

# DEVOPS

Best Practices per un prodotto migliore

Dario Pasquali

# AGENDA

## DAY 1 - POMERIGGIO

- 1 Continunuous Integration
- 2 Continuous Testing
- 3 Caso D'uso
- 4 Pratica



# 7 BEST PRACTICES

- 1 Continuous Management
- 2 Infrastructure As a Code
- 3 **CONTINUOUS INTEGRATION**
- 4 **CONTINUOUS TESTING**
- 5 Continuous Delivery
- 6 Continuous Deployment
- 7 Continuous Management



**CONTINUOUS LEARNING**



# CONTINUOUS INTEGRATION



**«IF SOMETHING HURTS, DO IT MORE  
OFTEN AND BRING THE PAIN  
FORWARD»**

*eXtreme Programming (XP)*



# CONTINUOUS INTEGRATION

## REQUISITI

3 Requisiti fondamentali per adottare il Continuous Integration:

1. Version Control System multibranch
2. Continuous Integration Server
3. Strutturare lo sviluppo in modo incrementale



# CONTINUOUS INTEGRATION

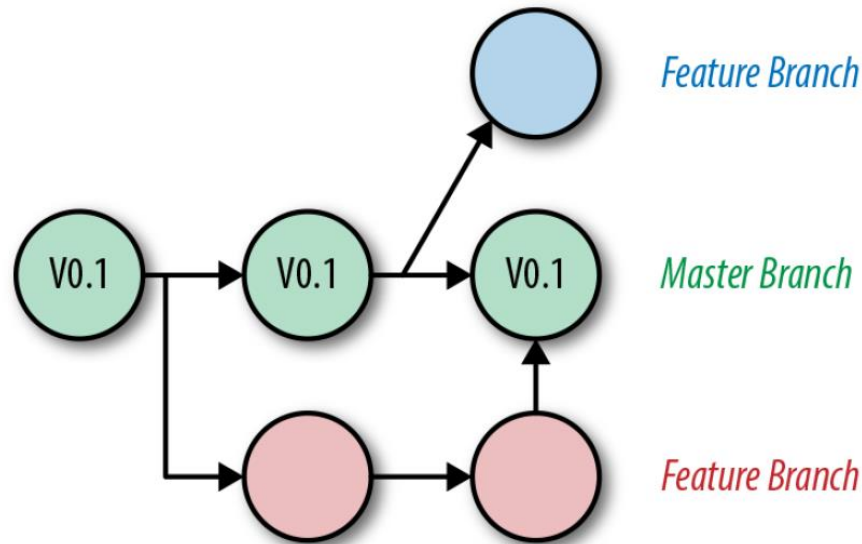
## VERSION CONTROL SYSTEM

### MASTER BRANCH

Ramo principale del progetto, mantiene la versione più aggiornata contenente le features di tutti i membri del team.

### FEATURE BRANCH

Ramo parallelo al master usato per sviluppare una specifica feature in maniera sicura.



# CONTINUOUS INTEGRATION

## VERSION CONTROL SYSTEM

- Suddividere il lavoro in Features standalone
- Strutturare un **VERSION CONTROL SYSTEM (VCS)** che mantenga la versione finale del progetto (Master Branch) e il flusso di produzione
- Creare Branches per ogni features
- **INTEGRARE** i Features Branch nel Master Branch **FREQUENTEMENTE** (almeno 1/gg) in modo che sia sempre pronto al Release.





# CONTINUOUS INTEGRATION

## GITHUB

Piattaforma web di supporto a GIT, permette di:

- Lavorare ai progetti in maniera condivisa all'interno del team, gestendo visivamente i commit
- Gestire le merge tra Branch tramite un sistema di **PULL REQUEST** per controllare al meglio chi e cosa viene integrato
- Strumenti di supporto come Issue tracking, Wiki, ...
- Nessun supporto al CI integrato
  - **MASSIMA LIBERTÀ DI PERSONALIZZAZIONE !**

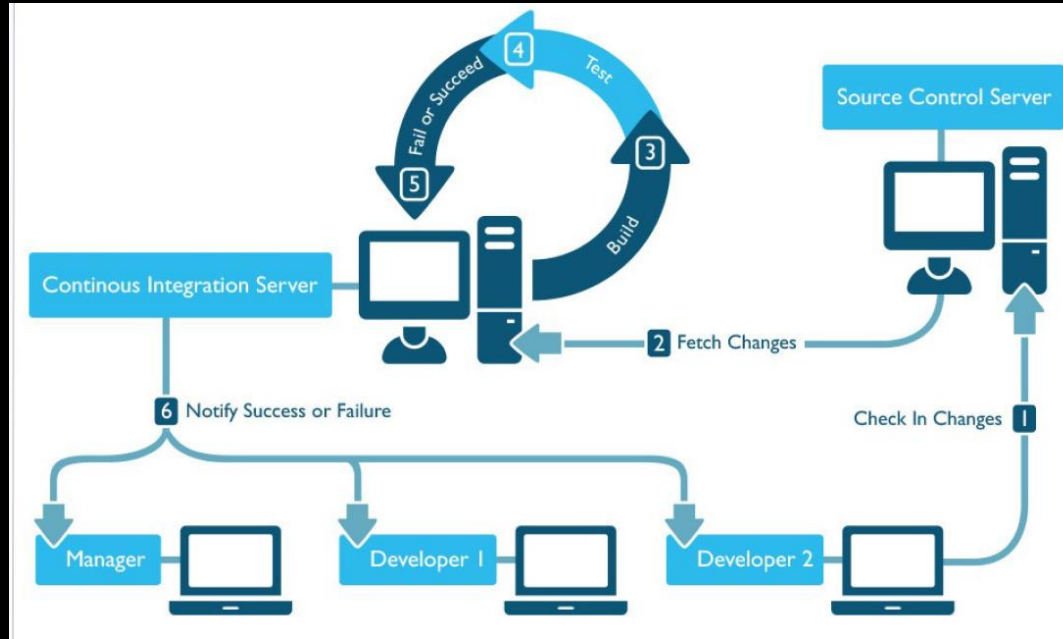


# CI PIPELINE

## INTEGRATION FLOW

Processo di Integrazione  
**AUTOMATIZZATO.**

Il commit sul VCS innesca una sequenza di step per verificare che il Branch siano integrabili in sicurezza.



# CI PIPELNE

## INTEGRATION FLOW

La pipeline esegue 3 passi fondamentali:

1. **BUILD**: fatta una e una sola volta in ambiente standardizzato (identico a quello di produzione)
2. **TEST**: Unit, Smoke, Integration, ... Assicurando la qualità del prodotto
3. **NOTIFY**: notifica successi e fallimenti **A TUTTO IL TEAM**



# CI SERVER

## INTEGRATION FLOW

Server che ospita e supporta l'esecuzione della pipeline di Continuous Integration.

Ampia scelta: servizi open source, enterprise, specifici per VSC, con supporto per lo scaling, container, plugin, ...

In questo periodo ho sperimentato GitLabCI, TravisCI, Bamboo, Go, TeamCity, CircleCI.

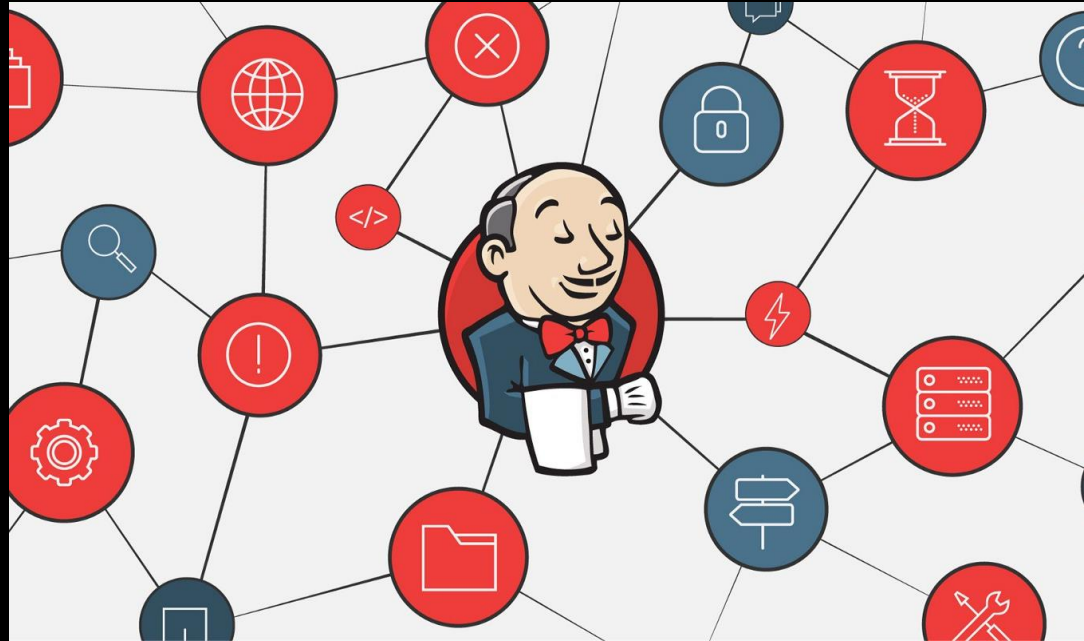
Scelta finale su **JENKINS**



# JENKINS

## CONTINUOUS INTEGRATION SERVER

- Sviluppato da CloudBees, disponibile in versione Enterprise e **OPEN SOURCE** (self – hosted)
- Totalmente customizzabile in base alle proprie esigenze grazie all'ecosistema di **PLUGINS**
- Integrabile con servizi esterni tramite **WEBHOOK**



# JENKINS

## CONTINUOUS INTEGRATION SERVER

Pipeline definita tramite il **JENKINSFILE**:

- Scritto in **GROOVY**, DSL simile al python
- Contenuto nel progetto da integrare
- Pipeline diverse per diversi Branch
- **CONDIVISIONE** trasparente del processo di sviluppo all'interno del team
- Possibilità di esecuzione in container Docker configurato «On the Fly»



# DECLARATIVE PIPELINE

## JENKINSFILE STYLE

Keyword «*pipeline*», flusso di esecuzione come sequenza di **STAGES**, definiti tramite il DSL Groovy

- Massima astrazione
- Facile comprensione da tutto il team
- Editor grafico di supporto (**BLUE OCEAN**)
- Necessità di Script per task complessi



# SCRIPTED PIPELINE

## JENKINSFILE STYLE

Keyword «*node*», estensione del DSL dichiarativo includendo costrutti tipici dei linguaggi imperativi.

- Massimo controllo del flusso di esecuzione
- Neccessaria conoscenza del linguaggio





# BLUE OCEAN

## DECLARATIVE MULTIBRANCH PIPELINE EDITOR

Restyle dell'interfaccia grafica di Jenkins specifico per le **PIPELINE MULTIBRANCH**

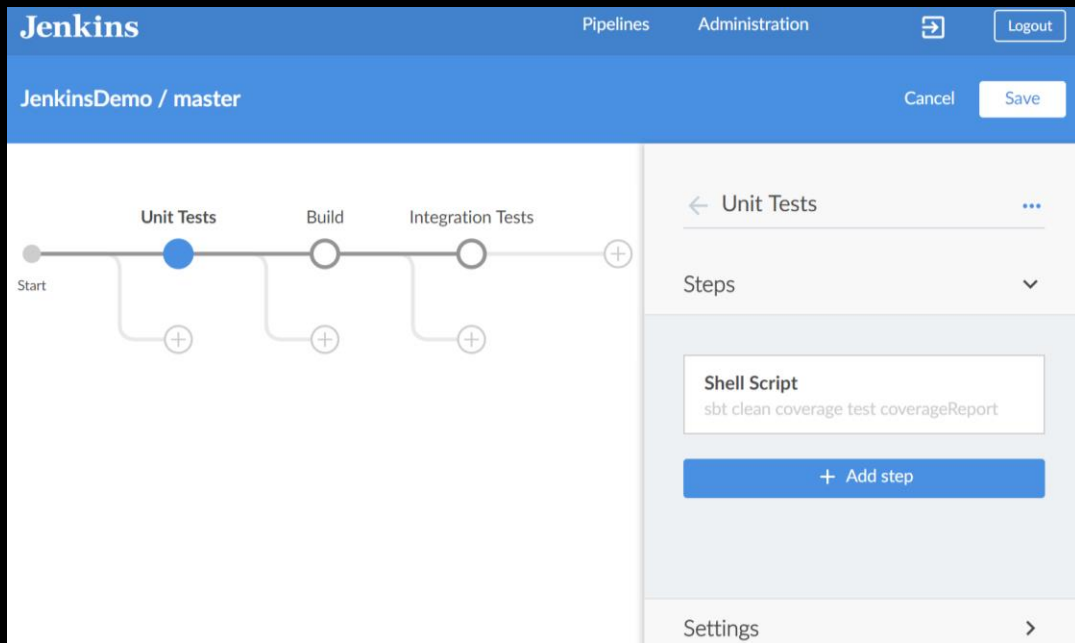
- Editor grafico per Jenkinsfile dichiarativi
- Integrazione automatica delle funzioni di plugin inseriti in jenkins
- Gestione delle Pipeline per i Singoli Branch
- Interfaccia di esecuzione con aggiornamento dei Log real time, divisi per Step/Stage
- Gestione degli artefatti archiviati



# BLUE OCEAN

## DECLARATIVE MULTIBRANCH PIPELINE EDITOR

```
pipeline {
  agent any
  environment { VAR = 'value' }
  stages {
    stage('Unit Tests') {
      steps {
        sh 'sbt clean coverage test coverageReport'
      }
    }
    stage('Build') {
      steps {
        sh 'sbt clean compile package assembly'
      }
    }
    stage('Integration Tests') {
      steps {
        sh 'cd IntegrationEnv/ && sbt clean test'
      }
    }
  }
  post {
    success { echo "Success" }
    failure { echo "Fail" }
  }
}
```



# CONTINUOUS INTEGRATION

## VANTAGGI

- Test automatici, in ambiente standard, eseguiti velocemente (< 10 min)
- Diminuzione del costo di Test
- **BUILD ONE-TIME** standard
- Meno bug/errori integrati nel Master (e portati in produzione)
- **SISTEMA** finale più stabile, **MODULARE** e mantenibile
- **FEEDBACK VISIVO** sullo stato della build (Trasparenza nel team)
- **MAGGIOR RESPONSABILITÀ** del singolo e diminuzione del debito tecnico
- Condizioni di lavoro migliori



# CONTINUOUS TESTING



# CONTINUOUS TESTING

## NON SOLO TEST

Test nella Pipeline di CI hanno un ruolo Fondamentale.

Sono l'**UNICA FONTE DI DIFESA** dall'integrazione di codice non funzionante o buggato.

Il Continuous Testing arricchisce l'Automated Testing con Tools e Cultura per garantire la **QUALITÀ** del software.



# CONTINUOUS TESTING

## NUOVA SEMANTICA DI TEST

Nell'ottica del Continuous Deployment i Test assumono nuova semantica:

- Sono la principale **GARANZIA DI QUALITÀ** del Software rilasciato

$$\text{COVERAGE} = \frac{\text{Righe di Codice Testato}}{\text{Righe di Codice Scritto}}$$

- Indicatore del **RISCHIO COMMERCIALE** legato al rilascio
- Indicatore di **PROGRESSIONE** dello sviluppo



# CONTINUOUS TESTING

## STRUTTURARE I TEST

Fondamentale strutturare saggiamente i Test:

- Avvalersi di un **FRAMEWORK DI TEST** (JUnit, ScalaTest, ...)
- Dividere i Test per **FEATURE** (penserà Git a unire al momento dell'integrazione)
- Adottare tecniche come la Test Driven Development (**TDD**)
- Tempo di esecuzione della batteria di test **< 10 MINUTI** (XP)



# CASO D'USO





# MOVIE ADVICER

## REQUISITI POC

- **RACCOMANDATORE BINARIO** basato sul Dataset Movielens 100k
- I dati di interesse devono essere memorizzati su un **DATA LAKE**, implementato in Kudu
- Si vuole presentare una semplice **INTERFACCIA** per interagire con il sistema



# MOVIE ADVICER

## SVILUPPO INCREMENTALE

Da Requisiti a Features Incrementali:

1. Il raccomandatore carica i dati da **CSV LOCALE**
2. Salvare i dati sul **DATA LAKE**
3. Raccomandatore che carica i dati dal Data Lake
4. **ENDPOINT WEB** per accedere al modello
5. **WEBAPP** per una migliore user experience



# MOVIE ADVICER

## SVILUPPO INCREMENTALE

Da Requisiti a Features Incrementali:

1. Il raccomandatore carica i dati da **CSV LOCALE**
  2. Salvare i dati sul **DATA LAKE**
  3. Raccomandatore che carica i dati dal Data Lake
  4. **ENDPOINT WEB** per accedere al modello
  5. **WEBAPP** per una migliore user experience
- 
- CORE**



# TDD – SCALATEST

## SVILUPPO CORE

- **BEFOREALL**  
init Spark e load csv
- **TEST**  
ad alta granularità per le varie features (pending per quelle non implementate)
- **AFTERALL**  
close safe di Spark

```
override def beforeAll(): Unit = {...}

"Spark Session" must "be initialized" in {...}

"Ratings Dataframe" must "be loaded from csv local file" in {...}
it must "contains at least 10K elements" in {...}
it must "be transformed, removing timestamp column" in {...}
it must "be transformed in a RDD of Spark MLlib Ratings" in {...}

"Rating RDD" must "be split in train and test set" in {...}

"Binary ALS recommender" must "be initialized" in {...}
it must "train a recommendation model" in {...}

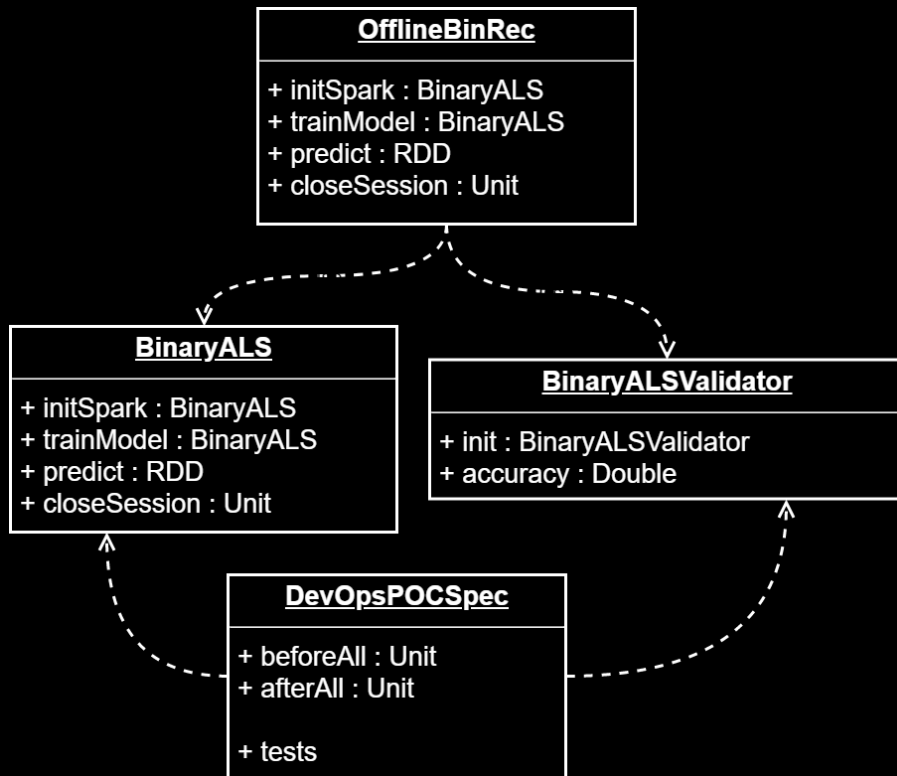
"The model" must "have a good accuracy" in {...}
it must "have a good precision" in pending
it must "have a good recall" in pending

override def afterAll(): Unit = {...}
```



# CLASSI

## SVILUPPO CORE



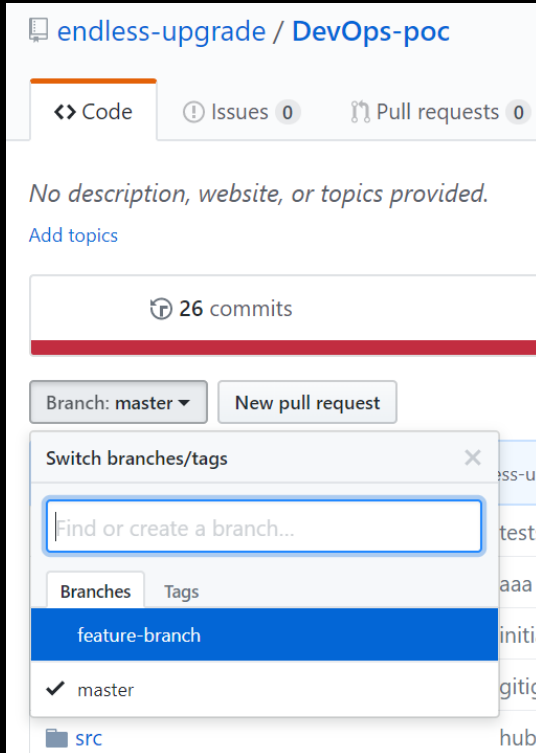
Architettura semplice.

Aggiunta di classi in modo modulare al progredire dello sviluppo.



# GITHUB

## STRUTTURA BRANCHES



Di base, predisposti due Branches:

- master  
contiene la versione più aggiornata
- feature-branch  
per lo sviluppo incrementale

(visti automaticamente da Jenkins – Blue Ocean



# JENKINSFILE

## PIPELINE DI CI

```
pipeline {
  agent any
  stages {
    stage('Config System') {
      steps {
        echo "${env.BRANCH_NAME}"
        echo 'Put here provisioning stuff'
        echo 'USE Ansible :)'
      }
    }
    stage('Test the System') {
      steps {
        echo 'SBT manage dependencies, just test if is reachable'
        sh 'java -version'
        sh 'sbt about'
      }
    }
    stage('Unit Tests') {
      steps {
        echo 'Tests'
        sh 'sbt clean test'
        archiveArtifacts 'target/test-reports/*.xml'
        junit(testResults: 'target/test-reports/DevOpsPOCSpec.xml',
              allowEmptyResults: true)
      }
    }
  }
}
```

### CONFIG SYSTEM

Eventuale provisioning ulteriore  
(Ansible ??)

### TEST THE SYSTEM

Verifica che java e SBT siano  
eseguibili (e dowload della versione  
SBT richiesta in build.sbt)

### UNIT TESTS

Esegue la batteria di test, archiviando  
i risultati.



# JENKINSFILE

## PIPELINE CI

```
stage('Build') {
  steps {
    echo 'Build'
    sh 'sbt clean compile package assembly'
    archiveArtifacts 'target/scala-*//*.jar'
  }
}

stage('Notify') {
  steps {
    script {
      if (env.BRANCH_NAME != "master") {
        sh "git checkout ${env.BRANCH_NAME}"
        sh 'source /etc/profile.d/exports.sh && +
          ' /opt/hub-linux-386-2.3.0-pre10/bin/hub pull-request' +
          ' -m "$(git log -1 --pretty=%B)"'
        notifyMessage = "Pull Request Sent"
      }
      else {
        notifyMessage = "Master ready for production"
      }
    }
  }
}
```

### BUILD

FAT jar con SBT assembly e archiviazione

### NOTIFY

In caso di branch, creazione di una pull request su GitHub.

Notifica verso Slack fatta in ogni caso, messaggio diverso in base al branch.





# JENKINSFILE

## PIPELINE

```
post {
    success {
        script {
            header = "Job <${env.JOB_URL}|${env.JOB_NAME}> <${env.JOB_DISPLAY_URL}|(Blue)>"
            header += " build <${env.BUILD_URL}|${env.BUILD_DISPLAY_NAME}> <${env.RUN_DISPLAY_URL}|(Blue)>:"
            message = "${header}\n :smiley: All test passed :smiley: $notifyMessage"

            author = sh(script: "git log -1 --pretty=%an", returnStdout: true).trim()
            commitMessage = sh(script: "git log -1 --pretty=%B", returnStdout: true).trim()
            message += " Commit by <@${author}> (${author}): `` ` ${commitMessage} `` ` "
            color = '#00CC00'
            slackSend(message: message,
                baseUrl: 'https://devops-pasquali-cm.slack.com/services/hooks/jenkins-ci/',
                color: color, token: 'ihoCVUPB7hqGz2xI1htD8x0F')
        }
    }
    failure {
        script {
            header = "Job <${env.JOB_URL}|${env.JOB_NAME}> <${env.JOB_DISPLAY_URL}|(Blue)>"
            header += " build <${env.BUILD_URL}|${env.BUILD_DISPLAY_NAME}> <${env.RUN_DISPLAY_URL}|(Blue)>:"
            message = "${header}\n :sob: The Build Failed, Release not ready for production! :sob: : `` ` ${failMessage} `` `\n"

            author = sh(script: "git log -1 --pretty=%an", returnStdout: true).trim()
            commitMessage = sh(script: "git log -1 --pretty=%B", returnStdout: true).trim()
            message += " Commit by <@${author}> (${author}): `` ` ${commitMessage} `` ` "
            color = '#990000'
            slackSend(message: message,
                baseUrl: 'https://devops-pasquali-cm.slack.com/services/hooks/jenkins-ci/',
                color: color, token: 'ihoCVUPB7hqGz2xI1htD8x0F')
        }
    }
}
```



#general | DevOps | endless-upgrade | jenkins / DevOps | DevOps-poc | Untitled Diagram | jenkins - No Build |

35.231.37.71:8080/blue/organizations/jenkins/DevOps-poc/detail/feature-branch/5/pipeline/

App | Install Spark on Windows | Get Started with PySpark | Shrew Soft Inc : Software | Reply Work | Meister | Apache Sqoop - Overview | Design Codes: HDInsights | How to Setup your Firewall | How to use SparkSession |

DevOps-poc 5

Pipeline | Changes | Tests | Artifacts | | | | | Logout |

Branch: feature-branch 4m 24s Changes by d.pasquali

Commit: 2c06141 - Replayed #4

Start

Config System

Test the System

Unit Tests

Build

Notify

End

Build - 1m 55s

> Build — Print Message <1s

> sbt clean compile package assembly — Shell Script 1m 55s

## ✓ DevOps-poc 5

[Pipeline](#)[Changes](#)[Tests](#)[Artifacts](#)[Logout](#)

Branch: feature-branch

5m 58s

Changes by d.pasquali

Commit: 2c06141

9 minutes ago

Replayed #4



Notify - 2s



&gt; Shell Script

1s



&gt; source /etc/profile.d/exports.sh &amp;&amp; /opt/hub-linux-386-2.3.0-pre10/bin/hub pull-request -m "\$(git log -1 --pretty=%B)" -- Shell Script

1s



&gt; Send Slack Message

&lt;1s

# SETUP



# SETUP

## JENKINS SERVER

- 1 Accedi all'istanza di questa mattina
- 2 Download repository git (<https://github.com/endless-upgrade/continuous-integration-and-testing.git>)
- 3 Esegui il playbook Ansible
- 4 Apri la porta 8080 tramite GCP (firewall settings)
- 5 User = admin, password = admin



# SETUP

## CONNESSIONE A GITHUB

- 1 Apri il tuo account GitHub
- 2 Account (icona in alto a destra) -> Settings -> Developer Settings -> Personal Access Token -> Genera Token (con un nome significativo)
- 3 Copia il **TOKEN**
- 4 Nell'istanza, inserisci il token in `/etc/profile.d/exports.sh`

```
export GITHUB_TOKEN=token  
source /etc/profile.d/exports.sh
```



# SETUP

## CONNESSIONE A GITHUB 2

- 1 Apri il tuo account GitHub
- 2 Account (icona in alto a destra) -> Settings -> Developer Settings -> Personal Access Token -> Genera Token (con un nome significativo)
- 3 Copia il **TOKEN**
- 4 Nell'istanza, inserisci il token in `/etc/profile.d/exports.sh`

```
export GITHUB_TOKEN=token  
source /etc/profile.d/exports.sh
```



# SETUP

## CONNESSIONE A GITHUB 3

- 1 Accedi alla console di Blue Ocean
- 2 Seleziona GitHub e inserisci il Token (quello di prima)
- 3 Seleziona l'account / organizzazione
- 4 E il progetto da integrare
- 5 Se già presente un Jenkins file, questo verrà automaticamente caricato
- 6 Altrimenti si apre l'editor.





# SETUP

## TRIGGER PIPELINE

A default la pipeline vede tutti i branch, ma non viene eseguita automaticamene.

1. Accedi a Jenkins tradizionale
2. Freccina a destra della pipeline -> Configure
3. Elimina i «Discover pull requests...» in Branch Sources
4. Seleziona «Periodically if not otherwise» e scegli il timeout di polling



# SETUP

## CONNESSIONE A SLACK

- 1 Accedi Slack
- 2 Aggiungi il Bot Jenkins
- 3 Nelle impostazioni del Bot copia **TOKEN** e **BASEURL**
- 4 Accedi a Jenkins (tradizionale)
- 5 Gestisci Jenkins -> Configura Sistema
- 6 Copia token e baseurl dove indicato



# PRATICA



# PRATICA

## SEMPLICE PROGETTO

- 1 Clona il PoC (<https://github.com/endless-upgrade/DevOps-poc.git>) e portalo sul tuo GitHub
- 2 Analizza test e pipeline
- 3 Esegui il SetUp del server e prova la pipeline
- 4 Pensa ad altre Features interessanti, magari usando anche gli altri csv del dataset Movielens
- 5 Sviluppa incrementalmente le Features, **MI RACCOMANDO PRIMA I TEST!!**



# PRATICA

## LINK UTILI

- Materiale GitHub: <https://github.com/endless-upgrade>
- Doc Jenkinsfile: <https://jenkins.io/doc/book/pipeline/jenkinsfile/>
- Doc Blue Ocean: <https://jenkins.io/doc/book/blueocean/>



