**Интернет** – это инфраструктура, объединяющая компьютеры в сети для передачи данных и обмена информацией.

**Схема организации сетевого оборудования:**

Компьютер – switch (может не быть) – роутер (может не быть) – цепочка маршрутизаторов (плюс DHCP –сервер и DNS-resolver) – роутер (может не быть) – switch (может не быть) - компьютер

*Компьютер* – любое устройство в сети, способное подключаться к ней и обмениваться данными с другими устройствами этой сети (сервер, компьютер, смартфон и тд)

*Switch* – сетевое устройство для соединения компьютеров по локальной сети, не имеет окна в глобальную сеть (работает до канального уровня модели OSI). Благодаря switch реализуется сетевая топология звезда.

*Маршрутизатор* – сетевое устройство, обеспечивающее маршрутизацию трафика. Основная функция – найти кратчайший путь, по которому нужно отравить пакет, исходя из конечной адресации пакета.

Маршрутизатор работает в купе с DHCP-сервером, который отвечает за назначение сетевым устройствам публичных ip адресов либо включает в себя его функционал. Также работает в купе с DNS-сервером, который хранит информацию о соответствии ip адресов и их доменных имен.

**Сетевая модель** – абстрактная модель взаимодействия сетевых протоколов. Описывает уровни архитектуры компьютерной сети.

Протокол – стандарт, определяющий по каким правилам, будут взаимодействовать различные программы в сети.

*Основные сетевые модели:*

OSI (Open System Interconnection) – Сетевая модель взаимодействия открытых систем.

7. Прикладной уровень (application layer) – уровень, который напрямую взаимодействует с данными от пользователей, то есть обеспечивает взаимодействие приложения с сетью (протоколы http, ftp, smtp, ssh и др.)

6. Представительский уровень или уровень представления (presentation layer) – преобразование данных предыдущего уровня во взаимно согласованные форматы (кодировка, шифрование, компрессия) и обеспечение обратных процессов (декодирование, дешифрование, декомпрессия).

5. Сеансовый уровень (session layer) – организует сеанс связи между компьютерами (сессия –время между открытием и закрытием соединения между устройствами)

4. Транспортный уровень (transport layer) – разбивает поток данных на сегменты и отмечает у каждого в заголовке порт получатель и отправитель. Обеспечивает надежность отправки данных за счет протоколов *TCP и UDP*.

Порт – некое число в диапазоне от 0 до 65535, которое используется для идентификации сетевого приложения на устройстве. Т.е. используются для направления сетевого трафика нужному приложению.

|  |  |  |
| --- | --- | --- |
|  | TCP | UDP |
| Надежность передачи данных | Обеспечивает надежную передачу данных с гарантией доставки без потерь и в правильном порядке. Использует механизмы подтверждения доставки и повторной передачи. | Передает данные без гарантии доставки, без подтверждения доставки или повторной передачи. |
| Управление потоком | Регулирует поток данных между отправителем и получателем, предотвращает перегрузку сети или переполнение буфера получателя. | Не управляет потоком данных. Отправляет данные на скорости, с которой они генерируются приложением. |
| Установка соединения | Установка соединения между клиентом и сервером (трехстороннее рукопожатие) | Соединение с сервером не устанавливается |
| Получение данных от сервера | Можно получать ответы от сервера в виде входящих пакетов данных | Не подразумевает получение пакетов от сервера |
| Затраты на накладные расходы | Обеспечение надежности и управление потоком приводят к большему объему заголовков и дополнительным затратам на накладные расходы. | Более легковесный и эффективный протокол без дополнительных механизмов надежности, что уменьшает накладные расходы. |
| Задержка и скорость | Из-за механизмов надежности и управления потоком может приводить к более высокой задержке и меньшей скорости передачи данных по сравнению с UDP. | Обычно имеет меньшую задержку и более высокую скорость передачи данных, так как отсутствуют механизмы надежности и управления потоком. |
| Применение | Чаще используется в приложениях, где надежность и последовательность доставки данных критически важны, таких как веб-серверы, электронная почта, передача файлов. | Применяется в приложениях, где более высокая скорость и меньшая задержка имеют большее значение, такие как приложения с реально временным аудио и видео взаимодействием (стриминг, приложения видеозвонков) |

3. Сетевой уровень (network layer) – разбиение сегментов предыдущего уровня на пакеты данных и определение маршрута передаваемых данных с помощью логической адресации (IP).

IP (Internet Protocol) – используется для маршрутизации доставки данных между устройствами сети на основе их IP адресов. С помощью IP адреса обеспечивается логическая адресация устройства, т.е. можно однозначно определить, куда отправлять пакет (за счет уникальности IP-адреса).

Приватные адреса – зарезервированные адреса для использования в локальных сетях и не могут напрямую использоваться в интернете. Если устройству с приватным IP адресом требуется выход в глобальную сеть, то роутер использует маску подсети, для определения находится ли запрос в пределах подсети или требуется выход в глобальную сеть - в случае необходимости применяется протокол NAT (каждому приватному IP адресу соответствует порт роутера).

Хост – устройство с публичным ip адресом. Подсеть – все устройства, которые обслуживает конкретное сетевое оборудование (т.е. по отношению к этому оборудованию все устройства являются подсетью, а он их хостом).

Маска подсети –используется для определения того, что запрос находится за пределами локальной сети (для получения адреса подсети путем применения логическое &)

Публичный адрес – уникальный адрес, используемый в глобальной сети интернет (уникальность контролируется организацией IPS).

127.0.0.1 – адрес localhost

192.168.xxx.xxx – адреса, зарезервированные для приватных ip

Версии протокола IP: IPv4 – использует 32 битную адресацию (4 миллиарда уникальных адресов), которое разделено на 4 октета, отделенные друг от друга точкой, каждый октет может принимать значения от 0 до 255.

245.14.23.61 – пример.

IPv6 – 128 битная адресация, каждая секция имеет 32 бита (4 шестнадцатеричных символа) разделенные двоеточием, что решает проблему исчерпания адресов в будущем.

2001:0db8:85a3:0000:0000:8a2e:0370:7334 – пример.

Сокет – комбинация IP адреса и порта.

DNS – ‘псевдоним ip адреса’ система сопоставления публичных ip адресов символьным обозначениям (система доменных имен) предоставляет возможность использования понятных человеку имен и решает проблему изменения ip адреса при смене локации (для компаний), в случае с доменным именем закрепленным за компанией, к этому доменному имени привязывается новый ip.

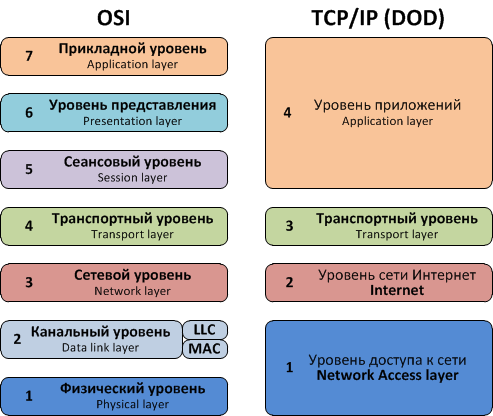
Получение Ip адреса по домену: сначала на компе был файл соответствия ip адресов и доменных имен – файл hosts. С ростом числа доменов стали использоваться DNS сервера, которых хранят эти соответствия.

2. Канальный уровень (data link layer) – разбивка пакета на кадры и доставка данных в пределах одной сети с помощью физической адресации (Ethernet).

MAC-адрес (MAC - Media Access Control) - физический адрес сетевого устройства, используется только на канальном уровне OSI (локальные сети) ввиду его непрактичности в глобальной сети (нужны огромные таблицы коммутации, чтобы знать, куда отправлять пакеты, обновление оборудования затруднит идентификацию устройства)

1. Физический уровень (physical layer) – реализован аппаратно и определяет методы передачи битов данных по физическим каналам.

TCP/IP (Transmission Control Protocol) – стандартное семейство сетевых протоколов, которые обеспечивают коммуникацию между устройствами в сети. Более практична и широко применяется на практике, в то время как модель OSI является более теоретической. Модель отличается от модели OSI кол-вом уровней, но они по-прежнему используют тот же набор протоколов, но имеют отличную группировку.



www. (world wide web) – распределенная система, представляющая собой систему взаимосвязанных посредством гиперссылок веб-страниц (гипертекстовых документов в интернете), для загрузки и просмотра которых необходима специальная программа — браузер. На прикладном уровне использует протоколы http или https.

То есть интернет != веб, это только одна из разновидностей сущностей, с которыми можно взаимодействовать с помощью интернета, помимо веб-сервисов так же присутствуют сервисы телефонии (технология VoIP), электронной почты (SMTP), обмена файлами (FTP) и др.

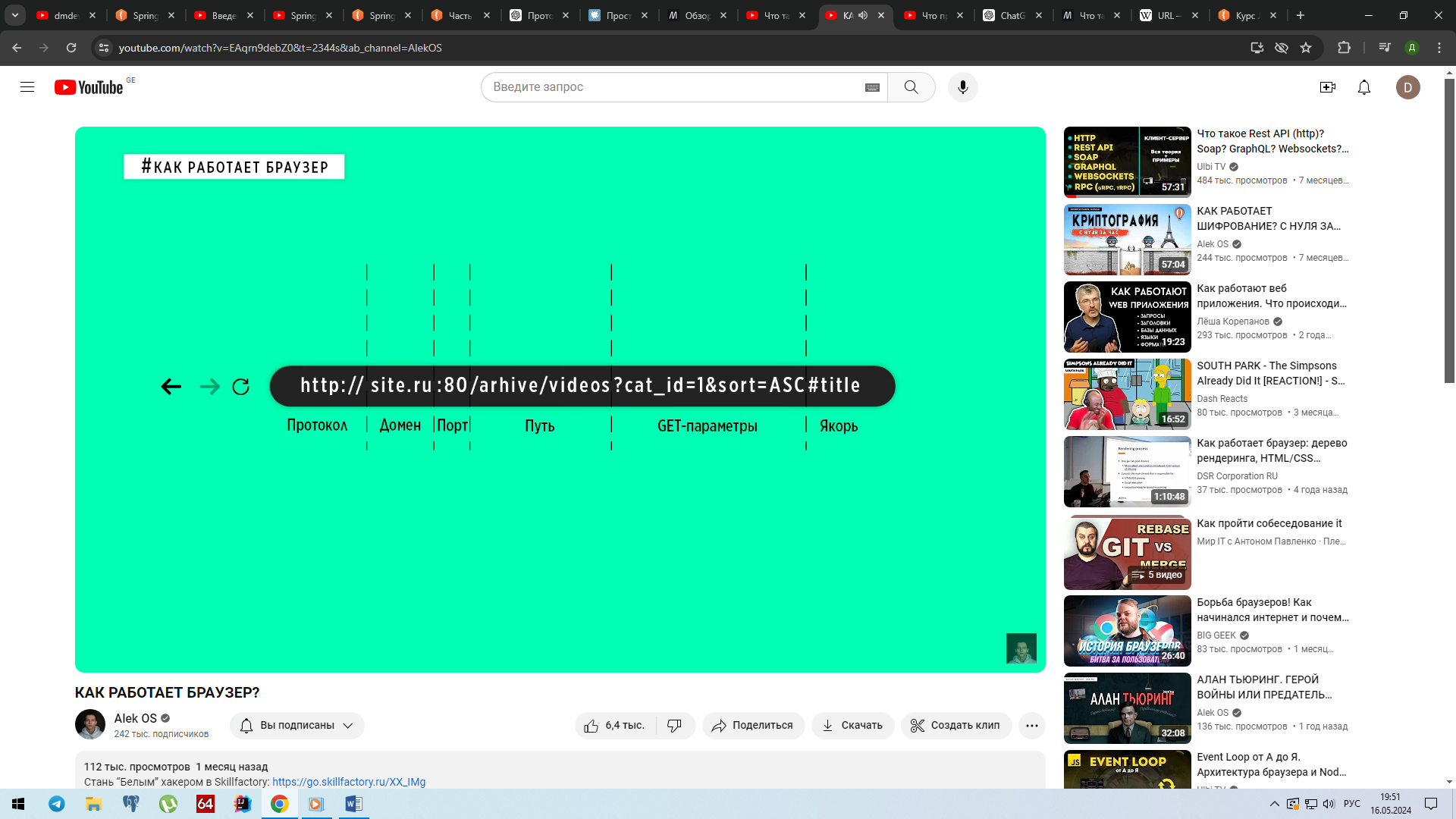
URI – уникальный идентификатор ресурса в интернете (URL + параметры)

URL (Uniform Resource Locator) – формат записи строки, которая предоставляет информацию о местонахождении ресурса в сети и способе взаимодействия с ним. Может включать только символы в кодировке ASCI, причем некоторые из них должны быть экранированы с помощью %, остальные кодируются с помощью последовательности байт в 16 формате, перед каждым байтом ставится %

<https://example.com/путь/с/символами/ы>

https://example.com/%D0%BF%D1%83%D1%82%D1%8C/%D1%81/%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%B0%D0%BC%D0%B8/%D1%8B

Состоит из:



Протокол – задает протокол взаимодействия между клиентом и сервером (http, https, ftp и др.)

Домен, порт – адрес, по которому расположен ресурс

Путь – указатель на конкретный ресурс на сервере

Get-параметры – дополнительные параметры, передаваемые на сервер

Якорь – метка на HTML-странице, которая указывает браузеру, что нужно проскролить страницу до этого якоря

Что происходит при обращении браузера к веб-сайту:

1) Определение поисковой запрос или url.

2) Если это введенная строка соответствует формату URL, то парсит его, если поисковой запрос, то обращение к веб-сервису поисковой системы по тем же шагам, введенный текст передается в качестве параметра этого запроса.

3) Парсинг: Определяется протокол взаимодействия (если протокол не указан, то браузер обратится к HSTS – preload list – механизм принудительно активирующие протокол https, который содержит домены, к которым должен быть применен https, если в списке его нет, то он попробует подсоединится с помощью https, если безуспешно, то http). Определение IP адреса с помощью DNS-адреса (ищется в кэшах, файл hosts, затем иерархически в DNS-resolver), если порт не был указан, то применяется порт по умолчанию http – 80, https -443.

4) Если использовался протокол https, то применяется TLS протокол

5) Установление соединения с сервером с помощью TCP-протокола

6) Формирование и отправка HTTP-запроса

7) Получение соответствующего ответа от сервера

**Протоколы Http и Https** – протоколы прикладного уровня для передачи данных (изначально применялся только к HTML документам) от клиента к серверу и обратно.

**Структура http сообщения** – состоит из стартовой строки, заголовков, тела.

Стартовая строка – строка, содержащая основную информацию о запросе и ответе

Заголовок – Его содержание характеризует тело сообщения и параметры его передачи и др. метаинформацию.

Заголовки и тело разделены пустой строкой.

Тело – содержит данные, передаваемые между клиентом и сервером.

Обязательными структурами http сообщения являются стартовая строка и заголовок со значением host (т.к. преобразование доменного имени в ip адрес происходит на стороне клиента и сервер не обладает информацией к какому доменному имени обратились).

|  |  |  |
| --- | --- | --- |
| Пример | Запрос по http://example.com/me | Ответ |
| Стартовая строка | GET/me HTTP/1.0 | HTTP 1.0 200 OK |
| Заголовок | Host: example.com | \*Заголовки\* |
| Тело | \*Тело запроса\* | \*Пересылаемый ответ\* |

|  |  |  |
| --- | --- | --- |
|  | *Запрос* | *Ответ* |
| Стартовая строка | Содержит метод, URI, версия HTTP | Содержит версию HTTP, код состояния, сообщение |
| Заголовки | Общие заголовки - Применяется для как для запроса, так и для ответа. | Cache-Control: Управляет кэшированием как клиентом, так и прокси-серверами.  Connection: Управляет аспектами управления соединением, например, поддержкой соединения открытым.  Date: Указывает дату и время, когда сообщение было отправлено.  Pragma: Используется для передачи директив, связанных с управлением кэшированием.  Trailer: Указывает, какие заголовки следуют после тела сообщения в формате chunked.  Transfer-Encoding: Указывает, какие типы преобразований были применены к телу сообщения, чтобы сделать его безопасным для передачи.  Upgrade: Указывает, что клиент хочет переключиться на другую протокольную версию.  Via: Указывает на промежуточные прокси-серверы, через которые прошел запрос или ответ.  Warning: Передает дополнительную информацию о статусе или преобразованиях сообщения. |
| Заголовок запроса клиента - Применяется только для запроса. | Accept: Указывает, какие типы медиа (MIME-типы) клиент может обрабатывать.  Accept-Charset: Указывает наборы символов, которые клиент может обработать.  Accept-Encoding: Указывает типы контент-кодирования, которые клиент может понять.  Accept-Language: Указывает языки, которые клиент предпочитает.  Authorization: Передает учетные данные для аутентификации пользователя на сервере.  Host: Указывает доменное имя сервера и (необязательно) номер порта, к которому клиент хочет подключиться.  If-Range: Выполняет запрос диапазона, если ресурс не изменился; иначе возвращает весь ресурс.  If-Modified-Since: Выполняет запрос только в случае, если ресурс изменился с указанного времени.  Range: Запрашивает определенный диапазон байтов ресурса.  Referer: Указывает URL, откуда был сделан запрос.  User-Agent: Передает информацию о клиенте, осуществляющем запрос (например, браузере или приложении)  И др. |
| Заголовок ответа сервера - Применяется только для ответа. | Accept-Ranges: Указывает, поддерживает ли сервер запросы диапазонов байтов.  Age: Указывает возраст объекта в кэше прокси-сервера.  ETag: Передает уникальный идентификатор версии ресурса (проверка актуальности кэшируемого ресурса)  Location: Указывает URL для перенаправления клиента (например, при статусе 3xx).  Proxy-Authenticate: Запрашивает у клиента учетные данные для аутентификации на прокси-сервере.  Retry-After: Указывает время, через которое клиент может повторить запрос.  Server: Передает информацию о серверном программном обеспечении.  Set-Cookie: Отправляет куки для сохранения клиентом.  Vary: Указывает, какие заголовки запроса использовались для выбора контента.  WWW-Authenticate: Запрашивает у клиента учетные данные для аутентификации на сервере. |
| Заголовок сущности - Определяет метаданные сущности. Если сущности нет, то метаданные URI запроса. | Allow: Указывает методы HTTP, поддерживаемые ресурсом.  Content-Encoding: Указывает кодировку, примененную к телу сообщения.  Content-Language: Указывает язык, используемый в теле сообщения.  Content-Length: Указывает длину тела сообщения в байтах.  Content-Location: Указывает альтернативный URL для тела сообщения.  Content-MD5: Указывает MD5 хэш тела сообщения для проверки целостности.  Content-Range: Указывает, какая часть полного ресурса передается.  Content-Type: Указывает MIME-тип тела сообщения.  Expires: Указывает дату и время, после которых ресурс считается устаревшим и его нельзя использовать из кэша.  Last-Modified: Указывает дату и время последнего изменения ресурса. |
| Тело | Содержит передаваемый контент | |

*URI* – путь до конкретного ресурса на сервере.

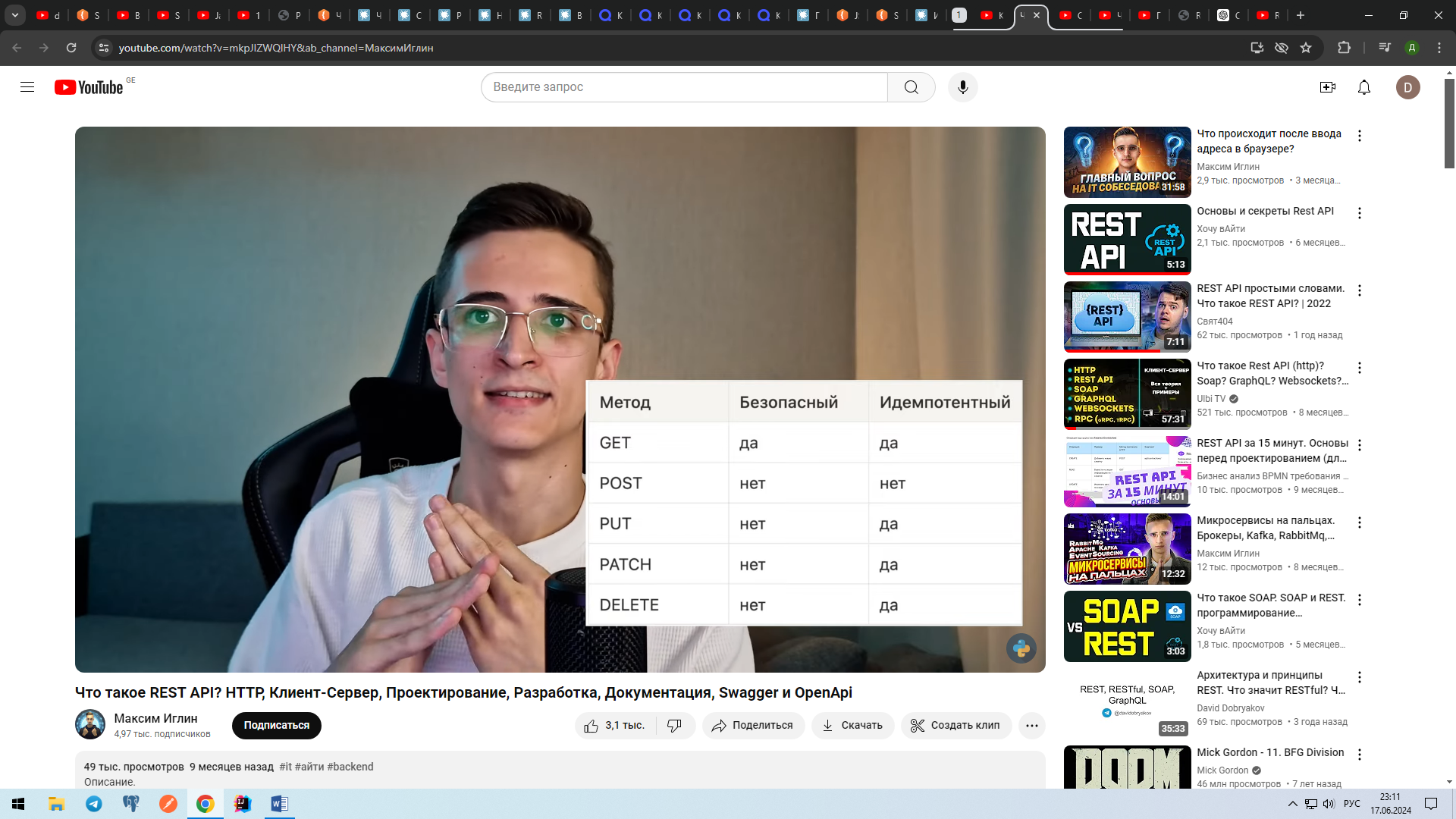
*Методы* – определяют действие, которое клиент хочет выполнить на стороне сервера.

|  |  |
| --- | --- |
| GET | Получение ресурса с сервера  Помимо обычного GET-запроса имеется также условные Get-запросы и частичные GET-запросы.  Условный - получение ресурса только в том случае, если ресурс изменился с последнего запроса клиента (с помощью заголовков If-Modified-Since или "f-None-Match) или использовать кэшированную версию.  Частичный – получение части ресурса с помощью определения диапазона байтов. |
| POST | Создание нового ресурса на сервере |
| HEAD | как GET, только сервер не отправляет тело, только заголовки (проверка существования ресурса или актуализация URL адреса) |
| PUT | обновление ресурса |
| PATCH | обновление фрагмента ресурса |
| DELETE | удаление ресурса |
| TRACE | позволяет увидеть путь прохождения запроса + его изменения |
| CONNECT | подключение через proxy |
| OPTIONS | определение возможностей сервера (какие методы и протоколы он поддерживает) |

*Безопасные и идемпотентные методы:*

Метод удовлетворяет критерию безопасности, если он не изменяет состояние ресурсов на сервере.

Метод удовлетворяет критерию идемпотентности, если его многократное использование приводит к такому же состоянию ресурса, если бы метод вызывался один раз.



Коды состояния – код, указывающий на результат обработки запроса. Информационное (100), успешное выполнение (200), перенаправление (300), ошибка на стороне клиента (400), ошибка на стороне сервера (500)

Сообщение – краткое текстовое описание кода состояния.

**Отличия протоколов**

|  |  |  |
| --- | --- | --- |
|  | HTTP | HTTPS |
| Безопасность | Не предполагает шифрование информации | Передает данные в нижестоящий протокол криптографический протокол ssl (устарел) или tls для обеспечения конфиденциальности передаваемых данных. Обеспечивает:  Шифрование данных (защита данных с помощью закрытого ключа) – предотвращает чтение перехваченных данных.  Сохранность данных (проверка целостности данных с помощью хэш-функции) – предотвращает возможность манипулирования данными во время передачи.  Аутентификация (защита от перенаправления) – см. сертификация. |
| Порт | 80 | 443 |
| Сертификация | Не имеет | Сервер аутентифицируется с помощью сертификатов, выданных специальными удостоверяющими центрами. Сертификация позволяет избежать атаки типа man in the middle.  Трехстороннее рукопожатие:  Установка соединения (обмен приветственными сообщениями с параметрами TLS соединения).  Аутентификация и обмен ключами - Сервер передает свой сертификат клиенту, содержит публичный ключ и информацию о удостоверяющем центре. Клиент обращается к центру сертификации, чтобы убедиться в подлинности.  Независимое вычисление закрытого ключа. |

**MIME type –** спецификация для передачи по сети файлов различного типа, которая указывает на тип и формат передаваемых данных в теле http. Указывается в заголовке Content-Type, состоит из type/subtype;parameter=value. Основные типы: application (бинарные данные любого вида), audio, example (для примеров), font (шрифты), image, model (описание 3d моделей), text, video.

**Версии http**

**Http 0.9 –** функционал был ограничен наличием метода GET, который возвращал только HTML страницу(отсутствовали заголовки, невозможно передавать данные отличные от HTML).

**Http 1.0** – Добавилась поддержка методов POST и HEAD, расширены типы передаваемого контента MIME-types, добавлены заголовки и коды состояния, базовая поддержка кэширования, при этом возможны только одноразовые соединения (клиент отправляет запрос, получает ответ и соединение закрывается).

**Http 1.1** – Добавлены новые методы, поддержка cookie, расширены возможности кэширования, поддержка постоянного соединения (дает возможность отправки нескольких запросов без пересоздания соединения), пайплайнинг запросов (возможность отправки нескольких запросов, не дожидаясь ответа на предыдущие), для его использования нужно точно знать границы сообщений (используются заголовки content-length или Transfer-encoding: chunked)

Cookie – служебная информация, которую сервер отправляет клиенту для ее сохранения и последующей передачи для идентификации этого клиента и его предыдущей активности, т.к. сервер не запоминает состояний (при авторизации пользователя на сайте сервер отправляет id сессии и при последующих запросах клиенту не требуется повторная авторизация)

По-прежнему актуальна проблема блокировки очереди запросов (если предыдущий медленный запрос блокировал последующие)

**Http 2.0** – Не изменилась общая семантика протокола, но изменился способ упаковки и передачи между клиентом и сервером, но сам набор данных остается прежним (для совместимости с протоколом 1.1). Рекомендуется использовать https соединение (не является стандартом).

*Бинарный формат вместо текстового –* представление http запроса в бинарном формате, а не в текстовом. *П*озволяет уменьшить объем передаваемой информации приходящейся на использование текстового формата.

*Физическое разделение на фреймы -* Использование физического разделения http запроса на фреймы (больше не используется логическое разделение на стартовую строку, заголовки и данные, которые по сути являются одним текстовым сообщением). Фрейм состоит из технического заголовка (тип фрейма, размер, флаги, идентификатор стрима) и данных, которые он должен передать.

*Мультиплексирование* – создание множества двунаправленных каналов между сервером и клиентом, в рамках одного соединения.

Позволяет отправлять несколько запросов одновременно (не последовательно, как в 1.1) и распараллеливать задачу отправки запроса и ответа.

*Server push –* принудительное наполнение кэша клиента данными не дожидаясь инициирующего воздействия со стороны клиента (отправка фрейма PUSH\_PROMISE) – можно реализовать логику работы сервера таким образом, что клиенту не придется создавать запрос на отправку ему какого-либо ресурса, т.к. сервер отправит его превентивно. Например, при отправке запроса на html страницу сервер может сразу же отправить ресурсы, которые включены в этот документ (js ресурсы, стили, изображения), что не требует формирование запроса клиентом.

*Сжатие заголовков методом HPACK –* Используется индексация заголовков, т.е. передается не символьное обозначение заголовка, а индекс соответствующий этому заголовку. Значение этих заголовков хранится в статической (часто повторяющиеся заголовки) и динамической таблицах, как на стороне сервера, так и на клиентской стороне для сопоставления индексов их значениям. Для значений этих заголовков используется сжатие методом Хаффмана.

Также уменьшает объем передаваемой служебной информации.

*Приоритизация запросов и выстраивание дерева зависимостей –* позволяет гибко управлять ресурсами сервера на основе полученной информации.

**Работа с сетевой моделью на транспортном уровне** (TCP и UDP) осуществляется за счет пакетов .net и .nio, и соответствующие им классы:

*В пакете .net:* Socket/ ServerSocket – протокол TCP

DatagramSocket – протокол UDP

*В пакете .nio:* SocketChannel/ServerSocketChannel – протокол TCP

DatagramChannel – протокол UDP

Классы позволяют управлять соединением на более низком уровне

|  |  |
| --- | --- |
| TCP | UDP |
| public class TCPClient {   public static void main(String[] args) throws IOException {  InetAddress address = Inet4Address.*getByName*("localhost"); //Класс для представления IP адресов  try (Socket socket = new Socket(address, 7777);  var outputStream = new DataOutputStream(socket.getOutputStream());  var inputStream = new DataInputStream(socket.getInputStream())) { //Созданеие клиентского сокета и получение потоков ввода и вывода сокета //для отправки запроса на сервер и получения ответа с помощью соответствующего потока  outputStream.writeUTF("Hello from client"); //отправка сообщения на сервер  System.*out*.println(inputStream.readUTF()); //обработка ответа от сервера   }  } } | public class UDPClient {   public static void main(String[] args) throws IOException {   try (DatagramSocket socket = new DatagramSocket()) {  //использует Datagram пакеты для отправки и получения сообщений, а не потоки  byte[] buffer = "Hello from UPD client".getBytes();  InetAddress address = Inet4Address.*getByName*("localhost");   DatagramPacket pack = new DatagramPacket(buffer, buffer.length, address, 7777);  //адрес передачи определяется в пакете, а не в сокете (в случае клиента)  // => используя один сокет можно отправлять пакеты разным серверам   socket.send(pack);  }  } } |
| public class TCPServer {   public static void main(String[] args) throws IOException {   try (ServerSocket serverSocket = new ServerSocket(7777);  Socket socket = serverSocket.accept(); //получение сокета клиента  DataOutputStream outputStream = new DataOutputStream(socket.getOutputStream());  DataInputStream inputStream = new DataInputStream(socket.getInputStream())) { //получение потоков клиентского сокета  System.*out*.println(inputStream.readUTF()); //обработка обращения  outputStream.writeUTF("Hello from server"); //Отправка ответа от сервера  }  } } | public class UDPServer {   public static void main(String[] args) throws IOException {  try (DatagramSocket socket = new DatagramSocket(7777)) { // в случае сервера нужно указывать порт  byte[] buf = new byte[512];  DatagramPacket pack = new DatagramPacket(buf, buf.length); //адрес передавать не нужно  socket.receive(pack); //получение переданного пакета данных  System.*out*.println(new String(pack.getData()));  }  } } |
| Блокирующие – сервер ожидает получения пакета, клиент ожидает ответа от сервера, если это TCP.  Для получения ответа от сервера необходимо на стороне сервера сформировать новый пакет и передать его клиенту (то есть клиент и сервер поменяются местами)  TCP и UPD порты – разные сущности, поэтому могут повторяться | |

**Классы прикладного уровня**

Классы URL и классы пакета .net.http используются для работы с соединением на прикладном уровне, класс URL предоставляет базовый функционал для работы с http – предоставляет минимальный набор функций для создания и управления http запросами.

Классы пакета .net.http предоставляют более удобный API (декларативный стиль), гибкие настройки http запроса (легко сконфигурировать http запрос), поддержку асинхронных запросов (не блокирует основной поток выполнения - completableFuture).

**Сервлет** – java программа, которая работает на серверной части приложения, обрабатывает запросы от клиента и формирует динамический ответ.

**ЖЦ сервлетов**:

1) Инициализация при вызове метода init() – вызывается контейнером сервлетов только 1 раз – происходит создание сервлета и инициализация необходимых для него ресурсов (создается либо при первом обращении к сервлету, либо при запуске сервера), после его создания контейнером сервлетов выделяются потоки исполнения для каждого нового клиентского запроса.

2) Обработка клиентского запроса вызовом метода service() – определение типа HTTP запроса и перенаправление его соответствующему http методу (doGet, doPut и др)

3) Уничтожение сервлета вызовом метода destroy() – вызывается контейнером сервлетов и выполняет операции по освобождению ресурсов перед уничтожением сервлета.

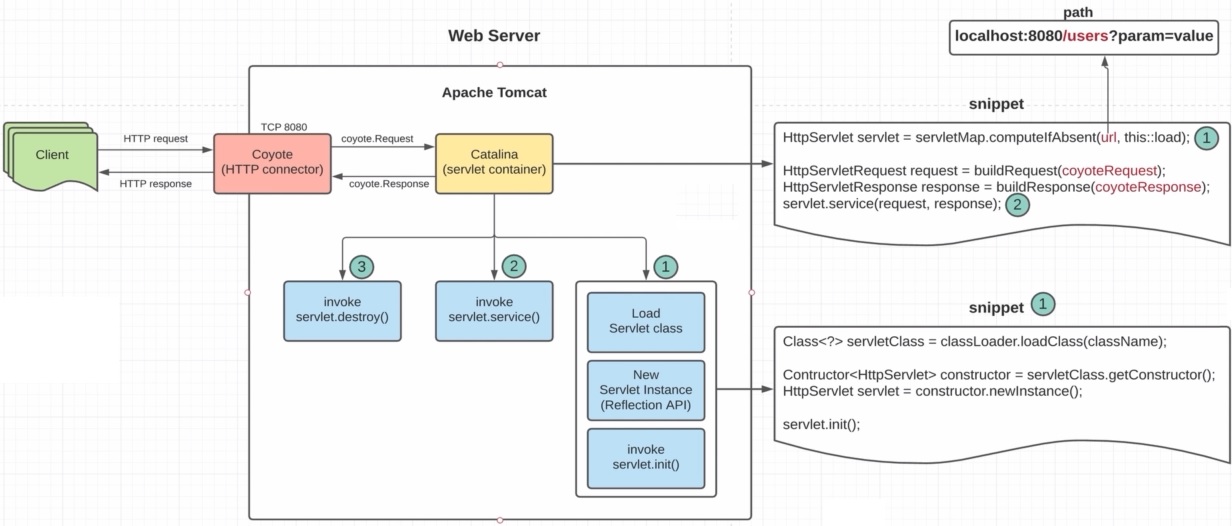
4) Полное уничтожение сервлета сборщиком мусора Java (Garbage Collector).

**Apache TomCat** – веб-сервер и контейнер сервлетов. Состоит из модулей:

Catalina – контейнер сервлетов. Управляет жизненным циклом сервлетов, обеспечивает взаимодействие coyote и jasper (является промежуточным звеном, которое перенаправляет запрос из coyote в jasper).

Coyote – Http connector – принятие/отправка запросов/ответов через http соединение (промежуточное звено, которое принимает запрос и передает его Catalina для обработки, также принимает ответ от Catalina и передает его клиенту).

Jasper – JSP движок – отвечает за компиляцию JSP (Java service pages) страниц в сервлеты и их последующее выполнение.



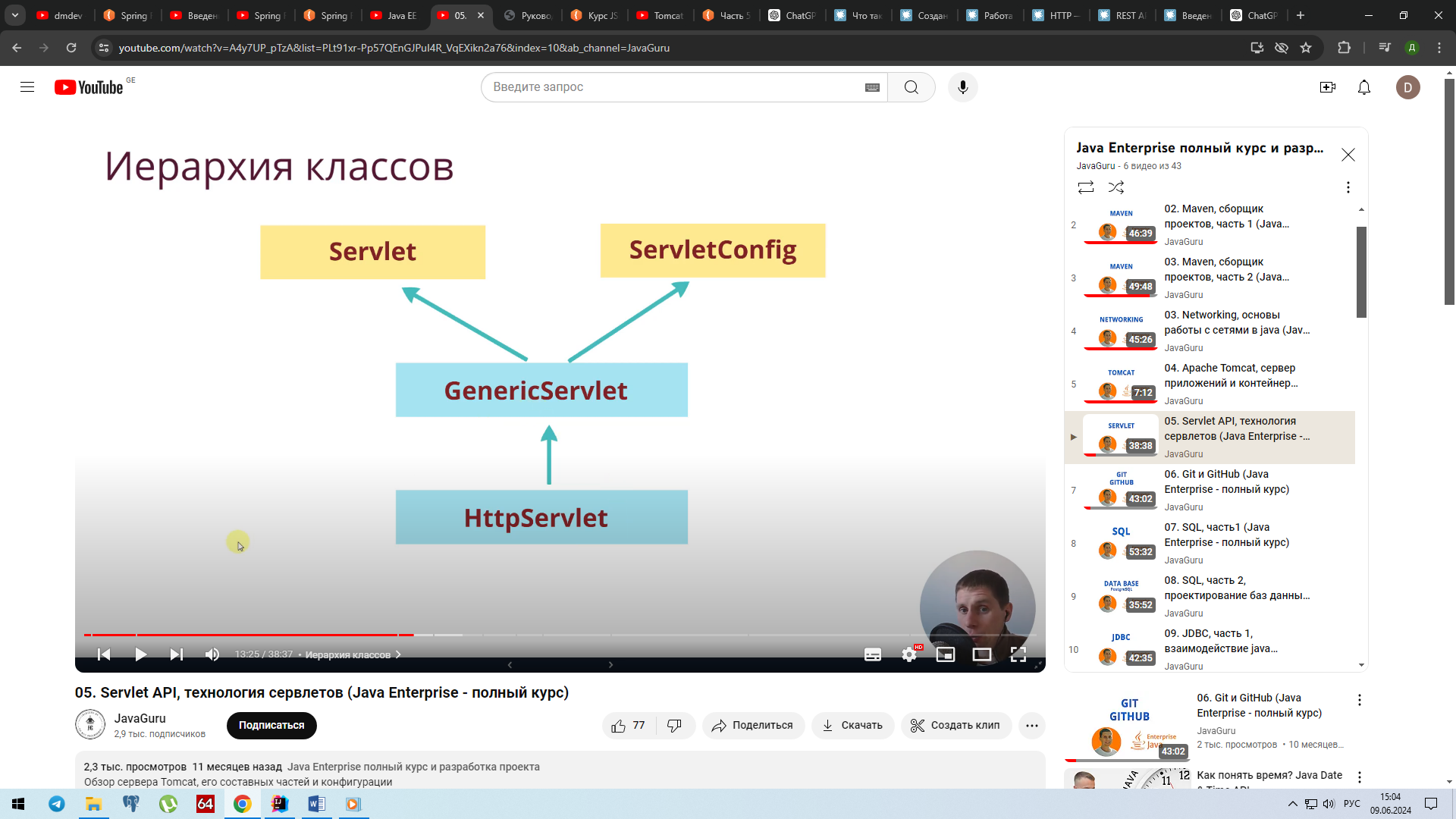
**Обработка http запроса:**

Http запрос поступает от клиента на коннектор Coyote (работает на транспортном уровне, используя классы Socket и ServerSocket), который принимает эти запросы от клиентов и оборачивает его в coyote.Request, что дает возможность абстрагироваться от деталей протокола HTTP (например унифицировать обработку запроса независимо от версии HTTP) и стандартизировать запрос, что дает возможность работать с запросом независимо от его свойств. Далее запрос принимает Catalina и так же оборачивает request и response для дальнейшей передачи в метод service.

Catalina представляет из себя ассоциативный массив, где URL (путь в нем) – ключ, а значение определенный сервлет, который должен отрабатывать по данному пути в URL. Если объект сервлета еще не был создан, то запускается механизм инициализации: загрузка класса с помощью ClassLoader, создание инстанса сервлета с помощью механизма рефлексии, вызов метода init.

Сервлет реализует концепцию существования единственного объекта, т.к. каждому сервлету соответствует определенный путь в URL (не является singleton, т.к. имеет открытый конструктор, создание объекта контролирует контейнер сервлетов, а не сам класс)

**Иерархия классов:**

**Servlet –** основнойинтерфейс, который определяет методы, которые необходимо реализовать сервлетам

**ServletConfig** – интерфейс, содержит конфигурационную информацию о сервлетах, используется для передачи информации об инициализации сервлета, параметры инициализации могут быть установлены в web.xml или с помощью аннотаций.

**GenegicServlet** – абстрактный класс, которые реализует некоторые методы, наследуется, если необходимо использовать сервлеты отличные от http

**HttpServlet** – абстрактный класс, включает реализации для обработки http запросов

**Виды сервлетов:**

\*\*\*\*\*\*\*

**Конфигурация сервлетов:**

Для конфигурации сервлетов используется дескриптор развертывания – web.xml (устаревший вариант) либо аннотации в коде.

Оба способа позволяют конфигурировать маппинг сервлета на URL, определять параметры инициализации и т.д.

**Запуск и развертывание**

Webapps – директория для развертывания веб-проекта, при добавлении war-архива в эту папку происходит ее развертывание для возможности дальнейшего использования, добавление war-архива в директорию ROOT приводит к тому, что обращение к ресурсу будет доступно по корневому URL.

Запуск и остановка TomCat осуществляется в директории bin (startup.bat, shutdown.bat)

Запуск и развертывание возможны с помощью IDE – edit configuration. Для ускорения развертывания можно использовать war-exploded (распакованная версия)

**Алгоритм работы:**

1) Унаследоваться от HttpServlet и переопределить основные методы жизненного цикла сервлета (init, service, destroy) – метод service() не переопределяется напрямую, так как в суперклассе определено поведение по умолчанию для него (определяет тип запроса и вызывает соответствующий метод doX, у которых также определено поведение по умолчанию в суперклассе), вместо негонеобходимо переопределить соответствующие методы doGet, doPost и т.д.

2) Должен присутствовать конструктор по умолчанию, для возможности использовать рефлексию.

3) Замапить сервлет на URL (с помощью аннотаций или web.xml)

**Работа с заголовками (**Headers**)**:

Получение заголовков http запроса с помощью объекта HttpServletRequest req и вызов у него методов по ключ слову Header (геттеры)

Отправка заголовков http ответа с помощью объекта HttpServletRequest resp и вызов у него методов по ключ слову Header (сеттеры и геттеры).

**Обработка параметров запроса:**

*Передача через URL (GET-запрос):*

example.com?id=123&id=234&pass=some – отделяются от пути знаком вопроса, между собой разделены &, у одного параметра могут быть несколько значений.

Параметры такого вида передаются в GET-запросе и обрабатываются в сервлете посредством методов типа getParameter();

*Передача через тело запроса (POST) – относится к работе с телом запроса, что подразумевает передачу параметров:*

**form-data –** Используется как подтип MIME-type multipart с заголовком Content-type. Позволяет передавать несколько различных типов данных в теле одного сообщения посредством объявления и использования разделителя (boundary).

Каждая часть данных имеет свой заголовок, который включает в себя метаданные содержимого в виде заголовков и передаваемый контент.

Пример:

POST /upload HTTP/1.1  
 Host: example.com  
 Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW  
  
------WebKitFormBoundary7MA4YWxkTrZu0gW  
 Content-Disposition: form-data; name="username"  
  
 JohnDoe  
------WebKitFormBoundary7MA4YWxkTrZu0gW  
 Content-Disposition: form-data; name="avatar"; filename="avatar.jpg"  
 Content-Type: image/jpeg  
  
 (здесь находится содержимое файла avatar.jpg в виде бинарных данных)  
------WebKitFormBoundary7MA4YWxkTrZu0gW--

**x-www-form-urlencoded** – используется как подтип MIME-type application с заголовком Content-Type. При использовании этого заголовка в теле запроса POST передаются параметры запроса, в виде аналогичном URL-параметрам.

Пример:

POST /submit-form HTTP/1.1  
Host: example.com  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 44  
Authorization: Bearer your\_access\_token  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36

username=johndoe&password=123456&rememberme=true

**raw –** для передачи данных в их исходном формате, используется для передачи данных текстового формата в теле запроса (text, JavaScript, JSON, HTML, XML).

**Binary –** для передачи данных в бинарном формате (изображения, аудио, видео)

**graphQL**

Для обработки тела запроса у HttpServletRequest req вызываются методы getReader или getInputStreamReader, в зависимости от получаемых данных. Для обработки x-www-form-unlencoded можно использовать методы типа getParameter()

**Загрузка файла с сервера:**

Для инициализации загрузки контента сервером на стороне клиента необходимо установить заголовки Content-Type, Content-length и заголовок Content-Disposition attachment; filename=xxx, который и будет указывать на то, что отправленный контент не нужно отображать, а необходимо сохранить в виде файла.

*Способы передачи:*

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 resp.setContentType("application/json");  
 resp.setCharacterEncoding("UTF-8");  
 resp.setHeader("Content-Disposition", "attachment; filename=employees.json");  
  
 //Так как приложением является TomCat, а текущий проект является просто war архивом,  
 // то для корректной отправки ресурсов необходимо указывать либо полный путь к ресурсам,  
 // либо пометить нужную директорию как resources root и получить stream этого ресурса с помощью  
 // FirstServlet.class.getResourceAsStream("/employees.json");  
  
 //с помощью указания полного пути  
  
 var writer = resp.getOutputStream();  
 writer.write(Files.*readAllBytes*(Path.*of*("C:\\Users\\dimir\\IdeaProjects\\demo\\src\\main\\resources", "employees.json")));  
  
 //с помощью стрима  
 InputStream resourceAsStream = FirstServlet.class.getResourceAsStream("/employees.json");  
 var stream = resp.getOutputStream();  
 stream.write(resourceAsStream.readAllBytes());  
}

**Работа с cookie –** На уровне http-сообщения серверу необходимо передать в ответе заголовок Set-cookie собязательнымипараметрами ключ-значение. Для более гибкого контроля над поведением cookie возможна установка опциональных параметров типа:

*Expires* (устанавливает дату истечения срока действия cookie – по умолчанию в течение всей сессии).  
*Max-Age* (время жизни в секундах, если установлен и Expires и Max-age, то приоритет на Max-age).  
*Domain* (домен для которого cookie должны быть отправлены, по умолчанию доступны для домена установившего cookie).  
*Path* (устанавливает путь для которого cookie должны быть доступны, по умолчанию для пути установившего cookie).  
*Secure* (только для https)  
*HttpOnly* (доступны только для http и недоступны для JavaScript)  
SameSite (контроль над cookie при крос-сайтовых запросах)

Средствами Java работа с cookie осуществляется с помощью класса Cookie, который позволяет конфигурировать отправляемый cookie методами типа set и получать клиентские с помощью методов типа get.

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 Cookie[] cookies = req.getCookies(); //получение клиентских cookie  
  
 Cookie cookie = new Cookie("name", "value"); //установка обязательных параметров  
 cookie.setMaxAge(20); //установка доп. параметра cookie  
 resp.addCookie(cookie); //установка cookie в ответное сообщение от сервера  
}

**Работа с сессией**

*Сессия в контексте соединения* – время, в течение которого между двумя узлами поддерживается связь для обмена данными.

*Сессия в контексте клиент-серверной архитектуры* – описание текущего состояния взаимодействия клиента и сервера, сохраненное на стороне сервера, которое осуществляется путем идентификации пользователя и объединения всех его запросов в единый процесс.

*Поддержание сессии –* это способ переслать серверу уникальный session id, которому соответствует объект типа Session.

Способы:

1) Аутентификация – для подтверждения себя пользователю необходимо вводить логин и пароль. Неудобно каждый раз логиниться.

2) Скрытое HTML поле – сервер отправляет session id путем внедрения его в HTML (добавляется скрытое поле, которое не рендерится браузером). Небезопасно, т.к. хранится в открытом виде.

3) URL Rewriting – передается с помощью параметра URL запроса. Небезопасно – передается открыто.

4) Cookies – передается с помощью cookie

5) Java Session Management API – механизм управления интернет-сессией (выбирает механизм управления между URL Rewriting или cookies, если отключены cookies) – TomCat использует этот механизм.

Взаимодействие с сессией осуществляется с помощью метода getSession() – позволяет настраивать атрибуты сессии (данные ассоциированные с пользовательской сессией) и время ее жизни.

Сессия на примере работы TomCat – если запрос от клиента инициируется впервые, то создается объект Session, генерируется session id и атрибуты сессии (Атрибуты используются для внутренней коммуникации сервлетов) сессия кладется в Map<String, Session>, где Sting – сгенерированный session id, этот id передается на клиент в виде cookie, который клиент затем передает на сервер, что позволяет ассоциировать его с определенной сессией.

*3 уровня атрибутов:*

ServletContext – глобальные настройки для приложения, доступны всем сервлетам, фильтрам, слушателям, сессиям.

HttpSession – доступны в рамках сессии.

ServletRequest – в рамках http запроса – используются при перенаправлении.

**Работа с файлами**

Для получения тела запроса используется getInputStream(), в таком случае будет взят весь контент, помещенный в тело запроса.

**getPart() –** Используется для получения части запроса из его тела, при использовании contentType multipart/form-data. Возвращает объект Part, который содержит все параметры полученной части запроса.

Для использования нужно использовать аннотацию @MultipartConfig(параметры)

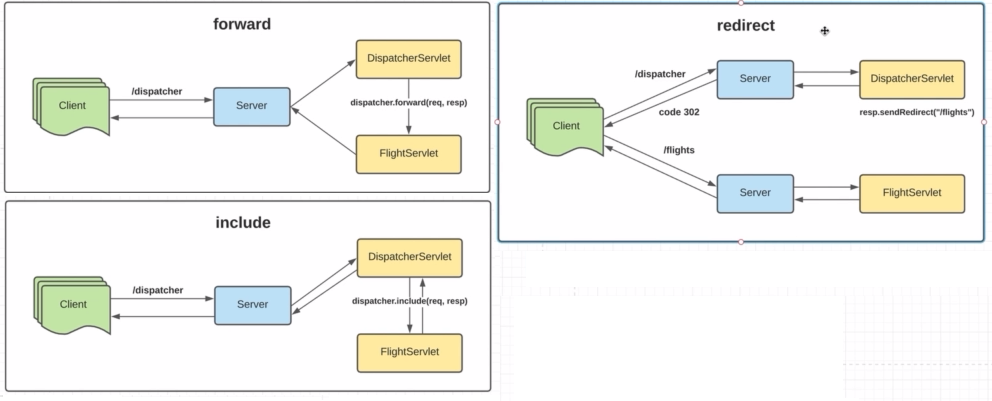
**Перенаправление запроса:**

Перенаправление может быть **серверное –** выполняется на стороне сервера (клиент не участвует в перенаправлении – не происходит перехода на другой URL). Используется для перенаправления запроса на другой сервлет в рамках одного приложения.

Серверное перенаправление осуществляется за счет интерфейса RequestDispatcher и его методов forward() и include().

forward() – передает управление указанному сервлету или другому серверному ресурсу (например, JSP странице). Новый ресурс обрабатывает запрос и формирует ответ для клиента. Оригинальный запрос и ответ передаются новому ресурсу.

include() – запрос передается на указанный ресурс, который обрабатывает запрос и передает результат обратно вызывающему сервлету, который включает результат его работы в свой ответ клиенту.



**Клиентское** – выполняется на стороне клиента. Сервер направляет сообщение о необходимости перехода на другой URL, что приводит к новому HTTP-запросу. Используется для перенаправления запроса на сервлет другого приложения или другой сервер.

Клиентское перенаправление осуществляется за счет resp.sendRedirect().

sendRedirect() – формирование ответа для клиента с кодом ответа 302 – redirect, что информирует его о необходимости перехода на указанный URL.

**JSP (Java Server Pages) –** технология для динамического создания веб-страниц, может содержать как статические (HTML), так и динамические компоненты (JSP-элементы).

JSP–страницы транслируются модулем Jasper в Java-классы, которые представляют из себя сервлеты.

**Директивы JSP**- управляют общими настройками и конфигурацией JSP страницы. Они предоставляют инструкции контейнеру JSP о том, как обрабатывать и управлять страницей**:**

*<%@page* – метаданные, конфигурация текущей JSP страницы (импорты, тип контента и др.)

*<%@Include* – для включения содержимого другой JSP-страницы в текущую

*<%@taglib* – подключение библиотеки тегов к JSP странице (тэгов, которые не относятся к html)

**JSP-элементы:**

*Скриптлеты* <% %> - для вставки многострочного Java-кода на стороне сервера.  
*Выражения* <%= %> - для вывода значений Java-выражений.   
*Декларации* <%! %> - для объявления классов, методов и полей.   
*Комментарии* <%-- --%> - для вставки комментариев в код.

\*Использование Скриплетов, выражений и деклараций в JSP не является хорошей практикой, так как в таком случае логика приложения перетекает на слой View, что нарушает MVC.

*EL (expression language) выражения* - ${} - является альтернативным и предпочтительным способом обращения с данными в JSP, т.к. при таком способе логика сервера не пишется в JSP, а происходит обращение к нужным переменным, которые содержат данные.

Контекст запроса JSP страница может получить с помощью различных видов scope (application/request/session/page) – и дополнительных объектов, определенных в requestScope – parameter, cookie, header.

Scope – используются для хранения и обмена данными между компонентами приложения (в частности между сервлетами). Контекст (данные) устанавливается посредством добавления атрибута (setAttribute()) к определенной области видимости.

${requestScope.user.name} – область видимости, получить объект по ключу, поле объекта (У поля должен быть геттер, иначе отвалится – Java Beans)

public class User {  
 private String name;  
   
 public String getName() {  
 return name;  
 }

${requestScope.SomeUser.name}

req.setAttribute("SomeUser", user);

*Теги JSTL* – библиотека стандартных тэгов для исполнения базовых задач на стороне jsp (некоторые функции, циклы, условия и др.). Используется для уменьшения и упрощения кол-ва кода на стороне JSP (для tomcat 10 2 зависимости: jakarta.servlet.jsp.jstl, jakarta.servlet.jsp.jstl-api)

@WebFilter()  
Сервлет-фильтр – сервлеты, реализующие интерфейс Filter, позволяют перехватывать запросы до их обработки сервлетом и модифицировать ответы после обработки сервлетом. Жизненный цикл схож с сервлетом, но вместо service() метода doFilter().

Цепочки фильтров: Использование аннотаций не гарантирует порядок вызова фильтров. Цепочки фильров создаются за счет указания адреса

Req – прямой порядок, resp – обратный порядок

Применение: установка глобальных атрибутов, логирование, аутентификация, шифрование

URL паттерн (“someURL/\*”)

Переменные метода doFilter представленные базовым классом ServletRequest, их можно приводить к HttpServletRequest, объект FilterChain для передачи запроса дальше по цепочке фильтров

**DispetcherType**

**Портлет**

**Архитектура ПО** – структурное проектирование системы, которое включает в себя определение ее компонентов (модулей, классов, сервисов), их взаимодействий (способы, которым модули взаимодействуют) и описание подхода к их разработке.

**Модули должны обладать:** сильной связанностью – компоненты внутри модуля направлены на решение одной четкой задачи, слабой зацепленностью – работа модулей не должна зависеть от работы других модулей.

Следование архитектуре ставит перед собой цель обеспечить максимальную простоту доработки ПО, развертывание приложения и масштабирования приложения.

**Клиент серверная архитектура** – структура распределенного приложения, в котором задачи распределены между двумя типами узлов: клиент и сервер.

Клиент – взаимодействует с пользователем и передает это взаимодействие на сервер, инициирует запрос серверу (выполнить действие, получить ответ).

Сервер – единый источник вычислений (независимо от того, сколько серверов в кластере) и хранения данных для множества клиентов. Отвечает на клиентские запросы.

Если сервер выступает в роли просящего, то он будет являться клиентом для отдающего сервера, то есть сервер также может быть клиентом по отношению к другому серверу.

Обмен данными происходит через сетевые протоколы.

С точки зрения Многоуровневой архитектуры клиент серверная архитектура может быть двухуровневая (физическое разделение уровней на клиентский и серверный) или трехуровневая (физическое разделение уровней на клиентский, бизнес уровень и уровень данных).

Физическое разделение означает, что уровни физически находятся на разных устройствах.

Причины отделение слоя данных – возросшая нагрузка на сервер из-за наличия БД: БД требуют много памяти и процессорного времени на обработку данных, у БД появилась своя вычислительная логика (тригеры, процедуры), отличающаяся процедура масштабирования (при возрастающей нагрузке на сервер логики ставится кластер серверов, на который распределяется вычислительная нагрузка, при масштабировании БД можно также поставить кластер (но нужно постоянно синхронизировать данные) либо использовать шардирование – вся БД размазывается на разные сервера.

*Архитектурные подходы для построения сервисной архитектуры:*

**Монолит**

|  |  |
| --- | --- |
| Описание | Все компоненты системы объединены в единый исполняемый файл (проект) – компилируется и развертывается как единое целое, имеют общую БД и части этого проекта не могут быть распределены на разные серверы или узлы. |
| Плюсы | Упрощенная разработка (не нужно разрабатывать взаимодействие и интеграцию), так как представляет из себя единую кодовую базу, которая не зависит от соседних сервисов.  Проще развертывать – требуется только настройка взаимодействия с БД  Проще поддерживать из-за отсутствия интеграции в другие приложения  Лучшая производительность за счет того, что внутренние вызовы и взаимодействие компонентов происходят без сетевых задержек. |
| Минусы | Сложность внедрения новых технологий (нет возможности писать часть приложения на одном языке, а другую на другом) – вся система должна использовать единый технологический стек. Громоздкая кодовая база – высокий порог вхождения в проект  Масштабируемость – Легко масштабируется технически, но неэффективно в случае, если нужно масштабировать только какой-то отдельный функционал, так как нет варианта масштабировать только его, нужно масштабировать всю кодовую базу, что является нерациональным |

**SOA (Service Oriented Architecture) - Сервис-ориентированная архитектура**

|  |  |
| --- | --- |
| Описание | Функции приложения представлены в виде независимых взаимосвязанных сервисов. Сервисы могут разрабатываться, масштабироваться и развертываться независимо друг от друга. Сервисы предоставляют интерфейсы для взаимодействия основанные на стандартах SOAP или REST.  Связь между сервисами происходит посредством специального сервиса – шины (ESB), наличие ее необязательно, сервисы могут общаться друг с другом.  Переиспользование сервисов |
| Плюсы | Гетерогенность - разбиение на сервисы дает возможность написания каждого сервиса на своем языке (использование множества технологий), может также негативно сказаться, так как для поддержания «зоопарка» технологий требует наличие большего кол-ва специалистов, что может затруднить разработку.  Независимость развертывания и масштабирования – каждый сервис может быть развернут на разных машинах, при высокой нагрузке на какой-либо сервис не нужно масштабировать все приложение, а сосредоточиться только на нагруженном сервисе.  Легкость подключения и отключения сервисов (если сервера общаются с помощь шины)  Независимость при командной разработке (каждая команда разрабатывает свой сервис и не мешают друг другу) |
| Минусы | Шина – точка отказа приложения логика – отказ шины приводит к отказу системы, также высокая нагрузка на шину является бутылочным горлышком приложения.  Сложность разработки из-за необходимости «дружить» сервисы друг с другом |

**MSA (Microservices Architecture) - Микросервисная архитектура**

|  |  |
| --- | --- |
| Описание | В целом схож с SOA, но сервисы представляют из себя небольшие и нацеленные на то, чтобы справляться только с одной задачей автономные совместно работающие сервисы.  Автономность выражается в избегании повторного использования сервисов (предпочитает дублирование, а не зависимость от других сервисов) и наличия у микросервисов всех необходимых компонентов для выполнения своей задачи |

**Различия SOA и MSA**

|  |  |  |
| --- | --- | --- |
|  | SOA | MSA |
| Переиспользование сервисов | Подразумевает переиспользование низкоуровневых сервисов, что влечет появление зависимостей | Отсутствие переиспользования сервисов – дублирование зависимостей вместо их переиспользования |
| Размер сервисов | Крупные сервисы, ответственные за несколько процессов (бизнес задач) | Сервис = процесс (бизнес задача) |
| Внесение изменений | Из-за наличия зависимостей в результате переиспользования изменения в одном сервисе могут приводить к необходимости изменений в другом | Отсутствие зависимостей позволяет избежать данной проблемы |
| Взаимодействие | Шина ESB или прямое обращение между сервисами – централизованный сервис для общения | Брокер сообщений – каждый сервис сам управляет взаимодействием, используя брокер сообщений |
| Развертывания | Состоит из взаимосвязанных модулей, что затрудняет процесс | Сервис может быть развернут независимо от других сервисов |
| Тестирование | Невозможность изолированного тестирования из-за зависимостей между сервисами | Каждый сервис имеет определенную границу и минимум зависимостей, что позволяет легко тестировать сервис в изоляции от других частей системы |
|  | Каждый сервис внутри может иметь любую структуру (монолит/может быть разделен на микросервисы) | Подразумевает использование MSA при разработке сервисов |

Хорошая практика – начинать новый проект на монолите, так как он быстрее и проще разрабатывается и по мере усложнения проекта разделять монолит на микросервисы, так как в случае масштабируемости микросервисы имеют преимущество

**API (application programming interface)** – программный интерфейс, который определяет способы взаимодействия с программой. Состоит из самого интерфейса и его описания. Работает по принципу изоляции реализации: рабочие процессы скрыты, а «наружу» торчит только предоставленные API функции или методы для взаимодействия с этими скрытыми процессами, т.е. API предназначена для взаимодействия программ.

**REST API** (Representational state transfer)– архитектурный стиль (набор ограничений, принципов проектирования, позволяющий добиться определенных свойств системы), описывающий взаимодействие компонентов распределенного приложения в сети. Создатель Рэй Филдинг.

*Предпосылки создания* – протокол http достаточно гибкий и не содержит в себе четких правил его использования.

*Цель* – унификация и стандартизация общения распределенных систем.

*Результат использования REST* – производительность, масштабируемость, гибкость к изменениям, отказоустойчивость, простота поддержки.

*Принципы:*

1. Клиент-серверная архитектура (Client-Server) – архитектурное разделение на клиент и сервер, и соответствующее разделение их функциональности. Клиент – пользовательский интерфейс, запрашивает данные. Сервер – принимает запрос, отдает данные.

+ : Масштабируемость – при возрастающей нагрузке возможно поставить несколько серверов.

Простота поддержки – изменение логики в работе клиента/сервера не несет в себе необходимость вносить изменения в соседних частях.

- : Единая точка отказа – отказ узла = неработоспособная система.

Увеличенная нагрузка на сеть.

2.Сервер не хранит информацию о клиентской сессии (Stateless) – каждый запрос от клиента должен быть независим от предыдущих (как будто он был сделан впервые), то есть при каждом запросе данные отправляются в полной форме (кто запрашивает данные и какие данные запрашиваются).

!Под утверждением, что сервер не хранит информацию о сессии имеется в виду, что он не сохраняет информацию о предыдущих запросах на сервер, при этом действия пользователя могут быть сохранены в виде сессии.

+: Масштабируемость сервера – т.к. каждый запрос несет в себе всю информацию о запросе, то сервер может быть просто клонирован и не важно на какой из клонов придет запрос.

Простота в поддержке – при возникновении ошибки будет залогировано сообщение от клиента и ответ от сервера, и не нужно дополнительно узнавать, какое состояние хранил сервер.

Позволяет использовать кэширование – не нужно запрашивать состояние у сервера

-: Увеличение нагрузки на сеть – из-за постоянной передачи информации в полной форме.

Усложнение логики клиентской части – информация о состоянии хранится на клиентской части (cookies, токены аутентификации)

Альтернативной реализацией, не относящейся к REST, является Stateful – сервер хранит информацию о сессии и использует ее при обработке последующих запросов. Позволяет не отправлять серверу запросы в полной форме.

-: Масштабируемость - необходима синхронизация состояний, либо писать дополнительную логику для направления запроса на необходимый сервер.

Поддержка - нужно узнавать состояние при возникновении ошибки.

3.Единообразие интерфейса/Универсальный интерфейс (Uniform Interface) – вводится ряд ограничений, благодаря которым язык общения между клиентом и сервером становится единообразным:

*Идентификация ресурса* (Identification of resources) – каждый ресурс должен иметь стабильный (не меняющийся при изменении состояния ресурса) уникальный идентификатор – URI.

*Манипуляция ресурсом через представление –* представление – полное текущее или желаемое состояние ресурса, то есть для изменения состояния ресурса необходимо предоставить не часть данных о нем (изменяемые данные), а его представление целиком, то есть все данные, которые должны характеризовать ресурс (представление)

*Самоописываемые сообщения* (Self-descriptive messages) – запрос и ответ должны хранить в себе всю информацию для его обработки. Т.е. не должно быть ситуаций, когда в первом сообщении содержатся данные, а во втором инструкции для взаимодействия с данными.

*HATEOAS* – гипермедиа как двигатель состояния приложения (Hypermedia as the engine of application state) – сервер должен подсказывать клиенту, какие действия он может предпринять дальше для взаимодействия с ним путем формирования ответа, в котором содержится не только ресурс, запрошенный клиентом, но и его связи с другими ресурсами и действия, которые с ним можно совершить.

4.Многоуровневость архитектуры (Layered system) – система может иметь более сложную структуру нежели клиент и сервер (несколько серверов, балансировщики, прокси сервера), каждый компонент не должен знать о том, как происходит цепочка вызовов дальше своих соседей.  
  
К примеру, приложение для вызова такси на телефоне в качестве клиента обращается к сервису поиска машин как к серверу, а этот сервис уже в качестве клиента взаимодействует с навигатором — сервером. При этом навигатор не видит данные, которые были отправлены в сервис поиска такси, а приложение на телефоне не знает, что сервис обращался к навигатору.

+: Возможность изменения общей архитектуры без доработок на стороне клиента или сервера

-: Нагрузка на сеть – больше участников = больше вызовов

Увеличение времени получения ответа – из-за увеличения цепочки прохождения запроса.

5.Кешируемость (Cache) – каждый ответ от сервера должен иметь пометку о возможности его кэширования на стороне клиента. (http заголовки или redis – разгрузка БД)

Важен при построении систем, которые обрабатывают большое кол-во однотипных запросов и данные не склонны к изменениям либо изменения в данных в формате реального времени не так актуальны.

+: Уменьшает кол-во сетевых взаимодействий

Уменьшение нагрузки на серверную часть

-: Усложнение логики сервера – необходимость прописывать алгоритмы кэширования

Риск получения неактуальных данных

6. (Необязательный) Код по запросу (Code on demand) - сервер в ответ на запрос может передать исполняемый код.

RESTfull – система, которая удовлетворяет всем принципам REST, то есть является реализацией данного стиля. Обязательными являются первые 5 пунктов, 6 опциональный.

RESTless – система, которая не полностью следует принципам REST, то есть может нарушать одно или несколько принципов. Возможно, более точное определение степени соответствия REST поможет определить модель зрелости RESTсервисов [Ричардсона](https://habr.com/ru/articles/319984/) (его модель рассматривает http как основной протокол, также определяет использование спецификации http протокола, как критерий соответствия REST)

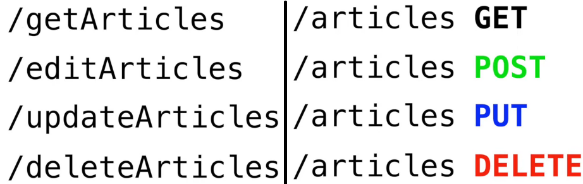
*Не является REST, но понимается по ним:*

1) REST - это только http протокол – ни один из принципов не говорит о том, какой протокол должен быть использован для соответствия REST (просто на практике это всегда http).

2) REST – это обязательно JSON – ни один из принципов об этом не говорит (может быть использован любой формат передачи данных).

3) Ресурсу должен соответствовать единственный URI, который должен описываться как существительное в единственном или множественном числе, которое будет интуитивно понятным и явно будет описывать сущность. За CRUD операции над этим ресурсом ответственны методы http, которые используются в соответствии со спецификацией:

Ресурсу Articles должно соответствовать не 4 разных URI, а 1 URI и на каждое действие свой http метод.



Также использование http-кодов ответа в соответствии со спецификацией http.

Так как использование http протокола не упоминается в контексте REST, а является скорее частой практикой, то этот пункт не является критерием REST, но, возможно, по причине того, что является в целом хорошей практикой, причисляется к REST.

**Альтернативы REST**

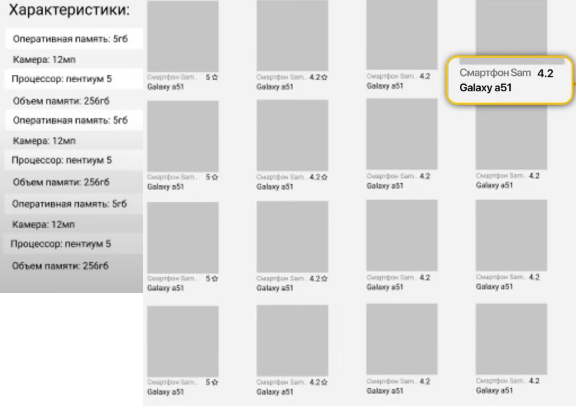
1) REST однонаправленный протокол (клиент инициирует обмен данными). Т.е. не подходит для систем, где необходимо, чтобы сервер самостоятельно инициировал передачу данных клиенту и эти изменения передавались в режиме «реального времени» (мессенджеры, онлайн игры). Для таких систем подойдет **WebSocket** (Stomp), если же время не критично, то можно реализовать long pooling с на базе REST – клиент опрашивает сервер на предмет изменений (нельзя делать слишком частые запросы, т.к. возрастает нагрузке, т.е. нужно искать баланс между нагрузкой на сервер и допустимым временем задержки).

2)Дорогая сериализация – сериализация объекта языка в JSON и наоборот накладывает большой оверхэд, альтернатива – **gRPC**, который более выгоден для межсервисного/межсерверного взаимодействия за счет облегченного формата Protobuf и использование http 2.0. (используется в микросервисах)

3)Ограниченная гибкость – каждый ресурс имеет фиксированную схему передачи данных, т.е. параметры остаются неизменными.

Есть ресурс телефон с определенным id, который содержит полный набор характеристик.

 В случае, если мы переходим на страницу с данной моделью телефона, нам нужен список всех его характеристик.

Но если мы хотим отобразить все доступные модели, то нам нужен ограниченный (отличный) набор характеристик того же ресурса.  


REST решения:

1) Сервер всегда возвращает полный набор данных для ресурса, а клиент выбирает, какие данные использовать (big data).

- увеличение нагрузки на сеть

- избыточность данных

2) Клиент запрашивает те данные, которые ему необходимы с помощью передачи параметров/фильтров (little data).

- сработает на малом кол-ве сущностей и параметров

3) Определить свой endpoint (URI) для каждого случая

- дублирование логики

- сложность масштабирования

Как альтернатива, возможно использование языка запросов GraphQL, который позволяет гибко описать необходимые фильтры и атрибуты.

**\*\*\***

**Стили (протоколы, языки, фреймворки), используемые для взаимодействия распределенных систем (помимо REST):**

**SOAP**

**WebSockets**

**GraphQL** язык запросов – позволяет в запросе описать необходимые фильтры и атрибуты

**gRPG**

\*\*\*

**MVC** – паттерн проектирования, который предполагает разделение логики приложения на 3 части в соответствии с их зоной ответственности. Можно рассматривать как специфическую реализацию слоистой архитектуры, которой соответствуют те же идеи.

Цель – соблюдение single responsibility, то есть разделение на слои в соответствии с его зоной ответственности. Стандартизация способа написания подобных друг другу систем – единообразная структура подобных проектов уменьшает время на то, чтобы вникнуть в неизвестный проект (упрощает понимание структуры проекта за счет известности его архитектуры).

**Слои:**

*View* (Представление) – Отвечает визуализацию данных и реализует интерфейс, посредством которого передаются действия пользователя.

*Model* – слой бизнес логики.

*Controller* – связующее звено между интерфейсом и логикой, обрабатывает - интерпретирует действия пользователя и передает их в Model, также полученные от Model данные передает на View.

**MVC на примере REST API на Java Servlets:**

*Слой View*: не подразумевается

*Слой Controller (Servlet):*

валидация данных – в идеале в контроллере не должно быть логики

формирование DTO-объекта и вызов сервиса

формирование ответа (успех/ошибка) по результатам ответа от Model

DTO (Data transfer object) – Объект, необходимый для передачи данных между слоями приложения либо между приложениями (отсутствие логики).

Цель – уменьшить кол-во сетевых вызовов между различными слоями приложения (делаем 1 вызов, упаковываем все данные в 1 объект). Подробнее про [DTO](https://www.youtube.com/watch?v=gD8xUkZW1GU&ab_channel=JPoint%2CJoker%D0%B8JUGru)

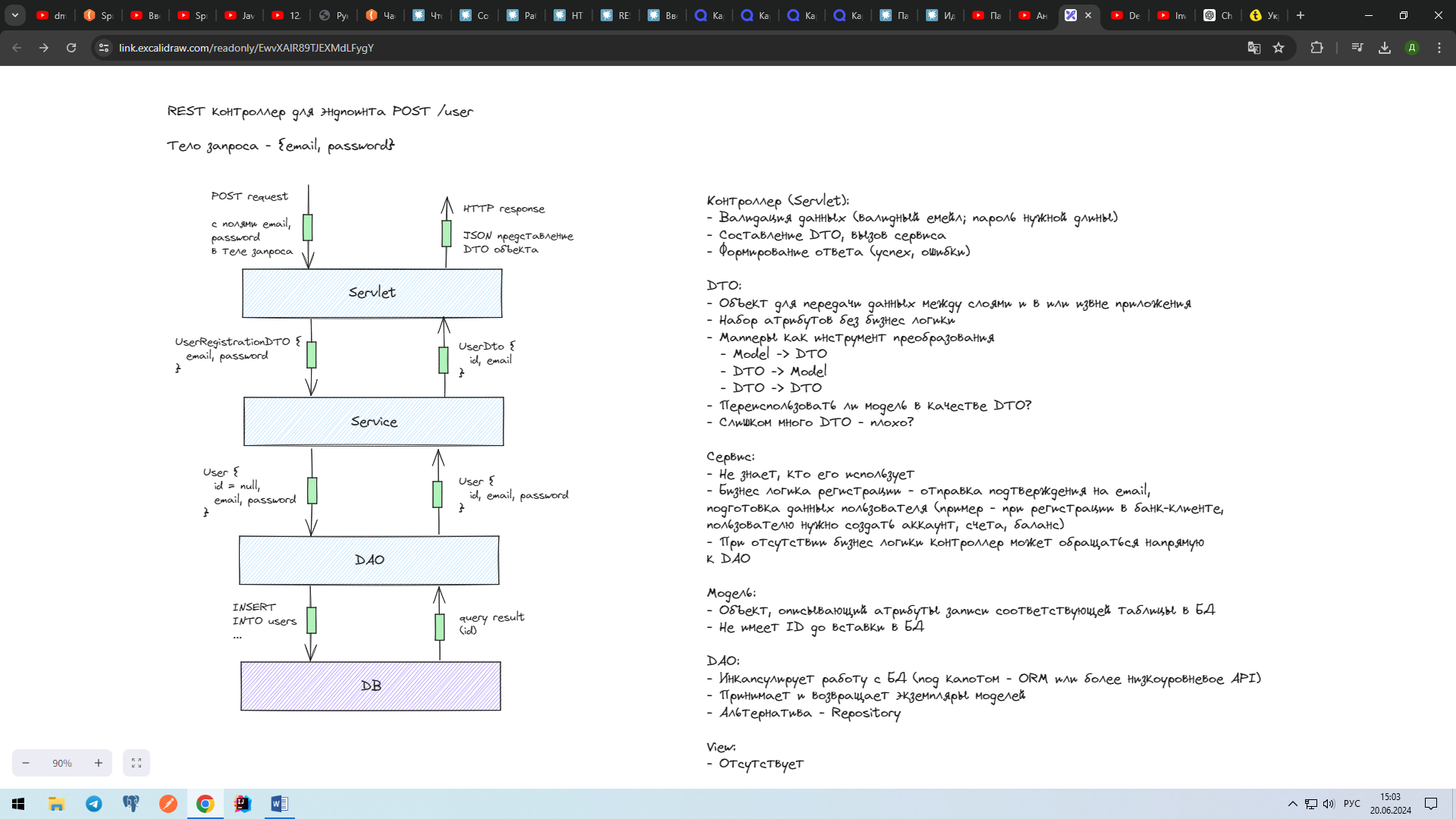
*Model* – представлена слоем Service, который работает с DTO (бизнес логика), DAO и DB.

Service – слой бизнес логики

DAO - инкапсулирует работу с БД

DB – База данных

Если не подразумевается логики обработки полученных от View данных, то Servlet может обращаться сразу к DAO, минуя слой логики, т.к. обрабатывать нечего.



MVVM MVP – разновидности MVC

редиректы

Аутентификация, авторизация, верификация

DRY  
KISS  
SOLID  
PATTERNS