-серверный поток HTTP

Фундаментальная часть понимания RESTful API — это понимание компонентов, составляющих клиент-серверный поток HTTP.

HTTP можно описать как pull-протокол, то есть первоначальный запрос данных производится клиентом, а ответ порождается сервером. Эти сообщения представляют собой текст, который компьютер впоследствии может интерпретировать в действия или, например, в мультимедийный контент.

Кратко обсудим некоторые из основных компонентов HTTP-запросов и ответов.

Каждое HTTP-сообщение состоит из строки запроса (Request line), заголовка сообщения (Header) и необязательного тела сообщения (Body). Последние два элемента разделяются, как показано на рисунке.

```
1 POST /cgi-bin/process.cgi HTTP/1.1      ← Request Line
2 User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
3 Host: www.tutorialspoint.com
4 Content-Type: application/x-www-form-urlencoded
5 Content-Length: length
6 Accept-Language: en-us
7 Accept-Encoding: gzip, deflate
8 Connection: Keep-Alive
9
10 licenseID=string&content=string&/paramsXML=string ← Body
```

Request Header

Empty Line

Источник: tutorialspoint.com

Первая строка заголовка HTTP-запроса — строка запроса (request line) — содержит:

- метод
- URI (Uniform Resource Identifier) — унифицированный идентификатор ресурса,
- номер версии HTTP.

После строки запроса есть необязательные заголовки запроса (optional request header). Это параметры для описания определенных свойств запроса. Заголовки запросов представлены в виде пар «имя-значение». Несколько значений могут быть разделены запятыми.

Пустая строка разделяет заголовок и тело HTTP-запроса. В теле запроса мы можем добавить любую другую информацию о запросе, которую хотим передать серверу.

Разберемся с методами HTTP. Они определяют тип операции, которую клиент запрашивает у конечной точки API. Четыре распространенных метода HTTP — GET, POST, PUT и DELETE.

- **GET** — метод для получения данных из конечной точки API. Самый распространенный метод HTTP. Позволяет получить информацию о ресурсе или наборе ресурсов, например, о списке книг или о конкретной книге.
- **POST** — метод для создания нового ресурса в API. Используется для отправки данных в конечную точку API, например, для создания новой книги или добавления рецензии.
- **PUT** — метод для обновления существующего ресурса в API. Используется для отправки данных в конечную точку API, например, для обновления информации о конкретной книге.
- **DELETE** — метод для удаления ресурса в API. Используется для запроса на удаление определенного ресурса из API, например, для удаления книги.

Когда сервер получает HTTP-запрос, он возвращает клиенту ответное сообщение. Этот ответ может содержать запрошенную информацию или сообщение об ошибке, если она была.

Как и HTTP-запрос, ответ состоит из заголовка сообщения и необязательного тела. Первая строка заголовка называется строкой состояния (Status line). За ней следуют необязательные заголовки ответа (optional response headers).

HTTP/1.1 200 OK	Status Line	HTTP Response
Date: Thu, 20 May 2004 21:12:58 GMT	General Headers	
Connection: close		
Server: Apache/1.3.27	Response Headers	
Accept-Ranges: bytes		
Content-Type: text/html	Entity Headers	
Content-Length: 170		
Last-Modified: Tue, 18 May 2004 10:14:49 GMT		
<pre><html> <head> <title>Welcome to the Amazing Site!</title> </head> <body> <p>This site is under construction. Please come back later. Sorry!</p> </body> </html></pre>		

Источник: blogs.sap.com

Строка состояния содержит **версию HTTP, код состояния** (Status Code) и **фразу причины** (Reason Phrase), которая объясняет код состояния на английском языке.

Распространенные коды состояния и фразы причины.

- 200 OK (все хорошо)
- 404 Not Found (не найдено)
- 403 Forbidden (запрещено)
- 500 Internal Server Error (внутренняя ошибка сервера)

Необязательные заголовки ответа имеют форму пар «имя-значение», как и аналогичные заголовки запроса.

Тело сообщения ответа содержит данные, которые запросил клиент, например веб-страницу.

Когда вы ищете информацию в интернете, ваш браузер заботится о создании HTTP-запросов и отображении ответов на экране. Однако дата-инженеру может понадобиться создавать и анализировать HTTP-запросы и ответы при сборе данных из API по нескольким причинам:

- **Валидация данных.** Создавая и анализируя HTTP-запросы и ответы, дата-инженер может проверить данные, возвращаемые API, чтобы убедиться в их точности и полноте.
- **Преобразование данных.** Некоторые API могут возвращать данные в формате, который не подходит для анализа или хранения данных. Создавая и анализируя HTTP-запросы и ответы, дата-инженер может определить нужные преобразования и применить их к собранным данным.
- **Оптимизация производительности.** Анализируя HTTP-запросы и ответы, дата-инженер может выявить узкие места или проблемы с производительностью и внести изменения для оптимизации процесса сбора данных.

Например, дата-инженер, собирающий данные через API, может столкнуться с высокой задержкой, которая приводит к медленному времени отклика. Анализируя HTTP-запросы и ответы, дата-инженер может определить, что API расположен далеко от клиента и причина высокой задержки в этом. Для решения проблемы дата-инженер может рассмотреть возможность использовать сеть доставки контента (CDN), расположенную ближе к клиенту. Это уменьшит задержку и повысит производительность.

- **Обработка ошибок.** При сборе данных из API могут возникать ошибки, например, тайм-ауты или недостоверные данные. Создавая и анализируя HTTP-запросы и ответы, дата-инженер может выявить ошибки и внедрить соответствующие стратегии их обработки, чтобы обеспечить надежность процесса сбора данных.

Создание HTTP-запросов в Postman

Браузер делает отличную скрытую работу по обработке HTTP, чтобы пользователь мог сосредоточиться на веб-серфинге. Но нам, как разработчикам, нужно уметь копать глубже, чтобы увидеть, что происходит с запросами и ответами.

К счастью, есть несколько инструментов для генерации HTTP-запросов и просмотра ответов:

- **Curl** — один из самых популярных инструментов командной строки для отправки и получения HTTP-сообщений, хотя он может передавать и несколько других протоколов.
- **Postman** — простое расширение Chrome, которое можно использовать для создания и просмотра HTTP-сообщений.



Документация

cURL:

- [curl.1 the man page](#)
- [Getting started with cURL](#)

Postman:

- [Overview | Postman Learning Center](#)

Query_string:

- [Query string - Wikipedia](#)

Разберемся на примере. Компания продает книги через интернет. Чтобы наполнить интернет-магазин информацией, нужно использовать данные из базы с помощью API. Дата-инженер может использовать интерфейс Postman, чтобы создать и отправить GET-запросы и получить данные о книгах, авторах и издателях.

Кроме того, дата-инженер анализирует производительность API, например, время отклика и использование ресурсов. С помощью такого анализа можно выявить узкие места или проблемы в производительности и внести изменения для оптимизации процесса сбора данных.

Пример службы книжных баз данных с бесплатным API — API [Open Library](#). Open Library — это онлайн-каталог литературы. API Open Library позволяет разработчикам выполнять поиск в библиотечном каталоге, получать информацию о книгах, а в случае общественного доступа и их полный текст.

Прежде чем начать работать с API, нужно познакомиться с его документацией. В нашем случае документация находится на странице [Developer Center / APIs | Open Library](#).

Чтобы использовать API, часто нужно зарегистрироваться на сервисе. Использование API может быть платным.

Как правило, после создания учетной записи сервис предоставляет ключи или ID (Key или Client ID). Обычно API-сервисы предоставляют Client ID и Client Secret.

- **Client ID** — общедоступный идентификатор приложений. Хотя он и общедоступный, лучше скрывать его от третьих лиц. Поэтому во многих реализациях используется что-то вроде 32-символьной шестнадцатеричной строки.
- **Client Secret** — ключ, известный только приложению и серверу авторизации. Это, по сути, пароль приложения. Он должен быть случайным, чтобы его нельзя было угадать.

Давайте воспользуемся API Open Library и выполним запрос по ISBN (Международному стандартному книжному номеру).

Из документации выясняем, что конечная точка API — openlibrary.org/api/books. Вы можете использовать ее, чтобы получить информацию о конкретных книгах по ISBN.

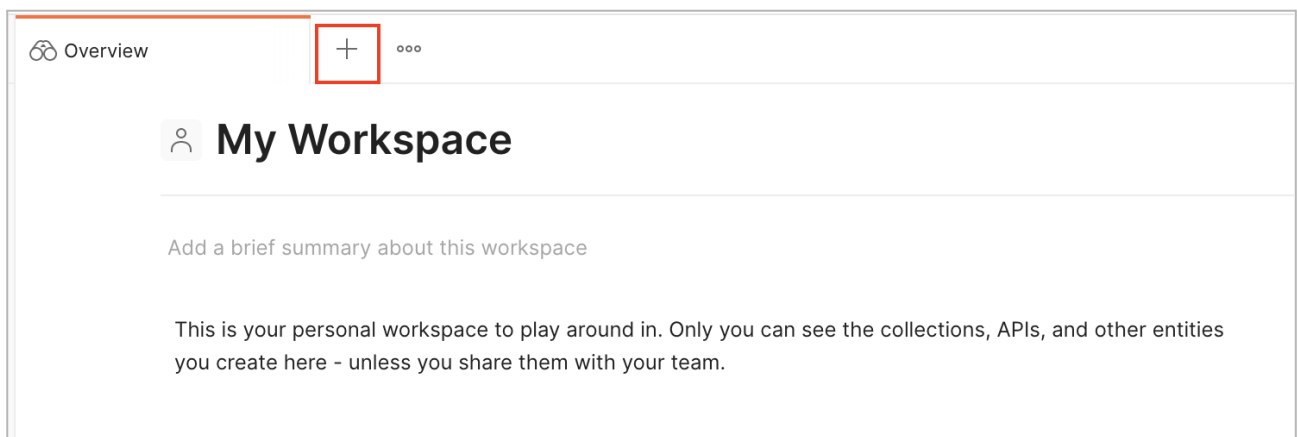
Также из документации выясняем, что API поддерживает следующие параметры запроса:

- **bibkeys** — список идентификаторов для запроса информации. API поддерживает ISBN, LCCN, номера OCLC и OLID (идентификаторы открытых библиотек).

- **format** — необязательный параметр, определяющий формат ответа. Возможные значения: JSON и JavaScript. По умолчанию — JavaScript.
- **callback** — необязательный параметр, задающий имя функции JavaScript для вызова с результатом. Учитывается только в том случае, если формат — JavaScript.
- **jscmd** — необязательный параметр, определяющий, какую информацию предоставлять для каждого совпадающего bibkey. Возможные значения: viewapi и data. По умолчанию — viewapi.

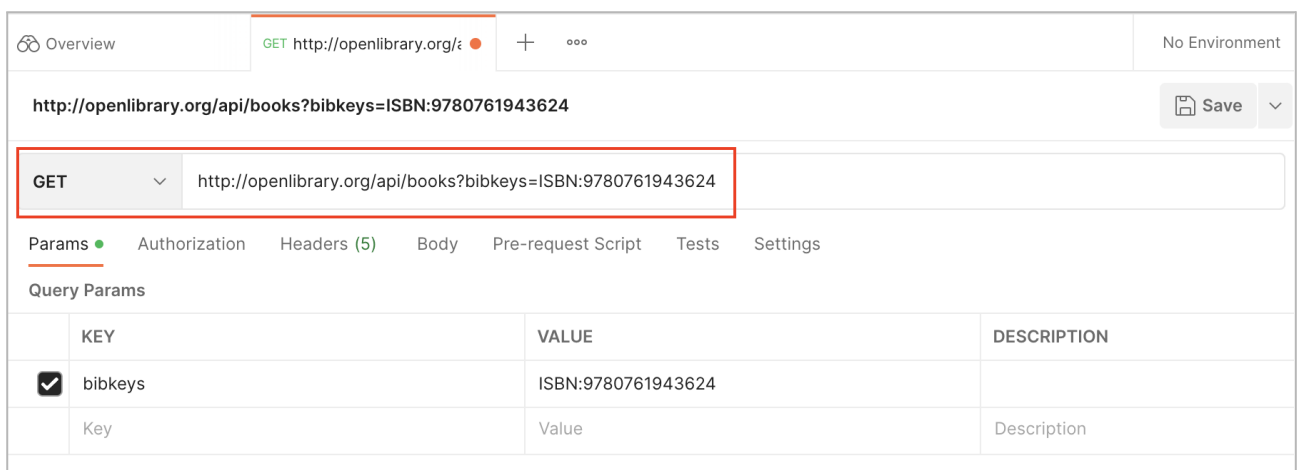
Перейдем в Postman и попробуем использовать API Open Library. Вот пример GET-запроса, который вы можете отправить API, чтобы получить информацию о книге по ISBN.

1. Создадим новый HTTP-запрос в Postman.

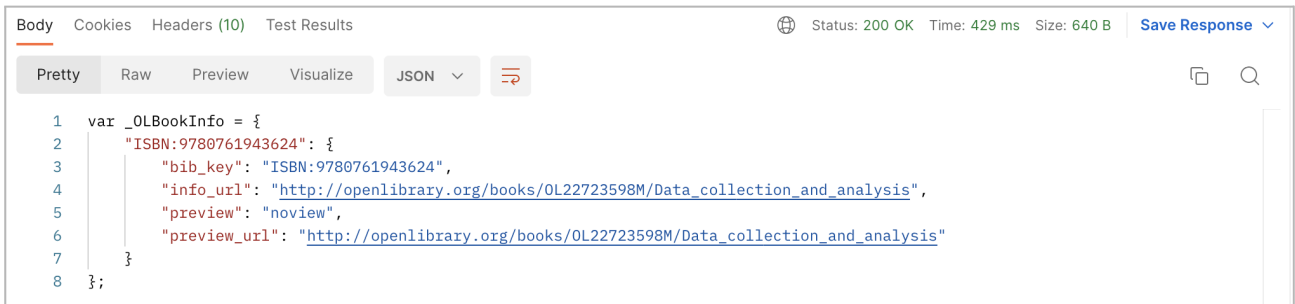


2. В строке запроса метод GET стоит по умолчанию. Прописываем строку запроса:

<http://openlibrary.org/api/books?bibkeys=ISBN:9780761943624>



3. Жмем Send. В случае успеха получаем ответ от сервера.



```
1 var _OLBookInfo = {
2   "ISBN:9780761943624": {
3     "bib_key": "ISBN:9780761943624",
4     "info_url": "http://openlibrary.org/books/OL22723598M/Data_collection_and_analysis",
5     "preview": "noview",
6     "preview_url": "http://openlibrary.org/books/OL22723598M/Data_collection_and_analysis"
7   }
8 };
```

В примере API возвращает информацию о книге «Data Collection and Analysis» Роджера Сепсфорда. Мы можем изменить запрос, например добавив в HTTP-запрос формат ответа JSON.

<http://openlibrary.org/api/books?bibkeys=ISBN:9780761943624&format=json>

Давайте проанализируем ответ сервера.

HTTP-запрос, который мы отправили в Open Library API, был успешным. Это мы поняли по ответу — Status: 200 OK. То есть сервер смог обработать запрос и вернуть запрошенные данные.

Время ответа относительно быстрое — 429 миллисекунд. Это хороший признак. Размер ответа небольшой — 640 байт. Он говорит о том, что объем возвращаемых данных минимальный.

Данные ответа представляют собой объект JavaScript, содержащий информацию о книге. В ответ включена следующая информация:

- "bib_key" — ISBN книги (ключ в запросе API).
- "info_url" — URL-адрес, предоставляющий дополнительную информацию о книге на сайте Open Library.
- "preview" — значение "noview" указывает, что для книги нет предварительного просмотра.
- "preview_url" — URL-адрес, аналогичный "info_url".

Таким образом, HTTP-запрос показывает, что API Open Library работает правильно и возвращенные данные относятся к запрошенной нами книге.

Теперь, когда мы изучили основы использования Postman для выполнения HTTP-запросов и анализа ответов, обратимся к другому важному инструменту для работы с API — Python. В следующем разделе мы рассмотрим, как делать HTTP-запросы и получать данные с помощью API в Python.

Создание HTTP-запросов в Python

Начнем с изучения отправки HTTP-запросов с помощью Python requests — популярной библиотеки для работы с HTTP в Python. Наша задача — научиться отправлять GET-запросы, а также анализировать и манипулировать данными ответа. Мы также познакомимся с некоторыми ключевыми параметрами и опциями, доступных при выполнении HTTP-запросов, и с тем, как работать с разными форматами ответов и обработкой ошибок.

Итак, библиотека requests в Python предоставляет несколько методов для отправки HTTP-запросов:

- **requests.get** — для отправки GET-запроса к конечной точке API.
- **requests.post** — для отправки данных в API, например, для создания нового ресурса.
- **requests.put** — для обновления существующего ресурса на сервере.
- **requests.delete** — для удаления ресурса с сервера.
- **requests.head** похож на запрос GET, но возвращает только заголовки ответа, а не фактические данные ответа.
- **requests.options** — для получения информации об опциях коммуникации, доступных для ресурса.

Каждый из этих методов принимает URL в качестве аргумента. Вы можете указать дополнительные параметры для запроса, если нужно.

Перейдем к практическому примеру — соберем данные через API Open Library.

API Open Library дает возможность получать данные о книгах, указывая тему или предмет. Например, мы можем получить данные о книгах по Artificial intelligence. В первую очередь нужно прочитать документацию API: [Subjects API | Open Library](#).

Перейдем в VS Code, создадим новый Python-файл и приступим к написанию кода.

1. Импортируем нужные библиотеки. Open Library возвращает данные в формате JSON.

```
import requests
import json
```

2. Определим конечную точку API и параметры для запроса API.

```
url = "http://openlibrary.org/search.json"
```

3. Определим параметры для запроса API в виде словаря params. Параметры включают тему для поиска, которая задана как «Artificial intelligence», и количество получаемых результатов (то есть количество книг), равное 10.

```
subject = "Artificial intelligence"
params = {
    "subject": subject,
    "limit": 10
}
```

4. Отправляем запрос API, используя метод requests.get.

```
response = requests.get(url, params=params)
```

5. Проверим код состояния ответа, чтобы убедиться, что запрос API успешный. Если код 200, код печатает «Успешный запрос API!». Иначе выводит сообщение об ошибке с указанием кода статуса.

```
if response.status_code == 200:
    print("Успешный запрос API!")
else:
    print("Запрос API отклонен с кодом состояния:",
          response.status_code)
```

6. Переходим к парсингу данных JSON. Response.text в коде обозначает текстовое содержимое HTTP-ответа, возвращаемого API. Этот текст представляет собой данные, возвращаемые API, в формате JSON.

```
data = json.loads(response.text)
```

Строка `data = json.loads(response.text)` использует функцию `json.loads` из библиотеки `json` для парсинга строки JSON, содержащейся в `response.text`, в словарь Python. Таким образом, мы осуществляем парсинг JSON в словарь. Это нужно, чтобы в дальнейшем было легко манипулировать данными, возвращаемыми API, вместо того, чтобы работать с необработанной строкой JSON. С полученным словарем можно обращаться как с любым другим словарем Python с ключами и значениями.

7. Код получает доступ к списку книг из данных ответа с помощью ключа "docs" и сохраняет его в переменной books.

```
books = data["docs"]
```

8. Код использует цикл for для итерации по списку книг и печати названия, автора и темы для каждой книги.

```
for book in books:
    print("Title:", book["title"])
    print("Author:", book["author_name"])
    print("Subject:", book["subject"])
    print("\n")
```

Создание датафрейма в Jupyter-ноутбуке

Теперь переместимся в Google Colab и попробуем сделать парсинг API платформы Binance. Допустим, перед нами стоит задача получить исторические котировки биткоина для создания модели машинного обучения.

Сначала нужно получить ключи API, а для этого — зарегистрироваться на платформе. После изучаем документацию по работе с API: [Change Log – Binance API Documentation](#).

Перемещаемся в Colab. Для начала импортируем нужные библиотеки:

```
import requests
import json
import pandas as pd
import datetime as dt
```

Мы инициализируем переменные, содержащие символ криптовалюты, для которого хотим получить данные — 'BTCBUSD', интервал — например, 1 час, начальная и конечная даты — 01.01.2021 и 31.12.2021 (даты можно выбрать произвольно).

```
url = 'https://api.binance.com/api/v3/klines'
symbol = 'BTCBUSD'
interval = '1h'
start = str(int(dt.datetime(2021,1,1).timestamp()*1000))
end = str(int(dt.datetime(2021,12,31).timestamp()*1000))
```

Запрос должен содержать необходимые нам параметры в формате словаря:

```
params = {'symbol': symbol, 'interval': interval, 'startTime':
start, 'endTime': end}
```

Наконец, отправляем запрос.

```
response = requests.get(url, params = params).text
```

Если посмотрим на результат, увидим, что переменная `response` содержит ответ сервера в текстовом формате. Понять в нем что-то сложно.

▶ response

```
↳ '[[1609459200000,"28961.73000000","29073.65000000","28731.25000000","29040
0","7388654.98258062","0"],[1609462800000,"29042.43000000","29526.00000000
39",26287,"645.47359000","18944209.67915610","0"],[1609466400000,"29459.53
99999","14784158.50198296",15999,"219.84238100","6448916.64318568","0"],[16
31.50194500",1609473599999,"9717157.51702325",11169,"171.81452000","503650
0","29272.06000000","448.90767500",1609477199999,"13156430.47067174",14438
00000","29140.35000000","29236.94000000","241.47007300",1609480799999,"...
```

Поэтому трансформируем ответ сервера в словарь и посмотрим результат.

```
json_res = json.loads(response)
```

▶ json_res

```
↳ [1611241200000,
'31146.79000000',
'32022.58000000',
'31050.00000000',
'31350.00000000',
'2198.03572600',
1611244799999,
'69286809.64278765',
55968,
'1103.36075600',
'34781238.01885618',
'0'],
```

В выдаче уже видна структура. Обратившись к документации, можно выяснить значение каждого параметра. Но напомним, что мы хотели получить датафрейм.

```
df = pd.DataFrame(json_res)
df.columns = ['datetime', 'open', 'high', 'low', 'close',
'volume', 'close_time', 'qav', 'num_trades', 'taker_base_vol',
```

```
'taker_quote_vol', 'ignore']
```

▶ df.head()

	datetime	open	high	low	close
0	1609459200000	28961.73000000	29073.65000000	28731.25000000	29040.44000000
1	1609462800000	29042.43000000	29526.00000000	29006.41000000	29458.58000000
2	1609466400000	29459.53000000	29511.32000000	29175.01000000	29245.40000000
3	1609470000000	29247.55000000	29400.00000000	29201.65000000	29331.51000000
4	1609473600000	29331.51000000	29441.45000000	29110.54000000	29272.06000000

Все еще не очень читабельно. В частности, время отображается в UNIX-формате, а тип котировок — object, хотя нужно представить их в числовом формате. Так или иначе у нас уже есть датафрейм, с которым мы можем выполнять любые доступные для этого формата данных действия.

```
df.index = [dt.datetime.fromtimestamp(x/1000.0) for x in
df.datetime]
df=df.astype(float)
```

▶ df.head()

	datetime	open	high	low	close	volume
2021-01-01 00:00:00	1.609459e+12	28961.73	29073.65	28731.25	29040.44	502.956408
2021-01-01 01:00:00	1.609463e+12	29042.43	29526.00	29006.41	29458.58	1115.361310
2021-01-01 02:00:00	1.609466e+12	29459.53	29511.32	29175.01	29245.40	503.948024
2021-01-01 03:00:00	1.609470e+12	29247.55	29400.00	29201.65	29331.51	331.501945
2021-01-01 04:00:00	1.609474e+12	29331.51	29441.45	29110.54	29272.06	448.907675

Заключение

Концепции и методы, которые мы сегодня изучили, формируют основу для работы с API и для сбора данных из интернета. Поняв, как работает взаимодействие «клиент-сервер» и как использовать API для доступа к данным, вы подготовитесь к изучению огромного мира онлайн-источников данных.

Главные тезисы лекции:

1. Понимание взаимодействия «клиент-сервер» и анализа API имеет важное значение для эффективного сбора, управления и обмена данными.
2. Цикл «запрос-ответ» — основа взаимодействия «клиент-сервер» и ключевое понятие при работе с API.
3. REST — популярный архитектурный стиль для создания API. Его преимущества — простота и масштабируемость. Разработчику, работающему с API, важно понимать преимущества и ограничения, которые накладывает REST.
4. Модель Open System Interconnection (OSI) дает полезную основу для понимания разных уровней связи, участвующих во взаимодействии «клиент-сервер».
5. Использование таких инструментов, как Postman, может значительно помочь дата-инженерам в создании и анализе HTTP-запросов и ответов, повысить производительность и эффективность сбора и обмена данными.
6. Выбор между JSON и XML в качестве формата ответа часто сводится к требованиям проекта. Но важно понимать сильные и слабые стороны каждого формата.
7. Использование Python и его популярных библиотек, таких как requests, значительно упрощает процесс выполнения HTTP-запросов и работы с API.

До встречи на следующей лекции!

Что можно почитать еще?

1. [Дизайн пагинации страниц в API](#)
2. [Использование разбиения на страницы в REST API](#)
3. [Функция loads\(\) модуля json в Python](#)