

InzynieriaOprogramowaniaAGH /  
MDO2023\_INO

&lt;&gt; Code

Issues

Pull requests

Actions

Projects

Security



This repository has been archived by the owner on Sep 15, 2023. It is now read-only.

MDO2023\_INO / INO / GCL2 / KP406287 / lab03 / Sprawozdanie.md



qamilciaq1 Update Sprawozdanie.md

last year



139 lines (78 loc) · 14.3 KB

Preview

Code

Blame

Raw



## PRZYGOTOWANIE

W poprzednim laboratorium przeprowadzono instalację i konfigurację Jenkinsa zgodnie z dokumentacją, aby przygotować środowisko do pracy z Jenkins Pipeline. W tym celu zostały wykonane kolejno przedstawione kroki z instrukcji dotyczącej instalacji Jenkinsa w środowisku Docker. Po zakończeniu tych kroków Jenkins był gotowy do użycia w celu stworzenia i uruchomienia Pipeline dla projektu <https://github.com/alt-romes/programmer-calculator>. Celem laboratorium było przetestowanie pełnego cyklu CI/CD, w tym budowania, testowania, wdrażania i publikowania projektu.

Przed rozpoczęciem pracy nad projektem należy spełnić pewne wymagania środowiskowe. Wymagane jest dostęp do internetu, aby pobrać i skonfigurować potrzebne narzędzia, takie jak Docker, który pozwala na pracę z kontenerami i obrazami. Należy również skonfigurować kontener Jenkins i DIND w sposób prawidłowy, aby móc pracować z obiektem projektowym w oparciu o pipeline.

W ramach projektu potrzebne będą również pliki Dockerfile, w tym plik konfiguracyjny dla Jenkinsa, a także plik Dockerfile do procesu budowania i testowania. Wymagane jest także sklonowanie repozytorium oraz posiadanie diagramów aktywności, które będą wykorzystane do porównania i wyciągnięcia wniosków na koniec projektu.

Po spełnieniu wymagań środowiskowych można rozpocząć tworzenie obiektu projektowego poprzez utworzenie w Jenkinsie nowego projektu - pipeline'a.

## Diagram CI/CD

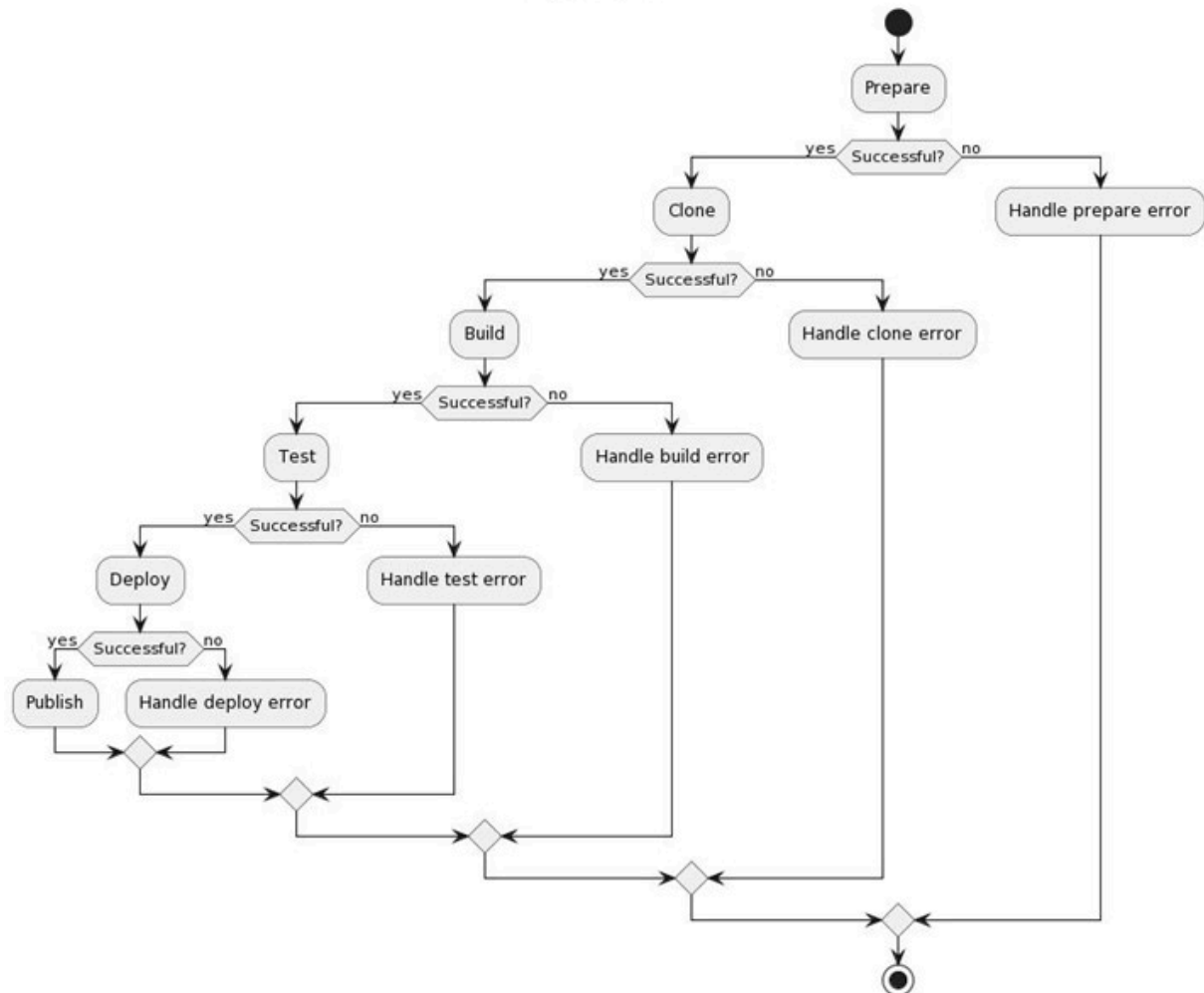
jest narzędziem, które pomaga wizualizować proces wdrażania aplikacji i automatyzacji cyklu życia oprogramowania. Diagram ten pokazuje całą ścieżkę od momentu wprowadzenia zmiany w kodzie źródłowym aż do wdrożenia aplikacji na produkcji.

Diagram CI/CD jest przydatny z kilku powodów. Po pierwsze, umożliwia on zrozumienie i zidentyfikowanie potencjalnych problemów w procesie dostarczania aplikacji. Dzięki temu można wprowadzić zmiany w procesie, aby zminimalizować ryzyko wystąpienia błędów w aplikacji.

Po drugie, diagram CI/CD pozwala na monitorowanie procesu wdrażania aplikacji w czasie rzeczywistym. W ten sposób można na bieżąco śledzić postęp prac i reagować na ewentualne problemy.

Po trzecie, diagram CI/CD pomaga w identyfikacji krytycznych punktów w procesie dostarczania aplikacji oraz w optymalizacji czasu potrzebnego na dostarczenie nowych wersji aplikacji.

## CI/CD Pipeline



## Uruchomienie z prawami administratora kontenerów DIND i Jenkins.

- Aby uruchomić kontener DIND, należy przypisać mu sieć (Jenkins) oraz zmienne środowiskowe przy użyciu flagi env. Taka konfiguracja umożliwi tworzenie kontenera wewnątrz innego kontenera.

```

kamila_partyka@Lenovo:~$ sudo docker run --name jenkins-docker --rm --detach \
> --privileged --network jenkins --network-alias docker \
> --env DOCKER_TLS_CERTDIR=/certs \
> --volume jenkins-docker-certs:/certs/client \
> --volume jenkins-data:/var/jenkins_home \
> --publish 2376:2376 \
> docker:dind --storage-driver overlay2
365d0adebcc8e1d3714fef4f2dda5d0e8c3c77a323bedcfbdef20fb5fe349b36

```

- Uruchomienie kontenera Jenkinsa pozwala na dostęp do interfejsu Jenkinsa poprzez wpisanie adresu localhost:8080 w przeglądarce internetowej. Dzięki temu możemy automatycznie przeprowadzać proces tworzenia oprogramowania w sposób uporządkowany i zautomatyzowany. Jenkins umożliwia wykonywanie zadań takich jak budowanie, testowanie, wdrażanie i dostarczanie oprogramowania w sposób ciągły. Jest to szczególnie przydatne w projektach zespołowych, w których wielu programistów pracuje nad jednym produktem. Jenkins

pozwała na monitorowanie postępów pracy, zarządzanie testami, śledzenie błędów oraz integrację z innymi narzędziami programistycznymi.

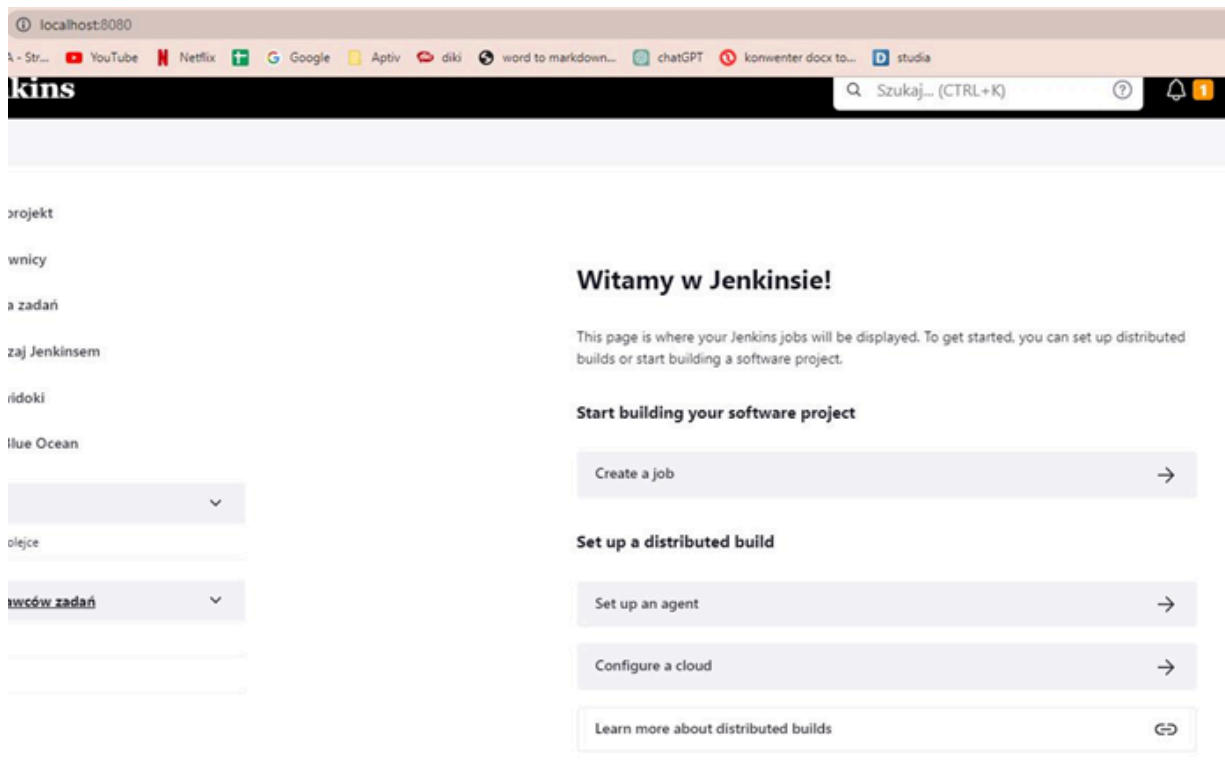
```
kamila_partyka@Lenovo:~$ sudo docker run \
> --name jenkins-blueocean \
> --rm \
> --detach \
> --network jenkins \
> --env DOCKER_HOST=tcp://docker:2376 \
> --env DOCKER_CERT_PATH=/certs/client \
> --env DOCKER_TLS_VERIFY=1 \
> --publish 8080:8080 \
> --publish 50000:50000 \
> --volume jenkins-data:/var/jenkins_home \
> --volume jenkins-docker-certs:/certs/client:ro \
> myjenkins-blueocean:2.387.1-1
```

- Wyświetlenie uruchomionych kontenerów za pomocą polecenia docker ps:

```
kamila_partyka@Lenovo:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND NAMES	CREATED	STATUS	PORTS
365d0adebcc8	docker:dind	"dockerd-entrypoint..."	14 seconds ago	Up 3 seconds	237
5/tcp, 0.0.0.0:2376->2376/tcp		jenkins-docker			
ecc6def21b4b	myjenkins-blueocean:2.387.1-1	"/usr/bin/tini -- /u..."	2 weeks ago	Up 36 minutes	0.0
.0.0:8080->8080/tcp, 0.0.0.0:50000->50000/tcp		jenkins-blueocean			

- Aby móc korzystać z Jenkinsa w trybie interaktywnym, należy przejść do strony o adresie <http://localhost:8080/>. Jeśli już wcześniej utworzyliśmy konto, zostaniemy przekierowani do tablicy nawigacyjnej, która umożliwia nam wybór zadania, które chcemy wykonać. W tablicy tej możemy wykonać różne czynności, takie jak: utworzenie nowego projektu, uruchomienie już istniejącego projektu, dodanie nowego węzła lub agenta, konfigurację ustawień i wiele innych. Jest to bardzo przydatne narzędzie dla programistów i inżynierów, którzy chcą zautomatyzować swoje zadania i procesy, aby oszczędzić czas i zwiększyć wydajność pracy.



## URUCHOMIENIE

- Aby zademonstrować działanie pipeline'a, utworzyłam przykładowy projekt, który pozwala na wyświetlenie aktualnej daty. Aby utworzyć nowy pipeline, wybrałam opcję "Nowy projekt" w menu Jenkinsa, a następnie podałam nazwę dla naszego pipeline'a i wybrałam typ projektu jako "Pipeline". Następnie skonfigurowałam nasz pipeline poprzez zdefiniowanie kroków, które mają zostać wykonane w określonej kolejności. Można to zrobić poprzez edycję pliku Jenkinsfile lub za pomocą interfejsu użytkownika Jenkinsa. W tym przypadku, w naszym pipeline'u, zdefiniowałam krok, który pobiera aktualną datę i wyświetla ją na ekranie. Po zapisaniu zmian i uruchomieniu pipeline'a, Jenkins automatycznie wykonał określone kroki i wyświetlił wynik na ekranie. Jest to przykład prostego pipeline'a, który pozwala na automatyzację procesów i zwiększenie wydajności pracy.

## Definition

Pipeline script

Script ?

```
1 pipeline {  
2   agent any  
3  
4   stages {  
5     stage('Display Date') {  
6       steps {  
7         sh 'date'  
8       }  
9     }  
10  }  
11 }  
12
```

☒ Use Groovy Sandbox ?[Pipeline Syntax](#)

## Pipeline przykład

### Stage View



### Prepare Clone i Build

- Stworzyłam plik Dockerfile, który zawiera skrypty potrzebne do skonfigurowania obrazu Dockerowego. W pliku tym zdefiniowałam listę zależności, które muszą zostać zainstalowane, aby móc uruchomić



projekt. Następnie skopiowałam repozytorium projektu do kontenera Dockera, aby można było wykonać na nim operacje budowania. Obraz Dockerowy, który stworzyłam, jest oparty na systemie operacyjnym Ubuntu. Dzięki temu, użytkownicy mogą łatwo uruchomić ten obraz i wdrożyć projekt na serwerze lub w chmurze. Dockerfile jest często używany przez programistów i inżynierów do tworzenia izolowanych środowisk, w których mogą testować i wdrażać swoje aplikacje bez wpływu na pozostałe części systemu.

## Jenkinsfile:

Script ?

```
1 pipeline {
2   agent any
3
4   stages {
5     stage('Prepare') {
6       steps {
7         sh "rm -rf MDO2023_INO"
8         sh "docker system prune --all --force"
9       }
10    }
11    stage('Clone') {
12      steps {
13        echo ' '
14        sh "git clone https://github.com/InzynieriaOprogramowaniaAGH/MDO2023_INO"
15        dir('MDO2023_INO'){
16          sh "git checkout KP406287"
17        }
18      }
19    }
20    stage('Build') {
21      steps {
22        echo ' '
23        dir('MDO2023_INO/INO/GCL2/KP406287/lab03'){
24          sh "docker build -t bldr ."
25        }
26      }
27    }
28  }
29 }
30
31
32
33
```

### 1. Etap "Prepare":

- W tym etapie wykonują się dwa polecenia shell: "rm -rf MDO2023\_INO" usuwające folder "MDO2023\_INO" oraz "docker system prune --all --force", które usuwają wszystkie niepotrzebne elementy systemu Docker.

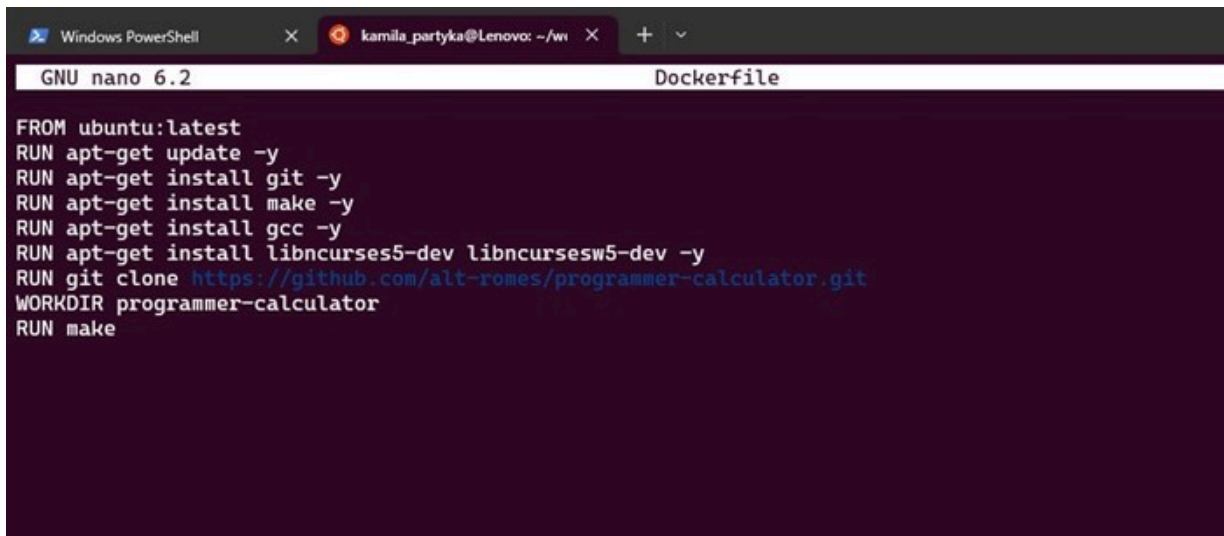
### 2. Etap "Clone":

- W tym etapie pobierany jest kod z repozytorium git pod adresem [https://github.com/InzynieriaOprogramowaniaAGH/MDO2023\\_INO](https://github.com/InzynieriaOprogramowaniaAGH/MDO2023_INO) i przechowywany w folderze "MDO2023\_INO".

- Następnie wchodzimy do folderu "MDO2023\_INO" i wykonujemy polecenie "git checkout KP406287", które przywraca stan kodu z gałęzi o nazwie "KP406287".

### 3. Etap "Build":

- W tym etapie wchodzimy do folderu "MDO2023\_INO/INO/GCL2/KP406287/lab03".
- Wykonujemy polecenie "docker build -t bldr ." tworzące obraz Dockera o nazwie "bldr" na podstawie pliku Dockerfile znajdującego się w bieżącym folderze.



```
GNU nano 6.2 Dockerfile
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install git -y
RUN apt-get install make -y
RUN apt-get install gcc -y
RUN apt-get install libncurses5-dev libncursesw5-dev -y
RUN git clone https://github.com/alt-romes/programmer-calculator.git
WORKDIR programmer-calculator
RUN make
```

Wydruki z konsoli:





## Logi konsoli

```
Started by user admin
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Prepare)
[Pipeline] sh
+ rm -rf MDO2023_INO
[Pipeline] sh
+ docker system prune --all --force
Deleted build cache objects:
cpfobtn1i67anb4xyf8440dok
wfbuumtj1su8onsj5xe7qmi94
ns8r3fh08ztkhoko1dm4c9dqt
pv55c0iqc24rzg69ajbtc53az
mz2e0nlyep6ra0eot263vy9np
7dxhfqkawuhakfvgq9oxi6d4q
1k2ot3v0xdgtth545m04vxqu2
nvwvfw8draageb1j00h1m7z1u

Total reclaimed space: 287.2MB
[Pipeline] }
[Pipeline] // stage
```



## Logi konsoli

```
Started by user admin
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Hello)
[Pipeline] echo
Hello World
[Pipeline] sh
+ git clone https://github.com/InzynieriaOprogramowaniaAGH/MDO2023_INO
Cloning into 'MDO2023_INO'...
[Pipeline] dir
Running in /var/jenkins_home/workspace/pipeline/MDO2023_INO
[Pipeline] {
[Pipeline] sh
+ git checkout KP406287
Switched to a new branch 'KP406287'
Branch 'KP406287' set up to track remote branch 'KP406287' from 'origin'.
[Pipeline] }
[Pipeline] // dir
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

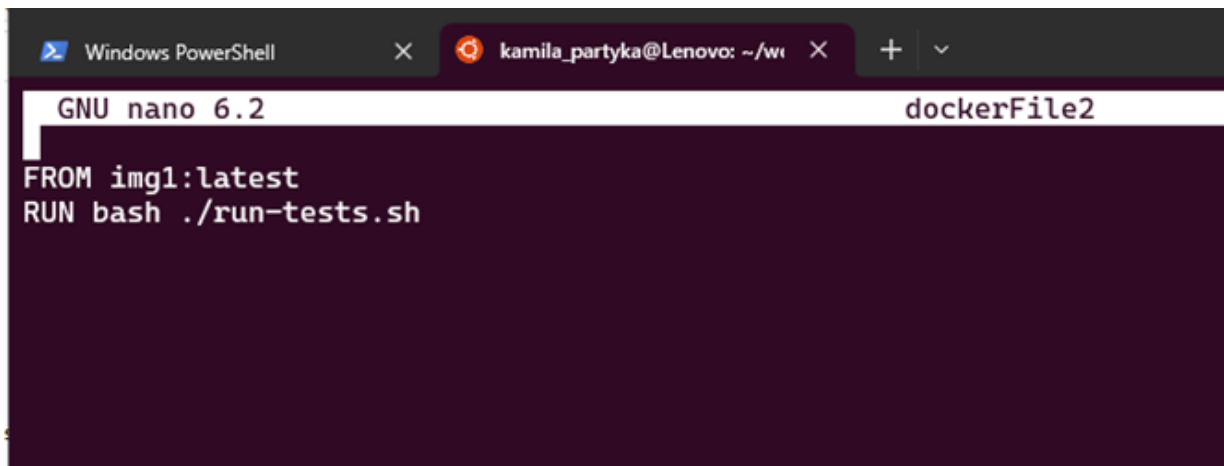
## Końcówka wydruku:

```
#12 0.756 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/draw.c -o build/draw.o
#12 0.905 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/history.c -o build/history.o
#12 1.061 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/main.c -o build/main.o
#12 1.270 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/numberstack.c -o build/numberstack.o
#12 1.386 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/operators.c -o build/operators.o
#12 1.561 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/parser.c -o build/parser.o
#12 1.712 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -c src/xmalloc.c -o build/xmalloc.o
#12 1.757 gcc -Wall -Wextra -g -Werror=missing-declarations -Werror=redundant-decls -Iinclude -o bin/palc build/draw.o build/history.o build/main.o
build/numberstack.o build/operators.o build/parser.o build/xmalloc.o -lncurses #
#12 1.805 Executing all complete!
#12 DONE 1.9s

#13 exporting to image
#13 exporting layers
#13 exporting layers 15.4s done
#13 writing image sha256:9b13d59ad41fbc911964aefcf4e00af70bf317a0fd78f49b0b286a5b1c791859 0.0s done
#13 naming to docker.io/library/bldr 0.0s done
#13 DONE 15.5s
[Pipeline] }
[Pipeline] // dir
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

## TESTY

- W pliku DockerFile2 uruchamiane są testy:



The screenshot shows a Windows PowerShell terminal window with two tabs. The active tab is titled 'kamil\_partyka@Lenovo: ~/w...' and contains a file named 'dockerFile2' opened in 'GNU nano 6.2' editor. The file content is as follows:

```
FROM img1:latest
RUN bash ./run-tests.sh
```

- W tym etapie wyświetlamy w konsoli informację "TESTS" za pomocą polecenia "echo 'TESTS'".
- Następnie wchodzimy do folderu "MDO2023\_INO/INO/GCL2/KP406287/lab03" za pomocą polecenia "dir('MDO2023\_INO/INO/GCL2/KP406287/lab03')".
- W tym folderze wykonujemy polecenie "docker build . -f DockerFile2 -t tester", które buduje obraz Dockera o nazwie "tester" na podstawie pliku Dockerfile2 znajdującego się w bieżącym folderze.

```
}
stage('Tests'){
  steps {
    echo 'TESTS'
    dir('MDO2023_INO/INO/GCL2/KP406287/lab03'){
      sh 'docker build . -f DockerFile2 -t tester'
    }
  }
}
```

```
TESTS
[Pipeline] dir
Running in /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03
[Pipeline] {
[Pipeline] sh
+ docker build . -f Dockerfile2 -t tester
#1 [internal] load .dockerignore
#1 transferring context: 28 done
#1 DONE 0.1s

#2 [internal] load build definition from Dockerfile2
#2 transferring dockerfile: 75B done
#2 DONE 0.1s

#3 [internal] load metadata for docker.io/library/builder:latest
#3 DONE 0.0s

#4 [1/2] FROM docker.io/library/builder
#4 DONE 0.3s

#5 [2/2] RUN bash ./run-tests.sh
#5 0.460 All tests passed
#5 DONE 0.5s

#6 exporting to image
#6 exporting layers 0.0s done
#6 writing image sha256:9b13d59ad41fbc911964aefcf4e00af70bf317a0fd78f49b0b286a5b1c791859 0.0s done
#6 naming to docker.io/library/tester done
#6 DONE 0.1s
[Pipeline] }
[Pipeline] // dir
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

## Deploy

"FROM ubuntu:latest" to linia używana w pliku Dockerfile, która określa obraz bazowy, na podstawie którego budowany jest nowy obraz. W tym przypadku "ubuntu:latest" oznacza, że budowany obraz Dockera będzie oparty na najnowszej wersji obrazu Ubuntu, dostępnej w repozytorium Dockera. W plikach Dockerfile często wykorzystuje się obrazy bazowe, aby uprościć proces budowania i utrzymać zgodność z systemem operacyjnym i bibliotekami, na których aplikacja będzie działać.



Ten fragment kodu to kolejny etap w skrypcie Jenkins pipeline, który zajmuje się wdrożeniem aplikacji na serwerze. Opisuje on trzy kroki:

1. Kopiowanie pliku wykonywalnego pcalc z kontenera zbudowanego na etapie Build (builder) i kopiowanie go na wolumin (lokalizację) /var/jenkins\_home/workspace/pipeline/MDO2023\_INO/INO/GCL2/KP406287/lab03 na hoście. W tym celu uruchamiana jest komenda docker run z flagą --volume, która tworzy połączenie między hostem a kontenerem, a następnie wywołuje polecenie mv wewnątrz kontenera, aby skopiować plik pcalc do woluminu.
2. Budowanie nowego obrazu Docker o nazwie deploy na podstawie pliku DockerFile3 znajdującego się w katalogu MDO2023\_INO/INO/GCL2/KP406287/lab03. W tym celu wykorzystywana jest komenda docker build.
3. Uruchomienie nowego kontenera z obrazem deploy, podpinając wolumin /var/jenkins\_home/workspace/pipeline/MDO2023\_INO/INO/GCL2/KP406287/lab03 do katalogu /v wewnątrz kontenera, a następnie wykonywanie komendy ps aux wewnątrz kontenera za pomocą docker exec. Na końcu zostaje zatrzymany i usunięty kontener.

```
stage('Deploy') {  
  steps {  
    sh 'docker run --volume /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03:/v builder mv pcalc /v'  
    dir('INO/GCL2/A0407699/lab5') {  
      sh 'docker build . -f DockerFile3 -t deploy'  
    }  
    sh '''  
    docker run -t --name deployPcalc --volume /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03:/v deploy v/pcalc&  
    '''  
    sh 'docker exec deployPcalc ps aux'  
    sh 'docker stop deployPcalc'  
    sh 'docker container rm deployPcalc'  
  }  
}
```

Wolumin /v w kontekście tego kodu w Jenkinsie jest punktem montowania (mount point) w kontenerze Docker, co oznacza, że jest to sposób na przekazywanie plików między hostem Jenkinsa a kontenerem Docker. W tym konkretnym przypadku, plik wykonywalny "pcalc" oraz inne pliki są przekazywane poprzez punkt montowania /v. Plik "pcalc" jest kopiowany z hosta Jenkinsa do woluminu /v na hoście, a następnie podmontowywany jako wolumin /v wewnątrz kontenera Docker, w celu wdrożenia aplikacji w środowisku docelowym.

W ten sposób, kontener Docker ma dostęp do pliku "pcalc" znajdującego się w woluminie /v, który został przekazany z hosta Jenkinsa. To umożliwia wdrożenie aplikacji w kontenerze Docker, w którym działa aplikacja, która jest testowana w poprzednich etapach Jenkinsa. Po zakończeniu wdrożenia aplikacji, wolumin /v jest odmontowywany z kontenera Docker i pliki zostają zaktualizowane na hoście Jenkinsa.

## Publish

Jeśli proces budowania, testowania i wdrażania przebiegnie pomyślnie, to następuje publikacja nowej wersji programu. W kontenerze kopiowany jest tylko plik z kodem wykonywalnym.



The screenshot shows a terminal window with two tabs: 'Windows PowerShell' and 'kamila\_partyka@Lenovo: ~/w'. The active tab displays the content of a Dockerfile and a Jenkins pipeline script. The Dockerfile has a dark background with light blue text, showing 'FROM builder:latest' and 'RUN tar rvf pcalc.tar pcalc'. The Jenkins pipeline script is on a light background with dark text, showing a 'Publish' stage with steps for echoing 'PUBLISH', changing the directory to the Jenkins workspace, building a Docker image named 'publisher', and running a container named 'publisher' with a volume mount for '/v'.

```
GNU nano 6.2 DockerFile4
FROM builder:latest
RUN tar rvf pcalc.tar pcalc

stage('Publish') {
  steps {
    echo 'PUBLISH'
    dir('MDO2023_INO/INO/GCL2/KP406287/lab03') {
      sh 'docker build . -f DockerFile4 -t publisher'
    }
    sh "docker run --volume /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03:/v publisher mv pcalc.tar /v"
  }
}
```

1. wyświetlenie tekstu "PUBLISH" za pomocą funkcji **echo** ,
2. zmiana bieżącego katalogu na 'MDO2023\_INO/INO/GCL2/KP406287/lab03' za pomocą funkcji **dir** ,
3. zbudowanie kontenera Docker o nazwie 'publisher' na podstawie pliku DockerFile4 za pomocą polecenia **docker build** ,
4. uruchomienie kontenera Docker o nazwie 'publisher' i zamontowanie woluminu z katalogiem '/v' wewnątrz kontenera,
5. przeniesienie pliku 'pcalc.tar' z kontenera do katalogu '/v' za pomocą polecenia **mv**.
6. W skrócie, ten fragment kodu buduje kontener Docker i przenosi plik 'pcalc.tar' do woluminu, który jest montowany wewnątrz kontenera.





Projekt został zakończony pomyślnie, gdyż udało się zrealizować efektywny proces, który pozwala na uzyskanie gotowego do wdrożenia artefaktu. W trakcie procesu publikacji, został wygenerowany plik "pcalc.tar.gz", który po zakończeniu sukcesem całego procesu, uznano za poprawny. Dzięki temu plikowi możliwe jest proste i skuteczne wdrożenie aplikacji na serwerze lub w chmurze. W ten sposób, proces tworzenia i publikacji aplikacji jest zautomatyzowany, co pozwala na oszczędność czasu oraz uniknięcie błędów wynikających z ręcznego wykonywania tych czynności.

```
pipeline {
  agent any

  stages {
    stage('Prepare') {
      steps {
        sh "rm -rf MDO2023_INO"
        sh "docker system prune --all --force"
      }
    }
    stage('Clone') {
      steps {
        echo ' '
        sh "git clone https://github.com/InzynieriaOprogramowaniaAGH/MDO2023_INO"
        dir('MDO2023_INO'){
          sh "git checkout KP406287"
        }
      }
    }
    stage('Build') {
      steps {
        echo ' '
        dir('MDO2023_INO/INO/GCL2/KP406287/lab03'){
          sh "docker build -t bldr ."
        }
      }
    }
    stage('Deploy') {
      steps {
        sh "docker run --volume /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03:/v builder mv pcalc /v"
        dir('MDO2023_INO/INO/GCL2/KP406287/lab03') {
          sh "docker build . -f DockerFile3 -t deploy"
        }
        sh '...'
        docker run -t --name deployPcalc --volume /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03:/v deploy v/pcalc&
        sh 'docker exec deployPcalc ps aux'
        sh 'docker stop deployPcalc'
        sh 'docker container rm deployPcalc'
      }
    }
    stage('Publish') {
      steps {
        echo 'PUBLISH'
        dir('MDO2023_INO/INO/GCL2/KP406287/lab03') {
          sh "docker build . -f DockerFile4 -t publisher"
        }
        sh "docker run --volume /var/jenkins_home/workspace/pipeline/MDO2023_INO/INO/GCL2/KP406287/lab03:/v publisher mv pcalc.tar /v"
      }
    }
  }
}
```

## PIPELINE from SCM

Pipeline script from SCM umożliwia przechowywanie skryptów pipeline w repozytorium kodu (na przykład w systemie kontroli wersji, takim jak Git), dzięki czemu można je wersjonować i kontrolować zmiany w skrypcie. Ponadto, skrypty pipeline są przechowywane poza Jenkins, co zapewnia większe bezpieczeństwo i niezawodność. W przypadku awarii Jenkinsa można łatwo odtworzyć skrypty pipeline z repozytorium kodu.

Pipeline script from SCM umożliwia również łatwiejszą współpracę i łatwiejsze zarządzanie skryptami pipeline w zespole. Każdy członek zespołu może łatwo zdalnie wprowadzać zmiany w skrypcie pipeline za pośrednictwem repozytorium kodu.

Dlatego warto rozważyć zmianę definicji pipeline'a z Pipeline script na Pipeline script from SCM, szczególnie w przypadku projektów, w których istnieje potrzeba współpracy i wersjonowania skryptów pipeline.

Definition

Pipeline script from SCM

SCM ?

Git

Repositories ?

Repository URL ?

[https://github.com/InzynieriaOprogramowaniaAGH/MDO2023\\_INO.git](https://github.com/InzynieriaOprogramowaniaAGH/MDO2023_INO.git)

Credentials ?

none

Add +

Zaawansowane ▾

Add Repository

Branches to build ?

Branch Specifier (blank for 'any') ?

\*/KP406287

Add Branch

Repository browser ?

(Automatycznie)

Additional Behaviours

Dodaj ▾

Script Path ?

/INO/GCL2/KP406287/lab03/Jenkinsfile