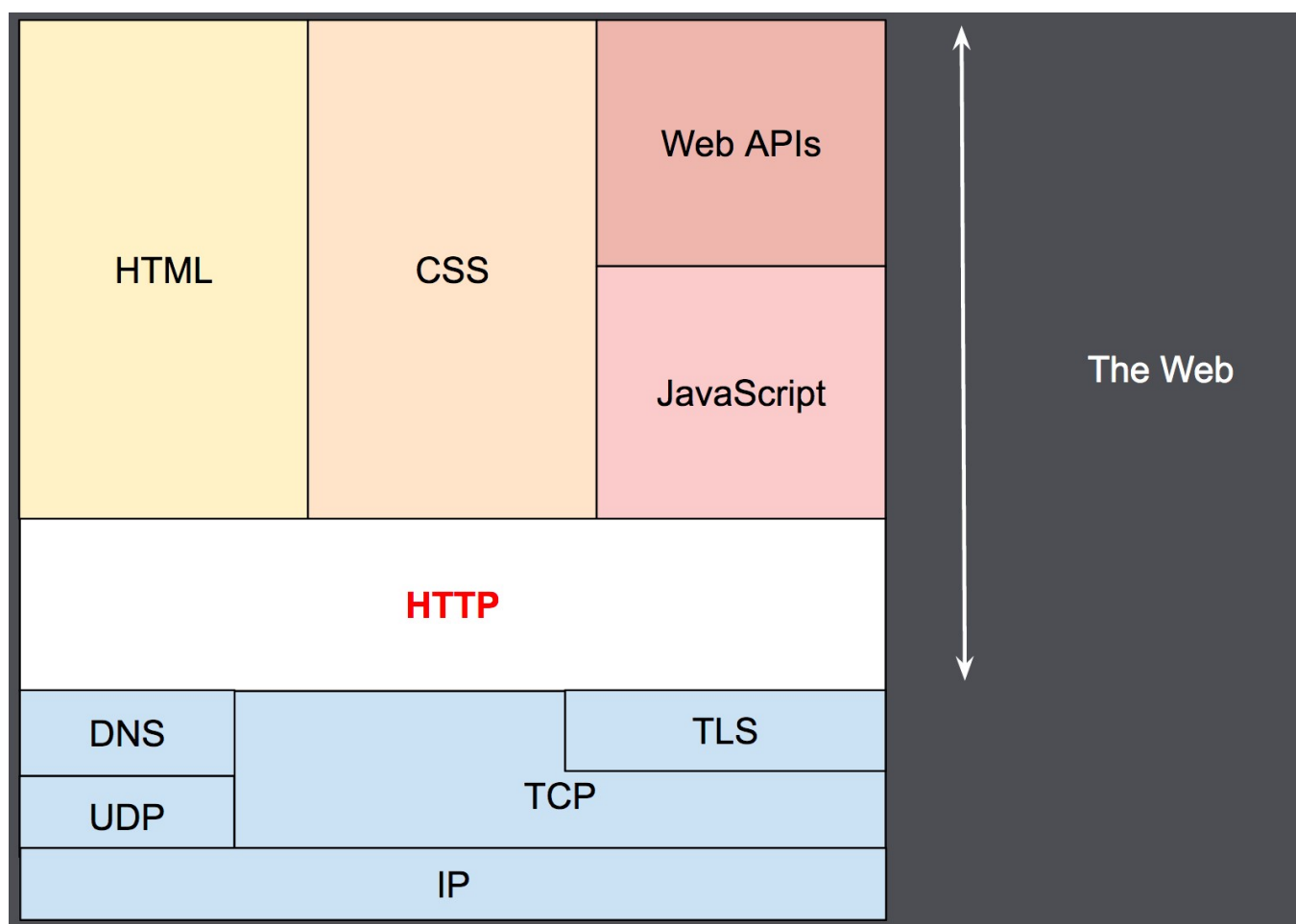


# Протокол HTTP

HTTP — широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов (то есть документов, которые могут содержать ссылки, позволяющие организовать переход к другим документам).

Аббревиатура HTTP расшифровывается как HyperText Transfer Protocol, "протокол передачи гипертекста". В соответствии со спецификацией OSI, HTTP является протоколом прикладного (верхнего, 7-го) уровня. Актуальная на данный момент версия протокола, HTTP 1.1, описана в спецификации RFC 2616.



Протокол HTTP предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает данный запрос, формирует ответ и передаёт его обратно клиенту. После этого клиентское приложение может продолжить отправлять другие запросы, которые будут обработаны аналогичным образом. Задача, которая традиционно решается с помощью протокола HTTP — обмен данными между пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-браузер) и веб-сервером. На данный момент именно благодаря протоколу HTTP обеспечивается работа Всемирной паутины.

Передача данных по протоколу HTTP осуществляется через TCP/IP-соединения. По-

умолчанию используются TCP-порт 80 или 443 для HTTP или HTTPS соответственно, при этом их можно явно не указывать. Если используются порты, отличные от стандартных, то их требуется явно указать в URL.

Пользователь ввел в адресной строке example.com, браузеру нужно подключиться к веб-серверу по адресу example.com по 80 порту. Попробуем эмулировать это поведение через telnet-терминал.

```
shell telnet example.com 80
```

После того как мы подключились к серверу, нужно отправить HTTP-запрос. Для формирования HTTP-запроса необходимо составить стартовую строку, а также задать по крайней мере один заголовок — это заголовок Host, который является обязательным, и должен присутствовать в каждом запросе. Дело в том, что преобразование доменного имени в IP-адрес осуществляется на стороне клиента, а на зарезолвленном IP-адресе могут находиться несколько виртуальных серверов. Таким образом заголовок Host позволяет определить к какому ресурсу обращаться по этому IP. Стартовая (начальная) строка запроса для HTTP 1.1 составляется по следующей схеме: *Метод URI HTTP/Версия*.

```
shell GET / HTTP/1.1 host: example.com
```

После обработки запроса веб-сервер пришлет ответ. Стартовая строка ответа имеет следующую структуру: *HTTP/Версия Код состояния Пояснение*.

```
```shell telnet example.com 80 Trying 93.184.216.34... Connected to example.com. Escape character is '^'. GET / HTTP/1.1 host: example.com
```

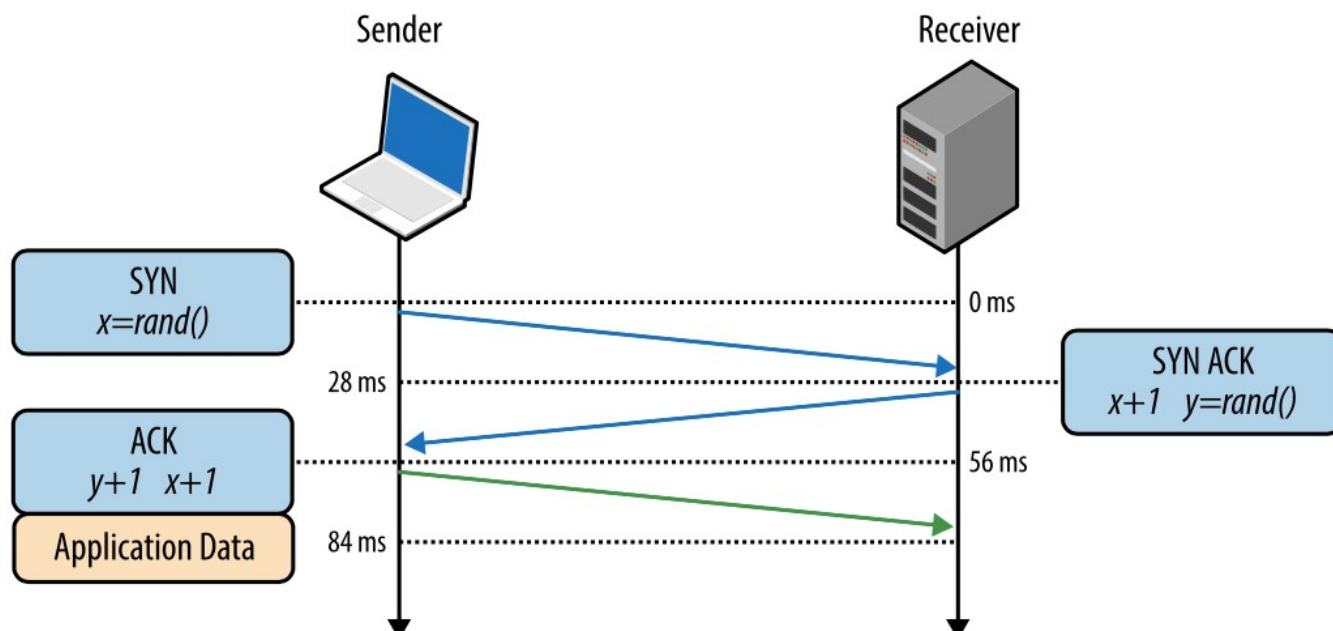
```
HTTP/1.1 200 OK Age: 433110 Cache-Control: max-age=604800 Content-Type: text/html; charset=UTF-8 Date: Tue, 14 Sep 2021 19:59:13 GMT Etag: "3147526947+ident" Expires: Tue, 21 Sep 2021 19:59:13 GMT Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT Server: ECS (dcb/7F37) Vary: Accept-Encoding X-Cache: HIT Content-Length: 1256 ```
```

## Протокол HTTP

---

Спецификация протокола HTTP 1.1 вышла в 1999 году, содержала в себе методы GET, POST, PUT, PATCH, OPTIONS, DELETE, идентификация хостов (заголовок `Host` стал обязательным), было добавлено управление кэшированием, согласование содержимого, Digest- и Proxy-авторизация.

### Установка соединения



Процесс начала сеанса TCP называется тройное рукопожатие, состоит из трех *hop*'ов:

- **SYN** – клиент выбирает случайное число  $X$  и отправляет SYN-пакет, который может также содержать дополнительные флаги TCP и значения опций.
- **SYN ACK** – сервер выбирает свое собственное случайное число  $Y$ , прибавляет 1 к значению  $X$ , добавляет свои флаги и опции и отправляет ответ.
- **ACK** – клиент прибавляет 1 к значениям  $X$  и  $Y$  и завершает handshake, отправляя ACK-пакет.

После того как процесс рукопожатия совершен, может быть начат обмен данными. Клиент может отправить пакет данных сразу после ACK-пакета, сервер должен дожидаться ACK-пакета, чтобы начать отправлять данные. Этот процесс происходит при каждом TCP-соединении и представляет серьезную сложность плане производительности сайтов. Ведь каждое новое соединение означает некоторую сетевую задержку.

Для решения этой проблемы используют заголовок `Connection: Keep-Alive`, который не закрывает уже установленное HTTP соединение, а использует его для следующих запросов.

## Методы

В HTTP метод определяет какую операцию нужно выполнить над запрашиваемым ресурсом.

### GET

Используется для запроса содержимого указанного ресурса. Согласно стандарту HTTP, запросы типа GET считаются идемпотентными — многократное повторение одного и того же запроса GET должно приводить к одинаковым результатам (при условии, что сам ресурс не изменился за время между запросами). Это позволяет кэшировать ответы на запросы GET. Кроме обычного метода GET, различают ещё условный GET и частичный GET. Условные запросы GET содержат заголовки `If-Modified-Since`, `If-Match`, `If-Range` и т.д. Частичные GET

содержат в запросе `Range` с указанием диапазона получаемых данных.

## POST

Применяется для передачи пользовательских данных заданному ресурсу. Например, в блогах посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего на формы вызывается submit они передаются серверу методом POST и он помещает их на страницу. При этом передаваемые данные включаются в тело запроса. Аналогично с помощью метода POST обычно загружаются файлы. В отличие от метода GET, метод POST не считается идемпотентным, то есть многократное повторение одних и тех же запросов POST может возвращать разные результаты (например, после каждой отправки комментария будет появляться одна копия этого комментария). При результатах выполнения 200 OK в тело ответа следует включить сообщение об итоге выполнения запроса. Если был создан ресурс, то серверу следует вернуть ответ 201 Created с указанием URI нового ресурса в заголовке `Location`. Сообщение ответа сервера на выполнение метода POST не кэшируется.

## PUT

Применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существовало ресурса, то сервер создаёт его и возвращает статус 201 Created. Если же был изменён ресурс, то сервер возвращает 200 OK или 204 No Content. Фундаментальное различие методов POST и PUT заключается в понимании предназначений URI ресурсов. Метод POST предполагает, что по указанному URI будет производиться обработка передаваемого клиентом содержимого. Используя PUT, клиент предполагает, что загружаемое содержимое соответствует находящемуся по данному URI ресурсу. Сообщения ответов сервера на метод PUT не кэшируются.

## PATCH

Аналогично PUT, но применяется только к фрагменту ресурса.

## DELETE

Удаляет указанный ресурс.

## OPTIONS

Используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса. Предполагается, что запрос клиента может содержать тело сообщения для указания интересующих его сведений. Формат тела и порядок работы с ним в настоящий момент не определён. Сервер пока должен его игнорировать. Аналогичная ситуация и с телом в ответе сервера. Для того чтобы узнать возможности всего сервера, клиент должен указать в URI звёздочку `*`. Запросы `OPTIONS * HTTP/1.1` могут также применяться для проверки работоспособности сервера и тестирования на предмет поддержки сервером протокола HTTP версии 1.1. Результат выполнения этого метода не кэшируется.

## TRACE

Возвращает полученный запрос так, что клиент может увидеть, что промежуточные сервера добавляют или изменяют в запросе.

## HEAD

Аналогичен методу GET, за исключением того, что в ответе сервера отсутствует тело. Запрос HEAD обычно применяется для извлечения метаданных, проверки наличия ресурса или, чтобы узнать, не изменился ли он с момента последнего обращения. Заголовки ответа могут кэшироваться. При несовпадении метаданных ресурса с соответствующей информацией в кэше копия ресурса помечается как устаревшая.

## Заголовки

Заголовки HTTP — это строки в HTTP-сообщении, содержащие разделённую двоеточием пару `<name>: <value>`. Все заголовки разделяются на четыре основных группы, при этом некоторые могут относиться к нескольким группам:

1. General Headers — должны включаться в любое сообщение клиента и сервера.
2. Request Headers — используются только в запросах клиента.
3. Response Headers — только для ответов от сервера.
4. Entity Headers — сопровождают каждую сущность сообщения.

Название параметра должно состоять минимум из одного печатного символа, регистр символов в названиях не имеет значения, заголовки с неизвестными именами игнорируются, после названия сразу должен следовать символ двоеточия. Значение может содержать любые символы ASCII кроме перевода строки и возврата каретки. Пробельные символы в начале и конце значения обрезаются. Последовательность нескольких пробельных символов внутри значения может восприниматься как один пробел. Регистр символов также не имеет значения (если иное поведение не предусмотрено форматом поля). Перечислим некоторые самые распространенные заголовки:

Заголовок	Описание	Тип заголовка	Пример
<code>Host</code>	Доменное имя и порт хоста запрашиваемого ресурса. Необходимо для поддержки виртуального хостинга на серверах	General Headers	<code>Host: example.com</code>
<code>Accept</code>	Список допустимых форматов ресурса	Request Headers	<code>Accept: application/json</code>
<code>Accept-Language</code>	Список допустимых языков с весами	Request Headers	<code>Accept-Language: ru-Ru,en;q=0.5</code>
<code>Accept-Charset</code>	Список допустимых кодировок символов	Request Headers	<code>Accept-Charset: UTF-8</code>
<code>Accept-Encoding</code>	Список допустимых кодировок сообщения	Request Headers	<code>Accept-Encoding: gzip</code>
<code>If-Match</code>	Выполнять метод, если сущность изменилась с указанного момента	Request Headers	<code>If-Match: "737060cd"</code>
<code>If-Modified-Since</code>	Выполнять метод если сущность изменилась с указанного момента	Request Headers	<code>If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT</code>
<code>Content-Disposition</code>	Способ распределения сущностей в сообщении при передаче нескольких фрагментов	Request / Response Headers	<code>Content-Disposition: form-data; name="AttachedFile1"; filename="photo-1.jpg"</code>
<code>Authorization</code>	Данные авторизации	Request Headers	

`Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==` || `Cache-Control` | Основные директивы для управления кэшированием | `General Headers` |

`Cache-Control: <no-cache | no-store | max-age=3600>` || `Content-Encoding` | Информация о кодировке возвращаемого ресурса | `Entity Headers` | `Content-Encoding: gzip` || `Content-Length` | Длина сообщения в байтах | `Entity Headers` | `Content-Length: 1348` || `ETag` | Уникальный идентификатор версии сущности, используемый при кэшировании | `Response Headers` |

`ETag: "56d-9989200-1132c580"` || `Upgrade` | Список предлагаемых клиентом протоколов | `Request Headers` | `Upgrade: websocket` || `Referer` | Полный или относительный URI ресурса, с которого клиент сделал текущий запрос | `Request Headers` |

`Referer: http://en.wikipedia.org/wiki/Main_Page` || `Origin` | Показывает откуда будет производиться загрузка. Не включает в себя какую-либо информацию о пути, содержит в себе лишь имя сервера | `Request Headers` | `Origin: https://developer.mozilla.org` || `User-Agent` | Указывает программное обеспечение клиента и его характеристики | `Request Headers` |

`User-Agent: Mozilla/5.0 (X11; Linux i686; rv:2.0.1) Gecko/20100101 Firefox/4.0.1` || `Allow` | Список поддерживаемых методов | `Entity Headers` | `Allow: OPTIONS, GET, HEAD` || `Location` | URI по которому клиенту следует перейти или URI созданного ресурса. | `Response Headers` |

`Location: http://example.com/about.html#contacts` || `Vary` | Определяет, как сопоставить будущие заголовки запроса, чтобы решить, можно ли использовать кешированный ответ, а не запрашивать новый с исходного сервера. Он используется сервером для указания того, какие заголовки он использовал при выборе представления ресурса в алгоритме согласования контента | `Response Headers` | `Vary: User-Agent` || `Access-Control-Allow-Origin` | Показывает, может ли ответ сервера быть доступен коду, отправляющему запрос с данного источника origin (CORS) | `Response Headers` |

`Access-Control-Allow-Origin: https://developer.mozilla.org` || `Access-Control-Allow-Methods` | Определяет метод или методы доступа к ресурсам (CORS) | `Response Headers` |

`Access-Control-Allow-Methods: POST, GET, OPTIONS` || `Access-Control-Allow-Credentials` | Определяет, можно ли вызываемому коду показывать cookies или заголовки авторизации (CORS) | `Response Headers` | `Access-Control-Allow-Credentials: true` |

## URL

Структура URL (Uniform Resource Locator) очень проста и состоит из следующих компонентов:

```
<schema>://<login>:<password>@<host>:<port>/<path>?<parameters>#<anchor>
```

Например:

```
https://program:test@example.com:8080/api/v1/users?name=Alex&surname=Romanov#Detailed_Info
```

- login – имя пользователя, используемое для доступа к ресурсу;
- password – пароль указанного пользователя;
- host – полностью прописанное доменное имя хоста в системе DNS или IP-адрес;
- port – порт хоста для подключения (для http / https протокола 80 или 443 порт обычно опускается);
- path – уточняющая информация о месте нахождения ресурса;
- parameters – строка запроса с передаваемыми на сервер (методом GET) параметрами. Начинается с символа ?, разделитель параметров — знак &;

- **anchor** – ссылка внутри HTML страницы, эта информация не передается на сервер.

## Статусы

Код состояния информирует клиента о результатах выполнения запроса и определяет его дальнейшее поведение. Каждый код представляется целым трехзначным числом. Первая цифра указывает на класс состояния, последующие – порядковый номер состояния. За кодом ответа обычно следует краткое описание на английском языке.

### 1xx: Informational

В этот класс выделены коды, информирующие о процессе передачи. Клиент должен быть готов принять этот класс сообщений как обычный ответ, но серверу отправлять что-либо не нужно. Сами сообщения от сервера содержат только стартовую строку ответа и, если требуется, несколько специфичных для ответа полей заголовка.

### 2xx: Success

Сообщения данного класса информируют о случаях успешного принятия и обработки запроса клиента. В зависимости от статуса сервер может ещё передать заголовки и тело сообщения.

- **200 OK** — успешный запрос. Если клиентом были запрошены какие-либо данные, то они находятся в заголовке и/или теле сообщения.
- **201 Created** — в результате успешного выполнения запроса был создан новый ресурс. Сервер может указать адреса (их может быть несколько) созданного ресурса в теле ответа, при этом предпочтительный адрес указывается в заголовке Location. Серверу рекомендуется указывать в теле ответа характеристики созданного ресурса и его адреса, формат тела ответа определяется заголовком Content-Type. При обработке запроса, новый ресурс должен быть создан до отправки ответа клиенту, иначе следует использовать ответ с кодом 202.
- **202 Accepted** — запрос был принят на обработку, но она не завершена. Клиенту необязательно дожидаться окончательной передачи сообщения, так как может быть начат очень долгий процесс.
- **204 No Content** — сервер успешно обработал запрос, но в ответе были переданы только заголовки без тела сообщения. Клиент не должен обновлять содержимое документа, но может применить к нему полученные метаданные.

### 3xx: Redirect

Коды этого класса сообщают клиенту, что для успешного выполнения операции необходимо сделать другой запрос, как правило, по другому URI. Из данного класса пять кодов 301, 302, 303, 305 и 307 относятся непосредственно к перенаправлениям.

Адрес, по которому клиенту следует произвести запрос, сервер указывает в заголовке `Location`. При этом допускается использование фрагментов в целевом URI. Клиент может производить перенаправление без запроса пользователя только, если второй ресурс будет запрашиваться методом GET или HEAD. При всех перенаправлениях, если метод запроса был

не HEAD, то в тело ответа следует включить короткое гипертекстовое сообщение с целевым адресом, чтобы в случае ошибки пользователь смог сам произвести переход.

- **300 Multiple Choices** — по указанному URI существует несколько вариантов предоставления ресурса по типу MIME, по языку или по другим характеристикам. Сервер передаёт с сообщением список альтернатив, давая возможность сделать выбор клиенту автоматически или пользователю.
- **301 Moved Permanently** — запрошенный документ был окончательно перенесен на новый URI, указанный в поле `Location` заголовка. Можно использовать кэширование.
- **302 Moved Temporarily** — запрошенный документ временно доступен по другому URI, указанному в заголовке в поле `Location`. Можно использовать кэширование только если указан заголовок `Cache-Control` или `Expires`.
- **303 See Other** — документ по запрошенному URI нужно запросить по адресу в поле `Location` заголовка с использованием метода GET несмотря даже на то, что первый запрашивался иным методом. Этот код был введён вместе с 307-ым для избежания неоднозначности, чтобы сервер был уверен, что следующий ресурс будет запрошен методом GET. Например, на веб-странице есть поле ввода текста для быстрого перехода и поиска. После ввода данных браузер делает запрос методом POST, включая в тело сообщения введённый текст. Если обнаружен документ с введённым названием, то сервер отвечает кодом 303, указав в заголовке `Location` его постоянный адрес. Тогда браузер гарантировано его запросит методом GET для получения содержимого. В противном случае сервер просто вернёт клиенту страницу с результатами поиска.
- **304 Not Modified** — сервер возвращает такой код, если клиент запросил документ методом GET, использовал заголовок `If-Modified-Since` или `If-None-Match` и документ не изменился с указанного момента. При этом сообщение сервера не должно содержать тела.
- **307 Temporary Redirect** — запрашиваемый ресурс на короткое время доступен по другому URI, указанный в поле `Location` заголовка. Метод запроса (GET/POST) менять не разрешается. Например, POST запрос должен быть отправлен по новому URI тем же методом POST. Этот код был введён вместе с 303 вместо 302-го для избежания неоднозначности. Можно использовать кэширование только если указан заголовок `Cache-Control` или `Expires`.

## 4xx: Client Error

Этот класс кодов предназначен для указания ошибок со стороны клиента.

- **400 Bad Request** — сервер обнаружил в запросе клиента синтаксическую ошибку.
- **401 Unauthorized** — для доступа к запрашиваемому ресурсу требуется аутентификация.
- **402 Payment Required** — предполагается использовать в будущем. В настоящий момент не используется. Этот код предусмотрен для платных пользовательских сервисов.
- **403 Forbidden** — сервер понял запрос, но он отказывается его выполнять из-за ограничений в доступе для клиента к указанному ресурсу. Если для доступа к ресурсу требуется аутентификация средствами HTTP, то сервер вернёт ответ 401, или 407 при использовании прокси. В противном случае ограничения были заданы администратором сервера или разработчиком веб-приложения и могут быть любыми в зависимости от



возможностей используемого программного обеспечения.

- **404 Not Found** — сервер понял запрос, но не нашёл соответствующего ресурса по указанному URI. Если серверу известно, что по этому адресу был документ, то ему желательно использовать код 410. Ответ 404 может использоваться вместо 403, если требуется тщательно скрыть от посторонних глаз определённые ресурсы.
- **405 Method Not Allowed** — указанный клиентом метод нельзя применить к текущему ресурсу. В ответе сервер должен указать доступные методы в заголовке Allow, разделив их запятой.
- **406 Not Acceptable** — запрошенный URI не может удовлетворить переданным в заголовке характеристикам. Если метод был не HEAD, то сервер должен вернуть список допустимых характеристик для данного ресурса.
- **409 Conflict** — запрос не может быть выполнен из-за конфликтного обращения к ресурсу. Такое возможно, например, когда два клиента пытаются изменить ресурс с помощью метода PUT.
- **415 Unsupported Media Type** — по каким-то причинам сервер отказывается работать с указанным типом данных при данном методе.
- **418 I'm a teapot** — Этот код был введен в 1998 году как одна из традиционных первоапрельских шуток IETF в RFC 2324, Hyper Text Coffee Pot Control Protocol.
- **422 Unprocessable Entity** — сервер успешно принял запрос, может работать с указанным видом данных, но, например, в теле запроса содержится XML-документ, который имеет верный синтаксис, но имеется какая-то логическая ошибка, из-за которой невозможно произвести операцию над ресурсом.
- **429 Too Many Requests** — клиент попытался отправить слишком много запросов за короткое время, что может указывать, например, на попытку DDoS-атаки. Может сопровождаться заголовком `Retry-After`, указывающим, через какое время можно повторить запрос.

## 5xx: Server Error

В этот класс кодов предназначен для указания ошибок выполнения операций по вине сервера.

- **500 Internal Server Error** — любая внутренняя ошибка сервера, которая не входит в рамки остальных ошибок класса.
- **501 Not Implemented** — сервер не поддерживает возможностей, необходимых для обработки запроса. Типичный ответ для случаев, когда сервер не понимает указанный в запросе метод. Если же метод серверу известен, но он не применим к данному ресурсу, то нужно вернуть ответ 405.
- **502 Bad Gateway** — сервер, выступая в роли шлюза или прокси-сервера, получил недействительное ответное сообщение от вышестоящего сервера.
- **503 Service Unavailable** — сервер временно не имеет возможности обрабатывать запросы по техническим причинам (обслуживание, перегрузка и прочее). В поле `Retry-After` заголовка сервер может указать время, через которое клиенту рекомендуется повторить запрос.
- **504 Gateway Time out** — сервер в роли шлюза или прокси-сервера не дождался ответа от

вышестоящего сервера для завершения текущего запроса.

## Согласование содержимого

Content Negotiation (согласование содержимого) — механизм автоматического определения необходимого ресурса при наличии нескольких разнотипных версий документа. Субъектами согласования могут быть не только ресурсы сервера, но и возвращаемые страницы с сообщениями об ошибках. При наличии нескольких версий ресурса сервер может анализировать заголовки запроса клиента, чтобы выдать, по его мнению, наиболее подходящую. В основном анализируются заголовки `Accept`, `Accept-Charset`, `Accept-Encoding`, `Accept-Languages` и `User-Agent`. Серверу желательно включать в ответ заголовок `Vary` с указанием параметров, по которым различается содержимое по запрашиваемому URI. Управляемое сервером согласование имеет несколько недостатков:

- Сервер только предполагает, какой вариант наиболее предпочтителен для конечного пользователя, но не может знать точно, что именно нужно в данный момент (например, версия на русском языке или английском).
- Заголовков группы `Accept` передаётся много, а ресурсов с несколькими вариантами — мало. Из-за этого оборудование испытывает избыточную нагрузку.
- Общему кэшу создаётся ограничение возможности выдавать один и тот же ответ на идентичные запросы от разных пользователей.
- Передача заголовков `Accept` также может раскрывать некоторые сведения о его предпочтениях, таких как используемые языки, браузер, кодировка.

## Cookie

Cookie — это небольшая порция текстовой информации, которую сервер передает браузеру. Браузер будет хранить эту информацию и передавать её серверу с каждым запросом как часть HTTP заголовка (заголовок `Cookie`).

Cookies используются веб-серверами для различения пользователей и хранения данных о них. К примеру, если вход на сайт осуществляется при помощи cookies, то после ввода пользователем своих данных на странице входа, cookie позволяют серверу запомнить, что пользователь уже идентифицирован, и ему разрешён доступ к соответствующим услугам и операциям. Многие сайты также используют cookies для сохранения настроек пользователя. Эти настройки могут использоваться для персонализации, которая включает в себя выбор оформления и функциональности. Cookie могут использоваться сервером для опознания ранее аутентифицированных пользователей. Это происходит следующим образом:

- Пользователь вводит имя пользователя и пароль в текстовых полях страницы входа и отправляет их на сервер.
- Сервер получает имя пользователя и пароль, проверяет их и, при их правильности, отправляет страницу успешного входа, прикрепив cookie с неким идентификатором сессии (через заголовок ответа `Set-Cookie`). Эта cookie может быть действительна не только для текущей сессии браузера, но может быть настроена и на длительное хранение.

- Каждый раз, когда пользователь запрашивает страницу с сервера, браузер автоматически отправляет cookie с идентификатором сессии серверу. Сервер проверяет идентификатор по своей базе идентификаторов и, при наличии в базе такого идентификатора, "узнаёт" пользователя.

Установка cookies выполняется через заголовок `Set-Cookie: <name>=<value>`. Строка `Set-Cookie` отправляется лишь тогда, когда сервер желает, чтобы браузер сохранил cookie. В этом случае, если cookie поддерживаются браузером и их приём включён, браузер запоминает строку `<name>=<value>` и отправляет её обратно серверу в заголовке Cookie с каждым последующим запросом. Кроме пары имя/значение, cookie может содержать срок действия, путь и доменное имя. В случае, если время жизни не задано, cookies удаляются по окончании сеанса. Время жизни cookies задается в формате ISO date time. Область видимости cookies можно ограничить установкой domain и path – cookies будут видны только для указанного домена и его поддоменов и подмножества документов. Имена cookies нечувствительны к регистру. Ограничения:

- В случае, если cookie принимает новое значение при имеющемся уже в браузере cookie с совпадающими параметрами name, domain и path, то старое значение заменяется новым. В остальных случаях новые значения cookie добавляются к старым.
- Использование expires не гарантирует сохранность cookie в течение заданного периода времени, поскольку клиент может удалить запись из-за нехватки выделенного места или каких-либо других причин.

Клиент (браузер) имеет следующие ограничения для cookies:

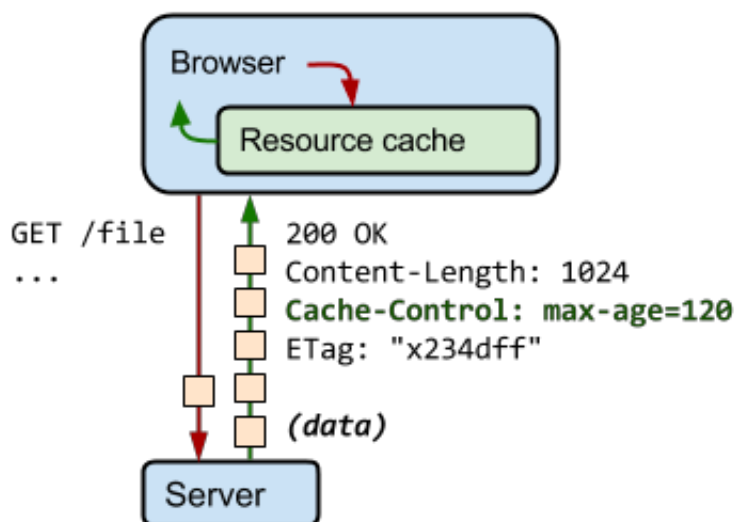
- всего может храниться до 300 значений cookies;
- каждый cookie не может превышать 4Кбайт; с одного сервера или домена может храниться до 20 значений cookie;
- если ограничение 300 или 20 превышает, то удаляется первая по времени запись;
- при превышении лимита объема в 4Кбайт корректность значения cookie страдает – отрезается кусок записи (с начала этой записи) равный превышению объема.

В случае кэширования документов, например, прокси-сервером, заголовок `Set-Cookie` не кэшируется. В случае, если у клиента отключены cookies, обычно важная информация встраивается как параметры URL. Т.к. cookies могут содержать конфиденциальную информацию, злоумышленники могут пытаться украсть их. Наиболее частые сценарии это:

- Если используется незащищенное соединение, то cookies можно просто перехватить с помощью анализатора трафика сети.
- Современные браузеры могут исполнять фрагменты кода, полученные с сервера. Если cookie доступны во время этого исполнения, их содержимое может в той или иной форме оказаться на серверах, которые не должны получать к ним доступ.
- Таким же образом вредоносные скрипты могут изменять значения cookies.
- Cookie могут вызвать противоречия между клиентом и сервером: если пользователь получает cookie, а затем нажимает кнопку "Назад" в браузере, то состояние браузера уже другое по сравнению с моментом получения cookie.

Кэш – промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Соответственно кэширование – это сохранение данных ответа сервера, для более быстрого ответа клиенту. Кеширование используется для снижения сетевого трафика и уменьшения времени ожидания ответа, т.к. данные не вычисляются, а берутся из быстрого хранилища. В web различают три вида кэширования:

- Во всех браузерах есть встроенный HTTP-кеш, управление им выполняется с помощью HTTP-заголовков. Они нужны, чтобы указать браузеру, когда и на какой период нужно кешировать ответ. Когда сервер возвращает запрос, он также отправляет набор HTTP-заголовков, описывающих тип контента (Content-Type), длину (Content-Length), команды для работы с кешем, маркер подтверждения и т. д.



## Как работает кэш

## ETag

Сервер отправляет маркер подтверждения в HTTP-заголовке `ETag: "x234dff"`. С помощью маркера подтверждения можно проверить, изменился ли ресурс. Допустим, после нашего первого вызова прошло 120 секунд, и браузер начал новый запрос к тому же ресурсу. Сначала браузер проверяет локальный кеш и находит предыдущий ответ. Но его использовать нельзя, потому что срок его действия уже истек. Теперь браузер мог бы просто отправить новый запрос и получить ещё один полный ответ. Однако это неэффективно, потому что ресурс не изменился, и не имеет смысла снова скачивать байты, которые уже есть в кеше. Чтобы избежать этой проблемы, нужно использовать маркеры подтверждения, указанные в заголовках ETag. Сервер создает и возвращает произвольный маркер. Обычно это хеш или другая идентификационная метка файла. Клиент может не знать, как она создается, ему просто нужно отправить её на сервер при следующем запросе. Если метка осталась прежней, то ресурс не изменился и скачивать его не надо. В примере выше клиент автоматически отправляет маркер ETag в HTTP-заголовке `If-None-Match: "x234dff"`. Сервер проверяет совпадение маркера с нужным ресурсом, и если тот не изменился, отправляет ответ 304 Not Modified. Это значит, что кешированный ответ остался прежним, и его можно снова использовать в течение следующих 120 секунд и скачивать ресурс ещё раз не нужно. Это уменьшает время загрузки страницы и экономит пропускную способность канала.

## Cache-Control

Правила кеширования ресурса можно указать с помощью заголовка `Cache-Control`. Директивы `Cache-Control` определяют, где, как и на какое время может быть кеширован ресурс.

### No-Cache и No-Store

Директива no-cache означает, что при повторном запросе к тому же URL ответ можно использовать только после проверки изменений. Таким образом, если указан соответствующий маркер подтверждения `ETag`, будет выполнена повторная проверка. Однако при отсутствии изменений данные не будут скачиваться ещё раз. Действие директивы no-store намного проще. Она запрещает браузеру и всем промежуточным кешам сохранять какую-либо версию ответа. Директива используется, если в ответе содержится личная или банковская информация. Каждый раз, когда пользователь хочет использовать этот ресурс, на сервер отправляется запрос и ответ полностью скачивается заново.

### Must-revalidate

При использовании директивы must-revalidate кеш обязан проверять статус ресурсов с истёкшим сроком действия.

### Public и Private

Если в ответе содержится директива public, его можно кешировать, даже если с ним связана HTTP-аутентификация и код статуса ответа обычно нельзя сохранять. Эта директива требуется редко, потому что другая информация, заданная для кеширования, например, max-age, показывает, что ответ можно сохранить. Директива private, наоборот, используется для ответов, которые можно сохранить в кеше браузера, но не в промежуточных кешах. Это происходит из-за того, что подобная информация предназначена для одного пользователя.

Например, HTML-страницу с личными данными пользователя можно кешировать в браузере, но не в сетях доставки контента.

## Max-Age

Эта директива указывает период времени в секундах, в течение которого скачанный ответ может быть повторно использован. Отсчет начинается с момента запроса. Например, `max-age=60` означает, что ответ может быть кеширован и использован в течение 60 секунд.

## Last-Modified

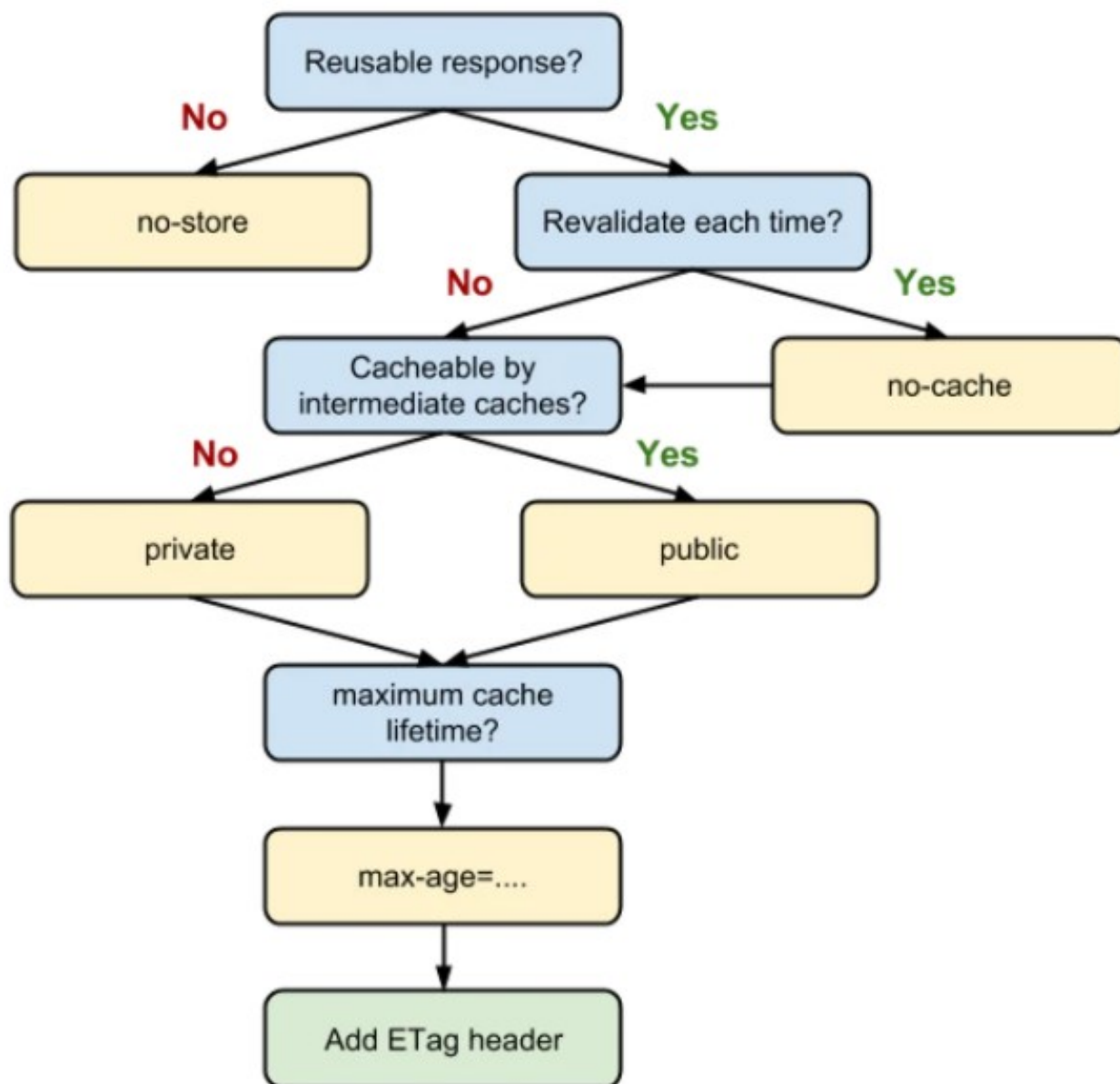
Когда кэш хранит контент, который содержит заголовок `Last-Modified`, он может использовать его для того, чтобы опросить сервер, чтобы узнать, был ли изменен контент со времени его последнего просмотра, с помощью заголовка `If-Modified-Since` в запросе. Если контент не изменился, то сервер отвечает 304 Not Modified.

## Аннулирование и обновление кешированных ответов

Кешированный ответ сохраняется в браузере до тех пор, пока не истечет срок его действия или пока он не будет удален, например при очистке кеша браузера. Таким образом, во время изменения страницы не все пользователи работают с одинаковыми версиями файла: те, кто загрузил ресурс недавно, видят новую версию, а те, кто кешировал предыдущую (но ещё действительную) копию, - старую. Для решения этой проблемы добавляют к имени файла некоторую идентификационную метку или номер версии, например: `style.v1.2.3.css`.

Советы по использованию:

- Не используйте POST, если это не является целесообразным. Ответы POST-методом не хранятся большинством кэшей.
- Если вы отправляете информацию в строке запроса (через GET), кэш может хранить эту информацию на будущее. Не вставляйте информацию, специфичную для пользователя, в URL-адрес, если сгенерированный контент не является полностью уникальным для этого пользователя.
- При использовании SSL промежуточные сервера не смогут хранить кэш, т.к. запрос для них будет приходить в зашифрованном виде.



## Сессия

Протокол HTTP не был рассчитан на хранение состояния между запросами, но для серверных приложений есть большая необходимость хранить на своей стороне какие-либо данные о клиенте. Сессия – некоторый объект, специфичный для каждого пользователя (к пользователю привязан уникальный ID), хранящий в себе некоторую информацию. Сами данные хранятся на сервере, а пользователю отправляется этот идентификатор.

На стороне клиента идентификатор хранится в cookies или является частью url. Время жизни сессии обычно делается небольшим, обычно около 30 минут, если пользователь не активен.

В распределенных системах с сессиями все обстоит сложнее, т.к. нет гарантии, что запрос придет на тот же сервер, который обрабатывал предыдущий запрос. Решается это либо хранением информации в некотором хранилище, либо созданием распределенных сессий. Второй вариант сложнее, т.к. крайне сложно отследить если что-то пошло не так:

информация о сессии не успела прийти на сервер, либо уже содержит устаревшие данные.

В сессиях не стоит держать большие объемы данных. Если есть открытая специфичная информация о пользователе, то лучше держать её в cookies. В случае неприменимости этого варианта стоит использовать какое-либо хранилище данных, а ID пользователя хранить в cookies.

## Stateful и Stateless

Для простоты объяснения можно привести две функции:

```
```java // Stateless class AtomicIncrementStateless { public synchronized int increment(int number) {  
return number + 1; } }
```

```
// Stateful class AtomicIncrementStateful { private int number = 0;
```

```
    public synchronized int increment() {  
        return ++number;  
    }  
}
```

```
}```
```

Stateless протокол представляет каждый запрос как отдельную транзакцию, не привязанную к предыдущим запросам. HTTP по своей природе является stateless протоколом, т.е. он не хранит никакую информацию о пользователе между запросами. В случае, если требуется сохранение информации, используются cookie или сессия. Но если для хранения информации используется сессия, то получаем stateful, т.к. сервер хранит состояние клиента.

Так как каждый запрос атомарный и каждый новый запрос может обрабатывать любой сервер, stateless сервера очень просто масштабировать. Если клиентский запрос падает в процессе обработки, нет необходимости восстанавливать предыдущее состояние. Но при этом мы получаем необходимость передавать дополнительную лишнюю информацию. Часто как компромисс между плюсами stateless и необходимостью передавать большой объем лишней информации между клиентом и сервером, используется некоторый идентификатор, хранящийся в cookie. По этому идентификатору в базе лежит вся нужная информация о пользователе. При каждом запросе значение cookie передается на обрабатывающий запрос сервер и он из базы получает всю нужную информацию.

## WebSocket

WebSocket — протокол двунаправленной связи (может передавать и принимать одновременно) поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и сервером в режиме реального времени. Для установления соединения WebSocket клиент и сервер используют протокол, похожий на HTTP. Клиент формирует HTTP-запрос с заголовками на изменение протокола `Connection: Upgrade` и указанием на какой именно протокол нужна замена `Upgrade: websocket`. В случае успеха в ответ клиент получает ответ со статусом 101 Switching Protocols с заголовками `Upgrade: websocket` и



```
Connection: Upgrade.
```

```
```shell docker run -p 8080:8080 jmalloc/echo-server
```

## заходим в браузер, открываем console в Developer Tools

```
var socket = new WebSocket("ws://localhost:8080") socket.onmessage = (event) =>
console.log(Response from server: ${event.data}); socket.onopen = (event) => {
console.log("Connected") socket.send("Hello, world") } socket.onclose = (event) =>
console.error(Connection close, code=${event.code}, reason=${event.reason}) socket.onerror =
(error) => console.error(Error: ${error.message})
```

## В ответ вернется

```
```
```

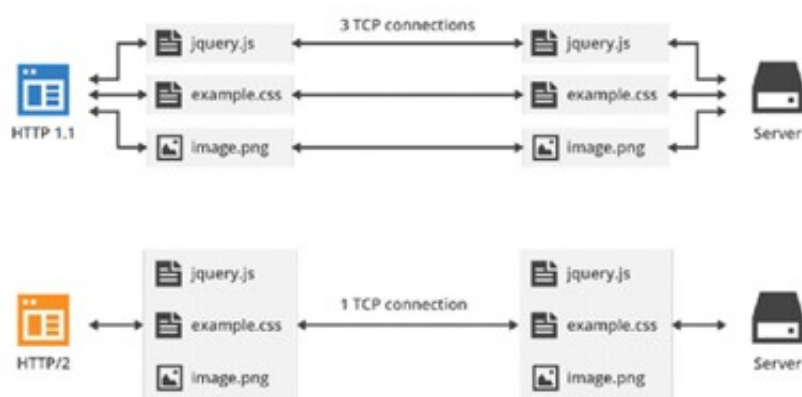
| General                      |  |
|------------------------------|--|
| Request URL:                 | ws://localhost:8080/   |
| Request Method:              | GET  |
| Status Code:                 | 🟢 101 Switching Protocols  |
| Response Headers View source |  |
| Connection:                  | Upgrade  |
| Sec-WebSocket-Accept:        | DA4+UEiYSDbwe7Ma1EL9/AYD4ec=   |
| Upgrade:                     | websocket  |
| Request Headers View source  |  |
| Accept-Encoding:             | gzip, deflate, br  |
| Accept-Language:             | ru-RU, ru;q=0.9, en-US;q=0.8, en;q=0.7   |
| Cache-Control:               | no-cache   |
| Connection:                  | Upgrade  |
| Host:                        | localhost:8080   |
| Origin:                      | chrome://new-tab-page  |
| Pragma:                      | no-cache   |
| Sec-WebSocket-Extensions:    | permessage-deflate; client_max_window_bits   |
| Sec-WebSocket-Key:           | IK0Re3yW6NHojVv5opzk0w==   |
| Sec-WebSocket-Version:       | 13   |
| Upgrade:                     | websocket  |
| User-Agent:                  | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.63 Safari/537.36 |

## HTTP/2

Основными целями для HTTP/2 являются снижение задержки за счет обеспечения полного мультиплексирования запросов и ответов, минимизации накладных расходов протокола посредством эффективного сжатия полей заголовков HTTP и добавления поддержки приоритизации запросов и отправки на сервер. Для реализации этих требований существует большое число других улучшений протокола, таких как новые механизмы управления потоками, обработки ошибок и обновления, но это наиболее важные функции, которые каждый веб-разработчик должен понимать и использовать в своих приложениях.

HTTP/2 никак не изменяет семантику приложения HTTP. Все основные понятия, такие как методы HTTP, коды состояния, URI и поля заголовка, остаются на месте. Вместо этого HTTP/2 изменяет способ форматирования (кадрирования) данных и их транспортировки между клиентом и сервером, которые управляют всем процессом, и скрывает всю сложность наших приложений в новом слое кадрирования. В результате все существующие приложения могут быть доставлены без изменений.

### Multiplexing



Разработка протокола HTTP/2 основывалась на другом протоколе SPDY, который был разработан Google.

## Бинарный протокол

HTTP/2 пытается решить проблему выросшей задержки, существовавшую в HTTP/1.x, переходом на бинарный формат. Бинарные сообщения быстрее разбираются автоматически, но, в отличие от HTTP/1.x, не удобны для чтения человеком. Основные составляющие HTTP/2 — фреймы (Frames) и потоки (Streams).

## Фреймы и потоки

Сейчас HTTP-сообщения состоят из одного или более фреймов. Для метаданных используется фрейм HEADERS, для основных данных — фрейм DATA, и ещё существуют другие типы фреймов (RSTSTREAM, SETTINGS, PRIORITY и т.д.), с которыми можно ознакомиться в спецификации HTTP/2. Каждый запрос и ответ HTTP/2 получает уникальный идентификатор потока и разделяется на фреймы. Фреймы представляют собой просто бинарные части данных. Коллекция фреймов называется потоком (Stream). Каждый фрейм содержит идентификатор потока, показывающий, к какому потоку он принадлежит, а также каждый фрейм содержит общий заголовок. Также, помимо того, что идентификатор потока уникален, стоит упомянуть, что каждый клиентский запрос использует нечётные id, а ответ от сервера — чётные. Помимо HEADERS и DATA, стоит упомянуть ещё и RSTSTREAM — специальный тип фреймов, использующийся для прерывания потоков. Клиент может послать этот фрейм серверу, сигнализируя, что данный поток ему больше не нужен. В HTTP/1.1 единственным способом заставить сервер перестать посылать ответы было закрытие соединения, что увеличивало время задержки, ведь нужно было открыть новое соединение для любых дальнейших запросов. А в HTTP/2 клиент может

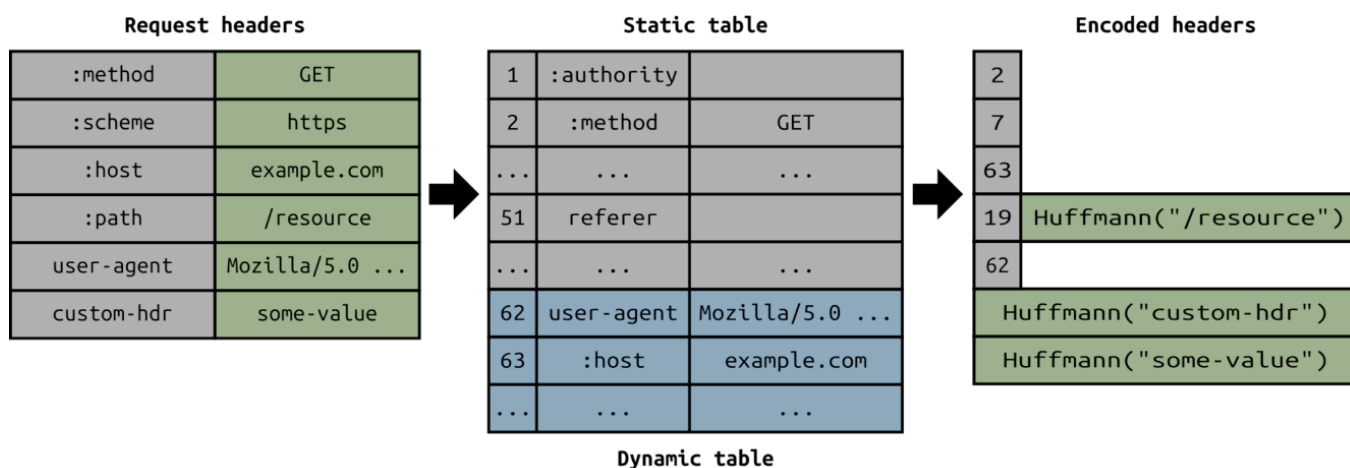
отправить RST\_STREAM и перестать получать определённый поток. При этом соединение останется открытым, что позволяет работать остальным потокам.

## Мультиплексирование

Поскольку HTTP/2 является бинарным протоколом, использующим для запросов и ответов фреймы и потоки, как было упомянуто выше, все потоки посылаются в едином TCP-соединении, без создания дополнительных. Сервер, в свою очередь, отвечает аналогичным асинхронным образом. Это значит, что ответ не имеет порядка, и клиент использует идентификатор потока, чтобы понять, к какому потоку принадлежит тот или иной пакет. Это решает проблему блокировки начала очереди (head-of-line blocking) — клиенту не придётся простаивать, ожидая обработки длинного запроса, ведь во время ожидания могут обрабатываться остальные запросы.

## Сжатие заголовков методом HPACK

Т.к. часто к серверу происходят обращения из-под одного и того же клиента, посылается огромное количество одинаковых данных, а если при этом передаются еще и cookies, то объем данных разрастается еще больше, что снижает пропускную способность сети и увеличивает время задержки. Чтобы решить эту проблему, в HTTP/2 появилось сжатие заголовков.



В отличие от запросов и ответов, заголовки не сжимаются в gzip или подобные форматы. Для сжатия используется иной механизм — литеральные значения сжимаются по алгоритму Хаффмана, а клиент и сервер поддерживают единую таблицу заголовков. Повторяющиеся заголовки (например, user agent) опускаются при повторных запросах и ссылаются на их позицию в таблице заголовков.

## Server Push

Сервер, зная, что клиент собирается запросить определённый ресурс, может отправить его, не дожидаясь запроса. Вот, например, когда это будет полезно: браузер загружает веб-страницу, он разбирает её и находит, какой ещё контент необходимо загрузить с сервера, а затем отправляет соответствующие запросы. Server push позволяет серверу снизить количество дополнительных запросов. Если он знает, что клиент собирается запросить данные, он сразу их посылает. Сервер отправляет специальный фрейм `PUSH_PROMISE`,

извещая клиента, что этого ресурса будет отправлен. Фрейм `PUSH_PROMISE` связан с потоком, который вызвал отправку push-а, и содержит идентификатор потока, по которому сервер отправит нужный ресурс.

## Приоритизация запросов

Клиент может назначить приоритет потоку, добавив информацию о приоритете во фрейм `HEADERS`, которым открывается поток. В любое другое время клиент может отправить фрейм `PRIORITY`, меняющий приоритет потока. Если никакой информации о приоритете не указано, сервер обрабатывает запрос асинхронно, т.е. без какого-либо порядка. Если приоритет назначен, то, на основе информации о приоритетах, сервер принимает решение, сколько ресурсов выделяется для обработки того или иного потока.

## Асинхронный обмен на базе HTTP

Есть ряд ситуаций, когда требуется на базе синхронного протокола реализовать некоторые асинхронные или долгоживущие операции. Для этого используются следующие решения:

- Опрос сервера – клиент отправляет на сервер долгий запрос, сервер принимает его в обработку, отдает клиенту некоторый ID, по которому клиент раз 3-5 секунд приходит к серверу и узнает статус. Если запрос выполнен, то данные отдаются в ответ, иначе какое-нибудь сообщение или статус код, отличный от успешного (т.е. не 200 OK, а например 204 No Content).
- Webhook (callback) – решаемая задача аналогична п. 1, но используется при межсерверном взаимодействии, клиент при запросе передает свой endpoint, куда сервер выполнит запрос с данными после окончания выполнения операции.
- WebSocket – надстройка над протоколом HTTP, при соединении передается запрос с заголовком `Connection: upgrade`, и между клиентом и сервером устанавливается “канал”, по которому можно передавать сообщения в обе стороны. Используется, например, для чатов или в системах, где важно обновлять данные в режиме реального времени.
- Server Sent Events – клиент делает запрос к серверу, открывается постоянное соединение (`Connection: Keep-Alive`), сервер отвечает с заголовком `Content-Type: text/event-stream` и отправляет данные клиенту. Это соединение однонаправленное, т.е. данные отдает только сервер, клиент выступает в роли слушателя.

## Литература

---

1. Протокол HTTP
2. Внутренние механизмы TCP, влияющие на скорость загрузки: часть 1
3. HTTP Status Codes
4. Согласование контента
5. WebSocket
6. HTTP-кэширование
7. Introduction to HTTP/2

## 8. Истөрпывасуе руковосво по исользованию HTTP/2 Server Push