

Stanisław Marek	Grupa 05
Metodyki DevOps	Projekt Jenkins

JENKINS

Tak powinien prezentować się główny widok w Jenkinsie po poprawnym odtworzeniu kroków

[Dodaj opis](#)

Wszystkie +

S	P	Nazwa ↓	Ostatni sukces	Ostatni błąd	Czas trwania	Fav
		devops-golang-microservice	4 min 16 sek #61	14 min #59	2 min 40 sek	
		kozacka nazwa	—	—	nd.	

Widok na pipeline'y, które zostały wykonane w czasie tworzenia projektu

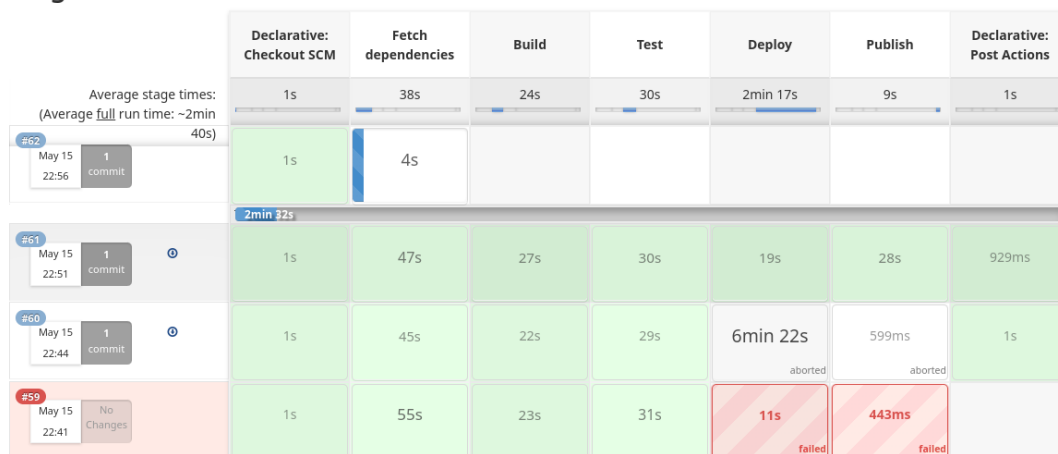
Pipeline devops-golang-microservice

Last Successful Artifacts

devops-golang-project 16.37 MB [view](#)

Recent Changes

Stage View



Ze względu na to że używany wcześniej projekt okazał się niemożliwy przystępnego uruchomienia zdecydowałem się wykorzystać inny projekt, który również został napisany w GoLangu. Uważam, że zmiana projektu na inny natomiast napisany w tym samym języku nie powinna sprawić problemu

Poprzedni projekt:

```
[stan@manjaro terraform-provider-azurerm]$ ./terraform-provider-azurerm
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

KONFIGURACJA PIPELINE'A


W Jenkinsie tworzymy nowy projekt i wybieramy pipeline oraz nadajemy nazwę

Jenkins Szukaj...


Tablica > Wszystkie >

Podaj nazwę projektu

» Pole wymagane

**Ogólny projekt**

To jest podstawowa funkcja Jenkinsa. Jenkins stworzy projekt łączący dowolny SCM z dowolnym systemem budującym, może to być również wykorzystane do czegoś innego niż budowanie oprogramowania.

**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.

Pipeline konfiguruję w taki sposób żeby korzystał ze skryptu umieszczonego bezpośrednio w podpiętym repozytorium. Jako branch wybieram master - jest to gałąź z której będzie publikowany projekt. Dodatkowo dodaję credentials do dockerhuba (o tym później)

General Build Triggers Advanced Project Options **Pipeline**

Pipeline

Definition

Pipeline script from SCM

SCM ?

Git

Repositories ?

Repository URL ?

https://github.com/StanMarek/devops-golang-microservice

Credentials ?

stanmarek/***** (use to login and push image) Add

Zaawansowane...

Add Repository

Branches to build ?

Branch Specifier (blank for 'any') ?

*/master

POBRANIE ZALEŻNOŚCI

Pierwszym etapem budowy aplikacji jest pobranie zależności potrzebnych do utworzenia pliku wykonywalnego, który jest domyślnym formatem budowania aplikacji pisanych z wykorzystaniem języka GO.

Rootem całej gałęzi budowy aplikacji jest obraz GoLangowy wykorzystujący Alpine Linux. Jest to najlepszy wybór gdy chcemy dockeryzować budowę aplikacji ze względu na rozmiar i brak niepotrzebnych rzeczy (przykładowo nie zawiera snapa ale również gita i make'a które w moim przypadku należy zainstalować)

Celem kontenera pobierającego zależności jest zapewnienie, że podczas budowy nie zabraknie niczego co jest konieczne do zbudowania pliku lub wykonania testów.

W przypadku aplikacji w Go potrzebny jest kompilator C oraz specjalna biblioteka wykorzystująca język C. Dodatkowo potrzebny jest program make, dzięki któremu możliwe jest wykonanie skryptów zawartych w Makefile.

Pierwszy stage - pobieranie zależności

Tworzymy dwa foldery, które wykorzystane zostaną jako woluminy wejściowy i wyjściowy.

Następnie buduję obraz alpine linuxa (wewnątrz niego dzieje się pobranie wszystkich zależności). UWAGA - dlaczego dodana jest flaga --no-cache? Podczas aktualizowania kodu przy budowaniu obrazu wykorzystywane było repo bez zmian (stara wersja). Użycie tej flagi sprawia, że kod zawsze (w moim przypadku) był aktualny, natomiast obarczone było to dłuższym trwaniem całego procesu.

Po tym uruchamiany jest kontener, który wykonuje polecenie kopiowania plików źródłowych na wolumin

```
stage("Fetch dependencies") {  
    steps {
```

```

        script {
            sh 'echo Downloading dependencies'

            sh 'mkdir -p shared_volume'
            sh 'mkdir -p shared_volume_out'

            sh 'docker build -t project-dep:latest -f
Dockerfile.dep --no-cache .'
            sh "docker run -v
\$(pwd) /shared_volume:/volumein project-dep:latest"

            sh 'ls ./shared_volume -la'
            sh 'ls shared_volume/source -la'
        }
    }
    post {
        success {
            sh 'echo Dependencies downloaded'
        }
        failure {
            sh 'echo Dependencies downloading failure'
        }
    }
}

```

Dockerfile wykorzystany w pierwszym stage'u

```

FROM golang:1.17-alpine

RUN apk add git make gcc libc-dev

RUN git clone
https://github.com/StanMarek/devops-golang-microservice.git
/source

WORKDIR /source

RUN go mod tidy

CMD ["cp", "-r", "/source", "/volumein/"]

```

BUILD

Wykonanie budowy możliwe jest dopiero po pobraniu zależności. Odpowiada za to drugi stage, gdzie wykorzystywany jest obraz utworzony w poprzednim. Skracać - bez zależności nie przeprowadzimy builda i tu się zatrzymamy.

Dlaczego wykorzystujemy obraz poprzedni a nie nowy obraz golang:<version>-alpine?

Odpowiedź jest prosta - nie po to wcześniej jest tworzony obraz zawierający niezbędne zależności żeby go nie wykorzystać. Przez to nie musimy ponownie pobierać przykładowego make'a, który w procesie budowania będzie niezbędny. Wykorzystany obraz w dalszym ciągu jest oparty na golang:alpine ale kryje się pod inną nazwą (w tym przypadku project-dep)

Skrypt budujący z pliku Make:

Wyjaśnienie: go build - może zostać wywołany bezargumentowo. Ja zdecydowałem dodać flagę -o, która nadaje nazwę dla tworzonego pliku binarnego. Argumentuję to na dużo czytelniejsze i łatwiejsze wykonanie kolejnego stage (Deploy) gdzie wykonywalny plik zostanie uruchomiony.

```
build:
go build -o devops-golang-project
```

Budowa drugiego stage - build

Budujemy obraz odpowiedzialny za zbudowanie/utworzenie pliku wykonywalnego.

Zbudowany obraz jest uruchamiany z podłączonymi dwoma woluminami. Polecenie z Dockerfile'a kopiuje plik binarny do woluminu wyjściowego - stage od deploya zobaczy jedynie ten plik. Nie ma potrzeby, żeby deploy miał dostęp do całego kodu źródłowego, podczas gdy potrzebny mu jest jedynie plik exe.

```
stage('Build') {
    steps {
        script {
            sh 'echo Building'

            sh 'docker build -t project-build:latest -f
Dockerfile.build .'

            sh "docker run -v
\\$(pwd)/shared_volume:/volumein -v
\\$(pwd)/shared_volume_out:/volumeout project-build:latest"

            sh 'ls shared_volume_out/ -la'
        }
    }
    post {
        success {
            sh 'echo Build finished'
        }
    }
}
```

```

        failure {
            sh 'echo Build failure'
        }
    }
}

```

Dockerfile przeznaczony do builda

```

FROM project-dep:latest

RUN make build

CMD ["cp", "devops-golang-project", "/volumeout/"]

```

TEST

Kontener testujący jest oparty na pierwszym - odpowiadającym za pobranie zależności a nie drugim - buildzie. Go zapewnia możliwość testowania kodu bez wcześniejszej kompilacji, więc wykorzystanie obrazu wyłącznie z dependencjami wydaje się słuszne. W wykorzystanym projekcie nie znalazło się za dużo testów (jedynie dwa pliki), natomiast nie przeszkadza to aby stage został wykonany ze względu na wartość pokazową (ważne, że skrypt się wykonuje)

W Makefile znajdują się dwa skrypty do testowania

```

unit-tests:
    go test ./...

functional-tests:
    go test ./functional_tests/transformer_test.go

```

Budowa stage'a do testów

Zasada działania analogiczna do poprzedniego stage. Tutaj jedynie entrypointem kontenera jest wydruk w konsoli, że testy się powiodły.

```

stage('Test') {
    steps {
        script {
            sh 'echo Testing'

            sh 'docker build -t project-test -f
Dockerfile.test .'

```

```

        sh "docker run -v
\$(pwd) /shared_volume:/volumein project-test:latest"

        sh 'ls ./shared_volume -la'
        sh 'ls shared_volume/source -la'
    }
}
post {
    success {
        sh 'echo Tests finished'
    }
    failure {
        sh 'echo Tests failure'
    }
}
}

```

Dockerfile dla testów

```

FROM project-dep:latest

RUN make unit-tests

RUN make functional-tests

ENTRYPOINT [ "echo", "tested in container" ]

```

DEPLOY

Deploy jest oparty bezpośrednio na buildzie. Tworzony obraz będzie bezpośrednio wykorzystany do publikacji w dockerhub pod warunkiem, że stage przejdzie poprawnie. Tym razem obraz umieszczany przyjmuje tag bezpośrednio związany ze mną i projektem. Dodatkowo aby rozróżnić jaką wersję publikuję dodaję ID commita, z którego utworzony został niniejszy obraz

Budowa stage'a deploy

Warto zdefiniować zmienne, które wykorzystamy do budowania obrazu i uruchomienia kontenera. Utworzyłem dwie zmienne - tag obrazu oraz nazwę kontenera (przydatne gdy nie chcemy męczyć się z ID kontenerów złożonymi z losowych cyfr i liter).

Po utworzeniu obrazu uruchamiam kontener z flagami --rm (kontener zostanie usunięty po zakończeniu jego działania przez co nazwa będzie wielokrotnego użytku); -d (detach - tryb

dzięki któremu testowy kontener działa w tle i możliwe jest jego zatrzymanie aby nie blokował dalej pipelinea. UWAGA - to był jeden z moich błędów gdzie początkowo nie użyłem tej flagi - skutkowało to nieskończonym pipelinem w stage'u deploy przez co musiałem go ręcznie "ubić"; --name (nazwa kontenera pomaga nim zarządzać wykorzystując tę nazwę zamiast ID)

Po uruchomieniu kontenera wypisuję logi z niego - warto to zrobić w celu sprawdzenia poprawności działania (w tym przypadku brakowało działającego Redisa ale zakładam, że produkt jest działający zakładając, że klient dostarczy go [redisa] z zewnątrz)

Następnie kontener jest zatrzymany i automatycznie usunięty żeby nie blokować nazwy

```
stage('Deploy') {
    steps {
        script {
            sh 'echo Deploying'

            def TAG_COMMIT = GIT_COMMIT
            def CONTAINER_NAME = 'deploy-test'

            sh "docker build -t
stanmarek/devops-golang-project:${TAG_COMMIT} -f Dockerfile.deploy
."

            sh "docker run --rm -d --name ${CONTAINER_NAME}
-v \$(pwd)/shared_volume_out:/volumeout
stanmarek/devops-golang-project:${TAG_COMMIT}"
            sh "docker logs ${CONTAINER_NAME}"

            sh 'sleep 10'

            sh "docker stop ${CONTAINER_NAME}"
        }
    }
    post {
        success {
            sh 'echo Deploy finished'
        }
        failure {
            sh 'echo Deploy failure'
        }
    }
}
```

Dockerfile do deploy


```
FROM project-build:latest

EXPOSE 9090

CMD ["/devops-golang-project"]
```

PUBLISH

Publikacja na dockerhub jest bardzo prosta. Dzięki uprzejmości Jenkinsa jesteśmy w stanie zdefiniować tzw. Credential do różnych serwisów, dlatego dodałem login oraz hasło (zaszyfrowane) do dockerhub. Aby to zrobić w ustawieniach użytkowników Jenkinsa w zakładce Credentials należy dodać nowe z unikatowym ID.

Scope ?
Global (Jenkins, nodes, items, all child items, etc) ▼

Username ?
stanmarek

☐ Treat username as secret ?

Password ?
Concealed Change Password

ID ?
dockerhub

Description ?
use to login and push image

Save

Następnie bezpośrednio w pipeline pobieramy login oraz hasło jako zmienne środowiskowe. Logujemy się poprzez docker login i wypychamy obraz do rejestru za pomocą docker push.

Budowa stage'a publikacji

```
stage('Publish') {
    agent any
    steps {
        withCredentials([usernamePassword(credentialsId:
'dockerhub', passwordVariable: 'dockerhubPassword',
usernameVariable: 'dockerhubUser')]) {
            sh "docker login -u ${dockerhubUser} -p
${dockerhubPassword}"
        }
    }
}
```

```
        sh "docker push
stanmarek/devops-golang-project:${GIT_COMMIT}"
    }
}
}
```

W naszym profilu powinien ukazać się spis obrazów

The screenshot shows the Docker Hub interface for the repository `stanmarek/devops-golang-project`. The repository is public and has no description. It was last pushed a few seconds ago. The **Tags and Scans** section shows two tags: `f5b2a518bf073fedfac...` (pushed a few seconds ago) and `latest` (pushed 10 minutes ago). The **Automated Builds** section indicates that manual pushes are being used instead of automated builds. A **Docker commands** section provides the command `docker push stanmarek/devops-golang-project:tagname`.

TAG	OS	PULLED	PUSHED
<code>f5b2a518bf073fedfac...</code>	Linux	---	a few seconds ago
<code>latest</code>	Linux	---	10 minutes ago

W moim przypadku obraz `latest` jest starszy niż ostatnio dodany, wynika to ze względu na to, że podczas tworzenia pipeline'a tagowanie obrazów przez ID commita zostało dodane później niż opublikowany został pierwszy obraz. Możliwe jest tagowanie najnowszego poprzez `latest` a poprzednich poprzez ID commita.

POST STAGES - Zapisanie artefaktu

```
post {
    always {
        archiveArtifacts artifacts:
'shared_volume_out/devops-golang-project', fingerprint: true
    }
}
```

Uzyskany artefakt - plik wykonywalny uzyskany z builda

✓ Build #62 (15 maj 2022, 20:56:31)



Build Artifacts

devops-golang-project 16.37 MB [view](#)



Changes

1. Finish complete ([details](#) / [githubweb](#))



Wystartowane przez użytkownika [Stanisław Marek](#)



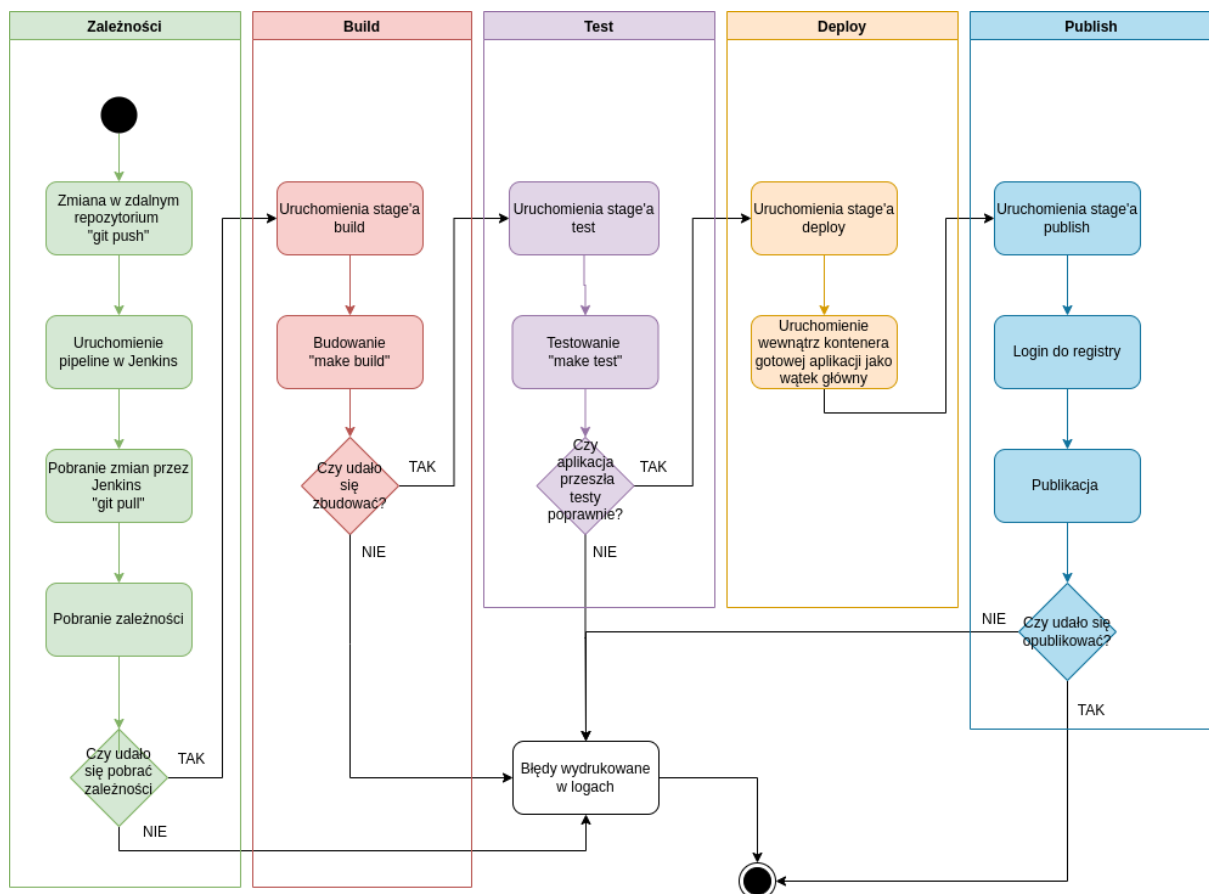
Revision: 44eb7e683fce4973d89fbe2efdb28c8e627b9083

Repository: <https://github.com/StanMarek/devops-golang-microservice>

• refs/remotes/origin/master

Logi z całego pipeline znajdują się w pliku tekstowym

DIAGRAM AKTYWNOŚCI



Omówienie pipeline:

- Początek pipeline – kiedy wykryte są zmiany na zdalnym repozytorium git push uruchamia się Jenkins i wykonuje instrukcje zawarte w Jenkinsfile.
- Repozytorium jest zaktualizowane do najnowszej wersji git pull

- c) W odpowiednim folderze głównym z plikami Dockerfile.dep uruchamiany jest proces pobierania zależności
- d) Instalacja wymaganych zależności oraz budowanie projektu, testowanie i deploy, kolejne Dockerfiles
- e) W przypadku gdy budowanie lub inny z w/w stage'y się nie powiedzie Jenkins kończy swoją pracę pozostawiając logi z błędami. W przeciwnym wypadku możemy zobaczyć logi o poprawnym zbudowaniu itp. i przechodzimy dalej
- f) Jenkins utworzony obraz po stage'u Deploy loguje się do dockerhuba i publikuje go

DIAGRAM WDROŻENIA

