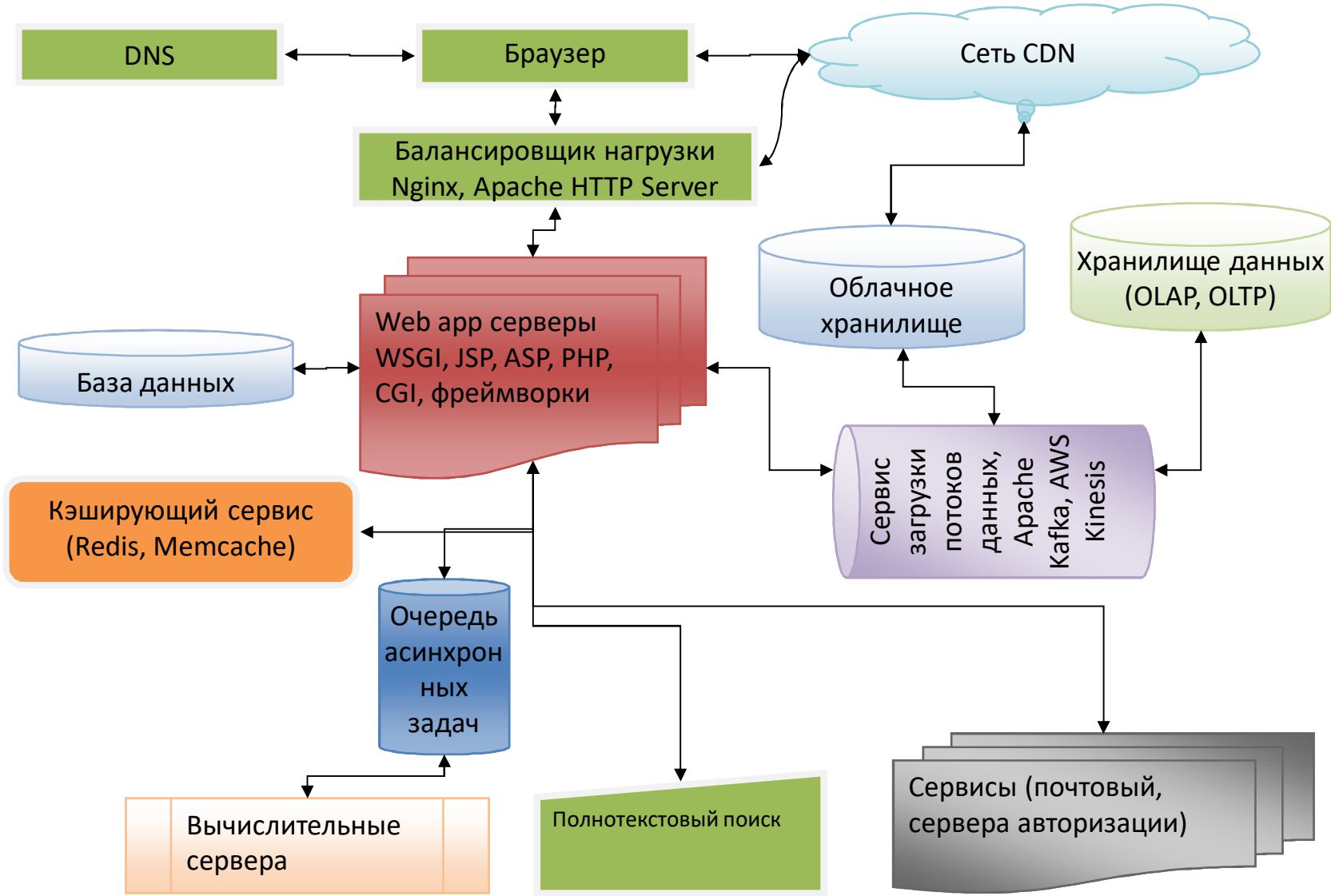
Современная архитектура веб-сервиса

Суханов А.Я.

Типичная монолитная архитектура

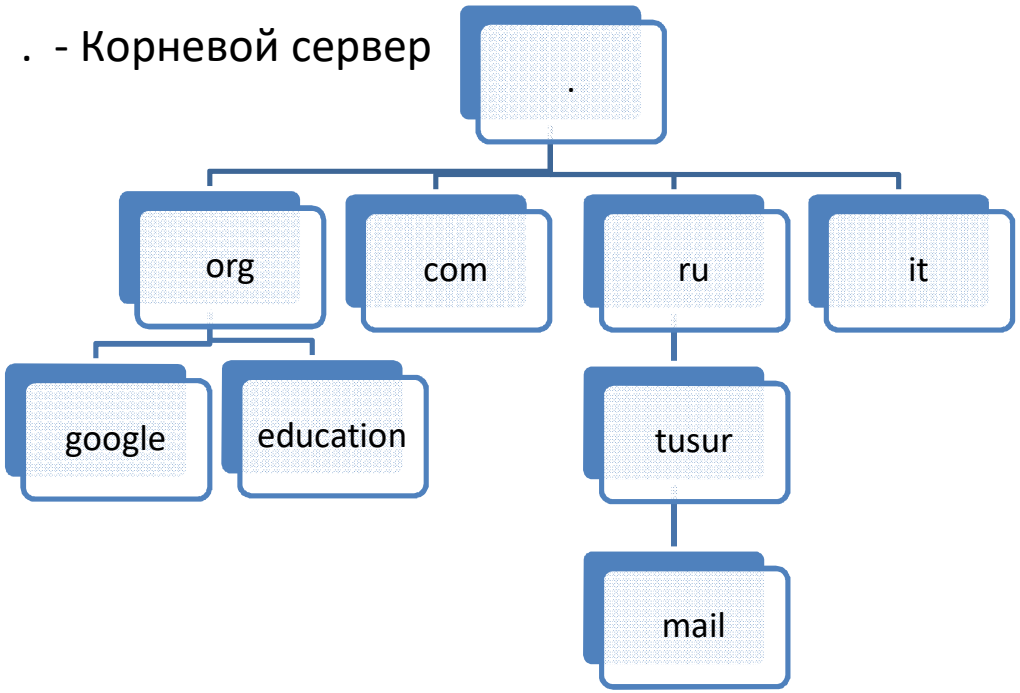


DNS

- DNS (англ. Domain Name System «система доменных имён») — компьютерная распределённая система для получения информации о доменах. Чаще всего используется для получения IP-адреса по имени хоста (компьютера или устройства), получения информации о маршрутизации почты и/или обслуживающих узлах для протоколов в домене (SRV-запись).
- Распределённая база данных DNS поддерживается с помощью иерархии DNS-серверов, взаимодействующих по определённому протоколу.
- Основой DNS является представление об иерархической структуре имени и зонах. Каждый сервер, отвечающий за имя, может делегировать ответственность за дальнейшую часть домена другому серверу (с административной точки зрения — другой организации или человеку), что позволяет возложить ответственность за актуальность информации на серверы различных организаций (людей), отвечающих только за «свою» часть доменного имени.
- Начиная с 2010 года в систему DNS внедряются средства проверки целостности передаваемых данных, называемые DNS Security Extensions (DNSSEC). Передаваемые данные не шифруются, но их достоверность проверяется криптографическими способами. Внедряемый стандарт DANE обеспечивает передачу средствами DNS достоверной криптографической информации (сертификатов), используемых для установления безопасных и защищённых соединений транспортного и прикладного уровней.

Иерархия доменных имен и серверов

. - Корневой сервер



DNS записи хранятся как в соответствующих узлах иерархии, так и кэшируются локальными DNS серверами, например, серверами провайдера. Время хранения кэша присутствует в ответах DNS. Бывает два типа запроса – рекурсивный и итеративный. Рекурсивный отправляется серверу и он сам разрешает доменное имя обращаясь к ответственным серверам по иерархии. Итеративный запрос отправляет адрес следующего ответственного севера к которому нужно обратиться самостоятельно.

Некоторые ресурсные записи DNS

- A (address record/запись адреса) - адресная запись, которая указывает на соответствие между доменным именем и IP-адресом (IPv4). На VDS вы можете настроить несколько A-записей, если, например, у вас несколько веб-серверов, обрабатывающих запросы для одного домена. Такая запись называется round-robin. В этом случае преобразование доменного имени в IP-адрес производится в произвольном порядке с равной вероятностью распределения.
- AAAA (IPv6 address record) - аналогична записи A, но указывает соответствие доменного имени для IPv6.
- MX (mail exchange) - запись, указывающая на адрес почтового шлюза для домена. Состоит из двух частей: приоритета (чем число больше, тем ниже приоритет) и адреса узла.
- NS (name server/сервер имен) - указывает на DNS-сервер, обслуживающий данный домен, т.е. указывает серверы, на которые домен делегирован. Данный тип записи критически важен для функционирования самой системы доменных имён.
- PTR (pointer) — запись, которая "связывает" IP-адрес с доменным именем. Многие почтовые серверы при фильтрации входящей почты от спама проверяют наличие PTR-записи и ее соответствие имени сервера-отправителя.
- Для определения имени сервера по его IPv4-адресу существует специальная доменная зона in-addr.arpa. Если для адреса задана PTR-запись, то IP-адрес хоста, например, 92.53.96.111, будет транслирован в обратной нотации и преобразован в 111.96.53.92.in-addr.arpa.

Клиент-Сервер

- Вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных (например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных) или в виде сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине). Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, её размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики её оборудования и программного обеспечения, её также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

Тонкий и толстый клиент

- *Тонкий клиент* — это компьютер-клиент, который переносит все задачи по обработке информации на сервер. Примером тонкого клиента может служить компьютер с *браузером*, использующийся для работы с *веб-приложениями*.
- *Толстый клиент*, напротив, производит обработку информации независимо от сервера, использует последний в основном лишь для хранения данных.

Многоуровневая архитектура

- Разновидность архитектуры «клиент — сервер», в которой функция обработки данных вынесена на один или несколько отдельных серверов. Это позволяет разделить функции хранения, обработки и представления данных для более эффективного использования возможностей серверов и клиентов.

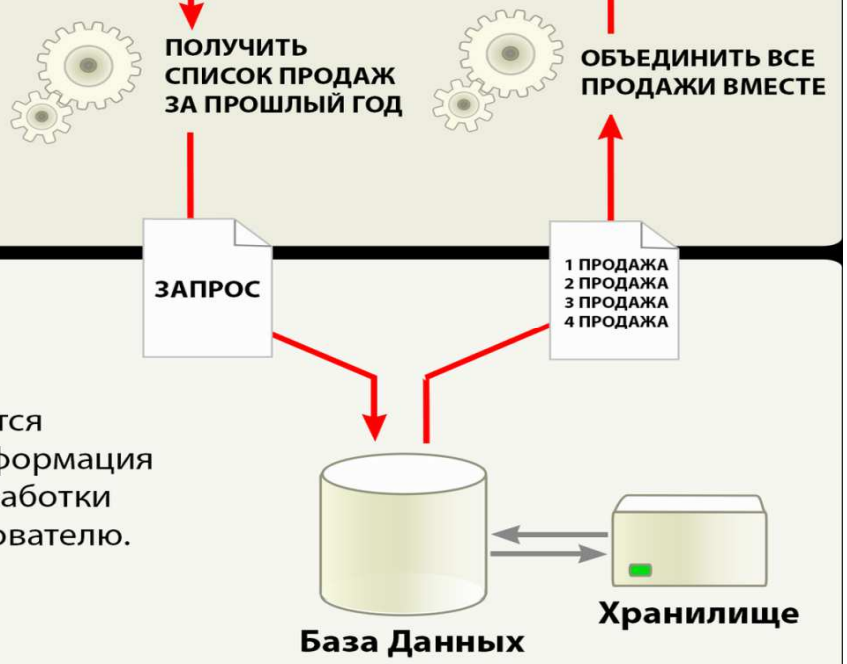
Трехуровневая архитектура

Слой клиента

Самый верхний уровень приложения с интерфейсом пользователя. Главная функция интерфейса представление задач и результатов,понятных пользователю.

Слой логики

Этот слой координирует программу, обрабатывает команды, выполняет логические решения и вычисления, выполняет расчеты. Она также перемещается и обрабатывает данные между двумя окружающими слоями.



Слой данных

Здесь хранится информация и извлекается из базы данных и файловой системы. Информация отправляется в логический слой для обработки и в конечном счете возвращается пользователю.

Трехуровневая архитектура

- Трёхуро́вневая архитекту́ра (трёхзве́нная архитекту́ра, англ. three-tier) — архитектурная модель программного комплекса, предполагающая наличие в нём трёх компонентов: клиента, сервера приложений (к которому подключено клиентское приложение) и сервера баз данных (с которым работает сервер приложений).
-
- По сравнению с двухзвенной клиент-серверной архитектурой или файл-серверной архитектурой трёхуровневая архитектура обеспечивает, как правило, бо́льшую масштабируемость (за счёт горизонтальной масштабируемости сервера приложений и мультиплексирования соединений), бо́льшую конфигурируемость (за счёт изолированности уровней друг от друга). Реализация приложений, доступных из веб-браузера или из тонкого клиента, как правило, подразумевает развёртывание программного комплекса в трёхуровневой архитектуре.

Сокеты

- Сокетом является абстрактная единица сетевого взаимодействия, фактически переводится как разъем. Можно провести с файловой переменной, но в данном случаем с сокетом связано не имя файла и тип (бинарный, текстовый) или режим (чтение, запись), а адрес удаленного приложения, тип соединения и передачи данных, протокол взаимодействия. Обычно рассматриваются на транспортном уровне взаимодействия.
- Сокеты предоставляют весьма **мощный и гибкий механизм межпроцессного взаимодействия (IPC)**. Они могут использоваться для организации взаимодействия программ на одном компьютере, по локальной сети или через Internet, что позволяет вам создавать распределённые приложения различной сложности. Кроме того, с их помощью можно организовать взаимодействие с программами, работающими под управлением различных операционных систем.

- **Сокеты поддерживают многие стандартные сетевые протоколы (конкретный их список зависит от реализации) и предоставляют унифицированный интерфейс для работы с ними**. Наиболее часто сокеты используются для работы в IP-сетях.

- <!-- _footer: Шаргин А. Программирование сокетов в Linux. [Электронный ресурс]. URL: https://rsdn.org/article/unix/sockets.xml (дата обращения: 23.03.2020)-->

Сокет на python

python client.py
Или
pypy client.py

Сервер

```
import socket

sock = socket.socket()
sock.bind(('', 9090))
sock.listen(1)
conn, addr = sock.accept()

print 'connected:', addr

while True:
    data = conn.recv(1024)
    if not data:
        break
    conn.send(data.upper())

conn.close()
```

Клиент

```
import socket

sock = socket.socket()
sock.connect(('localhost',
9090))
sock.send('hello, world!')

data = sock.recv(1024)
sock.close()

print data
```

```
from socket import *
host = 'localhost'
port = 777
addr = (host,port)
#socket - функция создания сокета
#первый параметр socket_family может быть AF_INET или AF_UNIX
#второй параметр socket_type может быть SOCK_STREAM(для TCP)
или SOCK_DGRAM(для UDP)
udp_socket = socket(AF_INET, SOCK_DGRAM)
#bind - связывает адрес и порт с сокетом
udp_socket.bind(addr)
#Бесконечный цикл работы программы
while True:
    #Если мы захотели выйти из программы
    question = input('Do you want to quit? y\\n: ')
    if question == 'y': break
    print('wait data...')
    #recvfrom - получает UDP сообщения
    conn, addr = udp_socket.recvfrom(1024)
    print('client addr: ', addr)
    #sendto - передача сообщения UDP
    udp_socket.sendto(b'message received by the server', addr)
udp_socket.close()
```

```
from socket import *
import sys

host = 'localhost'
port = 777
addr = (host,port)

udp_socket = socket(AF_INET, SOCK_DGRAM)
data = input('write to server: ')
if not data :
    udp_socket.close()
    sys.exit(1)
#encode - перекодирует введенные данные в байты, decode -
обратно
data = str.encode(data)
udp_socket.sendto(data, addr)
data = bytes.decode(data)
data = udp_socket.recvfrom(1024)
print(data)


udp_socket.close()
```

Java – сокет сервер

```
import java.io.*; import java.net.*;
class SampleServer extends Thread
{
    Socket s;
    int num;
    public static void main(String args[])
    {
        try
        {  int i = 0; // счётчик подключений
            // привинтить сокет на локалхост, порт 3128
            ServerSocket server = new ServerSocket(3128, 0,
                    InetAddress.getByName("localhost"));
            System.out.println("server is started");
            // слушаем порт
            while(true)
            {// ждём нового подключения, после чего запускаем обработку
             клиента в новый вычислительный поток и увеличиваем счётчик
                                                          на единичку
                new SampleServer(i, server.accept());
                i++;
            }
        }
        catch(Exception e)    {System.out.println("init error: "+e);}
  – // вывод исключений
    }
```

```
public SampleServer(int num, Socket s)
    {
        // копируем данные
        this.num = num;
        this.s = s;

        // и запускаем новый вычислительный поток (см. ф-ю run())
        setDaemon(true);
        setPriority(NORM_PRIORITY);
        start();
    }
```

```
public void run()
    {
        try
        {
            // из сокета клиента берём поток входящих данных
            InputStream is = s.getInputStream();
            // и оттуда же - поток данных от сервера к клиенту
            OutputStream os = s.getOutputStream();
            // буффер данных в 64 килобайта
            byte buf[] = new byte[64*1024];
            // читаем 64кб от клиента, результат - кол-во реально
принятых данных
            int r = is.read(buf);
            // создаём строку, содержащую полученную от клиента
информацию
            String data = new String(buf, 0, r);
            // добавляем данные об адресе сокета:
            data = ""+num+": "+"\n"+data;
            os.write(data.getBytes()); // выводим данные
            s.close(); // завершаем соединение
        }
        catch(Exception e)
        {System.out.println("init error: "+e);} // вывод исключений
    }
}
```

```
import java.io.*; import java.net.*;
class SampleClient extends Thread
{
    public static void main(String args[])
    {
        try
        {   // открываем сокет и коннектимся к localhost:3128,
получаем сокет сервера
            Socket s = new Socket("localhost", 3128);
            // берём поток вывода и выводим туда первый аргумент
            // заданный при вызове, адрес открытого сокета и его порт
            args[0] =
args[0]+"\n"+s.getInetAddress().getHostAddress()
+":"+s.getLocalPort();
            s.getOutputStream().write(args[0].getBytes());
            // читаем ответ
            byte buf[] = new byte[64*1024];
            int r = s.getInputStream().read(buf);
            String data = new String(buf, 0, r);
            // выводим ответ в консоль
            System.out.println(data);
        }
        catch(Exception e)
        {System.out.println("init error: "+e);} // вывод исключений
    }
}
```

- Компилируем javac Server
- Компилируем javac Client
- Запускаем сервер
- java Server

- а потом, дождавшись надписи "server is started", и любое количество клиентов:

- java Client test1
- java Client test2
- ...
- java Client testN

HTTP

- HTTP (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных изначально — в виде гипертекстовых документов в формате «HTML», в настоящий момент используется для передачи произвольных данных. Основой HTTP является технология «клиент-сервер», то есть предполагается существование:
- Потребителей (клиентов), которые инициируют соединение и посылают запрос;
- Поставщиков (серверов), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.
- HTTP в настоящее время повсеместно используется во Всемирной паутине для получения информации с веб-сайтов. В 2006 году в Северной Америке доля HTTP-трафика превысила долю P2P-сетей и составила 46 %, из которых почти половина — это передача потокового видео и звука.
- HTTP используется также в качестве «транспорта» для других протоколов прикладного уровня, таких как SOAP, XML-RPC, WebDAV.

HTTP

- В отличие от многих других протоколов, HTTP является протоколом без памяти. Это означает, что протокол не хранит информацию о предыдущих запросах клиентов и ответах сервера.
- Компоненты, использующие HTTP, могут самостоятельно осуществлять сохранение информации о состоянии, связанной с последними запросами и ответами.
  - Клиентское веб-приложение, посылающее запросы, может отслеживать задержки ответов.
  - Сервер может хранить IP-адреса и заголовки запросов последних клиентов.

Протокол http

- Всё программное обеспечение для работы с протоколом HTTP разделяется на три основные категории:
  - Серверы - поставщики услуг хранения и обработки информации (обработка запросов).
  - Клиенты — конечные потребители услуг сервера (отправка запросов).
  - Прокси-серверы для поддержки работы транспортных служб.

Протокол http

- Основными *клиентами* являются *браузеры* например: Google Chrome, *Opera*, *Mozilla Firefox*, и др.
- Наиболее известными реализациями *веб-серверов* являются: *Internet Information Services* (IIS), *Apache*, *lighttpd*, *nginx*.
- Наиболее известные реализации *прокси-серверов*: *Squid*, *UserGate*, *Multiproxy*, *Naviscope*.

"Классическая" схема HTTP-сеанса

1. Установление TCP-соединения.
2. Запрос клиента.
3. Ответ сервера.
4. Разрыв TCP-соединения.

- Таким образом, клиент посылает серверу запрос, получает от него ответ, после чего взаимодействие прекращается.
- Обычно запрос клиента представляет собой требование передать HTML-документ или какой-нибудь другой ресурс, а ответ сервера содержит код этого ресурса.

Структура протокола http

- Каждое HTTP-сообщение состоит из трёх частей, которые передаются в указанном порядке:
  - *Заголовок сообщения*, который начинается со *строки состояния*, определяющей тип сообщения, и *полей заголовка*, характеризующих тело сообщения, описывающих параметры передачи и прочие сведения;
  - *Пустая строка*;
  - *Тело сообщения* — непосредственно данные сообщения.
- *Поля заголовка* и *тело* сообщения могут отсутствовать, но *строка состояния* является обязательным элементом, так как указывает на тип запроса/ответа.

Структура запроса клиента http

Методы запроса клиента

- Метод, указанный в строке состояния, определяет способ воздействия на ресурс, URL которого задан в той же строке.
- Метод может принимать значения *GET*, *POST*, *HEAD*, *PUT*, *DELETE* и др.
- Несмотря на обилие методов, для Web-программиста по-настоящему важны лишь два из них: *GET* и *POST*.

Методы запроса клиента

- GET. Согласно формальному определению, метод GET предназначается для получения ресурса с указанным URL. Получив запрос GET, сервер должен прочитать указанный ресурс и включить код ресурса в состав ответа клиенту. Ресурс, Несмотря на то что, по определению, метод GET предназначен для получения информации, он вполне подходит для передачи небольших фрагментов данных на сервер.
- POST. Согласно тому же формальному определению, основное назначение метода POST - передача данных на сервер. Однако, подобно методу GET, метод POST может применяться по-разному и нередко используется для получения информации с сервера. Как и в случае с методом GET, URL, заданный в строке состояния, указывает на конкретный ресурс.
- Методы HEAD и *PUT* являются модификациями методов GET и POST.

Поля заголовка запроса клиента

- Поля заголовка, следующие за строкой состояния, позволяют уточнять запрос, т.е. передавать серверу дополнительную информацию. Поле заголовка имеет следующий формат:

  Имя_поля: значение

- Назначение поля определяется его именем, которое отделяется от значения двоеточием.

Поля заголовка запроса клиента

Поля заголовка HTTP-запроса	Значение
Host	Доменное имя или *IP*-адрес узла, к которому обращается клиент
Referer	*URL* документа, который ссылается на ресурс, указанный в строке состояния
From	Адрес электронной почты пользователя, работающего с клиентом
Accept	*MIME*-типы данных, обрабатываемых клиентом. Это поле может иметь несколько значений, отделяемых одно от другого запятыми. Часто поле заголовка Accept используется для того, чтобы сообщить серверу о том, какие типы графических файлов поддерживает клиент
Accept-Language	Набор двухсимвольных идентификаторов, разделенных запятыми, которые обозначают языки, поддерживаемые клиентом
Accept-Charset	Перечень поддерживаемых наборов символов
Content-Type	*MIME*-тип данных, содержащихся в теле запроса (если запрос не состоит из одного заголовка)
Content-Length	Число символов, содержащихся в теле запроса (если запрос не состоит из одного заголовка)
Range	Присутствует в том случае, если клиент запрашивает не весь документ, а лишь его часть
Connection	Используется для управления *TCP*-соединением. Если в поле содержится *Close*, это означает, что после обработки запроса сервер должен закрыть соединение. Значение *Keep-Alive* предлагает не закрывать *TCP*-соединение, чтобы оно могло быть использовано для последующих запросов
User-Agent	Информация о клиенте

Пример запроса

- `GET /ru/latest/net/http.html HTTP/1.1`
- **Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8**
- **Accept-Language: en-US,en;q=0.5**
- **Connection: keep-alive**
- **Host: lectureswww.readthedocs.org**
- **User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:35.0) Gecko/20100101 Firefox/35.0**

Структура ответа сервера http

Знание структуры ответа сервера необходимо разработчику веб-приложений, так как программы, которые выполняются на сервере, должны самостоятельно формировать ответ клиенту.

- Получив от клиента запрос, сервер должен ответить ему.
- Подобно запросу клиента, ответ сервера также состоит из четырех перечисленных ниже компонентов.
  - Строка состояния.
  - Поля заголовка.
  - Пустая строка.
  - Тело ответа.

Структура ответа сервера http

- Ответ сервера клиенту начинается со строки состояния, которая имеет следующий формат:

  Версия_протокола Код_ответа Пояснительное_сообщение

- **Версия_протокола** задается в том же формате, что и в запросе клиента, и имеет тот же смысл.
- **Код_ответа** - это трехзначное десятичное число, представляющее в закодированном виде результат обслуживания запроса сервером.
- **Пояснительное_сообщение** дублирует код ответа в символьном виде. Это строка символов, которая не обрабатывается клиентом. Она предназначена для системного администратора или оператора, занимающегося обслуживанием системы, и является расшифровкой кода ответа.

Тело ответа веб-сервера

- Из трех цифр, составляющих код ответа, первая (старшая) определяет класс ответа, остальные две представляют собой номер ответа внутри класса. Так, например, если запрос был обработан успешно, клиент получает следующее сообщение:

HTTP/1.0 200 OK

- Как видно, за версией протокола HTTP 1.0 следует код 200. В этом коде символ 2 означает успешную обработку запроса клиента, а остальные две цифры (00) — номер данного сообщения.

Тело ответа веб-сервера

- В используемых в настоящее время реализациях протокола HTTP первая цифра не может быть больше 5 и определяет следующие классы ответов.
  - 1 - специальный класс сообщений, называемых *информационными*. Код ответа, начинающийся с 1, означает, что сервер продолжает обработку запроса. При обмене данными между HTTP-клиентом и HTTP-сервером сообщения этого класса используются достаточно редко.
  - 2 - успешная обработка запроса клиента.
  - 3 - перенаправление запроса. Чтобы запрос был обслужен, необходимо пред-принять дополнительные действия.
  - 4 - ошибка клиента. Как правило, код ответа, начинающийся с цифры 4, возвра-щается в том случае, если в запросе клиента встретилась синтаксическая ошибка. Например, клиент послал неверный запрос. Можно исправить на стороне клиента.
  - 5 - ошибка сервера. По тем или иным причинам сервер не в состоянии выполнить запрос.

Классы кодов ответа сервера

Код	Расшифровка	Интерпретация
100	Continue	Часть запроса принята, и сервер ожидает от клиента продолжения запроса
200	OK	Запрос успешно обработан, и в ответе клиента передаются данные, указанные в запросе
201	Created	В результате обработки запроса был создан новый ресурс
202	Accepted	Запрос принят сервером, но обработка его не окончена. Данный код ответа не гарантирует, что запрос будет обработан без ошибок.
206	Partial Content	Сервер возвращает часть ресурса в ответ на запрос, содержавший поле заголовка Range
300	Multiple Choice	Запрос указывает более чем на один ресурс. В теле ответа могут содержаться указания на то, как правильно идентифицировать запрашиваемый ресурс
301	Moved Permanently	Затребованный ресурс больше не располагается на сервере
302	Moved Temporarily	Затребованный ресурс временно изменил свой адрес
400	Bad Request	В запросе клиента обнаружена синтаксическая ошибка
403	Forbidden	Имеющийся на сервере ресурс недоступен для данного пользователя
404	Not Found	Ресурс, указанный клиентом, на сервере отсутствует
405	Method Not Allowed	Сервер не поддерживает метод, указанный в запросе
500	Internal Server Error	Один из компонентов сервера работает некорректно
501	Not Implemented	Функциональных возможностей сервера недостаточно, чтобы выполнить запрос клиента
503	Service Unavailable	Служба временно недоступна
505	HTTP Version not Supported	Версия HTTP, указанная в запросе, не поддерживается сервером

Примеры

- 401 Unauthorized – Не авторизован, не представился (не прошел аутентификацию)

- 403 Forbidden (запрещено, не уполномочен), авторизованный пользователь не имеет права на такие действия с ресурсом

Поля заголовка ответа веб-сервера

Имя поля	Описание содержимого
Server	Имя и номер версии сервера
Age	Время в секундах, прошедшее с момента создания ресурса
Allow	Список методов, допустимых для данного ресурса
Content-Language	Языки, которые должен поддерживать клиент для того, чтобы корректно отобразить передаваемый ресурс
Content-Type	*MIME*-тип данных, содержащихся в теле ответа сервера
Content-Length	Число символов, содержащихся в теле ответа сервера
Last-Modified	Дата и время последнего изменения ресурса
Date	Дата и время, определяющие момент генерации ответа
Expires	Дата и время, определяющие момент, после которого информация, переданная клиенту, считается устаревшей
Location	В этом поле указывается реальное расположение ресурса. Оно используется для перенаправления запроса
Cache-Control	Директивы управления кэшированием. Например, *no-cache* означает, что данные не должны кэшироваться

Тело ответа веб-сервера

- В *теле ответа* содержится код ресурса, передаваемого клиенту в ответ на запрос.
- Это не обязательно должен быть HTML-текст веб-страницы. В составе ответа могут передаваться *изображение*, *аудио-файл*, фрагмент *видеоинформации*, а также любой *другой тип данных*, поддерживаемых клиентом.
- О том, как следует обрабатывать полученный ресурс, клиенту сообщает содержимое поля заголовка *Content-type*.

Пример ответа веб-сервера

- HTTP/1.1 200 OK
- **Server: nginx/1.4.6 (Ubuntu)**
- **Date: Mon, 26 Jan 2015 16:54:33 GMT**
- **Content-Type: text/html**
- **Content-Length: 48059**
- **Last-Modified: Mon, 26 Jan 2015 16:22:21 GMT**
- **Connection: keep-alive**
- **Vary: Accept-Encoding**
- **ETag: "54c669bd-bbbb"**
- **X-Served: Nginx**
- **X-Subdomain-TryFiles: True**
- **X-Deity: hydra-lts**
- **Accept-Ranges: bytes**

- <!DOCTYPE html>
- <!--[if IE 8]><html class="no-js lt-ie9" lang="en" > <![endif]-->
- <!--[if gt IE 8]><!--> <html class="no-js" lang="en" > <!--<![endif]-->
- <head>
- <meta charset="utf-8">
- <meta name="viewport" content="width=device-width, initial-scale=1.0">
- ...

Поля заголовка if_modifed_since, last_modified, if_none_match, Etag

- Данные поля используются для кэширования контента (браузером или прокси сервером). В данном случае когда сервер присылает ответ с полем last_modified там указывается дата модификации объекта, в Etag указывается хэш объекта. При следующем запросе того же контента сервер шлет запрос с заголовком if_modifed_since или if_none_match в зависимости от даты или сохранения хэша. При изменении объекта возвращается 200 ОК и новый измененный объект, в ином случае код перенаправления 304 Not Modified, самого объекта естественно не возвращается, таким образом, очевидно можно организовать кэширование, которое может существенно сократить трафик и время ожидания клиента при прогрузке страницы.

Поле Заголовка HOST

- HTTP-запрос отправляется на определенные IP-адреса. Но так как большинство серверов способны размещать несколько сайтов под одним IP, они должны знать, какое доменное имя ищет браузер.
-
- Host: net.tutsplus.com
-
- Это в основном имя host, включая домен и поддомен.
-
- Например, в PHP его можно найти, как $_SERVER['HTTP_HOST'] или $_SERVER['SERVER_NAME'].

URI, URL, URN

- *URI* (*Uniform Resource Identifier*) — единообразный идентификатор ресурса, представляющий собой короткую последовательность символов, идентифицирующую абстрактный или физический ресурс.
- Самые известные примеры *URI* — это *URL* и *URN*.
- *URL* (*Uniform Resource Locator*) - это *URI*, который, помимо идентификации ресурса, предоставляет ещё и информацию о местонахождении этого ресурса.
- *URN* (*Uniform Resource Name*) — это *URI*, который идентифицирует ресурс в определённом пространстве имён, но, в отличие от *URL*, *URN* не указывает на местонахождение этого ресурса.
- *URI* не указывает на то, как получить ресурс, а только идентифицирует его. Что даёт возможность описывать с помощью *RDF* (*Resource Description Framework*) ресурсы, которые не могут быть получены через Интернет (имена, названия и т.п.)

Структура URL

<схема>://<логин>:<пароль>@<хост>:<порт>/<URL-путь>

Где:

- *схема* - схема обращения к ресурсу (обычно сетевой протокол);
- *логин* - имя пользователя, используемое для доступа к ресурсу;
- *пароль* - пароль, ассоциированный с указанным именем пользователя;
- *хост* - полностью прописанное доменное имя хоста в системе *DNS* или *IP-адрес* хоста;
- *порт* - порт хоста для подключения;
- *URL-путь* - уточняющая информация о месте нахождения ресурса.

Структура URL

- Общепринятые схемы (протоколы) URL включают протоколы: *ftp*, *http*, *https*, *telnet*, а также:
  - *gopher* — протокол *Gopher*;
  - *mailto* — адрес электронной почты;
  - *news* — новости *Usenet*;
  - *nntp* — новости *Usenet* через протокол *NNTP*;
  - *irc* — протокол *IRC*;
  - *prospero* — служба каталогов *Prospero Directory Service*;
  - *wais* — база данных системы *WAIS*;
  - *xmpp* — протокол *XMPP* (часть *Jabber*);
  - *file* — имя локального файла;
  - *data* — непосредственные данные (*Data*: *URL*);

Порт TCP/IP

- TCP/IP *порт* — целое число от 1 до 65535, позволяющие различным программам, выполняемым на одном хосте, получать данные независимо друг от друга. Каждая программа обрабатывает данные, поступающие на определённый порт («слушает» этот порт).
- Самые распространенные сетевые протоколы имеют стандартные номера портов, хотя в большинстве случаев программа может использовать любой порт.
- Для наиболее распространенных протоколов стандартные номера портов следующие:
    - HTTP: 80
    - FTP: 21 (для команд), 20 (для данных)
    - telnet: 23
    - POP3: 110
    - IMAP: 143
    - SMTP: 25
    - SSH: 22

HTTPS

- HTTPS — расширение протокола *HTTP*, поддерживающее шифрование. Данные, передаваемые по протоколу *HTTP*, «упаковываются» в криптографический протокол *SSL* или *TLS*, тем самым обеспечивается защита этих данных. В отличие от *HTTP*, для *HTTPS* по умолчанию используется TCP-порт 443.
- Чтобы подготовить веб-сервер для обработки *HTTPS* соединений, администратор должен получить и установить в систему сертификат для этого веб-сервера.

SSL и TLS

- *SSL* (Secure Sockets Layer) — криптографический протокол, обеспечивающий безопасную передачу данных по сети Интернет.
- При его использовании создаётся защищённое соединение между клиентом и сервером. *SSL* изначально разработан компанией *Netscape Communications*. Впоследствии на основании протокола *SSL 3.0* был разработан и принят стандарт *RFC*, получивший название TLS.
- Протокол использует шифрование с открытым ключом для подтверждения подлинности передатчика и получателя. Поддерживает надёжность передачи данных за счёт использования корректирующих кодов и безопасных хэш-функций.

SSL и TLS

- На нижнем уровне многоуровневого транспортного протокола (например, TCP) он является протоколом записи и используется для инкапсуляции различных протоколов (например POP3, IMAP, SMTP или HTTP).
- Для каждого инкапсулированного протокола он обеспечивает условия, при которых сервер и клиент могут подтверждать друг другу свою подлинность, выполнять алгоритмы шифрования и производить обмен криптографическими ключами, прежде чем протокол прикладной программы начнет передавать и получать данные.
- Для доступа к веб-страницам, защищённым протоколом SSL, в URL вместо схемы http, как правило, подставляется схема https, указывающая на то, что будет использоваться SSL-соединение. Стандартный TCP-порт для соединения по протоколу https — 443.
- Для работы SSL требуется, чтобы на сервере имелся SSL-сертификат.

Методы аутентификации в WWW

- *Basic* — базовая аутентификация, при которой имя пользователя и пароль передаются в заголовках *http-пакетов*. Пароль при этом не шифруется и присутствует в чистом виде в кодировке *base64*. Для данного типа аутентификации использование *SSL* является обязательным.
- *Digest* — дайджест-аутентификация, при которой пароль пользователя передается в хешированном виде. По уровню конфиденциальности паролей этот тип мало чем отличается от предыдущего, так как атакующему все равно, действительно ли это настоящий пароль или только *хеш* от него: перехватив удостоверение, он все равно получает доступ к конечной точке. Для данного типа аутентификации использование *SSL* является обязательным.

Методы аутентификации в WWW

- *Integrated* — интегрированная аутентификация, при которой клиент и сервер обмениваются сообщениями для выяснения подлинности друг друга с помощью протоколов *NTLM* или *Kerberos*. Этот тип аутентификации защищен от перехвата удостоверений пользователей, поэтому для него не требуется протокол *SSL*. Только при использовании данного типа аутентификации можно работать по схеме *http*, во всех остальных случаях необходимо использовать схему *https*.

Cookie

- HTTP-сервер не помнит предыстории запросов клиентов и каждый запрос обрабатывается независимо от других
- Поэтому у сервера нет возможности определить, исходят ли запросы от одного клиента или разных клиентов
- Тем не менее механизм *cookie* позволяет серверу хранить информацию на компьютере клиента и извлекать ее оттуда.

Cookie

- Инициатором записи *cookie* выступает сервер.
- Если в ответе сервера присутствует поле заголовка *Set-cookie*, клиент воспринимает это как команду на запись *cookie*.
- В дальнейшем, если клиент обращается к серверу, от которого он ранее принял поле заголовка *Set-cookie*, помимо прочей информации он передает серверу данные *cookie*.
- Для передачи указанной информации серверу используется поле заголовка *Cookie*.

Пример использования cookie

1. Передача запроса серверу ***A***.
2. Получение ответа от сервера ***A***.
3. Передача запроса серверу ***B***.
4. Получение ответа от сервера ***B***. В состав ответа входит поле заголовка *Set-cookie*. Получив его, клиент записывает *cookie* на диск.
5. Передача запроса серверу ***C***. Несмотря на то что на диске хранится запись *cookie*, клиент не предпринимает никаких специальных действий, так как значение *cookie* было записано по инициативе другого сервера.

Пример использования cookie

6. Получение ответа от сервера ***C***.
7. Передача запроса серверу ***A***. В этом случае клиент также никак не реагирует на тот факт, что на диске хранится *cookie*.
8. Получение ответа от сервера ***A***.
9. Передача запроса серверу ***B***. Перед тем как сформировать запрос, клиент определяет, что на диске хранится запись *cookie*, созданная после получения ответа от сервера ***B***. Клиент проверяет, удовлетворяет ли данный запрос некоторым требованиям, и, если проверка дает положительный результат, включает в заголовок запроса поле *Cookie*.

Формат поля Set-Cookie

Set-cookie: имя = значение; expires = дата;

path = путь; домен = имя_домена, secure

где

- Пара *имя = значение* – именованные данные, сохраняемые с помощью механизм *cookie*. Эти данные должны храниться на клиент-машине и передаваться серверу в составе очередного запроса клиента.
- *Дата*, являющаяся значением параметра *expires*, определяет время, по истечении которого информация *cookie* теряет свою актуальность. Если ключевое слово *expires* отсутствует, данные *cookie* удаляются по окончании текущего сеанса работы браузера.

Формат поля Set-Cookie

- Значение параметра *domain* определяет домен, с которым связываются данные cookie.
- Чтобы узнать, следует ли передавать в составе запроса данные *cookie*, браузер сравнивает доменное имя сервера, к которому он собирается обратиться, с доменами, которые связаны с записями *cookie*, хранящимися на клиент-машине.
- Результат проверки будет считаться положительным, если сервер, которому направляется запрос, принадлежит домену, связанному с *cookie*.
- Если соответствие не обнаружено, данные *cookie* не передаются.

Формат поля Set-Cookie

- Путь, указанный в качестве значения параметра *path*, позволяет выполнить дальнейшую проверку и принять окончательное решение о том, следует ли передавать данные *cookie* в составе запроса.
- Помимо домена с записью *cookie* связывается путь.
- Если браузер обнаружил соответствие *имени домена* значению параметра *domain*, он проверяет, соответствует ли путь к ресурсу пути, связанному с *cookie*.

Формат поля Set-Cookie

- Сравнение считается успешным, если ресурс содержится в каталоге, указанном посредством ключевого слова *path*, или в одном из его подкаталогов.
- Если и эта проверка дает положительный результат, данные *cookie* передаются серверу. Если параметр *path* в поле *Set-cookie* отсутствует, то считается, что запись *cookie* связана с URL конкретного ресурса, передаваемого сервером клиенту.
- Последний параметр, *secure*, указывает на то, что данные *cookie* должны передаваться по защищенному каналу.

Формат поля *Cookie*

- Для передачи данных *cookie* серверу используется поле заголовка *Cookie*.
- Формат этого поля:

  Cookie: имя=значение; имя=значение; ...

- С помощью поля *Cookie* передается одна или несколько пар *имя = значение*. Каждая из этих пар принадлежит записи *cookie*, для которой URL запрашиваемого ресурса соответствуют имени домена и пути, указанным ранее в поле *Set-cookie*.

Пример использования COOKIE на php

```
Скрипт cookies.php выполняется на стороне сервера
<?php
if ($BeenSubmitted) {
# устанавливаем куки по полученным данным от клиента если была нажата кнопка submit
# у клиента сохраняется информация, эту информацию потом от клиента # же и получает сервер при последующих запросах от браузера
      setcookie("BGColor", "$NewBGColor");
      setcookie("TextColor", "$NewTextColor");
      $BGColor = $NewBGColor;
      $TextColor = $NewTextColor;
} else {
      if (!$BGColor) {
            $BGColor = "WHITE";
      }
      if (!$TextColor) {
            $TextColor = "BLACK";
      }
}
?>
```

```
<HTML>
<HEAD>
      <TITLE>User Customization</TITLE>
</HEAD>
Устанавливаем параметры страницы из сохраненных куки переданных клиентом в заголовках запроса, скрипт выполняется на стороне сервера подготовив страницу с тэгом body
<?php print ("<BODY BGCOLOR=$BGColor TEXT=$TextColor>\n"); ?>
Currently your page looks like this!
<FORM ACTION="cookies.php" METHOD=POST>
Select a new background color:
<!-- Элемент выбора меню  фона --><!-- -->
  <SELECT NAME="NewBGColor">
      <OPTION VALUE=WHITE>WHITE</OPTION>
      <OPTION VALUE=BLACK> BLACK </OPTION>
      <OPTION VALUE=BLUE> BLUE </OPTION>
      <OPTION VALUE=GREEN> GREEN </OPTION>
  </SELECT>
Select a new text color:
<!-- Элемент выбора меню  цвета текста --> <!-- -->
<SELECT NAME="NewTextColor">
      <OPTION VALUE=WHITE> WHITE </OPTION>
      <OPTION VALUE=BLACK> BLACK </OPTION>
      <OPTION VALUE=RED> BLUE </OPTION>
</SELECT>
<! -- при нажатии вызов скрипта cookie.php с установкой скрытой переменной флага и другими элементами формы включая выбранные цвета фона и текста-->
<INPUT TYPE=HIDDEN NAME=BeenSubmitted VALUE=TRUE>
<INPUT TYPE=SUBMIT NAME="SUBMIT" VALUE="Submit!">
</FORM>
</BODY>
</HTML>
```

WebSocket

- Отличие технологии Websocket от HTTP заключается в возможности использования асинхронных запросов, как со стороны клиента, так и со стороны сервера. Использование HTTP предполагает запрос от клиента и ответ, для реализации интерактивного приложения взаимодействующего с сервером необходимо постоянно обращаться к серверу с запросами, чтобы узнать его состояние. Таким образом, сервер сам инициирует событие, на которое реагирует клиент. В том, числе веб-соккеты позволяют инициировать обработку событий подключения клиента, отключения, отправки сообщения.

Код на Node .js

```
console.log("WebSocket Server ")
//библиотека работы с файлами
var fs = require('fs');
//синхронно читаем клиентский html файл
var contents = fs.readFileSync('webclient.html', 'utf8');
//библиотека работы с вебсоккетом сервера
var WebSocketServer = require('websocket').server;
var http = require('http');
//создаем http сервер
var server = http.createServer(function(request, response) {
//отдаем считанный документ клиенту (браузеру)
  response.writeHead(200, {'Content-Type': 'text/html' });
  response.write(contents)
  response.end() });
//прослушивание соккета сервера
server.listen(10556, function() { });
// create the веб соккет server ассоцированный с http
сервером
wsServer = new WebSocketServer({
  httpServer: server });
// WebSocket server - обработка события запрос
соединения
wsServer.on('request', function(request) {
//переменные для каждого клиента свои
  var fl = true;
  var index = 0;
  var connection = request.accept(null, request.origin);
  console.log(connection.toString());
// обработчик собатия от таймера
 function timecounter(arg) {
 console.log('arg was => %s', arg);
 console.log(fl);
//проверка закрыт ли соккет до отправки ему сообщения
 if(fl) {
 //отправляем json строку клиенту
 connection.sendUTF(
 JSON.stringify({ 'type': 'chat', 'data': String(index)} ));
 index=index+1;
 setTimeout(timecounter, 3000, index);
 }
 }
 setTimeout(timecounter, 3000, 'start');
 // This is the most important callback for us, we'll handle
 // all messages from users here.
 connection.on('message', function(message) {
    // process WebSocket message  , если это текст
    if(message.type === 'utf8') {
    console.log("client send json: "+message.utf8Data);
    msg = JSON.parse(message.utf8Data);
    console.log("client send data: "+msg.data);
    }
 });
 connection.on('close', function(connection) {
  // close user connection
   fl = false;
   console.log(connection.toString());
   console.log("User close connection")
 });
});
```

HTTP 2

- Вторая крупная версия сетевого протокола HTTP, используемая для доступа к World Wide Web. Протокол основан на SPDY. HTTP/2 был разработан рабочей группой Hypertext Transfer Protocol working group (httpbis, где bis означает «ещё раз», «повторно», «на бис») из Internet Engineering Task Force.
- HTTP/2 является первой новой версией HTTP с версии HTTP 1.1, которая была стандартизирована RFC 2616 в 1999. Рабочая группа представила протокол HTTP/2 на рассмотрение IESG как Proposed Standard в декабре 2014 и IESG утвердила его к публикации как Proposed Standard 17 февраля 2015. Спецификация HTTP/2 была опубликована как RFC 7540 в мае 2015.
- Усилия по стандартизации являются ответом на разработку SPDY (HTTP-совместимый протокол, разработанный Google и поддерживаемый браузерами Chrome, Opera, Firefox, Internet Explorer 11, Safari и Amazon Silk).
- 9 февраля 2015 года Google объявила о планах прекратить поддержку SPDY в Chrome в начале 2016 года в пользу HTTP/2 (Chrome 40+).
- По данным W3Techs на 1 марта 2020 года, 43,6 % из 10 млн самых популярных интернет-сайтов поддерживают протокол HTTP/2

Цели протокола

- Добавить механизмы согласования протокола, клиент и сервер могут использовать HTTP 1.1, 2.0 или, гипотетически, иные, не HTTP-протоколы.
- Поддержать совместимость с многими концепциями HTTP 1.1, например по набору методов доступа (GET, PUT, POST и т. п.), статусным кодам, формату URI, большому количеству заголовков
- Уменьшение задержек доступа для ускорения загрузки страниц, в частности путём:
- Сжатия данных в заголовках HTTP
- Использования push-технологий на серверной стороне
- Конвейеризации запросов
- Устранения проблемы блокировки «head-of-line» протоколов HTTP 1.0/1.1
- Мультиплексирования множества запросов в одном соединении TCP
- Сохранение совместимости с широко внедрёнными применениями HTTP, в том числе с веб-браузерами (полноценными и мобильными), API, используемыми в Интернете, веб-серверами, прокси-серверами, обратными прокси-серверами, сетями доставки контента

Свойства протокола HTTP 2

- Бинаризация
- В отличие от текстового HTTP 1.1, HTTP/2 — бинарный. Поэтому протокол более эффективен при парсинге, более компактный при передаче, подвержен меньшему количеству ошибок.

- Мультиплексирование
- В HTTP 1.1 браузеры используют множественные подключения к серверу для загрузки веб-страницы, причем, количество таких соединений ограничено. Но это не решает проблему с блокированием канала медленными пакетами. Тогда как в HTTP/2 используется мультиплексирование, которое позволяет браузеру использовать одно соединение TCP для всех запросов. HTTP/2 multiplexing
-
- Все файлы подгружаются параллельно. Запросы и ответы разделяются по фреймам с мета-данными, которые ассоциируют запросы и ответы. Так что они не перекрывают друг-друга и не вызывают путаницы. При этом ответы получаются по мере их готовности, следовательно, тяжелые запросы не будут блокировать обработку и выдачу более простых объектов.

- Приоритизация
- Вместе с мультиплексированием появилась приоритизация трафика. Запросам можно назначить приоритет на основе важности и зависимости. Приоритизация HTTP/2
- Так что при загрузке веб-страницы браузер будет в первую очередь получать важные данные, CSS-код, к примеру, а все второстепенное обработается в последнюю очередь.
- Компрессия заголовков HPACK

Протокол HTTP построен таким образом, что при отправке запросов также передаются заголовки, которые содержат дополнительную информацию. Сервер, в свою очередь, также прикрепляет заголовки к ответам. А учитывая, что веб-страницы состоят из множества файлов, все заголовки могут занимать приличный объем. Поэтому в HTTP/2 присутствует сжатие заголовков, которое позволит существенно сократить объем вспомогательной информации, так что браузер сможет отправить все запросы сразу.

- Server Push
- При использовании протокола HTTP 1.1 браузер запрашивает страницу, сервер отправляет в ответ HTML и ждет, пока браузер его обработает и запросит все необходимые файлы: JavaScript, CSS и фото. Поэтому в новый протокол внедрили интересную функцию под названием Server Push.
- HTTP/2 Server Push
- Позволяет серверу сразу же, не дожидаясь ответа веб-браузера, добавить нужные по его мнению файлы в кэш для быстрой выдачи.
- Шифрование
- Протокол HTTP/2 не требует шифрования канала. Тем не менее, все современные браузеры работают с HTTP/2 только вместе с TLS, как и Nginx. Так что массовое внедрение протокола должно поспособствовать распространению шифрования по Сети.
- Поэтому, если вы уже используете TLS, то стоит задействовать HTTP/2, который раскрывает весь потенциал шифрования. При создании зашифрованного соединения происходит только один TLS Handshake, что существенно упрощает весь процесс и сокращает время подключения.

- Оптимизация HTTP/2
-
- Главная оптимизация HTTP/2 по сравнению с HTTP 1.1 — отключение или модификация многих оптимизаций прошлой версии протокола.
-
- Стоит отказаться от доменного шардинга. Такой способ распределения множества файлов по различным доменам и CDN актуален для HTTP 1.1, так как решает проблему параллельных соединений. Но в случае с новым протоколом такое решение ухудшает производительность и нивелирует приоритизацию трафика.
-

- По возможности откажитесь или модифицируйте спрайты. Объединение множества маленьких картинок в одно большое изображение способно увеличить скорость загрузки страницы, но если пользователь заходил на веб-страницу с одной маленькой картинкой, то ему все-равно отправлялся весь спрайт. В случае с HTTP/2 такое решение будет менее полезным в виду появления мультиплексирования.

HTTP 3

- Решает недостатки HOL в HTTP2
- Отправляет данные по средством протокола QUIC на базе UDP

Data: uri

- Еще один метод оптимизации изображений — встраивание при помощи DataURI. Он также может быть полезен в HTTP/2, но точно будет менее эффективным, чем в случае с прошлой версией.

- data: URL — это определённая стандартом RFC 2397 схема, которая позволяет включать небольшие элементы данных в строку URL, как если бы они были ссылкой на внешний ресурс. Она гораздо проще альтернативных методов включения, таких, как MIME с cid: или mid:. Согласно букве RFC «data: URI» это фактически «data: URL» (URL — унифицированный указатель ресурса), хотя реально он ни на что не указывает.

- **Фрагмент внедрённого в XHTML небольшого изображения (Перенос на новую строку осуществлён для облегчения восприятия):**
- <img
- src="
- AAAC8IyPqcvt3wCcDkiLc7C0qwyGHhSWpjQu5yqmCYsapyuvUUlvONmOZtfzgFz
- ByTB10QgxOR0TqBQejhRNzOfkVJ+5YiUqrXF5Y5lKh/DeuNcP5yLWGsEbtLiOSp
- a/TPg7JpJHxyendzWTBfX0cxOnKPjgBzi4diinWGdkF8kjdfnycQZXZeYGejmJl
- ZeGl9i2icVqaNVailT6F5iJ90m6mvuTS4OK05M0vDk0Q4XUtwvKOzrcd3iq9uis
- F81M1OIcR7lEewwcLp7tuNNkM3uNna3F2JQFo97Vriy/Xl4/f1cf5VWzXyym7PH
- hhx4dbgYKAAA7"
- alt="Larry"/>

- Правило CSS с внедрённым фоновым изображением **(переносы сделаны для облегчения восприятия):**
- ul.checklist > li.complete {
- margin-left: 20px;
- background:
- url(
- iVBORw0KGgoAAAANSUhEUgAAABAAAAAQAQMAAAAlPW0iAAAABlBMVEUAAAD///+l2Z/
  dAAAAM0lEQVR4nGP4/5/h/1+G/58ZDrAz3D/ McH8yw83NDDeN
  Ge4Ug9C9zwz3gVLMDA/A6P9/AFGGFyjOXZtQAAAAAElFTkSuQmCC)
- top left no-repeat;
- }

- Лучше отказаться от объединения (конкатенации) файлов. Метод похож на спрайты — все необходимые файлы, CSS и JavaScript, объединяются в один большой для передачи одним потоком по одному соединению. Так что если пользователь зашел на страницу с одним небольшим кодом JS, ему все-равно будет отправлен весь объединенный файл. Еще одна сложность — все объединенные файлы нужно выгружать из кэша одновременно, а одно изменение в коде любого из них требует обновления всего набора. Так что благодаря все тому же мультиплексированию такой подход не имеет смысла.
- Также стоит отказаться от встраивания файлов в HTML код.
-
- Самое главное
-
- Протокол HTTP/2 уже значительно оптимизирован, по сравнению с HTTP 1.1, так что простое внедрение новой спецификации способно улучшить производительность веб-сервисов. А отключение дополнительных ухищрений, которые использовались для ускорения HTTP 1.1 поможет воспользоваться всеми преимуществами HTTP/2.

Proxy на Python

```
Python
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
from SocketServer import ThreadingMixIn
import urllib2, sys, re, httplib, urlparse


class ProxyHandler(BaseHTTPRequestHandler):
        server_version = ''
        sys_version = ''
        def do_HEAD( self ):
                print "HEAD"
        def log_message(self,format,*args):
                return
```

```
def do_POST( self ):
                print '------------------------------'
                print "POST"
                requested_url = self.requestline.split()[1]
                parsed_url = urlparse.urlsplit( requested_url )
                cutted_url = urlparse.urlunsplit( ( '', '', parsed_url.path,
parsed_url.query, '' ) )
                print parsed_url.hostname
                print parsed_url.path
                print parsed_url.query
                print requested_url
                port = 80 if None == parsed_url.port else parsed_url.port
                req_headers = {}
                for x in self.headers.items():
                        req_headers[x[0]] = x[1]
                print req_headers
                body = self.rfile.read(int(self.headers['content-length']))
                print 'body -- ',body
```

```
conn = httplib.HTTPConnection( parsed_url.hostname, port )
    conn.request( 'POST', cutted_url, body,headers = req_headers )
    response = conn.getresponse()
    print response.status
    self.send_response( response.status )
    for x in response.msg.items():
                self.send_header( x[0], x[1] )
    self.end_headers()

    html = response.read()
    self.wfile.write( html )
    self.connection.close()
    conn.close()
    print 'End - post'
    return
```

```
        def do_GET( self ):
                print '-----------------------------------'
                print 'GET'
                requested_url = self.requestline.split()[1]
                parsed_url = urlparse.urlsplit( requested_url )
                cutted_url = urlparse.urlunsplit( ( '', '', parsed_url.path,
parsed_url.query, '' ) )
                port = 80 if None == parsed_url.port else parsed_url.port
                req_headers = {}
                for x in self.headers.items():
                        req_headers[x[0]] = x[1]
                print req_headers

                conn = httplib.HTTPConnection( parsed_url.hostname, port )
                conn.request( 'GET', cutted_url ,'' ,req_headers )
                response = conn.getresponse()
```

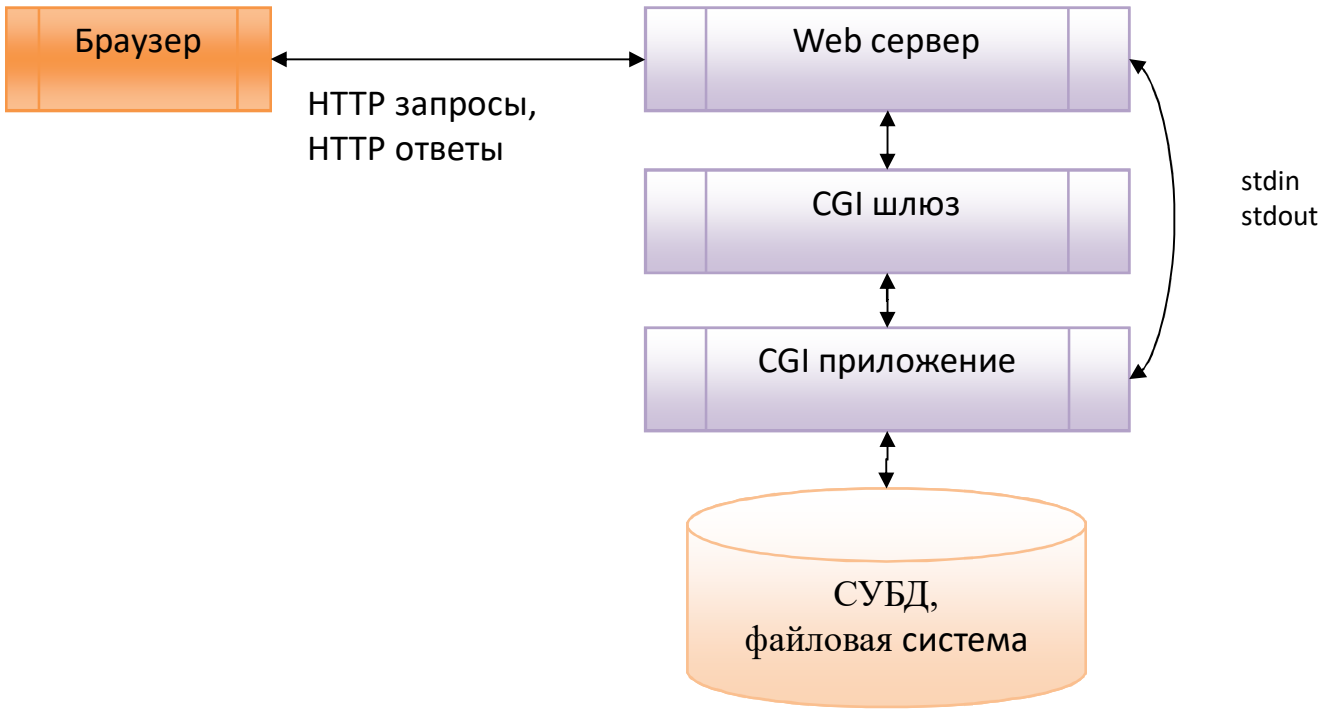
```
self.name = ''
self.send_response( response.status )
print response.status
print parsed_url.hostname, ':', port
print response.msg.items()
i=0
for x in response.msg.items():
        i=i+1
        if(x[0]!='transfer-encoding'):
                print '--- ', x[0],' : ' ,x[1]
                if(x[0]!='connection'):
                        self.send_header( x[0], x[1] )
                else: self.send_header( 'connection', 'close' )
self.end_headers()
html = response.read()
self.wfile.write( html )
conn.close()
self.connection.close()
print 'END -- GET'
return 0
```

```
class ThreadedHTTPServer( ThreadingMixIn, HTTPServer ):
#поток-демон уничтожается при уничтожении главного потока
#пользовательский поток может не уничтожаться, программа висит
        daemon_threads = True
if __name__ == '__main__':
        proxy = ThreadedHTTPServer( ( 'localhost', 19277 ), ProxyHandler )

        try:
                proxy.serve_forever()
        except KeyboardInterrupt:
                print 'End of server'
        proxy.server_close()
```

CGI

- CGI (Common Gateway Interface или общий интерфейс шлюза) — стандарт интерфейса, используемого для связи внешней программы с веб-сервером. Программу, которая работает по такому интерфейсу совместно с веб-сервером принято называть шлюзом, хотя многие предпочитают названия «скрипт» (сценарий) или «CGI-программа». Запускает на каждый запрос процесс.



Fast CGI

- Запускается один процесс, передача данных осуществляется посредством сокетов TCP. Дает дополнительные возможности безопасности давая возможность запустить процесс под другим пользователем с другими правами.

PHP

JSP

ASP

Python WSGI

Фреймворки Python

Веб сервис

- Веб-служба, веб-сервис (англ. web service) — идентифицируемая уникальным веб-адресом (URL-адресом) программная система со стандартизированными интерфейсами, а также HTML-документ сайта, отображаемый браузером пользователя.
-
- Веб-службы могут взаимодействовать друг с другом и со сторонними приложениями посредством сообщений, основанных на определённых протоколах (SOAP, XML-RPC и т. д.) и соглашениях (REST). Веб-служба является единицей модульности при использовании сервис-ориентированной архитектуры приложения.
-
- В обиходе веб-сервисами называют услуги, оказываемые в Интернете. В этом употреблении термин требует уточнения, идёт ли речь о поиске, веб-почте, хранении документов, файлов, закладок и т. п. Такими веб-сервисами можно пользоваться независимо от компьютера, браузера или места доступа в Интернет

Достоинства

- Веб-службы обеспечивают взаимодействие программных систем независимо от платформы. Например, Windows-C#-клиент может обмениваться данными с Java-сервером, работающим под Linux.
- Веб-службы основаны на базе открытых стандартов и протоколов. Благодаря использованию XML достигается простота разработки и отладки веб-служб.
- Использование интернет-протокола обеспечивает HTTP-взаимодействие программных систем через межсетевой экран. Это значительное преимущество, по сравнению с такими технологиями, как CORBA, DCOM или Java RMI. С другой стороны, веб-службы не привязаны намертво к HTTP — могут использоваться и другие протоколы.

Недостатки

- Меньшая производительность и больший размер сетевого трафика по сравнению с технологиями RMI, CORBA, DCOM за счёт использования текстовых XML-сообщений. Однако на некоторых веб-серверах возможна настройка сжатия сетевого трафика.
- Аспекты безопасности. Ответственные веб-службы должны использовать кодирование, возможно — требовать аутентификации пользователя. Достаточно ли здесь применения HTTPS, или предпочтительны такие решения, как XML Signature, XML Encryption или SAML — должно быть решено разработчиком.

DCOM

Corba

SOAP

Форматы данных

Спецификация MIME

- Поле с именем Content-type может встречаться как в запросе клиента, так и в ответе сервера. В качестве значения этого поля указывается MIME-тип содержимого запроса или ответа.
- *MIME-тип* также передается в поле заголовка *Accept*, присутствующего в запросе.
- Спецификация *MIME* (*Multipurpose Internet Mail Extension*) первоначально была разработана для того, чтобы обеспечить передачу различных форматов данных в составе электронных писем.
- Однако применение MIME не исчерпывается электронной почтой. Средства MIME успешно используются в WWW и, по сути, стали неотъемлемой частью этой системы.

Спецификация MIME

- В соответствии со спецификацией *MIME*, для описания формата данных используются *тип* и *подтип*. *Тип* определяет, к какому классу относится формат содержимого HTTP-запроса или HTTP-ответа. *Подтип* уточняет формат. Тип и подтип отделяются друг от друга косой чертой:

  тип/подтип

- Поскольку в подавляющем большинстве случаев в ответ на запрос клиента сервер возвращает исходный текст HTML-документа, то в поле *Content-type* ответа обычно содержится значение *text/html*. Здесь идентификатор *text* описывает *тип*, сообщая, что клиенту передается символьная информация, а идентификатор *html* описывает *подтип*, т.е. указывает на то, что последовательность символов, содержащаяся в теле ответа, представляет собой описание документа на языке HTML.

MIME типы данных

Тип/подтип	Расширение файла	Описание
application/pdf	.pdf	Документ, предназначенный для обработки Acrobat Reader
application/msexcel	.xls	Документ в формате Microsoft Excel
application/postscript	.ps, .eps	Документ в формате PostScript
application/x-tex	.tex	Документ в формате TeX
application/msword	.doc	Документ в формате Microsoft Word
application/rtf	.rtf	Документ в формате RTF, отображаемый с помощью Microsoft Word
image/gif	.gif	Изображение в формате GIF
image/ jpeg	.jpeg, .jpg,	Изображение в формате JPEG
image/tiff	.tiff, .tif	Изображение в формате TIFF
image/x-xbitmap	.xbm	Изображение в формате XBitmap
text/plain	.txt	ASCII-текст
text/html	. html , . htm	Документ в формате HTML
audio/midi	.midi, .mid	Аудиофайл в формате MIDI
audio/x-wav	.wav	Аудиофайл в формате WAV
message/rfc822		Почтовое сообщение
message/news		Сообщение в группы новостей
video /mpeg	.mpeg, .mpg, .mpe	Видеофрагмент в формате MPEG
video/avi	.avi	Видеофрагмент в формате AVI

multipart/form-data

- Тип содержимого multipart/form-data — это составной тип содержимого, чаще всего использующийся для отправки HTML-форм с бинарными (не-ASCII) данными методом POST протокола HTTP. Указывается в поле заголовка Content-Type (тип содержимого) и следует правилам для составных MIME-данных в соответствии с RFC 2045. Для форм, не имеющих больших бинарных (не-ASCII) данных, может использоваться тип содержимого application/x-www-form-urlencoded.
-
- Сообщение multipart/form-data содержит несколько частей, по одной на каждый задействованный в форме элемент управления.

XML

- <?xml version="1.0" encoding="UTF-8"?>
- <?xml-stylesheet type="text/xsl" href="file.xslt" ?>
- <people>
- <man id= "1">
- <name>John</name>
- <age>30</age>
- <work>Driver</work>
- </man>
- <man id = "2">
- <name>Lisa</name>
- <age>20</age>
- <work>Programmist</work>
- </man>
- </people>

XSL

- <?xml version="1.0" encoding="UTF-8"?>
- <xsl:stylesheet version = "1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
- <xsl:template match="/">
- <html>
- <head>
- <title>People</title>
- </head>
- <body>
- ########
- </body>
- </html>
- </xsl:template>
- </xsl:stylesheet>

```
<table border = "1">
    <tbody>
        <xsl:for-each select="people/man">
          <tr>
            <th>
                <xsl:value-of select="@id"/>
            </th>
            <th>
                <xsl:value-of select="name"/>
            </th>
            <th>
        <xsl:value-of select="age"/>
            </th>
             <th>
        <xsl:value-of select="work"/>
            </th>
          </tr>
        </xsl:for-each>
    </tbody>
</table>
```

JSON

Некоторые проблемы, костыли и лайфхаки, которые нужно (было) учитывать

Учет проблем доступа потоков к памяти при кэшировании

- Если данные которые содержат небольшие структуры данных находятся в одной линейке кэша, при этом один поток изменяет какую-то из структур, то вся линейка кэша блокируется, потому при чтении близких данных будут проблемы. Желательно сокращать критические секции.

Line ahead blocking

TCP syn flooding

SQL инъекции

Атаки на переполнение буфера

CSRS Token

- CSRF (англ. Cross Site Request Forgery — «Межсайтовая подделка запроса», также известен как XSRF) — вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP. Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника). Для осуществления данной атаки жертва должна быть аутентифицирована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя, который не может быть проигнорирован или подделан атакующим скриптом.

- Данный тип атак, вопреки распространённому заблуждению, появился достаточно давно: первые теоретические рассуждения появились в 1988 году[1], а первые уязвимости были обнаружены в 2000 году. А сам термин ввел Peter Watkins в 2001 году.

- Основное применение CSRF — вынуждение выполнения каких-либо действий на уязвимом сайте от лица жертвы (изменение пароля, секретного вопроса для восстановления пароля, почты, добавление администратора и т. д.). Также с помощью CSRF возможна эксплуатация отраженных XSS, обнаруженных на другом сервере.

- Атака осуществляется путем размещения на веб-странице ссылки или скрипта, пытающегося получить доступ к сайту, на котором атакуемый пользователь заведомо (или предположительно) уже аутентифицирован. Например, пользователь Алиса может просматривать форум, где другой пользователь, Мэллори, разместил сообщение. Пусть Мэллори создал тег <img>, в котором в качестве источника картинки указан URL, при переходе по которому выполняется действие на сайте банка Алисы, например:

- Мэллори: Привет, Алиса! Посмотри, какой милый котик: <img src="http://bank.example.com/withdraw?account=Alice&amount=1000000&for=Mallory">

- Если банк Алисы хранит ее информацию об аутентификации в куки и если куки еще не истекли, при попытке загрузить картинку браузер Алисы отправит запрос на перевод денег на счет Мэллори и подтвердит аутентификацию при помощи куки. Таким образом, транзакция будет успешно завершена, хотя ее подтверждение произойдет без ведома Алисы.

- Наиболее простым для понимания способом защиты от данного типа атак является механизм, когда веб-сайты должны требовать подтверждения большинства действий пользователя и проверять поле HTTP_REFERER, если оно указано в запросе. Но этот способ может быть небезопасен, и использовать его не рекомендуется[2].

- Другим распространённым способом защиты является механизм, при котором с каждой сессией пользователя ассоциируется дополнительный секретный уникальный ключ, предназначенный для выполнения запросов. Секретный ключ не должен передаваться в открытом виде, например если это GET запрос, то ключ следует передавать в теле запроса, а не в адресе страницы. Пользователь посылает этот ключ среди параметров каждого запроса, и перед выполнением каких-либо действий сервер проверяет этот ключ. Преимуществом данного механизма, по сравнению с проверкой Referer, является гарантированная защита от атак данного типа. Недостатком же являются: требование возможности организации пользовательских сессий и требование динамической генерации HTML-кода активных страниц сайта.

- Существует более жёсткий вариант предыдущего механизма, в котором с каждым действием ассоциируется уникальный одноразовый ключ. Такой способ более сложен в реализации и требователен к ресурсам. Способ используется некоторыми сайтами и порталами, такими как Livejournal, Rambler и др. В настоящий момент (2015 г.) нет сведений о преимуществе более жёсткого варианта, по сравнению с вариантом, где используется единственный для каждой сессии секретный ключ[3].

- Referer (от ошибочного написания англ. referrer — отсылающий, направляющий) — в протоколе HTTP один из заголовков запроса клиента. Содержит URL источника запроса. Если перейти с одной страницы на другую, referer будет содержать адрес первой страницы. Часто на HTTP-сервере устанавливается программное обеспечение, анализирующее referer и извлекающее из него различную информацию. Так, например, владелец веб-сайта получает возможность узнать, по каким поисковым запросам, как часто и на какие именно страницы попадают люди. Если HTTP-клиент загружает с сервера картинку, представленную на какой-либо странице, то referer будет содержать адрес этой страницы. Некоторые HTTP-серверы перед выдачей картинки анализируют referer и не показывают картинку, если запрос приходит с другого сайта (а, например, показывают маленькое изображение-заглушку).

- Любопытно, что написание английского слова referrer как referer — популярная ошибка. Настолько популярная, что вошла в официальные спецификации протокола HTTP.

- Как уже упоминалось, бывает, что сервер отказывается выдавать нужное содержимое без определённой строки referer, поэтому многое клиентское ПО имеет возможность выставить эту строку вручную. Например wget поддерживает опцию «--referer», позволяющую выставить нужную строку и получить доступ к требуемому содержимому веб-сервера.

Создаем шаблон

- В файл settings.py в mysite помещаем ссылку на каталог где хранятся шаблоны
- TEMPLATE_DIRS = (
- os.path.join(os.path.dirname(__file__), '../myappl/html').replace('\\','/'),
-)
- Можно сделать более логичную структуру каталогов соответственно приложениям

```
<html> <meta charset="UTF-8"> <title> Главная страница </title>
<body>
<p>Request {{ name }} </p> <p>ajax {{ ajax }} </p> <p>ajax {{ csrf_token }} </p>
 <button onclick="loadLidar()" id="button">
Start Lidar!
</button>
  <script>
    function loadLidar() {
      var xhr = new XMLHttpRequest();
      try {       csrftoken = document.cookie
      alert(csrftoken)
       }
                catch (err) {
                  alert('error');
                  }
xhr.open('POST', '', true);
xhr.setRequestHeader('X-CSRF-Token', document.cookie);
xhr.setRequestHeader('COOKIES', document.cookie);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
      var json = JSON.stringify({
                name: "Виктор",
                surname: "Цой"
      });
```

```
xhr.send('csrfmiddlewaretoken=4GUAATD63MNlMV1SOLRQAcQR1MORyMh1&json='+json);
    xhr.onreadystatechange = function() {
      if (xhr.readyState != 4) return;
      button.innerHTML = 'Готово!';
      if (xhr.status != 200) {
        // обработать ошибку
        alert(xhr.status + ': ' + xhr.statusText);
      } else {
        //вывести результат
        alert(xhr.responseText);
      }
    }
    button.innerHTML = 'Загружаю...';
    button.disabled = false;
  }
 </script> </body> </body> </html>
```

Семантическая паутина

- RDF
- OWL