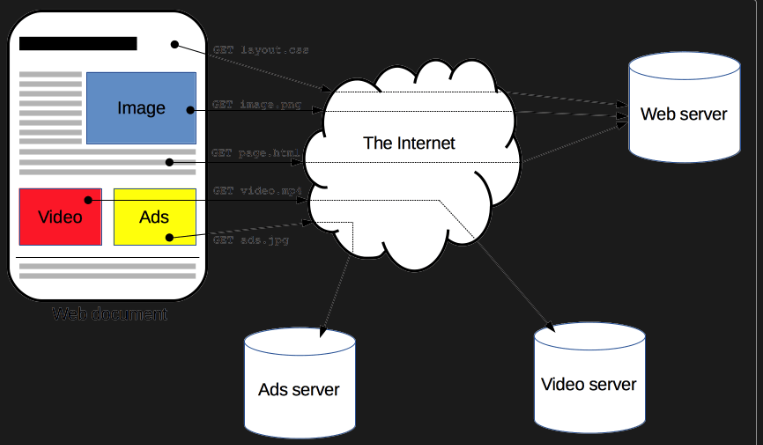
# **Ультимативный гайд по HTTP. Структура запроса и ответа**

**Введение**

Весь современный веб построен на протоколе HTTP. Каждый сайт использует его для общения клиента с сервером. Между собой сервера тоже часто общаются по этому протоколу. На данный момент существует четыре его версии и все они до сих пор используются. Поэтому статьи будут полезны инженерам любых уровней и специализаций, и помогут систематизировать знания об этой важной технологии.

### Что такое HTTP

**HTTP** - это [протокол](https://developer.mozilla.org/en-US/docs/Glossary/Protocol) для извлечения ресурсов, таких как HTML-документы. Это основа любого обмена данными в сети и это протокол клиент-сервер, который означает, что запросы инициируются получателем, обычно веб-браузером. Полный документ восстанавливается из различных выбранных вложенных документов, например текста, описания макета, изображений, видео, сценариев и многого другого.



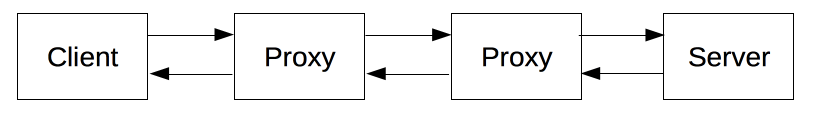
Клиенты и серверы взаимодействуют путем обмена отдельными сообщениями (в отличие от потока данных). Сообщения, отправляемые клиентом, обычно веб-браузером, называются *запросами*, а сообщения, отправляемые сервером в качестве ответа, называются *ответами*

Разработанный в начале 1990-х годов, HTTP представляет собой расширяемый протокол, который развивался с течением времени. Это протокол прикладного уровня, который передается по [TCP](https://developer.mozilla.org/en-US/docs/Glossary/TCP) или по TCP-соединению, зашифрованному с помощью [TLS](https://developer.mozilla.org/en-US/docs/Glossary/TLS), хотя теоретически можно использовать любой надежный транспортный протокол. Благодаря своей расширяемости он используется не только для извлечения гипертекстовых документов, но также изображений и видео или для размещения контента на серверах, например, в результатах HTML-формы. HTTP также можно использовать для извлечения частей документов для обновления веб-страниц по запросу.

## [Компоненты систем, основанных на HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#components_of_http-based_systems)

HTTP - это протокол клиент-сервер: запросы отправляются одним объектом, пользовательским агентом (или прокси-сервером от его имени). В большинстве случаев пользовательским агентом является веб-браузер, но это может быть что угодно, например, робот, который сканирует веб-страницы для заполнения и поддержания индекса поисковой системы.

Каждый отдельный запрос отправляется на сервер, который обрабатывает его и предоставляет ответ, называемый *response*. Между клиентом и сервером существует множество объектов, совместно называемых [прокси](https://developer.mozilla.org/en-US/docs/Glossary/Proxy_server), которые выполняют различные операции и действуют, например, как шлюзы или [кэши](https://developer.mozilla.org/en-US/docs/Glossary/Cache).



На самом деле между браузером и сервером, обрабатывающим запрос, находится больше компьютеров: есть маршрутизаторы, модемы и многое другое. Благодаря многоуровневому дизайну Интернета они скрыты на сетевом и транспортном уровнях. HTTP находится сверху, на прикладном уровне. Хотя он важен для диагностики сетевых проблем, нижележащие уровни в основном не имеют отношения к описанию HTTP.

### [Клиент: пользовательский агент](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#client_the_user-agent)

*User-agent* - это любой инструмент, который действует от имени пользователя. Эту роль в основном выполняет веб-браузер, но она также может выполняться программами, используемыми инженерами и веб-разработчиками для отладки своих приложений.

Браузер **всегда** является объектом, инициирующим запрос. Он никогда не является сервером (хотя с годами были добавлены некоторые механизмы для имитации сообщений, инициируемых сервером).

Для отображения веб-страницы браузер отправляет исходный запрос на получение HTML-документа, представляющего эту страницу. Затем он анализирует этот файл, делая дополнительные запросы, соответствующие сценариям выполнения, информации о макете (CSS) для отображения и вспомогательным ресурсам, содержащимся на странице (обычно изображениям и видео). Затем веб-браузер объединяет эти ресурсы для представления полного документа, веб-страницы. Скрипты, выполняемые браузером, могут извлекать дополнительные ресурсы на более поздних этапах, и браузер соответствующим образом обновляет веб-страницу.

Веб-страница - это гипертекстовый документ. Это означает, что некоторые части отображаемого содержимого являются ссылками, которые могут быть активированы (обычно щелчком мыши) для перехода на новую веб-страницу, позволяя пользователю управлять своим пользовательским агентом и перемещаться по сети. Браузер преобразует эти указания в HTTP-запросы и дополнительно интерпретирует HTTP-ответы, чтобы предоставить пользователю четкий ответ.

### [Веб-сервер](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#the_web_server)

На противоположной стороне канала связи находится сервер, который *обслуживает* документ по запросу клиента. Виртуально сервер выглядит как всего лишь одна машина; но на самом деле это может быть набор серверов, распределяющих нагрузку (балансировка нагрузки), или другое программное обеспечение (например, кэши, сервер базы данных или серверы электронной коммерции), полностью или частично генерирующее документ по запросу.

Сервер - это не обязательно одна машина, но на одной машине может быть размещено несколько экземпляров серверного программного обеспечения. С HTTP / 1.1 и [Host](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Host) заголовком они могут даже использовать один и тот же IP-адрес.

### [Прокси-серверы](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#proxies)

Между веб-браузером и сервером многочисленные компьютеры ретранслируют HTTP-сообщения. Из-за многоуровневой структуры веб-стека большинство из них работают на транспортном, сетевом или физическом уровнях, становясь прозрачными на уровне HTTP и потенциально оказывая значительное влияние на производительность. Те, которые работают на прикладных уровнях, обычно называются **прокси**. Они могут быть прозрачными, пересылающими полученные запросы без их какого-либо изменения, или непрозрачными, в этом случае они каким-либо образом изменят запрос, прежде чем передать его на сервер. Прокси-серверы могут выполнять множество функций:

* кэширование (кэш может быть общедоступным или частным, как кэш браузера)
* фильтрация (например, антивирусная проверка или родительский контроль)
* балансировка нагрузки (позволяет нескольким серверам обслуживать разные запросы)
* аутентификация (для контроля доступа к различным ресурсам)
* ведение журнала (позволяет хранить историческую информацию)

## [Основные аспекты HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#basic_aspects_of_http)

### [HTTP прост](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_simple)

HTTP, как правило, разработан таким образом, чтобы быть простым и понятным человеку, даже с учетом дополнительной сложности, привнесенной в HTTP / 2 путем инкапсуляции HTTP-сообщений во фреймы. HTTP-сообщения могут быть прочитаны и поняты людьми, что упрощает тестирование для разработчиков и снижает сложность для новичков.

### [HTTP является расширяемым](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_extensible)

[HTTP-заголовки](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers), представленные в HTTP / 1.0, позволяют легко расширять этот протокол и экспериментировать с ним. Новая функциональность может быть введена даже простым соглашением между клиентом и сервером о семантике нового заголовка.

### [HTTP не имеет состояния, но не не имеет сеанса](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_stateless_but_not_sessionless)

HTTP не имеет состояния: нет связи между двумя запросами, последовательно выполняемыми по одному и тому же соединению. Это сразу же может стать проблемой для пользователей, пытающихся согласованно взаимодействовать с определенными страницами, например, используя корзины покупок электронной коммерции. Но хотя ядро самого HTTP не имеет состояния, HTTP cookies позволяют использовать сеансы с отслеживанием состояния. Используя расширяемость заголовка, HTTP-файлы cookie добавляются в рабочий процесс, позволяя создавать сеансы для каждого HTTP-запроса с использованием одного и того же контекста или одного и того же состояния.

### [HTTP и соединения](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_and_connections)

Соединение контролируется на транспортном уровне и, следовательно, принципиально выходит за рамки HTTP. Протокол HTTP не требует базового транспортного протокола связи, основанного; он лишь требует, чтобы он был *надежный*, или не терять сообщения (как минимум, представляя ошибка в таких случаях). Среди двух наиболее распространенных транспортных протоколов в Интернете TCP является надежным, а UDP - нет. Поэтому HTTP полагается на стандарт TCP, который основан на соединении.

Прежде чем клиент и сервер смогут обмениваться парой HTTP запрос / ответ, они должны установить TCP-соединение, процесс, который требует нескольких обходов. Поведение HTTP / 1.0 по умолчанию заключается в открытии отдельного TCP-соединения для каждой пары HTTP-запрос / ответ. Это менее эффективно, чем совместное использование одного TCP-соединения, когда несколько запросов отправляются последовательно.

Чтобы устранить этот недостаток, в HTTP / 1.1 была введена *конвейеризация* (которая оказалась сложной в реализации) и *постоянные соединения*: базовым TCP-соединением можно частично управлять с помощью [Connection](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Connection) заголовка. HTTP / 2 пошел еще дальше, мультиплексируя сообщения по одному соединению, помогая поддерживать соединение теплым и более эффективным.

Ведутся эксперименты по разработке лучшего транспортного протокола, более подходящего для HTTP. Например, Google экспериментирует с [QUIC](https://en.wikipedia.org/wiki/QUIC), который основан на UDP, чтобы обеспечить более надежный и эффективный транспортный протокол.

## [Чем можно управлять с помощью HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#what_can_be_controlled_by_http)

Такая расширяемая природа HTTP со временем позволила расширить контроль и функциональность веб-интерфейса. Методы кэширования и аутентификации были функциями, которые использовались на ранних этапах истории HTTP. Возможность ослабить *ограничение происхождения*, напротив, была добавлена только в 2010-х годах.

Вот список общих функций, управляемых с помощью HTTP:

* [*Кэширование*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching): Способ кэширования документов может контролироваться HTTP. Сервер может указывать прокси-серверам и клиентам, что нужно кэшировать и на какой срок. Клиент может дать указание прокси-серверам промежуточного кэша игнорировать сохраненный документ.
* *Ослабление ограничения источника*: Для предотвращения слежки и других вторжений в частную жизнь веб-браузеры обеспечивают строгое разделение между веб-сайтами. Только страницы из **того же источника** могут получить доступ ко всей информации веб-страницы. Хотя такое ограничение является бременем для сервера, заголовки HTTP могут ослабить это строгое разделение на стороне сервера, позволяя документу превратиться в лоскутное одеяло из информации, полученной из разных доменов; для этого даже могут быть причины, связанные с безопасностью.
* *Аутентификация*: некоторые страницы могут быть защищены таким образом, что доступ к ним могут получить только определенные пользователи. Базовая аутентификация может быть обеспечена с помощью HTTP, либо с использованием [WWW-Authenticate](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/WWW-Authenticate) и подобных заголовков, либо путем настройки определенного сеанса с использованием [HTTP cookies](https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies).
* [*Прокси и туннелирование*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling): Серверы или клиенты часто расположены во внутренних сетях и скрывают свой истинный IP-адрес от других компьютеров. Затем HTTP-запросы проходят через прокси, чтобы преодолеть этот сетевой барьер. Не все прокси являются HTTP-прокси. Протокол SOCKS, например, работает на более низком уровне. Другие протоколы, такие как ftp, могут обрабатываться этими прокси.
* *Сеансы*: Использование HTTP cookies позволяет вам связывать запросы с состоянием сервера. Это создает сеансы, несмотря на то, что базовый HTTP является протоколом без состояния. Это полезно не только для корзин покупок электронной коммерции, но и для любого сайта, позволяющего пользователю настраивать выходные данные.

## [HTTP-поток](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_flow)

Когда клиент хочет установить связь с сервером, будь то конечный сервер или промежуточный прокси, он выполняет следующие шаги:

1. Открыть TCP-соединение: TCP-соединение используется для отправки запроса или нескольких запросов и получения ответа. Клиент может открыть новое соединение, повторно использовать существующее соединение или открыть несколько TCP-соединений с серверами.
2. Отправить HTTP-сообщение: HTTP-сообщения (до HTTP / 2) доступны для чтения человеком. С HTTP / 2 эти простые сообщения инкапсулированы во фреймы, что делает невозможным их прямое чтение, но принцип остается тем же. Например:
3. HTTPКопировать в буфер обмена
4. GET / HTTP/1.1  
   Host: developer.mozilla.org  
   Accept-Language: fr
5. Прочитайте ответ, отправленный сервером, например:
6. HTTPКопировать в буфер обмена
7. HTTP/1.1 200 OK  
   Date: Sat, 09 Oct 2010 14:28:02 GMT  
   Server: Apache  
   Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT  
   ETag: "51142bc1-7449-479b075b2891b"  
   Accept-Ranges: bytes  
   Content-Length: 29769  
   Content-Type: text/html  
     
   <!DOCTYPE html>… (here come the 29769 bytes of the requested web page)
8. Закройте или повторно используйте соединение для дальнейших запросов.

Если включена конвейерная обработка HTTP, можно отправить несколько запросов, не дожидаясь полного получения первого ответа. Конвейерную обработку HTTP оказалось сложно реализовать в существующих сетях, где старые части программного обеспечения сосуществуют с современными версиями. Конвейерная обработка HTTP была заменена в HTTP / 2 более надежными запросами мультиплексирования внутри фрейма.

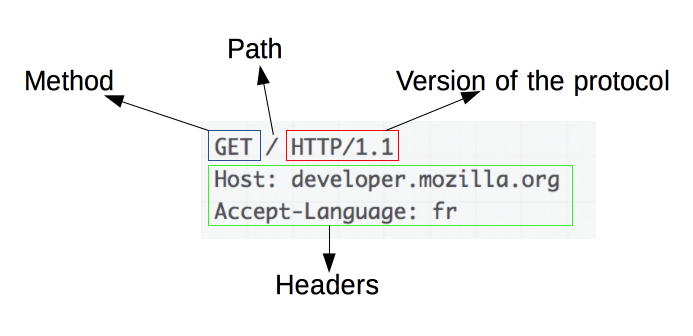
## [HTTP-сообщения](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_messages)

HTTP-сообщения, как определено в HTTP / 1.1 и более ранних версиях, доступны для чтения человеком. В HTTP/2 эти сообщения встроены в двоичную структуру, *фрейм*, что позволяет оптимизировать такие операции, как сжатие заголовков и мультиплексирование. Даже если в этой версии HTTP отправляется только часть исходного HTTP-сообщения, семантика каждого сообщения остается неизменной, и клиент восстанавливает (виртуально) исходный HTTP/ 1.1 запрос. Поэтому полезно понимать сообщения HTTP / 2 в формате HTTP / 1.1.

Существует два типа HTTP-сообщений, запросов и ответов, каждое со своим собственным форматом.

### [Запросы](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#requests)

Пример HTTP-запроса:

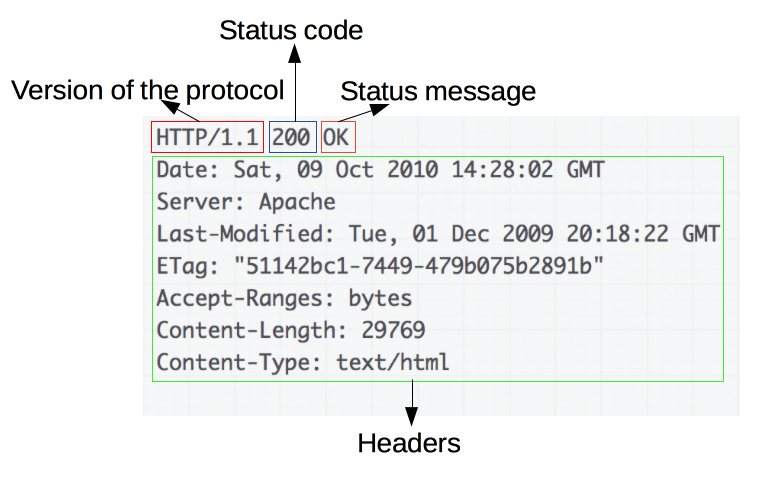


Запросы состоят из следующих элементов:

* [Метод](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) HTTP, обычно использующий глагол типа [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET), [POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) или существительное типа [OPTIONS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS) или [HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD), который определяет операцию, которую хочет выполнить клиент. Обычно клиент хочет получить ресурс (используя GET) или опубликовать значение [HTML-формы](https://developer.mozilla.org/en-US/docs/Learn/Forms) (используя POST), хотя в других случаях может потребоваться больше операций.
* Путь к ресурсу для извлечения; URL ресурса, лишенный элементов, которые очевидны из контекста, например, без [протокола](https://developer.mozilla.org/en-US/docs/Glossary/Protocol) (http://), [домена](https://developer.mozilla.org/en-US/docs/Glossary/Domain) (здесь, developer.mozilla.org) или [порта](https://developer.mozilla.org/en-US/docs/Glossary/Port) TCP (здесь, 80).
* Версия протокола HTTP.
* Необязательные [заголовки](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers), которые передают дополнительную информацию для серверов.
* Тело для некоторых методов, таких как POST, аналогичное тем, что указаны в ответах, которые содержат отправленный ресурс.

### [Ответы](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#responses)

Пример ответа:



Ответы состоят из следующих элементов:

* Версия протокола HTTP, которому они следуют.
* [Код состояния](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status), указывающий, был ли запрос успешным или нет, и почему.
* Сообщение о состоянии, неавторизованное краткое описание кода состояния.
* [Заголовки](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers) HTTP, подобные тем, что используются для запросов.
* При необходимости, тело, содержащее выбранный ресурс.

## [API-интерфейсы, основанные на HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#apis_based_on_http)

Наиболее часто используемым API, основанным на HTTP, является [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) API, который может использоваться для обмена данными между [пользовательским агентом](https://developer.mozilla.org/en-US/docs/Glossary/User_agent) и сервером. Modern [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) предоставляет те же функции с более мощным и гибким набором функций.

Другой API, [события, отправляемые сервером](https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events), представляет собой односторонний сервис, который позволяет серверу отправлять события клиенту, используя HTTP в качестве транспортного механизма. Используя [EventSource](https://developer.mozilla.org/en-US/docs/Web/API/EventSource) интерфейс, клиент открывает соединение и устанавливает обработчики событий. Клиентский браузер автоматически преобразует сообщения, поступающие по HTTP-потоку, в соответствующие [Event](https://developer.mozilla.org/en-US/docs/Web/API/Event) объекты. Затем он доставляет их обработчикам событий, которые были зарегистрированы для событий, [type](https://developer.mozilla.org/en-US/docs/Web/API/Event/type) если они известны, или [onmessage](https://developer.mozilla.org/en-US/docs/Web/API/EventSource/message_event) обработчику событий, если не был установлен обработчик событий для конкретного типа.

## [Заключение](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#conclusion)

HTTP - это расширяемый протокол, который прост в использовании. Структура клиент-сервер в сочетании с возможностью добавления заголовков позволяет HTTP развиваться вместе с расширенными возможностями Интернета.

Хотя HTTP / 2 добавляет некоторую сложность за счет встраивания HTTP-сообщений во фреймы для повышения производительности, базовая структура сообщений осталась неизменной со времен HTTP / 1.0. Поток сеансов остается простым, что позволяет исследовать и отлаживать его с помощью простого [монитора HTTP-сообщений](https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/index.html).

**Ультимативный гайд**

**HTTP** — это гипертекстовый протокол передачи данных прикладного уровня в сетевой модели OSI. Его представил миру Тим Бернерс-Ли в марте 1991 года. Главная особенность HTTP — представление всех данных в нём в виде простого текста. Через HTTP разные узлы в сети общаются между собой. Модель клиент-серверного взаимодействия классическая: клиент посылает запрос серверу, сервер обрабатывает запрос и возвращает ответ клиенту. Полученный ответ клиент обрабатывает и решает: прекратить взаимодействие или продолжить отправлять запросы.

Ещё одна особенность: протокол не сохраняет состояние между запросами. Каждый запрос от клиента для сервера — отдельная транзакция. Когда поступают два соседних запроса, сервер не понимает, от одного и того же клиента они поступили, или от разных. Такой подход значительно упрощает построение архитектуры веб-серверов.

Как правило, передача данных по HTTP осуществляется через открытое TCP/IP-соединение1. Серверное программное обеспечение по умолчанию обычно использует TCP-порт 80 для работы веб-сервера, а порт 443 — для HTTPS-соединений.

**HTTPS (HTTP *Secure***) — это надстройка над протоколом HTTP, которая поддерживает шифрование посредством криптографических протоколов SSL и TLS. Они шифруют отправляемые данные на клиенте и дешифруют их на сервере. Это защищает данные от чтения злоумышленниками, даже если им удастся их перехватить.

### HTTP/0.9

В 1991 году была опубликована первая версия протокола с названием HTTP/0.9. Эта реализация была проста, как топор. От интернет-ресурса того времени требовалось только загружать запрашиваемую HTML-страницу и HTTP/0.9 справлялся с этой задачей. Обычный запрос к серверу выглядел так:

GET /http-spec.html

В протоколе был определен единственный метод GET и и указывался путь к ресурсу. Так пользователи получали страничку. После этого открытое соединение сразу закрывалось.

### HTTP/1.0

Годы шли и интернет менялся. Стало понятно, что нужно не только получать странички от сервера, но и отправлять ему данные. В 1996 году вышла версия протокола 1.0.

**Что изменилось:**

1. В запросе теперь надо было указывать версию протокола. Так сервер мог понимать, как обрабатывать полученные данные.
2. В ответе от сервера появился статус завершения обработки запроса.
3. К запросу и ответу добавился новый блок с заголовками.
4. Добавили поддержку новых методов:

* HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа. Так можно получить только заголовки, без самого ресурса.
* POST добавляет сущность к определённому ресурсу. Часто вызывает изменение состояния или побочные эффекты на сервере. Например, так можно отправить запрос на добавление нового поста в блог.

### Структура запроса

Простой пример запроса:

GET /path HTTP/1.0  
Content-Type: text/html; charset=utf-8  
Content-Length: 4  
X-Custom-Header: value  
  
test

В первой строчке указаны метод запроса — GET, путь к ресурсу — /path и версия протокола — HTTP/1.0.

Далее идёт блок заголовков. Заголовки — это пары ключ: значение, каждая из которых записывается с новой строки и разделяется двоеточием. Они передают дополнительные данные и настройки от клиента к серверу и обратно.

HTTP — это текстовый протокол, поэтому и все данные передаются в виде текста. Заголовки можно отделить друг от друга только переносом строки. Нельзя использовать запятые, точку с запятой, или другие разделители. Всё, что идет после имени заголовка с двоеточием и до переноса строки, будет считаться значением заголовка2.

**В примере серверу передали три заголовка:**

1. Content-Type — стандартный заголовок. Показывает, в каком формате будут передаваться данные в теле запроса или ответа.
2. Content-Length — сообщает длину контента в теле запроса в байтах.
3. X-Custom-Header — пользовательские заголовки, начинающиеся с X- с произвольными именем. Через них реализуется специфическая логика обработки для конкретного сервера. Если веб-сервер не поддерживает такие заголовки, то он проигнорирует их.

После блока заголовков идёт тело запроса, в котором передается текст test*.*

**А так может выглядеть ответ от сервера:**

HTTP/1.1 200 OK  
Date: Thu, 29 Jul 2021 19:20:01 GMT  
Content-Type: text/html; charset=utf-8  
Content-Length: 2  
Connection: close  
Server: gunicorn/19.9.0  
Access-Control-Allow-Origin: \*  
Access-Control-Allow-Credentials: true  
  
OK

В первой строке — версия протокола и статус ответа, например, 200 ОК. Далее идут заголовки ответа. После блока заголовков — тело ответа, в котором записан текст OK.

### Статусы ответов

Клиенту зачастую недостаточно просто отправить запрос на сервер. Во многих случаях надо дождаться ответа и понять, как сервер обработал запрос. Для этого были придуманы статусы ответов. Это трёхзначные числовые коды с небольшими текстовыми обозначениями. Их можно увидеть в терминале или браузере. Сами коды делятся на 5 классов:

* Информационные ответы: коды 100–199
* Успешные ответы: коды 200–299
* Редиректы: коды 300–399
* Клиентские ошибки: коды 400–499
* Серверные ошибки: коды 500–599

Мы рассмотрим основные коды, которые чаще всего встречаются в реальных задачах. С остальными более подробно можно ознакомиться в реестре кодов состояния HTTP.

#### Информационные ответы

**100 Continue** — промежуточный ответ. Он указывает, что запрос успешно принят. Клиент может продолжать присылать запросы или проигнорировать этот ответ, если запрос был завершён.

Примечание

**101 Switching Protocol** присылается в ответ на запрос, в котором есть заголовок *Upgrade*. Это означает, что сервер переключился на протокол, который был указан в заголовке. Такая методика используется, например, для переключения на протокол *Websocket*.

**102 Processing** — запрос получен сервером, но его обработка ещё не завершена.

#### Успешные ответы

**200 OK** — запрос принят и корректно обработан веб-сервером.

**201 Created** — запрос корректно обработан и в результате был создан новый ресурс. Обычно он присылается в ответ на POST запрос.

#### Редиректы

**301 Moved Permanently** — запрашиваемый ресурс на постоянной основе переехал на новый адрес. Тогда новый путь к ресурсу указывается сервером в заголовке Location ответа.

Примечание

**302 Found** — указывает, что целевой ресурс временно доступен по другому URl. Адрес перенаправления может быть изменен в любое время, а клиент должен продолжать использовать действующий URI для будущих запросов. Тогда временный путь к ресурсу указывается сервером в заголовке Location ответа.

Примечание

**307 Temporary Redirect** — имеет то же значение, что и код 302, за исключением того, что клиент не может менять метод последующего запроса.

**308 Permanent Redirect** — имеет то же значение, что и код 301, за исключением того, что клиент не может менять метод последующего запроса.

#### Клиентские ошибки

**400 Bad Request** — запрос от клиента к веб-серверу составлен некорректно. Обычно это происходит, если клиент не передаёт необходимые заголовки или параметры.

**401 Unauthorized** — получение запрашиваемого ресурса доступно только аутентифицированным пользователям.

**403 Forbidden** — у клиента не хватает прав для получения запрашиваемого ресурса. Например, когда обычный пользователь сайта пытается получить доступ к панели администратора.

**404 Not Found** — сервер не смог найти запрашиваемый ресурс.

**405 Method Not Allowed** — сервер знает о существовании HTTP-метода, который был указан в запросе, но не поддерживает его. В таком случае сервер должен вернуть список поддерживаемых методов в заголовке Allow ответа.

#### Серверные ошибки

**500 Internal Server Error** — на сервере произошла непредвиденная ошибка.

**501 Not Implemented** — метод запроса не поддерживается сервером и не может быть обработан.

**502 Bad Gateway** — сервер, действуя как шлюз или прокси, получил недопустимый ответ от входящего сервера, к которому он обращался при попытке выполнить запрос.

**503 Service Unavailable** — сервер не готов обработать запрос (например, из-за технического обслуживания или перегрузки). Обратите внимание, что вместе с этим ответом должна быть отправлена удобная страница с объяснением проблемы. Этот ответ следует использовать для временных условий, а HTTP-заголовок Retry-After по возможности должен содержать расчётное время до восстановления службы.

**504 Gateway Timeout** — этот ответ об ошибке выдается, когда сервер действует как шлюз и не может получить ответ за отведенное время.

**505 HTTP Version Not Supported** — версия HTTP, используемая в запросе, не поддерживается сервером.

В HTTP из всего диапазона кодов используется совсем немного. Те коды, которые не используются для задания определенной логики в спецификации, являются неназначенными и могут использоваться веб-серверами для определения своей специфической логики. Это значит, что вы можете, например, придать коду 513 значение «Произошла ошибка обработки видео», или любое другое. Неназначенные коды вы можете посмотреть в реестре кодов состояния HTTP.3

### Тело запроса и ответа

Тело запроса опционально и всегда отделяется от заголовков пустой строкой. А как понять, где оно заканчивается? Всё кажется очевидным: где кончается строка, там и заканчивается тело. Однако, два символа переноса строки в HTTP означают конец запроса и отправляют его на сервер. Как быть, если мы хотим передать в теле текст, в котором есть несколько абзацев с разрывами в две строки?

POST /path HTTP/1.1  
Host: localhost  
  
Первая строка  
  
  
Вторая строка после разрыва

По логике работы HTTP соединение отправится сразу после второй пустой строки и сервер получит в качестве данных только строку Первая строка. Описанную проблему решает специальный заголовок Content-Length. Он указывает на длину контента в байтах. Обычно клиенты (например, браузеры) автоматически считают длину передаваемых данных и добавляют к запросу заголовок с этим значением. Когда сервер получит запрос, он будет ожидать в качестве контента ровно столько байт, сколько указано в заголовке.

Однако, этого недостаточно для того, чтобы передать данные на сервер. Поведение зависит от реализации сервера, но для большинства из них необходимо также передать заголовок *Content-Type*. Он указывает на тип передаваемых данных. В качестве значения для этого заголовка используют MIME-типы.4

**MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения интернет-почты)** — стандарт, который является частью протокола HTTP. Задача MIME — идентифицировать тип содержимого документа по его заголовку. К примеру, текстовый файл имеет тип text/plain, а HTML-файл — text/html.

Для передачи данных в формате обычного текста надо указать заголовок Content-Type: text/plain, а для JSON — Content-Type: application/json.

Можно ли передать тело с GET-запросом?

Популярный вопрос на некоторых собеседованиях: «Можно ли передать тело с GET-запросом?». Правильный ответ — да. Тело запроса не зависит от метода. В стандарте не описана возможность принимать тело запроса у GET-метода, но это и не запрещено. Технически вы можете отправить данные в теле, но скорее всего веб-сервер будет их игнорировать.

Представим, что на абстрактном сайте есть форма аутентификации пользователя, в которой есть всего два поля: email и пароль.

Если пользователь ввёл данные и нажал на кнопку «Войти», то данные из полей формы должны попасть на сервер. Самым простым и распространенным форматом передачи таких данных будет MIME application/x-www-form-urlencoded. В нем все поля передаются в одной строке в формате ключ=значение и разделяются знаком &.

**Запрос на отправку данных будет выглядеть так:**

POST /login HTTP/1.0  
Host: example.com  
Content-Type: application/x-www-form-urlencoded; charset=utf-8  
Content-Length: 26  
login=user&password=qwerty

Тут есть небольшая особенность. Как понять, где заканчивается ключ и начинается его значение, если в пароле будет присутствовать знак «=» ?

POST /login HTTP/1.0  
Host: example.com  
Content-Type: application/x-www-form-urlencoded; charset=utf-8  
Content-Length: 26  
login=user&password=123=45

В этом случае сервер не сможет понять, как разбить строку на параметры и их значения. На самом деле значения кодируются при помощи механизма url encoding.5 При использовании этого механизма знак «=» будет преобразован в код %3D .

**Тогда наш запрос приобретёт такой вид:**

POST /login HTTP/1.1  
Host: example.com  
Content-Type: application/x-www-form-urlencoded; charset=utf-8  
Content-Length: 28  
  
login=user&password=123%3D45

### Query string

Данные на сервер можно передавать через тело запроса и через так называемую строку запроса Query String. Это параметры формата *ключ=значение*, которые располагаются в пути к ресурсу:

GET /files?key=value&key2=value2 HTTP/1.0

При этом параметры можно передавать прямо от корня домена:

GET /?key=value&key2=value2 HTTP/1.0

Query String имеет такой же формат, как и тело запроса с MIME application/x-www-form-urlencoded*,* только первая пара значений отделяется от адреса вопросительным знаком.

Некоторые инженеры ошибочно полагают, что Query String являются параметрами GET-запроса и даже называют их GET-параметрами, но на самом деле это не так. Как и тело запроса, Query String не имеет привязки к HTTP-методам и может передаваться с любым типом запросов.

Обычно параметры Query String используются в GET-запросах, чтобы конкретизировать получаемый ресурс. Например, можно получить на сервере список файлов, имена которых будут начинаться с переданного значения.

GET-запросы по своей идеологии должны быть идемпотентными. Это значит, что множественный вызов метода с одними и теми же параметрами должен приводить к одному и тому же результату. Например, поисковые боты перемещаются по сайту только по ссылкам и делают только GET-запросы, потому что исходя из семантики они не смогут таким образом изменить данные на сайте и повлиять на его работу.

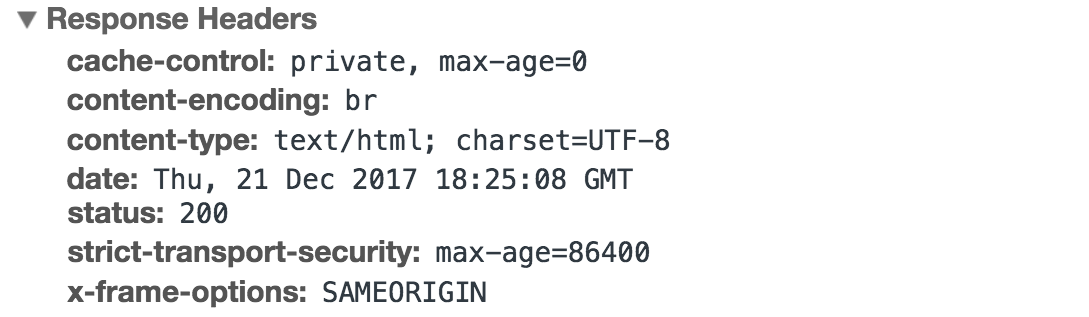
## **Могут ли DDoS-атаки запускаться через HTTP?**

Имейте в виду, что HTTP - это протокол без состояния, что означает, что каждая команда выполняется независимо от любой другой команды. В исходной спецификации каждый HTTP-запрос создавал и закрывал [TCP](https://www.cloudflare.com/learning/ddos/glossary/tcp-ip/) соединение. В более новых версиях протокола HTTP (HTTP 1.1 и выше) постоянное соединение позволяет передавать несколько HTTP-запросов по постоянному TCP-соединению, что улучшает потребление ресурсов. В контексте [DoS](https://www.cloudflare.com/learning/ddos/glossary/denial-of-service/) или [DDoS-атак](https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/) HTTP-запросы в больших количествах могут использоваться для организации атаки на целевое устройство и считаются частью атак прикладного уровня или атак [уровня 7](https://www.cloudflare.com/learning/ddos/what-is-layer-7/).

## Что такое заголовки HTTP-ответа?

Подобно HTTP-запросу, HTTP-ответ поставляется с заголовками, которые передают важную информацию, такую как язык и формат отправляемых данных в теле ответа.

Пример заголовков HTTP-ответа на вкладке сеть в Google Chrome:



## Что находится в теле HTTP-ответа?

Успешные HTTP-ответы на запросы ‘GET’ обычно имеют тело, содержащее запрошенную информацию. В большинстве веб-запросов это HTML-данные, которые веб-браузер преобразует в веб-страницу.