

CI/CD Process - Utilizing Jenkins, Docker, Jest and React.Js

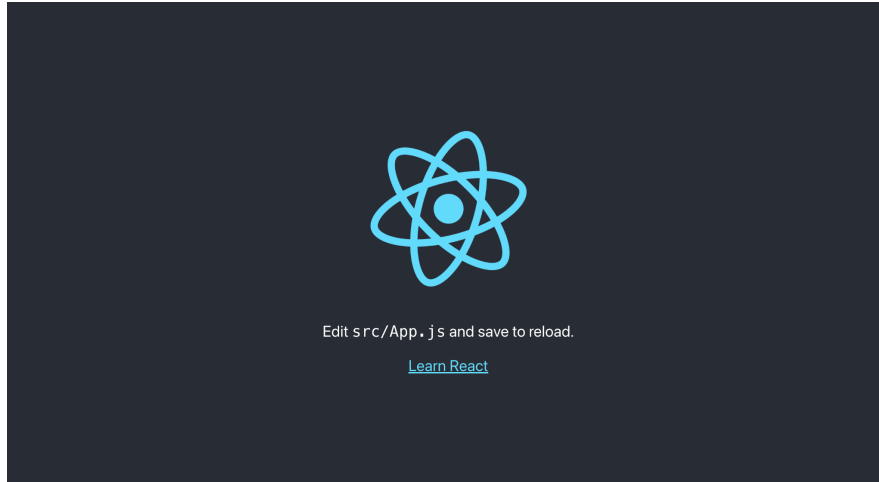
Required downloads and setup:

- Docker (<https://www.docker.com/products/docker-desktop/>). **Create an account as well.**
- Node.js (<https://nodejs.org/en/download>)
- Visual Studio Code (Or an editor you are comfortable with using.)
- Git Account (Github, Bitbucket, Gitlab, etc)

Let's create our Simple React Project

Open up your terminal and choose a directory you want your React Project to live. Once in that directory run the following commands to create your react application:

- `npx create-react-app simple-app`
- `cd simple-app`
- `npm start`



It should start the application on **http://localhost:3000** and provides a building block for creating a React application.

Initial Jenkinsfile

Create a file called '**Jenkinsfile**' within your React app directory. The Jenkinsfile will contain the instructions that will be used to automate the CI/CD process. A tool required to run our React application is **nodejs**. This will make npm accessible throughout the pipeline process. Each instruction is defined and split into stages in the Jenkinsfile. First, the pipeline will run **npm install** to download the needed dependencies.

It will then run **npm test** to make sure the application is running properly.

```
pipeline {
  agent any

  tools {nodejs "node"}

  stages {
    stage('Install dependencies') {
      steps {
        sh 'npm install'
      }
    }

    stage('Test') {
```

```
steps {  
  sh 'npm test'  
}  
}  
}  
}
```

This is the basic building blocks of the Jenkinsfile, more will be added later on.

Add Project to Github

Create a repository in Github and place the React code into the repository.

Documentation on how to do it [here](#). If you never used Github, you may need to [setup SSH](#) on your computer to push your code to Github.

You now have the Jenkinsfile and repository ready, let's move on to Docker.

Creating a Custom Jenkins Image with a Dockerfile

Jenkins is an automation server. It's most often associated with building source code and deploying the results. For many, Jenkins is synonymous with continuous integration and continuous delivery (CI/CD).

Let's set up the **Dockerfile** and put it in the parent folder outside the React folder. Here is how the directory structure should look:

./parent-directory

./simple-app (React app)

Jenkinsfile

Dockerfile (Custom Jenkins Dockerfile)

A **Dockerfile** is a set of instructions used for creating a Docker Image. A Docker Image is a template with instructions used to create a Docker Container. We are trying to create a Jenkins container so let's set up the instructions.

The first instruction is **FROM**. The **FROM** instruction specifies the parent image in which we are building. In this case, we are using **FROM** to extend from jenkins. We will be utilizing an image already created for Jenkins that gets constant updates. The last part “:lts” specifies that we want the latest version of Jenkins.

The second line gives the container **root** access.

The third line updates the system to provide the most up-to-date packages and versions for the operating system.

The fourth and last line, will install docker within the container using a curl command.

Docker can now be utilized within the container. It will be used to build our React application and push it to your own Docker registry.

```
FROM jenkins/jenkins:lts
USER root
RUN apt-get update
RUN curl -sSL https://get.docker.com/ | sh
```

Building the Docker Image

Open Docker and make sure that Docker is running. You can tell it is running by either having the Graphical Interface open or you will see a whale icon on the top right (Mac users).

With the Dockerfile, a Docker image can be created using the set of instructions within the file.

The **build** command creates a docker image based on the Dockerfile.

The **-t flag** tags the image to the name specified after the flag. The usual naming convention is to use **<Docker username> / <image name>**. The dot at the end specifies the directory where the Dockerfile is located for the build.

```
docker build -t <Docker username>/<image name> .  
// Example  
docker build -t giovannig/jenkins_uf .
```

After building the image successfully, the image can be used to run a Docker container. The **run** command uses the specified docker image as a template to run a container. Use the following command to start up the container (**Fill in the brackets**):

```
docker run -d  
-p 8080:8080  
-v /var/run/docker.sock:/var/run/docker.sock  
-v jenkins_home:/var/jenkins_home  
--name <container name>  
<Docker username>/<image name>  
  
// Example  
docker run -d  
-p 8080:8080  
-v /var/run/docker.sock:/var/run/docker.sock  
-v jenkins_home:/var/jenkins_home  
--name jenkins_container  
giovannig/jenkins_uf
```

The **-d** flag means detached, which will run the containers in the background. We will use **-p** to expose the Jenkins port which in this case will be on port 8080. The **-v** allows us to mount a volume. Volumes are used to make sure data is not deleted when the container stops.

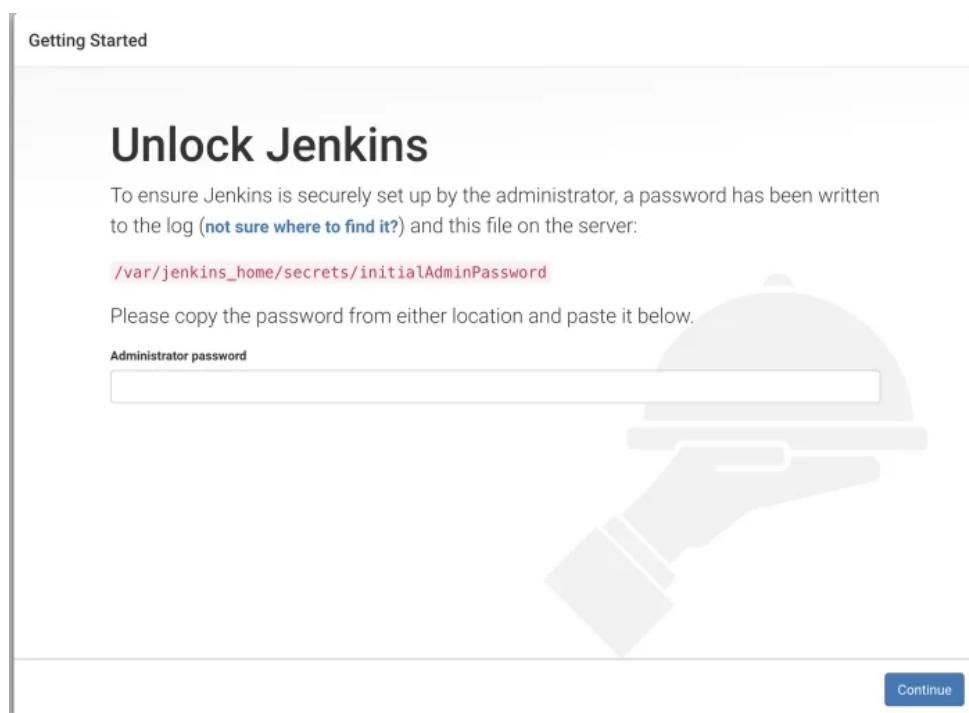
The `/var/run/docker.sock:/var/run/docker.sock` will host the docker daemon to the container so we can invoke docker api/client from our container. We will need this when we need to push the React application to Docker.

The `jenkins_home:/var/jenkins_home` directory maps the `/var/jenkins_home` directory inside the container to the Docker volume named `jenkins_home`. This allows this Docker container's Docker daemon to mount data from Jenkins. So even when you exit the container and reenter the container, the data is still available.

The **–name** flag will give the container a name. If not specified, Docker will give your container a random name, usually funny but not informative.

At the end, the image name being used is specified to create the Docker container. Use the same name of the image you built, which used the format of `<Docker username>/<image name>`

Go to <http://localhost:8080>:



The initial page informs you to find the initial password in a log file, but Jenkins prints the initial password using standard output too. So, it can be retrieved from the Docker logs.

Go back to your terminal and use the following command to get the password:

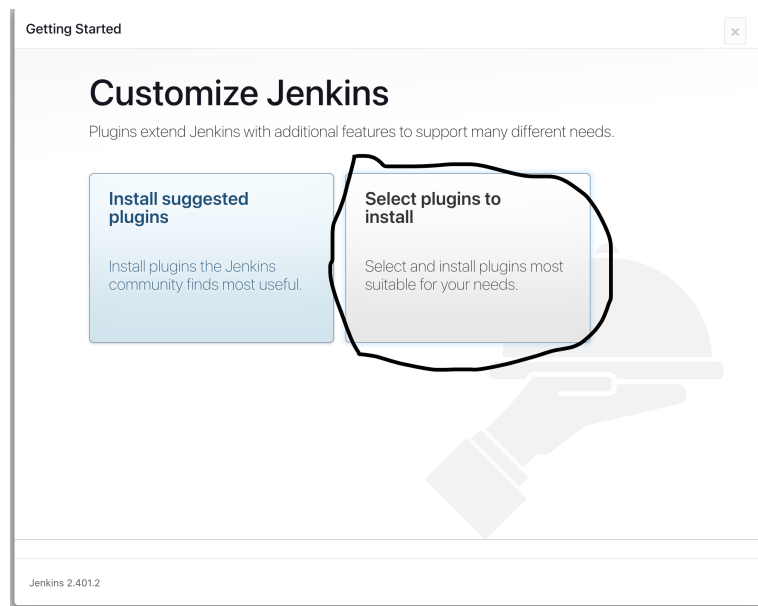
```
docker exec <container name> cat /var/jenkins_home/secrets/initialAdminPassword  
  
// or
```

```
docker logs <container name> // It should be in the logs
```

You should receive a **hash** output. Enter the password and click Continue.

Customize Jenkins

Select Plugins to Install



We will download the plugins we need to run our pipeline properly. The most important plugin is **NodeJS** since we will need it to run our React application.

Getting Started

Organization and Administration

Build Features

Build Tools

Build Analysis and Reporting

Pipelines and Continuous Delivery

Source Code Management

Distributed Builds

User Management and Security

Notifications and Publishing

Languages

All | None | Suggested

Selected (23/52)

Note that the full list of plugins is not shown here. Additional plugins can be installed in the **Plugin Manager** once the initial setup is complete. [See the documentation for more information.](#)

Organization and Administration (4/4)

☒ Dashboard View

Customizable dashboard that can present various views of job information.

25

☒ Folders

This plugin allows users to create "folders" to organize jobs. Users can define custom taxonomies (like by project type, organization type etc). Folders are nestable and you can define views within folders. Maintained by CloudBees, Inc.

1

☒ Configuration as Code

This plugin allows configuration of Jenkins based on human-readable declarative configuration files.

8

☒ OWASP Markup Formatter

Uses the [OWASP Java HTML Sanitizer](#) to allow safe-seeming HTML markup to be entered in project descriptions and the like.

7

Organization and Administration

Build Features

Build Tools

Build Analysis and Reporting

Pipelines and Continuous Delivery

Source Code Management

Distributed Builds

User Management and Security

Notifications and Publishing

Languages

All | None | Suggested

Selected (23/52)

Build Features (6/10)

☒ Build Name and Description Setter

This plug-in sets the display name and description of a build to something other than #1, #2, #3, ...

30

☒ Build Timeout

This plugin allows you to automatically terminate a build if it's taking too long.

6

☒ Config File Provider

Ability to provide configuration files (e.g. settings.xml for maven, XML, groovy, custom files,...) loaded through the UI which will be copied to the job workspace.

12

☒ Credentials Binding

Allows credentials to be bound to environment variables for use from miscellaneous build steps.

10

☐ Embeddable Build Status

This plugin adds the embeddable build status badge to Jenkins so that you can easily hyperlink/show your build status from elsewhere.

9

☐ Rebuilder

This plugin is for rebuilding a job using the same parameters.

26

☐ SSH Agent

This plugin allows you to provide SSH credentials to builds via a ssh-agent in Jenkins.

9

☐ Throttle Concurrent Builds









This plugin allows for throttling the number of concurrent builds of a project running per node or globally.

28















☒ Timestamper

13

Build Tools (3/4)

- ☒ **Ant**  5 
Adds Apache Ant support to Jenkins
- ☒ **Gradle**  23 
This plugin allows Jenkins to invoke [Gradle](#) build scripts directly.
- ☐ **MSBuild**  8 
This plugin makes it possible to build a Visual Studio project (.proj) and solution files (.sln).
- ☒ **NodeJS**  13 
NodeJS Plugin executes [NodeJS](#) script as a build step.

Pipelines and Continuous Delivery (4/7)

- ☒ **Pipeline**  47 
A suite of plugins that lets you orchestrate automation, simple or complex. See [Pipeline as Code with Jenkins](#) for more details.
- ☒ **GitHub Branch Source**  35 
Multibranch projects and organization folders from GitHub. Maintained by CloudBees, Inc.
- ☒ **Pipeline: GitHub Groovy Libraries**  31 
Allows Pipeline Groovy libraries to be loaded on the fly from GitHub.
- ☒ **Pipeline: Stage View**  23 
Pipeline Stage View Plugin.
- ☐ **Conditional BuildStep**  7 
A buildstep wrapping any number of other buildsteps, controlling their execution based on a defined condition (e.g. BuildParameter).
- ☐ **Parameterized Trigger**  26 
This plugin lets you trigger new builds when your build has completed, with various ways of specifying parameters for the new build.
- ☐ **Copy Artifact**  27 
Adds a build step to copy artifacts from another project.

Source Code Management (3/10)

☒ Bitbucket
 Integrates with BitBucket

☐ ClearCase
 This plugin makes it possible to retrieve files from a ClearCase SCM using a configspeg.

☐ CVS
 Integrates Jenkins with CVS version control system using a modified version of the Netbeans cvsclient.

☒ Git
 This plugin integrates Git with Jenkins.

☐ Git Parameter
 Adds ability to choose branches, tags or revisions from git repositories configured in project.

☒ GitHub
 This plugin integrates GitHub to Jenkins.

☐ GitLab
 This plugin allows GitLab to trigger Jenkins builds and display their results in the GitLab UI.

☐ P4
 Perforce Client plugin for the Jenkins SCM provider. The plugin includes extension points for: Perforce Password and Ticket Credentials storeWorkspace management for static manual template and streamAction point for

Press install and give it some time to download all of the plugins.

Getting Started

Getting Started

✓ Folders	✓ OWASP Markup Formatter	✓ Build Timeout	✓ Credentials Binding	** Plain Credentials
✓ Timestamper	✓ Workspace Cleanup	✓ Ant	⚙ Gradle	** Trilead API
⚙ Pipeline	⚙ GitHub Branch Source	⚙ Pipeline: GitHub Groovy Libraries	⚙ Pipeline: Stage View	** SSH Credentials
⚙ Git	⚙ SSH Build Agents	⚙ Matrix Authorization Strategy	⚙ PAM Authentication	Credentials Binding
⚙ LDAP	⚙ Email Extension	⚙ Mailer	⚙ Dashboard View	** SCM API
⚙ Configuration as Code	⚙ Config File Provider	⚙ Build Name and Description Setter	⚙ NodeJS	** Pipeline: API
⚙ GitHub	⚙ Bitbucket			** commons-lang3 v3.x Jenkins API
				Timestamper
				** Caffeine API
				** Script Security
				** JAXB
				** SnakeYAML API
				** Jackson 2 API
				** commons-text API
				** Pipeline: Supporting APIs
				** Plugin Utilities API
				** Font Awesome API
				** Bootstrap 5 API
				** JQuery3 API
				** ECharts API
				** Display URL API
				** Checks API
				** JUnit
				** Matrix Project
				** Resource Disposer
				Workspace Cleanup
				Ant
				** Durable Task
				** - required dependency

Jenkins 2.401.2

Now create an account and continue:

Create First Admin User

Username

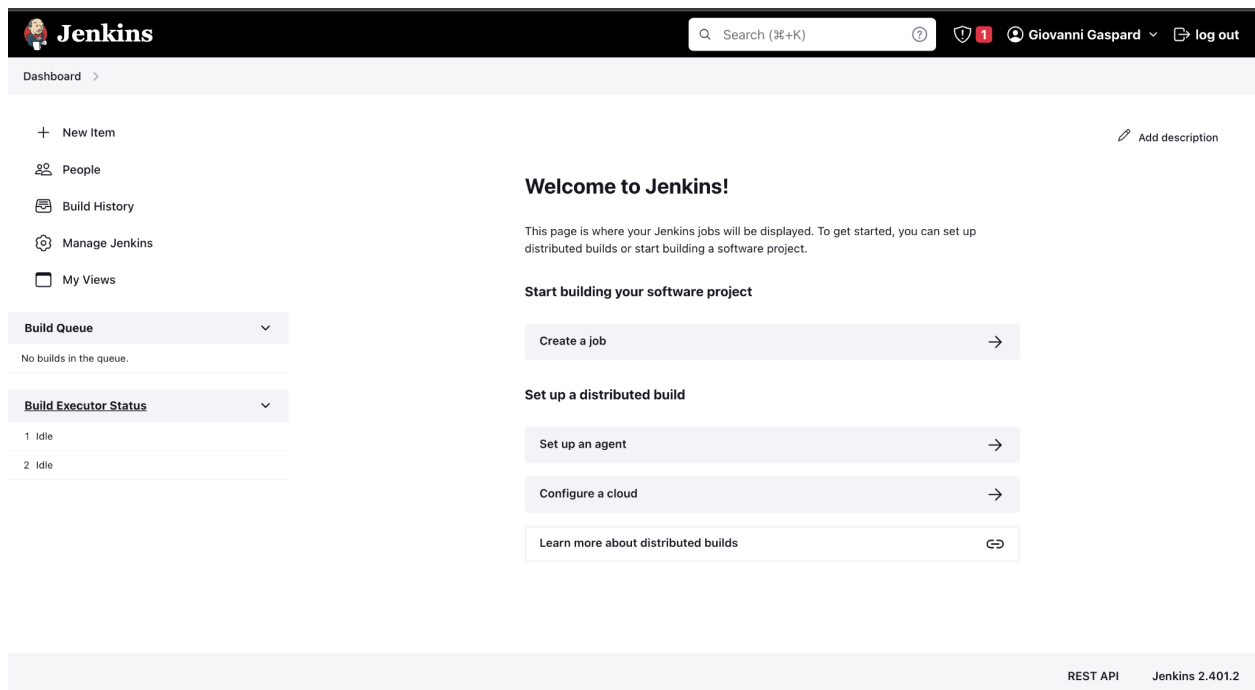
Password

Confirm password

Full name

E-mail address

Once you fill out the form we should be good to go and you should see the following:



The screenshot shows the Jenkins dashboard interface. At the top is a black header with the Jenkins logo, a search bar, and user information (Giovanni Gaspard). Below the header is a light gray sidebar with navigation links: New Item, People, Build History, Manage Jenkins, and My Views. The main content area has a 'Welcome to Jenkins!' message and a 'Start building your software project' section with buttons for 'Create a job', 'Set up a distributed build', 'Set up an agent', 'Configure a cloud', and 'Learn more about distributed builds'. On the left, there are two expandable sections: 'Build Queue' (showing 'No builds in the queue.') and 'Build Executor Status' (showing two idle executors). The footer contains links for 'REST API' and 'Jenkins 2.401.2'.

Jenkins Search (⌕+K) 1 Giovanni Gaspard log out

Dashboard >

+ New Item Add description

People

Build History

Manage Jenkins

My Views

Build Queue ▾

No builds in the queue.

Build Executor Status ▾

1 Idle

2 Idle

Welcome to Jenkins!

This page is where your Jenkins jobs will be displayed. To get started, you can set up distributed builds or start building a software project.

Start building your software project

Create a job →

Set up a distributed build

Set up an agent →

Configure a cloud →

Learn more about distributed builds ↗


REST API Jenkins 2.401.2


Click the + New Item button on the top left. Create a name for the pipeline and select Multibranch Pipeline. Press Ok.


Enter an item name


demo-pipeline


» 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.

 **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.

OK

Click the Branch Source. Select Git and provide the url of the git repo for the branch with the React project and Jenkinsfile.

Dashboard > demo-pipeline > Configuration

Configuration

General Branch Sources Build Configuration Scan Multibranch Pipeline Triggers Orphaned Item Strategy Appearance Health metrics Properties

General Enabled ☒

Display Name ?

Description

[Plain text] [Preview](#)

Branch Sources

Add source

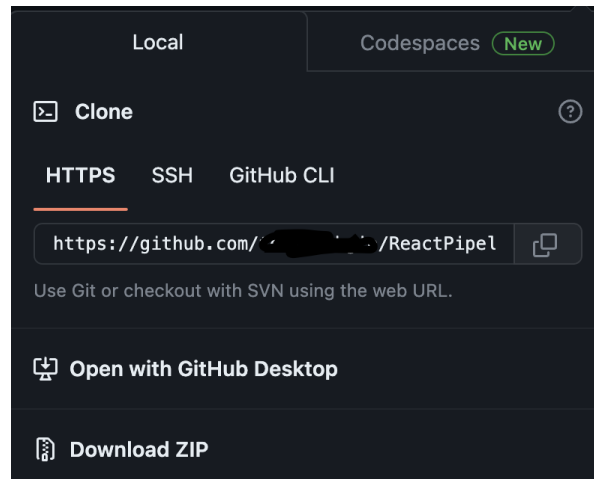
Git
GitHub
Mercurial
Single repository & branch

Mode

by Jenkinsfile

Save Apply

Get the Git Repository Url from the repository.



Place the Repository HTTPS Url from your Git Project Repository in the Project Repository input.

Branch Sources

Git

Project Repository ?

Credentials ?

- none -

Add

Behaviors

Discover branches ?

Add

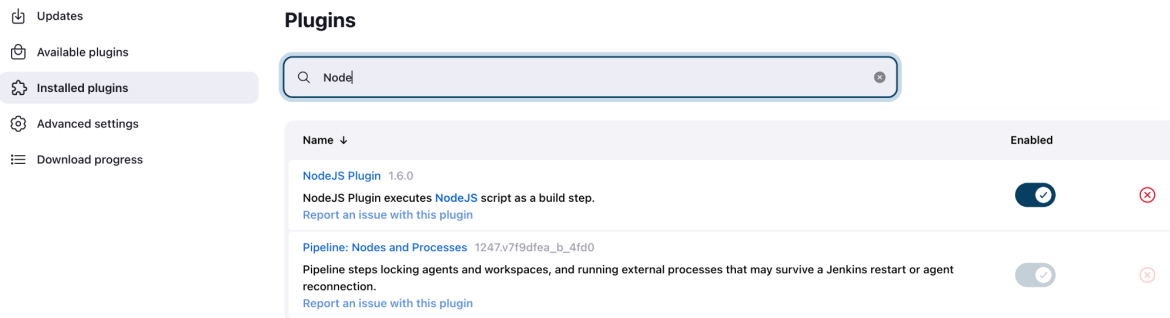
Property strategy

All branches get the same properties

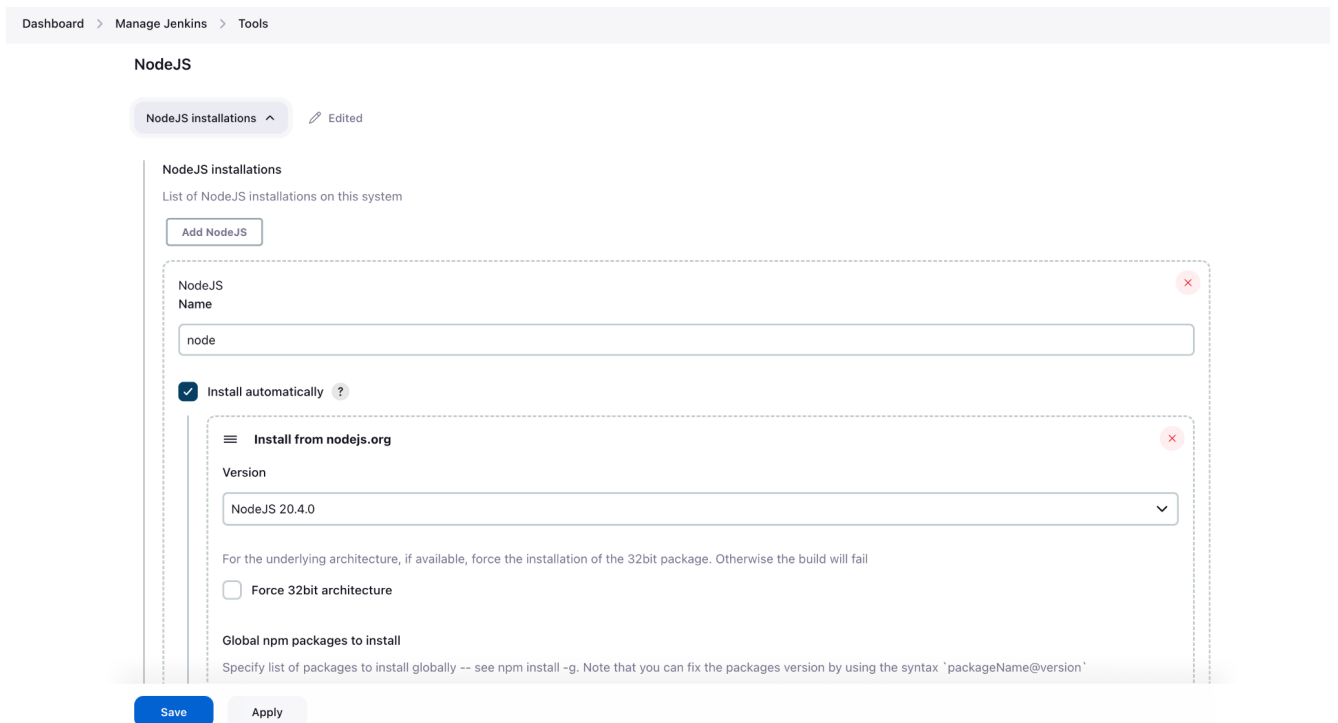
Add property

Save Apply

Go to the following url (**/manage/pluginManager/installed**) to check that Node was an installed plugin. Make sure Node is installed and enabled.



Go to the following url(**/manage/configureTools/**) and scroll down to add **Nodejs**. Make sure the latest version of **Nodejs** is available. Add the name “**node**” in the name input and save it. Now we will have **NodeJs** globally within our pipelines as long as we specify the name “**node**”.



Downloading the Necessary Jenkins Plugins

Go to Dashboard > Manage Jenkins > Plugins

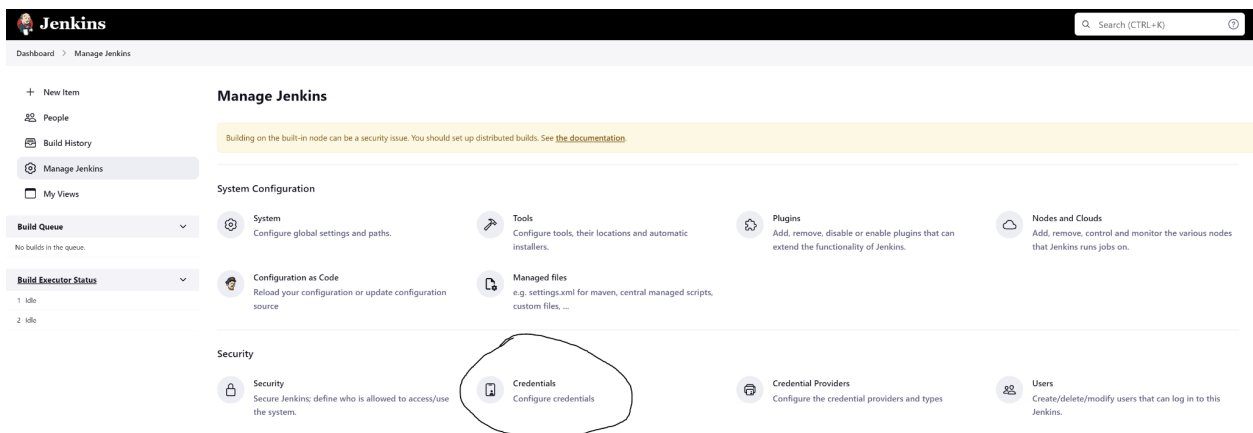


In the Search box, search for the following plugins and install them **without restart**.

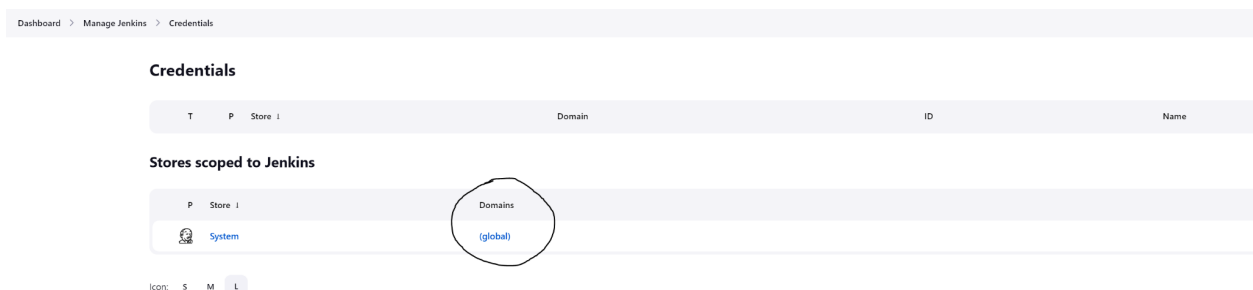
- Docker Plugin
- Docker Pipeline

Add Docker Credentials

Go to Dashboard > Manage Jenkins > Credentials



We want to add the credentials for global usage. Click **global** and add credentials.



Add your Docker Credentials so we can inject them into the pipeline. We will need the credentials to push to your Docker hub. Make sure the id is “**dockerhub-creds**”. It can be any name but it is the name used to retrieve the credentials in the Jenkinsfile.

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

New credentials

Kind: Username with password

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

Username: giovanni

☒ Treat username as secret

Password:

ID: dockerhub-creds

Description: Credentials for Docker

Create

Editing the Jenkinsfile

We can now add new stages to handle building the image and deploying it to the Docker registry. Update the **imageName** as well as the **dockerCredentialsName** to match your account information.

The imageName format is **<Docker username>/<application name>**. The **Docker username** is the name you chose when you created your Docker account. The **application name** is the name that you want your application to have in the Docker registry.

The **dockerCredentialsName** is the name that we gave our credentials when we created it in the previous step.

More details regarding the pipeline below:

```
pipeline {
  agent any

  tools {nodejs "node"}
  environment {
```



```

    imageName = <Docker username>/<application name>.
    dockerCredentialsName='dockerhub-creds'
    dockerImage = ''
}

stages {

    stage('Environment') {
        steps {
            echo "Branch: ${env.BRANCH_NAME}"
            sh 'docker -v'
        }
    }
    stage('Install dependencies') {
        steps {
            sh 'npm install'
        }
    }
    stage('Test') {
        steps {
            sh 'npm test'
        }
    }
    stage('Building image') {
        steps{
            script {
                dockerImage = docker.build imageName
            }
        }
    }
    stage('Deploy Image') {
        steps{
            script {
                docker.withRegistry('https://registry.hub.docker.com',
dockerCredentialsName
) {
                    dockerImage.push("${env.BUILD_NUMBER}")
                    dockerImage.push("latest")
                }
            }
        }
    }
}
}

```

The **environment** section is used to define variables that will be used within the stages.

The **imageName** variable is **<Docker username>/<application name>**. The first part is your Docker username and after the forward slash will be the image name.

The **dockerImage** variable will hold an empty initial value but it will be used to hold the built image later in the file.

Review of Each Jenkins Stage

The **Environment** stage is just used to confirm and print out the available tools versions. Since we are using a multibranch pipeline, we will print out the branch name that is running the Jenkins pipeline. We will also print out the docker version as confirmation that Docker is available within the pipeline.

The next two Stages '**Install dependencies**' and '**Test**' will run **npm install** and **npm test** to make sure our React application is properly constructed to continue. If any of the tests fails, it will skip the rest of the stages.

The '**Building Docker Image**' stage will build a docker image using the variable defined in the environment section. It will assign the instance of the image to the **dockerImage** variable.

The '**Deploy Docker Image**' is the final stage. In this stage, we specify the url of the registry and utilize the "**dockerhub-creds**" credentials we made earlier to push our image to our Docker registry. The **dockerImage instance** uses the build number to make sure each new push to the registry will be considered unique each time. Each build will increment the number and will be considered as the latest.

Push the new changes to Github!

Creating the React App Image

Time to create another custom **Dockerfile** to create an image for our React application. Make sure to place this Dockerfile **within** the React app directory. This is what our

pipeline will build and push to the Docker Registry. It will provide instructions on how to run and expose our React application. This is a simple development version of React, a production version will require a bit more steps.

The instruction for the Dockerfile is below and if you want to understand what each line does, a description is below as well.

```
# Extending image
FROM node:latest

# Create app directory
WORKDIR /app

# Copy Package to Working Directory
COPY package.json /app

# Install Dependencies in Package.json
RUN npm install

# Copy the rest of the files to the Working Directory
COPY . /app

# Port to listener
EXPOSE 3000

# Main command that will run when container starts
CMD [ "npm", "run", "start" ]
```

How the structure should look like:

./parent-directory

 ./simple-app (React app)

 Dockerfile (Custom React Dockerfile)

 Jenkinsfile

 Dockerfile (Custom Jenkins Dockerfile)

Let's set up the Dockerfile. Just like with the Jenkins Dockerfile, a Dockerfile is a set of instructions used for creating a Docker Image. A Docker Image is a template with instructions used to create a Docker Container.

Each instruction in a Dockerfile creates a layer in the image. When you change the

Dockerfile and rebuild the image, only those layers that have changed are rebuilt. It makes it faster to rebuild for minor changes due to not having to retrieve it unnecessarily. It is cached and reused which increases speed.

Do not get overwhelmed by the Dockerfile file. I will explain it so try to understand as much as you can, this will make more sense with time. Let's start!

The **FROM** instruction specifies the parent Image in which we are building. In this case, it will also utilize the node image which has all the tools and packages needed to run a React application.

The **RUN** command will execute commands.

The first RUN command will set the working directory for the container. It sets the **/app** directory as a working directory within the container using the **WORKDIR** command. By setting the **/app** directory as the working directory, all the commands will be directed to that directory. If the directory does not exist, the command will create it and assign it as the working directory.

The **COPY** command as the name implies will copy files to a new directory. In our case, it will copy the **package.json** to our working directory. This will be used to install the needed packages for the React application.

It is done by running **npm install** which will see the package.json that was copied over to the working directory and install all the dependencies specified within the file.

The rest of the React application is copied over from the parent directory to the working directory.

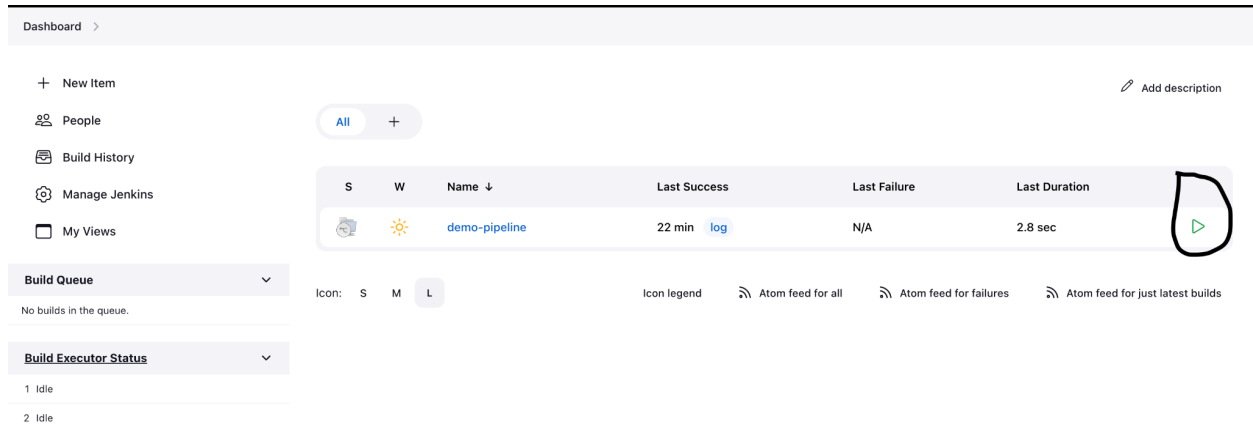
The **EXPOSE** keyword lets Docker know that the container will be listening for traffic on a specific port. For our container, it will expose port 3000.

Now that the setup is complete, it is instructed to run the **npm run start** command to start up the React application when the container is created. **CMD** is used to specify a command to run on the creation of the container. The format requires each item to be comma-separated.

So as you can see an image is just a set of instructions to create a container.

Running the Jenkins Pipeline

Now go back to the Dashboard and press the play button to get the pipeline running:



The screenshot shows the Jenkins Dashboard with a sidebar on the left containing links: + New Item, People, Build History, Manage Jenkins, and My Views. The main area displays a table of builds. The first build, 'demo-pipeline', is in the 'Build Queue' and has a green play button icon circled in the 'Last Duration' column. Below the table, there are sections for 'Build Queue' (No builds in the queue) and 'Build Executor Status' (2 Idle).

S	W	Name ↓	Last Success	Last Failure	Last Duration
		demo-pipeline	22 min log	N/A	2.8 sec

Running the React Image

Well if the pipeline ran successfully, you should now see the image in your Docker Repository. Go to <https://hub.docker.com> to confirm. To create a container using the image you just pushed, you can use the following command (**Replace in the brackets**):

```
docker run -d -p 3000:3000 --rm --name react-app <Docker username>/<application name>
```

You are running the application detached (**-d**) and utilizing the exposed port (**-p**) 3000 that was specified in the Dockerfile to run on the host port of 3000. The **--rm** flag will remove the container once the container is stopped. The command creates a custom container name using **--name** as react-app. Provide the image name at the end to create the container. Now if we go to **localhost:3000**, we should see our react application running!

What Now?

Well we have a pipeline that is automated to run builds. What now? Well if you are curious, you can make changes and test whether those changes are being processed.

You can even add other stages to do other checks such as code coverage. Code coverage is used to check if all areas of your code are tested. So if a file is only tested half way it may show the file to only have fifty percent code coverage. A popular software to display the coverage is **Sonarqube**. **Sonarqube** also tests for code quality and code smells.

You will learn in the future about the Cloud. With the React application now being an image in the registry, you can use that image to run your application in the cloud. A popular cloud provider is **Amazon Web Services (AWS)**. We can utilize **AWS** to host our Docker Image and create a domain (using AWS Route 53) to be a functional application that people can utilize. AWS has services like **AWS EC2** and **AWS ECS** which can be used to host your application using your Docker Image.

Hope you found this helpful. If you have any questions, do not be afraid to ask!