



Web service

API

REST

API

Application Programming Interface



Это программный интерфейс, который позволяет двум приложениям взаимодействовать друг с другом без какого-либо вмешательства пользователя.

API предоставляют продукт или услугу для связи с другими продуктами и услугами, не зная, как они реализованы.



Аббревиатура **API** расшифровывается как «**Application Programming Interface**» (интерфейс программирования приложений, программный интерфейс приложения). Чтобы понять, как и каким образом API применяется в разработке и бизнесе, сначала нужно разобраться, как устроена «всемирная паутина».

Всемирная паутина и удалённые серверы

WWW можно представить как огромную сеть связанных серверов, на которых и хранится каждая страница. Обычный ноутбук можно превратить в сервер, способный обслуживать целый сайт в сети, а локальные серверы разработчики используют для создания сайтов перед тем, как открыть их для широкого круга пользователей.

При введении в адресную строку браузера www.facebook.com на удалённый сервер Facebook отправляется соответствующий запрос. Как только браузер получает ответ, то интерпретирует код и отображает страницу.

Каждый раз, когда пользователь посещает какую-либо страницу в сети, он взаимодействует с API удалённого сервера. API — это составляющая часть сервера, которая получает запросы и отправляет ответы.

API содержит в себе некие «мостики», позволяющие программе А получить доступ к данным из программы Б или к некоторым ее возможностям. Таким образом, программисты могут расширять функциональность своего продукта и связывать его с чужими разработками.

Все это с разрешения создателей программы А и с соблюдением всех мер безопасности, чтобы разработчики, желающие использовать API, не смогли получить доступ к конфиденциальной информации.

Главный принцип работы API.

Почему его называют интерфейсом



Инкапсуляция сокрытие части функций ради упрощения работы в целом и минимизации участков программного обеспечения, где один из разработчиков мог бы допустить ошибку.

Главный принцип работы API. Почему его называют интерфейсом

Простыми словами, интерфейс – это «прослойка» между приложением А и приложением Б. В ней происходят процессы, которые позволяют двум программам обмениваться информацией и выполнять функции, связанные с обеими сторонами, скрывая «внутреннее строение» программ. Знакомо? Только что таким же образом мы описали API.

Такой подход позволяет наладить взаимодействие между несколькими утилитами, не задумываясь о том, как они устроены, какая программная логика ими движет и каким образом обрабатываются передаваемые данные. Интерфейсы упрощают работу как для простых пользователей, так и для программистов. Первым не приходится задумываться о том, что стоит за привычными функциями в их гаджетах, а разработчикам не нужно изучать код других программистов, чтобы подключить чужой продукт к своему. Это называется инкапсуляцией. Сокрытием части функций ради упрощения работы в целом и минимизации участков программного обеспечения, где один из разработчиков мог бы допустить ошибку.

Зачем нужен API?

Почему разработчики используют API?



Зачем нужен API?

Теперь нам знакомы принципы работы API и задачи, которые они помогают решить. Но они хороши не только этим. Программные интерфейсы используются еще по двум немаловажным причинам.

Во-первых, такой подход позволяет делать программы надежнее. Инкапсуляция в целом заметно упрощает жизнь разработчиков. Отдельные компоненты приложений становятся абстракциями. Создателям нового ПО не приходится лезть в логику низкоуровневых функций и разбираться в их реализации. Так заметно повышается безопасность выполняемых задач, что особенно заметно на уровне таких масштабных программных продуктов, как операционные системы. Программы постоянно выполняют сотни внутренних задач, при этом они проходят незаметно для пользователя и не могут навредить друг другу.

Во-вторых, на API можно заработать. Например, сервисы, предоставляющие информацию с метеовышек, берут плату за каждый запрос актуальной погоды, если их API используется в сторонних приложениях. Аналогичные условия могут предлагать и другие компании, предоставляющие услуги. Будь то навигация, конвертация файлов в другие форматы и прочие возможности, реализуемые через API.

Почему разработчики используют API?

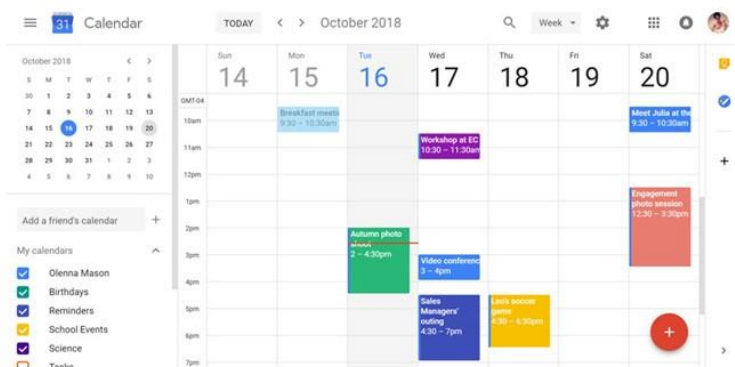
Есть как минимум еще 4 причины, объясняющие интерес программистов к API:

1. API упрощает и ускоряет создание новых продуктов. Разработчикам не приходится каждый раз изобретать велосипед. Можно взять API нейронной сети TensorFlow, к

примеру, и внедрить в свое программное обеспечение, а не создавать собственную систему машинного обучения.

2. программный интерфейс увеличивает безопасность разработки. С помощью него можно вынести ряд функций в отдельное приложение, сделав невозможным их некорректное использование. От человеческого фактора это тоже спасает.
3. API упрощает настройку связей между разными сервисами и программами. Интерфейс нивелирует необходимость в тесном сотрудничестве создателей различных приложений. Разработчики могут внедрять поддержку сторонних сервисов, вообще не контактируя с их создателями.
4. Наличие готовых интерфейсов позволяет экономить не только время и силы программистов, но и финансы, с которыми часто связано создание новых программных решений.

Google Календарь



Погодное приложение



Примеры API

Работа API представляет собой передачу данных по определенному запросу со стороны клиента или другого приложения. Допустим, нужно выудить информацию с существующего сайта и передать ее в программу.

В браузере будет дан запрос и ожидается ответ в виде HTML-страницы. Если же используется API в стороннем приложении, то ему может быть достаточно фрагмента данных в формате JSON. Более точное техническое описание работы любого из существующих API доступно только их создателям.

На стороне пользователя такая реализация интерфейса будет выглядеть как банальная возможность выполнить действие, связанное с программой А в программе Б. То есть убрать лишний переход в стороннюю программу.

Google Календарь

Те, кто использовал приложения-календари для iOS или Android, знают, что данные в них можно синхронизировать, подключив один из популярных сервисов: Apple iCal или Google Calendar. Обе компании предлагают разработчикам API, позволяющие подключить свой календарь напрямую к сторонним приложениям. Благодаря подобной интеграции люди могут использовать несколько разных программ со схожей функциональностью и иметь на руках актуальную информацию о всех своих делах.

API позволяют создавать новые события и напоминания, удалять уже существующие, редактировать их и т.п.

Погодное приложение

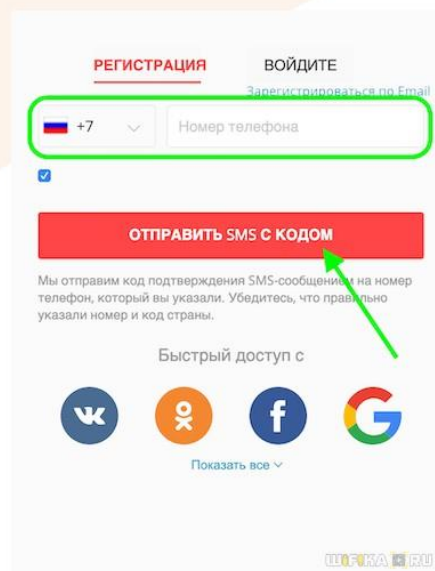
Существующие погодные приложения (встроенные в операционную систему или сторонние из App Store или Google Play) получают информацию о погоде из сторонних источников.

Есть сервисы, взаимодействующие напрямую с метеостанциями и обладающие информацией о текущей погоде. Разработчики приложений для мобильных устройств эту информацию покупают.

А чтобы весь процесс упростить, сервисы, сотрудничающие с метеостанциями, разработали соответствующие API. В них содержится набор функций, помогающий делать запросы о погоде в конкретных местах. Эти запросы через посредника (приложение) отправляются на «метеостанцию», а их результат возвращается пользователю тем же путем.



Сервис по заказу авиабилетов



Кнопки авторизации

Здесь аналогичная ситуация. Помимо сайтов и приложений, принадлежащих авиакомпаниям, есть так называемые агрегаторы. У нас популярен Aviasales, но есть и другие.

Такие сервисы собирают информацию о стоимости авиабилетов в разных авиакомпаниях и отображают ее в едином окне. Чтобы добыть эту информацию, разработчики используют функции сервисов авиакомпаний, которые помогают в реальном времени обновлять информацию о направлениях и стоимости билетов.

Кнопки авторизации

Наверняка вы видели на различных сайтах кнопки, позволяющие зарегистрироваться с помощью уже существующих аккаунтов на популярных площадках. Сейчас такие есть у Google, Facebook, Apple, Twitter, ВКонтакте и т.д. Набор доступных опций на конкретном ресурсе полностью зависит от его хозяев. Это тоже делается через API. Условная Apple создала набор защищенных функций, который можно с минимальными затратами подключить к своему проекту и предоставить пользователям доступ к удобному и безопасному способу авторизации.

При этом жизнь пользователей становится проще, а у владельцев ресурса остается доступ к почтовым адресам и другим персональным данным для взаимодействия с вновь зарегистрировавшимся человеком.

Навигация на сайтах и в приложениях

Тут почти как с погодой. Есть несколько крупных корпораций, предлагающих картографические данные. Те же Apple, Google, Yandex и парочка других. Некоторые из этих компаний разработали API, позволяющие подключить собственный картографический сервис к другим площадкам. Иногда они используются во внутренних продуктах. Яндекс.Транспорт построен на базе Яндекс.Карт, к примеру. Иногда API используются крупными партнерами. Uber использует для навигации сервис компании Google.

Web-сервис (служба) – программа, которая организует взаимодействие между сайтами. Информация с одного портала передается на другой.



По сути, веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети.

Именно с появлением веб-сервисов развилась идея SOA — сервис-ориентированной архитектуры веб-приложений (Service Oriented Architecture).

Информация в интернете разнородна. Сайты управляются разными системами. используются разные протоколы передачи и шифрования. Веб-сервисы упрощают обмен информацией между разными площадками.

Можно определить 3 инстанции, которые взаимодействуют между собой: каталог, исполнитель и заказчик. После создания сервиса, исполнитель регистрирует его в каталоге, а там сервис находит заказчик.

Механизм обмена данными формируется в описании Web Services Description. Это спецификация, охватывающая форматы пересылки, типы контента, транспортные протоколы, которые применяются в процессе обмена сведениями между заказчиком и транспортировщиком услуг.

Задачи веб-сервисов



- Интеграция процессов идет сразу, без участия людей
- Если в компании используются корпоративные программы, то веб-сервис поможет настроить их совместную работу

Задачи веб-сервисов

Веб-сервисы могут использоваться во многих сферах.

B2B-транзакции **B2B** (от англ. business-to-business, «бизнес для бизнеса») — это продажи, в которых и заказчиками, и продавцами выступают юридические лица.

Интеграция процессов идет сразу, без участия людей. Например, пополнение каталога интернет-магазина новыми товарами. Их привозят на склад, и кладовщик отмечает в базе данных приход. Автоматически информация передается в интернет-магазин. И покупатель вместо пометки “Нет на складе” на карточке товара видит его количество.

Интеграция сервисов предприятий

Если в компании используются корпоративные программы, то веб-сервис поможет настроить их совместную работу.

Создание системы клиент-сервер

Сервисы используются, чтобы настроить работу клиента и сервера. Это дает преимущества:

- можно продавать не само программное обеспечение, а делать платным доступ к веб-сервису;
- легче решать проблемы с использованием стороннего ПО;

- проще организовывать доступ к контенту и материалам сервера.

Веб-сервис — это приложение, которое упрощает техническую настройку взаимодействия ресурсов.

Веб-сервис Web-API	API
Все веб-сервисы являются API.	Все API не являются веб-сервисами.
Он поддерживает XML.	Ответы форматируются с использованием MediaTypeFormatter Web API в XML, JSON или любой другой заданный формат.
Он может использоваться любым клиентом, который понимает XML.	Может использоваться клиентом, который понимает JSON или XML.
Веб-сервис использует три стиля: REST, SOAP и XML-RPC для связи.	API можно использовать для любого стиля общения.
Он обеспечивает поддержку только для протокола HTTP.	Он обеспечивает поддержку протокола HTTP / s: заголовки запроса / ответа URL и т. Д.



Разница между API и веб-сервисами Вот важные различия между веб-сервисами и API.

Веб-сервис - это просто API в одежде HTTP.

Веб-сервис	API
Все веб-сервисы являются API.	Все API не являются веб-сервисами.
Он поддерживает XML.	Ответы форматируются с использованием MediaTypeFormatter Web API в XML, JSON или любой другой заданный формат.
Вам нужен протокол SOAP для отправки или получения данных по сети. Поэтому он не имеет легкой архитектуры.	API имеет легковесную архитектуру.
Он может использоваться любым клиентом, который понимает XML.	Может использоваться клиентом, который понимает JSON или XML.

Веб-сервис использует три стиля: REST, SOAP и XML-RPC для связи.	API можно использовать для любого стиля общения.
Он обеспечивает поддержку только для протокола HTTP.	Он обеспечивает поддержку протокола HTTP / s: заголовки запроса / ответа URL и т. Д.

Преимущества API Сервисов

Вот плюсы / преимущества использования API:

- API поддерживает традиционные действия CRUD (Create Read Update Delete), так как он работает с HTTP-глаголами GET, PUT, POST и DELETE.
- API помогает вам выставлять данные сервиса браузеру
- Он основан на HTTP, который легко определить, предоставить полный REST-способ.

КЛЮЧЕВАЯ РАЗНИЦА

- Веб-сервис — это набор протоколов и стандартов с открытым исходным кодом, используемых для обмена данными между системами или приложениями, а API — это программный интерфейс, который позволяет двум приложениям взаимодействовать друг с другом без какого-либо участия пользователя.
- Веб-сервис используется для REST, SOAP и XML-RPC для связи, в то время как API используется для любого стиля связи.
- Веб-сервис поддерживает только протокол HTTP, тогда как API поддерживает протокол HTTP / HTTPS.
- Веб-сервис поддерживает XML, а API поддерживает XML и JSON.
- Все веб-сервисы являются API-интерфейсами, но все API-интерфейсы не являются веб-сервисами.

СПОСОБЫ РЕАЛИЗАЦИИ ВЕБ-СЕРВИСОВ

XML-RPC (XML Remote Procedure Call)

SOAP (Simple Object Access Protocol)

JSON-RPC (JSON Remote Procedure Call)

REST (Representational State Transfer)

Специализированные протоколы для конкретного вида задач, такие как **GraphQL**.



Самые известные способы реализации веб-сервисов:

- XML-RPC (XML Remote Procedure Call) — протокол удаленного вызова процедур с использованием XML. Прародитель SOAP. Предельно прост в реализации.
- SOAP (Simple Object Access Protocol) — стандартный протокол по версии W3C. Четко структурирован и задокументирован. Данный протокол позволяет отправлять и получать сообщения, используя XML в качестве основы.
- JSON-RPC (JSON Remote Procedure Call) — более современный аналог XML-RPC. Основное отличие — данные передаются в формате JSON.
- REST (Representational State Transfer) — архитектурный стиль взаимодействия компьютерных систем в сети основанный на методах протокола HTTP.
- Специализированные протоколы для конкретного вида задач, такие как GraphQL.
- Менее распространенный, но более эффективный gRPC, передающий данные в бинарном виде и использующий HTTP/2 в качестве транспорта.
- **WSDL**. Это язык, который служит для описания сторонних интерфейсов на базе XML.



Почему выбрано название REST? Representational State Transfer, в переводе википедии «передача состояния представления»... Мысль сводится к тому, что запрос ресурса с сервера переводит клиентское приложение в определенное состояние (state), а запрос следующего ресурса меняет состояние приложения (transfer). А “Representational” означает то, что ресурс возвращается не просто так, а в каком-то представлении, например в представлении для машины или в представлении для человека. Я бы назвал как-то вроде «Стандартизированное оперирование данными»

А Филдинг в своей диссертации признается, что название придумано не для того, чтобы было понятно, о чем речь, а «is intended to evoke an image of how a well-designed Web application behaves».

Как было сказано, REST определяет, как компоненты распределенной системы должны взаимодействовать друг с другом. В общем случае это происходит посредством запросов-ответов. Компоненту, которая отправляет запрос называют **клиентом**; компоненту, которая обрабатывает запрос и отправляет клиенту ответ, называют

сервером. Запросы и ответы, чаще всего, отправляются по протоколу HTTP (англ. HyperText Transfer Protocol — "протокол передачи гипертекста").

Например, у вас может быть сервер, на котором могут размещаться важные документы, изображения или видео. Все это пример ресурсов. Если клиент, скажем, веб-браузер нуждается в любом из этих ресурсов, он должен отправить запрос на сервер для доступа к этим ресурсам. Теперь REST определяет способ доступа к этим ресурсам.

Нашлась и неплохая формулировка идеи по-русски – «представление данных в удобном для клиента формате».



REST – это не протокол и не стандарт, а архитектурный стиль



Важно понимать, что **REST – это не протокол и не стандарт, а архитектурный стиль.** Однако не каждая система, чьи компоненты обмениваются данными посредством запросов-ответов, является REST (или же RESTful) системой. Чтобы система считалась RESTful, она должна “вписываться” в шесть REST ограничений или же принципов

- **REST основывается на HTTP => доступны все плюшки:**
 - **Кэширование.**
 - **Масштабирование.**
 - **Минимум накладных расходов.**
 - **Стандартные коды ошибок.**



REST vs API



REST - это набор rules/standards/guidelines для создания веб-API. Поскольку существует множество способов сделать это, наличие согласованной системы структурирования API экономит время на принятии решений при ее создании и экономит время на понимании того, как ее использовать.

API-это очень широкий термин. Как правило, это то, как один фрагмент кода разговаривает с другим. В веб-разработке API часто относится к способу, которым мы получаем информацию из онлайн-сервиса.

REST-это тип API. Не все APIs являются REST, но все REST службы являются APIs. API-это очень широкий термин. Как правило, это то, как один фрагмент кода разговаривает с другим. В веб-разработке API часто относится к способу, которым мы получаем информацию из онлайн-сервиса. документация The API предоставит вам список URLs, параметры запроса и другую информацию о том, как сделать запрос из API, а также сообщит вам, какой ответ будет дан на каждый запрос.

REST - это набор rules/standards/guidelines для создания веб-API. Поскольку существует множество способов сделать это, наличие согласованной системы структурирования API экономит время на принятии решений при ее создании и экономит время на понимании того, как ее использовать.

Другие популярные парадигмы API включают SOAP и GraphQL.

REST в основном просто относится к использованию протокола HTTP так, как он был задуман.

REST В нем ничего не говорится о форматах возврата, которые с таким же успехом могут быть JSON.

Это противоречит, например, APIs, которые отправляют двоичные или XML сообщения на назначенный порт, не используя различия в методах HTTP или URLs вообще.



Ресурс — это ключевая абстракция, на которой концентрируется протокол HTTP. Ресурс — это все, что вы хотите показать внешнему миру через ваше приложение. Например, если мы пишем приложение для управления задачами, экземпляры ресурсов будут следующие:

Конкретный пользователь

- Конкретная задача
- Список задач

clients — клиенты;
orders — заказы клиентов;
items — товары.

РЕСУРС

URI

/clients — URI всех имеющихся клиентов;

/clients/23 — URI конкретного клиента, а именно клиента с ID=23;

/clients/4/orders — URI всех заказов клиента №4;

/clients/1/orders/12 — URI заказа №12 клиента №1.

Важные составляющие РЕСТ

Ресурс

Данные, которые получают или изменяют клиенты посредством запросов, называют ресурсами. Основа клиент-серверного взаимодействия — манипуляция над ресурсами. **Ресурсы** в REST — это все, чему можно дать имя. Это в каком то смысле как классы в Java. В Java мы можем создать класс для чего угодно. Так и в REST — ресурсом может быть что угодно: пользователь, документ, отчет, заказ. Все это может быть как абстракцией некоторой сущности, так и чем-то конкретным, например, файлом — картинкой, видео, анимацией, PDF файлом.

В рамках нашего примера у нас есть 3 ресурса:

- clients — клиенты;
- orders — заказы клиентов;
- items — товары.

Клиенты отправляют запросы на так называемые эндпоинты, или же конечные точки (end point). Если говорить очень просто, **ЭНДПОИНТ** — это что-то вроде адреса в сети. Если углубляться в суть, можно сказать, что эндпоинт — это **URI**: последовательность символов, идентифицирующая абстрактный или физический ресурс. Uniform Resource Identifier — унифицированный идентификатор ресурса. Иногда конечную точку, или URI называют путем (path) — путем до ресурса. У каждого конкретного ресурса должен быть уникальный URI. Ответственность за то, чтобы у каждого ресурса всегда был свой URI лежит на плечах разработчика сервера. В нашем примере разработчики это мы, поэтому будем делать это так, как умеем. Подобно тому, как в реляционной базе данных часто принято первичным ключом задавать некоторый числовой ID, в REST у каждого ресурса есть свой ID. Часто бывает так, что ID ресурса в REST совпадает с ID записи в базе данных, в которой хранится информация о данном ресурсе. URI в REST принято начинать с множественной формы существительного, описывающего некоторый ресурс. Например, со слова clients. Далее через слэш указывают ID — идентификатор некоторого конкретного клиента. Примеры:

- /clients — URI всех имеющихся клиентов;
- /clients/23 — URI конкретного клиента, а именно клиента с ID=23;
- /clients/4 — URI конкретного клиента, а именно клиента с ID=4.

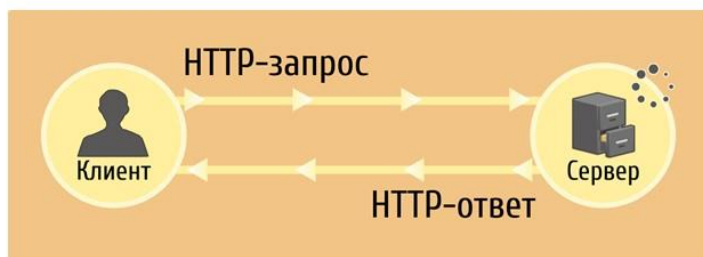
Но и это еще не все. Мы можем продолжить URI, добавив к нему заказы:

- /clients/4/orders — URI всех заказов клиента №4;
- /clients/1/orders/12 — URI заказа №12 клиента №1.

Если мы продолжим эту цепочку и добавим еще и товары, получим:

- /clients/1/orders/12/items — URI списка всех товаров в заказе №12 сделанного клиентом №1.

С уровнями вложенности главное — делать URI интуитивно понятными.



- **Формат обмена данными:** здесь нет никаких ограничений. JSON — очень популярный формат, хотя можно использовать и другие, такие как XML
- **Транспорт:** всегда HTTP. REST полностью построен на основе HTTP.
- **Определение сервиса:** не существует стандарта для этого, а REST является гибким.

Вот как обычно реализуется служба REST:

- **Формат обмена данными:** здесь нет никаких ограничений. JSON — очень популярный формат, хотя можно использовать и другие, такие как XML
- **Транспорт:** всегда HTTP. REST полностью построен на основе HTTP.
- **Определение сервиса:** не существует стандарта для этого, а REST является гибким. Это может быть недостатком в некоторых сценариях, поскольку потребляющему приложению может быть необходимо понимать форматы запросов и ответов. Однако широко используются такие языки определения веб-приложений, как WADL (Web Application Definition Language) и Swagger.

REST фокусируется на ресурсах и на том, насколько эффективно вы выполняете операции с ними, используя HTTP.

МЕТОДЫ HTTP

Запрос	Описание
GET /clients/23 Accept : application/json, application/xml	Получить информацию о клиенте №23 в формате json или xml
POST /clients { "name" : "Amigo", "email" : "amigo@jr.com", "phone" : "+7 (191) 746-43-23" }	Создать нового клиента с полями: Имя — Amigo Email — amigo@jr.com Тел. — +7 (191) 746-43-23
PUT /clients/1 { "name" : "Ben", "email" : "bigben@jr.com", "phone" : "+380 (190) 346-42-13" }	Редактировать клиента №1 в следующем образе: Имя — Ben Email — bigben@jr.com Тел. — +380 (190) 346-42-13
DELETE /clients/12/orders/6	Удалить из системы заказ №6 у клиента №12

Restful Методы.

HTTP метод

Метод HTTP (англ. HTTP Method) — последовательность из любых символов, кроме управляющих и разделителей, которая указывает на основную операцию над ресурсом. Существует несколько общепринятых методов HTTP. Перечислим те из них, которые наиболее часто используются в RESTful сервисах:

- GET — служит для получения информации о конкретном ресурсе (через ID) либо о коллекции ресурсов;
- POST — служит для создания нового ресурса;
- PUT — служит для изменения ресурса (через ID);
- DELETE — служит для удаления ресурса (через ID).

Заголовки

В запросах, как собственно и в ответах, присутствуют HTTP заголовки. В них отправляется дополнительная информация о запросе (либо ответе). Заголовки представляют собой пары ключ-значение. Применительно к REST клиенты часто могут слать в запросе к серверу заголовок **Accept**. Он нужен, чтобы дать серверу понять, в каком формате клиент ожидает получить от него ответ. Различные варианты форматов представлены в так называемом списке MIME-типов. **MIME** (англ. Multipurpose Internet Mail Extensions — многоцелевые расширения интернет-почты) — спецификация для кодирования информации и форматирования сообщений таким образом, чтобы их можно было пересылать по интернету. Каждый MIME тип состоит из двух частей, разделяемых слэшем — из типа и подтипа. Примеры MIME-типов для разных видов файлов:

- text — text/plain, text/css, text/html;
- image — image/png, image/jpeg, image/gif;
- audio — audio/wav, audio/mpeg;
- video — video/mp4, video/ogg;

- application — application/json, application/pdf, application/xml, application/octet-stream.

Итого, у запроса может присутствовать заголовок:

Accept:application/json

Данный заголовок говорит серверу, что клиент ожидает получить ответ в JSON формате.

Тело запроса

Пересылаемое клиентом сообщение на сервер. Есть у запроса тело или нет, зависит от типа HTTP запроса. Например, запросы GET и DELETE как правило не содержат никакого тела запроса. А вот PUT и POST могут содержать: тут все дело в функциональном назначении типа запроса. Ведь для получения данных и удаления по id (который передается в URL) не нужно слать на сервер дополнительные данные. А вот для создания нового ресурса (запрос POST) нужно этот ресурс передать. Также как и для модификации существующего ресурса. В REST для передачи тела запроса чаще всего используют форматы XML или JSON. Наиболее часто встречается JSON формат. Предположим, мы хотим отправить на сервер запрос, а в нем — создать новый ресурс. Если ты не забыл, в качестве примера мы рассматривали приложение, которое управляет заказами клиентов. Допустим, мы хотим создать нового клиента. В нашем случае мы храним следующую информацию о клиентах: Имя Email Номер телефона Тогда телом такого запроса может быть следующий JSON:

```
{
  "name" : "Amigo",
  "email" : "amigo@jr.com",
  "phone" : "+7 (191) 746-43-23"
}
```

Собираем запросы воедино

Итак, мы рассмотрели с тобой из чего может состоять клиентский запрос. Приведем теперь несколько примеров запросов с описанием

Запрос	Описание
GET /clients/23 Accept : application/json, application/xml	Получить информацию о клиенте №23 в
POST /clients { "name" : "Amigo", "email" : "amigo@jr.com", "phone" : "+7 (191) 746-43-23" }	Создать нового клиента с полями: Имя — Amigo Email — amigo@jr.com Тел. — +7 (191) 746-43-23
PUT /clients/1 { "name" : "Ben", "email" : "bigben@jr.com", "phone" : "+380 (190) 346-42-13" }	Редактировать клиента №1 в следующем Имя — Ben Email — bigben@jr.com Тел. — +380 (190) 346-42-13

DELETE /clients/12/orders/6

Удалить из системы заказ №6 у клиен

Ответы

Скажем пару слов об ответах сервера. Ответ как правило состоит из следующих частей:

- код ответа;
- заголовки;
- тело ответа.

В целом заголовки ответов мало чем отличаются от заголовков запросов. К тому же, некоторые заголовки используются и в ответах, и в запросах. С телом ответа тоже, думаю, все понятно. В теле часто возвращается информация которую запросил клиент. Может возвращаться в том же формате JSON информация на GET запросы. А вот последняя часть, немного более интересна.

Коды HTTP ответов **REST** тоже использует. Это стандартные 5 групп

Принципы REST



1. Client-Server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand

Существует 6 принципов написания RESTful-интерфейсов.

1. Client-Server

2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand

Приведение архитектуры к модели клиент-сервер

Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Само разграничение на "клиент" и "сервер" позволяет им развиваться независимо друг от друга.



Client-Server.

В основе данного ограничения лежит разграничение потребностей. Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Данное ограничение повышает переносимость клиентского кода на другие платформы, а упрощение серверной части улучшает масштабируемость системы. Само разграничение на "клиент" и "сервер" позволяет им развиваться независимо друг от друга.

Отделяя пользовательский интерфейс от хранилища данных, мы улучшаем переносимость пользовательского интерфейса на другие платформы и улучшаем масштабируемость серверных компонент за счёт их упрощения.

Приложение не должно беспокоиться о том как сервер сохраняет данные, особенности и тип базы данных, техническая часть и т.д. Точка взаимодействия между сервером и приложением – схема базы данных, то есть что и где храниться. Эта схема часть Апи документации.

Два самых важных пунктов взаимодействия между клиентом и сервером –

1. Формат данных, который нужен клиенту

Сервер должен поддерживать формат данных, который нужен клиенту, иначе взаимодействие будет бесполезным.

Приложение использует данные, а апи продукт. Для того чтобы этот продукт был успешным он должен удовлетворять потребности клиента.

Например если клиенту нужны картинки в формате jpeg это значит что бекенд должен их доставлять. Если на сервере хранятся картинки в формате RAW то нужно предусмотреть то, что их конвертируют клиенту в формате JPEG

Клиент должен заявлять серверу о желаемом формате. Например –
/api/user/id/profilepicture.jpg

2. Предоставлять только нужную информацию. Если клиенту нужна одна фотография, сервер не должен высылать весь альбом. Мораль сей басни такова - чтобы не перегружать трафик – минимизировать трафик и максимизировать пропускную способность) — объем информации, который способен пропустить канал за определенный временной промежуток

1. Client-Server

2. Stateless

- 3. Cacheable
- 4. Uniform interface
- 5. Layered system
- 6. Code on demand



Отсутствие состояния

Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения.



Stateless (без состояния). Архитектура REST требует соблюдения следующего условия. В период между запросами серверу не нужно хранить информацию о состоянии клиента и наоборот. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения.

Каждый запрос от клиента к серверу должен содержать в себе всю необходимую информацию и не может полагаться на какое-либо состояние, хранящееся на стороне сервера. Таким образом, информация о текущей сессии должна целиком храниться у клиента.

Если это было бы в реальной жизни то оно звучало бы так =

Где живет Вадим? Сколько ему лет?

В первом вопросе предполагается что мы должны ответить где живет Вадим. А второй вопрос понятно что идет речь о Вадиме и мы спрашиваем сколько ему лет.

В REST API, такой же диалог будет звучать так:

Где живет Вадим?

Сколько Вадиму лет?

1. Client-Server
2. Stateless
- 3. Cacheable**
4. Uniform interface
5. Layered system
6. Code on demand

Кэширование

Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные.



Cacheable (кэшируемость). Это ограничение требует, чтобы для данных в ответе на запрос явно было указано -- можно их кэшировать или нет. Если ответ поддерживает кэширование, то клиент имеет право повторно использовать данные в последующих эквивалентных запросов без обращения на сервер.

Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные.

Правильное использование кэширования помогает полностью или частично устранить некоторые клиент-серверные взаимодействия, ещё больше повышая производительность и расширяемость системы.

1. Client-Server
2. Stateless
3. Cacheable

4. Uniform interface

5. Layered system
6. Code on demand

Единообразие интерфейса

Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом



Uniform interface (единообразие интерфейса). Если применить к систем инженерный принцип общности/единообразия, то архитектура всего приложения станет проще, а взаимодействие станет прозрачнее и понятнее. Для выполнения этого принципа необходимо придерживаться нескольких архитектурных ограничений.

Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом

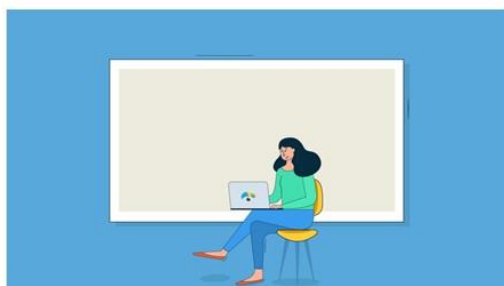
1. Client-Server
2. Stateless
3. Cacheable
4. Uniform interface

5. Layered system

6. Code on demand

Слои

Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования.



Layered system (многоуровневая система). Многоуровневость достигается за счёт ограничения поведения компонентов таким образом, что компоненты "не видят" другие компоненты, кроме расположенных на ближайших уровнях, с которыми они взаимодействуют.

Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом.

Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования.

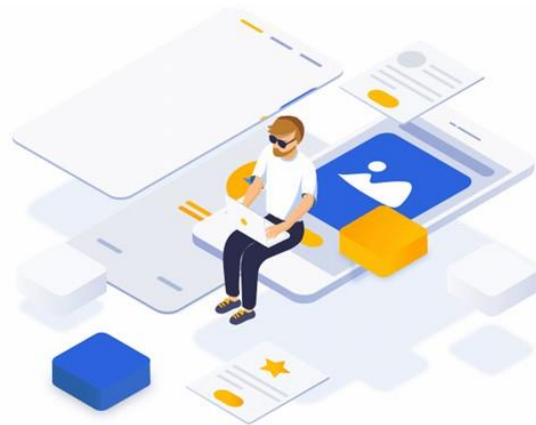
Приведем пример. Представим себе некоторое мобильное приложение, которое пользуется популярностью во всем мире. Его неотъемлемая часть — загрузка картинок. Так как пользователей — миллионы человек, один сервер не смог бы выдержать такой большой нагрузки. Разграничение системы на слои решит эту проблему. Клиент запросит картинку у промежуточного узла, промежуточный узел запросит картинку у сервера, который наименее загружен в данный момент, и вернет картинку клиенту. Если здесь на каждом уровне иерархии правильно применить кэширование, то можно добиться хорошей масштабируемости системы.

1. Client-Server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system

6. Code on demand

Код по требованию (необязательное ограничение)

Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев, скриптов



Code on demand (код по мере необходимости, необязательно). REST позволяет наращивать функциональность клиентского приложения по мере необходимости при помощи скачивания и исполнения кода в виде апплетов или скриптов. Это упрощает клиентские приложения, уменьшая количество заранее написанных возможностей.

POST -- create

GET -- read

PUT -- update/replace

DELETE -- delete

PATCH -- partial update/modify

Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев.

RESTful

всего лишь означает сервис, реализованный с использованием принципов REST.

Только что употребленный термин **RESTful** (веб-)сервис всего лишь означает сервис, реализованный с использованием принципов REST. **Так что же нам дает следование этим самым принципам REST?** Для начала я бы назвал простоту основным преимуществом архитектуры REST. Простоту идеи, простоту разработки и добавления функциональности к RESTful приложениям. Идея настолько проста и универсальна, что ее даже сложно сначала уловить. Мы не добавляем никакого нового слоя в наш и без того многослойный программный пирог, а просто используем уже давно признанные стандарты.

Источники

1. <https://coderlessons.com/tutorials/veb-razrabotka/arkhitektura-veb-servisov/8-api-protiv-veb-sluzhby>
2. <https://overcoder.net/q/442424/%D0%B2-%D1%87%D0%B5%D0%BC-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-api-%D0%B8-%D0%B2%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D0%B0%D0%BC%D0%B8>
3. <https://www.intervolga.ru/blog/projects/relsy-veb-integratsii-rest-i-soap/>
4. <https://coderlessons.com/tutorials/veb-razrabotka/izuchite-veb-servisy/veb-servisy-kratkoe-rukovodstvo>
5. <https://habr.com/ru/post/46374/>
6. <https://medium.com/future-vision/the-principles-of-rest-6b00deac91b3>
7. <https://javarush.ru/groups/posts/2486-obzor-rest-chastjh-1-chto-takoe-rest>
8. <https://javarush.ru/groups/posts/2487-obzor-rest-chastjh-2-kommunikacija-mezhdu-klientom-i-serverom->
9. <https://habr.com/ru/post/131343/>
10. <https://coderlessons.com/tutorials/veb-razrabotka/arkhitektura-veb-servisov/4-veb-servisy-restful>
11. <https://habr.com/ru/post/483202/>