Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

Автоматизированные системы управления

А.Я. Суханов

Разработка веб-сервисов для научных и прикладных задач

Учебное пособие

Томск 2022

Суханов А. Я.
Разработка веб-сервисов для научных и прикладных задач : Учебное пособие / А. Я. Суханов, . – Томск : ТУСУР, 2022. – 290 с.

Учебное пособие предназначено для студентов младших курсов бакалавриата направления 09.03.01 Информатика и вычислительная техника при изучении дисциплины «Разработка веб-сервисов для научных и прикладных задач», данное пособие может быть полезно для начального ознакомления с разработкой веб-приложений и изучения языка Python. Пособие знакомит читателя с основными современными технологиями разработки веб-сервисов и высоконагруженных приложений и использующимися при этом интернет технологиями.

© Суханов А. Я., 2022
© АСУ, ТУСУР, 2022

Введение 7
1 Краткий экскурс в историю телекоммуникаций, СЕТЕЙ и стандартизацию 9
1.1 Консорциум всемирной паутины W3C 19
1.2 IANA (Internet Assigned Numbers Authority) 21
1.3 Модель взаимодействия открытых систем 23
1.3 Модель DoD (Department of Defense) 30
2 Сетевые приложения и сервисы 35
2.1 Веб-приложение. 35
2.1.1 Мобильные веб-приложения. 36
2.1.2 Классические веб-приложения 40
2.1.3 Преимущество веб–приложений и их история. 41
2.1.4 Структура типичного веб приложения. 44
2.1.5 Типичная современная структура веб-приложения 45
2.1.6 Монолитное веб-приложение 51
2.1.7 Микросервисная архитектура веб-приложения 52
2.1.8 Одностраничное веб-приложение 52
2.1.9 Поставщик услуг приложений ASP (Application service providers) 52
2.2 Веб-сервис. 54
2.2.1 XML-RPC 55
2.2.2 SOAP 56
Примеры SOAP get и post 59
2.2.3 JSON-RPC 60
2.2.4 REST API 61
2.2.4.1 Примеры использования гипермедиа в REST 67
2.2.4.2 Примеры использования запросов HTTP в RESTful 69
2.2.5 JSON pure API 72
2.2.6 JSON API 73
2.2.6.1 Объекты ресурсов 75
2.2.6.2 Получение данных 76
2.3 Документирование веб-сервиса 78
2.3.1 Swagger (OpenAPI) 78
2.3.2 RAML 81
2.3.3 API Blueprint 84
3 XaaS (Anything as a Service, Что-либо как сервис). 86
3.1 Дата-центры 95

3.1.1 Стандарты оценки качества ЦОД 98
3.1.2 Уровни надежности ЦОД 103
3.1.3 Услуги которые предоставляет типичный дата-центр. 104
3.1.4 Colocation. 105
3.2 Типы вычислительных систем, размещаемых в ЦОД 108
3.2.1 Кластер 108
3.2.1.1 Кластеры высокой доступности. 109
3.2.1.2 Кластеры распределения нагрузки (Network Load Balancing, NLB) 110
3.2.1.3 Вычислительные кластеры 110
3.2.1.4 Устройство и состав кластера 111
3.2.2 Супер-компьютер 113
3.2.3 GRID-вычисления 116
3.3 Виртуализация 118
3.3.1 Миграция виртуальных машин (живая миграция) 119
3.3.2 Виды виртуализации. Виртуализация ресурсов 119
3.3.3 Виртуализация оборудования. 120
3.3.4 Виртуализация памяти 121
3.3.5 Виртуализация хранилищ данных 122
3.3.6 Виртуализация операционных систем 124
3.3.6.1 Программная виртуализация 125
3.3.6.2 Аппаратная виртуализация 128
3.3.7 Гипервизор 131
3.3.7.1 Xen 134
3.3.7.2 VMware Workstation. 139
3.3.8 Виртуализация на уровне ОС. Контейнерная виртуализация. 140
3.3.8.1 Docker 143
4. Примеры технологий реализации веб-приложений 146
4.1 Примеры реализации TCP соединений на основе сокетов 146
4.1.1 Пример сервера и клиента на Python 149
4.1.2 Пример сервера и клиента на java 149
4.2 Протокол DNS 151
4.3 Протокол HTTP 154
4.3.1 Структура HTTP запроса 156
4.3.2 Структура ответа и коды ответов HTTP. 158
4.3.3 URI 163

4.3.3.1 Data:URI, data: URL 164
4.3.4 Cookie 165
4.3.5 Авторизация и аутентификация в Web 168
4.3.5.1 HTTP аутентификация и SASL(Simple Authentication and Security Layer) аутентификация 168
4.3.5.2 JWT токен 171
4.3.5.3 Авторизация Auth 2.0 171
4.3.5.4 Open ID Connect 180
4.3 Протокол HTTPS 181
4.4.1 Пример поддержки HTTPS на python Flask 182
4.4.2 Пример создания самоудостоверяющего сертификата с использованием openssl 183
4.5 Протокол HTTP 2 187
4.6 Технологии формирования динамического контента 191
4.6.1 Технология CGI 191
4.6.2 PHP 193
4.6.3 JSP 194
4.6.4 ASP .Net 197
4.6.5 WSGI 198
4.6.6 Фреймворки для веб-приложений 199
4.7 Язык Python 201
4.7.1 Типы и структуры данных в Python 203
4.7.1.1 Аннотации типов. 203
4.7.1.2 Простые типы 204
4.7.1.3 Доступ к идентификатору объекта 204
4.7.1.4 Логический тип 204
4.7.1.5 Truthy/Falsy объекты. 206
4.7.1.6 Операции над числами. 206
4.7.1.7 Списки 207
4.1.7.8 Срезы. 207
4.1.7.9 Класс строки. 208
4.1.7.10 Tuple – класс кортежа 210
4.7.1.11 Класс множества (set). 211
4.7.1.12 Класс словаря (dict). 212
4.7.2 Основные операторы 213
4.7.2.1 Оператор if. 213

4.7.2.2 Оператор ternary if. 214
4.7.2.3 Цикл while и for. Break. Continue. 214
4.7.3 Функции Python. 215
4.7.3.1 Аргументы функции. Вызов функции. 215
4.7.3.2 Использование *args и **kwargs. 217
4.7.3.3 Область видимости переменных. 219
4.7.3.4 Лямбда выражение. 222
4.7.4 Декораторы python. 224
4.7.4.1 Модуль functools. 228
4.7.5 Генераторы Python. 232
4.7.6 Классы Python. 235
4.7.6.1 Декораторы методов. 239
4.7.7 Пример веб-приложения на Flask. 240
4.8 Балансировка нагрузки. 245
4.8.1 Уровни балансировки 245
4.8.1.1 Балансировка на сетевом уровне 245
4.8.1.2 Балансировка на транспортном уровне 246
4.8.1.3 Балансировка на прикладном уровне 247
4.8.2 Алгоритмы и методы балансировки 247
4.8.2.1 Round Robin 248
4.8.2.2 Weighted Round Robin 249
4.8.2.3 Least Connections и и другие алгоритмы 249
4.9 Аппаратное масштабирование архитектуры 252
4.10 CDN. 255
4.11 Потоковые сервисы 259
4.12 Кэширование. 263
ЗАКЛЮЧЕНИЕ. 268
Используемые источники литературы 269
ГЛОССАРИЙ 273

ВВЕДЕНИЕ

Современный мир трудно себе представить без информационных технологий, так как за несколько последних десятилетий они настолько прочно вошли в нашу жизнь, что мы порой не представляем ее без них. Усилия, направленные на информатизацию общества привели к развитию технологий поиска и предоставления раличной информации, требующейся в повседневной деятельности человека. Безусловно, одной из значимых вех в истории информатизации является появление глобальной сети интернет базирующейся на стеке протоколов TCP/IP. Конечно, без развития технологий связи, компьютерных систем было бы невозможно столь бурного ее роста. Но в данном учебном пособии мы лишь вкратце будем касаться этих технологий, не углубляясь подробно в различные уровни взаимодействия, в частности в нижние уровни близкие к физическому. Одна из самых знаменитых моделей описания уровней взаимодействия удаленных точек - модель взаимодействия открытых систем (ВОС англ. OSI – Open System Interaction), но она более подробно рассматривается в дисциплине «Сети и телекоммуникации», включая и различные физические аспекты взаимодействия. При этом уровни разделены между собой так, чтобы доступ к функциям нижестоящих уровней осуществлялся через унифицированный интерфейс и влияние изменений на нижестоящих уровнях на вышестоящие было бы минимальным или вовсе отсутствовало.

В реальности же в некоторых случаях, порой, и физический уровень может влиять на вышестоящие прикладные уровни и те решения, которые принимаются при проектировании тех или иных сетевых приложений. Хотя, как уже было сказано, сама концепция разбиения на уровни как раз и предполагает полную их независимость. Но, так называемая, проблема «колеи» [1, 2] присутствует в развитии человеческого общества и технологическом прогрессе и потому разработчики вынуждены порой мириться с решениями,

которые были приняты ранее или модифицировать свои разработки с учетом сложившихся реалий.

Данное же пособие посвящено созданию и разработке веб-приложений и веб-сервисов на основе современных программных технологий, это те сетевые приложения, которые непосредственно предоставляют какую-либо услугу сетевому пользователю или сетевым пользователям, или сетевым приложениям. Разработка таких приложений в основном подразумевает работу на высоких уровнях взаимодействия (прикладной, представительский и сеансовый уровни). Но в настоящее время все крупные сетевые сервисы представляют собой высоконагруженные приложения и порой обеспечивают работу и взаимодействие миллионов пользователей, что требует знания транспортного и сетевого уровней, технологий балансировки нагрузки, многопоточного программирования, принципов программирования для реализации кооперативной многозадачности, порой и принципов устройства КЭШей современных процессоров. Кроме того, требуется и знание Баз Данных, как реляционных (SQL) так и нереляционных, например, объектно-ориентированных или иерархических, но Базы данных в курс не входят.

В последние годы на смену информатизации приходит цифровизация, когда информация предоставляется в удобном виде не только пользователю, но и машинам и другим программам, обработка больших данных приводит к новым экономическим реалиям, давая толчок для развития цифровой экономики. Информационные сервисы, позволяющие максимально быстро и эффективно связать между собой поставщика и потребителя услуг становятся двигателями прогресса, а предприятия и компании, создающие такие сервисы одними из наиболее прибыльных.

1 КРАТКИЙ ЭКСКУРС В ИСТОРИЮ ТЕЛЕКОММУНИКАЦИЙ, СЕТЕЙ И СТАНДАРТИЗАЦИЮ

Известно, что телекоммуникации (telecommunication, tele – удаленный, communication – общение) начали развиваться раньше, чем электронные счетные машины, хотя устройства для различных вычислений вероятно реализовывались вместе с попытками доставить информацию участнику удаленного взаимодействия. Но вряд ли данные устройства участвовали в процессе доставки информации. Сейчас же вычислительные устройства являются непосредественными участниками телекоммуникационного взаимодействия.

Так же, если говорить о сравнительно недавних временах, то считается, что глобальные сети появились раньше локальных сетей, что было обусловлено громоздкостью и дороговизной вычислительных устройств, но порой возникала необходимость передать информацию от одного вычислительного устройства другому, из одного филиала организации в другой, либо при межведомственном взаимодействии и здесь пока вычислительные устройства тоже можно рассматривать лишь как участника взаимодействия. Но, в настоящее время, именно благодаря вычислительным устройствам стало возможным реализовать те методы цифровой обработки сигналов, которые увеличили надежность, скорость и помехоустойчивость при передаче информации. Кроме того, развитие самих вычислительных устройств, их миниатюризация, создание различных прикладных программ подтолкнули к созданию миниатюрных и надежных средств передачи информации, различных проводных и беспроводных линий связи, которые продолжают развиваться и на текущий момент.

На сегодняшний день вычислительная мощность некоторых миниатюрных устройств, размещающихся на ладони, во много раз превосходит вычислительную мощность устройств, занимавших целые комнаты в 50, 60-х годах XX века, давая возможность пользователю получать видео, звуковую и

текстовую информацию практически со всего мира, иметь доступ к различным услугам от заказа товаров до слежения за маршрутом своего следования.

Такое стремительное развитие обусловлено, в том числе, и стандартизацией в сфере телекоммуникаций, электросвязи и реализации вычислительных устройств. Очевидно, что без стандартизации каждый бы реализовывал свои средства связи, что создавало бы проблемы при взаимодействии, сопряжении устройств, требуя дополнительных затрат на создание таких устройств сопряжения.

Перечислим кратко основные институты и организации, занимающиеся стандартизацией в сфере телекоммуникаций. Наиболее старой и важной организацией является ITU (International Telecommunication Union, международный союз электросвязи, 1865 год) [3-4] отвечающий за выработку рекомендаций в области телекоммуникаций и радио, регулирующий распределение радиочастот. В настоящее время является специализированным учреждением ООН куда входит более 190 стран участниц и более 900 компаний, университетов и международных организаций. Несмотря на то, что данная организация выпускает рекомендации, следование им позволяет и облегчает компаниям предоставлять услуги связи по всему миру. Основной целью МСЭ является обеспечить каждому человеку эффективный доступ к информации и связи, для содействия в социально-экономическом развитии интересов всех людей. Это достигается либо путём разработки стандартов, используемых для создания инфраструктуры предоставления услуг электросвязи во всём мире и справедливого управления использованием радиочастотного спектра и спутниковых орбит, либо посредством предоставления поддержки странам в осуществлении их стратегий развития электросвязи. Целью Союза также является обеспечение и расширение международного сотрудничества в региональном использовании всех видов связи, совершенствование технических средств, их эффективная эксплуатация [3-4]. ITU ведет исследования по трем направлениям соответствующих трем секторам: сектор радиосвязи, сектор стандартизации и сектор развития. Вот что, например, приводится на са-

мом сайте ITU по поводу сектора стандартизации: «Стандарты МСЭ (называемые «Рекомендациями») имеют основополагающее значение для работы современных сетей ИКТ (информационно-коммуникационных технологий). Без стандартов Вы не сможете сделать телефонный вызов или получить доступ в интернет. Относящиеся к вопросам доступа в интернет, транспортных протоколов, сжатия голоса и изображений, создания домашних сетей, а также к огромному множеству других составляющих ИКТ, сотни стандартов МСЭ делают возможной работу систем на локальном и глобальном уровнях. Так, например, удостоенный премии «Эмми» стандарт МСЭ-Т Н.264 теперь является одним из наиболее популярных стандартов сжатия изображений. Обычно за год МСЭ создает и пересматривает более 150 стандартов, охватывающих все элементы – от функциональных свойств базовых сетей до услуг последующих поколений, таких как интернет-телевидение. Если вашему продукту или услуге в том или ином виде необходимо международное одобрение, Вы должны участвовать в обсуждении стандартов в Секторе стандартизации электросвязи МСЭ (МСЭ-Т)».

Еще одной известной организацией по стандартизации является ISO (International Organization for Standardization [5-6]), название сделано так, чтобы на всех языках оно звучало одинаково ИСО (с греческого равный). Потому на русском языке можно писать и говорить ИСО. Официальным языками являются английский, французский и русский, СССР являлся одним из основателем организации в 1946 году. Сама организация занимается только выпуском стандартов, но не сертификацией (проверкой на соответствие стандартам), этим занимаются независимые организации. Логотип ИСО является зарегистрированным товарным знаком и не может быть использован кем-либо за пределами ИСО без соответствующего разрешения. [5]. В организацию входит более 150 стран полноправных членов, кто может принимать участие в голосованиях за стандарт, влиять на содержимое стандарта. Работа по стандартизации проходит в соответствующих комитетах по различным направлениям. Начиналось все с порядка 60 комитетов, сейчас более 790

технических комитетов и подкомитетов. Разработка какого-либо стандарта проходит шесть стадий: предложения, подготовительная, комитета, вопросов, одобрения, публикации. Начинается все с решения о необходимости создания стандарта в какой-либо сфере, если голосование в данном техническом комитете за необходимость стандартизации даст более половину голосов за, и при этом, хотя бы пять участников согласятся работать над проектом, то создается рабочая группа и назначается лидер проекта. Далее создается рабочий черновик стандарта, который после согласования передается на стадию комитета, где регистрируется главным секретарем и рассылается по участникам технического комитета для голосования, здесь черновик может редактироваться пока не будет достигнут консенсус по техническому содержанию. После чего создается черновик международного стандарта, который в течение пяти месяцев проходит стадию вопросов для голосования, если две трети членов участников ИСО технического комитета проголосовали за и не более одной четверти против, то происходит переход на стадию одобрения, в ином случае обратно на предыдущую стадию для доработки. Стадия одобрения включает голосование всех членов ИСО за текст стандарта в течение двух месяцев, технические замечания не рассматриваются, но регистрируются для дальнейшего анализа при последующем пересмотре международного стандарта, здесь такой же количественный принцип по количеству за и против, как и на предыдущем шаге. В случае неодобрения стандарт возвращается в исходный технический комитет для исправления замечаний, голосовавших против. На шестой стадии публикации вносятся лишь небольшие редакторские правки и стандарт публикуется главным секретариатом ИСО. Каждые три года реализуется рецензирование стандартов, по прошествии которых осуществляется решение об отзыве или пересмотре путем голосования по принципу простого большинства голосов. Некоторые критикуют ИСО за влияние корпораций на стандарты и контроль за ними за счет наличия в комитетах большинства их представителей.

Важными стандартами с точки зрения телекоммуникаций была реализация ISO совместно с ITU модели и стека протоколов OSI и затем реализация ее в качестве международного стандарта ИСО/IEC 7498-1:1994 [6]. Сама же ИСО создала множество стандартов в соответствии со своими комитетами, начиная от ISO 4 — системы сокращения названий периодических изданий (журналов) и ISO 7 — трубной резьбы до ISO 8859 — кодовые таблицы символов или ISO 50001 — системы энергоменеджмента.

Еще одной организацией в мире стандартизации является IEEE (Институт инженеров электротехники и электроники), но стандарты, которые публикует данная организация больше относятся к стандартам физического и канального уровней, потому здесь мы просто приведем примеры некоторых стандартов. Например, IEEE 802.3 (Технология Ethernet), IEEE 802.11 (Беспроводные локальные сети Wi-Fi).

Рассмотрим организации, которые играют важнейшую роль в мире сетевых сервисов и технологий. Так уж исторически сложилось, что данные организации во многом занимаются развитием сети Интернет, в том числе благодаря тому, что стек протоколов TCP/IP набрал гораздо большую популярность, чем стек протоколов OSI или IPX/SPX (хотя второй какое-то время лидировал на рынке локальных сетей даже по сравнению с TCP/IP). Тем не мене, TCP/IP изначально был оптимизирован под глобальные сети в отличие от IPX/SPX, и был более прост, нежели стек протоколов OSI [7-8], что позволило занять ему пустовавшую нишу гораздо быстрее.

Когда была запущена сеть ARPANET управление перспективных исследовательских проектов (DARPA) Министерства обороны США под управлением Винта Сёрфа создало в 1979 году неофициальный комитет для наблюдения за сетью ICCB. В помощь ему для решения технических вопросов был назначен Дэвид Кларк из MIT. В 1984 году этот комитет был переименован в Совет по деятельности Интернета (Internet Activities Board, IAB). Перед советом была поставлена задача по удержанию исследователей, включенных в проекты ARPANET и Интернет, в более или менее одном направ-

лении. В последующем сокращение IAB было изменено на Совет по архитектуре Интернета (Internet Architecture Board) и его возглавил Дэвид Кларк, а редактором RFC стал Джон Постел. При этом каждый из приблизительно десяти членов IAB возглавлял специальную комиссию по отдельному важному вопросу. Совет по архитектуре Интернета собирался несколько раз в год для обсуждения результатов работы и представления отчета Министерству обороны и NSF (Национальный научный фонд США, реализовавший по технологии ARPANET сеть NSFNET ставшей хребтом (backbone) Интернета), которые в то время осуществляли их основное финансирование. Когда требовался какой-либо стандарт (например, новый алгоритм маршрутизации), члены совета прорабатывали этот вопрос, после чего объявляли об изменениях аспирантам, занимавшимся реализацией программного обеспечения сетей. Стандарты оформлялись в виде набора технических отчетов, называемых RFC (Requests for Comments). RFC доступны в Интернете для всех желающих (https://www.ietf.org/standards/rfcs/). Они пронумерованы в хронологическом порядке их создания. На сегодняшний день существует почти 9000 этих документов. К 1989 году Интернет вырос настолько, что подобный неформальный подход к его стандартам перестал работать. Например, многие производители уже стали предлагать продукцию на основе протокола TCP/IP и не хотели ее менять просто потому, что десятку исследователей пришла в головы одна хорошая идея. Летом 1989 года IAB был снова реорганизован. Исследователи были переведены в группу исследования Интернета (Internet Research Task Force, IRTF), и в группу проектирования Интернета (Internet Engineering Task Force, IETF). При этом IRTF стал подконтролен IAB. В совете IAB появились люди, представляющие более широкий спектр организаций, чем исследовательское сообщество. Вначале это была группа, в которой члены работали в течение двух лет, после чего сами назначали своих преемников. Затем в 1992 году было создано Интернет сообщество (Internet Society), в которое вошли люди, заинтересованные в развитии Интернета. Интернет-сообщество в каком-то смысле сравнимо с Ассоциацией по вычислительной

технике (ACM, Association for Computing Machinery) или IEEE. Оно управляется избираемыми доверенными лицами и предоставляет организационную основу для множества других консультативных и исследовательских групп, занимающихся развитием Интернета, включая IETF и IAB, в том числе доверенные лица утверждают состав IAB. Дело в том, что IETF и подобные ей организации были и остаются довольно неформальными с юридической точки зрения, но они нуждаются в финансовой поддержке и определённом правовом статусе. Для этих целей и было создано Общество Интернета [7].

Смысл же вышеуказанной реорганизации IAB заключался в том, чтобы сосредоточить IRTF на долгосрочных исследованиях, а IETF — на краткосрочных инженерных вопросах. Миссия IETF заключается в улучшении работы Интернета через создание высококачественных технических документов, которые оказывают влияние на то, как люди разрабатывают что-либо для Интернета, пользуются и управляют Интернетом. В настоящее время IAB совет является одной из комиссий Рабочей группы проектирования Интернета (англ. Internet Engineering Task Force, IETF) и осуществляет надзорные функции, в то же время обладая консультативным статусом при Обществе Интернета (англ. Internet Society, ISOC).

Проблемная группа IETF была разделена на рабочие группы, каждая из которых решала свою задачу. Первое время председатели рабочих групп встречались друг с другом в составе руководящего комитета для координации совместных исследовательских усилий. Рабочие группы занимались такими вопросами, как новые приложения, информационная пождержка пользователей, OSI-интеграция, маршрутизация и адресация, безопасность, управление сетью и стандарты. В настоящее время работа в основном ведётся через почтовые рассылки, но трижды в году проводятся собрания IETF. Результаты деятельности рабочих групп оформляются в виде рабочих проектов (англ. Internet drafts), которые затем используются ISOC для кодификации новых стандартов.

В конце концов, было сформировано так много рабочих групп, что их объединили по областям, после чего в руководящем комитете стали собираться председатели областей. Кроме того, был принят более формальный процесс стандартизации по аналогии с процедурой, принятой в ISO. Чтобы стать предлагаемым стандартом, основная идея должна быть полностью изложена в RFC и должна представлять достаточный интерес, гарантирующий ее рассмотрение. Затем, чтобы стать проектом стандарта, должна быть создана работающая реализация, которую нужно тщательно протестировать минимум двумя независимыми сайтами в течение 4 месяцев. Если IAB уверен, что идея адекватна и программное обеспечение работает, он может объявить RFC стандартом Интернета. Некоторые стандарты Интернета стали стандартами Министерства обороны США (MIL-STD), что сделало их обязательными к применению поставщиками министерства. Дэвид Кларк (David Clark) высказал замечание, ставшее ныне популярным, о стандартизации Интернета, состоящей из «грубого консенсуса и работающей программы».

Таким образом, Интернет сообщество, Internet Society (ISOC) - американская некоммерческая организация, основанная в 1992 году для обеспечения лидерства в областях, связанных с Интернетом, в том числе в области реализации стандартов, политики и образования. Миссия ISOC - «способствовать открытому развитию и использованию Интернета на благо всех людей во всем мире». Основные офисы находятся в Рестоне (Вирджиния, США), и Женеве (Швейцария). Девиз - «Интернет для всех». Является независимо финансируемым трастом, состоящим из отдельных членов, организаций-членов и отделений. Отдельные члены не имеют права голоса при определении политики, но организации-члены представлены в Консультативном совете, который может определять политику и направление, в котором будет развиваться Интернет-сообщество. Функции отделений Internet Society заключаются в реализации своих собственных планов в соответствии с политикой Internet Society, разработанной Консультативным советом, при условии утверждения и финансирования со стороны центрально аппарата. Internet

Society имеет большой оплачиваемый персонал и управляется Попечительским советом. Попечительский совет состоит из 13 человек. Четыре члена назначаются отделениями Internet Society, четыре члена назначаются Инженерным Советом Интернета (IETF, Internet engineering task Force), а четыре члена назначаются организационными членами Internet Society.

Членство в Internet Society со временем менялось. Internet Society потеряла 40 000 членов в 2018 году, а по состоянию на август 2020 года Internet Society указывает на своей домашней странице количество участников равное 72 334 и 97 организаций [7]. Данная организация обладает правами на все документы RFC.

Если говорить об историческом развитии Интернет, то кратко можно выделить следующие этапы:

После создания хребта NSFNET в 1985 году в 1989 году был реализован проект Всемирной паутины, разработанный Тимом Бернерсом-Ли. Появление Всемирной Паутины (World Wide Web, WWW) дало мощный толчок к популяризации и развитию Интернета.

В 1990 году научному сообществу был представлен первый текстовый браузер, позволяющий просматривать текстовые файлы, связанные гиперссылками. Уже в 1991 году пользователи получили свободный доступ к этому браузеру, однако распространение его вне научных кругов шло медленно.

Новый этап в развитии Интернета связан с выходом первой Unix-версии графического браузера Mosaic в 1993 году, разработанного Марком Андриссеном, стажировавшимся в Национальном центре суперкомпьютерных приложений (National Center for Supercomputing Applications, NCSA), США.

С 1994 года после выхода версий браузера Mosaic для операционных систем Windows и Macintosh, а вскоре вслед за этим – браузеров Netscape Navigator и Microsoft Internet Explorer, Интернет обрел популярность среди широкой публики сначала в США, а затем по всему миру.

В 1995 году NSF передала ответственность за Интернет в частный сектор. Это способствовало расширению круга коммерческих поставщиков и потребителей услуг сети Интернет, которая вскоре связала между собой миллионы компьютеров и сотни миллионов людей во всем мире.

После успешного проведения Всемирного дня IPv6 в 2011 году 6 июня 2012 года ISOC организовал Всемирный запуск IPv6, на этот раз с намерением оставить IPv6 постоянно включенным на всех участвующих сайтах.

В 2012 году ISOC запустил Deploy360, портал и обучающую программу для продвижения IPv6 и DNSSEC.

В 2016 году Deploy360 расширило свои кампании, включив в них взаимно согласованные нормы безопасности маршрутизации (MANRS) и аутентификацию именованных объектов на основе DNS (DANE).

В декабре 2017 года ISOC поглотил организацию по стандартизации Online Trust Alliance (OTA), которая проводит ежегодный онлайн-аудит доверия, Руководство по реагированию на киберинциденты и структуру доверия к Интернету вещей (IoT).

OTA определяет и продвигает передовые методы обеспечения безопасности и конфиденциальности, которые укрепляют доверие потребителей к Интернету. Ведущие американские государственные и частные организации, поставщики, исследователи и политики вносят свой вклад и следуют рекомендациям OTA, чтобы сделать онлайн-транзакции более безопасными и лучше защитить данные пользователей.

Цели OTA:

Посредством независимых исследований и сравнительных отчетов информировать заинтересованные стороны о проблемах Интернета, влияющих на доверие пользователей и репутацию бренда;

Стимулировать инициативы с участием многих заинтересованных сторон по разработке и продвижению передовых методов защиты безопасности, конфиденциальности и личности пользователей;

С помощью рабочих групп и комитетов развивать лидерство в отрасли и использовать возможности сотрудничества, значимого саморегулирования и управления данными;

Установливать и развивать партнерские отношения и возможности для совместного обучения;

Предоставить пользователям возможность действовать в своих интересах конфиденциальности и безопасности.

В 2018 году IETF обрела независимость от Internet Society, образовав собственное юридическое лицо (IETF Administration LLC). Internet Society обязалось производить платежи в IETF до 2020 года, чтобы помочь IETF создать благотворительный и резервный фонд, по истечении которого оно станет финансово независимым.

Далее рассморим отсальные важные для Интернета организации.

1.1 Консорциум всемирной паутины W3C.

Консорциум включает в себя более 400 организаций и занимается разработкой и внедрением стандартов Интернета, а также выработкой соответствующих открытых (то есть не защищенных авторским правом) рекомендаций, которые могут внедряться любым человеком без всяких финансовых отчислений консорциуму. Рекомендации W3C зачастую хорошо проработаны и детализированы [9]. С другой стороны, большинство рекомендаций доступны для любых категорий пользователей — от экспертов-программистов до начинающих веб-мастеров. Кроме технических спецификаций, Консорциум также публикует много руководств и советов, облегчающих внедрение рекомендаций. Для удобства пользователей консорциумом созданы специальные программы-валидаторы (англ. Online Validation Service), которые доступны по Сети и быстро могут проверить документы на соответствие популярным рекомендациям W3C. Консорциумом также созданы многие другие утилиты для облегчения работы веб-мастеров и программистов. Большинство утилит — это свободные программы, все они бесплатные.

Консорциум создан в 1994 году на основании соглашения между Массачусетским технологическим институтом США, Европейским консорциумом по исследованиям в области математики и информатики (Франция) и университетом Кейо (Япония). На 13 августа 2020 включает 434 действующих члена организации. Официальный сайт w3.org [9].

Консорциум World Wide Web (W3C) - это международное сообщество, в котором организации-члены, штатные сотрудники и общественность могут работать вместе над разработкой веб-стандартов. Миссия W3C, возглавляемая изобретателем и директором Интернета Тимом Бернерсом-Ли и генеральным директором Джеффри Джаффе, состоит в том, чтобы «раскрыть весь потенциал Интернета».

Стандарты W3C определяют открытую платформу для разработки приложений, позволяя реализовывать в них различные интерактивные возможности на основе обширных хранилищ данных, доступных на любом устройстве. Хотя границы платформы продолжают развиваться, лидеры отрасли почти в унисон говорят о том, что HTML5 станет краеугольным камнем этой платформы. Но платформа опирается на многие другие технологии, которые создают W3C и ее партнеры: веб разработка (CSS, SVG, Ajax, SMIL), архитектура веб (HTTP, URI), стек семантической паутины (RDF, SPARQL, OWL, SKOS), XML технологии (XSLT, DOM, XML Query, XPath) и различные виды API и веб-сервисов (SOAP, HTTP 1, 2, 3).

Одним из ключевых достижений консорциума является стандартизация языка гипертекстовой разметки HTML.

Цель консорциума - дать возможность компьютерным программам взаимодействовать в сети (сетевая интероперабельность) с этой целью консорциум взаимодействует с организацией Unicode, под интероперабельностью в стандарте ISO/IEC 24765-Systems and Software Engineering Vocabulary понимается способность двух или более систем или элементов к обмену информацией и к использованию информации полученной в результате обмена. При этом нет необходимости знать текущие рабочие характеристики задей-

ствованных функциональных устройств. Данное определение можно найти в ГОСТ Р 58539-2019 [10]. Другой целью является обеспечение полной «интернационализации Сети» и доступности для людей с ограниченными возможностями.

16 февраля 2012 W3C совместно с НИУ ВШЭ открыли представительство консорциума в России.

1.2 IANA (Internet Assigned Numbers Authority)

Еще одна важная организация в Интернет сфере это организация по стандартизации IANA [11], которая наблюдает за глобальным распределением IP-адресов, распределением номеров автономных систем, управлением корневой зоной в системе доменных имен (DNS) [12], медиа-типами (MIME) [13] и другими символьными обозначениями связанными с Интернет протоколами и Интернет-номерами. В настоящее время это функция принадлежит ICANN, некоммерческой частной американской корпорации, созданной с этой целью в 1998 году в соответствии с контрактом с Министерством торговли США. До этого управление IANA осуществлял в основном Джон Постел в Институте информационных наук (ISI) Университета Южной Калифорнии (USC), расположенном в Марина-дель-Рей (Лос-Анджелес), в соответствии с контрактом, который USC / ISI заключил с Департаментом обороны США. После перехода ICANN к глобальной модели управления с участием многих заинтересованных сторон функции IANA были переданы в Public Technical Identifiers, аффилированному лицу ICANN [14].

Кроме того, пять региональных интернет-реестров (RIR) делегируют номерные ресурсы своим клиентам, местным интернет-реестрам, поставщикам интернет-услуг и организациям конечных пользователей. Локальный интернет-реестр - это организация, которая назначает часть своего распределения из регионального интернет-реестра другим клиентам. Большинство местных Интернет-реестров также являются поставщиками Интернет-услуг.

Африканский сетевой информационный центр (AFRINIC) обслуживает Африку. Американский реестр интернет-номеров (ARIN) обслуживает Антарктиду, Канаду, часть Карибского бассейна и США. Сетевой информационный центр Азиатско-Тихоокеанского региона (APNIC) обслуживает Восточную Азию, Океанию, Южную Азию и Юго-Восточную Азию. Сетевой информационный центр Латинской Америки и Карибского бассейна (LACNIC) обслуживает большую часть Карибского бассейна и всю Латинскую Америку. Координационный центр сети Réseaux IP Européens (RIPE NCC) обслуживает Европу, Центральную Азию, Россию и Западную Азию.

1 октября 2016 года официально истёк срок действия договора о выполнении функций администрации адресного пространства интернета (IANA) между ICANN и Национальным управлением по телекоммуникациям и информации (NTIA) Министерства торговли США. При этом координирующая роль в исполнении функций IANA перешла в руки международного интернет-сообщества в связи с завершением срока действия договора с правительством США

С момента внедрения системы CIDR IANA делегировала свои полномочия по распределению IP-адресов региональным регистраторам в виде диапазонов класса A («/8»). Региональные регистраторы, в свою очередь, делегировали более мелкие диапазоны интернет-провайдерам.

Коллективно RIR создали Организацию номерных ресурсов, сформированную как орган для представления их коллективных интересов и обеспечения глобальной координации.

Поскольку адресное пространство версии 4 Интернет-протокола исчерпано, IANA не выделяет дополнительное адресное пространство IPv4, но выделяет /23 - /12 префиксных блоков из блока 2000::/3 IPv6 для запросов региональных реестров по мере необходимости.

Как уже было сказано IANA управляет также корневыми серверами доменных имен, образующими вершину иерархического дерева системы DNS, что включает в себя взаимодействие с доменами верхнего уровня «Ре-

гистраторами записи», операторами корневых серверов имен в соответствии с политикой ICANN.

Поскольку корневая зона криптографически защищена цифровой подписью в 2010 году, IANA также отвечает за жизненно важные части управления ключами для операций DNSSEC (в частности, это «Оператор KSK корневой зоны»), что включает в себя регулярное проведение реальных физических встреч, на которых члены группы доверенных представителей сообщества (TCR) в заранее определенном месте проходят процедуры по созданию ключей. TCR не могут быть связаны с ICANN, PTI (филиал ICANN) или Verisign из-за операционных ролей этих организаций в управлении ключами, а выбираются из более широкого сообщества DNS. Прошлые и нынешние TCR включают Винтона Серфа, Дэна Камински, Дмитрия Буркова, Анн-Мари Эклунд Лёвиндер и Джона Каррана. Очевидно, что создание DNSSEC было обусловлено тем, что сам по себе протокол DNS никак не защищен, потому легко можно было подделать ответ на запрос DNS. (А там сами фантазируйте к чему это может привести).

IANA управляет внутренним реестром международных договорных организаций, зоной arpa, службой обратного DNS (PTR запрос, получение по IP адресу доменного имени), и другими критическими зонами, такими как корневые серверы [4].

1.3 Модель взаимодействия открытых систем

Хотя набор протоколов Интернета стал стандартом для сетей и прагматичный подход TCP/IP к компьютерным сетям и независимой реализации упрощенных протоколов сделал его практической методологией [8], тем не менее, стоит рассмотреть и модель OSI [15]. Кроме того, некоторые протоколы и спецификации в стеке OSI остаются в использовании, одним из примеров является IS-IS, который был определен для OSI как ISO / IEC 10589: 2002 и адаптирован для использования в Интернете с TCP/IP как RFC 1142 [16]. Протоколы стека OSI отличает сложность и неоднозначность спецификаций.

Эти свойства явились результатом общей политики разработчиков стека, стремившихся учесть в своих протоколах все многообразие уже существующих и появляющихся технологий.

Идея модели OSI, как и многих других способов разбиения взаимодействия на уровни в информационных системах это обеспечить независимость прикладных программ от физических устройств (но не только). Многим наверняка уже знакомо понятие драйвера в операционных системах или модели представления данных в базах данных, разбивающих представление на логический и физический уровни. Давайте представим, что Вы хотите передать письмо на русском языке своему другу в Канаде. Первая идея и, очевидно, не самая удачная, это когда при пересечении границ, каждое почтовое отделение будет брать ваше письмо переводить его на основной язык той страны границу, которой это письмо пересекает, помещать на тот вид носителя или представлять его в той редакции, которая допустима для пересылки по данной стране. Ваш друг получит, конечно, письмо, но если ему хочется иметь письмо на русском, то и он вынужден будет его перевести. При этом каждая страна на границе должна будет иметь средство преобразования (переводчика). Другая очевидная идея - это брать письмо, помещать его вместе с остальной корреспонденцией в тот объект хранения, который легко перевозится по этой стране, затем извлекать на границе это письмо и помещать в другой объект хранения другой страны. В целом вторым способом и устроено все взаимодействие по модели OSI, с той лишь разницей, что здесь гораздо больше уровней вложений.

Каждый нижестоящий уровень предоставляет через унифицированный интерфейс доступ к своим возможностям, например, на сетевом уровне можно определить функцию send_packet(desitation_adrress, source_address, data).

Итак, в модели OSI присутствует семь уровней: прикладной, представления, сеансовый, транспортный, сетевой, канальный, физический. Обычно рассмотрение начинается с вышестоящего прикладного уровня, отвечающего

за протоколы предоставления определенных сетевых сервисов конечному пользователю. Рассмотрим сетевые сервисы прикладного уровня.

Сетевые сервисы можно разделить на сервисы удаленного хранения и предоставления информации, сервисы передачи сообщений, сервисы удаленного управления устройствами, сервисы приложений. Сервисы сетевого удаленного хранения - это файловые сервисы, обеспечивающие передачу файлов посредством таких протоколов как FTP, HTTP, распределенные системы хранения, обеспечивающие передачу файлов между пользователями, такие как torrent и p2p сети, различные сетевые диски. Сюда же можно отнести поисковые системы, помогающие найти нужную информацию, хотя, конечно, такие системы обеспечивают уже более сложные способы индексации, выборки данных, обеспечивая релевантность запросу предоставляемых данных. Кроме того, сюда можно отнести и различные СУБД, в соответствии с сервисами Кодда, реализующими и различные сетевые функции, такие как выполнение удаленных транзакций над СУБД и обеспечивающие распределенный доступ различных типов пользователей. К сервисам сообщений можно отнести сервис электронной почты, это протоколы SMTP, POP3, IMAP стека TCP/IP, X.400 стека протоколов OSI. Сервисы передачи мгновенных сообщений вроде SMS, различные диалоговые чаты. Кроме того, с повышением полосы пропускания линий связи уже прочно в обиход вошли видеочаты, онлайн передача звука и голоса. Конечно, все эти типы сервисов требуют различного уровня QoS (Quality of Service) [17] так как представляют собой различные виды трафика, но, тем не менее, их всех можно отнести к сервисам передачи сообщений. Сервисы удаленного управления устройствами включают в себя сервис печати, факс сервис, и так далее, несомненно, что некоторые устройства дороги или простаивают, переносить файлы для печати на флэшке или покупать в офисе каждому принтер дорогостоящее мероприятие и по времени и по деньгам, потому появление такого сетевого сервиса себя быстро оправдало. Но несомненно, что удаленно управлять можно не только принтером, когда реализуется диспетчеризация и постановка поступающих

заданий на устройство, но и другими приборами и устройствами. Для таких устройств, конечно, требуются свои операционные системы реального времени и надежные каналы связи и протоколы передачи команд и данных. Сервисы приложений дают возможность запускать и исполнять приложения, для этого мы рассмотрим далее SaaS, PaaS, IaaS системы.

На уровне представления решаются задачи, связанные с представлением информации, кодировкой и форматом данных, например, кодировка ASCII, UTF-8, КОИ-8. В стеке протоколов OSI это протокол ASN.1, сейчас описание форматов данных ASN.1 реализовано с помощью стандартов X.680 и X.690 (BER – базовые правила кодирования). В X.690 [18] была определена общая структура блока закодированной информации, состоящая из следующих 3 частей: Идентификатор – один или несколько октетов, в которых содержится информация о типе закодированных данных; Часть, содержащая информацию о длине блока – один или несколько октетов, в которых содержится информация о длине закодированных данных; Часть, содержащая закодированную информацию. В идентификаторе, например, указывается класс (универсальный, прикладной, контекстно-зависимый, частный), тип данных (простой, составной) и тэг (BOOLEAN, INTEGER, BIT STRING, REAL и т.д.). Очевидно, таким образом, можно описывать различные структуры данных. Кроме того, реализуются задачи по защите данных от внешнего доступа, это безопасная передача паролей и данных, включая технологии симметричного и ассиметричного шифрования, например, HTTPS, SSL, TLS, технологии создания цифровых подписей, реализуемые с помощью процедур хэширования.

Следующий уровень сеансовый, отвечающий за синхронизацию удаленного взаимодействия, завершение и начало сеанса передачи данных. Позволяет определять и передавать права на передачу данных, поддерживать сеанс в процессе неактивности приложений с помощью, например, отправки сообщений «heartbeat», добавлять контрольные точки, с которых можно восстановить и возобновить сеанс взаимодействия при потере или нарушении

соединения. Например, TCP протокол кроме функций транспортного уровня обеспечивает и сеансовый. В стеке OSI есть протокол X.255, обеспечивающий дуплексную и полудуплексную передачу данных и контрольные точки. Можно описать, например, сеансовый уровень примером по получению управления с помощью токенов. Допустим, вы прошли аутентификацию, и затем чтобы повторно ее не проходить вы высылаете уже полученный вами токен.

Транспортный уровень отвечает за доставку данных в том виде, в котором требует протокол, например, если нужно передать поток данных, то транспортный уровень может нарезать поток на фрагменты, пронумеровав их и отправив. Получатель соберет эти фрагменты в нужном порядке и передаст на следующий уровень в виде потока. Либо получив данные, из одного общего блока от нижестоящего сетевого уровня может передать их в виде фрагментов, если протокол транспортного уровня это предусматривает. Кроме того, может обеспечивать мультиплексирование, передачу нескольких потоков в рамках одного соединения, а также мультихоуминг, объединив несколько физических соединений для передачи одного потока данных. Кроме того, транспортный уровень может реализовывать как потоковую, так и дейтаграммную передачу, обеспечивая, либо не обеспечивая гарантированную доставку данных. Например, протокол TCP обеспечивает дуплексную надежную доставку потока данных, UDP обеспечивает дейтаграммную доставку без подтверждения доставки, SCTP обеспечивает мультиплексирование, мультихоуминг, надежную доставку. Кроме того, TCP и SCTP еще и обеспечивают функции сеансового уровня (хотя на самом деле функции сервисов с установлением соединения могут быть и на сетевом и канальном и на траспортном уровнях), так как стек протоколов TCP/IP соответствует модели DoD, которая не делит взаимодействие на семь уровней. UDP Lite обеспечивает возможность доставки с поврежденных дейтаграмм в отличие от UDP. Это бывает необходимо, например, при передаче звука, когда в звуковом потоке допустимы повреждения и шумы, или восстановление данных самим

приложением. Так же транспортный уровень рассматривает соединение как точка-точка, фактически он не рассматривает какие соединения, маршруты и структуры сетей проложены от получателя до передающего сообщение, можно считать, что для транспортного уровня соединение это прямая труба, по которой текут данные, просто они могут пропасть, задержаться или исказиться, но могут и успешно дойти до конца трубы, при этом в зависимости от этого и в зависимости от своих функций протокол транспортного уровня и будет определенным образом реагировать.

Сетевой уровень обеспечивает маршрутизацию и обычно пакетную коммутацию, за определение заторов и неполадок в сети, трансляцию логических адресов в физические и так далее. Думаю, из курса информатики вы помните, что кроме пакетной коммутации существует коммутация каналов и сообщений. Не будем подробно на этом останавливаться, потому что алгоритмы маршрутизации и коммутация рассматривается в курсе «Сети и телекоммуникации». Кратко же сеть обычно структурно представляет собой граф, где в вершинах находятся устройства маршрутизаторы (роутеры), а ребра представляют собой путь, который преодолевает пакет от одного маршрутизатора до другого, обычно этот путь (метрика) может быть представлен как величина обратная пропускной способности, время доставки или даже цена. Типичная решаемая задача здесь составить маршрут с минимальной ценой в том или ином смысле. Кроме того, существуют протоколы групповой доставки, обеспечивая тем самым возможности для онлайн вещания (цифровое телевидение или радио, видео стримы). Например, отличие стека OSI и стека TCP/IP здесь в том, что в первом каждый маршрутизатор имеет определенный адрес, фактически вершина графа, а во втором каждая сеть имеет адрес, фактически ребро графа. Обычно на сетевом уровне есть структурное деление адреса на сеть, и адрес узла в сети, либо сеть, подсеть и адрес узла.

На канальном уровне решается задача передачи кадров (фреймов) в рамках одного сегмента сети, при этом возможно решается задачи восстановления, подтверждения передачи. Здесь уже присутствует физическая ад-

ресация устройств, например, MAC или IMEI, такие устройства как коммутаторы, адаптеры, технологии Ethernet, Wi-Fi, FDDI и т.д.

На физическом уровне рассматривается передача сигналов, методы цифрового кодирования направленные на увеличение скорости и надежности передачи информации, различные среды передачи данных беспроводные и проводные (оптоволокно, витая пара, коаксиальный кабель), модуляция, манипуляция (амплитудная, частотная, фазовая), концентраторы, мультиплексирование (временное, частотное, кодовое).

Предполагается, что передача данных должна осуществляться путем инкапсулирования данных вышестоящего уровня в структуры данных нижестоящих уровней, при этом появляется понятие интерфейс, где каждый вышестоящий уровень через интерфейс получает доступ к сервисам нижестоящих уровней, и только так можно получить доступ к сети. Между одноименными уровнями на удаленных точках взаимодействие идет по протоколу, но через нижестоящие уровни, а не напрямую. Схему такого взаимодействия можно представить следующим рисунком (рисунок 1.1).

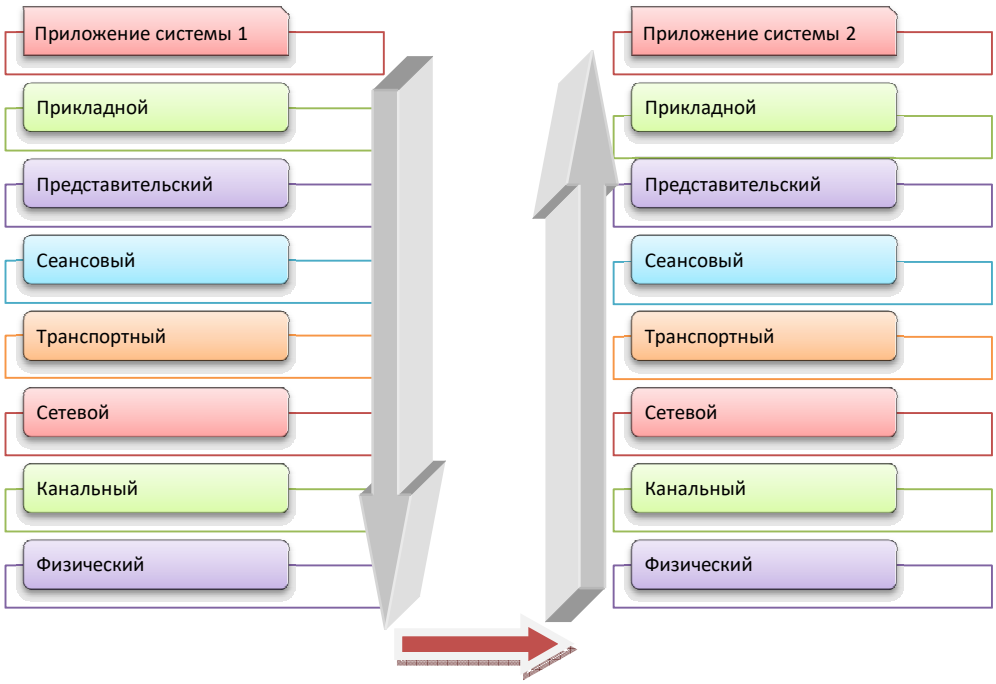


Рисунок 1.1 – Модель взаимодействия OSI

Такое разделение кроме того, что обеспечивает независимость приложений от физических устройств, обеспечивает и возможность развития каждого уровня независимо от другого. Появление новой технологии на физическом или канальном уровне практически не влияет на вышестоящий сетевой, кроме увеличения скорости и надежности передачи пакетов. Кроме того, специалисты, работающие на нижестоящих или вышестоящих уровнях, могут и не знать подробно об уровнях, с которыми они непосредственно не работают. Таким образом, можно обучать непосредственно специалистов для прикладных уровней и создания приложений не особо вникая в аспекты цифрового кодирования, независимо разрабатывать приложения и устройства. Хотя, конечно, некоторые знания физического и канального уровня могут быть полезны для создания специфических сетевых приложений.

1.3 Модель DoD (Department of Defense)

Данная модель разработана министерством обороны США и нашла свое отражение в распространенном ныне стеке протоколов TCP/IP [19]. В отличие от модели OSI она имеет четыре уровня: уровень приложений или прикладной уровень, транспортный уровень, межсетевой уровень, уровень сетевого доступа. Модель DoD представлена на рисунке 1.2.

Рисунок 1.2 – Модель DoD

Здесь нет такого жесткого разделения протоколов с отдельно сеансовыми или транспортными функциями, например, протокол TCP включает в себя процедуры установления и завершения соединения и поддержания и проверки соединения, а также чисто транспортные функции (хотя здесь можно установление сеанса считать как функцию транспортного уровня с установлением соединения). Но и протоколы, работающие поверх TCP, могут иметь сеансовые функции, в том числе и протоколы прикладного уровня, могут включать в себя как прикладные функции, так и представительские и сеансовые. Например, почти все значимые протоколы прикладного уровня текстовые и предполагают команды протоколов в кодировке ASCII. Например, SMTP, HTTP 1.x, POP3. Кстати это, например, породило некоторые пробле-

мы при передаче бинарных данных и файлов, требуя дополнительных форматов и кодировок, например, MIME и base64. Но, base64 увеличивает исходный размер файла на треть, что стало одной из причин перехода к бинароному HTTP 2 или использованию дополнительных соединений как в FTP.

В целом использование некоторых протоколов как обязательных для передачи протоколов прикладного уровня, например, HTTP 1 и 2 поверх TCP привело к появлению HTTP 3 поверх QUIC-UDP, в связи с так называемой проблемой HOL blocking. О ней мы поговорим в других главах.

Предполагается обычно, что на уровне сетевых интерфейсов происходит использование таких протоколов как Ethernet, Wi-Fi, FDDI. Условно можно считать, что в качестве блоков передаваемых данных на транспортном уровне используются сегменты, на межсетевом - пакеты, а на уровне сетевых интерфейсов используются кадры. При этом стек протоколов TCP/IP не описывает физический уровень, и фактически останавливается на том, что реализуются RFC документы, в которых описывается как вложить пакет в кадр данной технологии канального уровня. Хотя также, например, для IPv4 появляется необходимость в трансляции IP адреса в MAC адрес, что послужило появлению протоколов ARP и RARP. Кроме того, на транспортном уровне появляется дополнительное средство для уточнения приложения, которому поступают сегменты, это средство является дополнительным номером к IP адресу называемое портом. Сам же IPv4 адрес структурно представляет собой старшую и младшую часть четырех байтового числа, где старшая часть отвечает за номер сети, а младшая за номер узла в этой сети. Кроме того, выделяются специальные адреса, определяющие только локальные немаршрутизируемые адреса, широковещательные адреса и групповые адреса.

В целом можно упрощенно описать взаимодействие систем с использованием данной модели следующим образом: Приложение одного из удаленных компьютеров сети, реализуя какой-либо протокол прикладного взаимодействия, например, SMTP создает TCP соединение с удаленным серверным приложением. Очевидно, предварительно перед установлением соединения

приложение должно знать адрес сервера (это может быть и DNS и IP адрес) и порт приложения сервера. Если это DNS адрес система должна разрешить доменное имя используя протокол DNS (локальный DNS или указанный в качестве DNS сервера, например, 8.8.8.8) или локальный файл hosts, получив IP. Допустим, у нас уже есть IP адрес. В этом случае при установлении соединения формируется сегмент TCP для установления соединения со специальным битом SYN для инициализации процедуры тройного рукопожатия. В сегменте указывается также порт отправителя-клиента и порт назначения-сервера, порт отправителя нужен, чтобы получатель-сервер мог сформировать обратный сегмент, указав уже его в качестве порта получателя-клиента. Сегмент помещается в пакет, в котором указывается IP адрес отправителя и IP получателя. Не будем сейчас касаться особенностей формирования контрольных сумм или псевдозаголовков TCP. После того как сформирован пакет его нужно передать на уровень сетевых интерфейсов, фактически поместить в кадр, в кадре присутствует MAC адрес, с одной стороны необходимо отправить пакет на адрес указанного шлюза. Так как нам известен IP адрес шлюза, то необходимо получить и его MAC адрес (сетевого интерфейса) и отправить кадр, в кадре так же присутствует MAC адрес назначения и MAC адрес отправителя. После того как шлюз получает кадр, из него извлекается пакет, шлюз анализирует адрес назначения и используя таблицу маршрутизации отправляет пакет далее на тот интерфейс, который был указан в таблице маршрутизации, фактически следующему маршрутизатору. Естественно, между двумя маршрутизаторами может быть организована сеть на базе какого-либо протокола канального уровня, и пакет снова должен быть помещен в кадр этого протокола. После такого прохождения пакетом дистанции до получателя, из него извлекается сегмент на установление соединения. Далее сервером в ответ отправляется сегмент SYN ACK.

Мы здесь не рассмотрели, как строится таблица маршрутизации, что такое внутренние и внешние протоколы маршрутизации, что такое автономные системы, какие еще проблемы решаются при использовании протокола

TCP, не описали полную процедуру тройного рукопожатия и какая существует слабость данного способа установления соединения с точки зрения DDOS атак, каким образом настраивается оптимальное использование полосы пропускания и так далее. В большинстве случаев при разработке веб-сервисов нам это может и не понадобиться, все-таки деление модели на уровни успешно в основном решает задачу отделения приложения от механизмов сетевого взаимодействия. Но уже были упомянуты DDOS атаки, которые способны нарушить нормальное функционирование нашего приложения, и мы здесь уже видим пример, того что знание проблем и особенностей функционирования нижестоящих уровней может помочь нам сделать более успешное, безопасное и эффективное приложение.

В целом развитие стека протоколов TCP/IP происходит уже на протяжении десятков лет, классовую адресацию заменила бесклассовая (CIDR) [20], появилась шестая версия протокола, появлялись новые протоколы маршрутизации. Но именно данный стек протоколов тесно вошел в нашу жизнь.

Контрольные вопросы по главе 1

1. Какие уровни взаимодействия содержит модель OSI?
2. Функции каких уровней OSI обеспечивает протокол TCP?
3. Какие уровни предполагает модель DoD?
4. Может ли транспортный уровень обратиться к канальному минуя сетевой?
5. За что отвечает организация ICANN?
6. Что такое RFC?
7. Какие сервисы прикладного уровня можете назвать?

2 СЕТЕВЫЕ ПРИЛОЖЕНИЯ И СЕРВИСЫ

Существующие в настоящее время сетевые приложения и сервисы довольно разнообразны, они реализуют как интерфейсный GUI доступ к каким-либо вычислительным ресурсам и возможностям, так и интерфейсный доступ к функционалу посредством передачи данных с использованием различных форматов, таких как json или xml. Некоторые из ресурсов позволяют агрегировать огромные потоки данных и предоставляют свои ресурсы на не безвозмездной основе, какие-то ресурсы предлагают бесплатный доступ к онлайн интерпретаторам или вычислительным ресурсам вроде графических процессоров, на которых пользователь может реализовать обучение нейронных сетей или обработку данных, например, Google Colaboratory (colab.research.google.com) или Kaggle (www.kaggle.com). Существуют ресурсы по предоставлению возможностей размещения и разворачивания своих веб-приложений, так называемые PaaS и IaaS системы, примерами могут служить Amazon Web Services, Google App Engine, Heroku, Microsoft Azure, PythonAnyWhere. Некоторые позволяют пользоваться какими-то минимальными услугами забесплатно. Кроме того, среди разработчиков популярными являются сетевые сервисы контроля версий, GitHub, GitLab, а также системы непрерывной интеграции вроде Travis CI. Несомненно, что обилие такого количества сетевых приложений и сервисов было невозможно без развития стандартов сетевых технологий. Но рассмотрим область, связанную непосредственно с тематикой нашей дисциплины посвященной разработке веб-приложений и веб-сервисов и разберемся, что это такое. В том числе с точки зрения разработчика подобных приложений.

2.1 Веб-приложение.

Веб-приложение *представляет собой прикладное программное обеспечение, которое запускается на веб-сервере, в*

отличие от компьютерных программ, которые хранятся локально на усройстве пользователя и запускаются операционной системой (ОС).

. . . . . .

Пользователь получает доступ к веб-приложениям через веб-браузер с активным подключением к Интернету. Эти приложения программируются с использованием клиент-серверной архитектуры - пользователю («клиенту») предоставляются услуги через внешний сервер, размещенный на стороннем сервере.

. . . . . Пример . . . . . .

Примеры часто используемых веб-приложений включают в себя: электронную почту, розничные продажи в Интернете, онлайн-банкинг и онлайн-аукционы.

. . . . . .

Понятие динамической веб-страницы, «веб-приложения» и веб-сайта может быть размыто. Например, HTML5 [21] предоставляет явную языковую поддержку для создания приложений, которые загружаются как веб-страницы, но могут хранить данные локально и продолжать работать в автономном режиме.

Одностраничные сайты больше похожи на приложения, поскольку они отвергают обычную парадигму веб-сайта перемещения между отдельными страницами с разными URL-адресами (веб-сайт представляет собой набор взаимосвязанных документов веб-страниц, также это место расположения этих страниц). Одностраничные фреймворки могут использоваться для ускорения разработки веб-приложения для мобильной платформы.

2.1.1 Мобильные веб-приложения.

Несомненно, эти приложения сейчас наиболее часто используются на смартфонах и других планшетных устройствах. С точки зрения программи-

ста есть несколько концепций разрабатываемых веб-приложений, нацеленных на работу на мобильных устройствах: Адаптивный веб-дизайн, Прогрессивные веб-приложения (PWA), Нативные приложения, Гибридные приложения.

Веб-приложения с адаптивным веб-дизайном *[22] - это веб-приложения, которые одинаково отображаются на всех устройствах с различными геометрическими параметрами и разрешением.*

Адаптивный веб-дизайн можно использовать для создания обычных веб-сайтов или одностраничных приложений, просматриваемый на небольших экранах, которое хорошо работает с сенсорными экранами. При этом указанное приложение одинаково отображается на всевозможных устройствах с различными разрешениями и форматами. В этом случае нет необходимости в создании нескольких версий сайтов для каждого из устройств. При создании применяется гибкий макет на основе сетки, используются гибкие изображения (вообще в большинстве случаев стараются начать разработку сайта с адаптивной версии веб-сайта с небольшой страницей и одной колонкой). В качестве технологий можно рассмотреть Boostrap, использующий 12 колончатую сетку, плоский дизайн.

Прогрессивные веб-приложения (PWA) *[22] - это веб-приложения, которые загружаются как обычные веб-страницы или веб-сайты, но могут предлагать пользователю такие функции, как автономная работа и доступ к аппаратному обеспечению устройства, традиционно доступные только для собственных мобильных приложений.*

Фактически это технология, которая визуально и функционально трансформирует сайт в мобильное приложение в браузере и является гибридным решением между отдельным нативным мобильным приложением и загрузкой сайта с браузера, позволяя также отправку push-уведомлений, установку ярлыка на рабочий стол. Данная технология PWA была разработана Apple в 2007 году, но не набрав особой популярности была вытеснена нативными приложениям в AppStore. Но PWA набрало популярность в 2015 году благодаря расширению возможностей Google Chrome и продвижению Service Worker и Web App Manifest (предоставляет информацию о приложении в текстовом файле JSON, который необходим для того, чтобы веб-приложение было загружено и отображалось пользователю аналогично нативному приложению). Service Worker - это специальный Java Script файл, работающий в фоновом режиме, как кэширующий прокси между браузером и веб-сервером, фактически при запуске он начинает кэшировать контент, который в случае потери связи и будет запрошен, таким образом можно обеспечить независимую от потери связи работу приложения (например, транслировать кэшированную Музыку с сайта, а в некоторых случаях избежать недовольства пользователя, когда вместе с потерей связи отсутствует контент и работоспособность приложения).

В 2018 году произошли существенные изменения в области реализации web-стандартов. В обновлённой iOS 11.3 Apple добавила поддержку Service Worker в мобильную версию Safari и для macOS. В этом же году Microsoft добавила поддержку Service Worker в свой браузер Microsoft Edge. Кроме того, в Windows 10 появилась возможность распространения PWA через Microsoft Store. На апрель 2020 года офлайн-работу с PWA могут обеспечить iOS, Android, Windows, Linux, macOS и Chrome OS в браузерах Chrome, Safari, Firefox, Edge и Samsung Internet.

Для внедрения PWA можно воспользоваться каким-либо фреймворком, например, Ionic, Quasar или Vue Storefront. Vue Storefront — это бесплатный фреймворк для PWA интернет-магазина с открытым исходным кодом. Напи-

сан на Vue.js. Он довольно гибок и адаптивен, что делает его достаточно универсальным решением для интеграции с Pimcore/CoreShop, BigCommerce, PrestaShop, Shopware или, например, Magento через API.

Очевидны преимущества и недостатки такого подхода, с одной стороны универсальность, независимость от маркетов приложений, совмещение возможностей браузера и свойств нативного приложения, меньшие затраты на переработку дизайна, безопасность за счет использования HTTPS [23-24], с другой стороны, конечно, есть проблемы с обеспечением доступа к некоторым нативным возможностям устройств.

Нативные мобильные приложения *- это приложения, которые запускаются как обычные программы в операционной системе устройства, без использования браузера и обращаются к веб-серверу как клиенты при клиент-серверном взаимодействии.*

Нативные приложения или «мобильные приложения» запускаются непосредственно на мобильном устройстве, точно так же, как обычное программное обеспечение запускается непосредственно на настольном компьютере без веб-браузера (иногда без необходимости подключения к Интернету). Обычно они написаны на Java (для устройств Android), Objective-C или Swift (для устройств iOS). В последнее время фреймворки позволяют разрабатывать собственные приложения для всех платформ с использованием других языков (например, Kotlin). Очевидно, что преимущество нативных приложений в оптимизации для работы на конкретных устройствах, на которых они устанавливаются, но естественно такие приложения разрабатывать может быть сложнее, требует знания конкретных особенностей устройства, такое приложение может быть непереносимым уже на другое устройство, а разработка же с учетом кроссплатформенности может лишить пользователя дополнительных возможностей на его устройстве.

Гибридные приложения *- это приложения, которые встраивают непосредственно в себя веб-сайт.*

Гибридные приложения встраивают веб-сайт в собственное приложение. Это позволяет вести разработку с использованием веб-технологий, например, напрямую копируя код с существующего мобильного веб-сайта, но при этом сохраняя определенные преимущества нативных приложений (прямой доступ к оборудованию устройства, работу в автономном режиме, видимость магазина приложений). Платформы гибридных приложений включают Apache Cordova, Electron, Haxe, React Native и Xamarin.

2.1.2 Классические веб-приложения

Данные приложения начали развиваться раньше веб-приложений для мобильных устройств, в силу отсутствия последних и недостаточной их мощностью для того, чтобы запускать вообще какие-либо сложные приложения, мобильные устройства в основном выполняли роль телефона или мессенджера. Тем не менее, уже давно на то время существовали поисковые системы, форумы, чаты, доски объявлений, стали появляться социальные сети, как раз первые сайты реализовывали первые схемы клиент-серверного взаимодействия и разбиения разработки на две части, серверной и клиентской. Доступ к ресурсам реализовывался посредством браузеров и клиентской части веб-приложения. Появились понятия толстого и тонкого клиентов, понятие трехзвенной архитектуры (сервер с файловой базой данных, сервер СУБД, интернет сервер). Развивались технологии формирования динамического контента, которые набирали большую популярность даже в среде непрофессионалов. Именно в то время появились классические виды атак, от которых уже на стандартном уровне защищают современные фреймворки разработки приложений, например, sql-инъекции [25] и csrf-атаки [26], ве-

лись споры по поводу допустимости, безопасности и этичности использования cookie [27]. Некоторые из версий технологий формирования динамического контента, например, PHP подвергались атакам на переполнение буфера [28]. Почтовые протоколы передавали электронные письма в открытом виде, в том числе и пароли, кодируемые всего лишь в base64. При создании клиентских приложений приходилось проверять тип браузера, чтобы получить доступ к необходимому функционалу. Любой пользователь мог легко создать спам-бота, который легко обходил защиту подобных сайтов и вместо пользователей размещал сообщения. В настоящее время создание приложений подобного рода стало довольно обкатанным делом и реализуется фактически по готовым шаблонам, но, чтобы этого достичь индустрия прошла довольно длинный путь.

2.1.3 Преимущество веб–приложений и их история.

В ранних вычислительных моделях, таких как клиент-сервер, вычислительная нагрузка приложения распределялась между кодом на сервере и кодом локально установленного «десктопного» приложения на клиенте. Таким образом, приложение состояло из программы сервера и предварительно скомпилированной клиентской программы с пользовательским интерфейсом, которая отдельно устанавливалась на персональном компьютере каждого пользователя. В этом случае обновление серверного кода приложения также требовало обновления клиентского кода, установленного на каждой пользовательской рабочей станции, что увеличивало стоимость поддержки и снижало производительность. Кроме того, как клиентская, так и серверная программные компоненты приложения обычно были жестко привязаны к определенной компьютерной архитектуре и операционной системе, и перенос их на другую платформу часто было весьма дорогостоящей процедурой (хотя уже упоминалось, что в настоящее время и разработка нативных приложений для мобильных устройств обременена теми же проблемами).

С другой стороны веб-приложения используют веб-документы стандартного формата, такие как HTML и JavaScript, которые поддерживаются различными веб-браузерами. Веб-приложения можно рассматривать как особый вариант клиент-серверного программного обеспечения, в котором клиентское программное обеспечение загружается на клиентский компьютер при посещении соответствующей веб-страницы с использованием стандартных протоколов, таких как HTTP. Обновление клиентского веб-программного обеспечения может происходить при посещении веб-страницы. Во время сеанса веб-браузер выполняет действия, интерпретирует и отображает страницы как универсальный клиент для любого веб-приложения.

На заре развития Интернет каждая отдельная веб-страница доставлялась клиенту как статический документ, но возможность перемещения по последовательности страниц через ссылки, возможность отправки данных через элементы веб-формы встроенных в разметку страницы уже обеспечивали интерактивное взаимодействие. Однако какие-либо существенные изменения на веб-странице требовали обратного обращения к серверу для ее обновления.

В 1995 году Netscape представила клиентский язык сценариев под названием JavaScript [29], позволяющий программистам добавлять некоторые динамические элементы в пользовательский интерфейс, работающий на стороне клиента. Таким образом, вместо отправки данных на сервер для создания всей веб-страницы встроенные сценарии загруженной страницы могли выполнять различные задачи, такие как проверка ввода или отображение / скрытие частей страницы.

В 1996 году Macromedia представила Flash - проигрыватель векторной анимации, который можно было установить в браузере в качестве подключаемого модуля для встраивания анимации непосредственно в веб-страницу, что позволило использовать язык сценариев для программирования интерфейса на стороне клиента без необходимости связываться с сервером.

В 1999 году была представлена концепция «веб-приложения» на языке Java в спецификации сервлетов версии 2.2. К этому времени были разработа-

ны и JavaScript, и XML, объект XMLHttpRequest был представлен в Internet Explorer 5 как объект ActiveX [30-31]. Работа над согласованным стандартом асинхронной загрузки данных продолжалась и позднее -– версия 1.0 спецификации XMLHttpRequest Консорциума W3C была принята только в 2009 г. Разработка удачного веб-приложения в то время с обеспечением интерактивности порой требовала проверки производителя браузера, с которым работает пользователь для доступа к соответствующим объектам, обеспечивающим интерактивность на странице.

В 2005 году был впервые использован термин AJAX, объединяющий подходы для создания интерактивных пользовательских интерфейсов с фоновым обменом данными между браузером и сервером, без необходимости полной загрузки страницы, и такие сервисы как Gmail, Google Maps, начали делать свои клиентские части все более интерактивными.

В 2011 году была завершена разработка HTML5, обеспечивающего графические и мультимедийные возможности без необходимости использования плагинов на стороне клиента, первый рабочий черновик появился еще в 2008 году [32]. HTML5 также расширил семантические элементы документов (например, тэг <time>). API-интерфейсы и объектная модель документа (DOM) больше не являются второстепенными, а являются фундаментальными частями спецификации HTML5. WebGL API открыл возможности для реализации продвинутой трехмерной графики, основанной на Canvas HTML5 и языке JavaScript. Все это дало куда большие возможности для создания полнофункциональных веб-приложений независимых от платформы и браузера.

Благодаря Java, JavaScript, DHTML, Flash, Silverlight и другим технологиям появились специфические для приложений методы, такие как рисование на экране, воспроизведение звука и доступ к клавиатуре и мыши.

С ростом производтельности мобильных устройств появилось желание перенести часть вычислительных затрат на сторону клиента, появились такие JS (Java Script) фреймворки и библиотеки как Angular, Vue и React, исполь-

зующие концецию Virual DOM (Document Object Model), о различных DOM поговорим позже. Есть, конечно, и другие менее распространенные и менее популярные решения. Теперь «прорисовку» интерфейса пользователя можно реализовать на стороне клиента. Рост вычислительной мощности на стороне клиента дает возможность перенести часть задач с сервера, в частности, например, обрабатывать некоторые запросы. Например, не обрабатывать запросы по сортивроке SQL с операцией ORDER BY на стороне сервера, а выполнять только выборку данных, сортируя эти данные на стороне клиента.

2.1.4 Структура типичного веб приложения.

Приложения обычно разбиваются на логические блоки, называемые «уровнями», где каждому уровню назначается какая-либо роль. Традиционные приложения состоят только из одного уровня, который находится на клиентской машине, но веб-приложения по своей природе допускают n-уровневый подход. Хотя возможно множество вариантов, наиболее распространенной структурой является трехуровневое приложение. В наиболее распространенной форме эти три уровня называются презентацией (представлением), приложением и хранилищем (presentation, application, storage). На первом уровне веб-браузер, на втором уровне движок, использующий некоторые технологии динамического веб-контента (например, ASP, CGI, ColdFusion, Dart, JSP / Java, Node.js, PHP, Python или Ruby on Rails), а на третьем уровне база данных. Не стоит, наверное, это путать с трехзвенной архитектурой БД, где не первом уровне интернет сервер, на втором сервер СУБД, а на третьем файловая БД. Веб-браузер отправляет запросы на средний уровень, который их обслуживает, выполняя запросы и изменения в базе данных, а также создает пользовательский интерфейс, обычно динамически.

Для более сложных приложений трехуровневое разбиение может оказаться недостаточным, и может быть выгодно использовать n-уровневый подход, где наибольшее преимущество даст разбиение второго уровня бизнес-логики на более мелкие подуровни. Еще одним решением может быть

добавление уровня интеграции, который отделяет уровень данных от остальных уровней, предоставляя простой интерфейс для доступа к данным. Например, доступ к данным клиента можно получить путем вызова функции «list_clients ()» вместо выполнения запроса SQL непосредственно к таблице клиента в базе данных. Это позволяет заменять базовую БД без каких-либо изменений на других уровнях. В целом концепцию и смысл подобных разбиений на уровни мы уже разбирали.

Часто архитектура веб-приложения рассматривается как двухуровневая. Предполагается что есть «толстый» клиент, который выполняет всю работу и запрашивает простой сервер, или «тонкий» клиент, полагающийся на сложный сервер. Клиент будет работать на уровне представления, сервер будет иметь базу данных и работать на уровне хранения, а бизнес-логика (уровень приложения) будет находиться на одном из них или на обоих. Хоть это и улучшает масштабируемость приложений и разделяет отображение и базу данных, это по-прежнему не позволяет специализировать уровни более точно, поэтому большинство приложений переросли эту модель.

2.1.5 Типичная современная структура веб-приложения

Архитектура современного веб-приложения может быть достаточно сложной, особенно если оно является высоконагруженным приложением. Рассмотрим некоторые аспекты, связанные с таким приложением, более подробно эти аспекты будут раскрыты в последующих главах. Большинство современных архитектур веб-сервисов основаны на архитектуре REST [33] и разворачиваются как использованием PaaS, так и IaaS систем [34]. В целом в связи с большой нагруженностью запросами клиентов, при разработке таких приложений используют различные технологии для обеспечения свойств масштабируемости и расширяемости. Самих технологий так же большое количество и всех их рассмотреть получится только кратко, больший упор в нашем пособии будет сделан на интерпретируемый язык Python [35] и один из веб-фрейморков Flask [36]. Как и большинство технологий данная техно-

логия обладает как достоинствами, так и недостатками. Стоит отметить, что кроме указанного языка, существует естественно множество других языков, фреймворков и библиотек, использующихся для реализации так называемого «бэкенда» (PHP, Java JVM, ASP .Net C#, Go, Rust, Fast CGI C++, Node .js), на стороне клиента «фронте» более прочно закрепился язык Java Script, но так же используются другие Type Script, Coffee Script, ActionScript, Kotlin (для мобильных приложений) делаются попытки перейти на Dart, используются языки типа Haxe, которые можно транслировать на Java Script, кроме того, планируется использование технологии WebAssembly (wasm), языка низкого уровня, который будет использоваться вместо JavaScript и не требует как java script трудозатратной трансляции с высокоуровневого языка. По состоянию на июль 2021 год 94 % установленных браузеров поддерживают WebAssembly. Стековая виртуальная машина, исполняющая инструкции бинарного формата wasm, может быть запущена как в среде браузера, так и в серверной среде, WebAssembly ориентирована на Си, Си++. Таким образом, становится возможным писать «фронт» на си подобных языках Си, Си++, Go, Rust, Java, C#, TypeScript. Интерпретируемые языки, типа Python здесь сложнее использовать из-за их высокой динамичности, но если постараться, то можно, правда это приведет к большому файлу позволяющему запускать соответствующее приложение, самый оптимальный вариант на сегодняшний момент даст размер от 200 до 5000 кб для загрузки соответствующего js или wasm файла. Кроме того, для языков со сборщиком мусора (garbage collector) так же потребуется скачивание дополнительных данных реализующих соответствующую процедуру, несмотря на наличие сборщика мусора у самого браузера. Хотя при скачивании мобильного приложения или кэшировании это может не играть значительной роли. Кроме того, Mozilla запустила проект Pyodide, для трансляции научного стэка Python разработки в WebAssembly. В целом на стороне клиента лидером остается Java Script.

Как уже было сказано большинство современных веб-приложений используют многоуровневую архитектуру, но наиболее ее современный вид

можно представить рисунком 2.1. Очевидно, что данная архитектура представляет собой реализацию для довольно-таки сложного сервиса, в большинстве случаев можно было бы обойтись веб-сервером, субд и кэширующим сервисом, либо обычной трехзвенной архитектурой. Разберем кратко все эти системы, представленные на рисунке 2.1 и для чего они нужны.

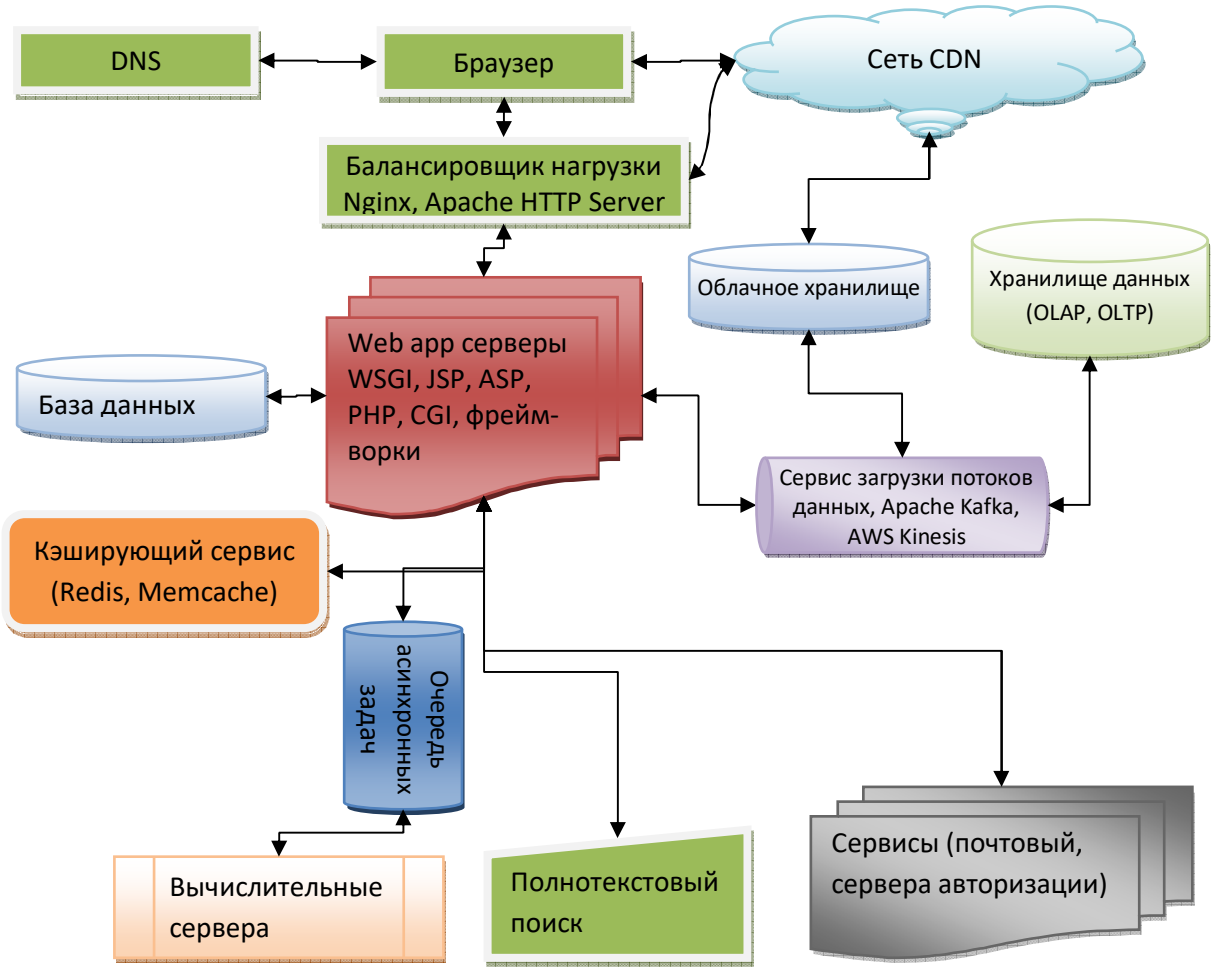


Рисунок 2.1 – Архитектура современных веб сервисов.

Очевидно, что в настоящее время для доступа к веб сервису или сайту используется протокол HTTP, либо протокол HTTPS, второй реализует функцию защиты соединения между сервером и клиентом, кроме того позволяет пользовательскому браузеру дать возможность проверить легальность данного сайта с помощью сертификата выданного доверенным Центром Авторизации. Пример формирования сертификата и общие принципы защиты с

помощью сертификатов мы разберем далее в данном учебном пособии. Пока же достаточно того, что пользовательский браузер, получая строку запроса ресурса URL, должен получить из URL доменное имя и далее с помощью запроса DNS соответствующий доменному имени IP адрес запрошенного сайта, с помощью запроса A (для получения почтового сервера используется запрос MX). Далее браузер осуществляет TCP соединение по полученному IP адресу и шлет известные запросы HTTP: GET, POST, PUT, PATCH, DELETE и т.д. Либо реализуется процедура защищенного соединения, где сервер предоставляет свой сертификат, который браузер проверяет с помощью сертификата центра авторизации. Предварительно сертификат сервера уже должен быть подписан сертификатом центра авторизации и браузер доверяет этому центру. Далее после обмена ключами шифрования между браузером и сервером реализуются те же HTTP запросы, но по уже защищенному соединению. В принципе при реализации данного взамодействия уже достаточно наличия непосредственно web-сервера, который выдаст нам запрошенный ресурс. Допустим, мы реализовали простейший сайт со статичным наполнением и базовой страницей index.html, и эту страницу и будет предоставлять по запросу HTTP веб сервер. Современные веб-сервисы и приложения более сложные, они как уже говорилось, позволяют динамически формировать страницы и возвращаемый контент. Основными технологиями для этого выступают такие как PHP, JSP, ASP .Net, CGI, WSGI, FastCGI. Более того в настоящее время изначально предназначенный для создания интерактивного наполнения на стороне клиента скриптовый язык javascript также получил технологию для создания так называемого back enda с помощью node.js. Также нагрузки на современные сайты гораздо выше, чем это было десятки лет назад, потому обойтись одним веб-сервером приложения уже не получится. Потому добавляют дополнительную надстройку, часто выполняющую роль балансировщика нагрузки и перераспределяющего запросы клиентов на разные дочерние процессы веб-приложений. Кроме того, такой сервер часто играет роль кэширующего прокси сервера, сервера обеспечивающего защи-

щенное HTTPS соединения, и веб-сервера с поддержкой технологий разработки веб-приложений на различных языках, например, примерами таких серверов является nginx и Apache HTTP-сервер. Приведем краткий описательный пример разработки веб приложения с использованием языка Python и фреймворка Flask. Данный фреймворк может запустить приложение Flask, которое будет обслуживать поступающие запросы от пользователей, очевидно для этого запускается серверный TCP соккет, который слушает входящие клиентские соединения и затем обслуживает их в различных потоках, допустим, синхронно, то есть после получения запроса get операции передачи контента завершает соединение. Фактически обслуживающая функция get будет запущена в отдельном потоке и вернет пользователю результат, если бы у нас было бы однопоточное приложение, то пользователь ждал бы пока обработается функция get для предыдущего клиента. Если к нашему приложению обращается немного клиентов, этого было бы достаточно. Но, обычно, как уже было сказано клиентов у веб-сервиса может быть очень много и соответственна сервер будет поступать множество запросов, которые он не успеет обработать.

При этом в рассматриваемом нами примере используются технологии, обладающие некоторыми недостатками. Во-первых, интерпретатор Python не распределяет потоки по разным ядрам процессора (см. GIL, global interpreter lock), что замедляет работу нашего приложения, а во-вторых наше приложение всякий раз генерирует только динамический контент и каждый раз необходимо обращаться к формированию страницы, в фреймворке это, например, делают шаблоны. Кроме того, к нашему приложению может обратиться какой-то очень медленный клиент, скачивающий по TCP соединению что-то очень долго и на этот поток веб-сервер будет вынужден тратить постоянно ресурсы, что может быть чревато проблемами задержки, система вынуждена переключаться на обработку этого потока, при этом еще и выполняя функции приложения, те функции за которые оно и отвечает. Поэтому перед веб-сервером ставится дополнительный балансировщик нагрузки, например

nginx. Он реализует кэширование статического контента, может кэшировать частично страницу выделяя динамический контент на странице с помощью вставок SSI, кроме того, способен использовать несколько запущенных приложений Python, реализуя затем балансировку нагрузки на процессы. При этом, выполнив быстро запрос к веб приложению и закэшировав его, сам будет отдавать клиенту данные, в то время как сервер приложения будет свободен и может выполнять другие задачи, в идеале, конечно, сервера должны быть разнесены.

Как вы поняли Python имеет недостаток с распоточиванием по ядрам. Но Python сообществом была разработана технология WSGI и различные сервера по типу gunicorn для того, чтобы обеспечить взаимодействие веб сервера и веб приложения, в том числе gunicorn позволяет запустить несколько отдельных процессов приложения. Gunicorn реализует запуск worker-ов которые обеспечивают обслуживание асинхронных и синхронных запросов в нескольких процессах – worker-ах, которые тоже являются либо синхронными, либо асинхронными. Но опять же, они предназначены для динамического контента. Разработчики веб-сервисов на Python используют gunicorn с целью запуска нескольких worker-ов выбирая их количество исходя из количества ядер процессора (по формуле количество ядер * 2+1). А вот nginx может обеспечить перенаправление запросов на несколько серверов, где запущен gunicorn или другое веб-приложение и также настроить считывание и кэширование статического контента. Такие системы взаимодействия можно реализовать, используя docker контейнеры и docker-compose, либо непосредственно на операционных системах семейства linux. Это, один из многочисленных примеров, которые существуют в мире реализации веб-серверов и приведен с целью понимания происходящих процессов.

Так же на представленной схеме (рисунок 2.1) присутствует кэширующий сервис между СУБД и веб-приложением, позволяя кэшировать часто использующие данные, например, списки пользователей или данные по запросам содержащих множественные экземпляры записей, в общем случае та-

кой кэш нужен, чтобы обеспечить быстрый доступ к данным которые хранятся сейчас на сервере СУБД.

Наряду с этим иногда присутствуют тяжеловесные задачи, решение которых может проходить независимо от веб-приложения, но веб-сервер не может тратить на это свои ресурсы, потому такую задачу проще поставить в очередь задач на какой-то вычислительный ресурс и когда задача будет решена, предоставить результат пользователю, например, распознавание класса объекта с помощью нейронной сети. Очередь задач может быть реализована с использованием RabbitMQ, хотя брокеров сообщений очень много.

Кроме того, приложение может собирать или передавать большие потоки данных, которые должны храниться так, чтобы можно было получить доступ к этим данным с использованием современных средств обработки больших данных, с этой целью используют, например, Apache Kafka, AWS Kenesis.

Все эти данные, например, видео могут доставляться непосредственному пользователю через сети доставки контента CDN (Content Delivery Network), обычно применяющие механизмы кэширования или хранения данных вблизи непосредственной близости расположения потребителя контента.

Так же используются различные сетевые сервисы, это и почтовые сервисы, предоставляющие возможность подтверждения регистрации и сервисы, предоставляющие проверку, того что вы человек на основе капчи (captcha), сервисы, позволяющие провести авторизацию на базе социальных сетей и прочих сервисов (OAuth). Существующие технологии чуть более подробно будут рассмотрены далее.

2.1.6 Монолитное веб-приложение

Приложение построено как нечто неделимое целое, вся логика обработки запросов помещена в единый процесс, который может быть предствлен конечно отдельными модулями, классами, но собирается весь проект в один условный исполнительный модуль. Отсюда и недостатки такого подхода,

связанные с тем, что разрастание проекта приводит к длительной сборке, сложности его сопровождения, сложностям с добавлением новых функциональных возможностей (модификации). Изменение какой-либо части проекта может приводить к изменению остальной части проекта из-за жестких зависимостей, отдельным группам программистов сложно работать с таким проектом. Вся бизнес логика реализуется фактически в одной монолитной части, в которой все сложнее разобраться. Балансировка нагрузки реализуется в основном за счет репликации, на логическом уровне присутствует единая база данных и такую систему сложно подвергать разделению по различным задачам в зависимости от бизнес логики, или по отдельным таблицам. С появлением облачных сервисов стали развиваться микросервисные архитектуры, которые представляли собой разделенные части приложения связанные собой через интерфейсы взаимодействия.

2.1.7 Микросервисная архитектура веб-приложения

Один из видов сервис ориентированных подходов (SOA) наряду с CORBA, web-сервисами, очередями сообщений, ESB (Enterprise Service Bus).

2.1.8 Одностраничное веб-приложение

Это один из паттернов проектирования пользовательского интерфейса, в том числе для микросервисной архитектуры.

2.1.9 Поставщик услуг приложений ASP (Application service providers)

В настоящее время веб-приложения набирают большую популярность. У компаний, производящих прикладное программное обеспечение появилась новая бизнес стратегия выхода на рынок, заключающаяся в предоставлении веб-доступа к программному обеспечению, которое ранее распространялось как «десктопное» приложение. При этом пользователи вносят ежемесячную или годовую плату за использование программного приложения без необходимости его установки на локальный жесткий диск. Компания, которая следует этой стратегии, известна как поставщик услуг приложений (ASP).

Нарушение безопасности в таких приложениях вызывают серьезную озабоченность, поскольку может затрагивать как корпоративную информацию, так и данные частных клиентов. Защита этих активов - важная часть любого веб-приложения, что требует включения в процесс разработки множества различных аспектов. Сюда входят процессы аутентификации, авторизации, обработки активов, ввода, регистрации и аудита. Встраивание безопасности в приложения с самого начала может быть более эффективным и менее разрушительным в долгосрочной перспективе.

Так широкое распространение получили сервисы облачных вычислений (SaaS, PaaS, IaaS).

Веб-приложения модели облачных вычислений представляют собой программное обеспечение как услугу (SaaS, Software As A Service). Существуют бизнес-приложения, предоставляемые как SaaS для предприятий за фиксированную плату или плату в зависимости от объема использования. Другие веб-приложения предлагаются бесплатно, часто принося доход от рекламы, отображаемой в интерфейсе веб-приложения.

Написание веб-приложений часто упрощается за счет использования фрейморков веб-приложений. Эти структуры облегчают быструю разработку приложений, позволяя группе разработчиков сосредоточиться на тех частях своего приложения, которые уникальны для их целей, без необходимости решать общие проблемы разработки, такие как управление пользователями. Многие из используемых фреймворков представляют собой программное обеспечение с открытым исходным кодом.

Непосредственно такие системы как PaaS предлагают наборы инструментов для разработки своего веб-приложения.

Использование фреймворков веб-приложений часто может уменьшить количество ошибок в программе, как за счет упрощения кода, так и за счет того, что одна группа программистов может сконцентрироваться на фреймворке, а другая - на конкретном варианте использования. Часто сетевые приложения подвергаются попыткам взлома, фреймворки обычно содержат

средства борьбы с типовыми атаками, например, CSRF токены. Фреймворки также могут способствовать использованию полезных нововведений, например, GET after POST или Post/Redirect/Get, способ борьбы с повторной отправкой данных формы при нажатии submit. При отправке данных формы, после получения результата, пользователь может нажать в браузере обновить, что вызовет повторную отправку данных и возможные нежелательные результаты, например, повторный перевод денег. Суть решения заключается в том, что в ответ на отправку данных веб-формы, сервер не просто генерирует HTML-страницу с результатом, а возвращает браузеру заголовок перенаправления «Location» (при этом используются коды состояния HTTP 302, HTTP 303, иногда HTTP 301), ведущий на страницу с результатом, но уже вызываемым обычным методом GET, а не POST, и без данных веб-формы.

2.2 Веб-сервис.

*Термин **веб-сервис** или **веб-службу** (WebService, WS) можно понимать, как услугу, предлагаемую одним электронным устройством другому электронному устройству, обменивающимся данными друг с другом через World Wide Web. Также можно понимать, как сервер, работающий на компьютерном устройстве, прослушивающий запросы к определенному порту по сети для передачи веб-документов (HTML, JSON, XML, изображения) и создающего службы веб-приложений, которые служат для решения конкретных проблем домена в Интернет (WWW, Интернет, HTTP).*

В веб-сервисе веб-технология, такая как протокол HTTP, используется для передачи машиночитаемых форматов файлов, таких как XML и JSON.

На практике веб-служба предоставляет объектно-ориентированный веб-интерфейс для сервера базы данных, использующийся, например, другим

веб-сервером или мобильным приложением, предоставляющим конечному пользователю пользовательский интерфейс. Многие организации, предоставляющие данные в виде HTML-страниц (например, в виде таблиц), предоставляют эти же данные в формате XML или JSON, часто через веб-службу, чтобы разрешить распространение, например, или экспорт данных. Сейчас такие службы имеют широкое распространение, предлагая различные научные данные, большие данные для обработки, данные относящиеся к деятельности государственных органов различных уровней, например, data.gov.ru. Приложение, работающее с конечным пользователем, может представлять собой гибрид, в котором веб-сервер использует несколько различных веб-сервисов в сети интернет и компилирует контент для пользователя в одном пользовательском интерфейсе.

Более строго термин «веб-сервис» описывает стандартизованный способ объединения веб-приложений с использованием открытых стандартов XML, SOAP, WSDL и UDDI через Интернет-протоколы.

Фактически веб-сервис - это программная функция, которая вызывается и предоставляет результат, функция вызывается по сетевому адресу через Интернет, при этом служба всегда включена, как в концепции распределенных вычислений.

В настоящее время наиболее популярными способами организации веб-сервисов является SOAP или архитектура REST, но REST становится более популярным.

2.2.1 XML-RPC

Протокол XML-RPC (XML Remote Procedure Call) был в первые опубликован в 1999 году. Все сообщение XML-RPC являются HTTP-POST запросами. Для кодирования сообщений используется XML. Параметрами процедуры могут быть скалярные значения, числа, строки, даты, массивы и структуры. Ответ веб-службы может хранить либо значение, возвращаемой проце-

дурой, либо код и сообщение ошибки. XML - это формат данных, используемый для хранения данных и предоставления метаданных вокруг них.

Пример запроса:

```
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
      </param>
    </params>
  </methodCall>
```

Пример ответа:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
      </param>
    </params>
  </methodResponse>
```

Недостатком может являться большой размер сообщений, в четыре раза больше чем обычный XML, отсутствие языка описания веб-сервиса по типу WSDL, который мог бы использоваться для генерации классов на стороне клиента.

2.2.2 SOAP

SOAP *— это стандартизированный протокол, который отправляет сообщения с использованием других протоколов, таких как HTTP и SMTP, является наследником XML-RPC и позволяет обеспечить доступ к удаленным функциям (процедурам).*

Спецификации SOAP являются официальными веб-стандартами, которые поддерживаются и разрабатываются Консорциумом World Wide Web (W3C World Wide Web Consortium).

SOAP используется для передачи данных, WSDL используется для описания доступных служб, а UDDI перечисляет доступные службы.

Разное сетевое программное обеспечение может быть реализовано с использованием различных языков программирования, и, следовательно, существует потребность в методе обмена данными, который не зависит от конкретного языка программирования. Таким образом, необходимы общие соглашения по представлению данных, например, XML.

Также необходимо определить правила связи между различными системами, например:

Каким образом, и в каком порядке одна система может запрашивать данные из другой системы?

Какие конкретные параметры необходимы в запросе данных?

Какой будет структура полученных данных? (Обычно данные обмениваются в файлах XML, и структура файла XML проверяется по файлу .xsd.)

Какие сообщения об ошибках представлять при несоблюдении правил передачи?

Все эти правила взаимодействия определены в файле WSDL (язык описания веб-служб), который имеет расширение .wsdl. WSDL был разработан совместно Microsoft и IBM, последняя опубликованная версия 2.0 датирована 2007 годом. Расшифровывается как язык описания веб-сервисов. Это стандартный формат для описания веб-службы. Клиентская программа, подключающаяся к веб-службе, может прочитать WSDL, чтобы определить, какие функции доступны на сервере. Все используемые специальные типы данных встраиваются в файл WSDL в форме XML-схемы. Затем клиент может использовать SOAP для фактического вызова одной из функций, перечисленных в WSDL.

Каталог под названием UDDI (универсальное описание, обнаружение и интеграция) определяет, с какой программной системой следует обращаться для какого типа данных. Поэтому, когда одной программной системе требуется один конкретный отчет / данные, она переходит к UDDI и выясняет, с какими другими системами она может связаться для получения этих данных. Как только программная система обнаруживает, с какими другими системами ей следует связаться, она затем связывается с этой системой, используя специальный протокол, называемый SOAP (Simple Object Access Protocol). Система поставщика услуг сначала проверит запрос данных, ссылаясь на файл WSDL, а затем обработает запрос и отправит данные по протоколу SOAP. UDDI была написана в августе 2000 года.

SOAP может использоваться с любым протоколом прикладного уровня: SMTP, FTP, HTTP, HTTPS и др. Однако его взаимодействие с каждым из этих протоколов имеет свои особенности, которые должны быть определены отдельно. Чаще всего SOAP используется поверх HTTP. Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур (RPC). Сейчас протокол используется для обмена произвольными сообщениями в формате XML, а не только для вызова процедур. Официальная спецификация последней версии 1.2 протокола никак не расшифровывает название SOAP. SOAP является расширением протокола XML-RPC. Использование SOAP для передачи сообщений увеличивает их объём и снижает скорость обработки. В системах, где скорость важна, чаще используется пересылка XML-документов через HTTP напрямую, где параметры запроса передаются как обычные HTTP-параметры. При использовании HTTP, поддерживается как метод GET, так и POST. GET допускается использовать только для получения данных, т.е. на стороне сервера не должно ничего меняться. POST можно использовать для всех случаев. На практике обычно используется только POST.

Все SOAP службы имеют описание на языке WSDL, которое тоже является XML-ом. Это позволяет клиентом автоматически сгенерировать прокси классы.

Примеры SOAP get и post

Пример SOAP GET.

```
GET /travelcompany.example.org/reservations?code=FT35ZBQ  HTTP/1.1
Host: travelcompany.example.org
Accept: text/html;q=0.5, application/soap+xml
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
         env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-30T16:25:00.000-05:00</m:dateAndTime>
  </m:reservation>
 </env:Header>
 <env:Body>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:x="http://travelcompany.example.org/vocab#"
    env:encodingStyle="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <x:ReservationRequest
 rdf:about="http://travelcompany.example.org/reservations?code=FT35ZBQ">
     <x:passenger>Åke Jógvan Øyvind</x:passenger>
     <x:outbound>
      <x:TravelRequest>
       <x:to>LAX</x:to>
       <x:from>LGA</x:from>
       <x:date>2001-12-14</x:date>
      </x:TravelRequest>
     </x:outbound>
     <x:return>
      <x:TravelRequest>
       <x:to>JFK</x:to>
       <x:from>LAX</x:from>
       <x:date>2001-12-20</x:date>
      </x:TravelRequest>
     </x:return>
    </x:ReservationRequest>
  </rdf:RDF>
 </env:Body>
</env:Envelope>
```

Пример SOAP POST.

```
POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn
```

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
  <t:transaction
        xmlns:t="http://thirdparty.example.org/transaction"
        env:encodingStyle="http://example.com/encoding"
        env:mustUnderstand="true" >5</t:transaction>
 </env:Header>
 <env:Body>
  <m:chargeReservation
    env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
       xmlns:m="http://travelcompany.example.org/">
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation">
   <m:code>FT35ZBQ</m:code>
  </m:reservation>
   <o:creditCard xmlns:o="http://mycompany.example.com/financial">
    <n:name xmlns:n="http://mycompany.example.com/employees">
        Any Name
    </n:name>
    <o:number>123456789099999</o:number>
    <o:expiration>2005-02</o:expiration>
   </o:creditCard>
  </m:chargeReservation
 </env:Body>
</env:Envelope>
```

Пример HTTP SOAP ответа.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
 <env:Header>
      ...
      ...
 </env:Header>
 <env:Body>
      ...
      ...
 </env:Body>
</env:Envelope>
```

2.2.3 JSON-RPC

Протокол JSON-RPC, опубликованный в 2009 году, по своим принципам работы очень похож на XML-RPC. Главными отличиями являются способ кодирования данных, независимость от транспортного уровня, способность передачи извещений (notification request) и возможность идентификации ответов при отправке нескольких запросов одновременно.

Для кодирования данных в JSON-RPC используется JSON. Кроме имени процедуры и параметров, в запросе указывается также значение id, который используется для идентификации ответа на стороне клиента. Другими словами, если вы отправили запрос с id=12345, то ответ этого запроса обязательно должен возвращать сообщение с id=12345.

Извещение – это специальные запросы, на которые сервер может не отвечать. Для отметки запроса, как извещение, значения параметра id указывается = null.

Независимость от транспортного уровня можно обусловить только тем, что в спецификации JSON-RPC HTTP не указывается обязательным протоколом. При использовании JSON-RPC над HTTP, необходимо использовать POST запросы.

Примеры запросов и ответов.

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}
```

2.2.4 REST API

REST API предлагает архитектуру программного обеспечения при создании веб-сервисов в виде определенного набора правил и ограничений. Веб-службы, соответствующие архитектурному стилю REST, называются веб-службами RESTful, они позволяют запрашивающим системам получать доступ и управлять текстовыми представлениями веб-ресурсов с помощью унифицированного и предопределенного набора операций без сохранения состояния клиента. Таким образом, в отличие от SOAP, REST — это не протокол, а архитектурный стиль. Архитектура REST устанавливает набор рекомендаций, которым необходимо следовать, если вы хотите предоставить веб-службу RESTful, например, существование без сохранения состояния и использование кодов состояния HTTP.

В веб-службе RESTful запросы к URI ресурса позволяют получить ответ с данными, отформатированными в HTML, XML, JSON или другом формате. В ответе может быть подтверждено, что состояние ресурса изменилось и содержаться гипертекстовые ссылки на другие связанные ресурсы. При обычном использовании HTTP доступны операции (методы HTTP): GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS и TRACE. RESTful службы ограничены фактически пятью операциями, GET (запрос на получение данных), POST (запрос на добавление нового ресурса), PUT (запрос на обновление всего ресурса), DELETE (запрос на удаление ресурса), PATCH (запрос на обновление части ресурса). Эти пять операций соответствуют четырем стандартным операциям над данными, это операции CRUD (Create, Read, Update, Delete). Create это POST, Read это GET, Update это PUT и PATCH, Delete это DELETE.

Используя протокол без сохранения состояния и стандартные операции, RESTful системы дают высокую производительность, надежность и масштабируемость за счет повторного использования ее компонентов. Появляется возможность управления и обновления этих компонент, без изменений системы в целом, даже во время работы.

Термин «репрезентативная передача состояния» («Representational State Transfer») был введен и определен в 2000 году Роем Филдингом в его докторской диссертации «Архитектурные стили и проектирование сетевых архитектур программного обеспечения», где описаны принципы REST известные еще в 1994 году как «объектная модель HTTP» и использовались при разработке стандартов HTTP 1.1 и универсальных идентификаторов ресурсов (URI). Данные принципы позволяют создать хорошо спроектированное веб-приложение, фактически представляющее собой сеть веб-ресурсов (виртуальную машину состояний), по которой пользователь осуществляет навигацию, выбирая идентификаторы ресурсов (например, http://www.example.ru/home/some), а также операции с ресурсами, такие как GET или POST (переходы между состояниями приложения). Каждый раз уже

следующее состояние приложения передается конечному пользователю для использования.

Ограничения архитектурного стиля REST улучшают следующие свойства приложения:

1) Производительность при взаимодействии различных компонентов приложения, которая может быть доминирующим фактором при восприятии пользователем производительности и эффективности веб-сервиса;

2) Масштабируемость, давая возможность поддерживать значительное количество компонентов и способов взаимодействия между ними. Напомним, что расширяемость дает возможность легко добавить новый компонент в систему, а свойство масштабируемости позволяет добавить дополнительный компонент без значительной потери производительности. Рой Филдинг описывает влияние REST на масштабируемость следующим образом: «Разделение задач клиент-сервер в REST упрощает реализацию компонентов, снижает сложность семантики связей, повышает эффективность настройки производительности и увеличивает масштабируемость чистых серверных компонентов. Ограничения многоуровневой системы позволяют добавлять посредников - прокси, шлюзы и брандмауэры - на различных этапах взаимодействия без изменения интерфейсов между компонентами, позволяя транслировать сообщения или повышать производительность за счет кэширования. REST дает возможность промежуточной обработки, за счет требования к самоописанию сообщений: взаимодействие между запросами не имеет состояния, для описания семантики и обмена информацией используются стандартные методы и медиа-типы, а ответы явно указывают на кэшируемость»;

3) простота унифицированного интерфейса;

4) возможность модификации компонентов при необходимости (даже во время работы приложения);

5) ясность и понятность связей между компонентами;

6) переносимость компонентов за счет перемещения программного кода вместе с данными;

7) надежность в устойчивости к отказам всей системы при наличии отказов в отдельных компонентах, соединениях или данных.

Систему RESTful определяют шесть основных ограничений, накладываемые на способы, с помощью которых сервер может обрабатывать запросы клиентов и отвечать на них, таким образом, что, работая в рамках этих ограничений, система приобретает желаемые нефункциональные свойства, такие как производительность, масштабируемость, простота, модифицируемость, видимость, переносимость и надежность. Если система нарушает какое-либо из обязательных ограничений, она не может считаться RESTful.

Формальные ограничения REST следующие:

1. Клиент-серверная архитектура.

Принцип, лежащий в основе ограничений клиент-сервер, заключается в разделении ответственности. Отделение пользовательского интерфейса от хранилища данных улучшает переносимость пользовательских интерфейсов на разные платформы, улучшает масштабируемость за счет упрощения серверных компонентов. Такое разделение позволяет компонентам развиваться независимо.

2. Отсутствие состояния

Взаимодействие клиент-сервер по REST предполагает отсутствие хранения на сервере клиентского контекста (состояния) между запросами. Каждый запрос от любого клиента содержит всю информацию, необходимую для обслуживания запроса (в том числе, например, данные требуемые для аутентификации), а состояние сеанса сохраняется в приложении клиента. Клиент начинает отправлять запросы, когда он готов к переходу в новое состояние. Пока один или несколько запросов не выполнены, считается, что клиент находится в процессе перехода.

3. Кэшируемость

Клиентские приложения и посредники (например, прокси) могут кэшировать ответы. Ответы должны, неявно или явно, указывать на то кэшируемые они или некэшируемые, чтобы клиенты не предоставляли устаревшие

или несоответствующие данные в ответ на последующие запросы. Хорошо управляемое кэширование частично или полностью позволяет исключить взаимодействие между клиентом и сервером, дополнительно улучшая масштабируемость и производительность.

4. Многоуровневая система

С учетом многослойности клиент не замечает, подключен ли он напрямую к конечному серверу или через посредника. Если между клиентом и сервером размещен прокси или балансировщик нагрузки, это не влияет на обмен данными, таким образом не требуется какое-либо обновление кода клиента или сервера. Промежуточные серверы могут улучшить масштабируемость системы за счет балансировки нагрузки и использования общих кэшей. Кроме того, можно поверх веб-сервиса добавить слой обеспечения безопасности, отделив бизнес-логику от логики безопасности. Добавление слоя безопасности в качестве отдельного уровня обеспечивает соблюдение политик безопасности. С учетом многоуровневости сервер может вызывать несколько других серверов для генерации ответа клиенту.

5. Код по запросу

Серверы могут временно расширять, изменять или настраивать функциональность клиента, передавая исполняемый код: например, скомпилированные компоненты, такие как апплеты Java, или сценарии на стороне клиента, такие как JavaScript.

6. Единый интерфейс

Единообразные правила ограничений интерфейса являются фундаментальными для разработки любой системы RESTful. Они упрощают и разделяют архитектуру, что позволяет каждой части развиваться независимо. Существует четыре ограничения унифицированного интерфейса RESTful:

1) Идентификация ресурса.

Отдельные ресурсы идентифицируются в запросах с помощью URI. Сами ресурсы концептуально отделены от представлений, возвращаемых клиенту. Например, сервер может отправлять данные из своей базы данных в

формате HTML, XML или JSON, ни один из которых не является внутренним способом представления или хранения данных сервера.

2) Управление ресурсами через представления

Когда клиент имеет представление ресурса, в том числе метаданные (описание данных), у него достаточно информации, чтобы изменить или удалить ресурс.

3) Самоописательные сообщения

Каждое сообщение содержит достаточно информации для его обработки. Например, тип парсера для вызова может быть определен MIME типом.

4) Гипермедиа как средство изменения состояния приложения (HATEOAS, Hypermedia As The Engine Of Application State)

Часто до 2016-2017 года встречались REST приложения, которые не использовали принцип единого интерфейса, что не совсем соответствовало REST. API описывалось в отдельной документации на отдельном ресурсе, и ее надо было скачать и изучить.

В гипермедия получение доступа к начальному URI для приложения REST - аналогично доступу обычного пользователя Web к домашней странице веб-сайта - клиент REST должен иметь возможность динамически использовать предоставляемые сервером ссылки для обнаружения всех необходимых ему доступных ресурсов. По мере доступа сервер возвращает содержимое, которое включает гиперссылки на другие доступные в настоящее время ресурсы. Нет необходимости в том, чтобы на клиенте была жестко закодирована информация о структуре или динамике приложения. Представление ресурса само себя описывает в достаточной степени, чтобы клиент понял, что с ним можно делать. Применение такого подхода обычно означает, что клиент знает некоторый конечный набор "точек входа" (можете считать их аналогами стартовых страниц на сайтах), с которых он начинает свое взаимодействие с API, используя предоставленную в представлении ресурса информацию для навигации к другим ресурсам и совершения действий. Для достижения этой задачи как раз и используются гиперссылки (hypermedia).

2.2.4.1 Примеры использования гипермедиа в REST

```
//список товаров
{
 "links": {
  "self"  "/recipes/products"
 }
 "items": [
     {
       "name": "milk",
       "rating": 5,
       "shortDescription": "...."
       "links": {
         "self": "/recipes/products/milk"
       }
     },
{
       "name": "bread",
       "rating": 6,
       "shortDescription": "...."
       "links": {
         "self": "/recipes/products/bread"
       }

 ]
}

//конкретный товар
{
  "links": {
    "self": "/recipes/cookies/milk"
    "next": { "href": "/recipes/cookies/milk?page=2" },
  }
  "name": "milk",
  "rating": 5,
  "shortDescription": "....",
  "description": "...."
  "composition": [
     {
      "name": "water",
      .....
     },
     {
      "name": "protein",
      .....
     },
     {
      "name": "carbohydrates",
      .....
     }
  ],
  "shelf life": [
   ....
  ]
}
```

Здесь как мы видим, используется специальное поле links, указывающее на URI ресурса, или даже на другой ресурс, таким образом, получив ответ от сервера, можно понять куда и как переходить или автоматически пе-

рейти на другие ресурсы по указанным ссылкам. В принципе можно реализовать автоматизированный унифицированный клиент, который будет сам обрабатывать переход по ссылкам, получаемым от сервера.

Еще один пример, который дает лучшее понимание этого процесса. Клиент получил вот такую JSON структуру.


```
{
  "content": [ {
    "price": 499.00,
    "description": "Apple tablet device",
    "name": "iPad",
    "links": [ {
      "rel": "self",
      "href": "http://localhost:8080/product/1"
    } ],
    "attributes": {
      "connector": "socket"
    }
  }, {
    "price": 49.00,
    "description": "Dock for iPhone/iPad",
    "name": "Dock",
    "links": [ {
      "rel": "self",
      "href": "http://localhost:8080/product/3"
    } ],
    "attributes": {
      "connector": "plug"
    }
  } ],
  "links": [ {
    "rel": "product.search",
    "href": "http://localhost:8080/product/search"
  } ]
}
```


Ясно, что можно перейти, в том числе, по ссылке поиска продуктов. При этом rel указывает на отношение, здесь имя свойства «rel» указано явно. В данном случае, гиперссылка делается на себя (self). Более сложные системы могут включать другие отношения. К примеру, отношение "rel":"customer" связывает запрос с другим клиентом. То же самое можно описывать, используя XML, а не JSON. Стандартом Hypermedia кроме links предусмотрены и другие свойства, например, title указывающее название, которое предназначено для чтения человеком, таким образом, на сервере можно определять все переходы для унифицированного клиента.

2.2.4.2 Примеры использования запросов HTTP в RESTful

Далее показано, каким образом методы HTTP могут быть использованы в API HTTP, включая RESTful.

Для серверов, которые проводят вычисления или осуществляют вызов процедур.

POST: Вызов операции, предоставляемой ресурсом. Если в результате запроса создается новый ресурс, его URI возвращается в заголовке Location ответа.

GET: Получить статус асинхронной операции в теле ответа.

DELETE: Отменить асинхронную операцию.

Для серверов, которые манипулируют данными.

GET: Получить представление данных в теле ответа.

PUT: Сохранить представление находящееся в теле запроса как новое состояние ресурса. Ожидается, что последующий доступ GET к ресурсу вернет это представление со статусом HTTP 200 до тех пор, пока не будет вызвано PATCHED, PUT, DELETE для данного ресурса.

PATCH: Обновляет некоторую часть ресурса, используя инструкции в теле запроса.

DELETE: Удалить ресурс. Ожидается, что последующий доступ GET к ресурсу будет возвращать статус HTTP 404 до тех пор, пока ему не будет установлено новое состояние PUT.

Метод GET безопасен, это означает, что его применение к ресурсу не приводит к изменению состояния ресурса (только для чтения). Методы GET, PUT и DELETE идемпотентны, что означает, что их многократное применение к ресурсу приводит к тому же изменению состояния ресурса, что и однократное применение, хотя ответ может отличаться.

В отличие от веб-служб на основе SOAP, для веб-API RESTful не существует «официального» стандарта, так как REST - это архитектурный стиль, а SOAP - это протокол. REST сам по себе не является стандартом, но реализации RESTful используют другие стандарты: HTTP, URI, JSON и XML. Мно-

гие разработчики указывают свои API-интерфейсы как RESTful, хотя на самом деле эти API-интерфейсы не удовлетворяют всем архитектурным ограничениям, описанным выше (особенно ограничениям унифицированного интерфейса).

Далее представлено как HTTP-методы обычно используются в REST API для ресурсов коллекция (collection, например, https://api.example.com/collection) и ресурсов-элементов (member, например, https://api.example.com/collection/item3). Указанные здесь ссылки приведены лишь для примера, в реальности их не существует. Напомним, что коллекция не имеет идентификаторов, а member имеет id для каждого элемента.

В общем и целом говоря, ресурс может быть либо одним элементом, либо коллекцией. Например, customers (клиенты) - это ресурс-коллекция, а customer (клиент) - ресурс-элемент. Мы можем идентифицировать ресурс коллекции customers, используя URI /customers, а ресурс одного клиента customer с помощью URI /customers/{customerId}. Хотя, можно еще рассмотреть хранилища и контроллеры.

Ресурс также может содержать вложенные коллекции. Например, вложенная коллекция accounts конкретного клиента (customer) может быть идентифицирована с использованием URI /customers/{customerId}/accounts. Аналогичным образом, одноэлементный ресурс account внутри вложенного ресурса accounts можно определить следующим образом: /customers/{customerId}/accounts/{accountId}.

Итак, использование HTTP для ресурсов коллекций.

GET используется для получения URI ресурсов-элементов ресурса коллекции в теле ответа.

POST используется для создания ресурсов-элементов в ресурсе коллекции, используя инструкции в теле запроса. В результате запроса URI созданного ресурса-элемента автоматически возвращается в поле заголовка ответа Location.

PUT используется для замены всех ресурсов-элементов в ресурсе коллекции или используется для создания ресурса коллекции, если он не существует.

PATCH используется для обновления всех ресурсов-элементов ресурса коллекции, или для создания ресурса коллекции, если он не существует.

DELETE используется для удаления всех ресурсов-элементов ресурса коллекции.

Использование HTTP для ресурсов-элементов.

GET: Получение ресурса-элемента в теле ответа.

POST: Создание ресурса элемента внутри другого ресурса элемента, используя инструкции в теле запроса. Происходит автоматическое связывание URI с ресурсом-элементом и URI возвращается в поле заголовка Location ответа.

PUT: Замена всех ресурсов-элементов или создание ресурса-элемента, если он не существует.

PATCH: Обновление всех ресурсов-элементов или создание ресурса-элемента, если он не существует, используя инструкции в теле запроса.

DELETE: Удаление ресурса-элемента.

Рассмотрим более подробно типы ресурсов.

Тип документ или ресурс-элемент соответствует одному экземпляру записи в базе данных, или экземпляр объекта, отдельный элемент в ресурсе коллекции. Именовать желательно существительным в единственном числе.

http://api.example.com/devices/{device-id}

http://api.example.com/users/{id}

http://api.example.com/users/admin

Тип коллекция - это каталог ресурсов, управляемый сервером, клиенты могут предлагать новые элементы в коллекцию, но обычно в самой коллекции определяется, создавать новый элемент или нет, коллекция будет определять URI каждого содержащегося элемента. Для именования ресурса коллекции лучше использовать существительное в множественном числе.

http://api.example.com/devices

http://api.example.com/users

http://api.example.com/users/{id}/accounts

Тип ресурса хранилища представляет собой репозиторий объектов, позволяющий пользователю вставлять, изменять, возвращать и добавлять ресурсы. При этом хранилище не генерирует новые URI, здесь URI назначается самим клиентом. Для именования хранилища используется множественное число.

http://api.example.com/users/{id}/carts

http://api.example.com//users/{id}/playlists

Ресурс контроллер используется для процедур. Фактически это функции возвращающие значения, для именования используется глагол.

http://api.example.com/users/{id}/cart/checkout

http://api.example.com/users/{id}/playlist/play

Дальнейшие соглашение, которые используются в RESTful:

Для обозначения иерархических отношений используется слэш «/», не использовать в конце ссылки слэш «/», для улучшения читаемости использовать дефис, а не нижнее подчеркивание (http://api.example.com/inventory-management/managed-entities/{id}/install-script-location), использовать буквы только в нижнем регистре, не использовать расширения файлов в названиях ресурсов и точку. При поиске, фильтрации, сортировке части ресурсов в коллекции по какому-либо атрибуту не создавать новую иерархию, а использовать запросы с параметрами вида http://api.example.com/devices?region=Russia.

2.2.5 JSON pure API

JSON-pure API использует только один метод для передачи данных — обычно POST для HTTP и SEND в случае использования Web Sockets.

Механизм передачи и содержимое запроса полностью независимы. Все ошибки, предупреждения и данные передаются в теле запроса, в формате JSON.

Используется лишь один код ответа, чтобы подтвердить успешную передачу, обычно это 200 OK.

2.2.6 JSON API

JSON:API - это спецификация процесса запроса клиентом выборок ресурсов или их изменения и того как сервер должен отвечать на эти запросы.

JSON:API предназначен для минимизации как количества запросов, так и объема данных, передаваемых между клиентами и серверами. Эта эффективность достигается без ущерба для удобочитаемости, гибкости или поиска.

JSON: API требует использования медиа формата JSON:API (application / vnd.api + json) для обмена данными. На данный момент существует версия 1.0.

Ответственность клиента:

Клиенты ДОЛЖНЫ отправлять все данные JSON:API в запросах с заголовком Content-Type: application / vnd.api + json без каких-либо параметров медиа-типа.

Клиенты, которые включают медиа тип JSON: API в свой заголовок Accept, ДОЛЖНЫ указать медиа-тип хотя бы один раз без каких-либо параметров медиа-типа.

Клиенты ДОЛЖНЫ игнорировать любые параметры для медиа-типа application / vnd.api + json, полученные в заголовке Content-Type ответных документов.

Ответственность сервера

Сервер ДОЛЖЕН отправлять все данные JSON:API в ответных документах с заголовком Content-Type: application / vnd.api + json без каких-либо параметров медиа-типа.

Сервер ДОЛЖЕН ответить кодом состояния 415 Unsupported Media Type, если в запросе указан заголовок Content-Type: application / vnd.api + json с любыми параметрами медиа-типа.

Сервер ДОЛЖЕН ответить кодом состояния 406 Not Acceptable, если заголовок Accept запроса содержит медиа-тип JSON:API, и все экземпляры этого медиа-типа изменены с параметрами медиа-типа.

Объект JSON ДОЛЖЕН находиться в корне каждого запроса и ответа JSON: API, содержащего данные. Этот объект представляет собой «верхний уровень» документа.

Документ ДОЛЖЕН содержать хотя бы один из следующих элементов верхнего уровня:

data: «первичные данные» документа.

errors: массив ошибок (объекты ошибки)

meta: мета-объект, содержащий нестандартную метаинформацию.

Данные и ошибки НЕ ДОЛЖНЫ находиться в одном документе.

Документ МОЖЕТ содержать любой из следующих элементов верхнего уровня:

jsonapi: объект, описывающий реализацию сервера

links: объект ссылок, относящийся к первичным данным.

included: массив объектов ресурсов, связанных с первичными данными и / или друг с другом («включенные ресурсы»).

Если документ не содержит ключа данных верхнего уровня, включенный член НЕ ДОЛЖЕН присутствовать.

Объект ссылок верхнего уровня МОЖЕТ содержать следующие элементы:

self: ссылка, по которой был создан текущий ответ (документ в ответе сервера).

related: ссылка на связанный ресурс, когда первичные данные представляют собой отношения между ресурсами.

Ссылки на страницы для первичных данных.

«Первичные данные» документа *- это представление ресурса или совокупности ресурсов, на которые направлен запрос.*

Первичные данные ДОЛЖНЫ быть либо:

одиночным объектом ресурса, одиночным объектом идентификатора ресурса или null, если запросы были, нацелены на отдельные ресурсы.

массивом объектов ресурсов, массивом объектов идентификаторов ресурсов или пустой массив ([]) если запросы были нацелены на коллекции ресурсов.

Ниже приведен пример первичных данных, которые представляют собой отдельный объект ресурса:

```
{
  "data": {
    "type": "articles",
    "id": "1",
    "attributes": {
      // ... this article's attributes
    },
    "relationships": {
      // ... this article's relationships
    }
  }
}
```

А следующий пример представляют собой одиночный объект идентификатора ресурса, который ссылается на тот же ресурс:

```
{
  "data": {
    "type": "articles",
    "id": "1"
  }
}
```

Логическая коллекция ресурсов ДОЛЖНА быть представлена в виде массива, даже если она содержит только один элемент или пустая.

2.2.6.1 Объекты ресурсов

«Объекты ресурсов» нужны в документе JSON:API для представления ресурсов.

Объект ресурса ДОЛЖЕН содержать как минимум следующие элементы верхнего уровня:

id

type

За исключением случая, когда член id не требуется, если объект ресурса приходит от клиента и представляет новый ресурс, который должен быть создан на сервере.

Кроме того, объект ресурса МОЖЕТ содержать любой из этих элементов верхнего уровня:

attributes: объект атрибутов, представляющий некоторые данные ресурса.

relationship: объект отношений, описывающий отношения между ресурсом и другими ресурсами JSON:API.

links: объект ссылок, содержащий ссылки, относящиеся к ресурсу.

meta: мета-объект, содержащий нестандартную метаинформацию о ресурсе, которая не может быть представлена как атрибут или отношение.

Например, ресурс статья может отображаться в документе следующим образом:

```
{
 "type": "articles",
 "id": "1",
 "attributes": {
  "title": "Rails is Omakase"
 },
 "relationships": {
  "author": {
   "links": {
    "self": "/articles/1/relationships/author",
    "related": "/articles/1/author"
   },
   "data": { "type": "people", "id": "9" }
  }
 }
}
```

2.2.6.2 Получение данных

Данные, включая ресурсы и отношения, можно получить, отправив запрос GET на сервер.

Ответы можно дополнительно уточнить с помощью дополнительных функций

Например, следующий запрос извлекает коллекцию статей:

```
GET /articles HTTP/1.1
Accept: application/vnd.api+json
```

Следующий запрос получает статью

```
GET /articles/1 HTTP/1.1
Accept: application/vnd.api+json
```

А этот запрос получает автора статьи.

```
GET /articles/1/author HTTP/1.1
Accept: application/vnd.api+json
```

Ответы сервера.

```
200 OK
```

Сервер ДОЛЖЕН ответить на успешный запрос на выборку отдельного ресурса или коллекции ресурсов ответом 200 OK.

Сервер ДОЛЖЕН вернуть на успешный запрос на выборку коллекции ресурсов с массив объектов ресурсов или пустой массив ([]) в качестве первичных данных ответного документа.

Например, запрос GET к коллекции статей может вернуть:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json
{
  "links": {
    "self": "http://example.com/articles"
  },
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON:API paints my bikeshed!"
    }
  }, {
    "type": "articles",
    "id": "2",
    "attributes": {
      "title": "Rails is Omakase"
    }
  }]
}
```

Пример возвращения пустой коллекции.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "http://example.com/articles"
  },
  "data": []
}
```

Другие примеры вы можете найти на официальном сайте JSON:API.

2.3 Документирование веб-сервиса

Часто разработанный веб-сервис необходимо документировать для того, чтобы другие системы и разработчики могли им воспользоваться, но и Вы сами при разработке могли помнить, что реализует та или иная функция, которую можно вызвать, обращаясь к вашему веб-сервису. Современные фреймворки в большинстве своем реализуют возможности проводить автоматизированное документирование и создание описания API функций на основе различных технологий, например, Swagger или RAML, более того некоторые из них, например, Swagger позволяет по декларативному описанию API (тех самых функций GET, POST и т.д.) создать и готовое серверное и готовое клиентское приложение фактически для любого фрейморка с использованием практически любой существующей технологии реализации серверов динамического контента (PHP, ASP .NET, JSP, Node.js и т.д.).

2.3.1 Swagger (OpenAPI)

В настоящее время OpenAPI и ранее известная как Swagger Specification представляет собой формализованную спецификацию и полноценный фреймворк для описания, создания, использования и визуализации веб-сервисов REST, фактически декларативная спецификация для описания REST API. Основная задача данной спецификации - позволить клиентским системам синхронизировать свое состояние и состояние документации с изменениями на сервере. Это достигается путем того, что методы, параметры, модели и другие элементы посредством OpenAPI интегрируются с программным обеспечением сервера и всё время с ним синхронизируются. Swagger появился в 2010 году и был разработан для внутренних нужд компании Wordnik компанией SmartBear и затем в 2015 переведен в open source. Данная спецификация не зависит от языка программирования и может быть

использована вне протокола HTTP. Используется декларативное описание, например, с использованием JSON, YAML.

Это декларативное описание представляет из себя единый файл в формате JSON или YAML, состоящий из трёх разделов:

- заголовок, содержащий название, описание и версию API, а также дополнительную информацию;
- описание всех ресурсов, включая их идентификаторы, HTTP-методы, все входные параметры, а также коды и форматы тела ответов, со ссылками на определения;
- все определения объектов в формате JSON Schema, которые могут использоваться как во входных параметрах, так и в ответах.

У OpenAPI есть серьёзный недостаток — сложность структуры и, зачастую, избыточность. Для небольшого проекта содержимое JSON-файла спецификации может быстро разрастись до нескольких тысяч строк. В таком виде поддерживать этот файл вручную невозможно. Это — серьёзная угроза для самой идеи поддержания актуальной спецификации по мере развития API.

Существует множество визуальных редакторов, позволяющих описывать API и формирующих в итоге спецификацию OpenAPI. На них в свою очередь основаны дополнительные сервисы и облачные решения, например, сам Swagger, Apiary, Stoplight, Restlet и другие. Swagger по сути, это интерфейс пользователя для OpenAPI, который позволяет визуализировать спецификацию.

На сайте https://petstore.swagger.io приводится пример api для хранения информации о питомцах, при этом rest api отображается визуально приглядно (рисунок 2.2).

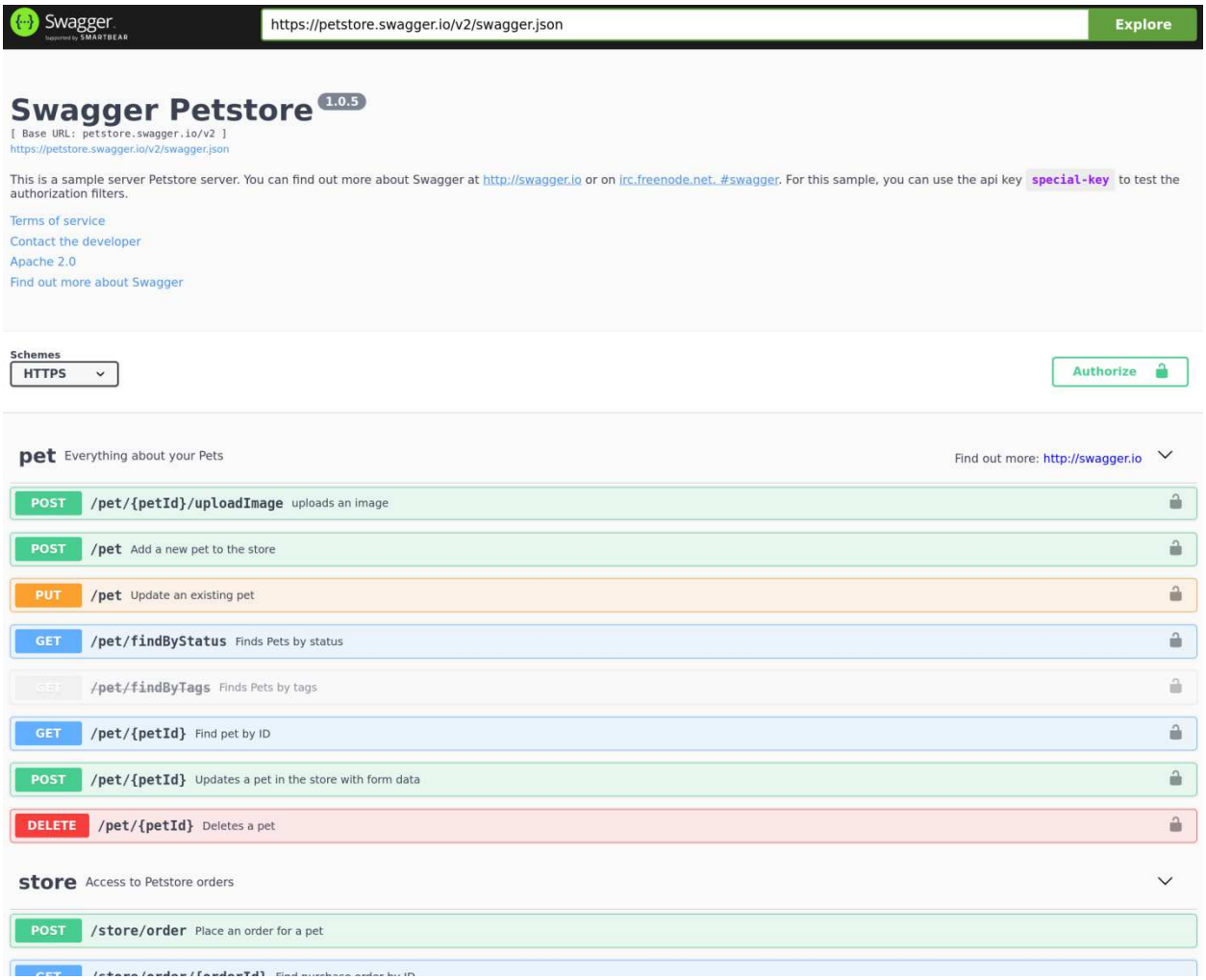


Рисунок 2.2 - Пример отображения REST API для сайта домашних животных

Swagger позволяет показывать и описывать Endpoints (фактически те запросы и URI, которые используются на веб-сервисе), авторизацию (Authorization), модели или схемы с которыми происходят манипуляции с помощью API (Models), описание примеров ожидаемых запросов и ответов API (Requests, Responses), кроме того, возможно через данный UI запускать в реальном времени запросы к создаваемому вами API (Execution).

В основном Swagger, очевидно, используется на стороне клиента (frontend), но и полезен на стороне сервера (backend). На этапе проектирования API можно описать endpoints, а также вести документацию, фактически предоставляя информацию об API другим пользователям или для фронтенда.

Можно сгенерировать непосредственно клиента или сервер по спецификации API Swagger, для этого нужен генератор кода Swagger-Codegen.

2.3.2 RAML

RAML расшифровывается как REST API Modeling Language и аналогичен спецификации OpenAPI. RAML поддерживается Mulesoft, компанией, предоставляющей полный комплекс услуг API.

Как и в OpenAPI, после создания файла RAML, описывающего API, он может использоваться различными платформами для анализа и отображения информации в интерактивном выводе. Формат RAML, в котором используется синтаксис YML, легко читаем человеком, эффективен и прост. Вот вывод RAML в консоли API похож на интерфейс Swagger.

Поддерживается строгая структура, YAML, простой удобный язык разметки markdown, допускается декомпозиция файлов с использованием подключения include, в этом есть некоторое преимущество swagger, где файл может значительно разрастаться.

Документация RAML легко обрабатывается программно.

Можно привести простое API, вот так будет выглядеть описание в файле example.raml.

```
#%RAML 0.8
title: Some Title
version: v3
baseUri: https://example.com/api/v1/
```

Добавим в файл вводную документацию и для каждого раздела укажем подзаголовок.

```
documentation:
 - title: Our Title
   content: |
     Some notes on this document.
 - title: Authorization
   content: |
     Some text to describe HTTP headers to access data.
     ~~~
     code goes here...
     ~~~
```

Далее можно задекларировать правила авторизации. Допустим, обычная базовая авторизация по протоколу HTTPS. В стандарте RAML присутствует данный вид авторизации:

```
securitySchemes:
 - Basic Authentication:
    type: Basic Authentication
```

Теперь, описывая URL, можно сослаться на данный тип авторизации. В скомпилированном HTML-документе напротив URL будет стоять значок замочка, а по клику откроется окно с описанием правил авторизации.

Далее можно описать требуемые URL.

```
/api/v1/user:
  securedBy: [Basic Authentication]
  get:
    description: |
      Retrieves a list of users.
      ### Syntax
      ~~~
      GET /api/v1/user/
      Host: {host}
      Authorization: {authorization}
      ~~~
    queryParameters:
      !include user_list_params.raml
    responses:
      200:
        body:
          application/json:
            example: !include examples/user_list_response.json
            schema: !include schemas/user_list_response.json
```

Здесь реализуется описание API, которое возвращает список пользователей. Притом доступ к api защищен базовой авторизацией. В описании пример запроса. Файл user_list_params.raml содержит параметры командной строки: лимит, смещение, сортировка.

Схема ответа может быть достаточно большой, поэтому здесь используется include. В файле examples/user_list_response.json лежит следующая схема:

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "additionalProperties": false,
   "definitions": {
      "item": {
         "additionalProperties": false,
         "properties": {
            "id": {
```

```
                "minimum": 0,
                "type": "integer"
            },
            "name": {
                "maxLength": 64,
                "type": "string"
            }
        },
        "required": [
            "id",
            "name"
        ],
        "type": "object"
    }
},
"properties": {
    "count": {
        "minimum": 0,
        "type": "integer"
    },
    "results": {
        "items": {
            "$ref": "#/definitions/item"
        },
        "type": "array"
    }
},
"required": [
    "count",
    "results"
],
"type": "object"
}
```

В начале схемы декларируются параметры пользователя, описываемые как item с именем и id. А результирующий ответ - это массив item.

Документ очень удобно изменять и расширять, добавляя методы (POST, PUT, DELETE), ссылаться на уже определенные структуры, выносить повторы в include файлы.

С помощью утилит можно скомпилировать документ в файл html. Утилита raml2html написана на node.js и ее можно установить через npm.

raml2html example.raml > example.html

Также при компиляции можно использовать свой собственный шаблон, а не шаблон по умолчанию.

raml2html -t custom-templates/template.nunjucks -i example.raml -o example.html

Здесь были приведены описания с использованием JSON. Можно использовать YAML.

Например, описание типов данных используя RAML 1.0 можно реализовать следующим образом:

```
#%RAML 1.0
title: New API
mediaType: application/json
types:
  Person:
    type: object #
    properties:
      firstname: string
      lastname: string
      is_our_employee: boolean
 Document:
    type: object
    properties: Author
     title: string
     signing_date: date
```

Как видите, здесь описывается объект персона и объект документ со свойствами: Автор, название и дата подписания.

Далее приведем пример использования типа данных документ для описания тела ответа.

```
/documents/{documentId}:
  get:
    responses:
      200:
        body:
          application/json:
            type: Document
```

2.3.3 API Blueprint

API Blueprint, так же, как и Swagger определяет спецификацию для описания REST API. Инструменты, которые поддерживают API Blueprint, могут читать и отображать информацию.

Спецификация API Blueprint написана с использованием синтаксиса Markdown. Blueprint представляет собой конкретную схему, которая является правильной или неправильной на основе используемых имен элементов, порядка расположения и т.д. и не является такой же гибкой, как Markdown.

Справочник по синтаксису можно посмотреть на https://apiblueprint.org/documentation/tutorial.html.

Проект имеет онлайн редактор. Поддерживает генерацию примеров запроса для языков: Java, Javascript, Node.js, Perl, Python. PHP, Ruby, Go, C#, Visual Basic, Groovy, Objective-C, Swift.

Пример описания простейшего api с самого сайта справочника приводится ниже.

```
# The Simplest API
This is one of the simplest APIs written in the **API Blueprint**. One plain
resource combined with a method and that's it! We will explain what is going on
in the next installment -
[Resource and Actions](02.%20Resource%20and%20Actions.md).
**Note:** As we progress through the examples, do not also forget to view the
[Raw](https://raw.github.com/apiaryio/api-blueprint/master/examples/01.%20Simplest%20API.md)
code to see what is really going on in the API Blueprint, as opposed to just
seeing the output of the Github Markdown parser.

Also please keep in mind that every single example in this course is a **real
API Blueprint** and as such you can **parse** it with the
[API Blueprint parser](https://github.com/apiaryio/drafter) or one of its
[bindings](https://github.com/apiaryio/drafter#bindings).
# GET /message
+ Response 200 (text/plain)
    Hello World!
```

Контрольные вопросы по главе 2

1. В чем назначение OpenApi и RAML?
2. Что такое прогрессивные мобильные веб-приложения PWA?
3. Назначение запроса HTTP PATCH в REST?
4. Основные ограничение REST?
5. Для чего предназначен SOAP?
6. Назвать основные операции HTTP REST.
7. Что такое нативные веб-приложения?
8. Что такое технология клиент-сервер?

3 XAAS (ANYTHING AS A SERVICE, ЧТО-ЛИБО КАК СЕРВИС).

Современные веб-сервисы и приложения разворачиваются в основном так же с использованием удаленных сетевых сервисов или облачных сервисов, что создают отдельную бизнес отрасль по предоставлению вычислительных возможностей, одна из концепций делящая по уровням предоставляемых функциональных возможностей или целей использования сервиса это XaaS.

XaaS является аббревиатурой от слова «что-либо как услуга» в контексте облачных вычислений (например, X как услуга), X определяет то, что представляется потребителю, обычно с точки зрения уровня предоставляемого сервиса. XaaS предоставляют клиентам/потребителям конечные вычислительные точки, интерфейс которых обычно управляется через API, но также чаще всего ими можно управлять через веб-консоль в веб-браузере пользователя. Являют собой довольно сложные системы, обладающие высокой степенью внутренней автоматизации, обеспечивая различные степени отказоустойчивости, возможность указания требуемой производительности или автоматического выделения необходимых вычислительных ресурсов, и обычно предназначены для выполнения своих повседневных функций без вмешательства человека. На рисунке 3.1 приводится иерархия сервисов.

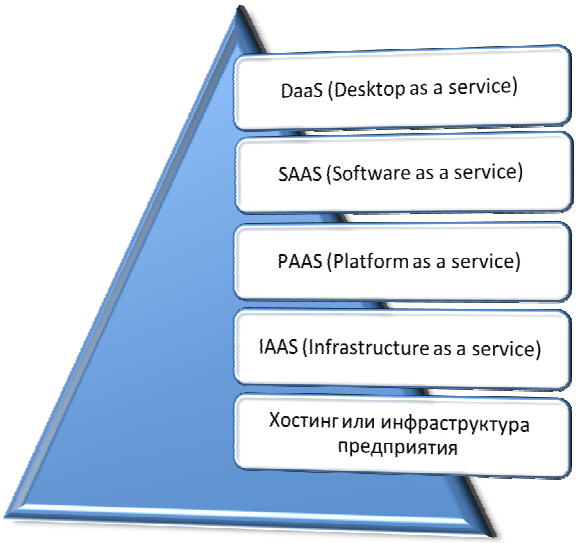


Рисунок 3.1 – Иерархия сервисов XaaS (что-либо как сервис).

Существует несколько видов подобных сервисов по уровню обслуживания, DaaS, SaaS, PaaS, IaaS.

DaaS (Desktop as a Service)

Переводится как «рабочий стол как услуга». Здесь сервисом является рабочее место, которое готово к использованию и снабжено всеми необходимыми средствами для работы пользователя.

Основные достоинства DaaS

- возможность быстро организовать офис с минимальными первоначальными затратами;
- возможность дать доступ к полноценному рабочему месту для разъездных сотрудников (командировки, торговые представители);
- дополнительная защита основного массива корпоративной информации;
- стандартизация рабочих мест;
- контроль над потоками данных пользователей и централизованное обслуживание;

Основные недостатки DaaS

- зависимость от качества канала связи;
- рост Интернет-трафика

SaaS (Software as s Service)

Переводится «программное обеспечение как услуга». В этой модели поставщик сервиса использует собственное интернет-приложение и предоставляет возможность потребителям пользоваться им через Всемирную сеть. Основные особенности SaaS:

- Пользователи услуги не платят за обновления, установку, обслуживание используемого аппаратного и программного обеспечения.

- Улучшение и обновление сервиса осуществляется прозрачно для пользователей — им не нужно вручную производить для этого какие-либо манипуляции.
- За использование сервиса поставщик взимает оплату. Цена определяется продолжительностью доступа к услуге (например, за месяц) или объемом выполненных операций.

Примерами могут служить Сервер 1С-Предприятие (Бухгалтерия, зарплата и управление персоналом, управление торговлей, комплексная автоматизация), Сервер 1С-Битрикс (Оптимальное решение для размещения сложных проектов на платформе 1С-Битрикс), Электронная почта (Электронные почтовые ящики с расширенной функциональностью или целый почтовый сервер), Сервер Asterisk (Облачная АТС позволяет гибко решить любые задачи телефонии с минимальными затратами), Файловый сервер (Безопасное хранение файлов с возможностью доступа с любых устройств), GitHub, GitLab.

PaaS.

Расшифровывается «платформа как услуга». В этой модели поставщик предлагает клиентам использовать свою облачную инфраструктуру с ОС, СУБД, а также всевозможными инструментами для разработки, которые клиент может выбрать.

Доступ к управлению облачной инфраструктурой PaaS имеет только провайдер. Он же задает набор доступных платформ, настроек и услуг.

Стоимость определяется объемом оказанных услуг, который может измеряться временем их использования, количеством операций.

Примеры:

Microsoft Azure

Помимо базовых функций операционных систем, Microsoft Azure имеет и дополнительные: выделение ресурсов по требованию для масштабирования, автоматическую синхронную репликацию данных для повышения отка-

зоустойчивости, обработку отказов инфраструктуры для обеспечения постоянной доступности и другое.

Heroku

Heroku утверждает, что предоставляет наибольшее количество разнообразных решений NoSQL на рынке информационных технологий.

Google App Engine

В отличие от многих обычных размещений приложений на виртуальных машинах, платформа App Engine тесно интегрирована с приложениями и накладывает на разработчиков некоторые ограничения.

Amazon web services

Ключевая инфраструктурная услуга — служба аренды виртуальных серверов. Подписчикам предоставляются виртуальные машины, работающие на гипервизоре Xen, доступен выбор различных по вычислительной мощности машин, а также машин с доступом к специализированному оборудованию.

PythonAnyWhere

Включает запуск и редактирование web-приложений на основе распространенных веб-фреймворков Python, базу данных MySQL и PostgreSQL, онлайн консоль Bash, веб-редактор кода.

IaaS (Infrastructure as a Service) Инфраструктура как сервис.

В данном случае клиент получает в распоряжение виртуальный сервер, виртуальную сеть и может разворачивать свои операционные системы или развернуть на базе систем виртуализации свое архитектурное решение для веб-сервиса.

Возникла идея подобной услуги в связи с тем, что для организации работы с информацией и доступа в сеть компании нужно обеспечить хранение и доступ к данным. При этом компания должна обеспечить себя нужной инфраструктурой — серверным и сетевым оборудованием, помещением для его размещения (дата-центр или серверная комната), специалистами для настройки и обслуживания, но организовывать собственную инфраструктуру

дорого и долго. Чтобы снизить расходы, можно арендовать место в дата-центре и установить там собственный сервер (colocation- свои стойки), можно арендовать сразу сервер (хостинг), а можно — вычислительные мощности: число ядер процессора, RAM и т. д. Последнее и будет IaaS. Услугами IaaS является «виртуальный дата-центр» от Selectel или CloudLITE, «виртуальный сервер» от ISPserver или RuVDS. Microsoft azure также предоставляет возможности IaaS. Главное отличие IaaS от традиционного хостинга — возможность быстро масштабироваться и брать плату только за потреблённые ресурсы. Возможность установить свою ОС в виртуальной машине.

Общая идея размещения веб-сервисов с помощью услуг отражена в таблице 3.1.

Таблица 3.1 – Типы размещения веб-сервисов.

Traditional IT	Hosting/ *Colocation*	IaaS	PaaS	SaaS
Данные	Данные	Данные	Данные	Данные
Приложения	Приложения	Приложения	Приложения	**Приложения**
Базы данных	Базы данных	Базы данных	**Базы данных**	**Базы данных**
Операционная система	Операционная система	Операционная система	**Операционная система**	**Операционная система**
Виртуализация	Виртуализация	**Виртуализация**	**Виртуализация**	**Виртуализация**
Физический сервер	**Физический сервер**	**Физический сервер**	**Физический сервер**	**Физический сервер**
Сети и хранилища	**Сети и хранилища**	**Сети и хранилища**	**Сети и хранилища**	**Сети и хранилища**
Дата-центр	***Дата-центр***	**Дата-центр**	**Дата-центр**	**Дата-центр**

Как видно из таблицы традиционная схема предполагает, что компания сама ответственна за создание всей необходимой инфраструктуры, включая создание Дата-центра. Colocation (размещение) дает инфраструктуру со стойками для размещения физических серверов и хранилищ. Традиционный хостинг обеспечивает наличие сетевых хранилищ и физический серверов в дата-

центре, а IaaS уже обеспечивает виртуализацию для того, чтобы клиент мог развернуть требуемую ему любую архитектуру веб-сервиса, включая виртуальные сервера, сети, операционные системы, балансировщики нагрузки, хранилища и прочее. PaaS предлагает системы разработки веб-сервисов, СУБД, сам реализует распределение нагрузки в случае чего, позволяет выделять необходимые вычислительные мощности и память. SaaS это уже полностью готовое приложение под ключ.

Но кроме рассмотренных сервисов с точки зрения уровней предоставляемых услуг от разработчика до пользователя существуют еще следующие XaaS.

BaaS – Backup-as-a-Service: резервное копирование

Этот сервис – позволяет обеспечить информационную безопасность предприятия, осуществляя регулярное резервное копирование данных в облаке и сохраняя их в удаленном хранилище. Удаленное хранение резервных копий повышает устойчивость информации к различного рода сбоям: даже если основная система пострадает от кибератаки, стихийного бедствия и прочих форс-мажорных неприятностей, данные уцелеют. И после восстановления из резервных копий система быстро вернет работоспособность.

RaaS – Recovery-as-a-Service: восстановление данных как услуга

RaaS позиционируют иногда как автономную от BaaS услугу, более продвинутую в плане защиты данных и скорости восстановления приложений. Иногда – как часть сервиса BaaS, часто – ассоциируют с DRaaS.

DRaaS – Disaster-Recovery-as-a-Service: услуга восстановления системы после аварий.

Это комплексный сервис, включающий в себя не только резервное копирование и инструменты восстановления системы, но и полную репликацию всех данных, бизнес-процессов и приложений. Фактически DRaaS создает клон основной инфраструктуры, которая непрерывно актуализируется. В случае сбоя или катастрофы работа компании продолжится в этой дополнительной инфраструктуре, пока будет восстанавливаться основная система.

Организовать отказоустойчивый клон основной инфраструктуры на собственной площадке – это колоссальные расходы, неподъемные для большинства фирм. В то же время облачный DRaaS гораздо более удобен, доступен и обеспечен дополнительной защитой, организованной провайдером.

MaaS – Monitoring-as-a-Service: мониторинг облачной инфраструктуры как услуга

Это очень перспективный сервис, который предлагают обычно крупные провайдеры. Используя MaaS, вы можете контролировать состояние ключевых параметров своей облачной инфраструктуры, нагрузку на приложения, а также работу установленного в облаке ПО. Назначение этой услуги – снижение рисков отказа инфраструктуры и простоев бизнеса, оптимизация процессов. Перспектива развития этого направления в будущем непосредственно связана с Big Data.

DBaaS – DataBase-as-a-Service: база данных как услуга

Данный сервис позволяет компаниям получать доступ к базам данных, обходясь при этом без настройки инфраструктуры или программного обеспечения. Администрирование, обслуживание, управление БД находится на стороне провайдера, клиент же просто пользуется БД как продуктом.

EaaS – Environment-as-a-Service: IT-среда как услуга

Это «факультативная» сервисная модель облачных решений. Иногда ее называют IaaS Plus, иногда – промежуточной версией между моделями PaaS и IaaS. Суть этого сервиса состоит в том, что клиент получает необходимую среду для разработки, тестирования, развертывания своих приложений, автоматизированную, интегрируемую с ведущими контейнерными решениями Kubernetes и Docker. EaaS-решения сокращают расходы бизнеса на облачные вычисления, оптимизируют использование инфраструктуры и повышают производительность.

Backend-as-a-Service: Бэкенд как услуга

Backend-as-a-Service и его мобильная вариация MBaaS – это облачная услуга, предоставляющая веб-разработчикам всю необходимую среду и компоненты, в том числе облачные хранилища, базы данных, API.

EaaS – Everything-as-a-Service: всё как услуга

Комплексное инфраструктурное решение с названием, которое носит, скорее, маркетинговый характер. EaaS в трактовке “всё-как-услуга” предлагают клиентам полную трансформацию существующей инфраструктуры предприятия от концепции on-premises к концепции cloud solutions. Это решение предусматривает радикальные изменения в стратегии управления компанией на всех уровнях, бизнес-процессах, подходах и методологии управления производством и т.д. В него включают подбор конфигурации, настройку, внедрение, миграцию с локальной платформы, приобретение и установку всего необходимого ПО (включая лицензии), поддержку, необходимые обновления системы и ее компонентов и другие параметры. Фактически это проект по переходу от локального размещения к гибкой облачной архитектуре, который выполнен “под ключ”.

MaaS – Metal-as-a-Service: выделенный сервер как услуга

MaaS – это решение для оркестровки серверов. Оно позволяет рассматривать физические серверы как виртуальные машины (инстансы) в облаке. То есть вместо того, чтобы управлять каждым сервером по отдельности, MaaS объединяет серверы в единый гибкий облачный пул ресурсов.

CaaS – Communication-as-a-Service: коммуникации как услуга

Это комплексное SaaS-решение для обеспечения услуг связи в компании, включающее интернет-телефонию, видеосвязь, чаты и мессенджеры, интерфейсы для совместной работы с документами и другие инструменты коммуникаций. Приобретение такого программного комплекса позволяет значительно экономить на инфраструктуре и ПО, при этом можно масштабировать мощности, как это обычно происходит в облачных решениях.

EaaS – Encryption-as-a-Service: шифрование как услуга

Пользователи, которым важно сохранять свою конфиденциальную информацию, принимают меры для ее защиты. Надежные и серьезные провайдеры облачных услуг обязательно предусматривают комплексные меры для защиты информации своих клиентов.

Информационная безопасность данных и их носителей обеспечивается на трех уровнях:

1. Технологическом: данные передаются по каналам связи, защищенных криптографическими протоколами, а для безопасности данных в хранилище используется аппаратное шифрование.
2. На уровне дата-центров: физический доступ к серверам с информацией ограничивается обязательными мерами, прописанными критериями сертификации Tier III+.
3. Законодательном: расположение на территории Евросоюза и в юрисдикции ФРГ дополнительно защищает данные от противоправного изъятия.

Можно заказать решение Encryption-as-a-Service у разработчика таких услуг. Такие продукты обычно легко интегрируются в облачную среду разных поставщиков, хотя есть и такие, которые зависят от платформы, где развернута облачная инфраструктура.

CaaS – Content-as-a-Service: контент как услуга

Иногда этот сервис называют еще MCaaS – managed content as a service, услуга управления контентом. Это настоящее спасение для маркетологов, которым приходится генерировать, адаптировать, оптимизировать разнообразный контент для самых разных каналов доставки потребителям. CaaS – сервис, который позволяет корпоративным пользователям размещать, управлять и структурировать контент на различных коммуникационных платформах, веб- и мультимедийных приложениях.

3.1 Дата-центры

Дата-центр *(от англ. data center), или* ***центр (хранения и) обработки данных*** *(ЦОД/ЦХОД) — это специализированное здание для размещения (хостинга) серверного и сетевого оборудования и подключения абонентов к каналам сети Интернет.*

Дата-центр исполняет функции обработки, хранения и распространения информации, как правило, в интересах корпоративных клиентов — он ориентирован на решение бизнес-задач путём предоставления информационных услуг. Консолидация вычислительных ресурсов и средств хранения данных в ЦОД позволяет сократить совокупную стоимость владения IT-инфраструктурой за счёт возможности эффективного использования технических средств, например, перераспределения нагрузок, а также за счёт сокращения расходов на администрирование.

Дата-центры обычно расположены в пределах или в непосредственной близости от узла связи или точки присутствия какого-либо одного или нескольких операторов связи. Основным критерием оценки качества работы любого дата-центра является время доступности сервера (аптайм).

Типичный датацентр состоит из:

- информационной инфраструктуры, включающей в себя серверное оборудование и обеспечивающей основные функции дата-центра — обработку и хранение информации;
- телекоммуникационной инфраструктуры, обеспечивающей взаимосвязь элементов дата-центра, а также передачу данных между дата-центром и пользователями;
- инженерной инфраструктуры, обеспечивающей нормальное функционирование основных систем дата-центра.

Инженерная инфраструктура включает в себя:

- кондиционирование для поддержания температуры и уровня влажности в заданных параметрах;
- бесперебойное электроснабжение для автономной работы дата-центра в случаях отключения центральных источников электроэнергии;
- охранно-пожарную сигнализацию и систему газового пожаротушения;
- системы удаленного IP-контроля, управления питанием и контроля доступа.

Некоторые дата-центры предлагают клиентам дополнительные услуги по использованию оборудования по автоматическому уходу от различных видов атак. Команды квалифицированных специалистов круглосуточно производят мониторинг всех серверов.

Необходимо отметить, что услуги дата-центров сильно отличаются в цене и количестве услуг. Для обеспечения сохранности данных используются системы резервного копирования. Для предотвращения кражи данных в дата-центрах используются различные системы ограничения физического доступа, системы видеонаблюдения.

В корпоративных (ведомственных) дата-центрах обычно сосредоточено большинство серверов соответствующей организации. Оборудование крепится в специализированных стойках и шкафах. Как правило, в дата-центр принимают для размещения лишь оборудование в стоечном исполнении, то есть в корпусах стандартных размеров, приспособленных для крепления в стойку.

Компьютеры в корпусах настольного исполнения неудобны для дата-центров и размещаются в них редко. На рисунке 3.2 отображены примеры таких стоек.

Рисунок 3.2 - Пример стоек размещения в ЦОД

ЦОД может располагаться и в довольно экзотических местах, например, под водой. Так по информации от NewAtlas в рамках Project Natick Microsoft разместила дата-центр в холодных водах у берегов Оркнейских островов, Шотландия. Это уже второй этап проекта: в ходе первой фазы в 2015 году на 105 дней погрузила в океан у побережья Калифорнии 3-метровый прототип дата-центра, таким образом, была подтверждена осуществимость концепта. Известно, что охлаждение, предотвращение перегревания компьютеров — это серьезная проблема в случае с крупными дата-центрами и компании решают ее, размещая такие центры в местах с холодным климатом — например, в прошлом году стало известно, что компания Kolos строит дата-центр за полярным кругом. На рисунке показано как сотрудники Microsoft занимаются погружением соответствующего датацентра в воде (рисунок 3.3).

Рисунок 3.3 – Пример подводного датацентра.

3.1.1 Стандарты оценки качества ЦОД

Кроме того, датацентры могут различаться по стандартам на оборудование помещений и оценке их качества.

В ряде стран имеются стандарты на оборудование помещений дата-центров, позволяющие объективно оценить способность дата-центра обеспечить тот или иной уровень сервиса. Например, в США принят американский (ANSI) стандарт TIA-942, несущий в себе рекомендации по созданию дата-центров и делящий дата-центры на типы по степени надёжности. В России не было такого же стандарта, дата-центры оснащались согласно требованиям для сооружений связи, а также ориентировались на требования TIA-942 и использовали дополнительную документацию Uptime Institute и ГОСТы серии 34.

В настоящее время Приказами Росстандарта утверждены следующие ГОСТ: ГОСТ Р 58811-2020 "Центры обработки данных. Инженерная инфраструктура. Стадии создания" регулирует стадии создания инженерной инфраструктуры центров обработки данных, этапы внутри стадий и содержание работ на каждом этапе; ГОСТ Р 58812-2020 "Центры обработки данных. Инженерная инфраструктура. Операционная модель эксплуатации. Спецификация" определяет требования к организации эксплуатации инженерных систем ЦОДов с целью обеспечения надлежащего качества их функционирования и

предоставления потребителям услуг ИИ ЦОДа приемлемого качества. Данный стандарт начал действовать 01 августа 2020 и опубликован в конце марта 2020 г.

TIA-942 воспринимается во всем мире как единый стандарт для дата-центров, однако следует отметить, что он достаточно давно не обновлялся и его достаточно сложно применить в условиях России. Активно развивается стандарт BICSI 002 2010 Data Center Design and Implementation Best Practices, появившийся в 2010 и обновленный в 2011. По словам создателей стандарта, «стандарт BICSI 002 2010, в создании которого участвовали более 150 экспертов, дополняет существующие стандарты TIA, CENELEC и ISO/IEC для центров обработки данных».

Для оценки качества инженерной инфраструктуры ЦОДов используются актуальные версии пяти групп стандартов в порядке убывания популярности, имеющих собственные концепции:

- Uptime's International Tier Standard, объединяющий в себе Data Center Site Infrastructure Tier Standard: Topology и Data Center Site Infrastructure Tier Standard: Operational Sustainability.
- ANSI/TIA-942 Telecommunications Infrastructure Standard for Data Centers и сопряженные с ним стандарты ANSI/EIA/TIA-568C Commercial Building Telecommunications Cabling Standard (February 2009), ANSI/EIA/TIA-606-A-1 The Administration Standard for the Telecommunications Infrastructure of Commercial Buildings, TIA/EIA-569 Commercial Building Standard for Telecommunication Pathways and Spaces и ANSI/TIA/EIA-607 Commercial Building Grounding and Bonding Requirements for Telecommunications.
- ANSI/BICSI 002 Data Center Design and Implementation Best Practices.
- ISO/IEC 24762 Information technology – Security techniques – Guidelines for information and communications technology disaster recovery services.
- SS 507 Singapore Standard for ICT Disaster Recovery Services.

Далее в таблице 3.2 приводятся некоторые характеристики стандартов.

Таблица 3.2 – Характеристики стандартов качества ЦОД

Стандарт	Uptime's International Tier Standard	ANSI/TIA-942-A	ANSI/BICSI 002-2014-v5
Обозначение уровней	Tier (I, II, III, IV)	Rating (1,2,3,4)	ClassF (0,1,2,3,4,5)
Принцип классифи-кации	Каждый Tier соотносится с определенной коммерческой функцией, эксплуатационными процедурами и рисками; устанавливает соответствующие требования к электропитанию, охлаждению, техническому обслуживанию и безотказности работы. Каждый следующий уровень сертификации включает в себя требования для всех предыдущих уровней.	Для каждого Rating устанавливаются технические и технологические требования к резервированию и доступности	Пять классов готовности ЦОДа определяются по: резервированию компонентов, резервированию систем, использованию продуктов с определенным уровнем качества, мерам противодействия любым внешним воздействиям.
Объём, стоимость, доступность	Topology – 12 с. Operational Sustainability – 15 с. Доступны на сайте Uptime Institute бесплатно	144 с. Доступен на сайте TIA за $521	500 с. Доступен на сайте BICSI за $575
Статус	Де-факто наиболее ранний и авторитетный общеотраслевой эталон классификации ЦОДов по уровням надежности	Наиболее популярный сборник правил по проектированию ЦОДов	Дополняет стандарты TIA-942, CENELEC и ISO/IEC

Стандарт BICSI 002 2010, в создании которого участвовали более 150 экспертов, дополняет существующие стандарты TIA, CENELEC и ISO/IEC для центров обработки данных.

При выборе площадки для ЦОД рекомендуется учитывать не только наличие необходимых для его работы энергетических ресурсов. Важную роль играют также транспортная доступность, от которой зависит организация снабжения топливом дизель-генераторов в условиях длительных чрезвычайных ситуаций, сейсмическая активность в данной местности, а также ряд других факторов.

Глава стандарта, посвященная безопасности, содержит данные, регламентирующие комплекс мероприятий, осуществляемых на основе анализа возможных рисков (эти меры необходимы для организации физической за-

щиты ЦОД, формирования планов противодействия потенциальным злоумышленникам и выхода из аварийных ситуаций), а также требования к обеспечению безопасности на всех этапах проектирования и строительства.

Модель, предложенная BICSI, определяет пять классов готовности ЦОД на основе четырех критериев:

- резервирования компонентов,
- резервирования систем,
- использования аппаратуры с определенным уровнем качества,
- мер противодействия любым внешним воздействиям, включая природные явления.

Исходя из этой классификации, разработчики стандарта предложили требования и рекомендации для электротехнического оборудования, которые способны обеспечить необходимый уровень его надежности.

Стандарты BICSI, использующие опыт более 500 специалистов по всему миру во всех отраслях промышленности по всему миру, определяют текущие практики для всех систем со спектром ИКТ, включая передачу голоса, данных, IoT, ESS и FLS, IP-системы и центры обработки данных, независимо от того, среда передачи данных - на основе оптоволокна, меди или беспроводной связи.

Представлена в 103 странах мира. 17 000 членов организации в мире, 700 в Европе. Основная локализация: 5 регионов США, Канада и Европа.

На рисунке 3.4 отображено звездочками где используется данный стандарт.

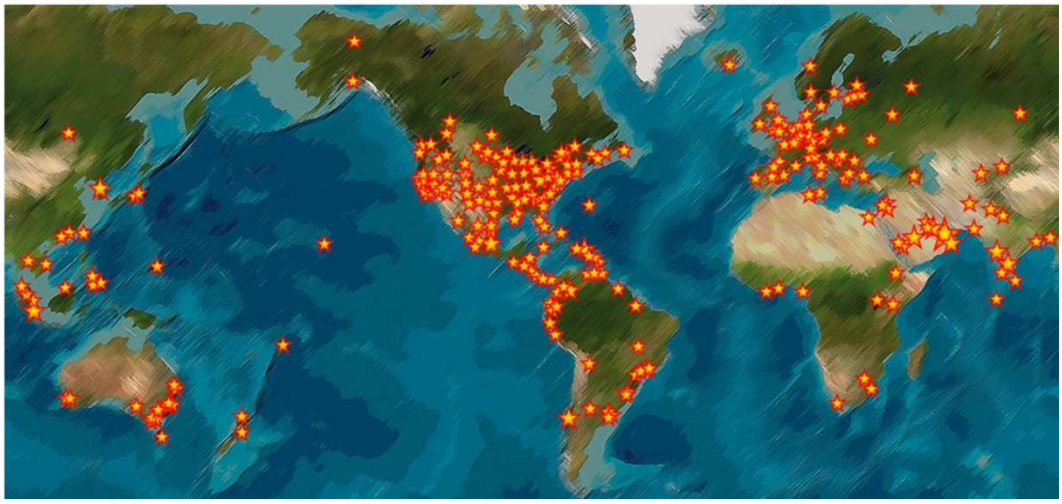


Рисунок 3.4 – Страны где используется стандарт BICSI для оценки качества ЦОД

Около четверти объема стандарта посвящено электротехническому оборудованию, включая организацию фидеров, распределительных устройств, систем бесперебойного электроснабжения, использование систем постоянного тока, заземления, средств подавления скачков напряжения, методики тестирования устройств.

В телекоммуникационной части документа BICSI, в дополнение к стандартам наподобие ANSI/TIA/EIA-942 Telecommunications Infrastructure Standard for Data Centers, рассматриваются вопросы формирования демаркационных точек с системами сервис-провайдеров, координация взаимодействия с этими организациями, требования к проектированию различных типов помещений для установки телекоммуникационного оборудования, организация кабельных трасс и ряд других проблем.

Раздел, посвященный информационным технологиям, содержит сведения о планировке помещений и размещении оборудования, кабельных системах, мерах по обеспечению заданного уровня готовности ЦОД. В состав последних входят, в частности, рекомендации по резервированию, включая создание и размещение резервных центров обработки данных.

Важной частью нового стандарта, являются рекомендации по организации комплекса мероприятий, необходимых для ввода в действие ЦОД и его последующего технического обслуживания.

3.1.2 Уровни надежности ЦОД

Основной показатель работы ЦОД — отказоустойчивость, также важна стоимость эксплуатации, показатели энергопотребления и регулирования температурного режима.

Например, стандарт TIA-942 предполагает четыре уровня надёжности дата-центров:

- Уровень 1 (N) — отказы оборудования или проведение ремонтных работ приводят к остановке работы всего дата-центра; в дата-центре отсутствуют фальшполы, резервные источники электроснабжения и источники бесперебойного питания; инженерная инфраструктура не зарезервирована;
- Уровень 2 (N+1) — имеется небольшой уровень резервирования; в дата-центре имеются фальшполы и резервные источники электроснабжения, однако проведение ремонтных работ также вызывает остановку работы дата-центра;
- Уровень 3 (2N) — имеется возможность проведения ремонтных работ (включая замену компонентов системы, добавление и удаление вышедшего из строя оборудования) без остановки работы дата-центра; инженерные системы однократно зарезервированы, имеется несколько каналов распределения электропитания и охлаждения, однако постоянно активен только один из них;
- Уровень 4 (2(N+1)) — имеется возможность проведения любых работ без остановки работы дата-центра; инженерные системы двукратно зарезервированы, то есть продублированы как основная, так и дополнительная системы (например, бесперебойное питание пред-

ставлено двумя ИБП, каждый из которых уже зарезервирован по схеме N+1).

3.1.3 Услуги которые предоставляет типичный дата-центр.

Виртуальный хостинг. Крупные дата-центры обычно не предоставляют подобную массовую услугу из-за необходимости обеспечения технического-консультационной поддержки.

Виртуальный сервер. Предоставление гарантированной и лимитированной части сервера (части всех ресурсов). Важная особенность данного вида хостинга — разделение сервера на несколько виртуальных независимых серверов, реализуемых программным способом.

Выделенный сервер. Дата-центр предоставляет клиенту в аренду сервер в различной конфигурации. Крупные дата-центры в основном специализируются именно на подобных типах услуг.

Colocation. Размещение сервера клиента на площадке дата-центра за определённую плату. Стоимость зависит от энергопотребления и тепловыделения размещаемого оборудования, пропускной способности подключаемого к оборудованию канала передачи данных, а также размера и веса стойки.

Аренда телекоммуникационных стоек. Передача клиенту стоек для монтажа собственного или клиентского оборудования. Формально это частный случай colocation, но с основным отличием в том, что арендаторы в основном юридические лица.

Выделенная зона (Dedicated area). В некоторых случаях владельцы дата-центра выделяют часть технологических площадей для специальных клиентов, как правило, финансовых компаний, имеющих строгие внутренние нормы безопасности. В этом случае дата-центр предоставляет некую выделенную зону, обеспеченную каналами связи, электроснабжением, холодоснабжением и системами безопасности, а клиент сам создает свой дата-центр внутри этого пространства.

3.1.4 Colocation.

. . . . . .

Колокейшн-центр *(также обозначается как co-location или colo) - это тип центра обработки данных (ЦОД), где оборудование, помещения и пропускная способность могут арендоваться клиентами, в том числе для размещения своего оборудования.*

. . . . . .

Колокейшн-центр обеспечивает размещение, электропитание, охлаждение и физическую безопасность для серверов, хранилищ и сетевого оборудования других фирм, дает возможность их подключения к различным поставщикам телекоммуникационных и сетевых услуг с минимальными затратами.

Многие Colocation центры предоставляют услуги широкому кругу клиентов, от крупных предприятий до небольших компаний. Обычно серверное оборудование принадлежит заказчику, а Colocation обеспечивает электроэнергию и охлаждение. Клиенты сохраняют контроль над своим оборудованием, но управление центром обработки данных и оборудованием ЦОД контролируется поставщиком услуг. В это оборудование входят Шкафы, Клетки,

Шкаф (Cabinet) - это блочный элемент, в котором находится серверная стойка. Как и лего, шкафы - это строительные блоки центров обработки данных. Они представляют собой закрытые конструкции, которые могут содержать несколько модулей для электронного оборудования с направляющими сделанными из металла, заземляющими элементами и др. Компания может использовать часть шкафа или весь шкаф в зависимости от своих потребностей.

Клетки (Cage) - это выделенное серверное пространство в традиционном центре обработки данных с фальшполом, корпус из металлического каркаса, в котором можно разместить несколько шкафов в центре обработки данных. Эти клетки могут быть общими или частными, они могут состоять

из металлических стержней и сетчатых стенок для максимальной эффективности охлаждения. Клетки обеспечивают полную видимость того, что находится внутри, с дверью для входа. В частных клетках содержатся серверы и оборудование только одной компании, а общие клетки могут использоваться для поддержки инфраструктуры нескольких компаний.

Private Suites (Аппартаменты) - это выделенное частное серверное пространство в традиционном дата-центре на фальшполе; он полностью закрыт прочными перегородками и туда можно попасть только через запирающуюся дверь. Если предприятиям нужна максимальная конфиденциальность для своей инфраструктуры, частные апартаменты - идеальное решение. Эти апартаменты могут содержать несколько шкафов (клетки), но они будут ограждены стенами со всех сторон. В отличие от предыдущих вариантов, частные апартаменты предлагают большую площадь и индивидуальные настройки в зависимости от предпочтений компании и даже могут включать другие удобства, такие как офисные помещения и дополнительные меры безопасности.

Здания с центрами обработки данных часто легко узнать по количеству охлаждающего оборудования, расположенного снаружи или на крыше.

Часто наличествуют системы противопожарной защиты, включая пассивные и активные детекторы. Детекторы дыма обычно устанавливаются для раннего предупреждения о развивающемся пожаре путем обнаружения частиц, образовавшихся от тлеющих компонентов, до возникновения пламени. Это позволяет найти источник тления, заранее отключить электроэнергию, вручную потушить пожар с помощью огнетушителей до того, как пожар разрастется до значительных размеров. Разбрызгивающая система пожаротушения часто используется для борьбы с полномасштабным пожаром, если он развивается. Пассивные элементы противопожарной защиты обычно представляют собой противопожарные перегородки в помещении, так что пожар может быть локализован на какое-то время в случае отказа активных систем противопожарной защиты.

Центры обработки данных Colocation часто проходят аудит, чтобы доказать, что они соответствуют определенным стандартам и уровням надежности; наиболее часто встречающимися системами являются SSAE 16 SOC 1 Type I и Type II (ранее SAS 70 Type I и Type II) и многоуровневая система, разработанная Uptime Institute или TIA. Другие стандарты соответствия центров обработки данных включают аудит по Закону о переносимости и подотчетности медицинского страхования (HIPAA) и стандарты PCI DSS.

На объектах Colocation обычно есть генераторы, которые запускаются автоматически при отключении электроэнергии и работающие на дизельном топливе. Эти генераторы могут иметь разные уровни резервирования в зависимости от того, как построено предприятие. Генераторы не могут запуститься мгновенно, поэтому обычно присутствуют системы резервного питания от батарей.

Некоторые клиенты предпочитают использовать оборудование, которое питается напрямую от аккумуляторных батарей 48В. Это может обеспечить лучшую энергоэффективность и уменьшить количество блоков, которые могут выйти из строя, хотя пониженное напряжение значительно увеличивает необходимый ток и, следовательно, размер (и стоимость) проводки для подачи энергии. Альтернативой батареям является мотор-генератор, подключенный к маховику и дизельному двигателю.

Объекты колокации для дополнительной надежности иногда подключаются к нескольким частям энергосистемы.

Владельцы объектов Colocation могут придерживаться разных правил в отношении перекрестных соединений между своими клиентами, некоторые из которых могут быть операторами связи. Эти правила могут позволять клиентам запускать такие подключения бесплатно или за ежемесячную плату. Могут позволять клиентам заказывать кросс-подключения к операторам связи, но не к другим клиентам. В некоторых центрах размещения есть «комната для встреч», где различные операторы связи, размещенные в центре, могут эффективно обмениваться данными.

Объекты Colocation, которые позволяют устанавливать соединение между клиентами без ограничений, обычно называют центрами обработки данных, нейтральными к операторам связи.

3.2 Типы вычислительных систем, размещаемых в ЦОД

Основные вычислительные системы на базе которых реализуется создание веб-сервисов:

- Кластер (узлы-процессора со своей памятью объединенные между собой высокоскоростной сетью через коммуникационное оборудование);
- Распределенная система (Grid) (любые узлы соединены любыми сетевыми технологиями);
- Суперкомпьютер (условно, процессоры имеют общую память или доступ к памяти других процессоров используя внутренние соединения).

3.2.1 Кластер

Вычислительный кластер *– это совокупность вычислительных узлов, объединенных высокоскоростными каналами связи, представляющая собой с точки зрения пользователя единый вычислительный (аппаратный) ресурс.*

По типу архитектуры кластер относится к системам с распределенной памятью («память распределена по узлам»), при этом каждый узел кластера в отдельности представляет собой систему с общей (разделяемой) памятью.

Кластер имеет одну общую систему питания и охлаждения и управления.

Кластерные системы занимают достойное место в списке самых высокопроизводительных систем, при этом значительно выигрывая у суперкомпьютеров в цене. На июль 2008 года на 7 месте рейтинга TOP500 находится кластер SGI Altix ICE 8200 (Chippewa Falls, Висконсин, США).

Различают следующие виды кластеров:

- отказоустойчивые кластеры (High-availability clusters, HA, кластеры высокой доступности);
- кластеры с балансировкой нагрузки (Load balancing clusters);
- вычислительные кластеры (High performance computing clusters, HPC);
- системы распределенных вычислений.

3.2.1.1 Кластеры высокой доступности.

Обозначаются аббревиатурой HA (англ. High Availability — высокая доступность). Создаются для обеспечения высокой доступности сервиса, предоставляемого кластером. Избыточное число узлов, входящих в кластер, гарантирует предоставление сервиса в случае отказа одного или нескольких серверов. Типичное число узлов — два, то минимальное количество, которое обеспечивает повышение доступности.

Отказоустойчивые кластеры и системы разделяются на 3 основных типа:

- с холодным резервом или активный/пассивный. Активный узел выполняет запросы, а пассивный ждет его отказа и включается в работу, когда происходит отказ. Примером могут служить резервные сетевые соединения, например, FDDI, когда во время отказа одного кольца, включается резервное кольцо.
- с горячим резервом или активный/активный. Все узлы выполняют запросы, в случае отказа одного нагрузка перераспределяется между оставшимися. То есть кластер распределения нагрузки с поддержкой перераспределения запросов при отказе. Примеры — практически все кластерные технологии, например, Microsoft Cluster Server. OpenSource проект OpenMosix.
- с модульной избыточностью. Применяется только в случае, когда простой системы совершенно недопустим. Все узлы одновременно выполняют один и тот же запрос (либо части его, но так, что результат достижим и при отказе любого узла), из результатов берется любой. Необходимо гаран-

тировать, что результаты разных узлов всегда будут одинаковы (либо различия гарантированно не повлияют на дальнейшую работу). Примеры — RAID и Triple modular redundancy.

3.2.1.2 Кластеры распределения нагрузки (Network Load Balancing, NLB)

Принцип их действия строится на распределении запросов через один или несколько входных узлов, которые перенаправляют их на обработку в остальные, вычислительные узлы.

Первоначальная цель такого кластера — производительность, однако, в них часто используются также и методы, повышающие надёжность. Подобные конструкции называются серверными фермами.

Программное обеспечение (ПО) для подобных систем может быть как коммерческим (OpenVMS, MOSIX, Platform LSF HPC, Solaris Cluster, Moab Cluster Suite, Maui Cluster Scheduler), так и бесплатным (OpenMosix, Sun Grid Engine, Linux Virtual Server).

3.2.1.3 Вычислительные кластеры

Кластеры используются в вычислительных целях, в частности в научных исследованиях.

Для вычислительных кластеров существенными показателями являются высокая производительность процессора в операциях над числами с плавающей точкой (flops) и низкая латентность (задержка реакции) объединяющей сети, и менее существенными — скорость операций ввода-вывода, которая в большей степени важна для баз данных и web-сервисов.

Вычислительные кластеры позволяют уменьшить время расчетов, по сравнению с одиночным компьютером, разбивая задание на параллельно выполняющиеся ветки, которые обмениваются данными по связывающей сети.

Одна из типичных конфигураций — набор компьютеров, собранных из общедоступных компонентов, с установленной на них операционной системой Linux, и связанных сетью Ethernet, Myrinet, InfiniBand или другими отно-

сительно недорогими сетями. Такую систему принято называть кластером Beowulf.

Специально выделяют высокопроизводительные кластеры (Обозначаются англ. аббревиатурой HPC Cluster — High-performance computing cluster). Список самых мощных высокопроизводительных компьютеров (также может обозначаться англ. аббревиатурой HPC) можно найти в мировом рейтинге TOP500. В России ведется рейтинг самых мощных компьютеров СНГ.

3.2.1.4 Устройство и состав кластера

Чаще всего кластер имеет как минимум одну головную ноду (head node) и отдельные узлы для файловой системы. Каждый узел (computation node, нода) представляет собой или SMP или NUMA систему (многопроцессорные системы с разной организацией по удаленности к памяти, о них речь пойдет далее). Так строятся практически все современные суперкомпьютеры, при этом наблюдается тенденция к увеличению количества процессоров в одной ноде. Между собой ноды связаны высокоскоростной локальной сетью, обычно применяется Gigabit Ethernet (GigE) или более быстрые сети Infiniband (Mellanox, QLogic), Myrinet или другие проприетарные способы сетевого взаимодействия.

Программное обеспечение кластера должно обеспечивать общий диск между узлами (shared space) и службу удаленного запуска приложений (это может быть telnet, ssh, rshell и т.п.). От размера кластера главным образом зависит то, каким будет выбранная топология его сети. Небольшие по размеру кластера обычно строятся на одном-двух коммутаторах (свичах). Для связывания же больших кластеров коммутаторы объединяются в несколько уровней.

В случае объединения двух стоек, можно воспользоваться одним из портов свича для создания связи свич-свич. При таком варианте обмен между стойками проходит через одну пару портов, что может приводить к замедлению взаимодействия, особенно коллективного (рисунок 3.5).

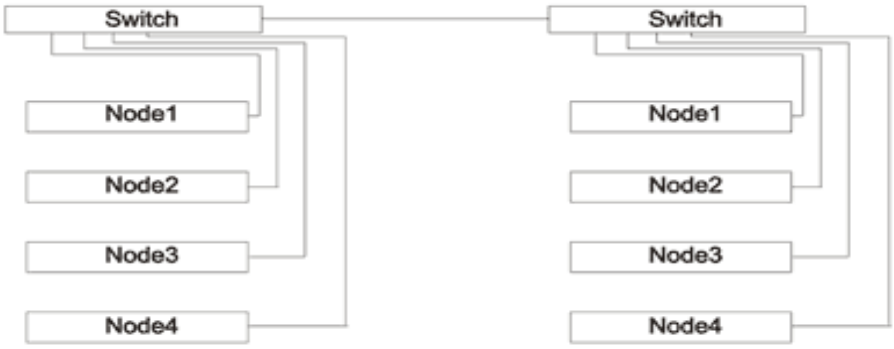


Рисунок 3.5 – Пример объединения нод в кластере с двумя свичами

Дальнейший рост сложности сети приводит к многоуровневым схемам построения коммутаторов (рисунок 3.6).

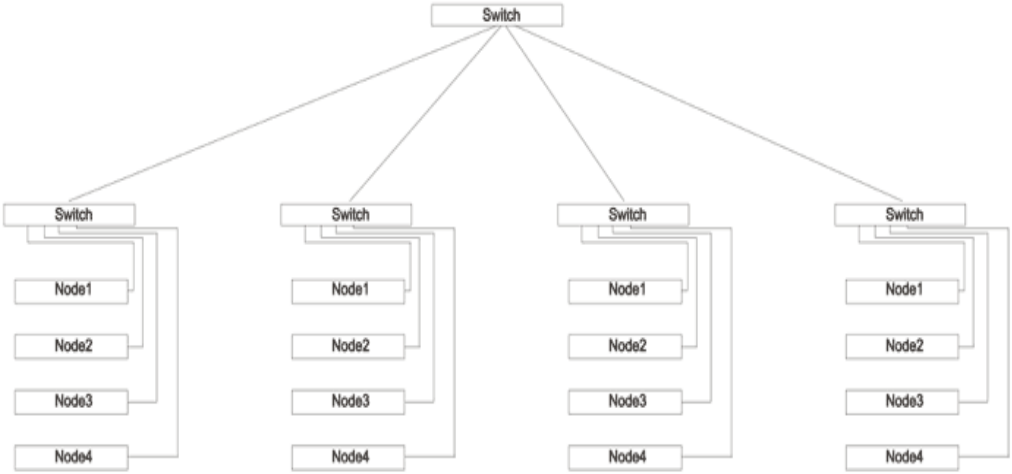


Рисунок 3.6 - Многоуровненвая схема объединения нод в кластере через коммутаторы.

Приведем типовое использование кластера через ssh.

Пользователь получает доступ к ВМ по SSH: SSH <user>@<address> -X. ("-X" для запуска графических приложений), далее можно использовать команды Linux.

Он имеет root привилегии для своей ВМ и может устанавливать и удалять программы, настраивать систему и т.д.

На виртуальную машину могут быть установлены и другие ОС кроме Linux.

Получив ВМ, пользователь может запускать свои задачи на ней.

Он сможет устанавливать требуемое ПО на ВМ.

3.2.2 Супер-компьютер

Суперкомпью́тер *(англ. Supercomputer, СверхЭВМ, Супер-ЭВМ, сверхвычисли́тель) — специализированная вычислительная машина, значительно превосходящая по своим техническим параметрам и скорости вычислений большинство существующих в мире компьютеров.*

Определение очень нечеткое и общее, потому порой вызывает разные шутки, например, что суперкомпьютер - это любой компьютер весящий больше тонны. Можно считать, что это объединение процессорных узлов через технологии обмена информацией отличные от стандартных технологий коммутации или через коммутаторы, как это делается в кластерах.

Начиная с 1993, суперкомпьютеры ранжируют в списке Top500. Список составляется на основе теста LINPACK по решению системы линейных алгебраических уравнений, являющейся общей задачей для численного моделирования.

Самым мощным суперкомпьютером в 2016 году по этому списку стал Sunway TaihuLight, работающий в национальном суперкомпьютерном центре Китая. Скорость вычислений, производимых им, составляет 93 петафлопс (10 в 15 степени вычислительных операций с плавающей запятой в секунду). По этому показателю он в два раза быстрее и в три раза эффективнее предыдущего рекордсмена — Tianhe-2, также разработанного в Китае и возглавлявшему список с 2013 года.

Типичные архитектуры суперкомпьютеров это NUMA и SMP.

В SMP (Symmetric Multi-Processing, симметричные процессорные системы) все процессоры имеют равноправный доступ к памяти. Память равноудалена от всех процессоров (рисунок 3.7).

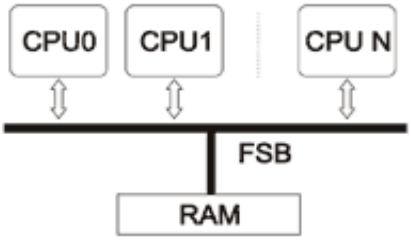


Рисунок 3.7 – Архитектура SMP

Как видно из иллюстрации все процессоры связаны общей памятью через FSB (Front Side Bus).

Эта же шина и является узким местом такой архитектуры, поскольку ее пропускная способность должна удовлетворять запросы каждого процессора, даже если они поступают одновременно.

Именно поэтому SMP системы почти не масштабируются, два-три десятка процессоров для них – это уже теоретический предел.

В NUMA (non-uniform memory access, системы с неодинаковым доступом к памяти или Non-Uniform Memory Architecture — «Архитектура с неравномерной памятью») каждый процессор имеет свою локальную память и более затратный доступ к памяти других процессоров (рисунок 3.8). Время доступа к памяти определяется её расположением по отношению к процессору.

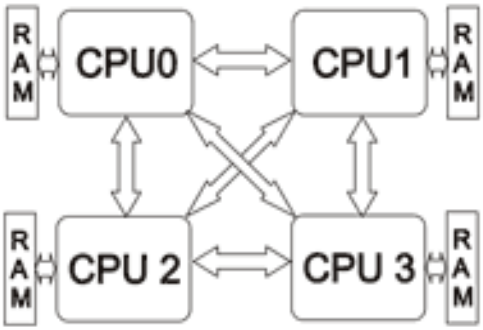


Рисунок 3.8 - Архитектура NUMA

В NUMA системах каждый процессор имеет локальную память и при правильной привязке процессов к процессорам всегда используется «ближняя» память. Доступ же в дальнюю память, происходит только при взаимодействии процессов. Если привязки процессов нет, то в результате так называемой «миграции», процесс может быть запущен на другом процессоре и работать со своими данными из дальней памяти.

Когда процессору нужно выполнить операции чтения или записи, он посылает запрос с нужным адресом своему контроллеру памяти. Контроллер анализирует старшие разряды адреса, по которым и определяет, в каком модуле хранятся нужные данные. Если адрес локальный, то запрос выставляется на локальную шину, в противном случае запрос для удаленного кластера отправляется через межкластерную шину. В таком режиме программа, хранящаяся в одном модуле памяти, может выполняться любым процессором системы. Единственное различие заключается в скорости выполнения. Все локальные ссылки отрабатываются намного быстрее, чем удаленные. Поэтому и процессор того кластера, где хранится программа, выполнит ее на порядок быстрее, чем любой другой.

CC-NUMA *(cache coherent NUMA) - система с кэш-когерентным доступом к неоднородной памяти. [37]*

В отличие от классической архитектуры NUMA, при использовании кэш-когерентного доступа к неоднородной памяти все процессоры объединены в один узел, причем первый уровень иерархии памяти образует кэш-память процессоров. Архитектура ccNUMA поддерживает когерентность кэш памяти внутри узла аппаратно. Аппаратная когерентность кэш-памяти означает, что не требуется никакого программного обеспечения для поддержки актуальности множества копий данных. Напомним, что когерентность КЭШа или памяти - это обеспечение всех процессоров одинаковым содержанием памяти при ее изменении одним процессором. То есть, очевидно, что изме-

нение в КЭШе должно обеспечить изменения в локальной памяти и согласованность хранимых данных.

В системе cc-NUMA физически распределенная память объединяется, как в любой другой SMP-архитектуре, в единый массив. Не происходит никакого копирования страниц или данных между ячейками памяти. Нет никакой программно-реализованной передачи сообщений. Существует просто одна карта памяти, с частями, физически связанными медным кабелем, и очень умные (в большей степени, чем объединительная плата) аппаратные средства. Аппаратно-реализованная кэш-когерентность означает, что не требуется какого-либо программного обеспечения для сохранения множества копий обновленных данных или для передачи их между множеством экземпляров операционной системы и приложений. Со всем этим справляется аппаратный уровень точно так же, как в любом SMP-узле, с одной копией операционной системы и несколькими процессорами.

3.2.3 GRID-вычисления

Грид-вычисления *(англ. grid — решётка, сеть) — это форма распределённых вычислений, в которой «виртуальный суперкомпьютер» представлен в виде кластеров, соединённых с помощью сети, слабосвязанных гетерогенных компьютеров, работающих вместе для выполнения огромного количества заданий (операций, работ).*

Эта технология применяется для решения научных, математических задач, требующих значительных вычислительных ресурсов. Грид-вычисления используются также в коммерческих направлениях для решения таких трудоёмких задач, как экономическое прогнозирование, сейсмоанализ, разработка и изучение свойств новых лекарств. [38]

Грид с точки зрения сетевой организации представляет собой согласованную, открытую и стандартизованную среду, которая обеспечивает гибкое, безопасное, скоординированное разделение вычислительных ресурсов и ресурсов хранения информации, которые являются частью этой среды, в рамках одной виртуальной организации.

Грид-вычисления можно создать на базе множества устаревших моделей персональных компьютеров, объединённых в иерархическую локальную вычислительную сеть (например, Ethernet и др.) с присутствием серверов. Эта сеть может иметь соединение с интернетом.

Грид является географически распределённой инфраструктурой, объединяющей множество ресурсов разных типов (процессоры, долговременная и оперативная память, хранилища и базы данных, сети), доступ к которым пользователь может получить из любой точки, независимо от места их расположения.

Идея грид-компьютинга возникла вместе с распространением персональных компьютеров, развитием интернета и технологий пакетной передачи данных на основе оптического волокна (SONET, SDH и ATM), а также технологий локальных сетей (Gigabit Ethernet). Полоса пропускания коммуникационных средств стала достаточной, чтобы при необходимости привлечь ресурсы другого компьютера. Учитывая, что множество подключенных к глобальной сети компьютеров большую часть рабочего времени простаивает и располагает большими ресурсами, чем необходимо для решения их повседневных задач, возникает возможность применить их неиспользуемые ресурсы. На 2016 год разработаны стандарты и уже в продаже коммутаторы 10Gbit, 25Gbit Ethernet, а в 2020 году объявлены спецификации на 800Gbit Ethernet.

В настоящее время выделяют три основных типа грид-систем:

- Добровольные гриды — гриды на основе использования добровольно предоставляемого свободного ресурса персональных компьютеров; Например, для технологий криптовалют, часто пользователи сами выделяют

ресурсы своей видеокарты или процессора, для решения простой задачи расчета хэш сумм, хотя за это они получают некие суммы криптовалюты. В настоящее время, так называемый майнинг на своем компьютере представляет собой альтруистическую задачу, где человек больше расходует на электричество.

- Научные гриды — хорошо распараллеливаемые приложения программируются специальным образом (например, с использованием Globus Toolkit);
- Гриды на основе выделения вычислительных ресурсов по требованию (коммерческий грид, англ. enterprise grid) — обычные коммерческие приложения работают на виртуальном компьютере, который, в свою очередь, состоит из нескольких физических компьютеров, объединённых с помощью грид-технологий.

3.3 Виртуализация

Для того чтобы обеспечить возможность абстрагироваться от конкретных аппаратных возможностей и иметь высокую степень управления вычислительными ресурсами чаще всего применяют виртуализацию: предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг от друга вычислительных процессов, выполняемых на одном физическом ресурсе.

Примером использования виртуализации является возможность запуска нескольких так называемых гостевых операционных систем на одном компьютере: при этом каждый из экземпляров таких гостевых операционных систем работает со своим набором логических ресурсов (процессорных, оперативной памяти, устройств хранения), предоставлением которых из общего пула, доступного на уровне оборудования, управляет хостовая операционная система — гипервизор. Также могут быть подвергнуты виртуализации сети

передачи данных, сети хранения данных, платформенное и прикладное программное обеспечение.

3.3.1 Миграция виртуальных машин (живая миграция)

Благодаря разнесению и отделению гостевой операционной системы от аппаратуры, практически возможно физически перемещать гостевые системы, но не только в моменты простоя или остановки, но и без остановки работы сервисов.

Живая миграция *(от англ. Live migration) — перенос виртуальной машины с одного физического сервера на другой без прекращения работы виртуальной машины и остановки сервисов; применяется в компьютерных системах с высокой доступностью (англ. High Availability, HA). Живая миграция возможна между серверами, находящимися в кластере.*

Живая миграция необходима в проектах, работу которых нежелательно прерывать: сервисы обслуживания магистральных и провайдерских сетей (например, DNS), высокопосещаемые web-ресурсы, крупные сервисы электронной почты.

Живая миграция также применяется для распределения нагрузки на физические серверы в кластере (например, при проведении научных расчётов).

3.3.2 Виды виртуализации. Виртуализация ресурсов.

Виртуализация ресурсов (или разделение ресурсов, англ. partitioning) может быть представлена как разделение одного физического узла на несколько частей, каждая из которых представлена для владельца в качестве отдельного сервера. Не является технологией виртуальных машин, реализуется на уровне ядра операционной системы. Иногда называется контейнейрной виртуализацией.

Например, в системах с гипервизором второго типа обе операционные системы (гостевая и гипервизора) отнимают физические ресурсы, и требуют отдельного лицензирования. В отличие от них виртуальные серверы, работающие на уровне ядра ОС, почти не теряют в быстродействии, что дает возможность запускать на одном физическом сервере сотни виртуальных, не требующих дополнительных лицензий.

Дисковое пространство или пропускной канал сети представляет собой разделенный ресурс. Например, к реализации разделения ресурсов можно отнести OpenSolaris Network Virtualization and Resource Control (Проект Crossbow), позволяющий создавать несколько виртуальных сетевых интерфейсов на основе одного физического.

Также применяется агрегация, распределение или добавление множества ресурсов в более крупные ресурсы или объединение ресурсов. Например, симметричные мультипроцессорные системы объединяют множество процессоров. RAID и дисковые менеджеры объединяют множество дисков в один большой логический диск. RAID и сетевое оборудование использует множество каналов, объединённых так, чтобы они представлялись, как единый широкополосный канал. На мета-уровне компьютерные кластеры делают все вышеперечисленное. Иногда сюда же относят сетевые файловые системы, абстрагированные от хранилищ данных на которых они построены, например, Vmware VMFS, Solaris/OpenSolaris ZFS, NetApp WAFL.

3.3.3 Виртуализация оборудования.

Эмуляция *— полная виртуализация (виртуализация всей платформы). Полная замена аналогами всех команд.*

Например, QEMU или эмуляторы игровых консолей.

QEMU — свободная программа с открытым исходным кодом для эмуляции аппаратного обеспечения различных платформ.

Включает в себя эмуляцию процессоров Intel x86 и устройств ввода-вывода. Может эмулировать 80386, 80486, Pentium, Pentium Pro, AMD64 и другие x86-совместимые процессоры; ARM, MIPS, RISC-V, PowerPC, SPARC, SPARC64 и частично m68k.

Помимо эмуляции, поддерживает технологии аппаратной виртуализации (Intel VT и AMD SVM) на x86-совместимых процессорах Intel и AMD (используется гипервизор KVM или Xen).

Пример запуска ISO образа:

qemu -m 512 -cdrom /путь/к_iso-образу/example.iso -boot d

3.3.4 Виртуализация памяти

Виртуализация памяти *(memory virtualization) — объединение оперативной памяти из различных ресурсов в единый массив.*

Реализации: Oracle Coherence, GigaSpaces XAP. Также это может быть расширение адресного пространства оперативной памяти за счет дискового хранилища.

Виртуальная память — изоляция адресного пространства приложения от всего адресного пространства. Применяется во всех современных ОС.

Идея виртуальной памяти возникла из желания выполнять программы большего размера чем ОЗУ. Вначале стали использовать оверлейную структуру программ, когда большая программа разбивается на отдельные модули перекрытия — оверлеи (overlays). Первый модуль загружается в память и работает. Если ему нужен код или данные, содержащиеся в других модулях, то работающий модуль вызывает функции, загружающие другие модули. При этом программист сам должен отследить: вызов функций загрузки, передачу управления вновь загруженным модулям, контроль суммарного объема загруженных модулей (объем не должен превышать имеющегося объема памя-

ти), размещение модулей в памяти, и другие подобные вопросы. Эта работа оказывается весьма трудоемкой. При этом отметим, что, поскольку вновь загружаемые модули "ложатся" в память на место находившихся там ранее, то разные объекты программы, находящиеся в разных модулях, но попадающие в одно и то же место физической памяти получают одни и те же физические адреса; расходуется время на перезагрузку модулей; фрагментируется память, поскольку модули имеют разные размеры; для работы в другой конфигурации памяти требуется перекомпоновка программы. Так же, например, в операционной системе MS-DOS применялась EMS память, где можно было отображать всю доступную память в сегмент 64 кб основной памяти, которая была не более 1 мб, в связи с использованием 20 битной адресной шины в реальном режиме работы процессора.

Как альтернатива оверлейной структуре, возникла концепция виртуальной памяти (1961 г, Манчестер). Фактически используется та же идеология перезагрузки модулей — свопинг (swapping), однако замена осуществляется аппаратурой и операционной системой автоматически, без какого-либо участия программиста и незаметно для него. При этом программист оперирует адресами в (линейном) виртуальном адресном пространстве задачи, размер которого не меньше размера программы (включающей как исполняемый код так и данные). Каждый объект программы в пространстве виртуальных адресов имеет уникальный адрес. Программа пишется в виртуальных адресах.

Технологии вытеснения также могут быть различными.

3.3.5 Виртуализация хранилищ данных

Виртуализация хранения данных, это объединение и представление набора физических носителей в виде единого физического носителя. Агрегирование множества физических устройств хранения данных (таких как JBOD, RAID, или ленточные накопители) с различными протоколами интерфейсов (таких как SCSI, iSCSI или Fibre Channel) в единый виртуальный пул хранения, из которого при необходимости можно производить создание и инициа-

лизацию (provisioning) виртуальных томов хранения. При этом для хост-сервера они представлены в виде локально подключенных логических устройств. Усовершенствованное решение виртуализации хранилищ данных предоставляет IT администраторам свободу выбора – обеспечивать доступные ресурсы хранения в виртуальном общем пуле в качестве томов SAN (сеть хранения данных) и/или NAS (сетевое хранилище). NAS представляет собой единое устройство хранения, которое обслуживает файлы по сети Ethernet и является относительно недорогим и простым в настройке устройством. Тогда как SAN представляет собой тесно связанную сеть из нескольких устройств, которые работают с данными на основе блоков, и является более дорогим и сложным в настройке и управлении устройством. Для пользователя NAS выглядят как тома на файловом сервере и используют протоколы, такие как NFS и SMB/CIFS, тогда как диски, подключенные к SAN, представляются пользователю как локальные диски. Поскольку SAN значительно сложнее и дороже, чем NAS, обычно он используется крупными корпорациями и требует администрирования со стороны ИТ-персонала. Для некоторых приложений, таких как редактирование видео, это особенно желательно из-за его высокой скорости и низкой задержки. Для редактирования видео требуется справедливое и приоритетное использование полосы пропускания в сети, что является преимуществом SAN.

Основным преимуществом сети SAN является то, что все согласования доступа к файлам осуществляются через Ethernet, в то время как файлы обслуживаются через чрезвычайно высокоскоростной Fibre Channel, что приводит к очень высокой производительности на клиентских рабочих станциях, даже для очень больших файлов. По этой причине SAN сегодня широко используется в средах совместного редактирования видео.

Когда администратор через централизованную консоль размещает ресурсы хранения из общего виртуального пула на сервер приложения, сервер принуждают воспринимать этот ресурс как реальное SCSI устройство хранения, физически подключенное к нему. И именно так этот ресурс отображает-

ся в списке устройств, подключенных к серверу, например, как новый диск F или G для сервера Windows, либо как новое mountable физическое устройство в Unix, Linux или Solaris. Виртуализация – это нечто, сродни щиту, который защищает сервер приложений от неблагоприятного влияния гетерогенного (если не полностью хаотического) мира хранения данных: массивов, смешанных дисков, контроллеров хранилищ, NIC, адаптеров шины узла (HBA), ленточных накопителей и т.д., работающих «за кулисами».

К виртуализации хранения не стоит относить маскирование файловых систем или возможность параллельного совместного использования файлов в нескольких платформах ОС, хотя некоторые производители и утверждают иное.

3.3.6 Виртуализация операционных систем

Это наиболее распространенная форма виртуализации. Она включает установку второй или нескольких экземпляров операционных систем, например, Windows, на одном компьютере, что дает предприятиям возможность уменьшить количество физического оборудования, необходимого для работы их программного обеспечения, за счет сокращения количества реальных машин. Также позволяет компаниям сэкономить на энергии, кабелях, оборудовании, пространстве в стойке и многом другом, при этом позволяя запускать такое же количество приложений.

Виды виртуализации ОС:

- Программная виртуализация
- Встроенная виртуализация
- Аппаратная виртуализация — виртуализация с поддержкой специальной процессорной архитектуры. В отличие от программной виртуализации, с помощью данной техники возможно использование изолированных гостевых систем, управляемых гипервизором напрямую.

- Виртуализация на уровне операционной системы: работа нескольких экземпляров пространства пользователя в рамках одной ОС. Контейнерная виртуализация. Примерами могут быть Docker, LXC.

3.3.6.1 Программная виртуализация

Классический процесс программной виртуализации подразумевает запуск платформы виртуализации поверх основной операционной системы. Именно эта платформа берет на себя работу по эмуляции аппаратных компонентов и управляет ресурсами в отношении гостевой системы.

Программная виртуализация достаточно сложна в реализации. Основным недостатком является существенные потери производительности, связанные с потреблением ресурсов основной системой.

Также следует отметить факт значительного снижения безопасности, ведь вследствие получения контроля над базовой операционной системой, автоматически перехватывается контроль над гостевыми системами.

В случае виртуализации программного обеспечения, хост-системе необходимо полностью эмулировать гостевую платформу (т. е. начиная с аппаратного обеспечения, инструкций процессора, его встроенного программного обеспечения). Преимущество заключается в том, что хостовая и гостевая платформы независимы. Недостатком является то, что этот подход очень медленный и требует много ресурсов (так как должно эмулироваться все).

Бывает несколько видов программной виртуализации.

Интерпретация или полная эмуляция *– метод программной виртуализации, при котором команды интерпретируются и преобразуются в ряд инструкций, воспринимаемых реальным процессором.*

В этом случае появляется возможность создавать виртуальные машины, имитирующие работу аппаратуры, не совместимой по архитектуре с ре-

альной ЭВМ, но данный метод является очень медленным. Например, можно запускать виртуальную машину, имитирующую работу процессора с RISC-архитектурой, на реальной ЭВМ с процессором CISC архитектуры. Это возможно за счет того, что эмуляция ведется на уровне базовых арифметико-логических инструкций, в том или ином виде, присутствующих, практически, в любом процессоре. Однако, интерпретация каждой инструкции приводит к значительному расходу ресурсов реальной ЭВМ и снижает быстродействие приложений, работающих в гостевой операционной системе.

Динамическая (бинарная) трансляция *- процесс, при котором проблемные команды гостевой ОС заменяются на безопасные. После того как эти команды заменяются на безопасные, происходит возврат управления гостевой системе.*

Бинарная трансляция исполняемого кода происходит перед его реальным исполнением. Простым языком у процессоров могут быть некоторые команды, которые при вызове в гостевой ОС будучи выполненными могут вызвать проблемы и для гипервизора, и для хостовой ОС. Типичным решением виртуализации является перехват гипервизором команд гостевой ОС с помощью прерываний, но некоторые команды могут быть даже не перехвачены или привести к сбою, потому эти проблемные команды заменяют.

Паравиртуализация *— процесс, при котором гостевые ОС модифицируют свое ядро с целью функционирования в виртуализированной среде.*

ОС взаимодействует с гипервизором, который обеспечивает гостевой API. Таким образом, исключается использование таблицы страниц памяти. Паравиртуализация гарантирует более высокую производительность в срав-

нении с динамической трансляцией, однако она уместна лишь тогда, когда гостевые ОС имеют открытые исходные коды, либо же гипервизор и гостевая ОС одного производителя. Термин сформировался в рамках проекта Denali.

Код, касающийся виртуализации, локализуется непосредственно в операционную систему. Паравиртуализация таким образом требует, чтобы гостевая операционная система была изменена для гипервизора, и это является недостатком метода, так как подобное изменение возможно лишь в случае, если гостевые ОС имеют открытые исходные коды, которые можно модифицировать согласно лицензии. Но зато паравиртуализация предлагает производительность почти как у реальной не виртуализированной системы. Как и при полной виртуализации, одновременно могут поддерживаться многочисленные различные операционные системы. Метод паравиртуализации позволяет добиться более высокой производительности, чем метод динамической трансляции.

Встроенная виртуализация *— новый метод, базирующийся на применении аппаратно-поддерживаемых возможностей виртуализации, что позволяет пользователям использовать любые версии ОС в сочетании с различными вариантами рабочих сред.*

По сути, встроенная виртуализация представляет собой полную виртуализацию, реализованную на аппаратном уровне. Данный подход был реализован в рамках проекта BlueStacks Multi-OS (MOS).

Преимуществами этого метода являются: совместное использование ресурсов несколькими гостевыми операционными системами (каталоги, принтеры и так далее).

Удобство интерфейса для окон приложений из разных систем (перекрывающиеся окна приложений, одинаковая минимизация окон, как в хост-системе). При тонкой настройке на аппаратную платформу производитель-

ность мало отличается от оригинальной операционной системы. Быстрое переключение между системами (менее одной секунды).

Простая процедура обновления гостевой операционной системы. Двухсторонняя виртуализация (приложения одной системы запускаются в другой и наоборот).

3.3.6.2 Аппаратная виртуализация

В отличие от программных методов, с помощью аппаратных средств виртуализации возможно получение изолированных гостевых систем, управляемых гипервизором напрямую.

Аппаратный процесс виртуализации практически не имеет никаких кардинальных отличий от программного. По факту, это процесс виртуализации, подкрепленный аппаратной поддержкой. При этом технологический способ решения основан на переводе гипервизора в уровень защиты выше 0-го кольца, таким образом, команды ядра гостевой операционной системы находясь даже на 0-м уровне, перехватываются более приоритетным гипервизором с поддержкой на аппаратном уровне.

Аппаратная виртуализация обеспечивает производительность, сравнимую с производительностью невиртуализованной машины, что дает виртуализации возможность практического использования и влечет её широкое распространение. Наиболее распространены технологии виртуализации Intel-VT и AMD-V, а также режим виртуального 8086.

В Intel VT (Intel Virtualization Technology) реализована виртуализация режима реальной адресации (режим совместимости с 8086). Соответствующая аппаратная виртуализация ввода-вывода — VT-d (кодовое название — Vanderpool). Часто обозначается аббревиатурой VMX (Virtual Machine eXtension).

AMD-V часто обозначается аббревиатурой SVM (Secure Virtual Machines). Кодовое название — Pacifica. Соответствующая технология виртуализации ввода-вывода — IOMMU. AMD-V проще и эффективнее, чем Intel VT. Поддержка AMD-V появилась в Xen 3.3.

Платформы, работающие на аппаратной виртуализации: IBM LPAR, VMware, Hyper-V, Xen, KVM.

VT-d (Virtualization technology for directed I/O) — технология виртуализации ввода-вывода

Технология, созданная корпорацией Intel в дополнение к её технологии виртуализации вычислений (VT), известной под кодовым названием Vanderpool.

Суть VT-решения заключается во введении нового режима функционирования процессора - VMX, предназначенного для поддержки виртуализации. В этом режиме команды процессора могут работать в двух новых режимах: VMX-root (обычный режим) и VMX-nonroot (режим исполнения виртуальной машины). По сути, эти режимы изменяют поведение некоторых команд. Переключения от VMX-root к VMX-nonroot называются входами в режим исполнения виртуальной машины (VM-entry), а обратные переключения - выходами из режима исполнения виртуальной машины (VM-exit). Сами переключения в терминологии Intel называются VMX-переходами (VMX-transition).

Поведение процессора в режиме VMX-root почти такое же, как и вовне режима VMX, за исключением случаев, когда имеется дополнительный набор VMX-команд или ограничения на значения, загружаемые в некоторые управляющие регистры (CR0, CR4).

Поведение процессора для режима VMX-nonroot изменено таким образом, чтобы осуществить работу гипервизора. Вместо того чтобы исполняться, некоторые инструкции (включая новую инструкцию вызова гипервизора - VMCALL, которая уменьшает накладные расходы таких обращений), попав в CPU и пытаясь быть исполненными в режиме VMX-nonroot, будут приводить к исключению - выходу из режима исполнения виртуальной машины и передаче управления гипервизору. Эти переходы остаются невидимыми для гостевого VMX-nonroot-кода, существенно ограничивая его прямой доступ к

физическому оборудованию и позволяя гипервизору полностью контролировать ресурсы компьютера.

Виртуализация ввода-вывода позволяет пробрасывать (pass-through) устройства на шине PCI (и более современных подобных шинах) в гостевую ОС, таким образом, что она может работать с ним с помощью своих штатных средств. Чтобы такое было возможно, в логических схемах системной платы используется специальное устройство управления памятью ввода-вывода (IOMMU), работающее аналогично MMU (memory management unit) центрального процессора, используя таблицы страниц и специальную таблицу отображения DMA (DMA remapping table — DMAR), которую гипервизор получает от BIOS через ACPI (интерфейс управления конфигурацией оборудования и питанием). Отображение DMA необходимо, поскольку гипервизор ничего не знает о специфике работы устройства с памятью по физическим адресам, которые известны лишь драйверу. С помощью DMAR он создает таблицы отображения таким образом, что драйвер гостевой ОС видит виртуальные адреса IOMMU аналогично тому, как бы он видел физические без него и гипервизора.

Intel Virtualization Technology for Directed I/O (VT-d) — это следующий важный шаг на пути к всеобъемлющей аппаратной поддержке виртуализации платформ на базе Intel. VT-d расширяет возможности технологии Virtualization Technology (VT), существующей в IA-32 (VT-x) и Itanium (VT-i), и добавляет поддержку виртуализации новых устройств ввода-вывода.

Гипервизор Xen поддерживает DMAR начиная с версии 3.3 для аппаратно-виртуализуемых доменов. Для паравиртуальных доменов отображение DMA не требуется.

Заявлена поддержка технологии ПО Oracle VirtualBox. Ядро Linux экспериментально поддерживает DMAR начиная с версии 2.6.28, что позволяет встроенному гипервизору (kvm) давать доступ виртуальным машинам к PCI-устройствам. Поддержка Intel VT-d есть в Parallels Workstation 4.0 Extreme и в Parallels Server 4 Bare Metal.

AMD Virtualization (AMD-V)

AMD разработала свои расширения виртуализации первого поколения под кодовым названием «Pacifica», и первоначально опубликовала их как AMD Secure Virtual Machine (SVM), но позже, на рынке, — под торговой маркой «AMD Virtualization», сокращенно «AMD-V».

23 мая 2006 года AMD выпустила Athlon 64 («Orleans»), Athlon 64 X2 («Windsor») и Athlon 64 FX («Windsor») в качестве первых процессоров AMD с поддержкой данной технологии.

Поддержка AMD-V также обеспечивается в семействе процессоров Athlon 64 и Athlon 64 X2 ревизий «F» или «G» на Socket AM2, Turion 64 X2, и Opteron второго поколения и третьего поколения, а также процессорами Phenom и Phenom II. Только две модели Sempron поддерживают её: Huron and Sargas.

Процессоры AMD Fusion также поддерживают AMD-V.

AMD-V не поддерживается в процессорах на Socket 939.

Процессоры Opteron, начиная с семейства 0x10 Barcelona, и процессоры Phenom II поддерживают второе поколение аппаратной виртуализации технология под названием Rapid Virtualization Indexing (ранее известная как Nested Page Tables во время его разработки), позже адаптированные Intel, как Extended Page Tables (EPT).

Скорость переключения гипервизора – виртуальная машина в Pacifica может быть заметно выше по сравнению с Intel VT так как таблицы трансляции виртуальных адресов в физические в Pacifica могут очищаться выборочно, в отличие от Intel VT, хотя программная реализация кэширования для последней может дать практически такой же результат. Кроме того, виртуальная машина может работать через DMA без контроля гипервизора.

3.3.7 Гипервизор

Гипервизор *- программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполне-*

ние нескольких операционных систем на одном и том же хост-компьютере.

. . . . . .

Гипервизор также обеспечивает изоляцию операционных систем друг от друга, защиту и безопасность, разделение ресурсов между различными запущенными ОС и управление ресурсами (рисунок 3.9).

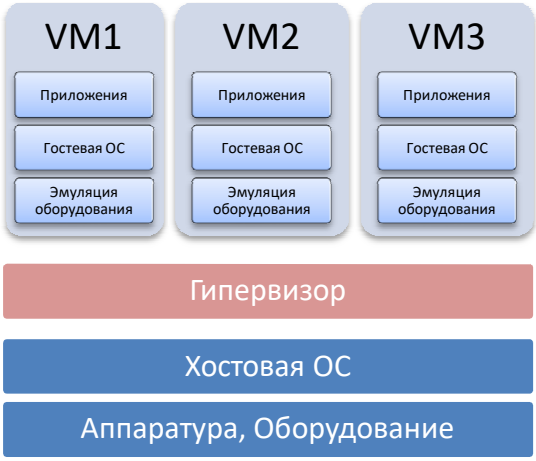


Рисунок 3.9 - Схема работы гипервизора

Гипервизор также обязан предоставлять работающим под его управлением на одном хост-компьютере ОС средства связи и взаимодействия между собой (например, через обмен файлами или сетевые соединения) так, как если бы эти ОС выполнялись на разных физических компьютерах.

Гипервизор сам по себе в некотором роде является минимальной операционной системой (микроядром или наноядром). Он предоставляет запущенным под его управлением операционным системам службу виртуальной машины, виртуализируя или эмулируя реальное (физическое) аппаратное обеспечение конкретной машины. И управляет этими виртуальными машинами, выделением и освобождением ресурсов для них. Гипервизор позволяет независимое «включение», перезагрузку, «выключение» любой из виртуальных машин с той или иной ОС. При этом операционная система, работающая в виртуальной машине под управлением гипервизора, может, но не обязана «знать», что она выполняется в виртуальной машине, а не на реальном аппаратном обеспечении.

Автономный гипервизор (Тип 1, X)

Имеет свои встроенные драйверы устройств, модели драйверов и планировщик и поэтому не зависит от базовой ОС. Так как автономный гипервизор работает непосредственно в окружении усечённого ядра, то он более производителен, но проигрывает в производительности виртуализации на уровне ОС и паравиртуализации. Например, кроссплатформенный гипервизор Xen может запускать виртуальные машины в паравиртуальном режиме (зависит от ОС).

Пример

VMware ESX, Citrix XenServer.

На основе базовой ОС (Тип 2, V)

Это компонент, работающий в одном кольце с ядром основной ОС (кольцо 0). Гостевой код может выполняться прямо на физическом процессоре, но доступ к устройствам ввода-вывода компьютера из гостевой ОС осуществляется через второй компонент, обычный процесс основной ОС — монитор уровня пользователя.

Пример

Microsoft Virtual PC, VMware Workstation, QEMU, Parallels, VirtualBox.

Гибридный (Тип 1+)

Гибридный гипервизор состоит из двух частей: из тонкого гипервизора, контролирующего процессор и память, а также специальной служебной ОС, работающей под его управлением в кольце пониженного уровня. Через служебную ОС гостевые ОС получают доступ к физическому оборудованию.

Пример

Microsoft Virtual Server, Sun Logical Domains, Xen, Citrix XenServer, Microsoft Hyper-V, VMware Workstation.

. . . . . .

3.3.7.1 Xen

Кроссплатформенный гипервизор, разработанный в компьютерной лаборатории Кембриджского университета и распространяемый на условиях лицензии GPL.

Основные особенности: поддержка режима паравиртуализации помимо аппаратной виртуализации, минимальность кода самого гипервизора за счёт выноса максимального количества компонентов за пределы гипервизора.

Xen разработчики отказались от того, чтобы функции работы с внешними устройствами были полностью возложены на гипервизор, который либо выполняет их самостоятельно, либо передает основной ОС, в свою очередь предусматривающей процессы для обслуживания гипервизора. Для работы с физическими устройствами Xen использует одну из гостевых ОС, обладающих соответствующими драйверами и полномочиями по доступу к оборудованию. В остальных гостевых ОС драйверы оборудования редуцированы до специальных "передающих" заглушек-фронтендов, которые вызывают гипервизор, а тот передает эти запросы гостевой ОС с настоящими драйверами.

Основной концепцией гипервизора Xen является домен. Доменом называется запущенная копия виртуальной машины. Если виртуальная машина перезагружается, то её домен завершается (в момент перезагрузки) и появляется новый домен. Более того, даже при миграции содержимое копируется из одного домена в другой домен. Таким образом, за время своей жизни практически все виртуальные машины оказываются по очереди в разных доменах. Xen оперирует только понятием домена, а понятие «виртуальной машины» появляется на уровне администрирования (прикладных программ, управляющих гипервизором).

Домены бывают нескольких типов. Самые известные dom0 и domU.

dom0 — первый запущенный Xen домен, обычно он автоматически создаётся и загружается сразу после загрузки и инициализации гипервизора. Этот домен имеет особые права на управление гипервизором и по умолчанию всё аппаратное обеспечение компьютера доступно из dom0. Фактически, dom0 — это место для ПО, управляющего Xen. dom0 всегда один.

domU — рядовой домен (сокращение от User domain), содержащий в себе домен выполняющихся виртуальных машин. Обычно не имеет доступа к реальному оборудованию и является «полезной нагрузкой» системы виртуализации. В отличие от dom0, domU может быть множество (обычно несколько десятков).

stub-domain — домен, в котором запущена очень специализированная ОС, обеспечивающая работу с каким-либо оборудованием или бэк-эндом драйвера. Является развитием модели безопасности Xen.

domain builder (конструктор доменов) — программа, которая создаёт domU (загружает в него нужный код и сообщает гипервизору о необходимости запуска). Помимо конструирования домена, обычно занимается подключением и конфигурированием виртуальных устройств, доступных для виртуальной машины. Она же отвечает за процесс миграции виртуальной машины с хоста на хост.

Гипервизор Xen (для версии 3.4) реализует минимальный набор операций для управления оперативной памятью, состоянием процессора, таймерами реального времени и счётчиками тактов (TSC) процессора, прерываниями и контролем за DMA. Все остальные функции, такие как реализация дисковых и блочных устройств, создание и удаление виртуальных машин, их миграция между серверами и т. д. реализуется, как уже было упомянуто выше, в управляющем домене. За счёт этого размер гипервизора получается весьма малым (для версии 3.4 размер двоичного кода всего гипервизора меньше 600 КБ), так же, как и размер его исходного текста. По замыслу авторов это увеличивает устойчивость системы виртуализации, так как ошибка в компонентах вне гипервизора не приводит к компрометации/повреждению самого ги-

первизора и ограничивает повреждения только вышедшим из строя компонентом, не мешая работать остальным.

Все функции, связанные с обеспечением работы сети, блочных (дисковых) устройств, эмуляции видеоадаптеров и прочих устройств вынесены за пределы гипервизора. Большинство таких устройств состоит из двух частей: драйверы в domU и программы в dom0. Драйвер (чаще всего встроенный в ядро ОС или загружающийся в виде модуля) реализует минимальный объём работы, фактически, транслируя запросы от ОС в программу в dom0. Программа в dom0 выполняет основную часть работы. При этом программа чаще всего запускается в виде отдельного процесса для каждого обслуживаемого устройства. Сбой в такой программе ведёт к сбою только одного устройства (блочного, сетевого) и не затрагивает работу других копий программы (то есть не затрагивает сетевые/блочные устройства остальных доменов, или даже другие устройства того же самого домена).

Xen (с помощью стека управления) поддерживает миграцию гостевых виртуальных машин по сети. Миграция паравиртуальных машин поддерживается с версии Xen 2, а HVM – с версии 3. Миграция может происходить с выключением гостевой системы, или прямо в процессе работы, так называемая «живая» миграция (англ. live migration) без потери доступности.

Необходимо, чтобы оба физических сервера Xen видели одно и то же хранилище, на котором находятся данные виртуальной машины. Это требуется потому, что при миграции виртуальной машины её файловая система не копируется, так как это требовало бы слишком много времени даже в случае быстрой сети. Общее хранилище может быть организовано на основе различных технологий SAN или NAS, например, Fibre Channel, iSCSI или DRBD.

Процесс живой миграции на примере гипервизора Xen. Для живой миграции необходимо:

- Минимум два сервера в кластере. Диск виртуальной машины должен находиться на ресурсе доступном обоим серверам кластера — старом

и новом местонахождении. Путь к диску виртуальной машины на физических серверах должен быть идентичен. Серверы должны иметь доступ к одной подсети, в которой находится сетевой интерфейс виртуальной машины. Гипервизоры должны быть одинаковых, либо совместимых версий.

- Совместимое оборудование серверов в кластере (в первую очередь архитектуры и расширения CPU).
- Принципиальная очередность процесса живой миграции (на практике требуются дополнительные операции). Остановка выполнения виртуальной машины. Передача параметров виртуальной машины с сервера исходного расположения на сервер целевого расположения. Передача образа оперативной памяти с сервера исходного расположения виртуальной машины на сервер целевого расположения. Создание виртуально домена и размещение образа оперативной памяти в RAM сервера целевого расположения. Запуск выполнения виртуальной машины на сервере целевого расположения.

В связи с тем, что сам гипервизор (около 500—600 КБ) реализует только «ядро» системы, вся остальная функциональность выносится на прикладной уровень, работающий в dom0. Набор программ, реализующий функциональность за пределами Xen называют англ. toolstack (устоявшегося перевода не имеется, иногда используется термин «стек управления»).

Существуют две версии toolstack для Xen: основанная на xend (входит в большинство поставок Xen) и основанная на xapi (входит в состав Citrix XenServer и Xen Cloud Platform). Xend развивался одновременно с Xen, написан на Python и с самого начала шёл под открытой лицензией. Xapi был проприетарной разработкой Xensource (в дальнейшем Citrix), но в 2009 году был опубликован под лицензией GPL. Xapi написан на OCaml, на момент написания имел меньший набор возможностей, но работал более стабильно.

В версиях toolstack присутствуют следующие утилиты:

xenstored — демон, реализующий интерфейс XenStore – простая древовидная база данных, напоминающая procfs. В xapi-toolstack XenStore переписан на ocaml, но реализует ту же самую функциональность.

xenconsoled — демон, обеспечивающий в dom0 доступ к консолям виртуальных машин. Xenconsoled реализует бэкэнд консольного устройства для domU и использует API unix98 для создания псевдотерминалов в dom0. Соответствие между номером псевдотерминала и виртуальной машиной записывается в XenStore.

Toolstack обеспечивает управление виртуальными машинами (создание/удаление, запуск/останов, миграция, подключение ресурсов и т.д.).

Кроме этого, инструментарий обеспечивает управление ресурсами для крупномасштабных систем: создаёт и поддерживает репозитории хранения образов дисков виртуальных машин (SR — storage repository), поддерживает пулы серверов для миграции виртуальных машин и может управлять сложными конфигурациями локальной сети, в том числе с поддержкой VLAN. Кроме того, поддерживается интерфейс удаленного управления XenApi на основе XML-RPC.

Являясь гибридным гипервизором типа 1, Xen запускается непосредственно на аппаратной платформе, но для своей работы требует управляющей операционной системы в dom0. Xen поддерживает процессоры, начиная от Pentium II, имеются версии для архитектур x86-64, PowerPC, Itanium (до версии 4.4) и ARM (стабильна с версии 4.4). Загрузка Xen осуществляется начальным загрузчиком типа GRUB или подобным. Непосредственно после загрузки Xen запускает операционную систему в dom0.

В большинстве инсталляций в качестве ОС управляющего домена dom0 используется Linux. Долгое время поддержка Xen не была включена в официальное ядро Linux и существовала в виде набора патчей для ядра v2.6.18. Начиная с v2.6.37 в ядре Linux появился механизм pv_ops для взаимодействия с гипервизорами. Данный механизм позволяет ядру работать как в паравиртуальном режиме, так и непосредственно на железе. Начиная с версии Xen 4.0 поддерживает механизм pv_ops для ядра Linux в dom0. Ядра Linux выше 3.0 также полностью поддерживают Xen и для dom0 и для domU.

Поддерживаются различные ОС для dom0, в основном дистрибутивы Linux, для гостевой ОС есть поддержка Windows.

Xen широко применяется как компонент виртуализации в облачных вычислениях и при организации служб выделенных частных серверов. Такие хостинговые компании как Amazon Elastic Compute Cloud, Liquid Web, Fujitsu Global Cloud Platform, Linode, SparkNode и Rackspace Cloud используют Xen как гипервизор виртуальных машин.

Сообщество Xen разрабатывает Xen Cloud Platform (XCP) — систему серверной виртуализации. Своё происхождение XCP ведет от бесплатной версии Citrix XenServer и выпускается полностью под GNU GPL.

3.3.7.2 VMware Workstation.

Ядром технологии компании VMware является полная эмуляция оборудования на уровне ПО. Это означает, что существует некий процесс уровня пользователя основной операционной системы, внутри которого, собственно, и проходит эмуляция. Это полууниверсальная технология (как правило, она позволяет эмулировать аппаратуру платформы того же типа, на котором запущена основная операционная система), предназначенная для запуска большинства современных ОС внутри такого эмулятора - поверх основной операционной системы. Для перехвата нежелательных команд применяется метод бинарной трансляции кода.

VMware Workstation позволяет пользователю установить одну или более виртуальных машин на один физический компьютер и запускать их параллельно с ним. Каждая виртуальная машина может выполнять свою операционную систему, включая Microsoft Windows, Linux, BSD, и MS-DOS. VMware Workstation разработана и продается компанией VMware, подразделением EMC Corporation.

VMware Workstation поддерживает мосты с сетевым адаптером реального компьютера, а также создание общих папок с виртуальной машиной. Программа может монтировать реальные CD или DVD диски или ISO образы в виртуальные оптические приводы, при этом виртуальная машина будет

считать, что приводы настоящие. Виртуальные жесткие диски хранятся в файлах .vmdk .

VMware Workstation в любой момент может сохранить текущее состояние виртуальной машины (снимок). Данные снимки позже могут быть восстановлены, что возвращает виртуальную машину в сохраненное состояние.

VMware Workstation включает в себя возможность объединять несколько виртуальных машин в группу, которую можно включать, выключать, приостанавливать или возобновлять как единый объект, что является полезным для тестирования технологий клиент-сервер

Технология не обеспечивает эффективного совместного использования всех аппаратных ресурсов, и масштабируемость решения принципиально ограничена из-за того, что оперативная память основной машины практически должна быть жестко разделена, чтобы каждой запущенной ВМ предоставить свою собственную неразделяемую память. Кроме того, около 20% ОЗУ следует зарезервировать для обслуживания виртуальных машин (накладные расходы системы виртуализации). То есть на обычных серверах фактически нельзя запустить более 15 виртуальных машин, а реально и еще меньше.

3.3.8 Виртуализация на уровне ОС. Контейнерная виртуализация.

При данном методе виртуализации, ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя. С точки зрения пользователя эти экземпляры, называемые контейнерами полностью идентичны реальному серверу. В некотором смысле для систем на базе UNIX, данная технология может рассматриваться как улучшенная реализация механизма chroot. Ядро обеспечивает полную изолированность контейнеров, поэтому прикладные программы из разных контейнеров не могут воздействовать друг на друга. При этом нет отдельного слоя гипервизора и хостовая операционная система сама отвечает за разделение ресурсов между контейнерами – гостевыми системами, ядро гостевых систем не должно отличаться от ядра базовой системы (рисунок 3.10).

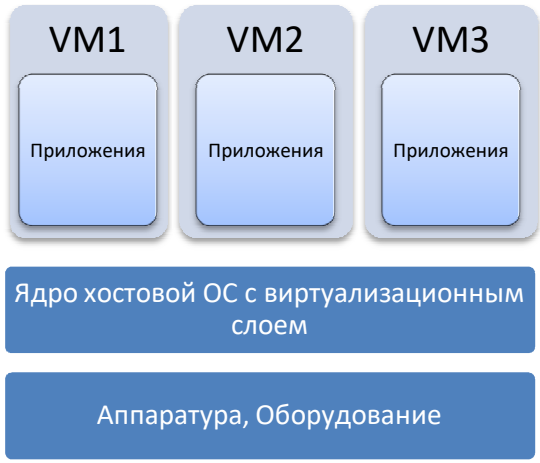


Рисунок 3.10 - Схема работы виртуализации на уровне операционной системы

Основные преимущества контейнерной виртуализации:

- Контейнеры выполняются на одном уровне с физическими серверами. Отсутствие виртуализованного оборудования и использование реального оборудования и драйверов позволяют получить достаточно высокую производительность.
- Каждый контейнер может масштабироваться до ресурсов целого физического сервера.
- Технология виртуализации на уровне ОС позволяет добиться одной из самых высоких плотностей размещения, среди доступных решений виртуализации. Возможно создание и запуск сотен контейнеров на одном обычном физическом сервере.
- Контейнеры используют единую ОС, что делает их поддержку и обновление очень простым. Приложения могут быть также развернуты в отдельном окружении.

Реализациями являются FreeBSD Jail (2000), Virtuozzo Containers (2000), Solaris Containers (2005), Linux-VServer[en], OpenVZ (2005), LXC (2008), iCore Virtual Accounts (2008), Docker (2013).

История подобного подхода начинается в 2005 году, когда компания Google занялась задачей массового предоставления Web-сервисов. Требовал-

ся способ эластичного масштабирования ресурсов в центре обработки данных Google, чтобы каждый пользователь имел возможность получить достаточный уровень сервиса в любой момент, независимо от текущей загрузки, а оставшиеся ресурсы можно было использовать для служебных фоновых задач.

Поэкспериментировав с традиционной виртуализацией, сотрудники Google сочли ее не подходящей. Главной проблемой стали слишком большие потери производительности и недостаточно гибкий и быстрый отклик для динамического переконфигурирования системы под изменившуюся нагрузку для массового предоставления Web-сервисов.

Динамическое гибкое выделение ресурсов очень важно, так как заранее предсказать количество запросов к сервису бывает проблематично, и это количество может варьироваться от десятков до миллионов. При этом пользователи всегда ожидают немедленного отклика, то есть незаметной для них задержки. Среднее время загрузки гипервизорной виртуальной машины может составлять десятки секунд, поэтому такой тип виртуализации здесь не подходит.

В то же самое время одна группа разработчиков экспериментировала с Linux и концепцией, основанной на механизме cgroups — так называемые контейнеры процессов. В январе 2008 года часть технологии cgroup, используемой Google, была перенесена в ядро Linux. Так родился проект LinuX Containers (LXC). Тем временем Parallels выпустила версию своей виртуализации Virtuozzo с открытым исходным кодом под названием OpenVZ. В 2011 году Google и Parallels пришли к соглашению о сотрудничестве в области контейнерных технологий. Результатом стал релиз ядра Linux версии 3.8, представленный в 2013 году. В нем были объединены все актуальные на тот момент контейнерные технологии для Linux, что позволило избежать повторения болезненного разделения ядер, как в случае с KVM и Xen.

По началу переход на новую технологию сопровождался проблемами, в связи с тем, что на новую технологию не хотели переходить и мало кто о ней слышал, отдавая предпочтения известным гипервизорам.

Но вышедшая на рынок компания-разработчик Docker продемонстрировала, как легко «упаковать» контейнеризованное приложение в Linux и развернуть его с возможностью масштабирования прямо в dotCloud (сервис Platform as a Service, PaaS от Docker). Предприятия заинтересовались данным подходом. Одновременно OpenStack пообещала объединить системы управления облаками (Cloud Management) в единую платформу, охватывающую обе технологии виртуализации. Когда бизнес наконец-то увидел возможность управлять своими центрами обработки данных на основе гипервизоров с помощью единого инструмента, который параллельно позволяет осуществлять масштабное развертывание контейнеризованных приложений, начался настоящий бум.

3.3.8.1 Docker

Как уже было упомянуто данное программное обеспечение прендназначено для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему с поддержкой cgroups в ядре, а также предоставляет среду по управлению контейнерами. Изначально использовал возможности LXC, с 2015 года применял собственную библиотеку, абстрагирующую виртуализационные возможности ядра Linux — libcontainer. С появлением Open Container Initiative начался переход от монолитной к модульной архитектуре.

Простыми словами Docker не реализует развертывание операционной системы, а фактически позволяет создать окружение для запуска вашего приложения на базе ядра хостовой ОС, со всеми необходимыми зависимостями, библиотеками и программным обеспечением. Основное преимущество Dock-

er это возможность быстро развернуть свое приложение на целевой платформе без затрат времени на настройки в целевой системе. Не требуется много времени на запуск виртуальной машины, или каких-либо дополнительных ресурсов на гостевую ОС, которая требует обновлений или проводит какие-то операции, которые могут быть и не нужны. Реализуется общий подход к развертыванию и сборки приложения в таком контейнере, в отличие от подхода с гостевыми системами с гипервизором. Это средство упаковки, доставки и запуска приложения, то есть любое приложение, созданное на любом языке программирования будет иметь общий механизм запуска, общий интерфейс предоставляющий возможность запуска приложения.

В docker присутствует docker image (образ) – сборка с упакованным приложением, и docker container – работающие и запущенное приложение на базе docker image (запущенных копий контейнеров может быть сколько угодно), образ не может быть изменен контейнером, либо изменения будут касаться самого контейнера и его копии образа. Сам по себе образ представляет собой многослойную структуру. Например, на базе ядра и образа дистрибутива операционной системы можно создать начальный образ, допустим ubuntu 18.04, затем установить на базе данного образа дополнительные зависимости apt-get install python3-dev python3-pip python3-venv и мы получим еще один образ. На базе нового образа можно добавить дополнительные зависимости, например, библиотеку tensorflow и keras, поддержку cuda и у нас появится возможность запускать приложения, работающие с нейронными сетями. Кроме того, можно использовать уже созданные кем-то образы, например, взять образ с python и добавить в него, то, что нужно, или взять чистый образ и добавить и python и все остальное. Образы, которые созданы вами хранятся локально и из них можно выбирать варианты, но также есть docker hub, общий реестр для хранения образов, они создаются как разработчиками docker, так и другими зарегистрированными на docker hub пользователями. Docker можно скачать на сайте https://www.docker.com/get-started и установить.

Для Ubuntu 20.04 можно выполнить следующие команды в командной строке терминала.

Обновить список пакетов:
sudo apt update
Установить пакеты для работы apt через https
sudo apt install apt-transport-https ca-certificates curl software-properties-common
Добавить ключ GPG для официального репозитория Docker в вашу систему:
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –
Добавить репозиторий docker в источники обновлений apt
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
Обновить базу пакетов apt
sudo apt update
Проверить что установка происходит из репозитория Docker а не Ubuntu
apt-cache policy docker-ce
Установить docker-ce
sudo apt install docker-ce
Можно проверить статус запуска docker
sudo systemctl status docker

Посмотреть какие образы есть в системе можно с помощью команды:

$ docker image

Посмотреть какие контейнеры запущены в системе можно с помощью команды:

$ docker ps

Контрольные вопросы по главе 3

1. В чем особенности контейнерной виртуализации?
2. В чем особенности паравиртуализации?
3. Что такое миграция виртуальных машин?
4. Что такое гипервизор?
5. Что такое кластер?
6. Что такое ЦОД?
7. Что такое SaaS, PaaS, IaaS?
8. В чем отличие систем NUMA от систем SMP?

4. ПРИМЕРЫ ТЕХНОЛОГИЙ РЕАЛИЗАЦИИ ВЕБ-ПРИЛОЖЕНИЙ

Технологии реализующие передачу веб-контента как уже было указано в предыдущих главах основываются на стеке протоколов TCP/IP и прикладном протоколе HTTP или HTTPS, данный протокол для передачи своих команд использует TCP соединение. Потому в данной главе сначала рассмотрим способы создания TCP соединения используя технологию сокетов, с примерами на различных языках программирования, рассмотрим протокол HTTP, создание сертификата для HTTPS, и те технологии, которые реализуют современную архитектуру веб-приложения. Основные примеры будут приводиться на языке python, php, java, большее внимание будет уделено python и этому языку будет посвящен отдельный параграф.

4.1 Примеры реализации TCP соединений на основе сокетов

Socket *(гнездо, разъем) - абстрактное программное понятие, используемое для обозначения в прикладной программе конечной точки канала связи с коммуникационной средой, образованной вычислительной сетью.*

При использовании протоколов TCP/IP можно говорить, что socket является средством подключения прикладной программы к порту локального узла сети.

Для создания сокета в операционной системе служит системный вызов socket(). Для транспортных протоколов семейства TCP/IP существует два вида сокетов:

UDP-сокет – сокет для работы с датаграммами, и TCP сокет – для работы с каналами. Соответственно в стеке TCP/IP протокол TCP отвечает за надежную передачу потока данных, реализует он эту надежность за счет повторной отправки данных, на которые не пришли подтверждения за выделенное специальным таймером время, протокол UDP позволяет отправить

дейтаграммы, но надежность передачи не обеспечивает, то есть дейтаграммы могут и не дойти.

При создании сокета необходимо точно специфицировать его тип. Эта спецификация производится с помощью трех параметров вызова socket(). Первый параметр указывает, к какому семейству протоколов относится создаваемый сокет, а второй и третий параметры определяют конкретный протокол внутри данного семейства.

Обычно с точки зрения взаимодействия двух удаленных приложений рассматривают клиентское и серверное приложение, при этом клиентское приложение подключается к серверному. И то и другое при этом имеет IP адрес и порт, который используется для разделения приложений на одном узле, операционная система следит за занятыми портами и выделяет приложению свободный порт или порт, который укажет само приложение. Сокет содержит три параметра, семейство протоколов или адресов, тип протокола (дейтаграммный или потоковый, с установлением или без установления соединения, DGRAM, STREAM) и указание конкретного протокола, например, UDP или TCP. Далее для TCP сервера необходимо реализовать привязку (bind) к локальному сетевому интерфейсу и порту, например, для HTTP сервера, порт 80, после чего выделяется очередь клиентов с прослушиванием входящих соединений путем вызова функции listen. А затем для так называемых блокирующих сокетов функция accept ожидающая подключение клиента, и извлекающая из очереди параметры клиента и возвращающая дескриптор сокета клиента. После чего можно начать взаимодействие с передачей и приемом данных, с использованием функций recv, send или их аналогов с данным клиентом. Так как протокол TCP дуплексный, и клиентов может быть несколько и в рамках одного потока команд обработать всех клиентов или одновременно передавать и получать данные невозможно, то обычно реализуется многопоточное приложение, после получения дескриптора сокета клиента взаимодействие с ним обслуживается в отдельных потоках. В linux также используется технология select, poll или более современная epoll, позволяющие в од-

ном потоке (бесконечном цикле) опрашивать состояние файловых дескрипторов, к коим относятся и дескрипторы сокетов, но первые из них работают по принципу возврата состояния всех запрошенных дескрипторов, а вторая возвращает дескрипторы только с измененными состояниями, сводя сложность алгоритма к O(1) от O(n). Первые две отличаются тем, что select обрабатывает не более 1024 дескрипторов по умолчанию, при этом передаваемые параметры и состояния дескрипторов изменяются вызовом select и затем им необходимо возвратить состояние, так же в select нужно передавать максимальное значение сокета. С точки зрения операционной системы сильный разброс номеров сокетов может так же приводить к замедлению, так как оцениваемые файловые дескрипторы передаются в виде битовой маски. В poll эти проблемы решены, но по-прежнему нужно проверять состояние каждого оцениваемого дескриптора. Количество дескрипторов не ограничено. Фактически после вызовов указанных блокирующих функций, нужно запустить цикл проверки каждой структуры на изменение соответствующего дескрипторы и выполнить нужное действие в зависимости от готовности сокета к выполнению какой-либо операции (например, на чтение или запись). При этом мы так же проверяем и неизмененные дескрипторы. В epoll нужно пройтись только по списку измененных дескрипторов.

Очевидно технология epoll предпочтительнее к использованию в высоконагруженных приложениях с большим количеством подключаемых клиентов. Клиент обычно проще и вызывает функцию создания сокета и затем функцию connect в которой указывается IP и порт сервера, порт клиента задается автоматически операционной системой из свободных портов, по RFC динамические порты начинаются с 49152. Вообще в TCP/IP порт занимает два байта, а IPv4 адрес четыре байта, а IPv6 адрес 16 байт. После подключения по протоколу TCP работа идет с использованием протокола прикладного уровня, обычно команды данных протоколов (SMTP, POP3, HTTP 1.x) используют кодировку ASCII. Чаще всего эта работа заключается в отправке запросов и получении ответов. Так как запросов может быть очень много, в

современных высоконагруженных системах этому уделяется большое внимание, например, используются балансировщики нагрузки, далее в этой главе мы рассмотрим методы балансировки.

4.1.1 Пример сервера и клиента на Python

Сервер представлен ниже, здесь осуществляется обработка подключения одного клиента, данные приходящие от клиента преобразуются так, что текст содержит заглавные буквы, затем преобразованный текст отправляется клиенту. Версия python вторая.

```
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()
```

Код клиента представлен ниже.

```
import socket
sock = socket.socket()
sock.connect(('localhost', 9090))
sock.send('hello, world!')
data = sock.recv(1024)
sock.close()
print data
```

4.1.2 Пример сервера и клиента на java

Сервер представлен ниже. Можно сохранить как Server.java.

```
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
}
}
```

Пример клиента представлен ниже. Можно сохранить как Client.java.

```
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
  }
```

```
}
```

Далее можно откомпилировать и запустить сервер и клиенты.

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

Можно было бы привести примеры на других языках программирования, но мы бы обнаружили, что общая концепция сокетов не сильно различается. Хотя стоит отметить, что на некоторых языках код будет короче.

4.2 Протокол DNS

Система DNS *(Domain Name System «система доменных имён») это компьютерная распределённая система для получения информации о доменах. Чаще всего используется для с целью сопоставления имени хоста и IP-адреса, получения информации о маршрутизации почты.*

Распределённая база данных DNS поддерживается с помощью иерархии DNS-серверов, взаимодействующих по одноименному протоколу посредством протокола UDP или TCP через порт 53.

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.

DNS записи хранятся как в соответствующих узлах иерархии (рисунок 4.1), 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 сервера, к которому нужно обратиться самостоятельно.

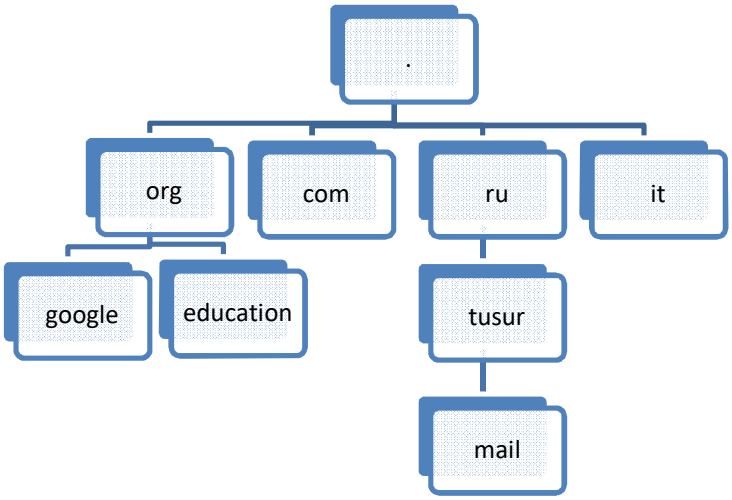


Рисунок 4.1 – Пример иерархических доменных серверов

Запросы DNS бывают следующих основных типов:

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.

Например, можно сделать соответствующий запрос используя команду nslookup.

nslookup -type=mx google.ru

Для получения записей об именах почтовых серверов домена google.

4.3 Протокол HTTP

HTTP *(HyperText Transfer Protocol или протокол передачи гипертекста) - это протокол прикладного уровня, предназначавшийся для передачи данных 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.

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.

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.

Пример

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 взаимодействия по данному протоколу состоит из следующих этапов:

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.

4.3.1 Структура HTTP запроса

Каждое HTTP-сообщение состоит из трёх частей, которые передаются в указанном порядке (рисунок 4.2):

- 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.

Рисунок 4.2 – Структура запроса протокола HTTP

Метод, указанный в строке состояния, определяет способ воздействия на ресурс, URL которого задан в той же строке.

Как уже упоминалось метод может принимать значения *GET*, *POST*, *HEAD*, *PUT*, *DELETE* и др.

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 предназна-

чен 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, 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.

Таблица 4.1 – Заголовки запросов

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.

```
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
```

4.3.2 Структура ответа и коды ответов 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.

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.

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 предпринять дополнительные действия.

4 - ошибка клиента. Как правило, код ответа, начинающийся с цифры 4, возвращается 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.

Таблица 4.2 – Примеры кодов ответов 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

Различие между 401 и 403 состояет в следующем, 401 Unauthorized – не авторизован, не представился (не прошел аутентификацию), 403 Forbidden (запрещено, не уполномочен), авторизованный пользователь не имеет права на такие действия с ресурсом, даже если он прошел аутентфикацию.

Некоторые заголовки, используемые в ответах сервера описаны в таблице 4.3.

Таблица 4.3 – Поля заголовков ответа 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

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 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">
        ...
```

Для нас интересными здесь могут быть 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 OK 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.

Также интересно 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'].

4.3.3 URI

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.)

<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.

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);

4.3.3.1 Data:URI, data: URL

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.

Далее приведен фрагмент внедрённого в XHTML небольшого изображения:

```
<img
src="
wCcDkiLc7C0qwyGHhSWpjQu5yqmCYsapyuvUUlvONmOZtfzgFzByTB10QgxOR0TqBQejhRNzOfkVJ+5
YiUqrXF5Y5lKh/DeuNcP5yLWGsEbtLiOSpa/TPg7JpJHxyendzWTBfX0cxOnKPjgBzi4diinWGdkF8kjdfny
cQZXZeYGejmJlZeGl9i2icVqaNVailT6F5iJ90m6mvuTS4OK05M0vDk0Q4XUtwvKOzrcd3iq9uisF81M1OI
cR7lEewwcLp7tuNNkM3uNna3F2JQFo97Vriy/Xl4/f1cf5VWzXyym7PHhhx4dbgYKAAA7"
alt="Larry"/>
```

Правило CSS с внедрённым фоновым изображением:

```
ul.checklist > li.complete {
margin-left: 20px;
background:
url(
```

iVBORw0KGgoAAAANSUhEUgAAABAAAAAQAQMAAAAlPW0iAAAABlBMVEUAAAD///+l2Z/dAAAAM0lEQVR4nGP4/5/h/l+G/58ZDrAz3D/McH8yw83NDDeNGe4Ug9C9zwz3gVLMDA/A6P9/AFGGFyjOXZtQAAAAAElFTkSuQmCC)
top left no-repeat;
}

4.3.4 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. 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.

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.

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.

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*.

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.

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*.

Приведем пример использования cookie в технологии 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>
```

4.3.5 Авторизация и аутентификация в Web

4.3.5.1 HTTP аутентификация и SASL(Simple Authentication and Security Layer) аутентификация

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. Но в данном случае можно перехватывать только текущую сессию, для другой сессии данный хэш не поможет, так как сервер посылает случайную строку, которая хэшируется с логином и паролем. Если говорить более точно, то можно привести в пример CRAM-MD5 или DIGEST-MD5. Напомним, что хэш MD5 является «однозначным» преобразованием строки в псевдослучайную строку, при этом обратного преобразования не существует, хотя в большинстве случаев это уже не так и MD5 не столь надежен, лучше использовать другие хэши, применяемые, например, в технологии блокчейн (SHA256 и другие). Первый протокол предполагает отправку случайной строки сервером, после чего эта строка хэшируется паролем пользователя, не будем вдаваться в подробности того, как преобразуется эта строка предварительно, но в итоге сервер должен таким же образом хэшировать отправленную им самим строку с паролем пользователя и соответственно его знать, очевидно, здесь можно перехватить сессию и при этом пароль хранится в открытом виде на сервере. Digest предполагает отправку клиенту случайной строки, но предварительно клиент хэширует вместе пароль и логин пользователя, затем полученный хэш хэширует вместе со случайной строкой сервера, затем все это вместе хэширует со своей случайной строкой, отправляет серверу полученный хэш и свою случайную строку в открытом виде, сервер проводит ту же операцию вместе со строкой полученной от клиента, при этом он может хранить хэш логина пароля, а не открытый логин пароль. Перехват от клиента хэша позволит перехватить сессию, типичная атака MITM (man in the middle, человек по середине). Все это выглядит на стороне клиента так H(H(login, pass)+server_string)+client_string. Здесь приведен пример, не учи-

тывающий различные параметры протоколов, которые так же входят в хэш, например, realm, но пример приведен для понимания концепции.

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*. Здесь используются доверенные центры распределения ключей.

Только при использовании данного типа аутентификации можно работать по схеме *http*, во всех остальных случаях необходимо использовать схему *https*.

Параметры требующиеся для Basic и Digest аутентификации передаются в заголовках сервера.

Например,

GET /images/image1.jpg HTTP/1.1

Authorization: Basic Zm9vOmJhcg==

Здесь в base64 передается логин и пароль пользователя, foo: bar.

Пример с Digest будет посложнее. Например, вы обратились к ресурсу и вам вернулося следующий ответ.

HTTP/1.0 401 Unauthorized

WWW-Authenticate: Digest realm="my@my.com",

qop="auth,auth-int",

nonce="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",

opaque="yyyyyyyyyyyyyyyyyyyyyyyyyy"

В ответ вы уже пытаетесь пройти процедуру аутентификации:

GET /dir/index.html HTTP/1.0

Host: localhost

Authorization: Digest username="my_user",
realm="my@my.com",
nonce=" xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
uri="/dir/index.html",
qop=auth,
nc=00000001,
cnonce="0a4f113b",
response="calculated by hash function response",
opaque="yyyyyyyyyyyyyyyyyyyyyyyyyyy"

Вычисление ответа response осуществляется следующим образом.

HA1 = MD5("my_user:my@my.com:password")

HA2 = MD5("GET:/dir/index.html")

Response = MD5("HA1:
cmFuZG9tbHlnZW5lcmF0ZWRub25jZQ:
nc:cnonce:auth:
HA2")

4.3.5.2 Cookie-based авторизация

4.3.5.3 JWT токен авторизация

4.3.5.4 Авторизация Auth 2.0

Является открытым фреймворком (протоколом) авторизации, позволяющим получить сторонним приложениям ограниченный доступ к ресурсам HTTP-сервиса, позволяет предоставить права на использование некоторого ресурса (например, API какого-либо сервиса). При этом в общем случае нельзя определить, кто в настоящий момент пользуется правами.

Схематично сценарий протокола OAuth 2.0 представлен на рисунке. Он описывает взаимодействие между упомянутыми выше четырьмя сторонамии включает следующие шаги.

1) Клиент запрашивает авторизацию у владельца ресурса. Запрос авторизации может быть направлен владельцу ресурса напрямую, как показано на рисунке, или косвенно через сервер авторизации. Предпочтительным является второй вариант.

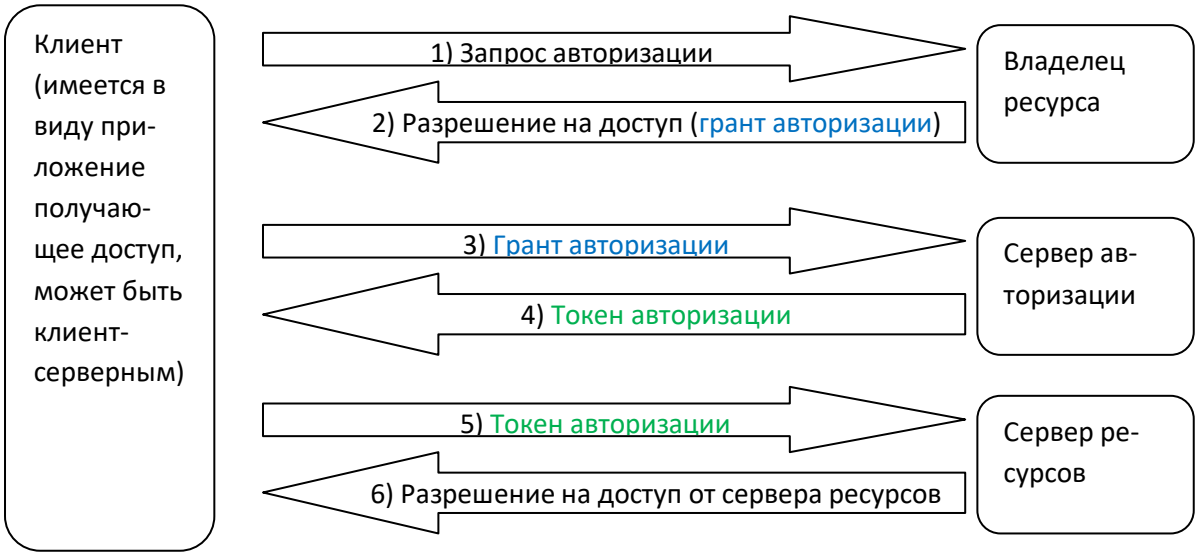
2) Клиент получает разрешение на доступ (grant), структуру данных, представляющую авторизацию владельца ресурса, выраженную с использованием одного из четырех типов разрешений: код авторизации (authorization code), неявное разрешение (implicit), пароль владельца ресурса (resource owner password credentials) и учетные данные клиента (client credentials), так же есть еще Device authorization для доступа устройств (если нет веб-браузера, RFC 8628) . Тип разрешения на доступ зависит от метода, используемого клиентом для запроса авторизации, и типов разрешений, поддерживаемых сервером авторизации. Типы разрешений, поддерживаемые сервером авторизации, определяются при его разработке, исходя из его прикладных целей и задач. Настоящий стандарт регламентирует использование в качестве типа разрешения код авторизации.

3) Клиент запрашивает токен доступа посредством аутентификации на сервере авторизации и предоставления разрешения на доступ.

4) Сервер авторизации аутентифицирует клиента, проверяет разрешение на доступ и, если оно действительно, выдает токен доступа.

5) Клиент запрашивает защищенный ресурс на сервере ресурсов и аутентифицируется, представляя токен доступа.

6) Сервер ресурсов проверяет токен доступа и, если он действителен, обслуживает запрос.



Стандарт OAuth 2.0 определяет следующие четыре роли:

• владелец ресурса — сущность, обладающая правом на выдачу доступа к защищенным ресурсам. В случае, если владелец является человеком, его называют конечным пользователем;

• сервер ресурсов — сервер, содержащий защищаемые ресурсы и обладающий возможностью получения и формирования ответа на запросы к защищаемым ресурсам посредством использования маркера доступа;

• клиент — приложение, осуществляющее доступ к защищенным ресурсам от имени Владельца. Термин "клиент" явно не определяет какое-либо конкретное исполнение (будь то сервер, персональный компьютер или мобильное приложение); Приложением может являться браузер, толстый клиент или мобильное приложение. Клиент может быть public и confedential. Public не может безопасно хранить свои учётные данные. Этот клиент работает на устройстве владельца ресурса, например, это браузерные или мобильные приложения. Confedential может безопасно хранить свои учетные данные, например бэкенд приложения.

• сервер авторизации — сервер, осуществляющий выпуск маркеров доступа для клиентских приложений после успешной аутентификации и авторизации Владельца ресурсов.

Протокол OAuth обладает возможностью аутентификации не только Пользователя, но и клиентского приложения, осуществляющего доступ к ресурсам.

Рассмотрим пять способов получения доступа (grant) в auth 2.0, начиная с самого простого.

Client credentials grant flow

Предполагает самый простой способ получения прав доступа к ресурсам. Клиент отправляет на сервер авторизации client id и client secret, на что возвращается access token с которым клиент может обратиться к серверу ресурсов. Данный способ используется для доступа к собственным ресурсам или предоставление доступа к серверу ресурсов согласованному с сервером авторизации, когда клиент совпадает с владельцем ресурсов. Допускается только для защищенных клиентов (confedential, клиент, который может безопасно хранить свои учётные данные, к такому типу клиентов относят web-приложения, имеющие backend.).

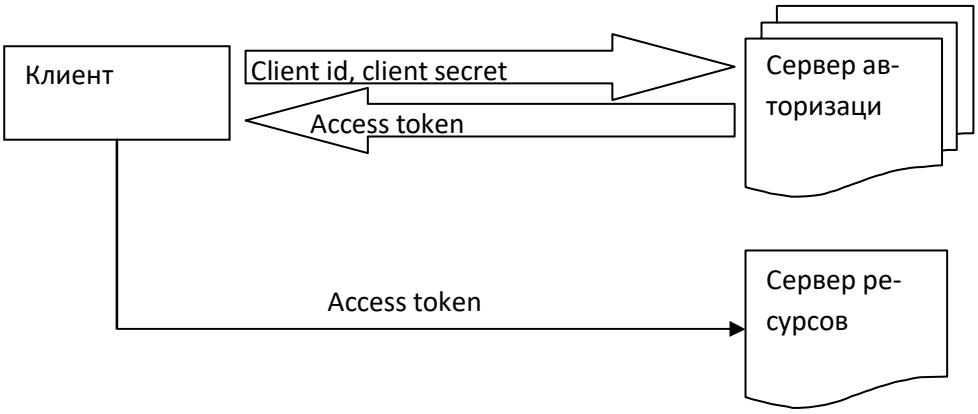


Рисунок – Схема AUTH 2.0 взаимодействия где клиент является владельцем ресурсов

Здесь не указано как и в других примерах взаимодействие сервера и авторизации и сервера ресурсов, но очевидно оно должно быть для согласования режима доступа передаваемого в scope. Кроме того, сервер ресурсов и сервер авторизации могут быть одним и тем же. Auth 2.0 не накладывает ограничения на роли указанных взаимодействующих частей.

Далее рассмотрим Authorization code flow, самая распространенная схема и одна из защищенных, за счет отправки дополнительно между клиентом и сервером авторизации client_id и client_secret, но и она может быть подвергнута атакам, связанным с неправильным использованием redirect_uri. На рисунке приводится схема взаимодействия с учтом того, что клиент представляет собой веб приложение с фронт и бэк частью.

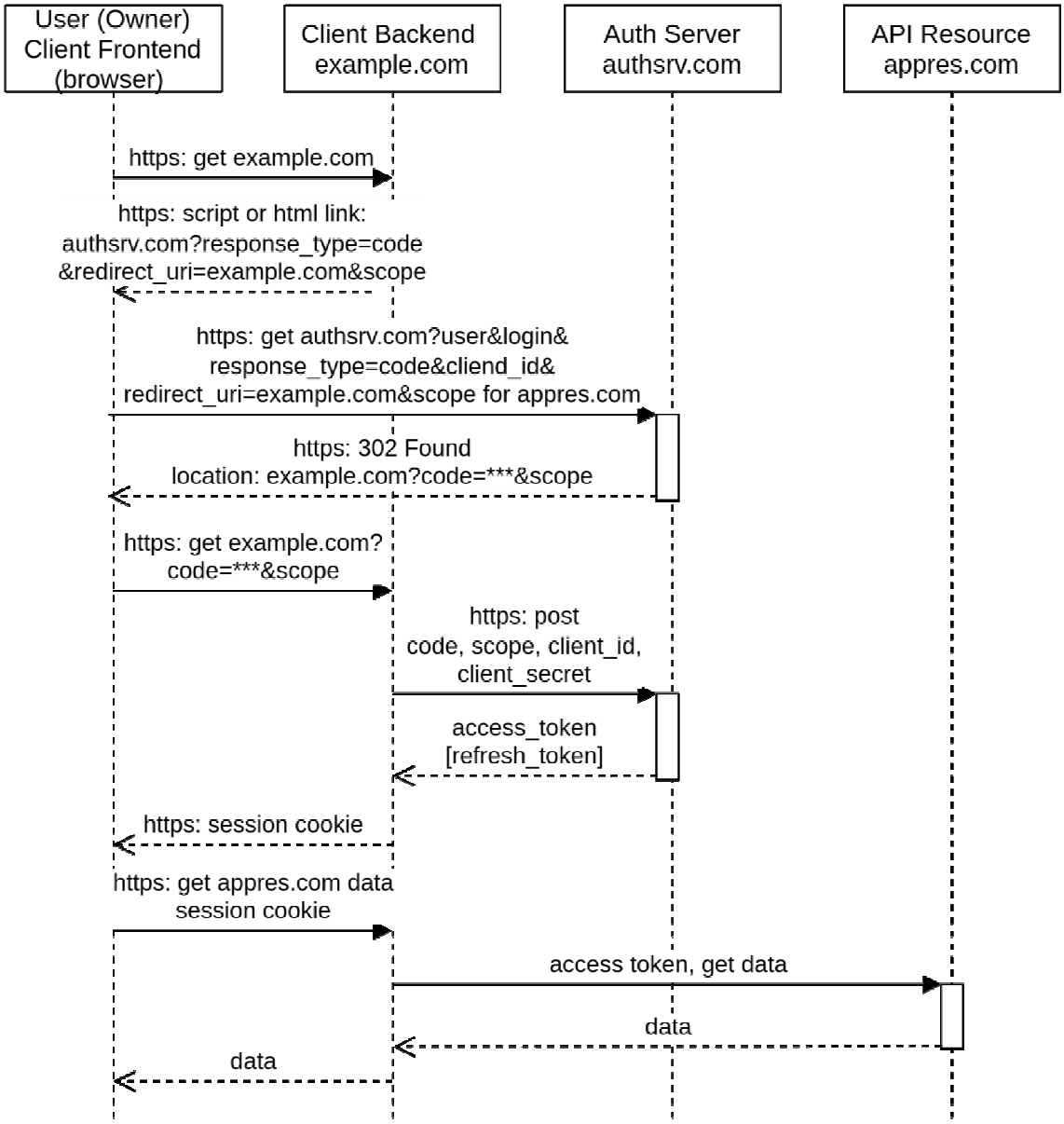


Рисунок – Схема взаимодействия получения прав доступа Authorization code flow

На первом шаге клиент перенаправляет resource owner с помощью user-agent на страницу аутентификации Authorization server. В URI он указывает client ID и redirection URI. Redirection URI используется для понимания, куда вернуть resource owner после того, как авторизация пройдёт успешно (resource owner выдаст разрешение на scope, запрашиваемый клиентом).

Взаимодействуя с сервером авторизации через user-agent, resource owner проходит аутентификацию на сервере авторизации.

Resource owner проверяет права, которые запрашивает клиент на consent screen и разрешает их выдачу.

Resource owner возвращается клиенту с помощью user-agent обратно на URI, который был указан как redirection URI. В качестве query-параметра будет добавлен authorization code — строка, подтверждающая то, что resource owner выдал необходимые права сервису.

С этим authorization code клиент отправляется на сервер авторизации, чтобы получить в ответ access token (ну и refresh token, если требуется).

Сервер авторизации валидирует authorization code, убеждаясь, что токен корректный и выдаёт клиенту access token (и опционально refresh token). С его помощью клиент сможет получить доступ к заветному ресурсу.

Еще одним способом доступа является implicit grant flow (неявный доступ), но он не рекомендуется. В данном случае вместо кода сразу возвращается аксес и рефреш токен, это небезопасно.

В другом варианте, который уже запрещен пароль владельца ресурса (resource owner password credentials) пароль и логин передается через клиента на сервер авторизации, что, совсем небезопасно потому, что пароль и логин пользователя становятся известны клиентскому приложению.

Появились схемы, которые усиливают защиту доступа по авторскому коду.

Authorization Code Flow with Proof Key for Code Exchange (PKCE). Это усовершенствованный вариант Authorization code grant flow. В нем происходит генерация еще двух случайных кодов: Code Verifier и Code Challenge. Авторизация реализуется с посылкой Code Challenge, после введения пользователем логина и пароля авторизационным сервером отдается Authorization Code, далее идет обращение за аксес токеном с только что вы-

данным Authorization Code и с Code Verifier. В результате Identity-провайдер (Авторизационный сервер) валидирует на основании трех кодов.

Если злоумышленник получит один код из системы, он ничего не сможет сделать с ним — нужны все три кода. Примерами, где можно задействовать такой flow, являются Single Page Applications: Angular приложения, React JS приложения, и нативные приложения под Android/iOS.

Полный процесс описанный в rfc7637:

A. Клиент создает и записывает секрет с именем «code_verifier» и получает преобразованную версию "t(code_verifier)" (или «code_challenge»), которая отправляется по протоколу OAuth 2.0 в запросе авторизации вместе с методом преобразования "t_m".

B. Конечная точка авторизации отвечает как обычно, отправляя код авторизации, но записывает "t(code_verifier)" и метод преобразования "t_m".

C. Затем клиент отправляет код авторизации как обычно в точку токена авторизации, но включает сгенерированный секрет «code_verifier» из шага (A).

D. Сервер авторизации преобразует «code_verifier» и сравнивает его в "t(code_verifier)" из (B). Доступ запрещен, если они не равны.

Злоумышленник, перехватывающий код авторизации в точке (B), не может обменять его на токен доступа, так как не владеет секретом "code_verifier".

То есть перед запросом авторизации клиенту надо сформровать code_verifier и t(code_verifier) = code_challenge (на бэке приложения) и выслать на аторизирующий сервер вместе с методом преобразования t. Затем сервер авторизации вышлет code (запомнив code_challenge). Приложение-клиент (бэк приложения) далее должно выслать на авторизационный сервер code и code_verifier. Авторизационный сервер проверит code и t(code_verifier) сравнив с code_challenge. Обратно высылает access_token.

Данный вид авторизации использует два вида токенов, access и refresh токен. Позволяет удобно реализовать концепцию микросервисной архитектуры, где будет существовать отдельный сервер авторизации. Данный сервер выдает клиенту или клиентскому приложению access токен и refresh токен (хотя рефреш токен опционален). Добавление refresh токена позволяет миновать процедуру запроса пароля пользователя и ввода его вручную. Обычно предполагается создание авторизирующего сервера, который на логин и пароль выдает данные токены. Первый выдается на короткое время и служит для того, чтобы аутентифицироваться на ресурсных серверах, когда его время заканчивается, то используется уже refresh токен для его обновления. Обычно в такие токены включают сведения о пользователе, например его идентификатор и прочую информацию, чтобы не искать ее в базе. Refresh токен как уже было сказано нужен для того, чтобы обновление jwt токена было реализовано автоматически, по прошествии времени. Очевидно, что тут нужно использовать асинхронные методы обмена. При этом если злоумышленник получил каким-то образом access токен, то через какое-то время он станет невалидным, так как будет выпущен новый токен и старый уже не будет приниматься, в этом случае доступ может быть заморожен. При получении злоумышлинником refresh токена, он может обновить токен, но опять же у легального пользователя устареет его access токен и в то же время его refresh токен и ему придется авторизоваться, так же можно получив от того же пользователя устаревший рефреш токен можно заморозить доступ на всех устройствах. Для дополнительной защиты часто используют так называемый finger print, уникальную информацию с устройства пользователя или клиента, дополнительно проверяя и ее.

Хотя refresh токен не обязательно делать невалидным после обновления, но в большинстве случаев отправляют и новый refresh и access токен и время жизни и после истечения этого времени, аксес и рефреш токен обновляются, либо по отправке рефреш токена, либо при заходе со страницы авто-

ризации (хотя рефреш токен можно оставлять прежним, а можно как уже было сказано вовсе не использовать). Таким образом, например, если аксес токен стал невалидным по истечении времени, то пользователя или клиент пользователя отправят страницу авторизации или будет выслан рефреш токен. Если, например, злоумышленник перехватил рефреш токен, то у легального пользователя рефреш токен станет невалидным и его отправят на страницу авторизации с требованием ввести логин и пароль.

4.3.5.4 Open ID Connect

Протокол OpenID Connect (OIDC)

OIDC – семейство протоколов, являющихся расширением протоколов OAuth 2.0, позволяющих расширить их функционал путем более точного описания процесса аутентификации владельца ресурса и возможности клиенту получить информацию о нем. Это этапы (1) – (4) протокола OAuth 2.0.

Схематично протокол OpenID Connect можно описать следующим образом

1) Клиент отправляет серверу авторизации запрос аутентификации.

2) Сервер авторизации аутентифицирует конечного пользователя и получает согласие пользователя на доступ клиента к запрошенному ресурсу.

3) Сервер авторизации отвечает клиенту ID токеном и (опционально) токеном доступа.

4) Клиент может отправить серверу авторизации запрос информации о пользователе по токену доступа.

5) Сервер авторизации возвращает клиенту информацию о конечном пользователе.

Сервер авторизации OpenID Connect поддерживает три сценария аутентификации, реализующие этот сценарий: с генерацией кода авторизации (Authorization Code Flow), неявный сценарий (Implicit Flow), гибридный сценарий (Hybrid Flow). Передача сообщений между клиентом и сервером авторизации (на конечных точках авторизации, токена и UserInfo) должна производиться с использованием протокола TLS.

4.3.5.5 Авторизация SAML

4.3.6 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент.

Примером обращения к ресурсу на другом домене может быть использование тэга <img src = “http://domenb.com/image.jpg”> на странице размещенной в домене domena.com.

4.4 Протокол 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 *(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.

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.

4.4.1 Пример поддержки HTTPS на python Flask

Добавить поддержку Https в приложениях на Flask можно следующим образом:

```
from flask import Flask
import ssl
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('server.crt', 'server.key')
app = Flask(__name__)
@app.route('/')
def index():
    return 'Hello, world!'
if __name__ == '__main__':
    app.run(host='127.0.0.1', debug=True, ssl_context=context)
```

В коде используется сертификат сервера и его секретный закрытый ключ.

Но такой способ является не очень хорошим способом реализовывать поддержку защищенного соединения на базе самого веб-приложения, лучше

воспользоваться для этого proxy серверами, например, NGINX https://nginx.org/ru/. Кроме того, понадобится ssl/tls сертификат, его можно купить или взять простенький сертификат на сайте подобном https://www.sslforfree.com/.

Для тестирования приложения на localhost, 127.0.0.1 можно воспользоваться созданием локального сертификата, самоподписанного сертификата, с помощью openssl, это утилита командной строки, которая позволяет создавать различные виды сертификатов, например, PKI или HTTPS. Сгенерированный сертификат необходимо будет добавить в доверенное хранилище браузера.

Так же можно воспользоваться различными средствами, такими как mkcert, minica.

4.4.2 Пример создания самоудостоверяющего сертификата с использованием openssl

Рассмотрим пример создания сертификатов с помощью openssl и напомним принцип использования подобных сертификатов.

Как вы помните асимметричное шифрование предполагает использование закрытого и открытого ключа, закрытый ключ расшифровывает сообщение, отрытый ключ шифрует, при этом открытый ключ может быть известен кому угодно. По открытому ключу нельзя восстановить закрытый. Применение закрытого ключа к тексту зашифрованному открытым ключом даст возможность получить открытый текст, и наоборот T=З(O(T)), T=O(З(T)). Таким образом, можно создать, например, цифровую подпись, зашифровать закрытым ключом какой-то текст, и отправить зашифрованную и открытую часть этого текста. Только расшифровка открытым ключом даст идентичные тексты и таким образом можно будет проверить подлинность документа.

Сертификаты SSL проверяются и выдаются **Центром сертификации (CA - Certificate Authorities)**. Вы подаете заявку, генерируя **CSR (Certificate Signing Request - запрос на получение сертификата)** с парой ключей на

вашем сервере. CSR содержит важные организационные детали, которые CA проверяет, то что это действительно ваш домен и так далее, сам по себе запрос на получение сертификата содержит в себе полное доменное имя вашего сайта, полное юридическое название организации, отдел в вашей организации, который занимается сертификатом, город нахождения организации, регион, страна (в виде двух буквенного обозначения), адрес электронной почты, публичный ключ, который должен быть подписан проверяющим центром сертификации. После проверки всей информации о вашем сайте Центр сертификации выдает сертификат, в котором подписан цифровой подписью, переданный вами публичный ключ, таким образом, любой владелец открытого ключа, соответствующий закрытому ключу официального центра сертификации, которым был подписан ваш сертификат, может проверить, что это действительно ваш публичный ключ, и это действительно ваш сайт. С помощью OpenSSL можно создать не только запрос на сертификат, но и создать закрытый и открытый ключи, а также сгенерировать самоподписанный сертификат, не обращаясь к официальному центру сертификации, в этом случае вашему сертификату сможете доверять только вы сами, другие сайты, пользователи и браузеры не будут принимать данный сертификат. Самоподписанный сертификат нужно будет поместить в доверенное хранилище ваших браузеров или туда где в вашей операционной системе хранятся сертификаты. Самоподисанный сертификат как уже было сказано может понадобиться в целях тестирования разрабатываемого веб-сервиса или в интрасети.

Сгенерируем закрытый ключ для нашего собственного «центра сертификации».

openssl genrsa -out rootCA.key 2048

Команда genrsa позволяет генерировать закрытый ключ, в данном случае длиной 2048 бит. Команда rsa, например, позволяет по закрытому ключу сгенерировать открытый.

Далее необходимо создать наш открытый публичный сертификат, который будет использоваться центром сертификации и его клиентами для подтверждения легальности публичных ключей серверов и их сертификатов.

Для этого используется команда req.

Данная команда при создании сертификата может использовать интерактивный режим для ввода информации или считывать данные с конфигурационного файла. Создадим конфигурационный файла req.cfg.

```
# Секция основных опций
[ req ]
default_bits = 2048 # Число бит ключа
default_keyfile = rootCA.key # Имя ключа, используемого для сертификата
distinguished_name = cad_name # DN организации, выдавшей сертификат
x509_extensions     = req_ext
prompt = no # Брать параметры из конфигурационного файла, неинтерактивный режим
# DN организации
[ cad_name ]
CN=RU # Страна
ST=Tomskaya # Область
L=Tomsk # Город
O=TUSUR # Название организации
OU=ASU # Название отделения
CN=HomeSayCA # Имя для сертификата (персоны, получающей сертификат)
emailAddress=mailmail@my.home.address.org # E-mail организации
[ req_ext ]
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints       = critical, CA:true      # центр сертификации
keyUsage               = keyCertSign, cRLSign
```

И запустим команду

openssl req -x509 -new -key rootCA.key -config req.cfg -out rootCA.crt -days 3

Здесь указано файл закрытого ключа для создания сертификата, и время действия сертификата 3 дня, конфигурация данных берется из файла req.cfg, можно убрать этот файл и вводить данные с клавиатуры.

Информацию о вашем CA сертификате можно просмотреть, используя команду

openssl x509 -in rootCA.crt -noout -text.

Далее можно создать запрос на сертификацию server.csr и приватный ключ для вашего сервера server.key.

Создадим для этого новый конфигурационный файл reqserver.cfg.

```
# Секция основных опций
[ req ]
default_bits = 2048 # Число бит ключа
default_keyfile = server.key # Имя ключа, используемого для сертификата
distinguished_name = server_cad_name # DN организации, выдавшей сертификат
req_extensions  = server_req_ext
prompt = no # Брать параметры из конфигурационного файла, неинтерактивный режим
# DN организации
[ server_cad_name ]
CN=RU # Страна
ST=Tomskaya # Область
L=Tomsk # Город
O=TUSUR # Название организации
OU=ASU # Название отделения
CN=localhost # Имя для сертификата (персоны, получающей сертификат)
emailAddress=servermailmail@my.home.address.org # E-mail организации
[ server_req_ext ]
subjectAltName      = @alternate_names
# альтернативные доменные имена для самоподписываемого сертификата
[ alternate_names ]
DNS.1  = localhost
IP.1  = 127.0.0.1
# расширения при запуске команды x509
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE # не используется для подписи сертификатов
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
subjectAltName=@alternate_names
```

Запускаем команду.

openssl req -nodes -new -newkey rsa:2048 -keyout server.key -config reqserver.cfg -out server.csr

Будет создан секретный ключ server.key и запрос на сертификат server.csr.

Осталось нашему импровизированному CA подписать сертификат сервера.

Выполняем команду

openssl x509 -req -in server.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -extensions v3_ext -extfile reqserver.cfg -out server.crt -days 3

Команда x509 создает на основе запроса на сертификацию секретного ключа центра сертификации подписанный сертификат для нашего сервера.

Здесь в данной схеме центр сертификации мы сами, в реальности можно их разнести и сделать доверенный центр в интранете.

В результате создается файл server.crt, его мы и используем совместно с файлом server.key в нашем приложении. Но пока браузеры не доверяют нашему сертификату. Например, в firefox можно добавить rootCA.crt в настройках→сертификаты→просмотр сартификатов→центры сертификации→импортировать.

В данной команде указывается корневой сертификат, корневой закрытый ключ, файл для подключения расширений при создании самоподписываемого сертификата, и секции в этом файле с дополнительными расширениями, и количество дней действия сертификата.

4.5 Протокол 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

Целями протокола являются:

Добавить механизмы согласования протокола, клиент и сервер могут использовать HTTP 1.1, 2.0 или, гипотетически, иные, не HTTP-протоколы.

Поддержать совместимость со многими концепциями HTTP 1.1, например, по набору методов доступа (GET, PUT, POST и т. п.), статусным кодам, формату URI, большому количеству заголовков.

Уменьшить задержки 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;

Сохраненить совместимость 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.

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. Каждое соединение при этом требует прохождения процедуры тройного рукопожатия, на что тратится время. Также соединение с более объемным контентом, может блокировать другие соединения или какое-то соединение будет медленным. Но 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.

Следует, наверное, отметить, что мультиплексирование допускает так же протокол SCTP, но он не настолько популярен как TCP, кроме того, SCTP допускает мультихоиминг, то-есть объединение сетевых интерфейсов для передачи общего трафика.

Приоритизация

Вместе с мультиплексированием появилась приоритизация трафика. Запросам можно назначить приоритет на основе важности и зависимости. 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.

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 нужно отказаться или модифицировать 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.

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 DataURI. Он также может быть полезен в HTTP/2, но точно будет менее эффективным, чем в случае с прошлой версией.

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.

4.6 Технологии формирования динамического контента

Технологии формирования динамического контента позволяют на стороне сервера сформировать готовую HTML страницу и предоставить ее пользователю. Данные технологии весьма разнообразны, какие-то из них более популярны, чем другие, все они базируются на каких-то языках программирования. Передает такую страницу обычно веб-сервер, отдающий запрос пользователя приложению, которое может быть написано на различных языках и формирует страницу или контент, связь веб-сервера и приложения реализуется с помощью различных интерфейсов. Рассмотрим основные технологии подобного рода.

4.6.1 Технология 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».

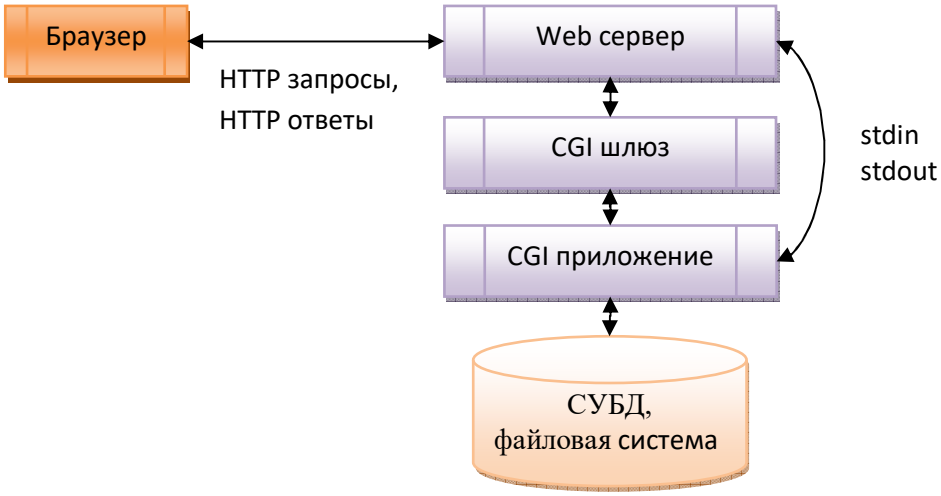


Рисунок 4.3 – Схема работы CGI

Общие принципы работы CGI (рисунок 4.3) можно описать следующим образом:

1. Клиент запрашивает CGI-приложение по его URI.

2. Веб-сервер принимает запрос и устанавливает переменные окружения, через них приложению передаются данные и служебная информация.

3. Веб-сервер перенаправляет запросы через стандартный поток ввода (stdin) на вход вызываемой программы.

4. CGI-приложение выполняет все необходимые операции и формирует результаты в виде HTML.

5. Сформированный гипертекст возвращается веб-серверу через стандартный поток вывода (stdout). Сообщения об ошибках передаются через stderr.

6. Веб-сервер передает результаты запроса клиенту.

Так как данный интерфейс использует стандартный поток ввода вывода, то может быть использован для приложений на любом языке.

Вот пример типичной CGI программы на python [39]:

вначале выводятся заголовки

```
print("Content-Type: text/html") # Определение типа HTML
print("\r\n") # Пустая строка означает окончание передачи заголовков
```

теперь выводим основной текст в формате HTML

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

Пример на Си может выглядеть так:

```
#include <stdio.h>
int main(void) {
  printf("Content-Type: text/plain\n\n");
  printf("Hello, world!\n\n");
  return 0;
}
```

Подход позволяет использовать практически любой язык программирования, но при этом неудобен тем, что требует заполнения заголовков, контента в строковом виде, это не очень удобно. Бывает достаточно затратным и требовательным к ресурсам из-за постоянной загрузки соответствующего исполнительного модуля приложения, каждый раз порождается новый процесс. В настоящее время редко где используется.

4.6.2 PHP

Скриптовый язык общего назначения, интенсивно применяемый для разработки веб-приложений [40]. В настоящее время поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов.

Преимуществами данной технологии является традиционность (исторически сложилось, что он с самого начала создавался для веб сервисов), гибкость, простота (очень простой синтаксис), безопасность (хотя было время, когда на php сервера успешно совершались атаки на переполнение буфера), эффективность. Сам язык является интерпретируемым и использует динамическую типизацию.

Недостатки связаны с неоднозначностью кода, моделью работы с памятью, отсутствием статической типизации.

Основные возможности – это:

автоматическое извлечение POST- и GET-параметров, а также переменных окружения веб-сервера в предопределённые массивы;

взаимодействие с большим количеством различных систем управления базами данных через дополнительные модули (MySQL, MySQLi, SQLite, PostgreSQL, Oracle (OCI8), Oracle, Microsoft SQL Server, Sybase, ODBC, mSQL, IBM DB2, Cloudscape и Apache Derby, Informix, Ovrimos SQL, Lotus Notes, DB++, DBM, dBase, DBX, FrontBase, FilePro, Ingres II, SESAM, Firebird / InterBase, Paradox File Access, MaxDB, Интерфейс PDO, Redis);

автоматизированная отправка HTTP-заголовков;

работа с HTTP-авторизацией;

работа с cookies и сессиями;

работа с локальными и удалёнными файлами, сокетами;

обработка файлов, загружаемых на сервер;

работа с XForms.

Новая версия PHP 8 в 2020 должна обеспечивать jit-компиляцию.

Пример php скриптов уже приводился в нашем пособии.

4.6.3 JSP

Java Server Pages *- это технология Java2 Platform, Enterprise Edition (J2EE) для создания приложений, генерирующих динамическое web-содержимое - HTML, DHTML, XHTML и XML.*

Технология Java Server Pages даёт возможность легко создавать динамическое содержимое web-страниц, предельно мощное и гибкое.

Эта технология основывается на Template Data\Шаблонные Данные – некоторая часть динамического содержимого является фиксированным, или шаблонным, содержимым (фрагменты текста или XML являются типичными шаблонными данными).

JSP-технология поддерживает естественное манипулирование шаблонными данными, возможность добавлять динамические данные к шаблонным

данным, два механизма для инкапсуляции функциональности: архитектура компонентов JavaBeans и библиотеки тэгов.

Хорошая поддержка утилитами ведёт к значительному повышению производительности. Соответственно, JSP технология имеет средства для создания качественных утилит авторизации. Тщательная проработка этих концепций привела к созданию гибкой и мощной серверной технологии.

JavaServer Pages (JSP) позволяют отделить динамическую часть страниц от статического HTML. Пишется обычный код в HTML, с использованием для этого любой программы для создания Web страниц. Затем динамическая часть кода заключается в специальные тэги, большинство которых начинаются с < и завершаются >. Например,

Имя вашего хоста: <%= request.getRemoteHost() %>

Сам по себе язык Java является объектно-ориентированным и строго типизированным, освобождение выделенной памяти под объект реализуется автоматически сборщиком мусора, программа предварительно транслируется в промежуточный байт код, а затем исполняется jit-компилятором на лету машиной Java JVM (Java virtual machine). Это делает данную технологию платформо-независимой.

JSP страницы имеют расширение .jsp и размещаются там же, где и обычные Web-страницы. Структура таких страниц может состоять из пяти конструкций: HTML, комментарии, скриптовые элементы, директивы и действия. JSP-страница при компиляции преобразуется в сервлет со статическим содержимым, которое направляется в поток вывода, связанный с методом service. Поэтому при первом запросе этот процесс может вызвать небольшую задержку. Комментарии в документе или программе не являются причиной замедления программы, так как транслятор и исполнитель их игнорируют. Скриптовые элементы позволяют указать код на языке Java, который впоследствии станет частью конечного сервлета, директивы дают возможность управлять всей структурой сервлета, а действия служат для задания существующих используемых компонентов, а также для контроля над поведением

движка JSP. Для упрощения работы со скриптами имеются заранее определённые переменные, такие как request, response, pageContext, session, out, application, config, page, exception.

Спецификация JSP различает три типа скриптовых элементов:

Объявления <%! одна или несколько деклараций %>

Выражения <%= одно выражение %>

Скриплеты <% скриплет %>

Объявления обычно используются для определения переменных, методов, внутренних классов и остальных действующих Java конструкций на уровне класса.

```
<%! private int accessCount = 0 ;%>
Количество обращений к странице с момента загрузки сервера: <%=++accessCount %>
```

Выражения становятся аргументами для метода out.print(). Для вставки значений в вывод. Пример выводит текущее время и имя хоста.

```
Текущее время: <%= new java.util.Date() %>
Имя вашего хоста: <%= request.getRemoteHost() %>
```

С помощью скриплетов в JSP страницы вкладываются работающие части Java-кода, имеют доступ к переменным в объявлениях.

```
<%String queryData = request.getQueryString();
out.println("Дополнительные данные запроса: "+ queryData);%>
```

Кроме того, существуют еще директивы. Представляющие собой указание на действие с определенным атрибутами.

```
<%@ директива атрибут1 = "значение1"
атрибут2 = "значение2"
...
атрибутN = "значениеN"
%>
```

Существует три основных типа директив: page, которая позволяет совершать такие операции, как импорт классов, изменение суперкласса сервлета, и т. п.; include, которая даёт возможность вставить файл в класс сервлета при трансляции JSP файла в сервлет; и taglib, позволяющий расширить множество тегов своими собственными, которые JSP контейнер способен истолковать.

Итак, страница JSP содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов

HTML, SVG, WML, или XML, и JSP-элементы, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также Expression Language (EL), для внедрения Java-кода в статичное содержимое JSP-страниц.

Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц Jasper, и затем компилируется в байт-код виртуальной машины Java (JVM). Контейнеры сервлетов, способные исполнять JSP-страницы, написаны на платформенно-независимом языке Java. JSP-страницы загружаются на сервере и управляются из структуры специального Java server packet, который называется Jakarta EE Web Application. Схематичное изображение представлено ниже. Обычно страницы упакованы в файловые архивы .war и .ear.

4.6.4 ASP .Net

Данная технология также является по сути платформо-независимой, так как программа веб-приложения написанного на каком-либо языке транслируется в байт код CIL (Common Intermediate Language, ранее MSIL (Microsoft intermediate language)), это код исполняется с помощью CLR (common language runtime) путем jit-компиляции [41]. Первоначально ASP (англ. Active Server Pages — «активные серверные страницы») — технология, как уже стало понятно внимательному читателю была предложена компанией Microsoft в 1996 году для создания Web-приложений. Эта технология также, как и PHP основана на внедрении в обыкновенные веб-страницы специальных элементов управления, допускающих программное управление.

Технология ASP разработана для операционных систем из семейства Windows NT и функционирует под управлением веб-сервера Microsoft IIS.
И как раз ASP получила своё развитие в виде ASP.NET — платформы разработки веб-приложений, в состав которой входит: веб-сервисы, программная инфраструктура, модель программирования, от компании Майкрософт. ASP.NET входит в состав платформы .NET Framework.

ASP.NET на Pyhton

Одной из основных реализаций языка Python, предназначенной для платформы Microsoft .NET является IronPython, который может быть использован как основной язык для разработки ASP.NET-приложений.

Пакет интеграции IronPython и ASP.NET распространяется с открытым исходным кодом под лицензией Apache 2.0.

Пакет для ASP.NET поставляется с примером приложения. (Приведённый фрагмент кода реализует два обработчика для ASP.NET: загрузки страницы и нажатия на кнопку):

```
def Page_Load(sender, e)
  if not IsPostBack:
    Label1.Text = "...Your name here..."
def Button1_Click(sender, e)
  LAbel1.Text = TextBox1.Text
```

4.6.5 WSGI

WSGI (Web Server Gateway Interface) является стандартом взаимодействия между Python-программой, выполняющейся на стороне сервера, и самим веб-сервером [42].

Для Python существует много различного рода веб-фреймворков и библиотек. Из-за разного интерфейса они не умеют взаимодействовать между собой, поэтому при выборе одного фреймворка разработчик может оказаться вынужден использовать определенный веб-сервер и наоборот. WSGI — простой интерфейс между веб-сервером и python-сервером, при его использовании проблем совместимости не возникает.

По стандарту, WSGI-приложение должно удовлетворять следующим требованиям:

1. Должно быть вызываемым (callable) объектом (обычно это функция или метод).

2. Принимать два параметра: словарь переменных окружения (environ) и обработчик запроса (start_response).

3. Вызывать обработчик запроса с кодом HTTP-ответа и HTTP-заголовками.

4. Возвращать итерируемый объект с телом ответа.

Простейшим примером WSGI-приложения может служить такая функция-генератор:

```
def simplest_wsgi_app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    yield 'Hello, world!'
```

Помимо приложений и серверов, стандарт дает определение middleware-компонентов, предоставляющих интерфейсы как приложению, так и серверу. То есть для сервера middleware является приложением, а для приложения — сервером. Это позволяет составлять «цепочки» WSGI-совместимых middleware. Middleware - простая обертка над объектом приложения, реализовать её можно через замыкание.

Middleware могут брать на себя следующие функции (но не ограничиваются этим):

1. Обработка сессий.

2. Аутентификация/авторизация.

3. Управление URL (маршрутизация запросов).

4. Балансировка нагрузки.

5. Пост-обработка выходных данных (например, проверка на валидность).

6. Изменение заголовков.

4.6.6 Фреймворки для веб-приложений

Для облегчения разработки веб-приложений в том числе на Python применяются фреймворки.

Существует множество фреймворков для реализации веб-приложения на Python.

Фреймворк *- это каркас приложения, позволяющий наполнить его необходимой функциональностью.*

Django

Универсальный фреймворк [43]. Он может использоваться при создании любых сайтов, поскольку умеет доставлять веб-контент в различных форматах (например, JSON, XML, RSS). Также он дружественен к начинающим разработчикам. Даже если вы в Python еще новичок, с Django вы справитесь без всяких проблем.

Flask

Микрофреймворк, и поэтому большее внимание в нем уделено простоте работы, а не функциям. Основная идея Flask заключается в том, чтобы помочь создать прочную основу веб-приложений [44].

Лучше всего подходит для веб-разработчиков, которые хотят использовать самые лучшие практики, добиться быстрого прототипирования и создавать автономные приложения.

Pyramid

Полностью «open-source» каркас для создания Python-приложений. Его основная цель - сделать как можно больше с минимальной сложностью. Самой яркой особенностью Pyramid является его способность хорошо работать как с небольшими, так и с большими приложениями [45]. Он отлично подходит для однофайловых приложений, а еще он хорош в URL-адресах, расширении конфигурации, тестировании и документации по данным.

Лучше всего он подходит для тех, кто занимается разработкой API, а также прототипированием и разработкой крупных веб-приложений, таких как CMS.

4.7 Язык Python

Так как в нашей дисциплине за базовый язык и технологию создания веб-контента взят Flask и язык Python, то рассмотрим основные особенности данного языка.

Язык подходит для быстрой реализации какого-либо проекта, осуществления какой-либо идеи, оценки ее работоспособности. У языка высокая динамичность, возможность изменять любые объекты и возвращать любые объекты.

Множество реализованных библиотек для решения научных задач, машинного обучения, нейронных сетей, обработки данных, больших данных, библиотеки и фреймворки реализации сетевых сервисов, распределенных вычислений и тензорных вычислений. Возможности обращения к библиотекам Си, различным СУБД. Есть свой JIT-компилятор PyPy. На Python есть множество библиотек для решения научных задач Keras, Tensorflow, Theano, Skikit-Learn, pandas, matplotlib. Это интерпретируемый язык, хотя для него существуют вариант jit-компилятора pypy, а также возможность транслирования в исполнимый модуль, так же есть трансляторы в CIL и Java байт код.

Развитие языка Python происходит согласно чётко регламентированному процессу создания, обсуждения, отбора и реализации документов PEP. PEP - Python Enhancement Proposal - это предложения по развитию питона https://www.python.org/dev/peps/. Процесс PEP является основным механизмом для предложения новых возможностей и для документирования проектных решений, которые прошли в Python.

Самым известным PEP является PEP8 - это свод рекомендаций по оформлению кода (https://www.python.org/dev/peps/pep-0008/).

Python лаконичный, что дает возможность получать короткий код, который еще и легко читаемый. Практически все, что делает ниже приведенный код можно понять, зная английский язык.

```
def magic(top):
    acc = []
    for entry in os.scandir(top):
        if entry.is_file() and entry.name.endwith(".py"):
```

```
            acc.append(entry.path)
    return acc
```

В Python есть возможность интроспекции, то есть можно получить и изменить данные о программном объекте во время исполнения.

Необходимые для интроспекции данные хранятся в специальных атрибутах. Так, например, получить все пользовательские атрибуты большинства объектов можно из специального атрибута - словаря (или из другого объекта, предоставляющего интерфейс dict) __dict__.

```
>>> class x(object):pass
>>> f = x()
>>> f.attr = 12
>>> print(f.__dict__)
{'attr': 12}
>>> print(x.__dict__)
```

Так как классы тоже являются экземплярами объекта type, то они поддерживают этот тип интроспекции

```
{'__dict__': <attribute '__dict__' of 'x' objects>, '__module__' .....}
```

Получение кода и байткода функции.

```
def f(x):pass
print(f.__code__)
print(f.__code__.co_code)
```

Получение интроспекции на функцию. Модуль inspect.

```
import inspect
def function(x, y = 10)
   return x**2
val = inspect.getfullargspec(function)
val1 = inspect.signature(function)
print(f'{val}')
print(f'signature {val1}')
```

Получим:

```
FullArgSpec(args=['x', 'y'], varargs=None, varkw=None, defaults=(10,), kwonlyargs=[],
kwonlydefaults=None, annotation={}) signature (x, y=10)
```

Здесь мы видим список аргументов, и значение по умолчанию для одного из аргументов, при этом существует правило, что если не задать значения по умолчанию между параметрами, для которых задано значение по умолчанию то возникает ошибка.

```
import inspect
def fun(x, y=10, *unnamed, par1=4, par2=3, **named)->int:
   for val in unnamed:
```

```
        print(f'unnamed value {val}')
    for key, val in named.items():
        print(f'named value {key} = {val}')
    return x+y
print(inspect.getfullargspec(fun))
print(fun(2, 2, 'ul', 1, 2, par1=2, par2=2, name1=1, name2='s'))
```

Здесь выводятся неименованные неуказанные *args (*unnamed) и именованные неуказанные аргументы **kwargs (*named) позволяющие передать дополнительные произвольные значения параметров. В принципе * и ** выступают как разыменование списка или словаря соответственно, таким образом, внутри функции можно обратиться к нескольким переданным значениям, а при вызове можно передать произвольное количество параметров. Здесь передаются дополнительно неименованные значения ul, 1, 2 и именованные name1, name2, между ними находятся именованные указанные переменные функции par1, par2.

Результат работы скрипта.

```
FullArgSpec(args=['x', 'y'], varargs='unnamed', varkw='named', defaults=(10,), kwonlyargs=['par1', 'par2'],
kwonlydefaults={'par1': 4, 'par2': 3}, annotations={'return':<class 'int'>})
unnamed value u1
unnamed value 1
unnamed value 2
named value name1 = 1
named value name2 = s
```

4.7.1 Типы и структуры данных в Python.

4.7.1.1 Аннотации типов.

Необходимы для анализа кода, контроля типов, несмотря на динамическую типизацию, например, с помощью mypy.

```
python -m mypy test_type.py
```

Существует три способа аннотации переменных

1. `var = value  # type: annotation`

2. `var: annotation; var = value`

3. `var: annotation = value`

Примеры

```
>>> name "John" # type: str  #### Автоматическая аннотация
>>> name: str; name = "John" #### Вручную в 2 команды
>>> name: str = "John"       #### То же самое в 1 команду
```

Пример аннотации для функции

```
def func()->int
```

4.7.1.2 Простые типы

Различают типы `int`, `float`, `bool`, `str`, `complex`...

Каждый тип является объектом класса type.

```
>>> type(int)
<class 'type'>
```

Тип объекта определяется автоматически, здесь тип int.

```
>>> type(2+2)
<class 'int'>
```

Здесь тип complex для комплексных чисел.

```
>>> d = (1+2j)
>>> type(d)
<class 'complex(2.0)'>
```

4.7.1.3 Доступ к идентификатору объекта

В Python все является объектом, все является выражением (даже, например, определение функции - это выражение), поддерживается парадигма ООП, поддерживается функциональная парадигма.

None. Аналог NULL, но является полноценным объектом

```
>>> res = print(None) # Любая функция принимая None вернет None
None
>>> res == None      # Сравнивает значения
True
>>> res is None      # позволяет проверить данный ли объект
True
```

Получение идентификатора объекта.

```
>>> id(None)         # В CPython – адрес, is сравнивает фактически адреса
140503861072
```

4.7.1.4 Логический тип

Принимает значения `True` и `False`. На нем определены:

- операции алгебры логики `not`, `and`, `or` …
- операции сравнения `>`, `<`, `==`, `>=`, `<=` результатом их применения будет или True или False.

```
>>> to_be = False
>>> to_be or not to_be    # Получается использовать слова вместо значков
True
>>> x = 1
>>> y = 2
>>> x**2 + y**2 < 5 == True    # True синглтон, не делайте так
False
>>> x**2 + y**2 < 5 is True    # И так тоже не делайте
False
```

Лучше делать так:

```
>>> x**2 + y**2 < 5
False
```

Пример работы логических операций, если первый аргумент операции and (И) ложь, то ясно, что все выражение ложно, потому второй операнд не вычисляется.

```
>>> False and print('also')
False
```

Видим, что тут печать на экран уже выполняется.

```
>>> res = True and print('also')
also
>>> assert res is None, "print should return None"  # Assert предикат для проверки выполнения условия
```

Далее в примере для or при первом ложном аргументе, необходимо проверить второй аргумент, потому результатом будет второй аргумент. Если бы первый аргумент был True, то второе выражение бы не вычислялось.

```
>>> False or 92    # Работает для любого значения
92
```

Операции Python. Присвоение.

```
>>> flag = True
>>> flag
True
```

Арифметическая сумма двух истинных значений даст 2.

```
>>> flag + True
2
```

Арифметическая сумма истинного и ложного значения даст 1.

```
>>> flag + False
1
```

Приведенный выше пример Assert применяется для отлавливания ошибок, вызывая исключение, если какое-то условие было нарушено.

Рассмотрим функцию, в которой цена должна быть в определенных пределах, если это не так, будет вызвано исключение.

```
def apply_discount(product, discount):
    price = int(product['цена'] * (1.0 - discount))
    assert 0 <= price <= product['цена']
    return price
```

Вызываем.

```
>>> apply_discount(shoes, 0.25)   # Все верно, скидка 25%
11175
>>> apply_discount(shoes, 2.0)    # Скидка в 200% вызовет ошибку исполнения
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    apply_discount(prod, 2.0)
  File "<input>", line 4, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

4.7.1.5 Truthy/Falsy объекты.

Те объекты, которые воспринимаются как значения True или False.

```
>>> bool(True) True
>>> bool(0) False
>>> bool(1) True
>>> bool([]) False
>>> bool([0]) True
```

Итак, пустой список воспринимается как ложное значение в условных операторах или логических операциях, не пустой как истинное.

Таким образом, например, такой способ является плохим стилем:

```
if len(xs) == 0:   # Плохо
  pass
```

Лучше делать так:

```
if xs:
  pass
if not xs:
  pass
```

4.7.1.6 Операции над числами.

Возведение в степень 2**10 (== 1024),

Деление нацело 15//2 (== 7),

Вещественное деление 15/2 (== 7.5),

Взятие остатка 15%2 (== 1).

Стоит обратить внимание, что получение остатка реализуется как в алгебре, на остаток накладывается условие положительности.

```
>>> -3 // 2
-2
>>> -1 % 3    # C/Java/Rust скажут -1
2
```

Проверьте

```
d = x//y = -1//3 = -1
r = x%y = -1%3 = 2
x = r+d*y = 2+(-1)*3 = -1
```

Дополнительный синтаксис.

```
>>> x = 10
>>> 0 <= x and x < 100
```

```
True
>>> 0 <= x < 100
True
```

4.7.1.7 Списки.

Упорядоченный список элементов [], [1, 2, 'd'].

Взятие значения по индексу some_list[0].

Присвоение по индексу some_list[0] = 'Hello, world!'

Умножение дублирует [1, 2] * 3 = [1, 2, 1, 2, 1, 2]

Взятие длины len()

Оператор in, наличие данного элемента item in list

```
>>> 1 in [1, 2, 3]
True
```

Методы списка.

- добавить в конец .append()
- удалить элемент (удаляет последний по умолчанию) .pop()
- создать новый список и объединить с другим .insert(oldL, newL)

```
>>> xs = [[0] * 3] * 3    # Не делайте так, продублируются ссылки на изначальный список
>>> xs
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> xs[0][0] = 1    # Нежелательное изменение в 2 других местах
>>> xs
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]

>>> xs += [1]    # Не надо так делать
>>> xs
[92, 2, 3, 1]
```

Так лучше не делать, так как данная операция выполняется долго, создавая новый список, куда копируется старое и новое содержимое списка. Изначально список в python расширяющийся массив, и добавление элемента может быть реализовано в расширенную область с помощью append без копирования. Лучше xs.append(1)

4.1.7.8 Срезы.

С их помощью можно производить выборку элементов списков

по такому правилу item[START:STOP:STEP]

Параметры могут опущены (хоть все), могут быть отрицательными

```
>>> xs
[0, 1, 2, 3, 4]
>>> xs[-1]     # xs[len(xs) - 1]
4
>>> xs[2:4]    # Start Stop
[2, 3]
>>> xs[:-2]    # Stop len(xs)-2
[0, 1, 2]
>>> xs[::2]    # Step 2 от 0 до последнего элемента.
[0, 2, 4]
>>> xs[:]      # Defaults
[0, 1, 2, 3, 4]
>>> new_list = old_list    # скопируется ссылка на old_list
>>> new_list = old_list[:] # Это поэлементно скопирует список
>>> id(new_list) == id(old_list)
False
```

4.1.7.9 Класс строки.

Конкатенация строк str1 + str2.

Умножение строк по аналогии со списком str * 3 – репликация.

Наличие подстроки в строке substr in str.

```
>>> "Hello " + "world!"    # Конкатенация
'Hello world!'
>>> "multi " * 3           # Умножение
'multi multi multi '
>>> "world" in "Hello, world!"
True
```

Методы

Разбиение по стокам `.splitlines()`

Разбиение по словам `.split()`

Убрать пробелы `.strip()`

Соединить список строк через разделитель `.join()`

Явное приведение к типу `.str()`

Длина `.len()`

```
>>> "Hello \nworld".splitlines()    # По строкам
['hello ', 'world']
>>> "a, b, c".split()               # По словам
['a', 'b', 'c']
>>> ", ".join(["a", "b", "c"])      # Соединить через разделитель
'a, b, c'
>>> str(123)                        # Приведение типа
'123'
>>> len("some_string")              # Не считая символа конца строки
11
```

Форматирование строк.

"%some_format" % (some_var)

% - оператор форматирования

```
>>> print("%d :: %s :: %d" % (flag, str, val))   # Выглядит почти по аналогии с Си
```

d – определяет число, s – строку.

В результате будет выведено.

```
1 :: some_str :: 123
```

Можно модифицировать оператор формата, указывая количество символов на вывод параметра. Видим, что после True идет заполнения еще 4-мя пробелами.

```
>>> print("%8s :: %s :: %05d" % (flag, str, val))
True    :: some_str :: 00123
```

Дополнительный вариант форматирования строк - метод формат.

"{Номер значения 1} {Номер значения 2} ...".format(значения по порядку)

```
>>> s = "{} {} {}?".format("what", "is", "it")   # Автоматический порядок
>>> s
'what is it?'
>>> s = "{2} {1} {0}?".format("what", "is", "it")   # Явно указанный порядок
>>> s
'it is what?'
>>> s = "{:>7} {} {}?".format("what", "is", "it")   # Формат фнутри
'   what is it?'
>>> s = "{2:>7} {1} {0}?".format("what", "is", "it")  # Формат внутри + явное указание порядка
'    it is what?'
```

Здесь знак >, <, ^ может быть использован как выравнивание по правому, левому краю и центру.

```
>>> print("{0:^15}\n{1:^15}\n{2:^15}".format("*","*"*3,"*"*5))
      *
     ***
    *****
```

F - строки появились в версиях Python 3.6+, они позволяют встраивать вычислимое выражение внутрь f строки с помощью фигурных скобок {}.

Напомним join — склеивание списка в строку. split разбиение строки на элементы списка.

```
>>> print("".join([f"{'*'*(i*2+1):^15}\n" for i in range(5)]))
      *
     ***
    *****
   *******
  *********
>>>x = 9
>>> print("".join([f"{'*'*(i*2+1):^{x*2-1}}\n" for i in range(x)]))
```

Ниже приведен пример использования лямбда выражения и рекурсии.

```
>>> treg = lambda x,y: "" if x==-1 else treg(x-1,y)+f"{'*'*(x*2+1):^{y}}\n"
print(treg(7,7*2+1))
```

Выполнение вывода одной командой

```
exec('treg = lambda x,y,c: "" if x==-1 else treg(x-1,y,c)+f"{(c)*(x*2+1):^{y}}\\n"\nprint(treg(6,6*2+1,"x"))')
```

Выполнение вывода одной командой с использованием Y комбинатора для реализации рекурсии в лямбда исчислении.

```
print((lambda a:lambda v:a(a,v))(lambda s,x:"" if x==0 else s(s,x-1)+f"{'*'*(2*x-1):^15}\n")(8))
```

Во многом работа со строками похожа на работу со списками, но в отличии от списков, строки не изменяемы.

```
>>> s = "Hallo!"
>>> s[1] = "e"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Однако, это опять же можно обойти, используя слайсеры.

```
>>> s = "Hallo!"
>>> s = s[0] + 'e' + s[2:]
>>> s
'Hello!'
```

4.1.7.10 Tuple – класс кортежа.

Те же операции как у списка, но кортеж неизменяемый объект.

пустой кортеж (),

кортеж из одного элемента (1,), просто (1) без запятой воспринимается как число.

скобки опциональны date = "September", 2018

```
>>> date = ("September", 2018)
>>> len(date)
2
```

Попытка изменения элемента кортежа вызывает ошибку.

```
>>> date[1] = 2019
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does bot support item assignment
```

Но можно изменить изменяемый объект внутри кортежа, например, список.

```
>>> xs = ([], [])
>>> xs[0].extend([1, 2, 3])    # Это работает, т.к. мы не меняем саму ссылку на объект
>>> xs
([1, 2, 3], [])
```

Функция возвращает кортеж значений (может возвращать набор объектов). Пример возвращения кортежа функцией. Реализация векторной функции векторного аргумента.

```
>>> def div_mod(x, y):
...     return x // y, x % y
```

Можно реализовать сразу присвоение двум переменным, так раскрывается кортеж.

```
>>> d, m = div_mod(10, 3)      # присваивание кортежем d = x // y, m = x % y
>>> assert (d, m) == (3, 1)    # Ассерты для такой функции
```

4.7.1.11 Класс множества (set).

По определению не может быть повторений и порядок элементов не важен.

xs = {1, 2, 3, ...}.

Пустое множество не имеет собственного особого литерала и обозначается set().

Операторы

наличие элемента во множестве in,

перегруженные операторы для методов объединение, пересечение |, &, объединение минус пересечение ^.

```
>>> xs = {1, 2, 3}
>>> 1 in xs
True
>>> 92 in xs
False
```

Используемые операции:

добавить элемент `add()`,

объединение множеств `|` или `.union()`,

пересечение множеств `&` или `.intersection()`,

исключающее или `^` или `.symmetric_difference`,

исключение элемента из множества `.discard()`.

```
>>> xs = {1, 2, 3}
>>> xs.add(1)    # Уже есть
>>> xs.add(92)
>>> xs
{1, 2, 3, 92}
>>> {1, 2, 3}.union({3, 4, 5})    # Объединение
{1, 2, 3, 4, 5}
>>> {1, 2, 3} | {3, 4, 5}
{1, 2, 3, 4, 5}
>>> {1, 2, 3}.intersection({3, 4, 5})    # Пересечение
{3}
>>> {1, 2, 3} & {3, 4, 5}
{3}
```

```
>>> {1, 2, 3}.symmetric_difference({3, 4, 5})    # XOR
{1, 2, 4, 5}
>>> {1, 2, 3} ^ {3, 4, 5}
{1, 2, 4, 5}
>>> xs.discard(2)    # В действительности операция редкая
>>> xs
{1, 3}
```

Нельзя добавить список во множество так как список изменяемый объект (mutable).

```
>>> xs = set()
>>> xs.add([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

4.7.1.12 Класс словаря (dict).

```
dict = {"key0": value_a, "key1": value_b}
```

- "длина" словаря - число входящих элементов len(),

- взятие значения по ключу dict[key],

- присвоение по ключу dict[key] = 14.

```
>>> date = {"year": 2018, "month": "September"}
>>> len(date)    # Длина словаря
2
>>> date["year"]    # Взятие по индексу
2018
>>> date["day"] = 14    # Присвоение по индексу
>>> date["year"]
14
```

Оператор наличие ключа в словаре in.

```
>>> 'day' in date.keys()    # так лучше не делать
True
>>> 'day' in date # лучше делать так
True
```

Основные методы словаря:

Получение значения или значения по умолчанию, если ключа нет .get(key, value),

удалить запись из словаря и вернуть ее значение .pop(key),

вывести все ключи .keys(),

вывести все значения .values(),

вывести все и сразу .items().

```
>>> date.get("day", 10)    # Значение по умолчанию или значение которое есть в словаре
14
>>> date.pop("year")
2018
>>> date.keys()
dict_keys(['month', 'day'])
```

```
>>> date.values()
dict_values(['September', 14])
>>> date.items()
dict_items([('month', 'September'), ('day', 14)])
```

Порядок элементов остается неизменнным и зависит только от времени присвоения ключу его первого значения.

```
>>> dict_ = {}
>>> dict_["a"] = 1
>>> dict_["b"] = 2
>>> dict_["c"] = 3
>>> list(dict_.keys())
['a', 'b', 'c']
>>> dict_["a"] = 3
>>> dict_["b"] = 2
>>> dict_["c"] = 1
>>> list(dict_.keys())    # Порядок гарантируется
['a', 'b', 'c']
>>> list(dict_.values())    # Порядок гарантируется
[3, 2, 1]
```

4.7.2 Основные операторы.

4.7.2.1 Оператор if.

Стандарный условный оператор, позволяющий проверить условие и выполнить действие, если условие выполнится и в отдельном блоке выполнить операции, если условие не выполнится. Здесь добавляется дополнительно elif с условием если не выполнится первое условие, их может быть несколько и наконец выполняется else если все что было в if и в elif не выполнилось.

```
if CONDIT1:
    expression1()
elif CONDIT2:
    expression2()
else:
    expression3()
```

Не следует записывать условия в одну строку, лучше записывать так:

```
if (x[0] < 100 and x[1] > 100
    and (is_full_moon() or not is_thursday())
    and user.is_admin):
    pass
```

4.7.2.2 Оператор ternary if.

Позволяет вернуть значение после проверки условия.

Возвращаемое значение this() if CONDIT1 else that()

Получить минимальное значение.

```
value = x if x < y else y    # int value = (x < y)? x : y; ## На Си
```

4.7.2.3 Цикл while и for. Break. Continue.

Пока выполняется условие выполняй действия.

while CONDIT:

  expression()

Перебрать значения в списке или в генераторе.

for var in list: expression

```
for x in range(10):    # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
   print(x)
for ch in "Hello world":    # H, e, l, l,o, ' ', w, o, r, l, d
   print(ch)
```

range(START, STOP, STEP) - генератор

Преждевременный принудительный выход из цикла break.

```
target = some_target
for item in items:
   if item == target:
      print("Found!", item)
      break
else:
   print("Not found")
```

Особый вариант else for сработает только если не было принудительного выхода из цикла break, то есть если цикл весь выполнится, удобно для поиска элементов, например, в циклах без else for условия пришлось бы завести дополнительную переменную логического типа, которая бы проверяла, что элемент не найден, и нужно было бы добавить дополнительный условный оператор и дополнительную переменную.

Принудительное завершение итерации цикла continue. В данном примере если не найден элемент равный target, то происходит переход на следующую итерацию и операция res.append(item) не срабатывает.

```
target = some_target
res = []
for item in items:
   if item != target:
      continue
   res.append(item)
```

4.7.3 Функции Python.

Документирование функций.

Зарезервированное слово def предваряет определение функции. За ним должны следовать имя функции и заключенный в скобки список формальных параметров. Выражения, формирующие тело функции, начинаются со следующей строки и должны иметь отступ. Первым выражением в теле функции может быть строковый литерал - этот литерал является строкой документации функции, или док-строкой (`docstring`)

```
def some_function(arg1, arg2, ..., argN):
    """Documentation"""
    expressions
```

Пример документации на функцию

```
def my_function():
    """
    Documentation

    with empty line.
    """
    pass
```

Получить метаданные функции

```
>>> my_function.__name__
'my_function'
>>> my_function.__doc__
'\n Documentation\n\n with empty line\n'
>>> help(my_function)
Help on function my_function in module __main__:
my_function()
    documentation

    with empty line
```

4.7.3.1 Аргументы функции. Вызов функции.

```
def min(x, y):
    return x if x < y else y
min(1, 2)       # Аргументы посылаются кортежем
min(1, y=2)     # Можно явно соотносить значение с именем
min(x=1, y=2)
min(y=2, x=1)   # И даже в любом порядке
```

Можно подать в функцию неименованные аргументы.

```
def min(*args):
    res = float('inf')
    for x in args:
```

```
        res = x if x < res else res
    return res
min(92, 10, 62) # Несколько неименованных аргументов для поиска минимума.
min()          # Возвращаем значение даже при отсутствии элементов, и это не всегда удачная ситуа-
ция
xs = [1, 2, 3]  #
min(*xs)  # Разыменовывание списка, на вход функции поступает три аргумента.
```

Функтор - функция применяется к элементам, находящимся в некотором контексте, в функторе (список - функтор), объект к которому применима fmap.

Аппликативный функтор - извлеченная упакованная функция (сама функция возвращает неупакованное значение), применяется к извлеченному упакованному значению и получает упакованные значения.

Монада - вы применяете функцию, возвращающую упакованное значение, к упакованному значению (PyMonad).

Монада - тип, который позволяет строить цепочки вычислений. Между этими вычислениями передаются только монады, что и делает эти цепочки в некотором смысле универсальными.

Пример функтора.

```
X = [1, 2, 3]
>>> mp = map(lambda x:x*2, x)
>>> list(mp)
[2, 4, 6]
def positive_and_negative(x):
    return List(x, -x)
List(9) >> positive_and_negative    # Результатом станет монада List(9, -9)
def add_and_sub(x, y):
    return List(y + x, y - x)
List(2) >> add_and_sub(3)   # Вернет список List(5, -1)

List(2) >> positive_and_negative >> add_and_sub(3)  # Результатом станет List(5, -1, 1, -5)
```

Еще одна попытка описания функции min. Один элемент обязателен, остальные опционально.

```
def min(first, *rest):
    res = first
    for x in rest:
        res = x if x < res else res
    return res
>>> min("hello", ",", " ", "world")
' '
>>> min()   # Ошибка в случае отсутствия элементов, но зато можно отследить ошибку.
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
TypeError: min() missing 1 required positional argument: 'first'
```

Но иногда нужен минимум по пустому значению. Указываем дефолтное значение в случае пустого списка аргументов. В этом случае если не указать значение default, то будет None по умолчанию, либо можно его указать при вызове.

```
def min(*args, default=None):
    if not args:
        return default
    res, *rest = args
    for x in rest:
        res = x if x < res else res
    return res
>>> min()   # Теперь работает
None
>>> min(xs, default=0)  # Обязательно присваивать по имени
```

Как заставить использовать только именованные ключевые аргументы где нам нужно? Поставить *. Код становится более понятным. Так как стоит *, то интерпретатор ожидает после * именованных переменных.

```
def min(a, *, b=6):
    if(a < b):
        return a
    else:
        return b
>>> min(4)
4
>>> min(2, b=3)
2
>>> min(2, 3)
Traceback (most recent call last):
    File "main.py", line 8m in <module>
        min(2, 3)
TupeError: min() takes 1 positional argument but 2 where given
```

*4.7.3.2 Использование *args и **kwargs.*

* преобразует в аргументы список без имен.

** преобразует словарь в именованные аргументы

```
def call_me(*args, **kwargs):
    return (args, kwargs)
>>> call_me(**{"a": 92})    # Передали в качестве аргументов пустой список и словарь
((), {'a': 92})
def add(x, y):
    return x+y
```

```
>>> mass = [1, 2]
>>> add(*mass)     # Аргументы в список без имен
3
>>> dict = {'x':1, 'y':2}
>>> add(**dict)    # Словарь в именованные аргументы
>>> print(*[1], *[2], 3)
1 2 3
>>> dict(**{'x': 1}, y=2, **{'z':3})
{'x':1, 'y':2, 'z': 3}
>>> *range(4), 4
(0, 1, 2, 3, 4)
>>> [*range(4), 4]
[0, 1, 2, 3, 4]
>>> {*range(4), 4}
{0, 1, 2, 3, 4}
>>> {'x': 1, **{y: 2}}
{'x': 1, 'y': 2}
>>> {'x': 1, **{'x': 2}}    # При совпадении ключей, последний идет в словарь
{'x': 2}
>>> {**{'x': 2}, 'x': 1}    #
{'x': 1}
def min1(**kwargs):
    print(kwargs)
    for (key, it) in kwargs.items():
        print(key, it)
    return
>>> print(min1(**{'x':1, **{'x': 2}})
{'x':2}
x 2
None
```

Рассмотрим присвоения такого рода:

```
>>> x, *xs = [1, 2, 3]  ## * - это шаблон - пример получения головы и хвоста списка
>>> x, xs
(1, [2, 3])
>>> x, *xs = "123"  ## Может быть любой итерабельный тип, но при применении звездочки получа-
ется список.
>>> x, xs
('1', ['2', '3'])
>>> first, *rect, last = [1, 2, 3]
>>> first, rect, last
(1, [2], 3)
>>> rectangle = ((0, 0), (2, 3))
>>> (x1, y1), (x2, y2) = rectangle
>>> y2
3
```

Зачем нужно присвоение такого рода? Извлекаем из кортежа координаты в отдельные переменные.

Рассмотрим следующий пример где, например, из файла или некоего строкового содержимого считываются строки.

```
"""
name,price,quantity
spam,8,92
"""
d = {}
for line in text.splitlines():
    cells = line.split(',')
    d[cells[0]] = cells[1]
```

Здесь могут склеиться две строки, и мы получим не тот результат, при этом обнаружить это будет достаточно сложно.

Исправим данную ситуацию при условии, что строк должно быть ровно три:

```
d = {}
for line in text.splitlines():
    name, price, _ = line.split(',')
    d[name] = price
```

Здесь обязательно должны быть три возвращенных переменных в кортеже, последняя по умолчанию не сохраняется, но указывается что она должна быть, если это не так, то возникнет ошибка.

Исправление при условии, что количество строк после первых двух не важно:

```
d = {}
for line in text.splitlines():
    name, price, *_ = line.split(',')   # Все последующие параметры экранируются *
    d[name] = price
```

4.7.3.3 Область видимости переменных.

```
def is_even(n):
    return n == 0 if n <= 1 else is_odd(n - 1)
def is_odd(n):
    return n == 1 if n <= 1 else is_even(n - 1)
assert is_even(92)

def is_even(n):
    return n == 0 if n <= 1 else is_odd(n - 1)
assert is_even(92)  # Этот вызов не может сработать, is_odd не существует
def is_odd(n):
    return n == 1 if n <= 1 else is_even(n - 1)
```

Здесь вызов не сработает, так как is_odd еще не добавлена и не именована.

Словарь глобальных и локальных объектов.

В Python есть области видимости Global, Local и Enclosing.

Local - область видимости локальных переменных функции.

Enclosing - область видимости для внутренней функции внутри функции.

Локальные переменные исчезают после вызова функции, если на них не ссылались из вложенной функции. Функция обходится сначала анализатором кода полностью и потом преобразуется в byte code.

```
x = 1
def foo():
    y = 2
    print(globals(), type(globals()))
    print(locals(), type(locals()))
>>> foo()
{'x': 1, 'foo': <function ...>, ...} <class 'dict'>
{'y': 2} <class 'dict'>
```

Python сначала ищет все присвоения локальной переменной и потому о всех своих переменных и их использовании знает заранее.

```
x = 1
def foo():
    print(x)
>>> foo()
1
x = 1
def foo():
    print(x)
    x = 2
>>> foo()
UnboundLocalError:
local variable 'x' referenced before assignment
```

Мы видим различие при работе второй функции если вызывается переменная до ее присвоения внутри функции, отсюда видно, что анализатор кода оценивает полностью всю функцию.

Замыкание (closure) - это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее параметрами. Далее такой же пример как в предыдущем случае для замыкания.

```
def fun():
    x = 2
    def bar():
        print(x)
        x = 2
    bar()
    return
>>> foo()
UnboundLocalError:
local variable 'x' referenced before assignment

def fun():
    x = 2
    def bar():
        print(x)
    bar()
    return
>>> fun()
2
```

Рассмотрим пример функции, которая использует функцию умножения, для создания функции умножения на константу.

```
def mul5(a)
   return mul(5, a)
>>> mul5(2)
10
>>> mul5(7)
35
```

Рассмотрим такой же пример с использованием замыканий.

```
def mul(a):
   print(a)
   def helper(b):
      print(b)
      return a*b
   return helper
```

Очевидно, вызывается функция mul(5), которая возвращает функцию helper, которая вызывается с аргументом 2.

```
print(mul(5)(2))
```

Можно создать функцию умножения на константу 5.

```
>>> new_mul5 = mul(5)
>>> new_mul5
<function mul.<locals>.helper at 0x000001A7548C1158>
>>> new_mul5(2)
10
>>> new_mul5(7)
35
```

Для изменений глобальных переменных и вышестоящей функции применяются global, nonlocal.

```
x = 1
def foo():
   global x # теперь переменная глобальная
   x = 2
   y = 1
   def bar(): # теперь переменная относится к вышестоящей функции
      nonlocal y
      y = 2
```

Приведем следующий пример.

```
def foo():
   res = []
   for i in range(3):
      def bar():
         return i
      res.append(bar)
   return res
for f in foo():
   print(f(), end=" ")
==> 2 2 2
```

Приведенный пример создает список функций с обращением к enclosing для последнего значения и с дефолтным значением.

Примерно то же самое с многомерными массивами и `[[]*3]*3`. В первом случае для вложенной функции глобальна, соответственно для функции берется ее последнее принятое значение. Потому получается 2 2 2.

```
def foo():
    res = []
    for i in range(3):
        def bar(i=i):
            return i
        res.append(bar)
    return res
for f in foo():
    print(f(), end=" ")
==> 0 1 2
```

Следующий пример показывает аналогичную ситуацию для x, при добавлении функций для каждой вложенной функции берется последнее присвоенное значение x.

```
def f():
    x = 2
    val = []
    def f1():
        return x
    val.append(f1)
    x = 9
    def f1():
        return x
    val.append(f1)
    return val
z = f()

for f in z:
    print(f)
    print(f())

==>
<function f.<locals>.f1 at 0x7f2090499790>
9
<function f.<locals>.f1 at 0x7f2090499790>
9
```

4.7.3.4 Лямбда выражение.

Схема объявления и использования лямбда выражений.

lambda arguments: expression # Лямбда функция

def _(arguments): # Ее аналог в виде именованной функции _()

  return expression

Пример описания лямбда выражения

```
lambda a, *args, b=1, **kwargs: 92
```

Функция высшего порядка map.

```
>>> range(3)
range(0, 3)
>>> list(range(3))
[0, 1, 2]
>>> map(lambda x: x + 1, [0, 1, 2])
<map object at 0x7fc54d060da0>
>>> list(map(lambda x: x + 1, [0, 1, 2]))   # x соотносится и применяется лямбда функция
[3, 5, 7]
>>> list(map(lambda x: x + y, [0, 1, 2], [3, 4, 5, 6]))
[3, 5, 7]
```

Фильтрация по условию

```
>>> list(filter(lambda x: x % 2 == 0, range(10)))
[0, 2, 4, 6, 8]
>>> list(filter(None, [0, 1, True, False, [], {None}])) # Фильтрация без функции, убивает все Falsy эле-
менты
[1, True, {None}]
```

Спаривание итерируемых объектов.

```
>>> list(zip("hello", range(10)))
[('h', 0), ('e', 1), ('l', 2), ('l', 3), ('o', 4)]
```

Пример перебора упакованных вместе значений

```
assert len(xs) == len(ys)
for x, y in zip(xs, ys):
    ...
```

Далее два примера с идентичным результатом, но первый понятнее и короче

```
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

То же самое, но с использованием map.

```
>>> list(map(
        lambda x: x**2,
        filter(lambda x: x % 2 == 0, range(10))
        ))
[0, 4, 16, 36, 64]
```

Аналогичный пример для нескольких циклов и условий.

```
res = [
    (x, y)
    for x in range(5)
    if x % 2 == 0
    for y in range(x)
    if y % 2 == 1
]
print(res)
==> [(2, 1), (4, 1), (4, 3)]
```

Эти же цикл и условия записанные стандартным способом.

```
res = []
for x in range(5):
    if x % 2 == 0:
        for y in range(x):
            if y % 2 == 1:
                res.append((x, y))
print(res)
```

4.7.4 Декораторы python.

Декораторы используются с целью замены какой-либо функции, чаще всего сторонней, добавив дополнительные особенности при ее выполнении.

Декоратор является функцией более высокого порядка. Приведем пример реализации без использования так называемого «синтаксического сахара».

```
form random import random
def decorator (func):
   def wrapper_for_func ():
      print ("Запускаем что-либо до вызова функции")
      x = func ()
      print (x)
      print ("Запускаем что0либо после вызова функции")
      return x
   return wrapper_for_func
```

Здесь мы видим как внутренняя функция получает на вход функцию аргумент и использует ее для вычислений, при этом декоратор возвращает внутреннюю функцию.

```
print (random ())
random = decorator (random)
print (random ())
```

Мы здесь переопределили стандартную функцию random, добавив дополнительную функциональность, связанную с печатью. Например, можно использовать для логгирования. Допустим у вас есть какая-то функция, ведущая расчеты, и она дает некоторую систематическую погрешность, исправлять такую функцию долго, и заменять все ее вхождения в проекте тоже, можно использовать декоратор, который просто изменит значение функции на данную погрешность или добавит дополнительную функциональность, при этом не надо будет менять весь проект. Декораторы очень часто используются в фрейморвках для добавления дополнительных возможностей, например, кэширования, авторизации, документирования api.

Результат работы

```
0.5948731855992081
Запускаем что-либо до вызова функции
0.5985262870159087
Запускаем что-либо после вызова функции
0.5985262870159087
```

Рассмотрим реализацию декоратора с использованием синтаксиса языка («синтаксического сахара»).

```
from random import random
def decorator_name (func):
   def wrapper_for_func ():
      print ("Запускаем что-либо до вызова функции")
      x = func ()
      print (x)
      print ("Запускаем что-либо после вызова функции")
      return x
   return wrapper_for_func
@decorator_name # декорируем функцию, не присвоением
def add ():
   return 1
print (add ())
```

Результат работы:

```
Запускаем что-либо до вызова функции
1
Запускаем что-либо после вызова функции
1
```

Рассмотрим пример декоратора для определения времени работы функции.

```
import time
def decorator_time (func):
   def wrapper_for_func ():
      print ("")
      t = time.time ()
      x = func ()
      dt = time.time () - t
      print (f"Время исполнения функции {func.__name__} равно {dt}")
      return x
   return wrapper_for_func

@decorator_time
def add ():
   i = 0.0
   for i in range (10000):
      i=i+i
   return i
print (add ())
```

Особенность в том, что теперь можно декорировать любую функцию данным декоратором и получать время работы какой-либо функции. Отсюда вытекает еще одна важная роль декораторов, это использование уже написанных декораторов с какими-то целями приложения, что существенно снижает объем кода, позволяет повторно использовать уже написанные кем-то и вами функции стандартной декорации.

Результат работы:

```
время исполнения функции add равно
0.0006504058837890625
19998
```

Пример применения декоратора к двум различным функциям.

```
@decorator_time
```

```
def add ():
    i = 0.0
    for i in range (10000):
        i=i+i
    return i
print (add ())

@decorator_time
def rand ():
    return random.random ()
print (rand ())
Время исполнения функции add равно 0.0008225440979003906
19998
Время исполнения функции rand равно 2.86102294921875e-06
0.8971549149038762
```

Декоратор для определения количества запусков функции. В данном случае нужно как-то возвращать внутренние переменные, которые у нас после исполнения функции удаляются. Поэтому это реализуется следующим образом:

```
def decorator_count (func):
    def wrap_func ():
        wrap_func.__ecount__ += 1
        return func ()
    wrap_func.__ecount__ = 0
    return wrap_func
@decorator_count
def my_func ():
    boomboom = 0
    return boomboom
for i in range (100):
    my_func ()
print (my_func.__ecount__)
100
```

Пример использования двух декораторов. Декорирование декорируемой функции.

```
import time
def decorator_time (func):
    def wrapper_for_func ():
        t = time.time ()
        print (f"time start")
        x = func ()
        dt = time.time() - t
        print (f"Время исполнения функции {func.__name__} равно {dt}")
        print (f"time end")
        return x
    return wrapper_for_func

def decorator_count (func):
    def wrap_func ():
        print ("count start")
        wrap_func.__ecount__ += 1
        x = func ()
        print ("count end")
        return x
```

```
    wrap_func.__ecount__ = 0
    return wrap_func

# Декоратор над декоратором над функцией.
# Если поменять местами count и time,
# __ecount__ уже не будет доступен.

@decorator_count
@decorator_time
def my_func ():
    boomboom = 0
    return boomboom
for i in range (2):
    print (my_func ())
    print (my_func.__ecount__)
```

Результат выполнения функции.

```
count start
time start
Время исполнения функции my_func равно 5.245208740234375e-06
time end
count end
0
1
count start
time start
Время исполнения функции my_func равно 6.87572021484375y-06
time end
count end
0
2
```

Декораторы с аргументами.

```
def decorator_with_args (func):
    def wrap_func (x=0, y=0):
        print (f"Get for {func.__name__} x = {x} y = {y}")
        return func (x, y)
    return wrap_func
@decorator_with_args
def add (x=0, y=0)
    return x+y
print (add (x=2, y=2))
```

Результат.

```
Get for add x = 2 y = 2
4
```

Использование неименованных аргументов в декораторе.

```
def decorator_with_args (func):
    def wrap_func (arg1, arg2):
        print (f"Get for {func.__name__} x = {arg1} y = {arg2}")
        return func (arg1, arg2)
    return wrap_func

@decorator_with_args
def add (x, y):
```

```
    return x+y
print (add (2, 2))
```

Неименованные аргументы

```
def decorator_with_args (func):
    def wrap_func (*arg):
        print (f"Get for {func.__name__} {arg}")
    return wrap_func

@decorator_with_args
def add (x, y):
    return x+y

print (add (2, 2))
```

Разыменовывание `*, **` в f {} строке использовать нельзя.

```
def decorator_with_args (func):
    def wrap_func (*arg, **kwargs):
        print (f"Get for {func.__name__} {arg} {kwargs}")
        return func (*arg, **kwargs)
    return wrap_func

@decorator_with_args
def add (z, x=0, y=0):
    return x+y+z

print (add (1, x=2, y=2))

Get for add (1,) {'x':2, 'y':2}
5
```

4.7.4.1 Модуль functools.

Содержит полезные декораторы, такие как: lru_cache - кэширование последних max вызовов функции, partials - создание функции с частично заданными аргументами, singledispatch - перегрузка функции в соответствии с типом первого аргумента, wraps - декоратор для внутренней wrap функции, чтобы, например, по имени возвращалось описание самой функции, а не врапера.

Декоратор Lru_cache. Кэширует результат работы функции. Можно использовать, например, при обращении к данным на сайте, после чего они будут закэшированы. При расчетах какой-либо функции, если ее какие-либо входные данные могут быть теми же самыми. Ниже приведен пример процедуры расчета интеграла методом Ромберга, относительно вычислительно сложная процедура для определенных функций.

```
@Lru_cache (30)
  def example (a, b, func, *args):
    e = 1e-9
    def romberg (a1, b1, n1. n2):
      iv = 0.0
      dx1 = (b1-a1)/n1
      f1 = func (a1, *args)
      for i in range (n1):
        x = a1 + (i+1) * dx1
        f2 = func (x, *args)
        sq = (f1+f2) * dx1 * 0.5
        ff1 = f1
        f1 = f2
        x1 = a1+i*dx1
        x2 = x
        dx2 = (x2-x1)/n2
        sqv = 0.0
        for j in range (n2):
          x = x1+(j+1)*dx2
          ff2 = func (x, *args)
          sqv = sqv+(ff1+ff2)*dx2*0.5
          ff1 = ff2
        if (abs (sqv-sq)>e):
          iv = iv+romberg (x1, x2, n1, n2)
        else:
          iv = iv + sqv
      return iv
    return romberg (a, b, 2000, 50)

t = time.time ()
print (example (-100, 11, gauss, 0.0, 0.00005))
t = time.time () - t
print (f"time of first function {t}")
t = time.time ()
print (example (-100, 11, gauss, 0.0, 0.00006))
t = time.time () - t
print (f"time of second function {t}")
t = time.time ()
print (example (-100, 11, gauss, 0, 0,000005))
t = time.time() - t
print (f"time of third function {t}")
```

Результаты

```
1.0000000000000795
time of first function
2.861814022064209
1.0000000000001261
time of second function
3.144050359725952
1.0000000000000795
time of third function
2.9802322387695312e-05
```

Как видим расчет для тех же значений аргументов происходит моментально, так как результат берется из КЭШа, аналогичным образом можно

реализовать, например, кэширование передаваемого контента, не прибегая к посредничеству Redis.

Декоратор Partial позволяет получать функции с меньшим числом аргументов чем исходная декорируемая функция.

```
from functools import partial

def add (x, y):
   return x+y

# Преобразуем в функцию без аргументов
# суммирующую две константы
p_add0 = partial (add, 2, 5)
print (p_add0 ())

# Преобразуем в функцию без аргументов
# суммирующую переменные в константу 2
p_add1 = partial (add, 2)
print (p_add1 (5))

# Оставляем функцию с тем же
# количеством аргументов
p_add2 = partial (add)
print (p_add2 (2, 5))
```

Атрибуты возвращаемого partial-объекта

partial (func, args, keywords)`

func - функция, к которой будет перенаправлен вызов с применением аргументов.

args - позиционные аргументы, которые будут переданы в вызываемую функцию при вызове объекта.

keywords - именованные аргументы, которые будут переданы при вызове объекта.

Можно реализовывать сценарии с помощью декоратора Partial.

```
from functools import partial
def add (x, y):
   return x + y

def multiply (x, y):
   return x * y

def run (func):
   print (func ())

a1 = partial (add, 1, 2)
m1 = partial (multiply, 5, 8)
```

```
run (a1)
run (m1)
```

Декоратор Singledispatch.

Сработает функция в соответствии с первым типом в register.

```
from functools import singledispatch
@singledispatch
def add (a, b):
   raise NotImplementedError ('Unsupported type error')

@add.register (int)
def _(a, b):
   print ("First argument is of type ", type (a))
   print (a + b)

@add.register (str)
def _(a, b):
   print ("First argument is of type ", type (a))
   print (a + b)

@add.register (list)
def _(a, b):
   print ("First argument if of type ", type (a))
   print (a + b)

add (1, 2)
add ("Python", "Programming")
add ([1, 2, 3], [5, 6, 7])
add (1.0, 1) # Вернет ошибку Unsupported type
```

Результат

```
First argument is of type <class 'int'>
3
First argument is of type <class 'str'>
PythonProgramming
First argument is of type <class 'list'>
[1, 2, 3, 5, 6, 7]
raise NotImplementedError ('Unsupported type error')
NotImplementedError: Unsupported type error
```

Декоратор Wraps. Используется для того, чтобы можно было получить доступ к данным или к описанию декорируемой функции. Например, если получить свойство doc, то без декоратора wraps, будет возвращена документация на оберточную функцию, а с декоратором на декорируемую функцию.

```
from functools import wraps
def another_function (func):
   """
   Функция которая принимает другую функцию
   """
   @wraps (func)
   def wrapper ():
      """
```

```
        Оберточная функция
        """
        val = "The result of %s is %s" % (func (), eval (func ()))
        return val
    return wrapper

@another_function
def a_function ():
    """
    Обычная функция
    """
    return "1+1"

print (a_function.__name__) # a_function (без wraps wrapper)
print (a_function.__doc__)  # Обычная функция (без wraps Оберточная функция)
```

4.7.5 Генераторы Python.

Генератор создает объект, который используется для итерирования, при этом он генерирует по одному элементу и не хранит их в памяти после итерирования. Фактически сгенерировав объект, происходит переход к следующему, предыдущие в памяти не хранятся.

Генератор range ().

```
generator = (i for i in range (5))
print (generator)
for item in generator:
    print (item)
```

Yield.

Функция выполняется до первого yield, генерирует объект и при следующем исполнении начинает с того места, где завершился последний выполненный yield. В этом плане это return, только следующих вызов будет с того места, где был последний вызов yield.

```
def fgenerator(n) :
    print ("start")
    for i in range (n):
        print ("in cycle")
        yield i
    yield 100
    print ("between")
    yield 200
    print ("end")

p = fgenerator(5)

for i in p: # перебираем весь генератор
    print (i)
```

```
print (list (p))
```

Результат

```
start
in cycle
0
in cycle
1
in cycle
2
in cycle
3
in cycle
4
100
between
200
end
[]
```

После перебора генератор больше ничего не возвращает при обращении к нему, список становится пустым.

Если поменять порядок вызова генератора, увидим, что после перебора генератора, элементов в нем для перебора не останется, а список заполнится значениями.

```
def fgenerator (n):
   print ("start")
   for i in range (n):
      print ("in cycle")
      yield i
   yield 100
   print ("between")
   yield 200
   print ("end")

p = fgenerator (5)

print (list (p)) # напечатается список
for i in p: # ничего не напечатается
   print (i)
```

Результат

```
start
in cycle
in cycle
in cycle
in cycle
in cycle
between
end
[0,1,2,3,4,100,200]
```

return прерывает итерацию генератора.

```
def fgenerator (n):
    print ("start"):
    for i in range (n):
        if (i>5):
            return
        print ("in cycle")
        yield i
    yield 100
    print ("between")
    yield 200
    print ("end")

p = fgenerator (7) # Будет каждый раз срываться при итерации > 5

print (list (p))
```

Результат

```
start
in cycle
in cycle
in cycle
in cycle
in cycle
in cycle
[0,1,2,3,4,5]
```

Генерация случайного гладкого сигнала.

```
from math import *
from random import *
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import UnivariateSpline


def func(n):
    nw = int(n / (random() * 10 + 2))
    w = (np.random.rand(nw) - 0.5) * 2
    x = np.linespace(-1, 1, nw)
    spl = UnivariateSpline(-1, 1, n)
    xs - np.linespace(-1, 1, n)
    ws = spl(xs)
    return ws


def funcgener(size, n):
    for i in range(size):
        yield func(n)


n = 100
res = [it for it in funcgener(10, n)]
print(len(res))

x = np.linespace(-1, 1, n)
y = func(n)

err = np.random.randn(n)
yerr = y + err * 0.1 * y
```

```
plt.plot(x, y)
plt.plot(x, yerr)
plt.show()
```

4.7.6 Классы Python.

Объявление класса начинается с ключевого слова class.

```
class Figure ():
   def square ():
      pass
   def perimetr ():
      pass
```

В скобках указывается класс, на котором основывается новый класс (родительский класс)

```
class Rectangle(Figure):

   # __init__ - особый метод - конструктор класса

   # Self - аргумент для ссылки объекта на самого себя

   def __init__(self, left, top, width, height):

      self.top = top

      self.left = left

      self.width = width

      self.height = height


   def square(self):

      return self.width * self.height


   def perimeter(self):

      return (self.width + self.height) * 2




class Rect(Rectangle):

   bottom = 0

   right = 0


   def __init__(self, left, top, rigth, bottom):

      # self.bottom = bottom
```

```
    # self.right = right

    # super - ключевое слово, которое используется для обращения к родительскому классу.
    super().__init__(left, top, right - left, bottom - top)
    pass

rect = Rectangle(0, 0, 10, 10)
print(rect.square())  # 100

rect1 = Rect(10, 10, 20, 20)
print(rect1.square())  # 100
rect2 = Rect(15, 15, 20, 20)
print(rect2.square())  # 25

rect1.bottom = 1
rect1.right = 2

print(rect2.bottom)  # 0
print(rect2.right)  # 0
Rect.bottom = 3
Rect.rigth = 3

print(rect1.bottom)  # 1
print(rect1.right)  # 2

print(rect2.bottom)  # 3
print(rect2.rigth)  # 3
```

Вызов методов базового (родительского) класса.

Надо вызвать метод базового класса из метода, который переопределен в производном классе. Из конструктора дочернего класса нужно явно вызы-

вать конструктор родительского класса. Обращение к базовому классу происходит с помощью `super ()`.

Видно, что без явного вызова конструктора класса `A` не вызывается `A.__init__` и не создается поле x класса `A`.

```
class A (object):
   def __init (self, x=5):
      print ('A.__init__')
      self.x = x

class B (A):
   def __init__ (self, y=2):
      print ('B.__init__')
      self.y = y

k = B (7)              # B.__init__

print ('k.y =', k.y)       # k.y = 7
print ('k.y =', k.x)       # AttributeError: 'B' object has no attribute 'x'
```

Реализация явного вызова.

```
class A (object):
   def __int__ (self, x=5):
      print ('A.__init__')
      self.x = x

class B (A):
   def __init__ (self, y=2):
      print ('B.__init__')
      super ().__init__ (y/2)
      self.y = y

      k = B (7)          # B.__init__, A.__init__

   print ('k.y =', k.y)    # k.y = 7
   print ('k.x =', k.x)    # k.y = 3.5
```

Конструктор базового класса стоит вызывать раньше, чем инициализировать поля класса-наследника, потому что поля наследника могут зависеть от (быть сделаны из) полей экземпляра базового класса.

Метод класса, помимо `super ()` можно вызывать, используя синтаксис вызова через имя класса.

```
class Base (object):
   def __init__ (self):
      print ('Base.__init__')

class A (Base):
   def __init__ (self):
      Base.__init__ (self)
      print ('A.__init__')

k = A ()       # Base.__init__
            # A.__init__
```

В следующем примере видно, что конструктор `Base.__init__` вызывается дважды. Иногда это недопустимо (считаем количество созданных экземпляров увеличивая в конструкторе счетчик на 1; выдаем очередное auto id какому-то нашему объекту, например, номер пропуска или паспорта, или номера заказа).

```
class A (Base):
   def __init__ (self):
      Base.__init__ (self)
      print ('A.__init__')

class B (Base):
   def __init__ (self):
      Base.__init__ (self)
      print ('B.__init__')

class C (A, B):
   def __init__ (self):
      A.__init__ (self)
      B.__init__ (self)
      print ('C.__init__')

X = C ()       # Base.__init__
            # A.__init__
            # Base.__init__ - !! Второй вызов
            # B.__init__
            # C.__init__
```

Для реализации наследования питон ищет вызванный атрибут начиная с первого класса до последнего. Этот список создается слиянием (merge sort) списков базовых классов:

Дети проверяются раньше родителей.

Если родителей несколько, то проверяем в том порядке, в котором они перечислены.

Если подходят несколько классов, то выбираем первого родителя.

При вызове `super()` продолжается поиск, начиная со следующего имени в MRO. Пока каждый переопределенный метод вызывает `super` и вызывает его только один раз, будет перебран весь список MRO и каждый метод будет вызван только один раз.

```
class Base (object):
   def __init__ (self):
      print ('Base.__init__')
```

```
class A (Base):
   def __init__ (self):
      super ().__init__ ()
      print ('A.__init__')

class B (Base):
   def __init__ (self):
      super ().__init__ ()
      printf ('B.__init__')

class C (A, B):
   def __init__ (self):
      super ().__init__ ()
      print ('C.__init__')

X = C ()          # Base.__init__
                  # B.__init__ - вызваы конструкторы обоих классов
                  # A.__init__ - порядок вызова
                  # C.__init__
```

Вызов метода дедушки.

```
class A (object):
   def spam (self):
      print ('A.spam')

class B (A):
   pass

class C (B):
   def spam (self):
      super ().spam ()
      print ('C.spam')

y = C ()
y.spam ()
print (C.__mro__)

A.spam
C.spam

(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>,
 <class 'object'>)
```

4.7.6.1 Декораторы методов.

```
def method_friendly_decorator (method_to_decorate):
   def wrapper (self, lie):
      lie -= 3
      return method_to_decorate (self, lie)
   return wrapper

class Lucy:
   def __init__ (self):
      self.age = 32

   @method_friendly_decorator
   def sayYourAge (self, lie):
      print ("Мне {} лет, а ты бы сколько дал?".format (self.age + lie))

l = Lucy ()
l.sayYourAge (-3)      # Мне 26, а ты бы сколько дал?
```

4.7.7 Пример веб-приложения на Flask.

Создадим отдельный проект, в котором рассмотрим кратко использование одной из библиотек позволяющей вести автоматическую документацию api с использованием swagger. Без документации ваше api бесполезно для других людей, впрочем, и для вас через какое-то время, когда вы забудете, что делали ранее, потому необходимо вести документацию, а еще проще ее вести если это будет делаться автоматически.

Папки, которые созданы в данном проекте не используются, и даны всего лишь для наглядности и понимания структуры проекта на flask. В данном случае используется некоторая отдельная часть сайта, сделанная с помощью blueprint (sitepart), которая содержит в себе свои шаблоны и свои статичные файлы, это и есть основная концепция использования blueprint. Когда проект разрастается и в одной папке или файле сложно хранить все элементы и части веб приложения, то можно использовать blueprint, который позволяет создавать отдельную часть сайта с использованием отдельного каталога, более того можно вести независимую разработку и потом подключить кусок такого blueprint к основному сервису просто со своим url.

Кроме того, в примере будет приведена возможность описания сайта с использованием swagger, благодаря библиотеке flasgger. Предварительно нужно создать environment и установить нужные библиотеки с помощью pip install.

```
sitepart
sitepart/static
sitepart/templates
sitepart/sitepart.py
static
templates
main.py
```

Содержимое main.py.

```
# Подключение библиотек для работы с Flask и Blueprint
```

```
from flask import Flask, jsonify, Blueprint
# Подключение библиотеки для создания автоматической документации api
from flasgger import Swagger
# Подключение части нашего веб сервиса с использованием Blueprint
from sitepart.sitepart import sitepart

# Приложение Flask
app = Flask(__name__)
# Инициализация для нашего api сервиса документации Swagger
swagger = Swagger(app)
# Создаем основной Blueprint сайта
main = Blueprint("/", __name__, template_folder='templates',static_folder='static')
# объявляем декоратор для метода http get
# Информация которая будет выдаваться по url /info/something
# Параметр в <> при вводе url будет передан в переменную about функции info
@main.route('/info/<about>/')
def info(about):
    """Example endpoint returning about info
    This is using docstrings for specifications.
    ---
    parameters:
      - name: about
        in: path
        type: string
        enum: ['all','version', 'author', 'year']
        required: true
        default: all
    definitions:
      About:
        type: string
    responses:
      200:
        description: A string
        schema:
          $ref: '#/definitions/About'
        examples:
          version: '1.0'
    """
    all_info = {
        'all': 'main_author 1.0 2020',
        'version': '1.0',
        'author': 'main_author',
        'year': '2020'
    }

    result = {about:all_info[about]}
    return jsonify(result)


# Регистрируем основной Blueprint и Blueprint другой части сайта
app.register_blueprint(main,url_prefix='/')
# url_prefix указывает url в контексте которого будет доступна часть данного Blueprint
app.register_blueprint(sitepart,url_prefix='/sitepart')

# Запуск приложения flask в режиме debug
app.run(debug=True)
```

Debug = True означает, что отладчик Flask работает. Эта функция полезна при разработке, так как при возникновении проблем она выдает дета-

лизированные сообщения об ошибке, что упрощает работу по их устранению.

Большинство функций здесь имеют комментарии, но, наверное, следует уделить внимание следующей части.

```
"""Example endpoint returning about info
   This is using docstrings for specifications.
   ---
   parameters:
     - name: about
       in: path
       type: string
       enum: ['all','version', 'author', 'year']
       required: true
       default: all
   definitions:
     About:
       type: string
   responses:
     200:
       description: A string
       schema:
         $ref: '#/definitions/About'
       examples:
         version: '1.0'
   """
```

Как видите, указанная часть, использующаяся для описания и документирования функций в python здесь описывает наше api. Здесь указан входной параметр (parameters), фактически, часть имени в пути url, он имеет тип string и может быть значением all, version, author или year, (например 127.0.0.1:5000/info/all). Более того, по адресу http://127.0.0.1:5000/apidocs можно получить документацию на api. Мы определяем тип About string в definitions и затем используем это определение в ответах responses. Тип ответа 200 OK, имеет описание и схему, говорящую о том, что в ответе json возвращается строка и даже приводится пример ответа на запрос version.

То же самое можно сделать, сославшись в документации к функции на файл, содержащий - - - и далее содержимое приводимое выше или в декораторе. На официальном сайте библиотеки приводится следующие примеры:

1.

```
from flasgger import swag_from

@app.route('/colors/<palette>/')
```

```
@swag_from('colors.yml')
def colors(palette):
    ...
```

2.

```
@app.route('/colors/<palette>/')
def colors(palette):
    """
    file: colors.yml
    """
    ...
```

Кроме того, приводятся и другие варианты использования определений swagger, в том числе через словарь python и др.

В файле sitepart.py в каталоге part содержится следующий код.

```
from flask import Blueprint,jsonify
# Создаем Blueprint для отдельной части веб api
sitepart = Blueprint("sitepart", __name__, template_folder='templates',static_folder='static')
# Возвращает цветовую палитру по имени палитры (rgb, cmyk ...)
@sitepart.route('/colors/<palette>/')
def colors(palette):
    """Example endpoint returning a list of colors by palette
    This is using docstrings for specifications.
    ---
    parameters:
      - name: palette
        in: path
        type: string
        enum: ['all', 'rgb', 'cmyk']
        required: true
        default: all
    definitions:
      Palette:
        type: object
        properties:
          palette_name:
            type: array
            items:
              $ref: '#/definitions/Color'
      Color:
        type: string
    responses:
      200:
        description: A list of colors (may be filtered by palette)
        schema:
          $ref: '#/definitions/Palette'
        examples:
          rgb: ['red', 'green', 'blue']
    """
    # содержимое цветов палитр
    all_colors = {
        'cmyk': ['cian', 'magenta', 'yellow', 'black'],
        'rgb': ['red', 'green', 'blue']
    }
    # что возвратить если url all
    if palette == 'all':
        result = all_colors
    else:
```

```
# возврат в зависимости от имени палитры
    result = {palette: all_colors.get(palette)}
# преобразуем словарь в json строку и возвращаем ее
return jsonify(result)
```

Здесь используется код с официального репозитория flasgger, как видим схема использует тип Palette, являющийся объектом, который содержит массив цветов, каждый при этом цвет является строкой. Соответственно для нашего сайта доступ к этой части api будет реализован через url http://127.0.0.1:5000/sitepart/colors/all/, или http://127.0.0.1:5000/sitepart/colors/rgb/ и т. д. При этом вы получите, например, такой результат (рисунок 4.4) или в формате json

```
{
  "cmyk": [
    "cian",
    "magenta",
    "yellow",
    "black"
  ],
  "rgb": [
    "red",
    "green",
    "blue"
  ]
}
```

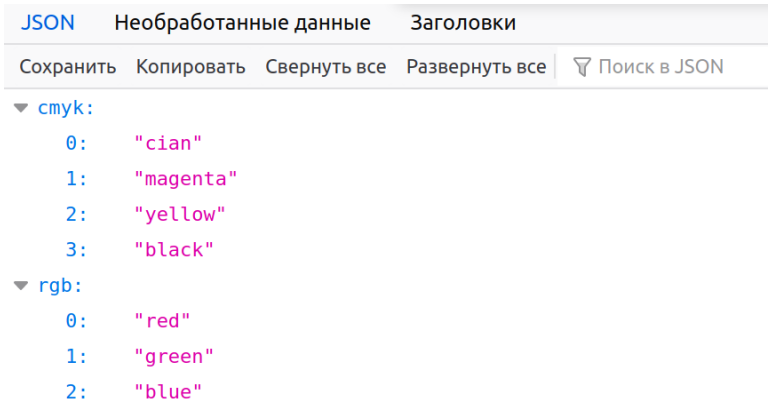


Рисунок 4.4 - Результат запроса по url к нашему api

Результат запроса apidocs отображен на рисунке 4.5.

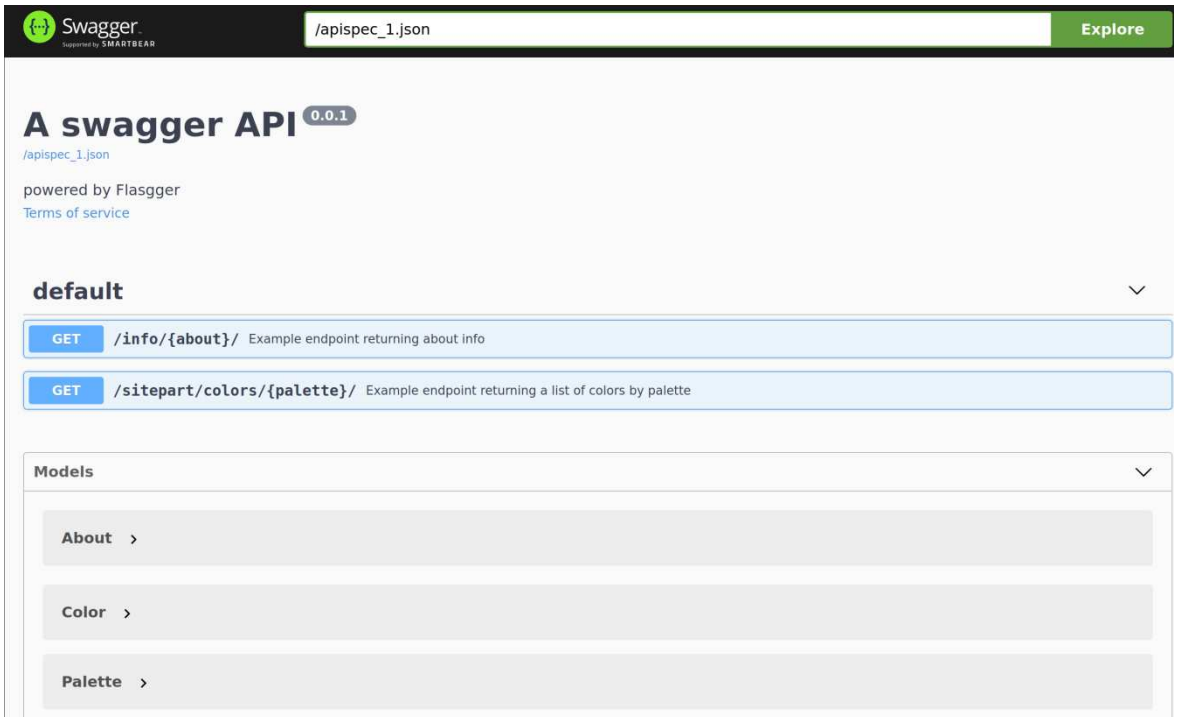


Рисунок 4.5 - Пример отображения информации об api.

4.8 Балансировка нагрузки.

В главе 2 были кратко рассмотрены основные элементы современного веб-приложения, в том числе в таких приложениях применяется технология балансировки нагрузки.

4.8.1 Уровни балансировки

Процедура балансировки осуществляется при помощи целого комплекса алгоритмов и методов, соответствующим следующим уровням модели OSI:

- сетевому;
- транспортному (L4);
- прикладному (L7).

4.8.1.1 Балансировка на сетевом уровне

Балансировка на сетевом уровне предполагает решение следующей задачи: нужно сделать так, чтобы за один конкретный IP-адрес сервера отвеча-

ли разные физические машины. Такая балансировка может осуществляться с помощью множества разнообразных способов.

DNS-балансировка. На одно доменное имя выделяется несколько IP-адресов. Сервер, на который будет направлен клиентский запрос, обычно определяется с помощью алгоритма Round Robin, предполагающий циклическую передачу на сервер со следующим по кругу IP для данного доменного имени, так как DNS выдает адрес клиенту при запросе браузера, то он обращается к «своему» IP серверу.

Построение NLB-кластера. При использовании этого способа серверы объединяются в кластер, состоящий из входных и вычислительных узлов. Распределение нагрузки осуществляется при помощи специального алгоритма. Используется в решениях от компании Microsoft.

Балансировка по IP с использованием дополнительного маршрутизатора. Например, по IP отправителя можно передавать пакеты одному какому-то серверу, а с другими IP передавать на другой сервер. В маршрутизаторах Cisco есть режим распределения потоков пакетов по хэшу от ip адресов, а также от порта отправителя и данных HTTP запросов (но это уже переводит на транспортный и прикладной уровень). Кроме того, существует балансировка в сетях по адресу назначения.

Балансировка по территориальному признаку осуществляется путём размещения одинаковых сервисов с одинаковыми адресами в территориально различных регионах Интернета (так работает технология Anycast DNS). Балансировка по территориальному признаку также используется во многих CDN.

4.8.1.2 Балансировка на транспортном уровне

При данном виде балансировки клиент обращается к балансировщику, тот перенаправляет запрос одному из серверов, который и будет его обрабатывать. Выбор сервера, на котором будет обрабатываться запрос, может осуществляться в соответствии с самыми разными алгоритмами: путём простого

кругового перебора, путём выбора наименее загруженного сервера из пула и т.п.

Различие между транспортной и сетевой балансировкой в том, что при сетевой балансировке просто идет перенаправление сетевого пакетного трафика и нет работы в проксирующем режиме. На транспортном уровне общение с клиентом замыкается на балансировщике, который работает как прокси. Он взаимодействует с серверами от своего имени, передавая информацию о клиенте в дополнительных данных и заголовках. Таким образом, работает, например, популярный программный балансировщик HAProxy.

4.8.1.3 Балансировка на прикладном уровне

При балансировке на прикладном уровне балансировщик работает в режиме «умного прокси». Он анализирует клиентские запросы и перенаправляет их на разные серверы в зависимости от характера запрашиваемого контента. Так работает, например, веб-сервер Nginx, распределяя запросы между фронтендом и бэкендом. За балансировку в Nginx отвечает модуль Upstream.

В качестве ещё одного примера инструмента балансировки на прикладном уровне можно привести pgpool — промежуточный слой между клиентом и сервером СУБД PostgreSQL. С его помощью можно распределять запросы по серверам баз данных в зависимости от их содержания: например, запросы на чтение будут передаваться на один сервер, а запросы на запись — на другой.

4.8.2 Алгоритмы и методы балансировки

При выборе алгоритмов балансировки, могут ставиться различные цели и выбираться различные критерии, связанные, например, со справедливостью распределения системных ресурсов между всеми запросами, эффективностью распределения нагрузки между серверами, чтобы какие-то сервера не простаивали, скоростью выполнения запросов и сокращения времени отклика на запрос пользователя.

Так же алгоритм балансировки должен быть предсказуемым, то есть давать ясное представление о нагрузках и задачах, при которых будет оставаться эффективным, масштабируемым, сохраняя работоспособность при увеличении нагрузки.

4.8.2.1 Round Robin

***Round Robin**, или **алгоритм кругового обслуживания**, представляет собой перебор по круговому циклу: первый запрос передаётся одному серверу, затем следующий запрос передаётся другому и так до достижения последнего сервера, а затем всё начинается сначала.*

Самой распространёной реализацией этого алгоритма является метод балансировки Round Robin DNS. Как известно, любой DNS-сервер хранит пару «имя хоста — IP-адрес» для каждой машины в определённом домене. Этот список может выглядеть, например, так:

example.com xxx.xxx.xxx.2

www.example.com xxx.xxx.xxx.3

С каждым именем из списка можно ассоциировать несколько IP-адресов:

example.com xxx.xxx.xxx.2

www.example.com xxx.xxx.xxx.3

www.example.com xxx.xxx.xxx.4

www.example.com xxx.xxx.xxx.5

www.example.com xxx.xxx.xxx.6

DNS-сервер проходит по всем записям таблицы и отдаёт на каждый новый запрос следующий IP-адрес: например, на первый запрос — xxx.xxx.xxx.2, на второй — xxx.xxx.xxx.3, и так далее. В результате все серверы в кластере получают одинаковое количество запросов.

Достоинством алгоритма является независимость от протоколов более высоких уровней, возможность использования любого протокола, обращающегося к серверу по доменному имени. Балансировка на основе алгоритма Round Robin никак не зависит от нагрузки на сервер: кэширующие DNS-серверы помогут справиться с любым наплывом клиентов, не требуется связи между серверами и его можно использовать как для локальной, так и для глобальной балансировки, низкая стоимость, достаточно просто добавить несколько записей DNS.

Но также алгоритм Round Robin имеет и целый ряд существенных недостатков. Для эффективного и справедливого распределения нагрузки у каждого сервера должен быть одинаковый набор ресурсов, совершенно не учитывается загруженность того или иного сервера в составе кластера, не учитывается количество активных на данный момент подключений сервера. Представим себе следующую гипотетическую ситуацию: один из узлов загружен почти на 100%, в то время как другие на порядок меньше. Алгоритм Round Robin такой ситуации не учитывает в принципе, поэтому перегруженный узел все равно будет получать запросы, потому сфера применения алгоритма Round Robin весьма ограничена.

4.8.2.2 Weighted Round Robin

Версия алгоритма Round Robin учитывающая весовой коэффициент каждого сервера в соответствии с его производительностью и мощностью, что помогает распределять нагрузку более гибко: серверы с большим весом обрабатывают больше запросов, хотя это всех проблем с отказоустойчивостью не решает.

4.8.2.3 Least Connections и и другие алгоритмы

Призван учесть количество активных на данный момент подключений. Рассмотрим практический пример. Имеется два сервера — обозначим их условно как А и Б. К серверу А подключалось меньше пользователей за какое-то время, чем к серверу Б. При этом сервер А оказывается более перегружен-

ным. Как это возможно? Ответ достаточно прост: подключения к серверу А поддерживаются в течение более долгого времени по сравнению с подключениями к серверу Б.

Алгоритм учитывает количество подключений, поддерживаемых серверами в текущий момент времени. Каждый следующий вопрос передаётся серверу с наименьшим количеством активных подключений.

Существует усовершенствованные варианты данного алгоритма, один из них называемый Weighted Least Connections учитывает при распределении нагрузки не только количество активных подключений, но и весовой коэффициент серверов.

Другой Locality-Based Least Connection Scheduling (созданный специально для кэширующих прокси-серверов) маршрутизирует пакет, направленный по определенному IP адресу, серверу обслуживающему этот адрес, однако если тот перегружен, и существует другой сервер загруженный наполовину, этот IP адрес будет закреплен за менее загруженным сервером.

В алгоритме Locality-Based Least Connection Scheduling with Replication Scheduling каждый IP-адрес или группа IP-адресов закрепляется не за отдельным сервером, а за целой группой серверов. Запрос передаётся наименее загруженному серверу из группы. Если же все серверы из группы данного IP перегружены, то будет зарезервирован новый сервер. Этот новый сервер будет добавлен к группе, обслуживающей IP, с которого был отправлен запрос. В свою очередь наиболее загруженный сервер из этой группы будет удалён — это позволяет избежать избыточной репликации.

Алгоритм Destination Hash Scheduling был создан для работы с кластером кэширующих прокси-серверов, но он часто используется и в других случаях. В этом алгоритме сервер, обрабатывающий запрос, выбирается из статической таблицы по IP-адресу получателя.

Алгоритм Source Hash Scheduling основывается на тех же самых принципах, что и предыдущий, только сервер, который будет обрабатывать за-

прос, выбирается из таблицы по IP-адресу отправителя. Этот алгоритм разработан для LVS маршрутизаторов с несколькими межсетевыми экранами.

Sticky Sessions — алгоритм распределения входящих запросов, при котором соединения передаются на один и тот же сервер группы. Он используется, например, в веб-сервере Nginx. Сессии пользователя могут быть закреплены за конкретным сервером с помощью метода IP hash. С помощью этого метода запросы распределяются по серверам на основе IP-адреса клиента. Как указано в документации, «метод гарантирует, что запросы одного и того же клиента будет передаваться на один и тот же сервер». Если закреплённый за конкретным адресом сервер недоступен, запрос будет перенаправлен на другой сервер. Пример фрагмента конфигурационного файла:

```
upstream backend {
ip_hash;
server backend1.example.com;
server backend2.example.com;
server backend3.example.com;
server backend4.example.com;
}
```

Начиная с версии 1.2.2 в Nginx для каждого сервера можно указывать вес.

Применение этого метода сопряжено с некоторыми проблемами. Проблемы с привязкой сессий могут возникнуть, если клиент использует динамический IP. В ситуации, когда большое количество запросов проходит через один прокси-сервер, балансировку вряд ли можно назвать эффективной и справедливой. Описанные проблемы, однако, можно решить с помощью портов, которые могут быть распределены более равномерно или используя cookies. В коммерческой версии Nginx имеется специальный модуль sticky, который как раз использует cookies для балансировки. Есть у него и бесплатные аналоги — например, nginx-sticky-module. Можно использовать метод sticky-sessions и в HAProxy.

4.9 Аппаратное масштабирование архитектуры

Различают вертикальное и горизонтальное аппаратное масштабирование. Рассмотрим основные различия между первой и второй архитектурой. Серверы с вертикальным масштабированием - это большие SMP- системы (с симметричной многопроцессорной обработкой или совместно используемой памятью), насчитывающие свыше четырех центральных процессоров. В них используется только одна копия ОС, управляющая работой всех процессоров, памяти и компонентов ввода-вывода.

Обычно все эти ресурсы размещены в одной стойке или шкафу. Соединения у таких серверов осуществляются по высокоскоростной центральной или объединительной панели с небольшим временем запаздывания и согласованным доступом к кэш- памяти. Добавить ресурсы можно путем установки внутрь шкафа дополнительных системных плат. В таких системах с вертикальной архитектурой (или SMP-системах) память используется совместно, т.е. все процессоры и компоненты ввода-вывода получают доступ ко всей памяти. Пользователь «видит» память как единый большой объект.

При альтернативном, горизонтальном масштабировании системы соединяются через сеть или объединяются в кластер. Для соединений обычно используются стандартные сетевые технологии, такие, как Fast Ethernet, Gigabit Ethernet (GBE) и Scalable Coherent Interconnect (SCI), дающие меньшую пропускную способность и большее запаздывание по сравнению с вертикальными системами. Ресурсы в этом случае распределяются между узлами, обычно содержащими от одного до четырех процессоров; каждый узел имеет собственный процессор и память и может иметь собственную подсистему ввода-вывода или использовать ее совместно с другими узлами. На каждом узле работает отдельная копия ОС. Ресурсы расширяются за счет добавления узлов, но не добавления ресурсов в узел. Память в горизонтальных системах распределена, т.е. у каждого узла есть собственная память, к которой напрямую обращаются его процессоры и подсистема ввода-вывода. Доступ к этим ресурсам с другого узла происходит намного медленнее, чем с уз-

ла, где они расположены. Кроме того, при горизонтальной архитектуре отсутствует согласованный доступ узлов к памяти, а используемые приложения потребляют относительно немного ресурсов, поэтому они «умещаются» на одном узле и им не нужен согласованный доступ. Если же приложению потребуется несколько узлов, то оно само должно обеспечить согласованный доступ к памяти. Горизонтальная система удовлетворяет требованиям приложений, такая архитектура предпочтительна, поскольку расходы на ее приобретение меньше. Обычно стоимость приобретения в расчете на один процессор у горизонтальных систем ниже, чем у вертикальных. Разница в цене объясняется тем, что в вертикальных системах применяются более мощные функции надежности, доступности и обслуживаемости - RAS (reliability, availability, serviceability), а также высокопроизводительные соединения.

Масштабирование баз данных

Зачастую узким местом в современных приложениях является БД. Проблемы с ней делятся, как правило, на два класса: производительность и необходимость хранения большого количества данных. В основе масштабирования данных лежит тот же принцип, что и в основе аппаратного масштабирования, это разделение данных на группы и выделение их на отдельные сервера. Существует две основные стратегии - репликация и шардинг.

Репликация

Репликация позволяет создать полный дубликат базы данных. Так, вместо одного сервера теперь будет несколько. Чаще всего используют схему Master - Slave:

а) Master - это основной сервер БД, куда поступают все данные. Все изменения в данных (добавление, обновление, удаление) должны происходить на этом сервере;

б) Slave - это вспомогательный сервер БД, который копирует все данные с мастера. С этого сервера следует читать данные. Таких серверов может быть несколько.

Репликация позволяет использовать два или больше одинаковых серверов вместо одного. Операций чтения (SELECT) данных часто намного больше, чем операций изменения данных (INSERT/UPDATE). Поэтому, репликация позволяет разгрузить основной сервер за счет переноса операций чтения на Slave.

Важно отметить, что репликация сама по себе не очень удобный механизм масштабирования. Причина - рассинхронизация данных и задержки в копировании с Master на Slave. Зато это отличное средство для обеспечения отказоустойчивости. Всегда можно переключиться на Slave, если Master ломается и наоборот. Чаще всего репликация используется совместно с шардингом именно из соображений надежности.

Шардинг - это другая техника масштабирования работы с данными. Суть его в разделении базы данных на отдельные части, так, чтобы каждую из них можно было вынести на отдельный сервер. Этот процесс зависит от структуры базы данных и выполняется прямо в приложении в отличие от репликации.

Рассмотрим несколько способов данного метода, различают вертикальный и горизонтальный шардинг.

Вертикальный шардинг - это выделение таблицы или группы таблиц на отдельный сервер. Например, в приложении есть такие таблицы:

а) users - данные пользователей;

б) photos - фотографии пользователей;

в) albums - альбомы пользователей.

Таблицу users можно оставить на одном сервере, а таблицы photos и albums перенести на другой. В таком случае в приложении необходимо будет использовать соответствующее соединение для работы с каждой таблицей.

В отличие от репликации, используются разные соединения для любых операций, но с определенными таблицами.

Горизонтальный шардинг - это разделение и перенос одной таблицы на разные сервера. Это необходимо использовать для огромных таблиц, которые

не умещаются на одном сервере. Разделение таблицы на части делается по такому принципу:

а) на нескольких серверах создается одна и та же таблица (только структура, без данных);

б) в приложении выбирается условие, по которому будет определяться нужное соединение (например, четные на один сервер, а нечетные - на другой);

в) перед каждым обращением к таблице происходит выбор нужного соединения.

Допустим, наше приложение работает с огромной таблицей, которая хранит фотографии пользователей. Существуют два сервера (обычно они называются шардами) для этой таблицы. Для нечетных пользователей – первый сервер, а для четных - второй. Таким образом, на каждом из серверов будет только часть всех данных о фотографиях пользователей.

Не следует применять технику шардинга ко всем таблицам. Правильный подход - это поэтапный процесс разделения растущих таблиц. Следует задумываться о горизонтальном шардинге, когда количество записей в одной таблице переходит за пределы от нескольких десятков миллионов до сотен миллионов.

4.10 CDN.

CDN *(Content Delivery Network) — это географически распределённая сетевая инфраструктура, обеспечивающая быструю доставку контента пользователям веб-сервисов и сайтов. Входящие в состав CDN серверы географически располагаются таким образом, чтобы сделать время ответа для пользователей сайта/сервиса минимальным*

Обычно сервер, на котором хранятся исходные файлы или данные, раздаваемые через CDN называют Ориджин (origin).

Сервер являющийся кэширующим в составе CDN и расположенный в определенной географической локации называют PoP (point of presence, точка присутствия). Для обозначения таких серверов также используется термин edge.

Доставляемый пользователям контент бывает твух типов, динамическим генериуруемым на сервере в момент получения запроса (либо изменяемый пользователем, либо загружаемый из базы данных) и статическим, хранимым на сервере в неизменяемом виде (например, бинарные файлы, аудио- и видеофайлы, JS и CSS).

Обычно CDN-хостинг предоставляют провайдеры. Они размещают сеть взаимосвязанных кеширующих CDN-серверов в разных точках мира. За счёт этого расстояние между клиентами и основным сервером не влияет на скорость передачи данных. Сайты, которые используют CDN-хостинг, загружаются быстрее, за счет автоматической переадресации к ближайшему CDN-серверу (например, в Новосибирске) и передачи части сайта (статический контент) на серверы из CDN-сетей, а динамический контент остаётся на основном сервере. Таким образом, еще и распределяется нагрузка, и страницы загружаются быстрее. Запросы к сайту будут обрабатывать сразу несколько серверов в зависимости от месторасположения «клиентов». Это значит, что нагрузка распределится равномерно, и сайт продолжит работу, несмотря на попытки снизить производительность ресурса.

Разумеется, далеко не всем сайтам требуется CDN-хостинг. В частности, проблем со скоростью загрузки контента может не быть на сайтах с небольшим количеством статического содержимого или интернет-магазинах, которые ориентируются на локальную аудиторию (город или область). Но крупным интернет-магазинам с аудиторией в разных регионах (например, Amazon, eBay, AliExpress) такая услуга необходима. Особенно чтобы сайт не «падал» при больших нагрузках в моменты распродаж на Чёрную пятницу

или под Новый год. Так же может требоваться стриминговым аудио- и видеосервисам, чтобы передавать большое количество контента в разные точки мира. Например, Spotify, игровым порталам. Например, облачному геймингу или платформам для дистрибуции ПО. Компаниям, у которых есть мобильное приложение. Технологию CDN также применяют, если у проекта есть мобильное приложение, и оно должно быстро работать.

Для настройки раздачи статического контента через CDN необходимо выполнить следующие шаги:

Шаг 1: Вынести статитечскую часть контента сайта на отдельный домен, например, static.example.com — это будет origin.

Шаг 2: Для работы через CDN создать домен вида cdn.example.com.

Шаг 3: Подключить CDN у провайдера. Для подключения владельцу веб-сервиса необходимо сообщить провайдеру следующее: домен, с которого он будет забирать статику — static.example.com; домен, с которого будет идти раздача — cdn.example.com.

Шаг 4: У своего DNS-регистратора настроить CNAME запись с cdn.example.com на домен CDN-провайдера, который CDN провайдер выделяет при подключении.

Например, в CDN Selectel такой домен имеет вид 85e77c09-bc03-43bf-b8f3-9492ae33390f.selcdn.net, где 85e72c09-bc03-43bf-b8f3-9492ae33390f генерируется автоматически.

Шаг 5: На своем сайте изменить домен для статического контента, который планируется раздавать через CDN, на cdn.example.com.

Пользователь набирает в строке браузера адрес www.example.com, с которого он получает HTML-страницу. При этом весь статический контент, например, графические изображения, подгружается из CDN (с адреса cdn.example.com).

Веб-сервис после подключения CDN будет работать на том же оригинальном сервере. Кэшированные части сайта будут загружены на серверы

CDN-сети. Система находит для пользователя ближайший сервер и максимально быстро загружает статику сайта с него.

Обратите внимание на один важный момент: серверы, входящие в состав CDN, не являются подобием файловых серверов, на которые контент размещается для последующего скачивания. CDN используются не для хранения контента, а для кэширования на основе конкретных алгоритмов.

Самой распространенной схемой кэширования является схема по первому обращению: максимальное количество времени на загрузку затрачивает пользователь, обратившийся к оригинальному серверу первым. Все последующие пользователи будут получать данные, кэшированные на ближайшей к ним точке присутствия.

Для преодоления ограничений, накладываемых этой схемой, используются технологии регионального извлечения: соседние серверы, входящие в состав CDN, забирают контент друг у друга, а не обращаются к оригинальному серверу.

В большинстве CDN пользователь, отправивший запрос на получение статического контента, переадресуется к ближайшей точке присутствия и получает кэшированную версию этого контента с неё.

CloudFlare CDN и его преимущества.

CDN – это краткая форма сети доставки контента. На рынке доступно много услуг CDN, и CloudFlare является одним из них. CloudFlare один из лучших бесплатных сервисов для использования.

Особенность CloudFlare CDN заключается в том, что они используют межсетевой экран веб-сайта и прокси-сервер облачной базы, что означает, что он отслеживает весь входящий трафик, поступающий на ваш сайт, и таким образом блокирует подозрительный трафик. Это эффективная система против атаки DDos и вредоносного трафика, которая заставляет их вручную вводить captcha, прежде чем они попадут на ваш сайт.

CloudFlare CDN блокирует вредоносных ботов, помечая вредоносные IP-адреса, что экономит вычслительные ресурсы и ресурсы, которые не затрачиваются на бесполезынй вредоносный трафик.

4.11 Потоковые сервисы

Потоковые сервисы предназначены для агрегирования больших потоков данных, среди них есть различные представители, некоторые из них мы рассмотрим.

Amazon Kinesis Data Streams (KDS) – это сервис для потоковой передачи данных в режиме реального времени и возможностями масштабирования. KDS может непрерывно выполнять сбор данных со скоростью несколько гигабайт в секунду из сотен тысяч источников, таких как истории посещений веб-сайтов, потоки событий баз данных, финансовые транзакции, ленты социальных сетей, ИТ-журналы и потоки событий отслеживания местоположения. Собранные данные через доли секунды становятся доступными для приложений с использованием анализа в режиме реального времени, например, для панелей управления или систем обнаружения аномалий в режиме реального времени, систем динамического ценообразования и других примеров использования.

В течение 70 миллисекунд с момента сбора потоковые данные становятся доступными для анализа в режиме реального времени с помощью различных приложений, сохранения в Amazon S3 или обработки с помощью AWS Lambda.

Сервис обеспечивает несколько уровней защиты от потери данных. Потоковые данные синхронно реплицируются в три зоны доступности в рамках региона AWS и хранятся до семи дней.

Для обеспечения соответствия законодательным и нормативным требованиям можно шифровать конфиденциальные данные в KDS и использовать частный доступ к ним в рамках Amazon Virtual Private Cloud (VPC). При хра-

нении данных можно использовать для их защиты шифрование на стороне сервера и главные ключи AWS KMS.

Можно динамически масштабировать приложения. Потоки данных Kinesis могут масштабироваться от нескольких мегабайт до нескольких терабайт в час и выполнять как тысячи, так и миллионы записей PUT в секунду. Пропускную способность потока можно динамически регулировать в любое время в зависимости от объема входящих данных.

Сбор данных журналов и событий

Kinesis Data Streams можно использовать для сбора данных журналов и событий из таких источников, как серверы, стационарные компьютеры или мобильные устройства. Можно создавать приложения Kinesis, которые будут выполнять непрерывную обработку данных, генерировать метрики, работать с информационными панелями в режиме реального времени и передавать сводные данные в хранилища, например, в Amazon S3.

Аналитика в режиме реального времени

В приложениях Kinesis можно выполнять аналитические вычисления в режиме реального времени, обрабатывая данные высокочастотных событий, например, показания сенсоров, собираемые Kinesis Data Streams. Это дает возможность получать аналитическую информацию каждые несколько минут, а не часов или дней.

Захват данных мобильных приложений

Мобильные приложения могут передавать данные в Kinesis Data Streams с сотен тысяч устройств. Данные будут доступны сразу же после их создания мобильными устройствами.

Игровые данные

Kinesis Data Streams может непрерывно собирать данные о взаимодействии игроков с приложением и передавать их на игровую платформу. Используя данные о действиях и поведении игроков, собираемые Kinesis Data Streams, можно проектировать увлекательные динамичные игры.

Amazon Kinesis Data Firehose предоставляет простейший способ надежной загрузки данных потоковой передачи в озера данных, хранилища и сервисы аналитики. Этот сервис позволяет захватывать, преобразовывать и доставлять данные потоковой передачи в Amazon S3, Amazon Redshift, Amazon Elasticsearch Service, на общие HTTP-адреса и поставщикам таких сервисов, как Datadog, New Relic, MongoDB и Splunk. Этот полностью управляемый сервис автоматически масштабируется в зависимости от пропускной способности потока данных и не требует постоянного администрирования. Amazon Kinesis Data Firehose позволяет создавать пакеты данных, а также сжимать, преобразовывать и шифровать потоки данных перед загрузкой, что сокращает объем используемой памяти и повышает уровень безопасности.

Всего за несколько минут с помощью Консоли управления AWS можно создать поток доставки данных Firehose и запустить передачу данных из сотен тысяч источников в указанные целевые объекты. Кроме того, в потоках данных можно настроить автоматическое преобразование входящих данных в открытые и основанные на стандартах форматы, такие как Apache Parquet и Apache ORC, перед доставкой.

В Amazon Kinesis Data Firehose не предусмотрена плата за настройку и нет минимальных платежей. Вы платите только за объем данных, переданных через сервис, а также за преобразование формата данных и за доставку и передачу данных через Amazon VPC, если эти возможности используются.

Простота использования

Amazon Kinesis Data Firehose позволяет захватывать, преобразовывать и загружать данные потоковой передачи за несколько щелчков мышью в Консоли управления AWS. Вы можете быстро создать поток доставки данных Firehose, выбрать целевое расположение и запустить передачу данных в режиме реального времени сразу из сотен тысяч источников. Сервис автоматически управляет потоком данных, в том числе масштабированием, сегментированием и мониторингом, необходимыми для последовательной загрузки данных в целевые расположения с заданными интервалами.

Интеграция с сервисами AWS и поставщиками сервисов

Сервис Amazon Kinesis Data Firehose интегрирован с Amazon S3, Amazon Redshift и Amazon Elasticsearch Service. С помощью Консоли управления AWS можно указать для Kinesis Data Firehose целевые объекты на выбор и использовать существующие приложения и инструменты для анализа потоковых данных.

Бессерверное преобразование данных

Amazon Kinesis Data Firehose позволяет подготавливать данные потоковой передачи перед их загрузкой в хранилища данных. С помощью Kinesis Data Firehose можно просто преобразовывать необработанные данные потоковой передачи, поступающие из источников данных, в форматы, необходимые для целевых хранилищ данных (таких как Apache Parquet и Apache ORC), без необходимости создания собственных конвейеров обработки данных.

Почти в режиме реального времени

Amazon Kinesis Data Firehose позволяет захватывать и загружать данные в режиме, близком к реальному времени. Данные, отправленные в этот сервис, загружаются в целевые объекты в течение 60 секунд. В результате вы можете оперативно получать доступ к новым данным и быстро реагировать на события, связанные с бизнесом и управлением.

Без постоянного администрирования

Amazon Kinesis Data Firehose – это полностью управляемый сервис, который автоматически выделяет и масштабирует вычислительные, сетевые ресурсы и память, необходимые для обработки и загрузки данных потоковой передачи, а также осуществляет управление ими. Достаточно один раз задать параметры Kinesis Data Firehose, и этот сервис будет непрерывно загружать поступающие данные в целевые объекты.

Оплата по факту использования

При использовании Amazon Kinesis Data Firehose оплате подлежит только объем данных, передаваемых через сервис, и преобразование формата

данных, если таковое используется. Кроме того, при необходимости оплачивается доставка и передача данных через Amazon VPC. Минимальные платежи и авансовые обязательства отсутствуют.

4.12 Кэширование.

Кэширование между веб приложением и между базой данных часто необходимо с целью ускорения доступа к данным и обеспечения малого времени отклика системы на пользовательские запросы. Представим ситуацию, что в некой реляционной СУБД хранится информация о пользователях, каждый раз для авторизации пользователя реализуется запрос sql в данное СУБД на который возвращается логин и хэш пароля или хэш логина и пароля пользователя, при этом в соответствии с трехзвенной архитектурой СУБД находится на другом физическом сервере. На подобный запрос тратится время, связанное с доступом к самой СУБД по сети, доступом СУБД к своему кэшу или физической базе данных. Потому в высоконагруженных системах чаще всего реализуют кэширование данных на стороне веб приложения, например, с использованием redis или memcached. Это так называемые не реляционные базы данных или NO SQL базы, хранящие информацию в оперативной памяти, в некоторых проектах их используют непосредственно без использования реляционной СУБД. Если ваша система не является высоконагруженной или вы не реализуете трехзвенную архитектуру, то возможно использование подобного кэша и не нужно, но при увеличении нагрузки может быть придется добавлять такой кэш и перестраивать проект.

Redis – быстрое хранилище в памяти с открытым исходным кодом для структур данных «ключ-значение». Redis поставляется с набором разнообразных структур данных в памяти, что упрощает создание различных специальных приложений. Самые распространенные примеры использования Redis включают кэширование, управление сессиями, системы «издатель-подписчик» и таблицы лидеров. Оно обладает лицензией BSD, написано на

оптимизированном коде C и поддерживает несколько языков разработки. Название Redis является акронимом от REmote DIctionary Server.

Благодаря высокой скорости и простоте Redis часто используется для мобильных и интернет-приложений, игр, рекламных платформ, «Интернета вещей», т.е. в тех случаях, когда необходима максимально возможная производительность. Большинство PaaS и фреймворков поддерживают Redis, например, Amazon ElastiCache для AWS или flask_cache, Django-redis, php5-redis.

Все данные Redis находятся в оперативной памяти сервера, в отличие от большинства систем управления базами данных, в которых данные хранятся на жестких дисках или твердотельных накопителях (SSD). Размещаемые в памяти базы данных, такие как Redis, не требуют доступа к дисковым накопителям, что позволяет избежать потерь времени на поиск. Доступ к данным можно организовать при помощи более простых алгоритмов, в которых используется меньшее количество инструкций ЦПУ. Для выполнения типовых операций требуется менее миллисекунды.

Redis позволяет пользователям хранить ключи, привязанные к различным типам данных. Основной тип данных – это строка, которая может состоять из текстовых или двоичных данных размером до 512 МБ. Redis также поддерживает списки строк (List of Strings), упорядоченные в порядке вставки; множества неупорядоченных строк (Sets of unordered Strings); упорядоченные по результату множества (Sorted Sets); хеш-таблицы (Hashes), содержащие список полей и значений, и тип данных HyperLogLogs для подсчета уникальных элементов в наборе данных. С помощью Redis в памяти можно хранить практически любые типы данных.

Redis поставляется с несколькими инструментами, ускоряющими и облегчающими разработку и эксплуатацию, включая шаблон «издатель-подписчик», для публикации сообщений в логических каналах, доставляемых подписчикам (хорошо подходит для чатов и систем обмена сообщениями); TTL: ключи могут иметь определенное время жизни (TTL), после чего они

самостоятельно уничтожаются, это используется, чтобы избежать заполнения базы данных ненужными данными; атомарные счетчики, которые обеспечивают непротиворечивость результатов в ситуации состязания; а также Lua, мощный, но несложный язык скриптов.

В Redis применяется архитектура «master–slave» и поддерживается асинхронная репликация, при которой данные могут реплицироваться на несколько подчиненных серверов. Это обеспечивает как улучшенные характеристики чтения (так как запросы могут быть распределены между серверами), так и восстановление при отключении основного сервера.

Для обеспечения надежности Redis поддерживает как снимки состояния на момент времени (копирование наборов данных Redis на диски), так и создание файла только для добавления (AOF) для последовательной записи на диск всех изменений данных. Любой из этих методов обеспечивает быстрое восстановление данных Redis в случае сбоя.

Кроме кэширования Redis используется для управления сессиями, в том числе как быстрое хранилище пар «ключ-значение» с соответствующим временем жизни (TTL) ключей сессии для управления передачей информации внутри сессии, в указанном примере с авторизацией можно создать соответствующий токен и сохранить его в redis. Управление сессиями обычно требуется для интернет-приложений, включая игры, сайты электронной коммерции и платформы социальных сетей.

С помощью структуры данных Redis Sorted Set можно хранить элементы в отсортированном списке. Это позволяет легко создавать динамические таблицы сортируемых данных, размещать наиболее понравившиеся сообщения.

Redis может измерять и при необходимости ограничивать скорость наступления событий. Используя счетчик Redis, связанный с ключом API клиента, можно подсчитать количество запросов за определенный период времени и принять меры при превышении лимитов. Ограничители интенсивности обычно используются для установки лимита по количеству публикаций

на форуме, лимитирования использования ресурсов и уменьшения влияния спам-атак.

Структура данных Redis List позволяет легко создавать упрощенные постоянные очереди. Списки Redis List обеспечивают выполнение элементарных операций, а также возможности блокировки, поэтому они подходят для различных приложений, в которых требуется надежный брокер сообщений или циклический список.

Redis поддерживает шаблон проектирования «издатель-подписчик», а также сопоставление с образцом. Это позволяет использовать Redis для создания высокопроизводительных комнат чата, лент комментариев, работающих в режиме реального времени, и систем взаимодействия серверов. Шаблон «издатель-подписчик» также можно использовать для запуска действий на основе опубликованных событий.

Пример листинга кода на Flask приведен ниже.

```
import time
from flask import Flask
from flask_cache import Cache

app = Flask(__name__)
# cache = Cache(app, config={'CACHE_TYPE': 'redis'})
cache = Cache(app, config={
    'CACHE_TYPE': 'redis',
    'CACHE_KEY_PREFIX': 'fcache',
    'CACHE_REDIS_HOST': 'localhost',
    'CACHE_REDIS_PORT': '6379',
    'CACHE_REDIS_URL': 'redis://localhost:6379'
    })

@cache.memoize(timeout=60)
def query_db():
    time.sleep(5)
    return "Results from DB"
```

```
@app.route('/')
def index():
    return query_db()

app.run(debug=True)
```

Здесь используется декоратор, который указывает на кэширование запросов из баз данных.

Контрольные вопросы по главе 4

1. Для чего может понадобиться кэширование между СУБД и Redis если сама СУБД обычно хранит все данные в промежуточном буфере?
2. Основное назначение сетей CDN?
3. Что такое соккет?
4. Основные запросы протокола HTTP?
5. Что представдяют из себя cookie и для чего они нужны?
6. Для чего используются атрибуты заголовка запросов и ответов HTTP if_modified_since, last_modified?
7. Для чего нужна сертификация и центры сертификации?
8. Что такое Round-Robin DNS?

ЗАКЛЮЧЕНИЕ

В данном учебном пособии мы рассмотрели всего лишь часть технологий, использующихся в интернет взаимодействии, но данное начальное пособие позволит начать путь, связанный с использованием и разработкой веб-приложений. Несомненно, существующих технологий веб-разработки огромное множество, мы не касались подробно протоколов нижних уровней взаимодействия, так как их более подробно рассматривают на дисциплинах, связанных с изучением компьютерных сетей. Тем не менее, были освещены современная архитектура веб-сервисов и многие из подходов, которые непосредственно используются в современной разработке веб-приложений. Технологии такого рода довольно быстро развиваются, например, те же технологии Swagger обеспечивают генерацию готового кода серверной и клиентской части приложения на основе декларативного описания. Несомненно, будущее данной области может быть связано с наполнением сервисов необходимой функциональной частью, а каркас приложения будет генерироваться автоматически, этот шаг является логическим продолжением, начавшийся с использования фреймворков. Более того, разрабатываются технологии на основе нейронных сетей, позволяющие генерировать непосредственно код и предлагать алгоритмическую реализацию на основе уже написанного кода, что еще более освобождает программиста от рутинных действий, возлагая на него функции больше связанные с архитектурными решениями или с непосредственным прикладным назначением сервиса. Дальнейшие задачи, с которыми сталкивается разработчик веб-сервисов связаны с такими дисциплинами как Базы данных, Сети и телекоммуникации, Защита информации, Операционные системы. Потому в данном пособии приведены лишь начальные сведения, которые позволят частично погрузиться в задачи, которые необходимо решать в будущем, дающие общее направление в которое необходимо погрузиться, чтобы на высоком уровне вести разработку веб-сервисов и высоконагруженных систем.

ИСПОЛЬЗУЕМЫЕ ИСТОЧНИКИ ЛИТЕРАТУРЫ

1. Бренделева Е. А. QWERTY-эффекты, институциональные ловушки с точки зрения теории трансакционных издержек // Пространство экономики. 2006. №2. URL: https://cyberleninka.ru/article/n/qwerty-effekty-institutsionalnye-lovushki-s-tochki-zreniya-teorii-transaktsionnyh-izderzhek (дата обращения: 12.08.2020).

2. Корнейчук Б.В. «Эффект колеи» в контексте эволюционной теории экономических изменений // Пространство экономики. 2016. №1. URL: https://cyberleninka.ru/article/n/effekt-kolei-v-kontekste-evolyutsionnoy-teorii-ekonomicheskih-izmeneniy (дата обращения: 12.08.2020).

3. О Международном союзе электросвязи (МСЭ) https://www.itu.int/ru/about/Pages/default.aspx (дата обращения: 12.08.2020).

4. ITU-T Recommendations. ITU-T X.200 (07/1994). https://www.itu.int/ITU-T/recommendations/rec.aspx?id=2820 (дата обращения: 12.08.2020).

5. Сертификация. https://www.iso.org/ru/certification.html (дата обращения: 12.08.2020).

6. ISO/IEC 7498-1:1994 Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Modelhttps://www.iso.org/standard/20269.html (дата обращения: 12.08.2020).

7. Internet society Home page. https://www.internetsociety.org (дата обращения: 12.08.2020).

8. Таненбаум Э., Уэзеролл Д. Компьютерные сети. 5-е изд. — СПб.: Питер, 2012. — 960 с.: ил. ISBN 978-5-459-00342-0.

9. Current Members of w3c. https://www.w3.org/Consortium/Member/List (дата обращения: 13.08.2020).

10. ГОСТР58539-2019(ИСО/МЭК 19763-1:2015) https://files.stroyinf.ru/Data/718/71813.pdf (дата обращения: 14.08.2020).

11. IANA. https://www.iana.org/ (дата обращения: 14.08.2020).

12. What is DNS? DNS Explained. https://ns1.com/resources/what-is-dns (дата обращения: 14.08.2020).

13. MIME types (IANA media types) https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types (дата обращения: 14.08.2020).

14. ICANN. https://www.icann.org/ru (дата обращения: 14.08.2020).

15. Open systems interconnection (OSI) https://www.iso.org/ics/35.100/x/ (дата обращения: 14.08.2020).

16. OSI IS-IS Intra-domain Routing Protocol https://tools.ietf.org/html/rfc1142 (дата обращения: 14.08.2020).

17. Олифер В Г, Олифер Н А. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. – СПб.: Питер, 2010. – 944 с.

18. X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) https://www.itu.int/rec/T-REC-X.690/ (дата обращения: 14.08.2020).

19. DoD Networking Model. https://www.freesoft.org/CIE/Course/Section1/5.htm (дата обращения: 14.08.2020).

20. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan https://tools.ietf.org/html/rfc4632 (дата обращения: 15.08.2020).

21. HTML5 Reference. The Syntax, Vocabulary and APIs of HTML5https://dev.w3.org/html5/html-author/ (дата обращения: 15.08.2020).

22. Progressive web apps (PWAs). https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps (дата обращения: 15.08.2020).

23. Hypertext Transfer Protocol -- HTTP/1.1. https://tools.ietf.org/html/rfc2616 (дата обращения: 15.08.2020).

24. HTTP Over TLS. https://tools.ietf.org/html/rfc2818 (дата обращения: 15.08.2020).

25. SQL-инъекция. Материал из Национальной библиотеки им. Н. Э. Баумана. https://ru.bmstu.wiki/SQL-инъекция (дата обращения: 21.08.2020).

26. Сергей Гордейчик. Cross-Site Request Forgery – много шума из-за ничего. https://www.securitylab.ru/analytics/292473.php (13 Марта, 2007) (дата обращения: 21.08.2020)

27. Kaspersky daily. Cookie: что нужно знать? https://www.kaspersky.ru/blog/cookie-chto-nuzhno-znat/979/ (7 июня 2013) (дата обращения: 21.08.2020)

28. Анатолий Ализар. Переполнение буфера в PHP. https://xakep.ru/2007/01/16/36240/ (16.01.2007)

29. Краткая история JavaScript. Часть 1. https://medium.com/webbdev/js-809d6101e64b (Jan 3, 2019)

30. Геопорталы: обзор. Геопортал ИВМ СО РАН. Краткая история веб-картографии. http://gis.krasn.ru/blog/review/history (дата обращения: 21.08.2020).

31. XMLHttpRequest Level 1. W3C Working Group Note 06 October 2016. https://www.w3.org/TR/XMLHttpRequest/ (дата обращения: 21.08.2020)

32. HTML 5. A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft 22 January 2008. https://www.w3.org/TR/2008/WD-html5-20080122/ (дата обращения: 21.08.2020)

33. Fielding Roy Thomas. Representational State Transfer (REST). CHAPTER 5. 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

34. Stephen Watts, Muhammad Raza. SaaS vs PaaS vs IaaS: What's The Difference & How To Choose. https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/ (дата обращения: 21.08.2020)

35. Python. https://www.python.org/ (дата обращения: 21.08.2020)

36. Flask's documentation. https://flask.palletsprojects.com/en/1.1.x/ (дата обращения: 21.08.2020)

37. What is NUMA? [Электронный ресурс]. URL: https://whatis.techtarget.com/definition/NUMA-non-uniform-memory-access (дата обращения: 14.04.2020)

38. Грид-вычисления [Электронный ресурс]. URL: https://ru.bmstu.wiki/Грид-вычисления (дата обращения: 14.04.2020)

39. Хабр. Пример простейшего cgi сервера [Электронный ресурс]. URL: https://habr.com/ru/post/254621/

40. Мэтт Зандстра. PHP: объекты, шаблоны и методики программирования — 3-е издание. — М.: «Вильямс», 2010. — С. 560.

41. Дино Эспозито. Microsoft ASP.NET 2.0. Базовый курс. — СПб: И. Д. Питер, 2007. — 688 с. — ISBN 978-5-91180-423-7.

42. Крупнейшая вики об WSGI. [Электронный ресурс]. http://wsgi.org/ (дата обращения: 28.03.2020)

43. Официальная документация Django [Электронный ресурс]. URL: https://www.djangoproject.com/ (дата обращения: 25.03.2020)

44. Официальная документация Flask [Электронный ресурс]. URL: https://flask.palletsprojects.com/en/1.1.x/ (дата обращения: 25.03.2020)

45. Официальная документация Pyramid [Электронный ресурс]. URL: https://trypyramid.com/documentation.html (дата обращения: 23.03.2020)

ГЛОССАРИЙ

TCP/IP – стек протоколов Интернета базирующийся на двух основных протоколах сетевого IP и транспортного TCP уровней, соответствующих модели сетевого взаимодействия DoD.

ВОС англ. OSI – Open System Interaction – Модель взаимодействия открытых систем, разбивающая взаимодействие на семь уровней.

Telecommunication – удаленное общение.

ITU - International Telecommunication Union, он же МСЭ

МСЭ - международный союз электросвязи

ООН – Организация объединенных наций

ИКТ - информационно-коммуникационные технологии

ISO - International Organization for Standardization

IEEE - Институт инженеров электротехники и электроники

DARPA - управление перспективных исследовательских проектов Министерства обороны США

ICCB - неофициальный комитет для наблюдения за сетью

IAB - Internet Activities Board

ARPANET

NSF - Национальный научный фонд США, реализовавший по технологии ARPANET сеть NSFNET

RFC - Requests for Comments, основные документы описывающие протоколы интернета

IRTF - Internet Research Task Force

IETF - Internet Engineering Task Force

ISOC - Internet Society

World Wide Web - распределённая система, предоставляющая доступ к связанным между собой документам, расположенным на различных компьютерах, подключённых к сети Интернет. Для обозначения Всемирной паутины также используют слово веб (англ. web «паутина») и аббревиатуру WWW.

IPv6 – протокол IP шестой версии, вводящий 16 байтные адреса, без контрольной суммы, и необходимости в протоколах ARP, RARP

DNS – Domain name system, протокол и система обеспечивающие трансляцию доменных имен в IP адреса и обратно

DNSSEC – защищенный DNS

Online Trust Alliance (OTA) – компания входящая в ISOC и продвигающая передовые методы обеспечения безопасности и конфиденциальности, укрепляющие доверие к сети Интернет.

W3C - Консорциум Всемирной паутины (англ. World Wide Web Consortium, W3C) — организация, разрабатывающая и внедряющая технологические стандарты для Всемирной паутины.

HTML - (от англ. HyperText Markup Language — «язык гипертекстовой разметки») — стандартизированный язык разметки веб-страниц во Всемирной паутине.

HTML5 – (англ. HyperText Markup Language, version 5) — язык для структурирования и представления содержимого всемирной паутины. Это пятая версия HTML вводящая, например, дополнительные элементы <canvas>, <video>, <audio> и т.д.

CSS – (Cascading Style Sheets «каскадные таблицы стилей») — формальный язык описания внешнего вида документа (веб-страницы), написанного с использованием языка разметки (чаще всего HTML или XHTML).

SVG - (от англ. Scalable Vector Graphics — масштабируемая векторная графика) — язык разметки масштабируемой векторной графики, созданный Консорциумом Всемирной паутины (W3C) и входящий в подмножество расширяемого языка разметки XML, предназначен для описания двумерной векторной и смешанной векторно/растровой графики в формате XML.

Ajax – (Asynchronous Javascript and XML — «асинхронный JavaScript и XML») — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером.

SMIL - (The Synchronized Multimedia Integration Language) — язык разметки для создания интерактивных мультимедийных презентаций. Является рекомендацией консорциума Всемирной паутины. SMIL похож на HTML, он выполнен на основе XML.

HTTP - (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

URI - Uniform Resource Identifier) — унифицированный (единообразный) идентификатор ресурса. URI — последовательность символов, идентифицирующая абстрактный или физический ресурс. Ранее назывался Universal Resource Identifier — универсальный идентификатор ресурса. Является либо URL, либо URN, либо одновременно обоими.

RDF - (RDF, «среда описания ресурса») — это разработанная консорциумом Всемирной паутины модель для представления данных, в особенности — метаданных. RDF представляет утверждения о ресурсах в виде, пригодном для машинной обработки. RDF является частью концепции семантической паутины.

SPARQL – (SPARQL Protocol and RDF Query Language) — язык запросов к данным, представленным по модели RDF, а также протокол для передачи этих запросов и ответов на них. SPARQL является рекомендацией консорциума W3C и одной из технологий семантической паутины. Предоставление SPARQL-точек доступа (англ. SPARQL-endpoint) является рекомендованной практикой при публикации данных во всемирной паутине.

OWL - (англ. Web Ontology Language) — язык описания онтологий для семантической паутины. Язык OWL позволяет описывать классы и отношения между ними, присущие веб-документам и приложениям.

SKOS - Simple Knowledge Organization System («простая система организации знаний») — это разработанная консорциумом W3C модель организации знаний для семантической паутины, призванная облегчить взаимодей-

ствие различных информационных систем за счёт стандартизации тезаурусов, систем классификации, таксономий, фолксономий и других видов нормализации лексики.

XML - eXtensible Markup Language) — расширяемый язык разметки. Рекомендован Консорциумом Всемирной паутины (W3C). Спецификация XML описывает XML-документы и частично описывает поведение XML-процессоров (программ, читающих XML-документы и обеспечивающих доступ к их содержимому). Все данные должны быть заключены в теги, значения атрибутов обязательно в кавычках, обязателен закрывающий тег.

XSLT - (eXtensible Stylesheet Language Transformations) — язык преобразования XML-документов. Спецификация XSLT входит в состав XSL и является рекомендацией W3C. При применении таблицы стилей XSLT, состоящей из набора шаблонов, к XML-документу (исходное дерево) образуется конечное дерево, которое может быть сериализовано в виде XML-документа, XHTML-документа (только для XSLT 2.0), HTML-документа или простого текстового файла.

DOM - (от англ. Document Object Model — «объектная модель документа») — это независящий от платформы и языка программный интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML-, XHTML- и XML-документов, а также изменять содержимое, структуру и оформление таких документов. Считывает весь документ в память.

SAX - (англ. «Simple API for XML») способ последовательного чтения/записи XML-файлов. Обычно SAX-парсеры требуют фиксированного количества памяти для своей работы, но не позволяют изменять содержимое документа. Используется обычно событийный механизм обработки тегов.

XML Query - язык запросов и функциональный язык программирования, разработанный для обработки данных в формате XML, простого текста, JSON или других предметно-специфичных форматах. XQuery использует XML как свою модель данных. Предназначен для запроса и преобразования коллекций структурированных и неструктурированных данных.

XPath - (XML Path Language) — язык запросов к элементам XML-документа. Разработан для организации доступа к частям документа XML в файлах трансформации XSLT и является стандартом консорциума W3C. XPath призван реализовать навигацию по DOM в XML. В XPath используется компактный синтаксис, отличный от принятого в XML.

SOAP - (от англ. Simple Object Access Protocol — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде. Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур (RPC). Сейчас протокол используется для обмена произвольными сообщениями в формате XML, а не только для вызова процедур.

IEC - Международная электротехническая комиссия (МЭК; англ. International Electrotechnical Commission, IEC; фр. Commission électrotechnique internationale, CEI) — международная некоммерческая организация по стандартизации в области электрических, электронных и смежных технологий. Некоторые из стандартов МЭК разрабатываются совместно с Международной организацией по стандартизации (ISO).

ГОСТ - Межгосударственный стандарт (индекс категории «ГОСТ») — региональный стандарт, принятый Межгосударственным советом по стандартизации, метрологии и сертификации Содружества Независимых Государств. На территории Евразийского экономического союза межгосударственные стандарты применяются добровольно.

IANA - (от англ. Internet Assigned Numbers Authority — «Администрация адресного пространства Интернет») — функция управления пространствами IP-адресов, доменов верхнего уровня, а также регистрирующая типы данных MIME и параметры прочих протоколов Интернета. Исполняется компанией Public Technical Identifiers, которая находится под контролем ICANN.

MIME – (англ. Multipurpose Internet Mail Extensions — многоцелевые расширения интернет-почты) — стандарт, описывающий передачу различ-

ных типов данных по электронной почте, а также, в общем случае, спецификация для кодирования информации и форматирования сообщений таким образом, чтобы их можно было пересылать по Интернету. Позволяет описать содержимое content-type, вкладывать разнородные данные путем использования multipart формата и разграничивающих строк boundary.

ICANN - (Internet Corporation for Assigned Names and Numbers) — международная некоммерческая организация, созданная 18 сентября 1998 года при участии правительства США для регулирования вопросов, связанных с доменными именами, IP-адресами и прочими аспектами функционирования Интернета. С 1 октября 2016 года — независимая международная организация.

RIR - Региональный интернет-регистратор (англ. Regional Internet Registry) — организация, занимающаяся вопросами адресации и маршрутизации в сети Интернет. Региональные регистраторы занимаются технической стороной функционирования Интернета: выделением IP-адресов, номеров автономных систем, регистрацией обратных зон DNS и другими техническими проектами. Статус RIR присваивается ICANN.

CIDR - (англ. Classless Inter-Domain Routing, англ. CIDR) — метод IP-адресации, позволяющий гибко управлять пространством IP-адресов, не используя жёсткие рамки классовой адресации. Использование этого метода позволяет экономно использовать ограниченный ресурс IP-адресов, поскольку возможно применение различных масок подсетей к различным подсетям.

FTP - (англ. File Transfer Protocol) — протокол передачи файлов по сети, является одним из старейших прикладных протоколов, появившихся задолго до HTTP, и даже до TCP/IP, в 1971 году, и сейчас широко используется для распространения ПО и доступа к удалённым хостам.

p2p – одноранговая, децентрализовання или пиринговая (англ. peer-to-peer, P2P — равный к равному) компьютерная сеть, основанная на равноправии участников. Часто в такой сети отсутствуют выделенные серверы, а каждый узел (peer) является как клиентом, так и выполняет функции сервера. В

отличие от архитектуры клиент-сервера, такая организация позволяет сохранять работоспособность сети при любом количестве и любом сочетании доступных узлов. Участниками сети являются все пиры.

SMTP - (англ. Simple Mail Transfer Protocol — простой протокол передачи почты) — это широко используемый сетевой протокол, предназначенный для передачи электронной почты в сетях TCP/IP. SMTP впервые был описан в RFC 821 (1982 год); последнее обновление в RFC 5321 (2008) включает масштабируемое расширение — ESMTP (англ. Extended SMTP).

POP3 - (Post Office Protocol 3) — протокол получения электронной почты. При помощи POP3 протокола письма скачиваются в устройство (например, компьютер или телефон) и затем удаляются с сервера. Главным недостатком и является обстоятельство, что по умолчанию письма на сервере не сохраняются.

IMAP - (англ. Internet Message Access Protocol) — протокол прикладного уровня для доступа к электронной почте. Базируется на транспортном протоколе TCP и использует порт 143, а IMAPS (IMAP поверх SSL) — порт 993. IMAP работает только с сообщениями и не требует каких-либо пакетов со специальными заголовками. IMAP предоставляет пользователю широкие возможности для работы с почтовыми ящиками, находящимися на почтовом сервере.

QoS (Quality of Service) - (англ. quality of service «качество обслуживания») — технология предоставления различным классам трафика различных приоритетов в обслуживании, также этим термином в области компьютерных сетей называют вероятность того, что сеть связи соответствует заданному соглашению о трафике, или же, в ряде случаев, неформальное обозначение вероятности прохождения пакета между двумя точками сети. Реализуется управлением очередями на коммутаторах и маршрутизаторах.

X.400 - протокол, представляет собой набор рекомендаций по построению системы передачи электронных сообщений, не зависящей от используемых на сервере и клиенте операционных систем и аппаратных средств. Реко-

мендации X.400 являются результатом деятельности международного комитета по телефонии и телеграфии (CCITT во французской транскрипции или ITU в английской), созданного при Организации Объединённых Наций.

SaaS - software as a service — программное обеспечение как услуга; также англ. software on demand — программное обеспечение по требованию) — одна из форм облачных вычислений, модель обслуживания, при которой подписчикам предоставляется готовое прикладное программное обеспечение, полностью обслуживаемое провайдером. Поставщик в этой модели самостоятельно управляет приложением, предоставляя заказчикам доступ к функциям с клиентских устройств, как правило через мобильное приложение или веб-браузер.

PaaS - Platform as a Service (PaaS, «платформа как услуга») — модель предоставления облачных вычислений, при которой потребитель получает доступ к использованию информационно-технологических платформ: операционных систем, систем управления базами данных, связующему программному обеспечению, средствам разработки и тестирования, размещённым у облачного провайдера. В этой модели вся информационно-технологическая инфраструктура, включая вычислительные сети, серверы, системы хранения, целиком управляется провайдером, провайдером же определяется набор доступных для потребителей видов платформ и набор управляемых параметров платформ, а потребителю предоставляется возможность использовать платформы, создавать их виртуальные экземпляры, устанавливать, разрабатывать, тестировать, эксплуатировать на них прикладное программное обеспечение, при этом динамически изменяя количество потребляемых вычислительных ресурсов.

IaaS - (англ. Infrastructure-as-a-Service; IaaS) — одна из моделей обслуживания в облачных вычислениях, по которой потребителям предоставляются по подписке фундаментальные информационно-технологические ресурсы — виртуальные серверы с заданной вычислительной мощностью, операционной системой (чаще всего — предустановленной провайдером из шаблона)

и доступом к сети. Популярным программным решением для создания IaaS является OpenStack. Здесь можно устанавливать свои гостевые ОС.

UDP - (англ. User Datagram Protocol — протокол пользовательских датаграмм) — один из ключевых элементов набора сетевых протоколов для Интернета. С UDP компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по IP-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных. Сообщение может как дойти до получателя, так и быть потеряно.

SCTP - (англ. Stream Control Transmission Protocol — «протокол передачи с управлением потоком») — протокол транспортного уровня в компьютерных сетях, появившийся в 2000 году в IETF. RFC 4960 описывает этот протокол, а RFC 3286 содержит техническое вступление к нему. Протокол обеспечивает мультихоуминг, возможность передавать данные через несколько сетевых интерфейсов, многопоточность, и использует четверное рукопожатие с учетом подтверждения, что избавляет от атак вроде TCP SYN флудинга.

MAC - (от англ. Media Access Control — надзор за доступом к среде, также Hardware Address, также физический адрес) — уникальный идентификатор, присваиваемый каждой единице активного оборудования или некоторым их интерфейсам в компьютерных сетях Ethernet. Старшие 24 бита выдаются организации выпускающей оборудование, 0-й бит указывает на то групповой это адрес или нет.

IMEI - (англ. International Mobile Equipment Identity — международный идентификатор мобильного оборудования) — это номер, обычно уникальный, для идентификации телефонов GSM, WCDMA и IDEN, а также некоторых спутниковых телефонов. Имеет 15 цифр в десятичном представлении, где последняя цифра контрольная по алгоритму Луна (Luhn algorithm), либо 17, где ещё две цифры - это версия.

DoD – модель взаимодействия открытых систем сети Интернета.

ARP - (англ. Address Resolution Protocol — протокол определения адреса) — протокол в компьютерных сетях, предназначенный для определения MAC-адреса по IP-адресу другого компьютера.

RARP - (англ. Reverse Address Resolution Protocol — Обратный протокол преобразования адресов) — протокол сетевого уровня модели OSI, выполняет обратное отображение адресов, то есть преобразует физический адрес в IP-адрес.

IPv4 – (англ. Internet Protocol version 4) — четвёртая версия интернет протокола (IP). Первая широко используемая версия. Протокол описан в RFC 791 (сентябрь 1981 года), заменившем RFC 760 (январь 1980 года). Использует 4 байтовые адреса, контрольную сумму по заголовку, промежуточную фрагментацию, максимальный размер пакета 64 кбайта, протоколы ARP и RARP для соответствия физическим адресам, время жизни пакета.

SYN – бит поля флагов протокола TCP для установки синхронизирующей последовательности при установлении соединения.

ACK – бит поля флагов протокола TCP указывающий на то, что это подтверждающий прием данных сегмент.

DDOS – (distributed denial-of-service attack) распределенная атака на отказ в обслуживании. Обычно реализуется с использованием Bot-net.

URL - Унифицированный указатель ресурса (от англ. Uniform Resource Locator) — система унифицированных адресов электронных ресурсов, или единообразный определитель местонахождения ресурса (файла). Используется как стандарт записи ссылок на объекты в Интернете (Гипертекстовые ссылки во «всемирной паутине» www). Для обозначения электронного адреса используют аббревиатуру «URL» по ГОСТ Р 7.0.5-2008.

PWA - (англ. progressive web app, PWA) — технология в web-разработке, которая визуально и функционально трансформирует сайт в приложение (мобильное приложение в браузере).

ASP .NET - (Active Server Pages для .NET) — платформа разработки веб-приложений, в состав которой входит: веб-сервисы, программная инфра-

структура, модель программирования, от компании Майкрософт. ASP.NET входит в состав платформы .NET Framework и является развитием более старой технологии Microsoft ASP.

CGI - (от англ. Common Gateway Interface — «общий интерфейс шлюза») —технология используемая для связи внешней программы с веб-сервером через стандартный поток ввода вывода. Программу, которая работает по такому интерфейсу совместно с веб-сервером, принято называть шлюзом, хотя многие предпочитают названия «скрипт» (сценарий) или «CGI-программа».

JSP - JSP (JavaServer Pages) — технология, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница JSP содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP-элементы, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также Expression Language (EL), для внедрения Java-кода в статичное содержимое JSP-страниц.

Node.js - программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API, написанный на C++, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль веб-сервера, но есть возможность разрабатывать на Node.js и десктопные оконные приложения.

PHP - PHP: Hypertext Preprocessor — «PHP: препроцессор гипертекста»; первоначально PHP/FI (Personal Home Page / Form Interpreter), а позже названный Personal Home Page Tools — «Инструменты для создания персональных веб-страниц») — скриптовый язык общего назначения с динамиче-

ской типизацией (интерпретатор), интенсивно применяемый для разработки веб-приложений. В настоящее время поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов.

SQL – (structured query language — «язык структурированных запросов») — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных, управляемой соответствующей системой управления базами данных.

HTTPS - (аббр. от англ. HyperText Transfer Protocol Secure) — расширение протокола HTTP для поддержки шифрования в целях повышения безопасности. Данные в протоколе HTTPS передаются поверх криптографических протоколов TLS или устаревшего в 2015 году SSL. В отличие от HTTP с TCP-портом 80, для HTTPS по умолчанию используется TCP-порт 443.

GET – запрос HTTP для получения ресурса.

POST – запрос HTTP для создания ресурса.

PUT – запрос HTTP для обновления ресурса.

PATCH – запрос HTTP для частичного обновления ресурса.

DELETE – запросе HTTP для удаления ресурса.

MX – запись DNS о почтовых доменных именах.

WSGI - (англ. Web Server Gateway Interface) — стандарт взаимодействия между Python-программой, выполняющейся на стороне сервера, и самим веб-сервером, например Apache.

GIL - является самым простым способом избежать конфликтов при одновременном обращении разных потоков к одним и тем же участкам памяти. Когда один поток захватывает его, GIL, работая по принципу мьютекса, блокирует остальные. Нет параллельных потоков — нет конфликтов при обращении к разделяемым объектам. Очерёдность выполнения потоков определяет интерпретатор в зависимости от реализации, переключение между потоками может происходить: когда активный поток пытается осуществить ввод-вывод, по исчерпании лимита выполненных инструкций, либо по таймеру.

Главный недостаток подхода обеспечения потокобезопасности при помощи GIL — это ограничение параллельности вычислений. GIL не позволяет достигать наибольшей эффективности вычислений при работе на многоядерных и мультипроцессорных системах.

SSI - (Server Side Includes — включения на стороне сервера) — несложный язык для динамической «сборки» веб-страниц на сервере из отдельных составных частей и выдачи клиенту полученного HTML-документа. Реализован в веб-сервере Apache при помощи модуля mod_include. Включённая в настройках по умолчанию веб-сервера возможность позволяет подключать HTML-файлы, поэтому для использования инструкций файл должен оканчиваться расширением .shtml, .stm или .shtm.

CDN (Content Delivery Network) - Сеть доставки (и дистрибуции) содержимого (англ. Content Delivery Network или Content Distribution Network, CDN) — географически распределённая сетевая инфраструктура, позволяющая оптимизировать доставку и дистрибуцию содержимого конечным пользователям в сети Интернет. Использование контент-провайдерами CDN способствует увеличению скорости загрузки интернет-пользователями аудио-, видео-, программного, игрового и других видов цифрового содержимого в точках присутствия сети CDN.

OAuth - открытый протокол (схема) авторизации, который позволяет предоставить третьей стороне ограниченный доступ к защищённым ресурсам пользователя без необходимости передавать ей (третьей стороне) логин и пароль. Работа над протоколом началась в ноябре 2006 года, а последняя версия OAuth 1.0 была утверждена 4 декабря 2007 года. Как последующее развитие в 2010 году появился протокол OAuth 2.0, последняя версия которого в качестве в RFC 6749 опубликована в октябре 2012 года.

CSRF - (англ. cross-site request forgery — «межсайтовая подделка запроса», также известна 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, которое не может быть проигнорировано или подделано атакующим скриптом.

JSON - (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом. Лаконичнее XML.

REST - (от англ. Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы.

Swagger - (с англ. — «спецификация OpenAPI»; изначально известная как Swagger Specification) — формализованная спецификация и экосистема множества инструментов, предоставляющая интерфейс между front-end системами, кодом библиотек низкого уровня и коммерческими решениями в виде API. Вместе с тем, спецификация построена таким образом, что не зависит от языков программирования, и удобна в использовании как человеком, так и машиной.

RAML - (RESTful API Modeling Language) используется для проектирования и документирования веб-сервисов REST.

ЦОД (от англ. data center), или центр (хранения и) обработки данных (ЦОД/ЦХОД) — это специализированное здание для размещения (хостинга) серверного и сетевого оборудования и подключения абонентов к каналам сети Интернет.

Colocation Колокейшн-центр (также обозначается как co-location или colo) - это тип центра обработки данных (ЦОД), где оборудование, помеще-

ния и пропускная способность могут арендоваться клиентами, в том числе для размещения своего оборудования.

Кластер - группа компьютеров, объединённых высокоскоростными каналами связи, представляющая с точки зрения пользователя единый аппаратный ресурс.

RAID - (англ. Redundant Array of Independent Disks — избыточный массив независимых (самостоятельных) дисков) — технология виртуализации данных для объединения нескольких физических дисковых устройств в логический модуль для повышения отказоустойчивости и производительности.

SMP - (англ. Symmetric Multiprocessing, сокращённо SMP) — архитектура многопроцессорных компьютеров, в которой два или более одинаковых процессора сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам) и выполняют одни и те же функции (почему, собственно, система и называется симметричной).

NUMA - (англ. Non-Uniform Memory Access «неравномерный доступ к памяти» или Non-Uniform Memory Architecture «архитектура с неравномерной памятью») — схема реализации компьютерной памяти, используемая в мультипроцессорных системах, когда время доступа к памяти определяется её расположением по отношению к процессору.

CC-NUMA (cache coherent NUMA) - система с кэш-когерентным доступом к неоднородной памяти.

GRID-вычисления) — это форма распределённых вычислений, в которой «виртуальный суперкомпьютер» представлен в виде кластеров, соединённых с помощью сети, слабосвязанных гетерогенных компьютеров, работающих вместе для выполнения огромного количества заданий (операций, работ). Живая миграция (от англ. Live migration) — перенос виртуальной машины с одного физического сервера на другой без прекращения работы виртуальной машины и остановки сервисов; применяется в компьютерных системах с высокой доступностью (англ. High Availability, HA). Живая миграция возможна между серверами, находящимися в кластере.

SAN (сеть хранения данных) - (англ. Storage Area Network, SAN) — представляет собой архитектурное решение для подключения внешних устройств хранения данных, таких как дисковые массивы, ленточные библиотеки, оптические приводы к серверам таким образом, чтобы операционная система распознала подключённые ресурсы как локальные.

NAS (сетевое хранилище) - NAS (англ. Network Attached Storage) — является сервером для хранения данных на файловом уровне. По сути, представляет собой компьютер с некоторым дисковым массивом, подключённый к сети (обычно локальной) и поддерживающий работу по принятым в ней протоколам. Несколько таких компьютеров могут быть объединены в одну систему.

SCSI - (англ. Small Computer System Interface) — представляет собой набор стандартов для физического подключения и передачи данных между компьютерами и периферийными устройствами. SCSI-стандарты определяют команды, протоколы и электрические и оптические интерфейсы. Разработан для объединения на одной шине различных по своему назначению устройств, таких, как жёсткие диски, накопители на магнитооптических дисках, приводы CD, DVD, стримеры, сканеры, принтеры и т. д.

Паравиртуализация — процесс, при котором гостевые ОС модифицируют свое ядро с целью функционирования в виртуализированной среде.

Встроенная виртуализация — новый метод, базирующийся на применении аппаратно-поддерживаемых возможностей виртуализации, что позволяет пользователям использовать любые версии ОС в сочетании с различными вариантами рабочих сред.

Гипервизор - программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполнение нескольких операционных систем на одном и том же хост-компьютере.

RAM – random access memory, память произвольного доступа

Docker программное обеспечение прендназначено для автоматизации развёртывания и управления приложениями в средах с поддержкой контей-

неризации. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему

Socket (гнездо, разъем) - абстрактное программное понятие, используемое для обозначения в прикладной программе конечной точки канала связи с коммуникационной средой, образованной вычислительной сетью.

Select – старая программная технология опроса состояния файловых дескрипторов, не более 1024, описывается битовой маской, передаваемые данные дескриптора изменяются вызовом select и надо их восстанавливать, каждый дескриптор необходимо проверять, сложность алгоритма O(n)

poll – программная технология опроса состояния файловых дескрипторов, передаваемые данные дескрипторов не изменяются, каждый дескриптор необходимо проверять, сложность алгоритма O(n)

epoll - программная технология опроса состояния файловых дескрипторов, возвращаются только измененные дескрипторы, сложность алгоритма O(1)

ASCII - (англ. American standard code for information interchange) — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Часто используется в интернет протоколах.

UTF-8 - (от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII.

Cookie - — небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса.

CSR (Certificate Signing Request - запрос на получение сертификата) - — это запрос на получение сертификата, который представляет собой текстовый файл, содержащий в закодированном виде информацию об администраторе домена и открытый ключ.

SSL - (англ. Secure Sockets Layer — уровень защищённых сокетов) — криптографический протокол, который подразумевает более безопасную связь. Он использует асимметричную криптографию для аутентификации ключей обмена, симметричное шифрование для сохранения конфиденциальности, коды аутентификации сообщений для целостности сообщений.

TLS - (англ. transport layer security — Протокол защиты транспортного уровня), как и его предшественник SSL, криптографический протокол, обеспечивающий защищённую передачу данных между узлами в сети Интернет. TLS-протокол основан на спецификации протокола SSL версии 3.0, разработанной компанией Netscape Communications