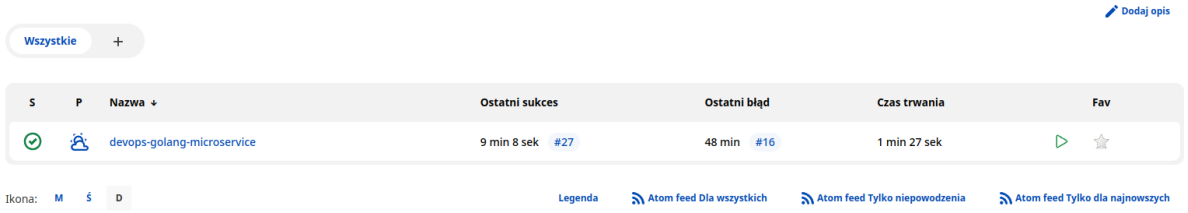
Stanisław Marek	Grupa 05
Metodyki DevOps	Projekt Jenkins

JENKINS

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



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

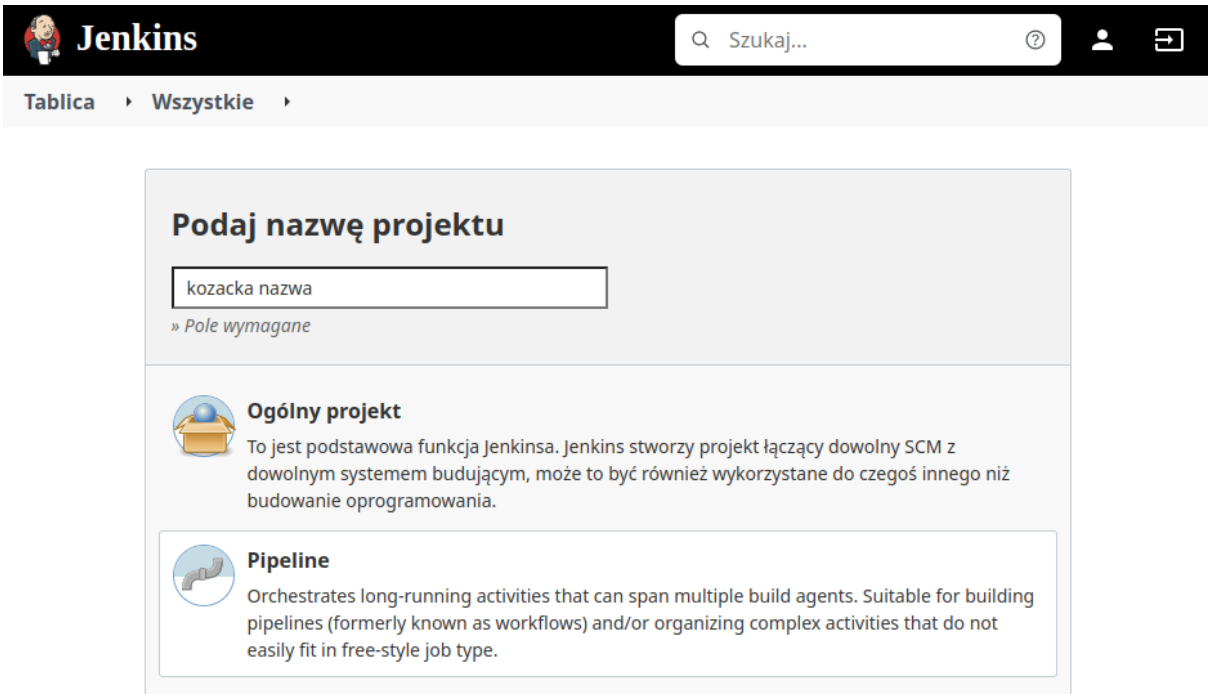
Ze względu na to że używany wcześniej projekt okazał się niemożliwy przystępnego uruchomienia zdecydowałem się wykorzystać inny projekt, który również został napisany w GoLangu. Uważam, że zmiana projektu na inny natomiast napisany w tym samym języku nie powinna sprawić problemu

Poprzedni projekt:

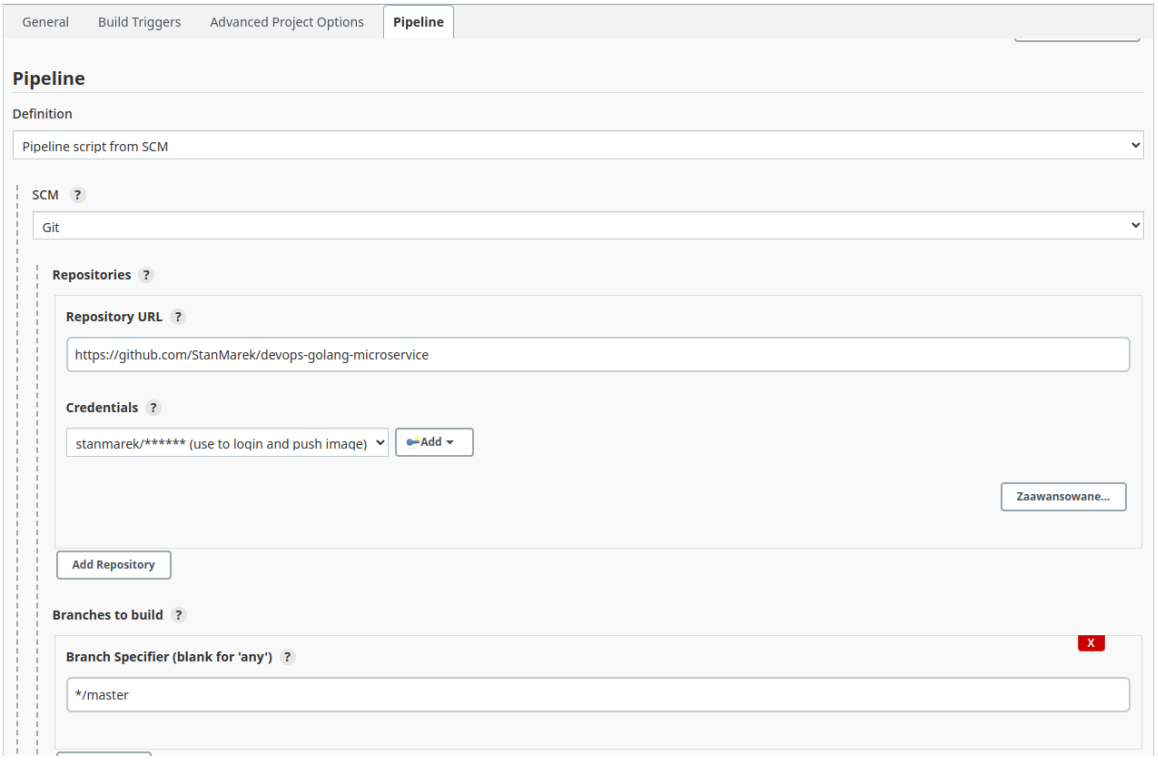
```
[stan@manjaro terraform-provider-azurerm]$ ./terraform-provider-azurerm
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

KONFIGURACJA PIPELINE'A

W jenkinsie tworzymy nowy projekt i wybieramy pipeline oraz nadajemy nazwę



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)



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 gita lub programu make)
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

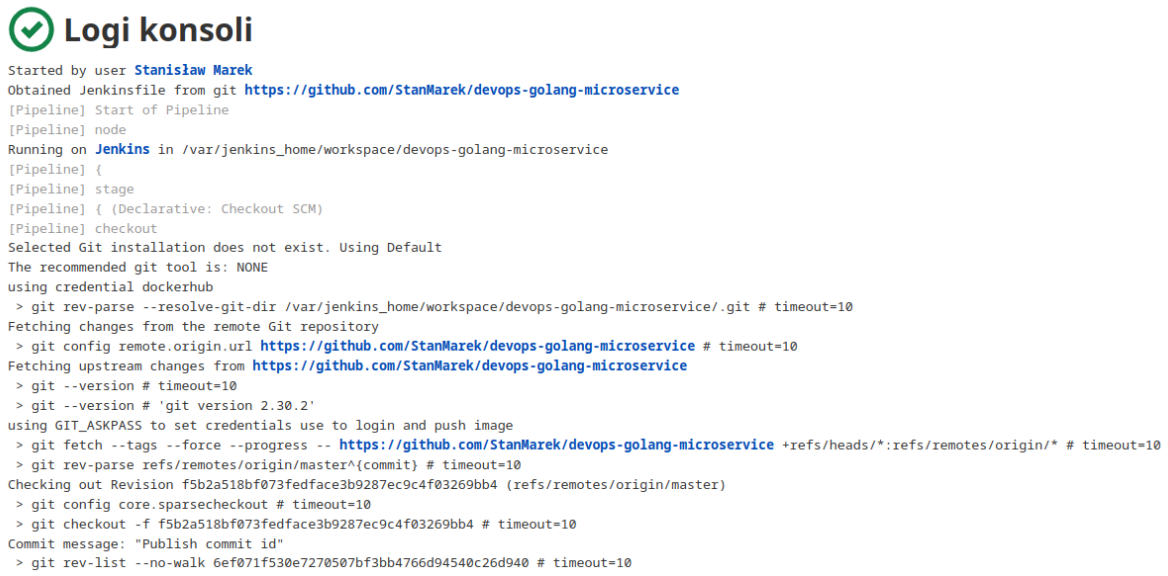
```
stage("Fetch dependencies") {
            steps {
                script {
                    sh 'echo Downloading dependencies'
                    sh 'docker build -t project-dep:latest -f
Dockerfile.dep .'
                }
            }
            post {
```

```
                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 make gcc libc-dev

COPY . /server
```

Przebieg pobierania zależności



Logi konsoli

```
Started by user Stanisław Marek
Obtained Jenkinsfile from git https://github.com/StanMarek/devops-golang-microservice
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/devops-golang-microservice
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Checkout SCM)
[Pipeline] checkout
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
using credential dockerhub
 > git rev-parse --resolve-git-dir /var/jenkins_home/workspace/devops-golang-microservice/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/StanMarek/devops-golang-microservice # timeout=10
Fetching upstream changes from https://github.com/StanMarek/devops-golang-microservice
 > git --version # timeout=10
 > git --version # 'git version 2.30.2'
using GIT_ASKPASS to set credentials use to login and push image
 > git fetch --tags --force --progress -- https://github.com/StanMarek/devops-golang-microservice +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision f5b2a518bf073fedface3b9287ec9c4f03269bb4 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f f5b2a518bf073fedface3b9287ec9c4f03269bb4 # timeout=10
Commit message: "Publish commit id"
 > git rev-list --no-walk 6ef071f530e7270507bf3bb4766d94540c26d940 # timeout=10
```

```
+ echo Downolading dependencies
Downolading dependencies
[Pipeline] sh
+ docker build -t project-dep:latest -f Dockerfile.dep .
Sending build context to Docker daemon  4.146MB

Step 1/3 : FROM golang:1.17-alpine
 ---> a15b7f2336ee
Step 2/3 : RUN apk add make gcc libc-dev
 ---> Using cache
 ---> 5249e088c5ae
Step 3/3 : COPY . /server
 ---> 60633510179f
Successfully built 60633510179f
Successfully tagged project-dep:latest
[Pipeline] }
[Pipeline] // script
Post stage
[Pipeline] sh
+ echo Dependencies downloaded
Dependencies downloaded
```

BUILD

Wykonanie budowy możliwe jest dopiero po pobraniu zależności. Odpowiada za to drugi stage, gdzie wykorzystywany jest obraz utworzony w poprzednim. Skracając - bez dependencji 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)
Brak wykorzystania volumes w tym stage: zdecydowałem, że nie będę wykorzystywać woluminów in oraz out, ponieważ repozytorium jest bezpośrednio podłączone do Jenkinsa. Oznacza to, że cały source jest już widoczny i nie istnieje potrzeba pobierania go (ponownie) do woluminu wejściowego tylko po to żeby skopiować go do woluminu wyjściowego jeżeli wszystko już jest w kontenerze. Zostało to zapewnione w poprzednim pipeline gdzie za pomocą ostatniego polecenia COPY wszystkie pliki zostały przeniesione do katalogu roboczego, który nazwałem 'server' - Deploy jest oparty na buildzie

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

```
stage('Build') {
            steps {
                script {
                    sh 'echo Building'
                    sh 'docker build -t project-build:latest -f
Dockerfile.build .'
                }
            }
            post {
                success {
                    sh 'echo Build finished'
                }
                failure {
                    sh 'echo Build failure'
                }
            }
        }
```

Dockerfile przeznaczony do builda

```
FROM project-dep:latest

WORKDIR /server

RUN go mod tidy

RUN make build
```

Przebieg builda

```
+ echo Building
Building
[Pipeline] sh
+ docker build -t project-build:latest -f Dockerfile.build .
Sending build context to Docker daemon  4.146MB

Step 1/4 : FROM project-dep:latest
 ---> 60633510179f
Step 2/4 : WORKDIR /server
 ---> Running in 34730a0fda97
Removing intermediate container 34730a0fda97
 ---> 72d11d80e24c
Step 3/4 : RUN go mod tidy
 ---> Running in fb0b7ef3fde3
[91mgo: downloading go.uber.org/zap v1.19.1
[0m[91mgo: downloading github.com/go-redis/redis/v8 v8.11.4
[0m[91mgo: downloading github.com/jasonlvhit/gocron v0.0.1
[0m[91mgo: downloading github.com/go-redis/redis v6.15.5+incompatible
[0m[91mgo: downloading github.com/jarcoal/httpmock v1.1.0
[0m[91mgo: downloading gopkg.in/yaml.v2 v2.4.0
[0m[91mgo: downloading github.com/jinzhu/gorm v1.9.16
[0m[91mgo: downloading github.com/gorilla/handlers v1.5.1
[0m[91mgo: downloading gopkg.in/tomb.v1 v1.0.0-20141024135613-dd632973f1e7
go: downloading github.com/fsnotify/fsnotify v1.4.9
[0m[91mgo: downloading golang.org/x/xerrors v0.0.0-20200804184101-5ec99f83aff1
[0m[91mgo: downloading golang.org/x/text v0.3.6
[0mRemoving intermediate container fb0b7ef3fde3
 ---> 42d63d8497c3
Step 4/4 : RUN make build
 ---> Running in ccb1d8e97f10
go build -o /devops-golang-project
Removing intermediate container ccb1d8e97f10
 ---> 0315d1e1fb2d
Successfully built 0315d1e1fb2d
Successfully tagged project-build:latest
[Pipeline] }
[Pipeline] // script
Post stage
[Pipeline] sh
+ echo Build finished
Build finished
```

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

```
stage('Test') {
            steps {
                script {
                    sh 'echo Testing'
                    sh 'docker build -t project-test -f
Dockerfile.test .'
                }
            }
            post {
                success {
                    sh 'echo Tests finished'
                }
                failure {
                    sh 'echo Tests failure'
                }
            }
        }
```

Dockerfile dla testów

```
FROM project-build:latest

WORKDIR /server

RUN make unit-tests

RUN make functional-tests
```

Przebieg testów

```
Testing
[Pipeline] sh
+ docker build -t project-test -f Dockerfile.test .
Sending build context to Docker daemon  4.146MB

Step 1/4 : FROM project-build:latest
 ---> 0315d1e1fb2d
Step 2/4 : WORKDIR /server
 ---> Running in b5fb504e0207
Removing intermediate container b5fb504e0207
 ---> 59a1da011472
Step 3/4 : RUN make unit-tests
 ---> Running in 6fc844d75f9c
go test ./...
?       github.com/shadowshot-x/micro-product-go        [no test files]
?       github.com/shadowshot-x/micro-product-go/authservice    [no test files]
?       github.com/shadowshot-x/micro-product-go/authservice/data       [no test files]
?       github.com/shadowshot-x/micro-product-go/authservice/jwt        [no test files]
?       github.com/shadowshot-x/micro-product-go/authservice/middleware [no test files]
?       github.com/shadowshot-x/micro-product-go/clientclaims   [no test files]
?       github.com/shadowshot-x/micro-product-go/couponservice  [no test files]
?       github.com/shadowshot-x/micro-product-go/couponservice/couponregionclient       [no test files]
?       github.com/shadowshot-x/micro-product-go/couponservice/store    [no test files]
?       github.com/shadowshot-x/micro-product-go/monitormodule  [no test files]
ok      github.com/shadowshot-x/micro-product-go/ordertransformerservice        0.003s
?       github.com/shadowshot-x/micro-product-go/ordertransformerservice/store  [no test files]
?       github.com/shadowshot-x/micro-product-go/productservice [no test files]
?       github.com/shadowshot-x/micro-product-go/productservice/store   [no test files]
Removing intermediate container 6fc844d75f9c
 ---> 515e4cff6f73
Step 4/4 : RUN make functional-tests
 ---> Running in c5c37b7a530d
go test ./functional_tests/transformer_test.go
ok      command-line-arguments  0.002s
Removing intermediate container c5c37b7a530d
 ---> 4081f70fc01d
Successfully built 4081f70fc01d
Successfully tagged project-test:latest
[Pipeline] }
[Pipeline] // script
Post stage
[Pipeline] sh
+ echo Tests finished
Tests finished
```

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

```
stage('Deploy') {
            steps {
                script {
                    sh 'echo Deploying'
                    def TAG_COMMIT = GIT_COMMIT
```

```
                    sh "docker build -t
stanmarek/devops-golang-project:${TAG_COMMIT} -f Dockerfile.deploy
."
                }
            }
            post {
                success {
                    sh 'echo Deploy finished'
                }
                failure {
                    sh 'echo Deploy failure'
                }
            }
```

Dockerfile do deploy

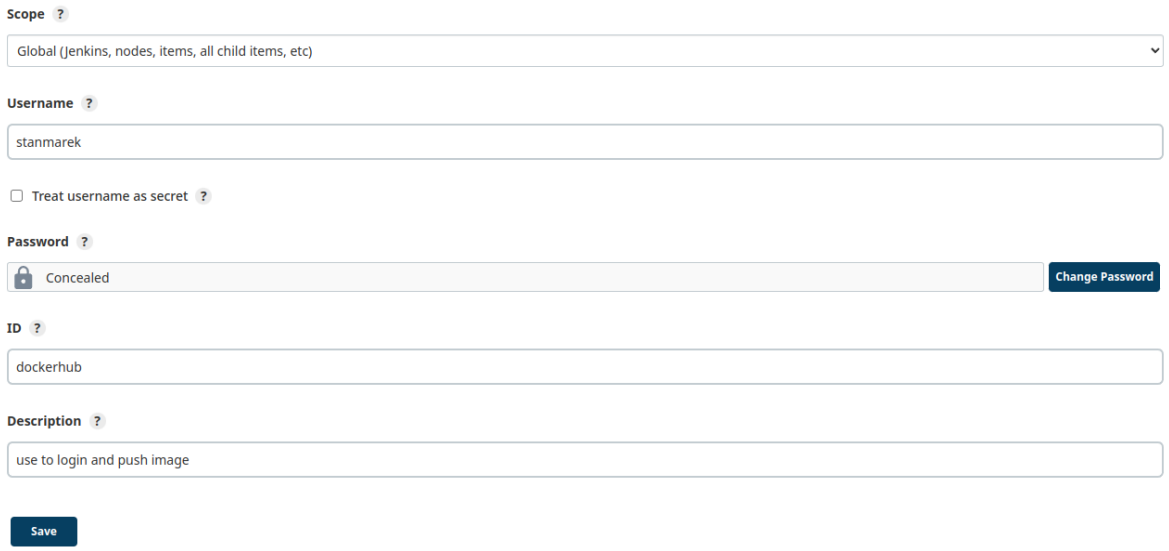
```
FROM project-build:latest

WORKDIR /server

EXPOSE 9090

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

Przebieg deploy

```
+ echo Deploying
Deploying
[Pipeline] sh
+ docker build -t stanmarek/devops-golang-project:f5b2a518bf073fedface3b9287ec9c4f03269bb4 -f Dockerfile.deploy .
Sending build context to Docker daemon  4.146MB

Step 1/4 : FROM project-build:latest
 ---> 0315d1e1fb2d
Step 2/4 : WORKDIR /server
 ---> Using cache
 ---> 59a1da011472
Step 3/4 : EXPOSE 9090
 ---> Running in 11c9e55860d9
Removing intermediate container 11c9e55860d9
 ---> 151e97a53ab1
Step 4/4 : CMD ["/devops-golang-project"]
 ---> Running in 6026c591eec6
Removing intermediate container 6026c591eec6
 ---> e5ccd4f49f3b
Successfully built e5ccd4f49f3b
Successfully tagged stanmarek/devops-golang-project:f5b2a518bf073fedface3b9287ec9c4f03269bb4
[Pipeline] }
[Pipeline] // script
Post stage
[Pipeline] sh
+ echo Deploy finished
Deploy finished
```

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.



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

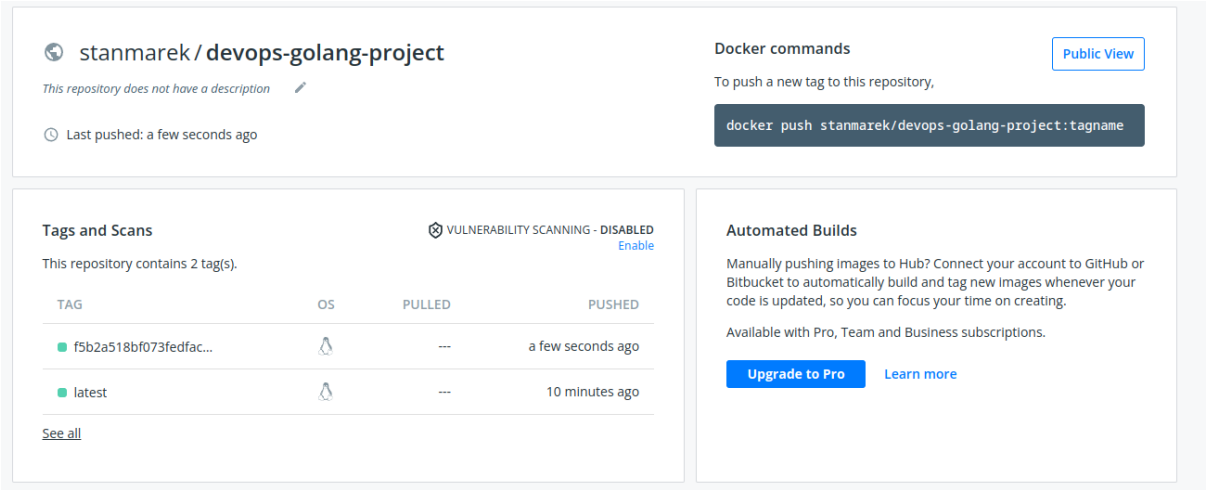
Budowa stage'a publikacji

```
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}"
                }
            }
        }
```

Przebieg publish

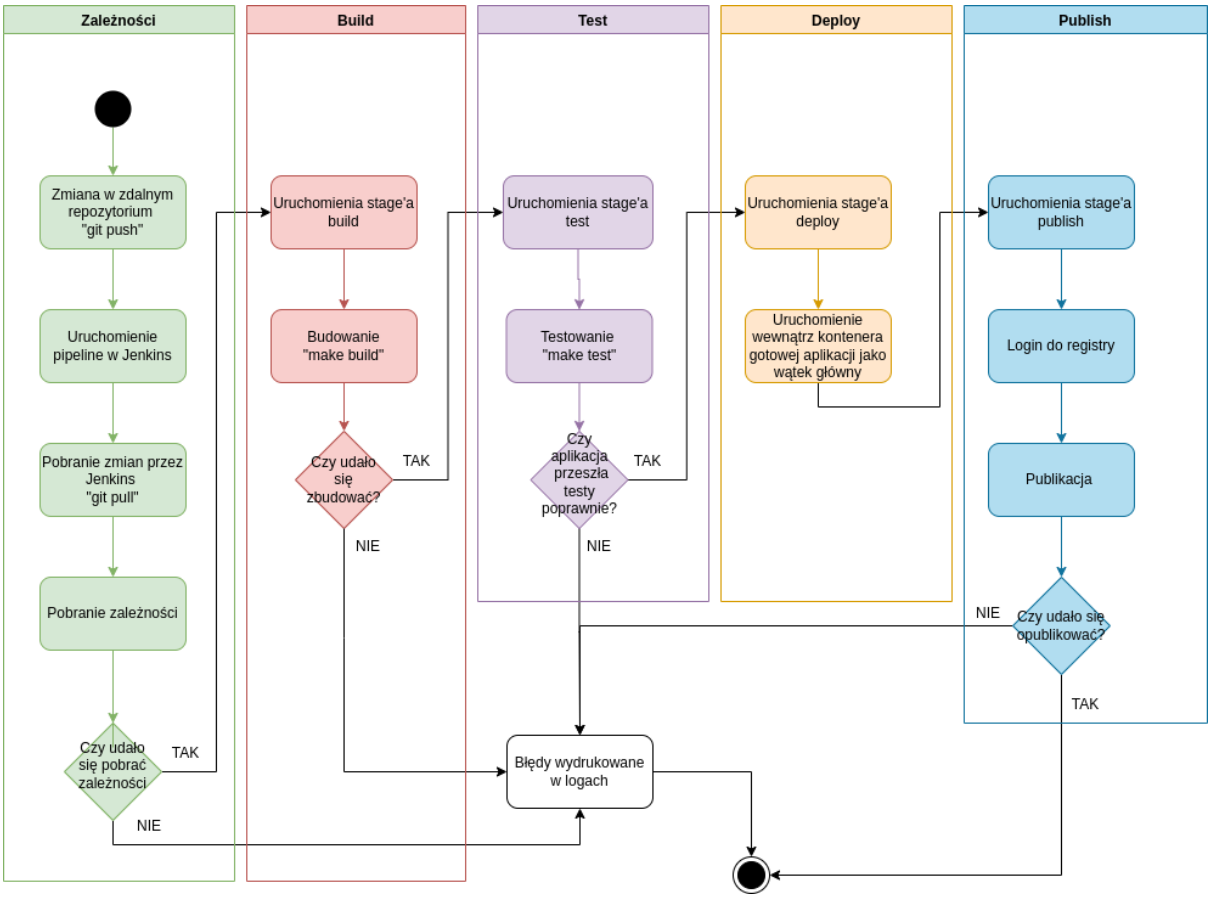
```
Running on Jenkins in /var/jenkins_home/workspace/devops-golang-microservice@2
[Pipeline] {
[Pipeline] checkout
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
using credential dockerhub
 > git rev-parse --resolve-git-dir /var/jenkins_home/workspace/devops-golang-microservice@2/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/StanMarek/devops-golang-microservice # timeout=10
Fetching upstream changes from https://github.com/StanMarek/devops-golang-microservice
 > git --version # timeout=10
 > git --version # 'git version 2.30.2'
using GIT_ASKPASS to set credentials use to login and push image
 > git fetch --tags --force --progress -- https://github.com/StanMarek/devops-golang-microservice +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision f5b2a518bf073fedface3b9287ec9c4f03269bb4 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f f5b2a518bf073fedface3b9287ec9c4f03269bb4 # timeout=10
Commit message: "Publish commit id"
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withCredentials
Masking supported pattern matches of $dockerhubPassword
[Pipeline] {
[Pipeline] sh
Warning: A secret was passed to "sh" using Groovy String interpolation, which is insecure.
                 Affected argument(s) used the following variable(s): [dockerhubPassword]
                 See https://jenkins.io/redirect/groovy-string-interpolation for details.
+ docker login -u stanmarek -p ****
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /var/jenkins_home/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[Pipeline] sh (hide)
+ docker push stanmarek/devops-golang-project:f5b2a518bf073fedface3b9287ec9c4f03269bb4
The push refers to repository [docker.io/stanmarek/devops-golang-project]
7281277fa00b: Preparing
3145398e7473: Preparing
c082bc6e7cd9: Preparing
ded76dea467d: Preparing
e047119c9fda: Preparing
ad8b951eb5e4: Preparing
4e797c290268: Preparing
09e3373b9d9a: Preparing
4fc242d58285: Preparing
ad8b951eb5e4: Waiting
4e797c290268: Waiting
09e3373b9d9a: Waiting
4fc242d58285: Waiting
e047119c9fda: Layer already exists
ded76dea467d: Layer already exists
ad8b951eb5e4: Layer already exists
4e797c290268: Layer already exists
09e3373b9d9a: Layer already exists
4fc242d58285: Layer already exists
c082bc6e7cd9: Pushed
7281277fa00b: Pushed
3145398e7473: Pushed
f5b2a518bf073fedface3b9287ec9c4f03269bb4: digest: sha256:d9b9f74e995629a7489dda1e749f3834ce8d1cadfbca19add2f3032dc1cf554d size: 2212
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

W naszym profilu powinien ukazać się spis obrazów



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.

DIAGRAM AKTYWNOŚCI



Omówienie pipeline:

a) Początek pipeline – kiedy wykryte są zmiany na zdalnym repozytorium git push uruchamia się Jenkins i wykonuje instrukcje zawarte w Jenkinsfile.
b) 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