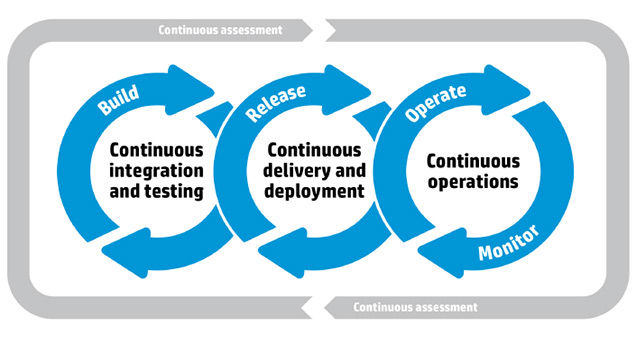
Всем привет на модуле CI/CD!

Меня зовут Надежды Балышева и мы с вами вместе, вы со мной, а я с вами будем постигать важные темы CI/CD и делиться опытом в рамках DQE программы. На онлайн сессии мы с вами рассмотрим задачи, области применения и основные инструмены для построения ci/cd процессов на проекте. Мы рассмотрим практический пример построения пайплайна на примере python репозитория.

Пожалуйста, ознакомьтесь с теоретическими материалами приведенными ниже и до встречи на сессии!

Поскольку DevOps закрепляет свои позиции в мире разработки программного обеспечения, то нам следует привыкнуть к новому термину “Continuous”. Непрерывность присутствует, наверное, во всех процессах, связанных с DevOps, и на слуху практически каждый день.

Хотя это слово стало широко распространенным, но некоторым до сих пор не понятно что именно оно означает? Когда оно используется в понятиях **Continuous Delivery**, **Continuous Deployment** и **Continuous Integration**, то как меняется его смысл? И какие именно различия между этими тремя терминами? В этой статье сделана попытка разобраться в данных терпинах и понять как их можно сочетать в одной среде.



**Что значит непрерывный?**

Прежде, чем мы начнём разбираться в различных концепциях **DevOps**, следует понять, что значит “*непрерывный*” в области программного обеспечения. Проще говоря, термин “непрерывности” относится к изменениям программного обеспечения, которые происходят в течении всего процесса разработки **ПО**.

Конечно же есть доля лукавства в термине “непрерывный”. На самом деле после реализации функционала может пройти некоторое время до того, как код попадёт в продакшен, но это время всё-таки гораздо меньше, чем было до появления *DevOps*.

**Continuous delivery (непрерывная доставка)**

В большинстве случаев **непрерывная доставка** — это серия практик, направленных на то, чтобы обновления программного обеспечения происходили практически постоянно. Данные методы гарантируют быстрое развёртывание на продакшене не меняя существующий функционал. *Непрерывная доставка* осуществима благодаря различным оптимизациям на ранних этапах процесса разработки.

Разработчик, сделав какую-либо фичу, отправляет её *QA-инженерам* для тестирования. Тестировщикам легче досконально оттестировать небольшой новый функционал и написать к нему *тест-кейсы*. Как только все проверки – прошли, новая фича попадает на дальнейшее тестирование авто-тестами и потом уже в релизный брэнч в системе контроля версий.

**Continuous delivery** поставляет бизнесу каждый функционал постепенно. Это позволяет получить сразу отклик от клиента и, при необходимости, сделать некоторые изменения.

**Другие преимущества Continuous delivery:**

* Внесение нового функционала в back-end для проверки совместимости с системой;
* Быстрое реагирование на потребности рынка;
* Возможность подстраивания под изменение бизнес-стратегии;
* Низкое количество потенциальных ошибок.

**Continuous deployment (непрерывное развёртываение)**

**Continuous deployment** часто путают с *continuous delivery*, хотя между ними существуют чёткие различия, которые следует знать и понимать.

Как выше было уже сказано *непрерывная доставка* обеспечивает постоянный выпуск обновлений пользователям. А **непрерывное развёртывание** отвечает за то, чтобы весь новый функционал после тестирования сразу же попал в основную программу без ручного вмешательства инженеров DevOps.

Тот же *Docker* создан для неприрывного развёртывания. DevOps инженеры могут обновлять контейнеры и разворачивать их сразу на продакшене в автоматическом режиме. Такой процесс является ключом к непрерывной доставке, т.к. весь процесс может занять всего лишь несколько минут.

Не всегда непрерывное развёртывание имеет смысл. Использование *фича-тоглинга* сводит на нет все преимущества. Всегда надо исходить из потребностей бизнеса и процессов внедрения нового функционала.

**Continuous integration (непрерывная интеграция)**

**Непрерывная интеграция** является ключевым компонентом практики *Agile Development*. Основой данной практики является постоянное попадание кода в центральный репозиторий после успешного запуска тестов. Основные цели **continuous integration** – поиск и устранение потенциальных проблем как можно быстрее, улучшение качества ПО и сокращение время для выпуска обновлений.

До того, как непрерывная интеграция стала широко распространённой, разработчики обычно работали изолировано, а только по окончанию работы объедениняли свои наработки. Порой это был очень трудоёмкий и длительный процесс.

При непрерывной интеграции разработчики часто заливают свои изменения в *центральный репозиторий*, выполняя до этого *unit – тесты*. Затем *система контроля версий* автоматически проверяет код на возможно безопасной интеграции с существующим в репозитории. При этом идёт постоянное поступление кода, что облегчает тестирование и сводит к минимуму возможные риски.

**Как это всё работает вместе?**

Перед тем как вы дойдёте до непрерываного развёртывания вам предстоит длинный путь. Сначала вы должны будете многое автоматизировать и пройти фазы непрерываной интеграции и доставки.

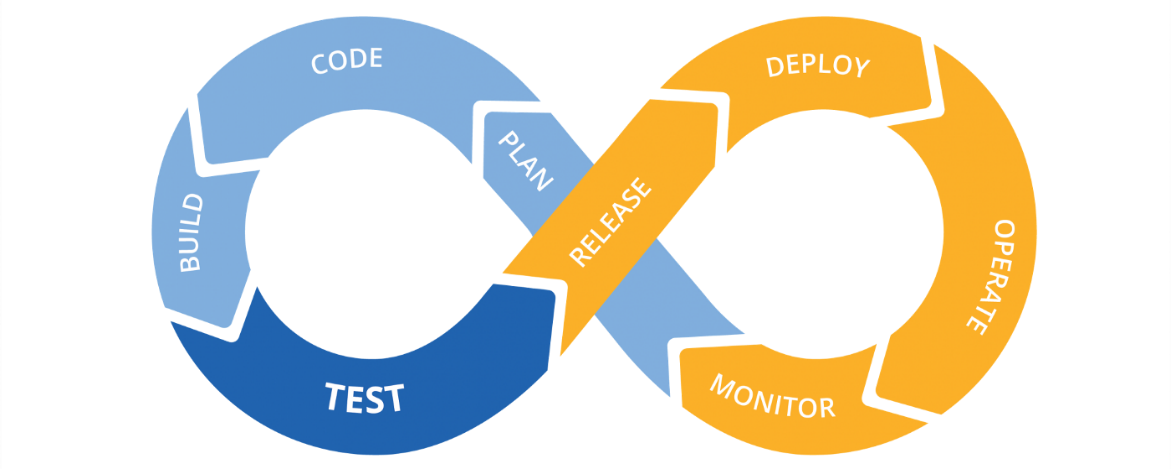
|  |  |  |
| --- | --- | --- |
| **Continuous Integration** | **Continuous Delivery** | **Continuous Deployment** |
| Необходимы автотесты на каждую новую фичу или баг-фикс | Тестами должно быть покрыто достаточный объём кода | Качество тестов влияет на качество готового продукта |
| Из-за ранней регрессии меньше багов попадает на продакшен | Развёртывание должно быть автоматизировано, обновления должны быть выпущены вручную | Развёртывание автоматизировано и тригеры настроены на каждое изменение |
| Разработчики должны мержится как можно чаще (минимум раз в день) | Команда может использовать фича — тоглинг | Фиче – тоглинг является неотъемлимой частью больших изменений |

**Идеальный процесс выглядит примерно так:**

* разработчик отправляет код в центральный репозиторий;
* на сервере непрерывной интеграции изменения объединяются с основным кодом, выполняются юнит – тесты и всё заливается на стэйжинг среду;
* в стэйжинг среде QA инженеры тестируют приложение;
* дальше всё проверяется для попадания на продакшен;
* развёртывание на продакшене.

## Core Concept

CI/CD — концепция, которая реализуется как конвейер, облегчая слияние только что закомиченного кода в основную кодовую базу. Концепция позволяет запускать различные типы тестов на каждом этапе (выполнение **интеграционного** аспекта) и завершать его запуском с развертыванием закомиченного кода в фактический продукт, который видят конечные пользователи (выполнение **доставки**).



## Принципы CI/CD

**1.сегрегация ответственности заинтересованных сторон**  
Одним из основных преимуществ CI/CD является своевременное участие различных заинтересованных сторон в любом проекте. Каждый этап конвейерной обработки CI/CD создает среду, настроенную на то, чтобы **группы брали ответственность за соответствующую стадию тестирования**, обеспечивая целостность процесса.

**2. снижение риска**

Каждый этап конвейерной обработки CI/CD создается для снижения риска в определенном аспекте. Разработчики отвечают за логические и письменные тесты, чтобы снизить риск нарушения логики. QE отвечают за целостность потока пользователей и записывают тесты для снижения риска сломанных потоков/ историй пользователей. BAs и POs отвечают за удобство использования и принимают участие в приемочных тестах пользователей, чтобы снизить риск создания непригодных / нежелательных функций. Ops/ DevOps несут участвуют в обслуживании CI/CD, связанные с развертыванием операций (анализ схемы данных/ миграции данных) и масштабирование, чтобы снизить риск недоступности продукта.

**3. короткий цикл обратной связи**

Причина внедрения конвейерной обработки CI/CD — использование машин для работы с людьми. Это позволяет сократить время, затрачиваемое на обратную связь по разрабатываемым функциям.

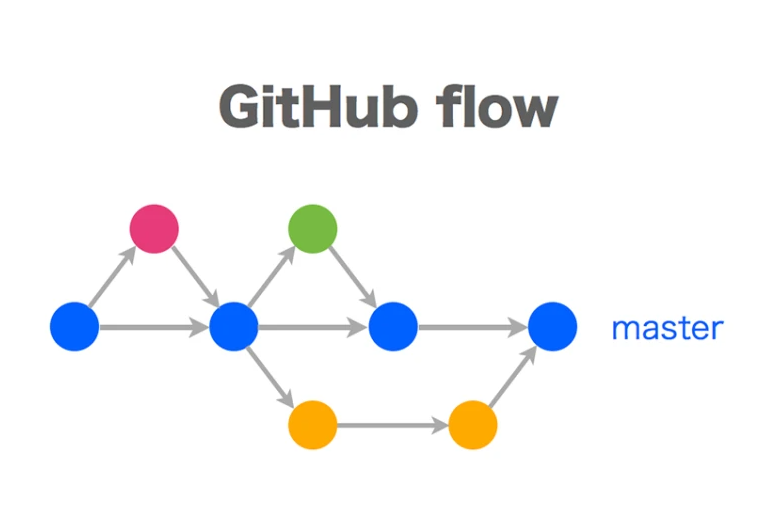
Но почему выгоднее использовать машины? Потому что люди не масштабируются, как машины. С помощью масштабирования сокращается время, затрачиваемое на тестирование программного обеспечения, что позволяет автоматизировать процесс развертывания. Эти процедуры займут гораздо больше времени, если будут выполняться человеком.

## ****Реализации среды в CI/CD. Стратегии ветвления****

При встрече ми рассмотрим различные среды реализации CI/CD и то, что происходит на каждом этапе, ссылаясь на реализацию в проектах, наг которыми мы работаем.

Очевидно, что фундаментом для построения неприрывной разработки, интеграции и деплоймента кода является выбранная стратегия работы с репозиториями кода. Рассмотрим наиболее популярные и академические подходы.

**GitHub Flow стратегия**



Стратегия ветвления, как бы это ни было странно, предпочитаемая в GitHub :) К ней прилагается [набор правил](https://githubflow.github.io/" \t "_blank), которым нужно следовать:

1. Код в master ветке должен быть не поломанным и готовым к развертыванию в любое время (то есть нельзя туда положить код, который помешает собрать проект и развернуть его на сервере).
2. Когда планируется работа над новой функциональностью, необходимо создать новую ветку (feature ветку) на основе master ветки и дать ей понятное имя. Коммитить свой код локально и регулярно пушить свои изменения на эту же ветку в удаленный репозиторий.
3. Открыть Pull-Request (что такое pull-request, можно почитать [здесь](https://habr.com/ru/post/125999/)), когда есть четкое ощущение, что работа готова и может быть смерджена в master ветку (или если уверенности нет, но хочется получить отзывы о проделанной работе).
4. После того, как новую фичу в пул-реквесте заапрувили, ее можно смерджить в master ветку.
5. Когда изменения смерджены в master ветку, их нужно развернуть на сервере немедленно.

По GitHub Flow получается, что прежде чем начать работу над чем-то новым, будь то исправление или новая фича, нужно создать новую ветку на основе master’а и дать ей подходящее имя. Далее, начинается работа над реализацией. Нужно постоянно отправлять коммиты на удаленный сервер с тем же именем. Когда приходит понимание, что все готово, нужно создать пул-реквест в master ветку. Потом хотя бы один, а лучше — два человека должны посмотреть этот код и нажать Approve. Обычно обязательно должен посмотреть тимлид проекта и кто-то еще, и тогда уже можно завершать пул-реквест. GitHub Flow еще известен тем, что драйвит [Continuous Delivery(CD)](https://rollout.io/blog/continuous-integration-continuous-delivery-continuous-deployment/" \t "_blank) на проекте. Потому что когда изменения заходят в master ветку, они должны сразу же быть развернуты на сервере.

## GitFlow стратегия

## 

В целом GitFlow состоит из двух постоянных веток и нескольких типов временных веток (В контексте GitHub Flow, master ветка — постоянная, а другие — временные). Постоянные ветки:

* master: эту ветку просто так никто не должен трогать/ничего не пушить туда. В этой стратегии master отображает последнюю стабильную версию, которую используют в продакшене (то есть на реальном сервере);
* development — это ветка для разработки. Потенциально она может быть нестабильная.

Разработка ведется при помощи трех вспомогательных временных веток:

1. Фиче ветки (feature branches) — для разработки новой функциональности.
2. Релизные ветки (release branches) — для подготовки выпуска новой версии проекта.
3. Хотфикс ветки (hotfix branches) — быстрое решение дефекта, который нашли уже реальные пользователи на реальном сервере.

**Фиче ветки (feature branches)**

Фиче ветки создаются разработчиками для нового функционала. Они всегда должны создаваться на основе development ветки. После завершения работы над новой функциональностью, нужно создать пул-реквест в development ветку. Понятно, что в больших командах одновременно может быть больше одной фиче ветки. Еще раз обратите внимание на картинку в начале описания стратегии GitFlow.

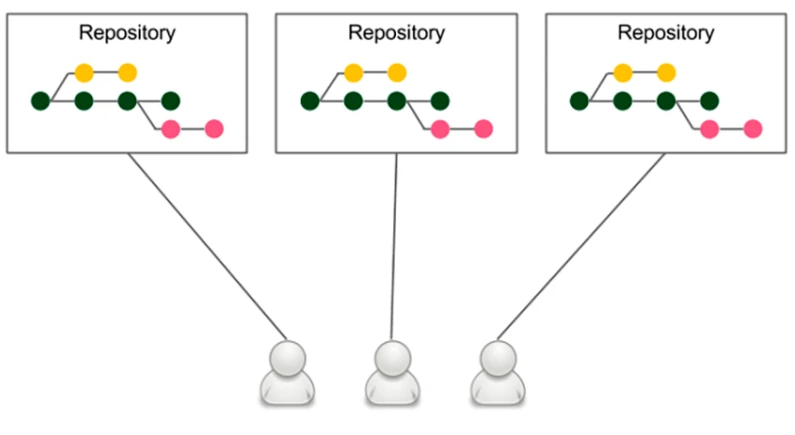
**Релизные ветки (release branches)**

Когда необходимое количество новых фич подготовлено в development ветке, можно подготовиться к выпуску новой версии продукта. В этом нам поможет релизная ветка. которая создается на основе development ветки. В ходе работы с релизной веткой нужно найти и починить все дефекты. Все новые изменения, которые требуются для стабилизации релизной ветки, нужно также смерджить обратно в development. Делается это для того, чтобы стабилизировать и development ветку. Когда тестировщики скажут, что ветка достаточно стабильная для нового релиза, ее смердживают в master ветку. Далее на этом коммите создается метка (tag: подробнее можно почитать об этом [здесь](https://git-scm.com/book/ru/v2/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%A0%D0%B0%D0%B1%D0%BE%D1%82%D0%B0-%D1%81-%D0%BC%D0%B5%D1%82%D0%BA%D0%B0%D0%BC%D0%B8" \t "_blank)), которой присваивается номер версии. Как пример, можно посмотреть на картинку в начале стратегии. Так вот, там Tag 1.0 — это как раз метка, которая указывает на версию 1.0 проекта. И последнее — это хотфикс ветки.

**Хотфикс ветки (Hotfix branches)**

Хотфикс ветки также предназначены для релиза новой версии в master. Только разница в том, что этот релиз не планируется. Бывают ситуации, когда дефекты доходят до релиза и уже обнаруживаются в работе. Например, iOS: как только выпустят новую версию, так сразу тебе куча обновлений с фиксами дефектов, которые обнаруживаются после релиза. В связи с этим нужно быстро пофиксить этот дефект и выпустить новую версию.

## The Forking Workflow стратегия



В рамках Forking Workflow стратегии разработка ведется так, что есть два репозитория:

1. Оригинальный репозиторий, в который будут смердживаться все изменения.
2. Форк репозиторий (это копия оригинального репозитория во владении другого разработчика, который хочет внести изменения в оригинальный).

Такая стратегия дает следующее премущество: разработка может вестись в форк-репозитории и без предоставления прав на совместную разработку в оригинальном. Разумеется, что владелец оригинального репозитория вправе отклонить предлагаемые изменения. Или согласиться и смерджить их. Это удобно и владельцу оригинального репозитория, и разработчику, который хочет поучаствовать в создании какого-то продукта.

*Подумайте, какая из стратегий ветвления лучше всего подходит для организации ci/cd процесса? С какими проблемами мы можем сталкнуться?*

## +/- CI/CD

В заключение хотелось бы обратить внимание на позитивные и негативные стороны внедрения ci/cd.

**Главные плюсы CI/CD:**

Скорость вывода новой функциональности от запроса клиента до запуска в эксплуатацию. CI/CD позволяет запускать обновления за считанные дни или недели по сравнению с целым календарным годом при классическом waterfall-подходе. Новые сервисы – новые конкурентные бизнес-преимущества. Появляется возможность не просто воспроизводить функциональность решений конкурентов, но и значительно опережать их в разработке и внедрении новых инструментов.

Возможность выбора оптимального варианта за счет оперативного тестирования и большего числа итераций. Отказавшись от работы над бесперспективными решениями, вы сэкономите ресурсы компании.

Качество итогового результата выше: автотестирование охватывает все аспекты продукта, что труднореализуемо при стандартном релизном подходе. Все ошибки и тонкие места выявляются и удаляются ещё на ранних этапах разработки.

**Главные минусы CI/CD:**

Искушение перевести на Agile, DevOps и CI/CD сразу всё, что связано с корпоративными ИТ-системами, включая core-уровень, без приобретения первичного опыта. Это может серьёзно нарушить работу компании, особенно при плохой организации перехода на новую методологию.

Поддержка должного уровня координации между CI и CD. Быстрые и качественные результаты от применения методики возможны только после длительной и тщательной настройки взаимодействия между командами DevOps, инженерами, scrum-экспертами и руководством компании. Самое сложное в CI/CD – человеческий фактор, налаживание здоровой командной работы, которую запрограммировать и автоматизировать невозможно.

**Контейнеризация с Docker**

Что же такое Docker и с чем его едят?



Docker - программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы; позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, а также предоставляет среду по управлению контейнерами.

Простыми словами, Докер это инструмент, который позволяет разработчикам, системными администраторам и другим специалистам деплоить их приложения в песочнице (которые называются контейнерами), для запуска на целевой операционной системе, например, Linux. Ключевое преимущество Докера в том, что он позволяет пользователям упаковать приложение со всеми его зависимостями в стандартизированный модуль для разработки. В отличие от виртуальных машин, контейнеры не создают такой дополнительной нагрузки, поэтому с ними можно использовать систему и ресурсы более эффективно.

### Что такое контейнер?

Стандарт в индустрии на сегодняшний день — это использовать виртуальные машины для запуска приложений. Виртуальные машины запускают приложения внутри гостевой операционной системы, которая работает на виртуальном железе основной операционной системы сервера.

Виртуальные машины отлично подходят для полной изоляции процесса для приложения: почти никакие проблемы основной операционной системы не могут повлиять на софт гостевой ОС, и наоборот. Но за такую изоляцию приходится платить. Существует значительная вычислительная нагрузка, необходимая для виртуализации железа гостевой ОС.

Контейнеры используют другой подход: они предоставляют схожий с виртуальными машинами уровень изоляции, но благодаря правильному задействованию низкоуровневых механизмов основной операционной системы делают это с в разы меньшей нагрузкой.

### Терминология

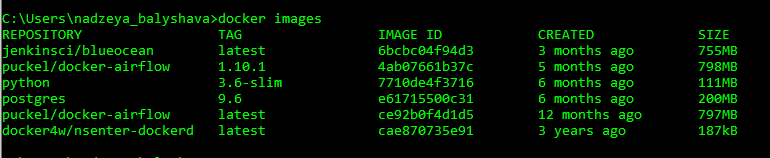
В предыдущем разделе мы использовали много специфичного для Докера жаргона, и многих это может запутать. Перед тем, как продолжать, давайте разберем некоторые термины, которые часто используются в экосистеме Докера.

* Images (образы) - Схемы нашего приложения, которые являются основой контейнеров.
* Containers (контейнеры) - Создаются на основе образа и запускают само приложение. Мы создали контейнер командой docker run, и использовали образ busybox, скачанный ранее. Список запущенных контейнеров можно увидеть с помощью команды docker ps.
* Docker Daemon (демон Докера) - Фоновый сервис, запущенный на хост-машине, который отвечает за создание, запуск и уничтожение Докер-контейнеров. Демон — это процесс, который запущен на операционной системе, с которой взаимодействует клиент.
* Docker Client (клиент Докера) - Утилита командной строки, которая позволяет пользователю взаимодействовать с демоном. Существуют другие формы клиента, например, [Kitematic](https://kitematic.com/), с графическим интерфейсом.
* Docker Hub - [Регистр](https://hub.docker.com/explore/) Докер-образов. Грубо говоря, архив всех доступных образов. Если нужно, то можно содержать собственный регистр и использовать его для получения образов.

### Образы

Мы касались образов ранее, но в этом разделе мы заглянем глубже: что такое Докер-образы и как создавать собственные образы.

Образы это основы для контейнеров. В прошлом примере мы скачали (**pull**) образ под названием Busybox из регистра, и попросили клиент Докера запустить контейнер, **основанный** на этом образе. Чтобы увидеть список доступных локально образов, используйте команду docker images.



Это список образов, которые я скачал из регистра, а также тех, что я сделал сам (скоро увидим, как это делать). TAG — это конкретный снимок или снэпшот (snapshot) образа, а IMAGE ID — это соответствующий уникальный идентификатор образа.

Для простоты, можно относиться к образу как к git-репозиторию. Образы можно [коммитить](https://docs.docker.com/engine/reference/commandline/commit/) с изменениями, и можно иметь несколько версий. Если не указывать конкретную версию, то клиент по умолчанию использует latest.

Скачать нужный образ можно с помощью команды

*docker pull*

Важно понимать разницу между базовыми и дочерними образами:

* **Base images** (базовые образы) — это образы, которые не имеют родительского образа. Обычно это образы с операционной системой, такие как ubuntu, busybox или debian.
* **Child images** (дочерние образы) — это образы, построенные на базовых образах и обладающие дополнительной функциональностью.

Существуют официальные и пользовательские образы, и любые из них могут быть базовыми и дочерними.

* **Официальные образы** — это образы, которые официально поддерживаются командой Docker. Обычно в их названии одно слово. В списке выше python, ubuntu, busybox и hello-world — базовые образы.
* **Пользовательские образы** — образы, созданные простыми пользователями вроде меня и вас. Они построены на базовых образах. Обычно, они называются по формату user/image-name.

### Dockerfile

Dockerfile — это простой текстовый файл, в котором содержится список команд Докер-клиента. Это простой способ автоматизировать процесс создания образа. Самое классное, что команды в Dockerfile почти идентичны своим аналогам в Linux. Это значит, что в принципе не нужно изучать никакой новый синтаксис чтобы начать работать с докерфайлами.

Простой пример сборки image для python проектов доступен здесь:

<https://www.docker.com/blog/containerized-python-development-part-1/>

## Jenkins



Jenkins — это инструмент автоматизации с открытым исходным кодом, написанный на Java, с плагинами, созданными для непрерывной интеграции. Jenkins используется для непрерывной сборки и тестирования ваших программных проектов, что облегчает разработчикам интеграцию изменений в проект и облегчает пользователям получение новой сборки. Это также позволяет вам непрерывно поставлять программное обеспечение, интегрируя с большим количеством технологий тестирования и развертывания.

С помощью Jenkins организации могут ускорить процесс разработки программного обеспечения за счет автоматизации. Jenkins объединяет процессы жизненного цикла разработки всех видов, включая сборку, документацию, тестирование, пакет, этап, развертывание, статический анализ и многое другое.

Jenkins достигает непрерывной интеграции с помощью плагинов. Плагины позволяют интегрировать различные этапы DevOps. Если вы хотите интегрировать определенный инструмент, вам нужно установить плагины для этого инструмента. Например: Git, проект Maven 2, Amazon EC2, HTML-издатель и т. д.

Настройка и интеграция Jenkins c GitHub хорошо описана тут:

<https://regularshow.ru/personal-account/razbiraemsya-s-jenkins-chto-eto-jenkins-continuous-integration-eto-prosto-nu-pochti.html>

Ознакомьтесь пожалуйста с основной концепцией Jenkins Pipelines:

<https://www.jenkins.io/doc/book/pipeline/>

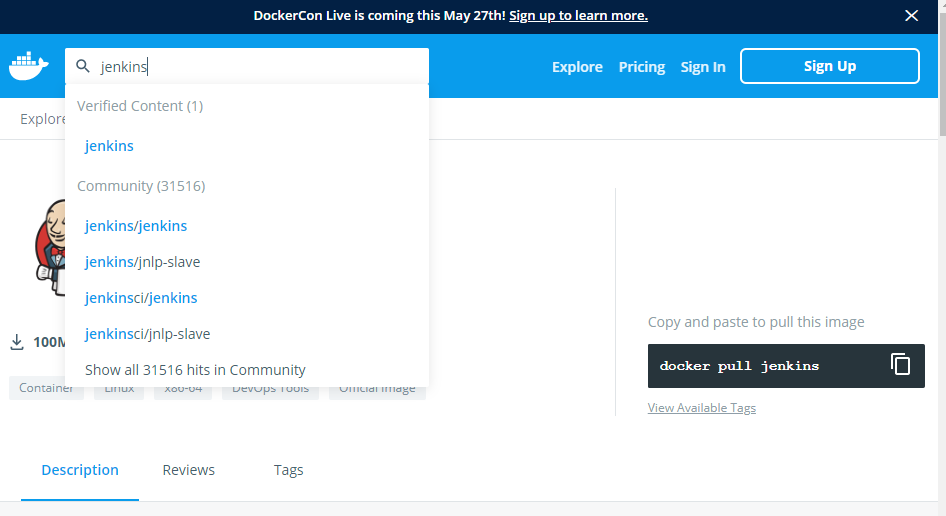
И основными bash командами тут:

<https://kmb.cybber.ru/bash/main.html>

C их применением на практике мы подробнее познакомимся на вебинаре.

## *Инструкции для выполнения практического задания*

* Скачайте и установите Docker для вашей операционной системы <https://www.docker.com/get-started>
* Убедитесь, что у вас установлен python и pip.
  + *python –version*
  + *pip –version*
* Найдите контейнер с Jenkins в реестре Docker <https://hub.docker.com/search?q=&type=image>



* Скачайте образ и запустите его командой Docker run. Переопределите внешний порт сервиса и домашнюю директорию:

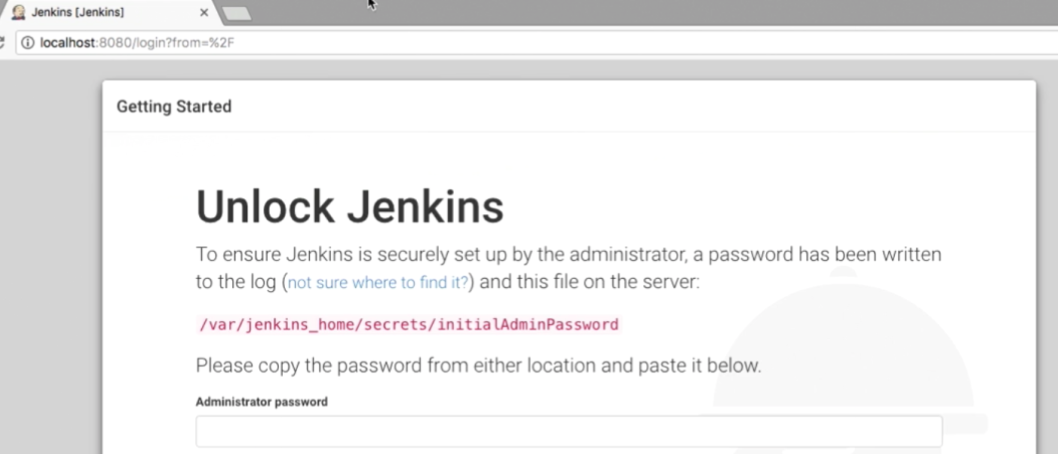
*docker run -p 8080:8080 -p 50000:50000 -v /your/home:/var/jenkins\_home jenkins/jenkins:lts-jdk11*

*docker run -p 8080:8080 -p 50000:50000 -v C:\Users\Aliaksandr\_Kikar\Documents\jenkins\_home:/var/jenkins\_home jenkins/jenkins:lts-jdk11*

*d11a7acb908d4698a06656f6c73cac86*

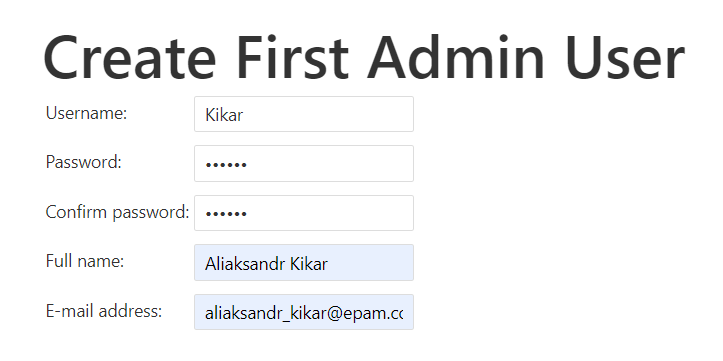
*C:\Users\Aliaksandr\_Kikar\Documents\Courses\_for\_training\DQE\_Mentoring\_Program*

* Настройка Jenkins: установите имя пользователя и пароль при первом запуске Jenkins

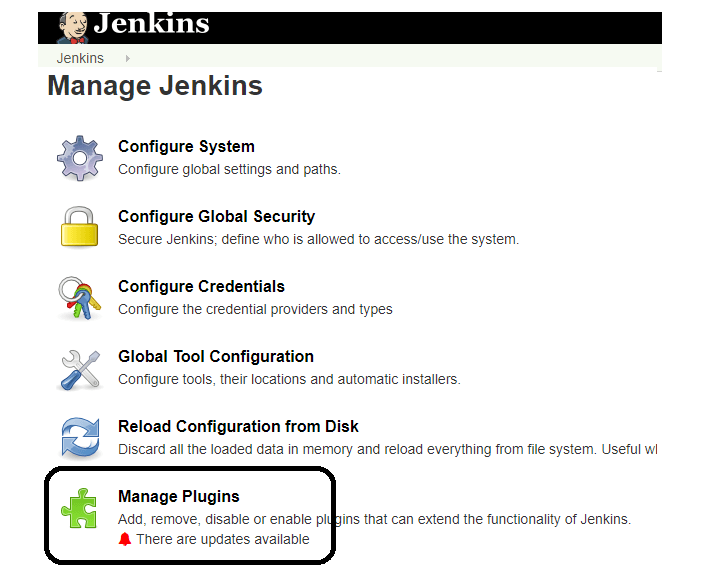


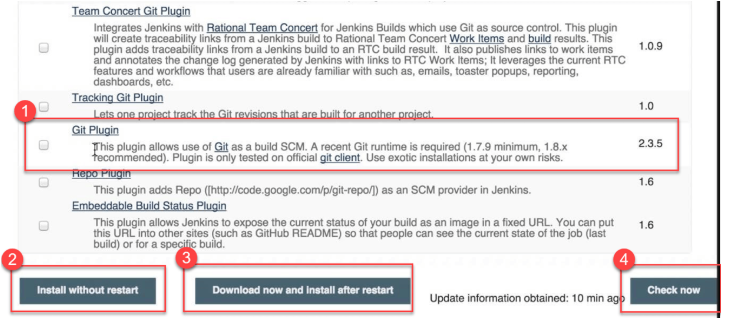
[Dashboard [Jenkins]](http://localhost:8080/) - host

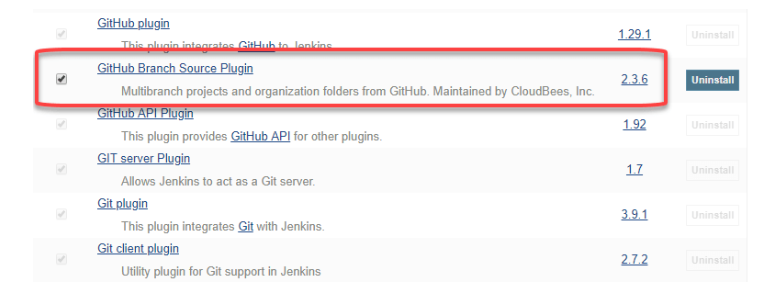
P – 101213



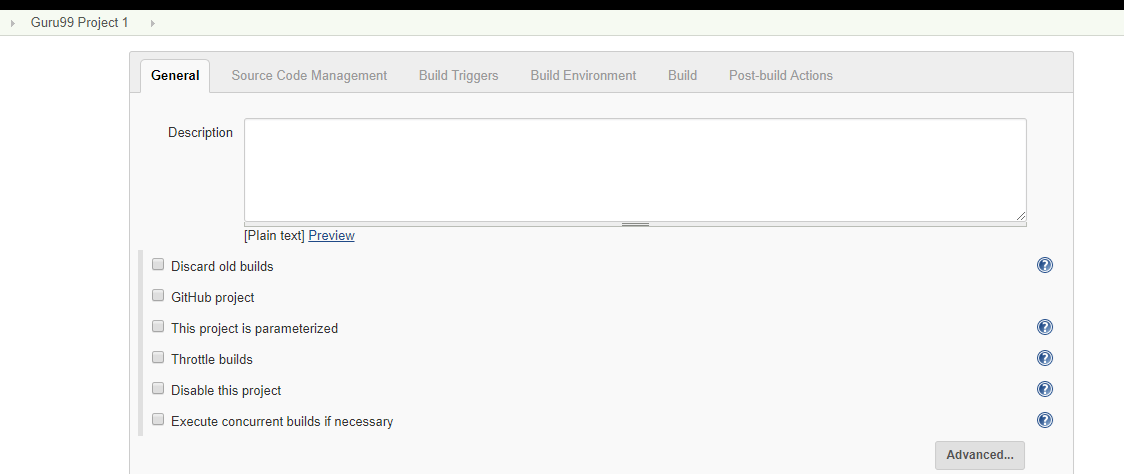
* Установите Git на свой локальный компьютер: перейдите по [ссылке https://git-scm.com/downloads.](https://translate.googleusercontent.com/translate_c?depth=1&pto=aue&rurl=translate.google.ru&sl=en&sp=nmt4&tl=ru&u=https://git-scm.com/downloads&usg=ALkJrhiaRE30dS6DAFPoRn_GrMcd_4mGTA) Исользуйте те того же пользователя, что и для Jenkins
* Создайте репозиторий в GitHub. Переместите свои скрипты в репозиторий.
* Настройте ветки в репозитории; Запуште примеры комитов и замерджите их с соотвестствии с выбранной стратегией.
* Установите плагин GIT для доступа к GitHub репозиторию



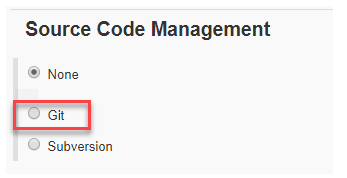
* 
* После установки плагинов перейдите в раздел «Управление Jenkins» на панели инструментов Jenkins. Вы увидите ваши плагины в списке среди остальных.

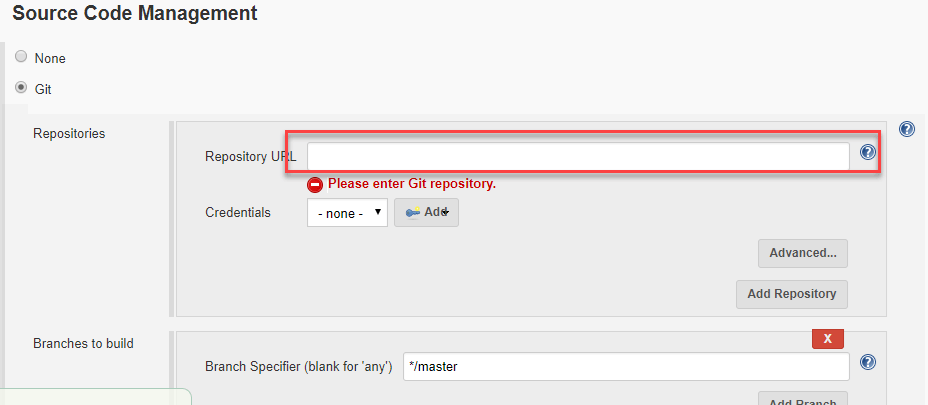


* Интеграция Jenkins с GitHub: создайте новый Jenkins Job. Тип проекта – freestyle project и введите информацию о проекте:



На второй вкладке вы увидите опцию **Git в** разделе « **Управление исходным кодом»,** если ваш плагин Git был установлен в Jenkins:



Введите URL-адрес хранилища Git, чтобы получить код из GitHub. 

* Разработайте JenkinsFile и добавьте его с свой репозиторий на GitHub. Команды и примеры доступны тут: <https://www.jenkins.io/doc/book/pipeline/>

Для теста :

pipeline {

agent any

stages {

stage('Test') {

steps {

echo 'Hello World ...'

}

}

}

}

######################

*pipeline {*

*agent any*

*stages {*

*stage ('GIT Checkout'){*

*steps {*

*git changelog: false, poll: false, url: 'https://github.com/...git'*

*}*

*}*

*stage('build') {*

*steps {*

*sh 'pip install -r requirements.txt'*

*}*

*}*

*stage ('Test'){*

*steps {*

*sh 'python unit-test.py'*

*}*

*}*

*}*

*}*

* Настроить триггер Git плагина для запуска jenkinsfile при push/pr в репозиторий.

Сгенерировать github Personal Access Token (PAT)

В GitHub репозитории задать URL вэб хуку *путь к дженкинсу*/github-webhook/

Выбрать ‘Let me select individual events’ ->‘Pull Request’ ...

* По аналогии установить и использовать почтовый плагин для отправки нотификаций.

Дополнительная пошаговая инструкция по настройке Jenkins:

[*https://faun.pub/basic-ci-cd-for-python-projects-with-docker-and-jenkins-38eeb547fb28*](https://faun.pub/basic-ci-cd-for-python-projects-with-docker-and-jenkins-38eeb547fb28)