М. С. НИКОЛЮКИН, А. Д. ОБУХОВ, Ю. В. ЛИТОВКА

ОБЛАЧНЫЕ ТЕХНОЛОГИИ

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет»

М. С. НИКОЛЮКИН, А. Д. ОБУХОВ, Ю. В. ЛИТОВКА

ОБЛАЧНЫЕ ТЕХНОЛОГИИ

Утверждено Ученым советом университета
в качестве учебного пособия для бакалавров 3 курса
направления подготовки 09.03.01 «Информатика и вычислительная техника»,
изучающих дисциплины «Облачные технологии» и «Интернет-технологии»,
всех форм обучения

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

УДК 004.77
ББК 32.971.3
Н63

Р е ц е н з е н т ы :

Доктор физико-математических наук, профессор,
начальник НИИ математики, физики и информатики
ФГБОУ ВО «ТГУ имени Г. Р. Державина»
Е. С. Жуковский

Доктор технических наук, профессор, заведующий кафедрой
«Компьютерно-интегрированные системы в машиностроении»
ФГБОУ ВО «ТГТУ»
В. Г. Мокрозуб

Николюкин, М. С.

Н63 Облачные технологии [Электронное ресурс] : учебное пособие / М. С. Николюкин, А. Д. Обухов, Ю. В. Литовка. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2024. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 2,4 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2757-3

Рассмотрены основные понятия облачных технологий, приведены общие сведения по работе и взаимодействию с ними.

Предназначено для бакалавров 3 курса направления подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплины «Облачные технологии» и «Интернет-технологии», всех форм обучения.

УДК 004.77
ББК 32.971.3

ISBN 978-5-8265-2757-3

ВВЕДЕНИЕ

В современном мире, где информационные технологии развиваются с невиданной скоростью, облачные технологии становятся краеугольным камнем для предприятий и индивидуальных разработчиков по всему миру. Этот учебник представляет собой путеводитель по миру облачных технологий, предназначенный для того, чтобы обеспечить читателям фундаментальное понимание этой революционной области, а также оснастить их практическими навыками для реализации облачных решений.

Начиная с основных концепций и принципов, таких как *IaaS, PaaS,* и *SaaS*, и продвигаясь к более сложным аспектам, включая масштабирование, контейнеризацию и автоматизацию процессов разработки, пособие предлагает шаг за шагом разобраться в том, как облачные технологии меняют подходы к *IT*-инфраструктуре, разработке программного обеспечения и командной работе.

Мы рассмотрим *Linux* и основы работы с командной строкой, которые являются неотъемлемой частью работы с облачными серверами, и изучим, как использовать *Git* и *GitHub* для управления версиями кода и совместной работы. Так же будут освещены ключевые стратегии настройки и оптимизации веб-серверов и баз данных для обеспечения высокой доступности и производительности облачных приложений.

Кроме того, пособие затрагивает тему масштабирования приложений и данных в облаке и их безопасность, что является критически важным для поддержания работоспособности бизнес-систем в условиях растущего спроса.

Далее будет рассмотрен *Docker* – инструмент, который революционизировал процесс развертывания и управления приложениями благодаря контейнеризации.

И, наконец, мы исследуем концепции *CI*/*CD*, которые позволяют автоматизировать процессы тестирования и развертывания, сокращая время вывода продуктов на рынок и повышая их качество.

1. ПОНЯТИЕ ОБЛАЧНЫХ ТЕХНОЛОГИЙ

1.1. ОБЛАЧНЫЕ ТЕХНОЛОГИИ: ОПРЕДЕЛЕНИЕ И ОСНОВНЫЕ ПРИНЦИПЫ

Облачные технологии – это модель предоставления удобного сетевого доступа по требованию к общему пулу конфигурируемых вычислительных ресурсов (например, сетям, серверам, хранилищам, приложениям и сервисам), которые могут быть быстро предоставлены и освобождены с минимальными усилиями по управлению и минимальным взаимодействием с поставщиком услуг.

Под облачными технологиями понимается предоставление информационных технологий для бизнеса как интернет-сервиса. Вместо покупки дорогостоящих серверов, лицензий на программное обеспечение и прочей ИТ-инфраструктуры компания просто арендует все необходимые ей вычислительные мощности, платформы и приложения у стороннего провайдера облачных услуг.

Технологической основой облачных сервисов являются центры обработки данных (дата-центры) провайдеров, объединенные в высокоскоростные сети передачи данных. Для повышения производительности и эффективности используется виртуализация – разделение физических серверов на изолированные виртуальные среды.

Облачные технологии базируются на ряде ключевых принципов:

1. Самообслуживание по требованию. Пользователь самостоятельно определяет и изменяет необходимые ему вычислительные ресурсы, без взаимодействия с поставщиком услуг. Это существенно ускоряет процесс получения и настройки ресурсов.

2. Широкий доступ по сети. Ресурсы доступны пользователю по сети Интернет с использованием стандартных механизмов и протоколов. Это позволяет работать из любой точки, где есть доступ в Интернет.

3. Объединение ресурсов. Ресурсы провайдера объединены для обслуживания множества пользователей и динамически распределяются между ними. Это дает более эффективное использование ресурсов.

4. Эластичность. Возможность гибкого масштабирования ресурсов для соответствия текущему спросу. Для пользователя ресурсы выглядят практически безграничными.

5. Измеряемость сервиса. Автоматический мониторинг и оптимизация использования ресурсов с предоставлением отчетности по запросу. Это позволяет оптимизировать расходы на инфраструктуру.

Благодаря этим принципам облачные технологии обеспечивают предприятиям гибкость и масштабируемость ИТ-инфраструктуры, а также возможность платить только за фактически использованные ресурсы.

Концепция облачных технологий начала формироваться в конце 1990 годов. Однако их бурное развитие стало возможным только в последние 10 – 15 лет с распространением высокоскоростного Интернета, доступных вычислительных мощностей и новых технологий виртуализации.

Сегодня облачные сервисы активно применяются для решения самых разных бизнес-задач. Они помогают компаниям оптимизировать расходы на ИТ, масштабировать инфраструктуру и ускорять вывод цифровых продуктов на рынок. Роль облачных технологий будет только возрастать в ближайшие годы.

1.2. КАТЕГОРИИ ОБЛАЧНЫХ УСЛУГ: IAAS, PAAS, SAAS

Благодаря использованию систем облачных вычислений разработчики и сотрудники ИТ-отделов могут сконцентрироваться на выполнении самых важных задач и не тратить усилий на рутинную и трудоемкую работу по материально-техническому снабжению, обслуживанию и планированию мощности вычислительных ресурсов. По мере роста популярности данных систем возникло несколько различных моделей и стратегий развертывания, позволяющих удовлетворить потребности различных категорий пользовате-

лей. Каждый тип облачных услуг и каждый способ развертывания обеспечивают свой уровень контроля, гибкости и управляемости. Понимание различий между моделями *Iaas, Paas* и *Saas* и особенностей стратегий развертывания поможет принять решение о том, какой набор услуг наиболее полно удовлетворит потребности.

IaaS (*Infrastructure as a Service*) – модель, в которой потребителю предоставляется инфраструктура (вычислительные мощности, хранилища данных, сети и другие базовые ресурсы) для размещения произвольного ПО, включая операционные системы и приложения. Примеры сервисов: *Amazon EC2, Microsoft Azure, Google Compute Engine*.

PaaS (*Platform as a Service*) – модель, в которой потребителю предоставляется готовая платформа для разработки, развертывания и управления приложениями. Обычно включает ОС, среду исполнения, базы данных и средства разработки. Примеры: *Heroku, AWS Elastic Beanstalk, Microsoft Azure App Service.*

SaaS (*Software as a Service*) – модель, в которой потребителю предоставляется готовое к использованию облачное приложение. Потребитель может настраивать приложение, но не управляет инфраструктурой. Примеры: *Gmail, Slack, Salesforce, Dropbox*.

Главное различие этих моделей в уровне управления инфраструктурой и степени абстракции для пользователя. В *IaaS* пользователь получает базовые «чистые» ресурсы и максимальную гибкость. В *SaaS* пользователь получает готовое приложение без доступа к инфраструктуре. *PaaS* занимает середину, предоставляя платформу для разработки.

Выбор модели зависит от потребностей бизнеса и готовности к управлению ИТ-ресурсами. *IaaS* требует больших усилий по настройке, зато дает максимум контроля. *SaaS* снимает заботы об инфраструктуре, но ограничивает гибкость. *PaaS* – компромиссный вариант. Комбинируя услуги разных моделей, можно гибко выстраивать ИТ-инфраструктуру.

1.3. ПРЕИМУЩЕСТВА И РИСКИ ОБЛАЧНЫХ ТЕХНОЛОГИЙ

Переход на облачные технологии – это важное стратегическое решение для любого бизнеса. Использование облачных сервисов дает компаниям массу новых возможностей, но в то же время несет определенные риски, о которых важно помнить. Разберемся подробнее в ключевых преимуществах и недостатках облачных технологий.

Преимущества:

1. Экономия на инфраструктуре. При использовании облака нет необходимости вкладывать средства в дорогостоящее оборудование и лицензии на ПО, арендовать серверные мощности, нанимать большой ИТ-штат для обслуживания инфраструктуры. Все это предоставляется провайдером облачных услуг по модели «оплата по факту». Это позволяет компаниям оптимизировать расходы на ИТ.

2. Гибкость и масштабируемость. Облако легко растет вместе с бизнесом. Можно оперативно наращивать или уменьшать вычислительные мощности, хранилища данных, количество пользователей – и платить только за фактическое потребление. Это избавляет от необходимости прогнозировать потребности бизнеса на годы вперед.

3. Высокая доступность сервисов. Облачные провайдеры гарантируют непрерывную работу сервисов и надежное резервное копирование данных, используя современные средства отказоустойчивости. Это позволяет минимизировать простои бизнес-систем.

Однако у облачных технологий есть и потенциальные недостатки:

1. Зависимость от провайдера и Интернета. В облаке данные и сервисы находятся под контролем сторонней компании. Сбои у провайдера или нарушения связи могут привести к недоступности ваших систем.

2. Риски безопасности данных. Хранение данных на сторонних серверах несет потенциальные угрозы утечки или взлома. Необходимо тщательно изучать политику безопасности провайдера.

3. Технологическая зависимость от вендора. Со временем может возникнуть «вендор-лок» – сложности с переходом от одного облачного провайдера к другому.

Чтобы минимизировать риски, компаниям стоит тщательно подходить к выбору провайдера облачных услуг, настраивать резервное копирование, использовать гибридные облачные решения. Взвешенный подход поможет максимально реализовать потенциал облачных технологий для бизнеса.

1.4. ОБЗОР КРУПНЕЙШИХ ОБЛАЧНЫХ ПРОВАЙДЕРОВ

На сегодняшний день существует множество крупных игроков на рынке облачных технологий, предлагающих широкий спектр сервисов для бизнеса. Рассмотрим подробнее ключевых глобальных и российских провайдеров облачных услуг.

Лидерами в области облачных технологий по праву считаются такие компании, как *Amazon Web Services*, *Microsoft Azure* и *Google Cloud Platform*. Эти гиганты предлагают огромный выбор облачных сервисов – от инфраструктуры и платформ до готовых приложений. При этом они имеют колоссальные вычислительные мощности и надежную поддержку.

Amazon Web Services (*AWS*) – это первопроходец и мировой лидер в облачных вычислениях. *AWS* предлагает самый широкий спектр сервисов – более 200 различных продуктов от хранения данных до искусственного интеллекта. Глобальная инфраструктура *AWS* включает десятки регионов по всему миру.

Microsoft Azure – второй по популярности облачный провайдер, который активно наращивает свою долю рынка. Платформа *Azure* интегрирована с другими сервисами *Microsoft*, такими как *Office* 365, *Dynamics* 365, *Power BI* и др. Это удобно для организаций, которые уже используют продукты *Microsoft*.

Google Cloud Platform специализируется на передовых технологиях вроде искусственного интеллекта, машинного обучения, анализа больших

данных. *GCP* широко применяется для создания высоконагруженных веб-сервисов и продуктов.

Среди российских облачных провайдеров лидируют Яндекс.Облако и *VK Cloud Solutions*. Эти компании развивают облачную инфраструктуру в российских дата-центрах, что важно для локальных законодательных требований. Их услуги оптимизированы для нужд российского рынка.

Подводя итог, отметим, что ведущие облачные провайдеры предоставляют надежные и масштабируемые технологические платформы для создания цифровых сервисов и продуктов. При этом стоит тщательно анализировать свои потребности и выбирать оптимальное сочетание глобальных и локальных облачных сервисов.

1.5. ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ОБЛАЧНЫХ РЕШЕНИЙ

Облачные технологии широко применяются в различных сферах. Рассмотрим несколько примеров их использования (рис. 1):

1. Масштабируемые веб-сервисы. Компании, предоставляющие онлайн-сервисы с большим трафиком (соцсети, видеохостинги, интернет-магазины), используют облачные решения для гибкого масштабирования инфраструктуры. Примеры: *Netflix*, *X*, *Dropbox*.

2. Мобильные приложения. Разработчики мобильных приложений опираются на облачные сервисы для хранения данных, *push*-уведомлений, бэкенд-сервисов. Это ускоряет разработку и масштабирование.

3. Анализ больших данных. Обработка и анализ больших массивов данных требует значительных вычислительных мощностей, которые эффективно предоставляют облачные сервисы.

4. Бэкапы и аварийное восстановление. Хранение резервных копий в облаке обеспечивает надежность и доступность данных при сбоях.

5. Гибридные облака. Компании используют сочетание локальной инфраструктуры и общедоступного облака для оптимизации затрат и повышения отказоустойчивости.

6. Удаленная работа. Сотрудники получают гибкий доступ к корпоративным данным и приложениям для удаленной работы с помощью облачных сервисов.

Таким образом, облачные технологии позволяют компаниям любого масштаба повысить эффективность бизнеса за счет гибкости и оптимизации ИТ-инфраструктуры.

Рис. 1. Применение облачных технологий

2. LINUX НА ОБЛАЧНЫХ СЕРВЕРАХ

2.1. ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ LINUX

Linux – одна из самых популярных операционных систем в мире, особенно в облачных вычислениях. В отличие от *Windows* или *MacOS*, *Linux* – это открытая система, которая бесплатна для использования и модификации.

Первая версия *Linux* была выпущена в 1991 году финским программистом Линусом Торвальдсом (рис. 2). С тех пор сообщество разработчиков по всему миру внесло огромный вклад в развитие и улучшение *Linux*. Сегодня существует множество дистрибутивов *Linux*. Они отличаются графическим интерфейсом, установленным ПО, подходом к обновлениям и другими деталями.

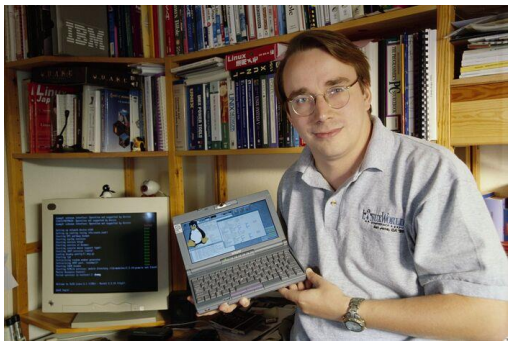


Рис. 2. Линус Торвальдс

Популярность *Linux* на облачных серверах обусловлена стабильностью, надежностью и безопасностью. Linux меньше подвержен вирусам и уязвимостям. Linux гибок и настраиваем под конкретные задачи и масштабируется на серверах с множеством ядер процессора и оперативной памяти.

Дистрибутивы *Linux* существуют, чтобы удовлетворить потребности разных групп пользователей (рис. 3).

Рис. 3. Множество дистрибутивов *Linux*

Рассмотрим некоторые из них:

1. *Debian* – один из самых популярных и стабильных дистрибутивов Linux, особенно для серверов. Он славится надежностью, безопасностью и открытостью. *Debian* использует собственный менеджер пакетов *APT* и имеет очень строгий процесс тестирования обновлений перед релизом. Это гарантирует стабильность системы. Debian отлично подходит для критически важных серверных задач, где простои недопустимы.

2. Еще один популярный дистрибутив – *Ubuntu*, который основан на *Debian*, но сделан более дружелюбным для настольных систем. *Ubuntu* регулярно выпускает новые версии с обновлениями. Он часто используется начинающими пользователями *Linux*.

3. Для компаний, нуждающихся в профессиональной поддержке и сертификации, подойдет *Red Hat Enterprise Linux*. Это коммерческий дистрибутив с гарантированным SLA, строгими проверками безопасности и долгосрочной поддержкой.

4. *CentOS* – еще один популярный выбор, особенно для веб-серверов. Это бесплатная ОС, основанная на *RHEL* и совместимая с ней. *CentOS* фокусируется на стабильности и предсказуемости.

Чтобы управлять сервером *Linux* в облаке, используется протокол *SSH* (*Secure Shell*). Он позволяет подключаться к удаленному серверу и выполнять команды через терминал. *SSH* шифрует трафик между клиентом и сервером, обеспечивая безопасность.

Чтобы начать работу, нужно установить *Linux* на виртуальный или выделенный сервер в облаке (*VPS*/*VDS*). Популярные облачные провайдеры предлагают образы с предустановленным *Linux*. Затем настраивается *SSH*-доступ для администрирования. Дальше можно устанавливать нужные приложения – веб-серверы, СУБД, языки программирования.

Linux открывает огромные возможности для создания надежных и производительных систем в облаке. Освоив основы работы с ним, можно эффективно использовать все преимущества облачных вычислений.

2.2. УСТАНОВКА LINUX НА VPS/VDS

Установка *Linux* на виртуальный или выделенный сервер в облаке – важный первый шаг в создании гибкой и масштабируемой ИТ-инфраструктуры. Давайте подробно разберем этапы и нюансы этого процесса.

Прежде всего, необходимо определиться с типом сервера – *VPS* или *VDS*. *VPS* – это виртуальная машина, работающая в среде виртуализации на физическом сервере провайдера облака. Ресурсы физического сервера (ядра процессора, ОЗУ, хранилище) распределяются между всеми *VPS* с помощью гипервизора.

VDS – выделенный физический сервер без виртуализации. Все его ресурсы полностью принадлежат клиенту *VDS*. Это обеспечивает предсказуемую производительность и минимизирует влияние «соседей», но стоит дороже *VPS*.

Следующий шаг – выбор дистрибутива *Linux*. В примерах далее будет использоваться *Debian*, так как это один из самых популярных и стабильных дистрибутивов для серверов.

На *VPS* часто уже установлен *Linux*. Но можно переустановить его или установить с нуля. Для этого загружаем *ISO*-образ выбранного дистрибутива в виртуальный привод *VPS*. На *VDS* процесс установки ничем не отличается от физического железа. Отличия *VPS* и *VDS* приведены на рис. 4.

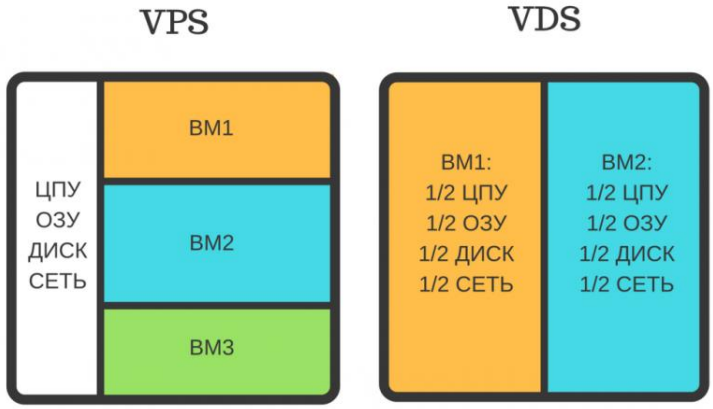


Рис. 4. Отличия *VPS* и *VDS*

После базовой установки следует выполнить постинсталляционную настройку – обновить ядро и ПО до актуальных версий, настроить сеть и брандмауэр, развернуть систему мониторинга, механизмы резервного копирования и восстановления.

Особое внимание стоит уделить безопасности – использовать *SSH*-ключи для доступа, регулярно менять пароли, установить защиту от *DDoS* и вредоносного ПО. Также важно правильно настроить логирование для упрощения диагностики проблем.

Грамотно установив и настроив *Linux*, можно приступать к развертыванию нужных приложений и сервисов поверх него – веб-серверов, СУБД, *CI*/*CD*-систем. *Linux* предоставляет огромную гибкость в конфигурации и адаптации под бизнес-задачи заказчика.

2.3. КОМАНДНАЯ СТРОКА LINUX: ОСНОВЫ

Работая с облачными серверами на *Linux*, важно знать базовые команды для навигации по файловой системе и управления приложениями (рис. 5).

Рис. 5. Вид командной строки

Рассмотрим наиболее полезные из команд:

Работа с файлами и каталогами:

1. *ls* – просмотр содержимого каталога:

ls /home – выводит список файлов и подкаталогов в каталоге */home*.

2. *cd* – перейти в указанный каталог:

cd /tmp – переходит в каталог */tmp*.

3. *mkdir* – создать новый каталог:

mkdir test – создаст каталог *test* в текущем каталоге.

4. *rm* – удалить файл:

rm file.txt – удалит файл *file.txt*.

5. *mv* – переместить или переименовать файл:

mv file.txt new_file.txt – переименует *file.txt* в *new_file.txt*.

6. *cp* – копировать файл:

cp file.txt /home/copy.txt – скопирует *file.txt в /home/copy.txt*.

7. *pwd* – вывести путь текущего каталога, показывает абсолютный путь текущего каталога.

8. *find* – поиск файлов:

*find /home -name *.jpg* – найдет файлы с расширением *.jpg* в каталоге */home*.

Управление пользователями:

1. *useradd* – создать пользователя:

useradd testuser – создаст пользователя с именем *testuser*.

2. *passwd* – установить пароль:

passwd testuser – установит пароль для пользователя *testuser*.

3. *su* – сменить пользователя:

su testuser – сменит текущего пользователя на *testuser*.

Управление правами:

– *chown* – сменить владельца файла:

chown testuser file.txt – сделает пользователя *testuser* владельцем файла *file.txt*.

– *chmod* – изменить права доступа к файлу:

chmod 755 file.txt – установит для файла права доступа 755.

– *chgrp* – назначить группу для файла:

chgrp users file.txt – назначит группу *users* владельцем файла *file.txt.*

Работа с процессами:

– *ps* – посмотреть запущенные процессы:

ps aux – выводит список всех текущих процессов в системе.

– *top* – список активных процессов, выводит динамически обновляемый список запущенных процессов.

– *kill* – принудительно завершить процесс по *PID*:

kill 12345 – посылает сигнал завершения процессу с *PID* 12345.

– *bg* – перевести процесс в фоновый режим:

bg %1 – переводит процесс с *jid* 1 в фоновый режим.

– *crontab* – редактор планировщика задач *cron*:

crontab -e – открывает текстовый редактор для создания заданий *cron.*

Получение информации:

– *df* – проверить использование дискового пространства:

df-h – выводит информацию об использовании дисков в человеко-читаемом формате.

– *du* – проверить размер каталога:

du -sh /home – выводит общий размер каталога */home*.

– *man* – вывести справочную страницу по команде:

man ls – показывает справку по использованию команды *ls*.

– *uname -a* – вывести информацию о ядре ОС, показывает версию ядра, название ОС и архитектуру процессора.

– *history* – вывести историю использованных команд.

Редактирование текстов:

– *nano* – простой консольный текстовый редактор:

nano test.txt – откроет файл test.txt для редактирования в *nano*.

– *vim* – универсальный консольный текстовый редактор, открывает файл для редактирования в *vim*.

– *grep* – поиск по шаблону в файлах:

*grep «text» *.txt* – найдет строки со словом *text* в *txt* файлах.

Работа с пакетами:

– *apt install* – установить пакет:

apt install nginx – устанавливает пакет *nginx*.

– *apt remove* – удалить пакет:

apt remove nginx – удаляет установленный пакет *nginx*.

– *apt update* – обновить список доступных пакетов в репозиториях.

– *apt upgrade* – обновить установленные пакеты до актуальных версий.

Также стоит упомянуть про каталоги, с которыми ведется работа в системе. Они играют важную роль в организации файловой системы. Корневая директория / содержит все остальные каталоги и файлы.

/home хранит персональные файлы пользователей. */etc* содержит конфигурационные файлы системы. */var* используется для хранения изменяемых данных, таких как логи, кеши, временные файлы.

/bin и */sbin* содержат исполняемые файлы основных утилит, */usr* – установленные пользовательские приложения. */tmp* используется для временных файлов, */dev* – для файлов устройств, */media* и */mnt* – для монтирования внешних носителей.

/opt хранит дополнительно установленное ПО. Такая структура позволяет логически и удобно организовать файлы и каталоги в *Linux*.

Таким образом, администраторам и пользователям легче ориентироваться в файловой системе благодаря стандартным каталогам с определенным назначением и управлять их содержимым с помощью команд.

2.4. УДАЛЕННОЕ УПРАВЛЕНИЕ СЕРВЕРОМ

Secure Shell, или *SSH*, представляет собой протокол, который используется для безопасного доступа к удаленному серверу и управления им. Он необходим для выполнения команд на сервере, редактирования файлов, перемещения данных и выполнения широкого спектра задач администрирования и обслуживания – все это с высоким уровнем шифрования, который предотвращает перехват и подмену данных.

Для подключения к *VDS* или *VPS* по *SSH* существуют различные инструменты. Пользователи *Windows* часто прибегают к использованию *PuTTY* (рис. 6), мощного и удобного в использовании клиента, который предоставляет графический интерфейс для установления *SSH*-соединения. В то же время в операционных системах *Linux* и *macOS* встроенная команда *ssh* в терминале позволяет подключаться к серверам напрямую, без необходимости установки дополнительного ПО.

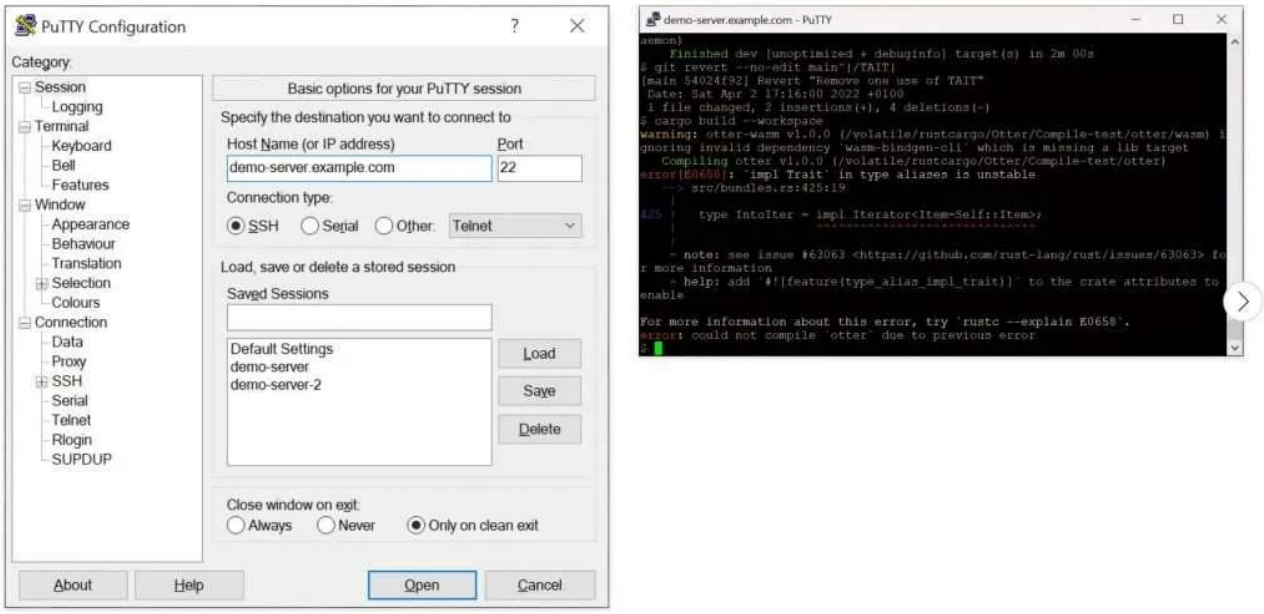


Рис. 6. Интерфейс *PuTTY*

Чтобы использовать *Putty* для подключения, вам нужно запустить программу и ввести *IP*-адрес вашего сервера в поле «*Host Name*». После этого вы выбираете тип подключения – обычно это *SSH*, который уже выбран по умолчанию, и нажимаете кнопку «*Open*» для начала сессии. При первом подключении *Putty* может запросить подтверждение ключа сервера; после его принятия вам будет предложено ввести свои учетные данные, чтобы завершить процесс подключения.

Далее рассмотрим процесс работы с командой *ssh*. Первым делом необходимо открыть терминал на вашем локальном компьютере. В системах на базе *Linux* или *macOS* это делается путем вызова приложения «Терминал» или его аналогов.

Допустим, *VDS* имеет *IP*-адрес 123.456.78.9, и вы хотите подключиться под пользователем *admin*. Команда будет выглядеть так:

ssh admin@123.456.78.9

После ввода этой команды система попросит вас ввести пароль пользователя удаленного сервера.

Если вы настроили аутентификацию по ключам, вместо ввода пароля система проверит ваш приватный ключ. Предполагая, что у вас есть приватный ключ с именем *id_rsa* в вашем каталоге ~/*.ssh*/, подключение будет выглядеть так:

ssh -i ~/.ssh/id_rsa admin@123.456.78.9

Это обеспечивает более высокий уровень безопасности, поскольку ключи гораздо сложнее подделать или украсть, чем пароли. После успешной аутентификации вы попадаете в командную оболочку удаленного сервера. Теперь вы можете выполнять любые команды, как если бы вы были непосредственно за его консолью.

Осуществив подключение по *SSH*, администратор получает полный контроль над удаленным сервером. Это означает, что можно устанавливать и настраивать программное обеспечение, управлять файлами и директориями, запускать и останавливать службы, настраивать сетевые параметры и многое

другое. Все операции выполняются в командной строке, что требует определенных знаний и навыков.

Когда вы закончите выполнение необходимых операций, вы можете безопасно отключиться от сервера, введя команду

exit

Безопасность подключения по *SSH* обеспечивается не только через использование шифрования. Существуют и другие механизмы, такие как аутентификация по ключам, которая позволяет избежать рисков, связанных с перехватом паролей. Помимо этого, настраиваемые правила брандмауэра и изолированные сетевые пространства обеспечивают дополнительный уровень защиты.

Владение навыками работы с *SSH* открывает широкие возможности для администрирования и гибкой настройки облачных серверов. Это фундаментальный инструмент, который должен быть в арсенале каждого специалиста, занимающегося облачными технологиями и системным администрированием.

2.5. УСТАНОВКА И НАСТРОЙКА ПАКЕТОВ

Установка пакетов на облачный сервер – это задача, с которой сталкиваются многие разработчики. В основе любой серверной среды лежат различные языки программирования и среды выполнения, поэтому рассмотрим процесс установки таких критически важных компонентов, как *PHP-FPM* для *PHP*, интерпретатор *Python* и среду выполнения *Java*.

PHP-FPM (*FastCGI Process Manager*) представляет собой альтернативу стандартному PHP и рекомендуется для более крупных сайтов и приложений из-за его способности эффективно обрабатывать запросы, сохраняя высокую производительность и гибкость в настройке. Это особенно заметно при работе с веб-серверами, которые используют архитектуру *FastCGI*, например *Nginx*.

Для установки *PHP-FPM* на большинстве дистрибутивов *Linux* достаточно выполнить команду

sudo apt install php-fpm,

что приведет к установке последней доступной версии *PHP-FPM*.

Python является еще одним фундаментальным инструментом разработчика. Многие современные веб-приложения и скрипты для серверного управления написаны на *Python*. Установить *Python* можно с помощью команды

sudo apt install python3

Java – незаменимый инструмент для запуска множества корпоративных приложений и сервисов. Установка *Java* осуществляется через команду:

sudo apt install default-jdk,

что установит *Java Development Kit*, позволяя компилировать и запускать приложения на *Java*.

Каждый из этих пакетов имеет свои конфигурационные файлы, которые можно редактировать для тонкой настройки работы приложений. Например, для изменения количества выделяемой оперативной памяти для *Java* можно отредактировать файл конфигурации *JVM*. Используя редактор *nano,* вы можете открыть файл настройки командой

sudo nano /etc/java-8-openjdk/jvm.cfg,

и изменить параметры, отвечающие за размер памяти.

Аналогично, для *PHP-FPM* и *Python* вы найдете файлы *.ini* или *.conf,* которые можно редактировать, чтобы настроить, например, лимиты памяти, время выполнения скриптов или параметры безопасности. Понимание того, как работать с этими конфигурациями, даст вам большой контроль над вашей серверной средой и позволит оптимизировать производительность ваших приложений.

3. СЕРВЕРЫ ПРИЛОЖЕНИЙ И СУБД В ОБЛАКЕ

3.1. ОБЗОР ОБЛАЧНЫХ СУБД

Обзор облачных систем управления базами данных (СУБД) представляет собой ключевую область знаний для любого разработчика, который стремится полностью использовать потенциал облачных технологий. Облачные СУБД предлагают гибкость, масштабируемость и высокий уровень доступности, что делает их идеальным выбором для различных приложений, от стартапов до крупных корпораций.

Существует несколько типов облачных СУБД, каждый из которых предлагает свои уникальные преимущества. Например, есть реляционные облачные базы данных, такие как *Amazon RDS* или *Google Cloud SQL*, которые предоставляют традиционные *SQL*-интерфейсы для управления данными с поддержкой стандартов *ACID*, обеспечивающих надежность транзакций. Эти сервисы автоматизируют рутинные задачи управления базами данных, такие как резервное копирование, шардинг и репликация, облегчая масштабирование и обеспечивая высокую доступность.

Для сценариев, требующих более гибких схем данных и возможности быстрого масштабирования, *NoSQL* базы данных, такие как *MongoDB Atlas, Amazon DynamoDB* или *Google Firestore*, предлагают документоориентированные, ключ-значение, широкостолбцовые и графовые базы данных. Они особенно хорошо подходят для приложений, которые должны быстро адаптироваться к изменениям в структуре данных и обрабатывать большие объемы неструктурированных данных.

Кроме того, многие облачные провайдеры предлагают специализированные решения для аналитики данных, такие как *Google BigQuery* или *Amazon Redshift*, которые оптимизированы для выполнения больших и сложных запросов к данным, что идеально подходит для бизнес-аналитики и обработки больших данных.

Выбор облачной СУБД зависит от конкретных требований приложения, включая тип данных, ожидаемый объем трафика, требования к консистентности и доступности, а также от специфических предпочтений разработчиков и бизнес-задач. Важно отметить, что облачные СУБД также предлагают различные опции в плане безопасности, такие как шифрование данных в покое и передаче, управление доступом и аутентификация, что критично для соблюдения нормативных и бизнес-требований в отношении данных.

PostgreSQL является одной из самых популярных и мощных открытых систем управления базами данных, которая часто выбирается для развертывания на виртуальных выделенных серверах (*VDS*). Ее предпочитают за продвинутые функции, такие как надежные механизмы транзакций, поддержка различных расширений и способность обрабатывать большие нагрузки и сложные запросы.

Когда дело доходит до облачных развертываний, *PostgreSQL* также выделяется своей гибкостью и масштабируемостью, что делает ее привлекательной для использования в облачной среде. Многие облачные провайдеры предлагают управляемые инстансы *PostgreSQL,* которые упрощают задачи по настройке, мониторингу, резервному копированию и восстановлению, а также обеспечивают автоматическое масштабирование ресурсов.

На *VDS* или в облачной среде *PostgreSQL* позволяет разработчикам использовать сложные *SQL*-запросы и процедуры, хранение *JSON*, геопространственные данные и многое другое. Благодаря своим возможностям по обеспечению высокого уровня консистентности данных и расширенной поддержке конкурентных операций, *PostgreSQL* становится идеальным выбором для приложений, требующих высокой надежности и производительности. Использование *PostgreSQL* в облаке также обеспечивает дополнительные преимущества в виде улучшенных механизмов безопасности и соблюдения нормативных требований, так как облачные

провайдеры часто включают в свои услуги сложные решения для защиты данных.

Выбор *PostgreSQL* для работы на *VDS* или в облачной среде не только обеспечивает разработчикам мощный инструмент для управления данными, но и предоставляет гибкость в управлении ресурсами и оптимизации производительности приложений.

3.2. НАСТРОЙКА СУБД POSTGRESQL ДЛЯ РАБОТЫ В ОБЛАКЕ

Настройка СУБД *PostgreSQL* на *VDS* в облачной среде включает не только установку и настройку параметров аутентификации, но и задание паролей для повышения безопасности системы.

Установка *PostgreSQL* выполняется через системный пакетный менеджер

sudo apt update

sudo apt install postgresql postgresql-contrib

После установки, чтобы задать или изменить пароль для пользователя «*postgres*», следует войти в интерактивный режим командой

sudo -u postgres psql

В интерактивном режиме задание пароля выполняется с помощью команды *SQL*

\password postgres

Система предложит ввести новый пароль и подтвердить его, устанавливая таким образом защиту для административного пользователя.

PostgreSQL по умолчанию использует порт 5432. Если вы хотите изменить порт, на котором сервер будет «слушать» входящие подключения, вам нужно отредактировать файл *postgresql.con* (рис. 7)*f*:

sudo nano /etc/postgresql/12/main/postgresql.conf

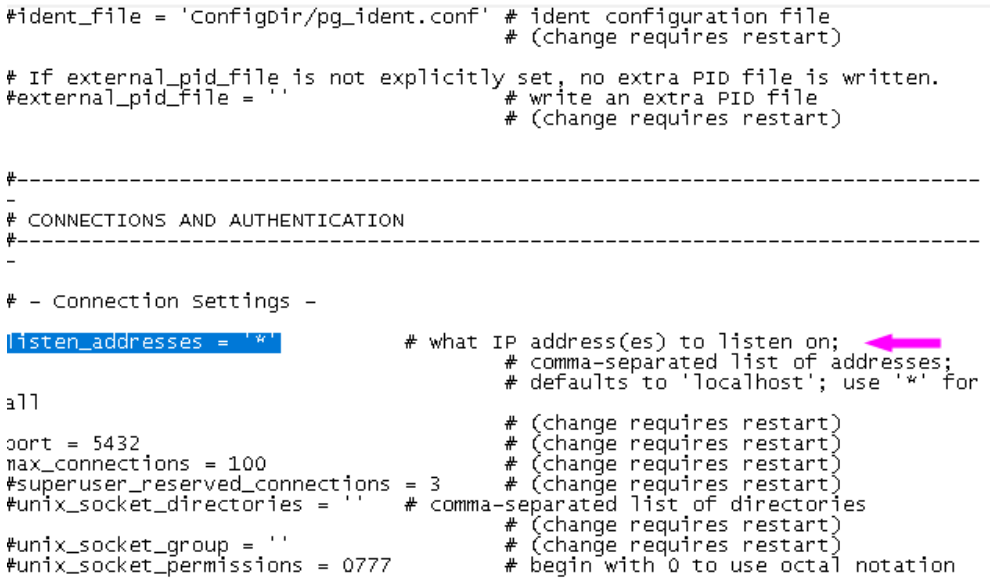


Рис. 7. *Postgresql.conf*

Найдите строку с *port* и измените ее на нужный вам порт, например:

port = 5433 # (change requires restart)

Следующий шаг – редактирование файла *pg_hba.conf* (рис. 8), который управляет политикой аутентификации клиентов. Файл можно открыть с помощью текстового редактора, например *nano:*

sudo nano /etc/postgresql/12/main/pg_hba.conf

В этом файле можно настроить методы аутентификации, такие как *MD*5 или *SCRAM-SHA*-256, для различных хостов и пользователей. Это позволяет более точно контролировать доступ к серверу баз данных.

В файле *pg_hba.conf* можно также настроить доступ для конкретных *IP*-адресов или диапазонов *IP*. Например, чтобы разрешить подключения для пользователя *myuser* с определенного *IP*-адреса 192.168.1.50 используя *MD*5-аутентификацию, добавьте следующую строку:

host all myuser 192.168.1.50/32 md5,

/32 означает, что доступ разрешен только для одного *IP*-адреса. Если вы хотите разрешить доступ для всей подсети, используйте соответствующую маску подсети, например /24 для 192.168.1.0/24. Для того чтобы разрешить все подключения, достаточно на месте *IP*-адреса указать «*all*».

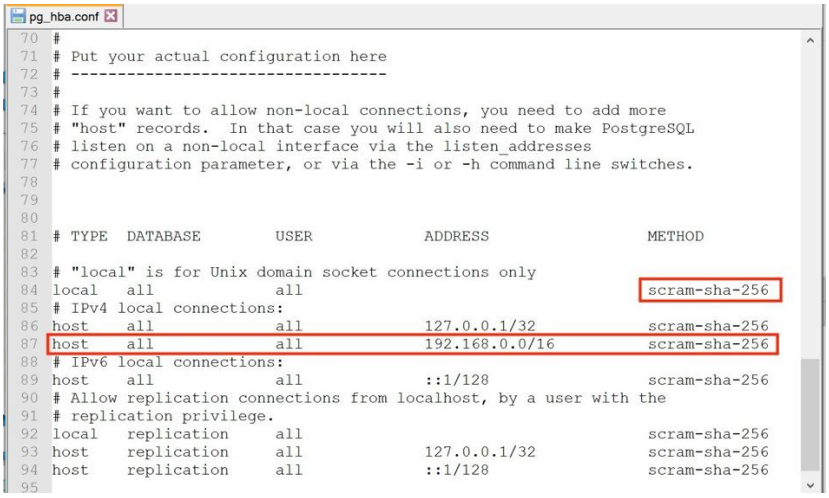


Рис. 8. *pg_hba.conf*

После внесения изменений в файл *pg_hba.conf* необходимо перезапустить службу *PostgreSQL* для применения новой конфигурации:

sudo systemctl restart postgresql

Теперь, когда система настроена и защищена, можно использовать инструменты вроде *pgAdmin* для удобной работы с базой данных через графический интерфейс. *pgAdmin* позволит управлять базами данных, выполнять запросы, настраивать схемы и многое другое, обеспечивая удобную среду для администраторов и разработчиков (рис. 9).

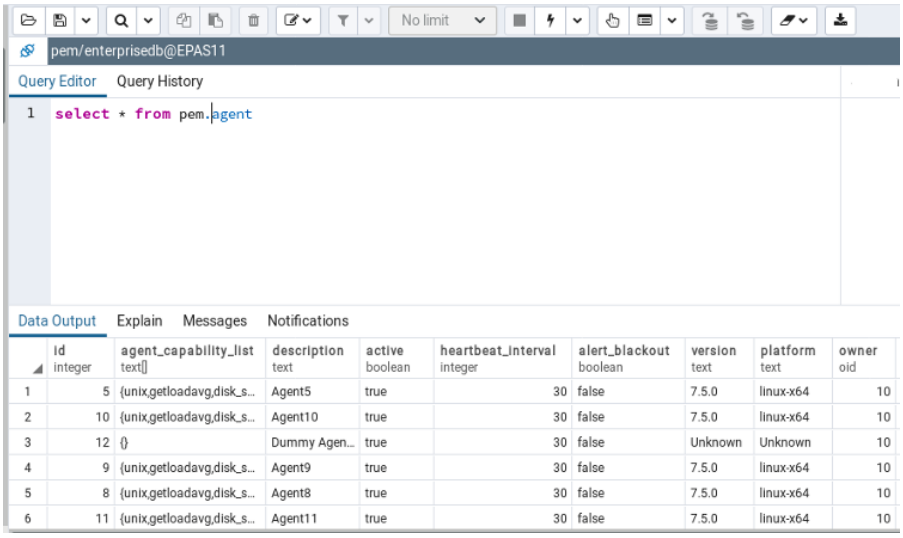


Рис. 9. Интерфейс *pgAdmin*

3.3. ОБЗОР СЕРВЕРОВ ПРИЛОЖЕНИЙ

Серверы приложений являются фундаментальным звеном, которое соединяет пользовательские запросы с бэкенд-логикой и базами данных, выполняя код приложений для обработки этих запросов и возвращая результаты. Это сердце любого веб-приложения, поскольку они обеспечивают необходимую инфраструктуру для развертывания и управления веб-приложениями.

Apache HTTP Server, часто просто называемый *Apache*, является одним из самых старых и надежных серверов приложений, который широко используется в Интернете. Он известен своей модульностью, поддержкой широкого спектра модулей для безопасности, кеширования и оптимизации производительности.

Nginx – относительно новый, но быстро набирающий популярность сервер, выделяется своей производительностью и масштабируемостью, особенно в средах с высоким трафиком и для статического содержимого. Он часто используется в качестве обратного прокси-сервера перед *Apach*e, что позволяет сочетать мощность *Apache* для сложных приложений с эффективностью *Nginx* для обработки статических ресурсов и балансировки нагрузки.

Java-серверы приложений, такие как *Apache Tomcat*, предоставляют среду для запуска *Java*-приложений, включая те, которые используют спецификации *Java EE*, например сервлеты и *JSP*. *Tomcat* особенно популярен в корпоративной среде, где требуется надежность и поддержка обширного набора *Java*-стандартов.

Для *Python* существуют различные веб-серверные решения, такие как *Gunicorn* или *WSGI*, которые часто используются в сочетании с веб-фреймворками вроде *Django* или *Flask*. Эти серверы действуют как шлюзы между веб-серверами и *Python*-приложениями, управляя *WSGI*-запросами, которые являются стандартом для веб-приложений на *Python*.

Серверы приложений представляют собой многослойные и многофункциональные системы, каждая из которых предлагает свои уникальные

преимущества и может быть настроена для обеспечения максимальной производительности и безопасности в зависимости от потребностей организации и ее инфраструктуры.

3.4. НАСТРОЙКА СЕРВЕРОВ ПРИЛОЖЕНИЙ ДЛЯ РАБОТЫ В ОБЛАКЕ

Облачные сервисы часто предлагают предварительно настроенные и оптимизированные для работы в облаке виртуальные машины, на которых уже запущены серверы приложений, такие как *Apache* или *Nginx*. Это позволяет разработчикам сосредоточиться на развертывании и управлении своими приложениями, а не на поддержке самого сервера приложений. Однако для максимальной эффективности и безопасности важно понимать, как настраивать и поддерживать эти серверы, а также как использовать инструменты и сервисы облака для обеспечения масштабируемости и отказоустойчивости.

Настройка *NGINX* начинается с его установки. Команда

sudo apt install nginx

обеспечит установку последней доступной версии.

После установки *NGINX* автоматически запускается, и его можно протестировать, открыв в браузере *http*://[ВАШ_*IP*_АДРЕС]. Если вы видите стандартную страницу приветствия *NGINX*, сервер работает корректно (рис. 10).

Рис. 10. Корректная работа *NGINX*

Настройка конфигурации *NGINX* осуществляется через файлы в каталоге */etc/nginx*. Основной конфигурационный файл – это */etc/nginx/nginx.conf*, но для управления отдельными сайтами чаще используются файлы в каталоге */etc/nginx/sites-available*, которые затем символически связываются с каталогом */etc/nginx/sites-enabled*.

Для связи с *PHP-FPM* необходимо настроить блок сервера в конфигурационном файле *NGINX*, который будет передавать запросы *PHP* к *PHP-FPM*. Это делается с помощью директивы *location*. Пример блока сервера, который настраивает обработку *PHP*, приведен на рис. 11.

В этом примере *fastcgi_pass* указывает на сокет, который использует *PHP-FPM*. Путь к сокету зависит от версии *PHP* и конфигурации *PHP-FPM*. Обратите внимание, что версия *PHP* в пути к сокету (*php7.4-fpm.sock*) должна соответствовать установленной версии *PHP-FPM* на вашем сервере.

После внесения изменений в конфигурацию необходимо перезапустить *NGINX* командой *sudo systemctl restart nginx* для применения новых настроек.

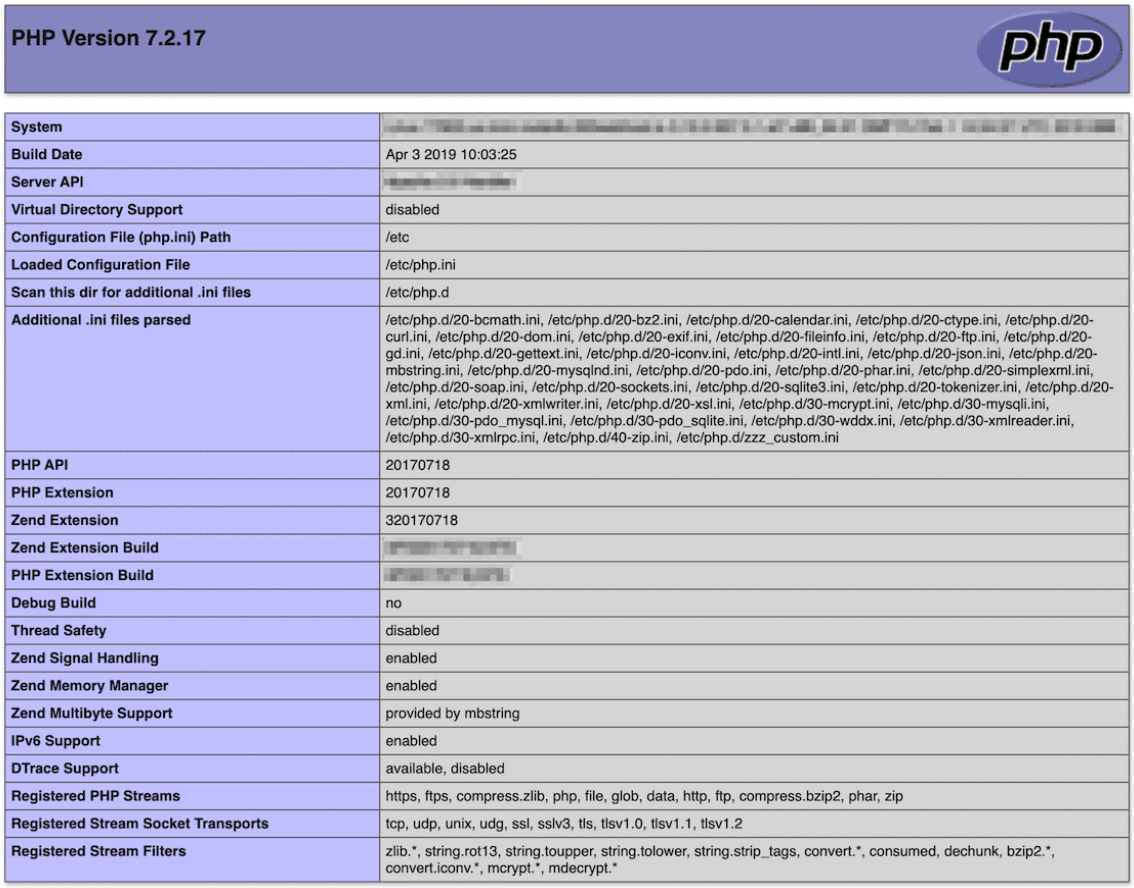
```
server {
    listen 80;
    server_name example.com www.example.com;

    root /var/www/html;
    index index.php index.html index.htm;

    location / {
        try_files $uri $uri/ =404;
    }

    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/run/php/php7.4-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

Рис. 11. Настройка связки NGINX с *PHP-FPM*



PHP Version 7.2.17

System	[redacted]
Build Date	Apr 3 2019 10:03:25
Server API	[redacted]
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini
Scan this dir for additional .ini files	/etc/php.d
Additional .ini files parsed	/etc/php.d/20-bcmath.ini, /etc/php.d/20-bz2.ini, /etc/php.d/20-calendar.ini, /etc/php.d/20-ctype.ini, /etc/php.d/20-curl.ini, /etc/php.d/20-dom.ini, /etc/php.d/20-exif.ini, /etc/php.d/20-fileinfo.ini, /etc/php.d/20-ftp.ini, /etc/php.d/20-gd.ini, /etc/php.d/20-gettext.ini, /etc/php.d/20-iconv.ini, /etc/php.d/20-intl.ini, /etc/php.d/20-json.ini, /etc/php.d/20-mbstring.ini, /etc/php.d/20-mysqlnd.ini, /etc/php.d/20-pdo.ini, /etc/php.d/20-phar.ini, /etc/php.d/20-simplexml.ini, /etc/php.d/20-soap.ini, /etc/php.d/20-sockets.ini, /etc/php.d/20-sqlite3.ini, /etc/php.d/20-tokenizer.ini, /etc/php.d/20-xml.ini, /etc/php.d/20-xmlwriter.ini, /etc/php.d/20-xsl.ini, /etc/php.d/30-mcrypt.ini, /etc/php.d/30-mysqli.ini, /etc/php.d/30-pdo_mysql.ini, /etc/php.d/30-pdo_sqlite.ini, /etc/php.d/30-wddx.ini, /etc/php.d/30-xmlreader.ini, /etc/php.d/30-xmlrpc.ini, /etc/php.d/40-zip.ini, /etc/php.d/zzz_custom.ini
PHP API	20170718
PHP Extension	20170718
Zend Extension	320170718
Zend Extension Build	[redacted]
PHP Extension Build	[redacted]
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	enabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
IPv6 Support	enabled
DTrace Support	available, disabled
Registered PHP Streams	https, ftps, compress.zlib, php, file, glob, data, http, ftp, compress.bzip2, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, tls, tlsv1.0, tlsv1.1, tlsv1.2
Registered Stream Filters	zlib.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk, bzip2.*, convert.iconv.*, mcrypt.*, mdecrypt.*

Рис. 12. *phpinfo()*

Теперь *NGINX* настроен на работу с *PHP-FPM*, и при обращении к *PHP*-скриптам на вашем веб-сервере запросы корректно перенаправляются на *PHP-FPM* для обработки. Чтобы убедиться, что все работает, можно создать простой *PHP*-файл *info.php* с функцией *phpinfo()* в корневой директории веб-сервера и обратиться к нему через браузер. Если конфигурация верна, в браузере отобразится страница с информацией о текущей конфигурации *PHP* (рис. 12).

Настройка веб-сервера *Apache* также начинается с его установки:

sudo apt install apache2

После установки *Apache* запускается автоматически. Чтобы проверить его работу, можно открыть веб-браузер и ввести адрес *http*://[ВАШ_*IP*_АДРЕС]. Если вы видите стандартную страницу *Apache*, значит, веб-сервер работает корректно (рис. 13).

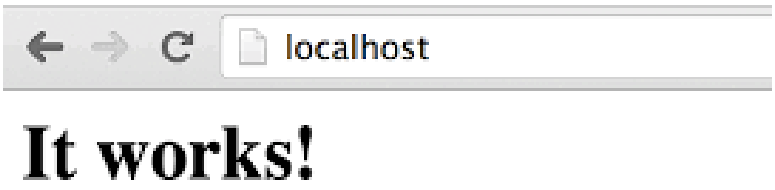


Рис. 13. Корректная работа *Apache*

Для настройки *Apache* используется каталог */etc/apache2*. Основной конфигурационный файл веб-сервера находится в /*etc/apache2/apache2.conf*, а дополнительные конфигурации виртуальных хостов находятся в каталогах */etc/apache2/sites-available* и /*etc/apache2/sites-enabled*. Для активации конфигурации виртуального хоста используется команда *a2ensit*e.

Связывание *Apache* с *PHP-FPM* требует дополнительной настройки. Во-первых, необходимо установить модуль *proxy_fcgi* и *php-fpm*:

sudo apt install php-fpm

sudo a2enmod proxy_fcgi

Далее, вам нужно настроить виртуальный хост для работы с *PHP-FPM*. Пример конфигурации приведен на рис. 14.

```
<VirtualHost *:80>
    ServerName example.com
    DocumentRoot /var/www/html

    <Directory /var/www/html>
        Options -Indexes +FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>

    <FilesMatch \.php$>
        SetHandler "proxy:unix:/run/php/php7.4-fpm.sock|fcgi://localhost/"
    </FilesMatch>
</VirtualHost>
```

Рис. 14. Настройка связки *Apache* с *PHP-FPM*

Здесь директива <*FilesMatch*> настраивает *Apache* для перенаправления всех запросов к *PHP*-файлам на *PHP-FPM* через *Unix*-сокет. Подобно *NGINX*, путь к сокету зависит от вашей версии *PHP-FPM*.

После внесения изменений в конфигурационный файл необходимо перезагрузить *Apache*, чтобы применить изменения:

sudo systemctl restart apache2

Теперь *Apache* настроен и готов обрабатывать *PHP*-запросы через *PHP-FPM*.

Apache Tomcat является менее популярным в современной веб-разработке по сравнению с *NGINX* и *Apache HTTP Server*, но он остается важным сервером приложений, специализирующимся на запуске *Java*-приложений, в частности тех, которые используют *Java Servlet* и *JavaServer Pages* (*JSP*). Его настройка и использование требуют определенного внимания, особенно если речь идет о запуске *Java*-базированных веб-приложений.

Для установки *Tomcat* можно использовать следующие команду:

sudo apt install tomcat9 tomcat9-admin

После установки *Tomcat* автоматически запустится и будет слушать на порту 8080. Чтобы проверить его статус, можно использовать

sudo systemctl status tomcat9

Или открыть в браузере http://[ВАШ_*IP*_АДРЕС]:8080, можно увидеть стартовую страницу *Tomcat* (рис. 15).

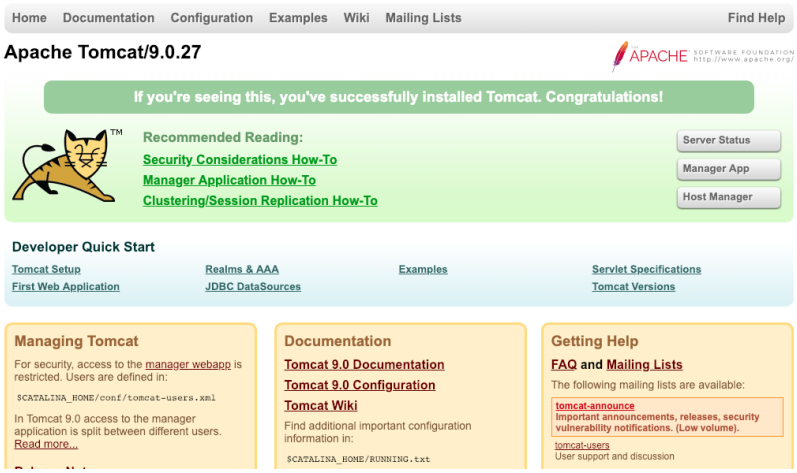


Рис. 15. Корректная работа *Tomcat*

Для более детальной настройки *Tomcat* основные конфигурационные файлы расположены в */etc/tomcat9*. Например, *server.xml* – это файл, в котором вы можете настроить сетевые порты, к которым *Tomcat* будет привязан, а также другие аспекты поведения сервера.

Конфигурация безопасности, включая пользователей и роли, находится в файле /*etc/tomcat9/tomcat-users.xml*, где вы можете определить пользователей, которые могут входить в систему для управления приложениями через веб-интерфейс *Tomcat*.

Tomcat также использует файлы *context.xml* для настройки контекста, т.е. окружения, в котором запускаются *Java*-приложения.

Для развертывания *Java*-веб-приложения в *Tomcat* обычно достаточно скопировать *WAR*-файл (*Web Application Archive*) в директорию */var/lib/tomcat9/webapps*. *Tomcat* автоматически распакует его и запустит приложение.

После изменения конфигурации или развертывания нового приложения вам потребуется перезагрузить *Tomcat:*

sudo systemctl restart tomcat9

Apache HTTP Server, NGINX и *Apache Tomcat* представляют собой три основных сервера приложений, которые часто используются в разработке веб-*проектов.*

Apache HTTP Server – это один из самых старых и надежных веб-серверов. Он тесно интегрирован с *PHP* через модуль *mod_php*, что делает его простым в использовании для *PHP*-разработчиков, особенно для тех, кто предпочитает традиционную стек-технологию *LAMP* (*Linux, Apache, MySQL, PHP*). *NGINX* – это более современный веб-сервер, известный своей высокой производительностью и масштабируемостью. Он часто используется для обслуживания статического контента и в качестве обратного прокси-сервера для *Apache* или других серверов приложений. *NGINX* работает с *PHP* через *FastCGI*, используя *PHP-FPM* (*FastCGI Process Manager*), что обеспечивает лучшую производительность и масштабируемость по сравнению с традици-

онным *mod_php*. *Apache Tomcat* – это контейнер сервлетов для *Java*-приложений, который не поддерживает *PHP* напрямую, поскольку он ориентирован на работу с *Java Servlet* и *JSP*. Однако *PHP* может быть использован на *Tomcat* через *Quercus*, *Java*-реализацию *PHP*, или *Tomcat* может быть сконфигурирован для работы в комбинации с другим веб-сервером, который поддерживает *PHP*.

В контексте облачных вычислений, где масштабируемость и управление ресурсами являются ключевыми, *NGINX* с *PHP-FPM* часто является предпочтительным выбором из-за его асинхронной архитектуры и эффективности работы с высоким числом одновременных соединений. *Apache* остается популярным благодаря своей гибкости и широкому сообществу, в то время как *Tomcat* предпочитают для специфических задач, связанных с *Java*.

4. МАСШТАБИРОВАНИЕ И БЕЗОПАСНОСТЬ

4.1. МАСШТАБИРОВАНИЕ ОБЛАЧНЫХ СЕРВИСОВ

Современные вычислительные системы, в том числе и облачные, постоянно сталкиваются с повышенными нагрузками. Чтобы удовлетворить потребности всех пользователей, они вынуждены постоянно расширяться. Достигается это видоизменением базовой архитектуры. Она постепенно становится более сложной и разветвленной. Применяются новые программные ресурсы, позволяющие корректировать производительность под потребительские запросы. В расширении нуждаются и серверы. Один из наиболее эффективных способов, позволяющих реализовать все это на практике, – масштабирование.

Масштабируемость облака – наиболее целесообразный способ повышения производительности системы путем добавления вычислительных ресурсов. Речь идет как об аппаратных, так и о программных изменениях, дополнениях, расширениях. Чтобы реализовать это на практике, требуется переписать существующий код. Но далеко не каждый бизнес решается на столь кардинальные перемены. Они просто прекращают наращивать аппаратную структуру и вносить изменения в серверную систему, которая в данный момент времени обеспечивает их потребности.

Многие представители бизнеса уверены, что к масштабированию стоит прибегать тогда, когда общая производительность облачной системы оказывается недостаточной для эффективного ведения рабочих процессов. Но это не совсем так. Даже при стабильной и эффективно работающей архитектуре, возможны сбои, вызванные повышением пользовательского трафика. А от этого не застрахована ни одна компания.

Проверить, насколько структура стойкая к перегрузкам, можно специальными утилитами-тестерами нагрузки. Они искусственно формируют повышенный поток пользователей на сервер, на порядок увеличивая запросы. После запуска приложения в работу необходимо оценить два параметра:

общую численность запросов и количество запросов, поступающих одновременно.

На основании этих данных определяется количество запросов за 1 секунду – *RPS*. Это показатель, позволяющий узнать число запросов, которые сможет обработать серверная система вашего бизнеса в определенный момент времени. Так вы узнаете, при каком одновременном наплыве пользователей ваш сервер «рухнет». Останется только оценить возможности компании. Если такая картина окажется вполне реальной, то уже в ближайшее время необходимо задуматься о видоизменении архитектуры настолько, чтобы ее можно было легко и быстро масштабировать параллельно с появлением такой необходимости. К такому решению стоит прибегнуть в случае, когда не представляется возможным изменение конфигурации серверов и оптимизация процедур обналичивания.

Обеспечить масштабирование облачных ресурсов можно с помощью:

1. Расширения по вертикали. Вертикальное масштабирование на практике реализуется путем увеличения общей мощности бизнес-процессов через повышение эффективности работы внутренних серверных ресурсов. Речь идет об аппаратных решениях: процессоры, диски, память, емкость сети. При этом сам сервер остается неизменным в своей базовой форме. Наращивание идет вверх, т.е. по вертикали.

2. Расширения по горизонтали. В том случае, когда трафик и цифровая нагрузка растут постоянно и с достаточно высокой скоростью, масштабирование по вертикали обречено на провал. Текущая спецификация базового сервера не может корректно отреагировать на такое увеличение запросов, что станет причиной появления ограничений и проблем в работе. В таком случае стоит использовать наращивание мощности по горизонтали. То есть горизонтальное масштабирование – это повышение вычислительных ресурсов добавлением дополнительных идентичных узлов поверх работающей архитектуры. Чтобы такая задумка сработала и дала желаемый эффект, необходимо иметь стабильно работающую серверную инфраструктуру,

т.е. центр обработки данных, и выстроить согласованную схему взаимодействия между отдельными серверами.

3. Расширения по диагонали. Наиболее инновационный уровень сервиса, которым пользуются в случае, когда даже горизонтальное масштабирование не позволяет охватить все пользовательские запросы и обеспечить стабильное протекание внутренних бизнес-процессов. При диагональном расширении одновременно выполняется увеличение с двух сторон: горизонтальное и вертикальное масштабирование. На практике вертикально наращиваются горизонтально увеличенные узлы в инфраструктуре сервера.

Чтобы определить оптимальный способ масштабирования, необходимо оценить потребности бизнеса в перспективе.

Виды масштабирования приведены на рис. 16.

После того, как будет выполнена комплексная оценка бизнеса и определен оптимальный способ масштабирования, можно переходить к этапу практической реализации. И здесь важно не только достичь ожидаемых результатов, но и сделать это максимально правильно и с минимально

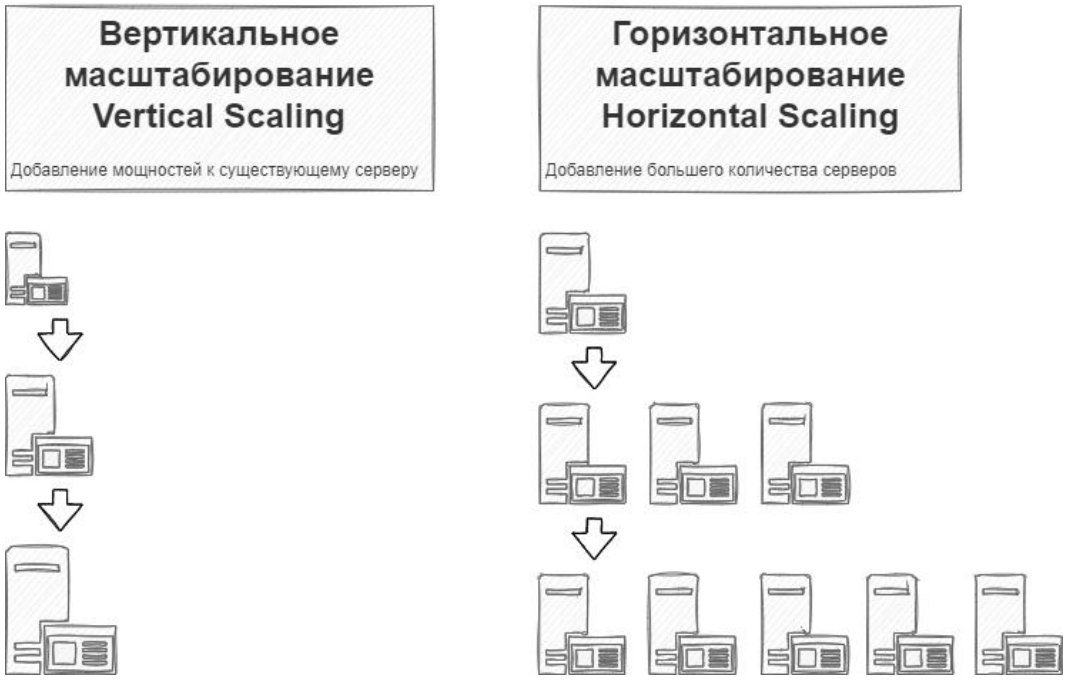


Рис. 16. Виды масштабирования

возможными тратами времени, усилий и денег. Обеспечить эффективное масштабирование поможет:

1. Достижение баланса нагрузки.
2. Автоматическое масштабирование.
3. Кластеризация.
4. Применение кеширования.
5. Привлечение *CDN*.

В каждом кластере, сокете, сосредоточено несколько аппаратно-программных ресурсов. Речь идет как о самих процессорах и дисках, так и о серверах. И если равномерно распределить нагрузку вычислительной сети между всеми ими, будет достигнут баланс, способный гарантировать:

– оптимизацию расходов вычислительных мощностей;

– снижение временных трат на обработку входящих пользовательских запросов;

– повышение производительности системы;

– минимизацию вероятности отклика сервера на *DDoS*-атаки;

– повышение доступности услуг вашей компании для пользователей.

Если обеспечить равномерное распределение нагрузки между несколькими узлами, то в случае выхода из строя одного из них его нагрузка автоматически будет распределена между другими аппаратно-программными ресурсами, что предотвратит «падение» системы.

Обеспечить балансировку позволяют специальные методы и алгоритмы. На примере модели *OSI*, существующие методики должны обладать несколькими уровнями: Сеть, Транспорт, Приложение. Реализовать это на практике позволяют несколько реальных серверов, оснащенных специализированным программным обеспечением.

Автоматическое масштабирование предполагает самостоятельную настройку системой вычислительных мощностей исходя из объема нагрузки на сеть. Это один из вариантов динамического масштабирования, реализованного на базе облачных серверов. На сегодня наиболее известными поль-

зователями, применяющими на практике возможности автоматического масштабирования, являются корпорации *Google Cloud Platform, Amazon Web Service, Microsoft Azure*. Если в масштабировании возникает необходимость, пользователям их служб выделяются дополнительные виртуальные серверы, которые в автоматическом порядке выводятся из контейнера или кластера. Как только ситуация стабилизируется, интенсивность запросов и трафик возвращаются к изначальному уровню, система сама выполняет обратный процесс, отключая от работы дополнительные мощности.

Автоматическое масштабирование позволяет бизнесу обеспечивать максимально высокую доступность сервисов для пользователей. Исключаются отказы в работе, достигается ощутимая экономия денежных ресурсов. Компания получает столько серверной мощности, сколько ей надо в определенный рабочий момент. При этом не требуется покупка дополнительного оборудования, проведение программных преобразований. А это одно из наиболее весомых преимуществ виртуального масштабирования перед физическим.

Кластеризация или использование контейнеров микросервисов – один из ресурсоэффективных методов повышения производительности системы. Технология предполагает объединение отдельных серверов в контейнеры, которые в последующем объединяются в кластеры. После реализации кластеризации необходимо определить сценарии будущей работы:

1. Добавлять ресурсы, которых не хватает для удовлетворения потребностей бизнеса.

2. Уменьшать выделение дополнительных ресурсов, чтобы исключить их избыток.

В случае горизонтального масштабирования не представляется возможным одновременно организовать простое кеширование памяти для нескольких компонентов. В этом случае выполняется оптимизация. Как пример, можно использовать хранилища Memcached либо Redis, чтобы равно-

мерно распределить информацию кеша между рабочими циклами программы. Учитывая тот факт, что инструменты в работе используют принципиально разные алгоритмы, объем данных, которые следует подвергнуть кешированию, постепенно будет уменьшаться. К тому же подобные хранилища имеют повышенную защиту от ошибок в процессе репликации и хранения информации.

Одна из основных проблем, с которыми можно столкнуться в процессе работы с кеш-хранилищами, – одновременный запрос некешированной информации от разных итераций программы. Предотвратить такое явление поможет обновление потоковых данных отдельно от производительности программы и использование кеширования внутри нее. При таком подходе вы получите инструмент виртуального масштабирования, эффективно работающий с большими объемами нагрузок, обеспечивая при этом отличную мощность.

CDN – это сеть физического аппаратного оснащения, удаленного друг от друга, предназначенного для передачи контента пользователям. То есть это распределенное хранилище, использующее кеширование. На практике такие системы получили применение в случае обслуживания интернет-службами, сайтами или программными комплексами пользователей с разных стран. Чем больший трафик будет проходить через сервис, тем более высокой будет и стоимость услуг по привлечению *CDN*.

Такие решения нецелесообразно использовать в случае, если трафик, хоть и распределяется по территории с большой площадью, но имеет несколько локальных концентраций пользователей. Как пример: 50% трафика приходится на Россию, 40% – на Беларусь, а оставшиеся 10% распределены по другим странам СНГ. Чтобы удовлетворить запросы такой сети, требуется установить дополнительные серверы в России и Беларуси, в то время как для последних 10% пользователей применить *CDN*.

В заключение стоит отметить, что успешное масштабирование облачных систем – это комплексная задача, требующая стратегического подхода.

Прежде всего, необходим тщательный анализ потребностей бизнеса и прогнозирование роста нагрузок. На этой основе вырабатывается стратегия масштабирования с учетом имеющихся технических и финансовых ресурсов. Сам процесс масштабирования должен быть поэтапным и гибким. Сначала оптимизируются существующие мощности, затем при необходимости добавляются новые ресурсы. Важно сохранять баланс производительности и затрат. Необходимо также предусмотреть резервирование критически важных компонентов и обеспечить отказоустойчивость на всех уровнях инфраструктуры. Мониторинг системы позволит своевременно обнаруживать узкие места.

Грамотный подход к масштабированию поможет компаниям эффективно использовать преимущества облачных технологий. Облака открывают поистине безграничные возможности для бизнеса любого масштаба.

4.2. БЕЗОПАСНОСТЬ ОБЛАЧНЫХ СЕРВИСОВ

Безопасность облака – это раздел кибербезопасности, посвященный защите облачных вычислительных систем. Сюда входит защита конфиденциальности и данных во всех объектах сетевой инфраструктуры, онлайн-приложениях и платформах. Участвовать в этом должны как поставщики облачных услуг, так и пользователи, будь то частные лица, малые и средние предприятия или корпорации.

Облачные службы размещаются на серверах с постоянным подключением к Интернету. Поставщики рассчитывают на доверие пользователей, поэтому в их интересах обеспечивать неприкосновенность хранящихся в облаке личных данных. Тем не менее облачная безопасность отчасти находится в руках самих пользователей. Для надежной защиты важно, чтобы обе стороны понимали свою ответственность.

Безопасность облака – это следующая совокупность категорий:

– безопасность данных;

– управление идентификацией и доступом *(IAM*);

– административный контроль (политика предотвращения, обнаружения и устранения угроз);

– планирование хранения данных и обеспечение непрерывности бизнеса;

– соблюдение нормативно-правовых требований.

На первый взгляд может показаться, что для обеспечения безопасности в облаке подходят те же методы, что и в традиционных IT-средах, но это не так. Прежде чем углубляться в тему, давайте разберемся, что же такое облачная безопасность.

Безопасность облака – это целый набор технологий, протоколов и наработок для защиты облачных сред, приложений и данных. Для начала необходимо понять, что именно нужно защищать и какие аспекты систем требуют управления.

В целом борьба с уязвимостями происходит преимущественно на серверной стороне: это обязанность поставщика облачных услуг. Но и у клиентов есть свои обязанности, помимо выбора надежного поставщика. Клиенты должны правильно использовать настройки защиты, уметь безопасно пользоваться службами, а также заботиться о защите всех устройств и сетей конечных пользователей.

Безопасность данных – это аспект облачной безопасности, связанный с технической стороной предотвращения угроз. Технологии позволяют поставщикам и клиентам делать конфиденциальные данные невидимыми и недоступными. Самая мощная из доступных технологий такого рода – шифрование. Шифрование делает ваши данные полностью нечитаемыми, и восстановить их сможет только тот, у кого есть ключ шифрования. Даже если ваши зашифрованные данные будут украдены, воспользоваться ими не получится.

В облачных сетях также важны средства защиты передаваемых данных, такие как виртуальные частные сети (*VPN*).

Управление идентификацией и доступом (*IAM*) связано с правами доступа, предоставляемыми пользователям. Сюда также относится управление аутентификацией и авторизацией учетных записей. Контроль доступа позволяет ограничить пользователям доступ к закрытым данным и системам и распространяется как на проверенных пользователей, так и на потенциальных злоумышленников. Управление паролями, многофакторная аутентификация – эти и другие методы защиты относятся к *IAM*.

Административный контроль сосредоточен на политике предотвращения, обнаружения и устранения угроз. Такие подходы, как анализ угроз, могут помочь предприятиям разных размеров в отслеживании и приоритизации угроз, чтобы защитить важнейшие системы. Индивидуальным клиентам тоже не помешает знать, как безопасно пользоваться облачными службами. Обычно это касается организаций, однако правила безопасного пользования системой и реагирования на угрозы пригодятся любому пользователю.

Планирование хранения данных и обеспечение непрерывности бизнеса включают меры восстановления утерянных данных в случае технического сбоя. При планировании опираются на методы дублирования информации, например на создание резервных копий. Кроме того, не помешают технические средства обеспечения бесперебойной работы. Хороший план обеспечения непрерывности бизнеса должен также включать проверку действительности резервных копий и подробные инструкции по восстановлению данных для сотрудников.

Соблюдение нормативно-правовых требований гарантирует защиту конфиденциальных данных пользователей в соответствии с законом. Государство заботится о том, чтобы личные данные людей не использовались в коммерческих целях, обязывая организации соблюдать установленные тре-

бования, например маскировать данные, т.е. использовать шифрование для скрытия личности пользователя.

С переходом на облачные вычисления традиционный подход к IT-безопасности претерпел огромные изменения. Облачные среды удобнее, однако постоянное подключение к Интернету требует новых мер безопасности. Безопасность облака как более современное решение в сфере кибербезопасности отличается от традиционных подходов рядом аспектов.

Главное отличие в том, что более ранние модели IT полагались на локальное хранение данных. Тем не менее, несмотря на возможность полноценного контроля безопасности, локальные IT-платформы дороги и не отличаются гибкостью. Облачные платформы помогают сэкономить на разработке и эксплуатации систем, но при этом частично лишают пользователей контроля.

Аналогичным образом безопасность облака требует особого внимания при масштабировании корпоративных IT-систем. В облаке используется модульная инфраструктура и приложения с возможностью быстрой мобилизации. Это облегчает адаптацию системы к организационным переменам, однако, вследствие потребности организации в постоянных обновлениях и повышении удобства работы, постоянно приходится задумываться об уровне безопасности.

Облачные системы взаимодействуют со многими другими системами и службами, которые также необходимо защищать, причем это справедливо как для организаций, так и для индивидуальных пользователей. Необходимо управление правами доступа на всех уровнях: на устройствах конечных пользователей, для ПО и даже в сети. Кроме того, поставщикам и пользователям нужно отслеживать уязвимости, возникающие из-за небезопасных установок приложений и доступа к системам.

Постоянная связь между облаком и пользователями создает угрозу даже для поставщика облачных услуг. В сетевой среде один единственный

уязвимый компонент может стать брешью для компрометации всей системы. Предоставляя клиентам услуги, включая хранение данных, поставщики облачных услуг постоянно подвергаются опасности. Сохраняя данные на собственные системы вместо систем конечных пользователей, поставщики вынуждены принимать дополнительные меры безопасности.

Для решения большинства проблем облачной безопасности – как в персональной, так и в деловой среде – требуется проактивное участие и клиентов, и поставщиков. Это означает, что и те, и другие равным образом должны уделять внимание:

– безопасной настройке и обслуживанию систем;

– обучению пользователей безопасному поведению и техническим мерам безопасности.

Наконец, и от тех, и от других требуется прозрачность и ответственность как гарантия безопасности обеих сторон.

Для облака главным риском является отсутствие периметра. Традиционная киберзащита в первую очередь направлена на обеспечение безопасности периметра, но облачные среды очень тесно взаимосвязаны, а значит, небезопасные *API* (интерфейсы программирования приложений) и кража учетных записей представляют серьезную опасность. Учитывая специфику рисков, специалисты по кибербезопасности теперь должны делать упор именно на контроль данных.

Взаимосвязанность также представляет проблему для сетей. Часто преступники проникают в сеть через взломанную или незащищенную учетную запись. Если злоумышленник получит доступ к облаку, он сможет воспользоваться его плохо защищенными интерфейсами, чтобы заполучить нужные данные из различных баз данных или узлов. Более того, он даже может использовать свои собственные облачные серверы для экспортирования и хранения похищенных данных. Система безопасности должна защищать все облако, а не только хранящиеся в нем личные данные.

Сторонние услуги хранения данных и онлайн-доступ также представляют угрозу. Если в работе какой-либо службы произойдет сбой, вы не сможете получить доступ к своим файлам. Например, в случае перегрузки мобильной сети вы можете в самый неподходящий момент оказаться без доступа к облаку. Или отключение электроэнергии может затронуть центр обработки данных, в котором хранятся ваши данные, и даже привести к их безвозвратной потере.

Подобные сбои могут иметь и долговременные последствия. Так, например, отключение электроэнергии в облачном хранилище данных Amazon привело к потере данных многих клиентов вследствие повреждения аппаратной части серверов. Этот пример наглядно показывает, почему в любом случае необходимы локальные резервные копии хотя бы части данных и приложений.

5. РАБОТА С GIT И GITHUB

5.1. ПОНЯТИЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

Контроль версий, также известный как управление исходным кодом, – это практика отслеживания изменений программного кода и управления ими. Системы контроля версий – это программные инструменты, помогающие командам разработчиков управлять изменениями в исходном коде с течением времени. В свете усложнения сред разработки они помогают командам разработчиков работать быстрее и эффективнее. Системы контроля версий наиболее полезны командам *DevOps*, поскольку помогают сократить время разработки и увеличить количество успешных развертываний.

Программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок, сводя к минимуму проблемы для всех участников команды.

Практически во всех программных проектах исходный код является сокровищем: это ценный ресурс, который необходимо беречь. Для большинства команд разработчиков программного обеспечения исходный код – это репозиторий бесценных знаний и понимания проблемной области, которые они скрупулезно собирали и совершенствовали. Контроль версий защищает исходный код от катастрофических сбоев, от случайных ухудшений, вызванных человеческим фактором, а также от непредвиденных последствий.

Разработчики программного обеспечения, работающие в командах, постоянно пишут новый исходный код и изменяют существующий. Код проекта, приложения или программного компонента обычно организован в виде структуры папок или «дерева файлов». Один разработчик в команде может работать над новой возможностью, а другой в это же время изменять код для исправления несвязанной ошибки, т.е. каждый разработчик может вносить свои изменения в несколько частей дерева файлов.

Контроль версий помогает командам решать подобные проблемы путем отслеживания каждого изменения, внесенного каждым участником, и предотвращать возникновение конфликтов при параллельной работе. Изменения, внесенные в одну часть программного обеспечения, могут быть не совместимы с изменениями, внесенными другим разработчиком, работавшим параллельно. Такая проблема должна быть обнаружена и решена согласно регламенту, не создавая препятствий для работы остальной части команды. Кроме того, во время разработки программного обеспечения любое изменение может само по себе привести к появлению новых ошибок, и новому ПО нельзя доверять до тех пор, пока оно не пройдет тестирование. Вот почему процессы тестирования и разработки идут рука об руку, пока новая версия не будет готова.

Хорошее программное обеспечение для управления версиями поддерживает предпочтительный рабочий процесс разработчика, не навязывая определенный способ работы. В идеале оно также работает на любой платформе и не принуждает разработчика использовать определенную операционную систему или цепочку инструментов. Хорошие системы управления версиями обеспечивают плавный и непрерывный процесс внесения изменений в код и не прибегают к громоздкому и неудобному механизму блокировки файлов, который дает зеленый свет одному разработчику, но при этом блокирует работу других.

Группы разработчиков программного обеспечения, не использующие какую-либо форму управления версиями, часто сталкиваются с такими проблемами, как незнание об изменениях, выполненных для пользователей, или создание в двух несвязанных частях работы изменений, которые оказываются несовместимыми и которые затем приходится скрупулезно распутывать и перерабатывать. Если вы как разработчик ранее никогда не применяли управление версиями, возможно, вы указывали версии своих файлов, добавляя суффиксы типа «финальный» или «последний», а позже появлялась новая финальная версия. Возможно, вы использовали комментирование блоков

кода, когда хотели отключить определенные возможности, не удаляя их, так как опасались, что этот код может понадобиться позже. Решением всех подобных проблем является управление версиями.

Программное обеспечение для управления версиями является неотъемлемой частью повседневной профессиональной практики современной команды разработчиков программного обеспечения. Отдельные разработчики ПО, привыкшие работать в команде с эффективной системой управления версиями, обычно признают невероятную пользу управления версиями даже при работе над небольшими сольными проектами. Привыкнув к мощным преимуществам систем контроля версий, многие разработчики не представляют, как работать без них даже в проектах, не связанных с разработкой ПО.

Программное обеспечение контроля версий рекомендуется для продуктивных команд разработчиков и команд *DevOps*. Управление версиями помогает отдельным разработчикам работать быстрее, а командам по разработке ПО – сохранять эффективность и гибкость по мере увеличения числа разработчиков.

За последние несколько десятилетий системы контроля версий (*Version Control Systems*, *VCS*) стали гораздо более совершенными, причем некоторым это удалось лучше других. Системы *VCS* иногда называют инструментами *SCM* (управления исходным кодом) или *RCS* (системой управления редакциями). Один из наиболее популярных на сегодняшний день инструментов *VCS* называется *Git*. *Git* относится к категории распределенных систем контроля версий, известных как *DVCS*. *Git*, как и многие другие популярные и доступные на сегодняшний день системы *VCS*, распространяется бесплатно и имеет открытый исходный код. Независимо от того, какую систему контроля версий вы используете и как она называется, основные ее преимущества заключаются в следующем.

1. Полная история изменений каждого файла за длительный период. Это касается всех изменений, внесенных огромным количеством людей за долгие годы. Изменением считается создание и удаление файлов, а также ре-

дактирование их содержимого. Различные инструменты *VCS* отличаются тем, насколько хорошо они обрабатывают операции переименования и перемещения файлов. В историю также должны входить сведения об авторе, дата и комментарий с описанием цели каждого изменения. Наличие полной истории позволяет возвращаться к предыдущим версиям, чтобы проводить анализ основных причин возникновения ошибок и устранять проблемы в старых версиях программного обеспечения. Если над программным обеспечением ведется активная работа, то «старой версией» можно считать почти весь код этого ПО.

2. Ветвление и слияние. Эти возможности полезны не только при одновременной работе участников команды: отдельные сотрудники также могут пользоваться ими, занимаясь несколькими независимыми направлениями. Создание «веток» в инструментах *VCS* позволяет иметь несколько независимых друг от друга направлений разработки, а также выполнять их слияние, чтобы инженеры могли проверить, что изменения, внесенные в каждую из веток, не конфликтуют друг с другом. Многие команды разработчиков ПО создают отдельные ветки для каждой функциональной возможности, для каждого релиза либо и для того, и для другого. Имея множество различных рабочих процессов, команды могут выбирать подходящий для них способ ветвления и слияния в *VCS*.

3. Отслеживаемость. Возможность отслеживать каждое изменение, внесенное в программное обеспечение, и связывать его с ПО для управления проектами и отслеживания багов, например *Jira*, а также оставлять к каждому изменению комментарий с описанием цели и назначения изменения может помочь не только при анализе основных причин возникновения ошибок, но и при других операциях по исследованию. История с комментариями во время чтения кода помогает понять, для чего этот код нужен и почему он структурирован именно так. Благодаря этому разработчики могут вносить корректные и совместимые изменения в соответствии с долгосрочным планом разработки системы. Это особенно важно для эффективной работы

с унаследованным кодом, поскольку дает специалистам возможность точнее оценить объем дальнейших задач.

Разрабатывать программное обеспечение можно и без управления версиями, но такой подход подвергает проект огромному риску, и ни одна профессиональная команда не порекомендует применять его. Таким образом, вопрос заключается не в том, использовать ли управление версиями, а в том, какую систему управления версиями выбрать.

Среди множества существующих систем управления версиями мы сосредоточимся на одной: системе *Git*.

5.2. УСТАНОВКА И НАСТРОЙКА GIT

Git – абсолютный лидер по популярности среди современных систем управления версиями. Это развитый проект с активной поддержкой и открытым исходным кодом. Система *Git* была изначально разработана в 2005 году Линусом Торвальдсом – создателем ядра операционной системы *Linux*. *Git* применяется для управления версиями в рамках колоссального количества проектов по разработке ПО, как коммерческих, так и с открытым исходным кодом. Система используется множеством профессиональных разработчиков программного обеспечения. Она превосходно работает под управлением различных операционных систем и может применяться со множеством интегрированных сред разработки (*IDE*).

Git – система управления версиями с распределенной архитектурой. В отличие от некогда популярных систем вроде *CVS* и *Subversion* (*SVN*), где полная история версий проекта доступна лишь в одном месте, в *Git* каждая рабочая копия кода сама по себе является репозиторием. Это позволяет всем разработчикам хранить историю изменений в полном объеме.

Разработка в *Git* ориентирована на обеспечение высокой производительности, безопасности и гибкости распределенной системы.

Git показывает очень высокую производительность в сравнении со множеством альтернатив. Это возможно благодаря оптимизации процедур

фиксации коммитов, создания веток, слияния и сравнения предыдущих версий. Алгоритмы *Git* разработаны с учетом глубокого знания атрибутов, характерных для реальных деревьев файлов исходного кода, а также типичной динамики их изменений и последовательностей доступа.

При разработке в *Git* прежде всего обеспечивается целостность исходного кода под управлением системы. Содержимое файлов, а также объекты репозитория, фиксирующие взаимосвязи между файлами, каталогами, версиями, тегами и коммитами, защищены с помощью криптографически стойкого алгоритма хеширования *SHA*1. Он защищает код и историю изменений от случайных и злонамеренных модификаций, а также позволяет проследить историю в полном объеме.

Использование *Git* гарантирует подлинность истории изменений исходного кода.

Гибкость – одна из основных характеристик *Git*. Она проявляется в поддержке различных нелинейных циклов разработки, эффективности использования с малыми и крупными проектами, а также совместимости со многими системами и протоколами.

В отличие от *SVN*, система *Git* рассчитана прежде всего на создание веток и использование тегов. Поэтому процедуры с участием веток и тегов (например, объединение и возврат к предыдущей версии) сохраняются в истории изменений. Не все системы управления версиями обладают настолько широкими возможностями отслеживания.

Функциональность, производительность, безопасность и гибкость *Git* удовлетворяют требованиям большинства команд и разработчиков. Эти качества системы подробно описаны выше. При сравнении системы с большинством альтернатив многие команды приходят к выводу, что *Git* обладает значительными преимуществами.

Пакеты *Git* доступны через утилиту apt:

apt-get install git

Убедитесь, что установка прошла успешно, набрав команду

git --version

Настройте для *Git* имя пользователя и адрес электронной почты с помощью следующих команд (замените имя своим, эта информация будет связана со всеми созданными вами коммитами):

git config --global user.name "Ivan Ivanov "

git config --global user.email "mail@gmail.com "

5.3. ОСНОВНЫЕ КОМАНДЫ GIT

Работа с *Git* через терминал – это обязательная часть практики современного разработчика. Однако для начинающих разработчиков этот инструмент может показаться сложным.

Работа с любой программой всегда начинается с ее настройки. *Git* можно настроить один раз и менять что-то только по мере необходимости.

Указать имя пользователя – *git config --global user.name* "*Ivan Ivanov*". Задает имя пользователя, от которого будут идти коммиты. Вместо Ivan Ivanov нужно написать свои данные на латинице. Если имя состоит из одного слова, кавычки можно не ставить. Указать электронную почту – g*it config --global user.email "mail@gmail.com"*. Вместо *mail@gmail.com* нужно указать вашу почту. Посмотреть настройки – *git config --list*.

Инициализирует пустой репозиторий:

git init

Склонировать удаленный репозиторий. Проект появится в директории, где вы находились в момент клонирования:

git clone [ссылка на удаленный репозиторий]

Связать удаленный и локальный репозитории:

git remote add origin [ссылка на удаленный репозиторий]

Любая работа с изменениями начинается с получения последней версии проекта из удаленного репозитория. Далее вы можете внести правки

в проект, добавить изменения в индекс и сделать коммит. В конце нужно отправить изменения в удаленный репозиторий или удалить, если они больше не нужны.

Подтянуть изменения:

git pull

Подтягивает в локальный репозиторий последнюю версию проекта. Будьте внимательны, вызов этой команды сотрет все незафиксированные изменения. Иногда после ввода этой команды появляется конфликт.

Посмотреть статус файлов:

git status

Вы увидите, какие файлы изменили, удалили или добавили в проект. При этом статус «закоммичен» не отобразится.

Добавить файлы в индекс. После ввода этой команды вы можете сделать коммит:

git add [название файла]

Есть похожие команды, например *git add*, индексирует сразу все измененные файлы и папки в директории, где вы находитесь. Обратите внимание, между точкой и словом add нужно ставить пробел. Команда *git add* :/ добавляет в индекс все файлы независимо от того, в какой директории вы находитесь.

Сделать коммит (фиксация изменений). До выполнения этой команды локальные изменения никуда не запишутся:

git commit -m "Комментарий к коммиту"

Посмотреть историю коммитов:

git log

Она выводит список всех коммитов. У этой команды есть разные опции, самая используемая из них – *--oneline*. Она показывает хеш в укороченном формате, ветку, в которой сделан коммит, а также текст коммита. Чтобы использовать эту опцию (как и любую другую), нужно добавить ее после команды

git log--oneline

Отправить изменения:

git push

Команда отправляет все зафиксированные изменения с локального репозитория в удаленный. Это одна из самых важных команд, ведь все вышеописанные действия производятся в локальной копии репозитория. Когда вы закончите работу, эту копию нужно будет отправить в удаленный репозиторий. Только так другие участники процесса смогут получить актуальную версию.

Работая с *Git*, приходится постоянно создавать и перемещаться по веткам. А иногда ветки нужно удалять или сливать.

Создать ветку:

git switch --create branch-name

Команда добавляет новую ветку с названием *branch-name* и автоматически переключает на нее.

Переключить ветку (вы перейдете на уже созданную ветку *branch-name*):

git switch branch-name

Для создания и переключения веток также можно использовать *git checkout*. Эта команда появилась раньше, у нее есть множество дополнительных функций. Например, она может восстанавливать изменения в коммите. Как раз из-за такого разнообразия задач разработчики решили создать отдельную команду для переключения между ветками – *git switch*. Вы можете использовать любую из команд, однако *git switch* доступна только в версиях от 2.23.

Посмотреть все локальные ветки:

git branch

Переименовать ветку:

git branch -m [старое-название-ветки] [новое-название-ветки] –

переименовывает ветку

Отправить ветку (отправляет ветку в удаленный репозиторий):

git push origin [branch-name]

Удалить ветки:

git branch --delete [branch-name]

Команда удаляет ветку [*branch-name*] в локальном репозитории. Если нужно избавиться от ветки в удаленном репозитории, используйте

git push --delete origin [branch-name]

Влить ветки (вливает ветку *branch-name* в ветку, в которой вы находитесь):

git merge [branch-name]

Перебазировать коммиты (перебазирует коммиты из ветки, в которой вы находитесь, в ветку [*branch-name*]):

git rebase [branch-name]

Создать точную копию коммитов:

git cherry-pick

Команду часто совмещают с *git merge* и *git rebase*, чтобы сохранить линейную историю коммитов. То есть создается точная копия коммитов, выполняется перебазирование и слияние веток.

Отложить изменения:

git stash push

Откладывает изменения, чтобы вы, например, могли срочно перейти к другой задаче. Чтобы отложить только часть изменений, используйте *git stash --patch*.

Вернуть отложенные изменения:

git stash pop

Отменить изменения, не добавленные в индекс:

git restore [название файла]

Удалит изменения в одном файле. Чтобы удалить изменения во всех файлах, используйте:

git restore :/

Отменить изменения, добавленные в индекс (возвращает изменения из индекса и полностью их отменяет):

git reset--hard

Удалить коммит (вместо [*195dfb0*] указывается хеш коммита, его можно узнать с помощью команды *git log*):

git revert [195dfb0]

Отменить слияние с конфликтом (используется, когда нет времени решать конфликт прямо здесь и сейчас):

git merge --abort

Удалить неотслеживаемые файлы из рабочего каталога:

git clean

5.4. СТРАТЕГИИ ВЕТВЛЕНИЯ

Одной из самых популярных моделей ветвления в репозитории является модель *GitFlow* (рис. 17 – 19). Она предполагает выстраивание строгой модели ветвления вокруг релиза проекта, которая дает надежную схему управления крупными проектами. *Gitflow* отлично подходит для проектов, которые имеют спланированный цикл релиза.

Вместо использования одной ветки *master*, в этой модели используются две ветки для записи истории проекта. В ветке *master* хранится официальная история релиза, а ветка *develop* служит в качестве интеграционной ветки для новых функций. Также удобно тегировать все коммиты в ветке *master* номером версии.

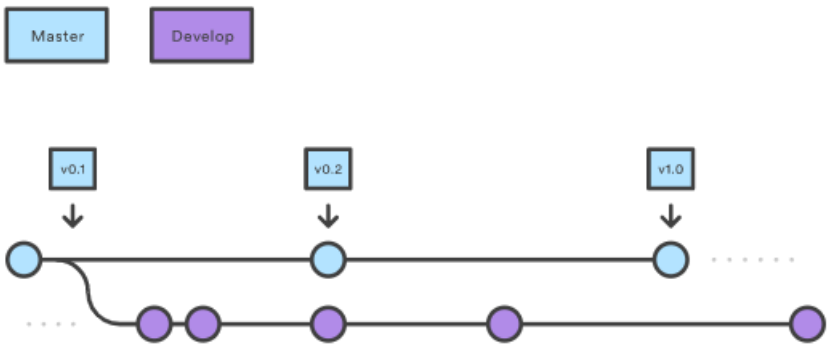


Рис. 17. Организация ветвления по *GitFlow*

Первым шагом является создание ветки *develop* от ветки *master*. Проще всего это сделать одному разработчику, локально создав пустую ветку и отправив ее в центральный репозиторий:

git branch develop

git push -u origin develop

В этой ветке будет находиться вся история проекта, в то время как *master* содержит частичную историю. Остальные разработчики теперь должны клонировать центральный репозиторий и создать отслеживающую ветку для ветки *develop*. Для перехода в ветку *develop*:

git checkout develop

Каждая новая функциональность должна разрабатываться в отдельной ветке, которую можно отправлять в репозиторий для создания резервной копии/для совместной работы команды. Ветки функций создаются не на основе *master*, а на основе *develop*. Когда работа над новой функциональностью завершена, она вливается назад в *develop*. Новый код не должен отправляться напрямую в *master*:

git checkout develop

git checkout -b feature_branch

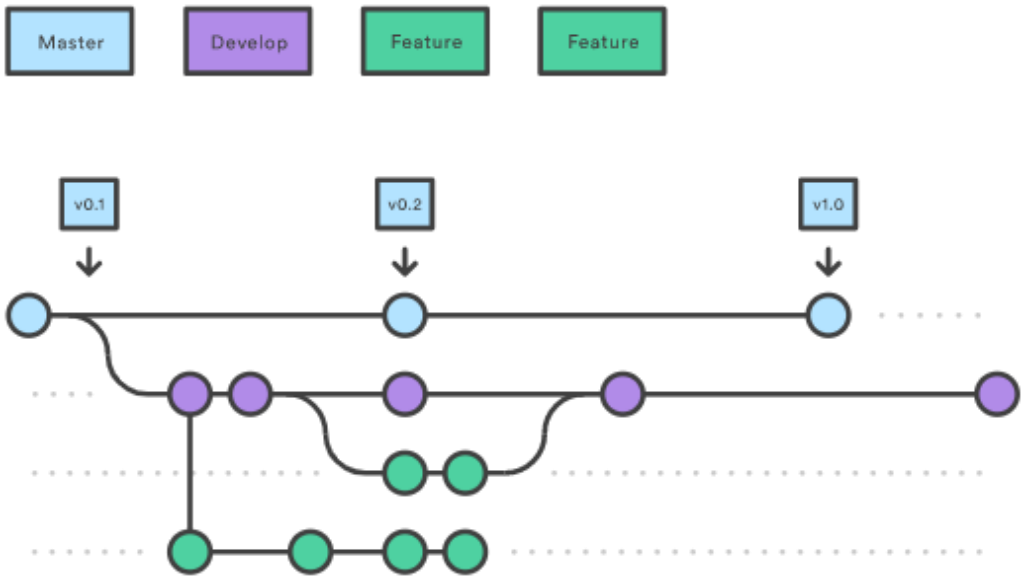


Рис. 18. Организация ветвления по *GitFlow*

Далее, продолжайте работу с *Git* как обычно. По окончании разработки новой функциональности следующим шагом следует объединить ветку *feature_branch с develop*. Используйте команды:

git checkout develop

git merge release/0.1.0

Ветки *hotfix* используются для быстрого внесения исправлений в рабочую версию кода. Ветки *hotfix* очень похожи на ветки *realese* и *feature*, за исключением того, что они созданы от *master*, а не от *develop*. Это единственная ветка, которая должна быть создана непосредственно от *master*. Как только исправление завершено, ветка *hotfix* должна быть объединена как с *master*, так и с *develop* (или с веткой текущего релиза), а *master* должен быть помечен обновленным номером версии.

Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла подготовки к релизу. Можно говорить о ветках *hotfix* как об особых ветках release, которые работают напрямую с *master*.

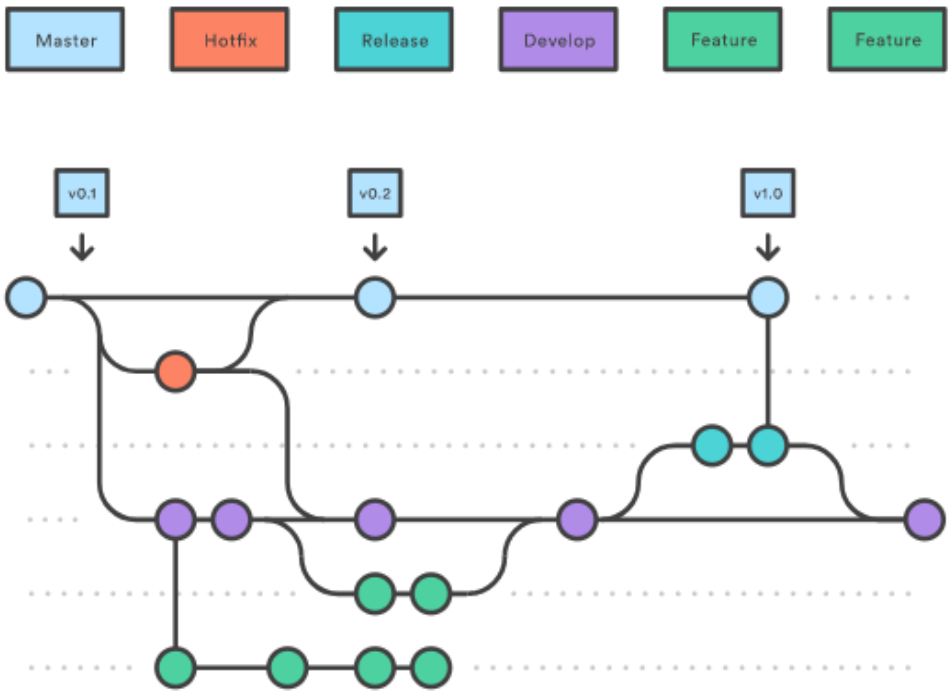


Рис. 19. Организация ветвления по *GitFlow*

git checkout master

git checkout -b hotfix_branch

Как и в работе с веткой *release*, ветка *hotfix* объединяется как с *master*, так и с *develop:*

git checkout master

git merge hotfix_branch

git checkout develop

git merge hotfix_branch

git branch -D hotfix_branch

$ git flow hotfix finish hotfix_branch

Пример команд, демонстрирующий полный цикл работы с веткой функции, выглядит следующим образом. Предположим, что у нас есть репозиторий с веткой *master*:

git checkout master

git checkout -b hotfix_branch

 # работа сделана, коммиты добавлены в hotfix_branch

git checkout develop

git merge hotfix_branch

git checkout master

git merge hotfix_branch

5.5. ИСПОЛЬЗОВАНИЕ GITHUB ДЛЯ РАБОТЫ В КОМАНДЕ

Разработчики программ используют в работе различные платформы для обмена исходным кодом, его хранения и распространения. Одной из таких платформ является *GitHub* (рис. 20).

Веб-сервис *GitHub* востребован для хостинга *IT*-проектов и совместной разработки. Разработчики системы называют ее «социальной сетью» для программистов. Здесь они объединяют репозитории, комментируют примеры «чужого» кода и используют платформу в качестве облачного хранилища с возможностью быстрой передачи заказчику.

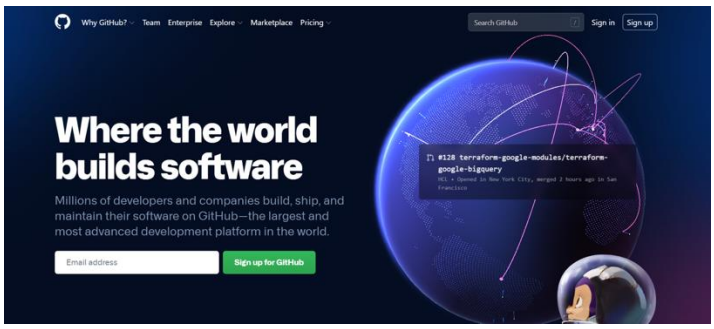


Рис. 20. *GitHub*

Первый шаг к использованию сервиса *GitHub* заключается в регистрации нового пользователя. В процедуре нет ничего сложного – достаточно зайти на официальный сайт *https://github.com/* и создать новую учетную запись. Система запросит рабочую электронную почту.

Пароль вводится на выбор пользователя, но с учетом правил. Так, рекомендуется комбинация размером в 15 символов или 8, но с использованием хотя бы одной цифры и строчной буквы. Имя пользователя, как и *email*, проверяется на занятость, и придется выбирать тот, с которым платформа позволит продолжать регистрацию.

Далее нужно указать, хочется ли получать новости об обновлениях продуктов и самой системы. Последним шагом становится подтверждение – пользователю предлагается собрать пазл, после чего станет активной кнопка «Зарегистрироваться».

Вход на платформу будет открыт только после подтверждения электронной почты, поэтому зайти анонимно не получится. Это своеобразная защита сервера от многочисленных ботов и гарантия для пользователей, что они будут общаться с реальными людьми. Теперь можно приступать к управлению настройками внутри личного кабинета.

Важно отметить, что сервис англоязычный, и пользоваться им без знания языка получится только при использовании обновленных версий браузеров типа *Google Chrome*, где есть встроенные функции по переводу страниц. В любом случае работа начинается с создания собственного репозитория – в бесплатном режиме доступны публичные, частные откроются только при активации платного тарифа (рис. 21).

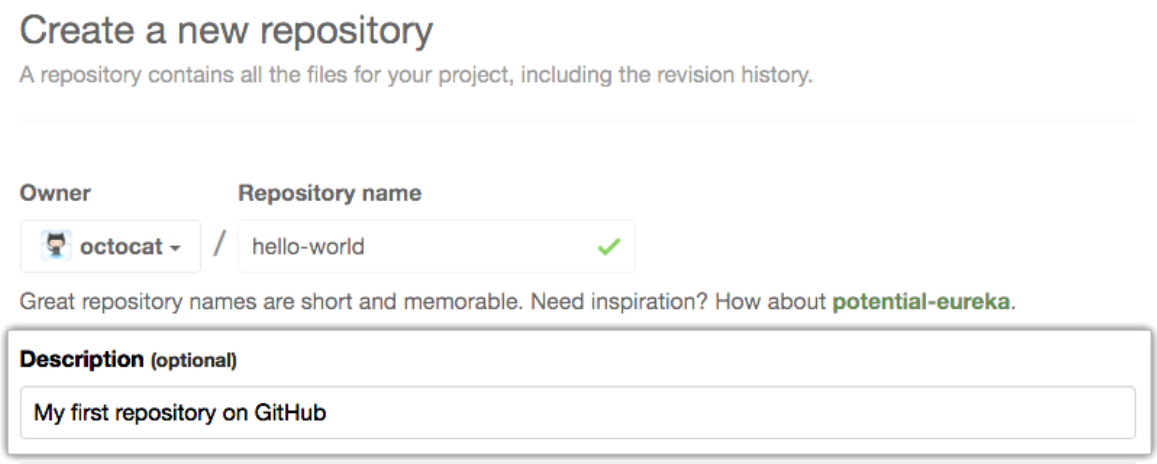


Рис. 21. Создание репозитория

Последовательность действий:

1. Нажать на кнопку «*Start a project*».
2. Ввести название и описание репозитория.
3. Поставить галочку на «*Initialize this repository with a README*».
4. Выбрать нужный тип лицензии и нажать на кнопку «*Create project*».

Тип лицензии (приватная или публичная) допускается заменить после, в процессе использования платформы. Единственная настройка, которую пользователи делают сразу, – это создание нескольких веток для размещения разных проектов. Например, для тестового кода и финальных релизов, чтобы не путать их при разработке и общении с другими кодерами.

Подобный подход часто используют создатели продуктов, которыми пользуются «массы». Им передается ссылка на проверенные стабильные версии, в то время как команда продолжает работу над таким же комплектом файлов без опасения нарушить функциональность системы в целом. При использовании платформы следует ориентироваться на отметку «*branch*».

Данная отметка обозначает текущую ветку. Создание новой инициируется просто – достаточно в списке начать набирать еще несуществующее название, и система выдаст сообщение «*Create branch*». Сразу после этого пользователь перекидывается в новую ветку (это стоит учитывать при работе, чтобы случайно не начать редактирование «не тех файлов»).

Корректировка файлов на *GitHub* выполняется с помощью коммитов. Это непосредственно само исправление и краткое описание изменений. Такой подход позволяет «внешним» пользователям ориентироваться в нововведениях кода и упрощает контроль командной работы, когда один и тот же файл может редактироваться разными исполнителями.

Система сохранения информации о корректировках удобна, когда они вносятся в различные участки кода, но связаны с определенной задачей. Фактически текстовый файл с описанием «связывает» разрозненные изменения и объясняет непосвященному программисту их суть, назначение. Чтобы запустить редактирование *README*, нужно в правой панели нажать на «кисточку».

После этого откроется текстовый редактор, где вносятся исправления. По завершении заполняется поле «*Commit*» внизу страницы (кратко, что изменилось) и нажимается кнопка «*Commit changes*». Сохраненные корректировки будут внесены в текущую (активную) ветку проекта, поэтому перед их внесением следует убедиться в правильном выборе (рис. 22).

Создав репозиторий, можно его клонировать на локальный компьютер и работать с ним локально через командную строку с помощью команд, рассмотренных ранее. *GitHub* в данном случае является лишь хостингом программного кода.

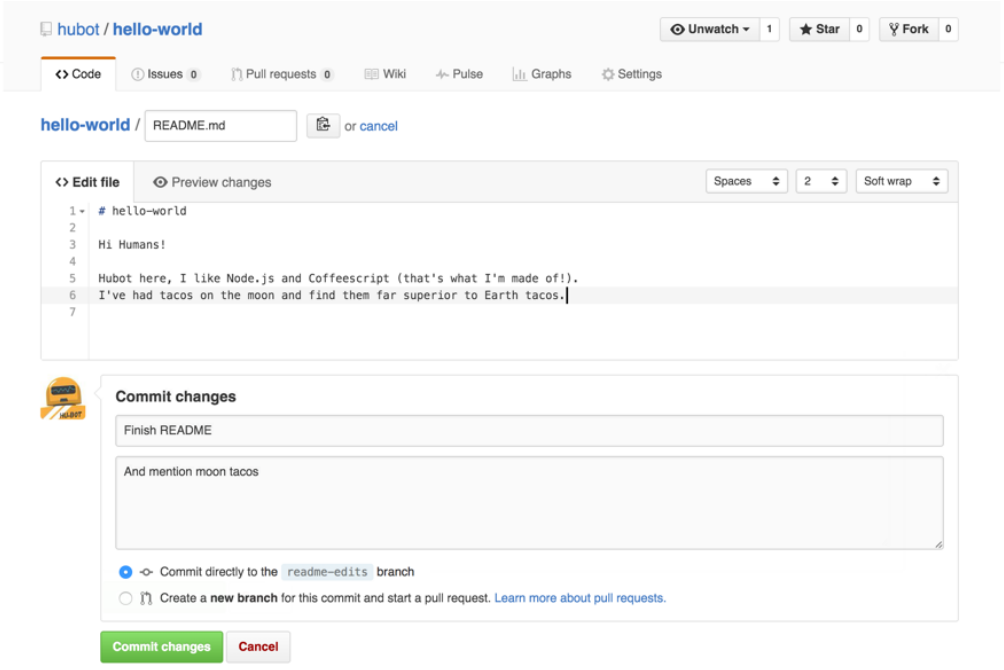


Рис. 22. Создание коммита

6. КОНТЕЙНЕРИЗАЦИЯ С DOCKER

6.1. ВВЕДЕНИЕ В КОНТЕЙНЕРИЗАЦИЮ

Контейнеризация – это легковесный метод виртуализации на уровне операционной системы, позволяющий запускать и управлять приложениями и их зависимостями в стандартизированных блоках, называемых контейнерами. Этот подход обеспечивает согласованность окружения независимо от инфраструктуры и упрощает развертывание приложений.

Представьте, что каждое приложение и его зависимости упакованы в прозрачный контейнер – подобно тому, как вы упаковываете обед в контейнер для еды, чтобы он оставался свежим и не смешивался с другими продуктами в вашем рюкзаке. Таким образом, контейнер с приложением может быть легко перемещен с одного компьютера на другой, будь то разработческая машина, тестовое окружение или продакшн-сервер, и приложение будет работать одинаково везде, поскольку все необходимое для его работы уже находится внутри контейнера.

Контейнер – это легковесный, автономный пакет, содержащий все необходимое для выполнения программы, включая код, среду выполнения, системные инструменты, системные библиотеки и настройки.

Docker – это одна из самых популярных платформ для контейнеризации, предоставляющая инструменты и сервисы для автоматизации развертывания приложений в контейнерах. *Docker* использует *Dockerfile*, файл с инструкциями для создания образа контейнера, который затем может быть запущен на любой системе с установленным *Docker*, обеспечивая тем самым консистентность и упрощая процессы *CI/CD* (*Continuous Integration/Continuous Deployment*). *Docker* обеспечивает изоляцию и безопасность приложений, позволяет быстро масштабировать и обновлять компоненты без влияния на другие части системы, что делает его идеальным инструментом для разработки современных микросервисных архитектур.

Архитектура Docker использует клиент-серверную модель и состоит из трех основных компонентов.

1. Клиент – это интерфейс пользователя, через который можно управлять и взаимодействовать с *Docker*. Клиенты *Docker* общаются с *Docker Daemon* (хостом), который выполняет тяжелую работу по сборке, запуску и распределению *Docker* контейнеров. Клиент *Docker* может находиться на том же хосте, что и демон, или общаться с демоном удаленно.

2. Хост принимает команды от клиента через *CLI* (командную строку) или через *API*, выполняет всю необходимую работу по созданию, запуску и мониторингу контейнеров. Демон также отвечает за управление объектами *Docker*, такими как образы, контейнеры, сети и тома.

3. Реестр – это хранилище для *Docker* образов. Он позволяет пользователям выкладывать и скачивать образы и использовать их для создания контейнеров. *Docker Hub* и *Docker Cloud* являются публичными реестрами, которые используются по умолчанию. Также существуют частные реестры, которые позволяют сохранять образы в закрытом доступе для организаций и команд.

Когда пользователь запрашивает создание контейнера из образа с помощью клиента, хост ищет этот образ локально. Если образ не найден локально, демон ищет его в конфигурированном реестре (по умолчанию Docker Hub), скачивает и использует для создания контейнера. После создания контейнера хост управляет его жизненным циклом по командам от клиента.

Эта модульная и распределенная архитектура обеспечивает гибкость и масштабируемость, позволяя *Docker* быть мощным инструментом для разработки, тестирования и развертывания приложений в различных средах (рис. 23).

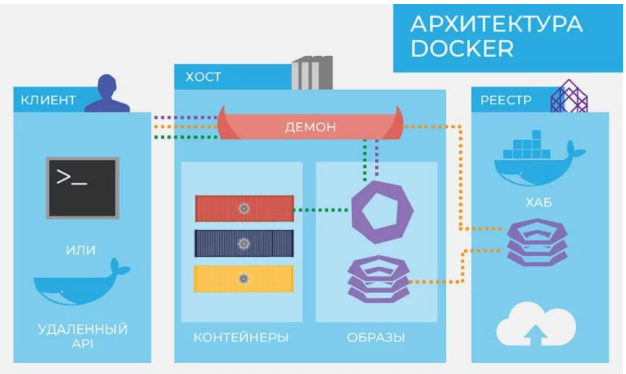


Рис. 23. Архитектура *Docker*

6.2. БАЗОВЫЕ КОМАНДЫ DOCKER

Контейнеризация с *Docker* позволяет создавать изолированные и легковесные среды для приложений. В этих контейнерах приложения могут работать независимо от остальной системы и других приложений, что упрощает разработку, тестирование и развертывание. В качестве примера мы создадим простой контейнер *Docker,* который запускает веб-сервер на базе образа *Nginx*.

Откройте *Docker Hub* и найдите официальный образ *Nginx*. Это можно сделать и командой

docker search nginx

Выполните команду загрузки образа (*pull*):

docker pull nginx

Используйте загруженный образ для создания нового контейнера. Запустите контейнер с портом, проброшенным на ваш локальный компьютер:

docker run --name my-nginx-container -p 8080:80 -d nginx

Эта команда создает и запускает контейнер под именем *my-nginx-container*, пробрасывает 8080 порт хоста на 80 порт в контейнере и запускает его в фоновом режиме.

Откройте веб-браузер и перейдите по адресу *http://localhost:8080*. Вы должны увидеть стартовую страницу *Nginx*, что означает, что ваш контейнер работает правильно.

Когда контейнер больше не нужен, его можно остановить и удалить:

docker stop my-nginx-container

docker rm my-nginx-container

Создание и управление контейнерами с *Docker* – процесс простой и стандартизированный. Использование контейнеров обеспечивает надежную и повторяемую разработку и развертывание приложений. В нашем примере мы создали базовый контейнер с веб-сервером *Nginx*, который можно использовать как отправную точку для более сложных приложений и сред.

Dockerfile – это текстовый документ, содержащий все команды, необходимые для сборки образа *Docker*. Используя *Dockerfile*, вы можете автоматизировать процесс упаковки вашего приложения и его зависимостей в контейнер. Это значительно упрощает создание консистентных и переносимых сред разработки и развертывания.

Каждый *Dockerfile* начинается с определения базового образа с помощью инструкции *FROM*. Базовый образ – это начальный образ, на основе которого создается ваш собственный. Для примера используем официальный образ *Python и* развернем *Flask*-приложени*е*:

FROM python:3.8-slim

Используйте инструкцию *RUN* для установки необходимого ПО. Например, установим *pip* и необходимые библиотеки:

RUN pip install --upgrade pip && \ pip install flask

С помощью инструкции *COPY* добавьте файлы исходного кода из вашего локального каталога в файловую систему образа:

COPY . /app

Укажите рабочий каталог для инструкций *RUN*, *CMD*, *ENTRYPOINT*, *COPY* и *ADD* с помощью *WORKDIR*:

WORKDIR /app

Используйте *ENV* для установки переменных окружения, которые будут использоваться в контейнере:

ENV FLASK_APP=app.py

ENV FLASK_RUN_HOST=0.0.0.0

ENTRYPOINT и CMD указывают команду и параметры, которые будут выполнены при запуске контейнера. Например, запустим *Flask*-приложение*:*

ENTRYPOINT ["flask"]

CMD ["run"]

С помощью инструкции *EXPOSE* можно указать порты, которые будут открыты в контейнере. *Flask* по умолчанию использует 5000:

EXPOSE 5000

```
FROM python:3.8-slim
RUN pip install --upgrade pip && \
    pip install flask
COPY . /app
WORKDIR /app
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENTRYPOINT ["flask"]
CMD ["run"]
EXPOSE 5000
```

Рис. 24. *Dockerfile*

Пример простого *Dockerfile* для веб-приложения на *Flask* приведен на рис. 24.

Dockerfile играет ключевую роль в экосистеме *Docker*, позволяя автоматизировать процесс сборки образов. Это обеспечивает быструю и эффективную разработку, а также гарантирует, что все разработчики и среды развертывания используют одни и те же зависимости и настройки.

Создание *Dockerfile* – это практика, которая стандартизирует и упрощает процесс доставки приложений.

6.3. DOCKER В ОБЛАКЕ

Docker в облачной среде становится все более популярным, поскольку он обеспечивает легкость, портативность и консистенцию развертывания приложений. Облачные платформы, такие как *AWS*, *Google Cloud* и *Azure*, предлагают интегрированные сервисы, которые упрощают управление контейнерами *Docker*, предоставляя инструменты для их развертывания, масштабирования и управления.

Преимущество использования *Docker* в облаке заключается в том, что разработчики могут упаковывать свои приложения в контейнеры на своем

локальном компьютере и быть уверенными, что эти контейнеры будут работать одинаково в любой облачной среде. Это снимает множество проблем, связанных с различиями в операционных системах и средах выполнения, и позволяет фокусироваться на создании и оптимизации самого кода.

Облачные провайдеры также предлагают сервисы оркестровки, такие как *Kubernetes*, которые позволяют автоматизировать развертывание, масштабирование и управление контейнеризированными приложениями. Это особенно важно для высоконагруженных приложений и микросервисной архитектуры, где необходимо управление большим количеством контейнеров, которые могут динамически масштабироваться в зависимости от нагрузки.

Безопасность в облачном *Docker* также получает особое внимание, так как контейнеры часто используются для изоляции приложений. Облачные провайдеры предлагают инструменты и практики для обеспечения безопасности на всех уровнях, от управления доступом до сканирования контейнеров на наличие уязвимостей.

Docker в облачной инфраструктуре значительно упрощает процесс разработки, тестирования и развертывания приложений, предлагая гибкость и масштабируемость, что делает его идеальным выбором для современных облачных решений.

7. ВВЕДЕНИЕ В CI/CD

7.1. ВВЕДЕНИЕ В DEVOPS

DevOps – это подход к разработке программного обеспечения, который фокусируется на тесной интеграции между командами разработки и операций. Цель *DevOps* – ускорить выпуск новых функций для пользователей и сделать этот процесс стабильным и надежным.

В традиционной модели разработки программного обеспечения команды разработчиков и ИТ-специалистов работали отдельно. Разработчики создавали новый код, после чего передавали его в ИТ-отдел для развертывания и поддержки. Такая модель часто приводила к проблемам и задержкам, поскольку команды находились в разных силосах организации и не были скоординированы.

DevOps пытается решить эти проблемы путем тесной координации между разработчиками и ИТ. Разработчики теперь также несут ответственность за развертывание и эксплуатацию своего кода. Они используют практики непрерывной интеграции и непрерывного развертывания (*CI/CD*), чтобы автоматизировать выпуск новых сборок. ИТ-специалисты больше не являются простыми исполнителями – они работают бок о бок с разработчиками над всем жизненным циклом приложения.

Внедрение *DevOps* требует значительных культурных изменений в организации. Необходимо ломать силосы между командами и поощрять сотрудничество. Процессы и инструменты должны меняться для поддержки итеративной, инкрементной модели разработки.

Рост популярности облачных технологий тесно связан с *DevOps*. Облачные платформы, такие как *AWS*, *Azure* и *Google Cloud*, предоставляют гибкую инфраструктуру, которая позволяет командам *DevOps* быстро масштабировать приложения. Инструменты облачных платформ, такие как контейнеризация и оркестровка *Kubernetes*, интегрированы с рабочим процессом

DevOps. Многие компании используют облачные платформы как ключевую часть своей стратегии *DevOps* для быстрой доставки ценности клиентам.

Когда *DevOps* внедрен правильно в сочетании с облачными технологиями, компании могут добиться более быстрой доставки ценности клиентам, повысить надежность систем и удовлетворенность сотрудников. Это позволяет организациям быть более гибкими и конкурентоспособными на рынке.

7.2. ПОНЯТИЕ CI/CD

CI/CD является собирательным термином, охватывающим несколько этапов *DevOps*. *CI* (непрерывная интеграция) – это способ интеграции изменений кода в репозиторий по несколько раз в день. У *CD* есть два значения: непрерывная доставка автоматизирует интеграцию в то время, как непрерывное развертывание автоматически выпускает финальную сборку для конечных пользователей. Регулярное тестирование в рамках *CI*/*CD* уменьшает количество ошибок и дефектов кода, что делает эту методику незаменимой для рабочего процесса *DevOps*.

CI – это методика *DevOps* и этап жизненного цикла *DevOps*, на котором разработчики регистрируют код в общем репозитории кода, часто по несколько раз в день. Желательно, чтобы каждый раз автоматический инструмент сборки проверял изменение или ветку на наличие ошибок и готовность к разработке. Отсюда и главное преимущество – проблемы выявляются на ранних этапах еще до того, как приведут к неприятным последствиям.

Реализация *CI* подразумевает интеграцию маленьких подгрупп изменений в компактные сроки вместо менее частых обновлений существенного объема, которые занимают больше времени. Благодаря автоматизации процессов тестирования, слияния и регистрации изменений в общем репозитории команды разработчиков могут ускоренно доставлять более читаемый код. Легкая читаемость кода позволяет быстрее проходить этап проверки, выпускать более стабильный продукт и повышать эффективность процесса разработки за счет упрощенного масштабирования.

CI – это методика *DevOps* и этап жизненного цикла *DevOps*, на котором разработчики регистрируют код в общем репозитории кода, часто по несколько раз в день. Желательно, чтобы каждый раз автоматический инструмент сборки проверял изменение или ветку на наличие ошибок и готовность к разработке. Отсюда и главное преимущество – проблемы выявляются на ранних этапах еще до того, как приведут к неприятным последствиям.

Реализация *CI* подразумевает интеграцию маленьких подгрупп изменений в компактные сроки вместо менее частых обновлений существенного объема, которые занимают больше времени. Благодаря автоматизации процессов тестирования, слияния и регистрации изменений в общем репозитории команды разработчиков могут ускоренно доставлять более читаемый код. Легкая читаемость кода позволяет быстрее проходить этап проверки, выпускать более стабильный продукт и повышать эффективность процесса разработки за счет упрощенного масштабирования.

За *CI* следует непрерывная доставка – своего рода контрольный этап в процессе разработки перед выпуском и развертыванием итогового продукта для пользователей. После подтверждения изменения кода автоматически доставляются в репозиторий.

Цель непрерывной доставки заключается в том, чтобы доставлять наборы изменений в главную сборку маленькими порциями, которые не нарушат статус «готово к коммерческому использованию» итогового продукта, не готового к выпуску. Готовый продукт может содержать небольшие ошибки, которые тем не менее не смогут поставить под угрозу удобство использования.

Реализация непрерывной доставки подразумевает, что разработчики будут тратить меньше времени на внутреннее тестирование, так как согласно этой методике до этапа доставки по определению доходит только стабильный код. Это упрощает процесс выявления ошибок и сокращает время на их исправление.

DevOps является культурой и процессом, нацеленным на повышение эффективности разработки программного обеспечения.

Конвейер *CI*/*CD* – это определенная цепочка этапов, связанная с инструментами и средствами автоматизации, на основе которых реализуется жизненный цикл *DevOps*. Хотя *CI*/*CD* и является неотъемлемой частью культуры *DevOps*, последняя гораздо шире охватывает жизненный цикл разработки программного обеспечения: от сотрудничества между разработчиками и структурой команд до мониторинга, контроля версий и т.д.

У разных компаний способ внедрения *DevOps* может сильно отличаться, но по своей сути *DevOps* невозможно реализовать без *CI*/*CD*. Конвейер *CI*/*CD* неразрывно связан с культурой *DevOps* и ее процессами небольших частых выпусков.

7.3. ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ CI/CD В ОБЛАКЕ

Интеграция непрерывной разработки и непрерывного развертывания (*CI/CD*) в облачной инфраструктуре становится ключевым фактором успеха в современной разработке программного обеспечения. Эта методология позволяет разработчикам быстрее и эффективнее доставлять программные продукты и обновления для них конечным пользователям, обеспечивая при этом высокое качество и стабильность программного продукта.

Введение *CI/CD* в облачную среду преобразует процесс разработки. Благодаря автоматизации процессов тестирования и развертывания команды могут сосредоточиться на разработке новых функций, а не на рутинных операциях. Облако предлагает ряд инструментов и сервисов, которые интегрируются с популярными системами контроля версий и поддерживают автоматическое развертывание приложений с каждым новым коммитом.

Кроме того, использование *CI/CD* в облаке значительно снижает время, необходимое для развертывания, поскольку инфраструктура облака позволяет легко и быстро масштабировать ресурсы. Это особенно ценно при управлении несколькими средами развертывания, такими как тестирование, стей-

джинг и производство, где каждая среда может быть автоматически обновлена с минимальными усилиями и простоями.

Облачные *CI/CD* пайплайны также улучшают сотрудничество в командах. Поскольку все процессы и история изменений хранятся в облаке, члены команды могут легко отслеживать прогресс и участвовать в процессе разработки, независимо от их географического положения. Это способствует лучшему пониманию и быстрому решению проблем, а также поддерживает прозрачность процесса разработки.

Преимущества внедрения *CI/CD* в облачной среде неоспоримы. Они приводят к более быстрому циклу разработки, высокой стабильности продукта и улучшенному сотрудничеству между участниками проекта. Облачные технологии предоставляют мощную платформу, которая помогает достигать этих преимуществ, делая *CI/CD* незаменимым элементом в арсенале современного разработчика.

7.4. ОБЗОР ИНСТРУМЕНТОВ CI/CD

В современной разработке программного обеспечения платформы непрерывной интеграции и доставки (*CI/CD*), такие как *GitHub Actions* или *GitLab CI*, играют центральную роль в поддержании эффективности процессов. Эти инструменты предоставляют не просто автоматизацию рутинных задач, но и способствуют культуре инноваций и быстрого реагирования на изменения, что критически важно для конкурентоспособности в быстро меняющемся цифровом мире.

Введение этих инструментов в рабочие процессы обеспечивает ряд преимуществ. *GitHub Actions*, например, позволяет автоматизировать тестирование и развертывание непосредственно в рамках репозиториев *GitHub*, используя преимущества интеграции с широко распространенной системой управления версиями. *GitLab CI*, с другой стороны, предлагает похожие воз-

можности в рамках экосистемы *GitLab*, облегчая создание сложных рабочих процессов для контроля качества и доставки продукта.

Эти инструменты делают возможным для разработчиков сосредоточиться на написании кода, в то время как механизмы *CI/CD* занимаются его проверкой, тестированием, сборкой и развертыванием. Интеграция с облачными сервисами усиливает эти преимущества, позволяя использовать масштабируемые ресурсы облака для увеличения производительности и надежности процессов.

Заключение работы с инструментами *CI/CD* в облачной среде подчеркивает их значимость: они не просто облегчают жизнь разработчикам, но и становятся стратегическим активом для организаций. Благодаря интеграции с облачными платформами, *GitHub Actions, GitLab CI* и подобные инструменты способны обеспечивать бесперебойную работу и постоянное улучшение процессов, что является ключом к успеху в динамичной среде цифровой экономики.

ЗАКЛЮЧЕНИЕ

В ходе изучения данного пособия читатели ознакомились с ключевыми аспектами облачных технологий, от их фундаментальных принципов до передовых методик и инструментов, необходимых для разработки и управления облачными решениями. Были рассмотрены практические сценарии использования *Linux*, *Git*, веб-серверов и баз данных, а также методы масштабирования и оптимизации приложений в облаке с применением *Docker*. Особое внимание было уделено процессам непрерывной интеграции и непрерывной доставки (*CI*/*CD*), подчеркивая их значимость для современного программного обеспечения.

Для дальнейшего развития и углубления знаний рекомендуется изучить специализированную литературу, научные журналы и патентную документацию, посвященные облачным вычислениям и IT-инфраструктуре. Они помогут оставаться в курсе последних научных исследований и практических разработок в области облачных технологий. Обращение к этим ресурсам позволит специалистам и студентам расширить свой профессиональный горизонт и способствовать инновациям в разработке и применении облачных систем.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Атаева, Г. И. Анализ возможности использования облачных технологий в высшем образовании Узбекистана / Г. И. Атаева, Х. Ю. Хамроева // Universum: технические науки. – 2022. – № 1-1(94). – С. 16 – 18.

2. Холодова, Е. А. Облачные технологии и их применение в бизнес-проектах / Е. А. Холодова, А. Н. Рыжков // E-Scio. – 2023. – № 5(80). – С. 153 – 158.

3. Хахина, А. М. Облачные технологии / А. М. Хахина, К. М. Сафонов // Интеграция мировой науки и техники: новые концепции и парадигмы. – 2023. – С. 70 – 73.

4. Володин, С. М. Внедрение в учебный процесс операционной системы Astra Linux, как эффективное решение программы импортозамещения / С. М. Володин, Е. В. Поколодина, Е. М. Баулин // Взаимодействие вузов, научных организаций и учреждений культуры в сфере защиты информации и технологий безопасности. – 2022. – С. 190 – 199.

5. Рыбаков, В. А. Разработка макета-тренажера для имитации информационного воздействия в ОС Windows и Linux / В. А. Рыбаков, М. А. Скосырских // Нанотехнологии. Информация. Радиотехника (НИР-22). – 2022. – С. 94 – 98.

6. Исследование протоколов сетевой безопасности / Ю. А. Андрусенко и др. // Auditorium. – 2023. – № 1(37). – С. 19 – 25.

7. Татарникова, Т. М. Кластеризация данных на лету для СУБД PostgreSQL / Т. М. Татарникова // Программные продукты и системы. – 2023. – Т. 36, № 2. – С. 196 – 201.

8. Хлопотов, Р. С. Особенности проектирования баз данных для автоматизированного рабочего места врача-нутрициолога / Р. С. Хлопотов // Известия Тульского государственного университета. Технические науки. – 2022. – № 9. – С. 84 – 89.

9. Семко, А. Е. Обзор алгоритмов балансировки нагрузки сервера для проекта по автоматизации конвейеров / А. Е. Семко, Ю. С. Гаврилова // Раз-

витие науки и практики в глобально меняющемся мире в условиях рисков. – 2023. – С. 345 – 349.

10. Дос, Е. В. Организация эластичных систем виртуальных облачных серверов / Е. В. Дос, К. Ш. Камалиденов, Д. Н. Мостовщиков // Наука, техника и образование. – 2022. – № 4(87). – С. 38 – 46.

11. Бедняк, С. Г. Система контроля версий Git / С. Г. Бедняк, А. А. Кузнецова // Актуальные проблемы информатики, радиотехники и связи. – 2023. – С. 148.

12. Бредихин, И. Р. Система контроля версий Git как средство взаимодействия преподавателя и студента при подготовке кадров СПО в области ИКТ / И. Р. Бредихин, М. В. Степанов // Конкурс научно-исследовательских работ студентов Волгоградского государственного технического университета. – 2022. – С. 201–202.

13. Голушко, А. П. Информационная безопасность контейнеров Docker на этапе их создания и запуска / А. П. Голушко, В. Г. Жуков // Редакционная коллегия. – 2022. – С. 389.

14. Болкусов, Г. Э. DevOps как инструмент для развития бизнеса / Г. Э. Болкусов // Современные проблемы экономического развития. – 2023. – С. 27 – 32.

15. Говядова, Ю. Г. Информационная безопасность с использованием DevOps / Ю. Г. Говядова // StudNet. – 2022. – Т. 5, № 1. – С. 332 – 338.

16. Кадыров, К. А. Сравнение инструментов для CI/CD GitLab и Jenkins / К. А. Кадыров, А. М. Сафин // Развитие современной науки и образования: актуальные вопросы, достижения и инновации. – 2022. – С. 123 – 125.

ОГЛАВЛЕНИЕ

Учебное электронное издание

НИКОЛЮКИН Максим Сергеевич
ОБУХОВ Артем Дмитриевич
ЛИТОВКА Юрий Владимирович

ОБЛАЧНЫЕ ТЕХНОЛОГИИ

Учебное пособие

Редактор Л. В. Комбарова
Графический и мультимедийный дизайнер Н. И. Кужильная
Обложка, упаковка, тиражирование Л. В. Комбаровой

ISBN 978-5-8265-2757-3

Подписано к использованию 29.03.2024.
Тираж 50 шт. Заказ № 38

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел./факс (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru