Pipeline: Jenkinsfile, etapy, artefakty

Poziom: operacyjny **Dotyczy: Jenkins LTS, Pipeline plugin** **Czas: ~35 min**

Pipeline zastąpił w Jenkinsie ręczne klikanie kroków zadania w UI plikiem tekstowym trzymanym w repozytorium razem z kodem — dzięki temu definicja builda przechodzi ten sam code review co reszta projektu.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
LTS	Long Term Support: wersja z długim wsparciem poprawkami
XML	eXtensible Markup Language: format tekstowy do zapisu ustrukturyzowanych danych w znacznikach, tutaj używany przez PowerShell do zapisu zrzutu obiektów poleceniem Export-Clixml

Fiszka

TYLKO ODCZYT

Wstrzyknięcie poświadczenia do kroku pipeline'u: credentials()

```
environment {
    TOKEN_REJESTRU = credentials('id-poswiadczenia-rejestru')
}
```

Wstawiamy identyfikator poświadczenia zapisanego w magazynie Jenkinsa — sama wartość nigdy nie trafia do treści Jenkinsfile, tylko jest podstawiana w trakcie wykonania. Więcej o typach poświadczeń i maskowaniu ich w logu na stronie Poświadczenia i bezpieczeństwo.

□□ TYLKO ODCZYT

Archiwizacja artefaktów builda: archiveArtifacts

```
steps {
    archiveArtifacts artifacts: 'target/*.jar', fingerprint: true
}
```

Zapisujemy wskazane pliki jako artefakty builda, dostępne do pobrania z poziomu UI Jenkinsa i widoczne w historii nawet po wyczyszczeniu workspace. Opcja `fingerprint: true` dodatkowo pozwala Jenkinsowi śledzić, w których kolejnych buildach dany plik został ponownie użyty.

□□ TYLKO ODCZYT

Publikacja wyników testów: junit

```
post {
    always {
        junit 'build/reports/**/*.xml'
    }
}
```

Wczytujemy raporty testów w formacie JUnit XML i pokazujemy je w UI jako wykres trendu zaliczonych/niezaliczonych testów w czasie — działa niezależnie od tego, jakim narzędziem testy faktycznie zostały wygenerowane, o ile zapisuje wynik w tym formacie.

□□ **XML** = eXtensible Markup Language: format tekstowy do zapisu ustrukturyzowanych danych w znacznikach, tutaj używany przez PowerShell do zapisu zrzutu obiektów poleceniem Export-Clixml

□□ TYLKO ODCZYT

Przekazanie plików między etapami tego samego builda: stash / unstash

```
stage('Build') {
    steps {
        sh 'make build'
        stash name: 'binarki', includes: 'dist/**'
    }
```

```
}

stage('Deploy') {
    steps {
        unstash 'binarki'
        sh './deploy.sh'
    }
}
```

> **Pojęcie: `stash`.** Tymczasowe przechowanie zestawu plików między etapami tego samego builda, przydatne zwłaszcza gdy poszczególne etapy wykonują się na różnych agentach i nie dzielą wspólnego workspace. Zawartość stasha żyje tylko do końca builda — to nie jest trwały magazyn artefaktów, do tego służy `archiveArtifacts`.

sh = shell: podstawowy interpreter poleceń Linuksa, tu uruchamia pobrany skrypt instalacyjny

TYLKO ODCZYT

Zabezpieczenie zasobu przed równoległym użyciem: lock

```
stage('Wdrożenie') {
    steps {
        lock('srodowisko-testowe') {
            sh './deploy.sh'
        }
    }
}
```

Blokujemy nazwany zasób na czas wykonania kroku — jeśli dwa buildy próbują jednocześnie wdrożyć na to samo środowisko testowe, drugi czeka w kolejce, zamiast nadpisać wdrożenie pierwszego w trakcie jego trwania. Wymaga wtyczki Lockable Resources.

sh = shell: podstawowy interpreter poleceń Linuksa, tu uruchamia pobrany skrypt instalacyjny

TYLKO ODCZYT

Ograniczenie czasu wykonania etapu: timeout

```
stage('Test') {
    options {
        timeout(time: 15, unit: 'MINUTES')
    }
    steps {
        sh 'make test'
    }
}
```

Przerywamy etap, jeśli przekroczy podany limit czasu — zabezpieczenie przed zawieszonym procesem testowym blokującym agenta w nieskończoność.

timeout = uruchom polecenie i przerwij je, jeśli nie skończy się w zadanym czasie · **sh** = shell: podstawowy interpreter poleceń Linuksa, tu uruchamia pobrany skrypt instalacyjny

TYLKO ODCZYT

Automatyczne ponowienie kroku: retry

```
stage('Pobranie zależności') {
    steps {
        retry(3) {
            sh 'npm ci'
        }
    }
}
```

Powtarzamy krok do podanej liczby razy, zanim uznamy go za ostatecznie nieudany — użyteczne przy krokach zależnych od niestabilnej sieci (pobieranie paczek, obrazów), nie przy krokach z efektami ubocznymi, które powtórzenie mogłoby zdublować.

sh = shell: podstawowy interpreter poleceń Linuksa, tu uruchamia pobrany skrypt instalacyjny

Czym jest Jenkinsfile

Pojęcie: Jenkinsfile. Plik tekstowy, zwykle trzymany w korzeniu repozytorium pod nazwą `Jenkinsfile`, opisujący cały proces budowania — jego etapy, kroki w każdym z nich oraz to, co ma się wydarzyć po sukcesie albo porażce. Zadanie typu „Pipeline" w Jenkinsie nie

> trzyma definicji kroków u siebie, tylko wskazuje repozytorium i ścieżkę do tego pliku — dzięki temu zmiana procesu budowania to zwykły commit, widoczny w historii repozytorium tak samo jak zmiana kodu.

Jenkinsfile pisze się w jednym z dwóch dialektów.

Deklaratywny opisuje pipeline strukturą o ściśle zdefiniowanych blokach (`pipeline`, `stages`, `steps`) — mniej elastyczny, za to czytelny nawet dla kogoś, kto nie zna Groovy, i to on jest dziś domyślnym wyborem dla nowych projektów.

Skryptowy to pełny skrypt Groovy z dodatkowymi funkcjami udostępnianymi przez Jenkinsa — daje pełną moc języka programowania (pętle, warunki, własne funkcje bez ograniczeń składni deklaratywnej) kosztem czytelności i wyższego progu wejścia.

Struktura deklaratywnego pipeline'u

Szkielet minimalnego, ale kompletnego pipeline'u:

```
pipeline {
    agent { label 'linux-docker' }

    environment {
        REJESTR = 'git.zielona.example'
    }

    parameters {
        booleanParam(name: 'WDROZ_NA_PROD', defaultValue: false, description: 'Wdrożenie po
testach')
    }

    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }
```

```
        stage('Build') {
            steps {
                sh 'make build'
            }
        }

        stage('Test') {
            steps {
                sh 'make test'
            }
        }
    }

    post {
        always {
            junit 'build/reports/**/*.xml'
        }
        failure {
            mail to: 'zespol@zielona.example', subject: 'Build nieudany', body: "Zobacz
${env.BUILD_URL}"
        }
    }
}
```

Pojęcie: `agent`. Blok wskazujący, gdzie mają wykonać się kroki — etykieta konkretnego agenta (`label`), dowolny wolny agent (`any`), wykonanie w efemerycznym kontenerze (`docker { image 'NazwaObrazu' }`) albo świadome pominięcie przypisania na poziomie pipeline'u, gdy każdy `stage` deklaruje własnego agenta osobno.

Pojęcie: `stages` i `steps`. `stages` to sekwencja etapów widocznych w UI jako osobne bloki na osi czasu builda (Checkout, Build, Test…) — każdy `stage` zawiera blok `steps` z faktycznymi poleceniami do wykonania, najczęściej krokiem `sh` uruchamiającym komendę powłoki.

Pojęcie: `when`. Warunek decydujący, czy dany `stage` w ogóle się wykona — najczęściej sprawdza nazwę gałęzi albo wartość parametru:

```
stage('Wdrożenie') {
    when {
        branch 'main'
        expression { params.WDROZ_NA_PROD == true }
    }
    steps {
        sh './deploy.sh'
    }
}
```

> **Pojęcie: `environment`.** Blok deklarujący zmienne środowiskowe widoczne we wszystkich krokach danego zakresu (całego pipeline'u albo pojedynczego `stage`) — wygodne miejsce na stałe konfiguracyjne niebędące sekretami.

> **Pojęcie: `parameters`.** Deklaracja parametrów, o które Jenkins zapyta przed uruchomieniem builda ręcznie wyzwolonego — typy obejmują tekst, wybór z listy, wartość logiczną i poświadczenie.

> **Pojęcie: `post`.** Blok wykonywany po zakończeniu wszystkich etapów, niezależnie od ich wyniku — z warunkami `always` (zawsze), `success`, `failure`, `unstable`, pozwalający np. zawsze zebrać wyniki testów, a mail wysłać tylko przy porażce.

Częste błędy

Trzymanie sekretów w zmiennych `environment` zamiast w `credentials()`. Wartość wpisana wprost w bloku `environment` trafia do logu builda w postaci jawnej za każdym razem, gdy jest wypisywana — tylko `credentials()` i magazyn poświadczeń Jenkinsa zapewniają automatyczne maskowanie w logu.

Zbyt agresywny `retry` na kroku z efektem ubocznym. Ponowienie kroku, który już częściowo wykonał nieodwracalną operację (np. wysłał żądanie do zewnętrznego API tworzącego zasób), potrafi tę operację zduplikować zamiast naprawić przejściowy błąd sieci.

Brak `lock` przy współdzielonym środowisku testowym. Dwa równoległe buildy wdrażające na to samo środowisko bez blokady kończą się wdrożeniem złożonym z fragmentów obu commitów, trudnym do odtworzenia przy diagnozowaniu awarii.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Piszemy szkielet deklaratywnego pipeline'u z trzema etapami: Checkout, Build, Test, oraz blokiem `post` archiwizującym wyniki testów niezależnie od wyniku builda.
2. Wyjaśniamy różnicę między `stash`/`unstash` a `archiveArtifacts` — kiedy sięgamy po jedno, a kiedy po drugie.
3. Etap wdrożenia ma wykonywać się tylko na gałęzi `main` i tylko gdy parametr `WDROZ_NA_PROD` jest ustawiony na `true`. Zapisujemy odpowiedni blok `when`.
4. Wskazujemy, dlaczego wartość zwrócona przez `credentials()` nie powinna być nigdy wypisywana krokiem `echo`.

Powiązane: Architektura Jenkinsa: kontroler, agenci, workspace · Poświadczenia i bezpieczeństwo · Utrzymanie i diagnostyka

---

Revision #37
Created 2026-09-25 07:39:53 UTC by Tomasz Konieczny
Updated 2026-10-26 14:15:06 UTC by Tomasz Konieczny