# **33. Виртуализация десктопов. Преимущества и недостатки.**

Виртуализация отделяет ПО от оборудования – это означает, что ПО помещается в отдельный контейнер, чтобы изолировать его от ОС.

1. **Виртуализация рабочего стола** – отделение рабочего стола с его ОС, приложениями и пользовательскими данными от базовой вычислительной системы. Виртуализация рабочего стола предоставляет пользователям операционную среду, независимую от их локальных физических систем. Он состоит из серверов, ПО виртуализации на серверах и виртуального образа на рабочих столах.
2. **Виртуализация представлений** – эмуляция интерфейса пользователя. Т.е. пользователю передается лишь картинка удаленного приложения на свой терминал, на самом деле приложение выполняется на удалённом сервере.

**Преимущества виртуализации десктопов:**

* доступ пользователя из любого места через LAN, WAN или широкополосный доступ.
* регулярное обновление приложений можно
* повышенная и быстрая безопасность благодаря централизованному администрированию
* упрощает резервное копирование пользовательских данных

=> эффективность использования ресурсов, масштабируемость, гибкость и простое резервное копирование и миграция

**Недостатки виртуализации десктопов:**

* фиксированное количество клиентских машин в сети
* иногда клиенты дорогие
* хорошая пропускная способность должна быть, чтобы не было перегрузки в локальной сети
* необходимо ограничивать количество ОС

=> проблема в распределении ресурсов при высокой загрузке и сложная настройка

# **34.Виртуализация приложений. Достоинства и недостатки.**

**Виртуализация приложений** – применение модели сильной изоляции прикладных программ с управляемым взаимодействием с ОС, при которой виртуализируется каждый экземпляр приложений. Данная технология позволяет использовать на одном компьютере, а точнее в одной и той же ОС несколько несовместимых между собой приложений одновременно.

Основные компоненты: файлы (включая системные), реестр, шрифты, INI-файлы, COM-объекты, службы.

Виртуализация приложений шаги:

1) Упаковка приложения: приложение устанавливается в специальном упаковщике, который записывает все файлы, реестры и настройки, связанные с приложением.

2) Доставка приложения в целевую систему: упакованное приложение доставляется в целевую систему через Интернет, USB или настраиваемый механизм push.

3) Выполнение приложения в виртуальной среде: приложение выполняется в виртуальной среде, и эта среда полностью изолирована от других приложений и базовой ОС.

**Преимущества** виртуализации приложений:

* нет необходимости устанавливать приложение
* быстрое развертывание приложений
* простое и эффективное управление приложением
* снижение затрат
* повышенная безопасность

**Недостатки**:

* необходима постоянная поддержка полосы пропускания

# **35.Использование сервис-ориентированной архитектуры в облачных приложениях и службах.**

SOA – это архитектурный стиль, в котором бизнес-логику можно разделить на четко определенную службу многократного использования, которой может пользоваться каждый разработчик. Т.е. это совокупность сервисов, реализуемых в виде компонентных объектов, систематизированных на основе обмена сообщениями. Общий пример сообщений – XML сообщение. SОА позволяет решить:

* проблему изоляция бизнес-логики
* проблему обеспечения совместимости компонентов
* проблему избыточности

Для каждого сервиса принято определять:

* поставщика сервиса (компонент сервис)
* потребителя сервиса (компонент клиент)
* брокера – компонент, обеспечив вз-ие поставщика и потребителя

Приложение можно описать трехуровневой архитектурой:

* Уровень интерфейса пользователя
* Уровень бизнес логики
* Уровень данных

**Организация работы:**

1. создается новый промежуточный слой сервисов,
2. реализуется разделение функций на уровне бизнес-логики
3. разделение доступа к данным

**Сервис в SOA имеет четыре свойства:**

1) логически – это бизнес-деятельность с заданным результатом

2) самодостаточен

3) это черный ящик для потребителей

4) может состоять из других базовых сервисов

**Преимущества:**

1. быстрее и проще создавать бизнес-приложения
2. легкое обслуживание и обновление
3. гибкость и расширяемость
4. снижение совокупной стоимости владения

# **36. Микросервисная архитектура в облачных приложениях**

**Микросервисная архитектура** – частный случай SOA c набором более строгих правил. У микросервисов есть особые свойства, они же преимущества:

* Гетерогенность: возможность построить систему с помощью разных языков программирования и технологий;
* Децентрализованное управление данными: каждый микросервис содержит свой набор данных, доступный другим микросервисам только через соответствующий интерфейс;
* Независимость инфраструктуры: каждый микросервис - независимая единица, поэтому вносить изменения и разворачивать его можно независимо от других;
* Масштабируемость: чтобы увеличить производительность системы, нужно расширить только те сервисы, которые в этом нуждаются.

Микросервисная архитектура **выигрывает** у монолита:

1. изменил один из микросервисов и сразу его внедрил
2. не зависят от какой-либо платформы

**Недостатки**:

* Сложность начальной разработки и создания инфраструктуры, т.к. нужно предусмотреть независимость одного микросервиса от сбоя в другом компоненте
* Для распределенной системы сложно поддерживать строгую согласованность

# **37. Бессерверная технология реализации облачных приложений и служб.**

**Бессерверная архитектура** – это способ создания и запуска приложений и сервисов без необходимости управления инфраструктурой. Приложение по-прежнему работает на серверах, но управление этими серверами AWS полностью берет на себя. Это избавляет от необходимости заниматься выделением ресурсов, масштабированием и обслуживанием серверов для запуска приложений, баз данных и систем хранения данных.

Для реализации бессерверных архитектур можно использовать облачные сервисы, такие как:

* AWS Lambda (основан на микросервисах);
* Amazon API Gateway;
* Amazon DynamoDB (NoSQL-СУБД);
* реляционная СУБД Aurora.

Другие облака:

* Microsoft Azure Functions
* Google Functions
* IBM OpenWhisk

# **38. Общая архитектура облачных систем и проблемы архитектурного дизайна облачных систем.**

Интернет-облако представляется пользователю как массивный кластер серверов, размещенных в одном или нескольких ЦОДах. Эти сервера предоставляют ресурсы для выполнения веб-сервисов или распределенных приложений по запросам. Облачная платформа формируется динамически из серверов, ПО и ресурсов БД. Сервера в облаке могут быть как физическими, так и виртуальными машинами.

**Облачная платформа** – масштабируемая программно-аппаратная инфраструктура на основе вычислительного кластера.

Цели создания платформ облачных вычислений – предоставление ресурсов по запросам пользователей обеспечивая:

* масштабируемость,
* виртуализацию ресурсов,
* эффективность использования ресурсов,
* надежность предоставления ресурсов,
* безопасность предоставления разделяемых ресурсов
* разделяемый доступ к ЦОДам.

Архитектура облака разработана как 3хуровневая:

* Уровень инфраструктуры (IaaS)
* Уровень платформы (PaaS)
* Уровень приложений (SaaS)

**Проблемы** архитектурного дизайна облаков:

* Проблема доступности и блокировки данных
* Конфиденциальность данных и их защита
* Непредсказуемая производительность
* Распределенные хранилища данных и широко распространенные ошибки ПО.
* Масштабируемость и стандартизация
* Лицензирование ПО и разделение репутации.

# **39. Возможности облачных систем, используемые для создания собственных служб и приложений.**

Облачные системы, такие как AWS, MA, GCP предоставляют спектр инструментов и услуг, которые можно использовать для создания собственных служб и приложений.

Ниже приведены некоторые возможности, которые предоставляют эти облачные системы:

* Виртуальные машины: возможность создания VM, в к-х можно выбирать различные конфигурации: размер, ОС и др.
* Контейнеры: они позволяют упаковывать приложения и их зависимости в единую сущность, которая может быть запущена на любом сервере, где установлен контейнерный движок.
* Базы данных: можно выбирать различные типы баз данных, включая реляционные, NoSQL и др.
* Хранилища данных: Amazon S3, Azure Blob Storage и Google Cloud Storage позволяют хранить и управлять файлами и объектами в облаке.
* Инструменты для анализа данных: Amazon EMR, Azure HDInsight и Google Cloud Dataproc – позволяют проводить анализ больших объемов данных, используя специализированные инструменты и технологии.
* Управление контейнерами: ECR, ACR и GCR – позволяют хранить, управлять и использовать их для развертывания приложений.
* Интеграция и доставка: AWS CodePipeline, Azure DevOps и Google Cloud Build – позволяют автоматизировать процессы сборки, тестирования и развертывания приложений.

# **40. Какие приложения не следует располагать в системах облачных вычислений.**

Типы приложений, которые не следует располагать в облачной среде:

1) Приложения с высокими требованиями к безопасности: если приложение обрабатывает или содержит конфиденциальную информацию, размещение его в облаке может оказаться небезопасным.

2) Приложения с высокими требованиями к производительности и отклику: некоторые приложения(высоконагруженные бд или приложения реального времени) могут требовать высокой производительности и низкой задержки. Облачные вычисления могут не всегда обеспечить необходимый уровень из-за разделения ресурсов между различными клиентами.

3) Приложения с ограничениями по локации данных: некоторые приложения могут иметь ограничения по местоположению хранения данных из-за законодательных требований или политики организации. В облаке данные могут храниться на серверах, расположенных в разных странах, что может противоречить требованиям ограничения доступа к данным.

4) Приложения с высокими затратами на обработку данных: некоторые приложения могут иметь высокие затраты на обработку данных(большой объема данных/сложные вычисления). В облаке это может привести к дополнительным расходам из-за использования вычислительных ресурсов облачного провайдера.

# **41. Архитектурные принципы и свойства веб-сервисов REST**

**Принципы**:

* Системы должны быть построены на основе масштабируемой архитектуры для поддержания роста пользователей, трафика или размера данных без снижения производительности.
* Архитектура должна обеспечивать линейный рост масштаба.
* Рост объемов потребляемых ресурсов должен обеспечивать экономию на масштабе.
* Стоимость должна соответствовать степени увеличения производительности и ценности ее для бизнеса.

Технология REST практически вытеснила дизайн интерфейса, основанный на SOAP и WSDL, из-за более простого стиля проектирования, к-я используется всеми публичными облачными системами.

Существует шесть ограничений (5 обязательных и 1 опционное):

1. Модель клиент-сервер

2. Отсутствие состояния

3. Кэширование

4. Единообразие интерфейса

5. Слои

6. Код по требованию (необязательное ограничение)

Архитектура REST основана на четырех принципах:

1. Идентификация ресурса посредством URI.
2. Унифицированный интерфейс – GET извлекает текущее состояние ресурса в некотором представлении. POST устанавливает новое состояние ресурса.
3. Информативные сообщения – Ресурсы отделены таким образом, что их содержимое может быть доступно в различных форматах (HTML, XML, текст, RDF, JPEG).
4. Взаимодействие через гиперссылки – Для обмена существуют различные технологии (переименование URI, cookies и скрытые поля формы).

# **42. Методы обеспечения масштабируемости облачных приложений и служб в зависимости от их архитектуры.**

**Масштабируемость** – это способность повышать производительность системы наиболее рациональным способом. Масштабируемость позволяет приложениям и службам адаптироваться к изменяющимся нагрузкам и требованиям, обеспечивая высокую производительность и доступность.

**Общие методы:**

1. Использование контейнеров.

2. Мониторинг и управление ресурсами.

**Общие методы в зависимости от архитектуры:**

1. Масштабируемость вертикального масштабирования:

* Увеличение ресурсов: при вертикальном масштабировании можно увеличить вычислительную мощность или объем памяти виртуальной машины или контейнера.
* Использование масштабируемых баз данных – позволяет увеличивать производительность путем увеличения ресурсов одного сервера.

1. Масштабируемость горизонтального масштабирования:

* Распределение нагрузки: используя балансировку нагрузки позволяет увеличить общую производительность за счет параллельной обработки.
* Автоматическое масштабирование: оптимизация использования ресурсов и обеспечение высокой доступности за счет динамического изменения количество экземпляров приложения или службы.
* Микросервисная архитектура: разработка приложения в виде набора микросервисов позволяет масштабировать каждую часть независимо друг от друга – облегчает горизонтальное масштабирование и управление отдельными компонентами приложения.

# **43. Обеспечение обмена данными между компонентами облачных систем с помощью ESB (Enterprise Service Bus).**

* ESB – выполняет функции посредничества при:
  + Обмене сообщениями;
  + Регистрации;
  + Маршрутизации;
  + Ведении журналов;
  + Аудите;
  + Управлении транзакционной целостностью.
* Современные приложения редко работают изолированно; приложение не может сделать что-либо значимое без взаимодействия с другими приложениями.
* Сервис-ориентированная архитектура интегрирует приложения для совместной работы и ускоряет их работу, разбивая приложение на части, которые могут быть объединены друг с другом.
* Шина обеспечивает возможность подключения конечных точек друг к другу без прямого контакта друг с другом. Это упрощает связь для конечных точек, поскольку они должны соответствовать стандартным коммуникационным интерфейсам - шине. (Это с любой технической шиной, а не только с ESB).
* ESB может расширять существующую инфраструктуру обмена сообщениями, предоставляя инструментарий для создания и развертывания инфраструктурного уровня систем, построенных на основе обмена сообщениями.
* Примеры таких систем могут включать сервисы маршрутизации и преобразования, сервисы журналирования и т.д.
* Это может быть простым расширением нижележащей инфраструктуры, построенной на обмене сообщениями, либо мостом между разными продуктами и даже технологиями. Для доступа к приложениям или сервисам, не поддерживающим обмен сообщениями, могут использоваться адаптеры.

# **44. Общая архитектура SOA облачного приложения но основе ESB.**



**Сервисная шина предприятия** — [связующее программное обеспечение](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), обеспечивающее обмен сообщениями между различными [информационными системами](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) на принципах [сервис-ориентированной архитектуры](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0).

При построении облачного приложения на основе SOA с использованием Enterprise Service Bus (ESB) можно применить следующую общую архитектуру:

**Компоненты архитектуры SOA на основе ESB:**

1) Сервисы: сервисы могут предоставлять различные возможности, такие как обработка заказов, аутентификация пользователей и т.д.

2) Enterprise Service Bus (ESB) – центральный компонент архитектуры, обеспечивающий интеграцию и коммуникацию между сервисами. ESB выполняет функции маршрутизации сообщений, преобразования данных, обеспечения безопасности и мониторинга.

3) Реестр сервисов: содержит информацию о доступных сервисах, их интерфейсах и местоположении.

4) Бизнес-процессы: определяют логику взаимодействия между сервисами для выполнения конкретной бизнес-задачи.

**Процесс взаимодействия в архитектуре SOA на основе ESB:**

1. Запрос сервиса: Клиентские приложения или другие сервисы могут отправлять запросы на выполнение определенных операций сервисов через ESB.

2. Маршрутизация сообщений: ESB маршрутизирует запросы к соответствующим сервисам на основе определенных правил и конфигурации.

3. Обработка запросов: Сервисы обрабатывают поступившие запросы, выполняют необходимые действия и возвращают результат обратно через ESB.

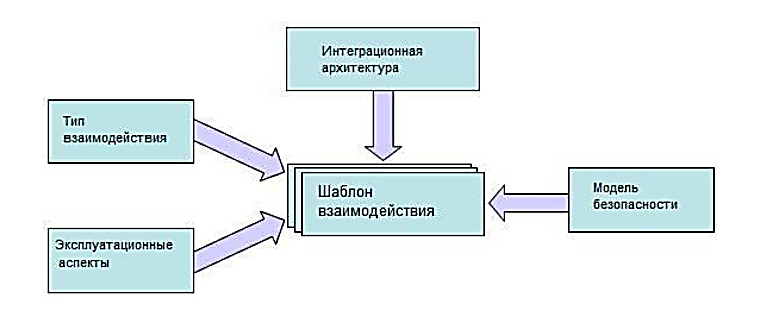
4. Обеспечение безопасности и мониторинг: ESB может обеспечивать безопасность коммуникации между сервисами, а также собирать данные о производительности и доступности сервисов для мониторинга.

# **45. Шаблоны взаимодействия облачных приложений с помощью службы ESB.**

Дизайн ESB реализует множество признанных шаблонов проектирования и спецификаций стандартов:

* + **Простой сервис-фасад**
  + **Сложный сервис-фасад**
  + **Клиентский доступ**

Ценность корпоративной сервисной шины (ESB) в интеграции систем автоматизации предприятия трудно переоценить, но чрезвычайная гибкость этого подхода приводит к огромным количеством вариантов реализации. Варианты реализации могут быть скомпонованы в шаблоны в зависимости от возможностей связующего ПО, поддерживающего эти архитектурные шаблоны.



# **46. Контейнеры. Возникновение контейнеров. Контейнерная виртуализация в Linux.**

Альтернативным подходом к изоляции приложений являются контейнеры. Лидером в продвижении контейнерных технологий является компания Parallels, представив продукт Virtuozzo.

Virtuozzo был создан изначально для Linux, в 2005 г. появилась версия для Windows , примерно тогда же компания Sun реализовала Solaris Containers для своей ОС (и появился термин «контейнеры»).

Название Virtuozzo порой бывает синонимом всего этого направления виртуализации, поскольку остальные средства намного уступают ему в популярности.

Экземпляры пространств пользователя (часто называемые контейнерами или зонами) с точки зрения пользователя полностью идентичны реальному серверу, но они в своей работе используют один экземпляр ядра операционной системы.

**Наиболее распространены сейчас OpenVZ, LXC, FreeBSD jail и Solaris Containers**

Для linux-систем, эта технология может рассматриваться как улучшенная реализация механизма chroot.

**Основные компоненты контейнерной виртуализации в Linux:**

1. **Ядро Linux**: Контейнеры в Linux основаны на функциях ядра, таких как cgroups (control groups) и namespace, которые обеспечивают изоляцию ресурсов и пространства имен для контейнеров.
2. **Docker / Podman**: Docker и Podman - это инструменты, к-е предоставляют удобный интерфейс для создания, запуска, остановки и удаления контейнеров.
3. **Образы контейнеров**: Образы контейнеров содержат файловую систему с необходимыми зависимостями и настройками для запуска приложения в контейнере.
4. **Dockerfile**: Dockerfile - это текстовый файл, который содержит инструкции для создания образа контейнера. Он определяет базовый образ, устанавливает зависимости, копирует файлы и настраивает окружение для приложения.

# **47. Существующие варианты контейеризации.**

1. **OpenVZ** позволяет на одном физическом сервере запускать множество изолированных копий операционной системы, называемых «виртуальные частные серверы» (Virtual Private Servers, VPS) или «виртуальные среды» (Virtual Environments, VE).

Поскольку OpenVZ базируется на ядре Linux, в отличие от виртуальных машин (напр. VMware, Parallels Desktop) или паравиртуализационных технологий (напр. Xen), в роли «гостевых» систем могут выступать только дистрибутивы Linux.

OpenVZ является базовой платформой для Virtuozzo — проприетарного продукта Parallels

1. **LXC** (Linux Containers) — система виртуализации на уровне операционной системы для запуска нескольких изолированных экземпляров операционной системы Linux на одном узле.

LXC основана на технологиях namespaces и cgroups, входящей в ядро Linux, начиная с версии 2.6.29.

# **48. Механизмы изоляции ОС, используемые при контейнеризации.**

В 1979 году в UNIX был добавлен системный вызов chroot() — с целью обеспечить изоляцию и предоставить разработчикам отдельную от основной системы площадку для тестирования. chroot представляет собой сокращение от change root, - «изменить корень». С  помощью системного вызова chroot() и соответствующей команды можно изменить корневой каталог

Попытки усовершенствовать механизм chroot и обеспечить более надёжную изоляцию предпринимались неоднократно: так, в частности, появились такие известные технологии, как FreeBSD Jail и Solaris Zones (про него не было в презентации).

Механизм виртуализации в системе FreeBSD, позволяющий создавать внутри одной операционной системы FreeBSD несколько независимо работающих FreeBSD на том же ядре операционной системы, но совершенно независимо настраиваемых с независимым набором установленных приложений.

В ядре Linux изоляция процессов была усовершенствована благодаря добавлению новых подсистем и новых системных вызовов:

* Механизм пространств имен (на основе использования различных привилегий, назначаемых разным пользователям)
* Механизм групп контроля (cgroups) (ограничивают количество ресурсов, которые разрешено использовать каждому отдельному контейнеру)

Принцип работы cgroups заключается в следующем: определённые процессы помещаются в группу, которую затем «встраиваются» в подсистемы.

# **49. Docker – как средство быстрого развертывания контейнеров с ПО.**

Docker — это программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на базе контейнеров LXC.

Docker позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любой Linux-системе с поддержкой cgroups в ядре, а также предоставляет среду по управлению контейнерами.

Docker, широко используется во многих облачных платформах – AWS, Azure, Cocaine (разработанной компанией Яндекс) и др.

Docker:

* позволяет отделить приложение от его инфраструктуры.
* дает возможность запускать практически любое приложение, безопасно как изолированное в контейнере.

Docker дает возможность достаточно просто упаковать приложения вместе со всеми его зависимостями, а затем продолжить их разработку, тестирование и регулярное использование в совершенно другой среде.

Благодаря тому, что Docker позволяет упаковывать каждый компонент и его зависимости по-отдельности, решаются следующие задачи:

* + **Исчезают конфликтующие зависимости**: нужно запустить один веб-сайт на PHP 4.3 и еще один на PHP 5.5? Нет проблем в случае, если вы запускаете каждую версию PHP в отдельном контейнере Docker.
  + **Нет потерянных зависимостей**: установка приложения в новой среде осуществляется при помощи Docker и поэтому в контейнере вместе с приложением упакованы все зависимости.
  + **Нивелируется различие платформ**: переход от одного дистрибутива к другому уже не является проблемой. Если в обеих системах работает Docker, то один и тот же контейнер будет этих системах выполняться без всяких проблем.

# **50. Архитектура Docker. Процесс создания образа. Запуск контейнера.**

**Базовая Архитектура Docker**

* Client Docker (CLI): Консольный (CLI) клиент использует Docker REST API для управления или взаимодействия с демоном Docker при помощи скриптов или непосредственных консольных команд
* Server Docker (Docker Engine): Сервер, по сути являющийся долго работающим приложением, называемым демоном
* REST API: REST API, определяющее интерфейсы для взаимодействия, которые могут использовать другие программы.

**Элементы архитектуры**

* Docker Клиент (REST API)
* Docker Engine (демон)
* Docker File
* Образы (images)
* Реестры (registries)
* Docker Hub
* Контейнеры

Образ контейнера – это «исходный» код на основе которого строятся контейнеры.

Образ является «шаблоном», а контейнер копия с этого шаблона.

**Как создать Docker образ?**

1. запуск команды с указанием:

* добавление файла или директории
* создание переменной окружения
* указания что запускать, когда запускается контейнер этого образа
* инструкции хранятся в файле Dockerfile

Docker считывает этот Dockerfile, выполняет находящиеся там инструкции и возвращает конечный образ. Docker ориентирован на контейнеризацию приложений, а не ОС.

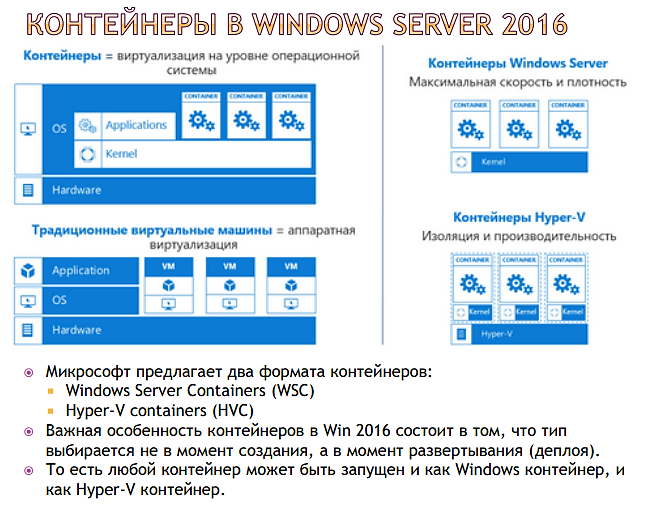
1. На основе образа диска с использованием в качестве описания:
   * Имеющегося контейнера (интерактивное создание);
   * Docker-файла (создание на основе docker-файла).

Запустить простой контейнер помощью команды:

$ sudo docker run –i -t ubuntu /bin/bash

# **51. Контейнеры Windows Server и Hyper-V.**

**Контейнеры Windows Server -** обеспечивают изоляцию приложений благодаря изоляции процессов и пространств имен. Контейнер Windows Server использует одно ядро ОС совместно с хостом, на котором он работает, и всеми остальными контейнерами на этом узле.



WSC – является аналогом Linux контейнеров. Контейнеры Windows используют одно ядро с ОС, которое динамично разделяют его между собой и хостом. Процесс распределения ресурсов (CPU, ОЗУ, сеть) берет на себя ОС. Файлы ОС и запущенные службы проецируются в пространство имен каждого контейнера.

****

В этой конфигурации каждый контейнер использует свою копию ядра, изолированную от других контейнеров, но при этом он обладает характеристиками обычного контейнера (быстрое развёртывание, использование библиотеки шаблонов Docker, stateless, мощные возможности по управлению).

# **52. Использование контейнеров в Microsoft Azure.**

Azure Container Service — облачный сервис контейнеризации на базе Docker Swarm ( кластеризация) и DC/OS (Apache Mesos).

DC/OS – это операционная система центра обработки данных, а также распределенная система. Операционная система основана на Apache Mesos distributed kernel.

Основные способы использования контейнеров в Microsoft Azure:

**Сервисы контейнеров в Microsoft Azure:**

1. **Azure Kubernetes Service (AKS)**: AKS - это управляемый сервис Kubernetes в Azure, который позволяет развертывать, масштабировать и управлять контейнерами с использованием Kubernetes. AKS обеспечивает автоматическое масштабирование, обновление и мониторинг кластеров Kubernetes.
2. **Azure Container Instances (ACI)**: ACI позволяет быстро запускать изолированные контейнеры без необходимости управления инфраструктурой. ACI идеально подходит для запуска одиночных контейнеров или задач, требующих быстрого развертывания.
3. **Azure Container Registry (ACR)**: ACR - это управляемый реестр контейнеров в Azure, который позволяет хранить и управлять образами контейнеров. ACR обеспечивает безопасное хранение образов и интеграцию с другими сервисами Azure.

**Интеграция с другими сервисами Azure:**

1. **Azure DevOps**: Azure DevOps обеспечивает интеграцию с контейнерами для автоматизации процесса развертывания и CI/CD. С помощью Azure DevOps можно настроить непрерывную поставку приложений в контейнерах.
2. **Azure Monitor**: Azure Monitor предоставляет мониторинг и управление контейнерами в реальном времени. Он позволяет отслеживать производительность, доступность и состояние контейнеров в Azure.

# **53. Облачные хранилища данных назначение, основные характеристики, преимущества и недостатки.**

**Облачное хранилище данных** (cloud storage) — модель онлайн-хранилища, в котором данные хранятся на многочисленных распределённых в сети серверах, предоставляемых в пользование клиентам.

**Плюсы**:

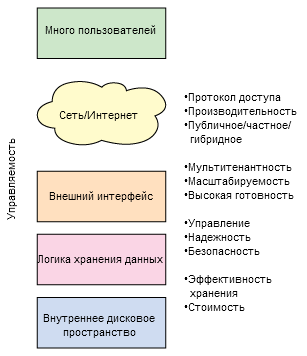
1. Возможность бесплатно хранить определённый объём файлов, у каждого провайдера есть свои условия использования и ограниченное свободное место;
2. Оплата услуг идёт только за фактическое использование хранилища, а не за аренду всего сервера;
3. Уменьшение общих издержек на ИТ инфраструктуру организации, т.е. клиентам нет необходимости заниматься приобретением, поддержкой и обслуживанием собственной ресурсов по хранению данных;
4. Неограниченный доступ к данным отовсюду;
5. резервирование и все процедуры, связанные с сохранностью и целостностью данных, производятся самим провайдером облачного центра, клиент в этот процесс никак не вовлекается;
6. гибкость облачных хранилищ, сервис подходит практически, под все операционные системы, как персональные компьютеры, так и мобильные;
7. синхронизация файлов и доступ к облачным файлам, через операционную систему пользователя, зависит от самого клиента облака, поэтому нужно обращать внимание при выборе облака, на его готовность к работе с десктопным клиентом.

**Минусы**:

1. Если нет интернета, то нет доступа к данным в облаке.
2. Конфиденциальность информации - не все сервисы шифруют данные в облаке, что создает потенциальную угрозу перехвата информации.

# **54. Общая архитектура хранения данных в облаке.**

Обобщенно облачная архитектура хранения данных представляет собой внешний интерфейс, который предоставляет API для доступа к накопителям



|  |  |
| --- | --- |
| **Характеристика** | **Описание** |
| Управляемость | Способность управлять системой при наличии минимальных ресурсов |
| Метод доступа | Протокол, через который предоставляются услуги облачного хранения данных |
| Производительность | Измеряется пропускной способностью и временем задержки |
| Мультитенантность | Поддержка множества пользователей (арендаторов) |
| Масштабируемость | Возможность постепенного наращивания для удовлетворения новых требований или обработки повышенной нагрузки |
| Готовность данных | Измеряется временем безотказной работы системы |
| Управление | Возможность управлять системой — в частности, выбирая стоимость, производительность или другие характеристики |
| Эффективности хранения | Мера эффективности использования накопителей |
| Стоимость | Мера стоимости хранения данных (обычно в долларах за гигабайт) |

# **55. Методы доступа к облачным хранилищам данных.**

Для доступа к ОХД используются следующие методы доступа:

1. API Web-сервисов в том числе RESTfull-API (Amazon S3), Windows Azure, Mezeo Cloud Storage Platform;

Использование API Web-сервиса, требует интеграции с приложенем, поэтому используются и общие методы доступа:

1. API На основе файлов:

NFS;

Common Internet File System(CIFS);

FTP;

1. API На основе блоков:

iSCSI

1. Другие API:

WebDAV (Web-based Distributed Authoring and Versioning)

# **56. Облачные модели хранения данных.**

* 1. Частное облако ЧД (IBM, Parascale и Cleversafe (которая предлагает программное обеспечение и/или оборудование для внутреннего облака));



* 1. Публичное облако ХД (Amazon и Nirvanix);



* 1. Гибридное облако ХД (Nirvanix, Egnyte).



# **57. Типы облачных хранилищ**

Все типы облачных хранилищ делятся на:

1) блочные (диски);

2) файловые;

3) «базовые» (СУБД);

4) объектные.

|  |  |  |
| --- | --- | --- |
| **Тип облачного ресурса** | **Тип данных** | **Примечание** |
| Диск | Файлы, папки | Можно управлять файловой системой диска, разбивать его на логические диски и т. д.; доступное дисковое пространство ограничено размером предоставленного диска |
| Папка | Файлы, папки | Нельзя управлять файловой системой и логической структурой дискового пространства; размер хранилища ограничен тарифом, а не размером диска |
| Набор данных | Числа, строки и другие атомарные данные | Приложение должно «знать» логические связи между этими данными, чтобы их правильно использовать |
| Объект | Набор связанных данных | Как правило, этот набор соответствует чему-то из реального мира: пользователю, документу, организации, товару и т. д. |

Между собой они различаются степенью агрегации данных — если дисковые и файловые хранилища содержат в себе атомарные данные, то в СУБД и, тем более, в объектных хранилищах данные собраны в связанные структуры, которыми можно оперировать на более высоком, прикладном, уровне Невозможно заранее сказать, какой из названных типов хранилищ данных является наилучшим — его выбор зависит от характера решаемых прикладных задач и применяемого программного обеспечения.

# **58. Обеспечение производительности облачных хранилищ данных. Протокол FASP.**

Главная задача облачной системы хранения данных ― это перемещение данных между пользователем и удаленным поставщиком облачных услуг.

Проблема в том, что TCP управляет потоком данных на основе подтверждения приема пакетов из удаленного узла. Потеря или задержка пакетов приводит к применению мер по ограничению скоплений пакетов с дополнительным ограничением производительности во избежание глобальных сетевых проблем.

Amazon с помощью Aspera Software решила эту проблему, исключив TCP при передаче данных.

Был разработан новый протокол *Fast and Secure Protocol* (FASP™) на базе UDP. UDP позволяет управлять заторами протоколу прикладного уровня FASP

Работая со стандартными сетевыми адаптерами (без ускорения), FASP эффективно использует доступную приложению полосу пропускания и исключает главные узкие места традиционных схем массовой передачи данных.

# **59. Методы обеспечения высокой готовности облачных хранилищ данных. Алгоритм рассредоточения данных IDA.**

**Методы обеспечения высокой готовности облачных хранилищ данных:**

1.Резервное копирование данных: Регулярное создание резервных копий данных и их хранение на отдельных серверах или облачных хранилищах для восстановления в случае сбоев.

2. Репликация данных: Дублирование данных на нескольких узлах или центрах обработки данных для обеспечения доступности и отказоустойчивости.

3. Геораспределение данных: Распределение данных по разным географическим регионам для защиты от региональных катастроф и обеспечения доступности в разных частях мира.

4. Использование технологий RAID: Применение технологий RAID (Redundant Array of Independent Disks) для обеспечения отказоустойчивости и высокой доступности данных.

**Алгоритм рассредоточения данных (IDA - Information Dispersal Algorithm):**

Алгоритм рассредоточения данных (IDA) - это метод разделения и распределения данных на несколько узлов для обеспечения безопасности и отказоустойчивости. Он используется для защиты данных от потери или повреждения путем разделения данные на фрагменты и распределения их по разным узлам.

1. Разделение данных: Исходные данные разделяются на фрагменты с помощью математических операций.

2. Распределение фрагментов: Фрагменты данных распределяются по разным узлам или серверам в облаке.

3.Восстановление данных: Для восстановления исходных данных необходимо собрать достаточное количество фрагментов с разных узлов.

4. Обеспечение безопасности: IDA обеспечивает безопасность данных путем их рассредоточения и устойчивости к отказам узлов.

# **60. Apache Hadoop. Распределенная файловая система HDFS. Репликации HDFS. Технологии работы с большими данными. MapReduce.**

Является одной из основополагающих технологий «[больших данных](https://ru.wikipedia.org/wiki/%D0%91%D0%BE%D0%BB%D1%8C%D1%88%D0%B8%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5)». Разработан на [Java](https://ru.wikipedia.org/wiki/Java) в рамках вычислительной парадигмы [MapReduce](https://ru.wikipedia.org/wiki/MapReduce), согласно которой приложение разделяется на большое количество одинаковых элементарных заданий, выполнимых на узлах кластера и естественным образом сводимых в конечный результат.

Проект состоит из четырёх модулей:

1. Hadoop Common ([связующее программное обеспечение](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) — набор инфраструктурных программных библиотек и утилит, используемых для других модулей и родственных проектов),
2. HDFS ([распределённая файловая система](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D0%B0%D1%8F_%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)),
3. YARN(система для планирования заданий и управления кластером)
4. Hadoop MapReduce(платформа программирования и выполнения распределённых MapReduce-вычислений).

* В HDFS вместо таблицы используется специальный сервер — сервер имён (NameNode), а данные разбросаны по серверам данных (DataNode).
* Данные разбиты на блоки (обычно по 64Мб или 128Мб)
* HDFS имеет классическую Unix-овскую древовидную структуру директорий, пользователей с триплетом прав, и даже схожий набор консольных команд.
* Выполняет 2 главные задачи: запись метаданных и хранение собственно данных.

**MapReduce** обеспечивает параллельные локальные вычисления на машинах кластера Hadoop с блоками данных, что обеспечивает быструю обработку больших объемов данных.

Принцип работы MapReduce состоит из двух шагов: map и reduce. Вот как происходит процесс:

**Map**: получение входных данных главным узлом (master node) -> разделение информации на части -> передача данных рабочим узлам (worker node).

**Reduce**: получение главным узлом ответа от рабочих узлов -> формирование результата.

# **61. Apache Spark. Назначение, архитектура и возможности.**

[Apache Spark](http://spark.apache.org/) - альтернатива Hadoop. представляет собой движок для распределённой обработки данных. Apache Spark обычно использует компоненты HDFS и YARN для своей работы.

Архитектура Spark включает в себя 3 главных компонента:

* + хранилище данных;
  + API;
  + менеджер кластера.

В специфических задачах, построенных на многократных обращениях к одному набору данных, «чистый» Spark работает быстрее Hadoop до **30 раз**. Например, это интерактивный дата-майнинг и итерационные алгоритмы, которые активно используются в системах машинного обучения.

Наиболее часто используется поверх Hadoop, заменяя модуль MapReduce и добавляя большее количество операций.

Spark требует наличия менеджера кластера и распределенной системы хранения данных. Если вопрос управления кластерами решается путем использования нативных средств, Hadoop YARN или Apache Mesos (при работе с многоузловыми кластерами), то распределенная система хранения данных может быть исключительно посторонней.

Именно по этой причине в большинство проектов с использованием технологий Big Data, Spark устанавливается поверх «слона»: связка из передовых приложений для аналитики от Spark и Hadoop Distributed File System позволяет программам на кластерах Hadoop выполняться до 100 раз быстрее в RAM и до 10 раз быстрее на ROM.

Spark предлагает ряд операций по обработке данных. Поэтому при разработке приложений с технологиями Больших Данных, Spark чаще всего заменяет именно Hadoop MapReduce, а не весь Hadoo

# **62. Средства работы с BigData в Azure и AWS.**

Windows Azure позволяет быстро создавать Hadoop-кластеры на базе полнофункционального Apache Hadoop и затем с помощью Windows Azure PowerShell и интерфейса командной строки Windows Azure можно без труда интегрировать HDInsight в существующие рабочие процессы аналитики, чтобы получать действенные прогнозы от HDInsight путем интеллектуального анализа данных в Microsoft Excel.

Amazon EMR предоставляет управляемую инфраструктуру Hadoop, которая способна эффективно, быстро и экономично обрабатывать большие объемы данных на динамически масштабируемых инстансах Amazon EC2. В EMR можно также запускать другие известные распределенные инфраструктуры, включая [Apache Spark](https://aws.amazon.com/emr/details/spark/), [HBase](https://aws.amazon.com/emr/details/hbase/), [Presto](https://aws.amazon.com/emr/details/presto/) и [Flink](https://aws.amazon.com/blogs/big-data/use-apache-flink-on-amazon-emr/). С Amazon EMR можно использовать разные хранилища данных, в том числе Amazon S3, распределенную файловую систему Hadoop (HDFS) и Amazon DynamoDB.

# **63. Обзор возможностей, преимущества и недостатки персональных облачных хранилищ: OneDrive, Dropbox, Google Drive, Яндекс Диск, Mega.**

**Google Drive**: Облачное хранилище делит пространство между Google Drive, Gmail и Google Photo.

* В сервисе можно хранить документы, фотографии, музыку, видео и многие другие файлы – всего 30 типов.
* Максимальный обьем файла 5 Тбайт.
* Доступен в веб - браузерах, Windows, Mac OS, Android, iOS и др. 15 u, бесплатно

**OneDrive**: базирующийся на облачной организации интернет-сервис хранения файлов с функциями файлового обмена.

* Интегрирован в Office 365 и в службу Windows Live OneDrive. 5u, бесплатно

**DropBox**: 256 - битное шифрование AES и шифрование SSL; Лучшая технология синхронизации; Неограниченное восстановление файлов и журнал версий; Ссылки доступа с паролем и сроком действия; Настраиваемые уровни доступа; Управление уровнями доступа и пр. 2u, бесплатно. Недостаток Dropbox – подход к выбору папок для синхронизации.

**Mega**: шифрует весь контент прямо в браузере с помощью алгоритма AES; все данные хранятся в «облаке»; ключи доступа к файлам не публикуются в открытом доступе, а распространяются по схеме Friend-to-Friend, защищает личные данные пользователя.

**Яндекс Диск:** Позволяет хранить свои данные на серверах в облаке и передавать их другим пользователям в интернете. Работа построена на синхронизации данных между различными устройствами, возможность автоматической загрузки фото и видеофайлов с цифровых камер и внешних носителей информации на Яндекс. Диск. 10 гб

# **64. Управление пользовательскими данными в хранилище «Dropbox» через REST API.**

Управление пользовательскими данными в хранилище Dropbox с помощью Dropbox REST API предполагает взаимодействие с конечными точками Dropbox API с помощью HTTP-запросов. Шаги:

* Создайте приложение Dropbox и получите токен доступа: После создания приложения, нужно сгенерировать токен доступа, который будете использовать для аутентификации ваших запросов к API.
* Делайте запросы API для взаимодействия с пользовательскими данными: с помощью токена доступа, можно использовать конечные точки Dropbox REST API для взаимодействия с пользовательскими данными.
* Аутентифицируйте запросы API с помощью токена доступа: нужно включить токен доступа в HTTP-заголовки ваших запросов API, используя заголовок авторизации со значением <access-token>.
* Проверить соответствующие разрешения на доступ к данным пользователя: можно использовать поток авторизации Dropbox OAuth 2.0 для получения необходимых разрешений.
* Документация Dropbox API: подробная информация о доступных конечных точках и их параметрах.