Бесплатная электронная книга

УЧУСЬ

HTTP

Free unaffiliated eBook created from
Stack Overflow contributors.

#http

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: http

It is an unofficial and free HTTP ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official HTTP.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с HTTP

замечания

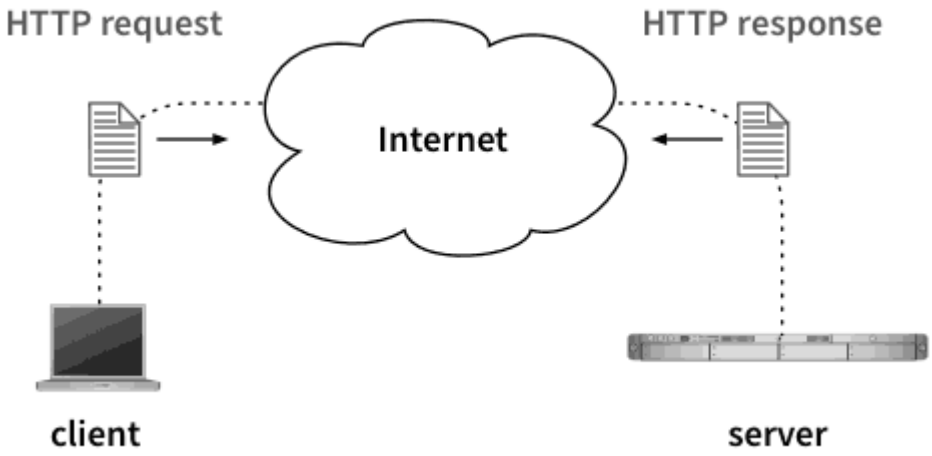
Протокол передачи гипертекста (HTTP) использует модель клиент-запрос / сервер-ответ. HTTP - это протокол без учета состояния, что означает, что сервер не требует сохранения информации или статуса каждого пользователя в течение нескольких запросов. Однако по соображениям производительности и для предотвращения проблем с задержкой соединения TCP могут использоваться такие методы, как постоянные, параллельные или конвейерные соединения.

Версии

Версия	Заметки)	Предполагаемая дата выпуска
HTTP / 0.9	«Как реализовано»	1991-01-01
HTTP / 1.0	Первая версия HTTP / 1.0, последняя версия, которая не попала в RFC	1992-01-01
HTTP / 1.0 r1	Первый официальный RFC для HTTP	1996-05-01
HTTP / 1.1	Усовершенствования в обработке подключений, поддержка виртуальных хостов на основе имен	1997-01-01
HTTP / 1.1 r1	Исправлено использование двуязычного ключевого слова, устранены возможные проблемы с кадрированием сообщений	1999-06-01
HTTP / 1.1 r2	Капитальный ремонт	2014-06-01
HTTP / 2	Первая спецификация для HTTP / 2	2015-05-01

Examples

HTTP-запросы и ответы



HTTP описывает, как HTTP-клиент, такой как веб-браузер, отправляет HTTP-запрос через сеть на HTTP-сервер, который затем отправляет HTTP-ответ клиенту.

HTTP-запрос обычно представляет собой запрос на онлайн-ресурс, такой как веб-страница или изображение, но может также содержать дополнительную информацию, такую как данные, введенные в форме. Ответ HTTP обычно представляет собой представление онлайн-ресурса, например веб-страницы или изображения.

HTTP / 1.0

HTTP / 1.0 был описан в RFC 1945 .

HTTP / 1.0 не имеет некоторых функций, которые сегодня де-факто требуются в Интернете, например заголовка `Host` для виртуальных хостов.

Тем не менее, HTTP-клиенты и серверы по-прежнему заявляют, что они используют HTTP / 1.0, если они имеют неполную реализацию протокола HTTP / 1.1 (например, без кодированной передачи или конвейерной обработки), или совместимость считается более важной, чем производительность (например, при подключении к локальному прокси-серверу серверы).

```
GET / HTTP/1.0
User-Agent: example/1

HTTP/1.0 200 OK
Content-Type: text/plain

Hello
```

HTTP / 1.1

HTTP / 1.1 изначально был указан в 1999 году в RFC 2616 (протокол) и RFC 2617 (аутентификация), но эти документы устарели и не должны использоваться в качестве ссылки:

> Не используйте RFC2616. Удалите его с жестких дисков, закладок и запишите (или ответственно переработайте) все распечатанные копии.
>
> - Марк Ноттингем, председатель рабочей группы по HTTP

Современная спецификация HTTP / 1.1, которая соответствует тому, как HTTP реализуется сегодня, находится в новых RFC 723x:

- RFC 7230: Синтаксис сообщений и маршрутизация
- RFC 7231: семантика и контент
- RFC 7232: условные запросы
- RFC 7233: Запросы диапазона
- RFC 7234: Кэширование
- RFC 7235: аутентификация

HTTP / 1.1 добавил, среди прочих возможностей:

- chunked transfer encoding, что позволяет серверам надежно отправлять ответы неизвестного размера,
- постоянные соединения TCP / IP (которые были нестандартными расширениями в HTTP / 1.0),
- запросы диапазона, используемые для возобновления загрузки,
- кеш-контроль.

HTTP / 1.1 попытался ввести конвейерную обработку, что позволило HTTP-клиентам сократить время ожидания запроса-ответа, отправив сразу несколько запросов, не дожидаясь ответов. К сожалению, эта функция никогда не была правильно реализована в некоторых прокси-серверах, в результате чего конвейерные соединения удаляли или переупорядочивали ответы.

```
GET / HTTP/1.0
User-Agent: example/1
Host: example.com

HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 6
Connection: close

Hello
```

HTTP / 2

HTTP / 2 (RFC 7540) изменил проводной формат HTTP с простого текстового запроса и заголовков ответов на двоичный формат данных, отправленный в фреймах. HTTP / 2 поддерживает сжатие заголовков (HPACK).

Это уменьшило накладные расходы на запросы и позволило одновременно получать

несколько ответов по одному соединению TCP / IP.

Несмотря на значительные изменения в формате данных, HTTP / 2 по-прежнему использует HTTP-заголовки, а запросы и ответы могут быть точно переведены между HTTP / 1.1 и 2.

HTTP / 0.9

Первая версия HTTP, которая появилась, составляет 0,9, которую часто называют « HTTP as Implemented ». Общим описанием 0.9 является «подсекция полного протокола HTTP [т.е. 1.0]». Однако это не позволяет проиллюстрировать несоответствие возможностей между 0.9 и 1.0.

Ни в запросах, ни в ответах в 0,9 заголовках функций. Запросы состоят из одной линии `GET` завершенным CRLF, за которой следует пробел, за которым следует запрошенный URL ресурса. Ожидается, что ответы будут единым HTML-документом. Конец указанного документа отмечен удалением серверной части соединения. Нет никаких возможностей указывать на успех или неудачу операции. Единственное интерактивное свойство - это строка поиска, которая тесно связана с `<isindex>` HTML.

Использование HTTP / 0.9 в настоящее время исключительно редко. Это время от времени видно на встроенных системах в качестве альтернативы tftp .

Прочитайте Начало работы с HTTP онлайн: https://riptutorial.com/ru/http/topic/984/начало-работы-с-http

глава 2: HTTP для API

замечания

HTTP API используют широкий спектр HTTP-глаголов и обычно возвращают ответы JSON или XML.

Examples

Создать ресурс

Не все согласны с тем, что наиболее семантически правильный метод для создания ресурсов. Таким образом, ваш API может принимать запросы `POST` или `PUT` или либо.

Сервер должен ответить на `201 Created` если ресурс был успешно создан. Выберите наиболее подходящий код ошибки, если это не так.

Например, если вы предоставляете API для создания записей сотрудников, запрос / ответ может выглядеть так:

```
POST /employees HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "name": "Charlie Smith",
    "age": 38,
    "job_title": "Software Developer",
    "salary": 54895.00
}
```

```
HTTP/1.1 201 Created
Location: /employees/1/charlie-smith
Content-Type: application/json

{
    "employee": {
        "name": "Charlie Smith",
        "age": 38,
        "job_title": "Software Developer",
        "salary": 54895.00
        "links": [
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "self",
                "method": "GET"
            },
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "delete",
```

```
                "method": "DELETE"
            },
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "edit",
                "method": "PATCH"
            }
        ]
    },
    "links": [
        {
            "uri": "/employees",
            "rel": "create",
            "method": "POST"
        }
    ]
}
```

Включение `links` полей JSON в ответе позволяет клиенту получить доступ к ресурсу, связанному с новым ресурсом и к приложению в целом, без предварительного уведомления об их URI или методах.

Редактировать ресурс

Редактирование или обновление ресурса является общей целью для API. Редактирование может быть достигнуто путем отправки запросов `POST` , `PUT` или `PATCH` на соответствующий ресурс. Хотя `POST` разрешено добавлять данные в существующее представление ресурса, рекомендуется использовать `PUT` или `PATCH` поскольку они передают более явный семантик.

Ваш сервер должен ответить `200 OK` если обновление было выполнено, или `202 Accepted` если он еще не применен. Выберите наиболее подходящий код ошибки, если он не может быть завершен.

Полные обновления

`PUT` имеет семантику замены текущего представления на полезную нагрузку, включенную в запрос. Если полезная нагрузка не имеет тот же тип представления, что и текущее представление ресурса для обновления, сервер может решить, какой подход принять. RFC7231 определяет, что сервер может либо

- Переконфигурируйте целевой ресурс для отражения нового типа носителя
- Преобразуйте представление PUT в формат, совместимый с форматом resouce, прежде чем сохранять его как новое состояние ресурса
- Отклоните запрос с ответом `415 Unsupported Media Type` указывающим, что целевой ресурс ограничен определенным (установленным) типом медиа.

Базовый ресурс, содержащий представление JSON HAL, например ...

```
{
    "name": "Charlie Smith",
    "age": 39,
    "job_title": "Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    }
}
```

... может получить запрос на обновление, подобный этому

```
PUT /users/1234 HTTP/1.1
Host: http://www.acmee.com
Content-Type: "application/json; charset=utf-8"
Content-Length: 85
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer"
}
```

Теперь сервер может заменить состояние ресурса данным объектом запроса, а также изменить тип контента из `application/hal+json` в `application/json` или преобразовать полезную нагрузку JSON в представление JSON HAL, а затем заменить содержимое ресурса с преобразованным или отклонить запрос на обновление из-за неприменимого типа носителя с ответом `415 Unsupported Media Type` .

Существует разница между заменой содержимого напрямую или первым преобразованием представления в модель определенного представления, а затем заменой существующего содержимого на преобразованный. Последующий `GET` вернет следующий ответ на прямую замену:

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer"
}
```

в то время как преобразование, а затем заменит подход, вернет следующее представление:

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: e0023aa4e

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    }
}
```

Побочные эффекты

Обратите внимание, что `PUT` разрешено иметь побочные эффекты, хотя он определен как идемпотентная операция! Это описано в RFC7231 как

> Запрос PUT, применяемый к целевому ресурсу, **может иметь побочные эффекты для других ресурсов** . Например, статья может иметь URI для идентификации «текущей версии» (ресурса), которая отделена от URI, идентифицирующих каждую конкретную версию (разные ресурсы, которые в одной точке разделяют то же состояние, что и ресурс текущей версии). Таким образом, успешный запрос PUT на URI «текущей версии» может создать ресурс новой версии в дополнение к изменению состояния целевого ресурса и может также привести к добавлению ссылок между связанными ресурсами.

Создание дополнительных записей в журнале не считается побочным эффектом, так как это, конечно, не состояние ресурса в целом.

Частичные обновления

RFC7231 упоминает об частичных обновлениях:

> Обновления частичного контента возможны путем таргетинга отдельно идентифицированного ресурса с состоянием, которое перекрывает часть более

крупного ресурса, или с помощью другого метода, который был определен специально для частичных обновлений (например, метода PATCH, определенного в RFC5789).

Поэтому частичные обновления могут быть выполнены в двух вариантах:

- Попросите ресурс внедрить несколько меньших под-ресурсов и обновить только соответствующий подресурс, а не полный ресурс через `PUT`
- Используя `PATCH` и проинструктируйте сервер, что обновлять

Частичное обновление с перекрывающимся состоянием

Если представление пользователя необходимо частично обновить из-за перемещения пользователя в другое место, вместо того, чтобы напрямую обновлять пользователя, связанный ресурс должен быть обновлен напрямую, что отражает частичное обновление представления пользователя.

Перед перемещением пользователь имел следующее представление

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept: application/hal+json; charset=utf-8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    },
    "_embedded": {
        "ea:address": {
            "street": "Terrace Drive, Central Park",
            "zip": "NY 10024"
            "city": "New York",
            "country": "United States of America",
            "_links": {
                "self": { "href": "/address/abc" },
                "google_maps": { "href": "http://maps.google.com/?ll=40.7739166,-73.970176" }
            }
        }
    }
}
```

По мере того, как пользователь переезжает в новое место, она обновляет свою информацию о местоположении следующим образом:

```
PUT /address/abc HTTP/1.1
Host: http://www.acmee.com
Content-Type: "application/json; charset=utf-8"
Content-Length: 109
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

{
    "street": "Standford Ave",
    "zip": "CA 94306",
    "city": "Pablo Alto",
    "country": "United States of America"
}
```

С семантикой transform-before-replace для несоответствующего медиа-типа между существующим ресурсом адреса и тем, который указан в запросе, как описано выше, ресурс адреса теперь обновляется, что имеет последствия для последующего запроса `GET` на пользовательском ресурсе возвращается новый адрес для пользователя.

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept: application/hal+json; charset=utf-8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    },
    "_embedded": {
        "ea:address": {
            "street": "Standford Ave",
            "zip": "CA 94306",
            "city": "Pablo Alto",
            "country": "United States of America"
            "_links": {
                "self": { "href": "/address/abc" },
                "google_maps": { "href": "http://maps.google.com/?ll=37.4241311,-122.1524475"
}
            }
        }
    }
}
```

Исправление частичных данных

`PATCH` определен в RFC5789 и не является непосредственно частью спецификации HTTP как таковой. Общим недоверием является то, что отправка только полей, которые должны быть частично обновлены, достаточно в рамках запроса `PATCH` . Поэтому в спецификации указывается

> Метод PATCH запрашивает, чтобы набор изменений, описанных в объекте запроса, был применен к ресурсу, идентифицированному Request-URI. Набор изменений представлен в формате, называемом «патч-документом», идентифицированным типом носителя.

Это означает, что клиент должен вычислить необходимые шаги, необходимые для преобразования ресурса из состояния A в состояние B и отправки этих инструкций на сервер.

Популярным медиа-типом на основе JSON для исправления является JSON Patch .

Если возраст и название работы нашего образца пользователя меняются, а дополнительное поле, представляющее доход пользователя, должно быть добавлено, частичное обновление с использованием `PATCH` с использованием JSON Patch может выглядеть следующим образом:

```
PATCH /users/1234 HTTP/1.1
Host: http://www.acmee.com
Content-Type: application/json-patch+json; charset=utf-8
Content-Length: 188
Accept: application/json
If-Match: "e0023aa4e"

[
    { "op": "replace", "path": "/age", "value": 40 },
    { "op": "replace", "path": "/job_title", "value": "Senior Software Developer" },
    { "op": "add", "path": "/salery", "value": 63985.00 }
]
```

`PATCH` может обновлять сразу несколько ресурсов и требует, чтобы эти изменения применялись атомарно, что означает, что все изменения должны быть применены или вообще отсутствуют, что ставит транзакционную нагрузку на разработчика API.

Успешное обновление может вернуть что-то вроде этого

```
HTTP/1.1 200 OK
Location: /users/1234
Content-Type: application/json
ETag: "df00eb258"

{
    "name": "Charlie Smith",
    "age": 40,
```

```
    "job_title": "Senior Software Developer",
    "salary": 63985.00
}
```

хотя не ограничивается только `200 OK` кодами ответа `200 OK` .

Чтобы предотвратить промежуточные обновления (изменения, сделанные между предыдущей выборкой состояния представления и обновлением), следует использовать заголовок `ETag` , `If-Match` или `If-Unmodified-Since` .

Обработка ошибок

Спецификация на `PATCH` рекомендует следующую обработку ошибок:

Тип	Код ошибки
Недопустимый файл патча	`400 Bad Request`
Неподдерживаемый патч-документ	`415 Unsupported Media Type`
Непроцессный запрос, т. Е. Если resoure станет недействительным, применив патч	`422 Unprocessable Entity`
Ресурс не найден	`404 Not Found`
Конфликтующее состояние, то есть переименование (перемещение) поля, которое не существует	`409 Conflict`
Конфликтная модификация, то есть, если клиент использует заголовок `If-Match` или `If-Unmodified-Since` проверка которого не удалась. Если предварительное условие не было доступно, последний код ошибки должен быть возвращен	`412 Precondition Failed` или `409 Conflict`
Параллельная модификация, то есть, если запрос должен быть применен до принятия дальнейших запросов `PATCH`	`409 Conflict`

Удалить ресурс

Другим распространенным применением HTTP API является удаление существующего ресурса. Обычно это делается с помощью запросов `DELETE` .

Если удаление было успешным, сервер должен вернуть `200 OK` ; соответствующий код ошибки, если это не так.

Если наш сотрудник Чарли Смит покинул компанию, и теперь мы хотим удалить его записи, это может выглядеть так:

```
DELETE /employees/1/charlie-smith HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    'links': [
        {
            'uri': '/employees',
            'rel': 'create',
            'method': 'POST'
        }
    ]
}
```

Список ресурсов

Последнее общее использование HTTP API - это получение списка существующих ресурсов на сервере. Списки, подобные этому, должны быть получены с использованием запросов `GET` , поскольку они только *извлекают* данные.

Сервер должен вернуть `200 OK` если он может предоставить список, или соответствующий код ошибки, если нет.

Таким образом, список наших сотрудников может выглядеть так:

```
GET /employees HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    'employees': [
        {
            'name': 'Charlie Smith',
            'age': 39,
            'job_title': 'Software Developer',
            'salary': 63985.00
            'links': [
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'self',
                    'method': 'GET'
                },
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'delete',
                    'method': 'DELETE'
                },
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'edit',
                    'method': 'PATCH'
```

```
                }
            ]
        },
        {
            'name': 'Donna Prima',
            'age': 30,
            'job_title': 'QA Tester',
            'salary': 77095.00
            'links': [
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'self',
                    'method': 'GET'
                },
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'delete',
                    'method': 'DELETE'
                },
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'edit',
                    'method': 'PATCH'
                }
            ]
        }
    ],
    'links': [
        {
            'uri': '/employees/new',
            'rel': 'create',
            'method': 'PUT'
        }
    ]
}
```

Прочитайте HTTP для API онлайн: https://riptutorial.com/ru/http/topic/3423/http-для-api

глава 3: HTTP-запросы

параметры

Метод HTTP	Цель
`OPTIONS`	Получить информацию об опциях связи (доступных методах и заголовках), доступных в указанном URI запроса.
`GET`	Получить данные, идентифицированные URI запроса, или данные, созданные скриптом, доступным в URI запроса.
`HEAD`	Идентично `GET` за исключением того, что сервер сообщений не будет возвращен никакому телу сообщения: только заголовки.
`POST`	Отправьте блок данных (указанный в теле сообщения) на сервер для добавления к ресурсу, указанному в URI запроса. Наиболее часто используется для обработки формы.
`PUT`	Храните закрытую информацию (в теле сообщения) в качестве нового или обновленного ресурса в соответствии с данным URI запроса.
`DELETE`	Удалить или очередь для удаления, ресурс, идентифицированный URI запроса.
`TRACE`	По сути, команда echo: работающий, совместимый HTTP-сервер должен отправить весь запрос обратно в качестве тела ответа 200 (OK).

замечания

Метод `CONNECT` зарезервирован спецификацией определений методов для использования с прокси-серверами, которые могут переключаться между режимами проксирования и туннелирования (например, для туннелирования SSL).

Examples

Отправка минимального HTTP-запроса вручную с помощью Telnet

Этот пример демонстрирует, что HTTP является текстовым протоколом интернет-связи и показывает базовый HTTP-запрос и соответствующий HTTP-ответ.

Вы можете использовать Telnet для ручной отправки минимального HTTP-запроса из командной строки, как показано ниже.

1. Запустите сеанс Telnet на веб-сервере `www.example.org` на порту 80:

   ```
   telnet www.example.org 80
   ```

   Telnet сообщает, что вы подключились к серверу:

   ```
   Connected to www.example.org.
   Escape character is '^]'.
   ```

2. Введите строку запроса, чтобы отправить URL-адрес URL-адреса запроса GET `/` , используя HTTP 1.1

   ```
   GET / HTTP/1.1
   ```

3. Введите строку поля заголовка HTTP, чтобы определить часть имени хоста требуемого URL-адреса, которая требуется в HTTP 1.1

   ```
   Host: www.example.org
   ```

4. Введите пустую строку для завершения запроса.

   Веб-сервер отправляет HTTP-ответ, который появляется в сеансе Telnet.

Полный сеанс выглядит следующим образом. Первая строка ответа - это *строка состояния HTTP* , которая включает в себя код состояния 200 и текст состояния « *OK*» , которые указывают, что запрос был успешно обработан. За ним следуют несколько полей заголовка HTTP, пустая строка и ответ HTML.

```
$ telnet www.example.org 80
Trying 2606:2800:220:1:248:1893:25c8:1946...
Connected to www.example.org.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Thu, 21 Jul 2016 15:56:05 GMT
Etag: "359670651"
Expires: Thu, 28 Jul 2016 15:56:05 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (lga/1318)
Vary: Accept-Encoding
X-Cache: HIT
x-ec-custom-error: 1
Content-Length: 1270
```

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
</head>
<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents. You may
use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

(элемент `style` и пустые строки удалены из ответа HTML для краткости.)

Основной формат запроса

В HTTP 1.1 минимальный HTTP-запрос состоит из строки запроса и заголовка `Host` :

```
GET /search HTTP/1.1 \r\n
Host: google.com \r\n
\r\n
```

Первая строка имеет такой формат:

```
Method Request-URI HTTP-Version CRLF
```

`Method` должен быть допустимым методом HTTP; один из [1] [2] :

- `OPTIONS`
- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `PATCH`
- `TRACE`
- `CONNECT`

`Request-URI` указывает либо URI, либо путь к ресурсу, который запрашивает клиент. Это может быть:

- полный URI, включая схему, хост, (необязательный) порт и путь; или же
- путь, в этом случае хост должен быть указан в заголовке `Host`

`HTTP-Version` указывает версию HTTP-протокола, который использует клиент. Для HTTP 1.1

запросов это всегда должно быть `HTTP/1.1` .

Линия запроса заканчивается парой обратной линии возврата каретки, обычно представленной `\r\n` .

Поля заголовка запроса

Поля заголовка (обычно называемые только заголовками) могут быть добавлены в HTTP-запрос для предоставления дополнительной информации с запросом. Заголовок имеет семантику, аналогичную параметрам, переданным методу на любом языке программирования, который поддерживает такие вещи.

Запрос с заголовками `Host` , `User-Agent` и `Referer` может выглядеть следующим образом:

```
GET /search HTTP/1.1 \r\n
Host: google.com \r\n
User-Agent: Chrome/54.0.2803.1 \r\n
Referer: http://google.com/ \r\n
\r\n
```

Полный список поддерживаемых заголовков запросов HTTP 1.1 можно найти в спецификации . Наиболее распространенными являются:

- `Host` - часть имени хоста URL-адреса запроса (требуется в HTTP / 1.1)
- `User-Agent` - строка, представляющая запрос агента пользователя;
- `Referer` - URI, от которого сюда ссылался клиент; а также
- `If-Modified-Since` - указывает дату, которую сервер может использовать, чтобы определить, изменился ли ресурс, и указать, что клиент может использовать кешированную копию, если она не указана.

Заголовок должен быть сформирован как `Name: Value CRLF` . `Name` - это имя заголовка, например `User-Agent` . `Value` - это назначенные ему данные, и строка должна заканчиваться CRLF. Имена заголовков нечувствительны к регистру и могут использовать только буквы, цифры и символы `!#$%&'*+-.^_`|~` (RFC7230 раздел 3.2.6 Компоненты значения поля).

Имя поля заголовка `Referer` является опечаткой для «referrer», введенной случайно в RFC1945 .

Органы сообщения

Некоторые HTTP-запросы могут содержать тело сообщения. Это дополнительные данные, которые сервер будет использовать для обработки запроса. Органы сообщения чаще всего используются в сообщениях POST или PATCH и PUT для предоставления новых данных, которые сервер должен применять к ресурсу.

Запросы, которые включают тело сообщения, должны всегда включать его длину в байтах

с заголовком `Content-Length` .

Тело сообщения включено *после* всех заголовков и двойной CRLF. Пример запроса PUT с телом может выглядеть так:

```
PUT /files/129742 HTTP/1.1\r\n
Host: example.com\r\n
User-Agent: Chrome/54.0.2803.1\r\n
Content-Length: 202\r\n
\r\n
This is a message body. All content in this message body should be stored under the
/files/129742 path, as specified by the PUT specification. The message body does
not have to be terminated with CRLF.
```

Запросы `HEAD` и `TRACE` не должны содержать тело сообщения.

Прочитайте HTTP-запросы онлайн: https://riptutorial.com/ru/http/topic/2909/http-запросы

глава 4: HTTP-ответы

параметры

Код состояния	Причина-фраза - Описание
100	**Продолжить** - клиент должен отправить следующую часть запроса с несколькими частями.
101	**Переключение протоколов** - сервер меняет версию или тип протокола, используемые в этом сообщении.
200	**OK** - сервер получил и выполнил запрос клиента.
201	**Создано** - сервер принял запрос и создал новый ресурс, доступный под URI в заголовке `Location` .
202	**Принято** - сервер получил и принял запрос клиента, но он еще не начал или не завершил обработку.
203	**Неавторитная информация** - сервер возвращает данные, которые могут быть поднабором или надмножеством информации, доступной на исходном сервере. В основном используется прокси.
204	**Нет содержимого** - используется вместо 200 (OK), когда тела нет ответа.
205	**Сбросить содержимое** - идентично 204 (без содержимого), но клиент должен перезагрузить активный вид документа.
206	**Частичное содержимое** - используется вместо 200 (OK), когда клиент запросил заголовок `Range` .
300	**Несколько вариантов** - запрашиваемый ресурс доступен в нескольких URI, и клиент должен перенаправить запрос на URI, указанный в списке в теле сообщения.
301	**Перемещено на постоянной основе** - запрошенный ресурс больше не доступен в этом URI, и клиент должен перенаправить этот и все будущие запросы в URI, указанный в заголовке `Location` .
302	**Найдено** - ресурс временно находится под другим URI. Этот запрос должен быть перенаправлен на подтверждение пользователя в URI в

Код состояния	Причина-фраза - Описание
	заголовке `Location` , но будущие запросы не должны быть изменены.
303	**См. Прочее** - очень похоже на 302 (Найдено), но не требует ввода пользователем для перенаправления на предоставленный URI. Предоставленный URI должен быть получен с помощью запроса GET.
304	**Не изменено** - клиент отправил `If-Modified-Since` или аналогичный заголовок, и ресурс не был изменен с этой точки; клиент должен отображать кешированную копию ресурса.
305	**Использовать прокси** - запрошенный ресурс необходимо запросить снова через прокси-сервер, указанный в поле заголовка « `Location` .
307	**Временное перенаправление** - идентично 302 (найдено), но клиенты HTTP 1.0 не поддерживают 307 ответов.
400	**Плохой запрос** - клиент отправил неверный запрос, содержащий синтаксические ошибки, и должен изменить запрос, чтобы исправить это, прежде чем повторять его.
401	**Неавторизованный** - запрошенный ресурс недоступен без аутентификации. Клиент может повторить запрос, используя заголовок `Authorization` чтобы предоставить данные аутентификации.
402	**Обязательный платеж** - зарезервированный, неуказанный код состояния для использования приложениями, для которых требуется подписка на пользователя для просмотра содержимого.
403	**Запрещено** - сервер понимает запрос, но отказывается выполнять его из-за существующих средств контроля доступа. Запрос не следует повторять.
404	**Не найдено.** На этом сервере нет ресурса, который соответствует запрошенному URI. Может использоваться вместо 403, чтобы не подвергать деталям контроля доступа.
405	**Метод не разрешен** - ресурс не поддерживает метод запроса (HTTP-глагол); В заголовке « `Allow` перечислены приемлемые методы запроса.
406	**Не приемлемо** - ресурс имеет характеристики, которые нарушают принимаемые заголовки, отправленные в запросе.
407	**Требуется прокси-аутентификация** - аналогично 401 (неавторизованный), но указывает, что клиент должен сначала пройти

Код состояния	Причина-фраза - Описание
	аутентификацию с промежуточным прокси.
408	**Тайм-аут запроса** - сервер ожидал другого запроса от клиента, но ни один из них не был предоставлен в течение приемлемого таймфрейма.
409	**Конфликт** - запрос не может быть завершен, поскольку он противоречит текущему состоянию ресурса.
410	**Gone** - аналогично 404 (Not Found), но указывает на постоянное удаление. Отсутствует адрес пересылки.
411	**Требуемая длина** - клиент не указал допустимый заголовок `Content-Length` и должен сделать это до того, как сервер примет этот запрос.
412	**Precondition Failed** - ресурс недоступен со всеми условиями, указанными условными заголовками, отправленными клиентом.
413	**Request Entity Too Large** - сервер в настоящее время не может обработать тело сообщения той длины, которую отправил клиент.
414	**Request-URI Too Long** - сервер отказывается от запроса, потому что Request-URI длиннее, чем сервер готов интерпретировать.
415	**Неподдерживаемый тип носителя** - сервер не поддерживает MIME или тип носителя, указанный клиентом, и не может обслуживать этот запрос.
416	**Запрошенный диапазон не удовлетворен** - клиент запросил диапазон байтов, но сервер не может предоставить контент этой спецификации.
417	**Expectation Failed** - клиент указал ограничения в заголовке `Expect` которые сервер не может выполнить.
500	**Внутренняя ошибка сервера** - сервер встретил неожиданное условие или ошибку, которая мешает ему выполнить этот запрос.
501	**Не реализовано** - сервер не поддерживает функции, необходимые для завершения запроса. Обычно используется для указания метода запроса, который не поддерживается на *каком-либо* ресурсе.
502	**Bad Gateway** - сервер является прокси-сервером и получил неверный ответ от восходящего сервера при обработке этого запроса.
503	**Сервис недоступен** - сервер находится под большой нагрузкой или

Код состояния	Причина-фраза - Описание
	проходит техническое обслуживание, и в настоящее время он не может выполнять этот запрос.
504	**Gateway Timeout** - сервер является прокси-сервером и не получил ответ от восходящего сервера своевременно.
505	**Версия HTTP не поддерживается.** Сервер не поддерживает версию HTTP-протокола, с которой клиент выполнил свой запрос.

Examples

Основной формат ответа

Когда HTTP-сервер получает правильно сформированный HTTP-запрос , он должен обработать информацию, содержащую запрос, и вернуть ответ клиенту. Простой ответ HTTP 1.1 может выглядеть как любое из следующего, обычно за которым следуют несколько полей заголовка и, возможно, тело ответа:

```
HTTP/1.1 200 OK \r\n
```

```
HTTP/1.1 404 Not Found \r\n
```

```
HTTP/1.1 503 Service Unavailable \r\n
```

Простой ответ HTTP 1.1 имеет такой формат:

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

Как и в запросе, `HTTP-Version` указывает версию используемого протокола HTTP; для HTTP 1.1 это всегда должно быть строка `HTTP/1.1` .

`Status-Code` - это трехзначный код, который указывает статус запроса клиента. Первая цифра этого кода - это *класс статуса* , который помещает код состояния в одну из 5 категорий ответа [1] :

- `1xx` **Информационный** - сервер получил запрос и обработка продолжается
- `2xx` **Success** - сервер принял и обработал запрос
- `3xx` **Redirection** - требуется дальнейшие действия со стороны клиента для завершения запроса
- **Ошибки клиента** `4xx` - клиент отправил запрос, который был искажен или не может быть выполнен

- **Ошибки сервера** `5xx` - запрос был действительным, но сервер не может выполнить его в настоящее время

`Reason-Phrase` - это краткое описание кода состояния. Например, код `200` имеет причину фразы `OK` ; код `404` имеет фразу « `Not Found` . Полный список фраз причины доступен в параметрах, ниже или в спецификации HTTP .

Линия заканчивается парой обратной линии возврата каретки, обычно представленной `\r\n` .

Дополнительные заголовки

Подобно HTTP-запросу, HTTP-ответ может включать в себя дополнительные заголовки для изменения или увеличения ответа, который он предоставляет.

Полный список доступных заголовков определен в разделе 6.2 спецификации . Наиболее часто используемые заголовки:

- `Server` , который функционирует как заголовок запроса `User-Agent` для сервера;
- `Location` , которое используется для ответов статуса 201 и 3xx, чтобы указать URI для перенаправления; а также
- `ETag` , который является уникальным идентификатором для этой версии возвращаемого ресурса, чтобы клиенты могли кэшировать ответ.

Заголовки ответов поступают после строки состояния, и поскольку заголовки запросов формируются как таковые:

```
Name: Value CRLF
```

`Name` содержит имя заголовка, например `ETag` или `Location` , а `Value` - значение, заданное сервером для этого заголовка. Линия заканчивается CRLF.

Ответ с заголовками может выглядеть так:

```
HTTP/1.1 201 Created \r\n
Server: WEBrick/1.3.1 \r\n
Location: http://example.com/files/129742 \r\n
```

Органы сообщения

Как и в структурах запросов , HTTP-ответы могут содержать тело сообщения. Это предоставляет дополнительные данные, которые клиент будет обрабатывать. Примечательно, что ответы 200 OK на хорошо сформированный запрос GET должны всегда предоставлять тело сообщения, содержащее запрошенные данные. (Если их нет, 204 No Content - более подходящий ответ).

Тело сообщения включено после всех заголовков и двойной CRLF. Что касается запросов, то его длина в байтах должна быть задана заголовком `Content-Length` . Следовательно, успешный ответ на запрос GET может выглядеть так:

```
HTTP/1.1 200 OK\r\n
Server: WEBrick/1.3.1\r\n
Content-Length: 39\r\n
ETag: 4f7e2ed02b836f60716a7a3227e2b5bda7ee12c53be282a5459d7851c2b4fdfd\r\n
\r\n
Nobody expects the Spanish Inquisition.
```

Прочитайте HTTP-ответы онлайн: https://riptutorial.com/ru/http/topic/3077/http-ответы

глава 5: Аутентификация

параметры

параметр	подробности
Статус ответа	`401` если исходный сервер требует аутентификации, `407` если промежуточный прокси требует аутентификации
Заголовки ответов	`WWW-Authenticate` сервером происхождения, `Proxy-Authenticate` промежуточным прокси-сервером
Запросить заголовки	`Authorization` для авторизации на сервере происхождения, `Proxy-Authorization` против промежуточного прокси
Схема аутентификации	`Basic` для базовой аутентификации, но могут использоваться другие, такие как `Digest` и `SPNEGO` . См. Реестр схем аутентификации HTTP .
область	Имя защищенного пространства на сервере; сервер может иметь несколько таких пространств, каждый из которых имеет различное имя и механизмы аутентификации.
полномочия	Для `Basic` : имя пользователя и пароль, разделенные двоеточием, base64-encoded; например, `username:password` base64-кодируется `dXNlcm5hbWU6cGFzc3dvcmQ=`

замечания

Базовая аутентификация определена в RFC2617 . Он может использоваться для аутентификации на сервере происхождения после получения `401 Unauthorized` а также прокси-сервера после `407 (Proxy Authentication Required)` . В (декодированных) учетных данных пароль начинается после первого двоеточия. Поэтому имя пользователя не может содержать двоеточие, но пароль может.

Examples

Базовая аутентификация HTTP

HTTP Basic Authentication предоставляет простой механизм аутентификации. Учетные данные отправляются в виде простого текста, и по умолчанию это небезопасно. Успешная аутентификация выполняется следующим образом.

Клиент запрашивает страницу, для которой ограничен доступ:

```
GET /secret
```

Сервер отвечает кодом состояния `401 Unauthorized` и запрашивает аутентификацию клиента:

```
401 Unauthorized
WWW-Authenticate: Basic realm="Secret Page"
```

Клиент отправляет заголовок `Authorization` . Учетные данные - это `username:password` base64 закодирован:

```
GET /secret
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Сервер принимает учетные данные и отвечает содержимым страницы:

```
HTTP/1.1 200 OK
```

Прочитайте Аутентификация онлайн: https://riptutorial.com/ru/http/topic/3286/аутентификация

глава 6: Коды ответов и сжатие

Examples

Сжатие HTTP

Тело сообщения HTTP может быть сжато (с HTTP / 1.1). Либо сервер сжимает запрос и добавляет заголовок `Content-Encoding` , либо прокси-сервер делает и добавляет заголовок `Transfer-Encoding` .

Клиент может отправить заголовок запроса `Accept-Encoding` чтобы указать, какие кодировки он принимает.

Наиболее часто используемые кодировки:

- gzip-deflate (LZ77) с контрольной суммой CRC32, реализованной в программе сжатия файла gzip (RFC1952)
- deflate - формат данных «zlib» (RFC1950), алгоритм дефляции (гибридный LZ77 и Хаффман) с контрольной суммой Adler32

Несколько методов сжатия

Можно сжимать тело сообщения ответа HTTP более одного раза. Затем имена кодировок должны быть разделены запятой в том порядке, в котором они были применены. Например, если сообщение было сжато с помощью deflate, а затем gzip, заголовок должен выглядеть так:

```
Content-Encoding: deflate, gzip
```

Несколько заголовков `Content-Encoding` также действительны, но не рекомендуется:

```
Content-Encoding: deflate
Content-Encoding: gzip
```

сжатие gzip

Клиент сначала отправляет запрос с заголовком `Accept-Encoding` который указывает, что он поддерживает gzip:

```
GET / HTTP/1.1\r\n
Host: www.google.com\r\n
Accept-Encoding: gzip, deflate\r\n
\r\n
```

Затем сервер может отправить ответ со сжатым телом ответа и заголовком `Content-Encoding` который указывает, что была использована кодировка gzip ::

```
HTTP/1.1 200 OK\r\n
Content-Encoding: gzip\r\n
Content-Length: XX\r\n
\r\n
... compressed content ...
```

Прочитайте Коды ответов и сжатие онлайн: https://riptutorial.com/ru/http/topic/5046/коды-ответов-и-сжатие

глава 7: Коды статуса HTTP

Вступление

В HTTP коды состояния являются машиночитаемым механизмом, указывающим результат ранее выданного запроса. Из RFC 7231, сек. 6 : «Элемент состояния-кода представляет собой трехзначный целочисленный код, дающий результат попытки понять и удовлетворить запрос».

Формальная грамматика позволяет кодам быть чем угодно между `000` и `999` . Однако только значение диапазона от `100` до `599` присваивало значение.

замечания

HTTP / 1.1 определяет число числовых кодов состояния HTTP, которые отображаются в строке состояния - первой строке ответа HTTP - чтобы суммировать то, что клиент должен делать с ответом.

Первая цифра кодов состояния определяет класс ответа:

- `1xx` Информационный
- `2xx` Клиентский запрос успешно
- `3xx` Запрос перенаправлен - необходимы дальнейшие действия, например новый запрос
- Ошибка `4xx` Client - не повторяйте один и тот же запрос
- Ошибка сервера `5xx` - возможно, повторите попытку

На практике не всегда легко выбрать наиболее подходящий код состояния.

Examples

500 - внутренняя ошибка сервера

Внутренняя ошибка сервера HTTP 500 - это общее сообщение, означающее, что сервер столкнулся с чем-то неожиданным. Приложения (или общий веб-сервер) должны использовать 500, когда возникает ошибка обработки запроса - то есть генерируется исключение или условие ресурса препятствует завершению процесса.

Пример строки состояния:

```
HTTP/1.1 500 Internal Server Error
```

404 Не Найдено

HTTP 404 Not Found означает, что сервер не смог найти путь, используя URI, запрошенный клиентом.

```
HTTP/1.1 404 Not Found
```

Чаще всего запрашиваемый файл был удален, но иногда это может быть неправильная конфигурация корневого каталога или отсутствие разрешений (хотя пропускающие разрешения чаще запускают HTTP 403 Forbidden).

Например, IIS Microsoft записывает 404.0 (0 является под-статусом) в свои файлы журнала, когда запрошенный файл был удален. Но когда входящий запрос блокируется правилами фильтрации запросов, он записывает 404.5-404.19 в файлы журнала, в соответствии с которым правило блокирует запрос. Более подробную ссылку на код ошибки можно найти в Microsoft Support .

Отказ в доступе к защищенным файлам

Использовать 403 Запрещено, если клиент запросил ресурс, который недоступен из-за существующих элементов управления доступом. Например, если ваше приложение имеет маршрут `/admin` который должен быть доступен только пользователям с правами администратора, вы можете использовать 403, когда обычный пользователь запрашивает страницу.

```
GET /admin HTTP/1.1
Host: example.com
```

```
HTTP/1.1 403 Forbidden
```

Успешный запрос

Отправьте ответ HTTP с кодом состояния `200` чтобы указать успешный запрос. Строка состояния ответа HTTP:

```
HTTP/1.1 200 OK
```

Текст состояния `OK` является только информативным. Тело ответа (полезная нагрузка сообщения) должно содержать представление запрашиваемого ресурса. Если нет представления 201 Нет содержимого.

Отвечая на условный запрос на кешированный контент

Отправьте **304 не измененный** статус ответа с сервера в ответ на запрос клиента,

содержащий заголовки `If-Modified-Since` и `If-None-Match` , если ресурс запроса не изменился.

Например, если запрос клиента на веб-страницу включает заголовок `If-Modified-Since: Fri, 22 Jul 2016 14:34:40 GMT` а с тех пор страница не была изменена, ответьте на строку состояния `HTTP/1.1 304 Not Modified` .

Top 10 Код состояния HTTP

2xx Успех

- **200 OK** - стандартный ответ для успешных запросов HTTP.
- **201 Создано** - запрос выполнен, в результате чего создается новый ресурс.
- **204 Нет содержимого** . Сервер успешно обработал запрос и не возвращает никакого содержимого.

Перенаправление 3xx

- **304 Not Modified** - указывает, что ресурс не был изменен с версии, указанной заголовками запроса `If-Modified-Since` или `If-None-Match` .

Ошибка клиента 4xx

- **400 Bad Request** - сервер не может или не будет обрабатывать запрос из-за очевидной ошибки клиента (например, синтаксис неправильного запроса, слишком большой размер, неправильное обращение к сообщениям запроса или маршрутизация ложных запросов).
- **401 Несанкционированный** - *аналогичен Запрещенному 403* , но специально для использования, когда требуется аутентификация, и не удалось или еще не предоставлено. Ответ должен включать поле заголовка `WWW-Authenticate` содержащее вызов, применимый к запрашиваемому ресурсу.
- **403 Запрещено** - запрос был действительным запросом, но сервер отказывается отвечать на него. Пользователь может войти в систему, но не имеет необходимых разрешений для ресурса.
- **404 Not Found** - запрошенный ресурс не найден, но может быть доступен в будущем. Возможны последующие запросы клиента.
- **409 Конфликт.** Указывает, что запрос не может быть обработан из-за конфликта в запросе, например, в конфликте редактирования нескольких одновременных обновлений.

Ошибка сервера 5xx

- **500 Внутренняя ошибка сервера. Общее** сообщение об ошибке, заданное при возникновении непредвиденного состояния и не более подходящее сообщение.

Прочитайте Коды статуса HTTP онлайн: https://riptutorial.com/ru/http/topic/2577/коды-статуса-http

глава 8: Кэширование HTTP-ответов

замечания

Ответы кэшируются отдельно для каждого URL-адреса и каждого метода HTTP.

HTTP-кэширование определено в RFC 7234 .

глоссарий

- **fresh** - состояние кэшированного ответа, которое еще не истекло. Как правило, свежий ответ может *удовлетворять* запросы без необходимости контактировать с сервером ответ, исходящий из него.
- **stale** - состояние кэшированного ответа, которое прошло по истечении срока его действия. Как правило, устаревшие ответы не могут использоваться для *удовлетворения* запроса без проверки с сервером, сохраняется ли он.
- **удовлетворяющий** запросу с кешированием, *удовлетворяет* запросу, когда все условия в запросе соответствуют кешированному ответу, например, они имеют один и тот же HTTP-метод и URL-адрес, ответ свежий или запрос позволяет выполнять устаревшие ответы, заголовки заголовков запросов заголовков, перечисленные в заголовке ответа `Vary` , и т. д. ,
- **revalidation** - проверка того, является ли кешированный ответ свежим. Обычно это делается с *условным запросом,* содержащим статус `If-Modified-Since` или `If-None-Match` и ответ `304` .
- **частный кеш-** кеш для одного пользователя, например, в веб-браузере. Частные кэши могут хранить персонализированные ответы.
- общий **кэш** - кеш, общий для многих пользователей, например, на прокси-сервере. Такой кеш может отправлять один и тот же ответ нескольким пользователям.

Examples

Кэш-ответ для каждого в течение 1 года

```
Cache-Control: public, max-age=31536000
```

`public` означает, что ответ одинаковый для всех пользователей (он не содержит персонализированной информации). `max-age` - через несколько секунд. 31536000 = 60 * 60 * 24 * 365.

Это рекомендуется для статических активов, которые никогда не должны меняться.

Персонализированный ответ кэша в течение 1 минуты

```
Cache-Control: private, max-age=60
```

`private` указывает, что ответ может быть кэширован только для пользователя, который запросил ресурс, и не может использоваться повторно, когда другие пользователи запрашивают тот же ресурс. Это подходит для ответов, которые зависят от файлов cookie.

Прекратите использование кэшированных ресурсов, не проверив сначала сервер

```
Cache-Control: no-cache
```

Клиент будет вести себя так, как будто ответ не был кэширован. Это подходит для ресурсов, которые могут непредсказуемо меняться в любое время и которые пользователи должны всегда видеть в последней версии.

Ответы `no-cache` будут медленнее (высокая латентность) из-за необходимости обращаться к серверу каждый раз, когда они используются.

Однако для экономии пропускной способности клиенты *могут* сохранять такие ответы. Ответы `no-cache` не будут использоваться для удовлетворения запросов без обращения к серверу каждый раз, чтобы проверить, можно ли повторно использовать кешированный ответ.

Запросить ответы не на всех

```
Cache-control: no-store
```

Предписывает клиентам не кэшировать ответ в любом случае и забыть об этом в самое ближайшее время.

Эта директива была первоначально разработана для чувствительных данных (сегодня вместо этого следует использовать HTTPS), но ее можно использовать, чтобы избежать загрязнения кэшей с ответами, которые нельзя использовать повторно.

Он подходит только в конкретных случаях, когда данные ответа всегда разные, например, конечная точка API, которая возвращает большое случайное число. В противном случае для `no-cache` и повторной аттестации можно использовать поведение «неприкасаемого» ответа, сохраняя при этом некоторую пропускную способность.

Устаревшие, избыточные и нестандартные заголовки

- `Expires` - указывает дату, когда ресурс становится устаревшим. Он полагается на

серверы и клиенты, имеющие точные часы и поддерживающие часовые пояса правильно. `Cache-control: max-age` имеет преимущество перед `Expires` и, как правило, более надежным.

- директивы `post-check` и `pre-check` являются нестандартными расширениями Internet Explorer, которые позволяют использовать устаревшие ответы. Стандартная альтернатива является `stale-while-revalidate` .
- `Pragma: no-cache` - устарел в 1999 году . Вместо этого следует использовать `Cache-control` .

Изменение кэшированных ресурсов

Самый простой способ обхода кеша - изменить URL. Это используется как наилучшая практика, когда URL-адрес содержит версию или контрольную сумму ресурса, например

```
http://example.com/image.png?version=1
http://example.com/image.png?version=2
```

Эти два URL-адреса будут кэшироваться отдельно, поэтому даже если `…?version=1` было кэшировано *навсегда* , новая копия может быть немедленно восстановлена как `…?version=2` .

Не используйте случайные URL-адреса для обхода кешей. Используйте `Cache-control: no-cache` или `Cache-control: no-store` вместо этого. Если ответы со случайными URL-адресами отправляются без директивы `no-store` , они будут излишне храниться в кэшах и выталкивать из кэша более полезные ответы, снижая производительность всего кеша.

Прочитайте Кэширование HTTP-ответов онлайн: https://riptutorial.com/ru/http/topic/3296/кэширование-http-ответов

глава 9: Перекрестное происхождение и контроль доступа

замечания

Совместное использование ресурсов ресурса, предназначенное для динамических запросов между доменами, часто использует такие методы, как AJAX . Хотя сценарий выполняет большую часть работы, HTTP-сервер должен поддерживать запрос с использованием правильных заголовков.

Examples

Клиент: отправка запроса на совместное использование ресурсов (CORS)

Запрос перекрестного происхождения должен быть отправлен, включая заголовок `Origin` . Это указывает, откуда возник запрос. Например, запрос перекрестного происхождения с `http://example.com` на `http://example.org` будет выглядеть следующим образом:

```
GET /cors HTTP/1.1
Host: example.org
Origin: example.com
```

Сервер будет использовать это значение, чтобы определить, разрешен ли запрос.

Сервер: ответ на запрос CORS

Ответ на запрос CORS должен включать заголовок `Access-Control-Allow-Origin` , который диктует, каким истокам разрешено использовать ресурс CORS. Этот заголовок может принимать одно из трех значений:

- Происхождение. Выполнение этого разрешает только запросы из *этого источника* .
- Символ `*` . Это разрешает запросы из *любого источника* .
- Строка `null` . Это не позволяет *запросам CORS* .

Например, при получении запроса CORS из источника `http://example.com` , если `example.com` является авторизованным источником, сервер отправит ответ:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.com
```

Ответ на любой источник также разрешает этот запрос, то есть:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
```

Разрешение учетных данных пользователя или сеанса

Разрешение учетных данных пользователя или сеанса пользователя, отправляемых с запросом CORS, позволяет серверу сохранять данные пользователя через запросы CORS. Это полезно, если серверу необходимо проверить, вошел ли пользователь в систему перед предоставлением данных (например, только выполнение действия, если пользователь вошел в систему), для этого требуется, чтобы запрос CORS был отправлен с учетными данными).

Это может быть достигнуто на стороне сервера для предполетных запросов, отправив заголовок `Access-Control-Allow-Credentials` в ответ на запрос предварительной `OPTIONS` . Возьмем следующий случай запроса CORS для `DELETE` ресурса:

```
OPTIONS /cors HTTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: DELETE
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.org
Access-Control-Allow-Methods: DELETE
Access-Control-Allow-Credentials: true
```

Строка `Access-Control-Allow-Credentials: true` указывает, что следующий запрос `DELETE` CORS может быть отправлен с учетными данными пользователя.

Запросы предварительной проверки

В базовом запросе CORS можно использовать один из двух методов:

- ПОЛУЧИТЬ
- СООБЩЕНИЕ

и только несколько заголовков. Запросы POST CORS могут дополнительно выбирать только из трех типов контента.

Чтобы избежать этой проблемы, запросы, которые хотят использовать другие методы, заголовки или типы контента, должны сначала выдать запрос предварительной *проверки* , который является запросом `OPTIONS` который включает заголовки запроса контроля доступа. Например, это запрос предполетной проверки, который проверяет, примет ли сервер запрос `PUT` который включает в себя заголовок `DNT` :

```
OPTIONS /cors HTTP/1.1
Host: example.com
```

```
Origin: example.org
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: DNT
```

Сервер: ответ на предполетные запросы

Когда сервер получает запрос перед полетом, он должен проверить, поддерживает ли он запрашиваемый метод и заголовки, и отсылает ответ, который указывает его способность поддерживать запрос, а также любые другие разрешенные данные (такие как учетные данные).

Они указаны в контроле доступа. Сервер также может отправить обратно заголовок `Max-Age` контроля доступа, указывающий, на сколько времени может быть кэширован запрос перед полетом.

Вот как выглядит запрос-ответ для запроса предполетной проверки:

```
OPTIONS /cors HHTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: DNT
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.org
Access-Control-Allow-Methods: PUT
Access-Control-Allow-Headers: DNT
```

Прочитайте Перекрестное происхождение и контроль доступа онлайн:
https://riptutorial.com/ru/http/topic/3424/перекрестное-происхождение-и-контроль-доступа

кредиты

S. No	Главы	Contributors
1	Начало работы с HTTP	Community, DaSourcerer, Kornel, Peter Hilton
2	HTTP для API	ArtOfCode, mnoronha, Peter Hilton, Roman Vottner
3	HTTP-запросы	artem, ArtOfCode, Jeff Bencteux, Peter Hilton
4	HTTP-ответы	ArtOfCode, Jeff Bencteux, Peter Hilton
5	Аутентификация	DaSourcerer, Peter Hilton, Stefan Kögl
6	Коды ответов и сжатие	Jeff Bencteux, Peter Hilton
7	Коды статуса HTTP	ArtOfCode, DaSourcerer, Deltik, Kornel, Lex Li, mnoronha, Peter Hilton, Rptk99, Sender, Xevaquor
8	Кэширование HTTP-ответов	DaSourcerer, Kornel
9	Перекрестное происхождение и контроль доступа	ArtOfCode