

# CI/CD с помощью Jenkins

Практика непрерывной интеграции и доставки



# Введение в Jenkins

- Процесс развертывания и обновления приложений (который мы также знаем под названием **CI/CD**) зачастую может быть сложнее разработки этих приложений. В этом процессе нам надо учитывать множество вещей - обеспечение одинакового окружения (операционная система и необходимые библиотеки) как в среде разработки, так и в боевой среде, сохранение уже существующих данных и осуществление миграций баз данных при обновлении, откат обновлений при ошибке и запуск тестов (при этом надо учесть, чтобы тестовые данные не попали в настоящее приложение). Безусловно, все это можно автоматизировать при помощи самописных программ, однако гораздо удобнее воспользоваться уже готовым и проверенным решением.
- Одним из самых популярных и проверенных временем инструментов **CI/CD** является программа Jenkins. Она появилась в 2011 году, распространяется бесплатно (лицензия MIT), позволяет запускать отдельные автоматически работающие задачи и даже целые конвейеры задач (pipeline), поддерживает систему бесплатных дополнений (плагинов), которые умеют работать с удаленными соединениями (например, с помощью SSH), подгружать данные из GitHub и запускать Docker контейнеры. Кроме того, Jenkins имеет удобный интерфейс, который позволяет настраивать вышеупомянутые сущности.

# Установка Jenkins

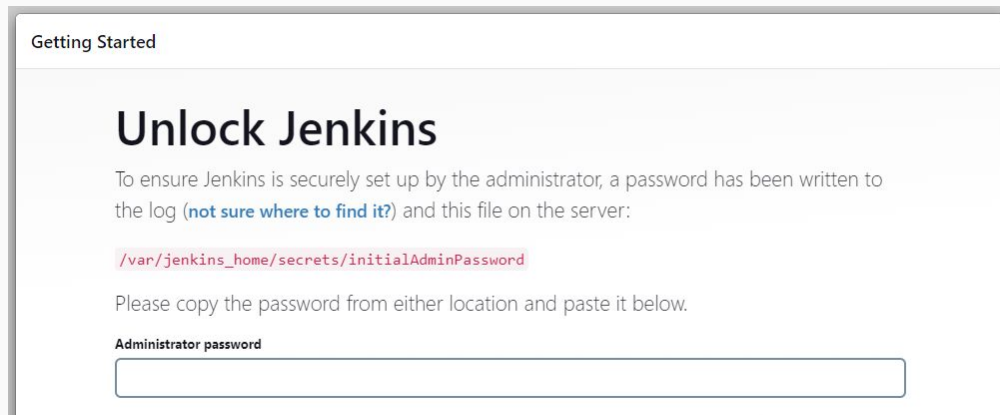
- Установка Jenkins может сильно различаться между разными операционными системами. Если нам потребуется конкретная специфика, то мы можем обратиться к документации <https://www.jenkins.io/doc/book/installing/>, однако для обучения гораздо удобнее развернуть Jenkins внутри Docker контейнера. Более того, на сайте Docker Hub существует готовый образ **jenkins/jenkins**, который содержит все необходимое для работы (хотя следует признать, что образ имеет некоторые недостатки, связанные с перезагрузкой системы после установки плагинов). Установим, наконец, Jenkins при помощи Docker:

```
docker run -d --name=jenkins -u root -p 8080:8080 -e DOCKER_HOST=tcp://host.docker.internal:2375 --restart=on-failure jenkins/jenkins:lts-jdk17
```

- При установке мы пробросили порт 8080 нашей операционной системы внутрь Docker контейнера (при этом используя 8080 порт самого контейнера). Дело в том, что Jenkins по умолчанию слушает именно этот порт - таким образом, если мы зайдём в браузер и перейдём по адресу ***http://localhost:8080***, то увидим относительно приятный web-интерфейс, посредством которого мы продолжим установку и настройку Jenkins, а после этого по этому же адресу будем создавать и автоматизировать задачи.

# Настройка Jenkins (применение секретного пароля)

- На странице <http://localhost:8080> перед нами предстанет предложение ввести пароль, который находится внутри файловой системы контейнера. Зайдет в контейнер, копируем пароль, а потом вводим его в поле:

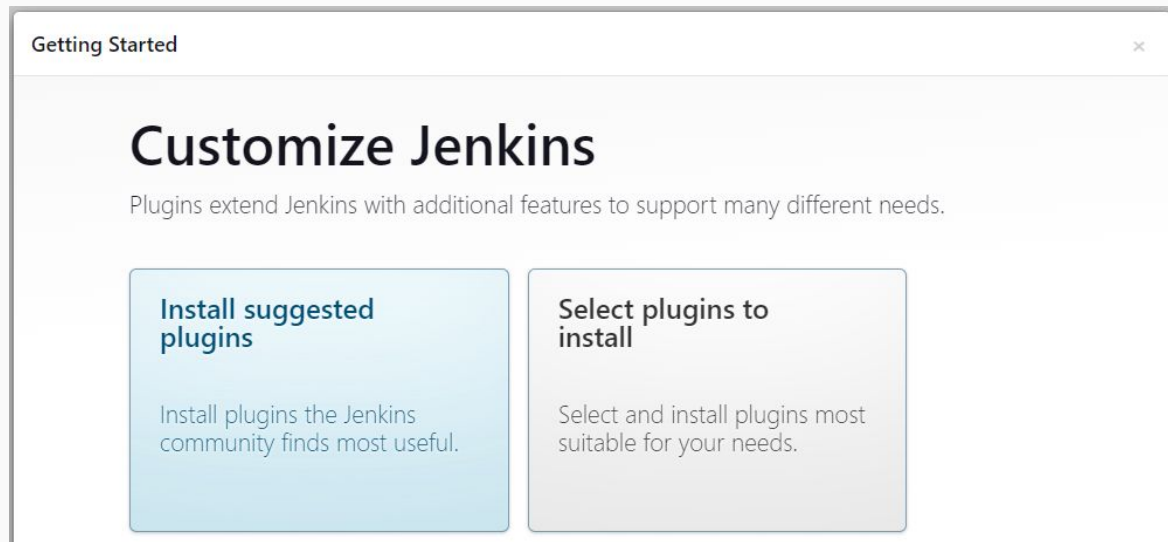


```
docker exec -it jenkins /bin/bash
```

```
cat /var/jenkins_home/secrets/initialAdminPassword
```

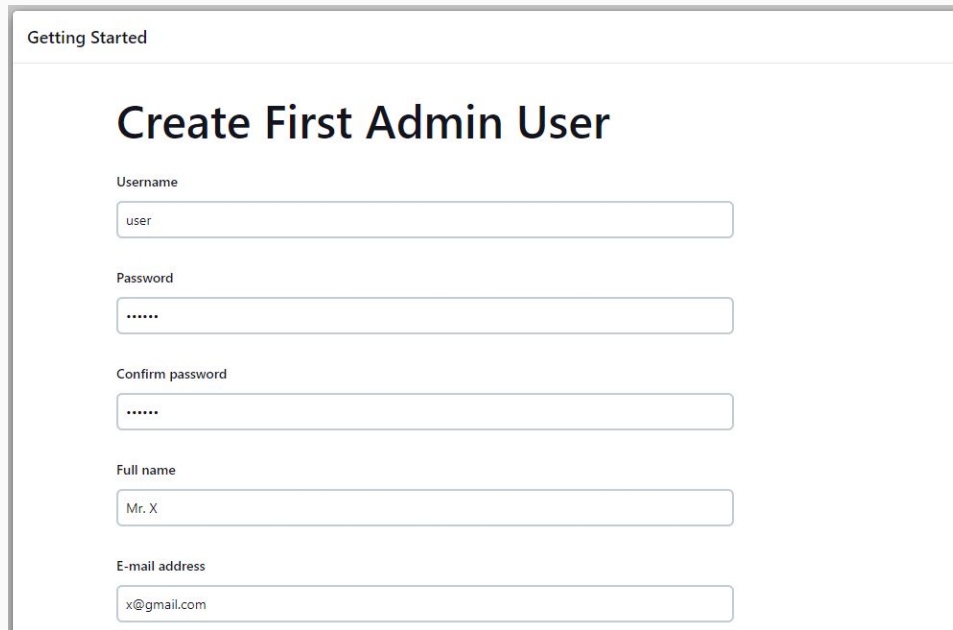
# Настройка Jenkins (установка рекомендованных плагинов)

- На следующей странице нам будет предложено установить плагины - выбираем установить рекомендованные (**Install suggested plugins**) - нам действительно потребуются многие из них:



# Настройка Jenkins (создание администратора)

- После установки плагинов нам будет предложено создать администратора. Условимся, что его именем будет **user**, а паролем - **secret**. Остальные данные можно указать по своему выбору:



The screenshot shows the 'Getting Started' page in Jenkins. The main heading is 'Create First Admin User'. Below it are five input fields with labels: 'Username' (containing 'user'), 'Password' (containing '\*\*\*\*\*'), 'Confirm password' (containing '\*\*\*\*\*'), 'Full name' (containing 'Mr. X'), and 'E-mail address' (containing 'x@gmail.com').

Getting Started

## Create First Admin User

Username

Password

Confirm password

Full name

E-mail address

# Настройка Jenkins (установка ссылки)

- Далее мы установим ссылку, по которой будет доступен Jenkins - в данном случае оставим то, что нам предлагает установщик - <http://localhost:8080>. Далее мы подтвердим наш выбор и на этом установка будет закончена:

Getting Started

## Instance Configuration

Jenkins URL:

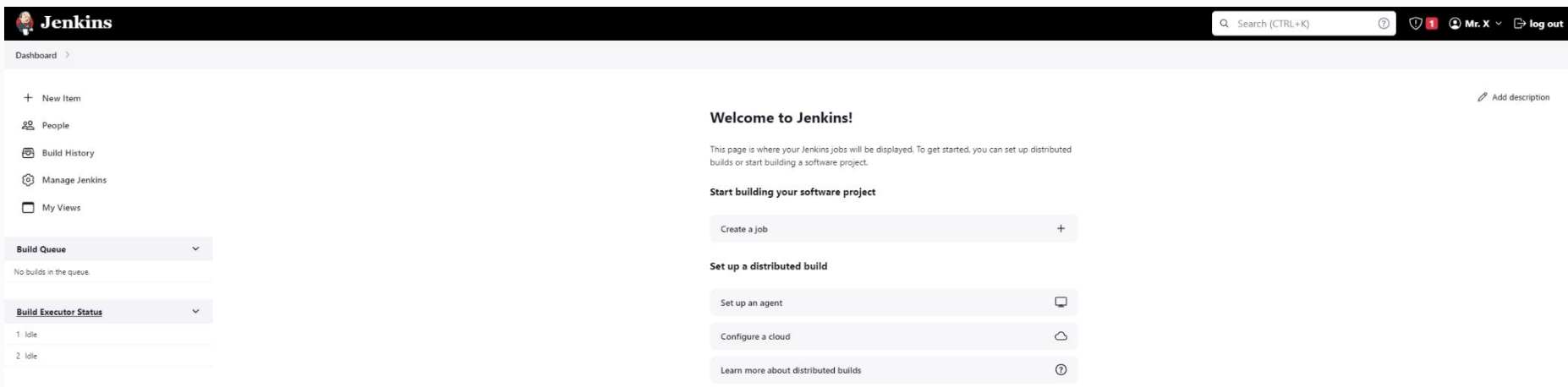
The Jenkins URL is used to provide the root URL for absolute links to various Jenkins resources. That means this value is required for proper operation of many Jenkins features including email notifications, PR status updates, and the `BUILD_URL` environment variable provided to build steps.

The proposed default value shown is **not saved yet** and is generated from the current request, if possible. The best practice is to set this value to the URL that users are expected to use. This will avoid confusion when sharing or viewing links.



# Вид административной панели Jenkins

- В конце концов мы попадем на главную страницу Jenkins, где в меню слева получим возможность создать задание (**New Item**), добавить и отредактировать параметры пользователей (**People**), просмотреть историю выполнения задач (**Build History**), настроить внутренние параметры Jenkins - установить новые плагины, отрегулировать требования к безопасности и т.д. (**Manage Jenkins**), а также объединить задания в группы для удобного просмотра (**My Views**):



# Основы управления задачами в Jenkins

- Нет сомнений, что основная задача Jenkins - выполнение задач. Поэтому мы нажмем на пункт меню **New Item** и попадем в раздел создания задач. Там нам будет представлен целый список возможных типов задач. Сразу же отметим, что сам по себе Jenkins предоставляет тип **Freestyle project**, а все остальные были добавлены теми плагинами, которые мы установили при установке Jenkins. Для начала выберем как раз **Freestyle project**, введем для него название **My First Project** и нажмём внизу кнопку с надписью **OK**:

The image shows a dialog box titled "Enter an item name". At the top, there is a text input field containing "My First Project". Below the input field, there is a small text label "+ Required field". Below this, there is a list of project types, each with an icon and a description:


- Freestyle project**: This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Pipeline**: Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
- Multi-configuration project**: Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Folder**: Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.
- Multibranch Pipeline**: Creates a set of Pipeline projects according to detected branches in one SCM repository.
- Organization Folder**: Creates a set of multibranch project subfolders by scanning for repositories.

At the bottom of the dialog, there is a blue button labeled "OK".

# Базовая настройка задачи (основные параметры)

- Внутри конфигурационной панели задачи перед нами предстанет множество настроек. В самом первом блоке **General** мы можем задать описание задачи (**Description**), настроить то, как будут храниться результаты выполнения задач (**Discard old build**), установить, ассоциирована ли наша задача с неким GitHub проектом (**GitHub Project**), а также решить, нужны ли нам дополнительные параметры при выполнении задачи (**This project is parameterised**). Остальные параметры на данный момент не столь важны, их мы рассматривать не будем:


General

Enabled 


Description


Plain text


[Preview](#)


☐ Discard old builds 

☐ GitHub project

☐ This project is parameterised 

☐ Throttle builds 

☐ Execute concurrent builds if necessary 

Advanced 

## Базовая настройка задачи (способы запуска задачи)

- В следующем блоке **Build Triggers** можно настроить механизм, посредством которого Jenkins будет запускать задачу. **Trigger builds remotely** устанавливает, что для старта задачи внешняя сила (человек или программа) должна перейти по определенной ссылке. **Build after other projects are built** означает, что задача должна запуститься после того, как отработает другая задача. **Build periodically** показывает, что задача будет выполняться периодически - формат для периода такой же, как у планировщика **cron**, например, ежеминутно - `* * * * *`. **GitHub hook trigger for GITScm polling** полагается на то, что задача запустится с помощью специального механизма GitHub, а конфигурация **Poll SCM** будет запрашивать изменения у некой системы контроля версий - если изменения были, то задача будет запущена:

### Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts) ?
- ☐ Build after other projects are built ?
- ☐ Build periodically ?
- ☐ GitHub hook trigger for GITScm polling ?
- ☐ Poll SCM ?

## Базовая настройка задачи (конфигурация окружения)

- Иногда перед выполнением задачи следует подготовить окружение - запустить те или иные процессы, создать или удалить файлы и т.д. Этому посвящен блок настроек **Build Environment**. Попробуем объяснить основные настройки. **Delete workspace before build starts** - позволяет удалить некоторые файлы перед стартом задачи. **Use secret text(s) or file(s)** - предоставляет возможность установить в качестве переменных окружения секретные данные. **Send files or execute commands over SSH before/after the build starts** - запустить процессы или создать файлы на удаленном компьютере до запуска или после окончания задачи. **Terminate a build if it's stuck** - прекратить выполнение задачи, если она выполняется слишком долго (можно указать максимальное время выполнения):

### Build Environment

- ☐ Delete workspace before build starts
- ☐ Use secret text(s) or file(s) ?
- ☐ Send files or execute commands over SSH before the build starts ?
- ☐ Send files or execute commands over SSH after the build runs ?
- ☐ Add timestamps to the Console Output
- ☐ Inspect build log for published build scans
- ☐ Terminate a build if it's stuck
- ☐ With Ant ?

## Базовая настройка задачи (конфигурирование хода задачи)

- Следующий блок настроек **Build Steps** посвящен решению самой задачи. В выпадающем меню **Add build step** можно выбрать множество вариантов, которые позволят запустить команды консоли Windows, воспользоваться командами Linux, инициализировать команды с помощью SSH соединения и много другое. Варианты можно добавлять многократно. Мы еще вернемся к этим опциям, когда будем создавать конкретные задачи:

Build Steps

Add build step ▾

## Базовая настройка задачи (действия после выполнения задачи)

- Последний блок настроек **Post-build Actions** связан с действиями, которые могут произойти после выполнения задачи. В выпадающем списке **Add post-build action** можно выбрать опции, которые позволят отправить сообщение по электронной почте, отправить на какой-либо удаленный компьютер результат выполнения задачи, связать каким-либо способом задачу с GitHub репозиторием, запустить другую задачу и многое другое. Варианты можно добавлять многократно. Как и в случае предыдущей секции мы еще вернемся к этим опциям, когда будем создавать конкретные задачи:

Post-build Actions

Add post-build action ▾

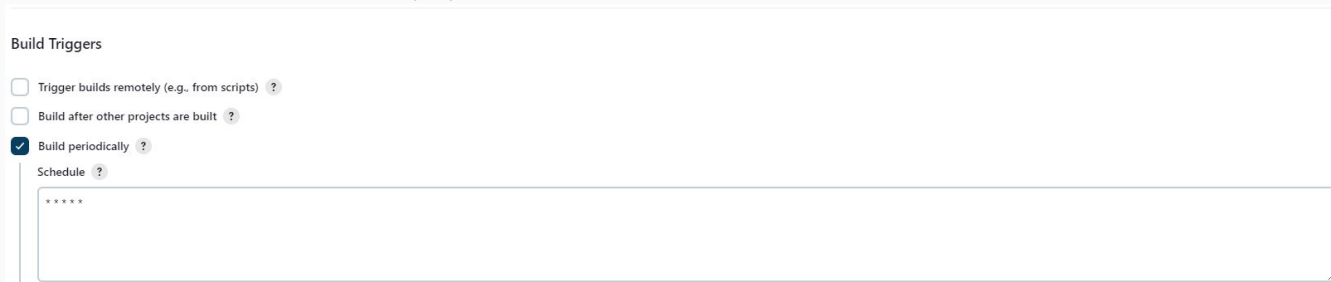


Создание, запуск и удаление  
простейшей Jenkins задачи

- В качестве самого первого нашего задания (которое будет выполняться на той же операционной системе, где работает Jenkins) попробуем записывать в файл **/var/jenkins\_home/load.log** нагрузку процессора за последнюю минуту и время, когда эта информация была получена. Условимся, что задание должно выполняться каждую минуту. Таким образом, через некоторое время мы сможем получить поминутную статистику по нагрузке нашего Jenkins сервера.

# Настройка задачи

- Внутри нашего задания **My First Project** в разделе **Build Triggers** установим, что выполнение будет происходить периодически (**Build periodically**). Внутри настройки пропишем пять звездочек - **\* \* \* \* \*** - это означает, что задание будет выполняться раз в минуту:



Build Triggers

☐ Trigger builds remotely (e.g., from scripts) ?

☐ Build after other projects are built ?

☒ Build periodically ?

Schedule ?

\*\*\*\*\*

- Затем внутри раздела **Build steps** в выпадающем списке выберем опцию **Execute shell** и внутрь вписываем Linux команду записи информации о процессоре и времени в файл **/var/jenkins\_home/load.log**:

```
echo $(cat /proc/loadavg | awk '{print $1}') `date +%Y-%m-%d %H:%M:%S` >> /var/jenkins_home/load.log
```



Build Steps

Execute shell ?

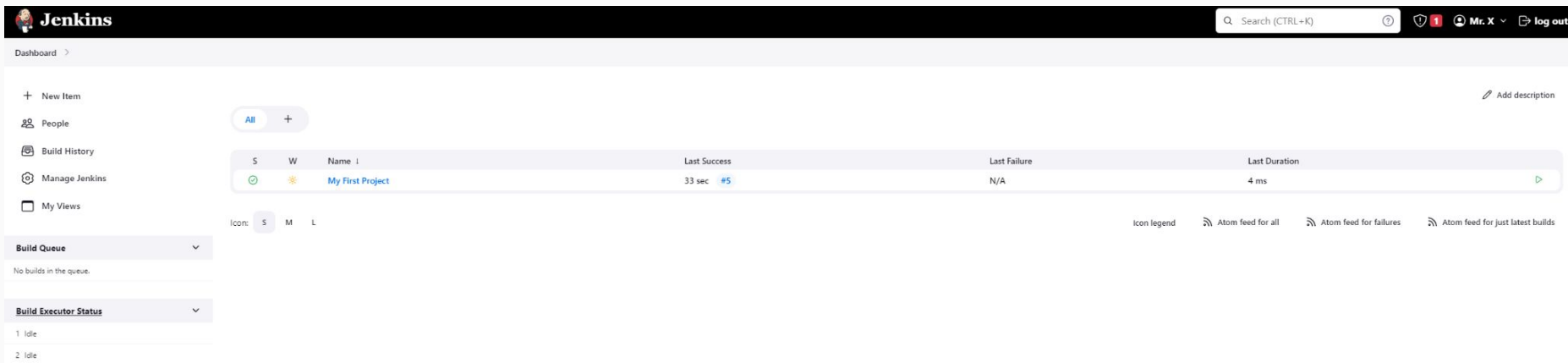
Command

See the list of available environment variables

```
echo $(cat /proc/loadavg | awk '{print $1}') `date +%Y-%m-%d %H:%M:%S` >> /var/jenkins_home/load.log
```

# Запуск задачи

- После того, как мы настроили все параметры, нам следует нажать нижнюю синюю кнопку **Save**. Т.к. мы выбрали периодическую сборку проекта, задача начнет обрабатывать немедленно. Чтобы посмотреть на результаты, мы должны перейти на главную страницу Jenkins - там мы увидим список из одной (только что созданной) задачи. Вполне возможно, что задача уже могла успеть отработать определенное количество раз, поэтому внутри описания задачи будет отображено ее состояние. Кстати, несмотря на автоматический режим выполнения задачи, мы все равно имеем возможность запускать ее вручную - для этого надо нажать на зеленый треугольник в правой стороне ряда задачи:



The screenshot shows the Jenkins Dashboard interface. At the top, there's a navigation bar with the Jenkins logo, a search bar, and user information. The main content area displays a table of jobs. The table has columns for status (S), week (W), name (Name), last success, last failure, and last duration. A single job, 'My First Project', is listed with a status of 'S' (Success) and a last success time of '33 sec #5'. The last failure is 'N/A' and the last duration is '4 ms'. To the right of the job name, there's a green play button icon. Below the table, there's a section for 'Build Queue' and 'Build Executor Status'.

| S | W | Name             | Last Success | Last Failure | Last Duration |
|---|---|------------------|--------------|--------------|---------------|
|   |   | My First Project | 33 sec #5    | N/A          | 4 ms          |

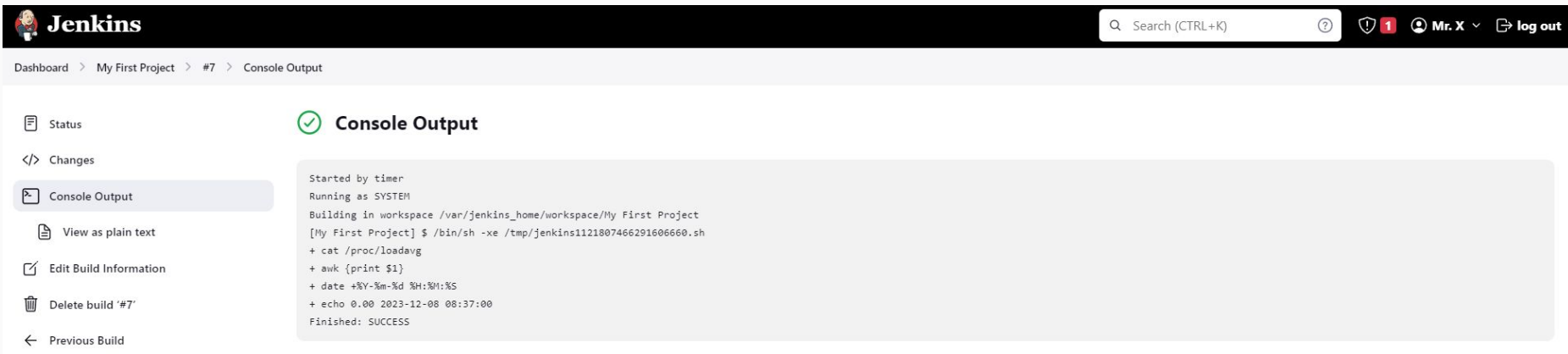
Icon: S M L

Build Queue: No builds in the queue.

Build Executor Status: 1 Idle, 2 Idle

# Просмотр хода выполнения задачи

- Если мы хотим детально проанализировать ход выполнения задачи, то нам следует привести курсор мышки на колонку **Last Success** или **Last Failure** (результат в случае успеха или неудачи) и нажать на появившийся треугольник. Затем в открывшемся меню следует выбрать пункт **Console Output**. После этих манипуляций откроется специальная страница с необходимой информацией:



The screenshot shows the Jenkins web interface. At the top is a black header with the Jenkins logo, a search bar, and user information. Below the header is a breadcrumb trail: Dashboard > My First Project > #7 > Console Output. On the left side, there is a sidebar with links: Status, Changes, Console Output (which is highlighted), View as plain text, Edit Build Information, Delete build '#7', and Previous Build. The main area displays the 'Console Output' for build #7, marked with a green checkmark. The output text shows the build was started by a timer, running as SYSTEM, and executed a series of commands in a shell, including cat, awk, date, and echo, all of which completed successfully.

**Jenkins** Search (CTRL+K) Mr. X log out

Dashboard > My First Project > #7 > Console Output

Status  
Changes  
Console Output  
View as plain text  
Edit Build Information  
Delete build '#7'  
Previous Build

✓ **Console Output**

```
Started by timer
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/My First Project
[My First Project] $ /bin/sh -xe /tmp/jenkins1121807466291606660.sh
+ cat /proc/loadavg
+ awk '{print $1}'
+ date +%Y-%m-%d %H:%M:%S
+ echo 0.00 2023-12-08 08:37:00
Finished: SUCCESS
```

## Просмотр результата выполнения задачи

- Как мы помним, мы настроили задачу таким способом, чтобы она сохраняла информацию о нагрузке процессора в том самом Docker контейнере, в котором находится сам Jenkins. Поэтому для просмотра результатов нам сначала следует перейти в командной строке в операционную систему контейнера:

```
docker exec -it jenkins /bin/bash
```

- Затем мы просто заглянем внутрь того самого файла, который используется задачей для заполнения информацией:

```
cat /var/jenkins_home/load.log
```

```
0.13 2023-12-08 21:38:00
0.12 2023-12-08 21:39:00
0.04 2023-12-08 21:40:00
0.01 2023-12-08 21:41:00
0.17 2023-12-08 21:42:00
0.15 2023-12-08 21:43:00
```

## Удаление задачи

- Когда задача теряет свою актуальность, мы можем захотеть ее удалить. Для этого мы должны перейти на главную страницу Jenkins, отыскать необходимую задачу и навести курсор мышки на название этой задачи. Затем в появившемся окошке следует отыскать опцию **Delete Project** и нажать на нее. Появится окно подтверждения, в котором мы подтвердим наши действия нажатием кнопки **OK**. В результате проект будет удален.

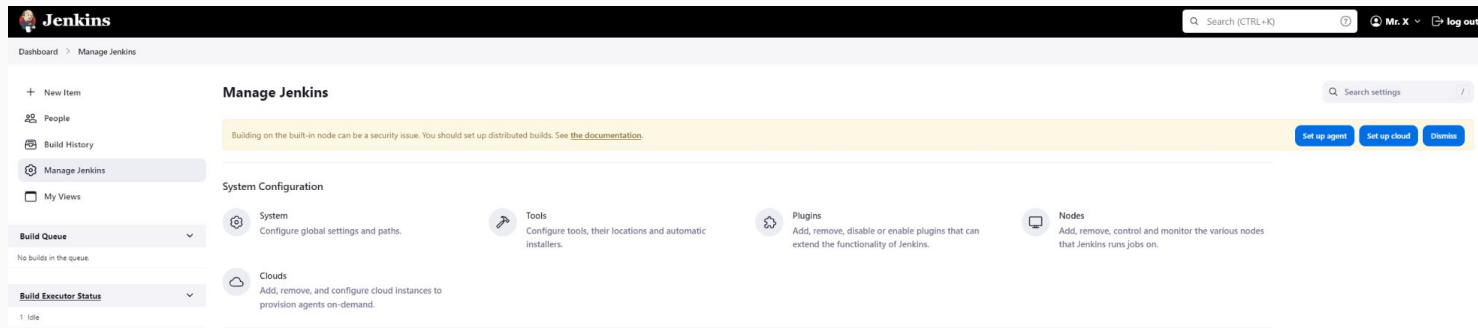
Выполнение Jenkins задачи на удаленном компьютере (при помощи SSH)



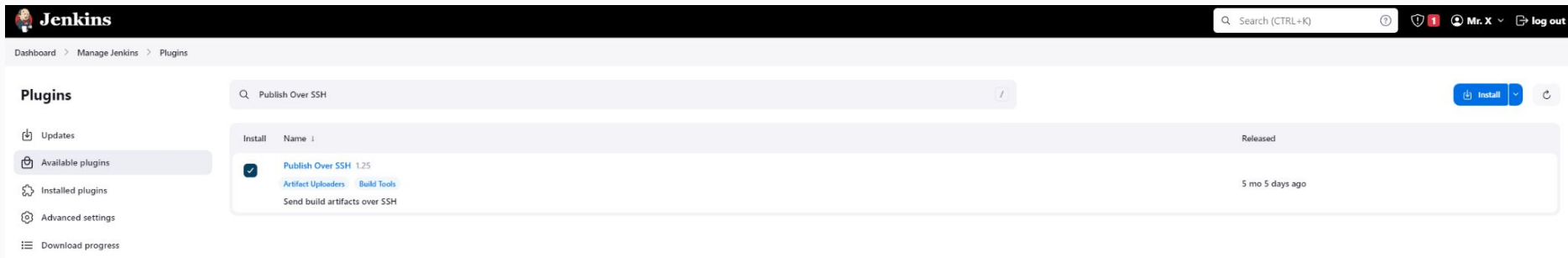
- В предыдущем разделе мы создали задачу, которая выполняется на том самом компьютере (пусть он и виртуальный), где установлен Jenkins. Очевидно, что в некотором количестве случаев нам не будет доступна такая роскошь - может случиться, что надо произвести какие-либо действия на удаленном компьютере. Представим, что есть некий удаленный MySQL сервер, на котором надо периодически делать резервные копии всех баз данных. Jenkins может помочь это осуществить - для этого он должен связаться с сервером и произвести необходимые действия.
- Для правильного осуществления вышеописанной задачи мы должны проинсталлировать для Jenkins специальный плагин, который умеет устанавливать соединения с помощью SSH протокола. Теоретически это можно осуществить при помощи обычных средств Jenkins, но с помощью плагина задача будет решена более удобным и элегантным способом.

# Установка плагина “Publish Over SSH”

- Для установки плагина нам надо перейти в раздел **“Manage Jenkins”** и выбрать подраздел **“Plugins”**:



- Затем слева выбрать пункт **“Available Plugins”** и в поисковой строке сверху написать *Publish Over SSH*. Потом поставить галочку в найденном варианте и нажать большую верхнюю кнопку **Install** - установка плагина будет начата:



## Завершение установки плагина

- После установки нового плагина нам следует перезапустить Jenkins, чтобы произведенные изменения вступили в силу. Однако если мы используем Jenkins, который установлен внутри Docker контейнера, то любой перезапуск приведет к тому, что контейнер будет остановлен. Чтобы решить эту проблему мы просто перезапустим сам Docker контейнер, содержащий Jenkins:

```
docker restart jenkins
```

Теперь мы можем использовать наш новый плагин “Publish Over SSH”!

# Эмуляция создания удаленного MySQL сервера

- Запустим новый Docker контейнер, который будет содержать работающий MySQL сервер. Для этого возьмем уже готовый образ **mysql:8.0.35-debian**, которому в качестве пароля для суперпользователя укажем слово **secret**:

```
docker run --name=mysql-ssh -d -e MYSQL_ROOT_PASSWORD=secret mysql:8.0.35-debian
```

- Когда мы запустили новый контейнер, мы должны подготовить его для удаленных соединений. Во-первых, для этого следует установить SSH сервер:

```
apt update  
apt install openssh-server
```

Во-вторых, следует его настроить. Самое главное - найти в конфигурационном файле **/etc/ssh/sshd\_config** настройку аутентификации с помощью публичного ключа и раскомментировать/включить ее:

```
PubkeyAuthentication yes
```

В самом конце надо не забыть запустить SSH сервер:

```
service ssh start
```

# Создание и настройка пользователя для удаленного доступа

- Теперь подготовим пользователя для удаленного доступа (мы можем использовать **root**, но это не очень безопасно, поэтому создадим нового - **user**):

```
# создание пользователя
useradd -m user

# назначение пароля пользователю
echo -e "secret\nsecret" | passwd user
```

- После создания пользователя нам надо где-нибудь (можно даже за пределами Docker контейнеров) создать публичный и закрытый ключ для SSH соединения:

```
ssh-keygen
```

- В самом конце мы должны перейти в контейнер, где работает MySQL сервер, в директории **/home/user/** создать директорию **.ssh**, в этой новой директории создать файл **authorized\_keys** и сохранить там созданный нами ранее публичный ключ.

# Подключение Jenkins к удаленному серверу (копирование закрытого ключа)

- Чтобы создать соединение с удаленным MySQL сервером (предположим, что его IP равен **172.17.0.3**), нам надо перейти в раздел **Manage Jenkins** -> **System** и пролистать страницу до блока **"Publish Over SSH"**. Затем в поле Key необходимо скопировать наш созданный ранее закрытый SSH ключ:

Publish over SSH

Jenkins SSH Key ?

Passphrase ?

Path to key ?

Key ?

```
AIIRFK50pi3rac4qoliz3Y5Vboq6PN8TAAAwQC6FIqG0v10Aelc80CDAw4TZYv5SP78nnv9
ijX+EbsSFZVApk8nR2wW5wON0hnIQa30PgLG71UuX9xcA7GSQ87a5SNT75QaQyEq/T18QP
RhzmJZBYy2iBeXbTnJh1SsXKtrvbmsl6lilYhVGZnhYITI1hS8kpv2ISbJ8skifgo9fjmU
Qkth9Gp45UbR5TmYhnMksFof6lhs1Ja+nw766itYp1i7Xrzqs64GA07J5E15U9Ki4fThgQ
/EDPykyUWR6gsAAAAUamVua2luc0A0NzM3YtQ3ZmVlQWQBAgMEBQYH
-----END OPENSSH PRIVATE KEY-----
```

# Подключение Jenkins к удаленному серверу (добавление сервера)

- Затем чуть ниже необходимо добавить сервер - нажимаем на кнопку **Add**, в появившемся новом блоке в поле **Name** вписываем **MySQL Server**, в поле **Hostname** - **172.17.0.3** (IP адрес сервера), в поле **Username** - **user** (имя нашего пользователя внутри контейнера, содержащего MySQL сервер), а в поле **Remote Directory** - **/** (разрешаем доступ ко всему компьютеру) В конце не надо забыть нажать кнопку **Save** (она находится в самом низу):

SSH Servers

SSH Server

Name ?

MySQL Server

Hostname ?

172.17.0.3

Username ?

root

Remote Directory ?

/

☐ Avoid sending files that have not changed ?

Advanced ▾

## Создание задачи для работы с удаленным сервером

- Для создания задачи мы идем стандартным путем - на главной странице выбираем пункт **New Item**, затем выбирает тип задания **Freestyle project**, а в качестве названия указываем *MySQL Database Backup*. В настройках проекта указывает, что задача будет выполняться периодически (**Build periodically**) и в качестве расписания ставим пять звездочек - \* \* \* \* \* (т.е. каждую минуту).
- Для установки непосредственного задания листаем страницу до блока **Build Step** и в выпадающем списке **"Add build step"** выбираем пункт **"Send files or execute commands over SSH"**. В открывшемся окне в разделе **SSH Server** в списке **Name** выбираем опцию MySQL Server, а в разделе **Transfer Set** в поле **Exec command** вписываем команду, которая должна удаленно сделать резервную копию всех баз данных:

```
mysqldump -uroot -psecret --all-databases >  
/home/user/backup-$(date +%Y_%m_%d_%H_%M_%S').sql
```

- После всех произведенных манипуляций нажимаем внизу большую синюю кнопку **Save**.



# Создание задачи для работы с удаленным сервером (визуализация)

Send files or execute commands over SSH ?

SSH Publishers

SSH Server

Name ?

MySQL Server

Advanced

Transfers

Transfer Set

Source files ?

Either Source files, Exec command or both must be supplied

Remove prefix ?

Remote directory ?


Exec command ?





`mysqldump -uroot -psecret --all-databases > /home/user/backup-$(date +%Y_%m_%d_%H_%M_%S).sql`

## Проблема при работе с удаленным сервером (открытое хранение реквизитов)

- Если мы все сделали правильно, наша система копирования должна работать отлично - в директории **/home/user** должны сохраняться соответствующие файлы. Однако при работе с соединением мы внутри самого задания открыто написали логин и пароль к MySQL серверу (root и secret). Таких вещей надо стараться избегать, и Jenkins может нам в этом помочь.
- Нам следует перейти в раздел **Manage Jenkins** -> **Credentials** и нажать на надпись **(global)**. Затем следует нажать на большую синюю кнопку вверху справа **Add Credentials**. В открывшемся окне в поле **Kind** выбираем опцию **Username with password**, **Scope** оставляем как есть, в поле **Username** вписываем слово **root** (логин базы данных), отмечаем галочкой опцию **Treat username as secret** (логин нигде не будет показан явно), а в поле **Password** вписываем слово **secret** (пароль базы данных). Затем в поле **ID** записываем название наших реквизитов, например, **MYSQL\_CREDENTIALS** и нажимаем синюю кнопку **Create** внизу.

# Создание реквизитов в Jenkins (визуализация)

 **Jenkins**

Search (CTRL+K)   1  Mr. X  log out

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

### New credentials

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

root

☒ Treat username as secret ?

Password ?

\*\*\*\*\*

ID ?

MYSQL\_CREDENTIALS

Description ?

Create

# Применение в задаче реквизитов Jenkins (создание переменных окружения)

- Для редактирования задания мы должны перейти на главную страницу, найти задание **MySQL Database Backup**, навести мышкой на название и в выпадающем списке выбрать пункт **Configure**. В результате мы попадем в наше задание.
- Далее мы должны пролистать до блока **Build Environment** и выбрать пункт **Use secret text(s) or file(s)**. Это необходимо, чтобы записать наши реквизиты (логин и пароль) в переменные окружения. В списке **Bindings** выбираем опцию **Username and password (separated)** - В открывшемся окне в поле **Username Variable** впишем **MYSQL\_LOGIN**, а в поле **Password Variable** впишем **MYSQL\_PASSWORD**. В списке **Credentials** следует указать **MYSQL\_CREDENTIALS** (те самые реквизиты, которые мы создали несколько ранее):

Build Environment

☐ Delete workspace before build starts

☒ Use secret text(s) or file(s) ?

Bindings

Username and password (separated) ?

Username Variable ?

MYSQL\_LOGIN

Password Variable ?

MYSQL\_PASSWORD

Credentials ?

☒ Specific credentials ☐ Parameter expression

MYSQL\_CREDENTIALS

+ Add

# Применение в задаче реквизитов Jenkins (внедрение переменных окружения)

- Чтобы внедрить созданные переменные окружения, нам следует пролистать страницу до того места внутри блока **SSH Server**, где ранее мы прописывали основную команду для выполнения задачи. Теперь вместо явного логина и пароля указываем переменные окружения. Затем сохраняем задание:

```
mysqldump -u$MYSQL_LOGIN -p$MYSQL_PASSWORD --all-databases > /home/user/backup-$(date +%Y_%m_%d_%H_%M_%S').sql
```

Exec command ?

```
mysqldump -u$MYSQL_LOGIN -p$MYSQL_PASSWORD --all-databases > /home/user/backup-$(date +%Y_%m_%d_%H_%M_%S').sql
```

- Если мы все сделали правильно, наши резервные копии продолжают создаваться, но уже более правильным способом.

# Взаимодействие Jenkins и GitHub

- На данный момент самым популярным хранилищем проектов является сайт GitHub (<https://github.com>), хранящий удаленные репозитории. Само собой, при помощи этого сайта и происходит разработка проектов - разработчики загружают изменения в определенную ветку в контексте некоего репозитория, а затем главный разработчик или администратор вставляет все изменения в основную ветку и выгружает самую свежую версию на боевой сервер.
- Попытаемся автоматизировать процесс обновления проекта, находящегося на GitHub, при помощи Jenkins. Предположим, что у нас есть простейший проект по разработке статического сайта (используются только HTML, CSS и JavaScript). Мы хотим, чтобы при обновлении определенной ветки GitHub репозитория, наш Jenkins брал бы последнюю версию проекта, определенным образом тестировал бы ее и, если тесты прошли успешно, выгружал бы эти изменения на условно боевой сервер.

## Подготовка Docker контейнера с сервером для сайта

- Для начала создадим с помощью Docker контейнера эмуляцию удаленного боевого сайта, который должен содержать сервер с необходимым нам сайтом:

```
docker run --name=web-server -d -p 80:80 nginx:1.25.3
```

- Когда мы запустили новый контейнер, мы должны подготовить его для удаленных соединений. Во-первых, для этого следует установить SSH сервер:

```
apt update  
apt install openssh-server
```

Во-вторых, следует его настроить. Самое главное - найти в конфигурационном файле **/etc/ssh/sshd\_config** настройку аутентификации с помощью публичного ключа и раскомментировать/включить ее:

```
PubkeyAuthentication yes
```

В самом конце надо не забыть запустить SSH сервер:

```
service ssh start
```



## Подготовка пользователя для удаленного доступа

- Далее нам надо где-нибудь (можно даже за пределами Docker контейнеров) создать публичный и закрытый ключ для SSH соединения:

```
ssh-keygen
```

- В самом конце мы должны перейти в контейнер, где работает MySQL сервер, в директории **/root/** создать директорию **.ssh**, в этой новой директории создать файл **authorized\_keys** и сохранить там созданный нами ранее публичный ключ (в этом случае не будем тратить время на создание дополнительного пользователя, но в реальной жизни это будет правильнее и безопаснее).
- Теперь мы должны добавить SSH сервер и его реквизиты в разделе **Manage Jenkins** -> **System**. Это мы уже делали ранее, поэтому подробно объяснять это здесь не будем. Однако условимся, что сервер будет называться **Web Server**.

# Создание GitHub репозитория


- Пришла пора создать наш GitHub репозиторий. Создание надо произвести стандартно, однако условимся, что репозиторий должен быть закрытым:

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

*Required fields are marked with an asterisk (\*).*

Owner \*

 itlat-mysql ▾

 /


Repository name \*

site


✔ site is available.

Great repository names are short and memorable. Need inspiration? How about [fuzzy-disco](#) ?

Description (optional)

☐  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☒  **Private**

You choose who can see and commit to this repository.

Initialize this repository with:

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None ▾

# Использование публичного ключа для подключения к GitHub репозиторию

- Теперь нам следует подключить наш публичный ключ к нашему GitHub профилю. Возьмем ранее созданный ключ, перейдем в раздел **Settings -> SSH and GPG keys** и нажмем зеленую кнопку **New SSH key**. В поле Title вписываем любое название, а в поле Key - наш публичный ключ. Затем подтвердим наши действия нажатием зеленой кнопки **Add SSH key**:

Public profile

Account

Appearance

Accessibility

Notifications

Access

Billing and plans

Emails

Password and authentication

Sessions

**SSH and GPG keys**

Organizations

Enterprises

Moderation

Code, planning, and automation

## Add new SSH Key

Title

Development Key

Key type

Authentication Key

Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQGCj4N9TAFijYnc1LpUicnmdcHBOoFwnlWC0MLCLFhOBD1KF95C8+1dWhRAivo+2zhtjFymg0U5ZB4sBhELZ
pYcFmxyPfqNRoP6FzjkaF7i/Ytbe0CAEZhsUO7KTqkOo4o6V8cRGf6cZb3LdjL4M0oEiYLC882k6i+ZuV0ODDDadbb2jyCEgiLwIOJWxbLyG5x9YsnoO69iL
vdlQutnJ7k9282sXqf1PLzzVwSRQM7Nj8AVIhTinODmw9cKLP7RnH1iuKk992ksWiKfPZKYpRPQDWnb3L8dCdcMVqbqyt8NvwNtgmVqANk7i+kfTh
0hRyjNzJHnntCDznXUj6T8wzosi2hoUxd0eiAaX11aZnLtFUFYVSgFcMg3tU+t2DcdBiNoVtZg6bORWze0M+nyzWNOSVxzRV62lqUcAo+zwmR8etTBrZ
FV3eXbgBOcthqCQ3iLOXp/Yx9fFE6WslGvV074MyNCXHGz91oQ8zbvixril/eEclgGeP16OPEtx1H89E= jenkins@4737a87fee9d
```

Add SSH key

# Создание задания (попытка подключение репозитория)

- Пришло время создать наше новое задание - на главной странице выбираем пункт **New Item**, затем выбирает тип задания **Freestyle project**, а в качестве названия указываем *GitHub Project*. Чуть ниже в поле Branch Specifier (blank for 'any') вписываем название нашей ветки на GitHub - **\*/main**.
- Далее нам надо найти блок **Source Code Management** и выбрать опцию **Git**. В открывшемся окне в поле **Repository URL** вставим SSH ссылку на наш репозиторий. Однако мы увидим, что доступ запрещен, т.к. наш репозиторий закрыт.

Source Code Management

☐ None

☒ Git ?

Repositories ?

Repository URL ?

git@github.com:itlat-mysql/site.git

❗ Failed to connect to repository : Command "git ls-remote -h -- git@github.com:itlat-mysql/site.git HEAD" returned status code 128:

stdout:

stderr: Host key verification failed.

fatal: Could not read from remote repository.

Please make sure you have the correct access rights  
and the repository exists.

Credentials ?

- none -

+ Add

## Создание задания (подключение репозитория через закрытый ключ)

- Для подключения закрытого репозитория нам надо создать реквизиты, которые будут содержать закрытый ключ, являющийся парой к уже загруженному публичному ключу на GitHub. Для этого не закрывая нашего нынешнего раздела идет в раздел **Manage Jenkins -> Security**, устанавливаем настройке **Host Key Verification Strategy** опцию **Accept first connection** и сохраняем наши изменения. Затем возвращаемся к нашему предыдущему разделу и нажимаем кнопку Add под полем **Credentials**.
- В открывшемся окне в списке **Kind** следует выбрать опцию **SSH Username with private key**, в поле **ID** вписать **GITHUB\_CREDENTIALS**, в поле **Username** вставить имя нашего GitHub пользователя, для безопасности пометить галочкой поле **Treat username as secret**, а в поле **Private Key** скопировать наш закрытый ключ. Затем подтверждаем наш выбор нажатием кнопки **Add**. Окошко автоматически закроется - после этого в списке **Credentials** нам следует выбрать опцию **GITHUB\_CREDENTIALS**. После этого ошибка доступа должна пропасть.

# Создание задания (подключение репозитория) (визуализация)

Kind

SSH Username with private key

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

ID ?

GITHUB\_CREDENTIALS

Description ?

Username

itlat-mysql

☒ Treat username as secret ?

Private Key

☒ Enter directly

Key

Enter New Secret Below

```
UKtSGP4SUDR5IMV7HKKSTFOT0IM5L1a#IW/00LYP11/X/ZUSQ49A8/JSELSULPK1aT1M8Q
/EDPkyUwR6gsAAAAUamVua2Iuc0A0NzI3YTe3ZmV1OWQBAgMEBOYH
-----END OPENSSH PRIVATE KEY-----
```

Passphrase

Add Cancel

## Создание задания (расписание выполнения задания)

- Теперь мы должны указать тот способ, посредством которого наше задание будет запускаться - для этого нам надо пролистать нашу страницу до блока **Build Triggers**. К сожалению, нам недоступна опция **GitHub hook trigger for GITScm polling**, которая позволяет самому сайту GitHub при определенных условиях (например при обновлении репозитория) запускать Jenkins задания - так случилось потому, что у нас нет публичного IP-адреса. Однако мы можем воспользоваться опцией **Poll SCM**, которая по определенному графику будет запрашивать возможные изменения у GitHub - если изменения были, задание будет запущено. В качестве расписания поставим пять звезд (ежеминутно):

### Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts) ?
- ☐ Build after other projects are built ?
- ☐ Build periodically ?
- ☐ GitHub hook trigger for GITScm polling ?
- ☒ Poll SCM ?

### Schedule ?

\* \* \* \* \*

# Создание задания (первый шаг - очистка старых файлов)

- Далее мы должны пролистать страницу до блока **Build Steps** и в списке **Add build step** выбрать опцию **Send files or execute commands over SSH**. Логично, что на первом шаге нам надо будет очищать старую директорию от находящихся там файлов. Для этого в поле SSH Server -> Name выберем опцию Web Server, а в поле Exec command пропишем следующую команду удаления файлов:

```
rm -rf /usr/share/nginx/html/*
```

The screenshot shows a configuration form for the 'Send files or execute commands over SSH' step. The form is titled 'Send files or execute commands over SSH' with a help icon. Below the title, there's a section for 'SSH Publishers' with a sub-section 'SSH Server'. In the 'SSH Server' section, the 'Name' field is set to 'Web Server'. There's an 'Advanced' dropdown menu. Below that, there's a 'Transfers' section with a sub-section 'Transfer Set'. The 'Transfer Set' section has three input fields: 'Source files', 'Remove prefix', and 'Remote directory'. At the bottom, there's an 'Exec command' field containing the command 'rm -rf /usr/share/nginx/html/\*'.



# Создание задания (второй шаг - копирование новых файлов)

- Теперь в списке **Add build step** мы должны опять выбрать опцию **Send files or execute commands over SSH**. В поле **SSH Server -> Name** также опять выберем опцию Web Server, в поле **Source files** пропишем \*/ (т.е. все файлы репозитория), а в поле **Remote directory** вставим путь **/usr/share/nginx/html/** (туда будут скопированы файлы репозитория):

The screenshot shows the configuration interface for the 'Send files or execute commands over SSH' build step. The interface is divided into several sections:

- SSH Publishers:** A section containing a list of SSH servers. The 'SSH Server' section is expanded, showing a dropdown menu for 'Name' with 'Web Server' selected.
- Advanced:** A button labeled 'Advanced' with a dropdown arrow.
- Transfers:** A section containing a 'Transfer Set' configuration.
  - Source files:** A text input field containing the wildcard pattern \*/.
  - Remove prefix:** An empty text input field.
  - Remote directory:** A text input field containing the path /usr/share/nginx/html/.
  - Exec command:** An empty text input field.

## Создание задания (завершение)

- Спустя примерно минуту после того, как мы нажмем кнопку **Save**, Jenkins запустит наше новое задание, которое подгрузит файлы из GitHub репозитория и загрузит их на удаленный сервер. Также Jenkins будет каждую минуту проверять репозиторий на изменения - если они будут замечены, задание будет запущено вновь.
- Нам вовсе необязательно ограничиваться пересылкой файлов. Например, мы в качестве первого шага задания можем подставить тест, который будет искать определенные слова в файлах репозитория. Если слова не будут найдены, то будет возвращена ошибка, которая остановит выполнение задания. Ниже представлен пример поиска словосочетания **Travel Agency** в файле **index.html**:

```
appearances=`grep 'Travel Agency' index.html | wc -l`  
if [ "$appearances" -ne 0 ]; then  
    echo 'Project is valid.'  
    exit 0  
else  
    echo 'Project is invalid.'  
    exit 1  
fi
```

# Взаимодействие Jenkins и Docker

- Те знания, которыми мы сейчас обладаем, уже позволяют нам подключаться к репозиторию, производить загрузку и тестирование любых приложений (CI), а также выгрузку этих приложений на боевой сервер (CD). Однако надо понимать, что приложения могут быть совершенно разными, использовать разные версии одних и тех же библиотек для разных случаев, а также требовать различного окружения.
- Обозначенная проблема решается при помощи уже хорошо знакомой нам программы Docker. Jenkins умеет работать с Docker и даже имеет специальный плагин для работы с контейнеризованными приложениями. В самом наилучшем случае мы должны запустить Docker на удаленном компьютере, а затем подключить этот компьютер в качестве агента для Jenkins. Однако пока мы не можем позволить себе такой роскоши, поэтому установим Docker на том же самом компьютере, на котором работает сам Jenkins. Это не очень правильно с точки зрения безопасности, но для начала обучения это вполне приемлемый вариант.

# Установка Docker внутри контейнера

- Сначала зайдём в наш Jenkins контейнер:

```
docker exec -it jenkins bash
```

- Затем внутри контейнера установим Docker и все его зависимости:

```
apt update && apt install -y lsb-release
```

```
curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \  
https://download.docker.com/linux/debian/gpg
```

```
echo "deb [arch=$(dpkg --print-architecture) \  
signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \  
https://download.docker.com/linux/debian \  
$(lsb_release -cs) stable" > /etc/apt/sources.list.d/docker.list
```

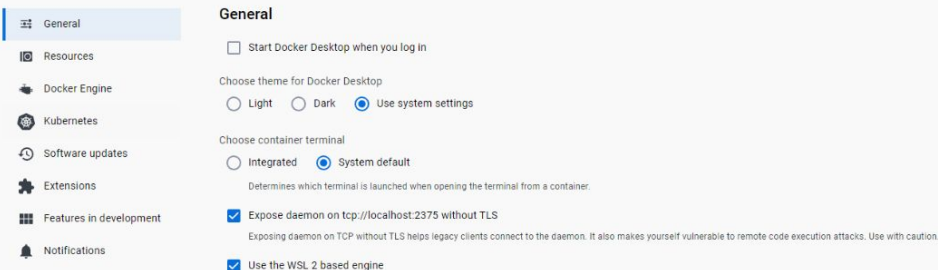
```
apt update && apt install -y docker-ce-cli docker-compose
```

# Использование основного сокета Docker демона внутри контейнера

- Мы успешно установили Docker внутри контейнера, но он не может полноценно работать, т.к. внутри отсутствует т.н. Docker демон, который отвечает за правильную работу всего приложения. Однако при создании Jenkins контейнера мы указали переменную **`DOCKER_HOST=tcp://host.docker.internal:2375`**, которая позволит контейнеру обратиться за демоном к основной операционной системе. Единственное, что нам надо сделать - разрешить использование этого демона. Для этого перейдем в настройки Docker Desktop (***Settings -> General***), включим настройку **Expose daemon on tcp://localhost:2375 without TLS**, а затем перезагрузим весь Docker Desktop, чтобы изменения вступили в силу:

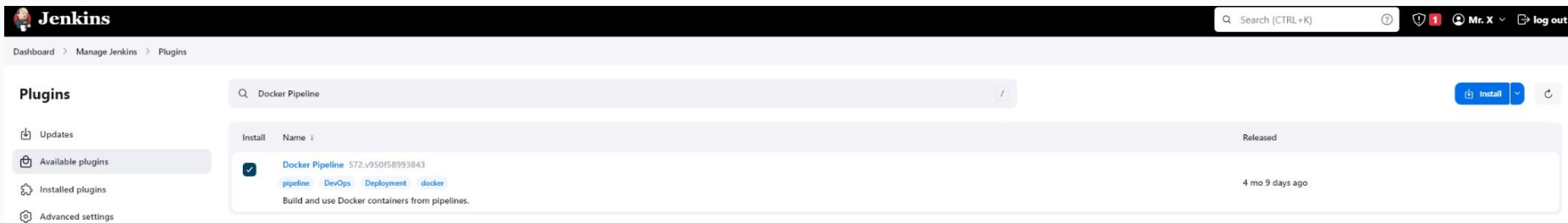
Settings [Give feedback](#)

×



# Установка плагина Docker Pipeline

- Теперь для работы с Docker нам надо установить плагин **Docker Pipeline** (хотя дальше мы увидим, что это не является обязательным условием). В очередной раз нам надо перейти в раздел “**Manage Jenkins**” и выбрать подраздел “**Plugins**”. Затем слева выбрать пункт “**Available Plugins**” и в поисковой строке сверху написать **Docker Pipeline**. Потом поставим галочку в найденном варианте и нажмем большую верхнюю кнопку **Install** - установка плагина будет начата:



- После установки плагина надо не забыть перезапустить Jenkins контейнер, чтобы стал доступен функционал нового плагина:

```
docker restart jenkins
```

# Создание задачи для работы с Docker

- Перейдем в раздел **New Item** и создадим задание, где будет использоваться Docker. Сразу же заметим, что в этом случае используется не такой тип задания, который был использован ранее (**Freestyle project**), а новый - **Pipeline**. Далее мы подробно разберем его особенности, а сейчас введем название задачи - **Docker Example**, выберем тип **Pipeline** и нажмем большую синюю кнопку **OK** внизу:

Enter an item name

» Required field



## Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.



## Pipeline

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.



## Применение конвейера при описании задач (введение)

- Основное отличие типа **Pipeline** от **Freestyle project** состоит в том, что шаги задачи описываются при помощи языка программирования Groovy. Код задачи должен быть расположен в секции **Pipeline** и обрамлен в конструкцию **pipeline { }** (есть еще один императивный способ с помощью конструкции **node**, который позволяет использовать продвинутые конструкции Groovy). Внутри конструкции указывается **agent**, с помощью которого будет выполняться задача - если указать **any** - Jenkins попытается задействовать любые доступные агенты, если указать **none**, то тогда надо будет указать агента для каждого конкретного шага задачи отдельно. Кроме того, может быть указан конкретный агент, например, **docker** и образ, где будет запускаться код. Если нам не нужно ничего специфического, то можем указать **any**.
- Сами шаги задачи описываются внутри конструкции **stages**. Для каждого шага прописывается свой блок **stage** с описанием, и внутри **stage** вставляется еще один блок **steps**, где можно писать конкретные команды (команд может быть сколько угодно). Например, **echo 'Hello'** выведет слово **Hello** в консоль, а **sh 'touch index.html'** создаст файл **index.html** при помощи знакомого нам языка командной строки в операционных системах на ядре Linux.

# Применение конвейера при описании задач (пример)

```
pipeline {
  agent any

  stages {
    stage('Initialization') {
      steps {
        echo 'Hello!'
        sh 'ls -la'
      }
    }
    stage('Build') {
      steps {
        sh 'echo "Hello, People!" > index.html'
        sh 'wc -l index.html'
      }
    }
    stage('Finalization') {
      steps {
        sh 'cat index.html'
        echo 'Finished!'
      }
    }
  }
}
```

# Применение конвейера при описании задач (императивный синтаксис)

```
node {  
  stage('Initialization') {  
    echo 'Hello!'  
    sh 'ls -la'  
  }  
  stage('Build') {  
    sh 'echo "Hello, People!" > index.html'  
    sh 'wc -l index.html'  
  }  
  stage('Finalization') {  
    sh 'cat index.html'  
    echo 'Finished!'  
  }  
}
```

## Применение конвейера при описании задач (работа с агентом docker)

- Как уже говорилось, для работы с Docker нам следует применить конструкцию с **agent**. Внутри конструкции надо вписать слово **image**, а затем в кавычках указать тот Docker образ, который нам нужен для запуска нашего приложения, например, если мы работаем с Python, это может быть **'python:3.12-bookworm'**.
- Надо четко понимать, что при таком подходе все действия будут происходить внутри контейнера, который будет автоматически построен на базе образа, указанного в предыдущем пункте. Кроме того, нам будут доступны все технологии, который входят в поставку этого образа (в нашем случае Python и его инструменты, например, **pip**). Также важно знать, что контейнер будет существовать только во время работы Jenkins задачи. Сразу же после выполнения задачи контейнер будет автоматически удален.
- В контексте этой задачи также будет полезно узнать, откуда мы можем брать данные нашего проекта. Логично, что из некоего репозитория на сайте GitHub. Для этого внутри первой задачи нам надо применить команду **git**, которой в качестве параметра **branch** надо указать ветку нашего репозитория, в качестве **url** - ссылку на наш репозиторий. Если наш репозитория закрытый, надо применить еще один параметр - **credentialsId**, которому надо подставить идентификатор реквизитов нашего репозитория.

# Применение конвейера при описании задач (пример работы с Docker)

```
pipeline {
  agent {
    // установка образа docker
    docker {
      image 'python:3.12-bookworm'
      args '--name=python_container'
    }
  }
  stages {
    stage('Load project from GitHub') {
      steps {
        // загрузка проекта из GitHub репозитория
        git branch: 'main', credentialsId: 'GITHUB_CREDENTIALS', url: 'git@github.com:itlat-mysql/blog.git'
      }
    }
    stage('Project Building') {
      steps {
        // установка зависимостей python
        sh 'pip install -r requirements.txt'

        // создание конфигурационного файла и заполнение его тестовыми данными
        sh 'cp .env.example .env'
        sh 'sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=my-secret-key/" .env'
        sh 'sed -i "s/DB_NAME=/DB_NAME=test/" .env'
        sh 'sed -i "s/DB_ENGINE=/DB_ENGINE=django.db.backends.sqlite3/" .env'
      }
    }
    stage('Project Testing') {
      steps {
        // запуск тестирования проекта
        sh 'python manage.py test'
      }
    }
  }
}
```

## Применение конвейера при описании задач (Docker образ для конкретного шага)

```
pipeline {
  agent any
  stages {
    stage('Test Project') {
      agent {
        // установка образа docker
        docker {
          image 'python:3.12-bookworm'
          args '--name=python_container'
        }
      }
      steps {
        // загрузка проекта из GitHub репозитория
        git branch: 'main', credentialsId: 'GITHUB_CREDENTIALS', url: 'git@github.com:itlat-mysql/blog.git'

        // создание конфигурационного файла и заполнение его тестовыми данными
        sh 'pip install -r requirements.txt'
        sh 'cp .env.example .env'
        sh 'sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=my-secret-key/" .env'
        sh 'sed -i "s/DB_NAME=/DB_NAME=test/" .env'
        sh 'sed -i "s/DB_ENGINE=/DB_ENGINE=django.db.backends.sqlite3/" .env'
        sh 'cat .env'

        // запуск тестирования проекта
        sh 'python manage.py test'
      }
    }
  }
}
```

# Применение конвейера при описании задач (Docker и императивный синтаксис)

```
node {
  docker.image("python:3.12-bookworm").inside {
    stage('Load project from GitHub') {
      git branch: 'main', credentialsId: 'GITHUB_CREDENTIALS', url: 'git@github.com:itlat-mysql/blog.git'
    }

    stage('Project Building') {
      sh 'pip install -r requirements.txt'
      sh 'cp .env.example .env'
      sh 'sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=my-secret-key/" .env'
      sh 'sed -i "s/DB_NAME=/DB_NAME=test/" .env'
      sh 'sed -i "s/DB_ENGINE=/DB_ENGINE=django.db.backends.sqlite3/" .env'
    }

    stage('Project Testing') {
      sh 'python manage.py test'
    }
  }
}
```

Настройка pipeline (конвейера) для  
развертывания и обновления  
приложений



- В этом разделе мы продолжим рассматривать настройку конвейера для сборки приложений, но уже не будет концентрироваться на технологии Docker. Как мы помним, процесс CI и CD состоит из двух главных частей - во-первых, тестирование после обновления репозитория и, во-вторых, выкладывание обновления на боевой сервер. Кроме того, не надо забывать, что иногда необходимо произвести первичное развертывание приложения, что может потребовать дополнительных манипуляций.
- Еще один важный момент, который мы до этого не упоминали, связан с уведомлением разработчика или администратора об успехе или неудаче при автоматической сборке проекта. Мы рассмотрим классический случай - отправку сообщения на электронную почту.
- В числе прочего важно рассмотреть логику восстановления после неудачного обновления боевого сервера. Обратим внимание на резервное копирование базы данных и возврат к предыдущей версии программы.

## Подключение SMTP сервера

- Чтобы использовать отправку сообщений в процессе выполнения задач, нам нужен плагин **Email Extension Plugin**. К счастью, он был установлен при установке Jenkins. Теперь нам нужны параметры какого-либо SMTP сервера, который позволит нам отправлять сообщения. Если нет подходящий, можно использовать сервис <https://mailtrap.io/> (нужный тестовый функционал там будет доступен бесплатно).
- Далее мы должны создать реквизиты, которые будут включать в себя пароль и логин SMTP сервера. Для этого перейдем в раздел **Manage Jenkins -> Credentials** и создадим реквизиты с ID равным **SMTP\_CREDENTIALS**, типом **Username with password** и теми самими паролем и логином от SMTP сервера.
- Затем переходим в раздел **Manage Jenkins -> System** и пролистываем до блока **Extended E-mail Notification**. В поле **SMTP server** вписываем доменный адрес нашего SMTP сервера, в поле **SMTP Port** - порт SMTP сервера. После этого нажимаем на кнопку **Advanced**, в списке **Credentials** выбираем опцию **SMTP\_CREDENTIALS**, а среди типов соединения - необходимый вариант, например, **Use TLS**.

# Подключение SMTP сервера (визуализация)

## Extended E-mail Notification

### SMTP server

sandbox.smtp.mailtrap.io

### SMTP Port

2525

Advanced ^

 Edited

### Credentials

SMTP\_CREDENTIALS



+ Add ▾

☐ Use SSL

☒ Use TLS

☐ Use OAuth 2.0

### Advanced Email Properties

## Подключение логики отправки эл. почты в конвейер

- Для настройки самой логики в нашем конвейере идеально подходит блок **post**, который мы разместим в конце нашего основного блока **pipeline**. Внутри этого блока можно разместить несколько других блоков, которые умеют реагировать на разные результаты выполнения задачи. Нас интересуют два из них - **success** (успешное выполнение задачи) и **failure** (неудачное выполнение задачи).
- Внутри каждого из противоположных блоков (**success** и **failure**) мы должны отправить соответствующую логику отправки электронной почты. Для этого используется команда **emailtext**. В параметр **body** следует вписать необходимое нам сообщение, в параметр **to** - адресата сообщения, а в параметр **subject** - тему сообщения.
- Когда мы полностью реализуем вышеупомянутую логику, то мы полностью (хотя и в несколько примитивном виде) завершим **Continuous Integration (CI)** часть развертывания приложения.

# Подключение логики отправки эл. почты в конвейер (пример)

```
pipeline {
  agent {
    docker {
      image 'python:3.12-bookworm'
      args '--name=python_container -u root'
    }
  }
  environment {
    EMAIL_TO = 'receiver@gmail.com'
  }
  stages {
    stage('Load project from GitHub') {
      steps {
        git branch: 'main', credentialsId: 'GITHUB_CREDENTIALS', url: 'git@github.com:itlat-mysql/blog.git'
      }
    }
    stage('Project Building') {
      steps {
        sh 'pip install -r requirements.txt'
        sh 'cp .env.example .env'
        sh 'sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=my-secret-key/" .env'
        sh 'sed -i "s/DB_NAME=/DB_NAME=test/" .env'
        sh 'sed -i "s/DB_ENGINE=/DB_ENGINE=django.db.backends.sqlite3/" .env'
      }
    }
    stage('Project Testing') {
      steps {
        sh 'python manage.py test'
      }
    }
  }
  post {
    success {
      emailxtext body: 'Build succeeded!', to: "${EMAIL_TO}", subject: 'Build succeeded: $PROJECT_NAME - #${BUILD_NUMBER}'
    }
    failure {
      emailxtext body: 'Build failed!', to: "${EMAIL_TO}", subject: 'Build failed: $PROJECT_NAME - #${BUILD_NUMBER}'
    }
  }
}
```

# Подключение логики отправки эл. почты в конвейер (императивный пример)

```
node {
  try {
    docker.image("python:3.12-bookworm").inside("-u root") {
      stage('Load project from GitHub') {
        git branch: 'main', credentialsId: 'GITHUB_CREDENTIALS', url: 'git@github.com:itlat-mysql/blog.git'
      }

      stage('Project Building') {
        sh 'pip install -r requirements.txt'
        sh 'cp .env.example .env'
        sh 'sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=my-secret-key/" .env'
        sh 'sed -i "s/DB_NAME=/DB_NAME=test/" .env'
        sh 'sed -i "s/DB_ENGINE=/DB_ENGINE=django.db.backends.sqlite3/" .env'
      }

      stage('Project Testing') {
        sh 'python manage.py test'
      }
    }
  } finally {
    def currentResult = currentBuild.result ?: 'SUCCESS'
    def emailTo = 'receiver@gmail.com'

    if (currentResult == 'SUCCESS') {
      emailx body: 'Build succeeded!', to: "${emailTo}", subject: 'Build succeeded: $PROJECT_NAME - #$BUILD_NUMBER'
    } else {
      emailx body: 'Build failed!', to: "${emailTo}", subject: 'Build failed: $PROJECT_NAME - #$BUILD_NUMBER'
    }
  }
}
```

# Конвейер для развертывания приложения (введение)


- Пришла пора написать наш первый полноценный конвейер, который будет включать в себя часть CD (Continuous Delivery) - выкладывание обновлений на боевой сервер (предположим, что на нем уже установлен Docker и SSH сервер). Но для начала надо описать конвейер для первичного развертывания приложения.
- Укажем также еще один важный момент. На самом деле Django приложение состоит из минимум трех частей - во-первых, код на Python, которому нужен сервер приложения (мы будем использовать Gunicorn), во-вторых, база данных, где будет храниться информация приложения (мы будем использовать СУБД MySQL) и, в-третьих, web-сервер, который будет передавать запросы из сети внутрь сервера приложения (мы применим Nginx). Все это говорит нам о том, что нам нужны сразу три Docker контейнера, связанные между собой посредством файла docker-compose.yml, который уже должен быть включен в сам проект.
- Кроме того, проект достиг такого уровня сложности, что описание самого конвейера на языке Groovy (Jenkinsfile) также следует вкладывать в сам проект. Сам Jenkins поддерживает такой подход - данный файл будет применен внутри задания и его команды будут успешно выполнены.

## Подготовка к развертыванию (создание необходимых реквизитов)



- Вначале нам надо создать несколько реквизитов (Credentials). Для начала нам нужен закрытый SSH ключ имя пользователя для подключения к удаленному серверу. Создадим его и назовем ***SITE\_CREDENTIALS***. После этого надо не забыть сходить на удаленный сервер и внутри пользовательской директории в директории ***.ssh*** в файл ***authorized\_keys*** добавить наш публичный ключ.
- Затем следует создать еще два варианта текстовых реквизитов - ***SITE\_USER*** (имя пользователя удаленного сервера) и ***SITE\_ADDRESS*** (IP адрес или доменное имя удаленного сервера).
- Также необходимо создать вариант файлового реквизита, куда мы загрузим файл ***.env*** с боевыми настройками для сайта. Назовем этот вариант ***SITE\_ENV\_FILE***.
- Ранее мы должны были уже создать два варианта реквизитов - ***SMTP\_CREDENTIALS*** (для отправки сообщений по электронной почте) и ***GITHUB\_CREDENTIALS*** (для подгрузки данных из GitHub). Если это не сделано, следует вернуться к соответствующим разделам и создать эти реквизиты.



# Подготовка к развертыванию (создание необходимых реквизитов) (пример)

 Jenkins

Search (CTRL+K)













Mr. X   log out

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

### Global credentials (unrestricted)

+ Add Credentials

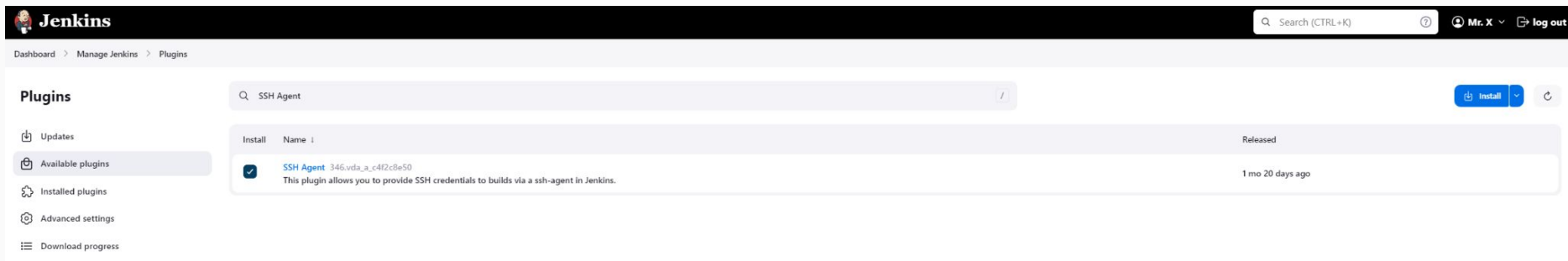
Credentials that should be available irrespective of domain specification to requirements matching.

| ID   | Name               | Kind                          | Description   |
|--|--------------------|-------------------------------|---|
|  <a href="#">SITE_CREDENTIALS</a>   | SITE_CREDENTIALS   | SSH Username with private key |  |
|  <a href="#">GITHUB_CREDENTIALS</a> | GITHUB_CREDENTIALS | SSH Username with private key |  |
|  <a href="#">SITE_USER</a>          | SITE_USER          | Secret text                   |  |
|  <a href="#">SITE_ADDRESS</a>       | SITE_ADDRESS       | Secret text                   |  |
|  <a href="#">SITE_ENV_FILE</a>      | .env               | Secret file                   |  |
|  <a href="#">SMTP_CREDENTIALS</a>   | SMTP_CREDENTIALS   | Username with password        |  |

Icons: S M L

# Подготовка к развертыванию (установка плагина для работы с SSH)

- Для того, чтобы нам было удобно и безопасно работать с удаленным сервером, следует установить еще один специальный плагин под названием SSH Agent. Сделаем это как обычно - перейдем в раздел **“Manage Jenkins”** и выберем подраздел **“Plugins”**. Затем слева выбрать пункт **“Available Plugins”** и в поисковой строке сверху написать *SSH Agent*. Потом поставим галочку в найденном варианте и нажмем большую верхнюю кнопку **Install** - установка плагина будет начата:



- После установки плагина надо не забыть перезапустить Jenkins контейнер, чтобы стал доступен функционал нового плагина:

```
docker restart jenkins
```

## Подготовка к развертыванию (настройка удаленного сервера)

- Наш удаленный сервер, на котором и будет находиться сайт, также требует дополнительной настройки. Во-первых, он тоже должен иметь доступ на GitHub, поэтому внутри пользовательской директории в директории **.ssh** надо либо создать новый закрытый ключ, а его публичную версию загрузить на GitHub, либо скопировать в эту директорию ключ, который уже используется на GitHub.
- Во-вторых, чтобы убрать возможные проблемы с подключением к GitHub, в той же директории **.ssh** следует создать файл **known\_hosts** и добавить туда ключи GitHub. Ключи можно отыскать по этой ссылке:  
<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/githubs-ssh-key-fingerprints>

## Задание по первоначальному развертыванию сайта

- Очевидно, что нам надо создать сайт, снабдить его нужными настройками (например, ***docker-compose.yml***) для запуска в Docker контейнерах и Jenkins окружении и выложен на GitHub. К счастью, сайт уже написан и будет предоставлен для использования. Теперь переходим в Jenkins и создаем задачу с названием ***"Site Initial Deploy"*** и типом ***Pipeline***.
- Находим внутри задания блок ***Pipeline*** и в списке ***Definition*** выбираем опцию ***Pipeline script from SCM***. В открывшемся окне в поле ***SCM*** выбираем ***Git***, в поле ***Repository URL*** вписываем ссылку на GitHub репозиторий для SSH соединения, в поле ***Credentials*** выбираем опцию с заранее созданными GitHub реквизитами (***GITHUB\_CREDENTIALS***), в поле ***Branch Specifier (blank for 'any')*** вписываем ветку нашего репозитория, например, ***\*/main***. Затем в поле ***Script Path*** вписываем путь к файлу, где находится файл описания конвейера для этого задания Jenkins (мы написали его заранее) в контексте GitHub репозитория - в нашем случае это будет ***jenkins/init***. Затем сохраняем задания и запускаем его - если все было настроено правильно, сайт будет автоматически развернут на целевом удаленном сервере.

# Задание по первоначальному развертыванию сайта (пример)

Pipeline

Definition

Pipeline script from SCM

SCM ?

Git

Repositories ?

Repository URL ?

git@github.com:titlat-mysql/blog.git

Credentials ?

GITHUB\_CREDENTIALS

+ Add

Advanced

Add Repository

Branches to build ?

Branch Specifier (blank for 'any') ?

\*/main

Add Branch

Repository browser ?

(Auto)

Additional Behaviours

Add

Script Path ?

jenkins/init

# Задание по первоначальному развертыванию сайта (jenkins/init) (1 часть)

```
pipeline {
    agent any

    environment {
        SITE_DIR = '/home/user/site'
        BACKUP_DIR = '/home/user/backup'
        DOCKER_COMPOSE_FILE = '/home/user/site/docker-compose.yml'
        EMAIL_RECEIVER = 'receiver@gmail.com'
    }

    stages {
        stage('Repository Loading') {
            steps {
                script {
                    executeCommandViaSSH("""
                        cd ${SITE_DIR}
                        git clone git@github.com:itlat-mysql/blog.git .
                    """)
                }
            }
        }

        stage('Copy Credentials') {
            steps {
                script {
                    withCredentials([file(credentialsId: 'SITE_ENV_FILE', variable: 'SITE_ENV_FILE')]) {
                        copyFileViaSSH(SITE_ENV_FILE, "${SITE_DIR}")
                    }
                }
            }
        }

        stage('Start Project') {
            steps {
                script {
                    executeCommandViaSSH("""
                        cd ${SITE_DIR}
                        docker compose up -d --build
                    """)
                }
            }
        }
    }
}
```

# Задание по первоначальному развертыванию сайта (jenkins/init) (2 часть)

```
    post {
        success {
            script {
                emailxtext body: 'Project initialized!', to: "${EMAIL_RECEIVER}", subject: 'Build succeeded: $PROJECT_NAME - #${BUILD_NUMBER}'
            }
        }
        failure {
            script {
                emailxtext body: 'Project initialization failed!', to: "${EMAIL_RECEIVER}", subject: 'Build failed: $PROJECT_NAME - #${BUILD_NUMBER}'
            }
        }
    }
}

def executeCommandViaSSH(commands) {
    withCredentials([
        string(credentialsId: 'SITE_ADDRESS', variable: 'SITE_ADDRESS'),
        string(credentialsId: 'SITE_USER', variable: 'SITE_USER'),
    ]) {
        sshagent(credentials: ['SITE_CREDENTIALS']) {
            sh "ssh " + SITE_USER + "@" + SITE_ADDRESS + " ${commands}"
        }
    }
}

def copyFileViaSSH(srcFilePath, remoteDir) {
    withCredentials([
        string(credentialsId: 'SITE_ADDRESS', variable: 'SITE_ADDRESS'),
        string(credentialsId: 'SITE_USER', variable: 'SITE_USER'),
    ]) {
        sshagent(credentials: ['SITE_CREDENTIALS']) {
            sh "scp '" + srcFilePath + "' " + SITE_USER + "@" + SITE_ADDRESS + " :${remoteDir}"
        }
    }
}
```

## Задание по обновлению сайта

- Мы уже умеет разворачивать сайт, но пока не умеем его обновлять. Однако внутри нашего сайта есть еще один файл, который позволит производить обновление - более того, обновление должно запускаться после того, как в репозитории произошли изменения. Создадим для этого новое задание с типом **Pipeline** и названием **"Site Update Deploy"**.
- В разделе задания **Build Triggers** выберем опцию **Poll SCM**, а в открывшемся поле **Schedule** впишем пять звездочек (**\* \* \* \* \***) - это означает, что Jenkins будет проверять наличие обновлений каждую минуту.
- Находим внутри задания блок **Pipeline** и в списке **Definition** выбираем опцию **Pipeline script from SCM**. Затем вписываем в поля открывшегося окна те значения, которые мы вписывали в задание для развертывания приложения. Однако в поле **Script Path** напишем путь к уже другому файлу - **jenkins/update**.
- После того, как мы сохраним задание, оно будет запускать при каждом обновлении ветки main нашего репозитория. Таким образом, мы реализуем простейшую логику **Continuous Deployment**.



# Задание по обновлению сайта (пример)

## Pipeline

### Definition

Pipeline script from SCM

### SCM ?

Git

### Repositories ?

#### Repository URL ?

git@github.com:itlat-mysql/blog.git

#### Credentials ?

GITHUB\_CREDENTIALS

+ Add

Advanced

Add Repository

### Branches to build ?

#### Branch Specifier (blank for 'any') ?

\*/main

Add Branch

### Repository browser ?

(Auto)

### Additional Behaviours

Add

### Script Path ?

jenkins/update

# Задание по обновлению сайта (jenkins/update) (1 часть)

```
pipeline {
  agent any

  environment {
    SITE_DIR = '/home/user/site'
    BACKUP_DIR = '/home/user/backup'
    DOCKER_COMPOSE_FILE = '/home/user/site/docker-compose.yml'
    EMAIL_RECEIVER = 'receiver@gmail.com'
  }

  stages {
    stage('Backup') {
      steps {
        script {
          executeCommandViaSSH("""
            rm -rf ${BACKUP_DIR}
            mkdir ${BACKUP_DIR}
            cp -r ${SITE_DIR}/.git ${BACKUP_DIR}/.git
            cd ${SITE_DIR}
            set +x
            DB_USER=`cat .env | grep -E '^DB_USER(.*)' | cut -f 2- -d '='`
            DB_NAME=`cat .env | grep -E '^DB_NAME(.*)' | cut -f 2- -d '='`
            DB_PASSWORD=`cat .env | grep -E '^DB_PASSWORD(.*)' | cut -f 2- -d '='`
            docker compose -f ${DOCKER_COMPOSE_FILE} exec -T db mysqldump \${DB_NAME} -u\${DB_USER} -p\${DB_PASSWORD} > \${BACKUP_DIR}/db.sql
            set -x
          """)
        }
      }
    }

    stage('Stop Old Project') {
      steps {
        script {
          executeCommandViaSSH("""
            cd ${SITE_DIR}
            docker compose down
            docker image prune --force
          """)
        }
      }
    }

    stage('Update Project') {
      steps {
        script {
          executeCommandViaSSH("""
            cd ${SITE_DIR}
            git pull
            docker compose up -d --build
          """)
        }
      }
    }
  }
}
```

# Задание по обновлению сайта (jenkins/update) (2 часть)

```
post {
    success {
        script {
            emailxext body: 'Build succeeded!', to: "${EMAIL_RECEIVER}", subject: 'Build succeeded: $PROJECT_NAME - #${BUILD_NUMBER}'
        }
    }
    failure {
        script {
            emailxext body: 'Build failed!', to: "${EMAIL_RECEIVER}", subject: 'Build failed: $PROJECT_NAME - #${BUILD_NUMBER}'

            executeCommandViaSSH("""
                cd ${BACKUP_DIR}
                VALID_COMMIT_CODE=`git rev-parse HEAD`
                cd ${SITE_DIR}
                docker compose down
                docker image prune --force
                git reset --hard \${VALID_COMMIT_CODE}
                docker compose up -d --build
            """)

            executeCommandViaSSH("""
                cd ${SITE_DIR}
                set +x
                DB_USER=`cat .env | grep -E '^DB_USER(.*)' | cut -f 2- -d '='`
                DB_NAME=`cat .env | grep -E '^DB_NAME(.*)' | cut -f 2- -d '='`
                DB_PASSWORD=`cat .env | grep -E '^DB_PASSWORD(.*)' | cut -f 2- -d '='`
                set -x
                cd ${BACKUP_DIR}
                docker cp ./db.sql `docker-compose -f ${DOCKER_COMPOSE_FILE} ps -q db`:db.sql
                set +x
                docker compose -f ${DOCKER_COMPOSE_FILE} exec -T db mysql -u\${DB_USER} -p\${DB_PASSWORD} \${DB_NAME} < db.sql
                set -x
                docker compose -f ${DOCKER_COMPOSE_FILE} exec -T db rm db.sql
            """)
        }
    }
}

def executeCommandViaSSH(commands) {
    withCredentials([
        string(credentialsId: 'SITE_ADDRESS', variable: 'SITE_ADDRESS'),
        string(credentialsId: 'SITE_USER', variable: 'SITE_USER'),
    ]) {
        sshagent(credentials: ['SITE_CREDENTIALS']) {
            sh "ssh " + SITE_USER + "@ " + SITE_ADDRESS + " ${commands}"
        }
    }
}
```