Лабораторная работа № 15

Организация Web-сервера. Протокол передачи гипертекстовой информации HTTP. Язык PHP.

Краткие теоретические сведения

1. Семантические компоненты Web.

В Web имеется три семантических компонента: унифицированные идентификаторы ресурсов (Uniform Resource Identifiers – URI), гипертекстовый язык разметки (Hypertext Markup Language – HTML) и протокол передачи гипертекста (Hypertext Transfer Protocol – HTTP). URI представляют собой механизм именования ресурсов в Web с целью их идентификации. HTML – это стандартный язык для создания гипертекстовых документов. HTTP – это язык для взаимодействия между Web-клиентами и Web-серверами.

1.1. Унифицированные идентификаторы ресурсов (URI).

Доступ к ресурсам в Web требует их идентификации. Web-ресурс идентифицируется с помощью унифицированного идентификатора ресурса (URI). Не являясь какой-либо физической сущностью, URI может быть воспринят как указатель па некий черный ящик, к которому применимы методы запросов с целью генерирования потенциально различных ответов в разное время. Метод запроса – это простая операция, такая как выборка, изменение или удаление ресурса. URI идентифицирует ресурс независимо от его текущего местонахождения или содержания. На верхнем уровне URI представляет собой простую форматированную строку, такую как http://www.foo.com/coolpic.gif. URI обычно состоит из трех частей: протокола для взаимодействия с сервером (например, http), имени сервера (например, www.foo.com) и имени ресурса па этом сервере (например, coolpic.gif). Наиболее популярной формой URI является унифицированный указатель ресурса – Uniform Resource Locator (URL).

1.2. Гипертекстовый язык разметки (HTML).

Гипертекстовый язык разметки (Hypertext Markup Language – HTML) обеспечивает стандартное представление гипертекстовых документов в текстовом формате. HTML произошел от более общего языка разметки Standard Generalized Markup Language (SGML). HTML позволяет форматировать текст, встраивать в документ изображения, а также создавать гипертекстовые ссылки па другие документы. Синтаксис HTML достаточно прозрачен и прост в изучении. Простейший HTML-документ – это нечто большее, чем обычный текстовой документ без форматирования или ссылок на другие ресурсы.

1.3. Протокол передачи гипертекста (HTTP).

Функционирование Web зависит от наличия стандартного, устоявшегося способа для взаимодействия Web-компонентов. Протокол передачи гипертекста Hypertext Transfer Protocol (HTTP) представляет собой наиболее распространенный способ передачи ресурсов в Web. HTTP определяет формат и назначение сообщений, которыми обмениваются Web-компоненты, такие как клиенты и серверы. Протокол – это язык, схожий с естественными человеческими языками, за исключением того, что он используется программами. Подобно другим языкам, протокол имеет свой особый синтаксис и семантику, связанные с использованием элементов языка. HTTP определяет синтаксис сообщений и способ интерпретации полей каждой строки сообщения. HTTP представляет собой протокол типа запрос-ответ – клиент отправляет сообщение-запрос, а затем сервер откликается сообщением-ответом. Клиентские запросы обычно порождаются действиями пользователя, например, щелчком мышью на гиперссылке или вводом URI в адресной строке браузера. HTTP не сохраняет своего состояния – клиенты и серверы трактуют каждый обмен сообщениями независимо от других, и нет необходимости сохранять какое-либо промежуточное состояние между запросами и ответами.

2. Структура и описание протокола HTTP

HTTP используется для передачи информации в различных форматах, на различных языках и с различными наборами символов. Синтаксис HTTP-сообщения основан на стандарте MIME – Multipurpose Internet Mail Extensions. Содержимое HTTP-сообщения слепо воспринимается протоколом – никакой интерпретации при этом не производится

Документ RFC 1945 отражает общие принципы использования HTTP в середине 90-х годов. Текущей версией протокола HTTP является версия HTTP/1.1.

2.1. Свойства протокола

Основные свойства HTTP:

- **Глобальные URI.** HTTP основывается на механизме именования URI. HTTP использует URI во всех транзакциях для идентификации ресурсов в Web.
- **Обмен по схеме запрос-ответ.** HTTP-запросы отправляются клиентами, получая затем ответы от серверов. Направление потока – от клиента к серверу; сервер не инициирует Web-трафик.
- **Отсутствие сохранения состояния.** Состояние между запросами и ответами клиентами и серверами не сохраняется. Каждая пара запрос-ответ трактуется как независимый обмен сообщениями.
- **Метаданные ресурсов.** Информация о ресурсах часто включается в Web-транзакции и может быть использована различными способами.

2.1.1. Обмен запросами-ответами.

Протокол HTTP задает синтаксис и семантику, в соответствии с которыми компоненты Web, такие как клиенты и серверы, взаимодействуют друг с другом. HTTP-сообщение структурировано и обладает определенным синтаксисом. HTTP является запрос-ответным протоколом, в котором запрос – это сообщение, посылаемое клиентом принимающему серверу. Принимающим может быть исходный сервер – сервер, на котором размещаются или генерируются ресурсы, или промежуточное звено, такое как прокси-сервер. Сервер-получатель отправляет обратно сообщение-ответ. Клиентом может выступать агент пользователя, нечто, инициировавшее запрос, или какой-либо компонент на пути между инициатором и конечным сервером-получателем. Протокол определяет набор расширяемых методов запроса, которые используются клиентом для выполнения операций, таких как получение, изменение, создание или удаление ресурса. Ресурсом является объект, сервис или коллекция элементарных сущностей, которые могут быть четко идентифицированы и размещены в любом месте сети. Рассмотрим следующий HTTP-запрос:

```
GET /foo.html HTTP/1.0
```

Строка `/foo.html` идентифицирует запрашиваемый с помощью метода `GET` ресурс. Номер версии протокола `HTTP`, в данном случае `1.0`, используется для идентификации возможностей отправителя. Цель запроса – получить текущее содержимое ресурса, идентифицируемого строкой `/foo.html`, с сервера. Если быть более точным, запрос приводит к применению метода `GET` к ресурсу, идентифицируемому `/foo.html`. В общем случае сообщение-запрос состоит из метода, URI, идентификатора версии протокола, необязательных полей заголовка запроса и необязательных данных, отправляемых с запросом.

После получения сообщения-запроса сервер осуществляет его синтаксический анализ и применяет метод к ресурсу. Сформированный ответ передается обратно клиенту в качестве сообщения-ответа. Например,

```
HTTP/1.0 200 0K
Date: Wed, 22 Mar 2017 08:01:01 GMT
Last-Modified: Wed, 22 Mar 2017 02:16:33 GMT
Content-Length: 3913

<3913 байтов текущего содержимого /foo.html>
```

Сообщение-ответ состоит из числового кода ответа (например, указывающего на успех или неудачу), фразы, поясняющей числовой код ответа, необязательных нолей заголовка и необязательного тела сообщения. В примере первая строка, известная как строка состояния, содержит помер версии HTTP и код ответа `200 OK`, указывающий, что запрос выполнен успешно. Заголовок ответа `Date` указывает время создания ответа. Заголовки `Last-Modified` и `Content-Length` указывают время последней модификации ресурса и длину содержимого, включенного в сообщение-ответ (`3913` байта), соответственно. В версии HTTP/0.9 протокола не предусмотрена строка состояния и какие-либо заголовки.

Не все поля заголовка являются информационными. Некоторые поля заголовка запроса могут модифицировать запрос, ограничивая применение метода запроса в зависимости от обстоятельств. Такие поля заголовка называются модификаторами запроса. Например, клиент может не захотеть получать ответ, если он не изменился с момента последней его загрузки с сервера.

Чтобы лучше попять протокол, можно вообразить, что исходный сервер представляет собой черный ящик, содержащий ресурсы, идентифицируемые URI. Исходный сервер применяет метод запроса к ресурсу, идентифицируемому URI, и генерирует ответ. Операции по чтению ресурса из файла и записи ответа обратно клиенту «скрыты» внутри черного ящика. Подобное представление обобщает содержимое ресурса и отделяет его от ответа, посылаемого клиенту. Различные запросы с одним и тем же URI могут порождать различные ответы в зависимости от ряда факторов: полей заголовка запроса, времени запроса или имевших место изменений в ресурсе.

HTTP является протоколом прикладного уровня. HTTP может использовать любой транспортный протокол для передачи сообщения от отправителя получателю. В действительности практически все известные реализации HTTP используют в качестве протокола транспортного уровня протокол Transmission Control Protocol (TCP).

2.1.2. HTTP не сохраняет своего состояния.

HTTP представляет собой протокол, не сохраняющий своего состояния (stateless). Это означает отсутствие сохранения промежуточного состояния между парами запрос-ответ. Каждый новый запрос на ресурс инициирует отдельное применение метода запроса к URI ресурса и создание нового ответа. Компоненты, использующие HTTP, могут

и осуществляют сохранение информации о состоянии, связанной с последними запросами и ответами. Браузер, посылающий несколько запросов подряд, может отслеживать задержки ответов. Сервер может хранить информацию об IP-адресе клиента, отправившего последние десять запросов. Однако сам протокол не осведомлен о предыдущих запросах и ответах. В протоколе не предусмотрена внутренняя поддержка состояния, к нему не предъявляются такие требования. В отличие от HTTP, FTP осуществляет сохранение состояния.

Решение не поддерживать сохранение состояния в HTTP было принято на стадии разработки протокола с целью обеспечить его расширяемость. На момент появления Web существовало несколько конкурирующих систем, в которых были реализованы протоколы, не сохраняющие состояние. Также предполагалось осуществить поддержку использования конкурирующих систем для доступа к ряду ресурсов в Internet. Добавление сохранения состояния в HTTP создало бы помеху для расширяемости Web. Протокол, требующий сохранения состояния между несколькими, не связанными друг с другом соединениями, также привел бы к необходимости храпения значительных объемов информации на сервере.

По мере развития Web отсутствие сохранения состояния в HTTP стало представлять проблему для некоторых приложений. Например, приложения электронной коммерции требуют сохранения состояния между HTTP-запросами. Транзакция, состоящая из последовательности запросов и ответов, должна быть повторена полностью, если один из запросов был прерван в ходе его выполнения. Управление состоянием HTTP стало очевидной проблемой, что привело к появлению cookies.

2.1.3. Метаданные ресурса

Метаданные – это информация, относящаяся к ресурсу, но не являющаяся частью самого ресурса. Концепция метаданных возникла при разработке протокола SMTP, в котором имелась возможность определения характеристик полезной нагрузки. Метаданные могут быть включены и в запрос, и в ответ HTTP. Примерами метаданных являются: размер ресурса; тип содержания, например, `text/html`**;** время последней модификации ресурса и т.д. В зависимости от конкретного ресурса в сообщение могут включаться различные метаданные. Например, для динамически генерируемых ответов длину содержимого включать нежелательно, поскольку определение длины приведет к увеличению времени ожидания ответа. Для статических же ресурсов эта информация может быть легко получена. Точно так же включать время последней модификации имеет смысл для статического ресурса, но не для динамического ресурса. Метаданные ресурса возвращаются в ответе с помощью ряда заголовков. Присутствие некоторых метаданных для определенных запросов и ответов может быть обязательным.

Включение метаданных расширяет возможности взаимодействия между отправителями и получателями. Вот несколько вариантов применения метаданных:

- Знание информации о кодировании содержимого может облегчить ее обработку.
- Метаданные о длине содержимого могут быть использованы получателем, чтобы убедиться в получении именно того, что ожидалось.
- Сервер может указать время последней модификации ресурса, что окажет помощь при кэшировании. Кэш при этом сможет принять решение, нужно ли кэшировать ответ. Эта информация также позволит определить, насколько устарел ресурс.

3. Элементы протокола HTTP

3.1. Термины, относящиеся к HTTP.

Определения различных терминов HTTP идентичны тем, которые содержатся в двух документах RFC, описывающих HTTP/1.0 и HTTP/1.1 (соответственно RFC 1945 и RFC 2616).

3.1.1. Сообщение.

HTTP-сообщение представляет собой последовательность байтов, передаваемых но соединению транспортного уровня. Сообщение – это основная единица коммуникационного взаимодействия в HTTP. HTTP-сообщение может быть запросом, передаваемым от клиента серверу, или ответом, отправляемым сервером клиенту. Сообщение-запрос начинается со строки запроса, в то время как сообщение-ответ начинается со строки состояния. Сообщения запроса и ответа могут иметь нуль или более заголовков, отделенных от необязательного тела сообщения двумя символами: возврата каретки (CR) и перевода строки (LF).

Сообщение-запрос HTTP, показанное на рисунке 1, имеет следующий синтаксис:

```
Строка запроса
Заголовок (заголовки) общий/запроса/содержимого CRLF
Необязательное тело сообщения
```

Сообщение-запрос начинается со строки запроса, за которой следует ряд необязательных заголовков и необязательное тело заголовка. Строка запроса содержит метод запроса, запрашиваемый URI и версию протокола клиента. Например, в HTTP-запросе:

```
GET /motd HTTP/1.O
Date: Wed, 22 Mar 2017 08:09:01 GMT
Pragma: No-cache
User-Agent: Mozilla/54
```

CRLF

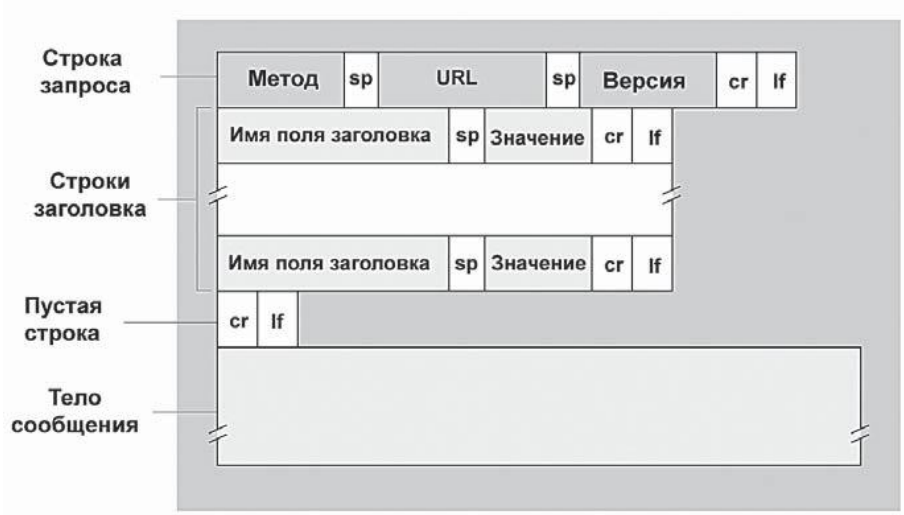

Рисунок. 1. Сообщение-запрос HTTP.

Сообщение-ответ, представленное на рисунке 2, имеет следующий синтаксис:

```
Строка состояния
Заголовок (заголовки) общий/ответа/содержимого CRLF
Необязательное тело сообщения
```

Сообщение-ответ начинается со строки состояния, которая содержит помер версии HTTP сервера и код ответа. Далее следуют необязательный общий заголовок, заголовок ответа, а также необязательное тело сообщения. Следует заметить, что из-за наличия промежуточных звеньев окончательное полученное сообщение не обязательно будет отражать номер версии протокола исходного сервера.

Рисунок. 2. Сообщение-ответ HTTP.

Сообщение не имеет тела содержимого, если отсутствует тело сообщения. Тело сообщения в запросе и ответе также называют телом запроса и телом ответа, соответственно. Не всем сообщениям разрешено иметь тело сообщения.

3.1.2. Содержимое.

HTTP/1.0 определяет содержимое как представление ресурса, которое заключено в сообщении запроса или ответа. Содержимое состоит из заголовков содержимого и необязательного тела содержимого. Заголовки содержимого состоят из метаданных о содержимом. Длина тела содержимого может быть задана в заголовке `Content-Length`**.** Очевидно, что ответ, у которого значение `Content-Length` равно пулю, не будет иметь тела содержимого. Определение содержимого несколько отличается в протоколах HTTP/1.0 и HTTP/1.1 главным образом из-за введения механизма согласования содержимого.

Тело содержимого, если оно присутствует, в известном смысле является наиболее важной частью HTTP-сообщения. Для сообщения-запроса телом содержимого могут быть данные, введенные пользователем в HTML-форму. В сообщении-ответе телом содержимого является тело сообщения, т.е. содержимое ответа без заголовков ответа.

3.1.3. Ресурс.

HTTP/1.0 определяет ресурс как «сетевой объект данных или сервис, который может быть идентифицирован унифицированным идентификатором ресурса (URI)». Термин «сетевой» означает, что объект данных или сервис могут располагаться в любой точке сети, а доступ к нему осуществляется через сетевое соединение. Решение расширить определение ресурса сервисом имело важное значение. Объект данных может быть статическим или генерироваться динамически, тогда как сервис может представлять собой любое приложение, которое просто использует Web в качестве транспортного средства для инициирования сервиса и предоставления ответа. Например, рассмотрим Web-сайт, который возвращает текущие котировки акций. Цены на акции постоянно меняются, и ресурс `http://www.cnnfii.com/quote=T` будет выдавать текущую цену акций AT&T. Сервис предоставления котировок может использовать любые внутренние механизмы для получения текущей цены акций. Он просто использует Web как механизм для передачи запроса (определенного набора акций) и ответа (текущей цены каждой из акций).

3.1.4. Агент пользователя.

Агент пользователя – это клиент, который инициирует запрос, и может быть браузером, слайдером или любым другим средством, участвующим в формировании запроса. Разница между агентом пользователя и клиентом очень важна. Обычно к инициирующему компоненту предъявляются дополнительные требования. Агент пользователя является единственной стороной, непосредственно связывающейся с пользователем, если запрос был инициирован пользователем. Агент пользователя может игнорировать сообщения об ошибке или передать их пользователю, предлагая пользователю сделать выбор: повторить запрос после неудачной аутентификации, переадресовать запрос или указать определенное альтернативное место назначения, либо изменить представление ответа. Например, агент пользователя должен определять, что запрос не содержит необходимой аутентификационной информации, на основе ответа, полученного от сервера. Агент пользователя может затем запросить у пользователя соответствующие данные и повторно отправить запрос, содержащий аутентификационную информацию.

Часто информация об агенте пользователя посылается как часть запроса в заголовке `User-Agent`. Заголовок содержит такую информацию, как названия браузера и операционной системы компьютера.

4. Методы запроса HTTP/1.0

Метод запроса уведомляет HTTP-сервер, какое действие следует выполнить над ресурсом, идентифицируемым URI запроса (т.е. URI, указываемом в строке запроса). Наиболее часто используется метод GET**,** который осуществляет выборку текущего содержимого ресурса, идентифицируемого URI. Хотя в HTTP/1.0 определены только три метода (GET, HEAD, POST)**,** в некоторых версиях клиентов и серверов, поддерживающих HTTP/1.0, были реализованы и другие методы, а именно: PUT, DELETE, LINK и UNLINK**.** Все семь методов уже применялись в различных реализациях на момент появления спецификации HTTP/1.0.

Достаточно сложно последовательно описывать методы, поля заголовков и коды ответов, поскольку любое рассмотрение одного аспекта неизбежно приводит к необходимости обращения к двум другим. На данный момент достаточно будет знать, из каких элементов состоит транзакция запрос-ответ:

- Метод запроса включается в клиентский запрос вместе с несколькими заголовками и URI.
- Метод применяется к ресурсу исходным сервером, после чего генерируется ответ. Ответ состоит из кода ответа, метаданных о ресурсе и других заголовков ответа.

Двумя важнейшими характеристиками метода являются безопасность и идемпотентность. Метод запроса, который лишь просматривает состояние ресурса (например, получает текущее содержимое), считается безопасным методом. Метод, который способен изменить состояние ресурса, не является безопасным. Идемпотентный метод, с другой стороны, имеет то свойство, что побочные эффекты для запроса являются точно такими же, что и для множества идентичных запросов. Другими словами, если последовательно выдается несколько идентичных запросов, применение метода к ресурсу либо не дает никаких побочных эффектов, либо дает одинаковый побочный эффект во всех случаях. Предсказуемость воздействия метода на ресурс является полезной для определения, должен ли ресурс поддерживать метод. Кроме того, если последовательность запросов идемпотентна, и если соединение было неожиданно закрыто, для клиента будет иметь смысл повторить набор запросов.

4.1. Метод GET.

Наиболее популярным на сегодняшний день является метод **GET.** Запрос **GET** применяется к ресурсу, задаваемому URI, а генерируемым ответом является текущее значение ресурса. Этот ответ возвращается обратившемуся с запросом клиенту. Если URI указывает на статический файл, запрос **GET** обычно приводит к чтению файла и возврату его содержимого. Если URI указывает на программу, то в теле ответа возвращаются данные (если они имеются). Метод **GET** *является* безопасным и идемпотептпым. Запрос на CGI-ресурс, например, может вызвать изменение ресурса при применении к нему метода **GET.** Поскольку такой побочный эффект соответствует *намерениям* пользователя, метод считается безопасным.

Запрос **GET** может содержать параметры, которые формируются на основе дан-пых, введенных пользователем. Это часто имеет место при запросе CGI-ресурса. Например,

```
GET http://www.altavista.com/cgi-bin/query?q=foo
```

передает пользовательскую строку запроса ("foo") ресурсу http://www.altavista.com/ cgi-bin**.**

Запрос **GET,** содержащий модификатор запроса, может привести к выполнению другого действия. Например, методу **GET** может быть предписано выбирать ресурс только в том случае, если время последней модификации запрашиваемого ресурса больше, чем значение, указанное в заголовке **If-Modified-Since.** Таким образом, запрос **GET на** ресурс **/foo.html** от клиента HTTP/1.0

```
GET /foo.html HTTP/1.0
```

может дать результат, отличный от того, который дает запрос **GET /foo.html HTTP/1.0**

```
If-Modified-Since: Sun, 12 Nov 2017 11:12:23 GMT
```

в зависимости от того, когда ресурс **/foo.html** был в последний раз модифицирован на исходным сервере. Модификатор запроса **If-Modified-Since** используется для снижения числа обращений к сети и уменьшения времени ожидания пользователем ответа, генерируемого и отправляемого исходным сервером. Если ресурс не был модифицирован с момента времени, указанного в заголовке **If-Modified-Since,** сервер отправляет код ответа, указывающий на это, не сопровождая ответ телом содержимого. Отметим, что знание времени последней модификации упрощает работу с периодически изменяемыми файловыми ресурсами. Для динамически генерируемого ресурса знание времени последней модификации особого значения не имеет. В HTTP/1.0 имеется несколько полезных модификаторов запроса. В HTTP/1.1 был добавлен еще ряд модификаторов.

Метод запроса **GET** не имеет тела запроса. Если тело запроса присутствует, серверами оно игнорируется.

4.2. Метод HEAD.

Метод **HEAD** был задуман для получения метаданных, ассоциированных с ресурсом. В результате выполнения запроса **HEAD** тело ответа не возвращается. Однако метаданные, возвращаемые сервером, могут оказаться теми же метаданными, которые были бы возвращены, если бы методом запроса был **GET.** Например, запрос **HEAD** для получения метаданных, ассоциированных с ресурсом `/foo.html`**,**

```
HEAD /foo.html HTTP/1.0
```

может вернуть

```
HTTP/1.0 200 0K Content-Length: 3219
Last-Modified: Sun, 12 Nov 2017 11:12:23 GMT
Content-Type: text/html
```

Ответ состоит из строки состояния **(HTTP/1.0 200 ОК),** указывающей на успешное выполнение запроса, и группы заголовков, представляющих метаданные для ресурса **/foo.html.** В этом примере метаданные содержат информацию о длине содержимого, времени последней модификации ресурса и типе ресурса. Метод **HEAD** безопасен и идемпотептеп.

Метод **HEAD** в основном используется при отладке для серверов с относительно малой загруженностью, а также для определения, был ли ресурс изменен с момента последней его загрузки. Метаданные могут кэшироваться или использоваться для обновления имеющихся кэшироваппых данных, если было установлено изменение ресурса. В HTTP/1.0 изменение может быть обнаружено путем анализа значений нолей заголовков **Last-Modified** или **Content-Length.** Однако в связи с тем, что ресурс может быть изменен без изменения его длины, анализ одного только ноля заголовка **Content-Length** не гарантирует обнаружения изменений.

В HTTP/1.0 метод **HEAD** нельзя использовать с модификаторами запроса, такими как **If-Modified-Since.** Это ограничение было снято в HTTP/1.1.

Метод запроса **HEAD** также не имеет тела запроса. Если тело запроса присутствует, серверами оно игнорируется.

4.3. Метод POST.

В отличие от методов **GET** и **HEAD,** которые используются для *извлечения* информации, метод **POST** применяется главным образом для модификации имеющегося ресурса или передачи данных обрабатывающему их процессу. Тело запроса содержит данные. Исходный сервер в зависимости от URI запроса, разрешает выполнение определенных действий. Метод **POST** может изменять содержимое ресурса, поэтому не может считаться безопасным методом. Поскольку побочные эффекты множества идентичных запросов могут отличаться, метод **POST** не является идемпотептпым методом.

Чтобы модифицировать ресурс, пользователь должен иметь необходимые полномочия. Не все пользователи могут обладать правами на изменение ресурса. Если пользователь имеет право на изменение ресурса, исходный сервер примет новую версию ресурса от клиента. Другое применение метода **POST** состоит в получении тела запроса сервером и использовании его в качестве входных данных для программы, идентифицируемой URI запроса. Такой программой может быть почтовая служба или менеджер доски объявлений, который создает файл, доступный другим приложениям, таким как программа для чтения электронной почты или новостей. Бывает также, что в результате выполнения запроса **POST** ресурсы не изменяются и не создаются. В этих случаях тело запроса трактуется как входные данные для программы, которая использует эти данные. Рассмотрим пример:

```
POST /foo/bar.cfm HTTP/1.0 Content-Length: 143

<тело содержимого>
```

Если принимающий сервер может успешно применить метод к ресурсу, он возвращает ответ, указывающий на успешное выполнение запроса. Предположим, что **/foo/bar.cfm** представляет собой ресурс, который не существует на исходном сервере. Сервер создаст ресурс и отправит ответ, указывающий на то, что ресурс был создан. Если, с другой стороны, **/foo/bar.cfm** является программой, ожидающей входных данных, то 143-байтное тело содержимого трактуется как входные данные для программы. Любой выход, генерируемый программой, отправляется обратно пользователю как

тело ответа. Следует иметь в виду, что заголовок **Content-Length** для запроса **POST** является обязательным, поскольку он дает возможность принимающему серверу HTTP/1.0 определить, что получен весь запрос.

Метод **GET** также может быть использован для отправки входных данных программе. Однако в использовании для этих целей методов **GET** и **POST** имеются различия. В запросе **GET** входные данные включаются в URI запроса. Предположим, что пользователь заполнил два ноля в форме для поиска: искомая строка и база данных, в которой следует вести поиск. Например, запрос

```
GET /search.cgi?string=iktinos&db=greek-architects HTTP/1.0
```

демонстрирует, как данные, введенные пользователем в форме, могут быть включены в URI запроса. Здесь запрашиваемым с помощью метода **GET** ресурсом является **search.cgi,** которому передаются значения двух нолей **(string** и **db).** При использовании метода **POST** этот же запрос выглядел бы следующим образом:

```
POST /search.cgi HTTP/1.0
Content-Length: 34
CRLF
query iktinos
db greek-architects
```

Оба запроса выдадут один и тот же ответ. Предположим, однако, что на пути между клиентом и исходным сервером имеются посредники. Посредник обычно регистрирует проходящие через пего запросы. Но в то время как URI запроса скорее всего будет занесен в журнал регистрации, тело содержимого вряд ли будет зарегистрировано. Таким образом, в случае запроса **GET** искомая строка будет занесена в журнал, а в случае запроса **POST** — пет. Некоторые посредники и серверы ограничивают длину подлежащего обработке URI, и это может стать еще одной причиной, чтобы предпочесть метод **POST** методу **GET** при передаче данных форм.

4.4. Метод PUT.

Метод **PUT** схож с методом **POST** в том, что выполнение метода обычно приводит к изменению ресурса, идентифицируемого URI запроса. Если запрашиваемый через URI ресурс не существует, он создается, а если ресурс существует, то модифицируется. При использовании метода **PUT** в результате выполнения запроса изменяется сам идентифицируемый URI ресурс.

В HTTP/1.0 метод **PUT** официально не определен. На момент выхода RFC 1945 несколько реализаций клиентов и серверов уже начали использовать этот метод, поэтому метод **PUT** (наряду с методами **DELETE, LINK, UNLINK)** был вкратце упомянут в приложении RFC 1945. Метод **PUT** не является безопасным методом. Метод **PUT** является идемпотентным, поскольку последовательность идентичных запросов **PUT** будет в каждом случае давать одно и то же содержимое, а побочные эффекты каждый раз будут одинаковыми.

4.5. Метод DELETE.

Метод **DELETE** используется для удаления ресурса, идентифицируемого URI запроса. Метод предоставляет возможность дистанционного удаления ресурсов. Однако принимая во внимание суть этого действия, исходные серверы контролируют, было ли в действительности выполнено запрашиваемое действие, и когда это произошло. Сервер может отправить ответ об успешном выполнении, в действительности не удалив ресурса. Имеется два вида ответов для успешного выполнения: один указывает на приемлемость запроса для последующей обработки, а другой указывает на реальное выполнение запроса. Подобная гибкость важна для исходных серверов при принятии решения, когда и как планировать действие, а также чтобы не приходилось держать открытым соединение с клиентом до тех пор, пока действие реально не будет завершено. Метод **DELETE** не является безопасным методом. Подобно методу **PUT,** метод **DELETE** является идемпотентным.

4.6. Методы LINK И UNLINK.

Метод **LINK** позволяет создавать связи между запрашиваемым URI и другими ресурсами. После того, как такая связь создана, можно запрашивать ресурсы но одному и тому же URI запроса. Метод **UNLINK** используется для удаления связей, созданных посредством метода **LINK.**

Хотя эти методы были определены в приложении HTTP/1.0, они не получили широкого распространения и в HTTP/1.1 отсутствуют.

5. Заголовки HTTP/1.0

Заголовок (или, точнее говоря, поле заголовка) — это ASCII-строка в свободном формате, в котором задано имя, а часто и значение. Заголовки играют важную роль в протоколе HTTP и являются основным средством для указания способа обработки запроса. Заголовки могут использоваться для предоставления метаданных о ресурсе, таких как его длина, формат кодирования и язык. Заголовки можно считать описателями ответа или запроса. Заголовок ответа может указывать, допустимо ли кэширование ответа, либо каким образом декодировать сообщение для получения исходного содержания (например, какой алгоритм сжатия был применен для его преобразования).

В противоположность фиксированному формату заголовков IP и TCP, протоколы прикладного уровня имеют более свободный формат представления заголовков. В протоколах нижних уровней размер пакетов часто ограничивается из соображений производительности. Фиксированный формат заголовков протоколов нижних уровней гарантирует, что сообщения не будут произвольно увеличиваться в результате добавления заголовков. Для протоколов прикладного

уровня такой проблемы не существует, добавление новых заголовков является типичным способом добавления новых функций.

В HTTP могут быть определены новые заголовки, которые могут иметь произвольную длину. Механизм расширяемости протокола дает возможность отражать новые идеи в виде новых заголовков. Некоторые реализации могут использовать эти новые возможности, а другие — игнорировать нераспознанные заголовки. Однако неограниченный рост числа полей заголовков и длины заголовков увеличивает общий размер сообщений и ведет к увеличению времени ожидания на стороне пользователя, а также к возрастанию загрузки сети. Большое число новых заголовков может также увеличить сложность протокола. Взаимодействие между функциями, которые зависят от различных заголовков, — существенный фактор в увеличении сложности.

HTTP-сообщение может иметь любое число заголовков, отделяемых символами CR и LF. Существуют определенные правила, связанные с передачей и интерпретацией заголовков сообщения по мере прохождения запроса или ответа в сети через серверы-посредники. Определенные типы сообщений запросов-ответов HTTP могут иметь обязательные заголовки. Большинство заголовков не являются обязательными, и Web-компоненты добавляют их по определенным причинам. **Web**-компоненты могут игнорировать и игнорируют заголовки, которые не являются обязательными. Заголовки, описанные в спецификации, должны быть попятными для всех Web-компонентов: клиентов, серверов-посредников и исходных серверов.

Заголовок имеет общий синтаксис, в соответствии с которым имя и значение отделяются друг от друга символом двоеточия Например, чтобы указать время создания сообщения, заголовок **Date** должен быть включен в сообщение примерно следующим образом:

```
Date: Thu, 23 Dec 2017 08:12:31 GMT
```

"Date" представляет собой поле имени заголовка, а строка "Thu, 23 Dec 2017 08:12:31 GMT" — поле значения. Вот пример заголовка с несколькими полями значений:

```
Accept-Language: de-CH, en-US
```

Здесь иоле имени — **Accept-Language,** а поля значений — *de-CH* и *en-US.*

В HTTP имеется иерархия заголовков. Заголовок сообщения в HTTP/1.0 является родовым именем для заголовка и может быть:

- *Общим* заголовком, используемым в сообщениях запросов и ответов.
- Заголовком *запроса,* присутствующим в сообщении-запросе для выражения предпочтительного варианта ответа, для включения дополнительной информации в запрос или для указания ограничений на обработку запроса сервером.
- Заголовком *ответа,* присутствующим в сообщении-ответе для предоставления дополнительной информации об ответе или для запроса дополнительной информации от пользователя.
- Заголовком *содержимого,* присутствующим в сообщениях запроса и ответа. Заголовки содержимого используются для предоставления информации о содержимом, например, время последнего изменения. Если поле общего заголовка или поле заголовка запроса-ответа не распознается, оно трактуется как поле заголовка содержимого.

Если заголовок не распознается получателем сообщения, он просто игнорируется. Однако если получателем является посредник, посредник должен переслать заголовок.

Хотя порядок следования различных полей заголовков не имеет значения, обычно сначала идут поля общих заголовков, потом следуют ноля заголовков запроса или заголовков ответа, а затем поля заголовков содержимого. Если для заданного ноля заголовка указано несколько полей значений, их порядок не должен изменяться посредниками, пересылающими сообщение.

5.1. Общие заголовки.

Заголовки, которые могут присутствовать в сообщениях запросов и ответов, называются *общими* заголовками. В HTTP/1.0 определены только два поля общих заголовков: **Date** и **Pragma.** Общие заголовки значимы только для самого сообщения, по *не* для содержимого, являющегося частью сообщения. Так, предположим, что запрашивается ресурс **/foo.html.** Заголовок **Date** в соответствующем сообщении-ответе указывает, что сообщение было сформировано в указанное время и не связано с тем, когда было создано или в последний раз модифицировано соответствующее содержимое.

• **Date.** Общий заголовок **Date** указывает на дату и время создания сообщения. Заголовок **Date** имеет такой же синтаксис, что и строка даты/времени в стандарте для текстовых сообщений Internet (документ RFC 822, позднее обновленный документом RFC 1123). Дата в этом формате имеет следующий вид:

```
Date: Tue 16 May 2017 11:29:32 GMT
```

Хотя это наиболее предпочтительный формат, HTTP/1.0 разрешает использование двух других форматов (определенном в документе RFC 1036 и в формате *asctime()* стандарта ANSI C). Формат RFC 1036 выглядит следующим образом:

```
Date: Tuesday, 16-May-17 11:29:32 GMT
```

Формату RFC 1036 присуща проблема, связанная с представлением года двумя цифрами. Как видно из примера, значение года трактуется как 17 вместо 2017. Неоднозначность, связанная с разрешением использования второго формата на основе RFC 1036, была впоследствии устранена в повой версии протокола (HTTP/1.1) в результате усилий по решению проблемы двухтысячного года. В соответствии с форматом **asctime()** стандарта ANSI C дата записывается следующим образом:

```
Date: Tue May 16 11:29:32 2017
```

HTTP/1.0 требует, чтобы клиенты и серверы генерировали строку данных либо в первом, либо во втором формате. Однако клиенты и серверы HTTP могут воспринимать все три формата, что необходимо по той причине, что некоторыми отправителями могут выступать приложения, не отвечающие спецификации HTTP. Спецификация протокола устанавливает правила для взаимодействия Web-компонентов друг с другом. Однако поскольку компоненты могут взаимодействовать и с не-HTTP приложениями, то должен быть определен интерфейс компонентов с ними.

• **Pragma.** Заголовок **Pragma** дает возможность отправлять директивы получателю сообщения. Директива — это способ указать для компонентов определенный вариант обработки запроса или ответа. Вообще директивы не являются обязательными для протокола, хотя в реальности есть несколько специфических директив, которым подчиняются большинство компонентов. Эта разница очень важна, поскольку, хотя протокол требует от компонентов пересылать директивы, он не обязывает компоненты следовать им, если они не являются уместными. Единственной директивой, определенной в протоколе, является

Pragma: no-cache

которая информирует серверы-посредники на маршруте следования сообщения не возвращать кэшированную копию; т.е. отправитель заинтересован в получении ответа непосредственно от исходного сервера. HTTP/1.0 не определяет назначение директивы **Pragma: no-cache** в сообщении-ответе.

5.2. ЗАГОЛОВКИ ЗАПРОСА.

Заголовок запроса может использоваться клиентом для отправки информации с запросом или задания ограничений при обслуживании запросов сервером. Передаваемая информация может содержать дополнительные сведения о клиенте, например, идентификационные данные пользователя или агента пользователя, либо информацию для авторизации, необходимую, чтобы запрос был обработан исходным сервером. В HTTP/1.0 определены пять заголовков запроса:

• **Authorization.** Заголовок **Authorization** используется агентом пользователя для включения соответствующих полномочий, необходимых для доступа к ресурсу. Для некоторых ресурсов сервера доступ разрешается только при наличии соответствующих полномочий. Вот пример заголовка **Authorization:**

```
Authorization: Basic YXZpYXRpS29IDizM1NA==
```

Здесь *Basic (обычная)* обозначает схему аутентификации, согласно которой данные представляются в виде идентификатора пользователя и пароля. Строка **YXZpYXRpS29IDizMlNA==** представляет собой кодированные идентификатор пользователя и пароль в формате Base64. Формат использует простой алгоритм кодирования и декодирования, а закодированные данные не намного длиннее исходных данных. Другие допустимые в HTTP схемы аутентификации предусматривают шифрование на транспортном уровне.

• **From.** Заголовок запроса **From** дает возможность пользователю включать в свои идентификационные данные адрес электронной почты. Это полезно для клиентских программ, работающих в качестве агентов (например, агент-робот в спайдере), для идентификации пользователя, в интересах которого функционирует программа. Вот пример заголовка **From:**

```
From: abc@mail.com
```

Следует заметить, что использование заголовка **From** нежелательно, поскольку он нарушает конфиденциальность пользователя, особенно если это делается без его ведома.

• **If-Modified-Since.** Заголовок **If-Modified-Since** является примером условного заголовка, указывающего, что запрос может быть обработан различными способами в зависимости от значения, заданного в поле заголовка. Если предыдущий ответ от сервера был кэширован клиентом или посредником, значение, заданное в заголовке *ответа* **Last-Modified,** используется в последующем запросе GET в качестве аргумента в заголовке **If-Modified-Since.** Например, для следующего запроса:

```
GET /foo.html HTTP/1.0
If-Modified-Since: Sun, 21 May 2017 07:00:25 GMT
```

сервер будет сравнивать значение, заданное в заголовке **If-Modified-Since,** с текущим значением времени последней модификации ресурса. Время последней модификации ресурса может быть доступно на уровне приложения и зависит от типа сервера. Во многих серверах это значение может быть получено с помощью системного вызова операционной системы (например, *stat()* в UNIX). Если ресурс не изменился с указанного времени **Sun, 21 May 2017 07:00:25 GMT,** сервер просто вернет ответ **304 Not Modified.** Для

резко меняющихся ресурсов это поможет избежать ненужных пересылок данных. Серверу не нужно повторно генерировать ресурс, и время ожидания на стороне пользователя снижается, поскольку клиент может получить содержимое локально.

• **Referer.** Поле заголовка **Referer** дает возможность клиенту включать URI ресурса, от которого был получен запрашиваемый URL Например, предположим, что пользователь посещает Web-страницу **http://www.cnn.com** и щелкает на гиперссылке на ресурс **http://www.disasterrelief.org/Disasters/world-glance.html** на этой странице. Заголовок **Referer** в запросе, передаваемом на **http://www.disasterrelief.org,** будет содержать строку **http://www.cnn.com:**

```
GET /Disasters/worldglance.html HTTP/1.0 Referer: http://www.cnn.com
```

Польза от применения ноля **Referer** состоит в выявлении устаревших гиперссылок. Однако чаще всего оно становится источником нарушения конфиденциальности пользователя. Поле **Referer** представляет собой заголовок,

который может быть использован исходным сервером для отслеживания действий пользователя через журнал регистрации.

Кроме того, сервер может осуществлять проверку ноля **Referer** с целью отклонения запросов на определенные ресурсы, если ссылки на ресурсы находились на страницах, не контролируемых сервером. Например, предположим, что ссылка на встроенное изображение **onlymine.gif,** изначально принадлежащее ресурсу А, было скопировано в другой документ, В. Теперь, если браузер попытается отобразить документ В, он должен сформировать запрос на ресурс **onlymine.gif** и включить в запрос заголовок **Referer** с нолем значения, соответствующим ресурсу В. Сервер может отказать в обработке запроса, поскольку поле **Referer** не содержит А.

• **User-Agent.** Поле **User-Agent** может быть использовано для включения информации о версии программного обеспечения браузера, версии операционной системы компьютера клиента и, возможно, каких-либо сведений об аппаратной конфигурации.

К ряду забавных моментов, связанных с HTTP, можно отнести орфографические ошибки, например **Referer** вместо *Referrer,* с которыми приходится мириться, поскольку они уже получили широкое распространение.

Эта информация достаточно полезна. Например, но ней можно получить статистику но используемым браузерам. Сервер может отправить альтернативную версию ресурса, если ему известно, что программное обеспечение определенного браузера не сможет отобразить версию, используемую по умолчанию. Сомнительность использования этого заголовка состоит в отслеживании действий пользователя и потенциальном вторжении в его частную жизнь. Например, предположим, что несколько пользователей, применяющих различные версии браузеров в большой системе с разделением времени, посылают запросы на исходный сервер. В этом случае иоле **User-Agent** может быть использовано для выявления пользователя, пославшего определенные запросы.

5.3. ЗАГОЛОВКИ ОТВЕТА.

Так же как заголовки запроса используются для отправки дополнительной информации о запросах, заголовки ответа применяются для отправки дополнительной информации об ответе и о сервере, создавшем ответ. Синтаксис строки состояния в заголовке строго фиксирован и не дает возможности включения дополнительной информации. Если заголовок ответа не распознан, он считается заголовком содержимого.

В HTTP/1.0 определены три заголовка ответа:

• **Location.** Заголовок **Location** используется для направления запроса в место расположения ресурса и полезен при переадресации ответов. Заголовок **Location** имеет следующий синтаксис:

```
Location: http://www.foo.com/level1/twosdown/Location.html
```

Поскольку в ответ на запрос могут быть выбраны различные варианты ресурса, заголовок **Location** предоставляет возможность идентификации местонахождения выбранного варианта. Если группа ресурсов реплицирована на нескольких зеркальных сайтах, заголовок **Location** может быть использован для указания на нужный сайт, с которого клиент должен получить ресурс. Если в результате выполнения запроса (например, **POST)** был создан новый ресурс, заголовок **Location** идентифицирует созданный ресурс.

• **Server.** Заголовок ответа **Server** аналогичен заголовку запроса **User-Agent.**

Заголовок **Server** песет информацию о версии программного обеспечения исходного сервера и любую другую информацию, связанную с его конфигурацией. Вот несколько типичных примеров заголовка **Server:**

```
Server: Apache/2.2.6 Red Hat Server: Netscape-Enterprise/3.5.1
Server: Apache/1.x.y mod_perl mod_ssl mod_foo mod_bar
```

Значение, указанное в заголовке **Server,** полезно для выявления и решения проблем в ответе, а также для сбора статистической информации, позволяющей определить наиболее популярные версии серверов. Минусом здесь является то, что, данная информация облегчает атаку на сервер. Подобно заголовку **User-Agent,** заголовок **Server** не является обязательным и может не включаться в ответы.

• **WWW-Authenticate.** Заголовок **WWW-Authenticate** используется для указания клиенту, что ресурс требует аутентификации. Клиенту возвращается ответ **401 Unauthorized,** и он может повторно обратиться с запросом, указав соответствующие данные в заголовке **Authorization.** Например, сервер может отправить такой ответ:

```
WWW-Authenticate: Basic realm="ChaseChem"
```

и клиенту следует включить в свой запрос необходимую аутентификационную информацию.

5.4. ЗАГОЛОВКИ СОДЕРЖИМОГО.

Заголовок *содержимого* используется для включения информации о теле содержимого или о ресурсе при отсутствии тела содержимого. Заголовок содержимого является принадлежностью не запроса или ответа, а конкретного ресурса, который запрашивается или отправляется. Заголовки содержимого могут иметься как в запросах, так и в ответах. Нераспознанный общий заголовок, заголовок запроса или заголовок ответа трактуется как заголовок содержимого. Другими словами, протокол задает иерархию для интерпретации полей заголовка. В сообщение запроса или ответа может быть включен новый заголовок, добавляющий новые метаданные о содержимом без опасения, что он будет интерпретирован как общий заголовок, заголовок ответа или заголовок запроса.

В HTTP/1.0 определено шесть заголовков содержимого:

• **Allow.** Заголовок содержимого **Allow** используется для указания списка доступных методов, которые могут быть применены к ресурсу. Он может использоваться и в запросах, и в ответах. Например, метод **PUT** может содержать заголовок **Allow** в запросе, перечисляющий методы, которые могут применяться к ресурсу. При получении запроса на

применение некорректного метода к ресурсу исходный сервер отправит в ответ заголовок **Allow** со списком допустимых для этого ресурса методов. Заголовок **Allow** также может быть применен в успешном ответе для указания полного списка приемлемых для данного ресурса методов.

Рассмотрим пример использования заголовка содержимого в сообщении-запросе

```
PUT /foo.html HTTP/1.0
Allow: HEAD, GET, PUT
```

Запрос информирует сервер, где будет сохранен ресурс **/foo.html,** что к этому ресурсу разрешается применять методы **HEAD, GET** и **PUT.** Другие методы запроса, которые могут попытаться применить клиенты, не должны допускаться исходным сервером.

• **Content-Type.** Заголовок **Content-Type** указывает на тип представления тела содержимого, например, **image/gif, text/html** или **application/x-javascript.** Метод **POST** может включать форму со следующим заголовком **Content-Type.**

```
POST /chat/chatroom.cgi HTTP/1.0 User-Agent: Mozilla/54
Content-Type: application/x-www-form-urlencoded
```

Различные типы представления содержимого, используемые в **Content-Type,** должны быть зарегистрированы организацией IANA.

• **Content-Encoding.** Заголовок **Content-Encoding** указывает, как было модифицировано представление ресурса, и как оно может быть декодировано в формат, указанный в заголовке содержимого **Content-Type.** Кодирование содержимого выполняется над документами для преобразования их без потерь информации. Типичным примером такого преобразования является сжатие. Чтобы преобразовать ответ обратно к его исходному виду, получатель должен располагать соответствующей технологией декодирования. Сообщение, содержимое которого было сжато с использованием *gzip,* может содержать заголовок

Content-Encoding: x-gzip

По мере регистрации IANA могут быть определены новые типы содержания, согласно документу RFC 1590.

• **Content-Length.** Заголовок **Content-Length** указывает на длину тела содержимого в байтах. Длима содержимого важна для обеспечения полной доставки отправленного тела содержимого получателю. Значение **Content-Length** может использоваться в качестве *валидатора* для сравнения кэшированного содержания с текущей его версией. Без заголовка **Content-Length** в запросе клиенту пришлось бы закрывать соединение, чтобы указать на завершение передачи запроса. Если ответ генерируется динамически, весь ответ должен быть сформирован до того, как станет известна длина содержимого. Поскольку длина содержимого **Content-Length** является значением ноля заголовка и должна быть вставлена в сообщение-ответ перед телом ответа, время ожидания на стороне получателя увеличивается. Обычно для динамических ответов заголовок **Content-Length** опускается. Если динамически сгенерированный ответ был буферизирован перед отправкой, длима содержимого может быть известна. В HTTP/1.0 в качестве индикатора конца содержимого используется закрытие соединения.

• **Expires.** Заголовок **Expires** предоставляет отправителю возможность заявить, что содержимое должно считаться устаревшим после истечения времени, указанного в заголовке. Клиент не должен кэшировать ответ позже даты, заданной в заголовке **Expires.** Разница между *хранением* ресурса в кэше и *кэшированием* весьма существенна. Ответ может храниться (т.е. удерживаться) в кэше и по истечении срока храпения, но он не может быть *возвращен* как ответ без проверки его актуальности на исходным сервере. **Expires** является заголовком содержимого, а не заголовком ответа по той причине, что истечение времени хранения ассоциируется с ресурсом, а не с сообщением. Сервер может отправить сообщение

```
HTTP/1.0 200 0K
Server: Microsoft-IIS/4.0
Date: Mon, 04 Dec 2017 18:16:45 GMT
Expires: Tue, 05 Dec 2017 18:16:45 GMT
```

указывающее, что время храпения ресурса составляет один день.

• **Last-Modified.** Заголовок **Last-Modified** указывает время последнего изменения ресурса. Если ресурсом является статический файл, то этим значением может быть дата последнего изменения ресурса в файловой системе. Динамически генерируемый ресурс может иметь время **Last-Modified,** совпадающее со временем создания сообщения-ответа сервером. Фактически время **Last-Modified** никогда не может быть большим, чем время создания сообщения. Протокол не определяет правила для вычисления времени истечения срока хранения. Заголовок **Last-Modified** в ответе подсказывает получателю сравнить это значение со значением **Last-Modified** кэшированной версии, чтобы проверить, не устарел ли кэшированный ресурс. Наличие заголовка **Last-**

Modified трактуется некоторыми кэшами в HTTP/1.0 как указание, что содержимое *не* было сгенерировано динамически, поскольку заголовок **Last-Modified** для динамически сформированного содержимого не имеет особого смысла.

Сервер может отправить сообщение, подобное следующему:

```
HTTP/1.0 200 0K
Date: Sun, 21 May 2017 08:09:12 GMT
Last-Modified: Sun, 21 May 2017 07:00:25 GMT
```

6. Классы ответов HTTP/1.0.

Каждое сообщение-ответ HTTP начинается со строки состояния, которая имеет три ноля: номер версии протокола сервера, код ответа и поясняющая фраза на естественном языке. Запрос на сервере может быть успешно обработан, вызвать отказ в обработке, либо быть переадресованным на другой сервер. В сообщении-запросе могут содержаться синтаксические ошибки, могут возникнуть проблемы при обработке запроса сервером. Различные виды ответов группируются в *классы ответов.* В HTTP каждый из пяти классов ответов имеет несколько кодов ответов, каждый из которых выражается трехзначным целым числом. К пяти классам ответов относятся: информационный класс, коды ответов для которого начинаются с 1 (записываются как 1xx); класс успешного выполнения запроса (2xx); класс переадресации (3xx); класс ошибок клиента (4xx); класс ошибок сервера (5xx).

Идея классов ответов была заимствована у почтового протокола SMTP. Протокол FTP имеет схожий механизм кодов ответов. Изначально в HTTP имелось только четыре класса, однако позднее был добавлен информационный класс (1xx). Выбор первоначальных четырех классов ответов был в определенном смысле произвольным, поскольку протокол HTTP отличается от почтового протокола. Однако адаптация хорошо известного и понятного механизма послужила хорошей отправной точкой. Хотя теоретически протокол допускает создание дополнительных классов, на практике имеющиеся реализации могут недостаточно четко использовать новые классы ответов. Правильно написанная реализация будет трактовать код ответа, принадлежащий неизвестному классу ответов, как ошибку.

В то время как коды ответов стандартизованы, строки состояния *не* являются стандартизованными. В различных программах могут использоваться различные строки, что не оказывает влияния на интерпретацию кода ответа. Например, ответ **404 Not Found** (Не найдено) означает то же самое, что **404 Missing Resource** (Ресурс отсутствует)**.**

Следует заметить, что хотя не все *коды* ответов понятны для всех HTTP-приложений, *класс* ответа, к которому принадлежит код, должен быть попятным. Класс определяется первой цифрой кода. Каждый класс ответа x имеет x00 в качестве ответа по умолчанию. Если приложение получает код ответа, который ему непонятен, оно ведет себя так, как если бы получило код ответа по умолчанию. Предположим, что сервер-посредник HTTP/1.0 получает код ответа **206 Partial Content,** который был определен в HTTP/1.1. Сервер-посредник HTTP/1.0 может не знать, как должным образом интерпретировать такой ответ. Однако посредник должен трактовать его как код ответа **200 OK** и не должен кэшировать ответ. HTTP-приложение, такое как браузер или сервер-посредник, должно вести себя *предсказуемо,* даже если получает неопознанный код ответа. Заметим, что такой образ поведения неявным образом запрещает добавление новых классов кодов ответов, поскольку старые HTTP-приложения не будут знать, как интерпретировать коды ответов для нового класса.

7. Cookies.

HTTP не сохраняет свое состояние, поэтому Web-серверу не нужно хранить какую-либо информацию о прошлых или будущих запросах. Однако на стороне сервера может иметься существенная причина для сохранения информации о состоянии между запросами в ходе сеанса или даже между сеансами. Например, может оказаться необходимым предоставлять доступ к ряду страниц на сервере только определенной группе пользователей. Если ввод идентификационной информации необходим при каждом обращении пользователя к любой из этих страниц, возникает дополнительная нагрузка как на пользователя (ему необходимо вводить эту информацию), так и на сервер (для обработки этой информации). Дополнительные транзакции также снижают пропускную способность сети. Сохранение определенной информации между запросами сокращает непроизводительные затраты для пользователя, сети и сервера. Браузер играет важную роль в предоставлении необходимой информации о состоянии вместе с пользовательским запросом.

Cookies являются одним из средств сохранения состояния HTTP. Cookie — это информация о состоянии, которая передается Web-сервером браузеру и хранится на компьютере пользователя в интересах сервера. Механизм работы с информацией о состоянии HTTP дает возможность клиентам и серверам сохранять информацию за пределами запроса и ответа на него. Cookies впервые были применены Netscape в 1994 г. Затем организацией Internet Engineering Task Force (IETF) был начат процесс стандартизации. Механизм работы с состоянием HTTP, формализующий использование cookies, подробно описан в документе RFC 2965, выпущенном в октябре 2000 г. и представляющем собой предложение по стандарту (Proposed Standard) IETF.

7.1. Причины использования cookies.

Сервер передает cookie браузеру вместе со своим ответом, требуя от браузера включать cookie в последующие запросы к серверу. Когда пользователь в следующий раз посещает Web-сайт, браузер включает информацию из cookie в заголовок запроса. Таким образом, Web-сервер способен различать пользователей в ходе сеанса и между различными сеансами. Информация, отправленная в cookie, может быть уникальной для каждого посетителя Web-сайта, что создает условие для индивидуального обслуживания пользователей.

Многие Web-сайты (например, The New York Times, **http://www.nytimes.com)** требуют, чтобы cookies использовались браузером при загрузке страниц. Сайты могут требовать, чтобы пользователи идентифицировали себя именами и паролями. Если это необходимо делать при каждой загрузке страниц с этого сайта, пользователь вряд ли захочет работать с таким сайтом. Вместо этого необходимая идентификационная информация может передаваться автоматически через cookie.

Типичный пример использования cookie — «магазинная тележка», в которую пользователь помещает купленные им товары. Имеются Web-сайты, на которых пользователи могут выбирать товары, которые они планируют купить, например, книги или компакт-диски. По мере выбора пользователем товаров виртуальная магазинная тележка содержит множество выбранных на данный момент товаров.

Без cookies Web-серверу пришлось бы сохранять состояние всех своих пользователей (их может быть тысячи) на своем компьютере в течение длительного периода времени. Содержимое магазинной тележки или перечень приобретенных в последнее время товаров являются примерами информации о состоянии. В других случаях может сохраняться более подробная информация, например, все купленные пользователем товары или страницы, которые пользователь посетил в ходе последнего сеанса. Реальное состояние не обязательно сохраняется в cookies полностью; cookies часто используются в качестве индекса для базы данных, в которой Web-сервером сохраняется информация о состоянии пользователя. Сохранение информации о пользователе дает возможность приложению совместно использовать ее с другими приложениями, в том числе с другими серверами. Подобное совместное использование информации без ведома пользователя ведет к угрозе конфиденциальности пользователя.

7.2. Использование cookies в браузере

На рисунке 3 представлен клиент, отправляющий запрос исходному серверу А (этап 1). Исходный сервер в ответ включает заголовок **(Set-Cookie)** со значением cookie **(XYZ)** (этап 2). Во все последующие запросы к исходному серверу А клиент включает cookie (этап 3, передача **Cookie** с запросом в заголовке).

Заметим, что клиент не интерпретирует строку cookie (XYZ) при ее сохранении и включении в последующие запросы. Сервер свободен в построении строки, предназначенной клиенту, сформировавшему запрос, и в изменении механизма построения строки cookie. На рисунке также показано, что обмен cookies фактически осуществляется без ведома пользователя, если только пользователь не потребовал уведомлять его каждый раз при передаче cookie. Такое уведомление мешает работе пользователя, поэтому лишь малая часть пользователей, осведомленных об этой возможности, применяет ее на практике.

Cookies первоначально хранятся в оперативной памяти компьютера пользователя и записываются на долговременное устройство храпения (файлы па диске) при выходе из браузера. В соответствии с указаниями но реализации агенты пользователя могут хранить часто используемые cookies столько, сколько нужно, однако существует ряд ограничений. Cookie может иметь размер до 4 Кбайтов. Браузеры дают возможность сохранять максимум 20 cookies на сервер (или домен) и не более 300 cookies, чтобы избежать перегрузки.

Рисунок. 3 Обмен cookies между клиентом и сервером

7.3. Контроль пользователя над cookies.

Как и другие семантические свойства, управление которыми может осуществляться в браузере, cookies допускают значительную степень контроля со стороны пользователя. Пользователи могут:

• **Решать, принимать ли какие-либо cookies вообще.** Подобное решение может сделать для пользователя невозможным загрузку страниц с некоторых сайтов.

• **Устанавливать ограничения на размер и количество принимаемых cookies.**

Тем самым пользователь управляет пространством, которое будет выделено для cookies на его компьютере, и уменьшает вероятность размещения произвольно больших cookies.

• **Решать, принимать ли cookies от всех сайтов, или только от определенных сайтов/доменов:** Это дает возможность пользователю принимать cookies с нужных сайтов и отклонять cookies с других сайтов.

• **Ограничить время жизни cookies только данным сеансом.** Более детальное управление на уровне сеанса дает возможность пользователю разрешать прием cookies только для выполнения определенной задачи. В конце сеанса осуществляется возврат к режиму по умолчанию, в соответствии с которым cookies для последующих сеансов не принимаются.

• **Потребовать, чтобы cookies исходили от того же сервера, с которого была получена текущая страница.** Тем самым гарантируется, что пользователь будет знать, откуда поступили cookies, а другим сайтам, с которыми браузер мог автоматически связаться, будет запрещено посылать cookies. Например, когда браузер загружает контейнерный документ, встроенные изображения могут извлекаться автоматически. Встроенные изображения могут размещаться на сервере, отличном от того, на котором находится контейнерный документ.

Реализация Web сервера (Apache) на базе ОС FreeBSD UNIX

1. Загружаем дистрибутив `httpd-2.2.11.tar.gz` в каталог `/usr/local/src`.
2. Переходим в каталог `/usr/local/src`:

```
# cd /usr/local/src
```
3. Распаковываем архив:
```
# tar –xvf httpd-2.2.11.tar.gz
```
4. Переходим в созданный каталог:
```
# cd httpd-2.2.11
```
5. Собираем и устанавливаем приложение:
```
# ./configure --prefix=/usr/local/apache2 --enable-so
# make
# make install
```
6. Загружаем дистрибутив `php-5.2.9.tar.gz` в каталог `/usr/local/src`.
7. Переходим в каталог `/usr/local/src`:
```
# cd /usr/local/src
```
8. Распаковываем архив:
```
# tar –xvf php-5.2.9.tar.gz
```
9. Переходим в созданный каталог:
```
# cd php-5.2.9
```
10. Собираем и устанавливаем приложение:
```
# ./configure --with-apxs2=/usr/local/apache2/bin/apxs --disable-libxml --disable-dom --disable-simplexml --disable-xml --disable-xmlreader --disable-xmlwriter --without-pear
# make
# make install
```
11. Копируем конфигурационный файл:
```
# cp php.ini-dist /usr/local/lib/php.ini
```
12. Добавляем строчку `AddType application/x-httpd-php .php .phtml` в конфигурационный файл `/usr/local/apache2/conf/httpd.conf`
13. Запускаем сервис:
```
/usr/local/apache2/bin/apachectl start
```
14. В корне Web сервера создаем файл `index.php`, содержащий код:
```
<?php
phpinfo();
?>
```

Задание на работу

1. Установить и настроить сервер Apache с модулем PHP. Создать виртуальные серверы согласно варианту.

2. Создать центр сертификации, подписать сертификаты для виртуальных серверов, в браузере в список доверяемых добавить сертификат центра сертификации, включить поддержку HTTPS для виртуальных серверов.

3. Проверить работу Web сервера с помощью стандартного клиента и утилиты telnet/openssl.

Варианты заданий.

№ варианта	Имя сервера	DNS суффикс	Порт	Корневой каталог	Ограничение доступа
1	alpha	zone01.com.ua	80	/web	-
	srv-01		8000	/usr/local/www	mercury, venus
2	mercury	zone02.net.ua	8080	/www	-
	srv-02		80	/usr/local/web	tiger, lion
3	tiger	zone03.kiev.ua	8088	/usr/home/web	-
	srv-03		80	/http	alpha, beta
4	rose	zone04.com	8000	/usr/home/www	-
	srv-04		8080	/usr/www	earth, saturn
5	beta	zone05.net	8080	/usr/home/http	-
	srv-05		8088	/usr/web	lynx, leopard
6	venus	zone06.org.ua	80	/usr/local/http	-
	srv-06		8000	/www	rose, tulip
7	lion	zone07.org	8080	/usr/http	-
	srv-07		80	/web	gamma, delta
8	tulip	zone08.edu	8088	/usr/apache	-
	srv-08		80	/usr/home/web	jupiter, mercury
9	gamma	zone09.org	8000	/http	-
	srv-09		8080	/usr/home/www	jaguar, tiger

№ варианта	Имя сервера	DNS суффикс	Порт	Корневой каталог	Ограничение доступа
10	earth	zone10.org.ua	8080	/usr/local/http	-
	srv-10		8088	/usr/home/http	narcissus, aster
11	lynx	zone11.net	80	/usr/home/http	-
	srv-11		8000	/web	omega, alpha
12	narcissus	zone12.com	8080	/usr/home/web	-
	srv-12		80	/usr/apache	venus, earth
13	delta	zone13.kiev.ua	8088	/web	-
	srv-13		80	/usr/home/http	lion, lynx
14	saturn	zone14.net.ua	8000	/www	-
	srv-14		8080	/usr/local/http	peony, rose
15	leopard	zone15.com.ua	8080	/usr/home/www	-
	srv-15		8088	/usr/web	beta, gamma

В столбце «Ограничение доступа» указаны пользователи, которым разрешен доступ к ресурсам Web сервера после аутентификации, «-» означает доступ без аутентификации с любого узла.

Литература

1. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 4-е изд. – СПб.: Питер, 2010. – 944 с.

2. Стивенс У.Р. Протоколы TCP/IP. Практическое руководство/ Пер. с англ. и коммент. А.Ю. Глебовского. – СПб.: «Невский диалект» – «БХВ–Петербург», 2003. – 672 с.: ил.