

## Теория

### Веб-сервер (Web Server)

Веб-сервер представляет собой программное обеспечение или аппаратное устройство, предназначенное для обработки запросов от клиентов (например, веб-браузеров) и предоставления им веб-страниц, данных или других ресурсов через протокол HTTP (Hypertext Transfer Protocol) или его безопасную версию HTTPS.

Основные функции веб-сервера включают прием запросов от клиентов, обработку этих запросов, взаимодействие с веб-приложениями или базами данных для получения данных, и отправку клиентам ответов в виде веб-страниц или других ресурсов.

Веб-серверы выполняют роль посредника между клиентами и веб-приложениями, обеспечивая передачу данных по сети. Они также могут обеспечивать дополнительные функции, такие как управление сессиями, обработка ошибок, аутентификация и авторизация пользователей.

### Конфигурация веб-сервера

**Виртуальные хосты** (Virtual Hosts): Эти объекты позволяют настроить один физический сервер для обслуживания нескольких доменов или поддоменов. Каждый виртуальный хост имеет свою собственную конфигурацию, определяющую параметры обработки запросов и корневой каталог для каждого сайта.

**Директивы** (Directives): Директивы представляют собой инструкции, определяющие параметры конфигурации веб-сервера. Это может включать в себя настройки безопасности, параметры производительности, местоположение файлов журналов и другие параметры.

**Модули** (Modules): Веб-серверы часто поддерживают модульную архитектуру, позволяющую добавлять или отключать функциональность.

Конфигурация модулей включает в себя активацию, деактивацию и настройку параметров.

**Настройки безопасности** (Security Settings): Эти параметры определяют, как веб-сервер обрабатывает запросы, чтобы предотвратить атаки или несанкционированный доступ. Конфигурация может включать в себя настройки SSL/TLS, ограничения доступа, фильтрацию запросов и другие меры безопасности.

**Настройки производительности** (Performance Settings): Опции, влияющие на производительность веб-сервера, такие как управление соединениями, буферизация данных, кэширование и балансировка нагрузки.

**Параметры обработки запросов** (Request Handling Settings): Конфигурация, связанная с обработкой запросов, такие как методы HTTP, обработчики запросов, перенаправления, и обработка ошибок.

**Настройки журналирования** (Logging Settings): Определение того, какие события и данные должны быть занесены в журналы сервера, включая формат и расположение журналов.

**Параметры сеансов** (Session Settings): Конфигурация параметров управления сессиями, включая хранение данных сеансов, время сеанса и другие связанные параметры.

## Настройки производительности веб-сервера на примере Nginx

**worker\_processes**: Определяет количество рабочих процессов, которые будут обрабатывать соединения. Значение этого параметра обычно устанавливается равным количеству ядер процессора.

Пример: `worker_processes 4;`

**worker\_connections**: Устанавливает максимальное количество соединений, которое каждый рабочий процесс может обслуживать одновременно.

Пример: `worker_connections 1024;`

**keepalive\_timeout**: Задает время ожидания (в секундах) для keep-alive соединений. Это время определяет, как долго соединение будет оставаться открытым после завершения запроса.

Пример: `keepalive_timeout 65;`

**sendfile**: Определяет, будет ли использоваться системный вызов `sendfile` для передачи файлов. Это может значительно улучшить производительность при передаче статических файлов.

Пример: `sendfile on;`

**tcp\_nopush** и **tcp\_nodelay**: Опции, связанные с передачей данных по TCP. `tcp_nopush` позволяет объединять маленькие пакеты данных в большие, что может быть полезно при передаче файлов.

`tcp_nodelay` отключает алгоритм Nagle, уменьшая задержки в передаче данных, но может привести к увеличению количества мелких пакетов.

Пример: `tcp_nopush on; tcp_nodelay on;`

**gzip**: Определяет параметры сжатия Gzip для передачи данных. Сжатие данных может существенно уменьшить объем передаваемых данных и ускорить загрузку.

Пример:

```
gzip on;  
gzip_comp_level 5;  
gzip_min_length 256;
```

**open\_file\_cache**: Позволяет кэшировать информацию о файлах для ускорения доступа к статическим файлам. Пример:

```
open_file_cache max=1000 inactive=20s;  
open_file_cache_valid 30s;  
open_file_cache_min_uses 2;
```

**proxy\_buffering**: Управляет буферизацией ответов при использовании Nginx в качестве прокси-сервера. Отключение буферизации может быть полезным при стриминге больших файлов. Пример: `proxy_buffering off;`

**client\_max\_body\_size**: Устанавливает максимальный размер тела запроса. Это полезно для предотвращения DDoS-атак и управления загрузкой сервера.  
Пример: client\_max\_body\_size 10M;

### Настройки безопасности веб-сервера на основе Nginx.

**server\_tokens**: Эта директива управляет отображением версии Nginx в HTTP-заголовках ответа. Рекомендуется отключить отображение версии для снижения риска атак, связанных с известными уязвимостями.  
Пример: server\_tokens off;

**location / { deny all; }**: Эта конфигурация запрещает доступ к корневому каталогу сервера. Обычно используется вместе с конфигурацией виртуальных хостов.

Пример:

```
server {
    listen 80;
    server_name example.com;

    location / {
        deny all;
    }
}
```

**limit\_req\_zone** и **limit\_req**: Ограничение запросов по времени для защиты от DDoS-атак.

Пример:

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

server {
    location / {
        limit_req zone=one burst=5;
    }
}
```

**ssl\_protocols** и **ssl\_ciphers**: Контроль использования протоколов и шифрования для HTTPS-соединений. Рекомендуется отключить устаревшие протоколы и шифры.

Пример:

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers 'TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384';
```

**add\_header Strict-Transport-Security**: Добавление HTTP-заголовка Strict Transport Security (HSTS) для защиты от атак посредника и принудительного использования HTTPS.

Пример:

```
add_header Strict-Transport-Security "max-age=31536000" always;
```

**client\_max\_body\_size**: Ограничение максимального размера тела запроса для предотвращения атак на основе переполнения буфера.

Пример: client\_max\_body\_size 10M;

**deny** и **allow**: Определение доступа к определенным IP-адресам или диапазонам IP.

Пример:

```
location / {  
    deny 192.168.1.1;  
    allow 10.0.0.0/24;  
    allow 192.168.1.0/24;  
    deny all;  
}
```

**proxy\_hide\_header:** Скрытие определенных HTTP-заголовков, чтобы уменьшить количество информации, доступной потенциальным злоумышленникам.

Пример:

```
proxy_hide_header X-Powered-By;
```

## Настройка балансировки в Nginx

Блок `upstream` в конфигурации Nginx используется для определения группы серверов, которые будут обслуживать запросы и служит основой для настройки балансировщика нагрузки. Этот блок позволяет Nginx распределить запросы между несколькими серверами для обеспечения более эффективного использования ресурсов и повышения отказоустойчивости.

**Варианты конфигурации upstream блока:**

1. **least\_conn:** Этот параметр выбирает сервер с наименьшей загрузкой. Он передает запрос на сервер с наименьшим числом текущих соединений.
2. **ip\_hash:** Этот параметр используется для постоянного маршрутизации запросов от одного и того же клиента к одному и тому же серверу. Это полезно, например, когда у вас есть сессионная информация, которую нужно сохранять на определенном сервере.

3. **round-robin**: Это значение по умолчанию. Он просто переключает запросы по кругу между серверами в блоке.
4. **hash**: Этот параметр позволяет определить кастомный ключ хеширования для балансировки нагрузки. Это может быть полезно, если вы хотите использовать другой критерий для распределения запросов, кроме IP-адреса или сессионной информации.

Пример

```
upstream app_servers {  
    least_conn;  
  
    server server1.site.com weight=1;  
  
    server server2.site.com weight=1;  
  
    server server3.site.com weight=1;  
}
```

Опция `weight=1` означает, что каждый сервер имеет одинаковый вес при балансировке нагрузки.

Ход работы:

### Установка Nginx

#### 1. Ubuntu/Debian

Откройте терминал (CTRL + T) и выполните следующие команды

```
sudo apt update
```

```
sudo apt install -y nginx
```

Конфигурационный файл хранится `/etc/nginx/nginx.conf`

## 2. Docker

Создайте файл с именем Dockerfile в корневой папке проекта с содержимым:

```
FROM nginx:latest
```

```
COPY nginx.conf /etc/nginx/nginx.conf
```

Выполните команду сборки образа:

```
docker build -t mynginx .
```

Выполните команду запуска контейнера:

```
docker run -d -p 80:80 mynginx
```

## 3. Windows

### 3.1. Загрузка бинарных файлов:

Перейдите на официальный сайт Nginx (<https://nginx.org/>) и загрузите последнюю версию для Windows. Обычно это архив в формате .zip.

### 3.2. Распаковка архива:

Распакуйте скачанный архив в удобное место на вашем компьютере, например, в C:\nginx.

### 3.3. Настройка переменных среды:

Добавьте путь к Nginx в переменные среды Windows, чтобы можно было запускать Nginx из любой директории.

- i. Перейдите в "Параметры системы" -> "Дополнительные параметры системы" -> "Переменные среды".
- ii. В разделе "Переменные среды пользователя" найдите переменную "Path" и нажмите "Изменить".
- iii. Добавьте путь к директории Nginx (например, C:\nginx) в список переменных среды.

### 3.4. Запуск Nginx:

- Откройте командную строку (cmd) с правами администратора.
- Перейдите в директорию, где у вас распакован Nginx (например, C:\nginx).

Запустите Nginx командой nginx

Конфигурационный файл хранится C:\nginx\conf\nginx.conf

### Разработка сервисов

Задача: разработать 3 сервиса на разных языках программирования (ЯП), которые будут доступны из браузера и выводить текст Hello World from <ЯП>.

Рекомендуется в качестве языков использовать PHP, NodeJs, Python.

### Пример приведен по ссылке

<https://drive.google.com/file/d/1GxwOrxQYXWd9PEFo1xuCSVoFqc6dhUv8/view?usp=sharing>

### Разработка конфигурации nginx

Задача: разработать конфигурацию nginx, которая бы позволяла nginx выступать в качестве балансировщика. Использовать секцию upstream.

Результат: по открытию ссылки <http://localhost> в браузере должен выводиться текст Hello World from <ЯП>.

### Произвести тестирование конфигурации

Задача: разработать программу, которая бы делала N запросов к адресу <http://localhost> определяла бы ЯП, на котором написан сервис (ЯП присутствует в ответе сервиса).

В качестве результата программа должна выдать сколько раз из N отработал сервис 1, сколько сервис 2 и сколько сервис 3.

Первое тестирование необходимо провести с weight = 1 у каждого сервиса.

Пример:

N = 90 при weight у каждого из сервисов = 1.

Результат:

PHP - 30 (weight = 1)

Python - 30 (weight = 1)

NodeJs - 30 (weight = 1)

Второе тестирование необходимо провести с различными weight у разных сервисов.

Пример:

N = 70 при weight у PHP = 1 , Python - 2, NodeJs - 4

Результат:

PHP - 10 (weight = 1)

Python - 20 (weight = 2)

NodeJs - 40 (weight = 4)

Вывод должен содержать Результаты тестирования и заключения о корректности работы балансировщика.

Контрольные вопросы:

1. Понятие и функции веб-сервера
2. Способы конфигурирования веб-сервера
3. Характеристики, влияющие на производительность Nginx
4. Характеристики, влияющие на уровень безопасности Nginx
5. Понятие балансировщика и роли Nginx в этом
6. Конфигурация блока Upstream Nginx