CERTIFIED JENKINS ENGINEER

Youssef ElZarka

KodeKloud Courses

Siddharth Barahalikar

🔒 Protected Document

This PDF is shared in a read-only format to protect the original work. If you'd like to access the unlocked version or collaborate on similar materials, feel free to reach out on LinkedIn

Important Notes:

I. You can consider that this summary includes (3) main courses for Jenkins:
- Jenkins for beginners .. *chapters (1 : 5)*
- Jenkins Pipeline project .. *chapters (6:11)*
- Advanced Jenkins .. *chapters (12:17)*

Chapter 1 : Introduction

Introduction 8
Problem Statement 8
Intro To Jenkins 10
Basics of SCM 12
SCM Technology 14
Basics of CI/CD 16
Software Testing 19

chapter 2 : Jenkins setup and interface

Jenkins Architecture 21
Jenkins Installation Option 23
Demo-Initial Setting & JVM Tunning 26
Running Junkins WAR as a standalone App 29
Jenkins User Interface Overview 30
System Settings 33
Build Timeout 36
Timestamp 39
Jenkins Project 41
Working with Freestyle Job 45
Chained Freestyle Project 50

Chapter 3 : Extending Jenkins and administration

Jenkins Plugins 51
Installing Plugins 53
Managing Plugins 56
Managing Old Data 59
Reload Configuration from Disk 60
Controller Failure – Freestyle Project 60
Jenkins Fingerprints 62

Chapter 4 : Jenkins Pipelines

Pipeline and Jenkinsfile 65
Additional pipeline Configuration 69
Simple Pipeline Job 74
Build & Test Via Pipeline 77
Pipeline Script from SCM 81
Controller Failure – Pipeline Project 84
Create Pipeline using Blue Ocean Graphical Editor 87
Create Pipeline with Parameters 91

Chapter 5 : Automation and Security

Automating jenkins using cli and APIs 94
Jenkins CLI Buila a job 97
Jenkins REST API-install a plugin 101
Jenkins CSRF - CRUMB 107
Jenkins Security Overview 110
Jenkins Authentication 112
Authentication-Matrix Authorization Strategy 114
Jenkins CLI-SSH Authentication 118
Jenkins CLI-Edit a Freestyle Job 121
Built-in Email Notification 124
Email Extention Notification-Freestyle 127
Email Extention Notification-Pipeline 130

Chapter 6 : Setting up CI Pipeline

NodeJS App Overview 134
Run and Test NodeJS App on Local Machine 136
Understanding DevOps Pipeline 139
Install&Setup NodeJS Build Tool 144
Organization Folder Project in Jenkins 146
Add Jenkinsfile to Solar System Repo 150
Install dependencies 153
Setup and Run Dependency Scanning 155
Fixing Vulnerabilities & Publish HTML Report 158
unit Testing and Analyze JUnit Reports 162
Using Option Directive 166

Chapter 7 : Code Quality and Testing

Code Coverage and Catch Errors 169
Refactoring Jenkinsfile 171
String Interpolation 174
SonarQube Intro/SAST 177
SAST Analysis 179
SonarQube Quality Gate Step and Refactoring 183

Chapter 8 : Containerization and Deployment

Build Docker Imange 187
Vulnerability Scan using Trivy 190
Push To Registry 193
Understanding Deployment Approach 195
Exploring AWS and Setting up Jenkins Instance 197
Build-in When Condition 201
Deploy AWS EC2 203
Integration Testing- AWS EC2 206

Chapter 9 : Kubernetes and GitOps

K8S Overview 209
Introduction To GitOps 211
Jenkins vs. ArgoCD 215
ArgoCD Concepts & Terminology 215
CICD With GitOps 217
Manifest Repository & Configure ArgoCD 220
K8s Deploy-Update Image Tag 223
K8S Deploy-Raise PR 226
Integration between Jenkins & ArgoCD 229
DAST & Manual Input 231
DAST Ignore Rules 235
Publish Reports to AWS S3 238
Deploy to Prod 242

Chapter 10 : AWS Lambda and advanced Deployment Techniques

AWS Lambda Basics 245
Using GenAI to Generate Steps for Lambda Deployment 247
Manual Lambda Deployment 251
Example on our App Solar System 254
Lambda Deployment Using Jenkinsfile 257
Update Lambda Configuration 260
Lambda-Invoke Function 263

Chapter 11 : Jenkins Administration and Monitoring (part1)

Jenkins Folder (Part1) 265
Jenkins Folder (Part2) 268
Jenkins Folder (Part3) 270
Jenkins Folder (Part4) 272
Script Console 274
Global Security Settings 277
Markup Formatters In More Details 280
Acess Control for Builds 283
Job Restrictions 286
Job Configuration History Plugin 289
Build Monitor View 291
Jenkins Scaling Capacity Planning 294
Monitoring using Java Melody 297
Slack Notification Setup 300
Send Slack Notification 302

Chapter 12 : Shared Libraries in Jenkins

Introduction to Shared Library 304
Refactor Existing Jenkinsfile 308
Create a Shared Library for Slack Notification 312
Configure Shared Library in Jenkins 314
Loading the Shared Library in Pipeline 317
Create Shared Library for Trivy Scan 320
Load Trivy Scan Library in Jenkins Pipeline 323
what&why&create Library Resource 326
Load Trivy Scan Script Library in Jenkins Pipeline 330

Chapter 13 : Agent and Nodes In Jenkins

Types of Agents 333
Create and Configure Node 337
Utilize Agents in Jobs 341
Utilize Docker Image Agent 344
Utilize Dockerfile Agent 347
Utilize NewContainerPerStage 350
NewContainerPerStage In More Details 352
Configure Cloud Instance-K8S 356
Utilize K8S Pod As Agent 360

Chapter 14 : Pipeline Enhancement and Caching

Sharing File Between Containers 364
Refactoring Solar System Pipeline 366
Refactoring Unit test Stage 369
Sequential Stages 371
Stash and Unstash 374
Pipeline Caching 376
Invalidate Cache 379

Chapter 15 : Pipeline Structure and Scripted vs. Declarative

Declarative vs Scripted Pipeline 382
Demo Declarative vs Scripted Pipeline 385
Post Section in Declarative Pipelines vs. Finally Block in Scripted Pipelines 388
reversing Stash In Declarative Pipeline 390
Scripted Pipeline Initalize 393
Scripted Pipeline-Static Agent 396
Scripted Pipeline-K8S Agent 398
Best Practice for Scripted Pipelines 402

Chapter 16 : Jenkins Administration and monitoring (part2)

Jenkins Supervision 404
Monitoring with Prometheus & Grafana 407
Log Recorder 409
Install & Configure Audit Trail 413
Forward Audit Logs to External Server like Elastic Cloud 415
Groovy Sandbox and In-Process Script Approval 417
Demo Groovy Sandbox and In-Process Script Approval (Part1) 420

Demo Groovy Sandbox and In-Process Script Approval (Part2) 423
Demo Groovy Sandbox and In-Process Script Approval (Part3)..... 425
Migrate Jenkins to Another Node 427

Chapter 17 : Backup and Configuration Management

GitHub Action Basics..... 430
Migrating Jenkins Pipeline to GitHub Action 434
Backing-Up & Restoring Jenkins 439
Demo Backing-Up & Restoring Jenkins (Thin Backup Plugin) 442
Validating Backup with Only Linux..... 445
Upgrading Jenkins 448
Jenkins Configure as code (JCasC) 451
Demo Configure and Explore (JCasC) 455
Real Example on JCasC (Creating a Job) & how to write It's YAML 458
Pipeline Durability..... 461

Important Notes:

II. In this course, it is highly recommended that you review the documentation for any of the following:
   - Jenkins plugins
   - Code used within stages and steps (Jenkinsfile)
   - Third-party tools (Trivy, SonarQube, ArgoCD....)

Introduction

📝 المقدمة

الجزء ده عبارة عن تقديم لكورس Certified Jenkins Engineer الهدف منه إنك تفهم خطة الكورس كاملة : إيه اللي هتدرسه من البداية للنهاية، وليه Jenkins يعتبر أداة أساسية في عالم الـ CI/CD

📖 الشرح بالتفصيل

- Jenkins هو أكتر سيرفر CI/CD مفتوح المصدر مستخدم في العالم (أكتر من مليون تنصيب شغال فعليًا).
- شركات كبيرة زي AWS ، IBM، GitHub بتدعمه رسميًا وبالتالي ده بيأكد استقراره واعتماديته.
- الكورس ماشي من الأساسيات .. لحد استراتيجيات متقدمة في التشغيل والإدارة.

محتوى الكورس بالترتيب:

1. مقدمة Jenkins : تبدأ تتعرف على Jenkins من أول مبادئه.
2. التحكم في الإصدارات (Source Control) + CI/CD Basics : الأساس اللي بيبنى عليه Jenkins
3. معمارية Jenkins + التنصيب + الواجهة : تفهم إزاي Jenkins مبني وإزاي تشتغل عليه.
4. الإضافات (Plugins) : إزاي توسع قدرات Jenkins عشان تناسب مشروعك.
5. Pipelines :
   - الفرق بين Declarative و Scripted Pipelines
   - إعدادات متقدمة للـ pipelines
   - استخدام CLI و REST API لإدارة Jenkins بشكل كامل.
6. CI Pipelines لتطبيقات Node.js + تحليل جودة الكود بـ SonarQube
7. Containerization & Deployment : باستخدام Docker + AWS
8. Kubernetes + GitOps : هتتعلم ArgoCD علشان Continuous Deployment
9. مراقبة Jenkins : بالأدوات Prometheus و Grafana
10. النسخ الاحتياطي + إدارة الإعدادات : باستخدام GitHub Actions + Jenkins Configuration as Code

---

🔗 مفاهيم إضافية لازم تعرفها

- Declarative vs Scripted Pipelines :
  - Declarative أبسط وأسهل كتابة ، معمولة علشان standard pipelines
  - Scripted أقوى ومرنة جدًا لكنها أعقد.
- SonarQube : أداة بتحلل جودة الكود وتكشف المشاكل الأمنية أو الـ bugs
- ArgoCD : أداة GitOps بتعمل Continuous Deployment للتطبيقات في Kubernetes
- Configuration as Code (JCasC) : بدل ما تعمل إعدادات Jenkins يدويًا، بتكتبها ككود في YAML

💡 مثال واقعي

تخيل إنك مطور في شركة:

- بتشتغل على تطبيق Node.js
- Jenkins Pipeline بتبني التطبيق، تشغل اختبارات، تعمل تحليل جودة الكود بـ SonarQube
- بعدين ينزل التطبيق كـ Docker Container على AWS
- ArgoCD في Kubernetes بيتابع الـ repo وينشر التغييرات أوتوماتيك.
- كل ده متراقب بـ Prometheus + Grafana
- ولو السيرفر وقع ؟ تقدر تسترجع Jenkins بالإعدادات من JCasC + GitHub Actions

Problem Statement

المقدمة

- ❖ التوبيك ده بيشرح بيمثل سيناريو عملي لشركة اسمها *Dasher Technology* ، وازاي بدأت رحلتها مع DevOps و CI/CD (دا مجرد مثال عشان تفهم المشاكل ظهرت ازاي والفروقات بين التولز وهكذا)
- ❖ في البداية كانوا بيعانوا من مشاكل في الكود ، الاختبارات، والـ deployments ، فقرروا يبنوا Pipeline حديثة باستخدام أدوات زي Jenkins و GitHub Actions

---

الشرح بالتفصيل

- Dasher Technology شركة سوفتوير بتقدم منصة لربط الـ data والـ apps والـ devices في بيئات On-Premise
- R&D Team قررت تنقل للخدمات السحابية وتستفيد من الـ containers (Docker + Kubernetes)
- المشروع الأول اللي طبقوا عليه كان Node.js Project، وبعد كده خطتهم يوسعوا التجربة لمشاريع Java و Python
- عيّنوا DevOps Team بقيادة Alice علشان تبني الـ pipeline من البداية.

❑ المشاكل اللي كانت عندهم قبل DevOps :

1. مفيش Version Control بمعني ان كل واحد بيكتب كود لنفسه.
2. Testing يدوي وبطيء وبالتالي Bugs كتير.
3. تعاون ضعيف هايؤدي الي Integrations قليلة ومشاكل كتير.
4. Deployments Manual → Development / Staging / Production
5. Releases كلها High Risk

❑ الحل اللي Alice خططته :

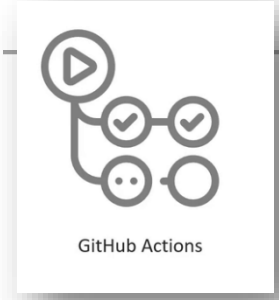
- استخدام GitHub للـ Version Control والتعاون بين المطورين.
- كتابة Unit Tests + Code Coverage
- Docker Build & Push علشان Containerization
- Deploy على Kubernetes
- إضافة Automated Integration Tests

❑ اختيار أداة CI/CD :

- قارنوا بين Jenkins, TravisCI, Bamboo, Spinnaker, CircleCI ...
- في البداية اختاروا Jenkins لأنه الأشهر والأوسع انتشارًا.
- بس Jenkins عيبه إنه محتاج VM + Java JDK + Firewall Rules + Plugins + Node.js + npm + Docker + Helm + kubectl + Tools تانية كتير.
- ده زوّد الـ Manual Setup & Configuration Workload

❑ التفكير الجديد :

- Alice بدأت تدور على أداة :
  1. سهلة الإعداد ، مفيش Installation & Config كتير.
  2. تركّز على Building Pipelines بدل ما يضيعوا وقت في إدارة Infrastructure
- النتيجة : قرروا يجربوا GitHub Actions



🔗 مفاهيم إضافية

- **CI/CD Pipeline** : سلسلة خطوات أوتوماتيكية لبناء الكود، اختباره، ونشره.
- **GitHub Actions** : أداة CI/CD مبنية جوه GitHub نفسها، بتوفر وقت كبير لأنك مش محتاج تجهّز Infrastructure
- **Docker vs Kubernetes** :
  - Docker = بيعمل Containerization للتطبيق.
  - Kubernetes = بيعمل Orchestration لإدارة الـ containers في بيئة Production
- **DevSecOps Tools** : زي Trivy و KubeSec علشان تعمل Security Scans في نفس الـ pipeline

---

💡 مثال واقعي

تخيل إنك شغال مع Dasher Technology :

- كودك معمول بـ Node.js
- GitHub Actions workflow بيتشغل أوتوماتيك أول ما تعمل Push
- بيعمل:
  1. Checkout الكود.
  2. Run Unit Tests
  3. Build Docker Image : Push لـ DockerHub
  4. Deploy للـ K8S كلاستر
  5. Run Integration Tests
- كل حاجة Automated من غير ما تلمس أي سيرفر أو تعمل Config يدوياً.

📌 الملخص

- Dasher Technology بدأت من بيئة Manual فيها مشاكل كتير.
- قرروا يعملوا Modern CI/CD Pipeline
- فكروا في Jenkins لكنه محتاج Setup & Maintenance كتير.
- لقوا إن GitHub Actions أسهل وأكتر مرونة.
- الكورس ده هيورّيك إزاي تبني Workflows حقيقية باستخدام GitHub Actions مع مشروع Node.js

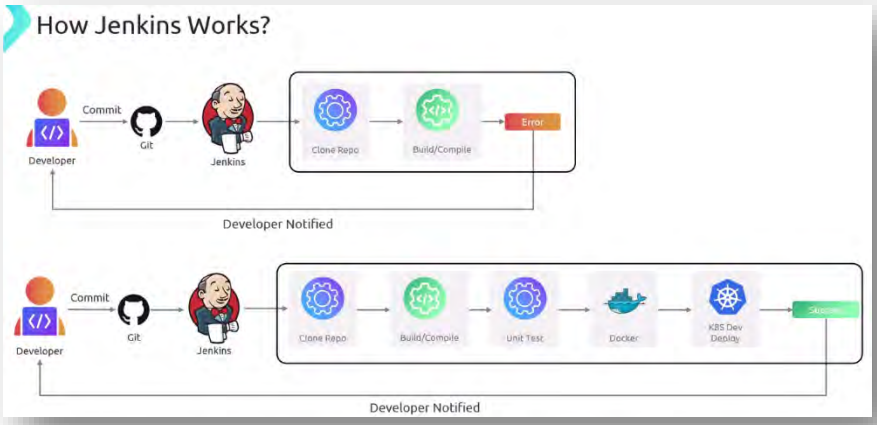
Intro To Jenkins

المقدمة

الجزء ده بيدي تعريف عملي عن Jenkins ، إزاي بيشتغل، وأهم المفاهيم الأساسية اللي لازم تبقى فاهمها قبل ما تدخل في بناء Pipelines .. بعد كده بيوضح مميزات Jenkins وعيوبه علشان تبقى عندك الصورة كاملة.

الشرح بالتفصيل

- Jenkins بيشتغل كـ Automation Server مفتوح المصدر و الهدف إنه يعمل Orchestrates لكل مراحل تطوير السوفتوير اللي هي (Build → Test → Deploy)
- يعني كل مرة تعمل Push للكود في Git ، Jenkins بيراقب الريبو:
  - يسحب التحديث الجديد.
  - يعمل Build/Compile
  - لو فيه Error : يبعث Alert للمطور.
  - لو الـ Build نجح : يكمل باقي المراحل (Testing, Deployment)
  - في الآخر يقدر يعمل Deploy على Kubernetes Cluster أو AWS EC2 مثلاً.
  - يبعث Notification عن النجاح أو الفشل.
- العملية دي Loop مستمر : أي تعديل بيتبعت بيتعمله Integration و Deployment بشكل أوتوماتيكي.



❑ المفاهيم الأساسية في Jenkins :

1. **Jobs** : هي الوحدات الأساسية ، كل Job عبارة عن Task زي Build أو Test أو Deploy
   - كل مرة Job يتنفذ اسمها Build
   - Jenkins بيحتفظ بـ Build History عشان تقدر تراجع النجاح والفشل.
2. **Freestyle Projects** : النوع الافتراضي من المشاريع في Jenkins واللي بتقدر من خلالها تحدد خطوات Build مرنة.
3. **Pipelines** : عبارة عن Workflows أوتوماتيكية بتربط Jobs مع بعض من أول Build لحد Deployment
   - بتتكتب في Jenkinsfile (يعني Pipeline as Code)
   - بتتخزن مع الكود في الـ Collaboration ← Git أفضل.
4. **Stages** : تقسيم الـ Pipeline لمراحل (Build / Test / Deploy)
   - بتدي Visualization أوضح.
5. **Nodes** : الماكينات اللي Jenkins بيشغل عليها الـ Jobs
   - Controller Node : اللي بينسق كل حاجة.
   - Worker Nodes : اللي بتنفذ المهام فعلياً.
   - تقدر تعمل Distributed Builds لزيادة الأداء.
6. **Plugins** : إضافات توسع قدرات Jenkins (مثلاً : Git, Docker, Kubernetes, Testing Frameworks)
   - ولاحظ إن أي Integration جديد تقريباً له Plugin

❑ **مميزات Jenkins :**

- مجاني و Open Source : مفيش License Fees
- Community كبيرة وداعمة.
- مكتبة Plugins ضخمة تغطي أي Use Case
- ممكن تكتب Groovy Scripts لحاجات معقدة.
- يدعم Pipeline as Code → Version Control + Collaboration
- Features كتير زي (Build Execution, Parallel Testing, Artifact Management, Reports)
- Scalable : تقدر تزود Worker Nodes لو عندك مشاريع كبيرة.

❑ **عيوب Jenkins :**

- صعب شوية على المبتدئين مقارنة بأدوات حديثة.
- الـ Configuration مع Plugins ممكن يبقى معقد ويحتاج وقت.
- لازم تحافظ على Updates دورية للـ Core والـ Plugins
- Performance Issues لو شغال على Server واحد بس.
- لازم تعمل Hosting و Configuration بنفسك (مش زي GitHub Actions/GitLab اللي بيجوا جاهزين)
- Security مسؤوليتك بالكامل : لازم تعمل Best Practices وتختار Plugins بعناية.

🔗 **مفاهيم إضافية**

- Pipeline as Code : اتجاه حديث بيخلي الـ pipeline جزء من الكود نفسه ودا أسهل في المراجعة (Code Review)
- Artifacts : الملفات الناتجة من الـ Build (زي JARs أو Docker Images) اللي Jenkins بيقدر يديرها.
- Distributed Builds : أسلوب لتوزيع الـ Jobs على أكتر من Node علشان السرعة.
- Notifications : Jenkins بيتكامل مع Slack, Email, Teams لإرسال تقارير.

💡 **مثال واقعي**

تخيل عندك فريق بيشتغل على Web App :

1. Developer يعمل Push للكود فهوبا Jenkins يلقط التغيير.
2. Jenkins يعمل Build : يعني يتأكد إن الكود بيتـ Compile صح.
3. Jenkins يعمل Unit Tests : يتأكد إن الكود شغال.
4. Jenkins يبني Docker Image : و يبعته على DockerHub
5. Jenkins يعمل Deploy على Kubernetes Cluster
6. Jenkins يبعت Report على Slack للفريق كله.

وبكده أي تعديل جديد بينزل Production في دقائق بدل ما كان بياخد أيام.

📌 **الملخص**

- Jenkins = Automation Server بيشغل CI/CD Pipelines
- بيحول الـ Software Delivery من Manual الي Automated
- عنده مكونات أساسية Jobs, Pipelines, Stages, Nodes, Plugins
- مميزاته : مجاني، قوي، Plugins كتير، Scalable
- عيوبه : Configuration معقد، محتاج Maintenance ، Security مسؤولية الفريق.
- هو الخيار الأشهر تاريخياً، لكن في المقابل أدوات زي GitHub Actions بتكون أسهل في البداية.

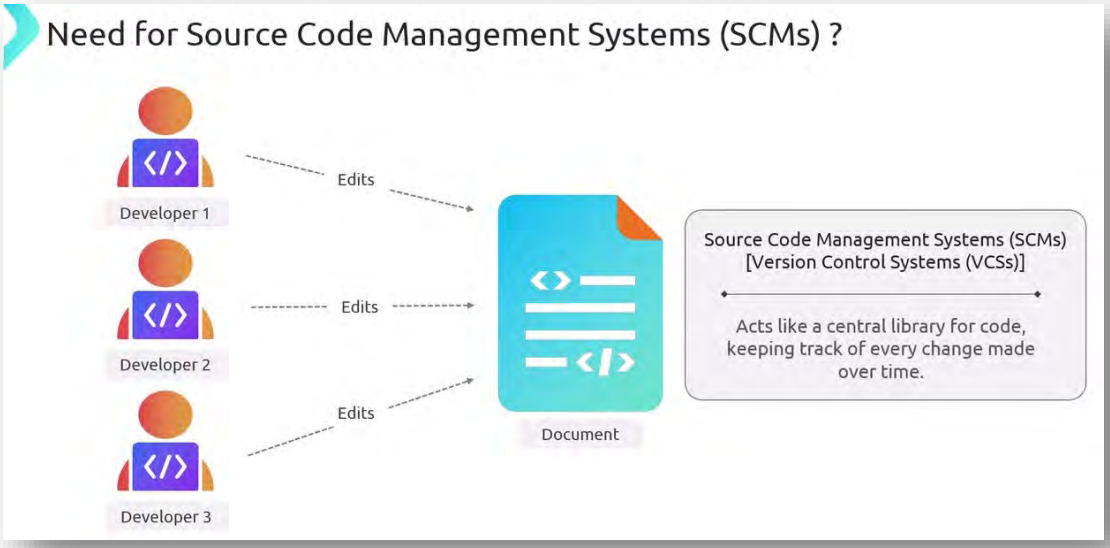
Basics of SCM

🎯 المقدمة

- ❖ الجزء ده بيشرح ليه بنحتاج Source Code Management Systems (SCM) أو Version Control Systems (VCS)في تطوير البرمجيات.
- ❖ بمعنى : إيه الفايدة الحقيقية من وجود نظام يتحكم في الكود بدل ما كل واحد يعدّل لوحده.

📝 الشرح التفصيلي

- تخيّل إنك شغال في مشروع، وكل الفريق بيعدل في نفس الملف في نفس الوقت من غير أي متابعة للتغييرات.
  - ◆ النتيجة : فوضى كاملة، مفيش حد عارف مين غيّر إيه.
- هنا بيجي دور SCM / VCS : اللي بيمنع الفوضى دي عن طريق إنه يعمل كأنه مكتبة مركزية للكود:
  - o كل تعديل بيتسجل.
  - o تقدر ترجع لأي نسخة قديمة.
  - o تقدر تعرف مين عدّل إيه وإمتى.
- الميزة التانية : التعاون بين الفريق (Collaboration) :
  - o كل المطورين يشتغلوا على نفس الكود من غير ما يحصل تعارض.
  - o أي تعديل ممكن يتراجع ويتراجع قبل الدمج .. وده بيتسمى Code Review
- كمان بيساعد في حل الـ conflicts اللي بتحصل لو 2 مطورين عدلوا نفس الجزء من الكود.
- بيخليك تقدر تشارك الكود بأمان مع باقي الفريق أو حتى مطورين من برّه.
- بتاخد تاريخ واضح للتغييرات (History)
- وأهم نقطة : لو حصل خطأ .. تقدر ترجع لنسخة سابقة بسهولة (Rollback)



🔑 مفاهيم إضافية لازم تبقى عارفها

- SCM = Source Code Management
- VCS = Version Control System

(الاتنين نفس المعنى تقريبًا)

- Single Source of Truth : يعني يبقى فيه مكان واحد معتمد للكود (زي Git repo) بدل ما يبقى نسخ متفرقة.
- Git هو أشهر نظام Version Control
- وبيتوفر له منصات Cloud زي:
  - o GitHub & GitLab & BitBucket & Gitea & Gogs

💡 مثال واقعي

تخيّل 3 مطورين شغالين على مشروع ويب :

- الأول بيعدل في صفحة الـ Login
- التاني بيضيف API جديد.
- التالت بيصلّح Bug

من غير SCM : كل واحد هيبعت الملف المعدل في شكل نسخة جديدة (Login_final_v2_fixed.js)

لكن مع Git : كل واحد يشتغل في branch خاص بيه، ولما يخلص يعمل merge بعد code review

---

📌 الملخص

- SCM/VCS بيمنع الفوضى في كتابة الكود.
- بيخلي الفريق يتعاون بشكل منظم.
- بيسجل كل التغييرات في تاريخ واضح.
- بيديك مرونة ترجع لأي نسخة قديمة.
- Git هو الأشهر، والمنصات زي GitHub و GitLab هي اللي بتسهل استخدامه.

SCM Technology

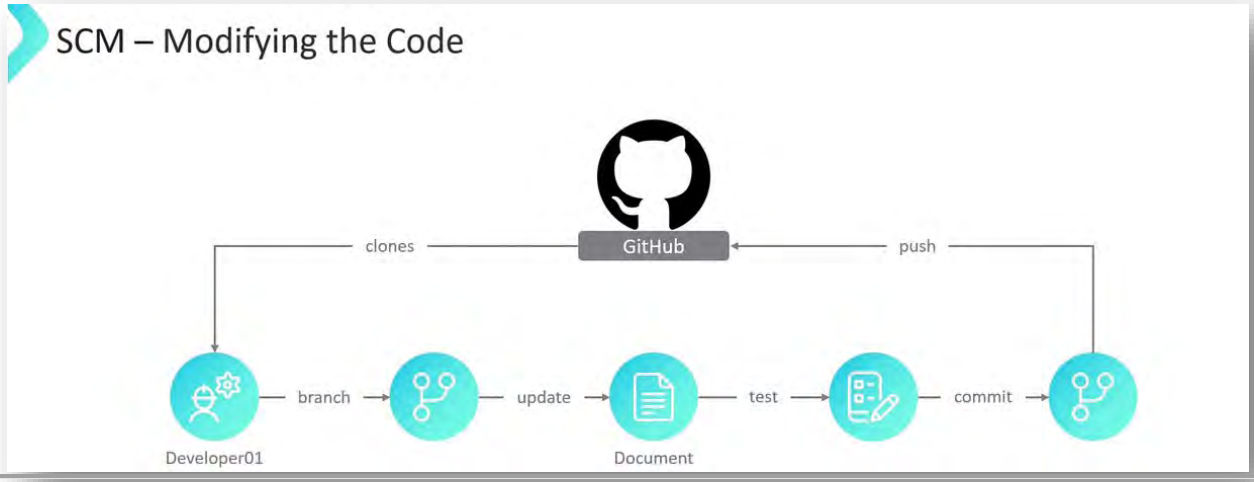
🎯 المقدمة

الجزء ده بيدخل أكتر في مصطلحات Source Control Management (SCM) خاصة مع Git ، وبيشرح إزاي بيتم التعامل مع الـ repository، الفروع(branches) ، التعديلات(commits) ، لحد ما نوصل لفكرة الـ pull request

---

📝 الشرح التفصيلي

- الكود في العادة بيتخزن في Repository (repo)
  - يعني مستودع للكود كله.
  - الأفضلية (Best Practice) : انك تعمل Repo لكل منتج أو مشروع منفصل عشان يبقى التحكم أسهل.
- SCM tools بتديك Access Control :
  - تحدد مين عنده صلاحية قراءة (read) .
  - مين يقدر يعدّل (Write)
  - ومين يقدر يدير (Admin)
- كل repo ممكن يبقى فيه فروع (Branches) :
  - بس فيه فرع رئيسي ثابت يمثل النسخة الرسمية الحالية (Main branch)
- الـ Workflow المعتاد في Git :
  1. تعمل Clone للـ repo على جهازك (نسخة محلية)
  2. تعمل Branch جديد عشان التغييرات متأثرش على الكود الأساسي.
  3. تعدّل الكود على الفرع المحلي.
  4. تختبر التعديلات محليًا.
  5. تعمل Commit : يعني snapshot من التغييرات + رسالة وصفية.
  6. تعمل Push للتعديلات للـ remote repo : وبكده تندمج مع باقي الفريق.



🔑 مصطلحات لازم تعرفها

- Diff : تقرير يوضح الفرق بين نسختين من الملف (إيه اللي اتضاف، اتمسح أو اتعدل).
- Commit : لقطة Snapshot بتسجل حالة الكود في لحظة معينة.
- HEAD : المؤشر اللي بيشاور على آخر commit في الفرع اللي إنت واقف عليه.
- Pull Request (PR) : طلب رسمي تدمج تغييراتك مع الكود الرئيسي.
  - بيتراجع ويتناقش من باقي المطورين.
  - ممكن يترفض أو يتعدل قبل الدمج.
  - الهدف هو Feedback + Code Quality

💡 مثال واقعي

- أنت شغال على Feature جديدة (مثلاً صفحة تسجيل مستخدم).
- تعمل Branch باسم feature/signup
- تكتب الكود وتختبره.
- تعمل Commit برسالة "Add signup page"
- تعمل Push وتفتح Pull Request
- زملاءك يراجعوا الكود، يكتبوا ملاحظات، وبعد التعديل يتم الـ Merge للـ main

---

📌 الملخص

- الـ Repo هو مكان الكود المركزي.
- Main branch يمثل النسخة الرسمية.
- بنشتغل على Branches لتقليل المشاكل.
- Diff يوضح الفروق.
- Commit = Snapshot + Message
- HEAD = آخر نقطة في الفرع.
- Pull Request = دمج منظم + مراجعة الكود قبل الدمج.

Basics of CI/CD

🎯 المقدمة

الجزء ده بيشرح أهمية CI/CD (Continuous Integration / Continuous Deployment & Delivery)
وبيوضح ليه بنحتاجه في المشاريع بدل ما نعتمد على شغل يدوي ومش منظم.

📝 الفكرة الأساسية

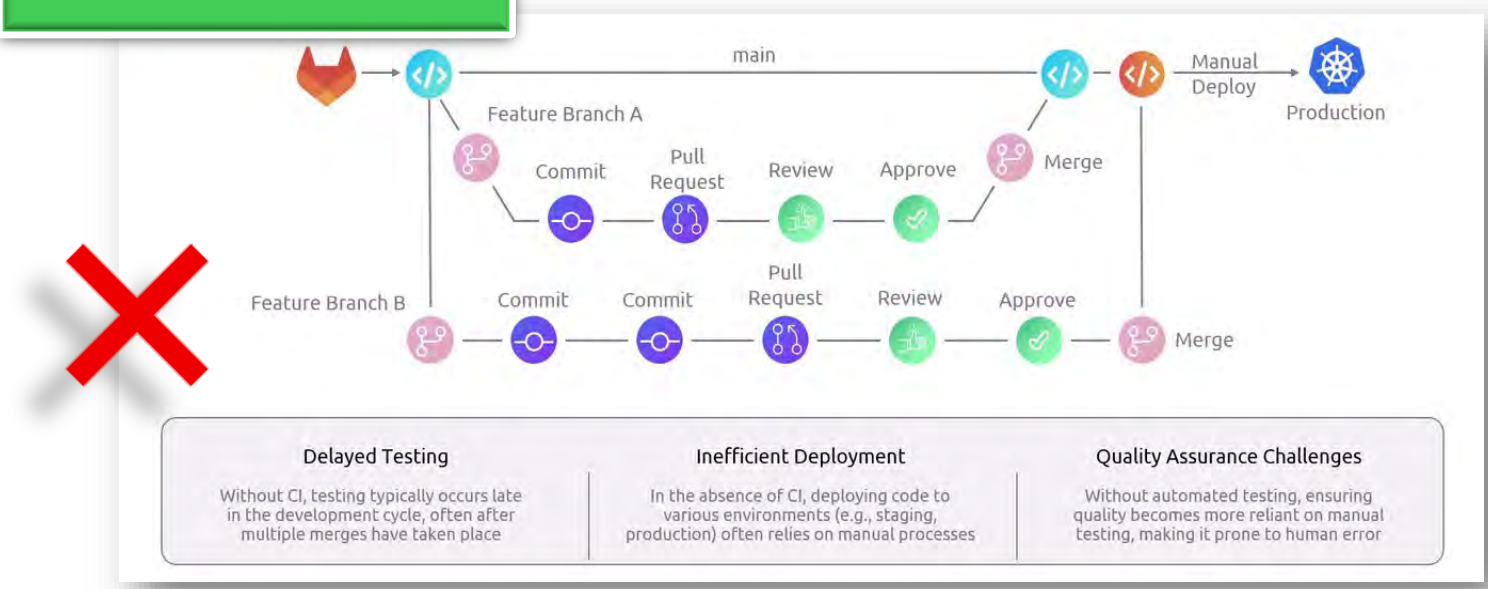
- إحنا إتفقنا ان الكود بيتخزن في Git Repository على (GitHub, GitLab, Bitbucket, ...)
- أي تغيير أو ميزة جديدة بتتعمل على Feature Branch
- بعد ما التعديلات تخلص : بيتعمل Pull Request (PR) عشان تندمج في الـ Main Branch

❑ المشكلة 💡 : في المشاريع التقليدية الكود أحيانًا بيتدمج ويتنشر على الإنتاج من غير اختبارات كافية .. وده خطر جدًا.

⚠️ التحديات من غير CI/CD

1. Delayed Testing : الاختبار بيتم متأخر بعد الدمج، فالأخطاء بتوصل للإنتاج.
2. (Manual Deployment) : ممكن يحصل اختلافات أو أخطاء في الإعدادات بين بيئة والتانية.
3. QA Challanges : لأن الاعتماد على اختبار يدوي بيزود نسبة الأخطاء البشرية.

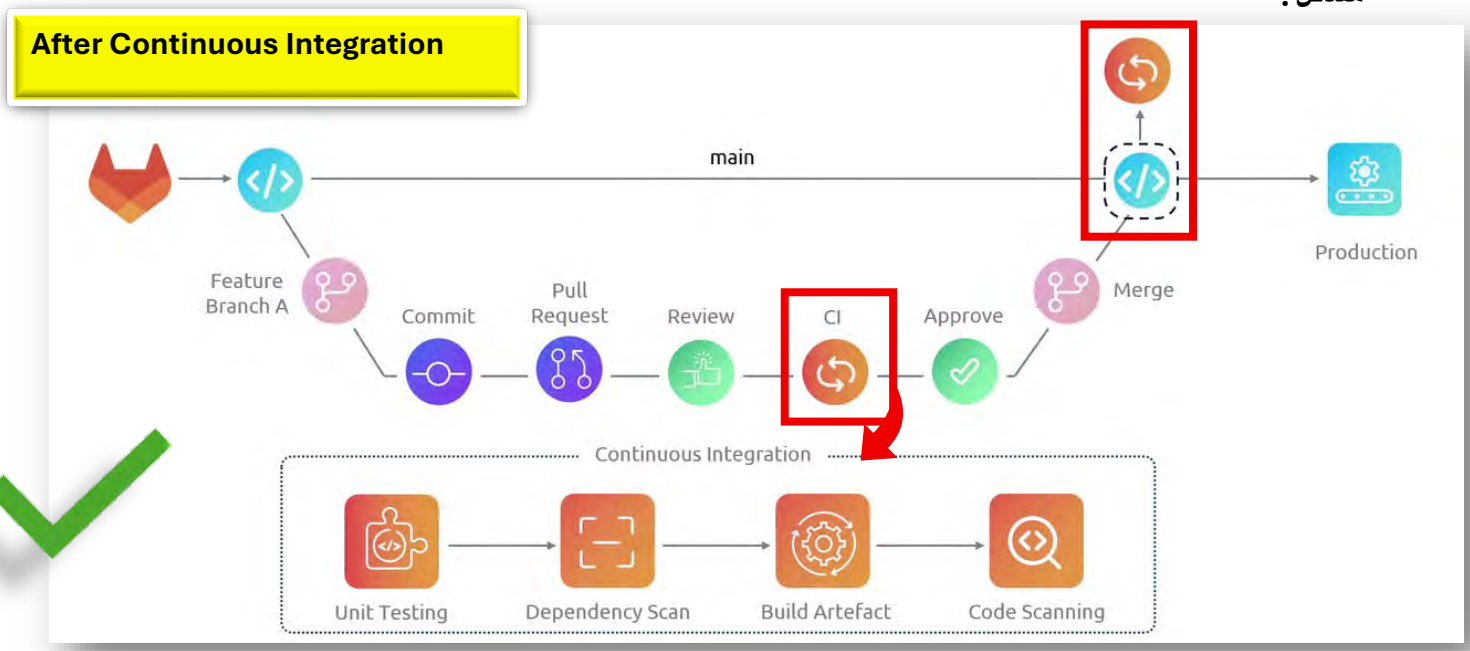
Before Continuous Integration



🔑 إزاي Continuous Integration بيحل المشكلة؟

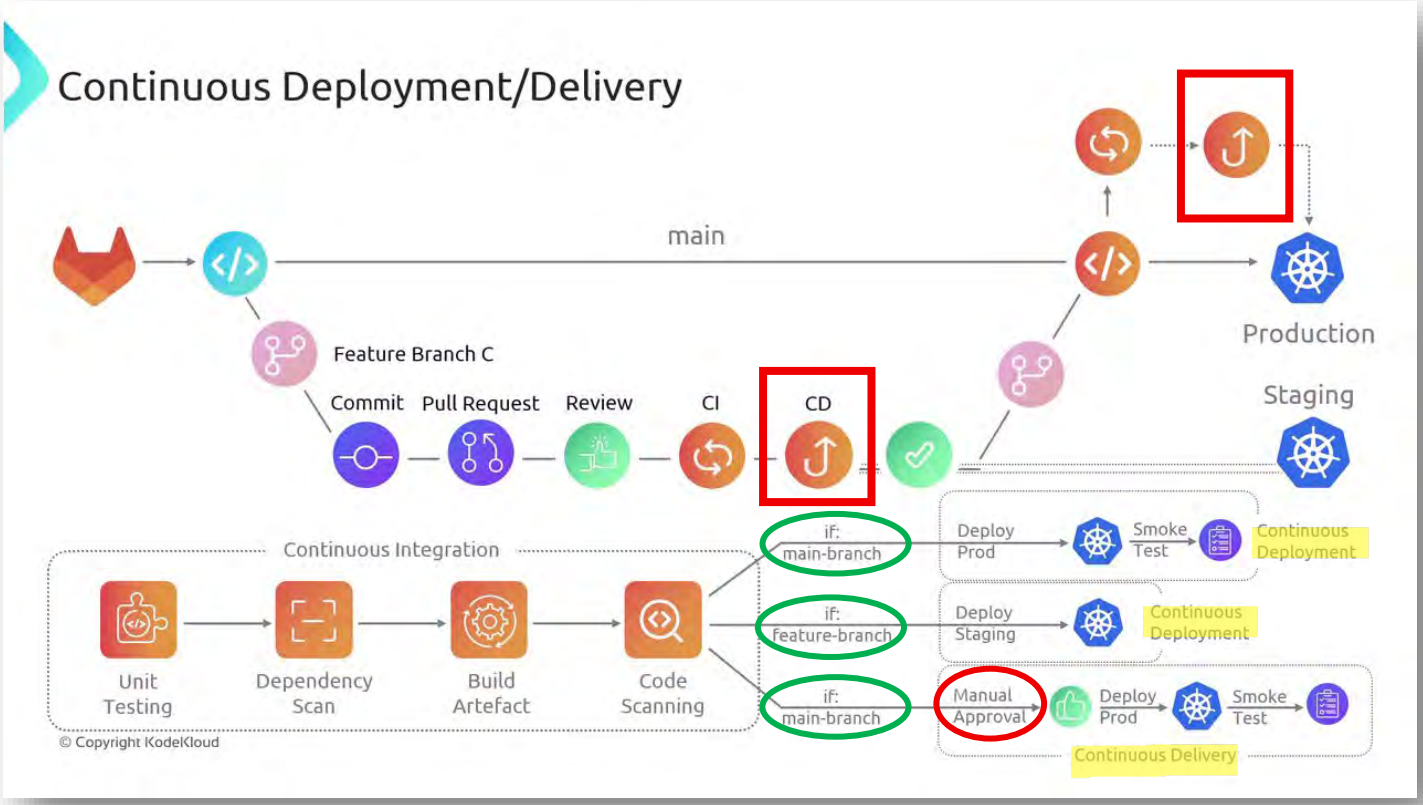
- أي PR بيتفتح : بيتشغل CI Pipeline أوتوماتيكياً.
- الـ CI Pipeline بيعمل :
  - Unit Tests
  - Dependency Scanning
  - Build Artifacts
  - Code Scans
- لو أي اختبار فشل .. فالمطور يعدل ويرفع Commit جديد .. وبعدين نفس الـ Pipeline يشتغل تاني.
- لما كله ينجح .. فالـ PR يتوافق عليه ويتدمج في الـ Main Branch

- بعد الدمج كمان .. نفس الاختبارات بتتكرر عشان نتأكد إن التغييرات من أكتر من فرع (A + B) بتشتغل مع بعض من غير مشاكل .



🔑 إزاي Continuous Delivery/ Continuous Deployment بيكمل الصورة؟

- بعد ما الكود ينجح في CI :
  - بيتعمل نشر (Deploy) لبيئة قريبة من الإنتاج (Staging/Dev) : للاختبار العملي.
  - بعد كده :
    - **Continuous Deployment : الكود يتنشر في الإنتاج أوتوماتيكياً بعد نجاح CI**
    - **Continuous Delivery : نفس الخطوات لكن قبل النشر في الإنتاج لازم (Manual Approval)**



💡 مثال عملي

❖ الخطوات الفعلية اللي كل Developer بيعملها :

1️⃣ Developer Commit

👷 المطور بيشتغل على جهازه المحلي (local machine)

- يعمل clone للـ repo
- يفتح branch جديدة عشان يشتغل على feature أو bug fix
- بعد ما يكتب الكود ويعمل test محليًا، بيعمل commit :
  - commit = snapshot للكود في لحظة معينة.
  - بيتسجل فيه رسالة (commit message) تشرح التغيير.

2️⃣ Pull Request (PR)

بعد ما يخلص التغييرات ، المطور بيبعت Pull Request للـ repo الرئيسي.

- الـ PR ده بمثابة طلب رسمي يدمج الكود بتاعه في main branch
- باقي الفريق أو reviewer بيشوفوا الكود ، يعملوا Code Review، يعلقوا أو يوافقوا.
- وممكن يطلبوا تعديلات قبل ما يتدمج.

3️⃣ CI Pipeline (Build + Test + Scan)

أول ما يتعمل PR أو commit جديد، بيشتغل CI Pipeline (Continuous Integration)

- Build : يتأكد إن الكود بيتبني (compile / package) بدون أخطاء.
- Test : يشغل unit tests, integration tests يتأكد إن التعديلات مبوظتش حاجة.
- Scan : أحيانًا يتعمل static code analysis أو security scan يتأكد من الجودة والأمان.

لو أي step في الـ CI فشلت .. الـ PR مايتدمجش غير لما المشكلة تتصلح.

4️⃣ Merge to Main Branch

بعد الموافقة على الـ PR ونجاح الـ CI pipeline :

- الكود الجديد بيتدمج في main branch
- الـ main branch = المصدر الأساسي / نسخة الإنتاج المستقرة (single source of truth)

5️⃣ CD (Deploy to Staging → Production)

هنا ييجي دور Continuous Deployment / Delivery :

- أولًا يتعمل deploy لـ staging environment (بيئة شبه الإنتاج للتجربة)
- لو كل حاجة تمام في الـ staging .. هايتعمل deploy لـ production (البيئة الحقيقية اللي بيشوفها المستخدمين)

ده بيوفر:

- سرعة في طرح التحديثات.
- تقليل الأخطاء وقت النشر.
- Rollback سريع لو حصلت مشكلة.

📌 الملخص

- CI = اختبارات أوتوماتيكية لكل تغيير .. (نمنع مشاكل بدري.)
- CD = نشر أوتوماتيكي (Deployment/Delivery) .. (يقلل التدخل اليدوي ويحافظ على الاستقرار.)
  - Continuous Deployment = نشر مباشر بعد الاختبارات في البرودكشن
  - Continuous Delivery = نشر بعد اختبار في البرودكشن بشررررط موافقة بشرية.

Software Testing

أهمية الـ Testing في دورة حياة تطوير السوفتوير

- الـ Testing جزء أساسي جدًا في أي Software Development Cycle
- ليه؟ لأنه بيضمن إن أي تغيير أو تعديل (Refactoring أو Upgrade) بيتم بشكل صحيح ومش بيكسر النظام.
- الهدف الرئيسي : اكتشاف الأخطاء (Defects) بدري وتحسين جودة السوفتوير.

📝 أنواع الـ Tests

الـ Testing مش نوع واحد، لأ ده فيه أنواع متعددة وكل نوع له دور:

- Unit Testing : بيختبر أجزاء صغيرة جدًا من الكود (زي Functions أو Methods) علشان يتأكد إنها شغالة صح.
- Integration Testing : بيتأكد إن الـ Components أو الـ Modules المختلفة لما تتجمع مع بعض تشتغل كويس.
- Smoke Testing : زي Check سريع للـ Features الأساسية عشان نعرف إذا السيستم يقدر يكمل Testing ولا لأ.
- Functional Testing : بيتأكد إن السوفتوير بيعمل المطلوب منه حسب الـ Requirements
- Non-Regression Testing : بعد أي تعديل، بيتأكد إن الحاجات القديمة لسه شغالة كويس.
- Acceptance Testing : ده من منظور المستخدم (User's Perspective) علشان نعرف هل السوفتوير بيلبي احتياجاته ولا لأ.
- Code Quality & Static Analysis : يراجع الكود نفسه (من غير تشغيله) عشان يكتشف أخطاء أو مشاكل محتملة.
- Performance & Security Testing : يختبر الأداء (سرعة، استهلاك موارد) وكمان الأمان.
- Manual Testing : Testing بايدينا، بس ده بيكون قليل جدًا وبالذات في حاجات زي الـ UX

🧩 تقسيم الـ Tests حسب السرعة

- Faster Automated Tests :
  - Unit Tests
  - Integration Tests
  - Smoke Tests

  دي سريعة جدًا ولازم تتعمل بشكل متكرر (حتى بعد كل Code Change)

- Slower Automated Tests :
  - Functional Tests
  - Non-Regression Tests
  - Acceptance Tests

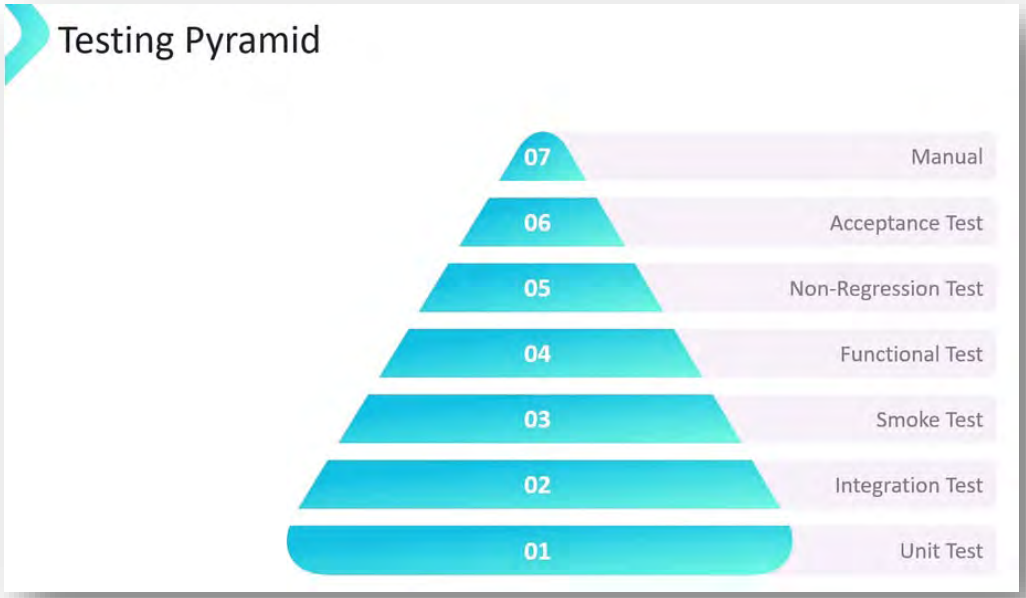
  * دي بتاخد وقت أطول، فمش لازم تتعمل كتير، غالبًا بتتعمل بعد ما Unit Tests تنجح*

- Manual Tests :
  - بتتعمل يدويًا، والأفضل استخدامها قليل جدًا.

Testing Pyramid 🔑

- ❑ **It's a model that guides software testing strategy. It emphasizes a balanced approach to testing, prioritizing different types of tests based on their speed, cost, and effectiveness.**
  - **تحت (القاعدة) : Unit Tests (كتيرة جدًا، أسرع، وأرخص)**
  - **فوق شوية : Integration / Functional Tests (عدد أقل)**
  - **فوق القمة : Manual / Acceptance Tests (عدد قليل جدًا)**

💡 **الفكرة : كل ما نطلع لفوق، الاختبارات تبقى أبطأ وأغلى، فبنقلل عددها.**



Key Principles في الـTesting

- **Speed & Frequency : الـ Unit Tests لازم تكون سريعة وتتعمل باستمرار.**
- **Failure Analysis : لو اختبار من المستوى العالي (زي Functional) فشل، غالبًا المشكلة في حاجة أقل (زي Unit Test) أو الكود نفسه**
- **Efficiency : كل ما الاختبارات عالية المستوى بتكلف وقت وموارد أكتر، فبنحاول نقلل استخدامها ونركز على الأساسيات.**

📌 الخلاصة :

الـTesting مش مجرد خطوة إضافية، ده أساس جودة السوفتوير.
المفروض يبقى عندنا إستراتيجية Testing متوازنة : نركز على Unit Tests الكتيرة والسريعة، وبعدين نضيف تدريجيًا الاختبارات الأعلى زي Integration و Functional، وفي الآخر Manual Testing يكون محدود جدًا.

Jenkins Architecture

🎬 مقدمة

Jenkins مش مجرد Tool CI/CD عادي، هو نظام بيوفرلك Distributed Architecture يعني بدل ما كل حاجة تشتغل على Server واحد، Jenkins بيوزع الشغل بين مكونات مختلفة عشان السرعة ، الـ scalability ، وحماية الـ controller

---

📌 الهدف (Purpose)

Jenkins Architecture معمولة عشان:

- تدير CI/CD Pipelines بكفاءة.
- توزع المهام بين أكثر من Machine (Scalability)
- تديك مرونة (تضيف Agents جديدة بسهولة)
- تحافظ على الـ Controller آمن (يفضل بعيد عن الـ Build Jobs)

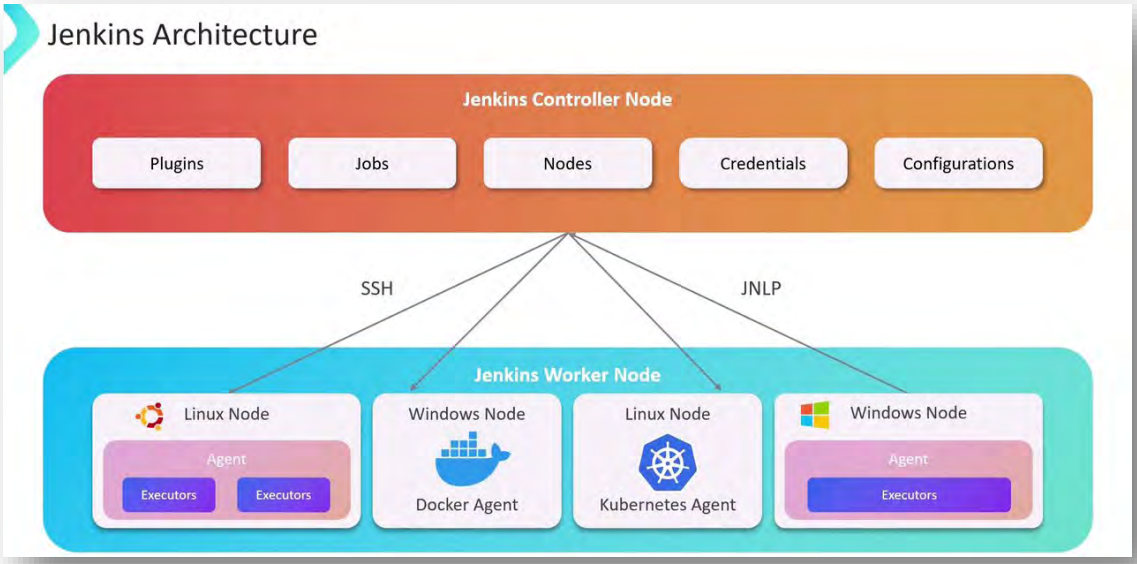
⚙️ الـ Mechanism (إزاي بيشتغل)

الـ Architecture مبنية من 4 مكونات أساسية:

1. Controller (Jenkins Server)
   - ده الـ "مخ" بتاع Jenkins
   - بيعمل :
     - Authentication & Authorization
     - إدارة Jobs و Pipelines (تعريف – جدولة – متابعة) .. (لكن تنفيذ الـ Job فعليا بيبقي في الـ Worker)
     - Web Interface للـ Configurations والـ Plugins تشوف منها كل حاجة
   - ممكن نشغل (Controller + Worker) في نفس الجهاز) في مشاريع صغيرة، لكن في الكبير لازم تفصل.
2. Nodes (Workers / Agents)
   - دي الماكينات اللي بتعمل الـ Build Jobs فعليًا.
   - ممكن تكون Linux, Windows, أو أي OS
   - بتتواصل مع الـ Controller بـ SSH أو JNLP (Java Network Launch Protocol)
3. Executors
   - جوا كل Node بيبقى فيه عدد معين من Slots اسمها Executors (Threads)
   - كل Executor يقدر يشغل Job واحد.
   - زيادة Executors = Parallel Jobs = سرعة أكتر.
   - يعني لو عندك Node فيها 2 Executors ← تقدر تشغل 2 Jobs في نفس الوقت.
   - Best practice : 1 Executor لكل Node (لأمان أعلى) ، أو 1 لكل CPU Core لو Tasks صغيرة.
4. Agents
   - هو الـ "وسيط" اللي بيحدد إزاي الـ Node يتصل بالـ Controller
   - ممكن يكون SSH Agent ، أو JNLP Agent
   - الأدوات (Tools + Dependencies) بتتسطب على الـ Node اللي فيه الـ Agent
   - كمان ممكن تستخدم Docker Agents (كل Build يشتغل جوه Container نظيف) أو Kubernetes Pods (On-Demand Scaling)

Options / Examples 🧩

- Node يتصل بـ SSH : مناسب للـ Linux servers
- Node يتصل بـ JNLP : مناسب لو مفيش SSH Access
- Docker Agent : مفيد لو عايز بيئة معزولة لكل Build (بنسخة Java معينة مثلًا)
- Kubernetes : للـ Enterprise setups اللي عايزة Auto-scaling للـ Agents



أنواع Modes 🔑

- Standalone Mode : Controller بيعمل كل حاجة. (مناسب لمشاريع صغيرة).
- Distributed Mode : Controller + Multiple Nodes (الموصى بيه للمشاريع المتوسطة والكبيرة)

مثال عملي (Use-case) 📝

افترض عندك: Jenkins Controller + 2 Nodes

1. Dev عمل Push للـ Code
2. Controller استقبل الـ Pipeline
3. شاف Node 1 فاضية فيها Executor
4. بعت الـ Job لـ Node 1
5. Node 1 شغل الـ Build وطلع Logs + Artifacts
6. Controller رجع عرض النتيجة في Dashboard

ولو الـ Node 1 مشغولة ← يبعت لـ Node 2

ولو محتاج تزود سرعة ← تزود Nodes أو Executors

ملخص + مقارنة ✅

- 🧠 Controller = Brain (المخ اللي بيدير)
- 👷 Nodes = Workers (اللي بينفذوا)
- Executors = Slots (كام Job هايشتغل في نفس اللحظة)
- Agents = طريقة الاتصال + إدارة Environment (SSH / JNLP / Docker / K8s)
- الفكرة كلها إن Jenkins مش محدود بجهاز واحد .. وممكن توسع وتوزع الشغل.

Jenkins Installation Option

🎬 مقدمة

السكريبت بيتكلم عن إزاي تبدأ مع Jenkins من أول المتطلبات (هاردوير + سوفتوير) لغاية طرق التثبيت المختلفة، وأخيرًا أين يخزن Jenkins ملفاته المهمة (JENKINS_HOME)

يعني باختصار: قبل ما تشتغل CI/CD ، لازم يكون السيرفر مجهز صح.

📌 المتطلبات (Requirements)

🖥️ المتطلبات الهاردوير

- أقل حاجة (Minimum) :
  - 2 CPU Cores
  - 256 MB RAM
  - 1 GB Disk

👉 عمليًا : ده يكفي للتجارب أو المشاريع الصغيرة جدًا.

مثال : لو عايز تجرب Jenkins على VM بسيطة أو لاب شخصي.

- الموصى به (Recommended) لفريق صغير:
  - 4 CPU Cores
  - 4 GB RAM
  - 50 GB Disk

👉 السبب Jenkins : بيخزن(Logs, Artifacts, Plugins..) .. لو القرص صغير، هتتزنق بسرعة.

🔧 نصيحة عملية:

في مشاريع حقيقية، بيوصلوا لـ 8 CPU + 16GB RAM + 200GB Disk (خصوصًا لو عندك Builds كبيرة أو Microservices كتيرة)

🥥 متطلبات السوفتوير

❖ Jenkins بيعتمد على Java Virtual Machine

- عندك خيارين:
  - JRE (Java Runtime Environment) : يكفي لتشغيل Jenkins
  - JDK (Java Development Kit) : فيه Tools إضافية للتطوير وحل المشاكل ← الأفضل للإنتاج (Production)

🖥️ أوامر عملية (Ubuntu) :

- تحديث النظام : sudo apt update
- تثبيت OpenJDK : sudo apt install openjdk-17-jdk -y
- تأكد من النسخة : java -version

💡 Jenkins بيشتغل كويس مع Java 11 أو Java 17 (LTS)

❖ الخيار الثالث ← Web Browser : ده اللي هتدخل منه على واجهة Jenkins

---

⚙️ طرق التثبيت (Installation Options)

1️⃣ war File.

- ممكن تشغل Jenkins في أي OS عنده Java
- بتنزل الملف jenkins.war وتشغله :

- wget https://get.jenkins.io/war-stable/2.452.2/jenkins.war
- java -jar jenkins.war

- هيشتغل على البورت 8080 افتراضيًا : http://localhost:8080

📌 مناسب : للتجارب أو لو مش عايز تربطه بالـ OS Service

2️⃣ OS Packages

- Jenkins عنده Native Packages :
  - Windows (Installer GUI سهل)
  - macOS (Installer GUI أو brew)
  - Linux (Debian, Ubuntu, RedHat...)

💻 أوامر عملية (Ubuntu/Debian) :

```
curl -fsSL https://pkg.jenkins.io/debian/jenkins.io-2023.key | sudo tee \
 /usr/share/keyrings/jenkins-keyring.asc > /dev/null

echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
 https://pkg.jenkins.io/debian binary/ | sudo tee \
 /etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt update  &  sudo apt install jenkins -y
```

- دي أفضل طريقة في الإنتاج عشان بتندمج مع Systemd

3️⃣ Cloud Templates

- AWS, Azure, GCP بيوفروا Jenkins كـ Managed Service
- تقدر تطلع Jenkins جاهز من الـ Marketplace
- بيوفر وقت وبيطبق Best Practices من الأول.

📌 مثال AWS : عندهم Jenkins on EC2 Template جاهز بالـ Security Groups و EBS Volume

---

4️⃣ Docker

- Jenkins عنده Official Docker Image
- أسهل طريقة عشان تبني بيئة ثابتة (Reproducible)

💻 أوامر عملية:

- docker run -d -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home \
  jenkins/jenkins:lts

📌 مميزات:

- كل Build في Container نظيف.
- تقدر تحدد Plugins/Tools مسبقًا.
- مناسب جدًا لو بتشتغل في فرق DevOps

```
Terminal
$ tree /var/lib/jenkins

+- builds                    (build records)
     +- [BUILD_ID]           (subdirectory for each build)
          +- build.xml       (build result summary)
          +- changelog.xml   (change log)
 +- config.xml               (Jenkins root configuration file)
 +- *.xml                    (other site-wide configuration files)
 +- fingerprints             (stores fingerprint records, if any)
 +- identity.key.enc         (RSA key pair that identifies an instance)
 +- jobs                     (root directory for all Jenkins jobs)
     +- [JOBNAME]            (sub directory for each job)
          +- config.xml      (job configuration file)
+- plugins                   (root directory for all Jenkins plugins)
      +- [PLUGIN].jpi        (.jpi or .hpi file for the plugin)
+- secrets                   (root directory for the secret+key for credential decryption)
     +- hudson.util.Secret            (used for encrypting some Jenkins data)
     +- master.key                    (used for encrypting the hudson.util.Secret key)
     +- InstanceIdentity.KEY (used to identity this instance)
 +- userContent              (files served under your https://server/userContent/)
 +- workspace                (working directory for the version control system)
```

Jenkins Home Directory 📂

❖ هو المكان اللي بيتخزن فيه كل حاجة:

- Jobs
- Plugins
- Configs
- Logs
- Artifacts

❖ المسار الافتراضي:

- war file → ~/.jenkins
- Linux Packages → /var/lib/jenkins

📻 أمر عملي للتأكد: echo $JENKINS_HOME

❖ تخصيص :

- ممكن تغير مكانه : export JENKINS_HOME=/data/jenkins

📌 نقطة مهمة Backup :

- لازم تعمل Backup للـ Jenkins Home دوريًا.
- لو ضاع ← كأنك فقدت Jenkins كله (Jobs, Configs, Plugins)

📻 أمر عملي للنسخ الاحتياطي : tar -czvf jenkins_backup_$(date +%F).tar.gz /var/lib/jenkins

---

✅ ملخص

1. جهز الهاردوير والجافا قبل أي حاجة.
2. اختر طريقة التثبيت حسب بيئتك:
   - war ← للتجارب.
   - OS Packages → Production
   - Cloud Templates → Cloud Teams
   - Docker ← بيئة قابلة لإعادة الإنتاج وسهلة.
3. اعرف مكان JENKINS_HOME وخليه جزء من خطة النسخ الاحتياطي.

Demo-Initial Setting & JVM Tunning

📌 المقدمة

❖ هانتكلم عن مشكلة شائعة بتحصل مع Jenkins وهي:
Out of Memory Error يعني Jenkins بيحاول يشتغل بس الـ JVM (Java Virtual Machine) اللي بيشغله مش واخد مساحة كافية من الـ RAM

❖ الشرح ده بيوريك:

1. إزاي تكتشف المشكلة من اللوجات.
2. إزاي تعرف الـ Heap Size الحالي.
3. إزاي تزود الـ Heap Size باستخدام systemd configs.
4. القيم الموصى بيها من Documentation.
5. تتأكد بعد التعديل إن Jenkins رجع يشتغل.

📝 الشرح خطوة بخطوة

1. ملاحظة المشكلة
   - Jenkins مش قادر يفتح الـ Dashboard (على localhost:8080)
   - في بعض الأحيان يدي Error 500
   - السبب ← JVM ran out of Java Heap Space : يعني الذاكرة المخصصة للـ Java قليلة جدًا .

---

2. التأكد من المشكلة من اللوجات
   - تستخدم الأمر : journalctl -u jenkins | grep -i memory
   - هتلاقي رسائل بتقول إن Jenkins ran out of memory .

💡 معلومة مهمة:
Java بتقسم الذاكرة لأجزاء (Heap / Non-Heap) ، و Jenkins بيعتمد بشكل أساسي على الـ Heap Memory

---

3. معرفة الإعدادات الحالية
   - شوف ملف الخدمة بتاع Jenkins : systemctl cat jenkins
   - هتلاقي فيه Environment Variables زي : JAVA_OPTS=-Xms90m -Xmx90m
     - -Xms: Initial Heap Size (اللي يبدأ بيه)
     - -Xmx: Maximum Heap Size (أقصى حاجة ممكن يستخدمها)
     - ❑ هنا الاتنين متظبطين على 90MB بس! وده قليل جدًا لـ Jenkins

---

4. تعديل الإعدادات
   - عشان تعدل القيم : sudo systemctl edit jenkins
   - هيظهرلك ملف Override
   - ضيف فيه السطر:

- [Service]
- Environment="JAVA_OPTS=-Xms1024m -Xmx1024m"

- هنا:
  - زودت الـ Heap Size لـ 1GB (ممكن تخليها 2 GB لو جهازك يسمح)
  - استخدمت نفس القيمة في البداية والنهاية عشان يبقى ثابت ومستقر.

5. القيم الموصى بها من Documentation

- الحد الأدنى :
  - 2GB لـ Operations Center
  - 4GB لـ Controllers / Production
- الحد الأقصى الموصى به 16GB :
- لو محتاج أكتر من كده ← استخدم Horizontal Scaling بدل ما تزود Memory لجهاز واحد.

---

6. خطوات عملية قبل وبعد التعديل

1. أوقف Jenkins قبل ما تعدل : sudo systemctl stop jenkins
2. عدل الإعدادات.
3. شغّله تاني: sudo systemctl restart jenkins
4. تأكد من الحالة: systemctl status jenkins
5. شوف الـ JVM options المحدثة: ps aux | grep -i jenkins

- ❑ هتلاقي إن -Xmx و -Xms بقوا m.1024

---

7. التحقق من الحل

- افتح http://<IP>:8080
- هتلاقي Jenkins رجع يفتح الـ Dashboard عادي.
- تقدر تعمل Login وتكمل الشغل.

---

8. Best Practices إضافية

- Memory Monitoring :
  استخدم Monitoring Tools (زي Prometheus + Grafana) عشان تتابع استهلاك الذاكرة.
- Disk & GC Tuning :
  مش بس الـ Heap ، لازم تظبط Garbage Collector settings و Disk space
- Containers :
  لو Jenkins شغال في Docker/K8s ، الأفضل تستخدم
  -XX:+UseContainerSupport و -XX:MaxRAMPercentage بدل -Xmx

🎯 الخلاصة

- سبب المشكلة Java Heap Size قليل (90MB)
- حلها : تزود القيم باستخدام -Xms و -Xmx في Jenkins service
- القيم الموصى بيها: من 2 GB لـ 4 GB حسب حجم الـ Instance
- بعد التعديل وإعادة التشغيل Jenkins ← يشتغل طبيعي.
- نصيحة : تابع استهلاك الموارد دايمًا وحط القيم حسب الـ workload

Running Junkins WAR as a standalone App

📌 المقدمة

هانتكلم عن طريقة تانية لتشغيل Jenkins :

- المرة اللي فاتت شغلناه على Ubuntu بـ apt install Jenkins
- المرة دي هنشغّله مباشرةً باستخدام ملف الـ WAR (وده ملف Java تنفيذي)

الفكرة:

- تشغيل Jenkins من WAR file بيديك مرونة أكتر ← تقدر تختار أي إصدار، وتخصص البورت أو تضيف prefix أو تعملHTTPS
- ودي طريقة مفيدة جدًا لو عايز تعمل تجربة على نسخة جديدة من Jenkins من غير ما تأثر على النسخة الأساسية اللي مثبتة.

📝 الشرح خطوة بخطوة

1. التأكد من النسخة اللي شغالة بالفعل

- استخدم : ps aux | grep jenkins
- بيظهرلك العملية اللي شغّالة.
- هنا بيوري:
  - الـ Heap Size = 1GB (متظبط قبل كده)
  - بيستخدم Jenkins WAR اللي نزل تلقائي مع apt install
  - شغال على البورت الافتراضي 8080

💡 معلومة مهمة:

بالتثبيت عبر apt install، Jenkins بيحط ملفات الـ WAR والـ configs في /usr/share/java/ وبيستخدم systemd service

---

2. تحميل نسخة جديدة من Jenkins WAR

War Jenkins Packages

1. روح على موقع War Jenkins Packages
2. اختار الإصدار اللي عايزه (هنا 2.479.3).
3. أنشئ فولدر خاص بالـ WAR الجديد : mkdir jenkins-war && cd jenkins-war
4. نزّل الملف: wget https://get.jenkins.io/war/2.479.3/jenkins.war

---

3. تشغيل Jenkins بالـ WAR

- الأمر الأساسي : java -jar jenkins.war
- لكن ده هيحاول يشغل Jenkins على الـ default port 8080 ← والنسخة القديمة أصلاً شغالة عليه ... طاب والعمل ؟!؟!؟!

---

4. تخصيص الإعدادات (Ports – Prefix – HTTPS)

تقدر تضيف Parameters مع الأمر:

- تغيير البورت: java -jar jenkins.war --httpPort=9090
- إضافة Prefix : java -jar jenkins.war --prefix=/myjenkins
- تشغيل HTTPS (مع Keystore)
- java -jar jenkins.war --httpPort=-1 --httpsPort=8443 --httpsKeyStore=/path/to/keystore.jks --httpsKeyStorePassword=yourpassword

📌 في الفيديو :

- الراجل اختار يشتغل بالبورت (7777) وكمان ضاف prefix = /DasherTechnologies
- النتيجة ← Jenkins الجديد بيفتح على: http://localhost:7777/DasherTechnologies/

5. أول تشغيل للنسخة الجديدة

- لما يشغّل الـ WAR ، بيطلع initial admin password (موجود في الـ console)
- افتح المتصفح:
  - النسخة القديمة (2.462.1) http://localhost:8080
  - النسخة الجديدة (2.479.3) http://localhost:7777/DasherTechnologies
- حط الباسورد ← هيطلب تنصيب ← Plugins زي التنصيب العادي.

---

6. التحقق من وجود النسختين

- شغّل: ps aux | grep -i jenkins
- هتلاقي:
  - نسخة قديمة (من apt) و بتستخدم WAR في /usr/share/java
  - نسخة جديدة (من الملف اللي حملناه) بتستخدم WAR اللي في jenkins-war + Prefix

❑ يعني عندك Jenkins servers شغالين في نفس الوقت، بس متفرقين في الـ paths أو الـ prefixes أو الـ Ports

---

💡 إضافات عملية (Tips & Best Practices)

- تجارب آمنة : تشغيل Jenkins من WAR بيسمحلك تختبر إصدار جديد من غير ما تضرب النسخة القديمة.
- Port Management : الأفضل تغيّر البورت دايمًا عشان متلخبطش بين النسخ.
- Service Integration : لو عايز النسخة الجديدة تشتغل كـ service (systemd) فممكن تعمل Unit file خاص بيها.
- Upgrade Strategy : لو هتعمل Upgrade رسمي، الأفضل توقف النسخة القديمة وتستبدلها بالكامل.
- ⚠️ Production Usage : في بيئات Production ، يفضل تستخدم apt install أو Docker image عشان تبقى متكاملة مع الـ system

---

🎯 الخلاصة

- Jenkins ممكن يتشغل بطريقتين:
  1. System package (apt install)
  2. مباشرة من ملف WAR
- تشغيله من WAR بيديك حرية تختار أي إصدار وتخصص إعدادات (port, prefix, https)
- في المثال:
  - نسخة قديمة 2.462.1 شغالة على 8080.
  - نسخة جديدة 2.479.3 شغالة على 7777 بـ prefix /DasherTechnologies
- كده تقدر تشغّل أكتر من Jenkins instance في نفس الوقت على نفس الجهاز.

Jenkins User Interface Overview

🎯 المقدمة

هانتكلم عن الـ Jenkins User Interface (UI) يعني بعد ما نعمل installation للـ Jenkins ونفتحه من المتصفح، إيه أهم المكونات اللي هنشوفها ونتعامل معاها.

الهدف إنك تفهم:

- إزاي تشوف نسخة Jenkins اللي شغالة.
- إزاي تدير Jenkins من خلال الـ dashboard
- إزاي تضيف system messages (زي Banner أو إعلانات داخلية)
- إزاي تشوف المعلومات عن الـ system, logs, plugins
- إزاي تغيّر الشكل (themes) باستخدام plugins

📝 شرح السطور بالتفصيل

1. الدخول للـ Jenkins بعد التثبيت

"post-installation, this is the page where you will be logging into."

◆ يعني بعد ما تفتح Jenkins من المتصفح (عادة على http://localhost:8080 هتجيلك صفحة login)

الـ default user هو admin، والـ password بتجيبه من الملف

/var/lib/jenkins/secrets/initialAdminPassword

---

2. معرفة نسخة Jenkins

"click on the Jenkins icon at the bottom right ... shows version ... mavenized dependencies."

◆ أي نسخة Jenkins بتستخدمها ، فلازم تبقى عارفها عشان:

- تعرف إمتى تعمل upgrade
- تشوف التوافق مع plugins

📌 الأمر العملي : java -jar jenkins.war --version

أو من GUI : تضغط على أيقونة Jenkins ← هتلاقي رقم النسخة + license info + dependencies

---

3. تسجيل الدخول / تسجيل الخروج

"logged in as admin ... logout → login page"

◆ Jenkins عنده Role-based Access

الـ admin يقدر يشوف كل الإعدادات، لكن الـ normal user ممكن صلاحياته تكون أقل.

💡 Best Practice : متخليش كل الناس تستخدم admin "user" اعمل حسابات مخصصة وأدوار (roles)

---

4. Manage Jenkins

"manage Jenkins option ... notifications ... distributed builds recommended"

◆ هنا بتلاقي أهم الـ Settings :

- System Configuration : زي الـ Home directory, executors
- Security : صلاحيات المستخدمين.
- Plugins : إضافة أو حذف plugins
- System Logs : مشاكل Jenkins

⚠️ Jenkins بيحذرك لو مش ماشي بالـ Best Practices (مثلاً : شغال على node واحدة فقط)
في الشركات الكبيرة دايمًا بيعملوا Distributed Builds (يعني Jenkins master + build agents)

---

5. System Configuration

"home directory ... add system message ... preview ... executors ... Jenkins URL."

◆ الحاجات المهمة هنا:

- **Home directory** : المكان اللي Jenkins بيخزن فيه الـ jobs, configs, logs غالبًا /var/lib/Jenkins
- **System Message** : رسالة تظهر لكل المستخدمين (مثلاً downtime أو ترحيب)
- **Executors** : عدد الـ parallel jobs اللي ممكن Jenkins node ينفذها.
- **Labels** : بتستخدم عشان تخصص jobs لآلات معينة.
- **Jenkins URL** : مهم جدًا عشان الروابط اللي بتتبعت بالإيميل (Build results)

📌 مثال إضافة رسالة:

Welcome to Dasher Team CI organization

---

6. System Information

"system properties ... environment variables ... plugins installed"

◆ صفحة مهمة جدًا:

- بتوريك كل Java system properties
- كل الـ environment variables
- قائمة بالـ plugins اللي متسطبة (بما فيهم اللي جايين pre-installed)

💡 **Trick** : لو عندك مشكلة في Pipeline ، دايمًا بص هنا تشوف إذا environment variables مضبوطة ولا لأ.

---

7. Logs من الـ UI

"system logs ... instead of journalctl"

◆ Jenkins بيحتفظ بالـ logs داخليًا.

لكن كـ DevOps/Administrator الأفضل تستخدم : journalctl -u jenkins -f

أو تشوف ملف : cat /var/log/jenkins/jenkins.log

---

8. Plugins

"plugins for adding, removing, disabling"

◆ **Plugins = قلب Jenkins**

- من غيرها Jenkins هيبقى مجرد scheduler بسيط.
- كل حاجة من Git integration لـ Docker لـ Kubernetes بيتم بالـ plugins

📌 أمر عملي لإدارة plugins من CLI : jenkins-plugin-cli --plugins git:latest docker-workflow:latest

---

9. Themes (UI customization)

"dark theme plugin ... darker UI ... show pipeline graphs"

◆ من الـ plugins اللي ركبناها Dark Theme plugin :

- بتقدر تغير شكل Jenkins من Dark ← Light
- كمان تقدر تعمل enable لعرض pipeline graphs بشكل واضح.

مفاهيم إضافية Best Practices

1. System Messages : استخدمها كـ communication channel مع الـ developers
2. Executors : متزودهاش قوي على نفس السيرفر ← هتخنق الـ resources .. والأحسن تضيف Agents
3. Plugins : متسطبش Plugins كتير من غير داعي ← بيزودوا attack surface + بطء.
4. Logs : دايمًا اربط Jenkins مع Logging system (ELK, Grafana Loki)
5. Themes : مهمة بس تجميلية، مش أساسية في الإنتاج.

الملخص

- بعد ما تثبت Jenkins وتدخل على الـ UI ← بتبدأ تعرف النسخة والمكونات.
- تقدر تتحكم في Jenkins من Manage Jenkins → Dashboard
- أهم صفحات Logs : System Info (vars, plugins) & System Config (home dir, executors, URL)
- Jenkins يعتمد بشكل كبير على Plugins ، ودي اللي بتدي له القوة.
- ممكن تخصص الـ UI (dark theme) عشان يبقى أسهل في الاستخدام.

بكده عندك صورة كاملة عن واجهة Jenkins من أول login لحد إدارة النظام والـ plugins

System Settings

المقدمة

الجزء ده بيركز على:

- إعدادات النظام (System Settings) في Jenkins
- خيارات زي Home Directory, System Message, Executors, Labels, Usage Policy, Project Naming, Jenkins URL, System Admin Email.
- الهدف إنك تتحكم في الـ Jenkins Master Node وتخلي الإعدادات أوضح وأسهل للمستخدمين.

شرح تفصيلي

Home Directory

- Jenkins بيخزن فيه كل الداتا زي : Jobs, Plugins, Configurations, Build history
- بيكون غالبًا في:
- /var/lib/jenkins
- لو عايز تتأكد من مكانه: echo $JENKINS_HOME
- Best Practice : خليه على Partition منفصلة أو اعمله Backup بشكل دوري (زي rsync أو snapshot على VM)

System Message

- أي رسالة بتكتبها هنا بتظهر على الصفحة الرئيسية لكل اللي بيدخل Jenkins
- مفيدة جدًا في:
  - تنبيه المستخدمين عن صيانة أو Downtime
  - عرض Contact Info بتاع الـ Admin

أمر عملي : لو عايز تضيف رسالة مباشرة من CLI :

jenkins-cli set-system-message " 🚨 Maintenance tonight at 11 PM"

Executors

- عدد الـ "Workers" اللي Jenkins Master يقدر يشغلهم في نفس الوقت.
- مثال : لو عندك 2 Executors ← Jenkins يقدر يشغل 2 Jobs في نفس الوقت.
- Best Practice :
  - خلى Executors قليلة على Master (حتى 0) ← وسيب كل التنفيذ على Agents/Slaves
  - ده بيحافظ على Stability بتاع الـ Master

Labels

- Labels بتستخدم لتصنيف Nodes (Agents)
- مثال عملي:
  - Agent عنده Java 11 ← هتسميه 11 java
  - Agent تاني عنده Docker ← تسميه docker
- لما تيجي تبني Job : تختار إنه يروح يشتغل على Agent بـ Label معين.
- مفيد جدًا في الـ CI/CD عشان توزع الأحمال صح.

🔹 Usage Policy

- خيار بيحدد Jenkins يستخدم الـ Node إزاي.
- أشهر اختيار:
  - "Use this node as much as possible" ← Jenkins يوزع الـ Jobs بشكل حر.
  - تقدر برضو تعمل Restriction بحيث تستخدم Node بس لما تتطلب Label معينة.

🔹 Project Naming Strategy

- تقدر تحدد Pattern للأسماء اللي المستخدمين يكتبوا بيها اسم الـ Project
- مثال : تمنع أي اسم فيه مسافات أو رموز غريبة.
- ممكن تستخدم Regex زي: ^[a-zA-Z0-9_-]+$
- ده يحافظ على Organization ونظافة الـ Jenkins Dashboard

🔹 Jenkins URL

- من أهم الإعدادات !
- URL لازم يكون:
  - بسيط وسهل التذكر.
  - دايمًا يفضل تستخدم HTTPS مش HTTP
- مثال جيد :
- https://jenkins.company.com
- مثال سيء (معقد وصعب يتذكر) :

https://10.0.0.1:8080/jenkins/app/dashboard/main

- أمر عملي لتفعيل HTTPS في Jenkins :
  - تضيف Keystore :

```
keytool -genkey -keyalg RSA -alias jenkins -keystore jenkins.jks -keysize 2048
```

  - بعدها تضيفه في إعدادات Jenkins config

🔹 System Admin Email

- هنا بتحدد إيميل الـ Admin عشان Jenkins يقدر يبعت Notifications (Errors, Build failures)
- بيتظبط مع SMTP Settings لما تعمل Configuration للـ Email Notification Plugin

أمر عملي (تجريب إرسال إيميل):

```
echo "Test Email from Jenkins" | mail -s "Jenkins Test" admin@example.com
```

🟣 إضافات مفيدة (Best Practices + Real Use Cases)

1. Backup Strategy
   - استخدم Plugin زي *ThinBackup* أو *JobConfigHistory*
   - أو حتى cron job بـ rsync : `rsync -avz /var/lib/jenkins /backup/jenkins_$(date +%F)`
2. Security
   - اقفل HTTP ← واستخدم HTTPS دايمًا.
   - اربط Jenkins بـ LDAP/AD للـ Authentication

3. **High Availability**
   - لو Jenkins مهم جدًا : استخدم Jenkins Controller + Multiple Agents
   - أو فكر في Jenkins on Kubernetes (Jenkins X)
4. **Documentation**
   - دايمًا سيب System Message واضح فيه : مين الـAdmin ، إزاي توصل له، أي Notes تخص المستخدمين.

الملخص

- **Home Directory** : مكان تخزين الداتا ← اعمل Backup دوري.
- **System Message** : وسيلة للتواصل مع المستخدمين.
- **Executors** : عدد الـ Jobs اللي يشتغلوا في نفس الوقت ← ومتنساش تقللها على Master
- **Labels** : تصنيف Agents حسب قدراتهم.
- **Usage Policy** : تتحكم في إزاي Jenkins يوزع الـ Jobs
- **Project Naming Strategy : Regex** ينظم أسماء المشاريع.
- **Jenkins URL** : لازم يكون HTTPS + بسيط.
- **Admin Email** : مهم للـ Notifications

❑ الصورة الكاملة : الإعدادات دي هي الـ "Foundation" لأي Jenkins Setup لو عملتها صح من البداية ← هتوفر وقت ومشاكل كتير بعدين.

Build Timeout

https://plugins.jenkins.io/build-timeout/

المقدمة

هانتكلم عن Build Timeout في Jenkins

الـ Build Timeout هو Plugin مهم جدًا لأنه بيمنع الـ Jobs إنها تفضل شغالة بشكل لا نهائي لو حصل Error أو Delay

لو ما عملتش Timeout ، ممكن تلاقي Job واحدة مسيطرة على Executor وموقفه الـ Queue كله.

شرح تفصيلي

1. Build Timeout Plugin

- هو Plugin بتسطبه في Jenkins عشان تحدد أقصى وقت مسموح للـ Job
- لو الـ Job عدى الوقت المحدد ← Jenkins يعمل Action (زي Abort أو Fail)
- معلومة مهمة:
  - Plugin ده مش بيشتغل مع Pipeline Jobs
  - لو بتستخدم Pipeline ← لازم تستخدم Step اسمه :

```groovy
timeout(time: 5, unit: 'MINUTES') {
    // build steps هنا
}
```

---

2. طرق التهيئة (3 Levels)

1. Global Configuration
   - تروح Manage Jenkins >> Configure System
   - تظبط الـ Timeout Strategy لكل Jenkins
   - ده بيأثر على كل Jobs لو ما حددتش حاجة محلية.
2. Job Level
   - تفتح الـ Job → Configure → Build Environment
   - تضيف خيار Terminate build if stuck :
   - وهنا تقدر تحط Timeout مختلف لكل Job
3. Build Step Level
   - تقدر تضيف Build Step: Run with Timeout
   - ده بيسمحلك تحدد Timeout لجزء معين من الـ Job مش كله.

---

3. مثال عملي (Sleep Job)

- عندنا Job اسمها sleep job بتعمل Sleep 300 ثانية (5 دقايق).
- عايزين نوقفها بعد 3 دقايق فقط.

خطوات:

1. تثبيت Timeout Plugin :
   - تروح Manage Jenkins >> Manage Plugins >> Available
   - تبحث عن Build Timeout Plugin :
   - تعمل Install → Restart Jenkins

   أو من CLI : jenkins-plugin-cli --plugins build-timeout
2. Global Timeout :
   - تروح Configure System
   - تبحث عن Global Build Timeout

- تختار Strategy :
  - Absolute : تحدد وقت ثابت (مثلاً 3 دقائق).
  - Deadline : تحدد وقت معين (مثلاً 11:00 PM)
  - Elastic : يحسب بناءً على متوسط وقت آخر Builds ناجحة.
  - Likely stuck / No activity : يوقف Job لو مفيش أي Activity
- مثال : نختار Absolute = 3 دقائق.

3. Timeout Actions :
   - Abort build (الأشهر)
   - Fail build : (ممكن تستخدمها لو عايز تفشل Job بشكل صريح)
   - Run additional build steps (تشغل سكربت معين بعد التوقف)
   - Write build description (مفيد عشان تكتب سبب التوقف)
   - مثال : اختار Abort build.

4. تشغيل Job :
   - تشغل الـ sleep job
   - المفروض يتوقف أوتوماتيك بعد 3 دقائق بدل 5.
   - في الـ Console Output هتلاقي رسالة زي:
   - Build was aborted due to timeout (after 3 minutes)

---

4. Pipeline Timeout (بديل للـ Plugin)

- زي ما قلنا Plugin : مش بيشتغل في Pipeline
- الحل : استخدم Step timeout

مثال:

هنا Jenkins هيوقف الـ Step بعد 3 دقائق.

```
pipeline {
    agent any
    stages {
        stage('Long Running') {
            steps {
                timeout(time: 3, unit: 'MINUTES') {
                    sh 'sleep 300'
                }
            }
        }
    }
}
```

إضافات مفيدة (Best Practices – Real Use Cases – Tricks)

1. Best Practice :
   - دايمًا حدد Timeout للـ Jobs خصوصًا لو بتتعامل مع Integration (مثلاً API call ممكن يهنج)
   - ما تسيبش Jobs مفتوحة بدون حد أقصى.
2. Real Use Case :
   - لو عندك Job بتبني Docker Image .. ساعات الـ Build ممكن يهنج بسبب Network Issue
   - بالـ Jenkins ← Timeout هيوقف الـ Job ويرجع Executors متاحة لغيرها.
3. Trick :
   - ممكن تخلي Action بعد Timeout تبعت Notification (Mail أو Slack)
   - مثال Slack notification في Pipeline :

```
timeout(time: 3, unit: 'MINUTES') {
    sh 'sleep 300'
}
catchError(buildResult: 'ABORTED', stageResult: 'ABORTED') {
    slackSend channel: '#jenkins', message: "Build aborted due to timeout!"
}
```

الملخص

- Build Timeout Plugin : بيوقف Jobs العادية لو عدت وقت معين.
- بيشتغل على 3 مستويات Global – Job – Step :
- Pipeline Jobs : تستخدم Step timeout بدل Plugin
- استراتيجيات متعددة Absolute, Deadline, Elastic, Stuck detection
- Actions: Abort – Fail – Additional Steps – Write Description
- فايدته : يمنع Jobs تفضل شغالة بلا نهاية وتستهلك Resources
- الصورة الكاملة : استخدام Timeout جزء أساسي من أي CI/CD عشان تحافظ على استقرار Jenkins

Timestamp

🎯 المقدمة

هانتكلم عن إضافة Timestamps داخل الـ Build Logs في Jenkins
الهدف من ده إنك تقدر تشوف وقت كل خطوة في اللوجز بالثواني والدقايق والساعات، وده بيساعدك جدًا في تحليل المشاكل ومعرفة الـ Performance لو أي خطوة أخدت وقت أطول من المتوقع.

📝 شرح تفصيلي

- في البداية ، المثال شغال على مشروع اسمه npm version test
الفكرة إننا لو فتحنا الكونسل لوجز .. هتلاقي إن مفيش أي توقيت ظاهر جنب الخطوات (مفيش Timestamps)
- علشان نضيف التوقيتات، بندخل على Configure للمشروع.
ومن جوه هتلاقي في Build Environment اختيار اسمه:

✅ "Add timestamps to the Console Output"

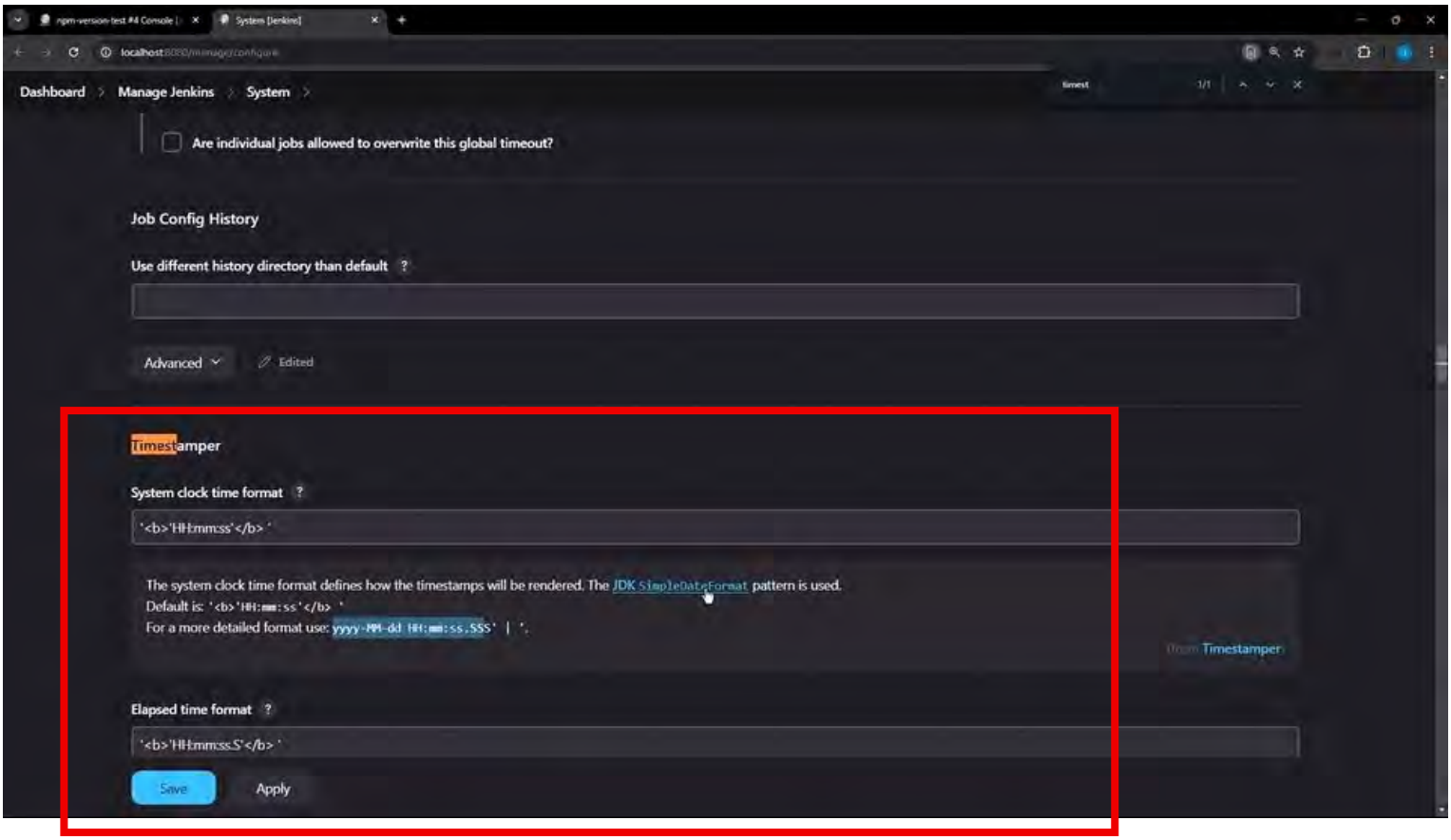
- بعد ما تفعل الاختيار ده وتعمل Save ، بتشغل الـ Job تاني.

❑ هتلاحظ دلوقتي إن الـ Console Output بقى فيه وقت بالثواني والدقايق والساعات جنب كل Log Line

---

⚙️ إعدادات أوسع(Global Settings)

- الـ Timestamps ليها إعدادات عامة تقدر تعدلها من:
ابحث عن Timestamper → Configure System → Manage Jenkins → Dashboard
- هنا هتلاقي إن التوقيت الافتراضي بيكون (ساعات:دقايق:ثواني).
- لكن في الحقيقة تقدر تغيّر الـ Format زي ما تحب.
مثلًا بدل ما يبقى "00:01:32" ، ممكن تخليه يظهر "28-09-2025 12:01:32 GMT "
- التغيير بيتم باستخدام JDK SimpleDateFormat Pattern
دي صيغة مرنة بتديك اختيارات كتير (تقدر تعرض السنة، الشهر، اليوم، الساعة، المنطقة الزمنية...).



🛠️ أمر عملي – تعديل الـ Format

لو عايز تعرض التاريخ + الوقت + التايم زون في اللوجز:

1. روح على Global Settings
2. في خانة Clock format اكتب مثلًا : yyyy-MM-dd HH:mm:ss z

- النتيجة هتبقى بالشكل ده: 2025-09-28 22:34:56 GMT+2

💡 إضافات مفيدة (Best Practices & Tricks)

- ليه أستخدم Timestamps ؟
  - عشان تقدر تعرف كل خطوة أخدت وقت قد إيه.
  - لو Build Failures بتحصل عشوائي، timestamps هتساعدك تعرف هل فيه delays أو hangs
  - لو بتقارن بين Builds مختلفة، timestamps هيساعدك تشوف لو فيه steps بقت أبطأ فجأة.
- Best Practice :
  - خليه مفعّل على مستوى Jenkins كله (Global) عشان يبقى عندك consistency في كل الـ Jobs
  - استخدم فورمات يشمل الـ Timezone ، خصوصًا لو الفريق بتاعك distributed في بلاد مختلفة.

- Real Use Case:
  - في CI/CD pipelines الكبيرة اللي فيها stages كتيرة (Build – Test – Deploy) ، timestamps بتخليك تحدد بسرعة المرحلة اللي بتستهلك وقت كبير.
  - في debugging لو job واقفة فترة طويلة في مكان ما، بتشوف بسهولة من التوقيت إمتى بدأت وأمتى اتعلقت.

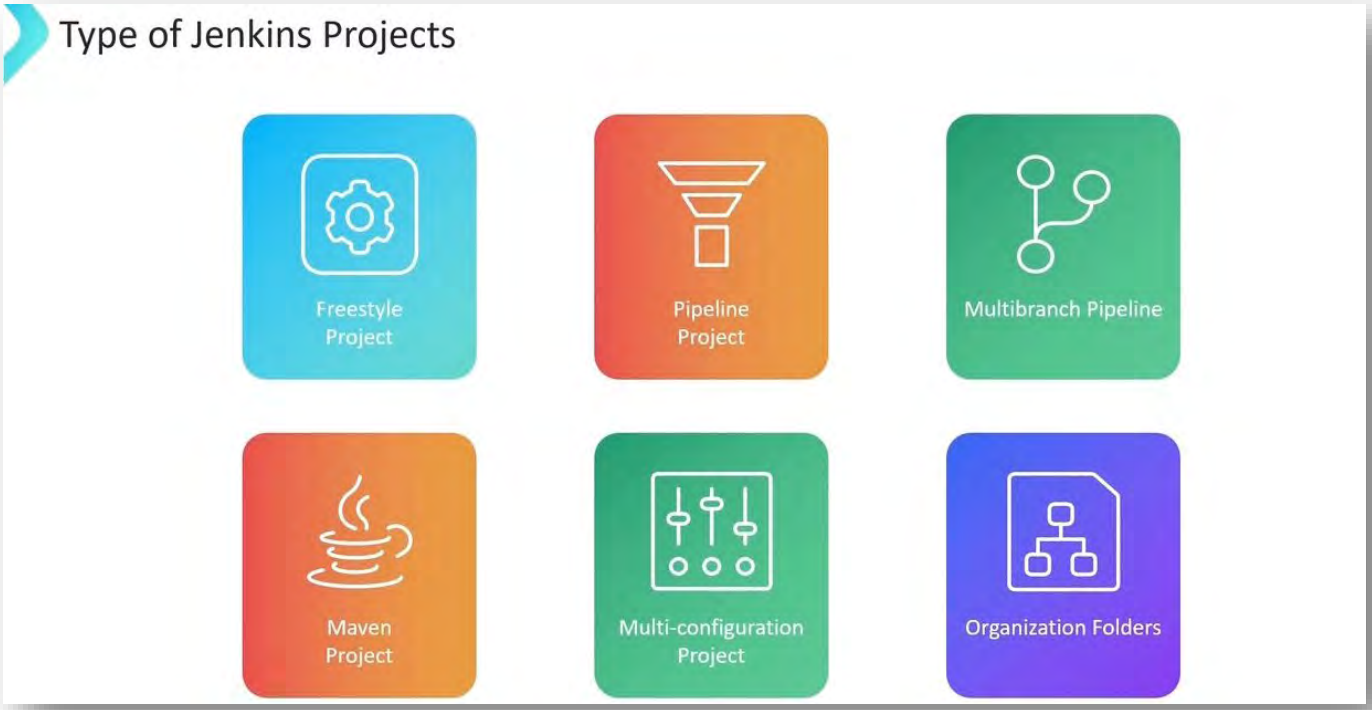
📌 الملخص

- السكريبت بيتكلم عن إضافة Timestamps للـ Console Output في Jenkins
- ممكن تفعّلها من Job Configuration أو تعملها بشكل عام من Global Settings
- تقدر تخصص الـ Format باستخدام SimpleDateFormat
- ده بيساعدك جدًا في تتبع الوقت، حل المشاكل، وتحسين الأداء.
- نصيحة : فعلها على مستوى Jenkins كله وخليها بصيغة واضحة تشمل التاريخ والـ Timezone

Jenkins Project

المقدمة

- ❖ هانتكلم عن أنواع الـ Projects (Jobs) في Jenkins
- ❖ Jenkins مش مجرد أداة CI/CD بتعمل Build & Deploy ، لكن عندها أنواع متعددة من الـ Jobs (Projects) اللي بدورها بتعمل Automate للـ Workflow على حسب طبيعة شغلك.
- ❖ الفكرة إن كل نوع ليه Use Case مختلف و مزايا وعيوب.



❑ الشرح التفصيلي

1. Freestyle Project (It's Something like To Do List)

- ده أبسط نوع من الـ Jobs
- تقدر تحدد فيه خطوات بناء (Build Steps) زي:
  - o سحب الكود من Git
  - o تشغيل Test Scripts
  - o Build للـ App
- مناسب جدًا للمشاريع الصغيرة أو التجارب السريعة.

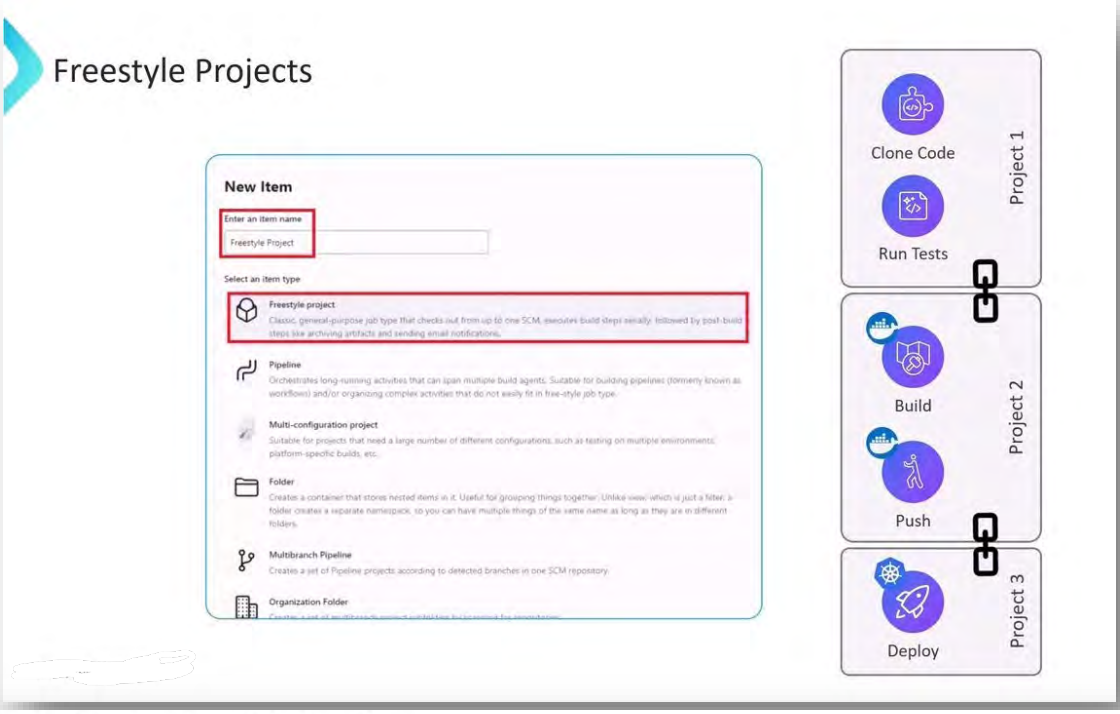
➢ عيوبه:

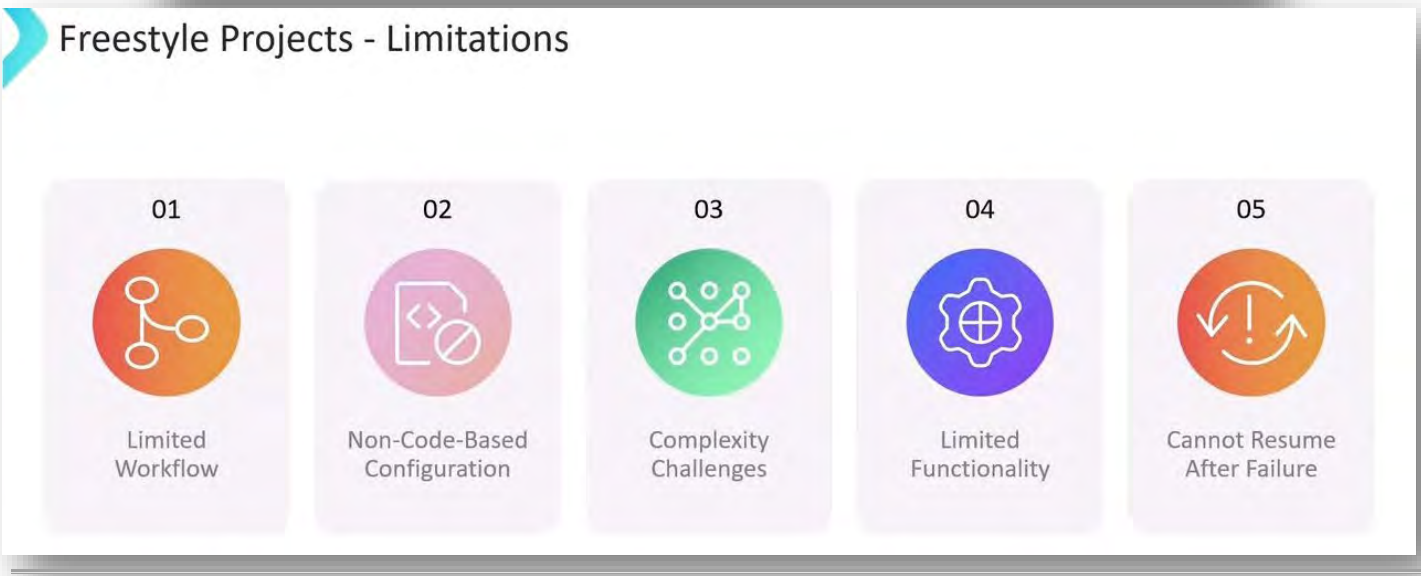
- بيشتغل بخطوات متسلسلة بس (Sequence)
- صعب تدير Workflows معقدة.
- كله بيتعمل من الـ GUI ، فـ مينفعش Infrastructure as Code

➢ مثال عملي:

لو عندك كود Java وعايز تبنيه:

1. تعمل Job جديد و تختار Freestyle
2. تحدد Git URL
3. تحط Build Command : mvn clean install
4. تحط Post-Build Action (مثلاً Send Email Notification)





2. Pipeline Project

- ده التطور الطبيعي للـ Freestyle
- بتكتب Jenkinsfile بالكود.
- بيحتوي على Stages و Steps
- مناسب للـ CI/CD Pipelines المعقدة.

➢ مثال Jenkinsfile بسيط :

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            steps {
                sh 'kubectl apply -f deployment.yaml'
            }
        }
    }
}
```

➢ مميزاته:

- كل شيء بالكود ← سهل يتخزن في Git
- مرن جدًا.
- تقدر تستخدم شروط (if/else) أو parallel stages

3. Multi-branch Pipeline

- ده بيبني على Pipeline
- بيتعامل مع Branches متعددة في Repo واحد.
- يعني لو عندك:
  - branch → dev
  - branch → staging
  - branch → main

➢ Jenkins هاتعملي detect وتبني كل واحدة بملف Jenkinsfile الخاص بيها.

➢ مثال عملي:

لو عندك GitHub repo فيه 3 branches

- تعمل Job جديد و تختار Multi-branch Pipeline
- تحدد GitHub repo
- Jenkins هيعمل Scan ويخلق Jobs أوتوماتيك.

---

4. Maven Project

- مخصص لمشاريع Java اللي بتستخدم Maven
- Jenkins بيقرأ pom.xml ويعرف الـ Build Commands من غير ما تكتب حاجة.

➢ مثال عملي:

لو عندك Java project :

1. تعمل Job جديد وتختار Maven Project
2. تحدد Git URL
3. Jenkins أوتوماتيك يشغل : mvn clean install

---

5. Multi-configuration Project (Matrix Project)

- بيسمحلك تشغل نفس الـ Build بخيارات مختلفة.
- ممتاز للـ Testing Across Environments

➢ مثال عملي:

لو عايز تختبر App على Java 8 و Java 11

- Jenkins هيعمل Matrix Builds :
  - Build + Test على Java 8
  - Build + Test على Java 11

---

6. Organization Folders

- مش Project في حد ذاته.
- ده طريقة لتنظيم الـ Jobs في شكل Hierarchy
- مثالي للشركات الكبيرة اللي عندها عشرات المشاريع.

---

7. **Legacy (Traditional Projects)**

- دي كانت الطريقة القديمة قبل Pipelines
- زي الـ Freestyle بس أكتر تعقيد.
- عيبها الأساسي:
  - صعبة الصيانة.
  - مفيش Resuming بعد Failure
  - Configuration معقد.

إضافات عملية(Best Practices)

1. دايمًا حاول تستخدم Pipeline Projects بدل Freestyle
2. لو بتتعامل مع فرق كبيرة و Branches كتيرة ← استخدم Multi-branch Pipeline
3. لمشاريع Java مع Maven ← Maven Project هيكون أسرع.
4. Multi-configuration مفيد جدًا للـ Testing Across OSs (Linux, Windows, Mac)
5. خليك فاكر إن Organization Folders بتنظم الشغل لكن مش هتديك Features جديدة.

الملخص

- Jenkins عنده كذا نوع من الـ Jobs
- Freestyle : بسيط وسهل بس محدود.
- Pipeline : مرن، بالكود، وده الأفضل دلوقتي.
- Multi-branch : مثالي لو عندك Git repo فيه أكتر من branch
- Maven Project : مخصص للـ Java + Maven
- Multi-configuration : بيساعدك تختبر بيئات متعددة.
- Organization Folders : بتنظم المشاريع مش أكتر.
- المشاريع القديمة (Legacy) موجودة لكن مش مناسبة للـ Modern DevOps

Working with Freestyle Job

📌 الدرس ده بيتكلم عن إزاي نعمل أول مشروع على Jenkins باستخدام Freestyle Job

---

1 إنشاء مشروع جديد

- أول ما تفتح Jenkins وتدخل على الـ Dashboard، هتلاقي زرار اسمه "New Item"
- من خلاله بتقدر تعمل مشروع جديد.
- بتكتب اسم المشروع اللي عايزه، وبعدين تختار نوع المشروع.
- الأنواع اللي بتظهر بتعتمد على الـ Plugins اللي عندك، لكن في حالتنا هنختار Freestyle Project (وهو أبسط وأكتر نوع مستخدم)

---

2 كتابة اسم ووصف للمشروع

- في المثال ده هانسميه Generate ASCII artwork
- وحط Description للمشروع: Generate ASCII artwork using Cowsay Library and AdviceSlip Rest API
  والوصف دا معناه إنه هيولد نصوص ASCII باستخدام مكتبة cowsay (دي مكتبة بتطبع رسائل بأشكال حيوانات في ASCII) + هيجيب الرسالة نفسها من REST API اسمه Advice Slip

---

3 إعدادات المشروع (Configuration)

المشروع فيه أقسام أساسية:

- General ← تضيف وصف للمشروع.
- Source Code Management (SCM) ← تختار منين تجيب الكود (GitHub, SVN ...) .. هنا هانختار None لأننا مش هانستخدم كود من بره.
- Build Triggers ← لو عاوز تشغل المشروع أوتوماتيك (مثلاً كل commit من GitHub) هنا سبناه فاضي)
- Build Environment ← هنا فعل Add timestamps عشان تشوف توقيت كل خطوة في الـ logs
- Build Steps ← الجزء الأهم، اللي بتكتب فيه الأوامر اللي هتتنفذ.

---

4 خطوات التنفيذ (Build Steps)

هو اختار إنه يعمل Execute shell (لأنه شغال على Linux)
كتب سكربت فيه 3 مراحل أساسية:

◆ مرحلة Build (إعداد البيانات) :

- استخدم أمر curl عشان ينادي الـ REST API بتاع Advice Slip
- بيخزن الاستجابة (JSON) في ملف اسمه advice.json
- بعدين بيستخدم jq ودي عبارة عن Library بيستخدمها عشان يقرأ الـ advice ذات نفسها من JSON .. ويحفظها في ملف تاني اسمه advice.message

◆ مرحلة Test (التحقق) :

- بيعمل اختبار بسيط : يشوف النص فيه أكتر من 5 كلمات ولا لأ. (من خلال امر ال wc اللي بيشتغل كا Counter)
- لو أقل ← يطبع رسالة ويفشل الـ build
- لو أكتر ← يكمل للمرحلة اللي بعدها.

◆ مرحلة Deploy (التنفيذ)

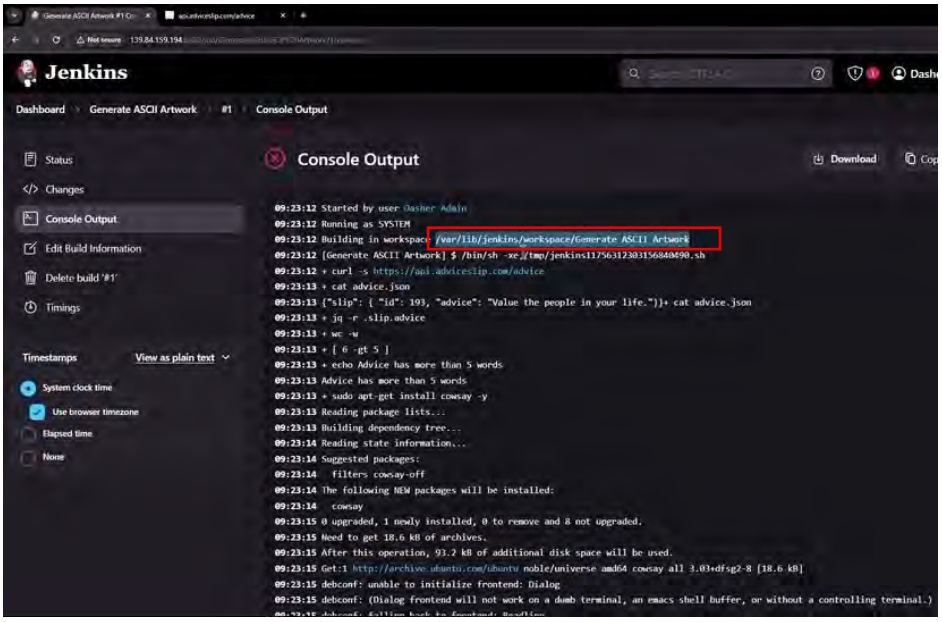
- بيعمل Install لمكتبة cowsay
- بعدين يطبع النص اللي جابه من API باستخدام cowsay ← في شكل ASCII artwork

متنساش تضيف الجزء دا هنا في السكريبت الأصلي .. هاتفهم قدام في نقطة رقم (6)

خلي بالك ان دا اللي موجود في
https://api.adviceslip.com/advice

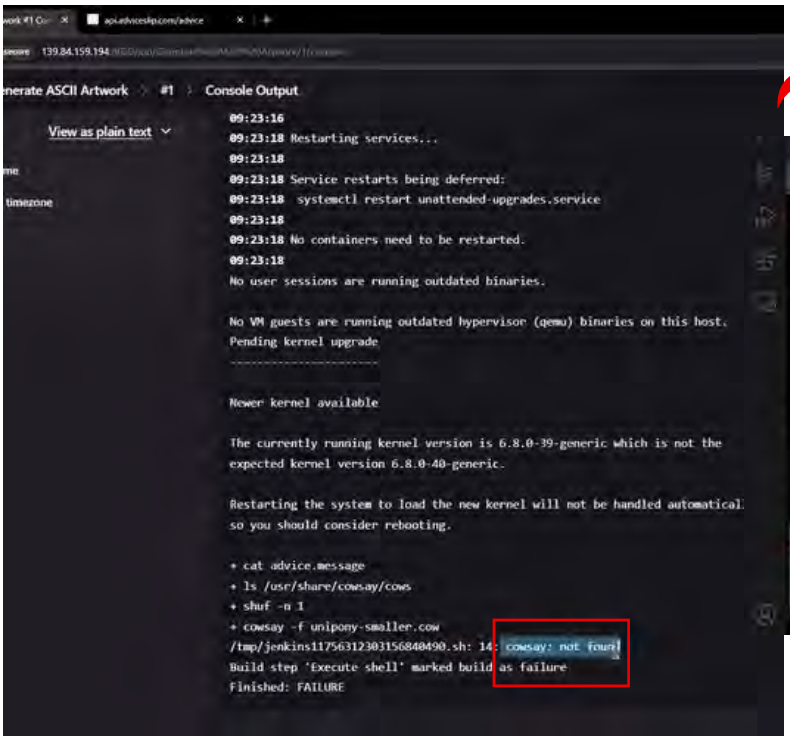
5 تشغيل المشروع

- لما عمل Build لأول مرة ← المشروع فشل. ❌
- السبب؟من الـ Console Output هانخش علي /var/lib/Jenkins/workspace/Generatic ASCII Artwork فهتلاقي ان كل حاجة تمام

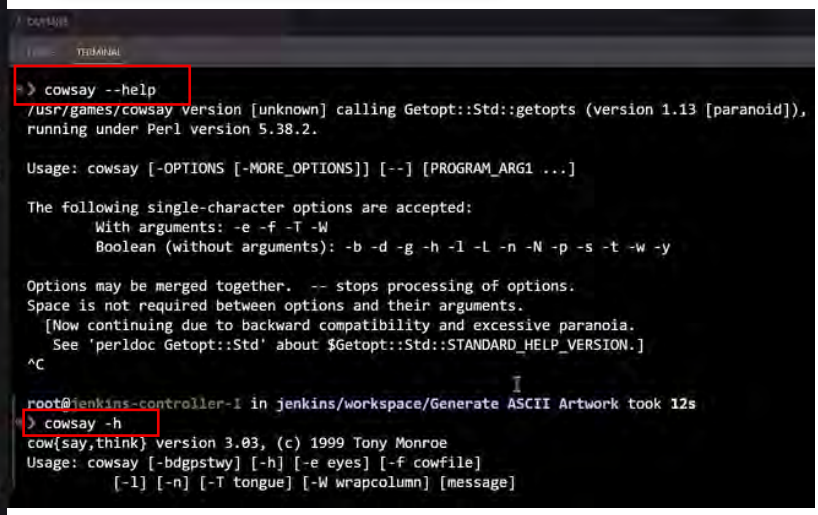



```
root@jenkins-controller-1 in ~
❯ cd /var/lib/jenkins/workspace/Generate ASCII Artwork
bash: cd: too many arguments

root@jenkins-controller-1 in ~
❯ cd /var/lib/jenkins/workspace

root@jenkins-controller-1 in lib/jenkins/workspace
❯ ls
'Generate ASCII Artwork'

root@jenkins-controller-1 in lib/jenkins/workspace
❯ cd /var/lib/jenkins/workspace/'Generate ASCII Artwork'/

root@jenkins-controller-1 in jenkins/workspace/Generate ASCII Artwork
❯ ls
advice.json  advice.message
```

- بس لو نزلت تحت شوية في الـ logs هتلاقي ان الـ cowsay: not found ودا معناه ان الـ Library متسطبة على النظام، لكن Jenkins مش قادر يوصل لها.



هتلاقيها موجودة اهوة عالنظام عادي



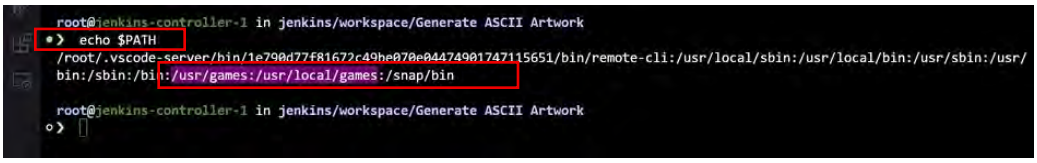
---

6 السبب والـ Fix

- ليه Jenkins مش لاقيه؟

لأن Jenkins بيشتغل كـ system service ومعندوش نفس الـ $PATH للمستخدم العادي او الـ (root)

- الحل :



  - تضيف المسار (/usr/games) مؤقتًا في الـ PATH جوه السكربت نفسه.
  - وخلي بالك انا قولتلك مؤقتا لانه كدا هايتضاف on shell level فقط .. ولو عملت أي بروجيكت تاني فلازم تضيف الـ PATH تاني في السكريبت

```
echo $PATH
export PATH="$PATH:/usr/games:/usr/local/games"
cat advice.message | cowsay -f $(ls /usr/share/cowsay/cows | shuf -n 1)
```

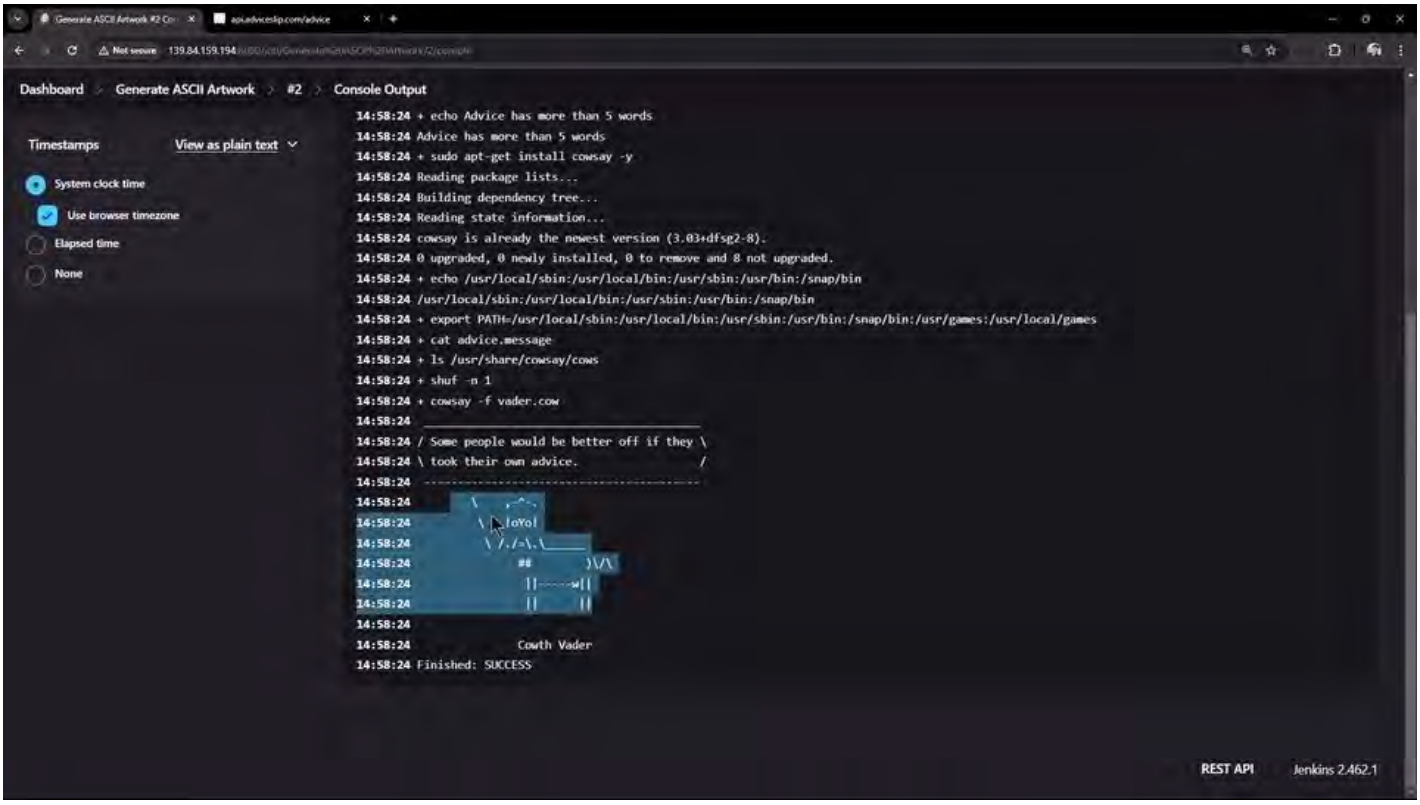
  - أو تضيفه بشكل دائم من
  - Jenkins → Manage Jenkins → System Configuration → Global Properties → Environment Variables.

- بعد ما عدلت السكربت وزودت السطر .. ضيف المسار للـ PATH ← هتلاقي الـ build اشتغل بنجاح. ✅

---

7 النتيجة

- Jenkins جاب النص من الـ API
- استخدم cowsay لطباعة الرسالة بشكل ASCII Art
- وبكده نجح أول مشروع Freestyle Job



🔥 الخلاصة :

اتعلمنا إزاي نعمل Freestyle Job، نكتب سكربت Shell فيه مراحل (Build – Test – Deploy) ، نستخدم API خارجي، ونتعامل مع مشكلة الـ PATH الخاصة بمكتبات النظام داخل Jenkins

Chained Freestyle Project

🎯 مقدمة

الشرح هنا بيتكلم عن Chained Freestyle Projects في Jenkins
يعني بدل ما كل المراحل (Build – Test – Deploy) تبقى جوا نفس الـ Job ، هنعمل لكل مرحلة Job منفصل ونربطهم مع بعض بحيث كل Job يشتغل بعد اللي قبله.

🛠️ الخطوات

1️⃣ مراجعة المشروع السابق

- قبل كده كنا عاملين Job واحد بيولد ASCII Artwork
- جوا الـ Job ده كان فيه سكربت Shell فيه 3 مراحل:
  - Build : يجيب بيانات من REST API ويخزنها في ملف.
  - Test : يتأكد إن النص فيه أكتر من 5 كلمات.
  - Deploy : يطبع الرسالة باستخدام cowsay

👉 كل ده كان بيحصل جوا Job واحد .. دلوقتي هنفصلهم.

2️⃣ إنشاء Job خاص بالـ Build

- من الـ Dashboard → New Item
- سمّاه ASCII Build Job :
- اختار Freestyle Project → OK
- انزل لجزء Build Steps ← ضيف Execute Shell
- اكتب أوامر الـ Build :
  - يعمل Call للـ REST API
  - يخزن النتيجة في ملف (مثلاً advice.json)
  - يقرأ محتوى الملف.

📂 بعد التشغيل:

- الـ Build Job نجح. ✅
- جوه الـ Workspace اتولد الملف advice.json (ملف JSON جواه الـ advice اللي رجعها الـ API)

3️⃣ إنشاء Job خاص بالـ Test

- ارجع للـ Dashboard → New Item
- سمّاه ASCII Test Job :
- انزل لـ Build Steps ← اختار Execute Shell
- اكتب أوامر الـ Test :
  - يستخدم ls عشان يتأكد إن ملف advice.json موجود.
  - يقرأ النص من JSON (باستخدام jq) ويحفظه في ملف جديد.
  - يتحقق إن النص فيه أكتر من 5 كلمات.

❑ لكن ❌ هو مش عاوز يشغل الـ Test Job يدوي.
والفكرة انه عاوزه يشتغل أوتوماتيك بعد الـ Build Job

4️⃣ ربط الـ Build Job بالـ Test Job

- افتح إعدادات ASCII Build Job ← انزل لجزء Post-build Actions
- اختار Build other projects :
- اكتب اسم المشروع ASCII Test Job :
- اختار الشرط Trigger only if build is successful:
- Apply → Save

📌 كده بقى عندنا Dependency :

- Build Job ← يشتغل الأول.
- بعد نجاحه ← يطلق Test Job

5️⃣ تشغيل السلسلة

- شغل الـ Build Job
- النتيجة:
  - الـ Build Job نجح. ✅
  - بعده مباشرةً اتشغل الـ Test Job (Downstream Project)

❑ لكن ⚠️ المشكلة:

- الـ Test Job فشل. ❌

6️⃣ ليه الـ Test Job فشل؟

- راح للـ Logs بتاعت الـ Test Job
- لقاه بيقول advice.json not found
- السبب:
  - الملف advice.json اتولد في Workspace بتاع Build Job
  - وبالتالي هاتكتشف ان كل Job في Jenkins ليه Workspace خاص بيه.
  - يعني الملفات مش بتتنقل أوتوماتيك من Job للتاني.

7️⃣ الحل

- عشان ننقل الملفات (Artifacts) من Job للتاني:
  - نستخدم Copy Artifact Plugin
- بالبلجن ده تقدر:
  - تخلي الـ Test Job ياخد الملف اللي اتولد في الـ Build Job
  - تستخدمه كأنه موجود عنده.

📌 البلجن ده هيتشرح في السيشنز القادمة.

✅ الخلاصة

- قسمنا الـ Pipeline لعدة Freestyle Jobs بدل ما يكون كله في Job واحد.
- ربطناهم باستخدام Post-build Actions (Upstream → Downstream)
- شغلنا Build ← بعد نجاحه شغل Test
- فشل الـ Test لأن الملفات مش بتتنقل بين Workspaces
- الحل : استخدام Copy Artifact Plugin

Jenkins Plugins

🎯 المقدمة

- ❖ الجزء ده بيتكلم عن Plugins في Jenkins
- ❖ الـ Plugins هي الإضافات اللي بتوسع قدرات Jenkins وتخليه يقدر يشتغل مع أدوات وتقنيات مختلفة في خط سير CI/CD
- ❖ Jenkins لوحده Engine قوي، لكن عشان يقدر يشتغل على كل احتياجاتك سواء كانت .. (Git – Maven – Cloud – Notifications...) فاهانحتاج Plugins

---

🛠️ الشرح التفصيلي

1 دور الـ Plugins

- Jenkins أساسه مجرد Engine
- الـ Plugins هي اللي بتديه القدرة على التعامل مع أدوات خارجية.
- شبه الـ Extensions في المتصفح : بتضيف Features جديدة.
- فيه أكتر من 1900 Plugin معمولين من الـ Community ، وده بيدي مرونة رهيبة إنك تبني أي Workflow يناسبك.

---

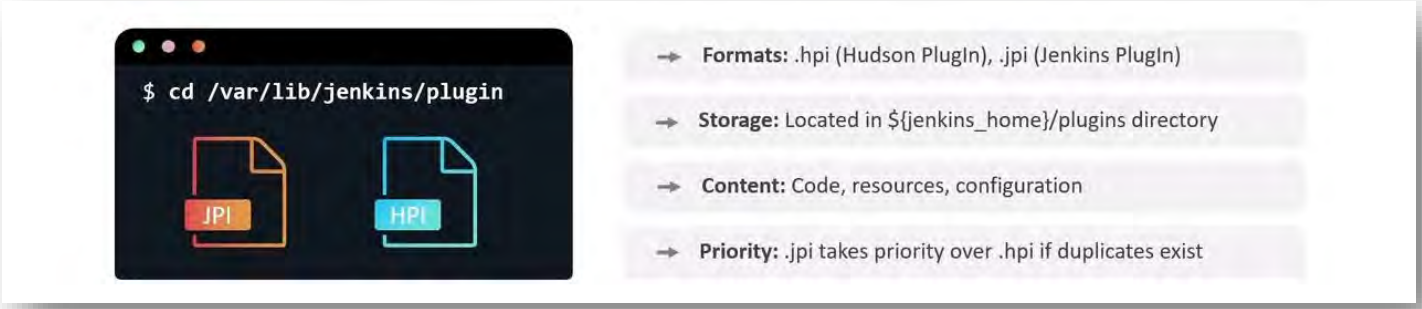
2 أمثلة على Plugins أساسية

- Source Control Management : للتعامل مع GitHub – GitLab – Bitbucket
- Build Tools : زي Maven – Gradle – Node.js
- Quality Checks : تحليل Static Code – Code Coverage
- Notifications : زي Slack – PagerDuty (عشان توصلك Alerts لو Build فشل أو خلص)
- Cloud Plugins : توصل Jenkins بـ AWS – GCP – Azure عشان Deploy أو تدير Resources
- Distributed Builds : لو عندك Worker Nodes ، تقدر توزع عليهم الشغل بالـ Plugins

---

3 تركيب الـ Plugins

- كل Plugin بيكون عبارة عن JAR File بامتداد:
  - .hpi → Hudson Plugin (قديم)
  - .jpi → Jenkins Plugin (حديث)
- بيتخزنوا في فولدر:
- $JENKINS_HOME/plugins
- الملف بيكون Archive جواه:
  - Code
  - Resources
  - Configurations
- لو فيه نسختين بنفس الـ Plugin (واحدة .hpi والتانية .jpi) .. فهاتجد ان Jenkins بياخد jpi كأولوية.



4️⃣ اختيار الـ Plugins أول مرة

- أول Setup لـ Jenkins بيعرضلك اختيارين:
  1. تختار Plugins يدوي.
  2. تختار Suggested Plugins (قائمة أساسية Jenkins بيرشحها)
- الاختيار التاني كويس لو لسه مش عارف احتياجاتك، لأنه بيغطي أساسيات زي:
  - Git.
  - Pipeline.
  - Gradle.
  - Maven.

5️⃣ إدارة الـ Plugins

- من Jenkins Management Console :
  - تقدر تعمل Enable/Disable للـ Plugins
  - تقدر تعمل Uninstall للـ Plugins اللي مش محتاجها.
- ⚠️ خلي بالك : بعض الـ Plugins معتمدة على Plugins تانية.
  - لو مسحت واحد أساسي ممكن تكسر Workflow بالكامل.
- إدارة كاملة (Install – Disable – Update) بتكون من Update Center (هتتشرح بالتفصيل في السيشن الجاية)

💡 معلومات إضافية (Best Practices + Tricks)

- دايمًا حاول تراجع Dependencies قبل ما تمسح أي Plugin
- اعمل Backup للفولدر $JENKINS_HOME/plugins قبل أي Upgrade كبير.
- لو عندك Jenkins Master–Slave setup ، تأكد إن نفس الـ Plugins موجودة ومتوافقة.
- فيه Plugins ممكن تسبب مشاكل في الـ Security أوPerformance ، فخليك حذر من أي Plugin مش مشهور أو مش معموله Update بانتظام.
- الأفضل تعتمد على أقل عدد ممكن من Plugins أساسية وتستبدل الباقي بـ Pipeline Groovy scripts لو ينفع، عشان تقلل الـ Complexity

---

✅ الخلاصة

- Jenkins لوحده Engine ، الـ Plugins هي اللي بتخليه يشتغل مع الأدوات المختلفة.
- فيه آلاف Plugins لكل حاجة زي SCM – Build – Quality – Cloud – Notifications
- بيتخزنوا كملفات hpi. أو jpi. في فولدر plugins
- Suggested Plugins خيار آمن في البداية.
- إدارة الـ Plugins بتتم من الـ Management Console والـ Update Center
- لازم تراجع Dependencies قبل أي تعديل، وتعمل Backup بشكل دوري.

Installing Plugins

🎯 المقدمة

- ❖ هانتكلم عن تنصيب وإدارة الـ Plugins في Jenkins بشكل عملي.
- ❖ الموضوع مش بس تحميل Plugin ، لكن كمان إزاي نستخدمه في الـ Jobs عشان نربطهم ببعض ونعمل الـ (Build → Test → Deploy).
- ❖ وبيعلمنا طريقتين للتنصيب :
  1. من الـ Plugin Manager UI
  2. يدويًا عن طريق تحميل ملف hpi. أو jpi.
- ❖ كمان بيعرضلك Use Case عملي باستخدام Copy Artifact Plugin، وإزاي نعمل Chained Jobs + Visualization بالـ Plugins

---

📝 الشرح التفصيلي

◆ بداية المشكلة

- عندنا Job اسمه Build Job شغال تمام وبينتج ملف (مثلاً advice.json)
- عندنا Job تاني اسمه Test Job فشل، والسبب إنه مش قادر يوصل للملف اللي اتعمل في Build Job
- الحل : نحتاج Plugin يسمح لنا ننقل الملفات بين الـ Jobs
  - ❑ وهنا بيظهر Copy Artifact Plugin

◆ تنصيب الـ Plugins

1. نروح على Manage Jenkins → Plugins
   - o من هنا نقدر : إضافة، إزالة، تعطيل، أو تحديث الـ Plugins
   - o نقدر نشوف الـ Plugins المتسطبة بالفعل.
   - o لو فيه تحديثات هتظهر هنا.
2. لو عايزين Plugin جديد :
   - o نبحث في Available Plugins
   - o نحدد Plugin (زي Copy Artifact) وبعدين نضغط Install
   - o عندنا خيارين:
     - ▪ Install مباشر.
     - ▪ Install بعد Restart

📌 Best Practice : لو Jenkins شغال Production ، يفضل نعمل Restart وقت Maintenance Window

---

◆ استخدام Copy Artifact Plugin

Copy Artifact | Jenkins plugin

❖ في Build Job :

- بعد ما يخلص، لازم نعمل Archive Artifact (Post-build Action)
  مثال : نعمل Archive للـ advice.json
- فيه Options :
  - o تقدر تختار تعمل Archive فقط لو الـ Build ناجح.
  - o ممكن تستبعد ملفات معينة.

❖ في Test Job :

- نضيف Step جديد Copy Artifact from another Project :
- نحدد:
  - اسم الـ Project اللي هنجيب منه (Build Job)
  - ناخد من آخر Build ناجح (Latest Stable Build)
  - الملف اللي عايزينه (advice.json)
- لازم Step ده يكون قبل أي Command بيستخدم الملف.

◆ إضافة Deploy Job

- نعمل Job جديد : Deploy Job
- أول Step : Copy Artifact من الـ Test Job (هنا الملف المطلوب advice.msg)
- بعدين نضيف Step تاني : Execute Shell فيه أوامر Deploy

مثال : تثبيت cowsay + استخدام الرسالة في شكل ASCII Art

➢ Build Triggers :

- نخلي Deploy Job يشتغل أوتوماتيك بعد Test Job
- الشرط : Test Job يكون Stable

📌 كده بقى عندنا :

Build Job → Test Job → Deploy Job

◆ Visualization Plugin

Yet Another Build Visualizer | Jenkins plugin

- في الآخر عرفنا ان فيه Yet Another Build Visualizer Plugin
- فايدته : يديك شكل رسومي يوضح العلاقات بين الـ Jobs (Upstream & Downstream)
- طريقتين لتنصيبه :
  - من UI (عادي)
  - Manual Upload → تحمل ملف .hpi → ترفعه في Jenkins (Manage Jenkins → Plugins → Advanced)

💡 معلومات إضافية (Best Practices + Real Use Cases)

- Best Practices :
  - دايمًا راجع Dependencies قبل ما تشيل أي Plugin
  - حافظ علي Jenkins Updated + Plugins محدثة لتجنب الثغرات.
  - اعمل Backup للـ Jenkins Home (فيه إعدادات + Plugins)
- Use Cases حقيقية:
  - نقل Artifacts بين مراحل مختلفة (Build → Test → Deploy)
  - رفع Reports (مثلاً JUnit, Coverage Reports) من Build Job لـ Test Job
  - ربط Jenkins مع Cloud Providers (AWS, Azure) أو أدوات Notifications (Slack, PagerDuty)
- Tricks :
  - ممكن تستخدم الـ CLI بدل الـ UI لتنصيب Plugins : jenkins-cli install-plugin copyartifact
  - دايمًا افحص الـ Logs لما Plugin يفشل في التنصيب.

📌 **الملخص**

- **Jenkins أساسي لكن قوته الحقيقية في الـ Plugins**
- **Copy Artifact Plugin حل مشكلة مشاركة الملفات بين الـ Jobs**
- **قدرنا نعمل Chained Jobs (Build → Test → Deploy)**
- **Visualization Plugin ساعدنا نشوف العلاقات بشكل رسومي.**
- **في الـ Production لازم نهتم بالـ Best Practices (Backup, Updates, Dependencies)**

Managing Plugins

🎯 المقدمة

- ❑ الدرس هنا بيجاوب على سؤال مهم :
  إيه اللي بيحصل لو شيلنا Plugin من Jenkins ؟؟!؟!؟!؟
- ➢ المثال بيستخدم Copy Artifact Plugin ضمن Chained Jobs
  - Build Job ← يطلع Artifact (ملف advice.json)
  - Test Job ← ينسخ الـ Artifact باستخدام الـ Plugin
  - Deploy Job ← يشتغل بعد الـ Test
- ➢ وبيوضح الفرق بين :
  - حذف الـ Binary File بتاع الـ Plugin
  - وبقاء إعدادات الـ Jobs المرتبطة بيه.

📝 الشرح التفصيلي

◆ قبل الإزالة

1. عندنا Build Job بيشتغل وبينتج ملف advice.json
   - الملف ده بيتعمله Archive
2. الـ Test Job .. أول Step فيه : Copy Artifact من Build Job
   - وده شغال كويس بسبب وجود Copy Artifact Plugin
3. ولو بصينا على Installed Plugins هنلاقي الـ Plugin متسطب (copyartifact.jpi)

◆ إلغاء تثبيت الـ Plugin

- من UI :
  - عندك خيارين : Disable مؤقتًا أو Uninstall نهائيًا.
- لما تختار Uninstall :
  - Jenkins يحذرك:
    - هيتم حذف ملف الـ Plugin من Jenkins Home
    - بس الـ Configuration Files هاتفضل زي ما هي.
- بعد Uninstall :
  - ملف copyartifact.jpi بيتمسح من /var/lib/jenkins/plugins
  - لكن Config موجود لسه جوه الـ Jobs

◆ ماذا يحدث للـ Jobs بعد الإزالة؟

1. رجعنا عملنا Build لـ Build Job
2. بعدين Test Job اشتغل ✅ ونسخ الـ Artifact بنجاح ❗
3. مع إن الـ Plugin نفسه مش موجود.

- ❑ إزاي؟
  - لأن الـ Config.xml بتاع الـ Job لسه فيه إعدادات Copy Artifact
  - Jenkins مش بيمسح Config بمجرد حذف الـ Plugin
  - فالـ Job يبان وكأنه شغال عادي.

🔹 كيف نحذف الإعدادات كمان؟

- الحل الوحيد Restart Jenkins :
- لما Jenkins يعمل Restart :
  - بيشوف إن Plugin مفقود.
  - فيلغي أي Config خاص بيه من الـ Jobs
- بالتالي أي Job كان بيعتمد على Plugin هيبطل يشتغل وهيظهر Error

---

💡 معلومات إضافية (Best Practices + Real Use Cases)

- Best Practices :
  - قبل إزالة أي Plugin لازم تعرف إيه الـ Jobs اللي معتمد عليه.
  - اعمل Backup للـ Config.xml بتاع الـ Jobs
  - في Production ، الأفضل تعمل Disable الأول وتشوف تأثيره.
- Use Cases :
  - لو حذفت Plugin بالغلط (زي Git أو Pipeline) ممكن تعطل CI/CD كامل.
  - في حالات التحديث ، لو Plugin بقى Deprecated ، ساعات الحل إنك تهاجر Config لـ Plugin أحدث.
- Tricks :
  - تقدر تعرف Dependencies بتاعت أي Plugin من Jenkins Plugin Manager
  - أو من CLI : jenkins-plugin-cli --list
  - Restart Jenkins دايمًا بيثبت التغييرات الحقيقية.

📌 الملخص

- Uninstall Plugin في Jenkins بيمسح الملف التنفيذي (JPI/HPI)
- لكن بيترك الإعدادات الخاصة بالـ Jobs
- عشان تتشال الإعدادات كمان : لازم تعمل Restart لـ Jenkins
- ده بيأكد إن أي Job معتمد على Plugin محذوف هيظهر Error بدل ما يبان شغال.
- لازم ناخد بالنا جدًا قبل ما نشيل Plugins في بيئة Production

Managing Old Data

📌 **الفكرة الأساسية :**

- ❖ في السيشن اللي فات شفنا إن حتى بعد ما نعمل Uninstall للـ Plugin من Jenkins ، بيكون لسه موجود Configurationبتاعه جوه الـ Job configs
- ❖ النهارده بنشوف إزاي نـ Purge / نحذف الكونفيج ده بالكامل.

🛠️ **الخطوات والشرح:**

1. المشكلة:
   - o بعد ما تعمل Uninstall للـ plugin (زي Copy Artifact)
   - o الـ Job configs لسه محتفظة بالإعدادات الخاصة بالـ plugin
   - o ولو رجعت ثبتت نفس الـ plugin تاني، هيشتغل بنفس الكونفيج القديم كإنه ما اتمسحش.

---

2. **Manage Old Data :**
   - o في Jenkins فيه أوبشن اسمه Manage Old Data تحت Troubleshooting
   - o الهدف منه : يوريك أي Data أو Configs قديمة مش مفهومة (Unreadable data) بسبب إن Plugin اتحذف.

---

3. **Restart Jenkins :**
   - o أول ما تروح لـ Manage Old Data بعد الـ uninstall ممكن ماتلاقيش حاجة.
   - o الحل ← لازم تعمل Restart لـ Jenkins
   - o بعد الريستارت لما تفتح Manage Old Data هتلاقي كل الـ entries اللي مرتبطة بالـ Plugin المحذوف (زي Copy Artifact)

---

4. **Discard Old Data :**
   - o Jenkins هيوريك Errors من نوع cannot resolve class exception أو configs مش متعرفة.
   - o ببساطة ← تعمل Discard unreadable data (هتلاقيها ف اخر الصفحة)
   - o كده Jenkins هيحذف الكونفيج القديم من الـ Jobs
   - o لو فتحت الـ Job config بعدها ← مش هتلاقي أي أثر للـ Copy Artifact

---

5. **الـ Behavior بعد الحذف:**
   - o قبل ما تعمل Discard : الـ Test Job كان بيقدر يلاقي ملف advice.json من Build Job حتى بعد ما مسحت الـ plugin (لأنه كان بيستخدم نفس الكونفيج القديم المخزن)
   - o بعد ما عملت Purge + مسحت Workspace القديم :
     - ▪ الـ Test Job بيفشل لأنه مش قادر ينسخ الملف (لأن الـ Plugin اللي بيعمل Copy مش موجود أصلاً)

🎯 **الملخص :**

- Uninstall Plugin من Jenkins بيشيل الـ binary (JPI file) بس بيخلي الـ Configurations جوه الـ Jobs
- وعشان تحذفهم بالكامل ← لازم :
  1. تعمل Restart لـ Jenkins
  2. تروح على Manage Old Data .. وتعمل Discard unreadable data

بكده بتضمن إن الـ Jobs مش هتحتفظ بأي إعدادات خاصة بالـ Plugin القديم.

Reload Configuration from Disk

🎯 **الهدف:**

- ❖ الـ Option ده بيُستخدم علشان تعمل Refresh لإعدادات Jenkins من الملفات على الـ Disk
- ❖ بمعنى تاني ← لو غيرت أي Config files أو نقلت Jobs بره الـ UI (من CLI مثلاً) .. Jenkins مش هيلاحظ التغيير غير لما تعمل Reload

---

🛠️ **إزاي بيشتغل:**

1. **فين تلاقيه؟**
   - o تروح Manage Jenkins → Tools and Actions → Reload Configuration from Disk

---

2. **المشكلة:**
   - o لو عدلت حاجة من بره الـ UI (زي config.xml أو نقلت Jobs بين Folders من CLI) ← التغيير مش بيبان في الـ Dashboard

---

3. **مثال عملي:**
   - o في UI كان فيه رسالة ترحيب Welcome to Jenkins CI/CD
   - o عدلناها من UI ← بقت Welcome to KodeKloud
   - o بس بعد كدا، لو دخلنا على config.xml من CLI وتحديدا من خلال cd var/lib/Jenkins ثم vim config.xml وعند الـ systemMessage .. غيّرها لـ Welcome to Dasher Technologies :
   - o النتيجة :
     - ▪ الملف على الـ Disk اتعدل.
     - ▪ لكن الـ UI لسه بيعرض الرسالة القديمة (مش متزامن).

---

4. **الحل:**
   - o تعمل Reload Configuration from Disk
   - o Jenkins هيعيد تحميل كل الـ configs من الـ Disk ← ويطبق التغييرات فوراً في الـ UI
   - o في المثال : بعد Reload ظهرت الرسالة الجديدة Welcome to Dasher Technologies :

✅ **الخلاصة:**

- استخدام Reload Config from Disk :
  - o لما تعدل Configs من بره الـ UI (زي CLI أو سكريبتات)
  - o بيخلي Jenkins يقرأ الملفات من جديد ويطبقها بدون ما تحتاج تعمل Restart كامل.

Controller Failure – Freestyle Project

🎯 **الهدف:**

نعرف إيه اللي بيحصل لو Jenkins Controller وقف فجأة (سواء Manual أو Crash) أثناء تشغيل Job من نوع Freestyle

---

🛠️ **الخطوات والتجربة العملية:**

1. **تحضير Job طويل المدة:**
   - هاتعمل Deploy Job وتضيف فيه:
   - sleep 3600

   يعني ينام لمدة ساعة كاملة (3600 ثانية) ← كدا عندنا Job طويل شغال.

---

2. **تشغيل Pipeline (Build → Test → Deploy) :**
   - اتعمل Build Job ← شغّل Test Job
   - Test Job في الأول فشل (بسبب مشكلة في التست نفسه، مش بسبب الـ Controller)
   - النتيجة : Deploy Job ما اشتغلش لأن شرطه إن Test ينجح.

🔄 بعد إعادة تشغيل الـ Build ، التست نجح، فالـ Deploy Job اتشغّل وبقى في حالة Running (Sleep 3600)

---

3. **محاكاة Controller Failure :**
   - من السيرفر اللي عليه Jenkins : sudo systemctl stop jenkins
   - النتيجة :
     - Jenkins Server وقف.
     - لما تعمل Refresh للـ UI ← "Site cannot be reached"

---

4. **إرجاع Jenkins :**
   - شغّل تاني : sudo systemctl start jenkins
   - استغرق ثواني ورجع Active
   - لما تدخل UI وتسجّل دخول ← هتلاقي إن الـ Deploy Job اللي كان شغال اتوقف وفشل.

🔴 السبب : Freestyle Jobs ما بيكملوش أوتوماتيكيا بعد ما السيرفر يرجع .. ولو الـ Controller وقف، أي Job شغال هايحصله Terminate

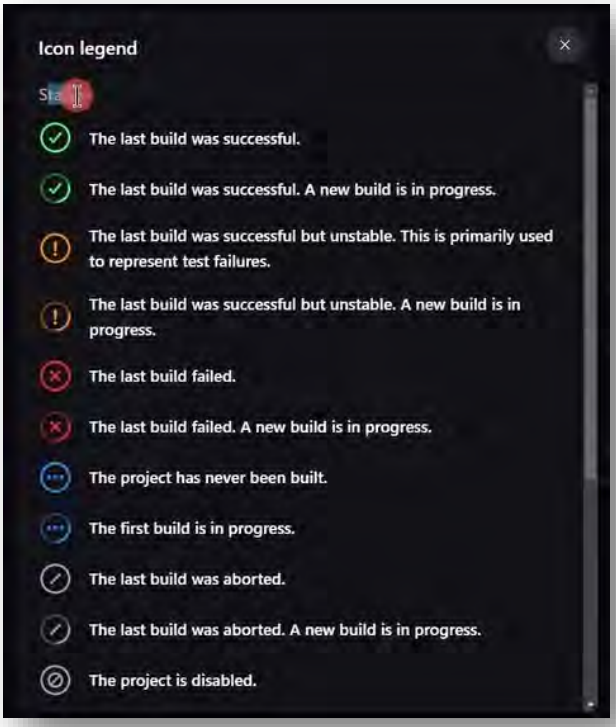
---

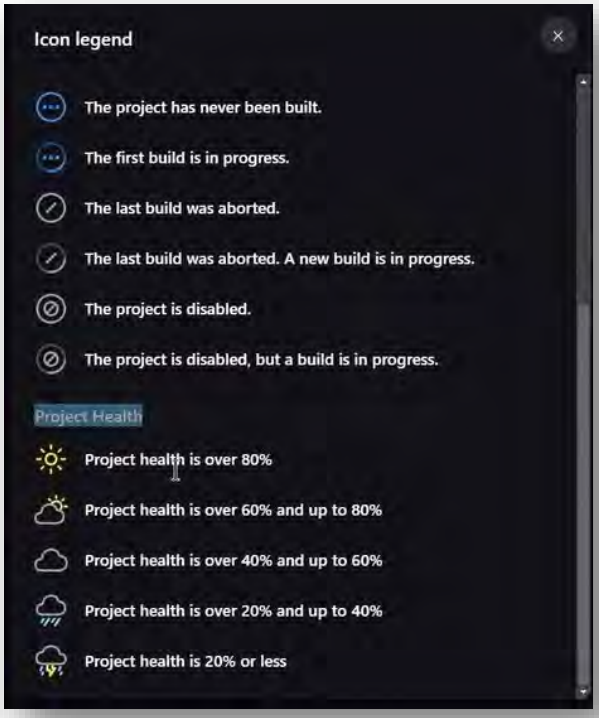
5. **مقارنة مهمة:**
   - دي مشكلة في Freestyle Projects
   - الـ Pipeline Projects عندها ميزة إنها تقدر تكمل من المكان اللي وقفت فيه أو تعمل Restart للـ Stage من جديد.
   - ودي واحدة من أسباب إن Pipelines أفضل وبتستخدم في الـ Production

---

6. **ملحوظة إضافية (Icons Legend) :**

- في Jenkins Dashboard هتلاقي Icons زي 🌞 أو ☁️.
- دول بيعبروا عن:
  - **Status** : آخر Build نجح ولا فشل.
  - **Health** : نسبة نجاح المشروع عموماً (History)
- تقدر تشوف الـ Legend من : More Actions → Icon Legend





✅ الخلاصة:

- **Freestyle Job + Controller Failure = Job** فشل ومش بيكمل.
- **الحل في بيئات Production** : استخدام **Pipeline Projects** عشان عندها **Resilience** أكتر وتقدر تستأنف أو تعيد تشغيل **Stages**
- كمان **Dashboard icons** بتديك فكرة سريعة عن حالة المشروع وصحته.

Jenkins Fingerprints

🎯 مقدمة

❖ الموضوع هنا بيتكلم عن Jenkins Fingerprints
❖ الفكرة ببساطة إن Jenkins يقدر يتتبع الملفات (Artifacts) اللي بتتولد في Jobs معينة ويشوف:

- أي نسخة من الملف اتولدت.
- أي Jobs تانية استخدمتها.
- إزاي الـ Jobs مترابطة مع بعض.

➢ وده مهم جدًا لو عندك مشاريع كتيرة مرتبطة ببعض، بحيث تعرف لو ملف اتغير، إيه اللي هيتأثر.

📖 شرح تفصيلي

❑ تعريف Fingerprints

- Jenkins بيستخدم fingerprints عشان يتتبع الملفات.
- الـ fingerprint هو MD5 checksum للملف (زي بصمة رقمية، كل ملف أو نسخة مختلفة بيبقى لها بصمة مختلفة).
- Jenkins ما بيخزنش الملف نفسه عشان ما يستهلكش مساحة، لكن بيخزن البصمة + مين استخدمها + فين.
- ممكن تشوف كل الـ fingerprints المخزنة في : $JENKINS_HOME/fingerprints

---

❑ السيناريو العملي

عندنا 3 Jobs مترابطة:

1. Build Job ← بيطلع artifact (مثلاً advice.json)
2. Test Job ← بياخد artifact من build job ويولد artifact جديد.
3. Deploy Job ← بيعتمد على artifact اللي طلع من test job

➢ يعني عندك سلسلة (Build → Test → Deploy)

❑ ازاي نفعل الـ Fingerprints

- بندخل على Configure Job
- ننزل لحد Post-build Actions
- نختار : Record fingerprints of files to track usage
- هنا بندخل أسماء الملفات اللي عايزين نتتبعها.

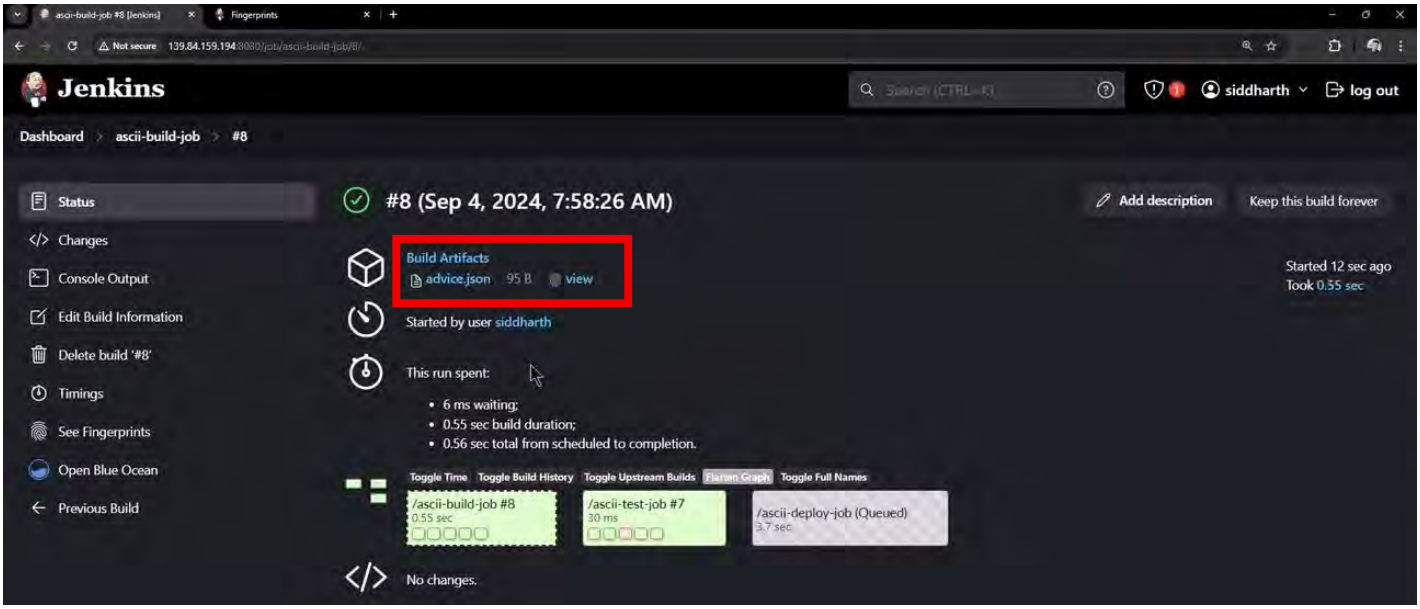
  مثال : advice.json
- نعمل Save

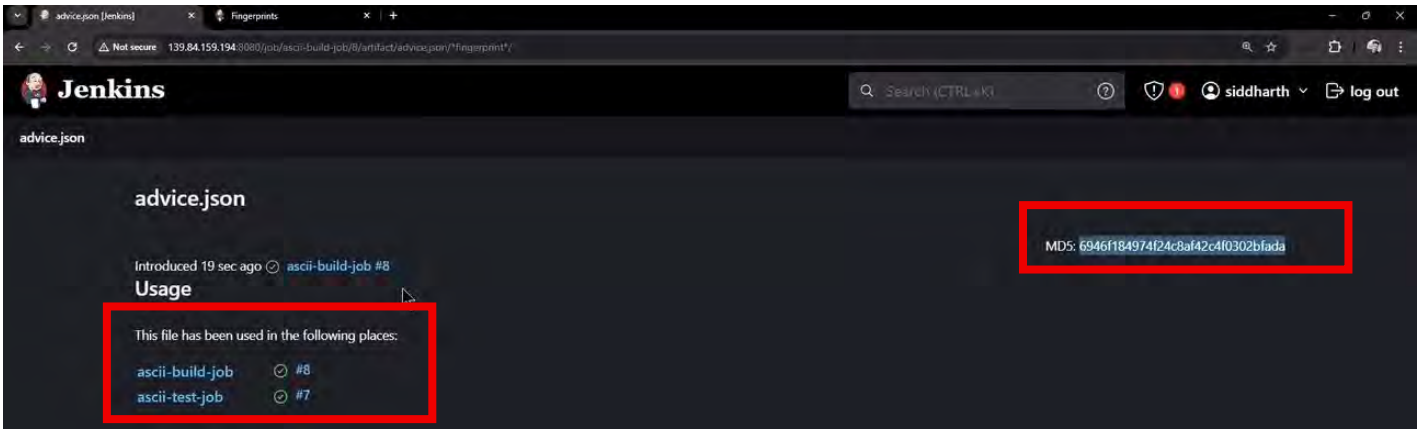
💡 معلومة :

ممكن تعمل كده لكل jobs (Build/Test/Deploy) ، أو تكتفي بواحد فيهم حسب متابعتك.

❑ التجربة العملية

- بعد ما عملنا الإعدادات، شغلنا build job
- خلص بنجاح، وولد ملف advice.json
- لما نفتح تفاصيل الـ build ده (مثلاً Build #8)
  - هنلاقي Jenkins مخزن الـ fingerprint (MD5)
  - كمان بيقولنا الملف ده استخدم فين.
  - المثال : الملف استخدم في Build Job و Test Job، مع IDs للبيلدات اللي استخدمته.





❑ طرق إضافية للتتبع

- من الـ Dashboard في Jenkins ، فيه اختيار : **Check file fingerprint**
- ترفع ملف عندك (مثلاً advice.json من جهازك)
- Jenkins يطلعلك:
  - أي Projects استخدمته.
  - أي Builds استخدمته.
  - إمتى اتولد / اتعدل.

Best Practices & Real Use Cases 💡

1. في Microservices :
   كل Service ممكن يولد artifact (JAR, JSON, Docker Image ...)
   وبالـ fingerprints تعرف أي Service اتأثر لما artifact معين اتغير.
2. التتبع أثناء الـ Debugging :
   لو حصل Bug في Production ، ممكن ترجع تشوف أي نسخة من الملف كانت مستخدمة وقتها.
3. Continuous Delivery :
   تقدر تتابع artifact من أول ما اتبنى (Build) لحد ما اتنشر (Deploy)
4. تجنب الـ Storage Issues :
   Jenkins ما بيخزنش الملفات نفسها، بس البصمة + معلومات الاستخدام ← أقل استهلاك للداتا.

📝 الملخص

- Jenkins Fingerprints = MD5 checksum للملف عشان يتتبع استخدامه.
- بيتفعل من Post-build Actions → Record fingerprints
- Jenkins يخزن:
  - بصمة الملف.
  - أي Job استخدمه.
  - أي Build IDs
- مفيد جدًا في متابعة الملفات بين Jobs مترابطة (Build → Test → Deploy)
- ميزة قوية خصوصًا في مشاريع كبيرة أو Microservices عشان تحدد dependencies والتأثيرات بسرعة.

Pipeline and Jenkinsfile

🎯 أولًا: مقدمة

- النهاردة بنبدأ مرحلة جديدة ومهمة جدًا في Jenkins وهي Jenkins Pipeline Projects وملفها الأساسي Jenkinsfile
- الفكرة إننا ننتقل من الشكل البسيط اللي كنا بنستخدمه في Freestyle Projects (اللي كل حاجة فيها linear و manual شوية) إلى شكل احترافي ومنظم أكتر باستخدام Pipeline as Code (يعني كل خطواتك في CI/CD تبقى مكتوبة في كود يمكن نسخه، مشاركته، وتتبع التعديلات عليه)

⚙️ المفهوم الأساسي

تقدر تتخيل الـ Pipeline زي وصفة طبخ (Recipe) فيها مراحل كتير:

- كل Stage = خطوة رئيسية في عملية البناء أو النشر.
- كل Step = أمر فعلي بيتنفذ (زي... build, test, deploy)

مثلاً :

- Stage اسمها ***Build*** فيها steps لتجميع الكود.
- Stage اسمها ***Test*** فيها steps لتشغيل اختبارات.
- Stage اسمها ***Deploy*** فيها steps لنشر التطبيق.

الميزة هنا إنك بتقسم كل حاجة لمراحل واضحة وسهلة تتبعها بدل ما تكون مجرد commands ورا بعض.

🧱 أنواع الـ Pipelines

1️⃣ Declarative Pipeline

- هي الأسهل والأوضح، مكتوبة بلغة قريبة للبشر.
- مناسبة جدًا للمبتدئين أو للـ workflows البسيطة والمتوسطة.
- مش بتحتاج خبرة كبيرة في البرمجة.
- مثال بسيط:

Jenkinsfile Terminal

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn build'
            }
        }
        stage('Unit Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Dockerize') {
            steps {
                sh 'docker build -t imageName:version .'
                sh 'docker push imageName:version'
            }
        }
        stage('Deploy') {
            steps {
                sh 'kubectl apply -f deployment.yaml
            }
        }
    }
}
```

💡 دي طريقة منظمة وسهلة القراءة، بس مش مرنة جدًا لما تكون عايز تعمل logic معقد.

2️⃣ Scripted Pipeline

- هنا بتكتب pipeline بالكامل بلغة Groovy
- أقوى ومرنة جدًا — تقدر تعمل loops, conditions, dynamic stages ....
- لكنها أصعب شوية وبتحتاج فهم برمجي.

مثال:

```
node {
    stage('Build') {
        echo 'Building...'
    }
    stage('Test') {
        echo 'Testing...'
    }
    stage('Deploy') {
        echo 'Deploying...'
    }
}
```

💪 دي بتديك تحكم كامل، ومناسبة جدًا لو عندك logic معقد أو automation متقدم .

الفرق بين Pipeline Projects وFreestyle Projects

Aspect	Pipelines	Freestyle
Structure	Stages allow parallel task execution	Sequential order of steps
Configuration	Configured via Jenkinsfile, supports version control	Configured via web interface, less flexible
Complexity	Ideal for complex workflows	Suitable for basic automation tasks

المقارنة	Freestyle Project	Pipeline Project
التنفيذ	خطوات ثابتة بالتسلسل	ممكن تعمل stages وparallel tasks
الإعداد	من الـ UI فقط	بالكود (Jenkinsfile)
المرونة	محدودة	عالية جدًا
التعافي بعد فشل السيرفر	بيتوقف	بيكمل تلقائي بعد الرجوع
المناسب للمشاريع	الصغيرة والبسيطة	الكبيرة أو متعددة الخطوات

مميزات Pipelines

1. **Pipeline as Code**
   كل الإعدادات في ملف Jenkinsfile داخل مشروعك ← وبالتالي تقدر تحفظه في Git وتتابع التعديلات.
2. **Resilience**
   لو السيرفر (controller) وقع، الـ Job بيكمل بعد ما السيرفر يرجع.
3. **Manual Approvals**
   ممكن توقف الـ pipeline في نقطة معينة وتستنى موافقة قبل ما تكمل (مثلاً قبل الـ deploy على production)
4. **Parallel Execution**
   تقدر تشغل اختبارات أو عمليات مختلفة في نفس الوقت لتقليل الزمن.
5. **Integrations** قوية
   بيتكامل مع باقي Plugins زي ... Docker, Kubernetes, Slack, SonarQube إلخ.
6. **Stage Visualization**
   بيعرض كل مرحلة بشكل جميل على الـ Dashboard ، فتقدر تشوف بسرعة فين المشكلة لو حصل فشل.
7. **Jobs** أقل = إدارة أسهل
   بدل ما تعمل 10 jobs منفصلة، ممكن تكتبها كلها في Jenkinsfile واحد منظم.

🧩 ملخص سريع

العنصر	المعنى
Jenkinsfile	ملف الكود اللي بيعرف الـ pipeline
Stage	مرحلة رئيسية (build, test, deploy...)
Step	أمر فعلي داخل المرحلة
Declarative Pipeline	شكل سهل القراءة للمبتدئين
Scripted Pipeline	شكل برمجي مرن ومتقدم
الميزة الكبرى	CI/CD code يمكن نسخه ومشاركته بسهولة

- ❑ عايز أقولك إن من أول الجزء ده، إحنا دخلنا في قلب Jenkins Automation الحقيقي.
- ❑ الـ Freestyle كانت مجرد مقدمة بسيطة.
- ❑ الـ Pipeline هو اللي بيخليك تشتغل زي الشركات الكبيرة — كل حاجة أوتوماتيكية ومكتوبة في كود 🔥 .

Additional pipeline Configuration

🎯 الفكرة العامة

- ❑ ده من أهم أجزاء Jenkinsfile اللي ممكن تتشرح، وده لأنه بيغطي كل المكونات الأساسية اللي لازم تكون فاهمها كويس جدًا.
- ❑ هانشرح بنية Jenkinsfile وكل المكونات (Directives) اللي بتستخدمها جوا الـ Declarative Pipeline

يعني ببساطة، بيشرحلك إزاي Jenkins بيفهم الكود اللي جوه Jenkinsfile خطوة بخطوة، وإيه معنى كل keyword فيه.

🧱 أولًا : فكرة الـ Jenkinsfile

- Jenkinsfile هو ملف نصي مكتوب بلغة قريبة من Groovy
- الهدف منه هو تعريف مراحل الـ CI/CD ككود بدل ما تكون إعدادات من واجهة Jenkins
- كل مرحلة (stage) فيه بتمثل خطوة أساسية في دورة حياة التطبيق:
  - Source Code Management ← سحب الكود من Git
  - Build ← تجميع وبناء التطبيق
  - Test ← تشغيل الاختبارات
  - Deploy ← نشر النسخة في الـ environment

⚙️ المكونات الأساسية (Syntax & Directives)

🧩 1. pipeline { }

ده الجذر الأساسي للكود، اللي جواه كل حاجة بتتكتب.
بيمثل بداية تعريف الـ Pipeline

🧩 2. agent

```
pipeline {
    agent any  // docker { image 'ubuntu:alpine' }
```

ده بيوضح Jenkins هيشغل الـ Job دي فين.
أنواع الـ Agent :

القيمة	المعنى
any	يشتغل على أي Agent متاح (حتى الـ Controller نفسه)
none	مفيش Agent افتراضي – لازم تحدد لكل stage Agent خاص بيه
`docker { image 'ubuntu:alpine' }`	يشغل الستيج جوه Docker container بالـ image المحددة

- ➢ 💡 مثال : لو كتبت : **agent any**
  يبقى الجوب هتشتغل على أي Node متاح.
- ➢ 💡 أما لو كتبت: **agent { docker { image 'ubuntu:alpine' } }**
  يبقى Jenkins هينشئ Container مؤقت من الصورة دي ويشغل الـ stage جواه.

```
Jenkinsfile                                Terminal
pipeline {
    agent any  // docker { image 'ubuntu:alpine' }

    stages {
        stage('Building') {



        }
        stage('Unit Testing') {



        }
        stage('Deployment') {



        }
    }
}
```

3. stages { }

جواها بنحط كل الـ stages اللي بتمثل مراحل الـ pipeline ..

وكل Stage عبارة عن Block صغير خاص بخطوة معينة وبيتكتب جواه الـ Steps .

4. stage ("اسم المرحلة") { }

ده العنصر اللي بيمثل خطوة معينة في خط سير الشغل.

زي مثلًا :

- وبالتالي هتلاقي ان كل Stage فيها Steps (الأوامر اللي تتنفذ فعليًا)

5. steps { }

ده الجزء اللي بيتكتب فيه الأوامر اللي Jenkins ينفذها.

ممكن تكون:

- Shell Commands (زي 'sh 'mvn clean install)
- أو DSL Commands من Jenkins نفسه (زي ...script, when, junit) .

```
Jenkinsfile                                Terminal
pipeline {
    agent any  // docker { image 'ubuntu:alpine' }

    stages {
        stage('Building') {
            agent { docker 'ubuntu:alpine' }
            steps {
                sh 'mvn clean package'
            }
        }
        stage('Unit Testing') {
            steps {
                script {
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }
        stage('Deployment') {
            when {
                expression { branch == 'main' }
            }
            steps {
                sh 'deploy.sh'
            }
        }
    }
}
```

6. script { }

- لو احتجت تعمل Logic معقد (...Loops, If conditions)
  فبتحط الكود جوه script block ، وده بيكون Groovy code

مثال:

```
script {
    for (f in files) {
        echo "Processing ${f}"
    }
}
```

7. when { }

بتتحكم إمتى Stage تتنفذ.

مثلاً:

```
stage('Deploy') {
    when {
        branch 'main'
    }
    steps {
        sh './deploy.sh'
    }
}
```

- يعني المرحلة دي هتتنفذ فقط لو الجوب شغالة على الـ main branch

8. { } environment

تستخدم لتعريف Environment Variables سواء على مستوى كل الـ Stages أو على مستوى الملف كله.

مثلاً:

```
Terminal
pipeline {
  environment {
    VAR1 = 'foo'
    VAR2 = 'bar'
  }
  stages {
        stage('Build') {
            steps {
                sh 'echo $VAR1' // result = foo
            }
        }
        stage('Test') {
            environment { VAR1 = "test" }
            steps {
                sh 'echo $VAR1'  // result = test
            }
        }
    }
```

- تقدر تستخدمها بعد كده جوه steps كـ $ ENV

---

9. { } post

ده بلوك بيتنفذ بعد ما الـ Pipeline تخلص — سواء نجحت أو فشلت.

بنستخدمه غالبًا في:

- إرسال إشعارات (Slack, Email)
- تنظيف الملفات المؤقتة
- أو تنفيذ خطوات بعد الفشل.

مثلاً :

```
Terminal
pipeline {
  // ... pipeline stages
  post {
    success {
      script {
        // Send success
notification
      }
    }
    failure {
      script {
        // Send failure
notification
      }
    }
  }
}
```

notification

```
Terminal
pipeline {
  // ... pipeline stages
  post {
    always {
      slackNotifier(
        channel: '#your-slack-channel',
        message: '${currentBuild.result}'
      )
    }
  }
}
```

Script Block

- بتستخدمه جوا طريقة الـ declarative pipeline لما تكون محتاج تعمل حاجة مش ممكن تتكتب بسهولة جوا طريقة الـ Declarative Pipeline زي :
  - كتابة Loops أو IF Condition متقدمة
  - استدعاء API Calls أو Function داخل Jenkins
  - التعامل المباشر مع متغيرات او بيانات داخل الـ Pipeline

يعني ممكن تعتبر ان من خلال الـ Scripted Block فأنت بتدمج طريقتي Scripted & Declarative Pipeline مع بعض.

script block

```
Terminal
stage('Process Files') {
  steps {
    script {
      files = ['file1.txt', 'file2.txt']
        files.each { file ->
          sh "echo Processing $file"
      }
    }
  }
}
```

When

- بنستخدمها لتحديد الظروف او الشروط اللي لاااازم تتحقق حتي يتم تنفيذ Stage معينة في الـ Pipeline
- يعني ببساطة بتخلي Jenkins يعرف امتا يشغل stage معينة وامتا يتخطاها وبالتالي بتديني إمكانية إني اقدر احدد الشروط الواجب حدوثها لتشغيل كل مرحلة
- كمان ممكن استخدمها في الـ Conditional Branching بناء علي اسم الفرع او الـ Build Result وهكذا.

```
Terminal
pipeline {
  // ... pipeline stages
  stage('Deploy') {
    when {
      expression { branch == 'main' }
    }
    // ... deployment steps
  }
}
```

---

10. { } parameters

بتخليك تمرر قيم ديناميكية وقت تشغيل الـ Pipeline

مثلاً:

```
Terminal
pipeline {
  parameters {
    string(name: 'ENV_NAME', defaultValue: 'dev',
description: 'Environment to deploy to')
  }
  // ... pipeline stages
  environment {
    ENV = params.ENV_NAME
  }
}
```

- كده تقدر تختار الـ Branch قبل التشغيل.

---

11. credentials

- بتستخدمها لو عندك بيانات حساسة (Passwords, API Keys...)
- وتقدر تستخدمها من خلال Jenkins Credentials Store

```
Terminal
pipeline {
 agent any
  stages {
   stage('Example') {
    steps {
     withCredentials([(credentialsId: 'myCredentials')
      echo "Username: $USERNAME"
     ...
```

مثلاً:

```
withCredentials([usernamePassword(credentialsId: 'git-creds', usernameVariable: 'USER', passwordVariable: 'PASS')]) {
    sh 'git push https://${USER}:${PASS}@repo-url'
}
```

---

12. input

- بتوقف تنفيذ الـ Pipeline مؤقتا وتطلب من المستخدام يتفاعل مع البيلد – مثلا يوافق او يرفض او يدخل قيمة معينة – قبل مالـ Pipeline يكمل باقي المراحل.
- وعلشان تشتغل بشكل صحيح فهي بتحتاج Pipeline Utility Steps Plugin
- مفيدة جدًا قبل النشر على production

مثال:

```
input {
    message 'Are you sure you want to deploy to Production?'
    ok 'Deploy'
}
```

13. stash & unstash

بيستخدموا لنقل الملفات بين stages أو nodes مختلفة.

- stash يخزن ملفات مؤقتًا.
- unstash يسترجعها.

مثلاً:

```
Terminal
stage('Build') {
  // ... build steps
  stash name: 'build-artifacts'
}

stage('Deploy') {
  unstash name: 'build-artifacts'
  // ... deployment steps using stashed
artifacts
}
```

14. parallel

- تشغيل stages في نفس الوقت بدل التتابع ... . بس خلي بالك ان الـ Stages دي لازم ميكونوش معتمدين علي بعض (Independent) أو ان يكون اي stage معتمدة علي الـ o/p بتاع stage قبلها
- مفيد جدًا لتقليل زمن التنفيذ.

مثلاً:

```
Terminal
pipeline {
  parallel {
    stage('Unit Testing') {
      // ... Unit test steps
    }
    stage('Vulnerability Testing') {
      // ... Vulnerability test steps
    }
  }
  // ... subsequent stages (optional)
}
```

الخلاصة :

- ملف Jenkinsfile هو اللي بيحوّل Jenkins من مجرد Tool لتشغيل Jobs إلى منصة CI/CD كاملة ومكتوبة بالكود.
- كل directive فيه ليه وظيفة بتخليك تتحكم في كل تفصيلة من عملية البناء والنشر.

العنصر	الوظيفة
pipeline	بداية تعريف الـPipeline
agent	بيحدد مكان التنفيذ
stages	يحتوي كل مراحل الـpipeline
stage	مرحلة معينة(Build/Test/Deploy...)
steps	الأوامر داخل المرحلة
environment	متغيرات بيئية
post	خطوات بعد التنفيذ
when	تشغيل مشروط لمرحلة
script	كود Groovy متقدم
input	انتظار تأكيد المستخدم
credentials	إدارة البيانات الحساسة
stash/unstash	تخزين واسترجاع الملفات
parallel	تنفيذ مراحل بالتوازي

Simple Pipeline Job

🎯 الفكرة العامة:

هانتكلم عن إزاي تنشئ أول Pipeline Job بسيط في Jenkins ، خطوة بخطوة، وتبدأ تفهم مبدأ الـ Declarative Pipeline، وكمان إزاي تربطه بـ Maven tool جوه Jenkins وتتعامل مع إعداداته.

🧩 أولاً: إنشاء Pipeline جديد

- بندخل على Jenkins Dashboard ، نختار New Item
- بنسمي الـ job اسم بسيط زي "Hello World Pipeline"
- بنختار نوع المشروع Pipeline ونضغط OK
- جوه الصفحة دي، عندنا Tabs زي:
  - General
  - Advanced
  - Project Options
  - Pipeline

❑ فيها إعدادات المشروع، والوصف، و Trigger options، وأهم تبويب فيهم هو تبويب Pipeline لأننا هنكتب فيه الكود نفسه.

⚙️ تعريف الـ Pipeline Script

- Jenkins بيديك خيارين:
  1. يا إما تكتب السكريبت بنفسك داخل Jenkins مباشرة (Pipeline script)
  2. يا إما تجيب السكريبت من GitHub أو أي Source Code Management (SCM)

❑ إحنا هنا استخدمنا الطريقة الأولى لأنها تجربة بسيطة.

🧠 بناء الـ Declarative Pipeline

- استخدمنا مثال بسيط "Hello World"
- بيكون عندنا syntax بالشكل ده:

```
pipeline {
    agent any
    stages {
        stage('Hello') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

الشرح:

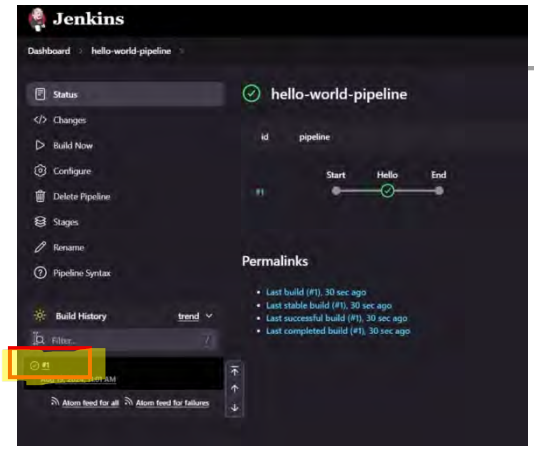
- pipeline {} : ده الـ structure الأساسي.
- agent any : معناها نفذ الخطوات دي على أي Agent متاح (وهنا هيكون الـ Controller node نفسه)
- stages {} : جواه مجموعة المراحل اللي بيتكون منها الـ pipeline
- stage('Hello') : مرحلة واحدة اسمها Hello
- steps {} : جواها الخطوات الفعلية، هنا مجرد أمر echo

🛡️ Groovy Sandbox

- Jenkins بيشغّل الـ pipeline سكريبت بلغة Groovy
- الـ Sandbox ده زي environment آمن بيمنع الكود من تنفيذ أوامر ممكن تكون خطيرة.
- لو الكود مكتوب من Admin أو مصدر موثوق، ممكن يتنفذ حتى بدون sandbox
- لكن لو المستخدم مش admin ، لازم يكون الـ Sandbox مفعّل وإلا Jenkins هيطلب موافقة Admin لتشغيله.

▶️ تشغيل الـ Pipeline

- بنضغط Build Now
- Jenkins بينفذ السكريبت ويعرض النتائج:
  - o بيظهر Stage اسمه Hello
  - o جوه اللوجز بتشوف نتيجة echo Hello World
- بيظهر كمان Visualization يوضح مراحل التنفيذ.



📊 صفحة التنفيذ (Build Details)

- فيها Tabs مهمة:
  - o Console Output : بيعرض اللوجز الكاملة.
  - o Pipeline Steps : يوضح الوقت اللي أخدته كل خطوة.
  - o Pipeline Overview : شكل رسومي للمراحل.
  - o Timing : وقت كل جزء.
  - o Workspace : الملفات اللي استخدمها أو أنشأها المشروع (وهنا فاضي).

🚧 تجربة تانية : GitHub + Maven Example

❑ بعد ما خلّصنا Hello World ، بيبدأ يورينا مثال أعقد شوية:

- Pipeline فيه integration مع GitHub و Maven
- الكود بيحاول يعمل build باستخدام Maven
  - ➢ لكن ظهر خطأ ❌

لأن Jenkins مش لاقي أداة Maven باسم m3 معرفة في إعداداته .. وتحديدا في الـ Tools

🧩 حل المشكلة – إعداد Maven Tool

- بندخل على Manage Jenkins → Tools
- هنا بنقدر نضيف أدوات زي:
  - o JDK
  - o Git
  - o Maven
- هتلاقي ان مفيش Maven متعرف ، فبنعمل "Add Maven"
  - o نديها اسم مثلاً m398 (عشان نعرف إن النسخة 3.9.8)
  - o نحدد إنها تتثبت أوتوماتيك من Apache
  - o نحط رقم النسخة 3.9.8
  - ➢ بعد كده نضغط Apply

🔗 تعديل الـ Pipeline ليستخدم الـ Tool الجديد

- نرجع للسكريبت، ونبدّل:
  - o m3 → m398
- نعمل Apply و Build Now من جديد.

✅ النتيجة

```
stage('echo version') {
    steps {
        sh 'echo print Maven version'
        sh 'mvn -version'
    }
}
```

- Jenkins أول ما يبدأ، بيعمل Tool Installation Stage :
  - بينزل Maven 3.9.8 من Apache
  - بيفكها في مسار .../lib/jenkins/tools
- بعد كده ينفذ الـ stage اللي فيها:
  (لاحظ انه ضاف الـ stage دي بنفسه في الـ script بس علشان يطبع ويوريلنا الفرق في الـ versions لما غيرها وتأكيدا علي كلامه يعني)
- النتيجة في اللوجز:
  - بيطبع"print Maven version"
  - بعدين بيعرض تفاصيل نسخة Maven المثبتة والمسار بتاعها.

🧱 الخلاصة

- كده أنشأنا أول Declarative Pipeline بسيط.
- فهمنا فكرة الـ stages والـ steps
- عرفنا يعني إيه Groovy sandbox
- وعملنا Integration بسيط مع Maven tool
- واتعلمنا إزاي نضيف tool جديد داخل Jenkins من Manage Jenkins → Tools

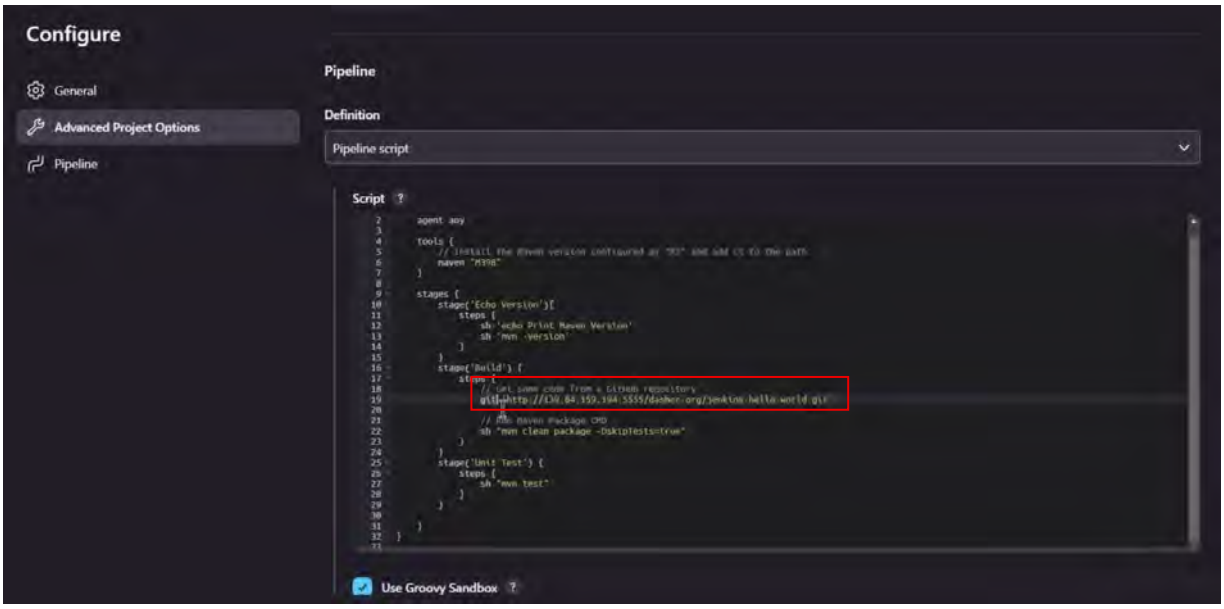
Build & Test Via Pipeline

dasher-org/jenkins-hello-world

1. 🧩 الفكرة العامة:

الـ Pipeline دي هدفها تعمل 3 مراحل:

1. Echo Version Stage : مجرد مرحلة تجريبية بتطبع إصدار الـ Maven
2. Build Stage : تسحب الكود من Git وتعمل build (باستخدام mvn clean package)
3. Unit Test Stage : تشغّل الـ test cases الخاصة بالمشروع.



---

2. 🏗️ مرحلة الـ Build :

- Jenkins بيستخدم Git Plugin عشان يسحب الكود من الـ repository
- الـ Repo موجود على سيرفر محلي (مش GitHub) .. اسمه Gitea وده self-hosted Git service (يعني زي GitHub بس على جهازك)
- الكود عبارة عن مشروع Java Spring Boot بسيط.
- عشان تعمل build : mvn clean package -DskipTests=true
  - هنا -DskipTests=true يعني ما تشغّلش الاختبارات دلوقتي.

---

3. 🧪 مرحلة الـ Test :

- في المرحلة التانية من الـ pipeline ، بنستخدم: mvn test
  - وده بيشغّل unit tests
- المشروع فيه 6 test cases :
  - 5 منهم ناجحين ✅
  - 1 فاشل ❌ (بيفشل لأنه متوقع "hola" بدل"hello")

---

4. ⚙️ المشكلة اللي حصلت:

- ❑ لما Jenkins حاول يسحب الكود من Git ، فشل في البداية لأنه كان بيحاول يسحب من فرع اسمه master ، بينما المشروع فعليًا على فرع اسمه main
- ❑ الحل كان ببساطة تعديل إعداد الـ git command في الـ pipeline كده:

```
git branch: 'main', url: 'http://<your_repo_url>'
```

5. 🧰 استخدام Snippet Generator :

❑ في Jenkins فيه أداة اسمها Pipeline Syntax Generator تساعدك تولّد الأكواد الجاهزة لأي خطوة بدل ما تكتبها من الصفر.

- هو استخدمها عشان يولّد سطر الـ git الصحيح (بفرع main)

---

6. 🧩 تصليح المشكلة اللي ظهرت في مرحلة الـ Unit Test :

لما فشل اختبار واحد بسبب الكلمة "hola" ، راح على Git وعدّل في الريبو الـ test case يخليه يتوقع "hello" بدالها.

وبعد التعديل:

- رجع شغّل الـ pipeline
- كل الـ test cases نجحوا ✅
- والـ pipeline بقت "SUCCESSFUL"

لااحظ اننا عرفنا المشكلة او بنشوف المشاكل عموما من Console Output

---

7. 🧠 النقطة الأخيرة:

- قال إننا لسه كاتبين الـ Pipeline script جوه واجهة Jenkins نفسها (GUI) ،
- لكن في المرة الجاية هينقلها إلى GitHub Repository كـ Jenkinsfile

وده الأفضل لأنه:

- بيسمح بالتعاون بين الفريق (collaboration)
- بيحتفظ بتاريخ التعديلات (history)
- أسهل في المراجعة والصيانة.

➕ ده شكل الـ Jenkins Pipeline اللي اتكلم عنه في الفيديو — مكتوبة بلغة Declarative Pipeline (اللي هي الشكل الأكثر استخدامًا في Jenkins) :

```
pipeline {
    agent any

    stages {

        stage('Echo Version') {
            steps {
                echo "Building Project with Maven Version:"
                sh 'mvn -v'
            }
        }

        stage('Build') {
            steps {
                git branch: 'main', url: 'http://<YOUR_GITEA_URL>/<repo_name>.git'
                sh 'mvn clean package -DskipTests=true'
            }
        }

        stage('Unit Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
}

    post {
        success {
            echo '✅ Build Successful!'
        }
        failure {
            echo '❌ Build Failed!'
        }
    }
}
```

🧩 تعال نحللها سطر بسطر:

- ❑ pipeline { ... }

ده الهيكل الرئيسي لأي Jenkins Pipeline، بيضم كل المراحل والخصائص.

- ❑ agent any

يعني الـ pipeline دي ممكن تشتغل على أي Jenkins agent (متاح) سواء master node أو worker node).

لو عندك agent مخصص (مثلاً build node عليه Maven) ، ممكن تكتب : agent { label 'maven-node' }

- ❑ stages { ... }

ده الجزء اللي بيحتوي على كل المراحل (build, test, deploy ...).

🧱 المرحلة الأولى Echo Version

- ❑ دي مرحلة بسيطة عشان تتأكد إن Maven شغال كويس على الـ agent
- ❑ echo بيطبع رسالة في الـ Jenkins console log

- sh 'mvn -v' بيشغّل أمر Maven في الـ shell وبيطبع الإصدار.

⚙️ المرحلة التانية Build

- git بتسحب الكود من الـ repository الموجود في Gitea

✅ مهم : لازم تحدد الفرع الصحيح main (بدل master)

- mvn clean package -DskipTests=true
  - clean : يمسح الملفات القديمة من build السابق.
  - package : يعمل compile و packaging للكود في ملف .jar.
  - -DskipTests=true : يتخطى الاختبارات مؤقتًا.

🧪 المرحلة التالتة Unit Test

- ❑ بتشغل unit tests الخاصة بالمشروع.

- لو أي test فشل، Jenkins هيعتبر الـ stage فاشلة ❌ والـ pipeline كلها ممكن تقف.

✅ Best Practices

- خليك دايمًا تستخدم post section عشان تسجّل النتايج أو تنظف الملفات بعد التنفيذ، زي:
- الأفضل تحط الـ Jenkinsfile ده جوه Git Repo نفسه بدل ما تكتبه يدوي في واجهة Jenkins

```
post {
success {
echo 'Build Successful ✅'
}
failure {
echo 'Build Failed ❌'
}
}
```

- ❑ كده Jenkins يقرأه أوتوماتيك كل مرة تعمل push جديدة .

🧠 ملخص الفكرة كلها:

الـ pipeline دي مثال بسيط يورّيك رحلة الكود من أول ما يتسحب من Git لحد ما يتبني ويتختبر:

1. بنجهز الـ environment (Echo Version)
2. بنسحب الكود ونعمل Build (Compile + Package)
3. بنشغّل الاختبارات نتأكد إن كل حاجة شغالة تمام.

طاب ازاي تخلي Jenkins يقرأ الـ Jenkinsfile من الـ Gitea (أو أي Git server) بدل ما نكتبه يدوي جوه الـ job.

مشروح بالتفصيل اكتر في الدرس الجاي

🎯 الفكرة العامة

- ❑ بدل ما تكتب السكربت في خانة "Pipeline script" جوه Jenkins ،
- ❑ هنخليه يروح يقرأه من ملف اسمه Jenkinsfile موجود في جذر المشروع في الـ repository
- ➢ كده أي تعديل في الكود أو الـ Jenkinsfile هيتسحب تلقائي في الـ build

---

⚙️ الخطوات بالتفصيل

1. في Gitea (أو GitHub)

- افتح المشروع بتاعك اللي فيه كود الـ Maven
  وفي جذر المشروع (root) اعمل ملف جديد اسمه : Jenkinsfile
- وحط فيه السكربت اللي كنا شرحناه .. وبعدين احفظه واعمل commit + push

---

2. في Jenkins

الخطوة 1 : Create a new Pipeline job

- روح لـ Jenkins Dashboard
- اضغط New Item
- اكتب اسم المشروع (مثلاً npm-version-test)
- اختار Pipeline
- واضغط OK

الخطوة 2 : Configure pipeline to use SCM

- في صفحة الإعدادات انزل تحت لجزء Pipeline
- غيّر الاختيار من : Pipeline script from SCM To Pipeline script

---

الخطوة 3 – اختار نوع الـ SCM

- اختار Git :
- في Repository URL، حط لينك الريبو بتاعك في Gitea مثل:

http://<your-gitea-url>/<username>/<repo-name>.git

- لو الريبو private ، حدد credentials (اسم مستخدم + توكن)

🧩 الخطوة 4 – الفرع

في Branches to build حط مثلاً : */main

🧩 الخطوة 5 : Jenkinsfile path

سيبها الافتراضي : Jenkinsfile (إلا لو انت مسمي الملف حاجة تانية أو حاطه في فولدر مختلف)

🧩 الخطوة 6 Apply & Build :

- اضغط Apply
- بعدين Save
- بعدين Build Now

📃 النتيجة

```
[Pipeline] Start of Pipeline
Building Project with Maven Version:
Apache Maven 3.9.5 ...
[INFO] Building jar: target/demo-1.0.jar
[INFO] Tests run: 10, Failures: 0
✅ Build Successful!
```

💡 Bonus tip

❑ لو عايز Jenkins يعمل build كل مرة تعمل push في Gitea :

1. روح على Settings > Webhooks في Gitea
2. Add webhook جديد من النوع Jenkins (Git).
3. حط فيه لينك Jenkins server : http://<jenkins-url>/gitea-webhook/post
4. فعّلها.

➢ كده كل push هيشغل build أوتوماتيك

Pipeline Script from SCM

dasher-org/jenkins-hello-world

🎯 المقدمة – الفكرة العامة

- ❑ هانتكلم عن إزاي Jenkins يقرأ الـ pipeline script من نظام إدارة الأكواد (SCM) زي GitHub أو Gitea بدل ما تكتب السكربت يدوي داخل واجهة Jenkins
- ❑ الفكرة ببساطة إنك بدل ما تعتمد على الـ "Pipeline Script" اللي بتكتبه جوه الـ job ، بتخلي Jenkins يروح يقرأ السكربت من ملف اسمه Jenkinsfile داخل الريبو بتاع المشروع.
- ❑ وده يعتبر أفضل practice لأن كده الكود والـ pipeline بيبقوا متزامنين، وأي تحديث في المشروع بيبقى متتبع تلقائيًا.

🧱 الشرح التفصيلي خطوة بخطوة

✅ الخطوة الأولى – نقل السكربت إلى GitHub أو أي Git Server

- في الأول، كنا بنكتب الـ pipeline script يدوي داخل Jenkins UI
  - o لكن ده غلط من ناحية الإدارة والمتابعة، لأن لو الـ job اتحذفت أو Jenkins وقع، السكربت كله يضيع.
  - o كمان صعب تشارك السكربت مع التيم أو تحتفظ بتاريخ تعديلاته .
- الحل إننا نعمل New File داخل المشروع في GitHub اسمه : Jenkinsfile
- بعد كده نحط جوه الملف الكود بتاع الـ pipeline اللي كنا بنستخدمه في Jenkins

📘 ملاحظة:

الملف اسمه لازم يكون Jenkinsfile بالضبط (بـ J كبيرة) لأن Jenkins بيدور عليه بالاسم ده تلقائيًا لما يعمل build من SCM

⚙️ الخطوة الثانية – تعديل السكربت قبل حفظه

- السكربت اللي كنا بنستخدمه قبل كده كان فيه خطوة اسمها : 'git branch: 'main', url: 'https://github.com/...
  - o وده معناه إن Jenkins كان بيسحب الكود يدويًا داخل الـ pipeline
- لكن لما نستخدم خيار "Pipeline script from SCM" ، Jenkins هيعمل checkout للكود أوتوماتيك من الـ repository فبالتالي السطر ده ملوش لازمة دلوقتي.

🧩 فهنعلّق أو نمسح خطوة الـ Git pull أو clone داخل السكربت.

📜 مثال على Jenkinsfile بعد التعديل

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package -DskipTests=true'
            }
        }

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

    post {
        success {
            echo "✅ Build successful!"
        }
        failure {
            echo "❌ Build failed!"
        }
    }
}
```

🧩 الخطوة الثالثة – إعداد Jenkins

1. روح لـ Jenkins Dashboard
2. اضغط New Item
3. اكتب اسم المشروع (مثلاً Hello-World-Pipeline)
4. اختار Pipeline
5. اضغط OK

⚙️ الخطوة الرابعة – ربط الـ Pipeline بـ SCM

- انزل لآخر الإعدادات عند قسم Pipeline
- بدلاً من اختيار : Pipeline script
- اختار: Pipeline script from SCM

🧠 الخطوة الخامسة – إعداد تفاصيل الـ SCM

- اختار نوع الـ SCM ← Git
- في خانة Repository URL حط لينك الريبو اللي فيه الكود : https://github.com/username/project-name.git
- لو الريبو private ← حدد الـ credentials
- في Branches to build اكتب: */main أو اسم الفرع اللي فيه الـ Jenkinsfile
- في Script Path سيبها الافتراضي : Jenkinsfile

❑ لو الملف مش في جذر المشروع (Root) ، بتحط المسار النسبي ليه زي ci/Jenkinsfile

---

🧪 الخطوة السادسة – تنفيذ الـ Build

- اضغط Save
- بعدين اضغط Build Now

➢ هتلاحظ إن Jenkins بدأ مرحلة جديدة اسمها : Checkout SCM

➢ ودي بتتعامل أوتوماتيك مع — Git تسحب الكود من الريبو وتبدأ تنفذ الـ Jenkinsfile

---

🔍 الخطوة السابعة – فحص اللوجات

لو فتحت الـ console log ، هتلاقي Jenkins عمل الخطوات دي:

1. git fetch ← سحب التحديثات الجديدة.
2. git checkout <commit-id> ← نزل آخر commit
3. بدأ تنفيذ الـ pipeline حسب المراحل Build ، Test .... إلخ
4. طبع commit message اللي اشتغل عليه (زي "added Jenkinsfile")

❑ وده بيديلك توثيق كامل لأي نسخة من الكود تم بناؤها بالظبط.

🧰 الخطوة الثامنة – فحص الـ Workspace

- الـ workspace هو المكان اللي Jenkins بيخزن فيه الملفات اللي اشتغل عليها.
- المسار العام ليه: /var/lib/jenkins/workspace/<job-name>/
- لو دخلت هناك هتلاقي:
- src/ ← ملفات الكود المصدر.
- pom.xml ← ملف إعدادات Maven
- Jenkinsfile
- README.md
- ومجلد target/ بيتكون بعد تنفيذ mvn package وبيحتوي على:
  - الـ jar. أو war. الناتج.
  - تقارير surefire الخاصة بالاختبارات.

💡 ملاحظات إضافية (Best Practices & Tips)

- دايمًا خزن الـ Jenkinsfile في نفس الريبو مع الكود عشان يفضلوا متزامنين.
- لو عندك مشاريع كتيرة ليها build steps شبه بعض، ممكن تعمل shared library (هاتتشرح في شابتر 12) وتستوردها في Jenkinsfile
- استخدم Git webhooks عشان Jenkins يشتغل أوتوماتيك مع كل push (Continuous Integration)
- خليك حريص إن Jenkins عنده permission يقرأ من الريبو، خصوصًا لو شغال self-hosted

---

🧩 الملخص النهائي

- بدل ما تكتب السكربت في Jenkins ← حطه في ملف اسمه Jenkinsfile داخل الريبو.
- Jenkins لما تستخدم خيار Pipeline script from SCM بيعمل checkout أوتوماتيك.
- مفيش داعي لكتابة git clone داخل السكربت.
- اللوجات بتوضح الكود اللي اشتغل عليه Jenkins والـ commit ID
- تقدر تراجع الملفات من الـ workspace وتشوف ناتج الـ build والاختبارات.

Controller Failure – Pipeline Project

🎯 المقدمة

- ❑ هانتكلم عن تأثير فشل الـ Jenkins Controller أو الـ Jenkins Master على نوعين من المشاريع:
  - Freestyle Project
  - Pipeline Project
- ❑ في السابق إتكلمنا عن إزاي الـ Freestyle بيتأثر لما السيرفر بتاع Jenkins يقع (الـ Build بيتوقف ومش بيكمل بعد الريستارت)
- ❑ أما هنا فهانشرح إيه اللي بيحصل لو نفس الموقف حصل لكن على Pipeline Project
  - ➢ الفكرة كلها هنا إن Jenkins Pipeline عنده قدرة على استئناف (Resume) الـ Build اللي كان شغال قبل ما السيرفر يقع، وده اللي هنشوفه خطوة بخطوة.

🧩 الشرح التفصيلي سطر بسطر

1 إعداد الـ Pipeline

- هانستخدم Pipeline بسيط اسمه Hello World Pipeline ، وعدّل عليه علشان تخليه Long-running Job (يعني بياخد وقت في التنفيذ) وده عن طريق تعديل الـ Stage الخاصة بالـ Unit Test
  - بدل ما الـ Unit Test تشتغل فورًا، هو ضاف قبلها أمر يخليها تنتظر 60 ثانية، باستخدام:

```
    }
    stage('Unit Test') {
        steps {
            for (int i = 0; i < 60; i++) {
                echo "${i + 1}"
                sleep 1
            }
```

➢ الفكرة هنا إننا نخلي الـ Job شغالة وقت كفاية علشان نقدر نعمل Restart للـ Jenkins في النص ونشوف التأثير.

---

2 الخطأ اللي حصل في البداية

- أول محاولة لتشغيل الـ Pipeline فشلت ❌
- والخطأ كان إن الـ for loop اتحطت مباشرة داخل block الـ steps ، وده مش مسموح في Declarative Pipeline
- ليه؟
- لأن Declarative Pipeline مش بيسمح بكتابة كود Groovy خام جوا steps
  أي منطق برمجي (زي loops أو conditions) لازم يتحط جوا script block كده:

```
stage('Unit Test') {
    steps {
        script {
            for (int i = 0; i < 60; i++) {
                echo "${i + 1}"
                sleep 1
            }
```

بكررها وبشرحهالك تاني علشان تفتكر لان موضوع الـ Script Block مهم جدا

💡 Concept مهم – الـ Script Block

- الـ script block في Jenkins Pipeline بيسمحلك تكتب كود Groovy حر داخل الـ Declarative Pipeline
- وده مفيد لما تكون عايز تعمل حاجة مش بسيطة زي أمر واحد، زي loops ، conditions، variables إلخ.

➢ باختصار:

- Declarative Pipeline ← بسيط وسهل للـ CI/CD
- Script Block ← عشان تضيف منطق برمجي متقدم جوه الـ Declarative Pipeline

3 تعديل الكود وإعادة التجربة

- بعد ما حط الكود جوا {} script، عمل commit للتغيير في GitHub ، ورجع يشغّل الـ Pipeline تاني.
- النتيجة: الـ Pipeline اشتغلت بنجاح ✅
  - الـ Stage بدأت، وبعد كده الـ Unit Test Stage بدأت تنفذ الـ loop لمدة 60 ثانية زي المتوقع.

---

4 التجربة الفعلية Restart – للـ Jenkins

- في النص وهو بيشغل الـ loop (لما كانت عدّت حوالي 37 ثانية)، وقف الـ Jenkins Controller يدويًا:

sudo systemctl stop jenkins

- وبعد كده تأكد من الحالة : sudo systemctl status jenkins
  - زي ما متوقع، الـ Jenkins UI وقف عن العمل، والـ Logs اتوقفت عند السطر اللي كانت واقفة عنده.

---

5 بعد الريستارت

- بعد كده رجع شغّل Jenkins تاني: sudo systemctl start jenkins
- وراح على الـ Web UI تاني، وبعد ما عمل Refresh هاتلاحظ إن الـ Pipeline كملت من نفس النقطة اللي كانت واقفة عندها
- يعني Jenkins حفظ حالة الـ Pipeline مؤقتًا، ولما السيرفر رجع اشتغل، استأنف التنفيذ مكان ما وقف (Resume Build)

---

6 مقارنة مع Freestyle Project

- في الـ Freestyle Project ، لما السيرفر بيقع أثناء الـ Build ، كل حاجة بتتوقف نهائيًا.
  - حتى لو السيرفر رجع، الـ Build القديم مش بيتكمل، لازم تبدأ Build جديد من الأول.
- لكن في Pipeline Project، Jenkins بيقدر يحفظ حالة الـ Build في ملفات عادة داخل
- JENKINS_HOME/jobs/<job_name>/builds/<build_number>/workflow$
- ولما السيرفر يرجع، بيكمل من المكان اللي وقف عنده.

---

7 خيار إضافي في الإعدادات

- Jenkins بيديك اختيار تتحكم فيه في السلوك دا:

"Do not allow the pipeline to resume if the controller restarts"

- لو فعلت الخيار ده، الـ Pipeline مش هتكمل بعد الريستارت، وهتعتبر فاشلة.
  - لكن بشكل افتراضي الخيار ده غير مفعل، فـ Jenkins بيحاول دايمًا يكمل أي Build كان شغال.

⚙️ أوامر عملية مهمة

الغرض	الأمر
إيقاف Jenkins	`sudo systemctl stop jenkins`
تشغيل Jenkins	`sudo systemctl start jenkins`
التأكد من الحالة	`sudo systemctl status jenkins`

💡 Best Practices و Real Use Cases

- ✅ استخدم Pipelines بدل Freestyle لو شغلك Critical أو بياخد وقت كبير.
- ✅ دايمًا احفظ Jenkins data على Storage دائم (Persistent Volume) علشان لو السيرفر وقع تقدر تكمل.
- ⚙️ في البيئات الكبيرة، استخدم Distributed Build Nodes علشان حتى لو Controller وقع، الـ Agent يقدر يكمل تنفيذ بعض الـ steps
- 🧠 استخدم Checkpoint Plugin في Jenkins علشان تعمل save لنقطة معينة في الـ Pipeline وتكمل منها لاحقًا لو حصل Failure

---

📜 الملخص

❑ كدا نكون وضّحنا الفرق العملي بين:

- Freestyle Projects ← بتفشل بالكامل لو السيرفر وقع.
- Pipeline Projects ← بتكمل التنفيذ تلقائيًا بعد ما Jenkins يرجع يشتغل.

❑ الفكرة الأساسية إن Pipeline مبني على Groovy + Workflow Engine بيخزن الحالة، وده اللي بيديه المرونة إنه يعمل Resume بعد أي Restart أو Crash

Create Pipeline using Blue Ocean Graphical Editor

🎯 المقدمة

- ❑ هانتكلم عن استخدام Blue Ocean Plugin في Jenkins لإنشاء Pipeline بشكل جرافيكي (GUI) بدل الطريقة الكلاسيكية اللي كنا بنكتب فيها الكود يدوي في Jenkinsfile
- ❑ الفكرة الأساسية إن Blue Ocean بيخلي التعامل مع Pipelines سهل جدًا من خلال واجهة فيها Visualization لكل مراحل الـ CI/CD (Build – Test – Deploy)

🧩 الشرح التفصيلي خطوة بخطوة

1 تنصيب Blue Ocean Plugin

- أول خطوة : بنفتح Jenkins → Manage Jenkins → Manage Plugins → Available tab
- بنبحث عن Plugin اسمه Blue Ocean ونقوم بعمل Install

◆ Blue Ocean مش مجرد Plugin واحد، هو مجموعة Plugins بتغير واجهة Jenkins كلها علشان تبقى أكتر وضوح وسهولة. وبتركز على User Experience في CI/CD وبتخلي التعامل مع Pipelines تفاعلي جدًا.

---

2 بعد التنصيب

- بعد ما تعمل Install ، لما ترجع للـ Dashboard هتلاقي اختيار جديد اسمه : Open Blue Ocean عالشمال وده بيفتحك الواجهة الجديدة اللي فيها Pipelines مرسومة ومقسّمة بشكل بصري.

---

3 إنشاء Pipeline جديدة

- بنختار New Pipeline
- Jenkins بيطلب منك تربط Blue Ocean مع Source Control (زي GitHub – Bitbucket – GitLab ... إلخ)

➢ في المثال ده استخدم GitHub، فبيطلب:

- Repository URL
- Credentials علشان Jenkins يقدر يتصل بالمخزن (ويعمل Pull و Push لو محتاج)

➢ الأوامر الفعلية اللي ممكن نستخدمها في السيرفر:

- مسار Plugins في Jenkins : `cd /var/lib/jenkins/plugins`
- لو عاوز تشوف الـ Plugin نزل `ls | grep blueocean`

---

4 أول Run للـ Pipeline

بعد الربط بـ GitHub :

- Jenkins بيدور على كل الـ Branches
- أي Branch فيها Jenkinsfile ← بيعمل لها Pipeline تلقائي.
- لو مفيش Jenkinsfile ← Blue Ocean بيسألك تعمل واحد جديد من الواجهة نفسها ؟!

---

5 تنفيذ Pipeline

➢ لو المشروع فيه Jenkinsfile أصلاً، Blue Ocean بيعرض المراحل (Stages) بالشكل الجديد.

مثلاً مراحل زي:

- Echo Version
- Build (بيشغل Maven)
- Test (بيعمل Unit Tests وينتظر 60 ثانية)

➢ كل Stage بتظهر في شكل مربع متصل بالسابق واللاحق، وتقدر تضغط عليه تشوف الـ Logs

✅ تقدر كمان:

- تشوف مدة كل مرحلة
- تعرف الـ Commit ID
- تتابع التغييرات اللي حصلت في GitHub

---

6 Multi-Branch Pipeline

➢ لما بيربط المشروع بـ GitHub ، Jenkins بيحوّله لـ Multi-Branch Pipeline

📘 المفهوم هنا:

- Jenkins بيعمل Scan دوري على الريبو.
- لو لقى أي Branch فيها Jenkinsfile ← يعملها Pipeline أوتوماتيك.
- يعني كل Branch بيتبني ويتختبر لوحده.

دي ميزة ضخمة في الـ CI لأنها بتخلي كل Feature Branch يتجرب قبل الدمج.

◆ أمر عملي:

- عايز تشوف الـ Workspaces اللي اتعملت لكل Branch : `ls /var/lib/jenkins/workspace/`

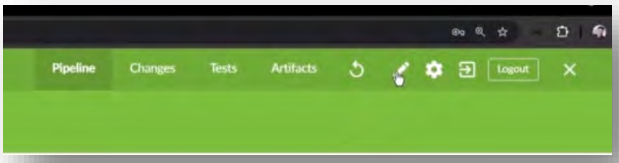
---

7 تعديل Pipeline من داخل Blue Ocean

تقدر تضغط على Edit (رمز القلم) وتغير الأوامر في كل Stage مباشرة من الواجهة.

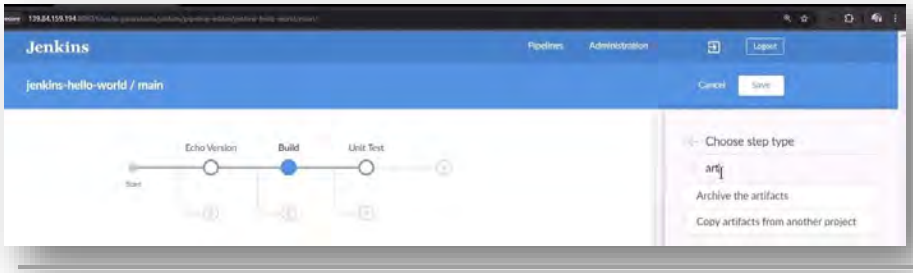
مثلاً:

- تضيف Step جديد .
- تحذف Loop أو Script
- تضغط Ctrl + S علشان تحفظ ← Jenkins بيولّد Jenkinsfile جديد ويعمل Commit في GitHub تلقائي.



8 إضافة Archive للـ Artifacts

- ❖ بعد مرحلة الـ Build ، بنضيف Step جديد من نوع : Archive Artifacts
  - o وده بيحتفظ بالملفات الناتجة من الـ Build (زي jar. أو war.)
- ❖ الأمر العملي للتحقق:
  - o cd /var/lib/jenkins/workspace/Jenkins_Hello_World_main/target
  - o ls
- ❖ بنلاقي الملف مثلاً: hello-demo-1.0.jar
  - o ففي GUI بنكتب في خانة الـ Artifact Path : target/hello-demo-*.jar وعرفناه من امر الـ cd
- ➢ لاحظ استخدام * كـ Wildcard لأي Version



---

9 تفعيل JUnit Reports

- علشان Jenkins يعرض نتيجة الـ Unit Tests ، بنضيف Step جديد : Archive JUnit formatted test results
- والمسار: target/surefire-reports/TEST-*.xml
  - o ده بيخلي Jenkins يعرض Test Summary + Graph يوضح Pass/Fail Trends

◆ التأكد من وجود Plugin :

- من Dashboard → Manage Jenkins → Plugins → Installed → Search "JUnit"

```
stage('Unit Test') {
  steps {
    sh 'mvn test'
    junit(testResults: 'target/surefire-reports/TEST-*.xml', keepProperties: true, keepTestNames: true)
  }
}
```

🔁 بعد التعديلات

كل تعديل بيتعمل في الواجهة بيتسجل في GitHub :

- Jenkins بيعمل Commit بالـ Message اللي كتبته.
- Pipeline بتشتغل أوتوماتيك بعد كل Commit

🧠 Best Practices + Real Use Cases

- دايمًا استخدم Multi-Branch Pipelines بدل Classic pipelines
- استخدم Blue Ocean في بيئات الـ Development أو Testing لأنها بتسهل المراجعة.
- خزن الـ Credentials في Jenkins Credentials Store بدل ما تكتبها في Jenkinsfile
- خليك دايمًا تعمل Archive Artifacts علشان تقدر ترجع تبص على الـ Outputs لأي Build قديم.

📋 الملخص النهائي

- Blue Ocean = واجهة جديدة في Jenkins تسهّل التعامل مع الـ Pipelines
- بتربط Jenkins بـ GitHub وبتتعامل مع الـ Jenkinsfile تلقائيًا.

- بتخليك تشوف الـ Stages بشكل Visual وتتفاعل معاها (تشغل، تعدل، تعمل Commit)
- Multi-Branch Pipeline بتخلي Jenkins يبني كل Branch لوحده.
- Archive Artifacts بيخزن نواتج الـ Build
- JUnit Plugin بيعرض نتائج الاختبارات في شكل Graph
- أي تعديل من الـ UI بيت Commitأوتوماتيك في الريبو.

Create Pipeline with Parameters

🎯 الفكرة العامة :

- ❖ في Jenkins ممكن تعمل (pipeline) job مش ثابت، إنما بياخد Parameters من المستخدم زي اسم الـ branch أو رقم الـ port أو مدة sleep قبل الاختبار...
- ❖ يعني بدل ما تعمل pipeline لكل فرع من المشروع (main & test) فتعمل واحد بس وتخليه ديناميكي بالـ parameters

---

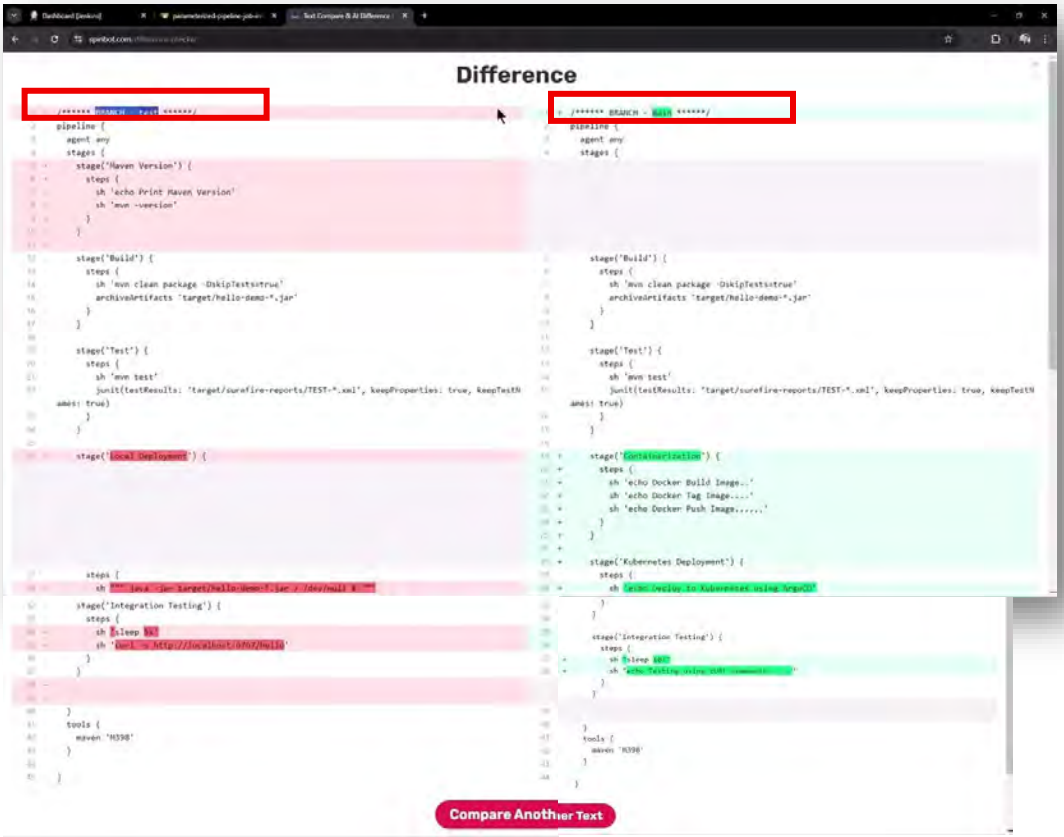
🧩 الجزء الأول : الفكرة وراء Parameters

- ليه نستخدم parameters ؟
  علشان نخلّي الـ build مرن وقابل للتخصيص.
- يعني بدل ما كل شغل يكون ثابت في Jenkinsfile ، نقدر نمرر parameters مختلفة وقت التشغيل
  (زي branch name أو port أو delay time)

🧱 الجزء الثاني : إعداد المشروع على GitHub

- عنده repo اسمه parameterized-pipeline-job-init
- فيه تطبيق Spring Boot Hello World
- يحتوي على فرعين:
  - main
  - test
- الفرق الوحيد بينهم هو Jenkinsfile بتاع كل فرع:
  - test branch : بيعمل build + test + local deploy + integration testing
  - main branch : بيعمل build + test + Docker build + Kubernetes deploy

dasher-org/parameterized-pipeline-job-init



⚙️ الجزء الثالث: إعداد Job في Jenkins

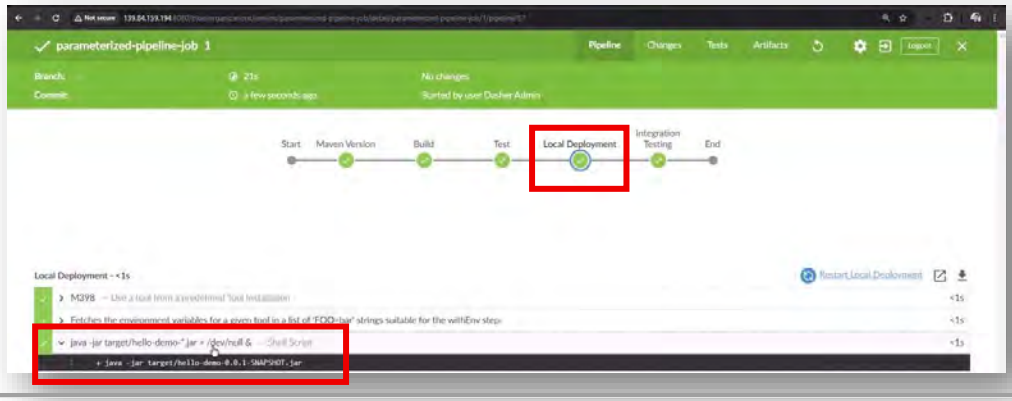
1. في Jenkins → Create New Item ← سميها : Parameterized Pipeline Job
2. نوعها : Pipeline
3. Source Code Management ← Git ← حط لينك الـ repo
4. في الأول خلى الـ branch test بـdefault
5. اضغط Apply ← بعدين Build Now
6. الـ Pipeline بتبدأ تشتغل وبتعمل build/test/deploy زي المعتاد.

💡 الجزء الرابع : تحليل أمر التشغيل داخل Jenkinsfile (test branch)

في مرحلة الـ deploy : `java -jar target/app.jar > /dev/null &`

- > /dev/null ← يعني تجاهل أي output من البرنامج.
- & ← يعني شغله في الخلفية.

➢ بعد كده بيعمل `curl localhost:6767/hello` علشان يعمل Integration Test



🧰 الجزء الخامس : إضافة Parameters

❑ بعد ما اتأكد إن الشغل شغال، بنبدأ نفعّل parameters ونضيفها :

1 Branch Name (String Parameter)

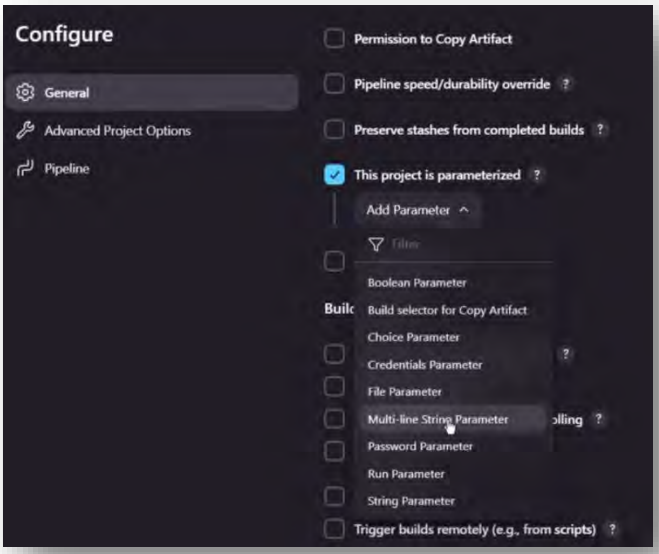
- الاسم : branch_name
- Default : main
- الهدف : نحدد الفرع اللي الـ build هيشتغل عليه (main أو test)

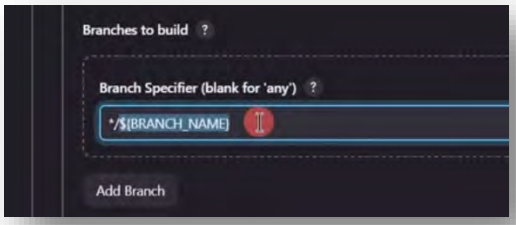
2 App Port (String Parameter)

- الاسم : app_port
- Default : 6767
- الهدف : نقدر نغيّر الـ port اللي الـ app بيشتغل عليه.

3 Sleep Time (Choice Parameter)

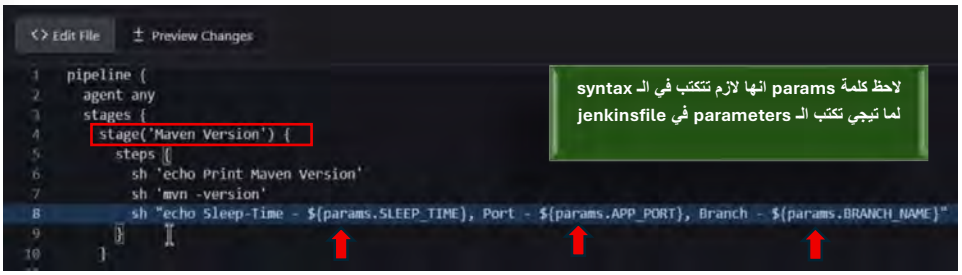
- الاسم : sleeptime
- القيم:
- 5s
- 10s
- 15s
- الهدف : نتحكم في الوقت اللي الـ pipeline تستناه قبل الـ integration test

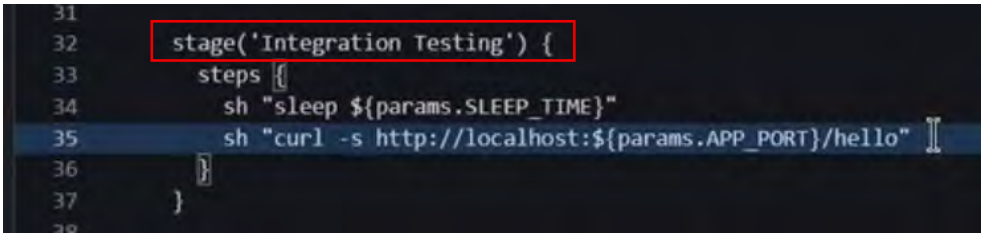




الجزء السادس : تعديل Pipeline ليستخدم Parameters

- بدل ما نكتب الـ branch ثابت في الكود، نستخدم : ${branch_name}
  - وده بيخلي Jenkins ياخد القيمة اللي المستخدم كتبها وقت التشغيل .
- وفي سكريبت الـ pipeline نفسه (داخل Jenkinsfile) هاتضيف السطور دي في test branch :





⚠️ لازم نستخدم " " double quotes مش ' ' single quotes

لأن الـ double quotes تسمح بعمل variable substitution (استبدال المتغير بقيمته)

الجزء السابع : تجربة الـ Build

بعد ما حفظ التعديلات:

1. لما ترجع للـ Job ← هتلاقي الزرار اتغيّر من Build Now إلى Build with Parameters
2. تقدر تختار:
   - branch_name = test
   - app_port = 6767
   - sleeptime = 5s
3. تعمل Build وتشوف النتيجة في Blue Ocean
   - الـ Pipeline هيطبع القيم اللي اخترتها ويستخدمها فعليًا في الأوامر:
   - echo "Sleeping for 5s on port 6767 (branch: test)"

الجزء الثامن : تطبيق نفس الفكرة على main branch

- بيروح يعدّل Jenkinsfile بتاع الـ main branch
- بيستبدل القيم الثابتة بنفس طريقة استخدام الـ parameters
- بيعمل build جديد، ويختار:
  - branch = main
  - sleeptime = 15s

النتيجة النهائية

بقى عندك Job واحدة فقط تقدر:

- تبني وتشغّل أي فرع (test أو main)
- تغيّر الـ port بحرية
- تتحكم في الـ sleep time .. وكل ده بدون ما تعمل Jenkinsfile جديد أو Job جديدة

Automating jenkins using cli and APIs

🎯 المقدمة

- ❑ هانتكلم عن Jenkins CLI و REST API ودي طريقتين مهمين جدًا للتعامل مع Jenkins بشكل أوتوماتيكي من غير الحاجة لإنك تدخل الـ Web UI (الـ Dashboard)
- ❑ الفكرة ببساطة إن بدل ما كل مرة تفتح Jenkins على المتصفح وتدوس manually عشان تعمل Build أو تنزل Plugin أو تعمل Configuration .. فأنت تقدر تعمل كل ده بالأوامر (CLI) أو بالـ API Requests (REST API)

🚀 وده بيخليك تشتغل بسرعة، تعمل Scripts ، وتدمج Jenkins مع أنظمة تانية بسهولة جدًا.

🧩 أولًا Jenkins CLI (Command-Line Interface) :

- ❑ Jenkins بيقدملك واجهة أوامر (CLI) تقدر من خلالها تتعامل مع Jenkins من الـ Terminal أو من Script Bash

فيه طريقتين أساسيين تقدر تستخدم بيهم CLI :

1. عن طريق SSH
2. عن طريق Jenkins CLI Client (ملف jar.)

🧠 1️⃣ SSH Access

- Jenkins بيكون فيه خدمة SSH تقدر تدخل منها بالأوامر.

بس خليك عارف إن الـ SSH في Jenkins بيكون Disabled افتراضيًا (يعني لازم تفعلّه بنفسك)

🔧 تفعيله :

- تطلب من Jenkins إنه يختار Port عشوائي للـ SSH ، بالأمر:
  - jenkins.model.Jenkins.instance.getDescriptor("jenkins.CLI").getSshd().getActualPort()
  - ده بيرجعلك الـ port اللي Jenkins اختاره.

```
Terminal
$ curl -Lv https://JENKINS_URL/login 2>&1 | grep -i 'x-ssh-endpoint'
< X-SSH-Endpoint: localhost:53801
```

🔐 إعداد الـ Authentication :

- Jenkins في حالة الـ SSH بيعتمد على Public/Private Key Authentication
- يعني لازم المستخدم يكون عنده SSH key pair، وتضيف الـ public key بتاعه داخل إعدادات Jenkins :

Jenkins → Manage Jenkins → Configure Global Security → SSH Public Keys

- بعد كده، المستخدم يقدر يدخل وينفذ أوامر CLI

🧰 تنفيذ الأوامر:

- بعد ما تضيف المفتاح وتفعل SSH ، تقدر تنفذ أوامر CLI زي مثلًا : ssh -p 2222 user@jenkins-server help
- ده بيعرضلك كل الأوامر المتاحة في CLI على السيرفر ده.
- أوامر CLI بتشمل حاجات زي:
  - بناء Jobs
  - عرض List بالـ Jobs
  - تنزيل Plugins أو حذفها

```
Terminal
$ ssh -l username -p 53801 localhost help
```

- إدارة Nodes
- وقف Builds .... إلخ

🧪 مثال عملي:

- لو عندك Job اسمه Hello_KodeKloud، تقدر تبنيه بالأمر:

ssh -p 2222 user@jenkins-server build Hello_KodeKloud

➢ كأنك دوست "Build Now" من الواجهة لكن هنا أوتوماتيكي.

```
Terminal
$ ssh -l username -p 53801 localhost build hello-kodekloud -f -v
Started hello-kodekloud #1
Started from command line by username
Building in workspace /tmp/jenkins/workspace/hello-kodekloud
[hello-kodekloud] /bin/sh -xe /tmp/hudson56238023482.sh
+ echo Hello KodeKloud
Hello KodeKloud
Finished: SUCCESS
Completed hello-kodekloud #1 : SUCCESS
```

2️⃣ 💡 Jenkins CLI Client (.jar file)

- الطريقة التانية والأكثر شيوعًا — استخدام ملف jenkins-cli.jar وده بيتوزع تلقائيًا مع الـ Jenkins Server بتاعك
- تقدر تنزله من: http://<jenkins-url>/jnlpJars/jenkins-cli.jar

◆ مميزاته:

- بيستخدم HTTP كبروتوكول افتراضي (فمش محتاج تفتح Port إضافي في Firewall)
- بيخليك تتعامل مع Jenkins بالأوامر من أي Terminal
- تقدر تكتبه داخل Scripts أو Jenkins Pipelines

◆ استخدامه :

مثال :

- قائمة كل الـ jobs :

```
Terminal
$ java -jar jenkins-cli.jar [-s JENKINS_URL] -auth user:token list-jobs
```

- بناء Job معين:

```
Terminal
$ java -jar jenkins-cli.jar [-s JENKINS_URL] -auth user:token build <job_name>
```

- تحميل تعريف الـ Job كـ XML (مفيد للـ backup أو النقل) :

java -jar jenkins-cli.jar -s http://localhost:8080/ get-job Hello_KodeKloud > job.xml

```
Terminal
$ java -jar jenkins-cli.jar [-s JENKINS_URL] -auth user:token get-job <job_name>
```

🔐 أنواع الـ Authentication :

1. Basic Authentication

بتستخدم username و password :

java -jar jenkins-cli.jar -s http://localhost:8080/ -auth user:password build Hello_KodeKloud

⚠️ مش آمن لأنه بيظهر الـ Password صريح.

2. API Token Authentication (الأفضل)

- تقدر تنشئ API Token من حسابك في Jenkins : User → Configure → API Token → Generate
- وتستخدمه كده:

java -jar jenkins-cli.jar -s http://localhost:8080/ -auth user:token list-jobs

✅ آمن أكتر لأنك تقدر تلغي الـ Token في أي وقت من غير ما تغير Password

---

🧩 ثانيًا Jenkins REST API :

- دي واجهة Jenkins البرمجية — تقدر تتحكم في Jenkins بالكامل عن طريق HTTP Requests
  يعني تقدر تعمل نفس اللي CLI بيعمله لكن من كود أو من برامج تانية.

🧰 تقدر تعمل بيها:

- تشغيل Build
- إنشاء أو تعديل Job
- حذف Plugin أو Job
- تحميل Logs أو Artifacts
- إدارة Nodes و Agents

🔧 مثال عملي:

لو عايز تنزل Plugin باستخدام REST API :

```
curl -s POST --data "<jenkins><install plugin='${plugin}' /></jenkins>" \
 -H 'Content-Type: text/xml' \
 http://localhost:8080/pluginManager/installNecessaryPlugins \
--user admin:$JENKINS_TOKEN
```

```
curl -X POST http://user:token@jenkins-server/pluginManager/install?plugin=git
```

أو لو عايز تشغل Job :

```
curl -X POST http://user:token@jenkins-server/job/Hello_KodeKloud/build
```

💡 استخدام مكتبات برمجية :

ممكن تستخدم لغات زي:

- Python (requests library) - Java (HttpClient) - Go / Node.js
  ودي بتسهل كتابة Scripts أو Integrations مع Jenkins

🔐 الـ Authentication في REST API :

زي CLI بالضبط:

1. Basic Auth (username + password)
   بيستخدم في Headers : Authorization: Basic base64(username:password)
2. API Token (الأفضل) : Authorization: Basic base64(username:token)

⚠️ يفضل متخزنش البيانات دي جوه كودك مباشرة — خزنها في Environment Variables أو ملفات Config مش عامة.

💎 Best Practices & Real Use Cases

- استخدم CLI أو REST API في الـ CI/CD automation scripts بدل ما تعمل خطوات Manual
- استخدم API Tokens دايمًا بدل passwords
- في بيئات Production ، اربط Jenkins بـ Credential Vault زي HashiCorp Vault أو AWS Secrets Manager.
- استخدم REST API لو بتحتاج Integration بين Jenkins و Systems تانية (زي GitHub, Jira, Slack)
- استخدم CLI في local automation أو quick maintenance tasks (زي build/delete jobs)

🧭 الملخص العام

❖ لو عايز تنفذ أوامر بسرعة وسكربتات إدارية ← استخدم CLI

❖ لو عايز تدمج Jenkins مع أنظمة تانية أو تعمل تطبيق بيتعامل معاه ← استخدم REST API

الجانب	Jenkins CLI	Jenkins REST API
الاستخدام	تنفيذ أوامر مباشرة من Terminal	تحكم برمجي من أي لغة
البروتوكول	SSH أو HTTP	HTTP/HTTPS
الأمن	SSH keys / Tokens	Tokens / Basic Auth
مناسب لـ	System Admins / Scripts	Developers / Integrations
أهم المزايا	سريع وسهل التنفيذ	مرن جدًا وقابل للدمج في أي تطبيق

Jenkins CLI Buila a job

🎯 المقدمة

❑ الجزء ده بيشرح إزاي تستخدم Jenkins CLI فعليًا علشان تبني Job من الـ Terminal أو من داخل Script

➢ يعني بدل ما تدخل الـ Jenkins Web UI وتدوس"Build Now" ، هتقدر تنفذ نفس الكلام من خلال أمر واحد بس من الـ Shell وده مهم جدًا لأي System Administrator أو DevOps Engineer لأنك ممكن تحتاج :

- تبني Job أوتوماتيكيًا بعد Deploy
- تشغل Scripts تعمل Monitoring أو Backup
- تدمج Jenkins مع أنظمة تانية (زي Git أو Ansible أو Python scripts)

🧩 الشرح التفصيلي

◆ الخطوة 1 – تحميل Jenkins CLI jar

➢ قبل ما تستخدم CLI ، لازم تنزّل ملف الـ jar الخاص بـ Jenkins CLI

➢ الملف ده موجود داخل الـ Jenkins نفسه.

تروح من : Manage Jenkins → Jenkins CLI

هناك هتلاقي:

- رابط لتحميل الملف jar (اسمه jenkins-cli.jar)
- رابط للـ documentation اللي بيشرح كل الأوامر اللي ممكن تستخدمها.

أو تقدر تحمّله مباشرة من اللينك: http://<jenkins_url>/jnlpJars/jenkins-cli.jar

🧠 ملاحظة :

الـ jar ده بيديك إمكانية إنك تتحكم في Jenkins بالكامل بالأوامر بدل من الواجهة.

◆ الخطوة 2 – تحميل الملف عمليًا

- هاتدخل على الـ VM اللي فيها Jenkins ، وتروح على الـ home أو root directory ،
- استخدم أمر التحميل : wget http://<jenkins_url>/jnlpJars/jenkins-cli.jar

بعد ما يتحمّل، تقدر تشغّله باستخدام Java

◆ الخطوة 3 – تشغيل Jenkins CLI لأول مرة

- الأمر المستخدم : java -jar jenkins-cli.jar -s http://<jenkins_url>/ help
- شرح المكونات:

الجزء	المعنى
`java -jar`	بيشغّل ملف jar
`jenkins-cli.jar`	الملف اللي نزّلته
`-s`	اختصار لـ "server" (رابط الـ Jenkins)
`help`	أمر بيعرض كل الأوامر المتاحة في الـ CLI

❖ ده هيعرضلك قائمة ضخمة بالأوامر اللي ممكن تستخدمها زي build, delete-job, install-plugin ... إلخ

```
Usage: java -jar jenkins-cli.jar [-s URL] command [opts...] args...
Options:
 -s URL              : the server URL (defaults to the JENKINS_URL env var)
 -webSocket          : connect using WebSocket (the default; works well with most reverse proxies; requires Jetty)
 -http               : use a pair of HTTP(S) connections rather than WebSocket
 -ssh                : use SSH protocol rather than WebSocket (requires -user; SSH port must be open on server)
 -i KEY              : SSH private key file used for authentication (for use with -ssh)
 -noCertificateCheck : bypass HTTPS certificate check entirely. Use with caution
 -noKeyAuth          : do not try to load the SSH authentication private key. Conflicts with -i
 -user               : specify user (for use with -ssh; must have registered a public key)
 -strictHostKey      : request strict host key checking (for use with -ssh)
 -logger FINE        : enable detailed logging from the client
 -auth [ USER:SECRET | @FILE ] : specify username and either password or API token (or load from them both from a file);
                                 for use with -http.
                                 Passing credentials by file is recommended.
                                 See https://www.jenkins.io/redirect/cli-http-connection-mode for more info and options.
 -bearer [ TOKEN | @FILE ]     : specify authentication using a bearer token (or load the token from file);
                                 for use with -http. Mutually exclusive with -auth.
                                 Passing credentials by file is recommended.

The available commands depend on the server. Run the 'help' command to see the list.
```

◆ الخطوة 4 – تجربة أول أمر

- هو استخدم أمر : **java -jar jenkins-cli.jar -s http://localhost:8080 who-am-i**
  - ودي أول تجربة عشان يعرف Jenkins شايفه كمستخدم مين.

النتيجة كانت:

```
root@jenkins-controller-1 in ~ via v17.0.12
❯ java -jar jenkins-cli.jar -s http://139.84.159.194:8080/ who-am-i
Authenticated as: anonymous
Authorities:
  anonymous
```

📍 المعنى:

هو مش داخل بأي user فعلي، فـ Jenkins شايفه مستخدم مجهول (anonymous) وده طبيعي لأن لسه مفيش Authentication

---

◆ الخطوة 5 – تجربة أمر بدون صلاحية

- جرب يعمل : **java -jar jenkins-cli.jar -s http://localhost:8080 list-jobs**
  - لكن ظهرله Error : **anonymous is missing the Overall/Read permission**

📌 السبب:

Jenkins من ناحية الأمن بيمنع أي مستخدم مش Logged in إنه يشوف أو يتحكم في الـ jobs

وده بيدخّلنا على المرحلة الجاية 👇

◆ الخطوة 6 – إضافة Authentication

❖ Jenkins CLI بيسمحلك تعمل Authentication بطريقتين :

1. Username + Password
2. Username + API Token (الأفضل والأكثر أمانًا)

✳ أولاً: تجربة Username & Password

- الأمر : **java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:password list-jobs**
  - لو اليوزر والباسورد صح .. Jenkins هيعرض كل الـ jobs

```
root@jenkins-controller-1 in ~ via v17.0.12
❯ java -jar jenkins-cli.jar -s http://139.84.159.194:8080/ -auth admin:password list-jobs
ascii-build-job
ascii-deploy-job
ascii-test-job
Generate ASCII Artwork
hello-world-pipeline
jenkins-hello-world
parameterized-pipeline-job
```

- بعد كده جرب تعمل: **java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:password who-am-i**
  - النتيجة :

    Authenticated as: admin

    Authorities: authenticated

✅ كده اتأكد إنه داخل كـ admin ، وبقى عنده صلاحيات كاملة.

◆ الخطوة 7 – تشغيل Build من CLI

- دلوقتي هنجرب نبني Job فعلي من CLI
- عشان تعرف شكل أمر الـ build ، تقدر تروح في Jenkins CLI Documentation وتشوف أمر build هيقولك إن

الصيغة العامة: **java -jar jenkins-cli.jar -s http://<jenkins> -auth user:pass build <job_name>**

```
java -jar jenkins-cli.jar -s http://139.84.159.194:8080/ build JOB [-c] [-f] [-p] [-r N] [-s] [-v] [-w]

Starts a build, and optionally waits for a completion.
Aside from general scripting use, this command can be
used to invoke another job from within a build of one job.
With the -s option, this command changes the exit code based on
the outcome of the build (exit code 0 indicates a success)
and interrupting the command will interrupt the job.
With the -f option, this command changes the exit code based on
the outcome of the build (exit code 0 indicates a success)
however, unlike -s, interrupting the command will not interrupt
the job (exit code 125 indicates the command was interrupted).
With the -c option, a build will only run if there has been
an SCM change.

 JOB : Name of the job to build
 -c  : Check for SCM changes before starting the build, and if there's no
       change, exit without doing a build (default: false)
 -f  : Follow the build progress. Like -s only interrupts are not passed
       through to the build. (default: false)
 -p  : Specify the build parameters in the key=value format. (default: {})
 -s  : Wait until the completion/abortion of the command. Interrupts are passed
       through to the build. (default: false)
 -v  : Prints out the console output of the build. Use with -s (default: false)
 -w  : Wait until the start of the command (default: false)
```

Option	الوظيفة
p-	بتمرر parameter بـ key=value
f-	تتابع تنفيذ الـ build مباشرة
s-	تنتظر لحد ما الـ build يخلص أو يفشل
v-	تعرض logs الـ console مباشرة
w-	تنتظر لحد ما الـ build يبدأ

◆ الخطوة 8 – إضافة Parameters

- لو الـ job من النوع "Parameterized Pipeline Job"، يعني بياخد input parameters (زي اسم branch مثلًا) فتقدر تمررها كده:
  - **java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:password build "name of your job" -p branch_name=test_branch**

◆ الخطوة 9 – تنفيذ الأمر ومتابعة النتيجة

نفّذ:

- **java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:password build \ "Parameterized_Pipeline" -p branch_name=test_branch -f**

النتيجة كانت :

```
Started build #4
Completed build #4 : SUCCESS
```

يعني:

- بدأ الـ job رقم 4
- وخلص بنجاح ✅

◆ الخطوة 10 – عرض الـ Console Output داخل الـ Terminal

- لو عايز تشوف الـ log الكامل للـ build نفسه، استخدم:

  - `java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:password build \ "Parameterized_Pipeline" -p branch_name=test_branch -v`

هيعرضلك في الـ Terminal كل خطوات الـ job كأنك بتشوفها من الـ UI .. وهايظهرلك فعلاً :

- Stage: Test
- Stage: Integration Test
- Sleep 5 seconds
- Build completed successfully

⚙️ الجزء العملي والسيناريو الواقعي

- في بيئة حقيقية، ممكن تستخدم CLI commands دي داخل Shell Script زي كده:

```
#!/bin/bash
JOB_NAME="Deploy_Pipeline"
BRANCH="feature-xyz"
JENKINS_URL="http://jenkins.local:8080"
AUTH="admin:MyToken123"

java -jar jenkins-cli.jar -s $JENKINS_URL -auth $AUTH build "$JOB_NAME" -p branch_name=$BRANCH -f -v
```

وتخلي السكريبت ده يشتغل كـ Cron Job أو كجزء من CI/CD pipeline

---

💎 Best Practices

- ✅ استخدم API Tokens بدل الـ password
- ✅ احفظ بيانات الـ auth في Environment Variables مش جوه السكريبت مباشرة.
- ✅ استخدم Jenkins CLI في المهام السريعة أو الـ automation scripts
- ✅ لو بتبني job كبير، استخدم -s و -v عشان تتابع التنفيذ لحظيًا.
- ✅ استخدم CLI في صيانة Jenkins (زي إيقاف Builds أو حذف Jobs قديمة)

🧭 الملخص النهائي

الموضوع	الفكرة الأساسية
Jenkins CLI	أداة بتسمحلك تتحكم في Jenkins من الـ Terminal
تحميل CLI	من صفحة "Manage Jenkins → Jenkins CLI"
أول تشغيل	`java -jar jenkins-cli.jar -s <url> help`
Authentication	`auth user:password-` أو `auth user:token-`
تشغيل Build	`<build <job_name` مع options زي `-v` , `-f` , `p`
الفايدة	تشغيل Jobs وعمليات Jenkins تلقائيًا من أي مكان

Jenkins REST API-install a plugin

🎯 المقدمة

- ❑ الجزء ده بيتكلم عن استخدام Jenkins REST API ، وده وسيلة قوية جدًا للتعامل مع Jenkins من خلال طلبات HTTP (زي GET, POST) بدل ما تستخدم الـ UI أو CLI
- ❑ يعني تقدر تدير Jenkins بالكامل وتعمل Trigger Jobs, Create Jobs, Retrieve Info, Install Plugins ... وكله من REST API
- ❑ وده مفيد جدًا في الحالات الآتية:
- لما تكون بتعمل Automation Scripts
- لما عايز تربط Jenkins مع أدوات تانية زي Ansible, Python, GitHub Actions
- لما تبني CI/CD system مخصص من الصفر.

🧩 الشرح التفصيلي – خطوة بخطوة

◆ الخطوة 1 – فهم REST API في Jenkins

- Jenkins بيوفر API قابل للاستهلاك الآلي (Machine Consumable API) وده بيسمحلك توصل لكل البيانات والوظائف اللي بتتعامل معاها عادة من الـ UI
- أنواع الـ APIs اللي بيوفرها Jenkins :
  1. XML API
  2. JSON API
  3. Python API
    - كل واحدة منهم بتسمحلك تشوف وتتعامل مع بيانات Jenkins بشكل برمجي.
- مثلاً:
  - تقدر تسترجع معلومات الـ Jobs
  - تشغل Build جديد.
  - تنشئ Job جديد.
  - تحدث Plugin أو تشوف حالته.

◆ الخطوة 2 – أمثلة على REST Calls

- الـ Endpoint الأساسي لـ Jenkins بيبقى بالشكل ده : http://<jenkins_url>/job/<job_name>/build
- ولو الـ Job بياخد Parameters : http://<jenkins_url>/job/<job_name>/buildWithParameters
- معاه:
  - Username و Token للـ Authentication
  - الـ Parameters في Body أو URL

❖ الطريقة الشائعة للتنفيذ باستخدام curl :

```
curl -u "admin:token" -X POST "http://localhost:8080/job/my-pipeline/buildWithParameters?branch=test"
```

◆ الخطوة 3 – واجهة REST API داخل Jenkins

- لو دخلت على Jenkins Dashboard ونزلت لتحت هتلاقي رابط اسمه REST API
هناك هتشوف:
  - XML API
  - JSON API
  - Python API

وفي نفس الصفحة Jenkins بيشرحلك إزاي تتحكم في كمية البيانات اللي هتسترجعها (عشان الـ output ساعات بيبقى ضخم جدًا)
فمثلاً ممكن تستخدم tree parameter لتحديد الحقول اللي عايزها بس : /api/json?tree=jobs[name,color]

◆ الخطوة 4 – تجربة استرجاع بيانات في JSON

- مثلاً لو عايز تجيب كل الـ Jobs كـ JSON : http://localhost:8080/api/json
- ده بيرجعلك قائمة بالـ Jobs وأنواعها:
  - Freestyle project
  - Pipeline project
  - Multibranch pipeline

ولو عايز تحدد اسم الـ Job : http://localhost:8080/api/json?tree=jobs[name]

◆ الخطوة 5 – تشغيل نفس الطلب من الـShell

- من الـ Terminal تقدر تعملها كده: curl http://localhost:8080/api/json
- ولو عايز تطبع النتيجة بشكل منسق استخدم jq : curl http://localhost:8080/api/json | jq
- بس هيطلب منك Authentication (لأنك مش Logged In زي ما في الـ Browser)

◆ الخطوة 6 – إضافة Authentication

- الطريقة الصحيحة : curl -u admin:password http://localhost:8080/api/json
- أو الأفضل باستخدام API Token بدل الباسورد : curl -u admin:myapitoken http://localhost:8080/api/json

بعد كده هتشوف رد JSON فيه:

- اسم الـ job
- URL بتاعه
- الـ color (status)
  - Blue = Success
  - Red = Failed
  - Yellow = Unstable

◆ الخطوة 7 – جلب تفاصيل Job محددة

- لو عايز تشوف بيانات Job بعينها، استخدم:

curl -u admin:token http://localhost:8080/job/MyPipeline/api/json

ده بيرجع تفاصيل كاملة:

- Display name
- URL
- عدد الـ Builds
- حالة كل Build

- الـ Parameters الموجودة
- Health Report

بمعنى تقدر تعرف كل حاجة عن Job من REST API مباشرة.

🔹 الخطوة 8 --- Trigger Job من REST API

- لو عايز تشغّل Job بياخد Parameters :

```
root@jenkins-controller-1 in ~ via
❯ curl -u admin:password -X POST http://139.84.159.194:8080/job/parameterized-pipeline-job/buildWithParameters -d BRANCH_NAME=
test -d APP_PORT=6767
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=ISO-8859-1"/>
<title>Error 403 No valid crumb was included in the request</title>
</head>
<body><h2>HTTP ERROR 403 No valid crumb was included in the request</h2>
<table>
<tr><th>URI:</th><td>/job/parameterized-pipeline-job/buildWithParameters</td></tr>
<tr><th>STATUS:</th><td>403</td></tr>
<tr><th>MESSAGE:</th><td>No valid crumb was included in the request</td></tr>
<tr><th>SERVLET:</th><td>Stapler</td></tr>
</table>
<hr/><a href="https://eclipse.org/jetty">Powered by Jetty:// 10.0.20</a><hr/>

</body>
</html>
```

📍 ملاحظات مهمة:

- لازم تستخدم POST مش GET
- لازم تمرر الـ Parameters بالشكل -d key=value أو في الـ URL

🔹 الخطوة 9 – مشكلة الـ Crumb Error 403

- لو شغلت الأمر وطلعلك : 403 No valid crumb was included in the request
  - ده معناه إن Jenkins رفض الطلب لأسباب أمنية.

📌 السبب:

- Jenkins عنده نظام أمان اسمه Crumb (زي CSRF Token) وده بيحميه من هجمات Cross-Site Request Forgery

✅ الحل إنك تضيف الـ Crumb في Header بتاع الطلب.

🔹 الخطوة 10 – تجاوز مشكلة الـ Crumb

عندك حلّين:

1. ✅ استخدام Crumb فعلي

تحصل عليه كده : curl -u admin:token http://localhost:8080/crumbIssuer/api/json

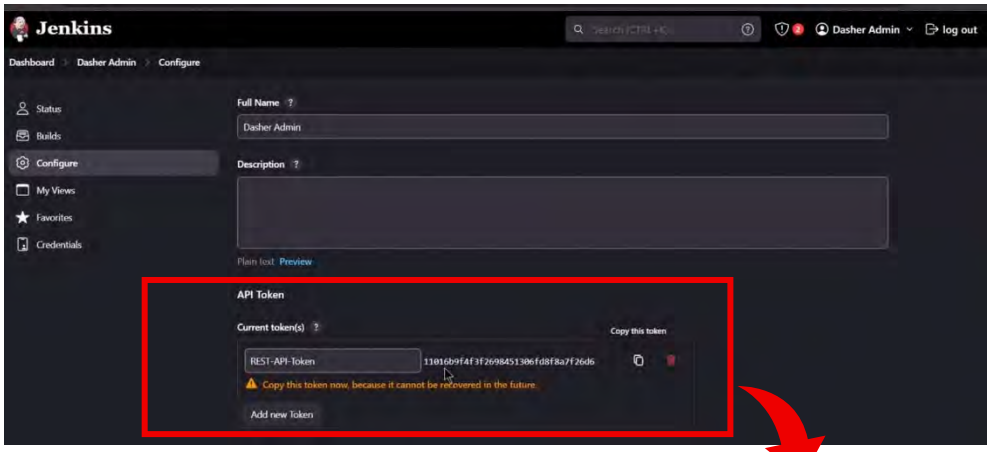
النتيجة فيها:

```
{
  "crumbRequestField": "Jenkins-Crumb",
  "crumb": "a1b2c3d4..."
}
```

وبعدين تستخدمه: curl -u admin:token -H "Jenkins-Crumb:a1b2c3d4..." -X POST ...

2. ✅ استخدام API Token فقط (الأفضل)

وده أسهل وأأمن لأن لو API Token مستخدم، Jenkins غالبًا مش بيحتاج Crumb




```
root@jenkins-controller-1 in ~ via v17.0.12
❯ curl -u admin:11016b9f4f3f2698451306fd8f8a7f26d6 -X POST  http://139.84.159.194:8080/job/parameterized-pipeline-job/buildWith
Parameters -d BRANCH_NAME=test -d APP_PORT=6767
```

◆ الخطوة 11 – إنشاء API Token

من داخل Jenkins UI :

1. اضغط على اسم المستخدم (admin)
2. اختار Configure
3. انزل لقسم API Token
4. اضغط Add New Token
5. سمه مثلاً "REST_API_Token"
6. اضغط Generate
7. انسخ القيمة اللي طلعتلك

دلوقتي استخدمها في curl : curl -u admin:your_generated_token -X POST ...

◆ الخطوة 12 – التأكد من نجاح التشغيل

- ارجع على الـ Jenkins UI ، وروح للـ Job هتلاقي Build جديد شغال (مثلاً Build #6)
- لو فتحت الـ console output ، هتلاقي إنه بدأ بواسطة : Started by user admin
- لكن مش هيقول "Started from command line" لأن ده REST API مش CLI jar

---

◆ الخطوة 13 – استخدام REST API لتثبيت Plugin

🔥 دا يعتبر الجزء الأهم || تثبيت Plugin باستخدام REST API بدل الواجهة.

الصيغة العامة:

```
curl -u admin:token -X POST  http://localhost:8080/pluginManager/installNecessaryPlugins \
    -H "Content-Type: text/xml" \
    -d '<install plugin="emotional-jenkins-plugin@1.2"/>'
```

شرح:

الجزء	الوصف
X POST-	لأننا بننشئ/نثبت Plugin
"H "Content-Type: text/xml-	نوع البيانات اللي رايحة
'<... d '<install-	XML payload فيه اسم الـ plugin والإصدار
pluginManager/installNecessaryPlugins/	Endpoint المسؤول عن التثبيت

◆ الخطوة 14 – التحقق من التثبيت

بعد تنفيذ الأمر بنجاح:

1. ارجع إلى Jenkins UI
2. افتح Manage Jenkins → Plugins → Installed / Download Progress
3. هتلاقي "Emotional Jenkins Plugin" بيتم تثبيته أو خلاص خلص ✅

➢ الـ Emotional Jenkins Plugin ببساطة بيغيّر وش Jenkins حسب حالة الـ Build (مبتسم لما ينجح – حزين لما يفشل)

◆ الخطوة 15 – استخدام REST API لتثبيت Plugins متعددة

- لو عندك Plugins كتير عايز تثبتها أوتوماتيك أثناء إعداد Jenkins لأول مرة،
  ممكن تبعت XML فيه أكتر من plugin :

```
<install plugin="git@5.2"/>
<install plugin="pipeline@2.6"/>
<install plugin="ssh@3.3"/>
```

وتستخدم نفس Jenkins — endpoint هيبدأ يحمّلهم واحد واحد.

💡 Best Practices و Tips مهمة

- ✅ دايمًا استخدم API Tokens مش Passwords
- ✅ احفظ الـ Token في Environment Variables (مش جوه السكريبت)
- ✅ لو بتستخدم Jenkins REST API بكثرة، فكّر تعمل Python Wrapper حواليه.
- ✅ استخدم jq لتنسيق JSON Outputs
- ✅ دايمًا راقب Permission الـ User اللي بيستخدم الـ Token (ما تدّيش admin لسكربت بسيط)
- ✅ لو بتبني automation setup لـ Jenkins ، استغل REST API لتثبيت Plugins أو إنشاء Jobs أول مرة.

🧭 الملخص النهائي :

الفكرة	الشرح
Jenkins REST API	طريقة للتعامل مع Jenkins برمجيًا عبر HTTP
الأنواع	XML, JSON, Python
الاستخدامات	Retrieve info, Trigger builds, Install plugins
Authentication	Username + API Token (الأفضل)
Crumb	Token أمني لتجنب CSRF attacks
Trigger build	`job/<job>/buildWithParameters/`
Install plugin	`pluginManager/installNecessaryPlugins/`
Automation	ممكن تجهز Jenkins كامل بالـ REST API

Jenkins CSRF-CRUMB

🎯 المقدمة:

- ❑ الجزء ده بيتكلم عن واحدة من أهم نُقط الأمان في Jenkins وهي CSRF Protection أو Cross-Site Request Forgery Protection
- وبيوضح إزاي Jenkins بيستخدم حاجة اسمها Crumb Token كوسيلة لحماية نفسه من الهجمات دي.
- كمان بيتكلم عن إزاي تولّد الـ Crumb Token من خلال الـ REST API ، وإزاي تستخدمه في أوامر Curl لتشغيل Jobs أو تنفيذ عمليات تانية على Jenkins
- في النهاية كمان بيشرح الفرق بين استخدام Username/Password + Crumb وبين استخدام API Token (اللي هو الأسهل والأكثر أمانًا)

🧩 الشرح التفصيلي

🔹 أولًا : فكرة الـ CSRF وسبب وجود الـ Crumb

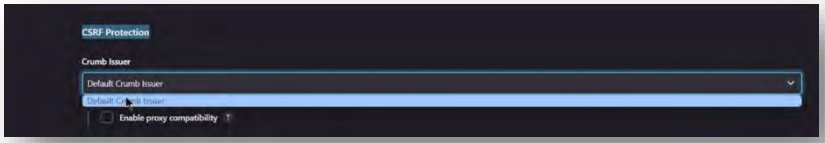
📘 الـ CSRF (Cross-Site Request Forgery)

- ده نوع من أنواع هجمات الويب اللي بتحاول تستغل جلسة المستخدم المفتوحة عشان تنفذ عمليات في الخلفية من غير إذنه.
- يعني لو أنت فاتح Jenkins في تبويب، وموقع تاني فيه كود JavaScript خبيث، الكود ممكن يحاول يبعت طلب لـ Jenkins وكأنه منك ... .عشان كده Jenkins بيستخدم Crumb Token

🔑 Crumb = رمز أمني فريد (Token)

- بيتولد لكل مستخدم ولكل Session بشكل عشوائي،
- وبيتحقق Jenkins إن الطلب اللي جاي فيه نفس الـ crumb اللي اتولد فعلاً في جلستك.
  - ➢ لو مش مطابق ← يرفض الطلب.

🔹 إعداد الـ CSRF Protection في Jenkins



- روح من لوحة Jenkins إلى :
- Manage Jenkins → Security
- هتلاقي خيار اسمه Prevent Cross-Site Request Forgery exploits
  ✅ بيكون مفعّل افتراضيًا.
- تقدر تطفئه (لكن مش بننصح بكده إطلاقًا) :
  - o يا إما من الواجهة أو عن طريق System Property ، لكن الأفضل تسيبه شغال دائمًا لحماية Jenkins

---

🔹 ما هو الـ Default Crumb Issuer ؟

- ده المكوّن المسؤول عن توليد وتحقق الـ Crumbs داخل Jenkins
- بيستخدم خوارزمية بتعمل Hash من شوية معلومات لتكوّن الـ Token
  - ➢ المعلومات اللي بتدخل في التوليد:

العنصر	المعنى
Username	المستخدم اللي تولّدله الـ crumb
Session ID	جلسة الويب الحالية
IP Address	عنوان الجهاز
Salt	قيمة فريدة خاصة بـ Jenkins نفسه

- كل ده بيتكوّن فيHash ، ولازم القيم دي تتطابق بالظبط لما Jenkins يستقبل crumb عشان يعتبر الطلب "صحيح."

◆ ازاي تولّد Crumb Token ؟

- الـ Crumb بيتولد من خلال REST API endpoint : /crumbIssuer/api/json
- يعني باستخدام curl : curl -s -u "admin:password" http://localhost:8080/crumbIssuer/api/json | jq
  - الناتج هيكون JSON فيه معلومتين مهمتين:

```
root@jenkins-controller-1 in ~ via v17.0.12
❯ curl -s \
-u admin:password \
http://localhost:8080/crumbIssuer/api/json | jq
{
  "_class": "hudson.security.csrf.DefaultCrumbIssuer",
  "crumb": "6286e097b759cb388daec3a4d4ef1fcde5da95edcbd779d8b9967c1239de5cff",
  "crumbRequestField": "Jenkins-Crumb"
}
```

- crumbRequestField : اسم الـ Header اللي Jenkins عايزه.
- Crumb : القيمة الفعلية للـ Token
-

◆ استلام الـ Cookie مع الـ Crumb

- عند تنفيذ الطلب، Jenkins بيرجع كمان Cookie داخل الـ Response Headers وده بيكون معاه Session ID للمستخدم.
- فعلشان أي طلب بعد كده يشتغل لازم تبعت:

1. الـ Crumb
2. الـ Cookie
3. الـ Credentials (username + password)

- لو نسيت حاجة فيهم ← Jenkins هيقولك

❌ "No valid crumb was included in the request".

◆ طريقة حفظ الـ Cookies

ممكن تخلي Curl يحفظ الـ Cookies تلقائيًا كده:

```
curl -u admin:password \
 --cookie-jar /tmp/cookies.txt \
 http://localhost:8080/crumbIssuer/api/json | jq
```

كده:

- الـ crumb بيتولد ويتعرض في الـ output
- الـ cookies بتتحفظ في /tmp/cookies.txt

لو عملت : cat /tmp/cookies.txt هتشوف الجلسة (Session info)

◆ استخدام الـ Crumb لتشغيل Job

- بعد مابقي معاك الـ Crumb و الـ Cookie ، تقدر تشغّل Job من خلال REST API كالتالي:

```
curl -u admin:password    --cookie /tmp/cookies.txt  -H "Jenkins-Crumb:a1b2c3d4e5..." \
  -X POST "http://localhost:8080/job/Parameterized-Pipeline- Job/buildWithParameters?BRANCH=dev"
```

الجزء	الشرح
`--cookie`	بيحمّل الكوكيز اللي اتخزنت
`-H "Jenkins-Crumb:...`	بيبعت الـ crumb token في الـ header
`-X POST`	عشان يشغّل الـ job
`BRANCH=dev?`	parameter للـ pipeline

لو كل حاجة مظبوطة، Jenkins هيبدأ build جديد.
هتلاقيه في الـ UI شغال عادي جدًا .

◆ التحقق من النتيجة

- افتح Jenkins UI ← ادخل على الـ Job ← هتلاقي build جديد بدأ
  مثلاً Build #6 أو #7
- وكمان هتلاقي في الـ Console Output مكتوب: Started by user admin

➢ وده معناه إن الـ REST API اشتغل بنجاح بالـ crumb اللي اتولّد.

◆ طيب ليه كل التعقيد ده؟

العملية فيها أكتر من خطوة:

1. Generate crumb
2. Save cookie
3. Reuse them مع الـ authentication

➢ وده ساعات بيبقى مجهد خاصة لو بتعمل Automation scripts

◆ البديل الأفضل API Token :

- عشان كده Jenkins بيسمح إنك تستخدم API Token بدل الباسورد والـ crumb .

📍 الميزة:

- أسهل.
- أكثر أمانًا.
- مش بيحتاج CSRF protection
- مخصص للاستخدام الآلي (Automation)

➢ يعني بدل كل اللي فوق، تقدر تعمل:

```
curl -u admin:apitoken -X POST \
“http://localhost:8080/job/MyPipeline/buildWithParameters?branch=test”
```

وخلاص Jenkins هينفذ الطلب بدون أي Crumb أو Cookie

💡 نصائح و Best Practices

النصيحة	الشرح
✅ استخدم API Token	أبسط وأأمن من Username/Password
🧱 سيب CSRF مفعّل	متطفيهوش أبدًا في بيئة Production
🧠 خزّن الـ Tokens في متغيرات بيئة	متحطّهاش جوه السكريبت
🔒 راقب صلاحيات المستخدم	تأكد إن الـ token مش admin لو مش ضروري
⚙️ استخدم jq	لتنظيم الـ JSON output
🚀 في الـ Automation	استخدم الـ REST API لتشغيل Jobs أو إدارة Jenkins بالكامل

الملخص النهائي :

المفهوم	الشرح
CSRF Attack	محاولة استغلال جلسة المستخدم لتنفيذ أوامر بدون إذنه
Crumb Token	رمز أمني يولده Jenkins لحماية الطلبات
Crumb Issuer	المكون اللي بينشئ ويتحقق من الـ crumb
Cookie Jar	ملف لتخزين الجلسات أثناء REST calls
API Token	بديل آمن وبسيط يغنيك عن استخدام الـ crumb
Practical Use	استخدم REST API لتشغيل Jobs أو تثبيت Plugins أو إدارة Jenkins أوتوماتيكيًا

Jenkins Security Overview

🧱 الفكرة العامة:

الجزء دا بيتكلم عن أمان Jenkins . يعني إزاي تحمي السيرفر من أي شخص مش مسموح له يدخل أو يعمل تغييرات ممكن تضر النظام.

- هو بيشبّه Jenkins بصندوق أدواتك (Toolbox) :

زي ما مش هتسيب أدواتك لحد ممكن يستخدمها غلط أو يسرقها، لازم برضه تحمي Jenkins من الوصول غير المصرح بيه.

🔐 1. Concept of Least Privilege (مبدأ أقل صلاحيات)

- القاعدة الأساسية في أمان Jenkins :

"ادّي المستخدم أقل صلاحيات ممكنة عشان يقدر يعمل شغله — مش أكتر".

- يعني مثلًا لو المستخدم دوره يعمل build فقط، متدّهوش صلاحية delete أو configure

---

🪪 2. Authentication vs Authorization

✅ Authentication (المصادقة) #انت مين#

- يعني إثبات الهوية .. بمعني هل الشخص ده فعلاً هو اللي بيقول عليه نفسه؟
- بيتحقق منها عن طريق:
  - Username + Password
  - أو API Token

💡 تشبيه : لما تروح أوتيل، بتورّي بطاقتك أو الباسبور ← كده الفندق عرف "إنت مين." .. ده هو الـ Authentication

---

🔑 Authorization (التفويض أو التصريح) #هاتعمل ايه#

- ده بيحصل بعد ما الـ Authentication يتأكد إنك فعلاً مستخدم حقيقي.
- المرحلة دي بتحدد: "طيب خلاص إنت متأكد، دلوقتي مسموحلك تعمل إيه؟"

💡 تشبيه:

بعد ما الهوتيل يتحقق منك، بيدّيك مفتاح أو كارت الغرفة — فده معناه إنك متفوض تدخل الغرفة دي بس (مش أي غرفة تانية).

يبقى المفتاح دا Authorization Token

👥 3. Managing Access in Jenkins

Jenkinsعنده كذا طريقة تتحكم بيها في الدخول، ودي حسب حجم شركتك:

النظام	الوصف	الأفضلية
Jenkins User Database	الافتراضي، مناسب للشركات الصغيرة	🟢 بسيط وسهل
Unix User/Group Database	يستخدم حسابات النظام نفسه	🟡 لو Jenkins شغال على Linux Server
Servlet Container Security	يعتمد على إعدادات الويب سيرفر (Tomcat, etc.)	🟡 متقدم شوية
LDAP / Active Directory	يتصل بدليل الشركات الكبير (Corporate Directory)	🔵 الأفضل للشركات الكبيرة

🔌 Plugins مفيدة:

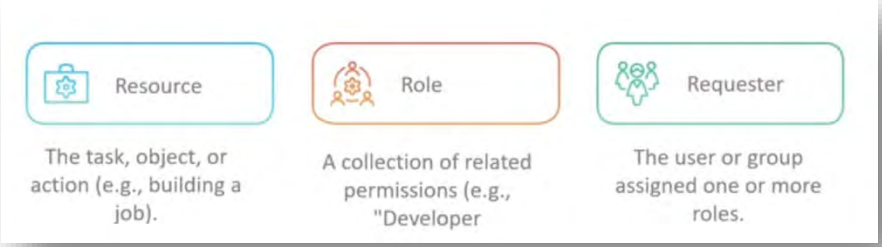
- Active Directory Plugin ← لو شركتك بتستخدم Microsoft AD
- SAML 2.0 Plugin ← لتفعيل Single Sign-On (SSO) بحيث تدخل Jenkins بنفس حساب الشركة.

4 🧩 Choosing the Right Method.

- لو عندك فريق صغير ← استخدم Jenkins Internal Database
- لو في شركة كبيرة فيها مئات المستخدمين ← استخدم LDAP أو Active Directory
  ده بيسهل إدارة المستخدمين ومافيش داعي تعمل حساب لكل واحد يدويًا .

---

5 🧱 Authorization Strategies (استراتيجيات التفويض)



🗂️ Matrix-Based Security

- بتظهر في شكل جدول فيه الأعمدة = الصلاحيات، والصفوف = المستخدمين.
- كل مستخدم ممكن تدي له صلاحيات معينة بشكل عام على كل المشاريع.

🧠 مثال:

User	Read	Build	Delete
admin	✅	✅	✅
dev	✅	✅	❌

🧩 Project-Based Matrix Authorization

- نفس الجدول، لكن الصلاحيات بتكون لكل مشروع على حدة.
- يعني dev1 ممكن يبني مشروع A لكن ما يقدرش يلمس مشروع B

---

6 ⚖️ Best Practice: Least Privilege Again.

- دائمًا خليك على مبدأ : "ادّي كل مستخدم أقل صلاحية ممكنة لإنجاز شغله فقط".
- ده بيحافظ على:
  - أمان Jenkins
  - ومن غير ما يبطّئ الـ pipelines أو يوقف الشغل.

🧭 الخلاصة

المفهوم	المعنى
Authentication	إثبات الهوية (إنت مين)
Authorization	الصلاحيات اللي تقدر تعملها
Least Privilege	أقل صلاحيات مطلوبة
Matrix-Based	صلاحيات عامة على كل المشاريع
Project-Based	صلاحيات لكل مشروع على حدة

Jenkins Authentication

🧩 أولاً: مفهوم الـ Authentication في Jenkins

- ❑ الجزء ده بيبدأ بشرح إن لما بتثبت Jenkins لأول مرة، بيكون عندك حساب admin واحد بس هو اللي تم إنشاؤه أثناء التثبيت.
- ❑ لكن لو فريقك عايز يبدأ يستخدم Jenkins ، مش هتلاقي زرار "Register" علشان المستخدمين يسجلوا نفسهم.

فالحل إنك تقدر:

- تفعّل خاصية التسجيل الذاتي للمستخدمين.
- أو تربط Jenkins بـ LDAP أو Active Directory في شركتك.

⚙️ إعداد الـ Authentication

1. تدخل على:
   - Manage Jenkins → Security
2. هتلاقي إن Jenkins بيستخدم افتراضيًا

   "Jenkins' own user database" 👉
   - يعني Jenkins بيخزّن بيانات المستخدمين داخليًا (مش من سيرفر خارجي).
3. تقدر بدلها تربط LDAP أو AD ، لكن في البداية جرب القاعدة الداخلية.

👤 إنشاء مستخدم جديد داخل Jenkins

- تدخل على Manage Jenkins → Users
- بتلاقي إن فيه مستخدم واحد فقط (admin)
- فعل خيار "Allow users to sign up" عشان الناس تقدر تسجل.
- كمان ممكن تلغي خيار "Keep me signed in" لو عايز تحافظ على الأمان.
- بعد كده لما تسجل خروج، هتلاقي زر "Register"

👷 مثال:

- سجّل مستخدم جديد اسمه John
- يدخل Username, Email, Password
- أول ما يعمل Register ← بيتضاف لحسابات Jenkins
- تقدر بعد كده تشوفه مع الـ admin في صفحة "Users"

🔐 ثانيًا: الـ Authorization (من له صلاحيات يعمل إيه)

➢ Jenkins بيديك 3 خيارات رئيسية للتحكم في الصلاحيات:

1️⃣ Anyone can do anything

يعني أي شخص حتى لو مش عامل login يقدر:

- يشوف المشاريع
- ويعمل build

⚠️ لكن ده خطر جدًا أمنيًا، لأن المستخدم المجهول ممكن يشغل build أو يعدّل حاجة في النظام.

2️⃣ Logged-in users can do anything

- ده الوضع الأفضل كبداية، لأنه بيخلي كل المستخدمين اللي داخلين فقط عندهم صلاحيات كاملة.

3️⃣ Allow anonymous read access

- تقدر تفعّل ده عشان الناس تشوف الـ jobs فقط بدون تعديلها.

زي:

- يقدر يشوف logs والنتائج
- لكن مش هيقدر يعمل build أو يضيف jobs جديدة

🧪 إعداد Mock Security Realm (بديل تجريبي للـ LDAP)

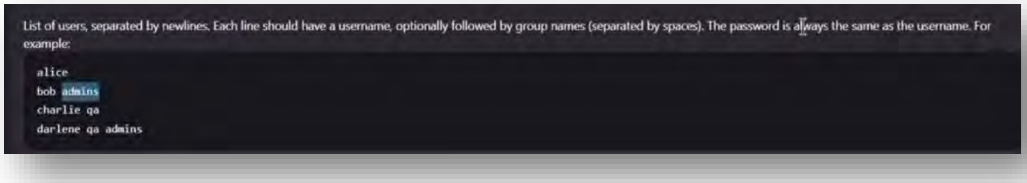
لما عايز تجرّب فكرة الـ Groups & Roles زي LDAP بس بدون سيرفر حقيقي:

- تروح على Manage Jenkins → Plugins → Available Plugins
- وتبحث عن Mock Security Realm :
- ده Plugin بسيط بيعمل محاكاة لقواعد بيانات LDAP

⚠️ فقط للتجارب، مش Production

👥 إعداد المستخدمين والجروبات في الـ Mock Realm

- بعد التثبيت، تروح على Security Realm وتختار Mock Security Realm
- هتضيف المستخدمين بالشكل ده:



📘 الملاحظات:

- كل user باسوردُه هو اسمه نفسه.
  (مثلاً اسم المستخدم "Ali" ← الباسورد "Ali")
- ممكن تربط كل مستخدم بجروب واحد أو أكتر.

🧱 بعد تطبيق الإعدادات:

- Jenkins مش هيعتمد على قاعدة بياناته القديمة.
- مش هتشوف "Users" Tab لأنك بقت تستخدم مصدر خارجي (mock realm)
- لازم تسجّل الدخول بمستخدم من اللي أضفتهم في الـ mock realm

🔑 مثال:

لو دخلت بـ Siddharth ← هيدخل عادي لأنه في قائمة المستخدمين الجديدة.
لكن admin أو John مش هيدخلوا لأنهم مش ضمن الـ mock realm

⚠️ ملحوظة مهمة جدًا

- الـ Mock Realm بيهتم فقط بـ Authentication (مين المستخدم؟ وجروبه إيه)
- لكنه مش بيتحكم في الـ Authorization (يقدر يعمل إيه داخل Jenkins)

ولذلك :

- كل المستخدمين يقدروا يبنوا ويحدفوا المشاريع حالياً.
- وده مش مظبوط من ناحية أمان.
- المفروض الجروب "admin" بس هو اللي يقدر يحذف المشاريع.

وده بالضبط اللي هنتكلم عنه في الدرس الجاي .. اللي هيتكلم عن Authorization Rules بالتفصيل (الصلاحيات المتقدمة).

Authentication-Matrix Authorization Strategy

🎯 أولًا: فكرة الـ Authorization في Jenkins

- بعد ما فهمنا في الجزء اللي فات الـ Authentication (مين اللي يقدر يدخل) .. الـ Authorization هو اللي بيحدد "اللي دخل فعلاً يقدر يعمل إيه."
- الـ Matrix Authorization Strategy بيديك تحكم تفصيلي (Fine-grained control) .. يعني تقدر تحدد صلاحيات معينة لكل User أو Group بالضبط.

مثلاً:

- الـ Admin يقدر يعمل أي حاجة
- الـ QA يقدر يعمل Build بس
- الـ Developer يقدر يعدل على Jobs بس مش يحذفها وهكذا

⚙️ الخطوة الأولى – تفعيل Matrix Authorization Strategy

1. تدخل كـ Admin (في المثال : Siddharth ← الاسم والباسورد نفس الشيء)
2. من Manage Jenkins → Security
   - هتلاقي الـ Authorization حاليًا على : "Any logged-in user can do anything"
   يعني كل اللي عامل Login يقدر يعمل كل حاجة.
3. عشان نغير ده، لازم نركّب Plugin جديد اسمه: Matrix Authorization Strategy 👉

🧩 تثبيت الـ Plugin

- تروح على Manage Jenkins → Plugins → Available :
- تبحث عن Matrix Authorization Strategy :
- وتعمل Install

🔍 وظيفة الـ Plugin :

- يسمحلك تعمل Access Control مفصل جدًا.
- تقدر تطبقه على مستوى Jenkins كله (Global) أو على مشروع معين (Project-based)

🎬 بعد التثبيت – إعداد الـ Authorization Modes

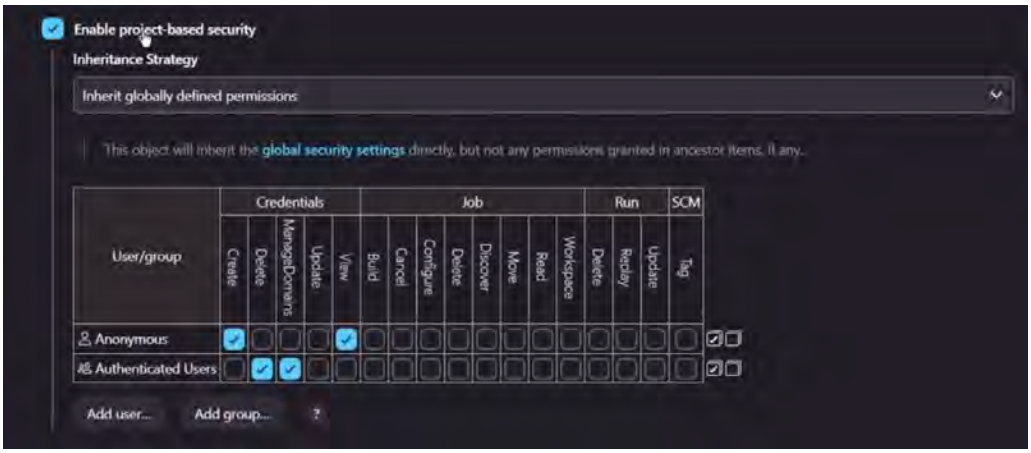
هتلاحظ إن فيه خيارين جداد:

1. Project-based Matrix Authorization Strategy
2. Matrix-based Security

الفرق بينهم 👇

الوضع	الوظيفة
Project-based	تقدر تحدد صلاحيات مختلفة لكل Job أو Project
Global Matrix	الصلاحيات بتتطبق على Jenkins كله

- لو فعلت **Project-based**، ودخلت على أي **Job name → Configure** :
  - هتلاقي خانة جديدة اسمها **"Enable project-based security"** ومنها تقدر تحدد الصلاحيات الخاصة بالمشروع ده فقط.
- أما لو اخترت **Matrix-based Security**، فده يطبق نفس الجدول على كل الـ **Jenkins**



🧱 إعداد الـ Matrix Table

لما تختار **Matrix-based Security**، هتشوف جدول فيه:

- على اليسار : المستخدمين أو الجروبات.
- في الأعلى : كل أنواع الصلاحيات (Overall, Job, View, Credential, Agent ...)

👥 القيم الافتراضية:

- **Anonymous users**
- **Authenticated users**

❑ لكن بنبدأ بإضافة الجروبات اللي عندنا (admin, manager, QA, developer)

---

1. 👑 صلاحيات مجموعة الـ Admin

نضيف **Group** باسم **admin** :

- نديها صلاحية:

✅ **Overall → Administer**

- يعني المجموعة دي تقدر تعمل أي حاجة في **Jenkins** بدون قيود.

💡 لو مش عارف صلاحية معينة بتعمل إيه،

- ممكن توقف بالماوس عليها ويظهر **Tooltip** يشرحها.

---

2. 🧭 صلاحيات مجموعة الـ Manager

نضيف **Group** باسم **manager** :

- الصلاحيات:
  - ✅ **Overall → Read**
  - ✅ **Job → Read**
  - ✅ **View → Read**
  - ✅ **Metrics → Read**
- يعني الـ **Managers** يقدروا يشوفوا الـ **Jobs** والـ **Views** والـ **Metrics**
  - لكن مش يغيروا أي حاجة.

3. 🧪 صلاحيات مجموعة الـ QA

نضيف Group باسم QA :

- صلاحيات محدودة جدًا:
  - ✅ Job → Read
  - ✅ Job → Build

🔹 يعني QA يقدروا يشوفوا الـ Jobs ويعملوا Build ،
لكن:

- ❌ ماينشؤوش Jobs جديدة
- ❌ مايحذفوش Jobs
- ❌ مايعرفوش يدخلوا على الـ Credentials

---

4. 👩‍🏫 صلاحيات مجموعة الـ Developer

نضيف Group باسم developer :

- ✅ Overall → Read
- ✅ Credentials → Create, Update, View
  - (علشان يستخدموا الـ credentials في الـ pipelines)
- ✅ Agents → Provision
- ✅ Jobs ← كل الصلاحيات ما عدا Delete
- ✅ View → Create, Read
- ✅ SCM → Tag
- ✅ Metrics → Read

🔹 يعني الـ Developers عندهم صلاحيات شبه كاملة في الشغل،
  - لكن ماينفعش يحذفوا أي Job لأن دي مسؤولية الـ Admin بس .

🧩 بعد الحفظ – اختبار الصلاحيات عمليًا

1️⃣ الـ Admin (Siddharth) :

- يقدر يعمل كل حاجة، بما فيها Delete Job

2️⃣ المستخدم Tina (عضوة في QA + Developer) :

- ما تقدرش تحذف أي Job (لأن المجموعتين مافيهمش delete)
- تقدر تعمل Build وتشوف Credentials فقط.
- مافيش عندها "Manage Jenkins" في القائمة لأنها مش Admin

3️⃣ المستخدم Bob (عضو في QA فقط) :

- أول ما دخل لقى Access Denied 😅
  السبب: مافيش "Overall Read" صلاحية لمجموعة QA
- بعد إضافة Read ← قدر يشوف الـ Dashboard
- لكنه لا يقدر يحذف أو يعدل أو يشوف Credentials
- يقدر فقط يعمل Build

4️⃣ المستخدم Ali (عضو في Manager)

- يقدر يشوف Dashboard والـ Jobs فقط.
- ما يقدرش يعمل Build أو Delete أو Configure لأن كل حاجة عنده Read-only

✅ النتيجة النهائية:

- الـ **Matrix Authorization Strategy** بيديك أقوى نظام صلاحيات في **Jenkins**

تقدر:

- تتحكم في كل **Permission** لكل **User** أو **Group**
- تمنع حذف أو تعديل **Jobs** من غير إذن.
- وتفصل بين صلاحيات التشغيل، القراءة، والإدارة.

المجموعة	الصلاحيات الأساسية
Admin	كل الصلاحيات
Manager	Read-only لكل الـ Jobs & Views
QA	Build + Read للـ Jobs
Developer	Build + Configure + Credential Management (بدون Delete)

Jenkins CLI-SSH Authentication

🎯 الهدف

- ❑ الجزء ده بيتكلم عن الطريقة التانية لتسجيل الدخول (Authentication) على Jenkins CLI ولكن المرة دي مش عن طريق الـ Username & Password (HTTP Basic Auth) زي قبل كده،
  لكن عن طريق 🔐 SSH Key Pair Authentication
- ❑ يعني هاتتعلم إزاي تعمل Authentication للـ Jenkins CLI باستخدام SSH key pair بدل ما تستخدم username/password كل مرة، وده بيخلي الاتصال أكثر أمانًا واحترافية.

🧱 الخطوة 1 – مراجعة فكرة Jenkins CLI

- ➢ قبل كده استخدمنا CLI (Command Line Interface) بتاع Jenkins عن طريق HTTP Basic Auth
  بصيغة زي دي : `java -jar jenkins-cli.jar -s http://localhost:8080/ -auth user:token list-jobs`
- ➢ لكن دلوقتي هنعمل نفس الكلام باستخدام SSH بدل HTTP

⚙️ الخطوة 2 – معرفة الـ SSH Port اللي Jenkins بيستمع عليه

- ➢ Jenkins ممكن يقبل اتصالات SSH ، لكن افتراضيًا السيرفر بتاع SSH بيكون disabled ❌
  - علشان تعرف إذا كان شغال أو لا، بنستخدم الأمر التالي `curl -lv http://localhost:8080/login | grep SSH`
    - ده بيعرض الـ HTTP Headers اللي راجعة من Jenkins

📍 في الأول، مش هيظهر أي mention لكلمة "SSH" .. لأن السيرفر لسه مش مفعّل .

```
>
< HTTP/1.1 200 OK
< Date: Thu, 06 Feb 2025 11:12:35 GMT
< X-Content-Type-Options: nosniff
< Content-Type: text/html;charset=utf-8
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< Cache-Control: no-cache,no-store,must-revalidate
< X-Hudson: 1.395
< X-Jenkins: 2.462.1
< X-Jenkins-Session: 6f02dd4f
< X-Frame-Options: sameorigin
< X-Instance-Identity: MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA1jdDzG56HPK94SULLWoeNTLhBlT4jge/WOkjYUtHkidmIfwdmMUIscU+JaGS9zmpHd1LGCX1VD0bLcPkSiVc
```

🧩 الخطوة 3 – تفعيل الـ SSH Server داخل Jenkins

1️⃣ افتح Jenkins كـ Admin

2️⃣ روح على :

Manage Jenkins → Security → SSH Server

3️⃣ افتراضيًا هتلاقي "Port: Disabled"

غيرها إلى : Port: Random

أو حدد رقم معين لو حابب تستخدم port ثابت (مثلاً 2222).

4️⃣ اضغط ✅ Apply & Save

🧮 الخطوة 4 – اختبار الـ SSH Endpoint بعد التفعيل

دلوقتي نرجع نكرر نفس الأمر: `curl -lv http://localhost:8080/login | grep SSH`

هتلاقي دلوقتي Jenkins بيرجع Header زي: X-SSH-Endpoint: localhost:41397

📍 ده معناه إن Jenkins بيستمع على Port رقم 41397 لأي اتصال SSH

```
< HTTP/1.1 200 OK
< Date: Thu, 06 Feb 2025 11:13:46 GMT
< X-Content-Type-Options: nosniff
< Content-Type: text/html;charset=utf-8
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< Cache-Control: no-cache,no-store,must-revalidate
< X-Hudson: 1.395
< X-Jenkins: 2.462.1
< X-Jenkins-Session: 6f02dd4f
< X-Frame-Options: sameorigin
< X-Instance-Identity: MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA1jdDzG56HPK94SULLWoeNTLhBlT4jge/WOkjYUtHkidmIfwdmMUIscU+JaGS9
2IsjCXhgUaWKfa9ASxr5VEmhCqvpBlCyqZ6nmvvbSJdDD4q9rHT/4V+xQ7n3GBhNUGBRH9UKKiuDZcrywdS+uOvVp9rSFyuxYhJXfMdCQGmq2AcsquEN9KVWozPXG90Z
gZaRqs0iKmLZGb9OwA2wYJAOndFN8wbm5VaHN69aH/wTvhRg8rCcrlgnXtFsGZNGK6p9hqUzXzVvVZBQIDAQAB
< X-SSH-Endpoint: localhost:41337
< Set-Cookie: JSESSIONID.8638589a=node0o3ab36qnyql7n0h9wsuhukng16.node0; Path=/; HttpOnly
< Content-Length: 2185
< Server: Jetty(10.0.20)
<
```

🧠 الخطوة 5 – تجربة الاتصال عبر SSH

- نحاول ندخل بالـ SSH مباشرة : ssh -p 41397 siddharth@localhost help
  - لكن هنا هتظهر رسالة: Permission denied (publickey)
- وده طبيعي لأننا لسه ما أضفناش Public Key الخاص بالمستخدم داخل Jenkins

---

🔐 الخطوة 6 – إنشاء SSH Key Pair

نولد key pair جديد باستخدام الأمر التالي : ssh-keygen -t rsa -b 4096

- هيطلب منك مكان الحفظ (خليه الافتراضي ~/.ssh/id_rsa)
- ممكن تسيب الـ passphrase فاضي لو مش محتاجه.

بعدها هيظهرلك ملفين:

- id_rsa 🔒 ← المفتاح الخاص (Private Key)
- id_rsa.pub 🔑 ← المفتاح العام (Public Key)

🧭 الخطوة 7 – إضافة الـ Public Key في Jenkins

1️⃣ افتح Jenkins

2️⃣ من فوق اضغط على Username (Siddharth Barahalikar)

3️⃣ اختار Configure

4️⃣ هتنزل لتحت لحد ما تلاقي : SSH Public Keys

5️⃣ الصق محتوى الملف: cat ~/.ssh/id_rsa.pub

6️⃣ اضغط ✅ Apply & Save

🧪 الخطوة 8 – إعادة تجربة الاتصال بعد إضافة المفتاح

دلوقتي نجرب نفس الأمر تاني: ssh -p 41397 siddharth@localhost help

📍 النتيجة:

- مش هتظهر رسالة "Permission denied" تاني.
- الاتصال هينجح ✅
- Jenkins هيرد بقائمة أوامر CLI الممكنة (زي help, list-jobs, build, ...)

🧰 الخطوة 9 – استخدام Jenkins CLI Jar مع SSH Authentication

- الآن نقدر نستخدم الـ Jenkins CLI jar file .. لكن بدل ما نستخدم HTTP Auth ، هنستخدم SSH

الأمر: 👇

```
java -jar jenkins-cli.jar -ssh -user siddharth -s http://localhost:8080 list-jobs
```

📍 الـ CLI هنا هيعمل الاتصال داخليًا عبر الـ SSH port اللي Jenkins محدده.

💡 مقارنة سريعة بين أوضاع الاتصال الثلاثة

Mode	طريقة الاتصال	نوع الـ Auth	الميزة
HTTP	`auth user:token-`	Username + Token	سهل ومباشر
WebSocket	`webSocket-`	Token	جديد وسريع
SSH	`ssh-`	SSH key pair	أكثر أمانًا 💪

✅ الخلاصة

الخطوة	الوظيفة
1️⃣	تأكد إن الـ SSH Server شغال داخل Jenkins
2️⃣	شوف رقم الـ Port اللي بيستمع عليه
3️⃣	أنشئ SSH Key Pair جديد
4️⃣	ضيف الـ Public Key لحسابك داخل Jenkins
5️⃣	استخدم الأمر `ssh` أو `jenkins-cli.jar -ssh` للاتصال

🔚 النتيجة النهائية:

- الـ Jenkins CLI بقى ممكن يتعامل مع الـ Server بأمان تام من غير Password ، باستخدام الـ SSH Key Pair Authentication
- وده مناسب جدًا للـ Automations, CI/CD Scripts, و الـ Secure Agents

Jenkins CLI-Edit a Freestyle Job

🎯 مقدمة

- ❑ الجزء ده بيتكلم عن إزاي نعدل Job في Jenkins باستخدام الـ CLI (Command Line Interface) بدل ما نعدلها من الواجهة الرسومية (GUI)
- ❑ الفكرة ببساطة إن Jenkins بيخزن كل Job عنده كـ ملف XML ، وده بيتضمن إعدادات الـ Job زي الـ commands اللي بتتنفذ، الـ build triggers ، الـ environment variables ، إلخ.
  - فإنت ممكن تنزل الملف ده، تعدله محليًا، وبعد كده ترفعه تاني علشان تحدث إعدادات الـ Job
- ❑ الخطوات الأساسية هنا هي:
  1. نستخدم أمر get-job علشان ننزل ملف الـ XML الخاص بالـ Job
  2. نعدله بإيدينا (نضيف أو نغير commands مثلًا)
  3. نرفع التعديل تاني باستخدام update-job

🧩 شرح تفصيلي

1 التحضير

- ➢ الـ Job اللي هيشتغل عليها اسمها : NPM version test
  - ودي Freestyle Job بسيطة جدًا بتطبع إصدارات NodeJS و NPM باستخدام الأمرين: node -v & npm -v

---

2 هدفنا

المطلوب نعمل الآتي:

- ننزل ملف إعدادات الـ Job (XML)
- نعدل عليه
- نرفعه تاني لـ Jenkins عن طريق CLI

---

3 تحميل إعدادات الـ Job

أمر التحميل بيكون بالشكل ده:

```
java -jar jenkins-cli.jar -s http://localhost:8080/ get-job "NPM version test" > NPM-job.xml
```

🧠 الشرح:

- java -jar jenkins-cli.jar ← لتشغيل Jenkins CLI jar
- -s http://localhost:8080/ ← السيرفر اللي Jenkins شغال عليه.
- get-job ← الأمر اللي بيجيب إعدادات Job
- "NPM version test" ← اسم الـ Job اللي عايزين نجيبها.
- > ← عشان نحفظ المخرجات في ملف بدل ما تتطبع على الشاشة.
- NPM-job.xml ← اسم الملف اللي هيحتوي على إعدادات الـ Job

💡 ملاحظة مهمة عن Authentication

❖ Jenkins CLI بيسمح بطريقتين للمصادقة:

- HTTP authentication (باستخدام username + API token)
- SSH authentication (باستخدام مفتاح SSH)

➢ وهنا، هو استخدم SSH authentication اللي كنا شايفينه في السيشن اللي قبلها.

مثال عملي : `ssh -p 41397 jenkins@localhost get-job "NPM version test" > NPM-job.xml`

🧠 الشرح:

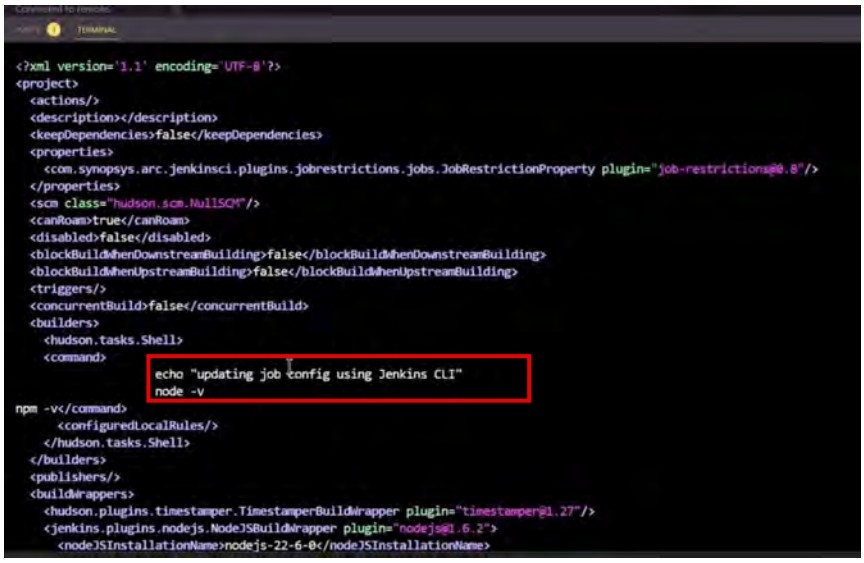
- ssh ← الدخول على Jenkins CLI عبر SSH
- -p 41397 ← البورت اللي Jenkins CLI بيسمع عليه (ممكن يتغير).
- jenkins@localhost ← المستخدم اللي بيتصل بالسيرفر.
- get-job ← اسم الكوماند.
- > ← حفظ الناتج في ملف XML

4️⃣ تعديل ملف الـ Job

- بعد ما نزلنا الملف كـ : NPM-job.xml
- نفتحه في أي محرر نصوص (nano, vim, VSCode...) ونضيف سطر جديد داخل سكريبت التنفيذ، مثل:

```
echo "updating job config using Jenkins CLI"
```

  - ده مجرد أمر بسيط بيطبع رسالة في الـ Console لما الـ Job تشتغل، علشان نقدر نعرف إن التعديل فعلاً اشتغل.



5️⃣ رفع التعديلات إلى Jenkins

- بعد ما حفظنا الملف المعدل، نستخدم أمر:

```
java -jar jenkins-cli.jar -s http://localhost:8080/ update-job "NPM version test" < NPM-job.xml
```

أو بالـ SSH :

```
ssh -p 2222 jenkins@localhost update-job "NPM version test" < NPM-job.xml
```

🧠 الشرح:

- update-job ← بيعمل العكس من get-job، بيرفع الإعدادات الجديدة.
- < NPM-job.xml ← بياخد البيانات من الملف ويبعتها كـ input للـ Jenkins CLI

💬 معلومة : مش هتشوف أي رد على الشاشة بعد الأمر ده، بس التعديل بيتطبق فورًا في الخلفية.

6️⃣ التأكد من التعديل

- ندخل على الـ Jenkins UI ← نفتح الـ Job ← نعمل Refresh
  - هنلاقي إن أمر echo اللي ضفناه ظهر ضمن أوامر الـ build steps
- بعدها ممكن نضغط:
  - Apply ✅ لتأكيد التغيير

- Build Now ▶️ لتشغيل الـ Job

- ثم نفتح الـ Logs ، وهنشوف السطر : updating job config using Jenkins CLI
  - وده معناه إن التعديل اتطبق بنجاح.

🧠 مفاهيم مهمة مرتبطة

◆ الفرق بين get-job و update-job

Command	الوظيفة
`get-job`	ينزل إعدادات الـ Job كملف XML
`update-job`	يرفع التعديل الجديد للـ Job

◆ فائدة الطريقة دي

- تقدر تعمل Backup لإعدادات Jenkins Jobs
- تقدر تعمل Automation لتعديل عشرات الـ Jobs مع بعض (باستخدام Scripts أو GitOps)
- مناسبة جدًا للـ CI/CD Pipelines لما عايز تكرر إعدادات في أكتر من سيرفر.

◆ Best Practices

- قبل أي تعديل : دايمًا خُد نسخة من الـ XML القديم.
- استخدم Git لتخزين ملفات الـ Job configs
- لو بتعدل XML manually ، تأكد من الـ indentation علشان Jenkins مايرفضش الملف.
- ممكن تستخدم --username و --password أو API token لو مش حابب SSH

🧩 أوامر عملية للتدريب

❖ تحميل job config

```
ssh -p 41397 jenkins@localhost get-job "NPM version test" > NPM-job.xml
```

❖ تعديل الملف `vim NPM-job.xml`

❖ رفع التعديلات

```
ssh -p 41397 jenkins@localhost update-job "NPM version test" < NPM-job.xml
```

🔚 الملخص

❑ وضّحنا خطوة بخطوة إزاي:

1. تنزل ملف إعدادات Job من Jenkins بصيغةXML
2. تعدل عليه محليًا (مثلاً تضيف أوامر جديدة)
3. ترفعه تاني باستخدام Jenkins CLI

➢ بكده تقدر تدير كل الـ Jobs في Jenkins من سطر الأوامر بسهولة، وده بيخليك تعمل Automation لإدارة الإعدادات بدل الشغل اليدوي.

Built-in Email Notification

المقدمة:

- الجزء دا بيتكلم عن الميزة المدمجة في Jenkins لإرسال إشعارات البريد الإلكتروني (Email Notifications)
- الفكرة إن Jenkins عنده إمكانية يبعث إيميلات بشكل أوتوماتيكي في حالات معينة — زي لما الـ Build يفشل، أو يبقى unstable، أو لما يرجع stable تاني . وده بيساعدك تعرف بسرعة حالة المشروع بدون ما تدخل Jenkins كل شوية.

الشرح التفصيلي:

1 تفعيل الـ Email Notification على مستوى Job

- في Jenkins ، تقدر تفعّل إرسال الإيميلات من داخل إعدادات أي Job كده:
  - ادخل على الـ Job اللي اسمه npm version test
  - اختار Configure
  - انزل لجزء Post-build Actions
  - دور على خيار اسمه E-mail Notification

هتلاقي إنك تقدر تضيف:

- قائمة Recipients (المستلمين اللي هيوصلهم الإيميل)
- وتقدر كمان تختار إن Jenkins يبعت إيميل منفصل لأي developer كسر الـ Build (في حال الـ push اللي عمله تسبب في مشكلة)

✅ نقطة مهمة : Jenkins هنا بيبعت فقط لما يحصل build فاشل أو غير مستقر أو يرجع Stable

---

2 تجربة الفشل علشان نشوف السلوك

علشان نجرب الميزة:

- جوه الـ Job هنضيف سطر exit 1 في الـ build script علشان نجبره يفشل.
- بعد كده نعمل Build Now ونشوف النتيجة.
- لما الـ build يفشل، المفروض Jenkins يحاول يبعت إيميل.
- لكن لو لسه ماعملناش إعدادات الـ SMTP الأساسية، هنلاقيه بيطلع Error : "Could not connect to host port"
  - وده طبيعي لأن Jenkins مش عارف يبعت إيميلات فعليًا لسه.

---

3 إعداد خدمة الإيميل على مستوى السيستم (SMTP)

- علشان Jenkins يقدر يبعت فعليًا لازم نضبط إعدادات SMTP :
  - نروح Manage Jenkins → System Configuration → System Settings
  - نبحث عن "Mail" أو "E-mail Notification"

هنلاقي إعدادات SMTP تشمل:

- SMTP Server (مثلاً smtp.gmail.com)
- Port (465 لو اخترت SSL)
- اختيار Use SSL أو TLS
- تفعيل SMTP Authentication
- إدخال Username + Password

⚠️ الـ Password هنا مش الباسورد العادي بتاع Gmail ، لازم تستخدم App Password من إعدادات حسابك في Google

---

4 إعداد App Password من Google

❖ Google ما بيسمحش لأي تطبيق يستخدم الباسورد العادي في SMTP لأسباب أمان.

➢ الحل هو إنشاء App Password كالتالي:

- افتح حسابك في Google
- روح لـ Manage your Google Account → Security
- فعّل 2-Step Verification
- بعد كده اختار App Passwords
- اختار نوع التطبيق (مثلاً "Mail") .. والاسم (مثلاً "Jenkins Email Service")
- هيظهرلك password عبارة عن 16 حرف .. انسخه

➢ وبعدين في Jenkins :

- استبدل الباسورد العادي بالـ App Password ده
- اختبر الإرسال بـ Test configuration

النتيجة: هيظهرلك إن الإيميل اتبعت بنجاح 🎉

---

5 ضبط From Address

لو استقبلت الإيميل ولاقيت خانة "From" فاضية، ده لأن الـ Admin Email Address مش متسجل.

الحل:

- ارجع Manage Jenkins → Configure System
- ابحث عن System Admin e-mail address
- اكتب مثلاً admin@jenkins.com

بعد كده جرّب الإرسال تاني هتلاقي خانة From بقت مظبوطة.

---

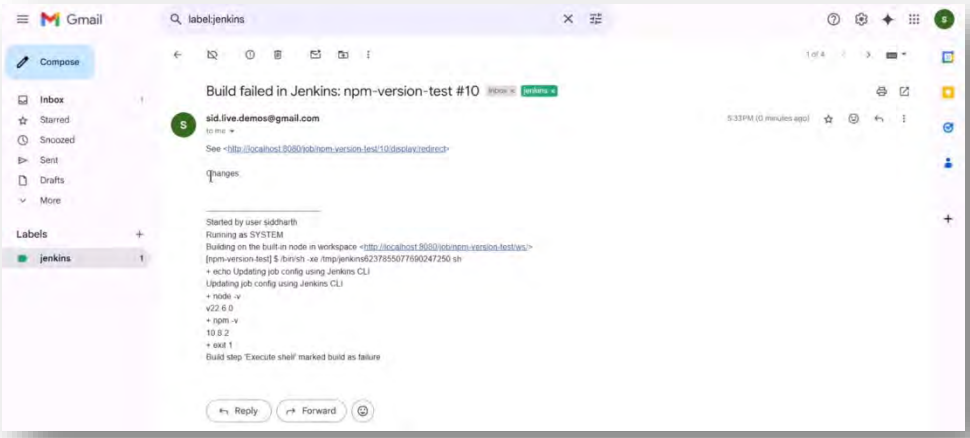
6 تجربة إرسال فعلي وقت فشل الـ Build

❖ نرجع لـ npm version test Job :

- نسيب exit 1 علشان الـ Build يفشل.
- نضغط Build Now

❖ لما يفشل، Jenkins هيبعت الإيميل تلقائيًا.

هيكون محتوى الرسالة:



ده بيخليك تفتح اللينك وتدخل مباشرة على Console Output بتاع build #10

---

7 لما الـ Build يبقى ناجح تاني

لما نحذف سطر exit 1 ونبني من جديد:

- Jenkins هيلاحظ إن الـ Build رجع لحالة Stable
- هيبعت إيميل بعنوان : "Jenkins build is back to normal"

وهيبقى ده بدون لوجز لأن الـ Build ناجح.

💡 إضافات و Best Practices :

- 🧱 ممكن تستخدم Extended Email Plugin (Email-ext) لو عايز تخصيص أكتر مثلا (template ، شرط معين، تنسيق HTML ...)
- 📧 احرص إنك تستخدم App Passwords بدل ما تفتح "Allow less secure apps"
- 🧠 ممكن تربط Jenkins مع SMTP سيرفر داخلي لو شغال في شركة مش بتسمح بـ Gmail مباشر.
- 🍃 تقدر تضيف في Email Template link للـ Build Artifacts أو commit اللي سبّب الفشل.

🧭 الملخص:

شرحنا خطوة بخطوة إزاي:

1. تفعل Email Notification في Jenkins job
2. تزبط إعدادات SMTP علشان Jenkins يقدر يبعث فعليًا.
3. تستخدم App Password من Gmail بدل الباسورد العادي.
4. تختبر إرسال الإيميل وتضبط From Address
5. تتابع سلوك Jenkins لما Build يفشل أو يرجع Stable

النتيجة النهائية:

بمجرد أي build يفشل أو يرجع طبيعي، Jenkins هيبعتلك إشعار إيميل جاهز بالمعلومات واللينك المباشر للـ logs 🔥

Email Extention Notification-Freestyle

🟩 الفكرة العامة:

- ❑ الجزء ده هو تكملة للي شرحناه قبل كده عن الـ Email Notifications في Jenkins ،
- ❑ بس المرة دي بيشرح Email Extension Plugin .. وده بديل أقوى وأكتر مرونة من الإشعارات العادية بتاعة Jenkins
- ❑ الـ plugin ده اسمه "Email Extension Plugin" .. وبيخليك تتحكم في كل تفصيلة تخص الإيميل:
  - إمتى يتبعت .. لمين يتبعت .. محتواه شكله إزاي (نص، HTML، attachments، إلخ)

🧩 الشرح خطوة بخطوة:

1️⃣ تثبيت الـ Plugin

- من داخل Jenkins :
  - o افتح Manage Jenkins → Manage Plugins
  - o من تبويب Available Plugins ابحث عن: Email Extension Plugin
  - o اعمله Install without restart
- بعد التثبيت، Jenkins بيطلب Restart
  - o اعمل restart علشان التغييرات تتطبق.

📘 الـ Plugin موثق كويس جدًا، وبيحتوي على كل الـ examples الخاصة بإعداداته سواء على مستوى النظام أو مستوى المشروع.

2️⃣ إعداد الـ Extended Email Notification على مستوى السيستم

بعد التثبيت ← روح إلى : Manage Jenkins → System Configuration → Configure System

- ابحث عن Extended E-mail Notification

هتلاقي إعدادات SMTP شبه اللي استخدمناها قبل كده:

- SMTP server: smtp.gmail.com
- Port: 465
- Use SSL ✅
- Authentication : استخدم Username + App Password

لو الـ App Password القديم خلاص، لازم تعمل جديد من Google Account زي ما شرحنا قبل كده
(لأن Google بتمنع استخدام الباسورد العادي)

3️⃣ إنشاء App Password جديد (لو لزم الأمر)

في حساب Gmail :

1. افتح Manage your Google Account → Security
2. فعّل 2-Step Verification
3. روح إلى App Passwords
4. اختار التطبيق "Mail" .. والاسم "Jenkins-2"
5. هيتولدلك Password من 16 رقم ← انسخه
6. ارجع Jenkins ← أضفه ضمن Credentials

بعد كده:

- اختار SSL
- اختار Content Type = HTML
- ضيف الـ default recipients (الإيميلات اللي دايمًا توصّلها الإشعارات)

---

4 خصائص إضافية متقدمة

الـ Plugin بيدي إعدادات ضخمة، زي:

- Reply-To list : ممكن تضيف إيميل الرد الافتراضي.
- Allowed Domains : لتحديد الدومينات اللي ممكن يتبعت ليها.
- Default Subject : بيكون فيه Variables زي:
  - $PROJECT_NAME - Build #$BUILD_NUMBER - $BUILD_STATUS
- Attachment Size Limit : تقدر تحدد أقصى حجم للملف، أو تسيبه مفتوح (-1)

✅ كمان ممكن تضيف Default Email Body، واللي بيحتوي على معلومات زي:

- اسم المشروع، رقم الـbuild ، الحالة ... (Success/Failure) إلخ.

---

5 إعداد الـ Triggers (المحفزات)

الـ plugin بيسمحلك تحدد:

- إمتى يتبعت الإيميل:
  - On Failure
  - On Success
  - On Unstable Build
- وتقدر تخصص رسائل مختلفة لكل حالة.

---

6 تفعيل الإيميل في Job معيّنة

❖ نرجع للـ Job اللي اسمها npm version test :

- ادخل على Configure
- شيل الإعداد القديم بتاع E-mail Notification
- وضيف الجديد: Editable Email Notification

❖ هتلاقي نفس الخيارات بس بتخص المشروع ده فقط:

- Project Recipient List :

تقدر تكتب الإيميلات هنا أو تستخدم: $DEFAULT_RECIPIENTS

❖ علشان يجيب الإيميلات اللي اتحطت في إعدادات السيستم.

- Reply-to List, Subject, Content Type, Body : تقدر تسيبهم افتراضيين أو تعدلهم.

---

7 تجربة فشل الـ Build

❖ علشان نشوف الإيميل بيتبعت:

- أضف سطر : exit 1

❖ في الـ build script

- اعمل Build Now
  - هتشوف إن الـ build فشل، والـ console هيعرض:

Email was triggered for failure

Attempting to send email to empty recipient list

❖ هنا فيه مشكلة:

بيقول "Empty recipient list" رغم إنك ضايف إيميل في الإعدادات العامة.

8 سبب المشكلة (والحل)

❖ السبب إن الـ trigger (failure/success) افتراضيًا بيبعت الإيميل لـ: "Developers who caused the build"
ومش بيبعت للـ default recipients

🔧 الحل:

- ارجع للـ Job → Post-build Actions → Editable Email Notification
- افتح Advanced Settings
- عند كل trigger (failure/success)
  - ضيف "Recipient List"
  - واكتب : $DEFAULT_RECIPIENTS

---

9 تجربة من جديد

- اعمل build تاني (مثلا build #13) ← دلوقتي Jenkins هيبعت الإيميل بنجاح 🎉

الإيميل هيكون فيه:

- العنوان (Subject) : npm version test - Build #13 - Still Failing
- المحتوى: الحالة + اللينك المباشر للـ console logs

---

10 إضافة الـ Attachments

لو عايز تبعت ملفات مع الإيميل:

- من إعدادات الإيميل، فعّل:
  - "Attach Build Log"
  - "Compress and Attach Log (optional)"
- لما تعمل build جديد، هتلاقي الإيميل جاي ومعاه ملف build.log كمرفق يحتوي على لوجات الـ build

---

💡 ملحوظات و Tips :

- ✉️ استخدم HTML content لو عايز تبني شكل مخصص للإيميل (جداول – ألوان – روابط).
- 🍃 لو المشروع كبير، ممكن تبعت report أو artifact (زي ملف jar. أو zip) كمرفق.
- ⚙️ لو عندك بيئة إنتاج، استخدم SMTP سيرفر داخلي مش Gmail
- 🔁 تقدر تعمل templates وتستخدمها في مشاريع متعددة.
- 🧩 الـ Plugin ده أساسي في Jenkins pipelines برضه (مش بس Freestyle Jobs)

🧭 الملخص:

شرحنا خطوة بخطوة إزاي:

1. تثبت Email Extension Plugin
2. تضبط إعدادات SMTP على مستوى السيستم
3. تضيف default recipients وتهيئ الـ format
4. تفعّل الإشعارات داخل Job معينة
5. تصلح مشكلة الـ empty recipients
6. تختبر الإرسال وتضيف attachments

❑ النتيجة:

Jenkins بقى يقدر يبعث إيميلات مخصصة بصيغة HTML + ملفات مرفقة + تنبيهات مفصلة عند النجاح أو الفشل.

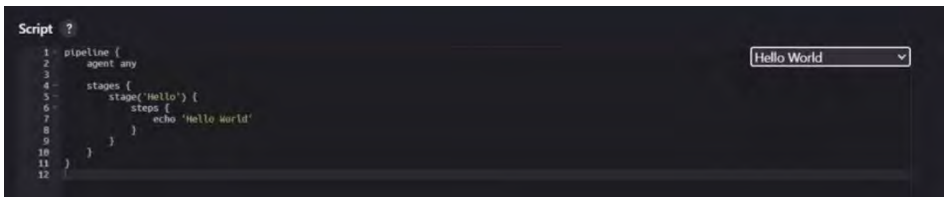
Email Extention Notification-Pipeline

المقدمة:

- الجزء ده بيشرح إزاي تستخدم Email Extension Plugin جوه Declarative Pipeline في Jenkins ، بحيث تبعت إيميلات أوتوماتيك بعد الـ build (سواء نجح أو فشل) وتقدر كمان تبعت ملفات مرفقة أو HTML formatted reports
- يعني باختصار:

هنحوّل فكرة الـ email notifications من "إعداد يدوي جوه الـ GUI" ← إلى "كود جوه الـ pipeline نفسه."

الشرح التفصيلي خطوة بخطوة:



1 إنشاء Pipeline جديد

من داخل Jenkins :

- اضغط New Item
- سمّي الـ Job مثلاً: Pipeline Email Notification
- اختار النوع : Pipeline
- في خانة الـ Script ممكن تكتب سكريبت بسيط في الأول للتجربة (زي"Hello World")

➢ الهدف دلوقتي بس نجهّز الـ Job اللي هنحط فيها سكريبت الإيميلات بعدين.

---

2 توليد سكريبت الإيميل (Email Extension Step)

❖ عشان نكتب الكود صح:

- اضغط Pipeline Syntax من تحت.
- اختار "Declarative: Directive Generator" أو "Snippet Generator"
- في خانة الـ sample step اكتب emailext: Extand Email

➢ ده هو الاسم الرسمي للـ pipeline step اللي بيستخدم Email Extension Plugin

❖ هتبدأ تملى الحقول:

- To : ممكن تكتب developers@example.com أو تسيبها متغيّر $DEFAULT_RECIPIENTS
- Subject : تكتب [Build Status - $BUILD_STATUS]
- Body : تكتب [Build Number: $BUILD_NUMBER] وهكذا

وبعدين اضغط Generate Pipeline Script، هيطلعلك كود شبه ده:

```
emailext(
    to: '$DEFAULT_RECIPIENTS',
    subject: '$DEFAULT_SUBJECT',
    body: '$DEFAULT_CONTENT'
)
```

❑ وده هو الشكل الأساسي اللي بيستخدمه Jenkins Pipeline

```
emailext attachLog: true, body: '$DEFAULT_CONTENT', subject: '$DEFAULT_SUBJECT', to: '$DEFAULT_RECIPIENTS'
```

---

3 تجهيز السكريبت النهائي (Pipeline كامل)

خليني أشرح كل جزء فيه بالتفصيل 👇

Script ?

```
pipeline {
    agent any

    stages {
        stage('Hello') {
            steps {
                sh "echo 'Testing pipeline-email-notifiication' > testFile.txt"
            }
        }
    }

    post {
        always {

            emailext (
                attachLog: true,
                body: '$DEFAULT_CONTENT',
                subject: '$DEFAULT_SUBJECT',
                to: '$DEFAULT_RECIPIENTS',
                attachmentsPattern: 'testFile.*'
            )
        }
    }
}
```

Use Groovy Sandbox ?

🧱 القسم الأول pipeline { ... } :

ده الهيكل الرئيسي للـ pipeline (Declarative format)

- agent any :
  معناها شغّل الـ build على أي Agent متاح في Jenkins
  (سواء master أو slave node)

🏗️ القسم الثاني stages :

هنا عندنا Stage واحدة بس:

الشرح:

- echo 'Hello World!' ← مجرد طباعة في الـ Console Output
- sh 'echo "Test file created" > test_file.txt' ← أمر shell بيعمل ملف اسمه test_file.txt
  فيه النص "Test file created"

🧠 الفكرة : نولّد ملف علشان نجرب نبعته كمرفق في الإيميل بعدين.

🧩 القسم الثالث post { always { ... } } :

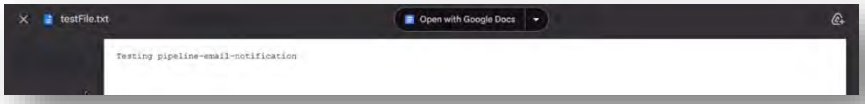
- ده الجزء اللي بيتنفذ بعد ما يخلص الـ build ، بغض النظر إذا كان ناجح أو فاشل.
- وده المكان المثالي لإرسال الإيميلات.
- كل باراميتر هنا ليه دور محدد:

الخيار	المعنى
`attachLog: true`	يضيف ملف لوجات الـ build كـ attachment تلقائيًا.
`'attachmentsPattern: 'test_file.txt`	يبعت أي ملف اسمه test_file.txt كمرفق.
`'to: '$DEFAULT_RECIPIENTS`	يبعته للإيميلات الافتراضية المحددة في إعدادات السيستم.
`'subject: '$DEFAULT_SUBJECT`	يستخدم عنوان البريد الافتراضي (زي اسم المشروع + رقم الـ build).
`'body: '$DEFAULT_CONTENT`	يستخدم النص الافتراضي اللي في إعدادات السيستم (المحتوى العادي).

📄 النتيجة بعد التشغيل

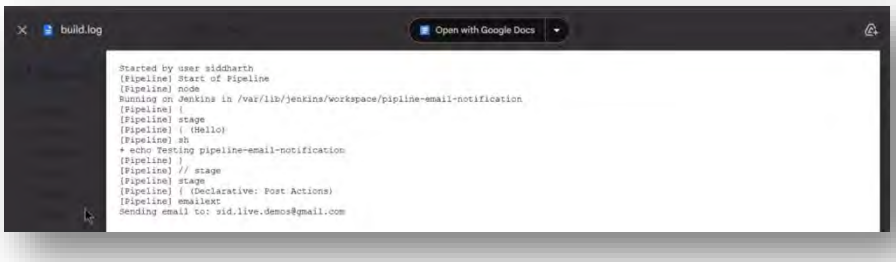
لما تشغّل الـ pipeline :

- Jenkins هيعمل Stage اسمها "Hello"
- هيولّد الملف test_file.txt
- بعد كده الـ Post section هيشتغل، ويبعت الإيميل.



الإيميل اللي هيوصل هيحتوي على:

- الـ Build Log كمرفق.
- ملف test_file.txt كمرفق تاني.
- عنوان البريد الافتراضي.
- نص بسيط في الـ body



🧱 الجزء الأخير HTML Email Format :

لو عايز تخلي الإيميل شكله احترافي أكتر:

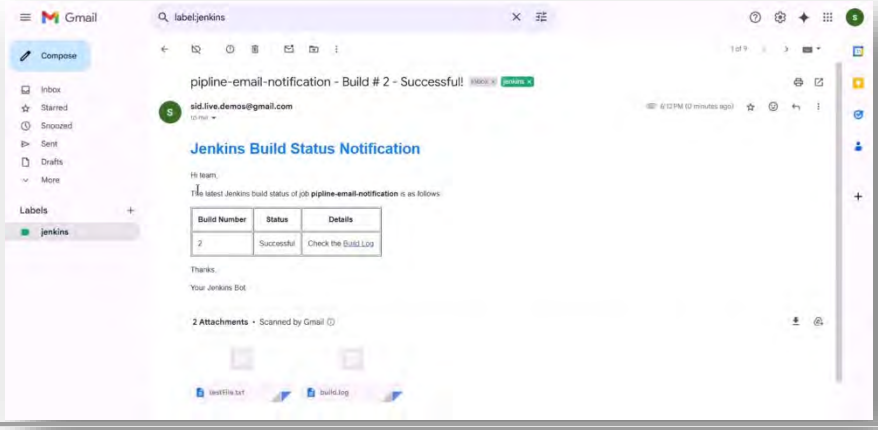
- روح لـ Manage Jenkins → Configure System → Extended E-mail Notification
- في Default Content، بدل النص العادي بـ HTML template زي ده:

```
<html>
  <body>
    <h3>Build Report</h3>
    <table border="1" cellpadding="5">
      <tr><th>Project</th><td>${PROJECT_NAME}</td></tr>
      <tr><th>Build #</th><td>${BUILD_NUMBER}</td></tr>
      <tr><th>Status</th><td>${BUILD_STATUS}</td></tr>
      <tr><th>URL</th><td><a href="${BUILD_URL}">View Logs</a></td></tr>
    </table>
  </body>
</html>
```

🟦 مهم جدًا:

- لازم تغيّر الـ Content Type من Plain text إلى HTML علشان Jenkins يعرف يرندر الـ HTML بدل ما يعرضه كنص خام.

➕ بعد كده لما تشغّل الـ job تاني، الإيميل هيجي شكله جدول منسّق، فيه الحالة واللينك المباشر للـ logs



⚙️ Best Practices & Tips

1. ✉️ استخدم attachLog: false ← في البيئات الكبيرة لتوفير مساحة (إلا لو فعلاً محتاج اللوجات).
2. 📁 استخدم pattern زي **/*.zip لو بتبعت تقارير أو build artifacts
3. 🔐 استخدم App Password أو SMTP داخلي آمن بدل Gmail الشخصي.
4. 🧩 تقدر تخصص body لكل حالة (Success / Failure) باستخدام:

```
post {
success {
emailext(subject: "Build Passed", body: "Everything OK")
}
failure {
emailext(subject: "Build Failed", body: "Check Jenkins logs")
}
}
```

5. 💡 استخدم المتغيرات الجاهزة زي:
   - ${BUILD_NUMBER}
   - ${BUILD_STATUS}
   - ${BUILD_URL}

🧭 **الملخص:**

كدا نبقي شرحنا :

1. **إزاي تولّد كود الـ Email Extension داخل Pipeline**
2. **إزاي تبني Pipeline بسيط فيه Stage واحدة.**
3. **إزاي تضيف Post Action تبعت إيميل تلقائي بعد الـ build**
4. **إزاي تبعت مرفقات (Logs أو ملفاتك)**
5. **إزاي تغيّر شكل الإيميل لـ HTML report**

📬 **النتيجة النهائية:**

كل build هيبعت إيميل منسّق فيه الحالة، اللوجات، والملفات المطلوبة بشكل تلقائي 🔥

NodeJS App Overview

🎯 الفكرة العامة:

الجزء دا بيتكلم عن مقدمة بسيطة لـ Node.js، وبيشرح إيه هي، بتستخدم في إيه، وازاي بنشغل وندير مشروع Node.js باستخدام أوامر زي npm test, npm install و npm start

---

🧩 شرح النقاط الأساسية :

➢ تعريف Node.js :

- Node.js عبارة عن بيئة تشغيل مفتوحة المصدر (Open-Source Runtime Environment) بتسمح إنك تشغل كود JavaScript خارج المتصفح.
- يعني بدل ما JavaScript تشتغل جوه الـ browser زي المواقع، هنا بتقدر تكتب بيها برامج بتشتغل على السيرفر.

➢ الاستخدامات:

Node.js بتخليك تقدر تبرمج الـ Front-end والـ Back-end بنفس اللغة (JavaScript) ، وده بيوفر مجهود كبير للمطورين لأنهم بيستخدموا لغة واحدة بس في الجانبين.

➢ المحرك الداخلي:

Node.js مبنية على محرك V8 Engine بتاع متصفح Google Chrome .. وده اللي بيخلّيها سريعة جدًا في تنفيذ كود JavaScript

---

➢ أنظمة التشغيل المدعومة : ممكن تثبّت Node.js على أي نظام تشغيل:

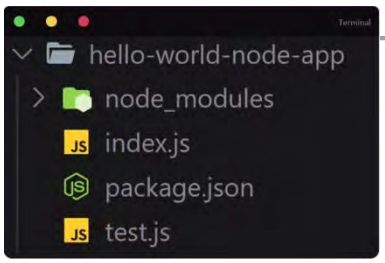
Windows

macOS

Linux

➢ NPM (Node Package Manager) :

- أول ما تثبّت Node.js ، بيتثبت معاها npm تلقائيًا . وده عبارة عن الـ Packet Manager (مدير الحزم) ، اللي بيستخدمه المطورين عشان:
  - ينزّلوا المكتبات الجاهزة (packages/libraries)
  - يديروا الـ dependencies الخاصة بالمشروع.
  - يشاركوا أكوادهم مع الآخرين.



➢ ملف package.json :

- ده أهم ملف في مشروع Node.js لأنه بيحتوي على كل البيانات الأساسية للمشروع زي:
  - اسم المشروع وإصداره.
  - المكتبات اللي بيعتمد عليها (dependencies)
  - السكريبتات اللي بتشغّل أو تختبر الكود.

➢ تنصيب الـ Dependencies :

- علشان تنزّل المكتبات اللي المشروع محتاجها، بتستخدم الأمر: npm install
- بعد ما يتنفّذ، بيتكوّن فولدر اسمه node_modules فيه كل المكتبات اللي المشروع بيعتمد عليها.

- الملفات الأساسية في المشروع:
  - index.js : فيه الكود الأساسي (business logic)
  - test.js : فيه أكواد لاختبار المشروع والتأكد إن كل حاجة شغالة صح.

- تشغيل الاختبارات : عن طريق الأمر : npm test
  - بيشغّل كل الـ test cases الموجودة في test.js

- تشغيل التطبيق : عن طريق الامر : npm start
  - بيشغّل التطبيق فعليًا.
  - بعدها تقدر تفتح المتصفح وتدخل على الـ port اللي التطبيق شغال عليه (مثلاً http://localhost:3000)

💡 الخلاصة:

- ❑ Node.js بتخليك تبني تطبيقات كاملة باستخدام JavaScript فقط.
- ❑ npm بيسهّل إدارة الحزم.
- ❑ كل مشروع Node.js بيمشي بنفس البنية دي:
  - package.json
  - index.js
  - test.js
  - node_modules/

Run and Test NodeJS App on Local Machine

sidd-harth/solar-system-gitea-jenkins-advanced

🎯 **الفكرة العامة:**

في الجزء ده، هانشرح خطوة بخطوة:

- إزاي نشغّل تطبيق Node.js بسيط (اسمه Solar System) على جهاز محلي أو VM
- إزاي نعمل clone للمشروع من GitHub
- إزاي نشغّل الاختبارات (Unit Tests) ونحسب الـ Code Coverage
- وإزاي نشغّل التطبيق ونشوفه من المتصفح.
- وأخيرًا، بيجهّزنا علشان نعمل نفس الخطوات دي أوتوماتيك في Jenkins في السيشن الجاية.

◆ أولًا: تشغيل التطبيق يدويًا

1. دلوقتي هانشغّل التطبيق manually على جهاز أو VM
   - بعد كده السيشن الجاية هتكون عن automation باستخدام Jenkins
2. المشروع اسمه Solar System App، وموجود على GitHub (بس بعدين هيتم نقله لـ GitLab)
3. المشروع يحتوي على:
   - HTML Frontend
   - app.js ← الكود الرئيسي
   - app.controller.js ← فيه الـ Business Logic (العمليات الرئيسية زي الاتصال بقاعدة البيانات)
   - ملفات مساعدة README + فيه تفاصيل التشغيل.

◆ ثانيًا: تجهيز المشروع

1. بيبدأ ينسخ الـ GitHub endpoint ويعمل: git clone <repository-url>
   - وبعدين يدخل على المجلد: cd solar-system-gitea/
2. بيشرح إن الـ structure بتاع المشروع نفس اللي شفناه قبل كده:
   - package.json
   - app.js
   - app.controller.js
   - app.test.js
   - Dockerfile
   - index.html

◆ ثالثًا: شرح أهم الملفات

1. package.json :
   - فيه بيانات المشروع (name, version, dependencies)
   - أهم المكتبات اللي مستخدمة:
     - express ← لإنشاء REST APIs
     - mocha + junit ← لاختبار الكود (Unit Tests)
     - mongoose ← للاتصال بـ MongoDB
     - nyc ← لقياس Code Coverage
     - serverless-http ← هيتستخدم بعدين لما ينشروا الكود على AWS Lambda

2. الـ Scripts داخل package.json :
   - "start" ← لتشغيل التطبيق.
   - "test" ← لتشغيل الـ Unit Tests
   - "coverage" ← لحساب نسبة الكود المغطاة بالاختبارات.
   - كمان محددين threshold = 90%، يعني لازم التستات تغطي على الأقل 90% من الكود.

◆ رابعًا: ملف app.js

1. الملف app.js فيه الكود اللي:
   - بيربط المشروع بمكتباته.
   - بيتصل بقاعدة بيانات MongoDB عن طريق Mongoose
   - بيحتاج 3 متغيرات بيئية (Environment Variables) :
     - MONGO_URI
     - USERNAME
     - PASSWORD
   - ➢ دول المفروض هيتمرروا بعدين من Jenkins Credentials

◆ خامسًا: تشغيل المشروع

1. أول حاجة: npm install
   - بتنزل كل المكتبات وتعمل مجلد node_modules
2. بعد كده : npm test
   - بيبدأ يشغل التستات، لكنه فشل في الأول لأنه مفيش بيانات الاتصال بـ MongoDB (URI, username, password)
3. للتجربة، هو عمل hardcoded للبيانات في الكود نفسه اللي في ملف package.json (كتبهم جوه app.js بشكل مؤقت)

```
package-lock.json
package.json
README.md
.bash_history
.bashrc
.cloud-locale-test.skip

mongoose.connect('mongodb+srv://supercluster.d83jj.mongodb.net/superData', {
    user: 'superuser',
    pass: 'SuperPassword',
    useNewUrlParser: true,
    useUnifiedTopology: true
```

4. بعد التعديل، شغّل تاني : npm test
   - هتلاقي الاختبارات نجحت ✅
5. علشان يتأكد إن الكوماند اشتغل بنجاح كتب: echo $?
   - النتيجة كانت 0 (يعني الكوماند نجح)
6. كمان اتولد ملف جديد: test-results.xml
   - وده عبارة عن تقرير JUnit Report فيه نتائج الاختبارات.

◆ سادسًا: الـ Code Coverage

1. بعد كده شغّل : npm run coverage
   - ده بيشغّل أداة nyc علشان يعرف نسبة الكود المغطاة بالتستات.
2. المفروض الـ threshold كان 90%، لكن اللي تحقق فعليًا 79 % ❌
   - وبالتالي لما عملنا echo $? ← رد علينا بـ 1 يعني الكوماند فشل .. وده متعمّد علشان بعدين يوريك في Jenkins إزاي نتعامل مع أخطاء زي دي بدون ما توقف البايبلاين.
3. بعد التشغيل، اتولد فولدرين : /coverage & /nyc_output
   - فيهم تقارير مفصّلة (coverage summary, LCOV report)

```
ERROR: Coverage for lines (79.06%) does not meet global threshold (90%)

File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |   79.54 |    33.33 |      70 |   79.06 |
 app.js   |   79.54 |    33.33 |      70 |   79.06 | 23,49-50,58,62-67
----------|---------|----------|---------|---------|-------------------
```

◆ سابعًا: تشغيل السيرفر

1. أخيرًا شغّل التطبيق : **npm start**
   - وده بيشغّل السيرفر على **port 3000**
2. فتح المتصفح على **localhost:3000** (أو IP الـ VM) هايظهرلك التطبيق البسيط .. واجهة HTML فيها معلومات عن الكواكب.

◆ ثامنًا: وظائف التطبيق

1. الـ Frontend (index.html) بيعرض:
   - قائمة بالكواكب ومعلومات عنها (صورة + وصف).
   - اسم السيرفر اللي التطبيق شغال عليه (jenkins-controller)
   - إمكانية البحث عن كوكب بالرقم.
2. الـ Endpoints الموجودة:
   - /os ← بيرجع اسم الجهاز أو الـ pod (لو على Kubernetes)
   - /live ← بيأكد إن التطبيق شغال (liveness)
   - /ready ← بيأكد إنه جاهز للاستقبال (readiness)

◆ تاسعًا: الخطوة الجاية

🎯 في الجلسة الجاية، هيعملوا automation لكل الخطوات دي في Jenkins pipeline :

- تثبيت الـ dependencies
- تشغيل التستات
- حساب Code Coverage
- تشغيل التطبيق ن ونشره (Deploy) على بيئات مختلفة

Understanding DevOps Pipeline

🎯 المقدمة — الفكرة العامة

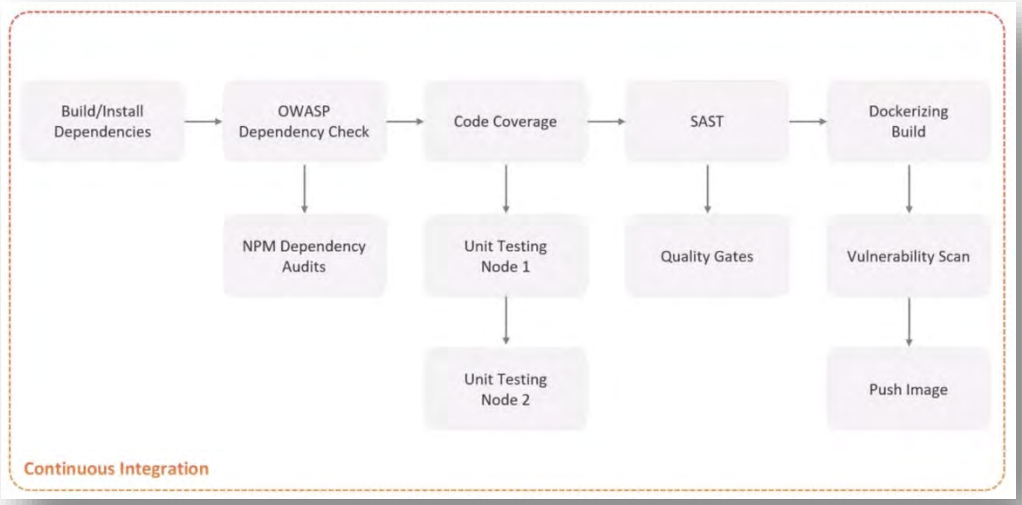
- ❑ الجزء ده بيتكلم عن Jenkins DevOps Pipeline متكامل هيتم بناؤه خلال الكورس خطوة بخطوة.
- ❑ الفكرة إنه بيشرح المراحل الكاملة اللي هتمر بيها أي Application من أول ما الـ Developer يكتب الكود لحد ما يتنشر في بيئة Production ، واللي بتنقسم لـ 4 مراحل رئيسية : (خلي بالك دا سؤال انترفيوهات)

1. Continuous Integration (CI)
2. Continuous Deployment (CD)
3. Continuous Delivery (CDel)
4. Post-Build Stage

- ➢ وفي النهاية بيشرح كمان التكامل مع أدوات خارجية زي (SonarCloud, ArgoCD, Snyk, OWASP ZAP, Slack, AWS (EC2 + Lambda + S3

✅ أولًا(CI) Continuous Integration :

❖ دي أول مرحلة في البايبلاين، وهيبدأ شرحها بالتفصيل لأنها الأساس اللي بيتبني عليه كل حاجة بعد كده.



1. 📌 Feature Branch Approach

- كل مطور لما يبدأ يشتغل على تعديل جديد، بيعمل فرع (Feature Branch) من الـ Main Branch
  مثلًا : git checkout -b feature/add-login
- بعد ما يخلّص التعديلات، بيعمل: git push origin feature/add-login
- هنا Jenkins بيتابع الـ repository ، ولما يشوف إن فيه push جديد على فرع Feature ← بيبدأ الـ CI Pipeline تلقائيًا.

💡 فائدة النظام ده:

بيخلي كل تعديل يتبنى ويختبر لوحده قبل ما يتدمج مع الكود الرئيسي، فبيقلل الـ bugs اللي ممكن توصل للـ main branch

---

2. 📌 Stage 1 – Install Dependencies

- ➢ بما إن المشروع Node.js App، أول خطوة لازم ننزل الـ dependencies : npm install
  - o ودي بتحمّل المكتبات اللي محتاجها المشروع من package.json

---

3. 📌 Stage 2 – Check Dependency Vulnerabilities

➢ بنراجع المكتبات اللي نزلناها لو فيها ثغرات أمنية باستخدام طريقتين:

- **OWASP Dependency Checker**
  - أداة بتفحص الـ dependencies وتقارنها بقاعدة بيانات الثغرات المعروفة (CVE Database)
  - ممكن تشغّلها في Jenkins Step كده:
  - `dependency-check.sh --project "solar-system" --scan .`
- **NPM Audit**
  - أداة مدمجة مع npm : `npm audit`
  - بتطلع لك تقرير فيه نوع الثغرة ومستوى الخطورة.

---

4. 📌 Stage 3 – Unit Testing + Code Coverage

- نشغّل التستات : `npm test`
- بعدين نقيّس الـ coverage : `npm run coverage`
- Jenkins بيتحقق إن الـ coverage ≥ 90% (زي ما كان محدد في الـ package.json)
  - لو أقل ← الـ Build هيفشل.

💡 **نقطة مهمة:**

تفعيل شرط الـ coverage في Jenkins خطوة ممتازة علشان تمنع الكود الناقص من إنه يعدي الـ CI

---

5. 📌 Stage 4 – Static Analysis (SonarCloud Integration)

- بيعمل تحليل للكود (Code Quality) عن طريق SonarCloud أو SonarQube
- الأدوات دي بتكشف مشاكل زي:
  - الـ Code Smells
  - الـ Security Vulnerabilities
  - الـ Duplications

بعد التحليل، Jenkins بيعمل مقارنة بنتيجة الـ Quality Gate :

- ✅ لو نجح ← يكمل
- ❌ لو فشل ← الـ Build يوقف هنا.

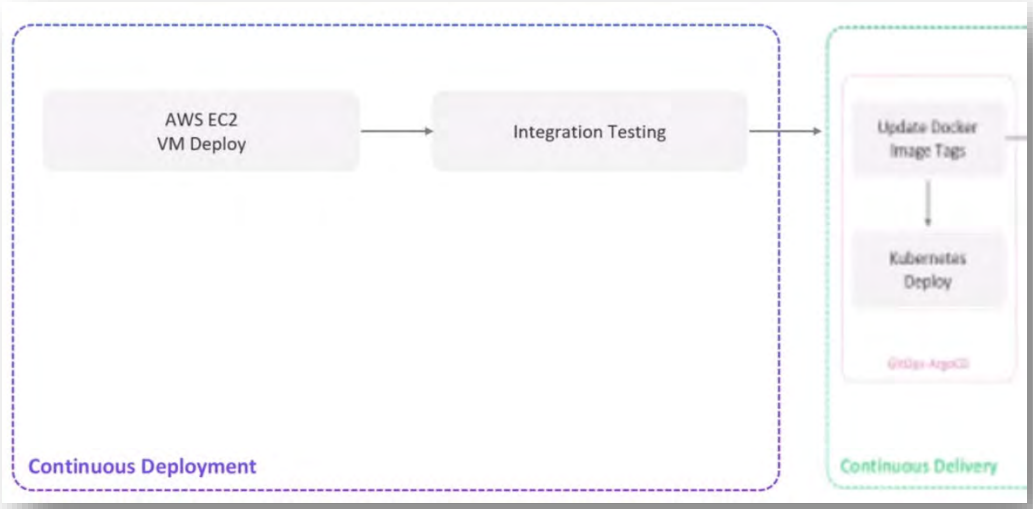
---

6. 📌 Stage 5 – Containerization + Security Scan

- نبدأ نبني Docker Image : `docker build -t solar-system:latest .`
- بعدين نفحصها ضد الثغرات: `snyk test --docker solar-system:latest`
- لو التحليل تمام ← ندفعها للـ Registry (زي DockerHub أو AWS ECR) :
  - `docker push <registry>/solar-system:latest`

🚀 ثانيًا(Continuous Deployment (CD :

❑ بعد ما الصورة (Docker Image) تترفع، بنبدأ ننشرها (Deploy) :



7. 📌 Deploy to AWS EC2

- بنشغّل الـ container جوه EC2 instance : **docker run -d -p 3000:3000 solar-system:latest**
- بعد التشغيل، Jenkins بيعمل Integration Testing يتأكد إن:
  - الـ app شغال.
  - الـ endpoints (زي /live أو /ready) بترجع استجابة سليمة.

---

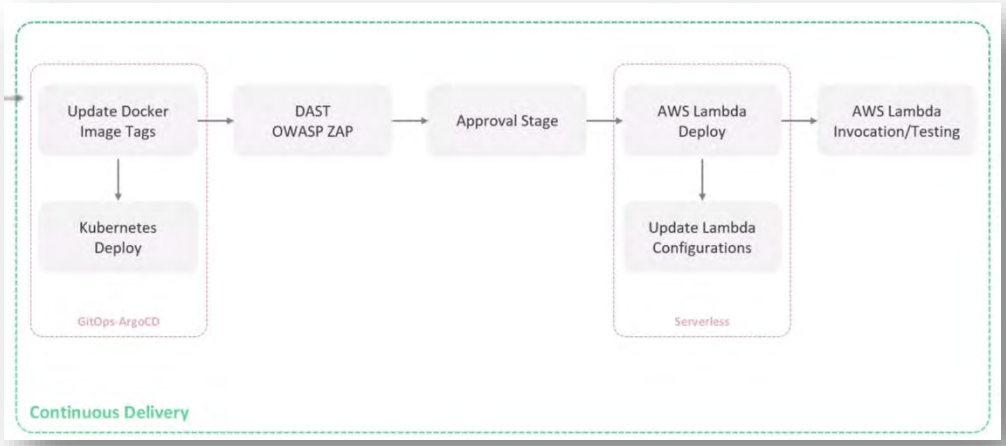
8. 📌 Pull Request & Merge

بعد ما المطور يتأكد إن كل حاجة تمام على فرع الـ Feature :

- بيرفع Pull Request عشان يدمج الكود في الـ Main Branch
- وقتها Jenkins بيبدأ مرحلة Continuous Delivery

🧭 ثالثًا (Continuous Delivery (CDel :

❑ دي المرحلة اللي بتحاكي بيئة Production الفعلية.



9. 📌 Deploy to Kubernetes via GitOps

- Jenkins هيحدث الـ Docker Image Tag داخل ملفات (manifest files) YAML
- Argo CD هيلاحظ التغيير وتسحب النسخة الجديدة وتنشرها تلقائيًا في الـ Cluster

GitOps Concept :

بدل ما Jenkins هو اللي ينشر مباشرة، إحنا بنخلي ArgoCD يتابع الـ Git repo ويعمل Sync أوتوماتيك لأي تحديث في الإعدادات أو الـ images .

---

10 . Dynamic Application Security Testing (DAST)

- بعد النشر، بنجري فحص حيّ باستخدام OWASP ZAP علي التطبيق اللي شغال فعلًا.
- ZAP بيحاول يحاكي هجمات فعلية (مثل SQL Injection أوXSS) ويطلع تقرير بالمخاطر.

---

11 . Merge & Approval

- بعد الفحص، المشرفين أو Product Leads بيراجعوا الـ Pull Request
- لما يوافقوا، بيتدمج الكود رسميًا مع الـ main branch

بعد الدمج (Main Branch Pipeline)

12 . Manual Approval Stage

- Jenkins بيبدأ Pipeline جديدة أوتوماتيك بعد الدمج.
- فيها مرحلة Approval لازم Admin يوافق يدويًا من داخل Jenkins Dashboard

---

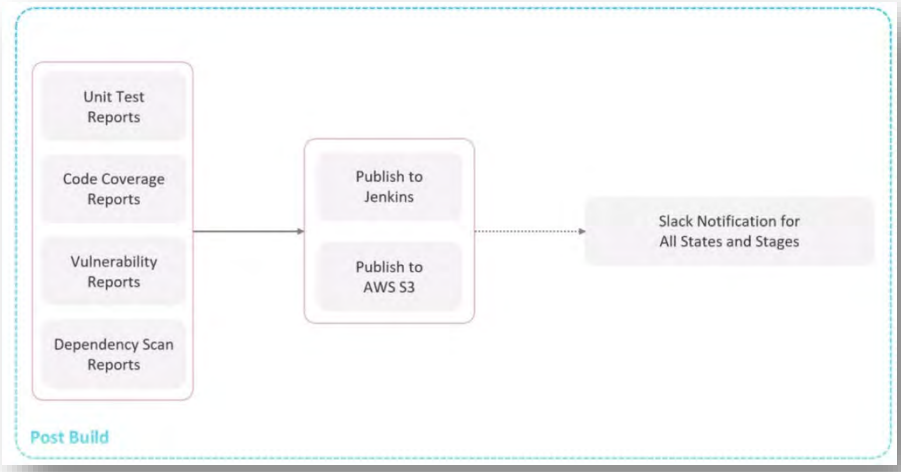
13 . Deploy to AWS Lambda

- بعد الموافقة ، Jenkins بيبدأ نشر النسخة الجديدة كـ Lambda Function :
  - بيحدّث متغيرات البيئة (Environment Variables)
  - بيضبط الـ Function URL
- بعد النشر، Jenkins بيعمل اختبارات بسيطة للتأكد إن الـ Lambda شغالة:
  - aws lambda invoke --function-name solar-system response.json

---

رابعا Post-Build Stage :

❑ هنا Jenkins بيجمع كل التقارير اللي اتولدت أثناء المراحل السابقة:



- Unit Test Reports
- Code Coverage Reports
- Vulnerability Scans
- Dependency Check Reports

وبيعمل حاجتين:

1. أرشفة داخل Jenkins (Artifacts)
2. رفعها على AWS S3 Bucket علشان تبقى محفوظة وممكن نرجع لها بعدين.

Notifications Integration :

❑ آخر خطوة هي التكامل مع Slack :

- Jenkins هيبعت إشعارات للـ Slack Channel لما يحصل:
  - Build Success
  - Build Failure
- ودي طريقة ممتازة لمتابعة حالة البايبلاين في الوقت الحقيقي من غير ما تفتح Jenkins Dashboard

---

الملخص الشامل

CI ← بناء الكود + اختبار + فحص جودة + إنشاء Docker Image

CD ← نشر الصورة على EC2 + اختبار التكامل.

CDel (GitOps) ← نشر تلقائي على Kubernetes عبر ArgoCD + اختبار الأمان الديناميكي.

Production Stage ← نشر نهائي على Lambda بعد الموافقة اليدوية.

Post-Build ← تجميع وتحليل التقارير + إرسال إشعارات Slack

أفضل الممارسات اللى بأكدلك عليها :

- استخدم Feature Branch Workflow دايمًا.
- اربط Jenkins بـ SonarCloud و Snyk و OWASP
- فعل Quality Gates و Code Coverage Thresholds
- استخدم GitOps لنشر الكود بدل الـ manual deploy
- اجمع كل الـ Reports وارفعها على S3 لحفظ السجلات.
- فعّل Slack Notifications علشان المتابعة السريعة.

Install&Setup NodeJS Build Tool

💡 الفكرة العامة:

- ❑ الجزء ده بيتكلم عن إزاي نثبت ونستخدم Node.js داخل Jenkins ،
- ❑ وبيوضح الخطوات من أول التحقق إن Node.js شغال فعلاً على السيرفر، لحد إعداد الـ Node.js plugin جوه Jenkins واستخدامه في مشاريعنا (freestyle project)

🧩 الجزء الأول – التحقق من وجود Node.js على السيرفر

- هاندخل على Jenkins اللي عندنا أصلاً جاهز ومثبت.
- هاتروح على الـ virtual machine اللي Jenkins شغال عليها.
- شيل أي repository قديمة مش محتاجها (clean setup) ← بـ rm -rf
- بعدها نفّذ الأمر : node -v
  - وطلعنا إن نسخة Node.js = 20.16.0
- وكمان: npm -v
  - وطلعت نسخة npm = 10.8.1

✅ كده اتأكد إن Node.js و npm مثبتين على نفس الجهاز اللي Jenkins عليه.

⚙️ الجزء الثاني – التحقق من حالة Jenkins

- شاف إذا كان Jenkins شغال بـ : systemctl status jenkins
  - وفعلاً طلع Active (running)
  - فده معناه إن الـ Controller بتاع Jenkins شغال على نفس الماكينة اللي فيها Node.js
  - بالتالي Jenkins يقدر يستخدم أوامر Node/npm بشكل مباشر من النظام نفسه.

🧱 الجزء الثالث – تجربة Node.js داخل Jenkins Job

- راح على واجهة Jenkins ، وعمل Freestyle Project جديد اسمه: npm version test
- جوه الـ Build step، اختار: Execute shell .. وكتب فيه : node -v & npm -v
- عمل Save ← Build
- النتيجة ✅ الـ build نجح، وظهر له نفس الإصدارات اللي كانت على السيرفر.
  - السبب إن Jenkins استخدم Node.js اللي موجود في النظام مباشرة.

🧱 الجزء الرابع – المشكلة المحتملة

- لكن أوقات الـ Jenkins job مش بتشتغل على نفس السيرفر اللي عليه Jenkins ،
- يعني ممكن تشتغل على Agent تاني (worker node) مفيهوش Node.js

➢ في الحالة دي لازم نركّب Node.js plugin داخل Jenkins نفسه.

🧰 الجزء الخامس – تثبيت Node.js Plugin في Jenkins

1. راح على : Manage Jenkins → Plugins → Available plugins
2. بحث عن Node.js .. لقاها (الإصدار 1.62) .. فاختار Install
3. بعد التثبيت ✅ راح على : Manage Jenkins → Tools
4. والمرة دي ظهرت له قائمة جديدة اسمها: NodeJS Installations

🧩 الجزء السادس – إعداد Node.js داخل Jenkins Tools

- ضغط على Add new Node.js
- سمّى التول مثلاً : Node.js-22.6.0
- اختار الإصدار 22.6.0
- وسيب باقي الخيارات Default (بيسحب النسخة من الموقع الرسمي)
- حفظ الإعدادات ✅

🧰 الجزء السابع – استخدام التول داخل المشروع

- رجع تاني للمشروع اللي كان عامله (npm version test)
- دخل على Configure
- نزل لتحت شوية، ولقى خيار جديد تحت Build Environment اسمه:
- Provide Node and npm bin/ folder to PATH
- فعّله ✅ واختار منه النسخة اللي سمّاها قبل كده (Node.js-22.6.0)
- حفظ المشروع، وبنى تاني.

⚙️ الجزء الثامن – إيه اللي بيحصل جوه الكواليس؟

- المرة دي الـ build أخد وقت أطول شوية.
- السبب إن Jenkins لأول مرة بيعمل:
  - تحميل Node.js من الموقع الرسمي.
  - فك الضغط عنه.
  - تخزينه في المسار : Jenkins/tools/NodeJS/Node.js-22.6.0
- بعد ما خلص التحميل، شغّل نفس أوامر node -v و npm -v
- وطلعت النسخة الجديدة:
  - Node.js v22.6.0
  - npm v10.8.2

✅ كده Jenkins اشتغل باستخدام Node.js اللي هو نفسه نزّله .. لأنه مكانش موجود على النظام.

🎯 الجزء التاسع – الفرق بين الطريقتين

الحالة	المصدر اللي Jenkins استخدمه
أول build	Node.js اللي على النظام (host)
تاني build	Node.js اللي اتثبت كـ plugin داخل Jenkins

🎯 الخلاصة

- لو Jenkins و Node.js على نفس السيرفر ← بيستخدم اللي في النظام.
- لو Jenkins على سيرفر منفصل أو Agent مش عليه Node.js ← استخدم Node.js Plugin
- الطريقة دي تنفع في:
  - Freestyle Projects
  - وكمان في Pipeline scripts (هتتشاف في الجزء الجاي من الكورس)

Organization Folder Project in Jenkins

🎯 الفكرة العامة:

- ❑ الجزء ده بيشرح نوع جديد من الـ Jenkins Projects اسمه Organization Folder Project 📂
- ❑ وده بيساعدك تربط Jenkins مع GitHub Organization بالكامل بحيث Jenkins :
  - يكتشف أوتوماتيك أي repository جديدة.
  - يعمل build لكل branch أو PR (Pull Request) جديد.
  - ينشئ أو يحدث أو يحذف الـ pipelines بشكل تلقائي من غير ما تعملها يدويًا.

🧩 الجزء الأول – مقدمة وأنواع المشاريع اللي استخدمناها قبل كده

sidd-harth/solar-system-gitea-jenkins-advanced

- قبل كده كنا شغالين على:
  - 🧱 Freestyle Project
  - 🔁 Pipeline Project
- النهاردة هنبدأ نوع جديد:
  - 📂 Organization Folder Project
  - بس قبل مانبدأ نشتغل عليه، هنجهز Repository جديدة اسمها solar-system 👉 وننقلها من GitHub إلى الـ Git Server الداخلي بتاعنا.

⚙️ الجزء الثاني – نقل (Migration) Repository من GitHub

1. فتح الـ Git Server وسجّل الدخول:
2. username: gitadmin
3. راح على Dasher Organization اللي كان شغال فيها قبل كده.
4. عمل New Migration ← اختار المصدر GitHub
5. لصق لينك الريبو القديمة ← وسمّى الريبو الجديدة: solar-system
6. والـ owner = dasher-org
7. ضغط "Migrate" ✅ .. وبعد ثواني الريبو ظهرت بنجاح داخل الـ Dasher Organization.

❑ 👀 دلوقتي عنده 3 repositories :

- Jenkins Hello World
- Parameterized Pipeline Job
- Solar System (اللي لسه متضافة)

🧱 الجزء الثالث – فكرة المشكلة

- بيقولك : ممكن أروح أعمل pipeline job جديدة يدويًا لكل Repository
  - لكن ده مش عملي ❌
- ليه؟
  - لأن كل org ممكن فيها:
    - Repositories كتير.
    - كل repo فيها branches و PRs كتير.

➢ يعني كل مرة هتضطر تعمل job جديدة يدويًا 😩

- الحل؟
  - ✨ Multi-Branch Pipeline أو Organization Folder Project

🧠 الجزء الرابع – الفرق بين الأنواع

- Manual Pipeline : بتنشئ كل pipeline يدويًا.
- Multi-Branch Pipeline : Jenkins بيعمل job لكل branch جوه repository واحدة
- Organization Folder Project :
  - Jenkins بيراقب كل الـ repositories جوه organization كاملة ويعمل jobs أوتوماتيك لأي branch / repo فيها Jenkinsfile

❑ 📘 حسب documentation بتاعت Jenkins :

➢ الطريقة المفضلة هي استخدام Organization Folder لأنها:

- أكثر أوتوميشن.
- بتتعامل مع Organizations زي GitHub و GitLab و Bitbucket بسهولة.

🧩 الجزء الخامس – إنشاء Organization Folder Project

1. رجع لـ Jenkins Dashboard
2. عمل : New Item → Name: GitHub Organization
3. اختار نوع المشروع: Organization Folder
4. ضغط OK ✅

❑ دلوقتي أول حاجة بيطلبها Jenkins هي : "Select Repository Source"

➢ لكن هنا لاحظ مشكلة 👇

- مافيش خيار اسمه "GitHub" في القائمة 😅 .. فيه بس Bitbucket و GitHub Organization قديمة .. يعني لازم نركّب GitHub Plugin

---

🔧 الجزء السادس – تثبيت Gitea Plugin

1. راح على : Manage Jenkins → Plugins → Available
2. بحث عن: Gitea Plugin
3. اختار أحدث إصدار ← Install
4. فعّل: Restart Jenkins when installation is complete

➢ بعد إعادة التشغيل ✅

سجّل الدخول تاني ← وتأكد إن Gitea Plugin بقى Installed

⚙️ الجزء السابع – إعداد Gitea Server في Jenkins

1. راح على : Manage Jenkins → System
2. بحث عن"Gitea"
   - هيلاقي Section اسمه : Gitea Servers
3. ضغط "Add Gitea Server"
   - الاسم : Gitea-Server
   - URL : عنوان السيرفر (مثلاً http://your-git-server)
4. Jenkins تعرّف على السيرفر وظهر إصدار Gitea ✅ (يعني الاتصال شغال تمام)

🧩 الجزء الثامن – تفعيل Webhooks

- فعّل خيار : Manage hooks automatically
- وده معناه:
  - Jenkins هيعمل Webhook لكل Repository في GitHub عشان أي Push أو PR يحصل ← Jenkins يشتغل أوتوماتيك.
  - لكن لازم نضيف Credentials

🔑 الجزء التاسع – إضافة Credentials

1. Add → Jenkins Credentials
2. النوع : Username and Password
3. البيانات :
   - username: gitea-admin
   - password : (كلمة المرور بتاع الـ Gitea Server بتاعك)
   - ID: gitea-server-creds
4. Save ✅

كده Jenkins بقى عنده Credential يقدر يدخل بيها على GitHub

🧱 الجزء العاشر – ربط Jenkins بالـ Organization Folder

1. رجع للمشروع اللي عمله : GitHub Organization Project
2. دخل على Configure
3. في "Repository Source" اختار: Gitea Organization
4. الإعدادات:
   - Server → Gitea Server
   - Credentials → Gitea-admin-credentials
   - Owner → dasher-org

باقي الإعدادات سبها Default :

- Scan branches ✅
- Discover pull requests ✅

🧩 الجزء الحادي عشر – طريقة اكتشاف المشاريع (Project Recognizers)

- Jenkins هيشوف كل repository جوه الـ org ويعمل Scan :
  - لو لقى Jenkinsfile في root folder ← هيعملها Job
  - لو مفيش Jenkinsfile ← هايتجاهلها (يتجاهل المشروع يعني)
- ممكن تحدد مسار مخصص لو Jenkinsfile مش في root
- كمان ممكن تحدد:
  - كل قد إيه يعمل scan (مثلاً كل يوم)
  - شكل الـ appearance أو libraries

حفظ التغييرات ← Save ✅

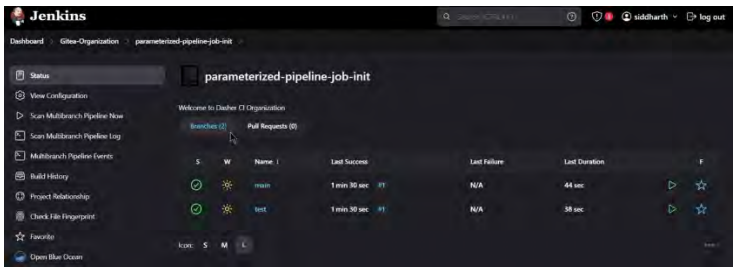
⚙️ الجزء الثاني عشر – النتيجة بعد Scan

- Jenkins بدأ يعمل Scan للـ Organization :
- لقى repo اسمها Jenkins Hello World ← بس مفيش Jenkinsfile ← فتجاهل المشروع ده ❌
- لقى Parameterized Pipeline Job ← فيها Jenkinsfile ✅ ← فأنشأ Pipeline ليها
- لقى Solar System ← مفيش Jenkinsfile ← فتجاهل المشروع ده ❌

❑ يعني : Jenkins بيبني Pipelines بس للـ repos اللي فيها Jenkinsfile

---

🧱 الجزء الثالث عشر – التحقق من النتائج



في صفحة Jenkins :

- هتلاقي مشروعين اتعملوا (لكل branch في الـ repo)
- كل Jenkinsfile اشتغل على حسب فروعه (main و test)

🌐 الجزء الرابع عشر – تأكيد الـ Webhook

- دخل على GitHub ← فتح إعدادات Parameterized Pipeline Job ← لقى Webhook جديد اتضاف أوتوماتيك 👇
  - http://<jenkins-ip>:8080/gitlab-webhook/post

- Webhook 🔁 ده بيتفعل مع:
  - أي push
  - أي branch creation / deletion
  - أي pull request

يعني أي تغيير في Gitea ← Jenkins يشتغل مباشرة 🚀

🧠 الجزء الخامس عشر – تجربة عملية على الـ Webhook

1. فتح repo → main branch
2. فتح ملف README.md
3. كتب كلمة : edited
4. وعمل commit + push
5. راح يشوف الـ Webhook ← لونه بقى أخضر ✅ (اشتغل)
6. راح على Jenkins ← لقى job جديدة اتعملت تلقائيًا.

🎯 كده تأكد إن الـ Webhook فعلاً شغال وبيـ Trigger Jenkins

✅ الخلاصة

العنصر	الشرح
💡 الهدف	أوتوماتيك Jenkins Pipelines لكل repos داخل Organization
🔧 الأدوات	GitHub Plugin + Credentials + Webhooks
⚙️ النتيجة	Jenkins يعمل Scan ويبني Pipelines لأي Repository فيها Jenkinsfile
🚀 الميزة	Zero manual setup — كل حاجة أوتوماتيك بمجرد وجود Jenkinsfile

Add Jenkinsfile to Solar System Repo

المقدمة

- الجزء ده بيشرح إزاي نضيف Jenkinsfile في Repository على GitHub (اسمها Solar System) عشان نقدر نفعّل الـ CI/CD عليها باستخدام Jenkins Organization Folder
- بمعنى بسيط جدًا:
  - عندنا Jenkins متوصل بـ GitHub Organization
  - فيه Repo (مثلاً PlanetApp) كانت فيها Jenkinsfile ، فاشتغلت تلقائي.
  - لكن Repo تانية (Solar System) لسه ما فيهاش Jenkinsfile فمش ظاهرة في Jenkins ولا عندها Webhook
- الخطوات هنا بتشرح:
  1. إزاي نضيف Jenkinsfile للـ repo دي.
  2. إزاي Jenkins يتعرف عليها تلقائي.
  3. إزاي الـ webhook يتكوّن أوتوماتيك.
  4. إزاي أول Pipeline يشتغل.
  5. وفي الآخر، نربطه بأداة Node.js اللي معمولة داخل Jenkins Tools

---

الشرح التفصيلي

أول فكرة:

مبدئيا كدا احنا مش هانبقي شايفين Solar System في Jenkins ، والسبب إنها مافيهاش Jenkinsfile

📌 النقطة دي مهمة جدًا:

Jenkins بيعمل scan لكل Repos في Organization ، بس بيتجاهل أي Repo مافيهاش Jenkinsfile

---

الخطوة التانية:

- هانعمل clone للـ repo دي : cd Solar-System THEN git clone <repo_link>
- ده بينزل المشروع محليًا عشان نقدر نضيف فيه Jenkinsfile ونرفعه تاني.

بعدها:

يا ريت دي تكون
الـ mindset بتاعتك دايما

- هيعمل branch جديدة مخصوص للـ CI/CD configuration
  - git checkout -b feature/enabling-CI-CD
- 💡 ليه؟
  - عشان نطبق الـ pipeline على فرع تجريبي الأول، وبعد كده نعمل Pull Request لما نتأكد إن كل حاجة شغالة.
- 📘 Best Practice :
  - دايمًا اعمل تغييرات الـ pipeline أو أي automation في Feature Branch ، مش على الـ main مباشرة.

بعدها أنشأ ملف جديد:

- touch Jenkinsfile
  - وأضاف فيه Pipeline بسيط جدًا فيه Stage واحدة بس اسمها "VM node version"

- وده شكلها:

```
pipeline {
    agent any
    stages {
        stage('VM node version') {
            steps {
                sh 'node -v'
                sh 'npm -v'
            }
        }
    }
}
```

📘 الشرح:

- **Pipeline** : بداية تعريف الـ Jenkins declarative pipeline
- **agent any** : معناها pipeline دي ممكن تشتغل على أي Agent متاح.
- **Stage** : بيمثل خطوة (أو مرحلة) من مراحل التنفيذ.
- **Steps** : الأوامر اللي هتتنفذ جوه المرحلة.
- **Sh** : معناها شغل أمر "shell" ، وهنا بيطبع نسخ Node.js و npm

---

🔹 **بعد ما خلص:**

- عمل commit و push
  - git add Jenkinsfile
  - git commit -m "Configured Jenkinsfile"
  - git push origin feature/enabling-CI-CD

📘 النتيجة:

- لما رفع الـ branch الجديد، Jenkins Organization Folder عمل scan ولقا Jenkinsfile
- فأنشأ pipeline جديدة تلقائيًا.
- وكمان عمل Webhook في Gitea (عشان أي push أو PR يفعّل build تلقائيًا)

🔹 **لما راح على Gitea :**

شاف إن فعلاً اتعمل Webhook جديد أوتوماتيك:

- واحد عام للـ repo
- والتاني خاص بالـ branch

➢ 📘 Concept مهم :

- Jenkins Organization Folder plugin بيعمل auto-management
- يعني أي repo أو فرع جديد فيه Jenkinsfile ← Jenkins هيكتشفه ويشغله.

🔹 **لما pipeline اشتغلت:**

➢ Jenkins نفذ الخطوات:

1. Checkout للـ source code
2. Run للـ stage اللي بتطبع نسخ Node.js و npm

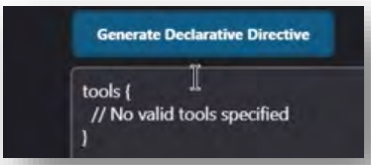
➢ لكن هنا نقطة مهمة:

- هو استخدم نسخة Node.js اللي موجودة أصلاً على الـ machine اللي بيشتغل عليها Jenkins (مش النسخة اللي معرفها في Tools)

🛠️ **ربط Node.js Tool اللي داخل Jenkins**

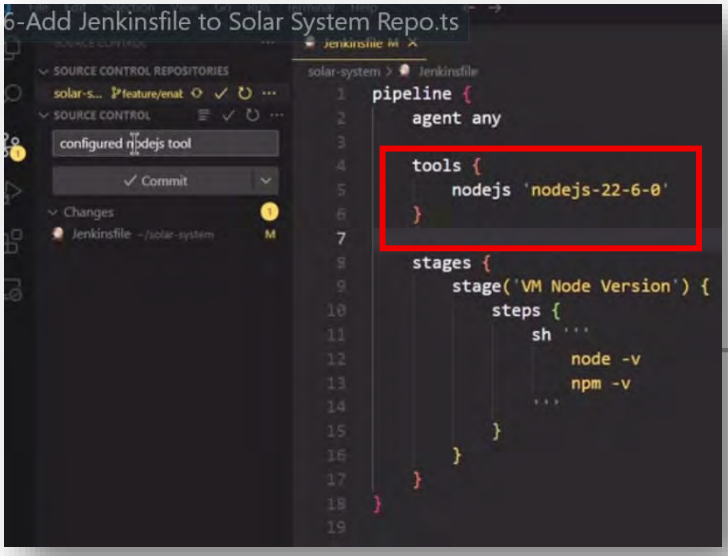
➢ الشرح هنا بيوضح إزاي نخلّي الـ pipeline تستخدم Node.js اللي مكوّنة داخل Jenkins Tools نفسها.

1. يفتح Jenkins → Pipeline Syntax Generator
2. يختار من هناك:
   - Declarative Directive Generator
   - وفي الـ Sample Directive يختار tools:Tools
   - وبعدين اضغط Generate ← هايطلعلك الـ syntax




```
tools {
nodejs "NodeJS"
}
```

3. يرجع يضيف الجزء ده داخل الـ pipeline :
   - لاااحظ ان الاسم "nodejs-22-6-0" هو اللي كنا مسمينه في Jenkins configuration



📘 شرح السطر ده:

- tools ← بتحدد Tool preconfigured في Jenkins
- nodejs "NodeJS" ← اسم Tool اللي جهزناه في Manage Jenkins > Global Tool Configuration

---

🔹 **بعد التعديل:**

➢ عمل commit تاني:

- git commit -m "Configured Node.js tool"
- git push origin feature/enabling-CI-CD

➢ الـ webhook اشتغل تاني وبدأ Build جديد تلقائيًا.

🔹 **النتيجة في Logs :**

➢ لما راح للـ Console Output :

شاف إن Node.js version المستخدم هو 22.6.0 .. وده بالضبط الإصدار اللي جهزه في Jenkins tools ✅

💡 **مفاهيم إضافية (Concepts & Best Practices)**

- Organization Folder :
  Jenkins بيعمل scan دوري للـ GitHub Org كلها، وأي Repo فيها Jenkinsfile بتضاف تلقائيًا كمشروع.
- Webhook Auto-Creation :
  Jenkins بيبعت Webhook للـ repo عشان كل push أو PR يفعّل build تلقائي.
- Tools Section :
  بدل ما تعتمد على tools اللي موجودة على الـ Agent (الماكينة نفسها)، تقدر تستخدم Tools معرفّة في Jenkins لسهولة الإدارة وتوحيد الإصدارات.
- Branching Strategy :
  الأفضل دايمًا تبدأ بفرع "feature/..." لتهيئة الـ pipeline وتجربتها، وبعد كده تدمجها في main
- Real Use Case :
  بيستخدم في المشاريع اللي فيها أكتر من microservice أو repo ، كل واحدة فيها Jenkinsfile خاص بيها، وكلها مربوطة بنفس Jenkins Organization Folder

---

الملخص

- عملنا Repository اسمها Solar System .. وكانت مش ظاهرة في Jenkins لأنها مافيهاش Jenkinsfile
- عملنا branch جديدة باسم feature/enabling-CI-CD .. ضفنا فيها Jenkinsfile بسيط فيه أمرين (node -v, npm -v)
- بعد الـ push ← Jenkins اكتشفها أوتوماتيك وعمل Webhook
- عدلنا Jenkinsfile تاني عشان يستخدم Node.js tool اللي جوه Jenkins
- بعد التعديل ← الـ build اشتغل بالنسخة المظبوطة.
- النتيجة:

Jenkins بقى بيعمل auto-scan، auto-create pipelines، auto-trigger builds لأي Repo جديدة فيها Jenkinsfile ✅

Install dependencies

🧩 مقدمة :

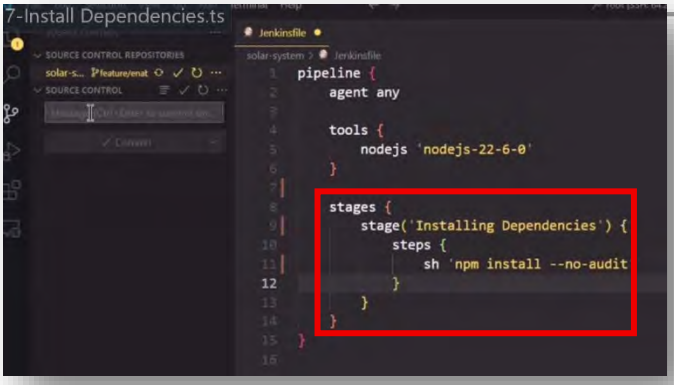
- ❑ الجزء ده بيتكلم عن إضافة ستيج جديدة في الـ Jenkins pipeline اسمها *Installing Dependencies*، والهدف منها هو تثبيت الـ Node.js dependencies اللي المشروع محتاجها قبل مراحل التنفيذ أو الاختبار.
- ❑ الشرح هنا بيكمل على المشروع اللي اتكلمنا عنه قبل كده لما كنا بنستخدم Node.js في بناء GitHub Actions workflow ، لكن المرة دي داخل Jenkins Pipeline

---

🧱 شرح خطوة بخطوة:

1. 🧩 إنشاء Stage جديدة:

الخطوة الأولى هي إننا بنضيف ستيج جديدة في الـ Jenkinsfile



- 🔹 الفكرة هنا إنه بدل ما نستخدم الستيج القديمة اللي كانت بتحدد نسخة Node.js (VM Node version stage) ، استبدلناها بستيج جديدة مخصوص لتثبيت المكتبات.
- 🔹 استخدم **npm install** علشان يحمل كل الـ packages اللي متعرفة في ملف package.json
- 🔹 استخدمنا **--no-audit** عشان يمنع عملية فحص الثغرات الأمنية (auditing) مؤقتًا.
  - o ده لأننا هنخلي الفحص الأمني (audit) في مرحلة مستقلة لاحقًا.
- 🔹 استخدم **--save** عشان يضيف أي مكتبات جديدة تتثبت تلقائيًا لقائمة الـ dependencies في ملف package.json (لو احتجنا)

---

2. 💾 حفظ التغييرات:

➢ بعد ما عدّلنا في ملف الـ Jenkinsfile ، بنعمل:

- o **git add .**
- o **git commit -m "Added stage for installing dependencies"**
- o **git push**

  - ▪ وده بيبوش التغييرات إلى الـ repository اللي Jenkins بيراقبها.

---

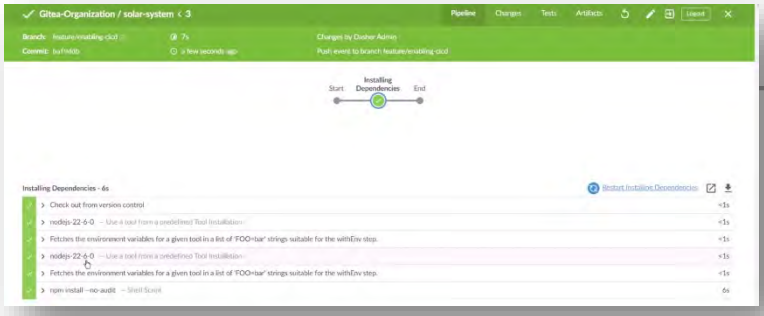
3. ⚙️ تشغيل الـ Pipeline تلقائيًا:

➢ أول ما تعمل push ، Jenkins بيفهم إن فيه تحديث في الكود، فيبدأ pipeline جديدة أوتوماتيك.

➢ لما نفتح Blue Ocean (الواجهة الرسومية للـ Jenkins pipelines) هنلاحظ إن:

- الـ pipeline اشتغلت وبتنفذ الـ stage الجديدة.
- الـ "Installing Dependencies" job اشتغل بنجاح.
- في الـ logs ، بيظهر إنه استخدم Node.js tool وشغّل الأمر : **npm install --no-audit**

⏱️ العملية أخدت تقريبًا 5 ثواني وأضافت 3 مكتبات .



4. 📁 التحقق من النتيجة:

➢ بعد ما الـ pipeline خلصت، نقدر نفتح من واجهة Blue Ocean :

- Workspaces → node_modules

وهتلاقي فعلاً فولدر node_modules اتعمل، وده دليل إن عملية التثبيت تمت بنجاح وكل الـ dependencies اتحمّلت جوه بيئة التنفيذ.

🧠 معلومات إضافية و Best Practices :

- دايمًا خلي مرحلة تثبيت الـ dependencies منفصلة عن الاختبارات أو الـ build ، عشان لو فشل التثبيت تعرف بسرعة إن المشكلة في الـ packages مش في الكود نفسه. (سؤال انترفيو : ازاي هاتعرف اذا كانت المشكلة في الـ packages ولا الكود نفسه ؟!؟!)
- استخدم الـ flag --no-fund كمان في حالة لو انت مش عاوز Jenkins يظهر رسائل تمويل للمكتبات المفتوحة المصدر.
- لو عايز تثبت مكتبات معينة بس بدون تحديث باقي الـ dependencies ، استخدم : npm ci
  وده مفيد جدًا في الـ CI/CD لأنه أسرع وثابت (_بيعتمد على lock file)
- تقدر تعمل caching لفولدر node_modules بين الـ builds لتقليل وقت التثبيت.

🎯 الخلاصة:

- ❑ الجزء ده كان هدفه إننا نضيف مرحلة "Installing Dependencies" في Jenkins pipeline
- ❑ المرحلة دي:

1. تستخدم npm install --no-audit لتثبيت مكتبات المشروع.
2. تجهّز بيئة التشغيل قبل أي اختبارات أو عمليات build
3. بتنشئ فولدر node_modules اللي بيحتوي كل المكتبات المطلوبة.

- ❑ بكده نضمن إن كل build بيبدأ في بيئة نظيفة ومجهزة بكل dependencies اللي محتاجها المشروع. 💪

Setup and Run Dependency Scanning

🧩 أولاً : فكرة الـ Dependency Scanning

- ❑ الهدف منها إنك تفحص الـ dependencies (المكتبات اللي بتستخدمها) في المشروع، وتشوف لو فيها أي ثغرات (vulnerabilities)معروفة علشان تصلحها قبل ما توصل للـ production
- ❑ وبنعمل كدا باستخدام طريقتين:

1️⃣ npm audit

2️⃣ OWASP Dependency Check

✅ الجزء الأول – باستخدام npm audit

📘 الفكرة:

- ➢ npm audit بيفحص الـ package.json و package-lock.json وبيبعت نسخة من dependencies لقاعدة بيانات الـ npm عشان يعرف لو في مكتبات فيها ثغرات أمنية.
- ➢ 📜 خطوات Jenkins stage :

```
stage('NPM Dependency Audit') {
    steps {
        sh '''
            npm audit --audit-level=critical
            echo $?
        '''
    }
}
```

- --audit-level=critical معناها إنه هيوقف الـ build فقط لو فيه ثغرات حرجة (critical).
- لو لقى critical issues ← هيعمل exit code 1 ← بالتالي الـ stage هيفشل.
- ➢ بعد ماتضيف الـ stage في الـ jenkinsfile .. متنساش تعمل commit & push

---

✅ الجزء الثاني – باستخدام OWASP Dependency Check

📘 الفكرة:

- ❖ أداة من OWASP بتفحص الـ dependencies لأي لغة (مش بس Node.js) وتعمل تقارير HTML و XML و JSON فيها تفاصيل الثغرات.

📦 الخطوات في Jenkins :

1. تثبيت الـ Plugin : من Jenkins UI ↓

Manage Jenkins → Plugins → Available → "OWASP Dependency-Check" → Install + Restart

2. إعداد الأداة : Manage Jenkins → Tools → Dependency-Check installation

- Name: owasp-dbcheck-10
- Version: 10.0.3 (من GitHub)

3. إنشاء Stage جديد في Pipeline :

```
stage('OWASP-dependency-check') {
    steps {
        dependencyCheck additionalArguments: '''
            --scan .
            --format ALL
            --prettyPrint
        ''',
        installationName: 'owasp-dbcheck-10'
    }
}
```

- --scan . ← بيفحص المشروع الحالي.
- --format ALL ← بيولد التقارير بصيغ JSON وHTML وXML.
- --prettyPrint ← بيخلي النتائج مقروءة أكتر في الكونسول.

❖ ⏱️ أول مرة تشغله:

هينزل قاعدة بيانات ضخمة (حوالي 260 ألف سجل)، فهياخد وقت (20–30 دقيقة).

❖ 📁 التقارير بتتخزن في workspace root (بتلاقي مثلًا: dependency-check-report.html)

```
stage('OWASP Dependency Check') {
    steps {
        dependencyCheck additionalArguments: '''
            --scan \'./\'
            --out \'./\'
            --format \'ALL\'
            --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'
    }
}
```

⚙️ Snippet Generator

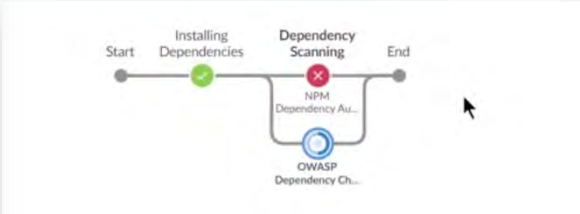
عشان تولّد الكود اللي فوق بسهولة:

- افتح أي Job ← Pipeline Syntax ← اختار dependencyCheck ← حدد الإعدادات ← Generate
  Pipeline Script وبيطلعلك الكود الجاهز تحطه في Jenkinsfile

⚙️ ملاحظة مهمة

- في البداية، Stage npm audit فشلت ← بالتالي Jenkins ما شغلش OWASP stage.
- علشان كده عمل تعديل بحيث يشغلهم في Parallel عشان مايعتمدوش على بعض :

```
stages {
    stage('Installing Dependencies') {...
    }

    stage('Dependency Scanning') {
        parallel {
    stage('NPM Dependency Audit') {
        steps {
            sh '''
                npm audit --audit-level=critical
                echo $?
            '''
        }
    }

    stage('OWASP Dependency Check') {
        steps {
            dependencyCheck additionalArguments: '''
                --scan \'./\'
                --out \'./\'
                --format \'ALL\'
                --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'
        }
    }
    }
```



- كده الاتنين بيشتغلوا في نفس الوقت ..
  و حتى لو واحد فشل، التاني هيكمل.

📊 النتائج:

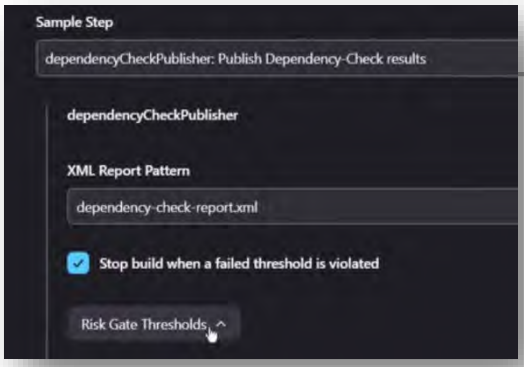
❖ التقارير بتتولد في الـ workspace داخل Jenkins:

- dependency-check-report.html
- dependency-check-report.json
- dependency-check-report.xml

❖ ممكن تفتح الـ HTML report وتشوف :

- عدد الثغرات
- نوعها (Low / Medium / High / Critical)
- في أي dependency موجودة

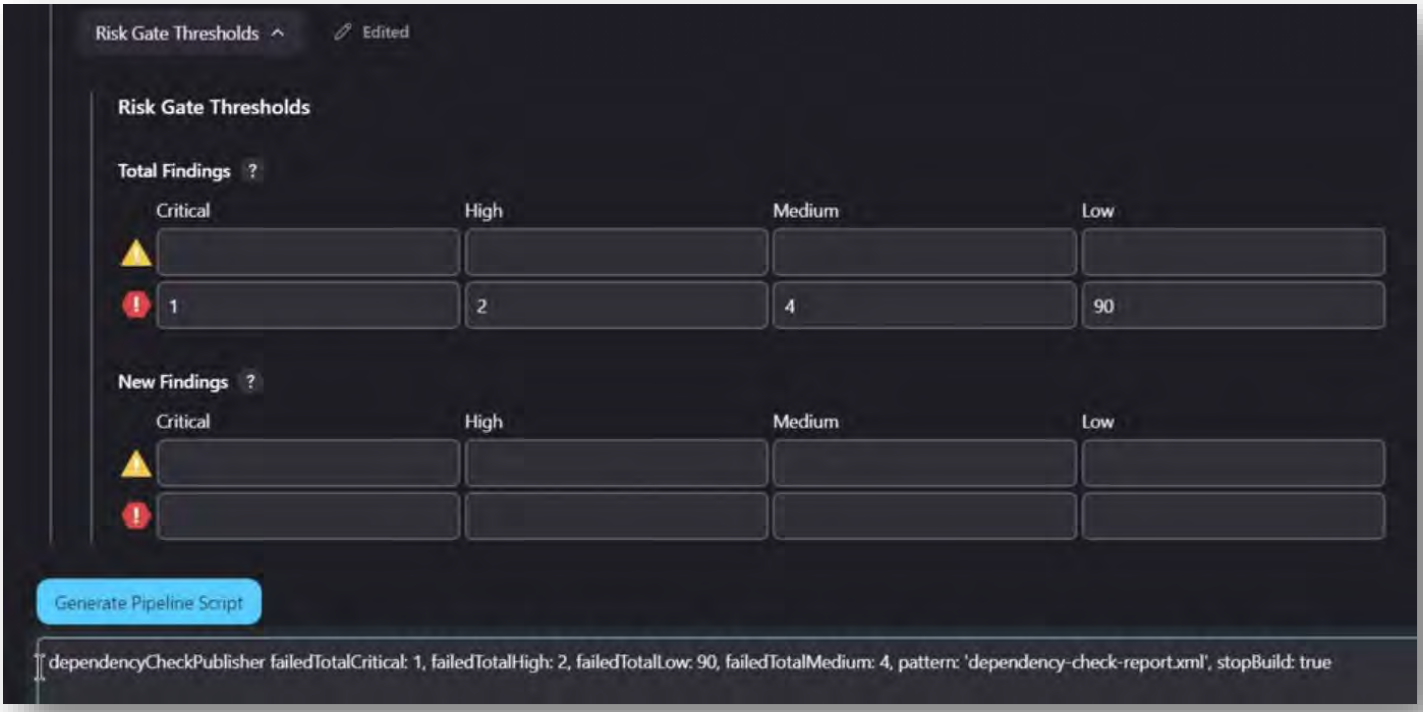
🚨 لو عايز تخلي Jenkins يفشل الـ Build لو لقى Vulnerabilities:



❖ استخدم Dependency-Check Publisher

- stopBuild: true ← يوقف الـ build لو في خطأ.
- failedTotalCritical: 1 ← لو لقى واحد على الأقل هايديني warning
  critical vulnerability: 1 ← لو لقى واحد على الأقل اذا هايفشل الـbuild

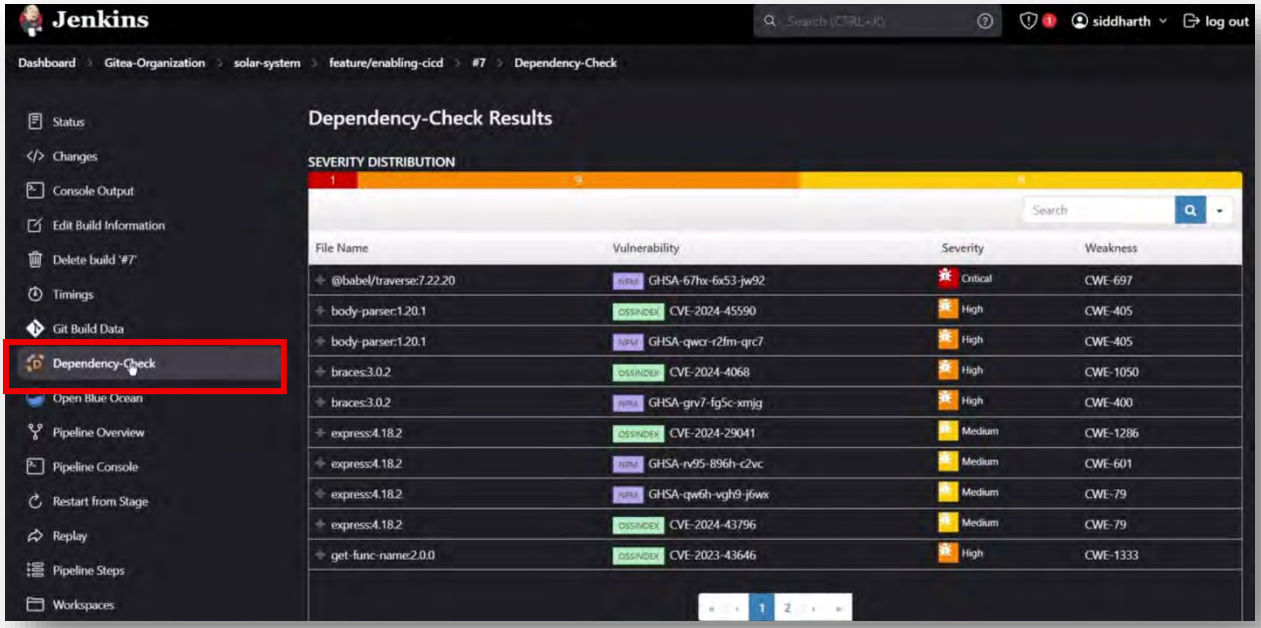
🔍 لو حبيت تعرف اكتر عن الـ arguments المتاحة ← Steps Reference



Then add this code to jenkinsfile, in stage('OWASP De...')

```
    stage('OWASP Dependency Check') {
        steps {
            dependencyCheck additionalArguments: '''
                --scan \'./\'
                --out \'./\'
                --format \'ALL\'
                --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'

            dependencyCheckPublisher failedTotalCritical: 1, pattern: 'dependency-check-report.xml', stopBuild: true
        }
    }
}
```

❖ كمان هتلاقي خيار Dependency Check اللي بيوفرلك Visual Interface علشان تشوف كل الـ Vulnerabilities :



Fixing Vulnerabilities & Publish HTML Report

🎯 المقدمة: الفكرة العامة

- ❑ الجزء ده بيكمل مراحل بناء الـ DevOps Pipeline اللي شغالين عليها، وبيغطي نقطتين أساسيتين:

1. إصلاح الثغرات (Fixing Vulnerabilities) اللي تم اكتشافها في مرحلة الـ Dependency Audit و OWASP Dependency Check
2. نشر تقارير النتائج (Publishing Reports) سواء كانت تقارير HTML (زي تقارير الثغرات) أو تقارير اختبارات (JUnit XML reports)

🧩 الشرح التفصيلي – خطوة بخطوة

1 اكتشاف الثغرات وبدء الإصلاح

- ❑ هانبدأ بأن مرحلتين (npm audit + OWASP dependency check) فشلوا لأنهم اكتشفوا critical vulnerability
- ➢ ⚙️ بمعنى : عند تشغيل الأوامر الخاصة بفحص الثغرات في مرحلة الـ CI ، النظام وجد حزمة (package) فيها ثغرة خطيرة.
- ➢ 🔹 هنا الخطوات بتكون كالآتي:
  - المطور (developer) بيعمل npm audit علشان يشوف تفاصيل الثغرات.
  - بعدها بيبدأ يختار إما:
    - o npm audit fix لإصلاح كل الثغرات تلقائيًا (لو ينفع).
    - o أو يصلح يدويًا كل dependency محتاجة تحديث لنسخة محددة.
  - 🔸 مثال عملي : npm audit --audit-level=critical
    - o ده بيعرض كل الثغرات ذات الخطورة critical فقط.
  - ولو عايز تصلحها : npm install <package>@<new_version>
- ➢ في المثال: npm install package@7.23.2
  - بعد التثبيت، الـ npm بيعمل audit تلقائي وبيأكد إن الثغرة الخطيرة اختفت.
  - تقدر تتأكد من كده عن طريق: echo $?
    - o لو رجع 0 ← مفيش ثغرات حرجة،
    - o لو رجع 1 ← لسه فيه مشاكل.

---

2 تعديل ملفات الـ package

- ➢ بعد الإصلاح، بيتغير:
  - ملف package.json (فيه تعريف الـ dependencies)
  - وملف package-lock.json (فيه نسخ الـ dependencies الفعلية المثبتة)
- ➢ فلازم المطور يعمل commit بالنسخ الجديدة:

```
o git add .
o git commit -m "fix critical vulnerability"
o git push
```

- ➢ ده بيشغل pipeline جديد في Jenkins بشكل تلقائي.

---

3 التأكد من نجاح الإصلاح

الـ Jenkins بيعيد تشغيل مراحل الفحص:

- الـ npm dependency audit بيعدي بنجاح. ✅
- والـ OWASP Dependency Check كمان. ✅

وده لأن خلاص مفيش critical vulnerabilities

❑ Best Practice 🧠 : في أي مشروع حقيقي، دايمًا اعمل auditing للـ dependencies في بيئة تطوير منفصلة قبل دمج الكود للـ main branch

---

4 نشر تقارير النتائج (HTML Reports)

❑ بعد ما الثغرات اتصلحت، بيبدأ الجزء التاني من شرحنا:

➢ نشر التقارير داخل Jenkins علشان المطورين يقدروا يشوفوها من الواجهة مباشرة بدل ما يدوروا في الـ workspace

➢ بيستخدم Jenkins step اسمه : publishHTML

➢ ◆ الخطوات:

- يفتح Pipeline Syntax Generator في Jenkins
- يختار "Publish HTML Reports"
- يحدد:
  - الـ directory اللي فيه التقرير ← ./ (root directory)
  - الـ index page ← اسم ملف التقرير (مثلاً dependency-check-jenkins.html)
  - report title ← "Dependency-Check HTML report"
- يولّد الكود ويضيفه للـ Jenkinsfile بعد مرحلة الفحص.

➢ الكود الناتج بيكون بالشكل ده:

```
publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './', reportFiles: 'dependency-check-jenkins.html', reportName: 'Dependency Check HTML Report', reportTitles: '', useWrapperFileDirectly: true])
```

وبعدين انسخه في الـ jenkinsfile .. وتحديدا في stage('OWASP DE...')

```
stage('OWASP Dependency Check') {
    steps {
        dependencyCheck additionalArguments: '''
            --scan \'./\'
            --out \'./\'
            --format \'ALL\'
            --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'

        dependencyCheckPublisher failedTotalCritical: 1, pattern: 'dependency-check-report.xml',

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: '
    }
```

➢ وبعد الـ commit :

- Jenkins بيشغّل pipeline جديد.
- بتظهر مرحلة جديدة اسمها Publish HTML Report ✅.
- جوه تبويب Artifacts هتلاقي التقرير منشور.

```
publishHTML([
  reportDir: '.',
  reportFiles: 'dependency-check-report.html',
  reportName: 'Dependency-Check HTML report'
])
```

➢ لكن التقرير ممكن يطلع من غير CSS 😅

لأن Jenkins بيمنع CSS وجافا سكريبت في التقارير بشكل افتراضي بسبب سياسة الأمان (Content Security Policy - CSP)

5 تفعيل الـ CSS في التقارير (تعديل الـ CSP)

علشان التقرير يظهر بشكل طبيعي:

- تفتح Manage Jenkins → Nodes → Built-in Nodes → Script Console
- تشغل الكود اللي بيغير إعداد الـ CSP :
  - System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "")

ده بيخفف القيود ويسمح بعرض CSS

⚠️ لكن لازم تعيد تشغيل الـ build علشان التغييرات تتطبق في تقرير جديد.

Junit : Provide graphical visualization of the historic test results.
Provide a publisher that consumes XML test reports which are generated during the builds

---

6 نشر تقارير الاختبارات (JUnit XML Reports)

➢ الهدف : عرض نتائج الـ XML test report من JUnit على واجهة Jenkins (تبويب "Test Results").

الخطوات:

1. تأكد إن الـ Plugin بتاع JUnit مثبت (junit Archive Junit-formatted test result)

```
junit allowEmptyResults: true, stdioRetention: '', testResults: 'dependency-check-junit.xml'
```

- allowEmptyResults: true ← عشان لو التقرير فاضي، الـ build مايفشلش.
- testResults ← هو اسم ملف النتائج.

2. أضف في الـ Jenkinsfile :

```
stage('OWASP Dependency Check') {
    steps {
        dependencyCheck additionalArguments: '''
            --scan \'./\'
            --out \'./\'
            --format \'ALL\'
            --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'

        dependencyCheckPublisher failedTotalCritical: 1, pattern: 'dependency-check-report.xml',
        stopBuild: true

        junit allowEmptyResults: true, stdioRetention: '', testResults: 'dependency-check-junit.
        xml'

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir:
        './', reportFiles: 'dependency-check-jenkins.html', reportName: 'Dependency Check HTML
        Report', reportTitles: '', useWrapperFileDirectly: true])
```

➢ نعمل commit جديد : git commit -m "configure 'JUnit reports' THEN git push

- Jenkins بيشغل pipeline جديد.
- بتظهر مرحلة اسمها Archive JUnit Test Results
- بتظهر تبويبة جديدة في Jenkins اسمها Tests فيها عدد الاختبارات اللي نجحت أو فشلت.

---

7 عرض التقارير في الواجهات المختلفة

- في Blue Ocean UI ← بتشوف نتائج الاختبارات بشكل مرئي لطيف.
- في Classic UI ← بتلاقي تبويب جديد على الشمال اسمه Test Results فيه نفس التفاصيل (عدد الاختبارات، أسباب الفشل، درجات الخطورة، تفاصيل الـ CVE ...)

🧠 معلومات إضافية و Best Practices

✅ نصائح عملية:

- شغل npm audit بشكل دوري حتى لو مفيش تغييرات في الكود.
- استخدم أدوات زي Snyk أو Dependabot عشان ترصد التحديثات التلقائية للـ dependencies
- حافظ على تقاريرك محفوظة في Jenkins و S3 للمراجعة المستقبلية.
- متسيبش ثغرات حرجة حتى لو "مش مؤثرة حالياً" — لأنها ممكن تُستغل لاحقًا.

📄 الملخص العام

كدا نبقي شرحنا دورتين متكاملتين داخل الـ CI/CD :

1. إصلاح الثغرات الأمنية (Fix Vulnerabilities) :
   - عمل npm audit
   - إصلاح النسخ المتأثرة.
   - commit + push ← Jenkins يعيد التشغيل.
   - تأكيد نجاح المراحل الأمنية.
2. نشر التقارير (Reports Publishing) :
   - HTML Reports (dependency check)
   - JUnit Reports (اختبارات الوحدة)
   - تفعيل CSS لعرض التقارير بشكل كامل.
   - عرض النتائج في Blue Ocean و Classic UI

📌 النتيجة النهائية:

بايبلاين كامل آلي بيعمل:

- فحص الثغرات 🔍 - إصلاحها 🔧
- اختبار الكود ✅ - توليد ونشر التقارير 📊 - عرض النتائج في Jenkins بكل وضوح 🎯

unit Testing and Analyze JUnit Reports

🎯 الفكرة العامة

- ❑ الجزء ده بيدخلنا على مرحلة Unit Testing Stage داخل الـ CI/CD Pipeline في Jenkins، وبيشرح خطوة بخطوة إزاي تضيف مرحلة اختبار الوحدات (Unit Tests)، وتتعامل مع Environment Variables و Jenkins Credentials عشان تقدر توصل لقاعدة البيانات أثناء الاختبارات.
- ❑ في المرحلة دي المطور بيضيف Stage جديدة داخل الـ Jenkinsfile الهدف منها هو تشغيل اختبارات الوحدة (Unit Tests) باستخدام أمر : npm test
- ❑ لكن علشان الاختبارات تشتغل، التطبيق محتاج يتصل بقاعدة بيانات MongoDB ، وده محتاج متغيرات بيئية (Environment Variables) زي:
  - mongoURI
  - mongo_username
  - mongo_password
- ➢ فبالتالي المطور لازم يمرر القيم دي جوه Jenkins بطريقة آمنة عن طريق Jenkins Credentials

---

🧩 الشرح التفصيلي – خطوة بخطوة

1 إنشاء مرحلة Unit Testing

- أول خطوة هي نسخ مرحلة "install dependencies" الموجودة بالفعل في الـ Jenkinsfile وتعديلها لتكون بالشكل ده:

```
stage('Unit Testing') {
    steps {
        sh 'npm test'
    }
}
```

- 📌 دي مرحلة بسيطة بتشغّل أوامر الاختبار الخاصة بـ Node.js (يعني نفس اللي بيحصل لما تكتب npm test يدويًا في التيرمنال)

---

2 فشل المرحلة – السبب؟

- ➢ بعد تشغيل الـpipeline ، المرحلة فشلت ❌
- ➢ 💡 السبب:

في اللوجز ظهر إن التطبيق حاول يتصل بـ MongoDB .. لكن مفيش URI أو credentials متعرفة داخل Jenkins

- ➢ التطبيق (app.js) محتاج Environment Variables زي:
  - process.env.mongoURI
  - process.env.mongo_username
  - process.env.mongo_password

لكن Jenkins مكنش عارف القيم دي، فالاتصال فشل.

> خلي باااالك انها بتتعرف علي الـ Global Level في الـ Jenkinsfile .. وليس في stage معينة (بالبلدي كدا بتتكتب قبل ماتكتب اي stage)

---

3 تعريف Environment Variables داخل Jenkinsfile

- ➢ الحل هو إنك تضيف الـ environment variables دي جوه Jenkinsfile
- تقدر تستخدم Declarative Directive Generator (الـ Snippet Generator) علشان يولّدلك كود بيعرّف متغيرات البيئة، بالشكل ده مثلًا:

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
}
```

- ◆ لكن فيه مشكلة مهمة:
- لو حطيت username و password هنا، هيبقوا plain text داخل Jenkinsfile وده خطر لأن الملف بيتخزّن في GitHub !!

4️⃣ الحل الآمن 🔐 Jenkins Credentials :

علشان كده استخدمنا Jenkins Credentials Plugin .. وده بيسمحلك تحفظ الـ secrets بشكل آمن ومش ظاهر لأي حد.

الخطوات:

1. من لوحة Jenkins :
   - روح على Manage Jenkins ← Credentials
   - اختار Global credentials (unrestricted)
   - Add Credentials ← نوع Username with password
2. اكتب:
   - Username: superuser
   - Password: superpassword
   - ID: mongo-db-credentials
   - Description: "MongoDB credentials"

🧠 كده بقى عندك Credential ID تقدر تستخدمه جوه الـ pipeline

---

5 استخدام الـ Credentials في الـ Pipeline

- في Jenkinsfile ، استخدم step اسمه : withCredentials()
- من الـ pipeline sytax ← Snippet Generator ← في الـ Sample step اختار withcredentials:Bind ... وبعدين ظبط بقية الاعدادات
  - في الـ bindings ← add username and password (seperated)
  - Username variable : MONGO_USERNAME
  - Passward Variable : MONGO_PASSWORD
  - Jenkins : Credentials (الخطوات اللي عملتها في خطوة رقم 4 .. اللي هي قبل دي علطول)
- وبعدين هاديها Generate فهايطلعلي الكود اللي هاضيفه في الـ jenkinsfile وتحديدا في stage('Unit Testing') :

```
withCredentials([usernamePassword(credentialsId: '', passwordVariable: 'MONGO_PASSWORD', usernameVariable: 'MONGO_USERNAME')]) {
    // some block
}
```

```
    stage('Unit Testing') {
        steps {
            withCredentials([usernamePassword(credentialsId: '', passwordVariable: 'MONGO_PASSWORD', username
                sh 'npm test'
            }
        }
    }
}
```

💡 المعنى:

- credentialsId = اسم الـ ID اللي سجلته في Jenkins
- usernameVariable و passwordVariable = أسماء المتغيرات اللي الكود محتاجها.
- جوا sh 'npm test' بيقدر Node.js يستخدمهم عادي من خلال process.env.mongo_username وهكذا.

6️⃣ التأكد من نجاح الاتصال بالـ MongoDB

➢ بعد تعديل الـ Jenkinsfile وتشغيل pipeline جديد:

- الـ unit test اشتغل بنجاح ✅
- اللوجز بتوضح إن التطبيق قدر يتصل بـ MongoDB
- السيرفر اتشغّل، والاختبارات اتنفذت بدون أي error

📋 النتيجة:

المرحلة "Unit Testing" نجحت في الـ pipeline 🎉

7️⃣ نشر تقارير نتائج الاختبار (JUnit Reports)

➢ كل اختبار بيولد ملف نتائج XML اسمه مثلاً : test-results.xml

الملف ده بيتضمن:

- عدد الاختبارات.
- اللي نجحت. ✅
- اللي فشلت. ❌

ولازم Jenkins يحتفظ بيه عشان تقدر تراجعه.

➢ هاتضيف ده في Jenkinsfile كده:

```
stage('Unit Testing') {
    steps {
        withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable: 'MONGO
            sh 'npm test'
        }

        junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
    }
}
```

📌 ده بيخلي Jenkins :

- يقرأ نتائج الاختبارات.
- يعرضها في تبويب خاص باسم Test Results
- حتى لو الملف فاضي (فبسبب allowEmptyResults: true) الـ build مش هيفشل.

8️⃣ عرض النتائج داخل Jenkins UI

➢ بعد تنفيذ الـ build :

- في Blue Ocean UI ← تشوف المرحلة "Unit Testing" ناجحة ✅
- في Classic UI :
  - تلاقي تبويب اسمه Test Results فيه كل التفاصيل.
  - تقدر تشوف كل test case على حدة، زي:
    - liveness endpoint
    - readiness endpoint
  - وكل واحدة منهم حالتها passed

ولو عندك آلاف الاختبارات، ممكن تستخدم الـ Classic View لأنها أسهل في التصفح.

🧠 Best Practices مهمة

✅ نصائح مطورين محترفين:

1. متخزنش أي بيانات حساسة (زي passwords أو tokens) جوه Jenkinsfile واستخدم Jenkins Credentials فقط.
2. خلي كل مرحلة من البايبلاين مستقلة وواضحة باسمها.
3. دايمًا استخدم تقارير JUnit علشان تتابع جودة الكود باستمرار.
4. اربط الاختبارات ببيئة Test Database مش Production

📋 الملخص العام

الهدف	الأوامر / الأدوات	النتيجة
إنشاء مرحلة Unit Testing	`npm test`	تشغيل الاختبارات
تمرير متغيرات البيئة	`environment {}`	تعريف الـ Mongo URI
إدارة البيانات الحساسة	`withCredentials()`	استخدام Jenkins Credentials
نشر تقارير الاختبار	`junit testResults: 'test-results.xml'`	عرض النتائج في Jenkins
التحقق من النجاح	Blue Ocean / Classic UI	مرحلة ناجحة ✅

✨ الخلاصة:

الجزء ده علّمنا إزاي نضيف مرحلة Unit Testing احترافية في Jenkins pipeline :

- متصلة بقاعدة بيانات.
- محمية بالـ credentials
- بتنتج تقارير منظمة في Jenkins

Using Option Directive

💡 الفكرة الأساسية:

- ❑ الـ Options Directive في Jenkins بتديك صلاحية إنك تتحكم في سلوك الـ pipeline نفسه أو سلوك stages معينة
  - يعني مش أوامر بتنفيذ شغل (زي build أو test) لكن أوامر بتنظم إزاي الشغل ده يتنفذ.

🧩 أولاً: إيه هي الـ Options Directive ؟

```
options {
    ...
}
```

- ➢ هي عبارة عن بلوك اسمه
- ➢ وده ممكن تكتبه في مستوى الـ pipeline كله أو جوه stage معينة.

الفرق بينهم إن:

- لو كتبته في المستوى العام ← بينطبق على كل الـ stages
- لو كتبته داخل stage ← بيأثر على الـ stage دي بس.

---

1. ⚙️ تفعيل الـ Timestamps

- ➢ الـ timestamps بتخليك تشوف وقت تنفيذ كل أمر في اللوجز.

وده مفيد جدًا لما تكون بتـ debug pipeline وتشوف كل خطوة استغرقت قد إيه .

- ➢ 👷 الكود بيكون كده:

```
stages {
    stage('Installing Dependencies') {
        options { timestamps() }
        steps {
            sh 'npm install --no-audit'
        }
    }
```

- ➢ ⏱️ النتيجة:

لما تفتح الـ console output هتلاقي كل سطر في اللوج متسبت قبله وقت التنفيذ:

```
[2025-10-14T12:05:32] npm install --no-audit
[2025-10-14T12:05:37] Added 3 packages in 5s
```

- ➢ 📍 ملحوظة:

الـ syntax generator في Jenkins مش بيضيف الأقواس () تلقائيًا، فلازم تكتبها بنفسك.

---

2. ⚙️ الـ Retry Option

- ➢ الـ retry() option بتستخدمها لما تكون عندك خطوة ممكن تفشل لأسباب خارجية (زي الاتصال بـ API أو Database)
- ➢ 👷 الشكل:

```
stage('Unit Testing') {
    options { retry(2) }
    steps {
        withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable: 'MONGO
            sh 'npm test'
        }
```

- ➢ 📘 المعنى:

لو الـ stage دي فشلت، Jenkins هيحاول يعيد تشغيلها مرتين قبل ما يعتبرها فشلت نهائيًا.

وده مفيد جدًا في حالات عدم استقرار الشبكة أو الـ external services

❑ 📍 مثال:

- الـ Unit Testing stage كانت بتعتمد على متغير MONGO_URI،
ولما اتشال المتغير ← الـ stage فشلت.

لاحظ انه عملها hash

```
environment {
//    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
}

stages {
    stage('Installing Dependencies') {
        options { timestamps() }
        steps {
            sh 'npm install --no-audit'
        }
    }

    stage('Dependency Scanning') {···
    }

    stage('Unit Testing') {
        options { retry(2) }
        steps {
            withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable: 'MONGO
                sh 'npm test'
            }

            junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
```

- لكن بما إن فيها retry(2) ، هتلاقي ان Jenkins أعاد المحاولة مرتين قبل ما يعلن فشلها النهائي.

---

3. Disable Concurrent Builds ⚙️

➢ ده خيار **Global** بيتحط على مستوى الـ pipeline ، ومقصوده:

- "ماينفعش يبقى في Build شغال و Build تاني يبدأ في نفس الوقت".
- يعني لو build رقم 15 شغال ولسه مخلصش، وبعدين push جديد حصل ← build 16 هيتعامل حسب الإعداد ده.

➢ 👷 الكود:

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
}

options {
    disableResume()
    disableConcurrentBuilds abortPrevious: true
}
stages {
    stage('Installing Dependencies') {
        options { timestamps() }
```

➢ 📘 المعنى:

- abortPrevious: true ← لو فيه build شغال، Jenkins هيوقفه ويبدأ الجديد.
- abortPrevious: false ← هيتجاهل الجديد لحد ما القديم يخلص.

❑ 📍 في الفيديو:

- الـ build رقم 15 كان بينفذ sleep 100s
- لما المستخدم عمل commit جديد ← build 16 بدأ.
- Jenkins قطع build 15 وقال في اللوج : Superseded by build #16

✅ يعني أوقف القديم وكمّل بالجديد.

---

4. Disable Resume on Controller Restart ⚙️

- ده خيار **Global** يردو ويمنع Jenkins من استكمال الـ pipeline لو السيرفر نفسه Restart

```
options {
    disableResume()
}
```

- المعنى: 📘

لو السيرفر حصل له restart أثناء تنفيذ pipeline ، Jenkins مش هيحاول يكملها بعد الإقلاع — بيبدأ من جديد.

🧱 النتيجة النهائية:

```
    agent any

    options {
        disableConcurrentBuilds(abortPrevious: true)
        disableResume()
    }

    stages {
        stage('Install Dependencies') {
            options { timestamps() }
            steps {
                sh 'npm install --no-audit'
            }
        }

        stage('Unit Testing') {
            options { retry(2) }
            steps {
                sh 'npm test'
            }
        }
    }
}
```

🧠 ملخص الفايدة العملية:

Directive	الوظيفة	الاستخدام الأفضل
`()timestamps`	يضيف الوقت في اللوج	تتبع وقت تنفيذ الأوامر
`retry(n)`	يعيد محاولة الـ stage	عند التعامل مع external services
`()disableConcurrentBuilds`	يمنع تشغيل أكتر من build في نفس الوقت	عند استخدام pipelines بتبني نفس الكود
`()disableResume`	يمنع استئناف build بعد restart	في البيئات الحساسة أو الـ production

Code Coverage and Catch Errors

🎯 الفكرة العامة

❑ الجزء ده بيشرح إزاي نضيف Stage جديدة في الـ Jenkins pipeline علشان نعمل:

1. Code Coverage (قياس تغطية الكود بالاختبارات)
2. ونتعلم كمان إزاي نتعامل مع الأخطاء (errors) اللي بتحصل جوه الـ pipeline بدون ما توقف باقي الـ stages

---

🧩 الخطوات بالتفصيل

1. 🧱 إنشاء Stage جديدة

في الـ Jenkinsfile، بنعمل Stage جديدة اسمها : stage('Code Coverage')

- هي نسخة من مرحلة الـ Unit Testing القديمة.
- لكن بيحذف منها:
  - retry ← لأننا مش محتاجين نكرر الخطوة في حالة الفشل.
  - junit ← لأننا مش بنحلل نتائج الـ unit test هنا.

---

2. 🧠 إعداد الـ Stage

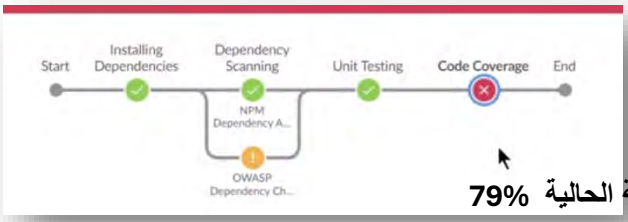
- داخل الـ stage الجديدة بنضيف:

```
stage('Code Coverage') {
    steps {
        withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable: 'MONGO
            sh 'npm run coverage'
        }
    }
}
```

- نستخدم withCredentials لأن السكريبت محتاج يتصل بقاعدة بيانات MongoDB أثناء تشغيل الـ coverage
- الأمر npm run coverage بيشغل الاختبارات ويحسب نسبة التغطية.

---

3. ⚠️ ملاحظة الخطأ



بعد تشغيل الـ pipeline :

- مرحلة code coverage بتفشل ❌
- السبب: النسبة أقل من الحد الأدنى (threshold) وهو 90%، بينما النتيجة الحالية 79%

يعني الـ build فشل عند المرحلة دي ومش هيكمل باقي الـ stages

---

4. 🚨 التعامل مع الخطأ باستخدام catchError

- علشان نمنع توقف الـ pipeline ، نستخدم : catchError
- بنجيبها من : Pipeline Syntax → snippet generator → sample step → catchError

Generate Pipeline Script

```
catchError(buildResult: 'SUCCESS', message: 'Oops! it will be fixed in future releases', stageResult: 'UNSTABLE') {
    // some block
}
```

```
stage('Code Coverage') {
    steps {
        withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable: 'MONGO_PASSWORD', usernameVar:
            catchError(buildResult: 'SUCCESS', message: 'Oops! it will be fixed in future releases', stageResult: 'UNSTABLE') {
                sh 'npm run coverage'
            }
        }
    }
```

✅ المعنى :

- لو حصل error : (لاحظ ان دي ردود الفعل اللي انا حدتهاله وانا بظبط الاعدادات)
  - الـ build كله هيفضل SUCCESSFUL
  - لكن الـ stage دي بس تبان إنها UNSTABLE
  - وبيطبع رسالة "Oops, will be fixed in future releases"
- ➢ دي طريقة ممتازة علشان تكمّل باقي المراحل حتى لو مرحلة واحدة فشلت جزئياً.

---

5. 📄 نشر (Publish) تقرير الـ HTML

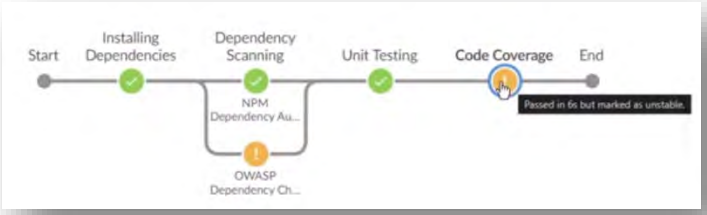
- بعد ما خلص الـ coverage ، Jenkins بيولّد تقرير HTML داخل مجلد: **coverage/lcov-report/index.html**
- فنستخدم نفس كود الـ publish اللي استخدمناه قبل كده:

```
stage('Code Coverage') {
    steps {
        withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable:
        'MONGO_PASSWORD', usernameVariable: 'MONGO_USERNAME')]) {
            catchError(buildResult: 'SUCCESS', message: 'Oops! it will be fixed in future releases',
            stageResult: 'UNSTABLE') {
                sh 'npm run coverage'
            }
        }
        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir:
        'coverage/lcov-report', reportFiles: 'index.html', reportName: 'Code Coverage HTML Report',
        reportTitles: '', useWrapperFileDirectly: true])
    }
}
```

وده بيخلي التقرير يظهر
في Jenkins UI كـ artifact

---

6. 🚀 تنفيذ وبناء جديد



بعد حفظ الكود وتشغيل build جديد:

- المرحلة دي بتظهر كـ unstable بدل ما كانت failed
- الـ pipeline كمل المراحل اللي بعدها بنجاح ✌️
- وكمان التقرير الـ HTML اتنشر، تقدر تفتحه من:
  - Blue Ocean ← في قسم Artifacts
  - أو Classic UI ← تضغط على Code Coverage HTML Report

---

7. 📊 محتوى التقرير

التقرير بيعرض:

- نسبة تغطية الكود (Statements, Branches, Functions, Lines)
- وأي سطور غير مغطاة بالاختبارات.
- في حالتنا، التقرير خاص بملف app.js

💡 النتيجة النهائية

- أضفنا Stage جديدة لقياس تغطية الكود.
- تعاملنا مع الخطأ من غير ما يوقف الـ pipeline
- نشرنا تقرير HTML بشكل منظم داخل Jenkins

Refactoring Jenkinsfile

🎯 الفكرة العامة

- ❑ الجزء ده بيركّز على Refactoring (إعادة تنظيم الكود) داخل الـ Jenkinsfile بحيث يكون الكود أنظف وأكثر قابلية لإعادة الاستخدام، عن طريق:

1. تعريف Environment Variables على مستوى الـ pipeline كله بدل ما نكرّر كود الـ withCredentials في كل stage
2. استخدام Secret Text Credentials بدلاً من username/password pair
3. وأخيراً، نقل أكواد الـ reports (JUnit و HTML) إلى post always section عشان تتنفذ بعد كل build تلقائيًا، سواء ناجح أو فاشل.

---

1. 🧱 مشكلة الكود الأصلي

- ❑ في النسخة القديمة من الـ Jenkinsfile :
  - كل stage (زي Unit Test و Code Coverage) كانت بتستخدم نفس الكود:
    - o withCredentials([usernamePassword(credentialsId: 'mongo-creds', usernameVariable: 'MONGO_USER', passwordVariable: 'MONGO_PASS')])
  - ده بيكرّر نفس البلوك في أكتر من مكان.
  - وبالتالي : لو credential ID تغيّر، لازم نعدّل في كل stage يدويًا.

📌 الهدف : نخلي الـ credentials دي متعرفة في مكان واحد على مستوى الـ Global

---

2. 🌍 تعريف متغيرات البيئة (Environment Variables)

- أول خطوة: نضيف جزء جديد في بداية الـ Jenkinsfile :

```
}

environment {
    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
    MONGO_DB_CREDS = credentials('mongo-db-credentials')
}
```

🧠 الفكرة:

- كده بنعرّف credential واحدة على مستوى الـ pipeline كله.
- المتغير ده بيولّد تلقائي 3 متغيرات:
  - o MONGO_DB_CREDS ← بتحتوي على"username:password"
  - o MONGO_DB_CREDS_USR ← اسم المستخدم فقط
  - o MONGO_DB_CREDS_PSW ← الباسورد فقط

---

3. 🧪 اختبار المتغيرات

- وعلشان الـ variables دي تسمع في الـ stages عندي .. هايبقي من خلال (أمر الـ echo للثلاث متغيرات اللي بيتولدوا) وبالتالي لو مثلا الـ credentials اتغيرت .. فأوتوماتيك الـ echo هاتطبعلي الـ credentials الجديدة
- بعد ماخلاص عرفت الـ Environment Variables علي مستوي الـ Global .. هاتيجي بقا في المراحل اللي مستخدمة الـ vars ده زي مرحلة الـ Unit Testing وتعرف المتغيرات بالطريقة دي :

```
stage('Unit Testing') {
    options { retry(2) }
    steps {
        sh 'echo Colon-Separated - $MONGO_DB_CREDS'
        sh 'echo Username - $MONGO_DB_CREDS_USR'
        sh 'echo Password - $MONGO_DB_CREDS_PSW'
        sh 'npm test'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
    }
```

🔍 النتيجة:

- المتغير الأساسي $MONGO_DB_CREDS بيطبع علامة # لأنه بيحتوي على بيانات حساسة (password masked)
- $MONGO_DB_CREDS_USR بيعرض الـ username عادي.
- $MONGO_DB_CREDS_PSW بيظهر masked برضه.

💡 كده تأكدنا إن الـ environment variables شغالة تمام.

---

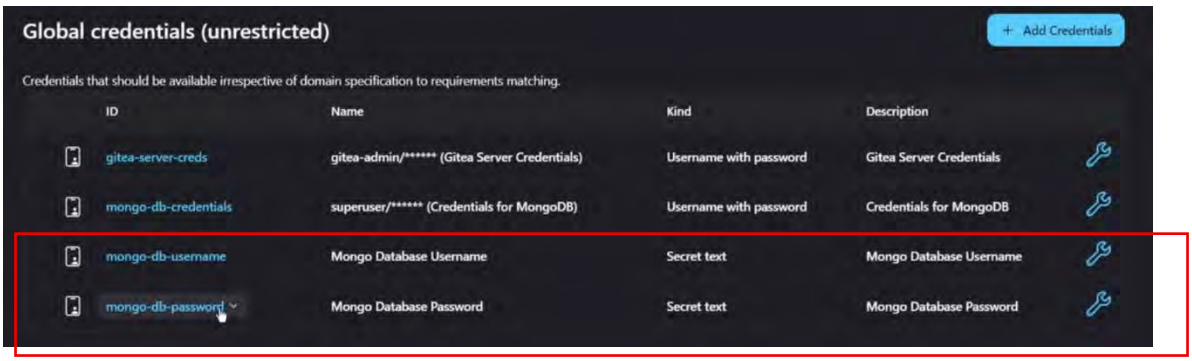
4. ⚠️ المشكلة الجديدة

- لكن مرحلة الـ Unit Test فشلت ❌
  - السبب إن سكريبت الـ test نفسه بيستخدم متغيرات مختلفة:
    - MONGO_USERNAME
    - MONGO_PASSWORD
- مش نفس أسماء اللي عرفناها (MONGO_DB_CREDS_USR و PSW)
- يعني الكود متوقع أسماء محددة .. عشان كده Jenkins مش قادر يمرر الـ credentials بشكل صحيح.

---

5. 🧩 الحل: Secret Text Credentials

- بدل ما نستخدم credential واحدة فيها username/password ، بنعمل اتنين منفصلين:
  1. Secret text فيه الـ username (مثلاً superuser)
  2. Secret text فيه الـ password
- في Jenkins GUI :
- تروح إلى Manage Jenkins → Credentials → Add Credentials :
- تختار Kind: Secret Text
- وتكتب القيمتين. (superuser w password)



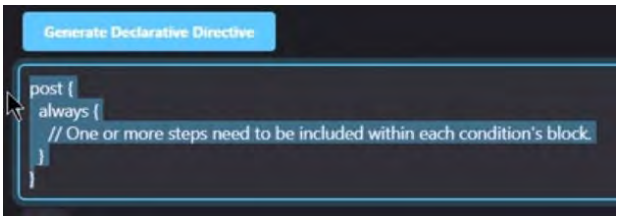
- بعدين نعرفهم في الـ Jenkinsfile :

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
    MONGO_DB_CREDS = credentials('mongo-db-credentials')
    MONGO_USERNAME = credentials('mongo-db-username')
    MONGO_PASSWORD = credentials('mongo-db-password')
}
```

➢ ✅ كده كل stages تقدر تستخدمهم بسهولة:

```
stage('Unit Testing') {
    options { retry(2) }
    steps {
        sh 'echo Colon-Separated - $MONGO_DB_CREDS'
        sh 'echo Username - $MONGO_DB_CREDS_USR'
        sh 'echo Password - $MONGO_DB_CREDS_PSW'
        sh 'npm test'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
    }
```

6. 🧱 تحسين إضافي – نقل تقارير الـ Reports إلى post always



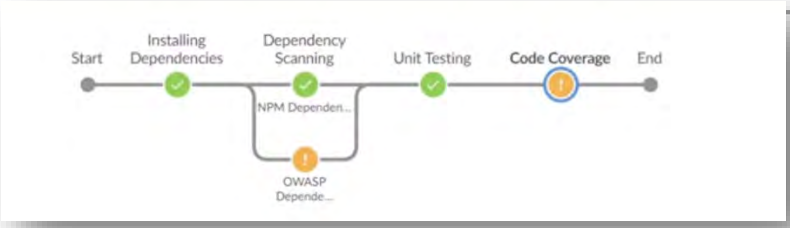
- قبل كده كنا بنكتب أكواد النشر داخل كل stage .. وده برضه تكرار مش ضروري.
- 🧠 الحل :
  - نضيف section في آخر الـ Jenkinsfile اسمه (Global) post always
  - بنجيب الـ syntax من : Pipeline Syntax → snippet generator → sample step → post: Post stage or build con

```
        stage('Code Coverage') {···
        }
    }

    post {
        always {
            junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
            junit allowEmptyResults: true, stdioRetention: '', testResults: 'dependency-check-junit.xml'
            publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './', reportF
            publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: 'coverage/lco
        }
    }
}
```

- 📌 النتيجة:
  - الأكواد دي هتتنفذ دايمًا بعد أي build
  - سواء الـ build ناجح، فشل، أو unstable

---

7. 🚀 بعد الحفظ والتشغيل



- بعد ما عمل commit و push :
  - Jenkins شغّل build جديد.
  - كل الـ stages اشتغلت بنجاح ✅
  - الـ Unit Testing و Code Coverage قدروا يستخدموا الـ credentials الجديدة.
  - بعد انتهاء الـ stages ، حصلت أوتوماتيكيًا:
    - Archiving للـ JUnit reports
    - ونشر الـ HTML Reports

✅ النتيجة النهائية

الهدف	النتيجة
تقليل التكرار في الكود	استخدمنا environment variables عامة
إصلاح فشل اختبار الاتصال بـ MongoDB	استخدمنا Secret Text Credentials
تنظيم التقارير	نقلناها إلى post always stage
تحسين الصيانة المستقبلية	كل حاجة متعرفة في مكان واحد

String Interpolation

🎯 الفكرة العامة

❑ الدرس بيتكلم عن:

- إزاي Jenkins بيتعامل مع النصوص (Strings) والمتغيرات داخل الـ pipeline script
- وده من خلال حاجة اسمها String Interpolation
- بمعنى بسيط:

لما تكتب متغير جوه نص، هل Jenkins هيعرض قيمته؟ ولا هيعرض الكود زي ما هو؟

❑ وده من أهم الدروس في الـ Jenkinsfile syntax لأنك هتحتاجه في كل مرة تستخدم متغيرات أو تكتب expressions داخل stages.

🧩 أولًا: ما هي الـ String Interpolation ؟

- هي طريقة تخلّي Jenkins أو Groovy عمومًا يستبدل المتغيرات داخل النصوص بقيمها الحقيقية.
- مثال بسيط:
  - def name = "Youssef"
  - echo "Hello $name"
  - الناتج ← Hello Youssef
- لكن لو كتبتها كده:
  - echo 'Hello $name'
  - الناتج ← Hello $name

لأن الـ single quotes مش بتسمح بالـ interpolation

💬 الفرق بين Single Quotes و Double Quotes

نوع الـ Quotes	مثال	النتيجة	التفسير
'single quotes'	'Hello $name'	Hello $name	الـ variable مش بيتفك. بيتطبع زي ما هو.
"double quotes"	"Hello $name"	Hello Youssef	بيستبدل name$ بقيمته.

📌 ملحوظة:

ده نفس السلوك في لغات زي Groovy و Bash .. double quotes تسمح بالـ interpolation ، single لأ.

🧱 مراحل الـ Pipeline اللي اتعملت في الفيديو

❑ New Item → Enter item name (string-interpolation-demo) → select Pipeline → ok

❑ الـ Jenkinsfile هنا معمول فيه Stages كتيرة عشان يجرب كل الحالات 👇

🔹 Stage 1: Basic String

```
stages {
    stage('Print Basic String') {
        steps {
            script {
                echo 'Basic String Interpolation Examples:'
            }
        }
```

echo "Hello Jenkins"

- بس بيطبع نص عادي، مفيش متغيرات أو interpolation
- الهدف منه إنه يكون baseline

🔹 Stage 2: Variables Interpolation

```
    stage('Interpolation with Variable') {
        steps {
            script {
                def name = 'Jenkins User'
                echo 'Hello, ${name}!' //using single quotes
                echo "Hello, ${name}!" //using DOUBLE quotes
            }
        }
```

```
def name = 'Jenkins User'
echo 'Hello $name'    // بدون interpolation
echo "Hello $name"    // فيها interpolation
```

🔍 النتيجة:

- الأولى ← Hello $name
- التانية ← Hello Jenkins User

🧠 الفكرة:

Jenkins أو Groovy ← بيفك المتغير بس جوه double quotes

🔹 Stage 3: Parameters Interpolation

```
    parameters {
        string (name: 'USER_NAME', defaultValue: 'Hello, Jenkins from Parameter', description: 'Name of the user')
    }
```

Globally

```
stage('Interpolation with Parameter') {
    steps {
        script {
            echo "Hello, ${params.USER_NAME}!"
        }
    }
```

In Stage

📌 الناتج:

Hello Jenkins from parameters

🎯 الفكرة:

- بنستخدم params.<parameter_name> للوصول لقيم الـ parameters اللي المستخدم بيدخلها وقت تشغيل الـ job
- وممكن نغيرها من الـ build interface

🔹 Stage 4: Environment Variables Interpolation

```
    environment {
        // Example environment variable
        GREETING = 'Hello, Jenkins! from environment variable'
    }
```

Globally

```
stage('Interpolation with Environment Variable') {
    steps {
        script {
            echo "Environment Variable Greeting: ${env.GREETING}"
        }
    }
```

In Stage

📌 الناتج:

Env variable: Hello Jenkins from environment variables

🎯 الفكرة:

- أي environment variable ممكن توصله كده:
- env.<VARIABLE_NAME>
- ضروري تستخدم double quotes عشان يتفك جوه النص.

🔹 Stage 5: Interpolation with Expressions

```
stage('Interpolation with Expression') {
    steps {
        script {
            def x = 5
            def y = 10
            echo "Sum of x and y is: ${x + y}"
        }
    }
```

📌 الناتج:

Sum: 15

🎯 الفكرة:

- ممكن تستخدم interpolation كمان مع عمليات حسابية أو expressions
- لكن لازم تكتبها جوه ${} وتكون جوه double quotes

🔹 Stage 6: Complex Interpolation (Lists & Loops)

```
stage('Complex Interpolation') {
    steps {
        script {
            def list = [1, 2, 3]
            echo "The list has ${list.size()} items: ${list.join(', ')}"
        }
    }
```

📌 الناتج:

List size: 3

Joined list: 1, 2, 3

🎯 الفكرة:

- ممكن تتعامل مع Objects و Functions جوه interpolation
- ${list.join(', ')} معناها "جمع عناصر الـ list بفاصل comma + space "

🔹 Stage 7: Job Parameters (Build Info)

```
stage('Job Parameters') {
    steps {
        script {
            def buildNumber = currentBuild.number
            echo "This is build number ${buildNumber}"
        }
    }
}
```

📌 الناتج:

This is build number 1

🎯 الفكرة:

- Jenkins بيخزن متغيرات جاهزة زي:
  - env.BUILD_NUMBER
  - env.JOB_NAME
  - env.BUILD_URL
- وتقدر تستخدمهم عادي داخل أي stage

🧠 الملخص

الحالة	مثال	بيشتغل؟
Single Quotes	`'Hello $name'`	❌
Double Quotes	`"Hello $name"`	✅
Parameter Access	`"{params.user_name}$"`	✅
Env Variable	`"{env.GREETING}$"`	✅
Expression	`"{x + y}$"`	✅
Complex Objects	`"{(' ,')list.join}$"`	✅

💡 أهم نصيحة

في Jenkinsfile : دايمًا استخدم double quotes لما تكتب متغيرات أو expressions داخل نصوص.

SonarQube Intro/SAST

🎯 الفكرة العامة

- ❑ الجزء ده بيتكلم عن مرحلة مهمة جدًا في الـ DevSecOps pipeline وهي تحليل الكود الثابت (Static Analysis)
- ❑ الفكرة ببساطة إننا بنفحص الكود نفسه (من غير ما نشغله) علشان نكتشف الأخطاء الأمنية والمشاكل في جودة الكود قبل ما نكمل باقي خطوات الـ pipeline
- ❑ الأداة اللي استخدمها هنا هي SonarQube ← منصة قوية مفتوحة المصدر بتعمل تحليل تلقائي للكود وتطلع لك تقارير تفصيلية.

🧩 الشرح خطوة بخطوة

1. 🧠 ما هو الـ Static Analysis ؟

هو نوع من أنواع الاختبارات اللي بيراجع كود المشروع نفسه من غير ما يشغّله، وبيفحص السطور سطر سطر علشان يكتشف:

- مشاكل أمنية (Security Vulnerabilities)
- Bugs محتملة
- Bad practices أو كود محتاج refactoring

🟢 تقدر تعتبره كأنه "code scanner" بيكشفلك الأماكن اللي فيها خطر في الكود.

---

2. 🛠️ الأداة المستخدمة SonarQube

- SonarQube هو منتج مفتوح المصدر من شركة SonarSource
- وظيفته إنه يعمل Continuous Code Quality Monitoring
- بيوفرلك Automated Code Reviews في كل build
- بيتكامل بسهولة مع أدوات CI/CD زي Jenkins أو GitHub Actions

---

3. 💡 ليه نستخدم Static Analysis ؟

الفوائد اللي اتذكرت:

1. 🔍 كشف الأخطاء مبكرًا ← بتكتشف المشاكل بدري وبتوفر وقت وتكلفة الإصلاح.
2. 🧱 تحسين بنية الكود ← بتحددلك الأماكن اللي محتاجة refactor أو تبسيط.
3. ⚙️ فرض قواعد الكود ← تقدر تعرف "coding standards" خاصة بمشروعك وسوناركوب بيتأكد إنها مطبقة تلقائيًا.

---

4. 🔐 التعامل مع النتائج في SonarQube

لما SonarQube يحلل الكود، بيعمل حاجات مهمة:

- يحدد السطور اللي فيها مشاكل أمنية بالضبط.
- يديك نصائح عملية لحلها.
- يطبق حاجة اسمها Quality Gates ← ودي زي checkpoints أو معايير للجودة لازم الكود يعديها علشان يُعتبر آمن وجاهز.

---

5 🧱 Quality Gates Components.

SonarQube بيقيس الجودة بعدة مقاييس، منها:

المكون	معناه
Code Smells	مؤشرات على practices سيئة أو كود محتاج تحسين
Security Hotspots	أماكن ممكن تكون معرضة لهجمات أو ضعف أمني
Code Coverage	نسبة الكود اللي تم اختباره فعلاً في الـ automated tests

🔴 **لو أي شرط من شروط الـ quality gate فشل ← الـ build كله بيفشل.**
وده بيضمن إن مفيش كود ضعيف أو غير آمن يدخل في الـ pipeline أو مرحلة الـ deployment

---

✅ النتيجة النهائية

الهدف من كل ده إنك:

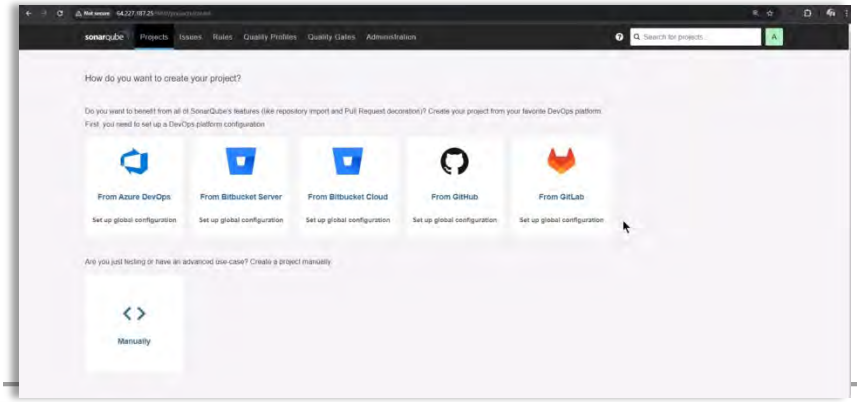
- **تحافظ على مستوى جودة الكود.**
- **تمنع دخول أكواد غير آمنة للبيئة الإنتاجية.**
- **تخلي الأمان وجودة الكود جزء أساسي من دورة التطوير (DevSecOps)**

SAST Analysis

🎯 الهدف العام

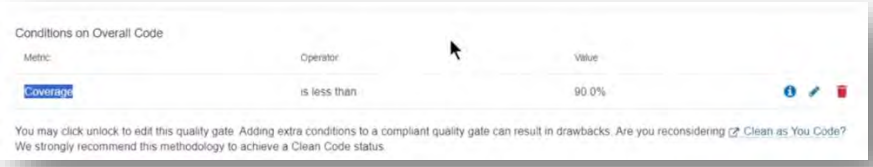
❑ عمل تحليل أمني (Static Analysis) للكود باستخدام SonarQube داخل Jenkins Pipeline ، بحيث:

- يعمل Jenkins build
- يشغّل SonarScanner لتحليل الكود.
- يرفع النتائج لـ SonarQube
- SonarQube يقرر هل الكود "Passed" أو "Failed" بناءً على Quality Gates (زي نسبة الـ Coverage ، عدد الـ Bugs، إلخ)



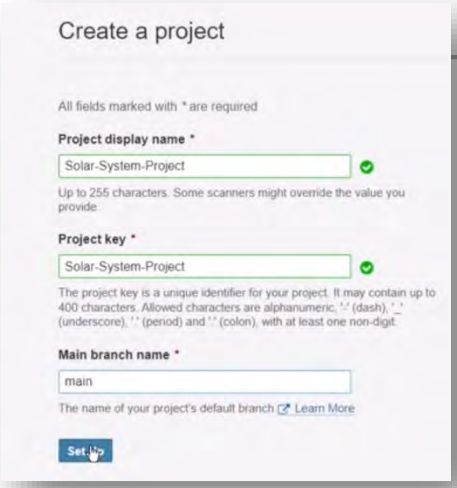
🧩 الخطوات التفصيلية

1. 🧱 إعداد SonarQube

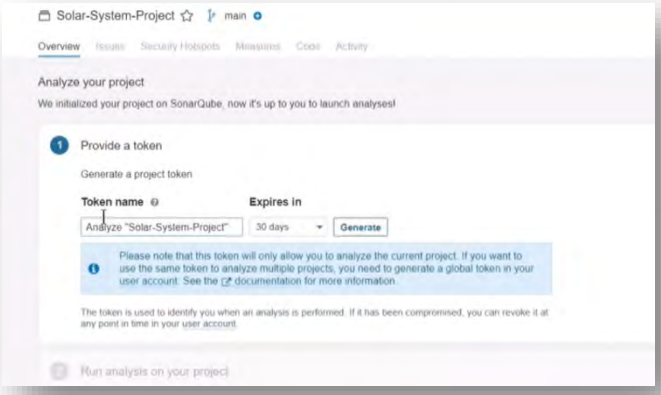


- عندنا SonarQube شغّال على port 9000
- في البداية مفيش أي project
- راح لـ Quality Gates وفعّل واحدة اسمها *Dasher Quality Gate* فيها شرط إن:
  - الـ Coverage لازم يكون ≥ 90%، وإلا المشروع هيفشل.
- خلاها هي الـ default gate

2. 🏗️ إنشاء Project جديد



- دخل على "Create project manually" ← SonarQube
- سمّى المشروع : Solar System Project
- حدّد الـ main branch باسم main
- لاااحظ انه اختار طريقة الربط : manual/local (مش Jenkins)
- النظام ولّد token جديد (هيتستخدم للربط بين Jenkins و SonarQube)



📌 ملاحظة: اختار *no expiration* لكن الأفضل دايمًا تختار صلاحية قصيرة.

3. ⚙️ اختيار نوع المشروع

- اختار إن المشروع من نوع Node.js (JavaScript)
- هيشتغل على Linux
- بعد كده SonarQube طلعله أمر جاهز لتشغيل التحليل، ممكن ينفذه من Jenkins أو من CLI :

Execute the Scanner

Running a SonarQube analysis is straighforward. You just need to execute the follo

```
sonar-scanner \
  -Dsonar.projectKey=Solar-System-Project \
  -Dsonar.sources=. \
  -Dsonar.host.url=http://64.227.187.25:9000 \
  -Dsonar.login=sqp_54484dbbbe3a5b3b3088c734cf5e4c3bbeba3fd6
```

Please visit the official documentation of the Scanner for more details.

---

4. 🔧 تعديل Jenkinsfile

- أضاف Stage جديدة بعد code coverage اسمها:

```
stage('Code Coverage') {...
}

stage('SAST - SonarQube') {
    steps {
        sh '''
            sonar-scanner \
                -Dsonar.projectKey=Solar-System-Project \
                -Dsonar.sources=. \
                -Dsonar.host.url=http://64.227.187.25:9000 \
                -Dsonar.login=sqp_54484dbbbe3a5b3b3088c734cf5e4c3bbeba3fd6
        '''
```

لاحظ الفرق بين السطر ده ونفس السطر في اخر صورة في الصفحة ومهم تعرف السبب

🔍 التحليل:

- sonar.projectKey : بيحدد اسم المشروع في SonarQube
- sonar.sources : بيحدد ملفات التحليل (هنا app.js)
- sonar.host.url : رابط السيرفر.
- sonar.login : التوكن اللي اتولد. (الدرس الجاي هانوضح ازاي نستخدم الـ token دي كـ credentials)

🚀 الأداه اللي بتنفذ التحليل هي SonarScanner CLI

---

5. 🧰 تثبيت SonarScanner في Jenkins

- فتح Jenkins → Manage Jenkins → Plugins
- بحث عن SonarQube Scanner → Installed → Restart Jenkins
- بعد الريستارت:
  - راح لـ Manage Jenkins → Tools
  - أضاف tool جديد:
    - Name: SonarQube Scanner 6.1.0
    - Install automatically → From Maven Central.

---

6. 🧩 تعريف متغير البيئة

في Jenkinsfile ، أضاف environment variable :

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
    MONGO_DB_CREDS = credentials('mongo-db-credentials')
    MONGO_USERNAME = credentials('mongo-db-username')
    MONGO_PASSWORD = credentials('mongo-db-password')
    SONAR_SCANNER_HOME = tool 'sonarqube-scanner-610';
}
```

ثم استخدمه في الـ shell :

```
stage('SAST - SonarQube') {
    steps {
        sh 'echo $SONAR_SCANNER_HOME'
        sh '''
            $SONAR_SCANNER_HOME/bin/sonar-scanner \
                -Dsonar.projectKey=Solar-System-Project \
                -Dsonar.sources=app.js \
                -Dsonar.host.url=http://64.227.187.25:9000 \
                -Dsonar.login=sqp_54484dbbbe3a5b3b3088c734cf5e4c3bbeba3fd6
        '''
```

📌 الهدف من الخطوة دي: عشان Jenkins يقدر يلاقي الـ SonarScanner CLI من الـ PATH الصحيح بدون Errors

7. 🚀 تشغيل Pipeline

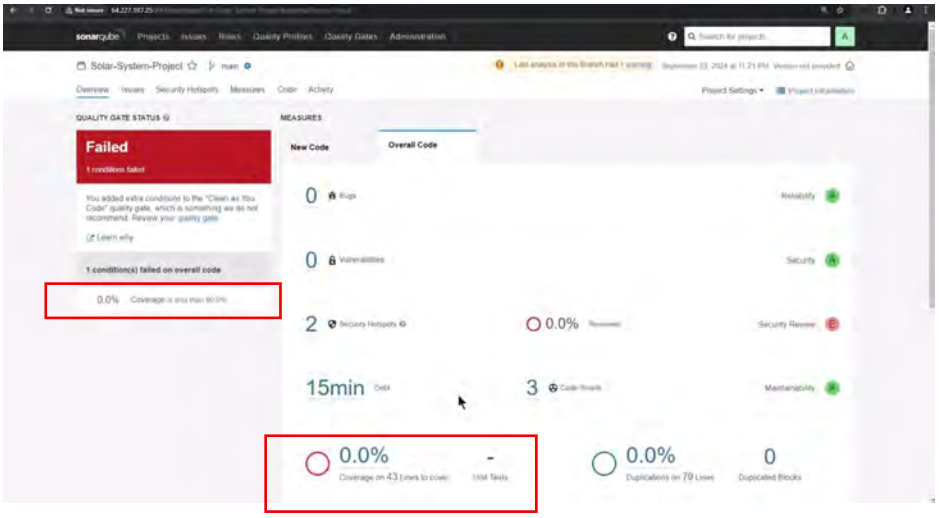
- عمل commit و push
- Jenkins نفذ الـ pipeline
- مرحلة SonarQube نجحت ✅ ورفعت التحليل للسيرفر.

---

8. 📊 تحليل النتائج على SonarQube Dashboard

بعد الريفريش:

- المشروع ظهر باسم Solar System Project
- فيه:
  - 🔒 2 security hotspots
  - 🧹 3 code smells
  - ⏱ 15 min debt
  - Coverage = 0%
- النتيجة : Quality Gate = FAILED لأن التغطية أقل من 90%



---

9. 🧠 حل مشكلة الـ Coverage = 0%

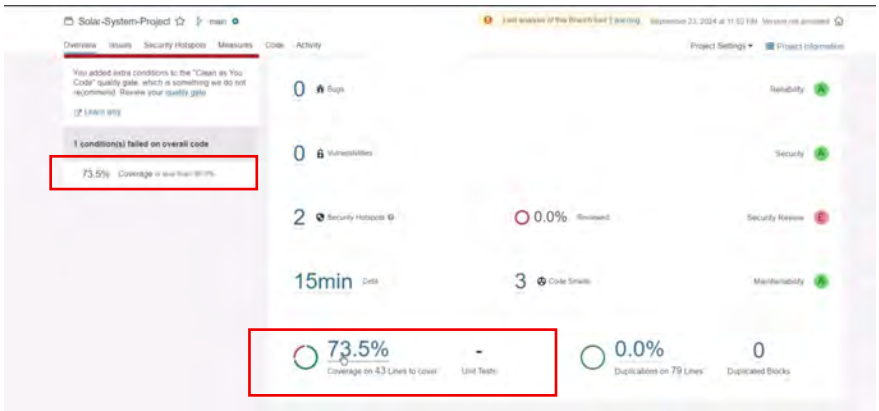
🔍 السبب : SonarQube مش عارف مكان ملف الـ coverage

- الملف بيتولد في : workspace/coverage/lcov.info
- SonarQube بيحتاج تعرفه عليه بإضافة argument ده في الـ Jenkinsfile :

```
stage('SAST - SonarQube') {
    steps {
        sh 'echo $SONAR_SCANNER_HOME'
        sh '''
            $SONAR_SCANNER_HOME/bin/sonar-scanner \
                -Dsonar.projectKey=Solar-System-Project \
                -Dsonar.sources=app.js \
                -Dsonar.host.url=http://64.227.187.25:9000 \
                -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info \
                -Dsonar.login=sqp_54484dbbbe3a5b3b3088c734cf5e4c3bbeba3fd6
        '''
    }
}
```

10. 📄 إعادة تشغيل الـ Pipeline

- عمل commit + push جديد.
- Jenkins شغّل الـ build
- SonarQube دلوقتي قرأ الـ coverage من الملف.
- بعد التحديث ← ✅ coverage = 73%
- لكن مازالت أقل من 90 % ← Quality Gate Failed



---

11. ⚠️ الملاحظة الأخيرة

- Jenkins شايف الـ stage ناجحة ✅ لأن SonarQube رجّع "analysis uploaded successfully"
- لكن SonarQube نفسه شايفها "Failed" بسبب الـ Quality Gate
- المشكلة : SonarQube مش بيرجع الحالة (Quality Gate status) لـ Jenkins
- الحل هيكون في الفيديو الجاي:
  - ربط SonarQube webhook أو integration بحيث الـ Quality Gate status يتبعت لـ Jenkins ويأثر على نتيجة الـ Pipeline

---

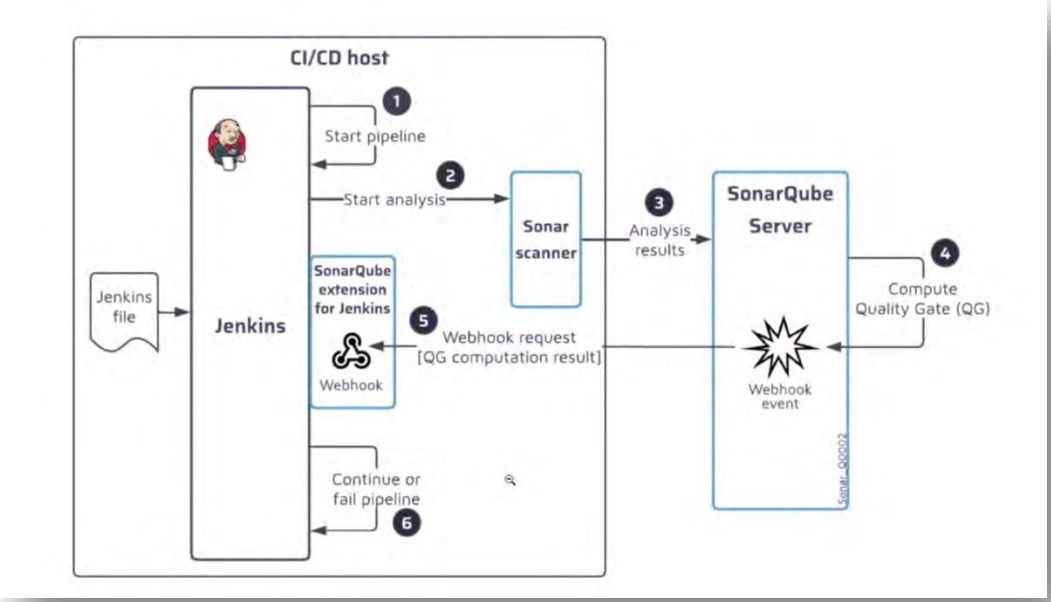
الخلاصة

المرحلة	الغرض	النتيجة
إعداد SonarQube	إنشاء مشروع وQuality Gate	جاهز للتحليل
إعداد Jenkins	تثبيت SonarScanner	✅
كتابة Jenkinsfile	تشغيل SAST stage	✅
تحليل النتائج	رؤية التقارير والـ coverage	✅
Quality Gate	فشل بسبب <90%	❌
المرة الجاية	ربط الحالة بين SonarQube وJenkins	SOON

SonarQube Quality Gate Step and Refactoring

🎯 الفكرة العامة

- ❑ احنا عايزين نربط Jenkins مع SonarQube بحيث لما SonarQube يحلل الكود ويكتشف مشاكل، يبعت الحالة (Pass / Fail) لجينكنز ← وجينكنز يا إما يكمل البايبلاين، يا إما يوقفها.



🧩 Steps Overview

🔹 Step 1: Trigger Jenkins Pipeline

- أول ما البايبلاين تبدأ في Jenkins ، بيبدأ تشغيل الـ job

🔹 Step 2: Run Sonar Scanner

- البايبلاين بتشغل SonarScanner لتحليل الكود (static analysis)

🔹 Step 3: Send Results to SonarQube

- الـ SonarScanner يبعث نتائج التحليل للسيرفر بتاع SonarQube

🔹 Step 4: SonarQube Calculates Quality Gate

- SonarQube بيحسب إذا كان المشروع Passed ولا Failed بناءً على قواعد الـ Quality Gate

---

🔹 Step 5: Send Status Back via Webhook

- بعد التحليل، SonarQube يبعث النتيجة إلى Jenkins عن طريق webhook URL

🔹 Step 6: Jenkins Acts on Result

- Jenkins يا إما يكمل البايبلاين (لو Passed)
- يا إما يوقفها (لو Failed)

Configuration Details ⚙️

1. 🧠 إعداد الـ Webhook في SonarQube

- جينكنز بيكون عنده endpoint كده : http://<jenkins-server>:8080/sonarqube-webhook
- ده اللي SonarQube بيستخدمه علشان يبعث نتيجة التحليل.
- بتروح في SonarQube → Administration → Webhooks → Create
  - Name: Jenkins Webhook
  - URL: http://<jenkins-server>:8080/sonarqube-webhook
  - بدون secret
- كده SonarQube جاهز يبعث النتيجة مباشرةً.

---

2. 🧠 إعداد الـ SonarQube Server داخل Jenkins

- تروح في Jenkins → Manage Jenkins → System Configuration
- تضيف SonarQube Server :
  - Name: SonarQube Server
  - Server URL: http://<sonarqube-server>:9000
  - Token : بتجيبه من SonarQube → My Account → Security → Generate Token
  - Server authentication token: SonarQube Server Token
- بعد كده بتحفظ.
- لااحظ انك بتضيف الـ token ده كـ Secret Text Credential (🔐 تأمين التوكن بدل ما يكون مكتوب في Jenkinsfile)

  في Jenkins → Manage Jenkins → Credentials → press Global then Add Credentials
  - Kind: secret text
  - Secret: paste token which generated from sonar-qube
  - ID: SonarQube-token
  - Description: SonarQube Server Token

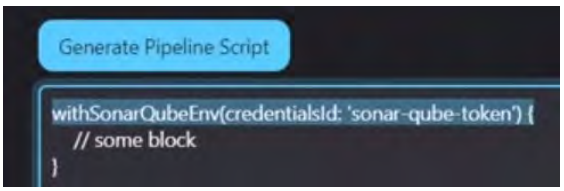
لاحظ ان الخطوة دي مش هاتظهرلك الا اما تعمل الخطوات دي

---

3. 🧠 تعديل Jenkinsfile (Refactoring)

➢ بدل ما تحط الـ token في الكود (وده unsafe) .. بتستخدم environment block اسمه (withSonarQubeEnv)

- بتجيبه من Pipeline Syntax → snippet generator → sample step → post: withSonarQubeEnv: Prepare..
- Name: sonar-qube-server
- server authentication token: SonarQube Server Token




```
stage('SAST - SonarQube') {
    steps {
        withSonarQubeEnv('sonar-qube-server') {
            sh 'echo $SONAR_SCANNER_HOME'
            sh '''
                $SONAR_SCANNER_HOME/bin/sonar-scanner \
                    -Dsonar.projectKey=Solar-System-Project \
                    -Dsonar.sources=app.js \
                    -Dsonar.host.url=http://64.227.187.25:9000 \
                    -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info \
                    -Dsonar.login=sqp_54484dbbbe2e5b3b2088e724cf5e4e3bbeba3fd6
            '''
        }
    }
}
```

هانمسح السطر ده لأننا عرفنا الـ host URL within Installation

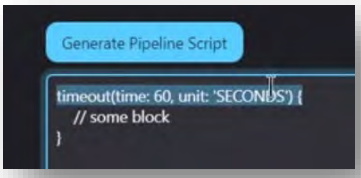
- كده Jenkins بيستخدم إعدادات السيرفر من الـ configuration ، مش من الكود نفسه ✅

وطبعا كمان هانمسح الـ Token لانه خلاص بقا جزء من الـ WithSonarQubeEnv

نفس الفكرة بس دي الطريقة الـ Declarative بتاعته وجيبناها او عرفناها من الـ Steps Reference هتلاقيها في الـ Pipeline Syntax

4. 🧠 انتظار نتيجة الـ Quality Gate

- بعد ما SonarQube يخلص التحليل، Jenkins يستنى النتيجة
- بتجيبه من Pipeline Syntax → snippet generator → sample step → timeout: Enforce time limit
  - Timeout: 60
  - Unit: Seconds



Example using declarative pipeline:

```
pipeline {
  agent none
  stages {
    stage("build & SonarQube analysis") {
      agent any
      steps {
        withSonarQubeEnv('My SonarQube Server') {
          sh 'mvn clean package sonar:sonar'
        }
      }
    }
    stage("Quality Gate") {
      steps {
        timeout(time: 1, unit: 'HOURS') {
          waitForQualityGate abortPipeline: true
        }
      }
    }
  }
}
```

```
stage('SAST - SonarQube') {
    steps {
        timeout(time: 60, unit: 'SECONDS') {
            withSonarQubeEnv('sonar-qube-server') {
                sh 'echo $SONAR_SCANNER_HOME'
                sh '''
                    $SONAR_SCANNER_HOME/bin/sonar-scanner \
                        -Dsonar.projectKey=Solar-System-Project \
                        -Dsonar.sources=app.js \
                        -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info
                '''
            }
            waitForQualityGate abortPipeline: true
        }
```

- waitForQualityGate : بيوقف البايبلاين لحد ما SonarQube يرد بالنتيجة.
- abortPipeline: true : يعني لو النتيجة فشلت ← أوقف البايبلاين فورًا.
- Timeout : علشان ما يستناش إلى الأبد (مثلاً أقصى انتظار = 60 ثانية).

---

5. 🧹 تحسين أداء OWASP Dependency Check

- في الـ pipeline القديم كان الـ OWASP analyzer بياخد وقت كبير لأنه بيحاول يشغل حاجة اسمها YARN Audit Analyzer (مش مستخدمة حيث أنه استخدم بدالها الـ npm audit), وبالتالي هايعملها Disable
- هو عمل optimization بسيط في الـ Dependency Check Stage :
  - كان الـ "YARN audit analyzer" بيطلع أخطاء كتير وبيبطأ الـ build
  - فتم تعطيله بإضافة : --disableYarnAudit
  - الهدف تقليل وقت التحليل وتجنب errors غير ضرورية .

```
stage('OWASP Dependency Check') {
    steps {
        dependencyCheck additionalArguments: '''
            --scan \'./\'
            --out \'./\'
            --format \'ALL\'
            --disableYarnAudit \
            --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'

        dependencyCheckPublisher failedTotalCritical: 1, pattern: 'dependency-check-report.xml',
```

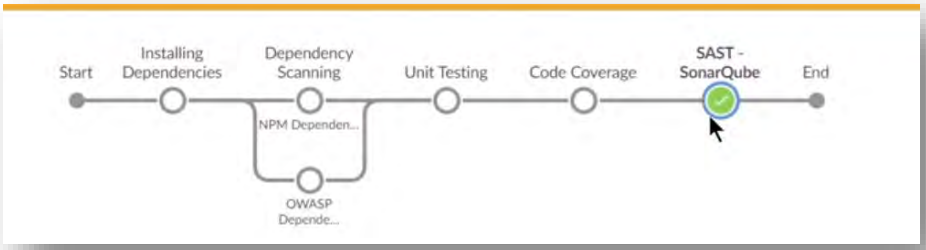
🚨 النتائج بعد التنفيذ

❑ الحالة الأولى Quality Gate Failed :

- SonarQube رجع إن النتيجة = Error (مثلاً بسبب '90%' > 'Code Coverage 73%')
- الـ webhook بعت النتيجة دي لـ Jenkins
- Jenkins شاف إنها fail ← أوقف الـ pipeline تلقائيًا
- 🔴 وبالتالي الـ SonarQube stage تظهر كـ Failure

لاحظ انك ممكن تشوف التفاصيل دي من:

- SonarQube → Projects → Quality Gate → Condition failed
- Webhooks → Delivery details (status = ERROR)



❑ الحالة الثانية Quality Gate Passed :

- بعد تعديل threshold لـ أقل من 70%
- الـ pipeline rerun تاني
- SonarQube رجع النتيجة OK
- Jenkins كمل بنجاح ✅

🧠 ملخص الفكرة

❑ الهدف كله إنك تضمن إن الكود اللي بيكمل في الـ CI/CD pipeline يكون نظيف وآمن،

❑ ولو الـ SonarQube لقى مشاكل خطيرة ← الـ pipeline يوقف تلقائيًا بدل ما يكمل نشر كود فيه ثغرات.

Summary

الخطوة	الهدف	الأداة
Configure Webhook	إرسال status من SonarQube → Jenkins	SonarQube
Secure Token	حماية الـ credentials	Jenkins Credentials
Wait for Quality Gate	منع الكود السيء من المرور	Jenkinsfile
Timeout	يمنع الانتظار الطويل	Jenkinsfile
Refactor YARN audit	تحسين وقت التحليل	Dependency Check

Build Docker Imange

- الجزء ده هو المرحلة اللي بعد تحليل SonarQube في الـ Jenkins Pipeline .. واللي فيها بنبدأ نعمل Docker image للتطبيق.
- يعني لحد دلوقتي، احنا كنا بنعمل Quality Check على الكود... لكن هنا بدأنا ندخل في مرحلة Build المرحلة الحقيقية للتطبيق داخل Container

---

🎯 الهدف

- الهدف هنا إننا نضيف Stage جديدة داخل الـ Jenkinsfile اسمها : "Build Docker Image"
  - وده علشان نخلّي Jenkins يبني Docker image للتطبيق بشكل أوتوماتيكي بعد ما يخلص تحليل الجودة (SonarQube)

```
stage('Build Docker Image') {
    steps {
        sh 'printenv'
        sh 'docker build -t siddharth67/solar-system:$GIT_COMMIT .'
    }
}
```

⚙️ الفكرة العامة

- Jenkins لما يوصل للـ stage دي، هيشغّل أمر:
  - docker build -t <username>/<image-name>:<git-commit> .
- وده بيعمل:
  - يبني Image من الـ Dockerfile اللي في المشروع.
  - يسميها باسم مستخدم DockerHub (هنا اسمه siddharth67)
  - ويضيف ليها Tag هو رقم الـ commit الحالي من Git ، علشان كل Build يكون له نسخة مميزة بالـ commit ID

🧩 شرح مكونات الـ Stage خطوة بخطوة

```
stage('Build Docker Image') {
    steps {
        sh 'printenv'
        sh 'docker build -t siddharth67/solar-system:$GIT_COMMIT .'
    }
}
```

1. 🧱 تعريف الستيج

✅ معناها:

- اسم الستيج "Build Docker Image"
- بتنادي أمر shell عادي.
- docker build بيستخدم الـ Dockerfile اللي موجود في نفس repo
- اسم الـ image = siddharth67/solar-system
- التاج هايتجاب من المتغير ده ← ${GIT_COMMIT}

🎯 الفكرة إن كل Build جوه Jenkins هيطلع Docker image مختلفة بناءً على commit ID ، وده بيساعدك بعد كده تعرف كل image اتبنت من أن نسخة كود بالضبط.

---

2. 🌍 الـ Environment Variables (المتغيرات اللي Jenkins بيستخدمها)

- استخدمنا هنا المتغير ${GIT_COMMIT}
  - وده متغير Jenkins بيجي تلقائيًا لما الـ pipeline يكون عامل checkout للـ repository

📌 المعلومة المهمة هنا:

- Jenkins دايمًا بيعمل checkout أوتوماتيكي للكود من الـ GitHub Repo في أول الـ pipeline
- بمجرد ما يعمل checkout ← كل متغيرات الـ Git تبقى متاحة لكل الـ stages :
  - GIT_COMMIT ← رقم الـ commit
  - GIT_BRANCH ← اسم الفرع.
  - WORKSPACE ← المسار اللي Jenkins بيبني فيه المشروع.
  - BUILD_ID, BUILD_NUMBER, BUILD_URL ... إلخ.

ممكن تعرف كل الـ variables اللي بتبقي متاحة من خلال :
Pipeline syntax → Global Vars Ref.

وده اللي خلاه يضيف كمان سطر printenv علشان يطبع كل الـ variables دي ويشوف اللي متاح فعلاً في الجلسة دي.

---

3. 🐳 محتوى الـ Dockerfile نفسه

الـ Dockerfile المستخدم بسيط جدًا (Node.js app) ، وكان كالتالي:

```
solar-system > Dockerfile
FROM node:18-alpine3.17

WORKDIR /usr/app

COPY package*.json /usr/app/

RUN npm install

COPY . .

ENV MONGO_URI=uriPlaceholder
ENV MONGO_USERNAME=usernamePlaceholder
ENV MONGO_PASSWORD=passwordPlaceholder

EXPOSE 3000

CMD [ "npm", "start" ]
```

📘 الشرح:

- FROM node:18-alpine ← بيستخدم إصدار خفيف من Node.js
- WORKDIR ← بينشئ مجلد العمل داخل الـ container
- COPY package.json ← ينقل ملفات الـ dependencies
- RUN npm install ← يثبّت الـ dependencies داخل الـ container
- COPY . . ← ينقل باقي ملفات المشروع.
- ENV ← متغيرات بيئية Placeholder (زي MongoDB credentials)
- EXPOSE 3000 ← بيفتح البورت اللي هيشتغل عليه السيرفر.
- CMD ← بيشغّل npm start لما الـ container يبدأ.

---

4. 🚫 ملف dockerignore.

➢ زي .gitignore بالظبط.

- بيستخدمه علشان يمنع نقل ملفات ملهاش لازمة في الـ image (زي node_modules القديمة أو ملفات config الحساسة)

مثلاً:

```
node_modules
.git
Jenkinsfile
.env
```

```
Jenkinsfile M   .dockerignore ×
solar-system > .dockerignore > #
npm-debug.log
# ignore .git and .cache folders
.git
.cache
# ignore all markdown files (md) beside all README*.md other than README-secret.md
*.md
!README*.md
README-secret.md
# github related files
.github/
# nodejs releated files
node_modules
solar-system.png
.nyc_output
.talismanrc
coverage
test-results.xml
#reports
reports*
zap*
dependency*
jenkins*
trivy-image*
```

🎯 النتيجة : حجم الـ image بيكون أصغر وأسرع في البناء.

5. 🧠 أمر Print Environment

➢ sh 'printenv'

- ده بيطبع كل الـ environment variables المتاحة وقت تشغيل الـ pipeline
- فمثلاً بيظهر:
  - BUILD_NUMBER
  - WORKSPACE
  - JENKINS_HOME
  - BUILD_URL
  - GIT_COMMIT
  - NODE_NAME
  - BRANCH_NAME

وده بيساعدك تعرف إيه المتغيرات اللي ممكن تستخدمها في أي stage بعد كده.

---

6. 🧩 تنفيذ فعلي للـ Pipeline

بعد الحفظ والـ commit :

- Jenkins شغّل pipeline جديدة.
- الستيج "Build Docker Image" اشتغلت.
- تم تنفيذ أمر:
  - docker build -t siddharth67/solar-system:9dbc4bb .
- وهنا الـ 9dbc4bb هو أول 7 حروف من الـ commit ID الأخير في Gitea server

📃 بعد الانتهاء:

- تقدر تشوف في Jenkins console log إن الـ image اتبنت بنجاح.
- ولو تروح على Gitea server ← هتلاقي نفس commit ID ظاهر هناك.

✅ النتيجة النهائية

بقى عندك:

- Jenkins pipeline فيها:
  1. مراحل الجودة (SonarQube, Tests...)
  2. ومرحلة Build Docker Image
- Image جاهزة بالتاج الخاص بكل commit
- متغيرات Jenkins واضحة وممكن تستخدمها بعد كده

🚀 المرحلة الجاية

❑ في الدرس الجاي ، بيبدأ يضيف مرحلة Vulnerability Scanning للـ Docker image اللي اتبنت دي (باستخدام أدوات زي Trivy أو Anchore أو OWASP Dependency Check)

❑ وبعدها يعمل Push للـ image على DockerHub

Vulnerability Scan using Trivy

🎯 الفكرة العامة :

- ❑ بعد ما تبني Docker image ، لازم تتأكد إنها مافيهاش ثغرات أمنية (vulnerabilities) قبل ما ترفعها أو تستخدمها في الـ production
- ❑ الأداة اللي بنستخدمها هنا اسمها Trivy

---

1. 🎯 مقدمة عن Trivy

كل الـ steps اللي في الـ stages بكل الكلام اللي فيها .. هتلاقيه في الدكيومنتيشن
Overview - Trivy

- Trivy هي أداة مفتوحة المصدر لعمل security scanning
- بتفحص حاجات زي:
  - Docker/container images
  - File systems
  - Git repositories
  - Kubernetes manifests
- وتقدر تكشف عن:
  - CVEs (الثغرات المعروفة)
  - مشاكل الـ OS packages
  - مشاكل الـ dependencies (زي مكتبات Node.js)
  - مشاكل في الـ Infrastructure as Code
  - مشاكل في تراخيص البرمجيات (licenses)

---

2. ⚙️ طريقة الاستخدام

- تقدر تشغلها بطريقتين:
  1. CLI (Command Line Tool)
  2. Docker image جاهزة

📌 هو استخدم الـ CLI اللي كان مسبت عنده.

---

3. 🧪 الأمر الأساسي لفحص الـ Image : trivy image <image-name>

مثلاً: trivy image myapp:latest

---

4. 🧩 مثال عملي داخل الـ Jenkins Stage

```
stage('Trivy Vulnerability Scanner') {
    steps {
        sh '''
            trivy image siddharth67/solar-system:$GIT_COMMIT \
            --severity LOW,MEDIUM,HIGH \
            --exit-code 0 \
            --quiet \
            --format json -o trivy-image-MEDIUM-results.json

            trivy image siddharth67/solar-system:$GIT_COMMIT \
            --severity CRITICAL \
            --exit-code 1 \
            --quiet \
            --format json -o trivy-image-CRITICAL-results.json
        '''
    }
}
```

➢ لاحظ اننا بنضيف الـ stage دي بعد stage('Build Docker Image')

ايوا زي مانت لااحظت بالظبط .. انه عمل severeity مرتين + لاحظ الـ exit code في الحالتين :

- واحدة للـ LOW, MEDIUM,HIGH
- والثانية للـ CRITICAL

5. خيارات مهمة في Trivy

- تقدر تحدد severity معينة أو تنسّق شكل التقرير (JSON / HTML / XML).
- **--severity** ← تحدد مستوى الخطورة اللي تفحصه (LOW, MEDIUM, HIGH, CRITICAL)
- **--exit-code** ← تحدد هل يحصل Error لو لقى ثغرات ولا لأ
  - **0** = تجاهل الثغرات (مايفشلش الـ pipeline)
  - **1** = يوقف الـ pipeline لو لقى ثغرات
- **--format json** ← يخرج النتيجة كـ JSON
- **--output <file>** ← يحفظ النتيجة في ملف

---

6. تحويل الـ JSON إلى تقارير HTML وXML

Reporting - Trivy

- هانعمل post{} علي مستوي الـ stage بتاع الـ Trivy

```
}
post {
    always {
        sh '''
            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/html.tpl" \
                --output trivy-image-MEDIUM-results.html trivy-image-MEDIUM-results.json

            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/html.tpl" \
                --output trivy-image-CRITICAL-results.html trivy-image-CRITICAL-results.json

            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/junit.tpl" \
                --output trivy-image-MEDIUM-results.xml  trivy-image-MEDIUM-results.json

            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/junit.tpl" \
                --output trivy-image-CRITICAL-results.xml trivy-image-CRITICAL-results.json
        '''
    }
}
```

ليه أربع o\p ؟!
لو ركزت هتلاحظ انه عمل 2 severity
..وهنا احنا هانحول كل severity الي
XML & HTML فبالتالي هايبقي عندنا
اربعة output

- ممكن تستخدم Templates جاهزة زي:
  - html.tpl
  - junit.tpl

هتلاقي الـ templates دي في الـ PATH ده

```
root@jenkins-controller-1 in ~
❯ ls /usr/local/share/trivy/templates
asff.tpl  gitlab-codequality.tpl  gitlab.tpl  html.tpl  junit.tpl
```

---

7. نشر التقارير في Jenkins

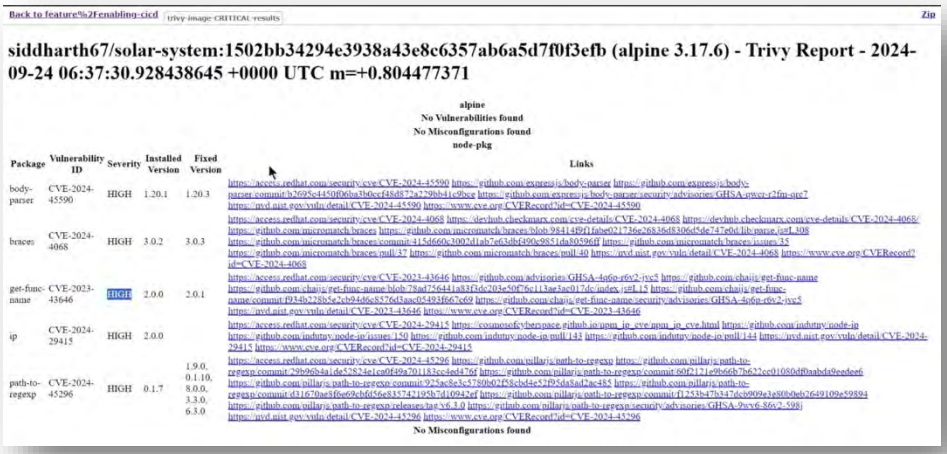
بعد ما تتولد التقارير، تقدر تستخدم:

- publishHTML لعرض تقارير HTML
- junit لعرض تقارير XML كاختبارات

```
post {
    always {
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'dependency-check-junit.xml'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'trivy-image-CRITICAL-results.xml'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'trivy-image-MEDIUM-results.xml'

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
        reportFiles: 'trivy-image-CRITICAL-results.html', reportName: 'Trivy Image Critical Vul Report',
        reportTitles: '', useWrapperFileDirectly: true])

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
        reportFiles: 'trivy-image-MEDIUM-results.html', reportName: 'Trivy Image Medium Vul Report',
        reportTitles: '', useWrapperFileDirectly: true])

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
        reportFiles: 'dependency-check-jenkins.html', reportName: 'Dependency Check HTML Report',
        reportTitles: '', useWrapperFileDirectly: true])
```

8. ✅ النتيجة النهائية

- لو فيه ثغرات High أو Critical ← الـ Pipeline هيفشل.
- لو فيه ثغرات Low أو Medium ← هيكمّل بس هيسجلها في التقرير.
- التقارير بتظهر داخل Jenkins Dashboard كـ HTML Report و Test Results



Back to feature%2Fenabling-cicd | trivy-image-CRITICAL-results | Zip

siddharth67/solar-system:1502bb34294e3938a43e8c6357ab6a5d7f0f3efb (alpine 3.17.6) - Trivy Report - 2024-09-24 06:37:30.928438645 +0000 UTC m=+0.804477371

alpine
No Vulnerabilities found
No Misconfigurations found
node-pkg

Package	Vulnerability ID	Severity	Installed Version	Fixed Version	Links
body-parser	CVE-2024-45590	HIGH	1.20.1	1.20.3	https://access.redhat.com/security/cve/CVE-2024-45590 https://github.com/expressjs/body-parser https://github.com/expressjs/body-parser/commit/b2695c4450f06ba3b0cef48d872a729bb4fc9bce https://github.com/expressjs/body-parser/security/advisories/GHSA-qwcr-r2fm-qrc7 https://nvd.nist.gov/vuln/detail/CVE-2024-45590 https://www.cve.org/CVERecord?id=CVE-2024-45590
braces	CVE-2024-4068	HIGH	3.0.2	3.0.3	https://access.redhat.com/security/cve/CVE-2024-4068 https://devhub.checkmarx.com/cve-details/CVE-2024-4068 https://devhub.checkmarx.com/cve-details/CVE-2024-4068/ https://github.com/micromatch/braces https://github.com/micromatch/braces/blob/98414ff9f1fabe021736e26836f8306d5de747e6d/lib/parse.js#L408 https://github.com/micromatch/braces/commit/415d660c300?d1ab7e63dbf490c9851da80596ff https://github.com/micromatch/braces/issues/35 https://github.com/micromatch/braces/pull/37 https://github.com/micromatch/braces/pull/40 https://nvd.nist.gov/vuln/detail/CVE-2024-4068 https://www.cve.org/CVERecord?id=CVE-2024-4068
get-func-name	CVE-2023-43646	HIGH	2.0.0	2.0.1	https://access.redhat.com/security/cve/CVE-2023-43646 https://github.com/advisories/GHSA-4q6p-r6v2-jvc5 https://github.com/chaijs/get-func-name https://github.com/chaijs/get-func-name/blob/78ad756441a83f3dc203e50f76c113ae3ac017dc/index.js#L13 https://github.com/chaijs/get-func-name/commit/f934b228b5e2cb94d6c8376d3aac05493f667c69 https://github.com/chaijs/get-func-name/security/advisories/GHSA-4q6p-r6v2-jvc5 https://nvd.nist.gov/vuln/detail/CVE-2023-43646 https://www.cve.org/CVERecord?id=CVE-2023-43646
ip	CVE-2024-29415	HIGH	2.0.0		https://access.redhat.com/security/cve/CVE-2024-29415 https://cosmonetcyberspace.github.io/oguu_ip_cve/oguu_ip_cve.html https://github.com/indutny/node-ip https://github.com/indutny/node-ip/issues/150 https://github.com/indutny/node-ip/pull/143 https://github.com/indutny/node-ip/pull/144 https://nvd.nist.gov/vuln/detail/CVE-2024-29415 https://www.cve.org/CVERecord?id=CVE-2024-29415
path-to-regexp	CVE-2024-45296	HIGH	0.1.7	1.9.0, 0.1.10, 8.0.0, 3.3.0, 6.3.0	https://access.redhat.com/security/cve/CVE-2024-45296 https://github.com/pillarjs/path-to-regexp https://github.com/pillarjs/path-to-regexp/commit/29b96b4a1de52824e1ca0f49a701183cc4ed476f https://github.com/pillarjs/path-to-regexp/commit/60f2121e9b66b7b622ce01080df0nabda9eedee6 https://github.com/pillarjs/path-to-regexp/commit/925ac8e3c5780b02f58cbd4e52F95da8ad2ac485 https://github.com/pillarjs/path-to-regexp/commit/d31670ae8f6e69cbfd56e835742195b7d10942ef https://github.com/pillarjs/path-to-regexp/commit/f1253b47b347dcb909e3e80b0eb26491096e59894 https://github.com/pillarjs/path-to-regexp/releases/tag/v6.3.0 https://github.com/pillarjs/path-to-regexp/security/advisories/GHSA-9wv6-86v2-598j https://nvd.nist.gov/vuln/detail/CVE-2024-45296 https://www.cve.org/CVERecord?id=CVE-2024-45296

No Misconfigurations found

- ❑ **تذكر** لو حبيت تظبط شكل الـ o/p اللي في الصورة السابقة دي فلازم تضيف الـ Jenkins content security اللي هو :

Unset the header:

```
System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "")
```

- ➢ ثم تضيف الكلام دا بقا في :

manage Jenkins → nodes → Build-in Node → Script Console → Add the line & Run

فهايبقي الشكل كدا :

Back to feature%2Fenabling-cicd | trivy-image-MEDIUM-results | Zip

siddharth67/solar-system:5f46fec8992ba6586668b5f218f44f1dd8ca1127 (alpine 3.17.6) - Trivy Report - 2024-09-24 06:43:55.605249151 +0000 UTC m=+0.850585867

alpine

Package	Vulnerability ID	Severity	Installed Version	Fixed Version	Links
busybox	CVE-2023-42363	MEDIUM	1.35.0-r29	1.35.0-r31	http://lists.busybox.net/pipermail/busybox/2024-May/090760.html https://access.redhat.com/security/cve/CVE-2023-42363 https://bugs.busybox.net/show_bug.cgi?id=15865 Toggle more links
busybox	CVE-2023-42364	MEDIUM	1.35.0-r29	1.35.0-r31	http://lists.busybox.net/pipermail/busybox/2024-May/090762.html https://access.redhat.com/security/cve/CVE-2023-42364 https://bugs.busybox.net/show_bug.cgi?id=15868 Toggle more links
busybox	CVE-2023-42365	MEDIUM	1.35.0-r29	1.35.0-r31	http://lists.busybox.net/pipermail/busybox/2024-May/090762.html https://access.redhat.com/security/cve/CVE-2023-42365 https://bugs.busybox.net/show_bug.cgi?id=15871 Toggle more links
busybox	CVE-2023-42366	MEDIUM	1.35.0-r29	1.35.0-r30	https://access.redhat.com/security/cve/CVE-2023-42366 https://bugs.busybox.net/show_bug.cgi?id=15874 https://nvd.nist.gov/vuln/detail/CVE-2023-42366 Toggle more links
busybox-binsh	CVE-2023-42363	MEDIUM	1.35.0-r29	1.35.0-r31	http://lists.busybox.net/pipermail/busybox/2024-May/090760.html https://access.redhat.com/security/cve/CVE-2023-42363 https://bugs.busybox.net/show_bug.cgi?id=15865

Push To Registry

🧱 أولاً: الهدف من المرحلة

- ❑ بعد ما تبني صورة الـ Docker وتعمل لها فحص أمان (باستخدام Trivy مثلًا) .. لازم تحفظها في Container Registry علشان تكون متاحة لأي عملية Deployment بعد كده.
- ❑ في المثال دا، بنستخدم Docker Hub كـ registry
- ❑ دي اخر مرحلة في الـ CI Pipeline وبعدها هانخش عال Deployment

---

⚙️ ثانياً : تعديل الـ Jenkinsfile

✳️ المرحلة الجديدة

بيضيف Stage جديدة في Jenkinsfile اسمها : stage('Push Docker Image')

```
stage('Push Docker Image') {
    steps {
        sh  'docker push siddharth67/solar-system:$GIT_COMMIT'
    }
}
```

لكن دا مش كفاية، لأن قبل ما تعمل docker push لازم تكون مسجل دخول (login) في Docker Hub

---

🔐 ثالثاً: التعامل مع الـ Credentials في Jenkins

1. 📦 تثبيت Plugin
   - لازم تثبت plugin اسمه : Docker Pipeline Plugin
   - وده بيسمحلك تستخدم أوامر Docker مباشرة داخل الـ Jenkins pipeline ، وكمان تضيف credentials بشكل آمن.

---

2. 🪪 إضافة بيانات الدخول

➢ تروح إلى:

- Jenkins Dashboard → Manage Jenkins → Credentials → Global
- وتضيف Username with Password

وتكتب مثلاً:

- Username: siddharth67
- Password: <DockerHubPassword>
- ID: dockerhub-credentials

🧰 رابعاً: استخدام الـ Plugin في الكود (WithDockerRegistry: Sets up Docker Registry Endpoint)

➢ هاتروح علي: manage Jenkins → pipeline syntax → snippet Generator → Sample Step: WithDockerRegistry: Sets...

- Docker registry URL: ""
- Registry credentials: Dockerhub-credentials (اللي معمولة في الخطوة رقم 2)
- وبعدين اديها generate .. هايديلك :

```
Generate Pipeline Script

// This step should not normally be used in your script. Consult the inline help for details.
withDockerRegistry(credentialsId: 'docker-hub-credentials', url: "") {
    // some block
}
```

➢ بعد كده، في الـ Jenkinsfile :

```
stage('Push Docker Image') {
    steps {
        withDockerRegistry(credentialsId: 'docker-hub-credentials', url: "") {
        sh  'docker push siddharth67/solar-system:$GIT_COMMIT'
    }
}
```

📘 المعنى:

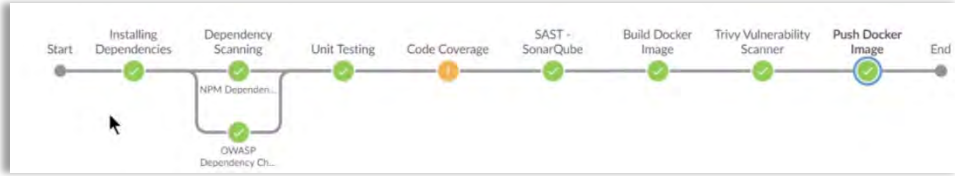
- withDockerRegistry ← بيعمل login تلقائي بالبيانات اللي حطيناها.
- credentialsId ← بيشير لـ ID اللي سجلناه في Jenkins
- url ← عنوان Docker Hub الافتراضي.
- الأمر داخل البلوك بيتنفّذ بعد تسجيل الدخول (push image)

📜 خامساً: تنفيذ الـ Pipeline

بعد ما تحفظ التعديلات وتعمل commit & push :

- Jenkins بيبدأ Pipeline جديد تلقائيًا.
- Stage "Push Docker Image" هتبدأ بعد نجاح المراحل السابقة.
- لو الـ credentials مظبوطة ← يعمل login ← يعمل push بنجاح.

📊 سادساً: التأكد من الـ Push



بعد ما الـ stage تنجح:

- تروح إلى Docker Hub account
- هتلاقي الـ repository solar-system
- آخر tag تم رفعه بيكون اسمه نفس Git Commit ID (مثلاً cf1715a)

وده بيأكد إن الصورة اللي اتبنت في الـ pipeline هي نفسها اللي اتعملها push

---

✅ سابعاً: ملخص الـ CI Pipeline اللي عملناه بالكامل:

المرحلة	الأداة / الهدف
Dependency installation	npm install
Dependency scanning	npm audit
Unit testing & Code coverage	Jest
Static code analysis	SonarQube
Build Docker image	Dockerfile
Vulnerability scanning	Trivy
Push image to registry	Docker Hub

النتيجة النهائية 🎯 →
صورة نظيفة وآمنة تم رفعها تلقائيًا إلى Docker Hub جاهزة للـ deployment.

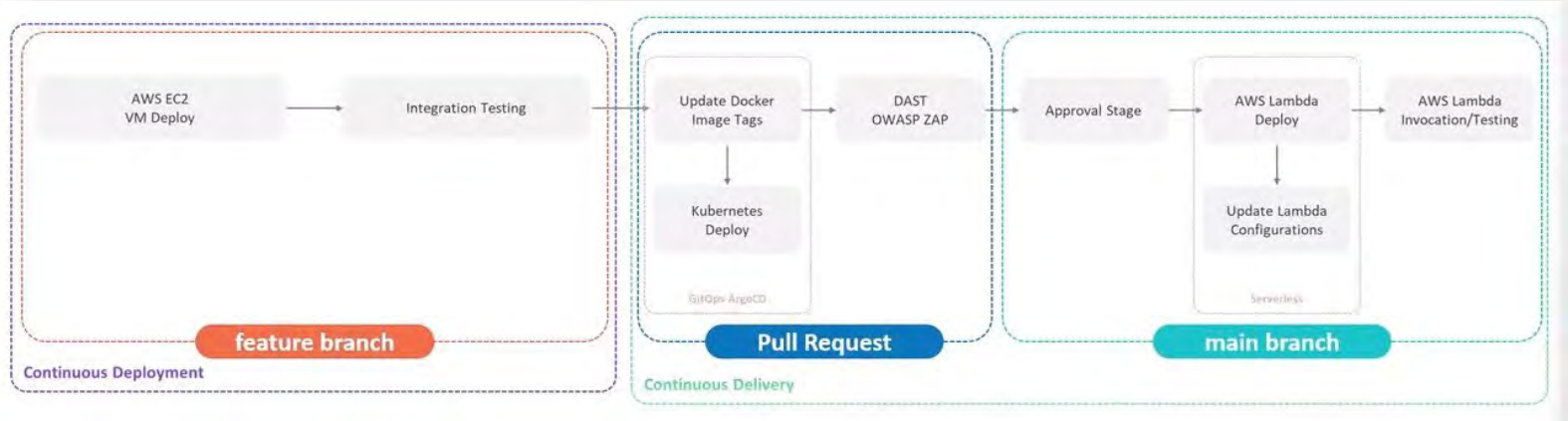
🎯 النتيجة النهائية : صورة نظيفة وآمنة تم رفعها تلقائيًا إلى Docker Hub جاهزة للـ deployment

Understanding Deployment Approach

الفكرة العامة:

- ❑ دلوقتي بعد ما خلصنا مراحل الـ CI
(build, testing, static & security analysis, build docker image, push image to Docker Hub زي)
- ❑ هنبدأ مرحلة الـ Deployment (يعني تشغيل التطبيق فعليًا على بيئة حقيقية)

تقسيم مراحل الـ Deployment :



❖ فيه 3 مراحل أساسية على حسب الفرع (branch) اللي بيشتغل عليه الـ developer :

1 Feature Branch Deployment (لما نعمل push على feature branch) :

- أول ما المطور يعمل push على الـ feature branch ، الـ pipeline هيبدأ يشتغل ويكمل مراحل الـ CI اللي عملناها قبل كده.
- بعد كده هيبدأ مرحلة deployment على AWS EC2 instance
- الهدف من المرحلة دي هو عمل integration testing على الـ EC2 (يعني اختبار التطبيق في بيئة شبه حقيقية)

➢ الهدف هنا : نتأكد إن الكود اللي في feature branch شغال فعلاً كويس قبل ما نرفعه للـ main

---

2 Pull Request Deployment

- لما المطور يعمل Pull Request (PR) علشان يدمج (merge) الكود في الـ main branch ،
ففي المرحلة دي الـ pipeline هيبدأ deploy التطبيق لكن المرة دي على Kubernetes cluster
- عملية الـ deployment هتتم باستخدام ArgoCD
- بعد ما التطبيق يتنشر على الـ K8s ، هيبدأ pipeline يعمل Dynamic Application Security Testing (DAST)
باستخدام OWASP ZAP ← علشان يختبر الثغرات الأمنية اللي بتحصل وقت التشغيل.

➢ المرحلة دي خاصة باختبار الـ security على بيئة Kubernetes قبل ما الكود يدخل الـ main

---

3 Main Branch Deployment

- لما يتم دمج الـ Pull Request في الـ main branch ، الـ pipeline هيتنفذ من تاني لكن الهدف دلوقتي مختلف:
  - o هيبدأ deploy التطبيق على AWS Lambda
  - o هيحدث إعدادات الـ Lambda function (configurations)
  - o وهيعمل invoke (تشغيل) للـ Lambda function علشان نعمل اختبار أخير بعد النشر.

➢ المرحلة دي تعتبر Production Deployment أو النشر النهائي.

🔁 الخلاصة:

المرحلة	الحدث	نوع الـ Deployment	مكان التشغيل	الهدف
🧪 Feature Branch	Push	Deploy على EC2	AWS EC2	Integration Testing
🧩 Pull Request	PR Created / Updated	Deploy على Kubernetes	K8s + ArgoCD	DAST Testing
🚀 Main Branch	Merge to main	Deploy على Lambda	AWS Lambda	Final Deployment

Exploring AWS and Setting up Jenkins Instance

- الجزء ده هو أول خطوة عملية في مرحلة الـ Deployment داخل الـ CI/CD pipeline
- وده اللي بنبدأ فيه نربط Jenkins مع AWS services (EC2 – S3 – Lambda) علشان نقدر ننفذ (Deployment) فعليًا من جوه Jenkins

---

🎯 المقدمة

- هنا بيمهد لمرحلة الـ deployment automation .. بمعنى إنه:
  - بيجهز Jenkins علشان يعرف يتعامل مع AWS
  - بيضبط الـ credentials (مفاتيح الدخول)
  - وبيثبت الـ plugins اللي هنحتاجها علشان Jenkins يقدر يتصل بـ EC2 و S3 و Lambda
  - كمان بيضيف الـ private key بتاع الـ EC2 علشان Jenkins يعرف يعمل SSH ويدخل السيرفر.
- كل ده مجرد تجهيز قبل ما نبدأ نشر التطبيق فعلاً على EC2

🧱 الشرح التفصيلي خطوة بخطوة:

🟢 أولًا : نظرة عامة على خدمات AWS المستخدمة

هانستخدم:

- EC2 ← علشان ينشر عليه Docker Image
- S3 ← علشان يخزن فيه artifacts (زي ملفات build أو صور Docker أو تقارير)
- Lambda ← علشان في مرحلة لاحقة نعمل deployment serverless
- دي 3 خدمات رئيسية في الـ AWS هنتعامل معاها من Jenkins

🟢 ثانيًا : عرض الإعدادات الحالية في AWS

- فتح الـ AWS console وبيوضح الآتي:
  - فيه EC2 instance باسم dev-deploy، شغالة حاليًا (Running state)
  - دخل عليها وعمل أمر : sudo docker ps
    - علشان يشوف الـ containers اللي شغالة على السيرفر.
  - فلقى إن فيه image قديمة (مثلاً solar-system) معمولة من 3 أيام.
- الهدف هنا إنه يتأكد إن السيرفر جاهز، فيه Docker شغال، وهنستخدمه لنشر الصور الجديدة بدل القديمة.

🟢 ثالثًا : إعداد IAM User لـ Jenkins

- دخل على IAM Dashboard في AWS ، وأنشأ User باسم Jenkins
- المستخدم ده عليه 3 سياسات (Policies) :
  1. AmazonEC2FullAccess
  2. AmazonS3FullAccess
  3. AWSLambdaFullAccess

✅ الهدف : السماح لـ Jenkins بالتحكم في الـ EC2 – S3 – Lambda

⚠️ ملاحظة مهمة : لاااااحظ إن دي Full Access policies (يعني صلاحيات كاملة) .. وده طبعا مش Best Practice في الحياة الواقعية، لكن هنا معمولة علشان الـ demo فقط.

🔒 أفضل ممارسة (Best Practice) :

تستخدم — Least Privilege Principle .. يعني تدي Jenkins صلاحيات محددة فقط للعمليات اللي محتاج يعملها، مش كل حاجة.

🟢 رابعًا: إعداد الـ Credentials (Access Keys)

- كل IAM User بياخد:
  - Access Key ID
  - Secret Access Key
- دي المفاتيح اللي Jenkins هيستخدمها علشان يتصل بـ AWS .. وهاتيخزنوا جوه Jenkins بطريقة آمنة.

🟢 خامسًا: تثبيت Plugin لربط Jenkins بـ AWS

- دخل على Jenkins → Manage Jenkins → Plugins
  ودور على plugin اسمه : AWS Steps Plugin (pipeline: AWS steps)

💡 ده Plugin بيسمحلك تستخدم أوامر AWS جوه Jenkinsfile
زي:

- رفع ملفات لـ S3
- النشر على ECR
- التعامل مع ELB أو SNS
- وحتى تشغيل Lambda functions

بعد التثبيت، عمل restart لـ Jenkins

🟢 سادسًا: إضافة الـ AWS Credentials داخل Jenkins

- بعد تثبيت الـ plugin ،
  راح على : Manage Jenkins → Credentials → Global → Add Credentials
- ومن الـ "Kind" اختار: AWS Credentials

وكتب:

- ID: aws-s3-ec2-lambda-creds
- Description: AWS credentials
- Access Key & Secret Key : القيم اللي نسخها من IAM User
- بعد الحفظ، Jenkins بيظهر رسالة تأكيد:

"These credentials are valid and have access to 6 availability zones."

يعني الاتصال بـ AWS تم بنجاح ✅

🟢 سابعًا: إعداد الاتصال بالـ EC2 Instance (SSH)

- علشان Jenkins يقدر يدخل على السيرفر الـ EC2 وينفذ أوامر deployment ، لازم يكون معاه Private Key :
  1. فبيروح لـ Manage Jenkins → Plugins
  2. وبيدور على plugin اسمه : SSH Agent Plugin
- وده اللي بيسمح لـ Jenkins يدخل على السيرفرات عن طريق SSH أثناء الـ pipeline
- بعد التثبيت، راح على : Manage Jenkins → Credentials → Add Credential
  - Kind : SSH Username with private key
  - ID: aws-devops-deploy-ec2-instance
  - Description: SSH username and private key for EC2
  - Username: ubuntu
  - Private Key : لصق المفتاح الخاص (.pem file)
  - Password : فارغ

كده Jenkins بقى يقدر يدخل السيرفر من غير ما يطلب باسورد 💪

🧩 ملخص سريع:

العنصر	الهدف	الأداة أو الخدمة
IAM User	تعريف Jenkins كمستخدم في AWS	AWS IAM
Access Key / Secret Key	مفاتيح اتصال Jenkins بـ AWS	Jenkins Credentials
AWS Steps Plugin	استخدام أوامر AWS من داخل Jenkinsfile	Jenkins Plugin
SSH Agent Plugin	تمكين Jenkins من الدخول على EC2 بـ SSH	Jenkins Plugin
SSH Key Credential	حفظ مفتاح الاتصال بالسيرفر	Jenkins Credentials

---

💡 Best Practices & Real Tips

- استخدم IAM Roles بدل الـ Access Keys لو Jenkins على EC2
- فعّل MFA على AWS Account
- خزن الـ credentials في Jenkins Credentials Store مش في Jenkinsfile
- استخدم Least Privilege Policy
- استخدم AWS CLI داخل Jenkins stages لتجربة الأوامر محليًا قبل إضافتها في pipeline

---

🔚 الخلاصة النهائية:

- ❑ الشرح ده كله تجهيز لـ المرحلة الجاية:
  اللي هنبدأ فيها فعلاً نربط Jenkins بالـ EC2 وننشر الـ Docker image هناك.
- ❑ يعني المرة دي عملنا الـ setup النظري والعملي:
  - AWS جاهز ✅
  - Jenkins جاهز ✅
  - Credentials & Plugins جاهزة ✅
- ❑ الخطوة الجاية بقى : نكتب Stage في الـ pipeline لعمل Deploy فعلي على EC2

Build-in When Condition

🧩 when directive في Jenkins Pipeline

- ❑ ده من أهم المفاهيم اللي لازم تكون فاهمها كويس في مرحلة الـ CI/CD automation

---

🎯 الهدف الأساسي من الـ when directive

- ❑ في أي Jenkins pipeline ، ساعات بتحتاج إن Stages معينة تشتغل بناءً على شرط معين او مش دايمًا.
- ❑ هنا بييجي دور الـ when
- ❑ الـ when directive بتتحكم إمتى الـ stage يشتغل، وإمتى يتعمله skip (يتعدّى عليه)

🧱 الاستخدام الأساسي:

داخل كل stage في Jenkinsfile ، تقدر تكتب:

```
stage('Deploy to AWS') {
  when {
    branch 'main'
  }
  steps {
    sh 'deploy.sh'
  }
}
```

🟢 معناها:

- الـ stage ده هيشتغل فقط لو الـ branch الحالي اسمه main
- لو أي branch تاني ← Jenkins هيـ skip الـ stage ده.

💡 الفكرة العامة:

- Jenkins بيقرأ كل Stage
- لو الـ stage فيه when { ... } ← بيشوف الشرط.
- لو الشرط اتحقق ✅ ← بيكمل تنفيذ الـ stage
- لو الشرط فشل ❌ ← بيعدي عليه (skip) ويروح للي بعده.

⚙️ أنواع الشروط (Conditions) اللي ممكن تستخدمها:

1️⃣ Branch condition

يشغّل الـ stage بس في فرع معين:

- يعني مثلاً تعمل build في جميع الفروع،
- لكن تعمل deploy فقط لو على main

```
when {
  branch 'main'
}
```

لاااحظ اننا بنجيب الـ patterns عموما من :
pipeline syntax → Declarative Directive Generator → Sample Directive → when:when condition → then choose your suitable pattern and generate it.

2️⃣ Environment Variable check

يشغّل الـ stage لو متغير بيئة (environment variable) ليه قيمة معينة:

```
when {
  environment name: 'DEPLOY_ENV', value: 'preprod'
}
```

✅ Example :

- لو DEPLOY_ENV=preprod ← stage هيشتغل
- غير كده ← هيتعمله skip

3 Expression (Custom Logic)

تقدر تكتب شرط مخصص بلغة Groovy :

```
when {
  expression {
    return params.MANUAL_TEST
  }
}
```

- ممكن تربط Jenkins بمتغيرات parameters ،
- أو تعمل شرط منطقي زي "لو المتغير الفلاني = true "

---

4 Change Request (Pull Request)

➢ يشغّل الـ stage بس لو الـ pipeline متشغل بسبب Pull Request :

```
when {
  changeRequest()
}
```

- كمان تقدر تحدد تفاصيل أكتر زي:

```
when {
  changeRequest author: "dev@example.com"
}
```

o يعني يشغل الـ stage فقط لو الـ PR جاي من شخص معين.

---

5 Tag check

➢ يشغل الـ stage لو الـ pipeline متشغل من tag بدل branch :

```
when {
  buildingTag()
}
```

- أو تطابق اسم الـ tag:

```
when {
  tag pattern: "release-*", comparator: "REGEXP"
}
```

---

6 شروط مركّبة (Combined Conditions) :

تقدر تستخدم منطق شرطي متقدم زي:

❑ anyOf

➢ يشغل الـ stage لو أي شرط من اللي جوه اتحقق :

```
when {
  anyOf {
    branch 'UAT'
    branch 'QA'
  }
}
```

- هيشتغل لو الفرع UAT أو QA (at least)
- لكن مش main

❑ allOf

➢ يشغل الـ stage لو كل الشروط صح:

- لازم الاتنين يتحققوا علشان الـ stage يشتغل.

```
when {
  allOf {
    branch 'main'
    environment name: 'DEPLOY_ENV', value: 'preprod'
  }
}
```

❑ **not** 🔸

➢ عكس الشرط :

- هيشتغل لو مش على الفرع main

```
when {
  not {
    branch 'main'
  }
}
```

❑ **beforeAgent** 🔸

➢ دي مهمة جدًا للأداء 💪

- معناها إن Jenkins هيفحص الشرط قبل ما يحجز agent (يعني قبل ما يعمل checkout أو يستهلك موارد)
- وده بيوفّر وقت ومصادر لو الشرط مش متحقق .

```
when {
  beforeAgent true
  branch 'main'
}
```

🧠 أمثلة عملية من الحياة الواقعية:

❑ 🔹 مثال 1:

➢ تشغيل اختبار أمني فقط في Pull Requests :

```
stage('Security Scan') {
  when {
    changeRequest()
  }
  steps {
    sh './run-zap-scan.sh'
  }
}
```

❑ 🔹 مثال 2:

➢ تشغيل نشر (Deploy) فقط على الفرع الرئيسي:

```
stage('Deploy to Production') {
  when {
    branch 'main'
  }
  steps {
    sh './deploy-prod.sh'
  }
}
```

❑ 🔹 مثال 3:

➢ نشر على بيئة معينة فقط:

```
stage('Deploy to QA') {
  when {
    environment name: 'DEPLOY_ENV', value: 'qa'
  }
  steps {
    sh './deploy-qa.sh'
  }
}
```

🧩 الخلاصة:

نوع الشرط	مثال	الاستخدام
`branch`	'branch 'main	تشغيل بناءً على الفرع
`environment`	'environment name:'DEPLOY_ENV', value:'preprod	تشغيل بناءً على متغير بيئة
`expression`	expression { params.RUN_TESTS }	شرط مخصص
`changeRequest`	()changeRequest	لما يكون في PR
`tag`	"*.tag pattern: "v	تشغيل في tags فقط
`anyOf`	anyOf { branch 'QA'; branch 'UAT' }	تشغيل لو أي شرط اتحقق
`allOf`	allOf { branch 'main'; env 'prod' }	تشغيل لو كل الشروط صح
`not`	not { branch 'main' }	عكس الشرط
`beforeAgent`	beforeAgent true	فحص الشرط قبل حجز agent

Deploy AWS EC2

🎯 الفكرة العامة

- ❑ الجزء دا بيشرح إزاي نعمل Deploy لتطبيق على AWS EC2 Instance باستخدام Jenkins Pipeline
- ❑ يعني بعد ما بنينا الـ Docker image ورفعناها على Docker Hub ، دلوقتي الدور إننا ننشرها على السيرفر (EC2)

⚙️ الخطوات الأساسية

1. 🧩 إضافة مرحلة جديدة في الـ Pipeline

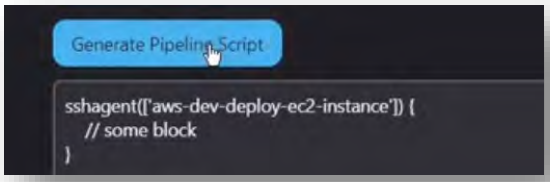
- ➢ بدأنا بإضافة stage جديدة اسمها "Deploy AWS EC2"
- ➢ المرحلة دي هتتنفذ بعد مرحلة "Push Docker Image"

```
stage('Deploy AWS EC2') {
    steps {
        ...
    }
}
```

---

2. 🔐 إعداد الاتصال بالسيرفر (EC2) عبر SSH

- ➢ عشان Jenkins يقدر يدخل على السيرفر، لازم نستخدم **SSH Agent Plugin**
- ➢ البلجن دي بتخلي Jenkins يستخدم الـ private key اللي خزنّاه قبل كدا كـ credentials
- هو راح على "Snippet Generator" في Jenkins وكتب كلمة ssh
- اختار الـ SSH Agent Step
- حدد الـ credentials اللي فيها:
  - o Username: ubuntu
  - o Private key : المفتاح الخاص بالـ EC2
- بعدين راح عالـ jenkinsfile وأضاف الـ steps دي :



🚀 وبكدا Jenkins بقى يقدر يتصل بالسيرفر.

```
stage('Deploy - AWS EC2') {
    when {
        branch 'feature/*'
    }
    steps {
        script {
            sshagent(['aws-dev-deploy-ec2-instance']) {
                sh '''
                    ssh -o StrictHostKeyChecking=no ubuntu@3.140.244.188 "
                        if sudo docker ps -a | grep -q "solar-system"; then
                            echo "Container found. Stopping..."
                                sudo docker stop "solar-system" && sudo docker rm "solar-system"
                            echo "Container stopped and removed."
                        fi
                        sudo docker run --name solar-system \
                            -e MONGO_URI=$MONGO_URI \
                            -e MONGO_USERNAME=$MONGO_USERNAME \
                            -e MONGO_PASSWORD=$MONGO_PASSWORD \
                            -p 3000:3000 -d siddharth67/solar-system:$GIT_COMMIT
                    "
                '''
            }
        }
    }
}
}
```

Explain in point 6

Explain in point 5

Explain in point 2

Explain in point 3&4

Explain in point 4

---

3. 🧠 تنفيذ وشرح أوامر SSH داخل السيرفر (داخل jenkinsfile)

❖ جوه الـ SSH block (اللي في jenkinsfile) ، بيستخدم shell command عادي لتشغيل أوامر داخل السيرفر.

➢ أول حاجة كتبها: ssh -o StrictHostKeyChecking=no ubuntu@<EC2-IP>

- الخيار StrictHostKeyChecking=no عشان يتجنب أي pop-up يسأل "هل تثق في السيرفر دا؟".
- الـ IP في المثال ثابت (hardcoded) ، لكنه وضّح إنه ممكن نخليه dynamic بعدين.

---

4. 🧱 منطق تنفيذ الـ Docker Container (داخل jenkinsfile)

➢ بمجرد ما يتصل بالسيرفر، بيبدأ ينفذ أوامر Docker :

🧩 الخطوات جوه السيرفر:

1. يتأكد إذا فيه container شغال بنفس الاسم: docker ps -a | grep -q "solar-system"
   - لو موجود ← يوقفه ويشيله: docker stop solar-system && docker rm solar-system
2. بعد كدا يشغل Docker container جديد:

```
sudo docker run -d --name solar-system    -e MONGO_URI=$MONGO_URI \
  -e MONGO_USER=$MONGO_USER    -e MONGO_PASS=$MONGO_PASS \
  -p 3000:3000 your_dockerhub_username/solar-system:$GIT_COMMIT
```

   - بينقل الـ environment variables (زي DB credentials)
   - بيفتح الـ port 3000
   - بياخد الـ image من Docker Hub

---

5. 💡 ملاحظة مهمة عن Groovy Syntax

➢ في النص، استخدم if condition داخل الـ pipeline

➢ لكن ده ماينفعش في Declarative Pipeline لأنه بيفهم بس structure ثابت.

➢ علشان كدا استخدم حاجة اسمها: script { ... }

- الـ script block بيسمحلك تكتب كود Groovy عادي (Scripted pipeline style) جوا declarative pipeline

---

6. 🧩 تقييد المرحلة بشرط (when)

```
when {
   branch pattern: "feature.*", comparator: "REGEXP"
}
```

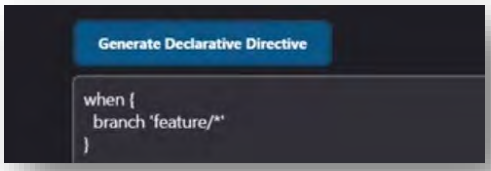
➢ أضاف كمان شرط when عشان المرحلة دي تتنفذ بس لو الفرع اسمه /feature

➢ لاحظ انه جاب الـ pattern من pipeline syntax → Declarative Directive Generator → Sample Directive → when:when condition

➢ بعدين اختار :

- When condition: allOf:Execute the stage if all nested conditions are true
- Branch Pattern: feature/*

➢ وبعدين ضغط generate فطلعله الـ pattern :



- يعني Jenkins هيشغل مرحلة الـ deploy دي فقط لما الفرع يبدأ بكلمة feature

---

7. اختبار وتشغيل الـ Pipeline

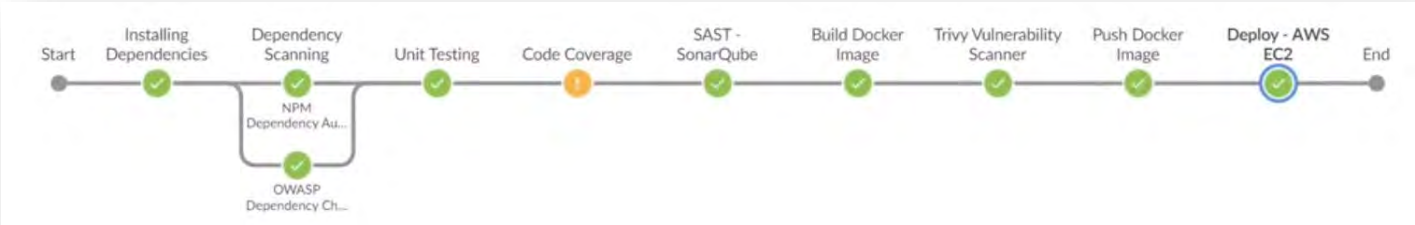
- بعد ما خلص الكود، عمل commit وشغل الـ pipeline
- طلع error بسيط في الـ syntax (نقص brace .. وبعد ما ظبطه، كله اشتغل تمام. ✅)

---

8. النتيجة

- الـ SSH اتنفذ بنجاح.
- Jenkins دخل على الـ EC2
- وقف الـ container القديم.
- شغّل الجديد.
- التطبيق بقى شغال على : http://<EC2-Public-IP>:3000

- شاف اللوج ولقى الـ container ID بيبدأ بـ CAB88، وفعلاً شافه بالسيرفر لما كتب: sudo docker ps
- الـ Container شغال ومكشوف على port 3000 .. ولما فتح الـ IP في المتصفح، التطبيق اشتغل تمام ✨



ملخص سريع

الخطوة	الهدف
إنشاء Stage جديدة	لنشر التطبيق على AWS
استخدام SSH Agent	للاتصال الآمن بالـ EC2
تنفيذ أوامر Docker	لإيقاف وتشغيل التطبيق الجديد
استخدام `script {}`	للسماح بكود Groovy داخل declarative pipeline
إضافة شرط `when`	لتشغيل المرحلة في فرع معين فقط
التحقق النهائي	الـ container اشتغل بنجاح على السيرفر

Integration Testing- AWS EC2

🎯 الفكرة العامة

- ❑ الـ developer عايز يعمل Integration Testing بعد ما الـ app اتعمله deploy على EC2
- ❑ بس بدل ما يكتب الـ IP أو الـ DNS بتاع الـ EC2 يدوي في الكود (hard-coded) ، هيخليه يتجاب أوتوماتيك من AWS CLI

---

1. 🧩 إنشاء سكريبت الاختبار integration-testing-ec2.sh

```bash
#!/bin/bash
echo "Intergation test........"

aws --version

Data=$(aws ec2 describe-instances)
echo "Data - "$Data
URL=$(aws ec2 describe-instances | jq -r ' .Reservations[].Instances[] | select(.Tags[].Value == "dev-deploy") |
echo "URL Data - "$URL

if [[ "$URL" != '' ]]; then
    http_code=$(curl -s -o /dev/null -w "%{http_code}" http://$URL:3000/live)
        echo "http_code - "$http_code
    planet_data=$(curl -s -XPOST http://$URL:3000/planet -H "Content-Type: application/json" -d '{"id": "3"}')
        echo "planet_data - "$planet_data
    planet_name=$(echo $planet_data | jq .name -r)
        echo "planet_name - "$planet_name

    if [[ "$http_code" -eq 200 && "$planet_name" -eq "Earth" ]];
        then
            echo "HTTP Status Code and Planet Name Tests Passed"
        else
            echo "One or more test(s) failed"
            exit 1;
    fi;

else
        echo "Could not fetch a token/URL; Check/Debug line 6"
        exit 1;
fi;
```

Explain in point 2&3

- اتعمل ملف جديد في الـ repository اسمه : integration-testing-ec2.sh
- ده سكريبت Bash بيعمل 3 حاجات رئيسية:
  1. يجيب الـ Public DNS name بتاع الـ EC2 instance اللي عليه التطبيق.
  2. يستخدم الـ cURL commands عشان يعمل requests للـ app
  3. يتحقق من إن الـ responses سليمة (status code = 200 و البيانات صح)

إجمد وأوعي تتخض .. كل الكلام والأوبشنز هتلاقيها في الـ Docs :

describe-instances — AWS CLI 2.31.25 Command Reference

---

2. ⚙️ شرح السكريبت + جلب الـ DNS name ديناميكيًا :

1. بيستخدم أمر AWS CLI : aws ec2 describe-instances
   - وده بيرجع JSON فيه كل تفاصيل الـ instances
2. عشان يفلتر منه الـ instance اللي هو عايزه، بيستخدم أداة : jq
   - jq '.Reservations[].Instances[] | select(.Tags[].Value=="dev-deploy") | .PublicDnsName'
   - وبيفلتر حسب Tag Name = "dev deploy" (يعني السيرفر بتاع الـ dev environment)
3. بعد ما نجيب الـ URL، نبدأ نعمل عليه شوية curl tests:
   - نتحقق إن السيرفر شغال على الـ port 3000 :
     - curl -s -o /dev/null -w "%{http_code}" http://<url>:3000/live
     - بنخزن الكود في متغير ونتأكد إنه 200.
   - نعمل POST request علشان نجيب بيانات كأننا بنتعامل مع API:
     - curl -X POST -H "Content-Type: application/json" -d '{"id":1}' http://<url>:3000/
     - نحفظ النتيجة ونشوف مثلاً إن القيمة اللي راجعة هي "planet":"Earth".
4. و كل حاجة تمام (HTTP code = 200 و planet name = Earth) ← الاختبار ناجح ✅
5. غير كده ← نخرج بكود خطأ exit 1 ❌

3. 🔁 اختبار الاتصال

- بيجرب يعمل curl على الـ URL على البورت 3000
- لو السيرفر مش شغال ← يطبع error message ويخرج.
- لو السيرفر شغال ← يبدأ يعمل:
  - GET request ← للتأكد إن الـ app بيرد.
  - POST request ← يبعث ID ويشوف الـ response بيرجع بيانات صح.
- بيخزن الرد في متغيرات زي planet_data و planet_name
- في الآخر، يتحقق:

```
if [[ "$http_code" -eq 200 && "$planet_name" -eq "Earth"  ]];
    then
        echo "HTTP Status Code and Planet Name Tests Passed"
    else
        echo "One or more test(s) failed"
        exit 1;
fi;
```

---

4. 🧱 إضافة Stage جديدة في Jenkinsfile علشان ينادي عالـ Shell script ده

- أضاف Stage اسمها : stage('Integration Testing AWS EC2')
- جوّاها بيشغّل السكريبت كده: sh 'bash integration-testing-ec2.sh'

```
stage('Integration Testing - AWS EC2') {
    when {
        branch 'feature/*'
    }
    steps {
        sh 'printenv | grep -i branch'
        withAWS(credentials: 'aws-s3-ec2-lambda-creds', region: 'us-east-2') {
            sh '''
                bash integration-testing-ec2.sh
            '''
        }
    }
}
```

Explain in point 5&6

📌 ملاحظات:

- الـ { branch 'feature' } when معناها إن الستيج دي هتشتغل بس على فرع feature
- withAWS بتضبط الـ region والمفاتيح مؤقتًا أثناء تشغيل الكود . (موضحة في نقطة 5)
- sh 'bash integration-testing-ec2.sh' ← بيشغل السكريبت اللي كتبناه.
- Sh printenv | grep -I branch ← بيطبعلي الـ env vars في الـ branch اللي انا عايزه فقط

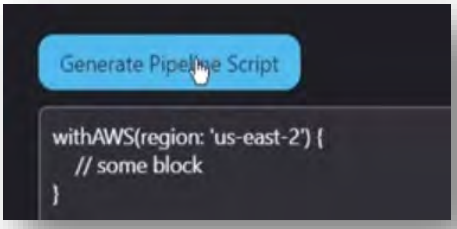
---

5. 🔐 إعداد الـ AWS Credentials داخل Jenkins

- AWS CLI محتاج credentials (Access Key + Secret Key + Region)
- وهو كان عاملهم قبل كده في Jenkins credentials بإسم aws-s3-ec2-lambda-creds
- بيستخدم plugin اسمه AWS Pipeline Steps هايجيبه من :

Manage Jenkins → pipeline syntax → snippet generator → sample step → withAWS: set AWS settings for nested block

ثم هايختار ← Region: based on your account region وبعدين generate



6. 🧩 الشرط (Condition)

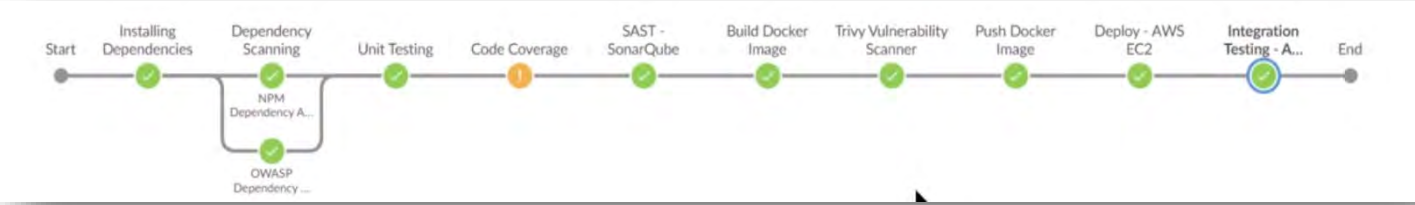
- المرحلة دي بتشتغل بس لو الفرع هو feature branch :

```
when {
branch 'feature'
}
```

---

7. 🧪 النتيجة النهائية

- Jenkins شغّل الـ pipeline
- في الـ logs :
  - طبع إن الفرع الحالي هو feature
  - طبع نتيجة أمر aws ec2 describe-instances
  - نفّذ الـ curl commands
  - الاختبارات نجحت. ✅
- النتيجة : "Integration Testing successfully completed."



💡 الخلاصة

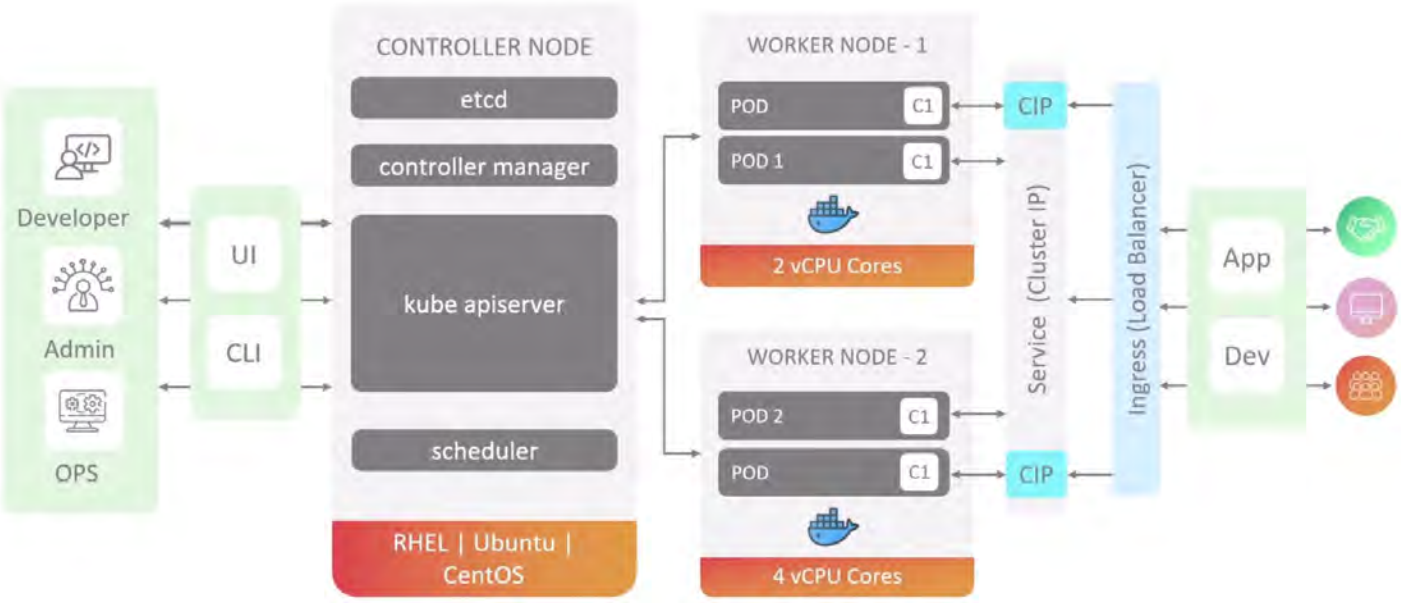
المرحلة	الوصف
Deploy	التطبيق اتنشر على EC2
Integration Testing	بيتأكد إن التطبيق شغال فعلاً
Dynamic Fetch	بيجيب الـ DNS name أوتوماتيك من AWS CLI
Validation	بيجرب `GET` و `POST` Requests ويتأكد من النتيجة
Automation	كله معمول داخل Jenkins stage بذكاء و بدون manual IPs

K8S Overview

Overview 🎯

- ❑ Kubernetes أو K8s هو نظام مفتوح المصدر بيُستخدم لإدارة الـ containers على نطاق واسع
- ❑ يعني لو عندك عشرات أو مئات الـ containers ، هو اللي بينظم نشرهم وتشغيلهم وتوزيعهم على السيرفرات (nodes)

Architecture Overview ⚙️



➢ Kubernetes بيتكوّن من نوعين أساسيين من الـ nodes :

1. Control Plane Node (Master)

هو اللي بيدير الكلستر، وبيتحكم في القرارات الكبيرة زي:

- فين الـ pods تشتغل
- إزاي يحصل scheduling
- مراقبة الحالة العامة للنظام

مكوناته:

- API Server :
  المدخل الأساسي لإدارة الـcluster ، كل أوامر الـ kubectl بتروح له.
- Controller Manager :
  بيتابع الحالة المطلوبة (desired state) ويعدل لو في حاجة مش مظبوطة.
- Scheduler :
  بيقرر الـ pod هيتشغل على أي Node بناءً على الموارد المتاحة.
- ETCD :
  قاعدة البيانات الخاصة بـ cluster و فيها كل المعلومات عن الحالة (state)

---

2. Worker Node

- هو المكان اللي فعليًا بيشغّل الـ containers
- بيستقبل الأوامر من الـ control plane ويشغل الـ pods عليها.

Pods 🧱

- الـ Pod هو أصغر وحدة تشغيل في Kubernetes
- ممكن يحتوي على Container واحد أو أكثر، وبيشاركوا نفس الـ IP والـ storage
- غالبًا بيبقى Container واحد في كل Pod (زي الـ app container)

📌 💡 لو الـ Pod حصل فيه crash أو failure ، بيتشال وما بيرجعش لوحده.

علشان كده بنستخدم Replication Controllers أو Deployments علشان يديروا عملية استرجاع الـ pods تلقائيًا.

Deployments 🚀

- Deployment = abstraction فوق الـ ReplicaSets
- هو اللي بيخليك تقول لـ Kubernetes :
  - " عاوز 3 نسخ من الـ app ده شغالة، وعاوز لو في تحديث يتم بدون downtime "
  - Kubernetes بيهتم يطابق الحالة الفعلية (actual state) مع الحالة اللي انت طلبتها (desired state)
    - يعني لو طلبت 3 Pods وشغّالين 2 بس، هو هيضيف الثالث أوتوماتيك.

Services 🌐

- بعد ما الـ Pods تشتغل، إزاي توصلها؟
- هنا ييجي دور الـ Service .. اللي بتديك عنوان ثابت للوصول للـ Pods (حتى لو الـ pods اتبدلت أو اتعمل لها restart)
- أنواع الـ services :
  - ClusterIP
    - للوصول الداخلي فقط داخل الكلستر.
  - NodePort
    - بيفتح port على كل Node علشان توصل من برّه.
  - LoadBalancer
    - بيستخدم Load Balancer خارجي (زي AWS ELB أو GCP LB) علشان يوزّع الترافيك على الـ Pods
    - بيكون مفيد في الـ Cloud لكن بيكلف أكتر.

Ingress 🌍

- بديل ذكي للـ Load Balancer لما يكون عندك أكتر من service لانه بيعمل route HTTP/s Traffic للـ Cluster
  - بيسمحلك تدخل كل الخدمات من Domain واحد (مثلاً app.example.com/api و app.example.com/web)
  - بيوجه الترافيك بناءً على path أو domain
  - بيشتغل غالبًا مع ClusterIP services (يعني internal) علشان يبقى فيه تحكم دقيق في مين اللي يتعرض للعالم الخارجي.

💡 ده بيقلل التكلفة وبيخليك تدير كل الـ routes من مكان واحد.

Summary ⚖️

الحالة	الأداة المستخدمة	الوظيفة
تشغيل الـ containers	Pods	الوحدة الأساسية للتشغيل
إدارة التكرار والتحديث	Deployment	تكرار أو تحديث التطبيق
توصيل الخدمة	Service	Endpoint ثابت للترافيك
توجيه الترافيك الذكي	Ingress	إدارة الـ HTTP routing
إدارة الكلستر	Control Plane	تنظيم وسكجولنج وتشغيل

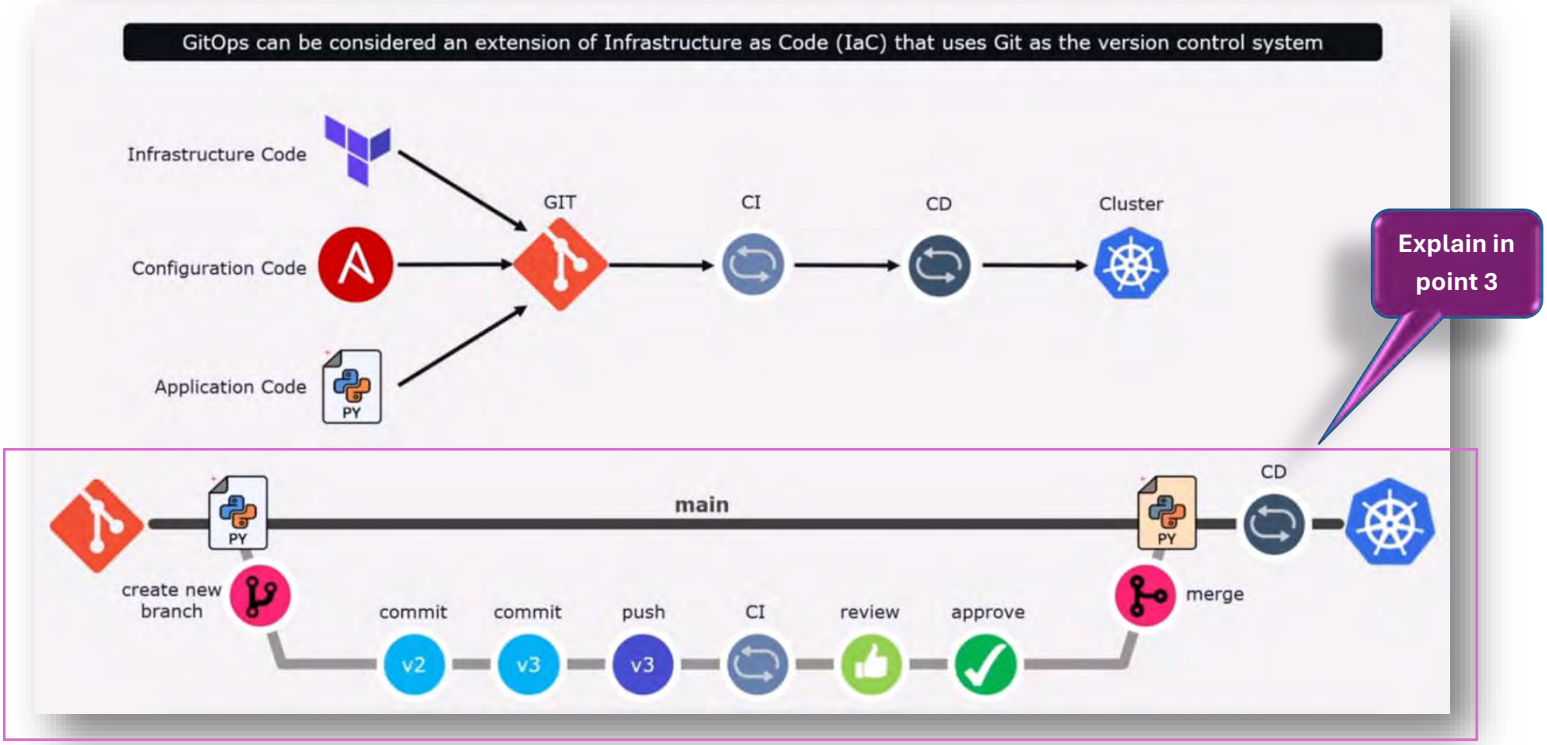
Introduction To GitOps

🎯 أولًا: المقدمة — الفكرة العامة

- ❑ GitOps ببساطة هي طريقة لإدارة الـ infrastructure والـ applications باستخدام Git كـ " (Single Source of Truth) "
- ❑ يعني أي تعديل في بيئة التشغيل أو التطبيق بيتم عن طريق تعديل في Git .. مش يدوي.
- ❑ الفكرة إننا نخلي الـ CI/CD Pipeline هو اللي ينفذ التغييرات دي تلقائيًا (deploy, rollback, etc.) بناءً على ما هو موجود في Git

بالتالي:

- Git = مصدر الحقيقة (Desired State)
- Environment = الحالة الفعلية (Actual State)
- Operator = اللي بيراقب ويضمن إن الاتنين متطابقين



🧩 ثانيًا: الشرح التفصيلي خطوة بخطوة

1 ◆ Git كـ مركز التحكم

يعني كل حاجة ← الكود، الـ infrastructure ، إعدادات الـ cluster ، ملفات الـ manifests ← بتتخزن في Git
وده بيدينا:

- Traceability (تاريخ التغييرات)
- Version control (ممكن نرجع لحالة سابقة)
- Consistency بين كل الـ environments

---

2. ◆ امتداد لمفهوم Infrastructure as Code

Infrastructure as Code (IaC) زي Terraform أو Ansible ، لكن GitOps بياخد الخطوة الأبعد:

- مش بس نكتب الـ infra كـ code
- كمان نربط التنفيذ والتحديث والـ rollback تلقائيًا بالـ Git repository

---

3. ◆ دورة العمل (Workflow)

1. Developer يكتب كود جديد في branch فرعي (feature branch)
2. يعمل commit + push
3. CI Service (زي Jenkins / GitHub Actions) يبدأ:
   - يبني الكود (build)
   - يشغّل اختبارات (unit tests)
   - يدمج (merge) بعد الموافقة (code review)
4. بعدها يبدأ الـ CD (Continuous Deployment) :
   - يبني Docker image
   - يرفعها إلى registry
   - يحدث ملفات الـ manifest في repo تاني

---

4. ◆ الـ Desired State vs Actual State

يعني الـ GitOps Operator (زي ArgoCD أو FluxCD)

- بيقرأ الـ manifests من الـ Git repo (desired state)
- ويقارنها بالحالة الفعلية داخل الـ cluster (actual state)
- ولو في فرق ← يطبق التحديث تلقائيًا

---

5. ◆ الـ GitOps Operator

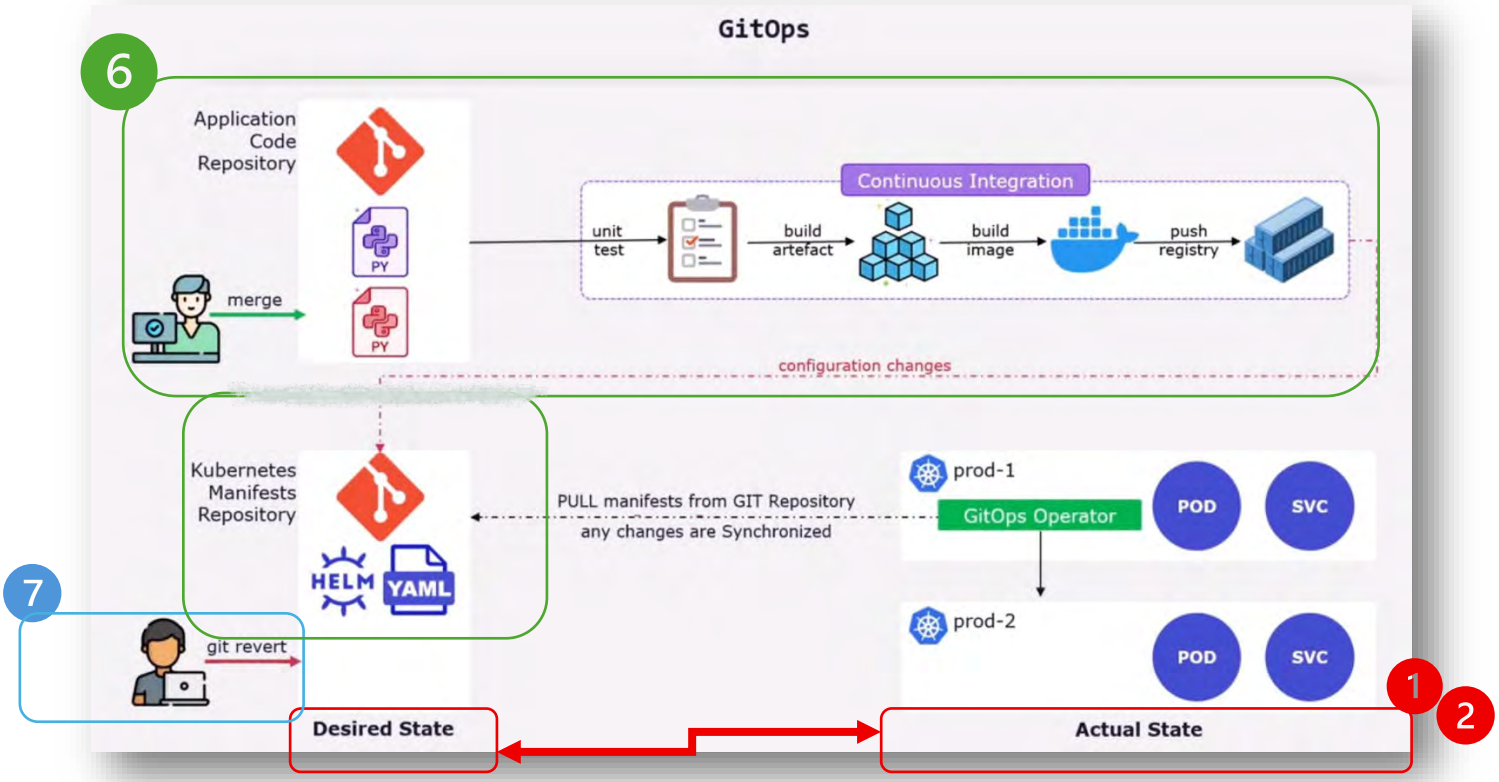
أشهر الأدوات:

- Argo CD
- FluxCD

الـ Operator بيعمل:

- مراقبة repo التحديثات
- تنفيذ sync للـ cluster
- مراقبة حالة الـ deployment

لو عندك أكتر من cluster ، ممكن يطبق نفس التحديث على الكل (multi-cluster sync)



6. ◆ التكامل مع CI/CD

الخطوات الكاملة:

1. Developer يدمج التغييرات في الـ app repo
2. Jenkins (أو أي CI tool) يبني التطبيق ← يعمل Docker image ← يرفعها للـ registry
3. Jenkins بعد كده يحدّث ملفات الـ manifests (في GitOps repo)
4. ArgoCD / FluxCD يشوف التغيير ← يطبقه على cluster

---

7. ◆ الـ Rollback السهل

ميزة قوية جدًا:

- لو deployment فيه مشكلة ← بس نعمل : git revert <commit-id>
- الـ Operator هيلقط التغيير ويرجع الـ cluster للحالة السابقة.

⚙️ ثالثًا: الأدوات والأوامر المستخدمة

الفئة	الأداة	الاستخدام
Version Control	Git	إدارة الكود والبنية التحتية
CI Tool	Jenkins / GitHub Actions / GitLab CI	بناء واختبار التطبيق
CD Tool	ArgoCD / FluxCD	تنفيذ النشر ومزامنة الحالة
Infra as Code	Terraform / Helm / Kustomize	تعريف الـ infrastructure والـ manifests
Containerization	Docker	بناء الـ app image
Container Registry	Docker Hub / ECR / GCR	تخزين الصور الجاهزة للنشر

💡 رابعًا: ملاحظات عملية (Best Practices + Real Use Cases)

✅ افصل الـ Repos بوضوح:

- Repo 1 → Application code
- Repo 2 → Deployment manifests

✅ Automation كاملة:

- Jenkins يبني ويحدث فقط الـ manifests
- ArgoCD هو اللي ينفذ النشر فعليًا (Pull model)

✅ استخدم Branch per Environment :

- main → production
- staging → staging cluster
- dev → development cluster

✅ Rollback سريع وآمن:

- كل حالة محفوظة في Git ← revert فوري بدون لمس cluster يدويًا.

✅ Real use case

شركات زي *Weaveworks* و *Intuit* بتستخدم ArgoCD عشان تنشر مئات التطبيقات على آلاف الـ clusters بدون تدخل بشري.

🧠 خامسًا: ملخّص

🔁 **GitOps Pipeline Flow:**

Developer -> Git Push -> Jenkins Build/Test -> Docker Image

↓

Update Manifest Repo

↓

GitOps Operator (ArgoCD/Flux)

↓

Cluster Syncs Desired State

↓

Automatic Deploy or Rollback

✨ **المحصلة:**

- **أي تعديل في Git ← نشر تلقائي.**
- **أي rollback في Git ← استرجاع تلقائي.**
- **بيئة دايمًا متزامنة، بدون تدخل يدوي.**

Jenkins vs. ArgoCD

- ArgoCD:

  هي **GitOps tool**، يعني شغلها الأساسي إنها تتابع الـ **Git repository** اللي فيه تعريفات الـ Kubernetes (YAML manifests, Helm charts, Kustomize, ...).

  - بتقارن بين الحالة الموجودة في الـ repo (Desired state) و الحالة الحالية في الـ cluster (Live state).
  - ولو لقت اختلاف، بتقوم بعملية Sync (تطبيق التغييرات تلقائيًا أو يدويًا حسب الإعدادات).
    - 🔹 بمعنى تاني: ArgoCD هي اللي **تمسك الكود وتطبّق التغييرات على الـ cluster فعليًا**.
  - هي ما تبنيش ولا تـ test ولا تعمل build steps — وظيفتها Deployment & Synchronization.

- Jenkins:

  هو **CI/CD tool عام**، وظيفته الأساسية **Automation** لأي عملية (build, test, deploy, etc).

  - Jenkins ممكن يعمل **build** لـ **code**، يعمل **push** لـ **image** إلى **Docker registry**، وبعد كده **يحدّث ملفات الـ deployment** داخل الـ **Git repo** (مثلاً يغيّر رقم الـ image tag).
  - لكن Jenkins نفسه ما بيطبّقش التغييرات على الـ cluster إلا لو كتبنا له سكريبت أو plugin مخصوص (زي `kubectl apply` داخل pipeline).
    - 🔹 ففي GitOps approach: Jenkins بيعدل في الـ repo فقط، و ArgoCD هو اللي بيشوف التعديل ويطبّقه على الـ cluster.

ArgoCD Concepts & Terminology

🧠 أولاً: الفكرة العامة

- ❑ قبل ما نغوص في تفاصيل ArgoCD ، لازم نفهم إن الأداة دي بتشتغل في إطار فكرة اسمها GitOps،
- ❑ يعني ببساطة كل حاجة في الـ infrastructure أو الـ deployments بتتدار من خلال Git repository
- ❑ ArgoCD هو الأداة اللي بتمسك الـ code اللي في الـ repo ده، وتشوف المفروض الـ cluster يكون شكله عامل إزاي، وبعدين تطبّق التغييرات دي تلقائيًا على Kubernetes

⚙️ ثانياً: المكونات الأساسية في ArgoCD

Application	A group of Kubernetes resources as defined by a manifest.
Application source type	The tool is used to build the application. E.g. Helm, Kustomize or Ksonnet
Project	Provide a logical grouping of applications, which is useful when Argo CD is used by multiple teams.
Target state	The desired state of an application, as represented by files in a Git repository.
Live state	The live state of that application. What pods, configmap, secrets, etc are created/deployed in a Kubernetes cluster.
Sync status	Whether or not the live state matches the target state. Is the deployed application the same as Git says it should be?
Sync	The process of making an application move to its target state. E.g. by applying changes to a Kubernetes cluster.
Sync operation status	Whether or not a sync succeeded.
Refresh	Compare the latest code in Git with the live state. Figure out what is different.
Health	The health of the application, is it running correctly? Can it serve requests?

🔹 Application

- دي تعتبر وحدة التشغيل الأساسية داخل ArgoCD
- كل Application بتمثل مجموعة الـ manifests أو الـ Helm chart اللي بتخص مشروع معين.
- لما بنثبت ArgoCD ، بيتضاف CRD (Custom Resource Definition) اسمه Application
- التطبيق ده بيحدد:
  - Source : مكان الكود (مثلاً GitHub repo)
  - Destination : الـ cluster اللي المفروض ArgoCD يطبق فيه التغييرات.

🧩 مثال عملي:

لو عندك repo فيه manifest لواجهة ويب (frontend) ، فهتعمل Application باسم frontend-app،
و ArgoCD هيبقى مسؤول عن التأكد إن الـ frontend في الـ cluster مطابق للي موجود في الـ Git

---

🔹 Application Source Type

- النوع ده بيحدد الأداة اللي بنبني بيها الـ application
- ممكن تكون:
  - Helm
  - Kustomize
  - أو ملفات YAML عادية.

💡 ملاحظة : في المشاريع الكبيرة، Helm بيكون المفضل لأنه بيسمح بمرونة أكبر في الـ templating

---

🔹 Project

- دي وسيلة لتنظيم التطبيقات جوه ArgoCD حيث أنها بتوفر logical grouping of applications
- تقدر تعتبرها namespace داخل ArgoCD نفسه.
- بتفيد لما يكون عندك فرق مختلفة (teams) بتستخدم نفس ArgoCD instance
- كل فريق ممكن يكون عنده project خاص بيه يضم الـ applications بتاعته بس.

🧩 ثالثاً: حالة التطبيق (States)

🔸 Target State (Desired State)

- ده الحالة المطلوبة للتطبيق، اللي متخزنة في الـ Git repo
- يعني الشكل اللي المفروض التطبيق يكون عليه حسب الكود اللي في Git

🔸 Live State (Actual State)

- دي الحالة الحالية الفعلية داخل الـ Kubernetes cluster
- هي اللي ArgoCD بيقارنها مع الـ target state

🧠 الفكرة الأساسية ArgoCD : دايمًا بيقارن بين الحالتين — لو فيه فرق، بيبدأ عملية اسمها Sync

---

🔄 رابعاً: الـ Sync

- عملية الـ Sync معناها إن ArgoCD بياخد الحالة اللي في Git (target state) ويطبقها في الـ cluster (live state)
- كده يضمن إن التطبيق في الـ cluster مطابق تمامًا للكود اللي في Git

💡 Best Practice :

فعل الـ *Auto-Sync* في البيئات التجريبية (staging) ، لكن خليه manual في الـ production لتفادي أخطاء غير مقصودة.

📋 خامساً Sync Status & Operation Status :

- Sync Status ← بتوضح هل الـ live state مطابق فعلاً للـ target state ولا لأ.
- Operation Status ← بتوضح هل عملية الـ sync نفسها نجحت ولا فشلت.

🔁 سادساً Refresh :

- الـ refresh هو اللي بيخلي ArgoCD يعيد مقارنة الكود اللي في Git بالحالة الحالية في الـ cluster
- بناءً على الفرق، ممكن:
  - يعمل Auto Sync لو مفعل.
  - أو يبلغ الـ admin إن فيه تغييرات محتاجة تطبيق يدوي.

❤️ سابعاً Health Status :

- ArgoCD بيقدّم تقييم صحي (Health Assessment) تلقائي لموارد Kubernetes زي:
  - Pods
  - Services
  - Deployments
- وفي الآخر بيحسب الحالة العامة للتطبيق بالكامل.

🧩 ملخص سريع (Pipeline Overview)

1. الكود بيتخزن في Git ← ده الـ Target State
2. ArgoCD يقارن بين الكود ده وبين اللي موجود فعلاً في الـ cluster → Live State
3. لو فيه فرق ← بيبدأ عملية Sync
4. بعد الـ Sync ← ArgoCD يحدث الحالة ويعمل تقييم صحي للتطبيق.

CICD With GitOps

🎯 أولاً: الفكرة العامة

الـ GitOps بيعتمد على وجود (Repositories) 2 بدل واحد:

1. **Application Repository** ← فيه الكود الفعلي للتطبيق، زي ملفات الـ source code ، والـ Dockerfile ، وأي configurations أو certificates خاصة بالتطبيق.
2. **Configuration Repository** ← فيه ملفات الـ Kubernetes manifests (زي deployment.yaml ، service.yaml .... إلخ) اللي بتوصف إزاي التطبيق يتنشر على الـ cluster

➕ الفصل بين الاتنين بيسمح إنك تتحكم في الكود والـ deployment بشكل منفصل وآمن.

⚙️ ثانياً : دور ArgoCD في GitOps

❖ ArgoCD شغال جوه الـ Kubernetes cluster ، وبيتابع باستمرار الـ configuration repository

كل ما يحصل فيه تعديل، ArgoCD بيعمل:

- Pull للتغييرات الجديدة.
- Apply للـ manifests على الـ cluster

❖ بكده بيضمن إن حالة الـ cluster دايمًا مطابقة للكود اللي في الـ Git repo

---

🔄 ثالثاً: خطوات الـ CI/CD في GitOps

🧩 المرحلة الأولى: الـ Developer يكتب الكود :

➢ المطور بيضيف أو يعدل كود التطبيق في Application Repository

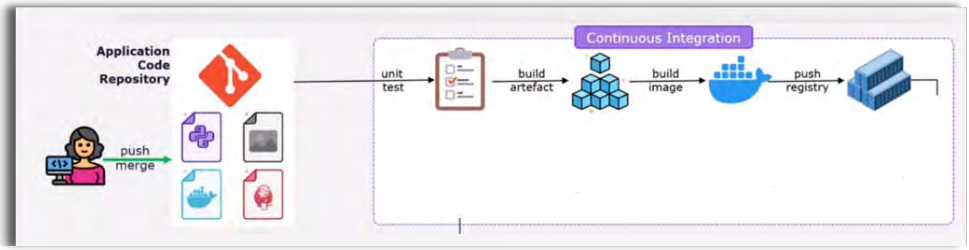
ممكن يضيف:

- كود جديد.
- Dockerfile
- Jenkinsfile (لتعريف مراحل الـ pipeline)

⚙️ المرحلة الثانية : الـ Jenkins Pipeline

➢ هنا بيبدأ الـ Jenkins CI (Continuous Integration) بينفّذ الخطوات دي :

1. تشغيل اختبارات Unit Tests للتأكد إن الكود شغال.
2. Build التطبيق.
3. Build Docker Image للتطبيق.
4. Push الـ (image) دي على الـ Container Registry (زي Docker Hub أو ECR)



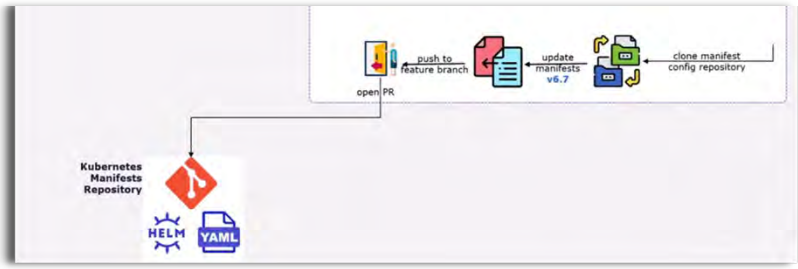
🧠 Best Practice

- استخدم tagging واضح للإصدار، مثلاً:

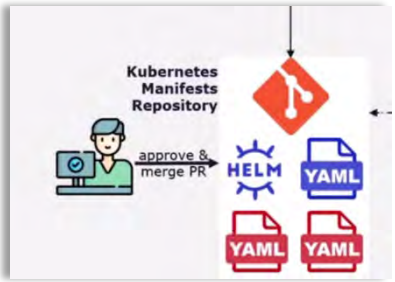
myapp:v1.2.3 عشان تسهل تتبع الإصدارات والـ rollback بعدين.

المرحلة الثالثة: تحديث الـ Kubernetes Manifests

- بعد ما (image) الجديدة تترفع، هانحتاج نحدث الـ manifests اللي في configuration repo عشان نستخدم الإصدار الجديد.
- الخطوات بتكون كده:
  1. Jenkins يعمل Clone للـ configuration repo
  2. يعدّل في ملف الـ deployment.yaml ، ويغيّر السطر اللي فيه :
     - image: myapp:old-version الي image: myapp:new-version
  3. يعمل Commit ويدفع التغييرات على feature branch
  4. يفتح Pull Request (PR) للـ main branch



المرحلة الرابعة : الـ Review والـ Merge



- الـ Project Manager أو الـ Architect بيراجع الـ PR :
  - لو التغييرات سليمة ← يعمل Approve & Merge
  - لو فيها مشكلة ← يرفض أو يرجع التعديلات للمطور.
- بمجرد ما الـ PR يتعمل له merge ، التغييرات تبقى في main branch

---

المرحلة الخامسة : ArgoCD يبدأ الـ Deployment

- ArgoCD بيراقب الـ main branch ، وأول ما يكتشف تغيير:
  1. يسحب التحديث الجديد.
  2. يطبقه تلقائيًا على الـ cluster
- يعني التحديث نزل في الـ Git ← ArgoCD طبّقه على طول ← اذا الكلاستر متزامن مع الكود الجديد.

المرحلة السادسة : الـ Rollback (لو حصلت مشكلة)

- لو الإصدار الجديد فيه Bug أو مشكلة في الأداء، ممكن نرجع بسرعة للإصدار السابق.
- في ArgoCD نقدر نعمل rollback بطريقتين:

من خلال الـ UI :
- تفتح التطبيق.
- تختار الإصدار القديم من الـ history
- تضغط Rollback

أو من خلال CLI : argocd app history myapp
- ده بيعرض كل الإصدارات اللي اتنشرت .
- بعدها: argocd app rollback myapp <revision-id>
- وده بيرجع التطبيق للحالة المستقرة السابقة فورًا.

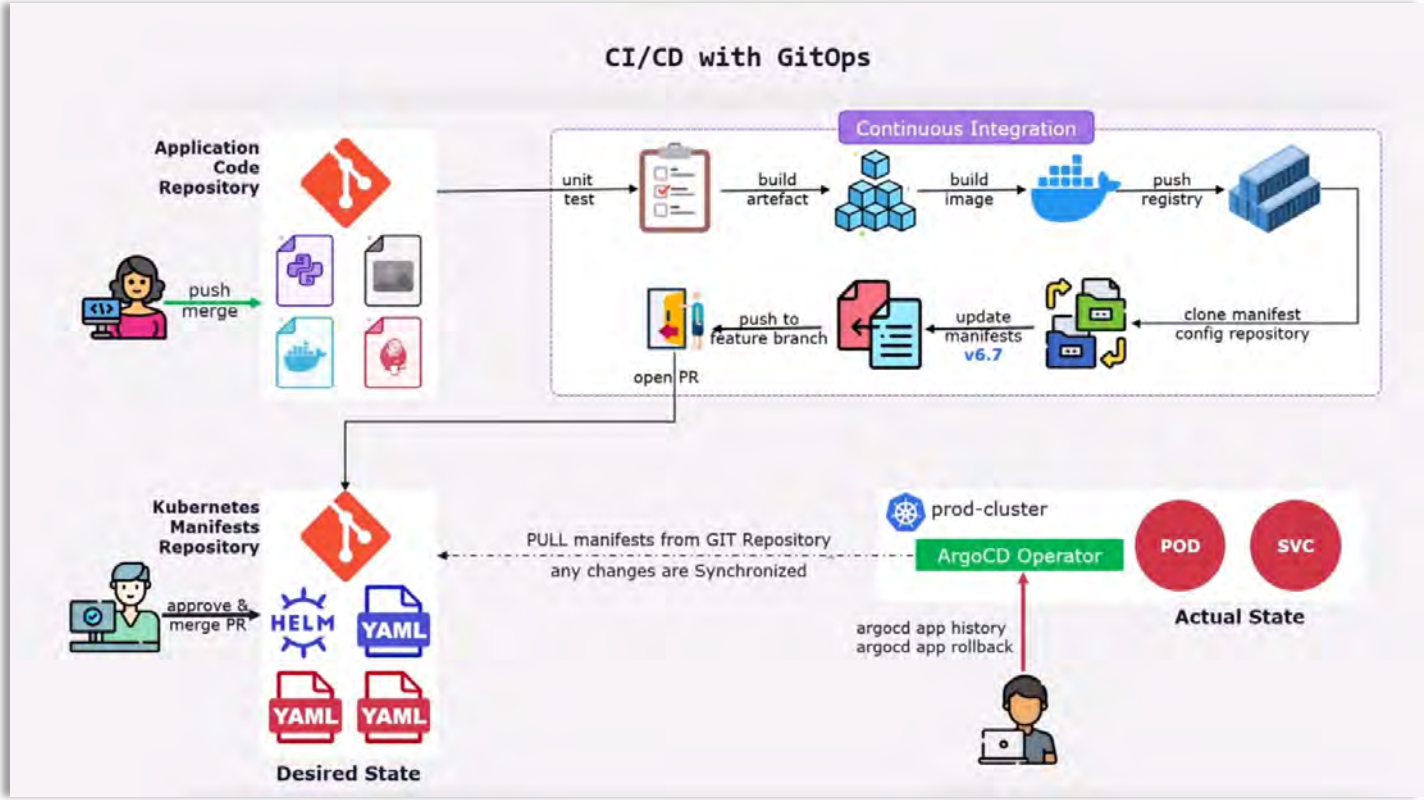
نصيحة عملية:

احفظ دومًا revision ID الخاص بكل إصدار في pipeline logs ← ده بيسهل عملية rollback السريعة.

🧩 المرحلة الأخيرة: الـ Flow بالكامل

ملخص الصورة الكاملة:

1. Developer يعمل تعديل على الكود ← push للـ Application repo
2. Jenkins يبني الصورة ويعمل push للـ registry
3. Pipeline يعدّل الـ Kubernetes manifests ← يفتح PR
4. Manager/Architect يعمل merge للـ PR
5. ArgoCD يلاحظ التغيير ← يعمل sync ← ينشر النسخة الجديدة.
6. Rollback سريع لو في مشكلة بالإصدار الجديد.



💡 Real-World Tips

- خلي Jenkins هو المسؤول عن Build Images فقط، وسيب ArgoCD مسؤول عن الـ deployment
- افصل الـ repositories لتقليل احتمالية نشر كود غير مجرّب.
- استخدم Git branching strategy واضحة (main/staging/dev)
- فعّل notifications من ArgoCD للـ Slack أو Teams لما يحصل deployment أو rollback

Manifest Repository & Configure ArgoCD

🎯 الفكرة العامة

- ❑ في الجزء ده إحنا بنجهز الـ Manifest Repository اللي فيه ملفات الـ YAML الخاصة بالتطبيق (Deployment + Service + Secrets) ،
- ❑ وبعد كده بنربطه مع Argo CD عشان يبدأ يـ manage التطبيق ويقدر يـ deploy التغييرات بشكل أوتوماتيكي (GitOps approach)

🧠 الخطوات بالتفصيل

sidd-harth/solar-system-gitops-argocd-gitea

1. تحضير الـ Manifest Repository

- بنبدأ نـ clone الـ repository اللي على GitHub ، واللي فيه فولدر باسم Kubernetes يحتوي على:
  - deployment.yaml
  - service.yaml
- لما بنراجع الـ deployment.yaml بنلاحظ إنه محتاج Secret اسمه mongodb-secrets، وده مش موجود.

---

2. إنشاء الـ Secret محليًا

- علشان نحل المشكلة دي، بنروح على الـ Kubernetes Cluster ونتأكد إنه شغال تمام:
  - kubectl get namespace
  - kubectl get nodes -o wide
  - ممكن نستخدم alias k بدل kubectl (لو متعرف في الـ shell)
- بعد كده بننشئ الـ Secret اللي المفروض التطبيق هيستخدمه، واللي بيحتوي على بيانات الاتصال بقاعدة MongoDB :

```
root@jenkins-controller-1 in ~
❯ kubectl -n solar-system create secret generic mongo-db-creds --from-literal=MONGO_URI=mongodb+srv://supercluster.d83jj.mongodb.net/superData --from-literal=MONGO
_USERNAME=superuser --from-literal=MONGO_PASSWORD=SuperPassword --save-config --dry-run=client -o yaml  > mongo-creds_k8s-secret.yaml
```

```
❯ cat mongo-creds_k8s-secret.yaml
apiVersion: v1
data:
  MONGO_PASSWORD: U3VwZXJQYXNzd29yZA==
  MONGO_URI: bW9uZ29kYitzcnY6Ly9zdXBlcmNsdXN0ZXIuZDgzamoubW9uZ29kYi5uZXQvc3VwZXJEYXRh
  MONGO_USERNAME: c3VwZXJ1c2Vy
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"kind":"Secret","apiVersion":"v1","metadata":{"name":"mongo-db-creds","namespace":"solar-system","creationTimestamp":null},"data":{"MONGO_PASSWORD":"U3VwZXJQ
YXNzd29yZA==","MONGO_URI":"bW9uZ29kYitzcnY6Ly9zdXBlcmNsdXN0ZXIuZDgzamoubW9uZ29kYi5uZXQvc3VwZXJEYXRh","MONGO_USERNAME":"c3VwZXJ1c2Vy"}}
  creationTimestamp: null
  name: mongo-db-creds
  namespace: solar-system
```

- لكن لاحظ هنا 👀
  إحنا مش بننشئه جوه الكلاستر مباشرة، إحنا بس بنخليه يولّد ملف YAML نحفظه عندنا.

🔐 ليه ماينفعش نرفع الـ Secret العادي على GitHub ؟

- القيم جوه الملف بتكون Base64 Encoded ، مش مشفّرة.
- يعني أي حد يقدر يفكّها ويرجع البيانات الأصلية — وده خطر جدًا في بيئة Production
- وعشان كده بنستخدم طريقة التشفير الآمن باستخدام Bitnami Sealed Secrets

---

استخدام Bitnami Sealed Secrets

- Bitnami Sealed Secrets هو أداة بتخليك ترفع الـ Secrets على GitHub لكن بشكل مشفّر.
- عندنا Pod اسمه sealed-secrets-controller بيكون شغال في الـ kube-system namespace
- نتأكد إنه موجود : kubectl get pods -n kube-system
- بعدين نجيب الشهادة العامة (Public Certificate) الخاصة بيه علشان نستخدمها في التشفير:

```
root@jenkins-controller-1 in ~
❯ k -n kube-system get secrets sealed-secrets-key8dv5k -o json | jq -r .data'."tls.crt"' | base64 -d > sealedSecret-publicCert.crt
```

- بعدها نستخدم أداة CLI اسمها kubeseal علشان نشفّر الـ secret :

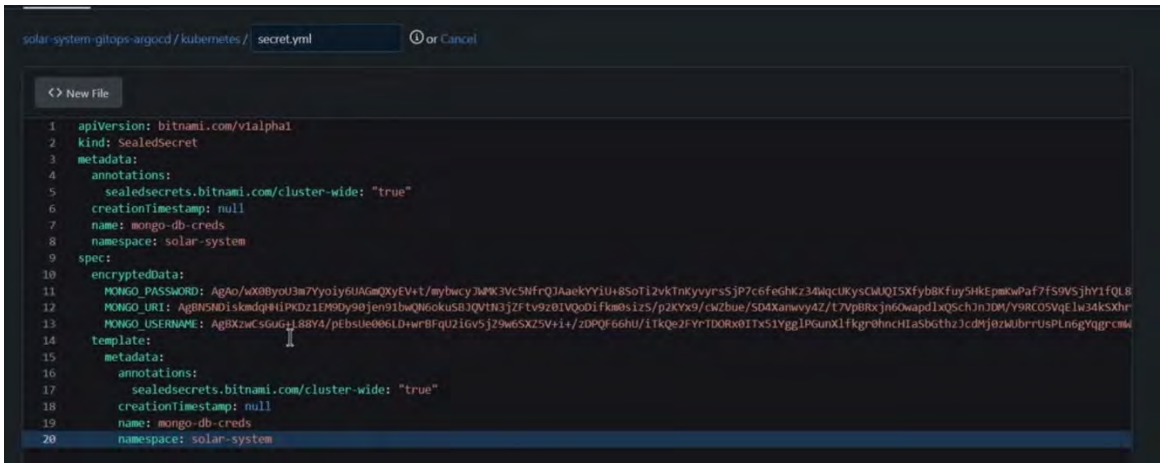
```
root@jenkins-controller-1 in ~
❯ kubeseal -o yaml --scope cluster-wide --cert sealedSecret-publicCert.crt < mongo-creds_k8s-secret.yaml > mongo-creds_sealed-secret.yaml
```

➢ النتيجة بتكون ملف sealed-secret.yaml، فيه البيانات مشفّرة بالكامل.

---

3. رفع الـ Sealed Secret على GitHub

- نضيف الملف باسم secret.yaml داخل الـ repo
- هانسخ ناتج الامر cat mongo-creds-sealed-secret.yaml ونحطه في ملف الـ secret.yaml اللي في الـ repo
- بنلاحظ أن الـ metadata بيكون فيه:
  - apiVersion: bitnami.com/v1alpha1
  - kind: SealedSecret
  - scope: cluster-wide
  - encryptedData : اللي فيها البيانات المشفرة .



---

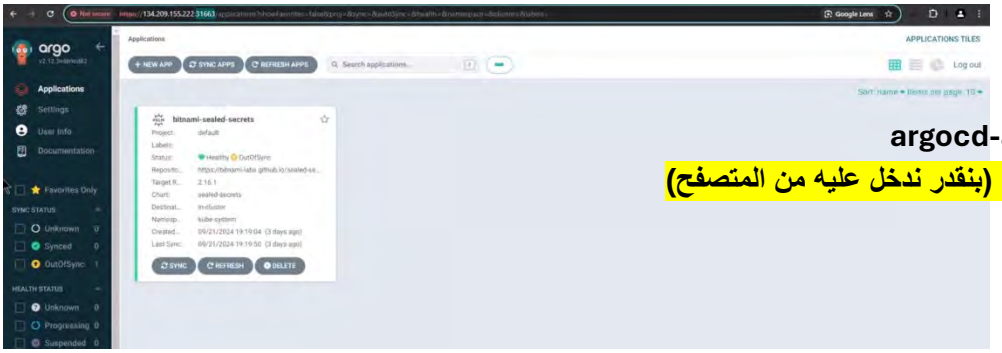
4. تجهيز Argo CD

- نتاكد إن Argo CD شغال داخل (cluster (namespace argocd
- نشوف الـ components بالأمر : kubectl get all -n argocd

هنلاقي:

- argocd-server
- argocd-repo-server
- argocd-application-controller

- الـ UI شغال على NodePort 31663 (بنقدر ندخل عليه من المتصفح)



5. **إنشاء Application جديدة في Argo CD**

- من واجهة Argo CD GUI :
  1. نضغط "+ NEW APP"
  2. نختار الاسم: solar-system-argo-app
  3. نحدد المشروع: default
  4. نختار نوع المزامنة Manual : (علشان نتحكم فيها يدوي)
  5. نحط رابط الـ Git repo
  6. نحدد الـ branch : main
  7. نحط الـ path : ./Kubernetes
  8. نحدد الـ namespace : solar-system
  9. Cluster URL: https://kubernetes.default.svc
- بعد الإنشاء، Argo CD يقرأ الملفات (deployment + service + sealed-secret) لكنه مش هايعمل Sync لحد ما نضغط "SYNC"

🛠️ الأوامر الأساسية

الهدف	الأمر
إنشاء secret بدون apply	`kubectl create secret generic mongodb --from-literal=user=... --dry-run=client -o yaml > mongodb-secret.yaml`
استخراج TLS cert من sealed-secrets	kubectl get secret -n kube-system -o json`
تشفير secret	`kubeseal --cert pub-cert.pem --scope cluster-wide -o yaml < mongodb-secret.yaml > sealed-secret.yaml`
عرض مكونات ArgoCD	`kubectl get all -n argocd`

💡 ملاحظات عملية Best Practices

- ممنوع رفع أي secrets عادية على Git حتى لو Base64
- 🔐 **Sealed Secrets أفضل طريقة لحماية secrets ضمن GitOps workflows**
- 🚀 يفضل ضبط ArgoCD على auto-sync في بيئات غير Production للتجربة، لكن في الـ prod استخدم manual sync + PR review
- 🧩 احرص إن الـ namespace اللي بتستخدمه في ArgoCD مطابق لما في الـ manifests

---

📋 الملخص

كدا شرحنا خطوة إعداد الـ Manifest Repository وربطه بـ Argo CD :

1. كل الملفات (deployment/service/secret) موجودة في GitHub repo
2. **الـ secrets اتعملها تشفير باستخدام Bitnami Sealed Secrets**
3. Argo CD بيقرأ الـ repo ويكوّن Application بناءً على الـ manifests
4. حالياً التطبيق جاهز لكنه مش متزامن (Manual Sync) لحد ما نحدث الـ image ونعمل Sync يدوي.

➢ بكده بقى عندنا بداية كاملة لبيئة GitOps حقيقية وآمنة 💪

K8s Deploy-Update Image Tag

1. الفكرة العامة

- ❑ إحنا عايزين نوصّل Jenkins بـ ArgoCD بحيث لما Jenkins يبني صورة جديدة (Docker image) للتطبيق، ArgoCD يعرف إن في تحديث، ويروح يعمل Deploy تلقائي على الـ Kubernetes Cluster

➢ طيب السؤال:

إزاي ArgoCD يعرف إن في صورة جديدة؟

➢ الإجابة 👇

ArgoCD مش بيتابع الـ DockerHub ، هو بيتابع Repository على GitHub اللي فيه ملفات الـ Kubernetes (manifest files).

يعني لازم Jenkins يعدّل ملف الـ deployment.yaml اللي جواه اسم الصورة (image name + tag)

```
matchLabels:
  app: solar-system
strategy: {}
template:
  metadata:
    labels:
      app: solar-system
  spec:
    containers:
    - image: siddharth67/solar-system:3e906e3be059342b1916f020c034344fb267ddca
      imagePullPolicy: Always
      name: solar-system
```

2. الـ Manifest Repository ⚙️

- فيه repo اسمه solar-system-gitops-argo-cd
- جواه مجلد kubernetes/ وفيه ملف deployment.yaml
- Jenkins لازم يفتح الملف ده ويحدث السطر

بتاع: image: sedat67/solar-system:<tag>

o ويخلي <tag> هو نفس الـ git commit اللي اتبنى في الـ pipeline

3. إضافة Stage جديدة في Jenkinsfile 🧱

```
stage('K8S Update Image Tag') {
    when {
        branch 'PR*'
    }
    steps {
        sh 'git clone -b main http://64.227.187.25:5555/dasher-org/solar-system-gitops-argocd'
        dir("solar-system-gitops-argocd/kubernetes") {
            sh '''
                #### Replace Docker Tag ####
                git checkout main
                git checkout -b feature-$BUILD_ID
                sed -i "s#siddharth67.*#siddharth67/solar-system:$GIT_COMMIT#g" deployment.yml
                cat deployment.yml

                #### Commit and Push to Feature Branch ####
                git config --global user.email "jenkins@dasher.com"
                git remote set-url origin http://$GITEA_TOKEN@64.227.187.25:5555/dasher-org/solar-system-gitops-argocd
                git add .
                git commit -em "Updated docker image"
                git push -u origin feature-$BUILD_ID
            '''
        }
    }
}
```

لاحظ اننا لازم نعمل configure GITEA TOKEN مشروحة في رابعا

➢ الـ Stage دي اسمها: K8s-update-image-tag

➢ بتعمل الآتي: الخطوات داخل الـ stage :

المرحلة دي بتعمل كل الآتي:

1. تنزل ملفات الـ manifests من GitHub
2. تفتح الملف اللي فيه الـ deployment (اللي بيقول ArgoCD يشتغل على أي صورة)
3. تغيّر اسم الصورة جوه الملف وتكتب التاج الجديد اللي Jenkins بناه.
4. ترفع التعديل ده تاني على GitHub
5. ArgoCD يشوف التغيير ويعمل Deploy جديد.

علشان تفهم الفكرة ببساطة فتخيل ان ArgoCD هو الشخص اللي هايشطب الشقة ويسلمها عالمفتاح .. بس هو مستني المقاول (اللي في حالتنا دي هو Jenkins) انه يخلص بناء الشقة الأول

⚙️ ثالثًا: الخطوات بالتفصيل (هنشرح كل أمر في السكريبت)

1. 🧩 ننزل الـ repo بتاع الـ manifests

- sh 'git clone https://github.com/sedat67/solar-system-gitops-argo-cd.git'

🔹 ده بينزل الملفات اللي ArgoCD بيستخدمها للنشر.

---

2. 🗂️ ندخل على المجلد اللي فيه ملف الـ deployment

- dir('solar-system-gitops-argo-cd/kubernetes') {

🔹 يعني كل الأوامر اللي بعد كده هتتنفذ جوه المجلد ده.

---

3. 🌿 نعمل branch جديد علشان نرفع التعديل عليه

- git checkout -b feature-${BUILD_ID}

🔹 ${BUILD_ID} هو رقم كل build في Jenkins

🔹 ده مفيد علشان ميحصلش تعارض بين أكثر من build

---

4. 📄 نعدّل سطر الصورة في ملف الـ deployment

- sed -i "s|sedat67.*|sedat67/solar-system:${GIT_COMMIT}|g" deployment.yaml

✅ شرحها:

- sed = أمر تعديل على النصوص.
- -i = يعدّل الملف مباشرة.
- s|...|...|g = استبدال.
- بيبحث عن أي سطر فيه sedat67 ← ويغيره إلى sedat67/solar-system:${GIT_COMMIT}
- ${GIT_COMMIT} = رقم الـ commit اللي اتبنى في Jenkins (يعني التاج الجديد)

➢ مثلاً:

- قبل التعديل 👇 : image: sedat67/solar-system:oldtag
- بعد التعديل 👇 : image: sedat67/solar-system:3f9c2d1

---

5. 👀 نعرض الملف بعد التعديل علشان نتأكد : cat deployment.yaml

---

6. 💾 نرفع التعديل على GitHub

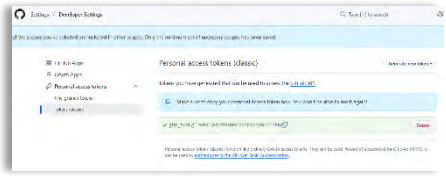
- Git remote set-url origin http://$GITEA_TOKEN@64.227.187.25:5555/dasher-org/solar-system-gitops-argocd
- git add .
- git commit -am "update image tag"
- git push -u origin feature-${BUILD_ID}

🔹 كده التعديل اتبعت على GitHub في فرع جديد.

🔐 رابعًا Jenkins : محتاج صلاحية للـ Push

➢ مينفعش Jenkins يعمل push من غير صلاحيات.

➢ فبنروح نعمل Personal Access Token من GitHub



- الخطوات:

1. نفتح GitHub → Settings → Developer Settings → Tokens
2. نعمل Token جديد (scope: repo)
3. نحفظه في Jenkins من خلال:

Jenkins → Credentials → Secret Text → paste the token generated from gitHub

- ونسميه مثلًا github-api-token

4. بعدين نستخدمه في Jenkinsfile ونضيفه علي مستوي الـ environment :

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
    MONGO_DB_CREDS = credentials('mongo-db-credentials')
    MONGO_USERNAME = credentials('mongo-db-username')
    MONGO_PASSWORD = credentials('mongo-db-password')
    SONAR_SCANNER_HOME = tool 'sonarqube-scanner-610';
    GITEA_TOKEN = credentials('gitea-api-token')
}
```

- وبكدا وقت الـ push اللي في الـ stage الدنيا هاتمشي مظبوط:

git remote set-url origin https://${GITEA_TOKEN}@github.com/sedat67/solar-system-gitops-argo-cd.git

---

Pipeline: Basic Steps

خامسًا: تنظيف الـ Workspace

- لو Jenkins شغّل نفس الـ pipeline أكتر من مرة، ممكن يحصل تضارب.
علشان كده بنحط في آخر السكريبت:

```
post {
    always {
        script {
            if (fileExists('solar-system-gitops-argocd')) {
                sh 'rm -rf solar-system-gitops-argocd'
            }
        }
```

- ده بيمسح المجلد في نهاية كل build علشان نبدأ جديد في اللي بعده.

- طاب المرحلة دي تشتغل امتى؟
  - الـ stage دي بتشتغل بس لما يكون Jenkins بيبني فرع اسمه PR-.*.. (يعني Pull Request)
  - ده بيتأكد بشرط: when { branch pattern: "PR-.*", comparator: "REGEXP" } اللي موجودة في السكريبت

Best Practices

1. خزن الـ GitHub Token في Jenkins Credentials ، متكتبوش جوه الكود.
2. استخدم Branch لكل Build علشان متكسرش الـ main
3. بعد ما تتأكد إن ArgoCD شاف التغيير، امسح الـ feature branch
4. استخدم commit message واضح زي : "update image tag to ${GIT_COMMIT}"

ملخص

الشرح دا ببساطة بيخلّي Jenkins بعد ما يبني الصورة الجديدة:

1. يدخل على الـ repo اللي ArgoCD بيستخدمه.
2. يعدّل ملف الـ deployment ويغيّر الـ image tag
3. يرفع التغيير على GitHub
4. ArgoCD يشوف التغيير ويعمل Deploy تلقائي.
5. يعني:

Jenkins يبني ← يعدّل manifest ← ArgoCD ينشر ✅

K8S Deploy-Raise PR

🎯 هانتكلم عن:

إزاي نعمل Stage في Jenkins Pipeline علشان نرفع Pull Request تلقائيًا في GitHub بدل ما نعملها يدويًا.

🧩 أولًا: الفكرة العامة

- في GitHub تقدر ترفع Pull Request يدوي من الـ UI بزرار New Pull Request
- لكن الهدف هنا إننا نخلّي Jenkins يعمل ده تلقائيًا باستخدام GitHub REST API

🧠 الخطوات الأساسية : (دا الشرح اللي طبقه في الفيديو علي Gitea Server) –يليه الشرح علي GitHub Server–

1️⃣ نفتح واجهة Gitea API

- في أسفل صفحة GitHub هتلاقي لينك اسمه API
- لما تضغط عليه، هيفتحلك الـ Swagger UI بتاع Gitea API ← وده فيه كل الـ endpoints اللي ممكن تستخدمها.

2️⃣ نبحث عن الـ API الخاصة بالـ Pull Request

- نكتب كلمة pull في البحث.
- هتلاقي POST /repos/{owner}/{repo}/pulls ← دي الـ API اللي بتعمل Pull Request

3️⃣ نضغط على "Try it out"

- نبدأ نملأ القيم المطلوبة:
  - Owner : اسم المنظمة أو المستخدم (مثلاً dasher-org)
  - Repo : اسم المشروع (مثلاً solar-system-gitops-argocd)
  - Body : وده فيه تفاصيل الـ PR زي:
    - title ← عنوان الـ PR
    - head ← الفرع اللي جاي منه الكود (مثلاً feature-flag-id)
    - base ← الفرع اللي هيتعمل عليه الـ merge (مثلاً main)
    - assignee ← الشخص المسؤول عن المراجعة

4️⃣ نجرب تنفيذ الطلب

- لما نضغط Execute ، Swagger بيورينا أمر cURL
  - النوع POST :
  - فيه الـ endpoint ، والـ headers ، والـ JSON payload اللي كتبناه.
- أول محاولة بتدي 404 Not Found لأن في خطأ في الاسم.
- بعد التصحيح، بتدي 401 Unauthorized لأن مافيش token

🔐 التوكن (Authentication Token)

- محتاجين GitHub Personal Access Token (اللي عملناه في الدرس اللي فات)
- بنضيفه في الـ header بالشكل ده: Authorization: token <YOUR_TOKEN>
- Swagger بيوضحلك ده لما تضغط على lock icon
- ممكن نحط التوكن في الـ Authorization header علشان المصادقة.

الخطوات الأساسية: (الشرح في حالة لو هاتطبق على GitHub Server)

- فعليًا GitHub ما عندوش Swagger UI تفاعلي داخل موقع docs.github.com.
- هو بيعرضلك فقط وصف الـ API، مش بيتيحلك تنفّذه من هناك.

REST API endpoints for pull requests - GitHub Docs

- وبالتالي ← ✅ الطريقة الصح لتجربة الـ API بنفسك (بدون Try it out)
  - هتستخدم curl (أو Postman لو تحب واجهة رسومية) علشان تبعت الطلب بنفسك.

الخطوة 1: جهّز Personal Access Token

1. افتح GitHub
2. روح على:

Settings → Developer settings → Personal access tokens → Tokens (classic)

3. اضغط Generate new token (classic)
4. ادّيه الصلاحيات التالية:
   - ✅ repo (لو الريبو خاص) أو public_repo (لو عام)
5. انسخ التوكن ← هتحتاجه كمان شوية (مثلاً ghp_xxxxxxxxxxxxxxxxxx)

الخطوة 2: افتح Terminal أو PowerShell

واكتب الأمر ده (بدّل القيم اللي بين علامات < >):

```
curl -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR_GITHUB_TOKEN>" \
  https://api.github.com/repos/<OWNER>/<REPO>/pulls \
  -d '{
    "title": "Test PR from API",
    "head": "feature-branch",
    "base": "main",
    "body": "This PR was created using GitHub REST API manually"
  }'
```

مثال عملي فعلي

لو عندك ريبو اسمه:

- Owner: youssefelzarka
- Repo: my-repo
- Base branch: main
- Feature branch: feature1 (موجود في الريبو)

```
curl -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer ghp_ABC123YOURTOKENHERE" \
  https://api.github.com/repos/youssefelzarka/my-repo/pulls \
  -d '{
    "title": "Auto PR from API",
    "head": "feature1",
    "base": "main",
    "body": "Created automatically via curl"
  }'
```

الخطوة 3: شوف النتيجة

لو كل حاجة تمام، هيظهر لك رد JSON زي ده:

```
{
  "url": "https://api.github.com/repos/youssefelzarka/my-repo/pulls/5",
  "id": 123456,
  "number": 5,
  "state": "open",
  "title": "Auto PR from API",
  "html_url": "https://github.com/youssefelzarka/my-repo/pull/5"
}
```

الخطوة 4 (اختياري): لو تحب واجهة رسومية بدل terminal

استخدم Postman :

1. افتح Postman واختار :
2. method = POST
3. URL : https://api.github.com/repos/<OWNER>/<REPO>/pulls
4. في التبويب Headers :
   - Accept: application/vnd.github+json
   - Authorization: Bearer YOUR_GITHUB_TOKEN
5. في التبويب Body اختار raw ← JSON .. واكتب:
6. اضغط send ← وشوف النتيجة في الأسفل

```
{
  "title": "PR from Postman",
  "head": "feature1",
  "base": "main",
  "body": "Created via Postman API test"
}
```

🧰 بعد كده نروح للـ Jenkinsfile

نضيف Stage جديدة باسم : stage('K8s Raise PR')

- نحطها بعد مرحلة update image tag
- الشرط إنها تشتغل فقط لما يكون الفرع Pull Request branch (PR)
- جواها بنكتب Shell Script بيشغل أمر cURL اللي جربناه.

شكل الأمر داخل Jenkins :

```
stage('K8S - Update Image Tag') {...
}

stage('K8S - Raise PR') {
    when {
        branch 'PR*'
    }
    steps {
        sh """
            curl -X 'POST' \
                'http://64.227.187.25:5555/api/v1/repos/dasher-org/solar-system-gitops-argocd/pulls' \
                -H 'accept: application/json' \
                -H 'Authorization: token $GITEA_TOKEN' \
                -H 'Content-Type: application/json' \
                -d '{
                "assignee": "gitea-admin",
                "assignees": [
                    "gitea-admin"
                ],
                "base": "main",
                "body": "Updated docker image in deployment manifest",
                "head": "feature-$BUILD_ID",
                "title": "Updated Docker Image"
            }'
        """
    }
```

- ${GITHUB_TOKEN} هو Credential مضاف في Jenkins (Environment variable) (شرحناه الدرس اللي فات)
- الـ body هو نفس الـ JSON اللي استخدمناه في Swagger

⚙️ لما نحفظ وندمج التغييرات:

- Jenkins هيشغل الـ pipeline
- المرحلة الجديدة (K8s Raise PR) هتعمل Pull Request تلقائي في GitHub
- تقدر تشوف الـ PR في الـ repository ← بيتكون باسم الفرع اللي اتعمل منه.

🧩 بعد كده:

- GitHub بيعمل Status Checks علشان يتأكد إن الـ Jenkins Build نجح قبل الدمج.
- نقدر نضيف Branch Protection Rules علشان نمنع الدمج إلا لو الـ Pipeline نجح.

✅ النتيجة النهائية:

- Jenkins بيرفع Pull Request تلقائيًا.
- Argo CD بيلاحظ التغيير الجديد في الـ repo
- بعد الدمج في الـ main branch ، Argo CD بيبدأ تحديث التطبيق في Kubernetes أوتوماتيك.
- نقدر نعمل Sync يدوي أو تلقائي من ArgoCD علشان نحدث الـ deployment (مشروحة تاليا)

Integration between Jenkins & ArgoCD

🎯 الفكرة العامة:

- ❑ بعد ما Jenkins يرفع الـ Pull Request ويتعمل merge في الـ main branch ، يبدأ دور ArgoCD
- ❑ الهدف إن ArgoCD يلاحظ أي تغييرات حصلت في الـ Git Repository
(زي تعديل نسخة الـ Docker image مثلاً) .. ويقوم بتحديث التطبيق على Kubernetes بشكل تلقائي أو يدوي حسب الإعدادات.

🧩 الخطوات بالتفصيل

1 ArgoCD بيراقب الـ Repository

- ArgoCD بيكون مربوط بـ Git repo اللي فيه ملفات الـ manifests الخاصة بالتطبيق.
- بمجرد ما يحصل merge في الـ main branch ،
ArgoCD بيعمل polling أو webhook trigger علشان يعرف إن فيه تغيير جديد.

📌 ملحوظة عملية:

- ➢ الأفضل دايمًا تستخدم webhook من GitHub بدل الـ polling لأن الـ webhook أسرع وأخف على السيرفر.
- ➢ تقدر تعمل ده من إعدادات GitHub ← Webhooks ← تضيف رابط ArgoCD API

---

2 ArgoCD يقارن الـ States

- بيشوف الـ target state اللي في الـ Git repo (يعني المفروض يكون إيه وضع الكلاستر)
- وبعدين يقارنها مع الـ live state (الوضع الفعلي جوه الـ Kubernetes cluster)
- لو لقى اختلاف (زي تحديث في tag بتاع الـ image مثلاً) .. بيبدأ عملية Sync علشان يخلّي الاتنين متطابقين.

---

3 عملية الـ Sync

عندك طريقتين:

- Automatic Sync : ArgoCD بيعمل التحديث لوحده فورًا.
- Manual Sync : بيعرضلك الفرق في الـ UI وتقرر بنفسك تعمل Sync أو لأ.

📌 أفضل ممارسة (Best Practice) :

- في بيئات الـ Production خلي الـ Sync يدوي
- علشان تراجع التغييرات الأول.
- لكن في بيئات الـ Dev أو Test ممكن تخليه تلقائي.

---

4 أثناء الـ Sync

- ➢ ArgoCD بيبدأ يطبّق ملفات الـ YAML الجديدة،

زي:

- Deployment
- Service
- ConfigMap
- Secrets
- وغيرهم...

وبيحدث الـ image version الجديدة اللي Jenkins عدّلها في الـ repo

5️⃣ التحقق من النتيجة

بعد الـ Sync ، ArgoCD بيعمل:

- Health Check على كل الـ resources
- بيعرض حالة التطبيق في الـ Dashboard (Healthy, Progressing, Degraded .... إلخ)

كمان ممكن تتابع من الـ CLI :

- argocd app get my-app
- argocd app history my-app
- argocd app sync my-app

6️⃣ في حالة وجود مشكلة

لو الـ image الجديدة فيها bug أو التطبيق فشل:

- ممكن نعمل rollback باستخدام أمر : argocd app rollback my-app <revision-number>
- أو من الـ UI تختار History → Rollback .. وده بيرجع التطبيق لحالة مستقرة سابقة.

📌 نصيحة عملية:

احتفظ بعدد كفاية من الإصدارات القديمة في الـ repo .. علشان تقدر ترجع بسهولة لأي نسخة مستقرة.

🧠 Best Practices

1. ✅ استخدم Webhooks بين GitHub و ArgoCD لتقليل التأخير.
2. ✅ خليك على Manual Sync في الـ Production لتقليل الأخطاء.
3. ✅ فعّل Notifications (Slack, Email) لكل Sync أو Rollback
4. ✅ خزن الـ tokens في Jenkins و ArgoCD كـ Secrets مش نصوص مكشوفة.
5. ✅ استخدم Branch Naming واضح (مثلاً /release , /hotfix, /feature)

📜 الملخص العام

1. Jenkins بيبني التطبيق ويعمل Docker image جديدة.
2. بيحدث ملف الـ Deployment في Git repo (YAML file)
3. Jenkins بيرفع Pull Request تلقائيًا باستخدام GitHub API
4. لما الـ PR يتدمج في الـ main branch ← ArgoCD يلاحظ التغيير.
5. ArgoCD بيقارن الحالة الفعلية باللي في Git ويعمل Sync
6. بيحدث التطبيق على Kubernetes أو يعمل Rollback لو حصل خطأ.

🎯 النتيجة النهائية:

❑ كده بقينا عاملين Pipeline متكامل 100 % بين Jenkins و GitHub وArgoCD
بشكل أوتوماتيكي — من الكود لحد النشر الفعلي على Kubernetes .. ومن غير تدخل بشري تقريبًا ✌️

DAST & Manual Input

🎯 الفكرة العامة :

❑ الهدف إننا نضيف في الـ Jenkins Pipeline :

1. مرحلة manual input ← المستخدم يوافق فيها إن الـ PR (Pull Request) تم دمجه وإن الـ ArgoCD Sync خلص.
2. مرحلة DAST Scan ← عشان نعمل اختبار أمان ديناميكي باستخدام أداة ZAP على التطبيق بعد ما يتم نشره.

🧩 شرح الجزء الأول – مفهوم الـ DAST :

🔹 DAST = Dynamic Application Security Testing

➢ هو نوع من أنواع اختبار الأمان اللي بيتم على تطبيق شغال فعلاً (running app) ، وبيفحص الثغرات زي:

- SQL Injection
- XSS (Cross-Site Scripting)
- Misconfigurations

👀 الفرق بين DAST و SAST :

النوع	يعتمد على	يحتاج التطبيق يشتغل؟
SAST (Static)	الكود نفسه	❌ لا
DAST (Dynamic)	التطبيق وهو شغال	✅ نعم

📌 يعني DAST بيشتغل كأنه "هاكر أبيض" بيجرب يدخل payloads خبيثة ويشوف هل التطبيق آمن ولا لا.

🧰 الأدوات المستخدمة:

ZAP

فيه نوعين من الأدوات:

- Open Source : زي OWASP ZAP (Zed Attack Proxy)
- Commercial : زي Burp Suite و Netsparker

📌 في الفيديو، استخدم OWASP ZAP (المجاني)

🐳 استخدام ZAP من خلال Docker :

- بدل ما تثبّت ZAP يدوي، تقدر تستخدمه كـ Docker image (شوف الـ Docs)
- ZAP عنده 3 أنواع من الـ scans :

نوع الـ Scan	الوصف
Baseline Scan	فحص سريع (passive only).
Full Scan	فحص شامل (active + passive).
API Scan	فحص APIs مبني على OpenAPI أو GraphQL أو SOAP.

💡 في الديمو استخدم API Scan فقط لتبسيط الشرح.

شرح الجزء الثاني – DAST Scan باستخدام ZAP :

- الجزء ده معمول عشان يضيف خطوة أمان (Security Testing) في الـ CI/CD pipeline ،
- الهدف منها إننا بعد ما نرفع التطبيق على Kubernetes ونتأكد إنه شغال، نستخدم أداة OWASP ZAP تعمل فحص أمان ديناميكي (DAST) على التطبيق .. يعني نحاول نكتشف ثغرات فعلية في التطبيق وهو شغال.
- الفكرة كلها إننا:

1. نستنى المستخدم يأكد إن الـ PR (Pull Request) اتدمج و ArgoCD عمل Sync
2. بعدها Jenkins يفتح Container فيه ZAP
3. ZAP يبدأ يعمل فحص للأمان على الـ API ويطلع تقارير مفصّلة.

- أضاف Stage جديدة في الـ Jenkinsfile بالشكل ده:

```
stage('DAST - OWASP ZAP') {
    when {
        branch 'PR*'
    }
    steps {
        sh '''
            #### REPLACE below with Kubernetes http://IP_Address:30000/api-docs/ #####
            chmod 777 $(pwd)
            docker run -v $(pwd):/zap/wrk/:rw  ghcr.io/zaproxy/zaproxy zap-api-scan.py \
            -t http://134.209.155.222:30000/api-docs/ \
            -f openapi \
            -r zap_report.html \
            -w zap_report.md \
            -J zap_json_report.json \
            -x zap_xml_report.xml
        '''
    }
}
```

شرح الأسطر:

- **chmod 777 .** ← عشان يعطي صلاحيات للـ container إنه يكتب الملفات الناتجة في الـ working directory
- **docker run -v $(pwd):/zap/wrk** ← بيربط المجلد الحالي داخل الـ container عشان يحفظ التقارير فيه.
- **zap-api-scan.py** ← سكريبت ZAP الخاص بفحص الـ APIs .. ويعتبر ده جزء من ZAP نفسه .. وبيقدر يقرأ OpenAPI spec أو Swagger ويبدأ يعمل اختبارات على الـ endpoints.
- **-t http://app-service/api/docs** ← الـ target (الـ API docs بتاعة التطبيق)
- **-f openapi** ← نوع الـ API specification
- **-r -w -J -x** ← توليد تقارير بصيغ HTML, Markdown, JSON, XML

مرحلة الـ Manual Approval :

- قبل تنفيذ الفحص، لازم المستخدم يأكد إن التطبيق تم نشره بنجاح.
- تمت إضافة Stage اسمها "App Deployed?" داخل الـ Jenkinsfile

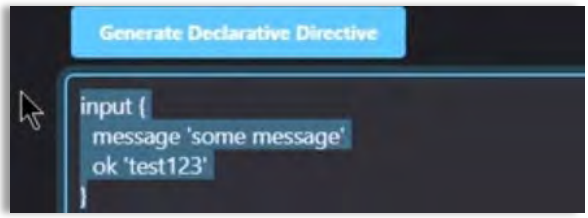
(في النص قبل الـ stage السابقة وبعد stage('k8s – Raise PR')):

```
stage('K8S - Raise PR') {...
}

stage('App Deployed?') {
    when {
        branch 'PR*'
    }
    steps {
        timeout(time: 1, unit: 'DAYS') {
            input message: 'Is the PR Merged and ArgoCD Synced?', ok: 'YES! PR is Merged and ArgoCD Appli
        }
    }
}

stage('DAST - OWASP ZAP') {...
}
```

➢ لاحظ ان السطر بتاع الـ input message ده احنا ولدناه من :

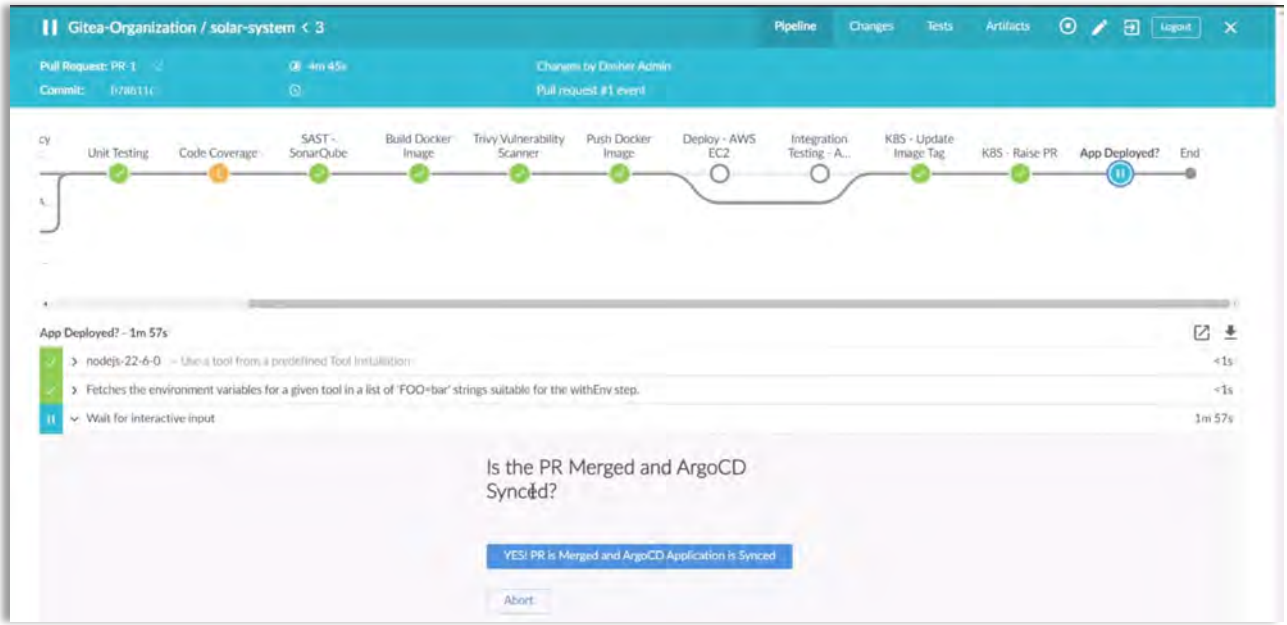
Pipeline syntax → Declarative Directive Generator → sample steps → input: Input

- Message: some message
- In Advanced → ok Button Caption: test123



📌 الهدف:

- الـ Jenkins بيوقف عند المرحلة دي ويطلب موافقة من المستخدم . (بيستني المستخدم يديله التأكيد ان التطبيق شغال وتمام)
- المستخدم لازم يضغط "Yes" بعد ما يتأكد إن الـ ArgoCD تم مزامنته (sync)
- لو ما وافقش خلال يوم ← الـ build بيتوقف تلقائيًا.



💡 نقطة مهمة:

في الـ real world ممكن تضيف Step ترفع التقارير دي كـ Jenkins artifacts :

- archiveArtifacts artifacts: '*.html, *.json, *.xml'

وده بيخلي التقارير محفوظة في Jenkins Job نفسه.

⚙️ الـ Flow الكامل للـ Pipeline

1. Developer يرفع كود جديد ويعمل Pull Request
2. Jenkins يبدأ الـ pipeline ويبني Docker image جديدة.
3. Jenkins يحدّث ملفات الـ manifest ويعمل PR في GitHub
4. المستخدم يراجع الـ PR ويدمجه.
5. ArgoCD يلاحظ التغيير ويعمل Sync للتطبيق على K8s
6. Jenkins يوقف عند المرحلة "App Deployed?"
7. المستخدم يضغط "Yes" بعد التأكد إن التطبيق شغال.
8. Jenkins يبدأ مرحلة DAST OWASP ZAP
9. ZAP يفحص الـ APIs ويولّد تقارير الأمان.
10. Jenkins يخزن التقارير كـ artifacts

🧩 السبب المنطقي (Why Manual Approval Before Scan?)

الـ Security Scan (زي OWASP ZAP) أو حتى الـ post deployment tests بيحتاجوا إن التطبيق يكون فعليًا شغال وسليم بعد عملية الـ deployment لكن أحيانًا :

- الـ deployment ممكن يفشل بصمت (يعني Jenkins ما يعرفش إن التطبيق واقع أو الـ pod مش جاهز)
- أو ممكن الـ app يشتغل جزئيًا (service شغالة لكن UI مش ظاهر)
- أو الـ deployment يتم بنجاح، لكن الـ environment نفسه (cluster أو ingress) فيه مشكلة.
- فلو Jenkins دخل على طول في مرحلة الـ scanning قبل التأكد، الـ scan هيفشل بنسبة كبيرة لأن الـ app مش جاهز فعليًا ⛔

Best Practices 🧠

1. ✅ شغّل ZAP على Staging مش Production
لأن الفحص النشط ممكن يسبب load أو تغييرات غير مقصودة على البيانات.
2. ✅ خلي التقارير جزء من الـ Pipeline Artifacts
عشان تقدر تراجعها من Jenkins UI بسهولة.
3. ✅ استخدم ZAP في وضع Automation داخل Container
عشان تضمن reproducibility ومفيش manual setup
4. ✅ حلل النتائج في Jenkins أو أداة خارجية (مثل DefectDojo)
عشان تطلع Dashboard حقيقية لحالة الأمان.
5. ✅ دمج DAST مع CI/CD مبكّرًا
عشان تكتشف الثغرات بدري بدل ما توصل للـ Production

📄 ملخص السكريبت بالكامل:

- السكريبت أضاف مرحلتين رئيسيتين في Jenkins Pipeline :
  1. App Deployed? ← تطلب تأكيد من المستخدم إن التطبيق تم نشره.
  2. DAST OWASP ZAP ← تشغّل فحص أمان ديناميكي على الـ API
- الفحص بيشتغل عن طريق Docker Container فيه OWASP ZAP ، والنتائج بتتحفظ في ملفات HTML / JSON / XML.
- الفكرة كلها إنك تتأكد إن كل إصدار جديد من التطبيق بيتفحص أمنيًا أوتوماتيك في Jenkins بعد نشره.

DAST Ignore Rules

🎯 أولاً: الفكرة العامة

- ❑ في الجزء السابق كنا عملنا DAST Scan باستخدام OWASP ZAP على التطبيق بعد نشره في Kubernetes ، لكن الفحص فشل لأن ZAP رجّع تحذيرات (Warnings) معينة، مثلاً:
  "Unexpected content type"
- ❑ التحذيرات دي ممكن تكون minor ، مش بالضرورة vulnerability حقيقية،
- ❑ فـ في بعض الحالات — خصوصًا في التدريب أو الـ demo environment ← ممكن تتجاهلها عشان الفحص يكمل.
- ❑ اهو الجزء ده بقى بيعلّمنا إزاي نعمل الكلام ده:
  - نستخدم config file نحدد فيه القواعد اللي عايزين نتجاهلها.
  - ونربطه بأمر Docker الخاص بـ ZAP Scan
  - وفي الآخر نضيف مرحلة جديدة عشان ننشر تقرير الـ HTML الناتج عن الفحص داخل Jenkins UI

🧱 ثانياً: شرح تفصيلي خطوة بخطوة

1. 🧩 تحديد المشكلة

- ➢ ZAP كان بيوقف الـ pipeline بسبب warning من نوع unexpected content type
- ➢ المفروض كـ مطور تصلح المشكلة دي في الكود (ده الحل الصحيح في الواقع).
  - ▪ لكن عشان ده تدريب، هنتعلم إزاي نتجاهل التحذير ده مؤقتًا.

---

2. 🧩 إعداد ملف التجاهل (Ignore File)

ZAP – ZAP - API Scan

- ➢ ZAP يسمحلك تعمل config file اسمه مثلًا: zap_ignore_rules
- ➢ الملف ده بيحتوي على قائمة بالقواعد (rules) اللي عايز تتجاهلها، وكل سطر فيه بيتكتب بالشكل ده:

<ID> ignore <endpoint>

- o لاحظ إن كل حاجة مفصولة بـ tab (مش space)

📘 مثال عملي:

- لو الـ ID بتاع التحذير هو 90033 والـ endpoint هو /api/v1/users
  يبقى السطر هيكون كده: 90033 ignore /api/v1/users

📍 لو عايز توقف البيلد لو القاعدة دي حصلت، تقدر بدل ignore تكتب fail

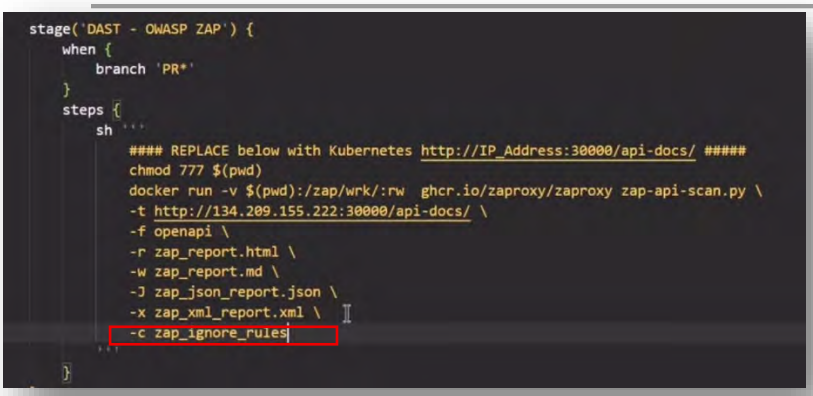
---

3. 🧩 إنشاء الملف فعليًا

- بنروح على المشروع وننشئ الملف : vi zap_ignore_rules
- ونكتب جوّاه القواعد اللي عايزين نتجاهلها.
- بعد ما نخلص بنحفظ الملف (:wq)

---

4. 🧩 تعديل Jenkinsfile لإضافة الملف في أمر Docker



```
stage('DAST - OWASP ZAP') {
    when {
        branch 'PR*'
    }
    steps {
        sh '''
            #### REPLACE below with Kubernetes http://IP_Address:30000/api-docs/ #####
            chmod 777 $(pwd)
            docker run -v $(pwd):/zap/wrk/:rw  ghcr.io/zaproxy/zaproxy zap-api-scan.py \
            -t http://134.209.155.222:30000/api-docs/ \
            -f openapi \
            -r zap_report.html \
            -w zap_report.md \
            -J zap_json_report.json \
            -x zap_xml_report.xml \
            -c zap_ignore_rules
        '''
    }
```

- في سكريبت الـ Jenkins اللي بيشغل ZAP ،بنضيف الخيار -c أو --config علشان نمرر ملف التجاهل:

📘 كده الـ Docker container هيقرأ الملف اللي فيه القواعد اللي هيتجاهلها أثناء الفحص.

5. 🧩 إضافة مرحلة نشر تقرير HTML في Jenkins

- بعد ما الفحص يخلص، بنستخدم plugin في Jenkins اسمه **Publish HTML Reports** عشان نعرض التقرير جوه واجهة Jenkins نفسها . (مشروح قبل كدا في جزئية الـ Trivy)
- بنضيف الكود ده في نهاية الـ pipeline داخل قسم post :

```
post {
    always {
        script {
            if (fileExists('solar-system-gitops-argocd')) {
                sh 'rm -rf solar-system-gitops-argocd'
            }
        }

        junit allowEmptyResults: true, stdioRetention: '', testResults: 'test-results.xml'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'dependency-check-junit.xml'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'trivy-image-CRITICAL-results.xml'
        junit allowEmptyResults: true, stdioRetention: '', testResults: 'trivy-image-MEDIUM-results.xml'

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
        reportFiles: 'zap_report.html', reportName: 'DAST - OWASP ZAP Report', reportTitles: '',
        useWrapperFileDirectly: true])

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
        reportFiles: 'trivy-image-CRITICAL-results.html', reportName: 'Trivy Image Critical Vul Report',
        reportTitles: '', useWrapperFileDirectly: true])

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
        reportFiles: 'trivy-image-MEDIUM-results.html', reportName: 'Trivy Image Medium Vul Report',
        reportTitles: '', useWrapperFileDirectly: true])

        publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
```

📌 الشرح:

- reportDir: '.' ← مكان التقرير في نفس مجلد الـ workspace
- reportFiles: 'report.html' ← اسم ملف التقرير اللي تولد من ZAP
- reportName ← الاسم اللي هيظهر في Jenkins UI

---

6. 🧩 تنفيذ بعض التغييرات في التطبيق

- هنا عملنا تعديل بسيط في واجهة التطبيق (frontend) بس للتوضيح البصري:
  - أضاف صواريخ إضافية في واجهة الـ Solar System علشان يشوف التحديث في الواجهة بعد النشر.
- ده تم من خلال تعديل ملف index.html وإضافة الأكواد الخاصة بالـ rocket قبل وبعد عنصر الـ solar system

---

7. 🧩 رفع التغييرات وتشغيل الـ pipeline

بعد ما خلص التعديلات:

- عمل git add .
- بعدين git commit -m "Added ZAP ignore rules and modified frontend"
- وأخيرًا git push.

ده بيخلي Jenkins يكتشف التغيير ويبدأ build جديدة.

---

8. 🧩 مراجعة خطوات Jenkins بعد التعديل

1. Jenkins بدأ pipeline جديدة.
2. وقف في مرحلة "App Deployed?" زي ما شرحنا قبل كده.
3. المطور دمج الـ PR في GitHub
4. ArgoCD لاحظ التغيير وعمل Sync للتطبيق.
5. اتأكد إن التطبيق نزل عليه النسخة الجديدة (اتغير شكل الواجهة).
6. رجع Jenkins وضغط "Yes, PR merged and synced"

---

9. 🧩 فشل مؤقت بسبب خطأ في تنسيق الملف

- في أول محاولة حصل Error :

unexpected number of tokens on the line. There should be at least three.

- ده معناه إن الملف zap_ignore_rules مكتوب بطريقة غلط — غالبًا المسافات بدل tabs حيث أن :

ZAP بيحتاج tab واحد بين كل عمود.

👨‍🔧 الحل:

- يفتح الملف باستخدام vi بدل VS Code لأن VS Code بيبدّل الـ tab بمسافات.
- ثم يكتب السطر بشكل صحيح، يحفظ، ويعمل commit & push من جديد.

10. 🧩 نجاح الفحص بعد التصحيح

بعد التصليح:

- Jenkins أعاد تشغيل الـ pipeline
- Docker command بتاع ZAP اشتغل بنجاح.
- الفحص خلص بدون ما يقف.
- الـ warning اللي كنا متجاهلينه ظهر كـ "ignored" مش "failed"

📄 في اللوجات شاف:

pass: 112

ignored: 1

يعني كل حاجة عدت عادي، والتحذير اتعمله ignore بنجاح.

---

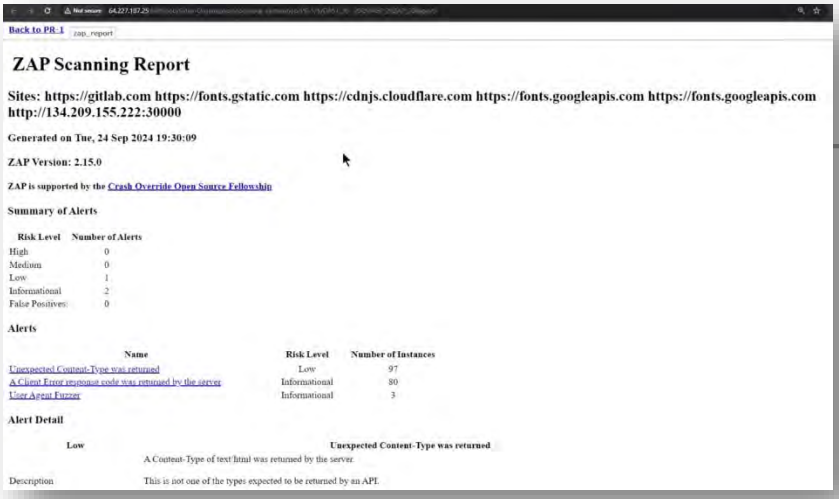
11. 🧩 استعراض التقرير

فتح تقرير report.html اللي تولد بعد الفحص.

📊 التقرير بيعرض:

- عدد الـ alerts المكتشفة.
- مستوى الخطورة (Low, Info, High)
- وصف كل alert بالتفصيل.
- عدد الـ endpoints المتأثرة.

حتى لو ملف CSS مش ظاهر (شكل التقرير بسيط)، المعلومات كلها موجودة.



Back to PR-1 zap_report

ZAP Scanning Report

Sites: https://gitlab.com https://fonts.gstatic.com https://cdnjs.cloudflare.com https://fonts.googleapis.com https://fonts.googleapis.com http://134.209.155.222:30000

Generated on Tue, 24 Sep 2024 19:30:09

ZAP Version: 2.15.0

ZAP is supported by the Crash Override Open Source Fellowship

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	0
Low	1
Informational	2
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
Unexpected Content-Type was returned	Low	97
A Client Error response code was returned by the server	Informational	80
User Agent Fuzzer	Informational	3

Alert Detail

Low — Unexpected Content-Type was returned

Description: A Content-Type of text/html was returned by the server. This is not one of the types expected to be returned by an API.

⚙️ ملاحظات وBest Practices

1. ✅ التجاهل لازم يكون مؤقت ومبرر
   استخدم ignore فقط في حالات التحذيرات الغير مؤثرة — مش عشان تتخطى مشاكل حقيقية.
2. ✅ اكتب السبب في الكومنت أو ملف توثيق
   عشان الفريق يعرف ليه فيه rule متعملها ignore
3. ✅ استخدم VI أو محرر بيحافظ على Tabs حقيقية
   لأن ZAP حساس جدًا لتنسيق الملف.
4. ✅ راجع التقارير دوريًا
   حتى التحذيرات الضعيفة ممكن تبقى بداية لثغرات مستقبلية.
5. ✅ استمر في تشغيل ZAP داخل Container
   ده بيسهّل التكرار ويمنع مشاكل بيئة التشغيل.
6. ✅ ادمج HTML Report مع Jenkins Dashboard
   بحيث الفريق الأمني يقدر يشوف النتائج بسهولة من Jenkins UI

📃 الملخص العام

الجزء ده علّمنا إزاي:

- نعمل config file نتحكم من خلاله في سلوك ZAP ونحدد التحذيرات اللي نتجاهلها.
- نمرّر الملف ده في أمر Docker أثناء تشغيل الفحص.
- نعرض تقرير HTML داخل Jenkins .. ونتعامل مع أخطاء التنسيق (tabs vs spaces)

وبالتالي ← Pipeline أمان ديناميكي متكامل في Jenkins ، بيعمل فحص OWASP ZAP ، وبيقدر يتعامل بمرونة مع التحذيرات بدون ما يوقف عملية الـ CI/CD

Publish Reports to AWS S3

🎯 الفكرة العامة والهدف

- ❑ الهدف هنا هو رفع كل التقارير (Reports) اللي بيولدها الـ Jenkins Pipeline بعد عملية الـ Build والاختبارات إلى Amazon S3 Bucket عشان نقدر نخزنها بشكل مركزي وآمن، ونرجع لها في أي وقت.
- ❑ وده بيدخل ضمن فكرة CI/CD + Reporting Automation، بحيث Jenkins مش بس يبني الكود ويعمل اختبارات، لكنه كمان يحتفظ بنتائج كل Build في الـ Cloud

---

🧱 الشرح التفصيلي

المرحلة الأولى : تحديد الملفات اللي هنرفعها

في البداية، بيورينا إن بعد ما البايبلاين خلص، عندنا مجموعة تقارير تم توليدها:

- Coverage reports داخل فولدر اسمه /coverage
- Dependency check reports زي ملفات الـ XML أو HTML الخاصة بتحليل المكتبات
- Unit test reports زي test-results.xml
- Trivy reports الخاصة بفحص الـ vulnerabilities
- ZAP reports الخاصة باختبار أمان التطبيق (DAST)

كل الملفات دي موجودة داخل الـ Workspace بتاع Jenkins

💡 معلومة مهمة:

Jenkins workspace هو المجلد اللي بيتخزن فيه الكود، ونتائج الـ build والاختبارات الخاصة بكل job أو pipeline

---

المرحلة الثانية : إعداد الـ S3 Bucket

- ➢ هايعمل Bucket جديد في AWS S3 وسماه : **solar-system-jenkins-reports-bucket**
- ➢ والموقع الجغرافي ← (region): us-east-2
  حالياً الباكت فاضية — ودي اللي إحنا هنرفعلها التقارير بعد شوية.

المرحلة الثالثة: إعداد صلاحيات Jenkins للوصول لـ AWS

- ➢ عشان Jenkins يعرف يتعامل مع AWS ، لازم يكون فيه IAM User عنده صلاحيات:
  - S3 Full Access
  - EC2 Access
  - Lambda Access
- ➢ وتم إنشاء المستخدم باسم : **jenkins-user**
- ➢ ثم تم تخزين بياناته (Access Key + Secret Key) داخل Jenkins من خلال:
  Manage Jenkins → Credentials
  باسم:
  AWS S3 EC2 Lambda creds

⚙️ نقطة عملية:

- ➢ الأفضل تخزن الـ credentials جوه Jenkins كـ "Secret Text" أو "AWS Credentials" مش في الكود مباشرة — لحماية البيانات.

المرحلة الرابعة: تثبيت البلجن الخاصة بـ AWS

تم استخدام Plugin اسمه : **Pipeline: AWS Steps**

وده بيوفر مجموعة Steps جاهزة زي:

- s3Upload
- s3Download
- s3Delete
- s3FindFiles

وده اللي بنستفيد منه هنا لرفع الملفات إلى الـ S3

المرحلة الخامسة : تجهيز الملفات اللي هترفع

- بما ان في الـ workspace بتاعتنا فيه فايلات كتيرة (عبارة عن تقارير مختلفة) .. فالفكرة بقا هي انك ازاي تحدد الفايل اللي هاتعمله upload في الـ S3 ؟!؟!؟!
- الحل : في Jenkinsfile ، بعد آخر Stage (اللي كانت DAST OWASP ZAP) بيضيف Stage جديدة اسمها :

stage('upload – Aws S3')

```
stage('DAST - OWASP ZAP') {...
}

stage('Upload - AWS S3') {
    when {
        branch 'PR*'
    }
    steps {
        sh '''
            ls -ltr
            mkdir reports-$BUILD_ID
            cp -rf coverage/ reports-$BUILD_ID/
            cp dependency*.* test-results.xml trivy*.* zap*.* reports-$BUILD_ID/
            ls -ltr reports-$BUILD_ID/
        '''
    }
}
```

📘 شرح الأوامر هنا :

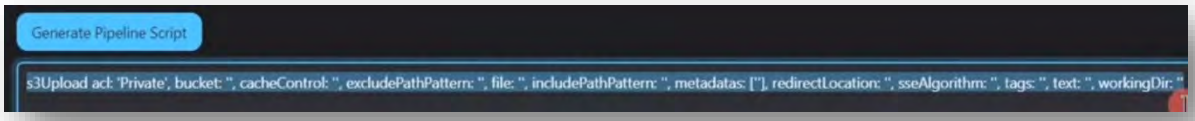
1. ls ← بيعرض محتويات workspace
2. mkdir reports-${BUILD_ID} ↓
   بيعمل فولدر جديد باسم التقرير الخاص بالـ Build ده.
   مثال : reports-6 لو الـ build رقمها 6.
3. cp ← بينسخ الملفات المطلوبة إلى الفولدر الجديد.
   - coverage/*
   - dependency*
   - test-results.xml
   - trivy*
   - zap*
4. ls reports-${BUILD_ID} ← عشان يتأكد إن النسخ تم فعلاً.

💡 نصيحة من أرض الواقع:

استخدم دائمًا ${BUILD_ID} أو ${BUILD_NUMBER} في اسم الفولدر عشان تتجنب تداخل التقارير بين الـ builds المختلفة.

المرحلة السادسة : رفع التقارير إلى S3

- بعد ما جهّز الفولدر، بيروح علي : pipeline sytax → snippet generator → sample step → s3Upload:copy file to S3
  - وبيعرفك انك ممكن تدخله arguments اللي عاوزها وفي الاخر يولدلك الكود (بس هو مش هايشتغل بالطريقة دي)



- الحل الثاني (اسهل) ← وهو انه يروح عالـ Docs وينسخ الكود علطول ويغير فيه حسب الـ use case اللي عنده

Pipeline: AWS Steps | Jenkins plugin

- وبعدين بيروح عالـ Jenkinsfile ويعمل paste علطول في نفس stage('Upload – Aws S3')

```
    ls -ltr reports-$BUILD_ID/
'''
s3Upload(
    file:'file.txt',
    bucket:'my-bucket',
    path:'path/to/target/file.txt'
)
```

```
    ls -ltr reports-$BUILD_ID/
'''
s3Upload(
    file:"reports-$BUILD_ID",
    bucket:'solar-system-jenkins-reports-bucket',
    path:"jenkins-$BUILD_ID"
)
```

- ده بيرفع الفولدر بالكامل إلى المسار : s3://solar-system-jenkins-reports-bucket/jenkins-6/
- وكل Build جديدة بتتحط في فولدر منفصل.

عملناها قبل كدا في
stage('Integration Testing – AWS EC2')

المرحلة السابعة: المصادقة مع AWS

- قبل تنفيذ Step الرفع، لازم Jenkins يعرف بيانات الدخول (credentials) .. وبالتالي هانضيف الكود دا في الـ steps بتاع نفس الـ stage :

```
stage('Upload - AWS S3') {
    when {
        branch 'PR*'
    }
    steps {
        withAWS(credentials: 'aws-s3-ec2-lambda-creds', region: 'us-east-2') {
            sh '''
                ls -ltr
                mkdir reports-$BUILD_ID
```

- الـ withAWS دي هي اللي بتربط Jenkins مؤقتًا بحساب AWS باستخدام الـ credentials المخزنة مسبقًا.

Explained in p.210

المرحلة الثامنة: معالجة مشكلة المتغيرات (String Interpolation)

- لما تستخدم متغيرات زي ${BUILD_ID} داخل Jenkinsfile (اللي مكتوب بلغة Groovy)
  - لازم تكتب السطر بين " " double quotes مش ' ' single
- ليه؟
  - لأن Groovy مش بيستبدل المتغيرات داخل single quotes

✅ مثال صحيح:

file: "reports-${BUILD_ID}"

❌ مثال خاطئ:

file: 'reports-${BUILD_ID}'

الـ jenkinsfile النهائي

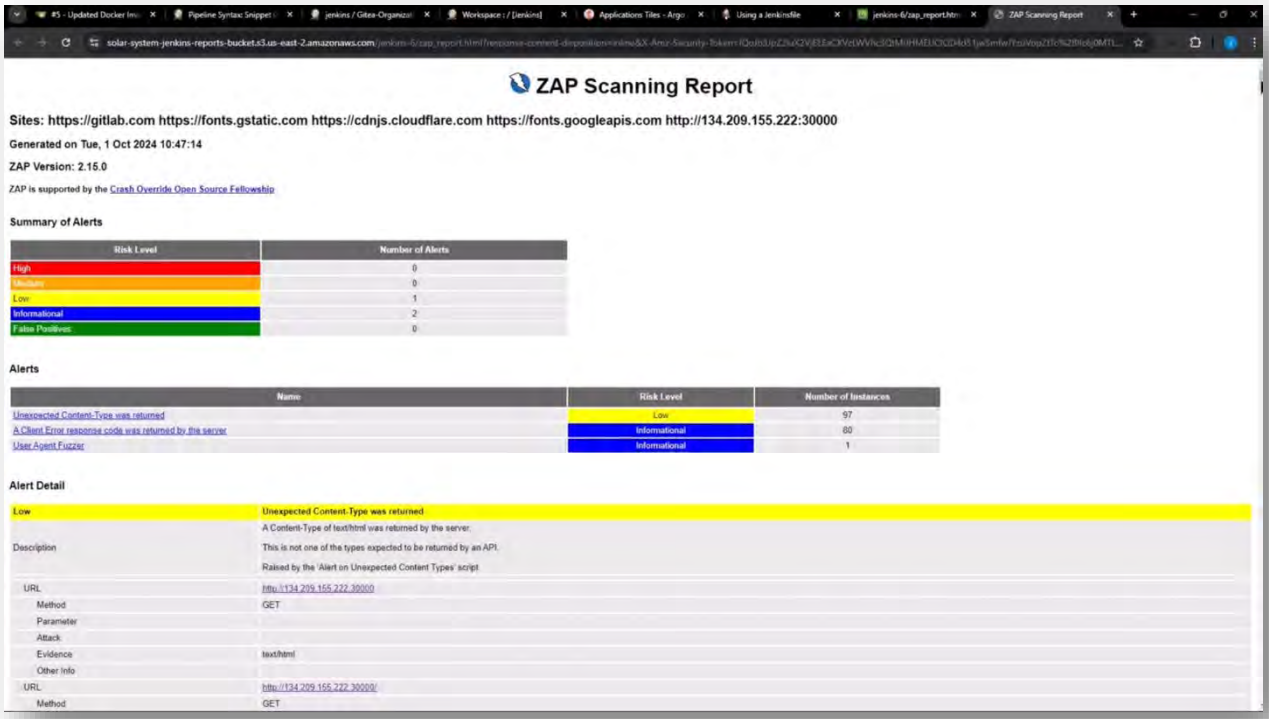
```
stage('Upload - AWS S3') {
    when {
        branch 'PR*'
    }
    steps {
        withAWS(credentials: 'aws-s3-ec2-lambda-creds', region: 'us-east-2') {
            sh '''
                ls -ltr
                mkdir reports-$BUILD_ID
                cp -rf coverage/ reports-$BUILD_ID/
                cp dependency*.* test-results.xml trivy*.* zap*.* reports-$BUILD_ID/
                ls -ltr reports-$BUILD_ID/
            '''
            s3Upload(
                file:"reports-$BUILD_ID",
                bucket:'solar-system-jenkins-reports-bucket',
                path:"jenkins-$BUILD_ID"
            )
        }
    }
}
```

المرحلة التاسعة: اختبار النتيجة

بعد تنفيذ الـ Pipeline :

- هتلاحظ إن كل الـ Stages خلصت بنجاح.
- Stage AWS S3 Upload اشتغلت ورفعت الملفات.
- في AWS Console → S3 ← Refresh ← هتلاقي فولدر جديد زي jenkins-6 فيه كل التقارير.

ممكن تفتح مثلًا تقرير ZAP-report.html وتشوف النتيجة مباشرة في المتصفح.



ZAP Scanning Report

Sites: https://gitlab.com https://fonts.gstatic.com https://cdnjs.cloudflare.com https://fonts.googleapis.com http://134.209.155.222:30000

Generated on Tue, 1 Oct 2024 10:47:14

ZAP Version: 2.15.0

ZAP is supported by the Crash Override Open Source Fellowship

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	0
Low	1
Informational	2
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
Unexpected Content-Type was returned	Low	97
A Client Error response code was returned by the server	Informational	80
User Agent Fuzzer	Informational	1

Alert Detail

Low	Unexpected Content-Type was returned
Description	A Content-Type of text/html was returned by the server. This is not one of the types expected to be returned by an API. Raised by the 'Alert on Unexpected Content Types' script
URL	http://134.209.155.222:30000
Method	GET
Parameter	
Attack	
Evidence	text/html
Other Info	
URL	http://134.209.155.222:30000/
Method	GET

Best Practices

1. ✅ خزن الـ AWS credentials جوه Jenkins بشكل آمن (Secret Text / IAM Role)
2. 📁 استخدم ${BUILD_ID} أو ${BUILD_NUMBER} لتقسيم التقارير لكل Build
3. 🌶️ استخدم Regex في أوامر الـ cp لتحديد الملفات بدقة.
4. 🔒 تأكد إن الـ IAM User permissions محدودة فقط بما تحتاجه (Least Privilege)
5. 🧰 استخدم withAWS لكل تعامل مع AWS لتفادي أخطاء المصادقة.
6. 🗑️ فعل Lifecycle Policy في الـ S3 لحذف التقارير القديمة أو أرشفتها.

الملخص العام

الجزء ده بيضيف مرحلة جديدة في Jenkins pipeline مسؤولة عن:

1. تجميع كل تقارير الـ build في مجلد واحد.
2. رفعها إلى AWS S3 bucket باستخدام plugin مخصص.
3. استخدام متغيرات Jenkins و AWS credentials بأمان.
4. تنظيم التقارير لكل Build على حدة.

النتيجة النهائية:

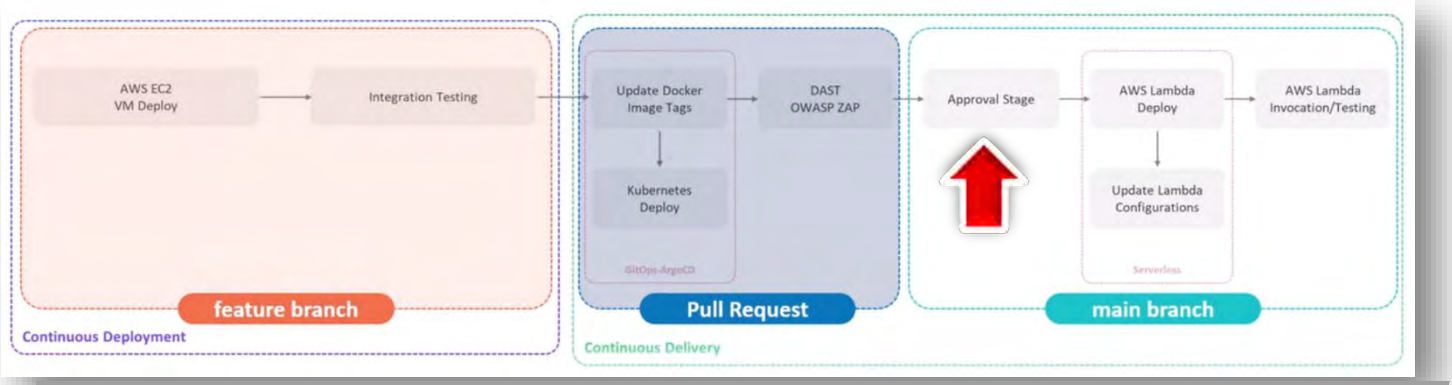
كل build من Jenkins أصبح ليه نسخة محفوظة من تقاريره في S3 ، وده يسهل عملية المتابعة والمراجعة من أي مكان، ويوفر Backup ممتاز لنتائج الاختبارات.

Deploy to Prod

🎯 أولًا: الفكرة العامة والهدف

- ❑ الفكرة هنا إننا بنوصل لآخر مرحلة في خط سير الـ CI/CD Pipeline :
  📦 نشر التطبيق على بيئة الإنتاج (Production)
- ❑ قبل ما التطبيق يتنشر في الـ Production ، لازم نمر بمرحلة approval .. يعني حد من الـ Admins لازم يراجع التغييرات ويوافق يدويًا على تنفيذ عملية النشر.
- ❑ المرحلة دي اسمها في Jenkins : Input Stage (manual approval step)
- ❑ فباختصار:

1. التطبيق اتبنى (Build)
2. اتعمل له اختبار (Integration + Security)
3. اتنشر على Kubernetes للاختبار.
4. وخلصنا كل الفحوصات (ZAP, Trivy, etc)
5. دلوقتي جاهزين نرفعه إلى AWS Lambda (Production)
6. بس قبل دا، فيه موافقة يدوية (Manual Approval) من Admin فقط.



🧱 ثانيًا: الشرح التفصيلي

1. 🟩 إضافة مرحلة "Deploy to Production"

- ➢ بنضيف Stage جديدة في Jenkinsfile بعد مرحلة رفع التقارير إلى AWS S3
- ➢ المرحلة دي هدفها تنفيذ نشر التطبيق إلى Production بعد ما الـ Admin يوافق.

```
stage('Upload - AWS S3') {···
}

stage('Deploy to Prod?') {
    when {
        branch 'main'
    }
    steps {
        timeout(time: 1, unit: 'DAYS') {
            input message: 'Deploy to Production?', ok: 'YES! Let us try this on Production', submitter:
                'admin'
        }
    }
```

تذكر ان الـ stage دي نفس الخطوات والفكرة بتاع stage('App Deployed') اللي عملناها في p.235

📘 الشرح:

- stage('Deploy to Prod') ← بنسمي المرحلة الجديدة "Deploy to Prod"
- when { branch 'main' } ← المرحلة دي تشتغل فقط لو إحنا على فرع main (الفرع الأساسي في Git)
- input message: ← هنا Jenkins هيوقف التنفيذ ويطلب موافقة المستخدم. .. الزرار اللي بيظهر اسمه Yes, Let us try this on Production

Explained in p.236

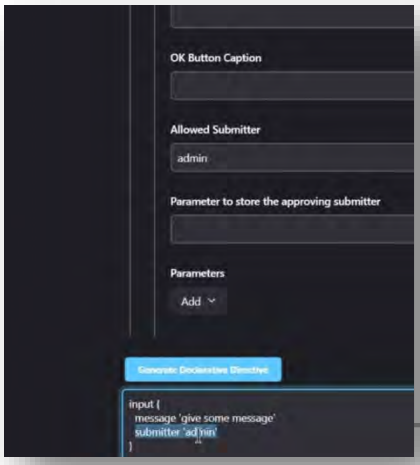
- بنضيف timeout للمرحلة، بحيث لو مفيش موافقة خلال فترة معينة، Jenkins يوقف الـ Pipeline تلقائيًا.
  ⏳ المدة هنا يوم واحد (1 Day) ← يعني لو محدش وافق خلال يوم، الـ Pipeline هيفشل تلقائيًا

3. 🟩 تقييد الموافقة لمجموعة معينة (Admins Only)

- افتراضيًا، أي مستخدم عنده صلاحية في Jenkins ممكن يضغط زر الموافقة .. لكن في الـ Production ، دا خطر جدًا.
- فنستخدم خيار allowed submitters لتحديد المستخدمين أو الجروب المسموح لهم بالموافقة.

input message: 'Deploy to production?', ok: 'Yes, deploy now', submitter: 'admin'

🔒 كده بس المستخدمين اللي في Group = admin يقدروا يضغطوا على الزر ويكملوا الـ Pipeline



للتذكير : 🧠 ازاي نجيب الصيغة دي؟

- لو مش متأكد من الكود، تقدر تستخدم : Jenkins Snippet Generator:

1. افتح Jenkins
2. روح إلى Pipeline Syntax
3. اختار Declarative Directive Generator ← Input
4. حدد كل الـ options (message, ok, submitter)
5. Jenkins هيولّدلك الكود الجاهز.

---

4. 🟩 اختبار عمل الـ Stage

- بعد ما نضيف الكود في الـ feature branch ونعمل commit ، Jenkins هيبدأ Build جديد.
- بس لأننا لسه في feature branch ، شرط when { branch 'main' } هيخلي المرحلة يتعملها Skip
- فهنوقف الـ Build يدويًا ونكمّل باقي الخطوات لما نعمل merge للـ main branch

---

5. 🟩 عمل Merge إلى Main Branch

بنروح إلى الـ GitHub Repository :

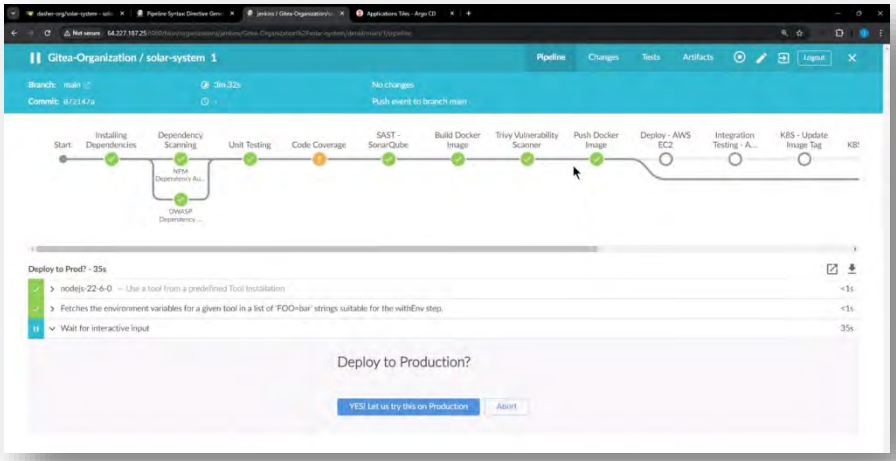
- نفتح الـ Pull Request
- نعمل Merge إلى main
- بمجرد الدمج، Jenkins هيلاحظ إن في تغييرات جديدة على الـ main branch
- وده هيشغل Build جديد مخصوص للـ main

---

6. 🟩 تشغيل الـ Pipeline على الـ Main Branch

لما Jenkins يبدأ البايبلاين الجديد:

- هينفذ مراحل الـ Build الأساسية.
- وهيتخطّى المراحل الخاصة بالـ Feature أو PR
- لحد ما يوصل إلى مرحلة Deploy to Prod
- وهنا هيقف ويعرض شاشة فيها زر الموافقة (Yes/Abort)



Remember: (Explained in Jenkins Authentication & Authorization Strategy Lessons)
احنا بنعرف الـ users منتمين لأنهي groups من خلال :
Mamage Jenkins → Security → Mock Security Realm

7. تجربة الـ Approval Stage

✅ أولًا : User من نوع Admin
لو المستخدم الحالي عضو في مجموعة admin :
- هيقدر يضغط على الزر Yes
- المرحلة هتتكمل، والـ Pipeline هيكمل نشر التطبيق.

❌ ثانيًا : User عادي (Developer)
لو المستخدم من مجموعة تانية (زي developer group)
- هيظهرله خطأ : You need to be an admin to submit this

وده اللي فعله المحاضر لما جرّب الدخول بمستخدم تاني اسمه Ema من الـ developer group ، ولما حاول يوافق، ظهرله التحذير ده.

---

8. تأكيد النتيجة

بعد الموافقة من الـ Admin :
- Jenkins هيكمل الـ Stage بنجاح.
- ولو مفيش مراحل بعدها (زي Lambda deployment) الـ Pipeline هيخلص.

لكن الخطوة الجاية في الكورس — واللي لسه هنشوفها في الشرح الجاي — هي نشر التطبيق فعليًا في AWS Lambda بعد الموافقة.

⚙️ ملاحظات إضافية (Real Use Cases - Tricks)

1. استخدم الـ approval stage لأي خطوة خطيرة (زي production deployment أو database migration)
2. دايمًا حدّد الـ submitter ، ومتسيبهاش مفتوحة لكل المستخدمين.
3. استخدم timeout عشان تمنع الـ Pipeline من الانتظار إلى مالانهاية.
4. ممكن تضيف message توضيحي للـ input ، مثلاً:
   "Please review the changes before deploying to production."
5. لو عندك stages مختلفة تتطلب approvals متعددة (security / QA / admin) ،
   ممكن تضيف stages متتالية بـ submitters مختلفين.

🧠 Best Practices

✅ استخدم manual approval فقط في مراحل الـ Production
✅ اربط كل user بجروب واضح (admin, developer, tester)
✅ خزن الـ permissions في Jenkins بطريقة منظمة (Matrix-based Security)
✅ فعّل audit logs لمعرفة مين وافق على النشر.
✅ استخدم descriptive messages عشان كل approval يبقى مفهوم.

🏁 الملخص النهائي

الجزء دا أضاف Manual Approval Stage في Jenkins Pipeline علشان:
1. يمنع النشر التلقائي على الـ Production بدون إذن.
2. يخلّي بس الـ Admins هما اللي يقدروا يوافقوا على العملية.
3. يزود طبقة أمان وتحكم في خط سير النشر (Release Control)
4. يتأكد إن الـ Pipeline يشتغل تلقائيًا على main branch بعد الدمج.
5. يمهّد للمرحلة اللي بعدها — نشر التطبيق في AWS Lambda

AWS Lambda Basics

🧠 الفكرة العامة

❑ الجزء ده مقدمة تمهيدية لفهم AWS Lambda، وبيشرحها بطريقة قصة علشان تبين الفكرة:

- إزاي Lambda خلتنا نشغل الكود بتاعنا من غير ما نحجز أو نجهز سيرفرات.
- بمعنى أبسط: AWS Lambda = Serverless Compute Service
  - بتخليك تشغل الكود فقط، وأمازون تتولى كل حاجة تخص السيرفر والبنية التحتية.

🔍 الشرح التفصيلي

➢ المطور زمان كان لازم يجهز سيرفر فعلي علشان يشغل التطبيق بتاعه — سواء في داتا سنتر أو على VM ، وده بياخد وقت ومجهود وصيانة.

➢ لكن بعد كده ظهر الحل السحري : AWS Lambda ⚡ اللي بدورها قالتلك ← "هات الكود بتاعك، وأنا هشغله من غير ما تشيل هم السيرفر".

- يعني:
  - مش هتشتري ولا تجهز ولا تصين سيرفر.
  - Lambda بتتعامل مع كل ده أوتوماتيك.
  - أنت تركز على الكود فقط.

🧩 وده معناه إن Lambda بتختصر دورة نشر الكود من أيام وساعات إلى ثواني.

⚙️ إيه هي AWS Lambda فعليًا؟

➢ Lambda هي خدمة Serverless Compute من AWS
بمعنى إنها بتنفذ كودك استجابةً لحدث معين (event) ، من غير ما تكون مسؤول عن أي سيرفر.

🔹 الأحداث (Triggers) اللي ممكن تشغل Lambda :

- رفع ملف على S3 Bucket
- تحديث في DynamoDB
- HTTP Request عبر API Gateway
- CloudWatch Event أو Alarm

🔹 الوظيفة : Lambda تشغل الكود، AWS توفر السيرفرات، تدير الـ scaling ، وتطفيها لما ماتحتجهاش.

🧰 المميزات العملية (Features & Advantages)

1️⃣ Auto Scaling

بتتعامل تلقائيًا مع عدد المستخدمين.

- لو 10 مستخدمين فقط ← Lambda بتشغل 10 Instances بس.
- لو مليون مستخدم ← بتوسع لآلاف الـ Executions في نفس اللحظة.
- لو مفيش حد بيستخدم ← بتوقف (scale to zero) ومش بتدفع حاجة.

2️⃣ Pay-as-you-go

➢ تدفع بس على الوقت اللي الكود بيشتغل فيه.

يعني:

- لو كودك اشتغل ثانية واحدة = هتدفع تمن الثانية دي بس. .. مش زي السيرفر اللي بيفضل شغال حتى لو مفيش ترافيك.

3️⃣ Multi-language Support

- Lambda بتدعم لغات برمجة كتير جدًا منها:
  - Python - Node.js – Java – C# - Go – Ruby - PowerShell
- وكمان لو لغتك مش مدعومة رسميًا،
  AWS بتوفر Runtime API Extension تقدر من خلاله تشغل أي لغة تانية— حتى لغات قديمة زي COBOL!
  📘 AWS عندها أمثلة في مدونتها بتوضح تشغيل كود COBOL على Lambda

---

💡 المغزى العملي

لو عندك كود جاهز بلغة مدعومة، مش محتاج غير ترفعه على Lambda .. وهي تتولى الباقي:

- بتشغله لما يحصل Event
- بتقيس الـ Logs والـ Metrics تلقائيًا
- وبتوقفه بعد ما يخلص التنفيذ

ده بيخلي Lambda مثالية للتطبيقات الصغيرة أو Microservices اللي مش محتاجة سيرفر دايم.

🧪 في كورس Jenkins ده

هنا Lambda جزء من الـ Production Deployment Pipeline، لأننا هنستخدم Jenkins علشان:

1. نجهز الـ Node.js Application
2. نرفعه كـ Lambda Function
3. نعمل Trigger لـ Lambda من Jenkins ونختبر النتيجة.

يعني Lambda هي الخطوة الأخيرة اللي التطبيق بيستقر فيها "في الإنتاج"، بدل ما نرفعه على EC2 أو Kubernetes

---

🧠 Best Practices

1. خلي الكود صغير وسريع
   - Lambda ليها حد زمني (15 دقيقة كحد أقصى).
   - كل ما الكود يكون خفيف، الأداء يكون أسرع والتكلفة أقل.
2. استخدم Layers للـ Dependencies
   - متحطش كل المكتبات جوه الكود مباشرة.
   - استخدم Lambda Layers عشان تفصل الأكواد المشتركة وتقلل الحجم.
3. راقب الأداء بالـ CloudWatch Logs
   - راقب زمن التشغيل واستهلاك الذاكرة بشكل مستمر.
4. استفد من الـ Triggers المتعددة
   - دمج Lambda مع API Gateway أو S3 بيديك نظام أوتوماتيكي كامل.
5. جرب الكود محليًا قبل الرفع
   - استخدم AWS SAM CLI أو Serverless Framework للتجربة المحلية.

📃 الملخص الشامل

- Lambda = Serverless Service لتشغيل الكود بدون سيرفرات.
- بتوفر لك auto scaling ، pay-per-use ، وبتشغل الكود بناءً على أحداث معينة.
- بتدعم لغات متعددة، حتى اللغات القديمة.
- بتخلي المطور يركز على الكود فقط مش البنية التحتية.
- في الكورس ده، هنستخدم Lambda كمرحلة الإنتاج النهائية داخل Jenkins Pipeline لتشغيل تطبيق Node.js تلقائيًا بعد الدمج في الفرع الرئيسي (main branch)

Using GenAI to Generate Steps for Lambda Deployment

🎯 الفكرة العامة

- ❑ الجزء ده بيتكلم عن تحضير Node.js App علشان نقدر ننشره على AWS Lambda باستخدام Jenkins Pipeline
- ❑ بمعنى تاني:

التطبيق اللي كان معمول يشتغل على EC2 أو Kubernetes داخل Docker containers، محتاج شوية تعديل بسيط في الكود والبنية علشان يبقى متوافق مع Lambda .. لأن Lambda عندها نموذج تشغيل مختلف (Serverless)

📌 فالجزء دا هو التحضير قبل ما نبدأ التنفيذ الفعلي.

🧠 الشرح التفصيلي خطوة بخطوة

فهم المشكلة

- ➢ التطبيق بتاعنا (Node.js + Express) كان شغال على سيرفر فعلي أو على Kubernetes ، لكن Lambda مش بتحتاج سيرفر ولا بتتعامل بنفس الطريقة.
- ➢ في Lambda ، لازم التطبيق يكون متوافق مع event-driven architecture .. يعني بيتعامل مع أحداث (Requests) بشكل مباشر من AWS ، مش من Web Server زي Express بس.
- ➢ إذن لازم نعمل تعديل في الـ business logic (الكود نفسه) علشان Lambda تعرف تتعامل معاه.

نفهم المتطلبات من AWS Documentation

Building Lambda functions with Node.js - AWS

- ➢ AWS عندها documentation ممتاز، لكن ساعات بيكون طويل ومليان تفاصيل كتير، ولذلك ممكن نستعين بـ Gen AI Tools (زي Copilot أو ChatGPT) علشان نجيب خطوات محددة وسريعة.

🧩 أولا : نسأل عن الخطوات العامة لنشر Node.js App على Lambda باستخدام Jenkins

```
lambda-steps.md
1 > 1. Steps required to deploy a Node.js app to AWS Lambda using a Jenkins pipeline. ...
11
12 > 2. Are there specific Lambda/Serverless dependencies that need to be added to a Node.js application for
     deployment? ...
19
20 > 3. Is it necessary to modify the app.js file in a Node.js application to make it compatible with Lambda? ...
28
29 > 4. What AWS CLI command can be used to update a Lambda function? ...
33
34 > 5. How can I use AWS CLI commands to update a Lambda function with a file stored in an AWS S3 bucket? ...
41
42 > 6. What AWS CLI command can be used to retrieve the function URL configuration? ...
```

These are the ques that he asked

🧩 أولا : ايه الخطوات المطلوبة ؟!

- ❑ الـ Copilot أجاب علي أول سؤال إن الخطوات هي :

📋 الخطوة 1: إعداد بيئة AWS

لازم يكون عندك AWS Account مفعّل، وتكون مجهز IAM Role عنده صلاحيات على Lambda و S3 (لو هتستخدمه)

1. Set Up Your AWS Environment

- **Create an AWS Account**: If you don't have one, sign up at the AWS Management Console.
- **Install AWS CLI**: Install the AWS Command Line Interface on your local machine.
- **Configure AWS CLI**: Run `aws configure` and enter your AWS Access Key ID, Secret Access Key, default region, and output format.

الخطوة 2: تجهيز التطبيق

عندنا التطبيق جاهز، لكن محتاجين نتأكد من إن الـ dependencies متسطبة.

أمر عملي: **npm install**

2. Create Your Node.js Application

- **Initialize Project**: Create a new directory and run `npm init` to set up a new Node.js project.
- **Install Dependencies**: Use `npm install` to add necessary dependencies.
- **Write Your Code**: Develop your Node.js application. For example, a simple Express server:

JavaScript

```javascript
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.send('Hello, AWS Lambda!');
});
app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

AI-generated code. Review and use carefully. More info on FAQ.

الخطوة 3: ضغط التطبيق في ملف ZIP

لأن Lambda بتحتاج الملف مضغوط علشان ترفعه كـ deployment package

أمر عملي : **zip -r myapp.zip .**

ده هيضغط كل ملفات المشروع (الكود + node_modules) في ملف zip

3. Package Your Application

- **Create Deployment Package**: Zip your application code and dependencies.

```
zip -r my-node-app.zip index.js node_modules
```

الخطوة 4: إنشاء Lambda Function ورفع الكود عليها

هنستخدم AWS CLI علشان ننشئ function ونرفع الكود.

أمر عملي:

4. Deploy to AWS Lambda

- **Create IAM Role**: In the AWS Management Console, create a new IAM role with the `AWSLambdaBasicExecutionRole` policy.
- **Deploy Function**: Use the AWS CLI to deploy your function.

```
aws lambda create-function --function-name my-node-app --zip-file fileb://my-node-app.zip --
handler index.handler --runtime nodejs14.x --role arn:aws:iam::<your-account-id>:role/<your-role-
name>
```

هنا:

- Runtime : بيحدد نوع اللغة.
- Role : صلاحيات التنفيذ.
- Handler : اسم الملف والدالة الأساسية اللي Lambda هتشغلها.
- zip-file : ملف الكود المضغوط.

الخطوة 5: إعداد Jenkins Pipeline

هنا بيقول إننا هنحتاج نضيف stages زي:

- Install dependencies
- Run unit tests
- Zip application
- Deploy using AWS CLI

وده بيتطلب Plugins زي:

- Git plugin (لسحب الكود)
- AWS Lambda plugin (للتعامل مع AWS)

5. Set Up Jenkins Pipeline

- **Install Jenkins**: Ensure Jenkins is installed and running.
- **Install Plugins**: Install necessary plugins like Git and AWS Lambda.
- **Create Pipeline Job**: In Jenkins, create a new pipeline job and configure the source code management to pull from your Git repository.
- **Configure Build Steps**: Add build steps to:
  - Install dependencies (`npm install`).
  - Run tests (if any).
  - Package the application (`zip -r my-node-app.zip index.js node_modules`).
  - Deploy to AWS Lambda using the AWS CLI commands.

Yes, when deploying a Node.js application to AWS Lambda, there are specific dependencies that can help streamline the process, especially if you're using the Serverless Framework. Here are some key dependencies you might need:

1. **Serverless Framework**
   - **serverless**: This is the core framework that simplifies the deployment of serverless applications.

```
npm install -g serverless
```

2. **Serverless HTTP**
   - **serverless-http**: This library allows you to wrap your existing Node.js HTTP frameworks (like Express) to run on AWS Lambda.

```
npm install serverless-http
```

3. **Express (Optional)**
   - **express**: If you're using Express to build your Node.js application, you'll need to include it.

```
npm install express
```

Example Setup

Here's a basic example of how you might set up a simple Express application to run on AWS Lambda using the Serverless Framework:

1. **Install Dependencies**:

```
npm init -y
npm install express serverless-http
```

ثانيا : معرفة Dependencies المطلوبة

❑ الـ Copilot أجاب علي ثاني سؤال إن الخطوات هي:

📦 أهم مكتبتين:

1. serverless
2. serverless-http

📘 أوامر عملية : npm install serverless serverless-http express

الـ serverless-http هي المكتبة اللي بتخلي تطبيق Express يعرف يشتغل جوه Lambda لأنها بتعمل wrap للتطبيق وبتخليه يفهم Requests اللي جايالـه من Lambda بدلاً من Web Server

---

2. Create app.js :

JavaScript

```
const express = require('express');
const serverless = require('serverless-http');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello, AWS Lambda!');
});

module.exports.handler = serverless(app);
```

ثالثا تعديل الكود في app.js

❑ الـ Copilot أجاب علي ثالث سؤال إن:

- بدل ما كنا بنستخدم: app.listen(3000)
- لازم نضيف في آخر الملف:

module.exports.habdler = serverless(app);

🚀 كدا Lambda لما تستدعي الـ function هتشغل التطبيق كـ serverless process

---

رابعا : تحديث الكود في Lambda (Deploy) جديد

❑ الـ Copilot أجاب علي رابع سؤال إن:

لو عايز تحدّث الكود بدون إنشاء Function جديدة:

📘 أمر عملي:

aws lambda update-function-code --function-name my-node-app --zip-file fileb://myapp.zip

لو الملف موجود على S3 بدلًا من الجهاز:

📘 بديل باستخدام S3 :

aws lambda update-function-code --function-name my-node-app --s3-bucket my-bucket \
--s3-key myapp.zip

خامسا : جلب الـ Function URL (علشان نختبر التطبيق)

📘 أمر عملي: aws lambda get-function-url-config --function-name my-node-app

ده بيرجعلك الـ public URL اللي ممكن تستخدمه لتجربة التطبيق بعد ما يتنشر.

---

1. Steps required to deploy a Node.js app to AWS Lambda using a Jenkins pipeline.

```
npm install
```

```
Update index.html page and append a version
```

```
zip -qr solar-system-lambda.zip app* package* index.html node*
```

2. Are there specific Lambda/Serverless dependencies that need to be added to a Node.js application for deployment?

```
const serverless = require('serverless-http');
```

```
module.exports.handler = serverless(app);
```

سادسا : تلخيص لكل الـ steps and commands :

1. جهز تطبيقك Node.js
2. ضيف serverless-http
3. عدّل الكود في app.js
   باستخدام module.exports.handler
4. اعمل zip للتطبيق.
5. أنشئ Lambda Function جديدة أو حدّث القديمة.

3. Is it necessary to modify the app.js file in a Node.js application to make it compatible with Lambda?

```
tail -5 app.js
sed -i "/^app\\.listen(3000/ s/^/\\/\\/\\//" app.js
sed -i "s/^module.exports = app;/\\/\\/\\/module.exports = app;/g" app.js
sed -i "s|^//module.exports.handler|module.exports.handler|" app.js
tail -5 app.js
```

4. What AWS CLI command can be used to update a Lambda function?

```
aws lambda update-function-code --function-name my-function --s3-bucket my-bucket --s3-key my-function.zip
```

5. How can I use AWS CLI commands to update a Lambda function with a file stored in an AWS S3 bucket?

```
aws s3 cp solar-system-lambda.zip s3://solar-system-lambda-bucket/solar-system-lambda.zip
```

```
aws lambda update-function-code --function-name solar-system-function --s3-bucket solar-system-lambda-bucket --s3-key solar-system-lambda.zip
```

6. What AWS CLI command can be used to retrieve the function URL configuration?

```
aws lambda get-function-url-config --function-name solar-system-function
```

6. استخدم AWS CLI علشان ترفع الكود وتشوف الـ URL
7. بعد كده، كل ده هيتحول لخطوات أوتوماتيكية في Jenkins Pipeline

---

⚙️ Best Practices

- ✅ استخدم Environment Variables داخل Lambda بدل hard-coded values
- ✅ حط node_modules الضرورية بس في الـ ZIP لتقليل الحجم.
- ✅ دايمًا استخدم IAM Role بصلاحيات محدودة (Least Privilege)
- ✅ اختبر الكود محليًا باستخدام serverless framework قبل النشر.
- ✅ راقب Logs في CloudWatch بعد كل نشر علشان تشوف أي Errors
- ✅ احفظ الـ deployment packages في S3 لأرشفة النسخ القديمة.

🧭 الملخص النهائي

الشرح دا كان بيجهزنا لنقل تطبيق Node.js من بيئة تقليدية (EC2/K8s) إلى بيئة Serverless (Lambda)
التحضير دا يشمل:

- تعديل الكود ليكون متوافق مع Lambda
- إضافة المكتبات الخاصة بـ serverless
- ضغط الكود في zip
- استخدام AWS CLI لنشره وتحديثه.
- وأخيرًا، الحصول على الـ URL لاختبار التطبيق.

بعد كده، هنتعلم إزاي نعمل نفس الخطوات دي أوتوماتيك من خلال Jenkins

Manual Lambda Deployment

🎯 الفكرة العامة

- ❑ إحنا دلوقتي هنبدأ ننفّذ فعليًا عملية نشر تطبيق Node.js على AWS Lambda يدويًا، قبل ما نربطه بـ Jenkins
- ❑ الهدف إنك تفهم كل خطوة على الأرض — من أول تجهيز الكود، لحد ما التطبيق يبقى شغال فعلاً على Lambda وتختبره من الـ URL بتاعه.
- ❑ يعني نعمل العملية manual عشان بعدين نعرف نعملها automated في Jenkins

---

1. 🧱 إعداد البيئة محليًا (Local Setup)

أول خطوة لازم نجهز التطبيق على جهازنا:

📘 أوامر : cd my-node-app & npm install

- ➢ ده بيأكد إن كل الـ dependencies متسطبة قبل ما نبدأ عملية الـ packaging
- ➢ لو اشتغلت على Lambda بدون تثبيت الـ dependencies ، الـ Function هتفشل لأنها مش هتلاقي المكتبات اللي محتاجاها.

💡 ملاحظة مهمة:

في Lambda لازم ترفع كل الملفات الضرورية فقط.

يعني لو عندك dev dependencies (زي eslint أو jest) متخليهاش في zip .. و استخدم:

- o npm install --production

ده بيقلل حجم الـ package ويحسّن سرعة النشر.

---

2. 🧩 تحويل التطبيق ليكون متوافق مع Lambda

زي ما قولنا، Lambda مش بتسمح باستخدام app.listen()
فلازم نعدل app.js بالشكل ده:

📘 كود معدل:

```
const express = require('express');
const serverless = require('serverless-http');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Lambda!');
});

module.exports.handler = serverless(app);
```

كده Lambda لما تستدعي الـfunction ، هتستخدم handler كمدخل لتشغيل التطبيق.

---

3. 🧳 عمل Package للتطبيق (ZIP)

دلوقتي لازم نضغط الملفات عشان نرفعها لـ AWS :

📘 أوامر: zip -r myapp.zip .

💡 نصايح عملية:

- تأكد إنك بتعمل الـ zip من داخل فولدر المشروع نفسه.
- متنساش النقطة . عشان تضيف كل الملفات.
- لو حجم zip كبير عن MB50 ، ارفعه على S3 بدل ما ترفعه مباشرة من CLI

---

4. ☁️ إنشاء Lambda Function جديدة

دلوقتي نبدأ بإنشاء Lambda Function من AWS CLI :

📘 أمر:

```
aws lambda create-function  --function-name my-node-app  --runtime nodejs18.x \
 --role arn:aws:iam::123456789012:role/lambda-role  --handler app.handler  --zip-file fileb://myapp.zip
```

شرح :

- function-name : اسم الـLambda function.
- Runtime : بيحدد اللغة والإصدار (هنا Node.js 18)
- Role : IAM Role عندها صلاحيات Lambda + CloudWatch
- Handler : الدالة اللي Lambda هتبدأ منها (زي main function)
- zip-file : مسار ملف الكود اللي ضغطناه.

Real Case 💡 :

- في بيئات الإنتاج، الـ IAM Role دي بيكون معمولها policy فيها صلاحيات محددة جدًا (Minimal Access)
- يعني مثلًا AWSLambdaBasicExecutionRole بس، علشان ما يبقاش عندها صلاحيات زيادة.

---

5. 🔄 تحديث الكود بعد التعديل (Update Function Code)

لو عندك Function جاهزة وعايز تحدث الكود بدل ما تنشئ واحدة جديدة:

📘 أمر:

```
aws lambda update-function-code  --function-name my-node-app  --zip-file fileb://myapp.zip
```

💡 ملاحظة:

- ده بيساعدك لما تكون بتعمل continuous deployment من Jenkins بعد كل commit
- يعني Jenkins هيعمل zip جديد ويرفعه بنفس الأمر ده.

---

6. 🧠 ربط Lambda بواجهة (URL Invocation)

Lambda ممكن تتنفذ بطرق مختلفة:

- إما من API Gateway
- أو Function URL مباشرة (لو فعلتها)

📘 أمر:

```
aws lambda get-function-url-config  --function-name my-node-app
```

- هتطلعلك حاجة زي : https://abc123.lambda-url.us-east-1.on.aws/
- وده هو الرابط اللي تقدر تجرب منه التطبيق مباشرة.

📘 تجربة التشغيل:

```
curl https://abc123.lambda-url.us-east-1.on.aws/
```

- هيطبعلك: Hello from Lambda!

Real Trick 💡 :

لو الـURL بيرجع Error 403 ، تأكد إن AuthType متظبط على NONE في إعدادات الـfunction URL :

```
aws lambda add-permission  --function-name my-node-app  --principal '*' \
 --action lambda:InvokeFunctionUrl  --statement-id function-url-access \
 --function-url-auth-type NONE
```

---

7. 🧰 (اختياري) إعداد API Gateway بدلاً من URL

- لو عايز تحكم أكتر (Routing, Auth, Headers...) استخدم API Gateway بدل الـ URL المباشر.
- تقدر توصله بـ Lambda بسهولة من الـ AWS Console أو Terraform
  - بس في الكورس هنا، إحنا بنكتفي بالـ Function URL عشان نبسّط التجربة.

---

8. 🧩 مراجعة Logs

بعد ما تختبر التطبيق، تقدر تشوف الـ logs في CloudWatch :

📘 أمر:

- aws logs describe-log-groups
- aws logs get-log-events --log-group-name /aws/lambda/my-node-app --log-stream-name \ <stream-name>

ده مهم جدًا لو الـ Lambda رجعت Error أو Timeout .. هتقدر تعرف بالضبط إيه اللي حصل جوه الكود.

Best Practices ✅

1. استخدم serverless-http بدل الكود اليدوي.
2. متحطش ملفات ضخمة أو secrets في zip .. و استخدم AWS Secrets Manager
3. دايمًا اختبر Function محليًا بـ serverless invoke local قبل النشر.
4. خليك على أحدث Runtime مدعوم من AWS
5. استخدم Jenkins stages صغيرة (build / package / deploy) بدل سكريبت واحد ضخم.
6. استخدم Parameter Store أو ENV Variables بدل hardcoded values

---

🧭 الملخص النهائي

- اللي عملناه هنا هو تنفيذ عملي لإعداد تطبيق Node.js على AWS Lambda يدويًا.
  - بدأنا بتثبيت dependencies ، عدّلنا الكود علشان يبقى compatible ، ضغطناه في zip ، رفعناه على Lambda ، وبعد كده اختبرناه من الـ URL
- ودي هي نفس الخطوات اللي Jenkins هيعملها أوتوماتيك في Pipeline بعدين.

📌 الفكرة إنك تكون فاهم كل خطوة "manually" قبل ما تعتمد على الأتمتة.

Example on our App Solar System

هانمشي
عالـ steps اللي
في p.252&253

الفكرة العامة:

- ❑ إحنا عندنا تطبيق اسمه Solar System مكتوب بـ Node.js ، وعايزين نرفعه على AWS Lambda يدويًا (manual deployment)
- ❑ الـ Lambda هي خدمة بتشغّل الكود من غير ما تحتاج سيرفر (Serverless)

المكونات اللي جهزها قبل البدء:

1. S3 Bucket
   - اسمه solar-system-lambda-bucket :
   - بيستخدمه عشان يرفع فيه ملف الـ zip. بتاع الكود.
2. Lambda Function
   - اسمها solar-system-function :
   - الـ Runtime: Node.js 20.x
   - الـ Handler: app.handler
     (لو اسم الملف index.js يبقى index.handler)
   - فيها كود شغال فعلاً (version 3.0)

ملاحظات مهمة:

- عنده Environment Variables (زي الـ MongoDB username/password/URI)
  مكتوبة plain text ← وده غلط جدًا.
- هيشيلها بعدين لما يعمل pipeline ويستخدم Secrets Manager

خطوات الـ Manual Deployment :

1 تجهيز الكود محليًا

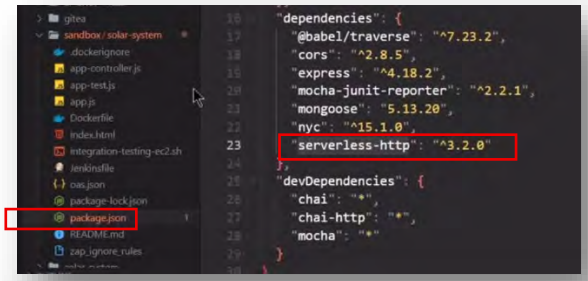
- أنشأ فولدر جديد اسمه sandbox ← mkdir sandbox
- عمل clone للـ repository بتاع المشروع من GitHub بداخل فولدر sandbox
- Git clone http://<your ip>/dasher-org/solar-system
- دخل على package.json وتأكد إن فيه dependency مهمة:
- "serverless-http"

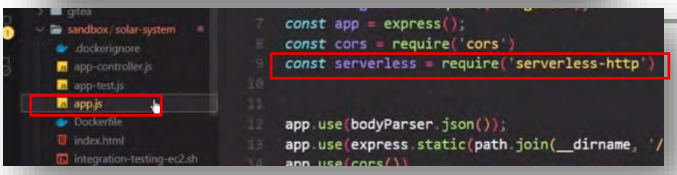
ودي مطلوبة عشان Node.js يشتغل كويس جوه Lambda

- بعدين فتح app.js وتأكد

إنه بيستخدم : const serverless = require("serverless-http");

عشان يقدر يلف الـ Express app جوه Lambda handler





2 نعمل install للـ dependencies :

npm install ده بيعمل فولدر node_modules

---

3 نعدّل في الكود (version update)

- فتح صفحة index.html وعدّل رقم الـ Version من 3.0 إلى 4.0
  عشان لما يرفع يعرف إن الـ Deployment اشتغل صح.

4 نضبط app.js للـ Lambda :

- عباستخدام أمر : tail -5 app.js ← جابله اخر خمس سطور في ملف index.html زي دي:

```
tail -5 app.js

app.listen(3000, () => { console.log("Server successfully running on port - " +3000); })
module.exports = app;

//module.exports.handler = serverless(app)
```

- لما يشغلها على Lambda ، لازم يعلّق (comment) الـ app.listen()
ويخلي السطر الأخير uncommented (module.exports.handler)
- عمل ده أوتوماتيك باستخدام أوامر sed عشان يعدّل السطور دي في الكود.

```
root@jenkins-controller-1 in solar-system on main [!] via v20.16.0 on (us-east-2)
❯ sed -i "/^app\\.listen(3000/ s/^/\\/\\//" app.js
  sed -i "s/^module.exports = app;/\\/\\/module.exports = app;/g" app.js
  sed -i "s|^//module.exports.handler|module.exports.handler|" app.js
```

---

5 نضغط الملفات في ملف ZIP :

المطلوب فقط:

```
app.js
app.txt
index.html
package.json
package-lock.json
node_modules/
```

واستخدم الأمر:

```
zip -r solar-system-lambda.zip app.js app.txt index.html package.json package-lock.json node_modules
```

---

6 نرفع الـ ZIP على S3 :

```
aws s3 cp solar-system-lambda.zip s3://solar-system-lambda-bucket/solar-system-lambda.zip
```

(لازم يكون AWS CLI متكوّن بـ credentials)

---

7 نحدث الـ Lambda Function :

```
aws lambda update-function-code --function-name solar-system-function \
 --s3-bucket solar-system-lambda-bucket --s3-key solar-system-lambda.zip
```

- ده بيخلي Lambda تستخدم الكود الجديد من الـ ZIP اللي على S3
- هترجع JSON فيه تفاصيل عن التحديث (زي آخر تعديل، env vars .... إلخ)

---

8 نشوف التحديث:

- من الـ Console : تقدر تفتح Lambda → Configuration → Function URL
وتشوف اللينك.
- أو من CLI : `aws lambda get-function-url-config --function-name solar-system-function`
هيطلعلك الـ URL

🧪 ✅ اختبار التطبيق:

فتح اللينك في المتصفح ← المفروض يشوف: Solar System 4.0
وده معناه إن الـ Lambda اتحدثت بنجاح.

⚡ في النهاية:

الخطوات اللي اتعملت يدويًا:

1. تجهّز الكود (dependencies + handler)
2. تضغطه فيZIP
3. ترفعه على S3
4. تحدث Lambda بالكود الجديد من S3
5. تتأكد إن التطبيق اشتغل

🎯 والسيشن الجاي هيشرح إزاي نعمل كل ده أوتوماتيك في Jenkins Pipeline

Lambda Deployment Using Jenkinsfile

🎯 الفكرة العامة

- ❑ الهدف هو إننا نخلّي Jenkins هو اللي يتولى نشر الـ Lambda Function بدل ما نعمل ده يدويًا من AWS Console
- ❑ يعني Jenkins هيعمل:

1. تعديل في كود التطبيق (app.js) حسب الحاجة.
2. يضغط الملفات في ZIP
3. يرفع الـ ZIP على S3 Bucket
4. يحدث كود الـ Lambda Function باستخدام الملف اللي اترفع على S3

🚀 وده كله أوتوماتيكي من الـ Jenkinsfile بدون تدخل يدوي.

🧩 شرح تفصيلي خطوة بخطوة

المرحلة الأولى : تعديل الإعدادات في الـ Lambda

- أول حاجة عملها انه دخل علي الـ configuration وبعدين حذف الـ Environment Variables من الـ Lambda Function وكل دا عادي من الـ AWS Console (خليك فاكر الحتة دي عشان هاتضرب error في الاخر)
- ليه؟ لأننا هنضيفها ونتحكم فيها بعدين من داخل Jenkinsfile بدل ما تبقى ثابتة في الـ Console
- وده Practice مهم جدًا : خلي كل حاجة Dynamic وتتحكم فيها من Pipeline

---

المرحلة الثانية: التحضير في الـ VM

في الماشين اللي عليها Jenkins ، بدأ الراجل يجهّز الشغل:

1. حذف أي ملفات أو فولدرات قديمة باستخدام : rm -rf sandbox
   - دي خطوة تنظيف، علشان يشتغل في بيئة نظيفة بدون بواقي من جلسة قديمة.
2. فتح الريبو الرئيسي: cd solar-system
3. التأكد إنه واقف على الفرع (branch) الصح:
   - git checkout main → git fetch → git pull

علشان يجيب آخر تحديثات، خصوصًا الـ Jenkinsfile اللي بيحتوي على الـ stages الجديدة.

المرحلة الثالثة: إنشاء Stage جديدة في Jenkinsfile

أضاف مرحلة (Stage) جديدة اسمها: stage('Lambda S3 Upload and Deploy')

المرحلة دي مش بتشتغل إلا على الـ main branch ، باستخدام شرط:

```
stage('Lambda - S3 Upload & Deploy') {
    when {
        branch 'main'
    }
    steps {
        withAWS(credentials: 'aws-s3-ec2-lambda-creds', region: 'us-east-2') {
        }
    }
}
}
```

المرحلة الرابعة: إعداد الـ AWS Authentication

➢ قبل أي أمر AWS ، لازم Jenkins يعمل Authenticate وده بيتم عن طريق الـ plugin بتاع AWS credentials :

- withAWS(credentials: 'aws-creds', region: 'us-east-2')

كده Jenkins بقى عنده صلاحية يستخدم AWS CLI بالأكاونت المربوط بيه.

المرحلة الخامسة: سكريبتات الـ Shell داخل الـ Stage

```
stage('Lambda - S3 Upload & Deploy') {
    when {
        branch 'main'
    }
    steps {
        withAWS(credentials: 'aws-s3-ec2-lambda-creds', region: 'us-east-2') {
            sh '''
                tail -5 app.js
                echo "*****************************************************************"
                sed -i "/^app\\.listen(3000/ s/^/\\/\\/\\//" app.js
                sed -i "s/^module.exports = app;/\\/\\/\\/module.exports = app;/g" app.js
                sed -i "s|^//module.exports.handler|module.exports.handler|" app.js
                echo "*****************************************************************"
                tail -5 app.js
            '''
            sh '''
                zip -qr solar-system-lambda-$BUILD_ID.zip app* package* index.html node*
                ls -ltr solar-system-lambda-$BUILD_ID.zip
            '''
            s3Upload(
                file: "solar-system-lambda-${BUILD_ID}.zip",
                bucket:'solar-system-lambda-bucket'
            )
            sh '''
                aws lambda update-function-code \
                    --function-name solar-system-function \
                    --s3-bucket solar-system-lambda-bucket \
                    --s3-key solar-system-lambda-$BUILD_ID.zip
            '''
        }
    }
```

1 تعديل كود التطبيق

- هنا استخدم أوامر Shell لتعديل الكود في ملف app.js
- الأوامر الأساسية:
  - tail : بيعرض آخر 3 سطور من الملف علشان يتأكد قبل وبعد التعديل.
  - Sed : بيعمل تعديل أو Comment/Uncomment لسطور معينة في الكود حسب الحاجة.

الهدف هنا : تفعيل أو تعطيل جزء من الكود حسب بيئة التشغيل .

---

2 ضغط الملفات

بعد التعديل، بيجهّز التطبيق للنشر عن طريق ضغط الملفات في ملف ZIP

الأوامر:

- ${BUILD_ID} متغير بيتم توليده أوتوماتيك في Jenkins ، بيميز كل Build
- بيضغط ملفات المشروع اللي Lambda محتاجاها فقط.
- بعدين بيعمل ls علشان يتأكد إن الملف اتعمل فعلاً.

---

3 رفع الملف إلى S3 Bucket

- الأمر ده بيرفع الملف الـ ZIP إلى الـ Bucket
- لو الباكت مش موجود، هيدي Error .. لذلك تأكد دايمًا إن اسم الباكت صحيح ومكتوب زي ما هو في AWS Console.

---

4 تحديث كود الـ Lambda Function

بعد ما الملف اترفع، بيحدث الـ Lambda Function نفسها:

- الأمر ده هو اللي بيعمل Deployment حقيقي.
- Jenkins بياخد الكود اللي في الـ S3 وبيحدث بيه الـ Lambda Function

---

المرحلة السادسة : اختبار النشر

- بعد تنفيذ الـ Stage ، Jenkins شغّل الـ Build للـ main branch وبدأ يعدّي على كل المراحل:
  - Build
  - Test
  - Deploy
- لكن لاحظ إنه لما جرّب يفتح التطبيق في المتصفح، حصل 🙁 Internal Server Error
  وده منطقي، لأننا حذفنا الـ Environment Variables اللي كانت بتحتوي على بيانات الاتصال بـ MongoDB (URI، username، password)

المرحلة السابعة: تحليل الخطأ

راح على CloudWatch Logs من AWS علشان يشوف السبب:

- اللوج بيقول إن الخطأ سببه MongoDB URI undefined
- يعني التطبيق مش قادر يوصّل بالداتابيز.

الحل: نرجع نضيف الـ Environment Variables بس المرة دي من داخل Jenkinsfile أو عبر الـ AWS CLI

المرحلة الثامنة: تجهيز للحل

في نهاية الشرح قال إن الخطوة الجاية هي:

- نستخدم Jenkinsfile لتحديث Lambda Configuration وإرسال الـ Environment Variables بشكل آمن.
- باستخدام Jenkins credentials اللي فيها بيانات MongoDB

⚙️ Best Practices

1. خلي Jenkins هو اللي يدير الـ Deployment بالكامل (ما تعتمدش على UI)
2. استخدم Build IDs لتسمية ملفاتك علشان تقدر تتبع كل إصدار بسهولة.
3. خزن الـ Secrets و Credentials جوه Jenkins securely باستخدام Credentials Plugin
4. استخدم S3 كوسيط آمن لتخزين الحزم (ZIPs) اللي هتنشرها على Lambda
5. اقرأ دايمًا الـ CloudWatch Logs بعد أي Error .. لأنها أول مكان تشوف فيه السبب الحقيقي.
6. استخدم شروط (when branch) علشان الـ Stages تشتغل بس على الفروع اللي محتاجاها.
7. احذف الـ stages اللي مش لازمة (زي build docker image هنا، لأنها مش مستخدمة في Lambda)

🧠 الملخص العام

الجزء دا بيشرح خطوة بخطوة إزاي:

1. نحذف الـ environment variables القديمة.
2. نجهّز Jenkins بيئة نظيفة ونجيب آخر كود.
3. نضيف Stage جديدة مخصوصة للـ Lambda Deployment
4. نعمل تعديل في الكود، نضغط الملفات، نرفعها على S3
5. نحدث كود الـ Lambda Function من S3
6. نشوف النتيجة ونفهم سبب الـ error لو حصل.
7. نجهز Jenkinsfile علشان يضيف الـ environment variables أوتوماتيك بعد كده.

Update Lambda Configuration

🎯 الفكرة العامة

- ❑ الجزء ده هدفه يشرح إزاي نحدث إعدادات (Configuration) خاصة بوظيفة AWS Lambda من خلال Jenkinsfile بشكل أوتوماتيك بدل ما نعملها يدوي من الـ AWS Console
- ❑ الموضوع بيتكلم بالتحديد عن تحديث المتغيرات البيئية (Environment Variables) الخاصة بالـ Lambda Function زي الـ Mongo Username و Password والـ URI
- ❑ الفكرة إن Jenkins هيبقى مسؤول عن:
  - رفع الكود الجديد للـ Lambda
  - تحديث الإعدادات الخاصة بيه (زي environment variables)
  - التعامل مع أي أخطاء أو مشاكل بتحصل في مراحل الـ pipeline زي فشل dependency check
  - وأخيرًا التأكد إن التطبيق شغال بعد التحديث بشكل سليم.

🧩 الشرح التفصيلي خطوة بخطوة

Working with Lambda environment variables - AWS Lambda

1️⃣ أول حاجة : تحديث إعدادات Lambda من Jenkinsfile

- عاوزين نحدّث إعدادات Lambda باستخدام AWS CLI بدل ما نفتح AWS Console ونعملها يدوي.
- الأمر المستخدم هو:

```
aws lambda update-function-configuration  --function-name <function-name>  --environment\
"Variables={mongo_username=$MONGO_USERNAME,mongo_password=$MONGO_PASSWORD,
mongo_uri=$MONGO_URI}"
```

📘 شرح الأمر:

- update-function-configuration ← ده الكوماند اللي بيحدّث إعدادات الـ Lambda
- --function-name ← بنكتب هنا اسم الـ Lambda Function اللي عاوزين نحدّثها.
- --environment ← هنا بندي JSON فيه كل الـ Environment Variables اللي عاوزين نحدّثها.
- كل variable بيبقى key=value داخل {} ومفصولين بفواصل.

💡 ملاحظة عملية:

- o لما تشتغل على Jenkins ، القيم دي عادة بتكون متخزنة كـ Jenkins Credentials علشان السرية.
- o يعني بدل ما تحط password في الكود، بتستخدم مثلاً ${env.MONGO_PASSWORD}

2️⃣ إضافة الكود داخل Jenkinsfile

➢ هايضيف الكوماند ده داخل Stage('Lambda - S3 Upload & Deploy') في Jenkinsfile وتحديدا قبل الـ step بتاع "Update Function Code" ، وده منطقي لأنك لازم تحدث الـ Configuration الأول قبل ما ترفع الكود.

مثال داخل Jenkinsfile :

```
        zip -qr solar-system-lambda-$BUILD_ID.zip app* package* index.html node*
        ls -ltr solar-system-lambda-$BUILD_ID.zip
    '''
    s3Upload(
        file: "solar-system-lambda-${BUILD_ID}.zip",
        bucket:'solar-system-lambda-bucket'
    )
    sh """
        aws lambda update-function-configuration \
            --function-name solar-system-function  \
            --environment '{"Variables":{ "MONGO_USERNAME": "${MONGO_USERNAME}",
            "MONGO_PASSWORD": "${MONGO_PASSWORD}","MONGO_URI": "${MONGO_URI}"}}'
    """
    sh '''
        aws lambda update-function-code \
            --function-name solar-system-function \
            --s3-bucket solar-system-lambda-bucket \
            --s3-key solar-system-lambda-$BUILD_ID.zip
    '''
}
```

🧠 ليه استخدمنا double quotes (") ؟

لأن الـ Groovy في Jenkins بيحتاج double quotes علشان يقدر يبدّل المتغيرات داخل النص (${env.MONGO_USERNAME})

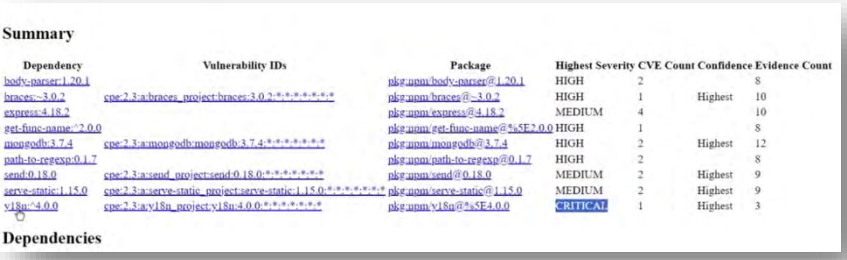
3 دمج المراحل داخل Stage واحد

- في السكريبت ده، كل العمليات (رفع للـ S3 ، تحديث الكود، وتحديث الإعدادات) بتحصل في Stage واحدة، وده لتبسيط الـ pipeline
- لكن في الشركات الكبيرة أو المشاريع المعقدة، الأفضل إنك تعمل كل خطوة في Stage منفصل علشان:
  - تسهل المتابعة والتصحيح.
  - وتقدر تعيد تشغيل Stage معينة لو فشلت.

---

4 التعامل مع فشل الـ Dependency Check

- هاتلاحظ إن مرحلة dependency scanning فشلت بسبب critical vulnerability في مكتبة i18n
- طاب عرف منين ؟!؟!
  - راح علي Artifacts → Dependency check HTML Report



Summary

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
body-parser:1.20.1		pkg:npm/body-parser@1.20.1	HIGH	2		8
braces:~3.0.2	cpe:2.3:a:braces_project:braces:3.0.2:*:*:*:*:*:*:*	pkg:npm/braces@~3.0.2	HIGH	1	Highest	10
express:4.18.2		pkg:npm/express@4.18.2	MEDIUM	4		10
get-func-name:^2.0.0		pkg:npm/get-func-name@%5E2.0.0	HIGH	1		8
mongodb:3.7.4	cpe:2.3:a:mongodb:mongodb:3.7.4:*:*:*:*:*:*:*	pkg:npm/mongodb@3.7.4	HIGH	2	Highest	12
path-to-regexp:0.1.7		pkg:npm/path-to-regexp@0.1.7	HIGH	2		8
send:0.18.0	cpe:2.3:a:send_project:send:0.18.0:*:*:*:*:*:*:*	pkg:npm/send@0.18.0	MEDIUM	2	Highest	9
serve-static:1.15.0	cpe:2.3:a:serve-static_project:serve-static:1.15.0:*:*:*:*:*:*:*	pkg:npm/serve-static@1.15.0	MEDIUM	2	Highest	9
v18n:^4.0.0	cpe:2.3:a:v18n_project:v18n:4.0.0:*:*:*:*:*:*:*	pkg:npm/v18n@%5E4.0.0	CRITICAL	1	Highest	3

Dependencies

- علشان يكمل البايبلاين (للتجريب فقط) عدّل Jenkinsfile وخلى النتيجة ما توقفش البيلد حتى لو فيه مشكلة:

```
stage('Dependency Scanning') {
    parallel {
        stage('NPM Dependency Audit') {
        }

        stage('OWASP Dependency Check') {
            steps {
                dependencyCheck additionalArguments: '''
                    --scan \'./\'
                    --out \'./\'
                    --format \'ALL\'
                    --disableYarnAudit \
                    --prettyPrint''', odcInstallation: 'OWASP-DepCheck-10'

                dependencyCheckPublisher failedTotalCritical: 1, pattern: 'dependency-check-report.xml',
                stopBuild: false
            }
        }
```

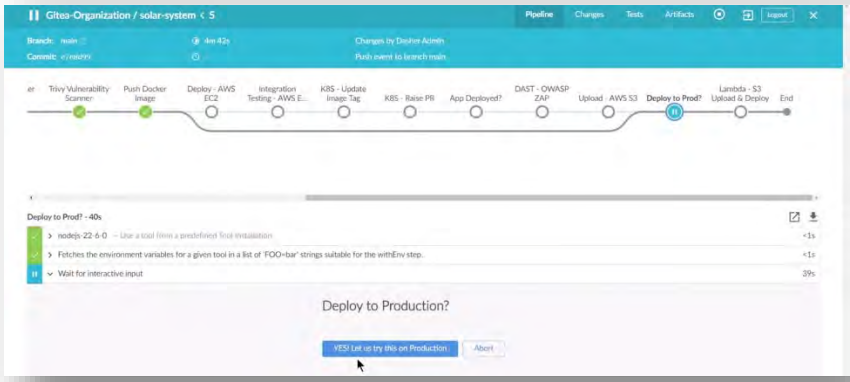
كانت true وخلاها false

⚠️ تحذير عملي:

- في بيئة الإنتاج، ما ينفعش تتجاهل الثغرات دي، لازم تتصلح.
- لكن أثناء التدريب أو التجريب ممكن تتجاوزها مؤقتًا.

---

5 متابعة تنفيذ البيلد



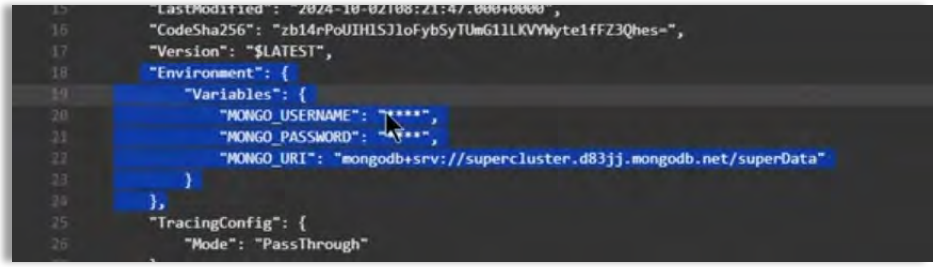
- بعد التعديل، بيرجع يبني الـ pipeline من جديد.
- المراحل اشتغلت بنجاح لحد ما وصلت مرحلة Deploy to Production
- وبعدها بدأ يرفع الكود الجديد ويحدّث الإعدادات.
- الأوامر اللي اشتغلت:
  - رفع الكود للـ S3 :

```
aws s3 cp solar-system-lambda.zip s3://my-lambda-bucket/
```

  - تحديث Lambda config :

```
aws lambda update-function-configuration --function-name myLambda --environment ...
```

- كل القيم السرّية (زي passwords) بتكون masked في Jenkins log ، وده اللي ظهر لما شاف إن القيم متغطية في الـ console output



```
"LastModified": "2024-10-02T08:21:47.000+0000",
"CodeSha256": "zb14rPoUIH1SJloFybSyTUmG11LKVYWyte1fFZ3Qhes=",
"Version": "$LATEST",
"Environment": {
    "Variables": {
        "MONGO_USERNAME": "****",
        "MONGO_PASSWORD": "****",
        "MONGO_URI": "mongodb+srv://supercluster.d83jj.mongodb.net/superData"
    }
},
"TracingConfig": {
    "Mode": "PassThrough"
```

6️⃣ التحقق من التحديث

- بعد تنفيذ Jenkinsfile ، راح على AWS Console → Lambda → Configuration → Environment variables
  ولقى إن القيم اتحدثت فعلاً بالبيانات الجديدة اللي جاية من Jenkinsfile
- كده أثبت إن الـ Jenkins pipeline بقى بيعمل تحديث تلقائي للـ Lambda config

---

7️⃣ التحقق من التطبيق بعد التحديث

اختبر التطبيق (API أو web app) من المتصفح:

- اشتغل بدون error
- قدر يتصل بقاعدة MongoDB
- رجع بيانات الكواكب من الـ API

بكده اتأكد إن الكود والـ config الجديد شغالين تمام.

⚙️ Best Practices

1. ✅ متخزنش الـ credentials في الكود
   استخدم Jenkins credentials أو AWS Secrets Manager
2. 🔄 افصل كل مرحلة في Stage منفصلة
   عشان تقدر تعيدها بسهولة لو فشلت (S3 Upload, Lambda Config, Update Code)
3. 🧪 اعمل اختبارات تلقائية بعد كل تحديث
   استخدم أدوات زي Postman أو curl من داخل Jenkinsfile لاختبار الـ Lambda بعد التحديث.
4. 🧰 راقب الـ Lambda logs في CloudWatch
   علشان تتأكد إن التحديثات ما سببتش أخطاء runtime
5. 🧨 ما تتجاهل الـ Dependency Vulnerabilities في production
   الحل الصحيح هو تحديث المكتبات أو استبدالها الآمنة.

📜 الملخص النهائي

- ❑ الجزء ده شرح إزاي نستخدم Jenkins pipeline لتحديث إعدادات AWS Lambda Function أوتوماتيكيًا باستخدام AWS CLI
- ❑ الخطوات الأساسية كانت:
  1. تحديد environment variables داخل Jenkinsfile
  2. استخدام aws lambda update-function-configuration لتحديثها.
  3. رفع الكود للـ S3 وتحديث Lambda code
  4. التعامل مع أخطاء الفحص الأمني (dependency check)
  5. اختبار التطبيق بعد التحديث للتأكد من نجاح العملية.

💡 النتيجة:

بمجرد ما أي متغير (زي password) يتغير في Jenkins ، بيتحدث أوتوماتيكيًا في Lambda بدون تدخل يدوي.

Lambda-Invoke Function

🧠 أولًا: الفكرة العامة

- ❑ الهدف من الجزء دا من الكورس إننا نضيف مرحلة جديدة في Jenkins pipeline علشان نختبر Lambda Function اللي تم رفعها ونشرها على AWS
- ❑ يعني بعد ما نعمل build ونرفع الكود على S3 ونعمله deploy على Lambda ، عاوزين نتأكد إن الـ Lambda دي شغالة فعلاً ومفيش Error
- ❑ وده بيتم عن طريق إن Jenkins :
  1. يجيب الـ Function URL بتاع الـ Lambda
  2. يستخدم cURL علشان يعمل request للـ endpoint /live مثلاً.
  3. ويتأكد إن الـ Status Code = 200 OK (يعني السيرفر شغال تمام)

🔍 ثانيًا: الشرح التفصيلي خطوة بخطوة

💡 شرح السكريبت ↓↓↓

```
stage('Lambda - S3 Upload & Deploy') {...
}

stage('Lambda - Invoke Function') {
    when {
        branch 'main'
    }
    steps {
        withAWS(credentials: 'aws-s3-ec2-lambda-creds', region: 'us-east-2') {
            sh '''
                sleep 30s
                function_url_data=$(aws lambda get-function-url-config --function-name
                solar-system-function)
                function_url=$(echo $function_url_data | jq -r '.FunctionUrl | sub("/$"; "")')
                curl -Is  $function_url/live | grep -i "200 OK"
            '''
        }
    }
}
}
```

1. 🧩 إنشاء Stage جديد في Jenkinsfile

- ➢ بدأنا بإضافة stage("Lambda - Invoke Function")
- ➢ ودي بيضيفها بعد stage("Lambda - S3 Upload & Deploy")

المرحلة دي:

- بتشتغل فقط على الفرع الرئيسي (main branch)
- بتستخدم AWS Credentials المسجلة في Jenkins ، علشان تقدر تنفذ أوامر CLI علي AWS

---

2. 🕒 Sleep 30 ثانية

- ➢ بعد ما يرفع الكود على Lambda ، Jenkins بينفذ أمر: sleep 30

📝 السبب :أحيانًا بعد ما ترفع كود جديد على Lambda ، بتاخد وقت بسيط (ثواني) علشان الـ Lambda تتحدث وتبقى جاهزة للتنفيذ.

- o فعلشان يتأكد إن التحديث تم بنجاح، بينتظر 30 ثانية قبل ما يختبرها.

---

3. 🔗 جلب الـ Lambda Function URL

- ➢ الخطوة اللي بعدها هي الحصول على الرابط (URL) اللي نقدر نستخدمه لاختبار الـ Lambda
- ➢ الأمر المستخدم : aws lambda get-function-url-config --function-name <function-name>
  - الـ AWS CLI بيرجع JSON كبير فيه تفاصيل كتير، منها الـ URL
  - لكن إحنا مش محتاجين غير الـ URL ، فبنخزنه مؤقتًا في متغير:
    - o function_url_data=$(aws lambda get-function-url-config --function-name my-lambda)

---

4. 📜 استخراج الـ URL الحقيقي من JSON باستخدام jq

- ➢ الـ jq أداة بتساعدك تعالج الـ JSON بسهولة في Linux .. هنا بنستخدمها علشان نطلع قيمة الـ URL :
  - o function_url=$(echo $function_url_data | jq -r '.FunctionUrl')

بس أحيانًا الـ URL بييجي في الآخر فيه .. / slash وده ممكن يعمل مشاكل في الـ curl command ، فبيشيلوه بالأمر:

- o function_url=${function_url%/} or like in script → sub("/$"; "")')

ده معناه : "لو آخر حرف في المتغير / ، شيله."

5. 🌐 اختبار الـ Lambda باستخدام cURL

- دلوقتي هنستخدم الـ URL اللي جبناه علشان نعمل request ونتأكد إنها بترد.
- الأمر المستخدم: `curl -I "$function_url/live" | grep -I "200 ok"`
  - الخيار -I معناه: "هاتلي الرؤوس (headers) بس، مش جسم الرد."
  - الـ /live هو endpoint المفروض بيرجع 200 OK لو كل حاجة تمام.
- بعد تنفيذ الأمر، لو النتيجة فيها:
  - HTTP/2 200
  - يبقى الـ Lambda شغالة تمام ✅
    - لو غير كده (زي 400 أو 500)، يبقى في مشكلة في الـ deployment أو الكود.

---

6. 🔁 متابعة التنفيذ في Jenkins

- Jenkins بينفذ المراحل واحدة ورا التانية.
- مرحلة "Lambda Invoke Function" هتظهر بعد "S3 Upload & Deploy"
- اللون الأحمر اللي ظهر في الفيديو مش بسبب Lambda ، لكن بسبب مشكلة في"OWASP Dependency Check" ، وده تحليل أمان مش بيأثر على الـ pipeline لأنه كان stopBuild=false (يعني حتى لو في Error متوقفش البيلد)

---

7. 💻 تنفيذ نفس الأمر يدويًا (اختبار عملي)

- بعد كده نسخ نفس أوامر الـ AWS CLI والـ cURL وشغلها في الـ Virtual Machine ، وشاف النتيجة: HTTP/ 200 .. وده دليل إن الـ Lambda شغالة ومربوطة بالـ Jenkins Pipeline بشكل سليم.

Best Practices 💡

1. ✅ دائمًا انتظر شوية بعد كل Deploy لـ Lambda (زي sleep 30s) قبل ما تعمل اختبار.
2. 🧠 استخدم jq لاستخراج البيانات بدل الـ grep علشان دقته في JSON أعلى.
3. 🔐 احفظ AWS credentials في Jenkins credentials store ، ومتكتبهاش في الكود.
4. 🧪 اختبر Endpoints بسيطة زي /live أو /health علشان تتأكد إن السيرفر شغال.
5. 📜 سجل نتائج الاختبار في Jenkins logs علشان لو حصل فشل، تعرف السبب بسهولة.
6. ⚙️ خلي المرحلة دي optional (يعني لو فشلت الاختبارات ما توقفش البيلد الأساسي لو كانت مرحلة Monitoring فقط)

🧭 ملخص

- الهدف : اختبار الـ Lambda Function بعد عملية الـ Deploy
- الطريقة Jenkins Stage : جديدة اسمها "Lambda Invoke Function"
- الخطوات:
  1. انتظر 30 ثانية بعد الـ Deploy
  2. استخدم AWS CLI علشان تجيب الـ Function URL
  3. استخرج الـ URL باستخدام jq
  4. شيل الـ Slash في الآخر.
  5. استخدم curl علشان تختبر الـ /live endpoint
  6. لو رجع HTTP 200 ← الـ Lambda تمام. ✅
- النتيجة النهائية : مرحلة Jenkins بتأكد إن الـ Lambda شغالة بنجاح بعد النشر.

Jenkins Folder (Part1)

🎯 أولًا: الفكرة العامة

- ❑ الجزء ده بيتكلم عن ميزة الـ Folders في Jenkins وهي واحدة من الحاجات القوية جدًا اللي بتخليك تنظم المشاريع والـ pipelines داخل Jenkins بطريقة مرتبة وواضحة.
- ❑ الفكرة إنك ممكن يكون عندك أكتر من تيم أو مشروع شغالين على نفس Jenkins controller ، لكن عايز تفصل كل تيم أو مشروع عن التاني:
  - كل تيم يبقى ليه فولدر خاص بيه.
  - جوه الفولدر ممكن تحط pipelines, credentials, libraries خاصة بيه.
  - ومفيش تداخل بين الفولدرات.
- ❑ ببساطة:

الفولدر = "مساحة مستقلة" جوه Jenkins فيها إعدادات وموارد خاصة بيها.

🧩 ثانيًا: الشرح التفصيلي خطوة بخطوة

1. 🧱 تعريف مفهوم الـ Folders

بكل بساطة ← الـ folders بتعمل namespace منفصل داخل Jenkins

يعني:

- ممكن يكون في job اسمه build في Folder A
- وكمان job تاني اسمه build في Folder B
- ومفيش تعارض بينهم 👌

فكل فولدر كأنه Jenkins صغير خاص بالتيم ده.

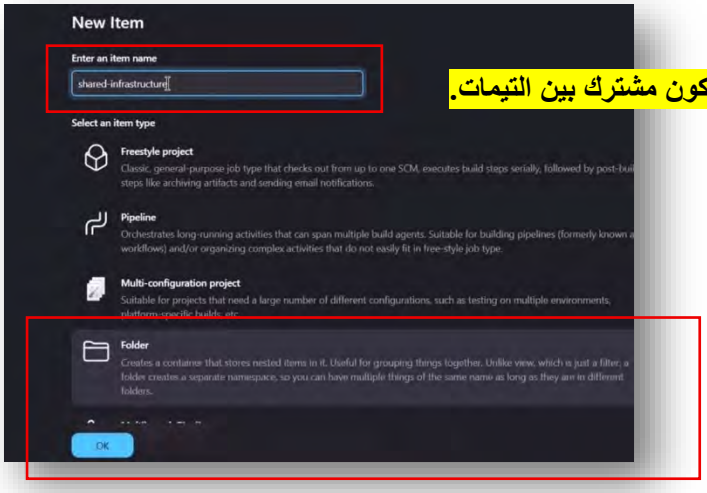
---

2. ⚙️ مميزات استخدام الـ Folders

- تنظيم المشاريع حسب التيمات أو المجالات.
  مثلاً : Frontend, Backend, DevOps, Infrastructure
- فصل الـ credentials :
  كل فولدر ممكن يكون ليه credentials خاصة بيه.
- تحكم في الصلاحيات:
  تقدر تحدد مين يشوف إيه داخل الفولدر.
- إضافة إعدادات خاصة بالفولدر:
  زي Docker registry, Kubernetes configs أو pipeline libraries

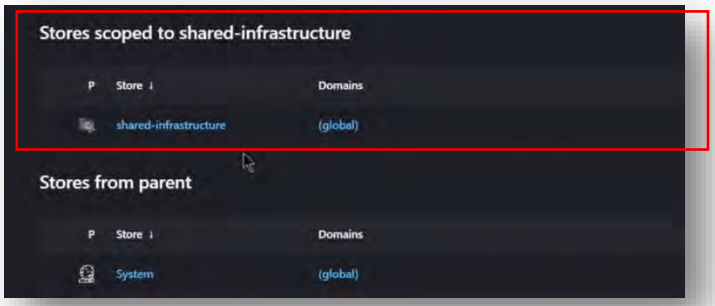
---

3. 🏗️ إنشاء أول Folder – "Shared Infrastructure"



- ➢ أنشأنا فولدر اسمه: Shared Infrastructure .. وده فولدر أساسي هيكون مشترك بين التيمات.
- ➢ الفولدر ممكن يحتوي على إعدادات زي:
  - Display name و Description
  - Health metrics (يقيس حالة الـ Jobs)
  - Docker registry labels
  - Pipeline libraries خاصة بالفولدر.
  - إعدادات Kubernetes cloud لو محتاجها.

وبعد إنشاءه، هايظهر فعلاً فولدر جديد في Jenkins



4. 🔐 إنشاء Credentials داخل الفولدر

- دلوقتي هنشوف نقطة مهمة جدًا:
  كل فولدر ممكن يكون فيه credentials خاصة بيه.
- يعني لو روحت لـ : Manage Jenkins → Credentials
- هتلاحظ إن فيه:
  - domain Global credentials (اللي بنستخدمه عادة)
  - ودلوقتي بقى فيه Domain خاص بـ Shared Infrastructure folder
- فبدأ ينشئ credential جديدة داخل فولدر Shared Infrastructure :
  - النوع : Username with password
  - الاسم : Shared-DATABASE-username
  - الـ ID : shared-db-creds
  - الـ Description : shared-db-creds
- فكده بقى عندنا credential خاصة بالفولدر ده فقط ✅
  يعني بس الـ pipelines اللي جوه Shared Infrastructure تقدر تستخدمها.

---

5. 📁 إنشاء Folder داخل Folder

- أنشأنا فولدر جديد جواه وسماه : Team-a
- الفكرة إن Team A محتاج يستخدم بعض الموارد المشتركة
  من فولدر Shared Infrastructure
  زي الـ credentials أو الـ libraries مثلًا.
- فـ Team A دلوقتي جوه Shared Infrastructure ← يعني وراثيًا يقدر يشوف الإعدادات الخاصة بالأب.

---

6. 🔐 إنشاء Credentials جديدة داخل Team A

- دخل على فولدر Team A .. واختار "Add Credentials" → Username with password
  - Username : TeamAUsername
  - Password : أي حاجة (لأننا مش هنقراه)
  - ID : TeamAcreds

وكده بقى عندنا:

- Credential في Shared Infrastructure
- و Credential تاني في Team A

---

7. 🚀 إنشاء Pipeline داخل Team A Folder

- دلوقتي هيختبر هل Jenkins هيقدر يقرأ credentials اللي فوق ولا لأ.
- أنشأ pipeline جديدة باسم : Team A Pipeline وحط فيها Jenkinsfile بالشكل ده :

Script ?

```
pipeline {
    agent any

    environment {
        SHARED_DB_CREDS = credentials('shared-db-creds')
        TEAM_A_CREDS = credentials('team-a-creds')
    }

    stages {
        stage('Accessing Credentials') {
            steps {
                script {
                    echo "Username from Shared folder Credentials: ${SHARED_DB_CREDS_USR}"
                    echo "Username from Team-A folder Credentials: ${TEAM_A_CREDS_USR}"
                }
            }
        }
    }

    // Add post-build actions
    post {
        success {
            echo "Build completed successfully"
        }
        failure {
            echo "Build failed"
        }
    }
}
```

🔍 الشرح:

- environment فيه متغيرين:
  - واحد بياخد credentials من Shared Infrastructure
  - والتاني بياخد credentials من Team A
- داخل الـ stage بيطبع usernames علشان نشوف مين متاح.

---

8. 📘 تجربة التنفيذ في Blue Ocean

بعد ما حفظ البايبلاين وبنى المشروع (Build Now) :

- النتيجة إن Jenkins قدر يوصل لكل الـ ✅ credentials سواء بتاعة Team A أو Shared Infrastructure

وده لأن:

Team A folder جزء من Shared Infrastructure folder ، فبيقدر يرث الـ credentials اللي فوقه.

Best Practices 💡

✅ استخدم Folders لتقسيم المشاريع حسب الفرق أو بيئات العمل

- زي : Dev, Test, Prod أو Team A, Team B

✅ احفظ credentials في أقرب مستوى للفولدر اللي بيستخدمها

- يعني بدل ما تحط كل حاجة في Global ، خلي كل فولدر مسؤول عن نفسه.

✅ استخدم IDs واضحة للـ credentials

- زي shared-db-creds, teamA-api-key علشان تبقى سهلة في Jenkinsfile

✅ خلي الفولدرات المرتبطة تحت فولدر رئيسي مشترك

- علشان الوراثة (inheritance) تشتغل بين الأب والابن.

✅ متحطش أسرار في النص مباشرة داخل Jenkinsfile

- دايمًا استخدم credentials() function

📘 الملخص العام

- الهدف : تنظيم المشاريع داخل Jenkins باستخدام Folders
- كل فولدر بيكون Namespace منفصل فيه Jobs, Pipelines, و Credentials خاصة بيه.
- الفولدرات الفرعية بتقدر تورّث من الفولدر الأب (زي Team A من Shared Infrastructure)
- كل فولدر ممكن يكون فيه إعدادات مستقلة زي Docker registry أو Kubernetes configs
- الـ Credentials داخل فولدر الأب بتبقى متاحة للأبناء.
- النتيجة النهائية : Team A قدر يوصل لكل الـ credentials بنجاح.

Jenkins Folder (Part2)

🎯 أولًا: الفكرة العامة

- ❑ الشرح ده هو الجزء التاني من موضوع Jenkins Folders
- ❑ الفكرة هنا إننا عاوزين نوضح إزاي الصلاحيات والـ credentials بتتوزع بين الفولدرات المختلفة داخل Jenkins
- ❑ يعني لما يكون عندك:
  - Folder رئيسي اسمه مثلًا Shared Infrastructure
  - وجواه فولدرات فرعية زي Team A و Team B

🚨 فهل Team B تقدر تستخدم الـ credentials اللي Team A عاملاها؟

- o الإجابة: لأ.

وهو ده بالظبط اللي هانبرهنه خطوة بخطوة 👇

🧩 ثانيًا: الشرح خطوة بخطوة

1. 🧱 إنشاء فولدر جديد داخل Shared Infrastructure

- ➢ الهدف : نحاكي وجود تيم جديد داخل نفس بيئة العمل.
- ➢ الخطوات:
  - افتح Jenkins
  - من الصفحة الرئيسية اختار : 🧱 New Item → Folder
  - سمّي الفولدر الجديد: team-b ← اضغط OK

📎 ملحوظة:

كل فولدر جديد بيعتبر Namespace مستقل، يعني Jenkins بيشوفه كبيئة منفصلة.

---

2. 🧱 إنشاء Pipeline جديد داخل Team B

الخطوات:

- جوه فولدر team-b ، اضغط New Item → Pipeline
- سمّي البايبلاين : team-b pipeline
- في خانة الـ Pipeline Script انسخ نفس الكود اللي استخدمناه في Team A :

Script ?

```
pipeline {
    agent any

    environment {
        SHARED_DB_CREDS = credentials('shared-db-creds')
        TEAM_A_CREDS = credentials('team-a-creds')
    }

    stages {
        stage('Accessing Credentials') {
            steps {
                script {
                    echo "Username from Shared folder Credentials: ${SHARED_DB_CREDS_USR}"
                    echo "Username from Team-A folder Credentials: ${TEAM_A_CREDS_USR}"
                }
            }
        }
    }

    // Add post-build actions
    post {
        success {
            echo "Build completed successfully"
        }
        failure {
            echo "Build failed"
        }
    }
}
```

---

3. 🧠 تشغيل الـ Pipeline وملاحظة النتيجة

- اضغط Build Now
- بعد شوية، الـ Build هيفشل. ❌

➢ ولما تفتح Blue Ocean UI أو Console Output هتلاقي Error بالشكل ده:

org.jenkinsci.plugins.credentialsbinding.impl.CredentialNotFoundException: Credentials 'Team A Creds' not found

➢ 📌 السبب:

- o الفولدر Team B ما عندوش صلاحية يشوف الـ credentials بتاعة Team A
- o لكن تقدر تشوف إن الـ pipeline قدر يوصل لـ Shared DB Creds عادي، لأنها كانت موجودة على مستوى Shared Infrastructure

💡 ثالثًا: ملاحظات و Best Practices من الواقع

1. ✅ كل Folder في Jenkins يعتبر Environment مستقلة:
   - يعني ممكن تخصص Credentials أو Variables أو Pipeline Libraries لكل تيم لوحده.
   - وده بيساعد في العزل الأمني (Security Isolation)
2. 🔒 إياك تسيب الـ credentials Global لو مش لازم:
   - الـ Global Domain بيخلي كل الـ jobs تشوف الـ credentials
   - الأفضل تحطها على مستوى الـ Folder الخاص بالتيم.
3. ⚙️ اعمل Shared Folder للبنية التحتية فقط (Infrastructure Folder) :
   - ده بيحتوي على الـ resources المشتركة زي docker registry أو قاعدة بيانات عامة.
   - كل تيم بعد كده يورّث منه اللي محتاجه فقط.
4. 🧩 Nested folders logic :
   - الفولدر الفرعي بيرث من اللي فوقه، لكن مش العكس.
   - Team B ورث من Shared Infrastructure
     لكن ما ورثش من Team A

📋 رابعًا: الملخص

النقطة	التوضيح
🎯 الهدف	اختبار وراثة الـ credentials بين الفولدرات
🧱 الإنشاء	Folder جديد اسمه Team B داخل Shared Infrastructure
⚙️ التنفيذ	Pipeline بنفس كود Team A
❌ النتيجة	فشل الـ Build بسبب عدم وجود صلاحية لـ Team A credentials
✅ الاستنتاج	الـ Credentials مرئية فقط داخل نفس الفولدر أو الفولدرات الفرعية منه

Jenkins Folder (Part3)

🎯 أولًا: الفكرة العامة

- ❑ الشرح ده هو الجزء الثالث من سلسلة "Jenkins Folders"
- ❑ في الديمو ده، هنشوف إزاي نقدر نعمل فولدر جديد (Team C) يكون نُسخة من فولدر موجود أصلًا (Team A) باستخدام ميزة اسمها "Copy from"
- ❑ الميزة دي بتوفر وقت ومجهود لما يكون عندك إعدادات (settings, pipelines, credentials) ممكن تعيد استخدامها بدل ما تبدأ من الصفر.
- ❑ الفكرة الأساسية 👇

لما تعمل Folder جديد بالـ "Copy from" من فولدر تاني ، Jenkins بينسخ:

- الـ Jobs (Pipelines)
- الـ Configurations
- الـ Credentials
- وأحيانًا الـ Properties العامة للفولدر

يعني الفولدر الجديد بيبقى تقريبًا نسخة طبق الأصل من القديم.

🧩 ثانيًا : الخطوات التفصيلية مع الأوامر والشرح

1. 📄 إنشاء فولدر جديد باسم Team C

➢ الخطوات داخل Jenkins UI :

1. من الـ Dashboard اختار: New Item 🧱
2. اكتب اسم الفولدر الجديد: team-c
3. قبل ما تضغط OK ، فعّل الاختيار : "Copy from" .. واكتب في الخانة دي: team-a
4. علشان الفولدر الجديد ياخد إعدادات Team A ← اضغط OK ✅

➢ 📎 اللي بيحصل هنا:

Jenkins بيعمل نسخة جديدة من كل حاجة جوه Team A :

- الـ Pipelines اللي Team A أنشأها
- الـ Credentials اللي تخص Team A
- الإعدادات الخاصة بالفولدر

---

2. 📄 التحقق من النسخ التلقائي

بعد الإنشاء:

- افتح فولدر Team C

هتلاحظ إن جوه نفس الـ pipeline اللي كان في Team A :

- Team A Pipeline
- وكمان لما تروح على Manage Credentials هتلاقي إن:
  - فيه صلاحية لـ Shared Infrastructure credentials
  - وكمان فيه access لـ Team A credentials ✅

📌 يعني Jenkins نقل كل الصلاحيات الخاصة بالفولدر المنسوخ منه.

---

3. 📋 تفعيل الـ Job المنسوخ

- الـ Jobs اللي بتتنسخ بالـ "Copy from" بيكون وضعها Disabled بشكل افتراضي.
  يعني مش هتقدر تبنيها غير لما تفعلها يدويًا.

خطوات التفعيل:

1. افتح الـ pipeline المنسوخ (Team A pipeline) داخل Team C
2. اضغط Configure
3. بعد كده اضغط Save

Jenkins ✅ هيفعل الـ job وهيظهر زر Build Now

---

4. 📋 تشغيل الـ Pipeline وملاحظة النتائج

- اضغط Build Now الـ Pipeline هيتنفذ بنجاح 🎉
- وفي الـ Blue Ocean UI أو في الـ Console Output ، هتلاحظ:

Shared DB username: ...
Team A username: ...

يعني:

- الـ pipeline قدر يوصل للـ credentials الخاصة بـ Shared Infrastructure ✅
- وكمان قدر يوصل للـ credentials الخاصة بـ Team A ✅

💡 ثالثًا: ملاحظات عملية و Best Practices

1. ⚙️ ميزة "Copy from" قوية جدًا لكن استخدمها بحذر:
   - لو الفولدر القديم فيه إعدادات حساسة (credentials، tokens، .... إلخ) ← فهي هتتنسخ تلقائيًا.
   - دايمًا راجع إعدادات الـ permissions بعد النسخ.
2. 🧱 استخدمها كـ Template للفولدرات:
   - ممكن تعمل فولدر Template جاهز اسمه مثلًا "Team Template" ، فيه كل الإعدادات الأساسية اللي عاوزها لأي تيم جديد، وبعدين تعمل "Copy from Team Template" بدل ما تعيد الإعداد من الصفر.
3. 🔒 راجع الـ credentials inheritance بعد النسخ:
   - Jenkins ممكن ينسخ الـ links للـ credentials مش الـ credentials نفسها.
     فلو الفولدر الأصلي اتغير أو اتمسح، الفولدر الجديد ممكن يتأثر.
4. 🧩 ميزة النسخ مش دايمًا آمنة في البيئات الكبيرة (Enterprise Jenkins) :
   - بعض الشركات بتمنعها لأن فيها مخاطرة نسخ إعدادات Security حساسة.
   - البديل هو استخدام Jenkins Configuration as Code (JCasC)

📋 رابعًا: الملخص السريع

النقطة	التوضيح
🎯 الهدف	إنشاء فولدر جديد بنسخ إعدادات فولدر قديم
📦 الأداة المستخدمة	"Copy from" أثناء إنشاء الفولدر
⚙️ النتيجة	فولدر جديد يحتوي على نفس الـ pipelines و الـ credentials
❌ الملاحظة	الـ jobs المنسوخة بتكون Disabled لازم تفعيلها يدويًا
✅ الفائدة	سرعة الإعداد، وتوحيد البيئة بين الفرق المختلفة
⚠️ التحذير	لازم مراجعة الـ credentials والصلاحيات بعد النسخ

Jenkins Folder (Part4)

🎯 أولاً: المقدمة

- ❑ هانشرح حاجة اسمها weather icons اللي بتظهر في Jenkins dashboard .. ودي عبارة عن أيقونات صغيرة بتدل على حالة أو "صحة" المشروع.
- ❑ يعني ببساطة، Jenkins بيستخدم أيقونات الطقس (شمس، سحاب، مطر...) كطريقة لعرض حالة الـ Build Stability

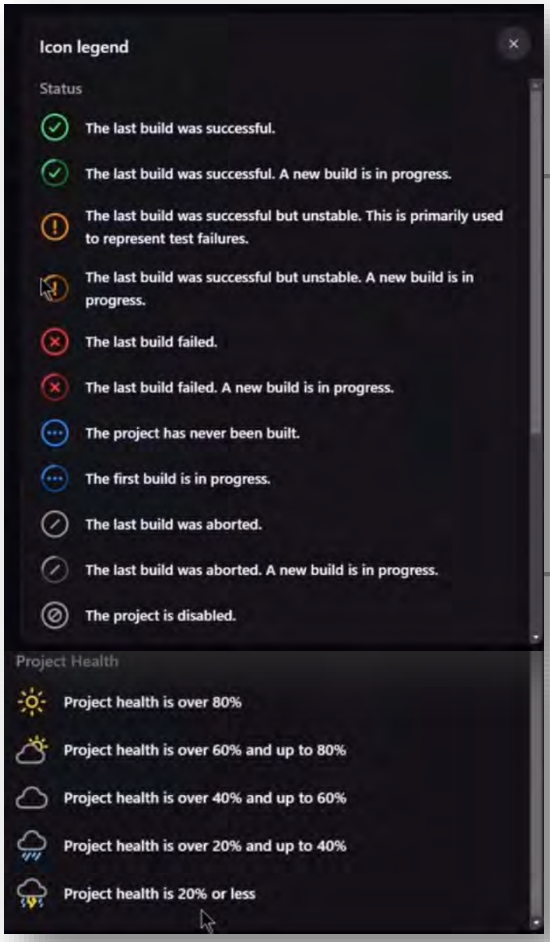
---

🌤️ أنواع الـ Weather Icons ومعناها:

الأيقونة	المعنى	النسبة المئوية للصحة
شمس	المشروع سليم جدًا	أكثر من 80%
جزئيًا مشمس	صحي لكن فيه بعض المشاكل	من 60% إلى 80%
غائم أو ممطر	مشاكل واضحة	أقل من 60%
برق	أغلب الـ Builds فشلت	أقل من 20%

🟢 الخلاصة:

- الـ "Weather Icon" بتعكس مدى استقرار الـ Builds الأخيرة للمشروع.
- كل ما الـ Builds الأخيرة تكون ناجحة، الأيقونة تبقى "أحسن".



📁 ثانياً: المشكلة في Folders

- ➢ لو دخلنا على Shared Infrastructure Folder و Team B Folder
- ➢ هتلاحظ إن:
  - كل الفولدرات ظهر عليها أيقونة شمس ☀️ (يعني صحية جدًا).
  - لكن لما تدخل على Team B Folder، هتلاقي إن فيه Job Failed!
- ➢ 😮 رغم كده، الفولدر نفسه (Team B) لسه بيظهر صحي جدًا في الـ Dashboard!

📌 وده لأن Jenkins بشكل افتراضي ما بيحسبش صحة الفولدر من صحة المشاريع اللي جواه.

⚙️ ثالثاً: الحل ← استخدام Health Metrics

- ➢ Jenkins عنده إعداد اسمه Health Metrics تقدر تفعّله لكل فولدر ..

وده بيخلي Jenkins يحسب صحة الفولدر بناءً على المشاريع اللي جواه.

الخطوات:

1. ادخل على الفولدر (مثلاً Team B)
2. اضغط على Configure
3. انزل لجزء اسمه Health Metrics
4. اضغط على Add Metric
5. اختار الخيار ده: Child item with worst health
6. فعل اختيار Recursive (عشان لو فيه فولدرات متداخلة)
7. اضغط Apply وبعدها Save

🧩 رابعاً: النتيجة

بعد ما تعمل الخطوات دي:

- **Jenkins هيبدأ يحسب حالة الفولدر حسب أسوأ Job جواه.**
- **لو فيه Job فاشل، الأيقونة هتتغير فورًا وتوضح إن الفولدر نفسه فيه مشاكل.**

والموضوع ده ممكن تعمله في أي مستوى من الفولدرات (Nested Folders)

يعني تقدر تخلي:

"Shared Infrastructure" ← يتأثر بصحة "Team B" ← اللي بيتأثر بصحة الـ Jobs اللي جواه.

💡 الخلاصة

📌 **Folder Health Metrics :**

بتخليك تشوف حالة مشاريعك كلها من نظرة واحدة . .. ومش بس على مستوى الـ Pipelines ،
لكن كمان على مستوى الفولدرات اللي بتنظمهم.

Script Console

jenkins-scripts/scriptler at main · jenkinsci/jenkins-scripts

🎯 أولاً: يعني إيه Script Console في Jenkins ؟

- ❑ الـ Script Console هي ميزة موجودة في Jenkins بتسمحلك إنك تشغل سكربتات بلغة Groovy مباشرةً على الـ Jenkins Controller (الماستر) أو على أي Agent متصل بيه.
- ❑ Groovy = لغة مبنية على Java ، و Jenkins نفسه معمول بـ Java ، فبالتالي تقدر تتفاعل مع كل Objects في Jenkinsعن طريق Groovy

---

⚠️ قبل ما تستخدم الـ Script Console :

فيه شوية نقاط مهمة لازم تعرفها:

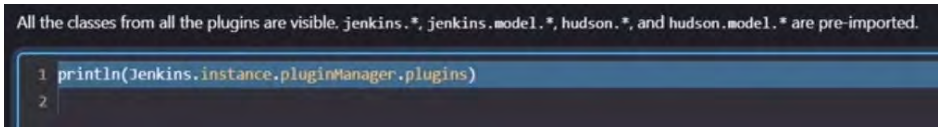
1. لازم تكون Administratorعلشان تقدر تستخدمها.
2. الـ Script Console قوية جدًا — بتقدر من خلالها تعمل أي حاجة جوه Jenkins :
   - o تعدل Configurations ، تمسح Jobs ، تضيف Users ، حتى توقف Jenkins نفسه!
3. علشان كده الوصول ليها لازم يكون محدود جدًا لأشخاص موثوق فيهم.

💻 إزاي نوصل لـ Script Console ؟

فيه طريقتين أساسيين:

1. من Dashboard ← Manage Jenkins ← تنزل تحت هتلاقي Script Console
   - o دي بتشغّل السكربت على الـ Controller
2. أو من صفحة Nodes ← اختار Built-in Node ← هتلاقي برضو Script Console
   - o دي تقدر تشغّل منها سكربت على Node معينة (Agent)

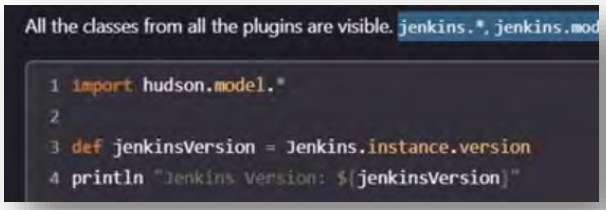
🧩 أول تجربة : طباعة كل الـ Plugins



- ➢ أول سكربت كان بسيط جدًا :
  - o لما تشغله، بيعرضلك كل الـ plugins اللي متسطبة على الـ Jenkins Controller ✅
- ➢ لاحظ إنه مش محتاج import لأن Jenkins بيدي شوية pre-imported classes جاهزين زي:
  - Jenkins.* & Jenkins.model.*
  - Hudson.* & Hudson.model.*

🔍 تجربة تانية : نطبع إصدار Jenkins

السكريبت كان بالشكل ده :



دا الاسم القديم للـ Jenkins

شرح الكود:

- import hudson.model.* : بيستورد الكلاسات اللي بتتعامل مع الـ Jenkins objects
- Jenkins.instance.version : بترجع إصدار Jenkins الحالي.
- println : بتطبع النتيجة في الـ console

💡 الناتج:

Jenkins version is 2.462.1

🧠 سكريبت مهم: طباعة كل الـ Jobs وتفاصيلها

➢ السكريبت ده أكبر شوية، وبيعمل loop على كل الـ Jobs :

All the classes from all the plugins are visible. jenkins.*, jenkins.model.*, hudson.*, and hudso

```
import jenkins.model.Jenkins

def jobs = Jenkins.instance.getAllItems()

jobs.each { job ->
    println "#######################################################"
    println "Job Name: ${job.fullName}"
    println "Job URL: ${job.absoluteUrl}"
    println "Job Last Build Number: ${job.lastBuild?.number}"
    println "Job Class: ${job.getClass().getName()}"
    println "#######################################################"
}
```

🧩 الشرح:

- getAllItems() : بتجيب كل الـ Jobs الموجودة في Jenkins
- each : loop على كل Job
- job.fullName : بيجيب المسار الكامل (بما في ذلك اسم الفولدر لو جوه فولدر).
- job.name : بيجيب الاسم فقط.
- job.absoluteurl : اللينك بتاع الـ job
- job.lastBuild?.number : رقم آخر build (الـ ? علشان لو مفيش build مايحصلش Error)
- job.class : بيقولك نوع الـ job (Freestyle / Pipeline)

💡 الناتج بيكون حاجة زي:

```
Full name: team-a/team-a-pipeline
Simple name: team-a-pipeline
URL: http://jenkins.local/job/team-a/job/team-a-pipeline/
Last build: 4
Class: org.jenkinsci.plugins.workflow.job.WorkflowJob
-----------------------------
```

🧮 سكريبت إضافي من GitHub : حساب عدد الـ Executor (ودا الريبو اللي فيه معظم الأمثلة اللي هاتحتاجها)

jenkins-scripts/scriptler/countExecutors.groovy at main · jenkinsci/jenkins-scripts

```
import jenkins.model.Jenkins

println("All Nodes - executor count:")
println()
Jenkins.instance.computers.eachWithIndex() { c, i -> println " [${i+1}] ${c.displayName}: ${c.numExecutors}" };
println()
println("Total nodes: [" + Jenkins.instance.computers.size() + "]")
println("Total executors: [" + Jenkins.instance.computers.inject(0, {a, c -> a + c.numExecutors}) + "]")
```

Result

Result

```
All Nodes - executor count:

 [1] Built-In Node: 2
 [2] ubuntu-agent: 1

Total nodes: [2]
Total executors: [3]
```

📊 سكريبت من آخر GitHub : Jenkins Stats

jenkins-script-console-scripts/jenkins-stats.groovy at main · samrocketman/jenkins-script-console-scripts

➢ السكريبت ده بيجمع إحصائيات كاملة عن Jenkins زي:

- عدد الـ GitHub organizations
- عدد الـ jobs
- عدد الـ builds
- عدد الـ users

...إلخ.

Result

```
Number of GitHub Organizations: 5
Number of GitHub Projects: 4
Number of Jenkins jobs: 31
Jobs with more than one build: 29
Number of users: 8
Global total number of builds: 192
Global total number of pull requests executed: null
Global total number of tag releases executed: null
Count of projects by type.
  FreeStyleProject: 8
  WorkflowJob: 23
  OrganizationFolder: 1
  WorkflowMultiBranchProject: 4
  Folder: 5
```

➢ فكرة السكريبت:

هو بيعمل queries على Jenkins objects ويحسب الـ counts ، زي:

Jenkins.instance.getAllItems(Job.class).size()

jenkins-script-console-scripts/disable-jenkins-cli.groovy at main · samrocketman/jenkins-script-console-scripts

🔒 **نقطة مهمة في الآخر:**

- ممكن تستخدم Script Console لتعطيل حاجات معينة لأسباب أمنية، زي مثلاً تعطيل Jenkins CLI (لأنه ممكن يُستغل)
- سكريبت بسيط زي كده:

```
Jenkins.instance.getDescriptor("jenkins.CLI").setEnabled(false)
```

بيوقف الـ CLI interface مؤقتًا أو دائمًا.

🧠 **الملخص:**

- Script Console = أداة قوية جدًا لإدارة Jenkins
- تقدر تعمل منها Monitoring, Debugging, Configuration, وحتى Automation
- لكن لازم تستخدمها بحذر، لأنها زي صلاحيات root على Jenkins

Global Security Settings

🧱 الفكرة العامة:

- ❑ (Jenkins) بييجي جاهز بإعدادات أمان قوية علشان يقلل فرص الاختراق أو استغلال الثغرات.
- ❑ لكن في بعض الحالات، ممكن تحتاج "تفتح" شوية من الأمان ده مؤقتًا علشان تشغل Job معينة أو Plugin بيحتاج صلاحيات معينة.
- ❑ فالمبدأ هنا بسيط جدًا : كل مرة تفتح فيها إعداد أمان مؤقتًا... لازم تقفله تاني أول ما تخلص.

🏠 تشبيه بسيط:

➢ تخيّل Jenkins زي البيت بتاعك:

- البيت مقفول كويس (ده الأمن الافتراضي).
- بس أحيانًا بتفتح الباب لحد يدخل يعمل شغل — زي ما بتفك إعداد علشان تشغّل Job
- بس بعد ما تخلص، لازم تقفله تاني و ترجع الإعدادات الآمنة زي ما كانت.

🔐 الإعدادات اللي اتكلم عنها:

➢ الشرح ركّز على إعدادين أساسيين ضمن Global Security Settings :

1. Markup Formatter
2. CSRF (Cross-Site Request Forgery) Protection

🧩 أولاً Markup Formatter :

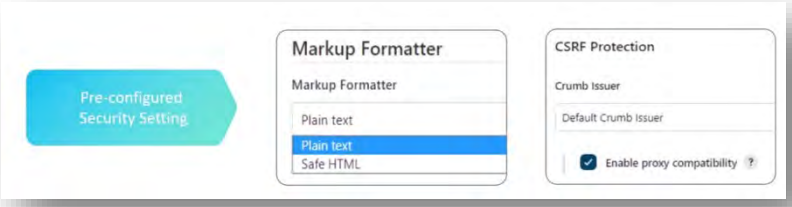
📘 الفكرة:

- Jenkins بيسمحلك تضيف Descriptions أو Messages داخل الـ jobs أو الـ views أو حتى النظام نفسه.
- الموضوع ده ممكن يكون فيه شوية تنسيق (Formatting) زي HTML tags

⚠️ المشكلة:

- لو المستخدمين قدروا يضيفوا HTML أو JavaScript جوه الوصف — فممكن جدا يحصل XSS Attack (Cross-Site Scripting) .. يعني المستخدم يحقن كود خبيث يتنفذ على المتصفح لما حد يفتح الصفحة دي.

💡 عشان كده Jenkins بيدي 3 اختيارات:



1. Plain Text (افتراضي وآمن جدًا)
   - أي حاجة تكتبها تتعرض كنص عادي.
   - مفيش HTML ولا تنسيق.
   - بيهرب (escapes) أي علامات ممكن تُقرأ ككود.
   - أكتر خيار آمن. ✅
2. Safe HTML
   - بيسمح بعناصر HTML بسيطة (زي <p>, <i>, <b>)
   - بس بيشيل أي عناصر خطيرة زي <script> أو <iframe>
   - 🟡 توازن بين الأمان والشكل.
3. Custom Markup Formatter (عبر Plugins)
   - Plugins معينة بتضيف دعم لتنسيقات أقوى.
   - بس لازم تتظبط كويس جدًا لأن فيها مخاطرة.
   - 🔴 ممكن تكون غير آمنة لو ما اتعملش إعدادها صح.

🧠 **مثال عملي:**

- لو استخدمت Plain Text وكتبت : Hello
  - هيتعرض كده في Jenkins : Hello
  - يعني الوسم مش هيتفسّر كـ HTML ، هيتعرض كنص عادي.
- لكن لو استخدمت Safe HTML، هيعرضها بالشكل ده : Hello

🧩 **ثانيًا CSRF Protection (Cross-Site Request Forgery) :**

⚠️ **المشكلة:**

- دي نوع من أنواع هجمات الويب .
- الفكرة إن مهاجم يقدر يستغل جلسة المستخدم اللي مسجل دخولها، ويخليه يعمل Action من غير ما يعرف — زي:
  - تشغيل Build بدون إذنه
  - حذف Artifacts
  - تعديل إعدادات Jenkins

🧠 **مثال بسيط:**

- افترض إنك فاتح Jenkins ، وفي نفس الوقت فتحت موقع تاني فيه كود خبيث بيبعت (Request) إلى Jenkins لتشغيل Job
- فللأسف ← جِنكنز هيفتكر إنك إنت اللي بعته، طالما عندك Session فعالة 😬

✅ **الحل: تفعيل CSRF Protection**

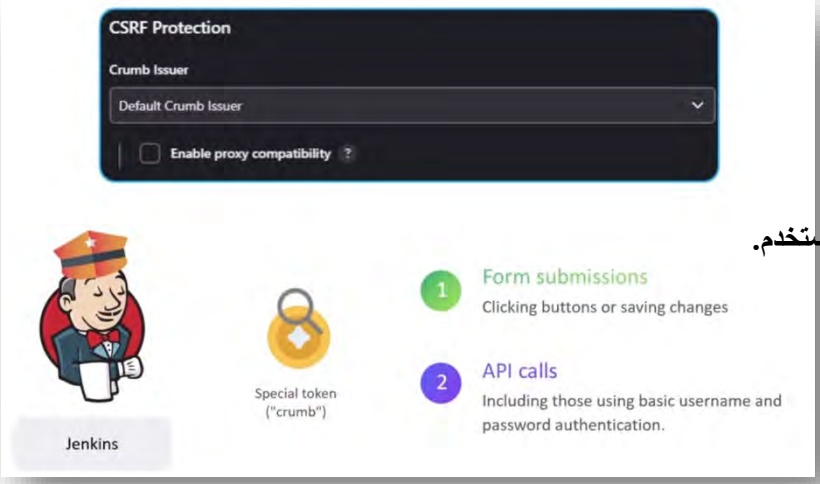
- Jenkins عنده ميزة اسمها : Prevent Cross-Site Request Forgery exploits بتكون مفعّلة افتراضيًا.

🔑 **إزاي بتحميك؟**

- Jenkins بيضيف token خاص (crumb) جوه كل فورم أو طلب بيغيّر بيانات.
  - وبالتالي فأي طلب جاي من خارج Jenkins من غير التوكن ده بيتمنع.
- الـ token ده بيتحقق في:
  - أي Form submission (زي الضغط على "Save")
  - أي API request حتى لو بـ username/password

💬 **المصطلح المهم:**

الناس أحيانًا يسموه : CSRF Protection أو Crumb Issuer ← الاتنين نفس الحاجة.



🧱 طريقة عملها:

كل ما تبعت Request لتغيير حاجة في Jenkins :

- Jenkins بيفتح التوكن ويشيّك عليه.
- لو مش موجود أو مش صحيح ← الطلب مرفوض.
- ده يمنع أي سكربت أو موقع خارجي إنه يستغل جلسة المستخدم.

💡 خلاصة الجزء ده:

- ✅ لازم تسيب CSRF Protection Enabled
- ⚠️ متعطلهاش إلا لو بتعمل Test على بيئة داخلية مش Production
- 🧠 ولو عطلتها مؤقتًا، ارجع فعّلها فورًا بعد الانتهاء

🔒 الخلاصة النهائية:

الإعداد	الوصف	الغرض	الحالة المثالية
Markup Formatter	يتحكم في تنسيق النصوص داخل Jenkins	يمنع إدخال أكواد خبيثة (XSS)	استخدم Plain Text أو Safe HTML فقط
CSRF Protection	يمنع طلبات غير مصرح بيها باسم المستخدم	يحمي Jenkins من تنفيذ أوامر عن بُعد	دايمًا مفعّل ✅

Markup Formatters In More Details

🎯 أولًا: فكرة عامة

- ❑ الميزة دي مسؤولة عن الطريقة اللي Jenkins بيعرض بيها النصوص في واجهته زي:
  - الـ System Message اللي بيظهر في الـ Dashboard
  - وصف (Description) الـ Jobs / Builds / Views
  - وأي Text Field تقدر تكتب فيه كلام يظهر في واجهة Jenkins
- ❑ الفكرة إن Jenkins ممكن يسمحلك تستخدم HTML tags (زي <b>, <i>, <ul>, ... إلخ) عشان تدي شكل أو تنسيق للنص
- ❑ لكن ده ممكن يكون خطر لو استخدمته غلط، لأن فيه احتمال إن المستخدم يضيف كود JavaScript ضار(XSS attack)
  - علشان كده Jenkins بيقدم أنواع مختلفة من الـ Markup Formatter بتوازن بين الجمال والأمان 👇

🧩 ثانيًا: إعدادات الـ Markup Formatter .. أنواعها والفرق بينهم

➢ في Jenkins عندك نوعين أساسيين:

1️⃣ Plain Text (الأفضل للـ Prod)

- ده الإعداد الافتراضي.
- بيمنع أي تنسيق HTML
- أي كود HTML بتكتبه بيظهر كـ نص عادي.
- آمن جدًا لأنه بيه escape لأي HTML tags

🧱 مثال عملي:

- لو كتبت في الـ System Message : Welcome!
  - Jenkins هيعرضها كده بالضبط: Welcome!
  - من غير أي تأثير ولا ألوان.

  وده لأن Jenkins بيعتبرها مجرد نص عادي مش كود HTML

2️⃣ Safe HTML

- ده بيسمح ببعض أكواد HTML البسيطة اللي ملهاش خطر.
- Jenkins في الوضع ده بيعمل Sanitization للنص.
  - يعني بيشيل أي عناصر أو أكواد خطيرة (زي <script> أو <iframe>)
- النتيجة إنك تقدر تدي شكل منظم وجميل للنصوص، من غير ما تفتح باب للهجمات.

🧱 مثال عملي:

- نفس الكود اللي فوق: Welcome!
  - بعد ما تختار Safe HTML ، Jenkins هيعرض الرسالة باللون الأحمر وبالـ bold : Welcome!
  - بس لو حاولت تضيف سكربت زي: <script>alert('hacked!')</script>
  - Jenkins هيشيله تلقائيًا.

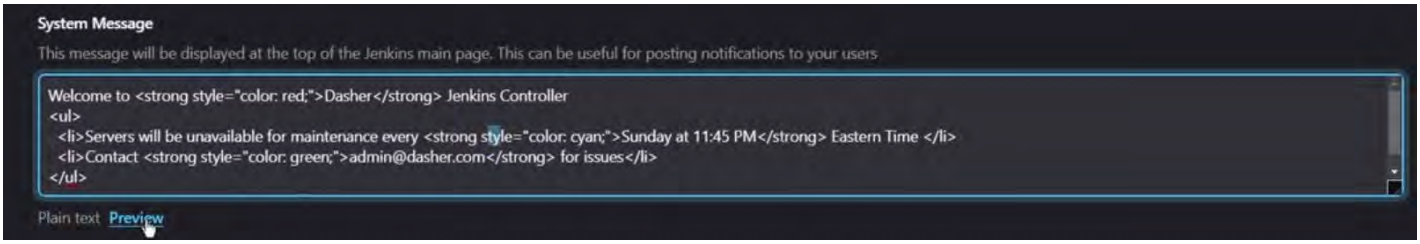
🧰 ثالثًا: خطوات عملية لتغيير الـ Markup Formatter

➢ تعالى نمشي خطوة بخطوة زي ما اتعمل في الفيديو 👇

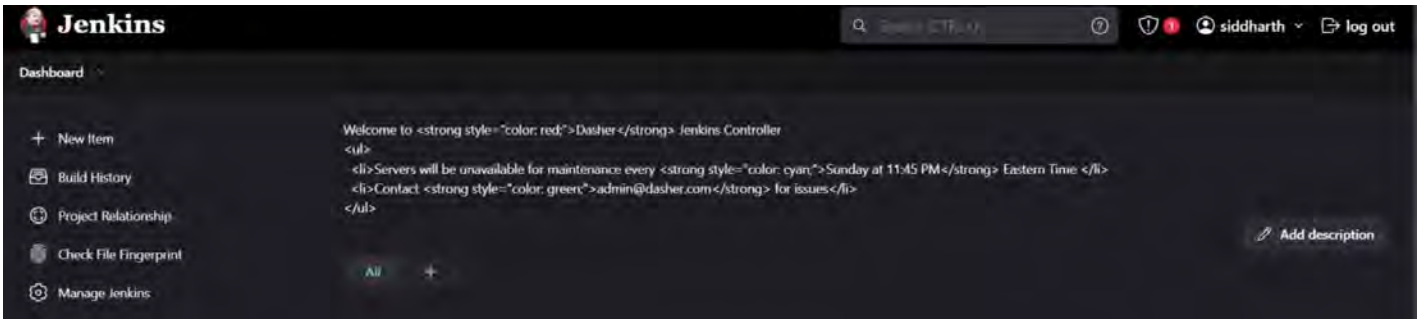
📋 الخطوة 1:

- ادخل على لوحة التحكم : Manage Jenkins → System

- هتلاقي هناك الـ System Message اللي بيظهر فوق الـ Dashboard
  - جرب تكتب HTML جوه الرسالة، زي مثلًا:



- لو سيبت الإعداد زي ما هو(Plain text) ، لما تعمل Save أو Preview
  هتلاقي Jenkins بيعرض الكلام ده كنص عادي، من غير ألوان ولا بولد.

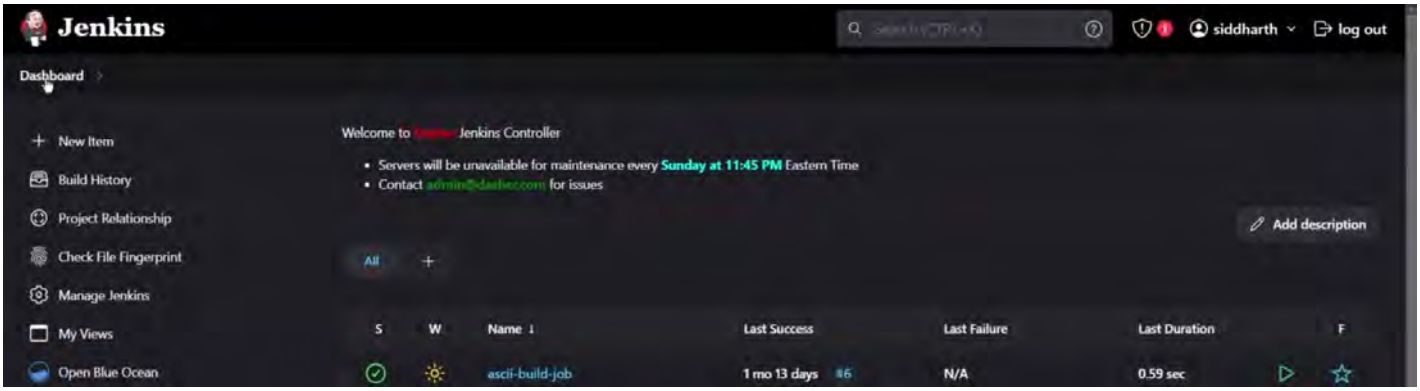


الخطوة 2 :

- روح على : Manage Jenkins → Configure Global Security
- وتحت قسم Markup Formatter غيّر القيمة من: Plain text → Safe HTML

الخطوة 3:

- اضغط Save ورجع تاني على الـ Dashboard
- دلوقتي هتلاحظ إن الرسالة اتغير شكلها فعلًا — بقت فيها ألوان وتنسيقات.



ملاحظات مهمة و Use Cases

- ممكن تستخدم نفس الفكرة في وصف (Description) أي Job أو Build
  مثال:
  - توضح مراحل الـ Pipeline بألوان.
  - أو تضيف جدول بسيط يوضح النتائج.
- Jenkins بيسمح بالـ Safe HTML في الأماكن اللي بيتم عرضها للمستخدمين فقط.
  يعني ما ينفعش تستخدمها لتشغيل كود أو تنفيذ أوامر.
- لو بيئتك فيها أكتر من Admin ، لازم تتأكد إن محدش يضيف Tags فيها محتوى غير موثوق.

🧠 Real-world Best Practices

1. ✅ استخدم Plain Text كإعداد افتراضي في بيئة Production
   - وغيّره لـ Safe HTML فقط في حالة الضرورة (زي Dashboard توضيحي أو View منسق)
2. ⚠️ متستخدمش Plugins خارجية لتنسيقات مخصصة إلا لو مضطر جدًا.
3. 🧹 راجع بانتظام المحتوى اللي فيه HTML في Jenkins dashboard عشان تتأكد مفيهوش أكواد غريبة.
4. 🔒 دايمًا اربط الإعداد ده بإعداد الأمان التاني اللي هو CSRF Protection
   (علشان تمنع استغلال أي HTML مدعوم في تنفيذ أوامر خبيثة)

📜 الملخص

العنصر	الوصف	ملاحظات
Markup Formatter	يتحكم في طريقة عرض النصوص والوصف داخل Jenkins	بيأثر على شكل الـ Dashboard والـ Jobs
Plain Text	يمنع أي HTML، آمن جدًا	الأفضل للـ Production
Safe HTML	يسمح ببعض التنسيقات البسيطة	مناسب للـ Views التوضيحية
الهدف الأساسي	التوازن بين الجمال والأمان	حافظ على Sanitization شغال دائمًا

Acess Control for Builds

🎯 الفكرة العامة

- ❑ الجزء ده بيتكلم عن Access Control داخل Jenkins Builds .. يعني إزاي نتحكم في الـ Build بحيث نربطه بنفس صلاحيات اليوزر اللي عامل الـ Build ده .
- ❑ بشكل افتراضي، Jenkins لما بيشغّل أي Build ، بيشغّله كـ System user داخلي عنده كل الصلاحيات (create/delete jobs, run builds, cancel builds ... إلخ)
- ❑ وده خطر جدًا، لأن أي سكريبت داخل الـ Build ممكن يعمل حاجات غير مسموحة لو تمّ اختراقه أو لو كان فيه Misconfiguration
- ❑ علشان كده بيتم استخدام Authorize Project Plugin عشان نربط كل Build بصلاحيات المستخدم اللي شغّله.

🧠 المرحلة 1: فهم الوضع الافتراضي

الوضع الافتراضي في Jenkins :

- أي Build بيتشغّل بـ System user :
- الـ System user عنده صلاحيات كاملة (Run on any node, create/delete jobs ... إلخ)

📌 خطر عملي:

لو سكريبت الـ Build فيه أمر زي rm -rf /var/lib/jenkins/jobs ← هيشتغل بدون أي منع لأن المستخدم عنده صلاحيات كاملة.

⚙️ المرحلة 2: مقدمة الـ Plugin المستخدم

- ➢ الـ Plugin اللي بنحتاجه هنا هو: Authorize Project Plugin 👉
- ➢ بيسمحلك تختار مين المستخدم اللي الـ Build هيتنفّذ باسمه (system - anonymous - specific user - trigger user)

🧩 المرحلة 3: اختبار ما قبل التهيئة

- ➢ ننشئ Job بسيط باسم: npm-version-test

ونضغط على: Build Now

📊 لما تفتح الـ Console Output هتلاقي حاجة زي:

- o Started by user Siddharth Barahalikar & Running as SYSTEM

🧠 النتيجة:

الـ Build بدأ باسم "Siddharth" لكن اتنفّذ بصلاحيات System user

---

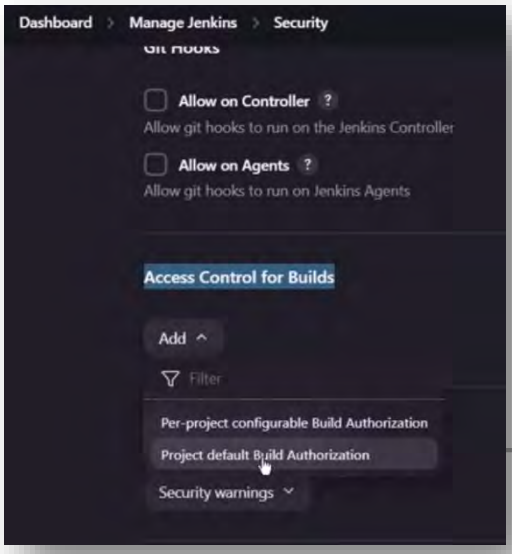
💻 المرحلة 3: تثبيت الـ Plugin

الخطوات العملية:

1. نفتح Jenkins Dashboard
2. نروح على : Manage Jenkins → Manage Plugins → Available
3. نبحث عن: Authorize Project
4. نضغط: Install without restart

✅ بعد التثبيت، الـ Plugin بيضيف إعدادات جديدة في: Manage Jenkins → Configure Global Security

---

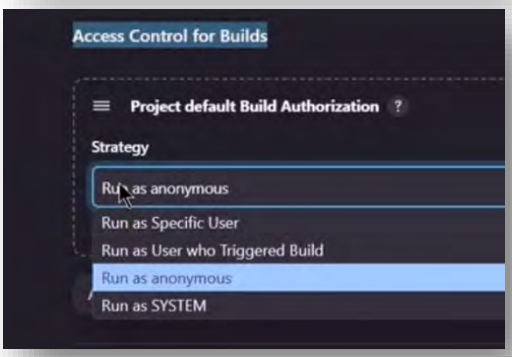


🔐 المرحلة 5 : تفعيل الـ Access Control

- نروح إلى : Manage Jenkins → Configure Global Security
- هتلاقي Section جديد اسمه: Access Control for Builds
- تضغط على: Add
- وهنا عندك خيارين:
  - Project Default Build Authorization
  - Project Configurable Build Authorization

➢ نختار دلوقتي الأول (Project Default Build Authorization)

⚙️ المرحلة 6: اختيار الاستراتيجية (Strategy)



هتظهر قائمة فيها خيارات زي:

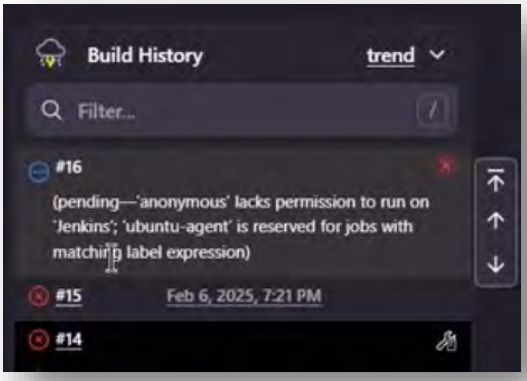
- Run as System (الوضع الافتراضي)
- Run as Anonymous
- Run as Specific User
- Run as User who Triggered the Build

نختار مبدئيًا:

Run as Anonymous

ونضغط: Apply → Save

🧪 المرحلة 7: تجربة التنفيذ بعد التغيير



➢ نرجع للـ Job "npm-version-test" ونبني تاني.

➢ اللي هيحصل:

- الـ Build مش هيبدأ أصلاً.
- هتلاقي الحالة Pending :
- والـ Console Log بيقول:

anonymous lacks permission to run on Jenkins

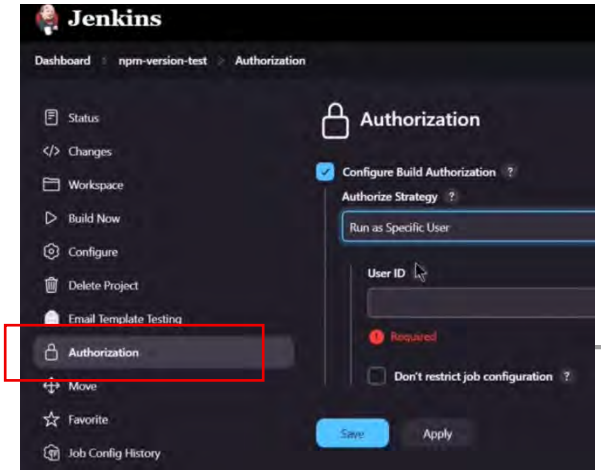
📌 السبب:

المستخدم Anonymous ملوش صلاحية تشغيل الـ Build على الـ Node

🔄 المرحلة 8: تعديل الإعداد للمستوى المحلي (Project level)

➢ نرجع إلى : Manage Jenkins → Configure Global Security

➢ ونضيف تاني Access Control جديد لكن المرة دي: Project Configurable Build Authorization



- وده بيسمحلك تختار من داخل كل Job استراتيجية مختلفة (بيظهرلك اوبشن في الـ dashboard اسمه Authorization)
- هنحدد السماح بالاستراتيجيات التالية:
  - Run as System
  - Run as Specific User
  - Run as User who Triggered the Build

➢ ونضغط Save

🧱 المرحلة 9: إعداد Job معين

- نفتح Job "npm-version-test"
- نروح إلى : Configure → Build Authorization
- ونختار: Run as user who triggered the build
- وبعدين نضغط: Save → Build Now
- النتيجة في الـ Console :

- Started by user Siddharth Barahalikar
- Running as Siddharth Barahalikar

💡 كده كل صلاحيات تنفيذ الـ Build بقت بنفس مستوى صلاحيات المستخدم اللي شغّله.

🧰 ملاحظات عملية (Best Practices + Real Use Cases)

🔸 Best Practice 1 :
متشغلش أي Job بصلاحيات الـ System إلا لو محتاج فعلاً (زي Admin maintenance jobs)

🔸 Best Practice 2 :
استخدم دايمًا "Run as user who triggered the build" في بيئات Production عشان تحمي Jenkins من إساءة الاستخدام.

🔸 Best Practice 3 :
لو عندك Jobs بتتعامل مع API أو Secrets ، استخدم Credentials Bound to that user فقط.

🔸 Real Case :
في شركات كبيرة، بيستخدموا Authorize Project Plugin مع LDAP/AD authentication عشان كل مستخدم يبني بس المشاريع المصرح ليه بيها.

📋 الملخص النهائي

العنصر	الوصف
الهدف	التحكم في صلاحيات تنفيذ البيلد
الأداة	Authorize Project Plugin
الاستراتيجيات	System, Anonymous, Specific User, Trigger User
التوصية	Run as user who triggered the build
الفايدة	حماية Jenkins من تنفيذ أوامر غير مصرح بيها

Job Restrictions

🎯 الفكرة العامة

- ❑ هنتكلم عن الـ **Job Restrictions Plugin** في Jenkins
- ❑ البلجن ده من أهم إضافات الـ Security Hardening في Jenkins لأنه بيخلّيك تتحكم في:
  - أي Jobs مسموح لها تشتغل على Node معيّنة.
  - أو مين المستخدمين اللي مسموح ليهم يشغّلوا Job معيّنة.
  - وكمان تقدر تستخدم شروط مركّبة (AND / OR / NOT) لتخصيص الصلاحيات.
- ❑ بمعنى أبسط:
  - البلجن ده بيمنع أي Job من إنها تشتغل على Node معينة إلا لو بتطابق شروط محددة انت حاططها.

🧠 المرحلة 1: تثبيت البلجن (Plugin Installation)

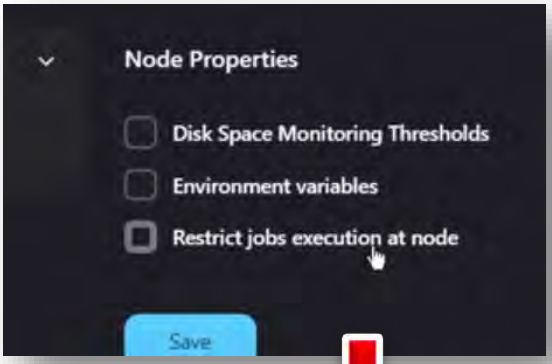
🔧 الخطوات:

1. افتح Jenkins Dashboard
2. روح على : Manage Jenkins → Manage Plugins → Available
3. اكتب في البحث: **Job Restrictions**
4. اختار البلجن واضغط: Install without restart
5. بعد التثبيت يفضل تعمل Restart لـ Jenkins

✅ بعد الريستارت ← البلجن هيبقى متاح داخل إعدادات الـ Nodes

⚙️ المرحلة 2: التهيئة على مستوى الـ Node (Node Configuration)

البلجن ده بيشتغل على مستوى الـ Node Configuration مش على مستوى الـ Job

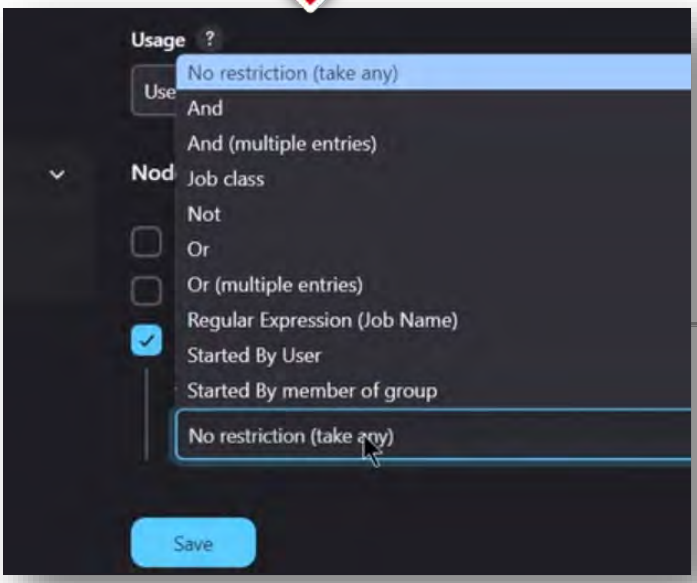


📍 الخطوات:

1. من الـ Dashboard : Manage Jenkins → Nodes
2. اختار الـ Node اللي عايز تقيّدها (مثلاً Built-in Node)
3. اضغط: Configure
4. هتلاقي Section اسمه: Restrict job execution on this node
5. فعِّل الاختيار ده ✅

وبعدها اضغط: Add restriction

🧩 المرحلة 3: إضافة أول شرط (Restriction Condition)



🎯 الشرط الأول: السماح فقط للـ Jobs اللي اسمها يبدأ بـ Dasher_

1. نختار : **Regular Expression (Job name)**
2. نكتب الـ Regex pattern : ^Dasher_.*
   - معناها: اسم الـ Job لازم يبدأ بـ "Dasher_"
3. اضغط : Save

🧪 المرحلة 4: اختبار الشرط الأول (Testing the Restriction)

- جرّب تشغّل Job اسمها npm-version-test
  ⛔ النتيجة: هتفضل في حالة Pending ومش هتبدأ.
- 📋 السبب:
  اسمها لا يطابق الـ Regex اللي بيبدأ بـ Dasher_

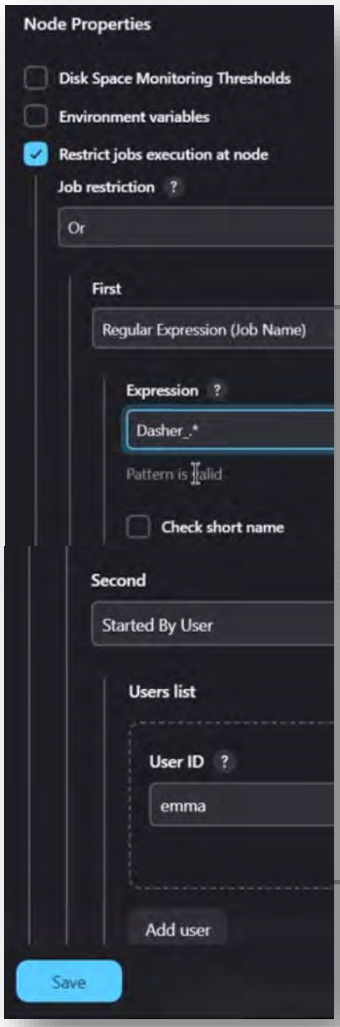
🧱 المرحلة 5: إنشاء Job مطابقة (Matching Job)

🔧 الخطوات:

1. من Dashboard اختار : New Item → Freestyle project
2. الاسم: Dasher_test_job
3. Build Step : echo "Hello from restricted job"
4. اضغط: Save → Build Now

✅ النتيجة:

الـ Job هتشتغل بنجاح لأن اسمها بيطابق الشرط ^Dasher_



🧩 المرحلة 6: إضافة أكثر من شرط (AND / OR / NOT Conditions)

➢ دلوقتي هنخلّي البلجن يسمح بالحالتين دول:

1. الـ Job تبدأ بـ Dasher_
2. أو المستخدم اللي شغّلها اسمه Emma

📍 الخطوات :

1. ارجع إلى : Manage Jenkins → Nodes → Built-in Node → Configure
2. في قسم Job Restrictions :
   - اختار: OR Condition
3. ضيف أول شرط: Regular expression on job name → ^Dasher_.*
4. ضيف شرط تاني: Started by user ID → Emma
5. اضغط: Save

🧪 المرحلة 7: تجربة الحالة الثانية (User-based Restriction)

1. لو المستخدم الحالي مش "Emma" ← الـ Job مش هتشتغل (هتفضل Pending)
2. لو عملت Logout ودخلت بحساب Emma ← الـ Job هتشتغل بنجاح حتى لو اسمها مش يبدأ بـ Dasher_.

📊 النتيجة:

- OR = أي شرط فيهم كفاية لتشغيل الـ Job
- AND = لازم الشرطين يتحققوا.

🧠 المرحلة 8: حالات استخدام حقيقية (Real-World Scenarios)

1. 🔸 حماية Nodes الحساسة:
ممكن تقيّد الـ Nodes اللي بتتعامل مع Production بحيث تشتغل عليها بس Jobs معينة تبدأ باسم Prod_.
2. 🔸 تقييد حسب المستخدم:
ممكن تسمح فقط لفريق DevOps بتشغيل Jobs على Agent معين.
3. 🔸 بيئة Multi-Tenant :
في Jenkins واحد يخدم فرق مختلفة، البلجن ده يضمن إن كل فريق يشتغل على Nodes معينة تخصه فقط.

🧰 Best Practices

✅ استخدم أسماء Jobs واضحة مرتبطة بالفريق (زي Dev_, Prod_, QA_)
✅ دايمًا اختبر القيود الجديدة على بيئة Test قبل تطبيقها على Production
✅ استخدم Conditions مركّبة بحذر — خصوصًا الـ NOT لأنها ممكن تمنع Jobs مهمة بالغلط.
✅ احتفظ بنسخة Backup من إعدادات Nodes قبل أي تعديل.

📄 الملخص النهائي

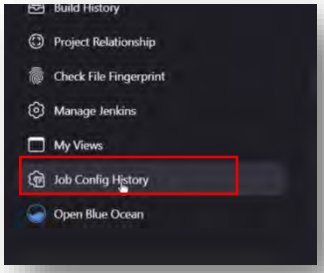
العنصر	الوصف
الهدف	تقييد تشغيل الـ Jobs على Nodes معينة بشروط محددة
البلجن	Job Restrictions Plugin
نوع الشروط	Regex job name, User ID, AND/OR/NOT logic
مثال عملي	^*_Dasher أو User = Emma
النتيجة	فقط الـ Jobs اللي تطابق الشروط تشتغل
الاستخدام العملي	حماية الـ Nodes من تشغيل Jobs غير مصرّح بيها

Job Configuration History Plugin

🎯 الفكرة العامة:

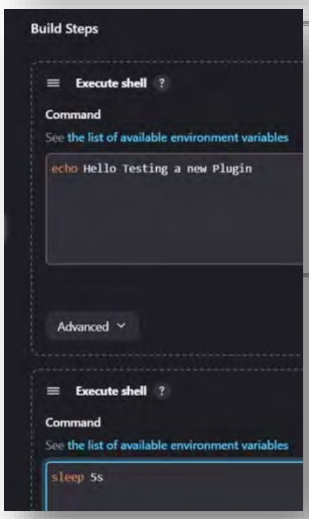
- ❑ الـ **Job Configuration History Plugin** في Jenkins بيعمل زي نظام version control لكن مخصوص لملفات الإعدادات (configurations) الخاصة بالـ jobs والـ system نفسه.
- ❑ يعني أي تعديل أو حذف بيحصل، Jenkins بيحتفظ بنسخة منه بحيث تقدر ترجع في أي وقت وتسترجع النسخة القديمة.

🧩 الفكرة بالتفصيل:



1. أول ما تثبت الـ plugin وتعمل restart للـ Jenkins ، هتلاقي في الـ Dashboard اختيار جديد اسمه "Job Config History"
2. لما تدخل عليه، هيعرضلك كل التغيرات اللي حصلت:
   - ممكن تختار تشوف بس تغييرات الـ system
   - أو بس تغييرات الـ jobs
   - أو الـ jobs اللي اتعملها Create
   - أو الـ jobs اللي اتعملها Delete

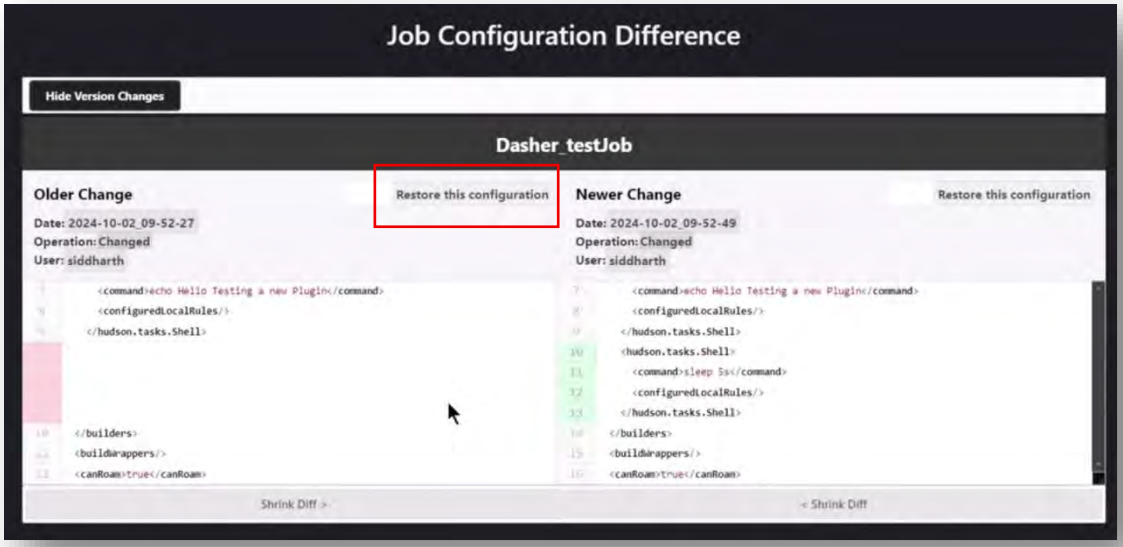
🧠 مثال عملي:



- المستخدم هنا عمل شوية تغييرات في job اسمه Dashboard Test Job
  - أولاً عدّل أمر التنفيذ (echo command) ، وبعدين ضاف أمر تاني (sleep 5)
- الـ plugin بيحتفظ بنسخة من ملف الإعداد (config.xml) قبل وبعد التعديل.

🔁 استرجاع النسخ القديمة:

- في صفحة الـ job نفسها، هتلاقي برضو اختيار Job Config History
- هناك تقدر:
  - تشوف الفرق بينهم (show diffs) ← هيورّيك السطر الجديد اللي اتضاف .
  - تعمل Restore لأي نسخة سابقة.
- لما تعمل Restore ، Jenkins هيرجع الملف زي ما كان في النسخة القديمة (مثلاً قبل إضافة أمر sleep)



🗑️ استرجاع job محذوف:

لو job اتمسحت بالغلط:

- ارجع للـ "Show only deleted jobs" ← Job Config History ← Dashboard
- هتلاقي كل الـ jobs اللي اتمسحت، وتقدر تعمل Restore لأي واحدة فيهم.

📂 فين بيتخزن الكلام دا؟

- كل نسخ الإعداد بتتخزن في المسار : /var/lib/jenkins/config-history .. ولكن تقدر تغيره عادي من خلال :
  - Manage Jenkins ← System ← Job History Config ←
- عموما جوا المسار الافتراضي دا هتلاقي:
  - فولدرات لكل job
  - جواها ملفات:
    - config.xml ← نسخة الإعداد.
    - history.xml ← تفاصيل مين عدّل وإمتى حصل التعديل.

✅ الخلاصة:

- 🔹 الـ Plugin دا مهم جدًا لأنه بيحافظ على سجل تاريخي لكل التغييرات في Jenkins
- 🔹 مفيد لو حصل تعديل غلط أو job اتحذف بالغلط ← تقدر ترجعه فورًا.
- 🔹 بيشتغل أوتوماتيك بدون أي إعدادات معقدة، لكن ممكن تغيّر مكان تخزين البيانات لو حبيت.

Build Monitor View

🎯 الفكرة العامة

- ❑ الجزء ده بيتكلم عن **Build Monitor View Plugin** في Jenkins ، وده Plugin بصري (Visual Plugin) بيساعدك تتابع حالة كل الـ jobs في Jenkins من شاشة واحدة.
- ❑ يعني بدل ما تدخل على كل Job وتشوف حالتها لوحدها (نجحت؟ فشلت؟ بتبني؟)، الـ Plugin دا بيعرض كل دا في Dashboard واحد بيحدّث نفسه تلقائي كل كام ثانية،
- ❑ وبيوريلك الحالة بالألوان:
  - 🟢 أخضر = Build ناجح
  - 🔴 أحمر = Build فشل
  - 🟡 أصفر = Build شغال

🧩 الشرح التفصيلي خطوة بخطوة

1. 🧠 تجهيز الـ Plugin

- أول خطوة: نثبت الـ Build Monitor View Plugin
- بعد التثبيت لازم نعمل Restart لـ Jenkins Controller عشان التحديث يتطبق.

🔧 معلومة مهمة

أي Plugin بيتعامل مع واجهة Jenkins لازم تعمل بعده Restart ، عشان يتسجل داخل الـ UI

---

2. 🧱 الهدف من الـ Plugin

الـ Plugin دا بيعمل:

- عرض حالة كل الـ jobs اللي تختارها (Running / Success / Failed)
- تحديث تلقائي كل عدة ثوانٍ.
- بيعرض كمان اسم الشخص اللي عمل التغيير أو مسؤول عن الـ job
- بيشتغل كـ Dashboard حيّ لمتابعة كل الـ pipelines

---

3. ⚙️ خطوات إنشاء Build Monitor View

🧩 الخطوة 1: إنشاء View جديدة

من الـ Dashboard :

1. اضغط على "+" أو"New View"
2. اختار نوع الـ View → Build Monitor View
3. اديله اسم مميز زي : Dashboard CI Build Monitor View
4. اضغط Create

🧩 الخطوة 2: ضبط الإعدادات

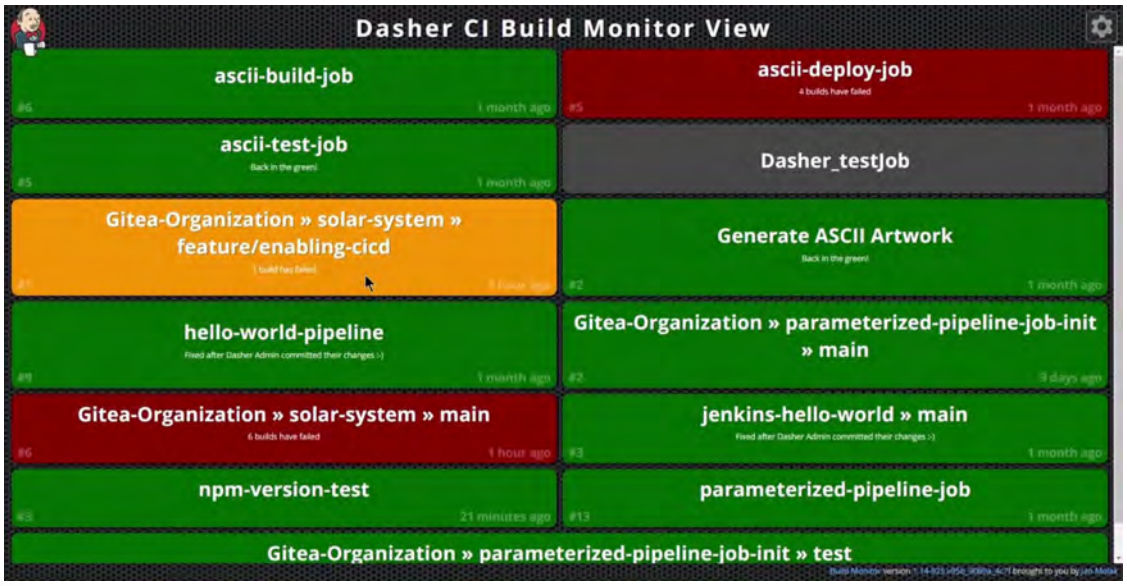
في الصفحة الجديدة:

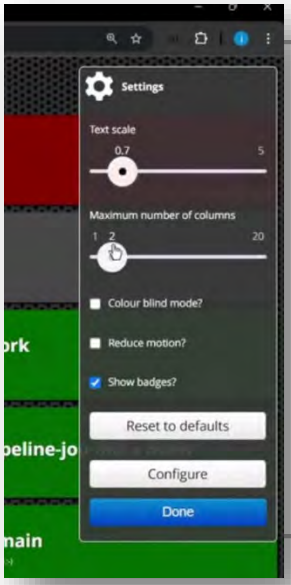
- ممكن تضيف وصف (Description) للـ View
- بعدين تختار الـ Jobs اللي عايز تتابعها.

عندك 3 اختيارات:

1. ✅ تختار كل الـ jobs ← لو عايز تتابع كل حاجة.

2. 🧩 تختار subset (جزء معين) ← لو عايز بس CI/CD jobs معينة.
3. 🔍 تستخدم **Regex** علشان تضم الـ jobs اللي اسمها بينطبق على نمط معين: **$Backend.^***
   - دا مثل لو عايز تضم كل الـ jobs اللي بتبدأ بـ "Backend"
4. بعد كده اضغط : Apply





4. 🎨 تخصيص واجهة العرض (Display Settings)

من نفس واجهة الـ View :

- ممكن تزود عدد الأعمدة: Columns → 1 / 2 / 3 / 4
- ممكن تكبر الخط (Text Scale)
- ممكن تفعل:
  - Colorblind Mode 🎯 (لضعاف الألوان)
  - Reduce Motion 🌀 (لأداء أفضل)
  - Show Badges 🏷️ (لعرض Build Numbers)
- ولو لخبطت الإعدادات ← اضغط "Reset to Default"

5. 🧩 متابعة الحالة أثناء التشغيل

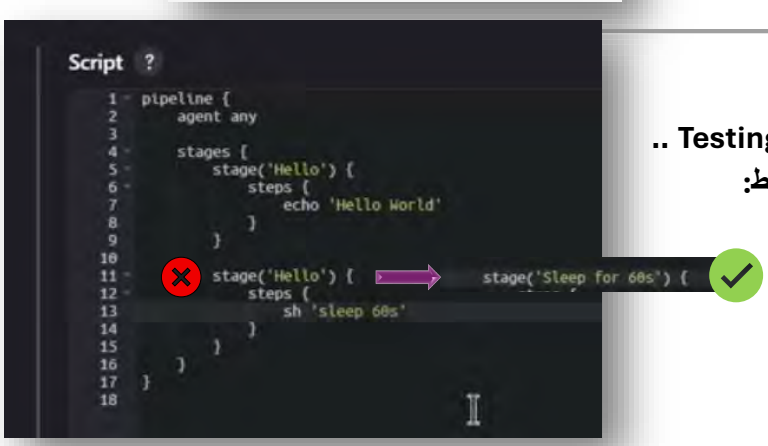
- الـ Plugin دا متكامل مع Pipeline Plugin، يعني لو عندك Job Pipeline ، هيعرض كل Stage داخلها أثناء التنفيذ.
- مثلاً لو عندك Pipeline فيه:

```
stage('Build') { ... }
stage('Test') { ... }
stage('Deploy') { ... }
```

  - هيعرضهم لحظة بلحظة في الـDashboard ، وهيكتبك كمان الزمن اللي كل مرحلة أخدته. ⏱️

6. 🧱 مثال عملي

- هنا عمل Job جديدة اسمها Testing_Build_Monitor_plugin ..
- ونوعها Pipeline .. وبعدين كتب سكريبت Pipeline بسيط:



⚠️ ملاحظة مهمة:

- **لازم كل stage يكون ليه اسم مختلف،**
  لأن Jenkins ما بيقبلش Stage Name مكرر.
- وبعدين Apply → save → Build Now

7. 🧩 متابعة النتيجة في Build Monitor View

لو الـ job مش ظهرت في الـ Dashboard :

- روح للـ Build Monitor View
- اضغط "Edit View"
- أضف الـ job يدويًا من القائمة.

هتبدأ تلاحظ:

- اسم الـ stage الحالي.
- الزمن اللي عدى.
- الحالة (جاري التنفيذ أو مكتمل).
- اللون بيتغير حسب الحالة :
  - 🔵 جاري التشغيل.
  - 🟢 نجح.
  - 🔴 فشل .

---

8. 🧠 بعد انتهاء التنفيذ

لما يخلص الـ job :

- اللون بيتحول لـ أخضر.
- بيظهر آخر Build Message
- بيعرضلك كمان:
  - اسم الفرع (branch)
  - عدد الـ builds اللي فشلت قبل كده.
  - آخر شخص عدّل على الـ job

Best Practices 💡

1. اقسم الـ Views حسب نوع المشروع
   مثلاً:
   - "Frontend Builds View"
   - "Backend Builds View"
   - "Production Deployments View"
2. خلي شاشة Jenkins Monitor دايمًا ظاهرة في التيم روم بحيث الناس تشوف حالة الـ builds أول بأول.
3. فعّل Auto Refresh (بيكون موجود ضمن الإعدادات) ← بيخلي الـ dashboard يحدّث نفسه تلقائي كل دقيقة.
4. استخدم Regex علشان تضم المشاريع الجديدة تلقائيًا.
5. لو السيرفر عليه حمل عالي، قلل معدل التحديث (refresh rate) لتجنب الضغط على Jenkins

---

الملخص النهائي

النقطة	الشرح المختصر
Plugin	Build Monitor View
الهدف	متابعة حالة كل الـ jobs من شاشة واحدة
الإعداد	jobs → Apply اختار الـ → Create → Build Monitor View
العرض	ألوان – مراحل – حالة التنفيذ – اسم المستخدم
التحديث	تلقائي كل عدة ثوانٍ
متكامل مع	Pipeline Plugin (يعرض الـ stages)
مفيد في	بيئات CI/CD عشان تشوف حالة المشاريع لحظة بلحظة

Jenkins Scaling Capacity Planning

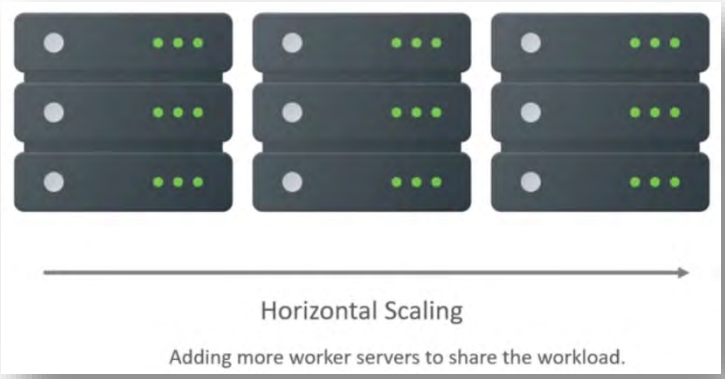
🎯 أولاً: الفكرة العامة

- ❑ لما بيزيد عدد الـ Jobs والمشاريع في Jenkins ، بيبدأ السيرفر يعاني :
  - ➢ CPU عالي 🔥 ، Memory بتتملى، Builds بطيئة، أو حتى UI بيقف.
- ❑ فبييجي السؤال : "نقوّي السيرفر نفسه؟ ولا نزود عدد السيرفرات؟"
  - ➢ وده اللي بيتسمّى: Scaling Jenkins 👉
    - يعني نزوّد قدرته عشان يتحمّل ضغط أكتر.

🧩 ثانياً: طريقتين رئيسيتين لـ Scaling

1. ⚙️ Horizontal Scaling (التوسّع الأفقي)

- ➢ الفكرة : نضيف أكتر من سيرفر Jenkins Agent (Worker Nodes) .. وكل واحد منهم يشيل جزء من الـ Jobs
- ➢ 🎯 المبدأ:
  - بدل ما جهاز واحد يعمل كل الشغل، نقسّمه على أكتر من جهاز.
- ➢ 💪 الفوايد:
  - سهل التوسيع : تضيف Agent جديد وخلاص.
  - Fault Tolerant : لو Agent واحد وقع، الباقي يكمّلوا.
  - Performance أعلى: كل Job بتتوزع على Node مختلف.
- ➢ ⚠️ العيوب:
  - محتاج إدارة أكتر (Agents كتير)
  - إعدادات وتحديثات على أكتر من Machine
  - ممكن يبقى مكلف لو الحمل مش كبير .



---

2. 🖥️ Vertical Scaling (التوسّع الرأسي)

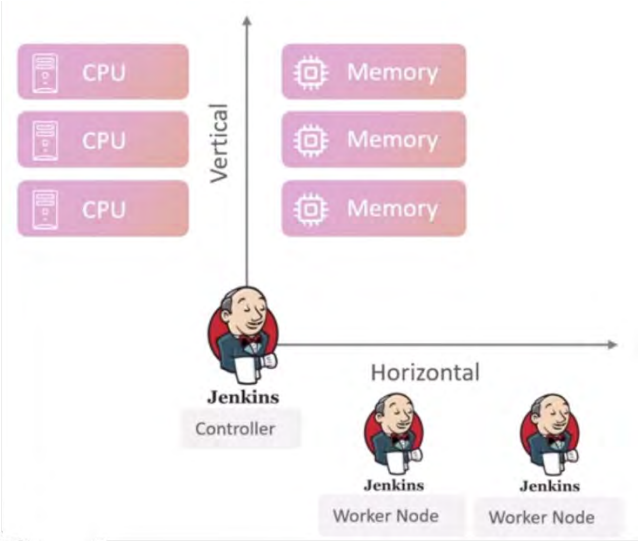
- ➢ الفكرة : نزوّد إمكانيات السيرفر الحالي (نفس الجهاز).
- ➢ 🎯 المبدأ:
  - نزوّد RAM و CPU للجهاز اللي شغال عليه Jenkins Controller
- ➢ 💪 الفوايد:
  - أسهل في الإدارة (جهاز واحد بس).
  - أرخص في البداية للمشاريع الصغيرة.
- ➢ ⚠️ العيوب:
  - فيه حد فيزيائي : مش هتقدر تزود رام أو CPU إلى ما لا نهاية.
  - Single Point of Failure : لو الجهاز دا وقع ← Jenkins كله هيقع.
  - مع زيادة الرام جدًا، Garbage Collection في JVM بياخد وقت أطول ← الأداء يقل.

🧱 ثالثاً: تطبيق الكلام دا على Jenkins (الـ Horizontal Scaling هو المفضل)

- خلينا نتخيل إن شركتك شغالة بـ Jenkins Controller واحد، وكل الـ plugins ، integrations، pipeline jobs شغالة عليه.
- بعد فترة:
  - الفريق كبر.
  - عدد الـ builds زاد.
  - كل الناس فاتحين Jenkins UI
  - CPU و RAM بدأوا يتخنقوا. 💀
- ممكن تفكر: "نزود RAM وCPU؟"

تمام، ولكن:

- بعد فترة .. الزيادة دي مش هتكفي.
- الـ Garbage Collection (GC) هيبدأ يستهلك وقت كبير.
- وفي النهاية عندك جهاز واحد بس = نقطة فشل واحدة.
- 🎯 الحل الأفضل؟
  - Horizontal Scaling



⚙️ رابعاً: الحل المثالي Jenkins Distributed Architecture

- 💡 المفهوم : نعمل Controller + Agents
  - Controller : يدير الـ pipelines
  - Agents (Nodes) : تشغل الـ builds فعليًا.
- يعني الـ Controller بس بيشغل الـ Jobs على Agents تانية.
- الشكل العام :

```
        +--------------------+
        | Jenkins Controller |
        +--------------------+
                  |
    -----------------------------------
    |                |                |
+----------+     +----------+     +----------+
| Agent #1 |     | Agent #2 |     | Agent #3 |
+----------+     +----------+     +----------+
```

✅ المزايا:

- الحمل بيتوزع.
- Jenkins بيبقى أسرع.
- لو Agent وقع، الباقي يكمّلوا.
- تقدر تضيف Agents جديدة بسهولة وقت الضغط.

💾 خامساً : ما تنساش الـ Storage

- الـ Jobs بتولّد:
  - Logs 📜
  - Artifacts 📦 (زي binaries, reports, builds)
- فلو بتعمل Scaling ، لازم يكون عندك Scalable Storage Solution زي:
  - AWS S3
  - NFS Shared Volume
  - أو أي Storage مركزي Jenkins Agents يقدروا يوصلوا له.

Best Practices 💡

1. ✅ استخدم Distributed Jenkins Setup (Controller + Multiple Agents)
2. 🚀 استخدم Kubernetes Jenkins Agents لو عايز Auto-Scaling (Pods تتخلق وتتمسح أوتوماتيك)
3. 🧱 خفّف الضغط عن الـ Controller ← متشغلش عليه Builds
4. 🧠 راقب الأداء (CPU/RAM Usage) من Monitoring Tools زي Prometheus + Grafana
5. 🗄️ استخدم External Storage عشان اللوجات والـ artifacts
6. 🔄 اعمل Backup دوري للـ Controller Configuration

الملخص النهائي 📄

المفهوم	الشرح السريع
⚙️ Horizontal Scaling	نضيف أكتر من Agent – توزع الشغل – أفضل للأحمال الكبيرة
🖥️ Vertical Scaling	نزود RAM وCPU – سهل بس محدود
🧱 في Jenkins	Controller يدير + Agents تشغل
⚠️ Single Point of Failure	مشكلة في Vertical Scaling
💾 Storage	استخدم Storage قابل للتوسّع للـ Logs و Artifacts
🚀 الأفضل للمؤسسات الكبيرة	Horizontal Scaling (Distributed Architecture)

Monitoring using Java Melody

أولاً: مقدمة

- الجزء دا بيتكلم عن مراقبة Jenkins باستخدام Java Melody
- وده جزء مهم جدًا لأي System Admin أوDevOps Engineer ، لأنك لازم تتابع أداء Jenkins .. هل بيستهلك CPU عالي؟ هل في مشاكل في الذاكرة؟ هل عدد الـ Threads كبير؟

---

1. ما هوJava Melody ؟

javamelody/javamelody: JavaMelody : monitoring of JavaEE applications

- Java Melody عبارة عن أداة (tool) خاصة بتطبيقات Java-based
- بتقيس أداء التطبيق فعليًا أثناء استخدامه (runtime statistics)
- بتعرض لك بيانات زي:
  - Memory usage
  - CPU load
  - Number of sessions
  - Threads
  - Errors and logs

بما إن Jenkins مكتوب بلغة Java ، يبقى طبيعي جدًا نقدر نراقبه مباشرة باستخدام Java Melody

---

2. تثبيت وإعداد Monitoring Plugin

- الـ Jenkins عنده Plugin رسمي اسمه Monitoring Plugin، وده بيشتغل باستخدام Java Melody
- علشان تبدأ تستخدمه:

1. تروح على Manage Jenkins → Manage Plugins
2. من Available plugins تعمل بحث عن : Monitoring
3. تعمل Install .. وبعدها Restart Jenkins

بعد التثبيت، الـ Java Melody هيبدأ يراقب Jenkins controller نفسه (اللي هو الـ master أو main node)

---

3. معلومات عن الـ Plugin

Monitoring | Jenkins plugin

- الـ Monitoring Plugin بيوفر Dashboard جاهزة فيها:
  - Memory usage : قد إيه من الـ RAM مستخدم.
  - CPU load average : متوسط استخدام المعالج.
  - Errors & Logs : أخطاء النظام والـ pipelines
  - Threads : العمليات الجارية داخل Jenkins
  - Reports : ممكن تصدرها كـ HTML أو PDF

كمان تقدر تختار اللغة (English, French, German...)

---

4. الوصول إلى أدوات المراقبة

- بعد ما تثبت الـ plugin :
  - تروح على Manage Jenkins
  - هتلاقي قسم جديد فيه خيارين:
    1. Monitoring of Jenkins Instance ← لمراقبة الـ Controller نفسه.
    2. Monitoring of Jenkins Agents ← لمراقبة الـ Nodes اللي بتبني عليها الـ jobs

5. 🖥️ داخل صفحة الـ Instance Monitoring

➢ هتلاقي Dashboard فيها بيانات Real-Time زي:

- 📊 Memory usage
- ⚙️ CPU utilization
- 🌐 Active HTTP sessions
- Thread count
- Error rate

📈 وفيها كمان Charts تفاعلية توضح الإحصائيات مع مرور الوقت.

---

6. 🧩 خيارات متقدمة في الواجهة

➢ Java Melody بيديك صلاحيات إضافية:

- 🔄 Execute Garbage Collector ← تنظف الذاكرة من الـ unused objects يدويًا.
- 🧠 Generate Heap Dump ← تعمل Snapshot لذاكرة JVM لتحليل الأداء.
- 📊 Memory Histogram ← توزع استخدام الذاكرة بين الـ classes المختلفة.
- 🌍 View HTTP Sessions ← تشوف المستخدمين المتصلين حاليًا، نوع المتصفح، والدولة.

---

7. 🧰 اختبار الأداة عمليًا

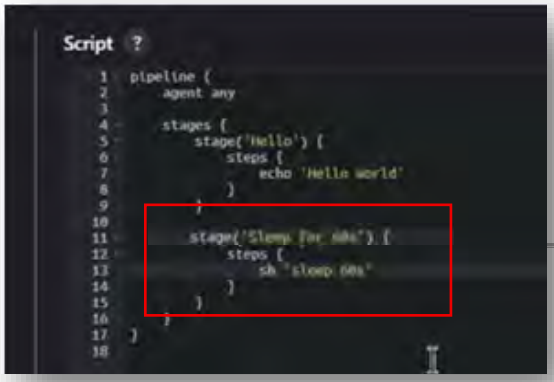
➢ هنا بيجرب يعمل Build Job عشان يراقب التغييرات في Dashboard

الخطوات:

1. بيروح على Jenkins ويشغل Job اسمه: test build monitoring plugin
2. جوّه الـ pipeline في Stage بتعمل Sleep لـ 60 ثانية (عشان يبقى عنده وقت يلاحظ التغير).
3. خلال الـ build بيروح يعمل Refresh في صفحة الـ Monitoring

بيلاحظ:

- عدد الـ threads زاد.
- الـ CPU usage ارتفع .
- الـ Job ظهر في الإحصائيات .



---

8. 📉 التحليل والنتائج

➢ مع الوقت، لما Jenkins يشغّل Jobs كتير، البيانات دي بتبقى مهمة جدًا لتحديد:

- هل الـ Controller محتاج Scaling ؟
- هل في bottleneck في أحد الـ nodes ؟
- هل Garbage Collector بيأثر على الأداء؟

🏁 الخلاصة

- Java Melody = أداة قوية جدًا لمراقبة تطبيقات Java.
- Jenkins Monitoring Plugin = واجهة جاهزة بتخليك تتابع أداء Jenkins بسهولة.
- تقدر تطلع تقارير PDF أو HTML
- تقدر تدير الذاكرة، تشوف الجلسات، وتعرف حالة كل Node
- مفيدة جدًا في بيئات CI/CD الكبيرة لتحديد مشاكل الأداء مبكرًا.

Slack Notification Setup

🎯 الفكرة العامة:

❑ الجزء ده بيتكلم عن إزاي نربط Jenkins مع Slack بحيث Jenkins يقدر يبعث notifications (إشعارات) على قناة معينة في Slack بعد كل build (زي نجاح أو فشل)

🧩 الخطوات بالتفصيل:

Slack Notification | Jenkins

1 تثبيت البلجن (Plugin)

- أول حاجة لازم نثبّت Slack Notification Plugin في Jenkins
- البلجن ده بيسمح إن Jenkins يتكلم مع Slack ويرسل له إشعارات عن حالة الـ build
- بعد ما البلجن يتثبت، لازم نعمل Restart لـ Jenkins

---

2 إنشاء حساب وقناة في Slack

- لو لسه ماعندكش حساب Slack ، تعمل واحد.
- بعد كده، جوه الحساب، بتعمل Workspace (زي مساحة شغل لفريقك)
- جوه الـ Workspace ده، تعمل Channel جديدة، مثلاً اسمها dasher-notifications
- دي القناة اللي Jenkins هيبعت فيها كل الإشعارات بتاعت الـ builds

---

3 إنشاء App في Slack

- Slack بيحتاج تطبيق (App) يكون هو اللي بيتكلم مع Jenkins
- فبنروح نعمل Create a New App من الـ Slack
- بنختار "From an App Manifest" ، وده بيسمح لنا نعمل التطبيق عن طريق ملف YAML جاهز.

---

4 ملف الـ YAML

- الـ YAML file ده بيحدد صلاحيات التطبيق اللي بنعمله:
  - يقدر يكتب رسائل (chat:write)
  - يقدر يقرأ المستخدمين (users:read)
  - يقدر يتعامل مع الملفات (files:write)
- يعني ده اللي بيحدد إيه اللي التطبيق يقدر يعمله داخل Slack

```yaml
display_information:
  name: Jenkins
features:
  bot_user:
    display_name: Jenkins
    always_online: true
oauth_config:
  scopes:
    bot:
      - channels:read
      - chat:write
      - chat:write.customize
      - files:write
      - reactions:write
      - users:read
      - users:read.email
      - group[illegible]read
settings:
  org_deploy_enabled: false
  socket_mode_enabled: false
  token_rotation_enabled: false
```

It's recommended to make It Unique

---

5 تثبيت التطبيق على Workspace

- بعد ما تعمل التطبيق، بتضغط على Install App to Workspace
- Slack هيدّيك حاجة اسمها OAuth Token (ده التوكن اللي بيمثّل الصلاحيات للتطبيق)
- التوكن ده مهم جدًا، وهنحتاج نحطه في Jenkins بعدين.

---

6 إعداد Jenkins بالـ Token

- نرجع لـ Jenkins ، نروح على : Manage Jenkins → Configure System
- هنلاقي جزء جديد اسمه Slack (ظهر بعد ما ثبتنا البلجن)
- هنا بنضيف:
  - اسم الـ workspace (مثلاً "Jenkins")
  - الـ Token اللي خدناه من Slack (بنضيفه كـ credential "Secret Text")
  - ID: Jenkins Slack Token
- بعدهل بنختار الـ channel اللي عايزين نبعت عليها الإشعارات (مثلاً "dasher-notifications")

---

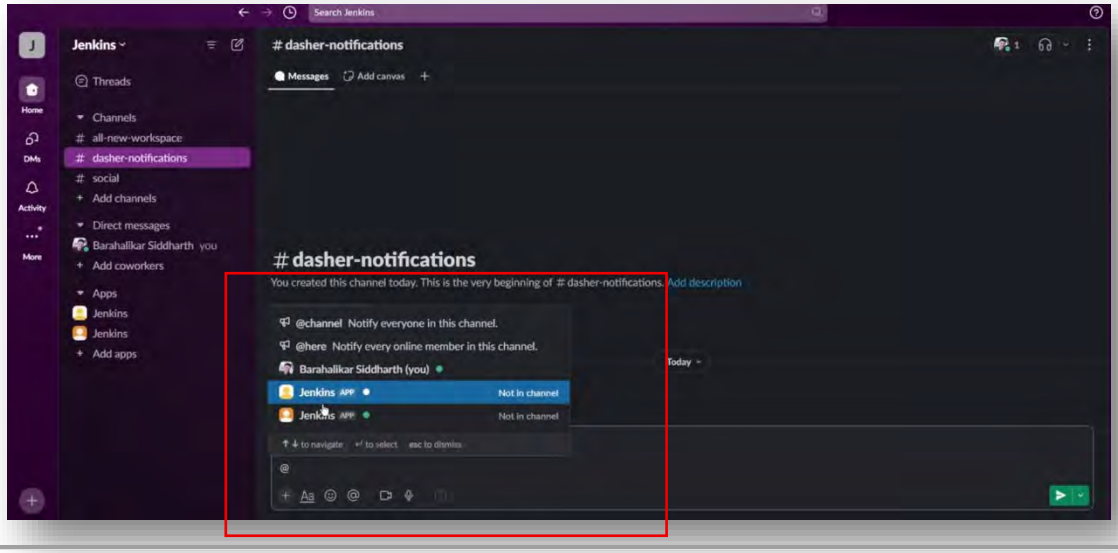
7 اختبار الاتصال (Test Connection)

- بنضغط على Test Connection عشان نتأكد إن Jenkins قدر يتصل بـ Slack
- لو التوكن مش صح، هيديك error بيقول "Invalid token"
- ولو التطبيق مش مضاف للقناة، هيديك Error بيقول إن المستخدم مش موجود في القناة.

---

8 إضافة التطبيق للقناة

- لو حصل الخطأ ده، الحل هو:
  - تفتح قناة Slack اللي أنشأتها.
  - تكتب فيها:
    @ وتضيف الـ Jenkins App اللي أنشأته.
- ده بيخلي التطبيق عضو في القناة، وبالتالي يقدر يبعث الرسائل فيها.



---

9 اختبار الاتصال تاني

- بعد ما تضيف التطبيق، ارجع لـ Jenkins وجرب Test Connection تاني.
- المفروض تظهر رسالة "Success"
- كده يبقى Jenkins متوصل تمامًا مع Slack ، والـ integration اشتغل.

في النهاية

- بعد كده، تقدر تستخدم البلجن في أي Pipeline أو Job في Jenkins
- بحيث كل ما يحصل Build Success أو Failure ، Jenkins هيبعت إشعار تلقائي في قناة Slack اللي اخترتها.

Send Slack Notification

🎯 الهدف

- ❑ النهارده احنا بنضيف Slack notifications داخل الـ Jenkinsfile، بحيث Jenkins يرسل إشعار أو رسالة تلقائيًا على Slack كل مرة الـ build ينجح أو يفشل.
- ❑ وده بيساعد الـ DevOps أو الـ Developers إنهم يتابعوا حالة الـ pipeline من Slack من غير ما يدخلوا Jenkins كل مرة.

🧩 الخطوات الأساسية اللي اتعملت

1. إنشاء فرع جديد (Branch)

- ⮜ مبدئيا كدا فالـ Developer الفهمان ما بيعدلش على الـ main branch مباشرة.
- ⮜ عمل فرع جديد مخصوص لتفعيل Slack notifications : git checkout -b feature/enabling-slack
  - ◆ هنا -b معناها "create new branch and switch to it"
  - ◆ اسم الفرع feature/enabling-slack بيعبر عن الغرض منه.

---

2. فتح الـ Jenkinsfile والتعديل عليه

- ⮜ هنا هنضيف أمر إرسال إشعارات Slack داخل الـ Jenkins pipeline
- ⮜ لكن طبعا زي ماتعودنا، بنستخدم Pipeline Syntax Generator في Jenkins علشان نولّد الكود الجاهز.

---

3. استخدام الـ Pipeline Syntax Generator

- ⮜ في الصفحة دي تقدر تختار الـ plugin اللي عايز تستخدمه (زي Slack Notification plugin)
- ⮜ Pipeline sytax → snippet generator → sample steps → slackSend: Send Slack Message
  - هانظبط الاعدادات
    - o Channel : اسم القناة اللي هيتبعت فيها الرسالة . ← dasher-notifications
    - o color : اللون اللي هيظهر في الرسالة (أخضر، أحمر، أصفر...).
    - o message : النص اللي عايز تبعته في الإشعار . ← testing

Generate Pipeline Script

```
slackSend botUser: true, channel: 'dasher-notifications', color: '#439FE0', message: 'Build Started: ${env.JOB_NAME} ${env.BUILD_NUMBER}'
```

---

4. تخصيص الألوان حسب حالة الـ Build

- ⮜ هنا استخدم Hex color codes (زي #FF00 00 للأخضر أو #FF0000 للأحمر)
  الفكرة إنك تبين الحالة بصريًا:
  - ✅ أخضر ← Build ناجح.
  - ⚠️ أصفر ← Build unstable
  - ❌ أحمر ← Build فاشل.

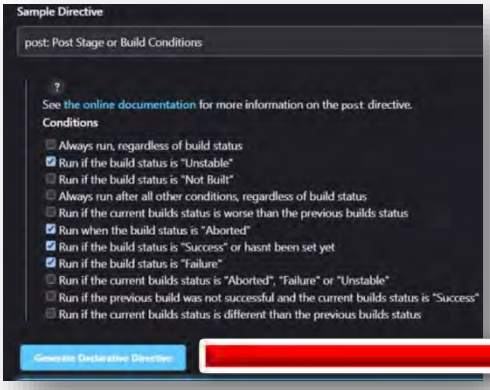
```
post {
    success {
        slackSend(color: '#00FF00', message: 'Build Success!')
    }
    failure {
        slackSend(color: '#FF0000', message: 'Build Failed!')
    }
}
```

5. إضافة الكود جوا الـ post داخل الـ Jenkinsfile

- زي ماعرفنا سابقا : ممكن تضيف في آخر كل Stage ← section اسمه post .. وجواه بتعرفه يعمل ايه بعد انتهاء الـ Build
  - لاحظ انت كدا معاك الكود اللي هاتكتبه من الخطوة رقم (3) .. بس الفكرة بقا هانكتبه ازاي جوا section الـ post ؟؟!
    - هاتروح علي :

Pipeline syntax → Declarative Directive Generator → sample steps → post: Post Stage or Build Conditions

    - هاتعلم عالحاجات اللي محتاجها زي ماهو موضح في الصورة وبعدين تضغط generate فهايطلعلك الـ structure اللي المفروض هايتعمل في الـ post
    - هاتنسخ الـ structure ده + تنسخ فيه الكود بتاع الخطوة رقم (3)
- وبعدين تضيف كل دا في الـ jenkinsfile




```
slackSend botUser: true, channel: 'dashe
slackSend botUser: true, channel: 'dasher-notifications', color: '#439FE0', message: 'Build Started: ${env.JOB_NAME} ${env.BUILD_NUMBER}'

post {
    unstable {
      // One or more steps need to be included within each condition's block.
    }
    aborted {
      // One or more steps need to be included within each condition's block.
    }
    success {
      // One or more steps need to be included within each condition's block.
    }
    failure {
      // One or more steps need to be included within each condition's block.
    }
}
```

🎯 وبالتالي هنا هاتظهر مشكلة :

❖ لو عندك 20 stage ، هتضطر تكرر ده في كل واحدة، وده مرهق جدًا وغير عملي.

---

6. حل المشكلة: إنشاء Groovy Method قابلة لإعادة الاستخدام

- بدل ما تكرر الكود، عمل دالة (method) في بداية Jenkinsfile (قبل الـ pipeline اصلا) اسمها slackNotificationMethod
- شرح الدالة:

```
Jenkinsfile
solar-system > Jenkinsfile
def slackNotificationMethod(String buildStatus = 'STARTED') {
  buildStatus = buildStatus ?: 'SUCCESS'

  def color

  if (buildStatus == 'SUCCESS') {
    color = '#47ec05'
  } else if (buildStatus == 'UNSTABLE') {
    color = '#d5ee0d'
  } else {
    color = '#ec2805'
  }

  def msg = "${buildStatus}: `${env.JOB_NAME}` #${env.BUILD_NUMBER}:\n${env.BUILD_URL}"

  slackSend(color: color, message: msg)
}

> pipeline {...
}
```

لاحظ ان ده هو الـ pipeline
بكل الحوارات اللي احنا قاعدين
نعملها فيه بقالنا 300 صفحة

- def ← 🧠
  تعريف دالة جديدة في Groovy
- String buildStatus = 'STARTED' ←
  متغير بياخد الحالة، ولو مش متحدد،
  يبقى القيمة الافتراضية "STARTED"
- ?: ← معناها لو المتغير فاضي، استخدم "SUCCESS"
- if ← بيحدد بيها بيحدد اللون المناسب
  (بناءً على الحالة SUCCESS, UNSTABLE, FAILURE)
- بعدين بيكوّن الرسالة اللي فيها:
  - حالة الـ build
  - اسم الـ job
  - رقم الـ build
  - الرابط في Jenkins
- وأخيرًا بيبعت الرسالة عن طريق slackSend

---

7. استدعاء الدالة داخل الـ Jenkinsfile

- بعد ما عرّفنا الدالة، بنستدعيها في post section كده:

```
post {
    always {
        slackNotificationMethod("${currentBuild.result}")
```

- always معناها إنها هتتنفذ بعد أي حالة (نجاح أو فشل أو unstable)
- currentBuild.result بيجيب نتيجة الـ build تلقائيًا.

---

8. تنفيذ الـ Pipeline ومتابعة النتيجة

- بعد حفظ التغييرات:
  - git commit -m "Added Slack notification method"
  - git push
- رجع Jenkins ، شغّل build جديد للفرع الجديد، وكل حاجة اشتغلت تمام.
  راح على Slack ، لقى رسالة بتقول:

```
Build Status: SUCCESS
Job: solar-system
Build Number: 34
URL: http://jenkins/job/solar-system/34/
```

  - بلون أخضر. ✅

---

9. اختبار حالة الفشل

- عشان يتأكد، حط في مرحلة من المراحل : exit 1
- وده بيخلي الـ build يفشل.
- رجع Slack ، وفعلاً وصلت رسالة فشل باللون الأحمر. ❌

💡 الملخص :

الجزء دا بيخليك:

1. تضيف Slack notifications في Jenkins بسهولة.
2. تبعت إشعارات تلقائية بعد أي build
3. تستخدم method واحدة بدل تكرار الكود.
4. تميّز الإشعارات بالألوان حسب النتيجة.

- ❑ لاحظ ان اللي احنا عملناه في الخطوة رقم (6) مش عملي اطلاقا .. ليه ؟!؟!
  - ❖ ودا لأن الكود ده مناسب فقط للـ Scenario ده (اللي احنا عملناه في الـ session ده يعني)
  - ❖ ولو حبيت تستخدم الكود دا في أي Jenkins file تاني ← مش هاينفع
- ❑ وبالتالي فالحل هو Shared Library in Jenkins

Introduction to Shared Library

🎯 الفكرة العامة

- ❑ الجزء دا بيتكلم عن Shared Libraries في Jenkins
- ❑ الفكرة ببساطة إنك بدل ما تكرّر نفس الكود في كل Jenkinsfile لكل مشروع أو Job ، تقدر تكتب الأكواد المشتركة مرة واحدة في مكان مركزي (زي Repository على GitHub) .. وكمان تخلي كل الـ Pipelines تستخدمها.
- ❑ ده بيخلّي الكود:
  - أسهل في الصيانة (maintainable)
  - موحد في كل المشاريع (consistent)
  - ومفيش تكرار (No duplication – DRY principle: *Don't Repeat Yourself*)

🧩 الشرح التفصيلي خطوة بخطوة

1. 🧠 ما هي Shared Libraries ؟

- ➢ هي عبارة عن مجموعة سكربتات مكتوبة بلغة Groovy (اللغة اللي Jenkins Pipeline مكتوب بيها)
- ➢ الغرض منها إنك تكتب فيها دوال (functions) أو خطوات (steps) متكررة وتستخدمها في أي Jenkinsfile بعد كده.
- ➢ 🔹 الفكرة:
  - بدل ما تكتب كل مرة stages زي build و test و deploy بنفس الأكواد،
  - تخليهم في مكتبة مشتركة (Shared Library) ، وتستدعيهم بسهولة في أي Pipeline

---

2. ⚙️ ليه نستخدم Shared Libraries ؟

- ➢ لأن بدونها، كل Pipeline هيبقى فيه نفس الكود مكرر.
- ➢ وده بيعمل مشاكل زي:
  - تكرار الكود (Duplication)
  - صعوبة التحديث (لو غيرت حاجة لازم تعدلها في كل Jenkinsfile)
  - عدم الاتساق (Inconsistency بين المشاريع)
  - صعوبة الإدارة لما العدد يكبر

📌 Shared Library بتحل ده لأنها مركزية، وكل الـ Pipelines بتسحب منها.

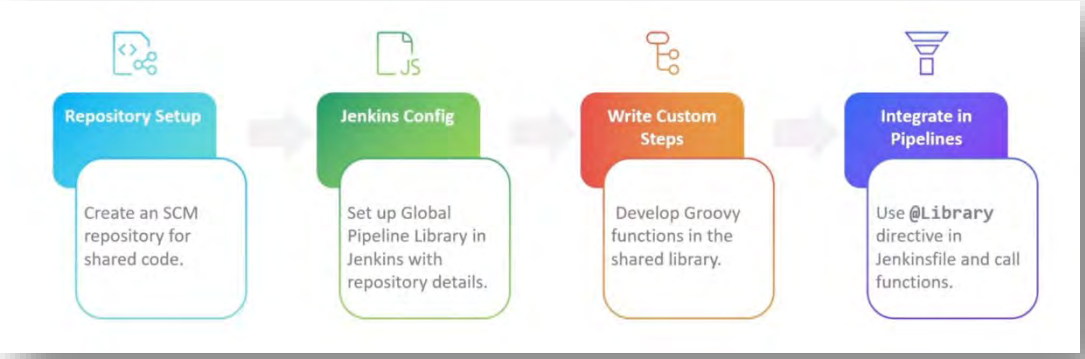
---

3. 📝 مثال عملي للفكرة

- ➢ افترض إنك شغال في شركة وكل Pipeline لازم يبدأ بـ "Welcome to the DevOps Team"
- ➢ في البداية ممكن تكتب الرسالة دي في أول كل Jenkinsfile يدويًا. بس بعد فترة، الشركة كبرت وبقى عندها مئات Pipelines
- ➢ لو الشركة غيرت اسمها مثلًا من Blue إلى KodeKloud، لازم تعدّل الرسالة دي في كل Jenkinsfile يدويًا. 😫
- ➢ هنا بقى بييجي دور Shared Library :
  - هتخزن الرسالة دي في دالة واحدة (function) داخل ملف Groovy ،
  - ولو الاسم اتغير، تعدّلها مرة واحدة بس، وكل الـ Pipelines تتحدث تلقائيًا.

---

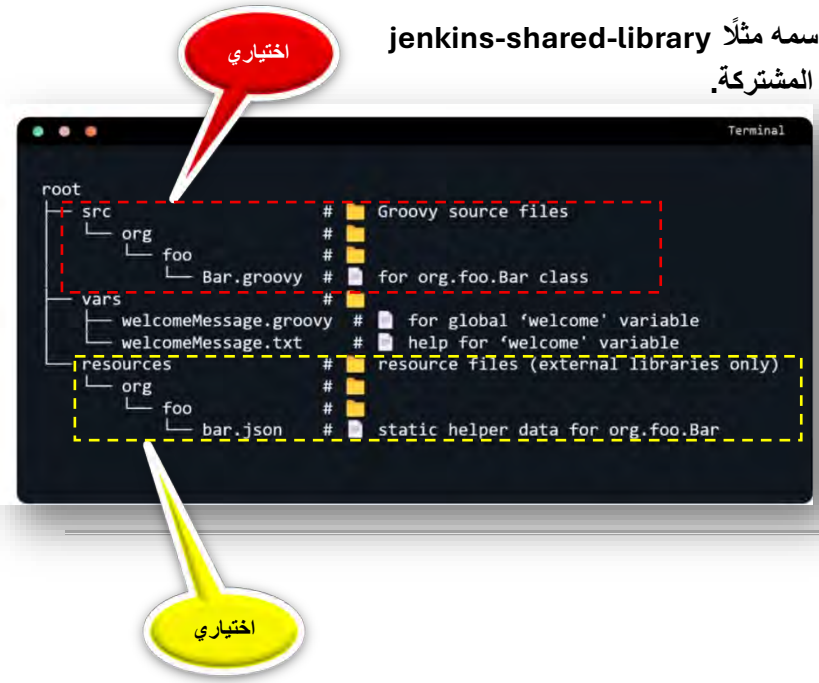
4. 🧰 الخطوات العملية لإنشاء Shared Library



1️⃣ الخطوة الأولى : إنشاء Repository مخصص للمكتبة

- اعمل Git repo جديد (مثلاً على GitHub) وخلي اسمه مثلًا jenkins-shared-library
  - جواه هتحط ملفاتك اللي فيها الـ functions المشتركة.

📁 هيكل المجلد الأساسي:




```
jenkins-shared-library/
├── vars/
│   └── welcome.groovy
├── src/          (اختياري)
└── resources/    (اختياري)
```

📁 شرح هيكل المجلدات

1. vars/
   - دا أهم فولدر
   - جواه بتحط ملفات groovy اللي فيها الدوال القابلة لإعادة الاستخدام.
   - كل ملف هنا بيكون كأنه function ممكن تستدعيه في Jenkinsfile
   - مفيش Subfolders جواه (مينفعش تعمّق الهيكل).
   - التسمية بتكون camelCase (زي welcomeMessage.groovy)
2. src/ (اختياري)
   - مخصص للكود الـ compiled (زي Java Classes)
   - نادر جدًا نستخدمه في الـ Pipelines البسيطة.
3. resources/ (اختياري)
   - بنخزن فيه ملفات نصية أو YAML أو Config نحتاجها داخل المكتبة.

⚙️ الخطوة الثانية: كتابة كود الـ function

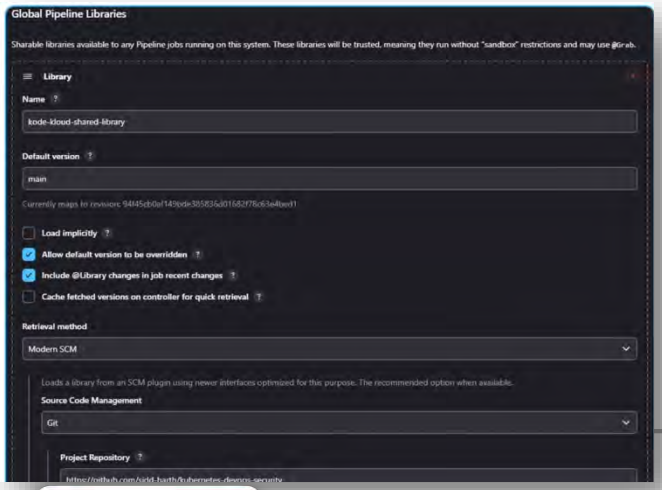
مثلاً في ملف : vars/welcome.groovy

- اسم الملف welcome.groovy
- اسم الدالة لازم يكون call()
  عشان Jenkins يتعرف عليها تلقائيًا.
- أي Jenkinsfile يقدر يستدعيها بعد كده كأنها step جاهز.

⚙️ الخطوة الثالثة : تعريف المكتبة في Jenkins

من واجهة Jenkins :

1. روح إلى: Manage Jenkins → Configure System → Global Pipeline Libraries
2. اضغط "Add" وأضف التفاصيل التالية:
   - Name : اسم المكتبة (مثلاً kode-kloud-shared-library)
   - Default version : main
   - Allow default version to be overridden : ✅ (اختياري)
   - Implicitly load : ✅ (عشان تتشحن تلقائيًا في كل Jenkinsfile)
   - Retrieval method :
     اختار Modern SCM
     - SCM : Git
     - URL : لينك الـ repo على GitHub



📜 الخطوة الرابعة : استخدام المكتبة داخل Jenkinsfile

دلوقتي، في أي مشروع، افتح Jenkinsfile واستدعي المكتبة بتاعتك

- وده معناه إن Jenkins هيروح يستخدم library اسمها kode-kloud-shared-library
  (لازم تكون متسجلة مسبقًا في Jenkins ودا اللي احنا عملناه في الخطوة الثالثة)
- بعد كده بدل ما تكتب كود الـ echo جوه Jenkinsfile (الصورة اللي عالشمال)
  فأنت هاتكتبه وتخزنه في الـ Shared Library (في function اسمها welcome()) وبعدين هاتستدعي الـ function دي في أي jenkinsfile بعد كدا علطول
- الـ function دي موجودة جوه ملف اسمه welcome.groovy داخل الـ shared library (ودا اللي عملناه في الخطوة الثانية)
- يعني Jenkins أول ما يشوف welcome()، هيروح يدور عليها في الـ library ويشغلها.

🧠 ملاحظات عملية (Best Practices / Real Use Cases)

1. ✳️ احفظ المكتبة في Repo منفصل
   عشان تبقى سهلة في المراجعة والـ versioning
2. 🔐 تحكم في الإصدارات (Version Control)
   - خليك دايمًا محدد الفرع أو التاج اللي تسحب منه المكتبة.
   - متسحبش من master/main مباشرة في بيئات Production
3. 🧱 استخدم CamelCase لتسمية الدوال ← زي buildApp(), deployApp(), runTests()
4. 🧩 اعمل ملفات .txt توثّق كل function
   - مثلاً : vars/welcomeMessage.txt تحط فيه شرح مختصر عن الـ function

5. 🔄 اعمل Testing للمكتبة قبل ما تعتمدها
   - Jenkins بيسمحلك تعمل Unit Tests باستخدام Jenkins Pipeline Unit
6. 🌱 احرص على Modular Design
   - يعني كل function تعمل حاجة واحدة بس.

🏁 الملخص النهائي

العنصر	الشرح المختصر
🔹 Shared Library	كود مشترك لإعادة الاستخدام في Jenkins Pipelines
📦 الموقع	Git repository مركزي
🧩 المحتوى	ملفات Groovy داخل مجلد /vars
🧠 الفائدة	تقليل التكرار – سهولة الصيانة – اتساق الكود
⚙️ الاستخدام	تعريف المكتبة في Jenkins ثم استدعائها بـ `@Library()`
🧱 أفضل الممارسات	استخدم camelCase – versioning – ملفات توثيق – Modular Functions

Refactor Existing Jenkinsfile

🎯 الفكرة العامة

- ❑ الجزء ده هو بداية دورة Jenkins Pipeline Advanced ، وهانبدأ بإعادة استخدام الـ pipeline اللي اتعملت في السابق.
- ❑ الفكرة هنا إننا نرجع على المشروع القديم (Solar System repo) ، ونبدأ نعمل Refactor الـ Jenkinsfile علشان نستخدمه كأساس للدروس الجاية.
- ❑ بمعنى بسيط:

هو بياخد Jenkinsfile كبير فيه حوالي 20 stage ، ويبدأ يقلّمه (يعمل hashing على stages مش مهمة دلوقتي) علشان يحتفظ بس بالأساسيات اللي هنكمل عليها في الدروس الجاية.

🧱 المرحلة 1: مراجعة المشروع السابق (Recap of Previous Setup)

- ➢ أول حاجة، فتح Jenkins instance اللي استخدمناه في الكورس القديم، واللي فيه:
  - Organization Folder اسمها مثلاً gitea organization :
    - o جواها Repository اسمها solar-system
    - o فيها فروع (branches) مختلفة .. آخر واحدة اشتغلوا عليها كانت feature/enabling-slack

📌 أوامر وأدوات مستخدمة هنا: Jenkins Organization Folder

- o بيتعرف تلقائيًا على الـ repos والفروع داخل GitHub organization
- Git branches (main, feature/enabling-slack, ...)

🧠 Best Practice :

- لما تعمل كورس أو مشروع تدريبي متكامل، حاول دايمًا تبني كل مرحلة على اللي قبلها (Reuse) ، وده اللي بيعمله هنا بالضبط.

🧩 المرحلة 2: تجهيز الـ Jenkinsfile للتطوير

- ➢ فتح الـ Jenkinsfile على Visual Studio Code
  - ولقى إنه ضخم جدًا (حوالي 20 Stage)
- ➢ الهدف دلوقتي:
  - نخلي الـ Jenkinsfile بسيط، ونحتفظ بس بالـ stages اللي هنحتاجها في الكورس المتقدم.
- ➢ ✅ الخطوات اللي اتعملت:
  - فتح ملف Jenkinsfile .. وبدأ يعلّق (comment out) مراحل مش محتاجها.

```
stages {
    stage('Installing Dependencies') {
        options { timestamps() }
        steps {
            sh 'npm install --no-audit'
            sh 'exit 1'
        }
    }
```

Take care & delete it, explained in p.312

⚙️ المرحلة 3: تعديل الـ Stages خطوة بخطوة

خلينا نفصلها:

🔸 المرحلة الأولى Dependency Installation :

- ✅ Install Dependencies : Keep stage
  - o Still need it (أساسي لأي pipeline)

🔸 المرحلة الثانية Dependency Scanning :

```
stage('Dependency Scanning') {
    parallel {
        stage('NPM Dependency Audit') {...
        }

        // stage('OWASP Dependency Check') {
        //     steps {
        //         dependencyCheck additionalArguments: '''
        //             --scan \'./\'
        //             --out \'./\'
        //             --format \'ALL\'
        //             --disableYarnAudit \
        //             --prettyPrint''', odcInstallation: 'OWASP
        //         dependencyCheckPublisher failedTotalCritical:
        //     }
        // }
    }
```

- ✅ NPM Dependency Audit → Keep
- ❌ OWASP Dependency Check → Comment out
  - o لأنه مش محتاجه حالياً.

📌 أمر محتمل في Jenkinsfile :

npm audit ← بيفحص الثغرات في مكتبة الـ npm packages

💡 Real Case Tip :

- في المشاريع الواقعية، بتحط الفحصين مع بعض (npm audit + OWASP) ، لكن ممكن تفصلهم لو بتجرب pipeline خفيفة أو بتبني Prototype

```
76 >    stage('Unit Testing') {…
84      }
85
86 >    stage('Code Coverage') {…
92      }
93
94 >    // stage('SAST - SonarQube') {…
109     // }
110
```

◆ المرحلة الثالثة Testing :

- ✅ Unit Testing + Code Coverage : Keep
- ❌ SonarQube Analysis : Comment
  - علشان مش محتاج تحليل جودة الكود دلوقتي .

◆ المرحلة الرابعة Container Build :

- ✅ Docker Build Image : Keep
- ✅ Trivy Vulnerability Scanner : Keep

📌 ملاحظات عملية:

- Trivy بيعمل فحص للـ Docker image بعد البناء.
- ده بيساعدك تكتشف ثغرات في الـ base image أو في الـ dependencies

◆ المرحلة الخامسة Deployment Stages :

كل المراحل التالية اتعملها (Commented Out) :

- Push Docker Image
- Deploy to AWS EC2
- Integration Testing
- Kubernetes Deployment
- OWASP ZAP (Security Scan)
- Deploy to Production
- Lambda Deployment

◆ المرحلة السادسة Post Section :

➢ هايحتفظ فقط بـ: ✅ Slack Notification

- وباقي الخطوات (زي Publishing reports) علّقها. ❌

➢ لكن قبل ما يعلّقها بالكامل، نقل خطوة Publish HTML report الخاصة بـ Trivy لمكانها الطبيعي (بعد Stage Trivy مباشرة)

📌 الخطوات اللي عملها:

- نسخ جزء الـ publishHTML(...)
- نقله بعد Stage Trivy
- علّق الباقي في الـ post section

```
post {
    always {
        slackNotificationMethod("${currentBuild.result}")

        // script {
        //     if (fileExists('solar-system-gitops-argocd')) {
        //         sh 'rm -rf solar-system-gitops-argocd'
        //     }
        // }

        // junit allowEmptyResults: true, stdioRetention: '', testRes
        // junit allowEmptyResults: true, stdioRetention: '', testRes
```

تذكر ان دي هي الـ slackNotificationMethod
اللي احنا معرفينها في أول الـ jenkinsfile ..
(هانبدأ نحطها في shared library الدرس القادم)

```
jenkinsfile
solar-system > jenkinsfile
def slackNotificationMethod(String buildStatus = 'STARTED') {
    buildStatus = buildStatus ?: 'SUCCESS'

    def color

    if (buildStatus == 'SUCCESS') {
    color = '#47ec05'
    } else if (buildStatus == 'UNSTABLE') {
    color = '#d5ee0d'
    } else {
    color = '#ec2805'
    }

    def msg = "${buildStatus}: `${env.JOB_NAME}` #${env.BUILD_NUMBER}:\n${env.BUILD_URL}"

    slackSend(color: color, message: msg)
}
```

✨ المرحلة 4: إنشاء فرع جديد للتعديلات

- قرر ميعملش التعديلات على الفرع القديم feature/enabling-slack
- وعمل فرع جديد اسمه : feature/advanced-demo

📌 أوامر Git المستخدمة: git checkout -b feature/advanced-demo ← بيعمل فرع جديد وينقلك عليه

ثم: git push origin feature/advanced-demo

💡 Best Practice :

- ما تعملش refactor على نفس الفرع اللي فيه شغل مستقر.
- الأفضل تعمل فرع جديد علشان تجرب التعديلات بأمان.

⚡ المرحلة 5: تشغيل الـ Pipeline الجديد

- بعد ما عمل الفرع ، Jenkins Organization Folder عمل scan تلقائي (Auto-discovery) ولقى الفرع الجديد فيه Jenkinsfile ← فبدأ Build تلقائي.

📌 معلومة مهمة:

- Jenkins Organization Jobs تكتشف الفروع أوتوماتيك لما يكون فيها Jenkinsfile
- الـ Build فشل في مرحلة install dependencies لأن فيه أمر بيعمل exit 1 (مقصود عشان تجربة Slack Notification لو متذكر)

🔔 بعد الفشل، Slack أرسل رسالة:

"Build failed on feature/advanced-demo branch."

وده بيأكد إن الـ notification شغالة تمام.

💾 المرحلة 6: تعديل الكود محليًا ورفع التغييرات

رجع لـ VS Code :

1. حذف سطر exit 1 من مرحلة الـ dependencies
2. حفظ الملف
3. عمل commit و push :

- git add Jenkinsfile
- git commit -m "Refactored pipeline"
- git push

- ثم راح لـ Jenkins وشاف إن build جديد بدأ أوتوماتيك.
- دلوقتي عنده pipeline مختصرة فيها stages 4 – 5 فقط.

🧠 المراحل الجديدة في الـ Jenkinsfile بعد الـ refactor

```
pipeline {
  agent any
  stages {
    stage('Install Dependencies') { ... }
    stage('Dependency Scan (NPM Audit)') { ... }
    stage('Unit Test & Coverage') { ... }
    stage('Docker Build') { ... }
    stage('Trivy Scan & Report') { ... }
  }
  post {
    always {
      slackNotify()
    }
  }
}
```

🧠 ملخص شامل

الهدف	الخطوات	الأدوات/الأوامر	ملاحظات عملية
إعادة استخدام pipeline قديم	فتح مشروع سابق في Jenkins	Jenkins Organization Folder	reuse improves efficiency
تقليل حجم Jenkinsfile	تعليق stages غير ضرورية	VS Code, Groovy `//` comments	ابني pipeline تدريجي
تنظيم التقارير	نقل publishHTML بعد Trivy	`()publishHTML`	خلي كل stage مسئولة عن تقاريرها
إنشاء فرع جديد للتجارب	`git checkout -b feature/advanced-demo`	Git	متعملش refactor في main branch
التحقق من Slack Notification	intentionally failed build	Slack plugin	Useful for alert testing
تعديل وإعادة تشغيل	`git commit && git push`	Jenkins auto build	Organization job auto-detects new branches

القسم	التفاصيل
🧱 المشروع	Dasher org تحت Solar System repo
🎯 الهدف	عمل Refactor لـ Jenkinsfile عشان يبقى خفيف وجاهز لـ advanced demos
⚙️ الخطوات	1️⃣ علّق stages مش محتاجها 2️⃣ نقل publishHTML داخل Stage Trivy 3️⃣ علّق post steps ماعدا Slack 4️⃣ أنشأ branch جديد `feature/advanced-demo`
🧩 المراحل اللي فضلت	Install Dependencies – NPM Audit – Unit Test – Docker Build – Trivy Scan – Slack Notification
🎉 ملاحظات	Trivy فشل بسبب API limit، Slack Notification اشتغل بنجاح
💬 النتيجة	Jenkinsfile جاهز للكورس الجديد اللي هيتكلم عن shared libs وtrivy وDocker advanced topics

Create a Shared Library for Slack Notification

🎯 الفكرة العامة:

- ❑ في الأول، هو بيقول إن في الـ Jenkinsfile بتاع مشروع (زي Solar System repo) كتب (Groovy code) لإرسال رسالة إلى Slack .. الكود دا بيبني الرسالة، بيحدد اللون، وبعدين بيبعتها.
- ❑ لكن المشكلة إن الكود دا مكتوب جوا Jenkinsfile نفسه، يعني:
  - ميقدرش يستخدم في أي مشروع تاني.
  - وأي تعديل فيه لازم يتعمل في كل مشروع يدوي.
- ❑ فالحل إننا نعمل Shared Library علشان الكود دا يتخزن في Repository منفصل، ويتعاد استخدامه في أكتر من Pipeline بسهولة.

⚙️ خطوات الإنشاء:

1. بيروح على Gitea organization (Dasher org)
2. بيعمل Repository جديدة باسم shared-libraries
3. جوه الريبو دي، بيضيف Folder اسمه vars/
4. جوا الفولدر دا، بيعمل File اسمه : slackNotification.groovy
   - o ولاحظ ان الاسم مكتوب بصيغة camelCase.groovy

```
shared-libraries/
└── vars/
    └── slackNotification.groovy
```

💡 الفكرة اللي هايعملها :

- ❑ هو بياخد نفس الكود اللي كان في الـ Jenkinsfile (اللي بيبعت Slack message) وبيحطه هنا في الملف الجديد.
- ❑ لكن بدل ما يسمي الفنكشن باسم معين، بيستخدم call method

```
New File
def slackNotificationMethod(String buildStatus = 'STARTED') {
    buildStatus = buildStatus ?: 'SUCCESS'

    def color

    if (buildStatus == 'SUCCESS') {
    color = '#47ec05'
    } else if (buildStatus == 'UNSTABLE') {
    color = '#d5ee0d'
    } else {
    color = '#ec2805'
    }

    def msg = "${buildStatus}: `${env.JOB_NAME}` #${env.BUILD_NUMBER}:\n${env.BUILD_URL}"

    slackSend(color: color, message: msg)
}
```

```
def call(String buildStatus = 'STARTED') {
```

📘 ليه استخدم call method؟

- ➢ في Jenkins Shared Libraries ، لو عايز تستخدم السكربت بتاعك كأنه Built-in step (زي sh, git, echo) ساعتها بتسمي الفنكشن call()
  - الميثود call() هي الطريقة اللي Jenkins بيقدر بيها يشغّل الكود اللي في ملف المكتبة لما تنادي اسمه في Jenkinsfile
- ➢ مثال من الـ Docs :

Extending with Shared Libraries

```
// vars/sayHello.groovy
def call(String name = 'human') {
    // Any valid steps can be called from this code, just like in other
    // Scripted Pipeline
    echo "Hello, ${name}."
}
```

🧠 إيه اللي بيحصل هنا بالضبط؟

- لما Jenkins يقرأ السطر ده : sayHello("Youssef")
- هو بيروح يدور في الـ Shared Library اللي اسمها my-shared-library ، ويقول:
  - o "هل في ملف اسمه sayHello.groovy جوه مجلد vars/ ؟"
  - o 📦 لو لقاه، بيشغله.
- لكن عشان يعرف إيه الكود اللي المفروض يتنفذ لما حد ينادي sayHello() ، هو لازم يلاقي function اسمها call() جواه.
- فبيعمل داخليًا كده: sayHello("Youssef") ==> sayHello.call("Youssef")

⚙️🧠 طيب وليه مش بنسميها أي اسم تاني بدل call() ؟

- لأن Jenkins مبرمج إنه يدور على function اسمها call() تحديدًا.
  - هي دي الـ entry point (نقطة البداية) لأي سكريبت جوه vars/
  - يعني Jenkins بيقول : " انا هشغّل الميثود اللي اسمها call() لما المستخدم ينادي اسم أي ملف داخل vars/ ."

🔍تخيلها كده:

- تخيل عندك زرار مكتوب عليه "sayHello" في ريموت كنترول.
  - الزرار ده مش بيعمل حاجة لوحده، لازم تربطه بحدث لما تدوس عليه.
- الميثود call() هي الحدث ده 👇
  - يعني لما تدوس على الزرار (تكتب sayHello()) ، يتنفذ الكود اللي جواه call()

➢ وساعتها في الـ Jenkinsfile تقدر تكتب كده :

sayHello("Youssef") بدون ما تحتاج تكتب sayHello.call()

✅ الخطوة الجاية:

في الدرس الجاي، هيشرح إزاي نربط الـ Shared Library دي بجينكنز (من خلال إعدادها في System Configuration داخل Jenkins) .. وبعدين إزاي نستدعيها من الـ Jenkinsfile

Configure Shared Library in Jenkins

🎯 الفكرة العامة

- ❑ الجزء ده بيشرح إزاي نربط الـ Shared Library اللي عملناها في الجلسة اللي فاتت (زي مكتبة إشعارات Slack) بـ Jenkins نفسه، علشان الـ Jenkins يقدر يتعامل معاها ويشغّلها من أي Pipeline
- ❑ بمعنى تاني:
  - قبل كده عملنا كود Groovy وحطيناه في Repository مخصوص.
  - دلوقتي بنقول لـ Jenkins "خد بالك، المكتبة دي موجودة في المكان الفلاني، واسمها كذا، وتقدر تستخدمها."

🧱 أول جزء: فهم فكرة المكتبات داخل Jenkins

- ➢ في Jenkins ، فيه نوعين من Shared Libraries :

النوع	الوصف	مستوى الأمان
Trusted Libraries	مكتبات انت اللي عاملها بنفسك أو واثق فيها	بتشتغل بدون قيود (No Sandbox)
Untrusted Libraries	مكتبات جاية من الإنترنت أو من مطورين تانيين	بتشتغل داخل Sandbox (محمية بقيود أمان)

- ➢ 🔒 يعني إيه Sandbox ؟
  - الـ Sandbox بيمنع تشغيل أوامر ممكن تكون خطر أو فيها Access للنظام.
  - بيخلّي Jenkins يراجع الكود قبل ما يشغله.
  - أي دالة "مش آمنة" لازم Admin يوافق عليها الأول.
- ➢ 👩‍🏫 ملحوظة عملية:
  - لو بتكتب مكتبتك بنفسك داخل الشركة، دايمًا استخدمها كـ Trusted Library
  - لكن لو بتسحب مكتبات عامة من GitHub أو Maven ، استخدمها كـ Untrusted

---

⚙️ الجزء التاني: الخطوات داخل Jenkins UI

1️⃣ افتح Jenkins → Dashboard

روح على : Manage Jenkins → System

2️⃣ في البحث اكتب "Libraries"

هتلاقي اختيارين:

- Global Pipeline Libraries (Trusted)
- Global Untrusted Pipeline Libraries

3️⃣ نختار Trusted Pipeline Libraries

لأن المكتبة دي احنا اللي عامليها بنفسنا (زي مكتبة Slack Notification)

🧩 إعداد الـ Trusted Library

➢ لما تضغط على "Add" وتضيف مكتبة جديدة، هتحتاج تملأ شوية بيانات:

الإعداد	معناه	مثال
Name	اسم المكتبة اللي Jenkins هيشوفها بيه	dasher-trusted-shared-library
Default version	اسم الـ branch اللي جوه الـ repo	`main`
Load implicitly	لو فعلتها، كل الـ pipelines هتشوف المكتبة تلقائيًا بدون ما تكتب @Library	❌ الأفضل تسيبها مقفولة
Allow default version to be overridden	لو عندك فروع مختلفة للمكتبة (مثلاً dev/test/prod) تقدر تغير الـ branch من الـ pipeline	✅ يفضل تفعّلها
Include library changes in job recent changes	لو فعلتها، أي تعديل في المكتبة هيعمل Build جديد	حسب استخدامك
Caching	بيخزن نسخة محلية من المكتبة عشان التحميل يبقى أسرع	✅ يفضل تفعلها
Source Code Management (SCM)	المصدر اللي هتسحب منه المكتبة	`Git`
Repository URL	لينك الريبو اللي فيه الـ Shared Library	مثلًا: `https://github.com/dasher-org/shared-libraries.git`
Credentials	لو الريبو Private بتحط Credentials هنا	لو Public مش محتاج

➢ 🧠 أمر عملي:

لو انت بتستخدم Jenkins CLI أو Jenkins Configuration as Code (JCasC) ، ممكن تضيف نفس المكتبة كده في الـ YAML :

```yaml
unclassified:
  globalLibraries:
    libraries:
      - name: "DasherTrustedLibrary"
        defaultVersion: "main"
        implicit: false
        retriever:
          modernSCM:
            scm:
              git:
                remote: "https://github.com/dasher-org/shared-libraries.git"
```

Extending with Shared Libraries

💡 ملاحظات حول الـ @Grab Annotation

🔍 إيه هي @Grab؟

- دي طريقة في Groovy تقدر بيها تسحب مكتبة Java خارجية من Maven Central وتستخدمها في كودك .. مثال:

```
@Grab('org.apache.commons:commons-math3:3.4.1')
import org.apache.commons.math3.primes.Primes
void parallelize(int count) {
  if (!Primes.isPrime(count)) {
    error "${count} was not prime"
  }
  // ...
}
```

  - كده انت فعليًا بتضيف مكتبة "Math" من Maven وتقدر تستخدم دوالها داخل الـ Shared Library بتاعتك.

⚠️ لكن خد بالك:

- **@Grab مينفعش تستخدمها في Untrusted Libraries لأنها بتحتاج صلاحيات خارج الـ Sandbox**
- يعني لازم المكتبة تبقى Trusted عشان Groovy يقدر يسحب الـ dependency دي.

🧠 حالات استخدام واقعية (Real Use Cases)

1. في الشركات الكبيرة:
   - بيبقى عندهم Shared Library واحدة مركزية فيها كل الأكواد العامة:
     - إشعارات Slack
     - Deployment scripts
     - Code quality checks
     - Notification templates
   - كل الـ teams بيستدعوا المكتبة دي بدل ما يكرروا نفس الكود في كل Jenkinsfile
2. في بيئة DevOps :
   - Shared Libraries بتوفر standardization وبالتالي أي تعديل في الـ library يتطبق تلقائيًا على كل المشاريع.

🧭 Best Practices

✅ استخدم Trusted Libraries للأكواد الداخلية اللي انت كاتبها.

✅ خليك منظم في هيكلة المشروع:

```
(shared-library-repo)
├── vars/
│    ├── slackNotification.groovy
│    ├── deployApp.groovy
│    └── testReport.groovy
├── src/
│    └── org/company/utils/ (متقدم Groovy كود)
└── resources/
     └── templates/
```

✅ لو بتستخدم @Grab، اربط الإصدارات بإصدارات محددة (Avoid latest)

✅ ما تفعّلش "Load implicitly" إلا لو المكتبة صغيرة وآمنة جدًا.

✅ اعمل test للـ Shared Library في pipeline تجريبي قبل اعتمادها في الـ production

---

📄 الملخص النهائي

- شرحنا إزاي نربط Shared Library بـ Jenkins
- فيه نوعين من المكتبات:
  - Trusted (بدون Sandbox) - Untrusted (داخل Sandbox)
- الخطوات داخل Jenkins :
  - Manage Jenkins → System → Global Pipeline Libraries → Add
  - تكتب الاسم، الـ branch ، والـ repo URL
- ممكن تستخدم @Grab لاستيراد مكتبات Java إضافية (ولكن فقط في Trusted)
- بعد الربط ، تقدر تستخدم المكتبة في Jenkinsfile كده:
  - **@Library('DasherTrustedLibrary') _** → *calling the library*
  - **slackNotification('SUCCESS')** → *calling the function*

Loading the Shared Library in Pipeline

🎯 المقدمة :

- ❑ ده شرح الجزء التاني من موضوع الـ Shared Library .. واللي بيكمل على الدرس السابق اللي كنا بنربط فيه المكتبة جوه Jenkins
- ❑ الجزء ده بيوضح إزاي فعليًا نستخدم المكتبة دي داخل Jenkinsfile بدل ما نكتب الكود بنفسنا.
- ❑ الهدف هنا إننا:

نستبدل الكود الـ Groovy اللي كنا كاتبينه جوه الـ Jenkinsfile ← بكود بسيط بيستدعي نفس الوظيفة لكن من الـ Shared Library اللي جهزناها قبل كده.

🧱 أولاً: التأكد إن Jenkins فعلاً شايف المكتبة

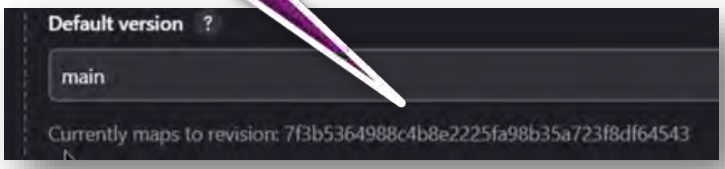
- ➢ بعد ما نضيف المكتبة في: Manage Jenkins → System → Global Trusted Pipeline Libraries
- ➢ وعلشان تفهم اللي بيحصل :
  - لما تعمل Save وتعمل Refresh للصفحة، Jenkins هيعمل Connect بالـ Git Repo
  - وهيجيب الـ Commit ID من الـ main branch

🧠 يعني إيه Commit ID هنا؟

ده رقم الإصدار (الـ hash) اللي Jenkins سحب منه المكتبة.

وده تأكيد إن الربط شغال صح .

It should be the same thing in your Git as well



🧩 ثانيًا: استخدام المكتبة داخل Jenkinsfile

- ➢ دلوقتي نروح على الـ pipeline اللي كنا شغالين عليه، وليكن الـ branch اسمه feature/advanced-demo

◆ الخطوة 1: نحذف الكود القديم

- o اللي هو الكود اللي كنا كاتبين فيه الـ Slack Notification أو أي function جوه Jenkinsfile لأننا خلاص هنستدعي نفس الكود من المكتبة.

◆ الخطوة 2: نستخدم Annotation اسمها @Library

- ➢ هي دي الطريقة اللي بتقول لـ Jenkins : استدعي المكتبة اللي اسمها كذا من الـ "Global Libraries"

📘 الصيغة : @Library('Dasher-trusted-shared-library') _

- Dasher-trusted-shared-library = الاسم اللي كتبناه في إعدادات Jenkins
- _ = Placeholder إجباري لتكملة الصيغة (مالوش وظيفة فعلية).

🧑‍🏫 ملحوظة:

لو نسيت الـ _ underscore ← Jenkins هيرمي Error في أول الـ build

```
1 @Library('dasher-trusted-shared-library') _
```

⚙️ ثالثًا: استدعاء Function من المكتبة

- ➢ قبل كده كنا بنستخدم مثلاً : slackNotification('SUCCESS')
- ➢ لكن دلوقتي بعد ما بقينا بنسحبها من المكتبة، لازم نراعي الصيغة الجديدة.
- ➢ ليه الصيغة بتتغير؟

لأن في الـ Shared Library ، كل ملف في مجلد vars/ بيتحوّل تلقائيًا لـ متغير أو method جاهز للاستخدام

لو جواه دالة اسمها call()

سابقًا كنا بنستدعيها بالشكل ده

```
post {
    always {
        slackNotificationMethod("${currentBuild.result}")
```

```
post {
    always {
        slackNotification("${currentBuild.result}")
```

دلوقت هانستدعيها بالشكل ده

🧩 مثال توضيحي للنقطة السابقة :

- ➢ لو عندنا في المكتبة ملف اسمه : **vars/slackNotification.groovy**
- ➢ وفيه الكود ده :

```
def call(String buildStatus = 'STARTED') {
    buildStatus = buildStatus ?: 'SUCCESS'

    def color

    if (buildStatus == 'SUCCESS') {
        color = '#47ec05'
    } else if (buildStatus == 'UNSTABLE') {
        color = '#d5ee0d'
    } else {
        color = '#ec2805'
    }

    def msg = "${buildStatus}: `${env.JOB_NAME}` #${env.BUILD_NUMBER}:\n${env.BUILD_URL}"

    slackSend(color: color, message: msg)
}
```

- • يبقى تقدر تستدعيه في Jenkinsfile ببساطة كده 👇

  slackNotification('SUCCESS')

  - o وده لأن Jenkins بيفهم إن الملف ده فيه دالة **call()**

🧠 رابعًا: ملخص خطوات التنفيذ السابقة :

1. احذف الكود القديم اللي كان بيبعت Slack Notification
2. أضف في أول Jenkinsfile :
   - o **@Library('Dasher-trusted-library') _**
3. استخدم الـ function الجاهزة:
   - o **slackNotification('SUCCESS')**
4. احفظ الملف (Save) واعمل Commit & Push

🚀 خامسًا: تجربة التنفيذ

- ➢ بعد الـ push ، Jenkins هيشوف التغيير ويعمل build جديد (Job #6 مثلاً)
- ➢ بعد ما الـ build يخلص:
  - • هتلاقيه نجح في console log
  - • وهتلاقي رسالة Slack وصلت بتقول إن الـ build رقم 6 نجح. ✅

📜 سادسًا: فحص الـ Logs

- ➢ عشان تتأكد إن Jenkins فعلاً حمّل المكتبة، افتح : Console Output → Filter → Search for "library"
- ➢ هتلاقي سطر بيقول: Loading library DasherTrustedLibrary@main
- ➢ وده معناه إنه فعلاً سحب الكود من:
  - • الـ Repo اللي انت محدده.
  - • الـ Branch = main
  - • والـ Commit ID اللي ظهر فوق.

🧩 سابعًا: محتوى الرسالة من المكتبة

- ➢ Jenkins هيجيب إعدادات الـ Slack (زي اللون، اسم القناة، إلخ) من نفس الكود اللي داخل المكتبة:

channel = "#jenkins-alerts"

color = "good"

وده معناه إن الكود اللي جوه المكتبة اشتغل واتنفذ بالكامل.

💡 ملاحظات مهمة

النقطة	التوضيح
لو حصل Error في تحميل المكتبة	راجع اسمها في Jenkins settings وتأكد إنه مطابق في Jenkinsfile
لو Slack Message ماوصلتش	راجع Token أو الـ Webhook URL في الكود داخل المكتبة
ممكن تعمل مكتبات كتير	كل واحدة لجزء مختلف (Slack – Deploy – Testing – Reports...)
ممكن تعمل فروع مختلفة	لكل إصدار أو بيئة (main = prod, dev = testing...)

🧭 الملخص النهائي

- **عملنا Verify إن Jenkins قرأ المكتبة من الـ Git repo**
- **استخدمنا _ @Library('Name') علشان نستدعيها.**
- **حذفنا الكود اليدوي واستدعينا الدوال من المكتبة.**
- **عملنا Commit ← Jenkins عمل Build ← Slack استقبل الإشعار.**
- **اتأكدنا من اللوجز إن Jenkins فعلاً شغّل المكتبة.**

Create Shared Library for Trivy Scan

🎯 المقدمة :

- ❑ الجزء ده بقى بيدخّلنا في مرحلة جديدة من موضوع Jenkins Shared Libraries وهي فكرة Versioning
- ❑ يعني إنك تقدر تعمل نسخ مختلفة من المكتبة وتستخدم كل واحدة حسب المشروع أو المرحلة.
- ❑ الهدف :
  - ينقل مرحلة الـ Trivy Scan (اللي بتفحص الـ Docker images من الـ vulnerabilities) إلى Shared Library مستقلة بدل ما تبقى مكتوبة في كل Jenkinsfile
  - وده بيخلينا نقدر نعيد استخدام نفس الكود في أي Pipeline بسهولة
  - نتحكم في الإصدارات (versioning) زي ما الشركات الكبيرة بتعمل.

🧩 أولاً: تجهيز الفرع الجديد للمكتبة (Versioning Concept)

🧠 فكرة بسيطة:

- بدل ما نفضل نعدل على نفس الـ main branch كل مرة،
- هنبدأ نعمل فروع (branches) لكل ميزة أو تحديث جديد في المكتبة.

◆ الخطوات:

```
git clone <repo-url>
cd <repo-folder>
git checkout -b Feature-TrivyScan
```

1. هو كان على الفرع الرئيسي main بتاع المكتبة .
2. عمل checkout لفرع جديد : git checkout -b feature/TrivyScan
   - ده معناه إنه أنشأ نسخة جديدة من المكتبة علشان يضيف فيها كود الـ TrivyScan
3. بعد كده دخل على مجلد vars/ ← المجلد ده في Shared Libraries هو المكان اللي بنحط فيه ملفات groovy. اللي بتحتوي على الـ pipeline steps أو functions الجاهزة.

📁 ثانيًا : إنشاء ملف جديد للمكتبة

- ➢ داخل مجلد vars/ ← أنشأ ملف اسمه: TrivyScan.groovy
  - ده اللي هيكون مسؤول عن تنفيذ كل خطوات الفحص (Scanning)

⚙️ ثالثًا: إنشاء Function للفحص ← vulnerability()

- ➢ جوه الملف الجديد كتب Function اسمها vulnerability() .. ودي وظيفتها إنها تشغّل الـ Trivy scan commands
- ➢ 📘 الكود الأساسي جايه Copy-Paste من Jenkinsfile القديم (اللي كان فيه stage الفحص)

Old jenkinsfile

```
stage('Trivy Vulnerability Scanner') {
    steps {
        sh '''
            trivy image siddharth67/solar-system:$GIT_COMMIT \
                --severity LOW,MEDIUM,HIGH \
                --exit-code 0 \
                --quiet \
                --format json -o trivy-image-MEDIUM-results.json

            trivy image siddharth67/solar-system:$GIT_COMMIT \
                --severity CRITICAL \
                --exit-code 1 \
                --quiet \
                --format json -o trivy-image-CRITICAL-results.json
        '''
```

◆ قبل التعديل:

لااحظ ان في الـ jenkinsfile القديم .. اسم الـ image مكتوب يدوي.

◆ بعد التعديل:

- لكن هنا بقى (بعد مانسخه في الـ trivyscan.groovy) بيعدّل عليه علشان يخليه ديناميكي مش Hardcoded
  - يعني هو عايز يخلي اسم الصورة يتبعت من Jenkinsfile نفسه (بدل مايكون مكتوب جوا الكود) ، فبيخلي الـ function تستقبل parameter اسمه **imageName**
  - وبالتالي كده الميثود دي تقدر تشتغل على أي صورة حسب الاسم اللي هيتبعت لها من Jenkinsfile.

```
def vulnerability(String imageName) {

    sh """
        echo image - ${imageName}

        trivy image ${imageName} \
            --severity LOW,MEDIUM,HIGH \
            --exit-code 0 \
            --quiet \
            --format json -o trivy-image-MEDIUM-results.json

        trivy image ${imageName} \
            --severity CRITICAL \
            --exit-code 1 \
            --quiet \
            --format json -o trivy-image-CRITICAL-results.json
    """
```

Example

اللي بيحصل بالضبط:

- Jenkinsfile بينادي vulnerability("nginx:latest")
- Jenkins يروح للمكتبة (trivyScan.groovy)
  - يلاقي إن الميثود فيها parameter اسمه imageName
  - فيبعت القيمة "nginx:latest" جوه الميثود اللي بتنفذ الأمر : trivy image nginx:latest ....

ملاحظات مهمة هنا

السطر	الشرح
`def vulnerability(String imageName)`	دي Function جديدة اسمها vulnerability بتستقبل باراميتر اسمه imageName.
`"echo "Scanning image: ${imageName}`	بتطبع اسم الصورة اللي بيتم فحصها (عشان يظهر في اللوجز).
`sh """ ... """`	بيشغّل أمر Trivy الحقيقي داخل الـ Jenkins stage.
`{imageName}$`	بنستخدمه بدل ما نكتب اسم الصورة يدوي → ديناميكي.

رابعًا: إضافة Function تانية لتحويل التقارير

- في Jenkinsfile القديم كان فيه بعد الفحص خطوة بتحوّل نتائج Trivy من JSON إلى HTML/XML

Old jenkinsfile

```
post {
    always {
        sh '''
            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/html.tpl" \
                --output trivy-image-MEDIUM-results.html trivy-image-MEDIUM-results.json

            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/html.tpl" \
                --output trivy-image-CRITICAL-results.html trivy-image-CRITICAL-results.json

            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/junit.tpl" \
                --output trivy-image-MEDIUM-results.xml  trivy-image-MEDIUM-results.json

            trivy convert \
                --format template --template "@/usr/local/share/trivy/templates/junit.tpl" \
                --output trivy-image-CRITICAL-results.xml trivy-image-CRITICAL-results.json
        '''
```

- فهو كمان نقل الخطوات دي لمكتبة علشان تبقى reusable .. بدل ما يعيدها في كل pipeline

```
Jenkinsfile    trivyScan.groovy U ●
shared-libraries > vars > trivyScan.groovy
            --quiet \
            --format json -o trivy-image-CRITICAL-results.json
        """
}

def reportsConverter() {
    sh '''
        trivy convert \
            --format template --template "@/usr/local/share/trivy/templates/html.tpl" \
            --output trivy-image-MEDIUM-results.html trivy-image-MEDIUM-results.json

        trivy convert \
            --format template --template "@/usr/local/share/trivy/templates/html.tpl" \
            --output trivy-image-CRITICAL-results.html trivy-image-CRITICAL-results.json

        trivy convert \
            --format template --template "@/usr/local/share/trivy/templates/junit.tpl" \
            --output trivy-image-MEDIUM-results.xml  trivy-image-MEDIUM-results.json

        trivy convert \
            --format template --template "@/usr/local/share/trivy/templates/junit.tpl" \
            --output trivy-image-CRITICAL-results.xml trivy-image-CRITICAL-results.json
    '''
}
```

➢ **كتب function تانية في نفس الملف:**

💡 دي وظيفتها ببساطة:

- تاخد الملف الناتج من Trivy (trivy-report.json)
- وتحوله لتقارير مفهومة أكثر بصيغة:
  - HTML ← لعرضها فيbrowser
  - XML ← لاستخدامها في أدوات تحليل Jenkins

🔧 خامسًا: حفظ ورفع التعديلات

بعد ما خلّص الدالتين:

1. عمل Save للملف.
2. نفّذ :
   - **git add .**
   - **git commit -m "Added TrivyScan shared library functions"**
   - **git push origin feature/TrivyScan**

كده الفرع الجديد feature/TrivyScan بقى موجود في GitHub وفيه كل الأكواد اللي تخص الفحص والتحويل.

📜 سادسًا: التحقق من الفرع

راح على GitHub أو CLI .. ولقى فعلاً:

- Branch جديد اسمه **feature/TrivyScan**
- فيه الملف vars/TrivyScan.groovy
- موجودة على فرع خاص (مش في main).
- فيها:
  - Vulnerability() ← بتعمل scan.
  - reportsConverter() ← بتحول التقارير.

🧠 كده اتعمل إيه بالضبط؟

الخطوة	النتيجة
أنشأ فرع جديد	ده الإصدار الجديد من المكتبة (versioned branch)
أنشأ ملف TrivyScan.groovy	فيه كل خطوات فحص الثغرات
كتب Function اسمها vulnerability()	للفحص باستخدام Trivy
كتب Function اسمها reportsConverter()	لتحويل تقارير الفحص
عمل Commit & Push	علشان Jenkins يقدر يستخدم الفرع الجديد

🔮 في الدرس الجاي (اللي هو مكمل للكود ده)

➢ هيشرح:

- إزاي نستدعي الفرع الجديد من الـ pipeline
  - باستخدام : **@Library('DasherTrustedLibrary@feature/TrivyScan') _**

🚀 وبكده تقدر تستخدم أي إصدار من نفس المكتبة في الـ pipelines المختلفة ✌️

Load Trivy Scan Library in Jenkins Pipeline

🎯 الفكرة العامة :

- ❑ الجزء دا بيشرح إزاي نحمل ونستخدم Shared Library اسمها TrivyScan داخل Jenkins Pipeline
- ❑ الفكرة إننا كنا في الدرس اللي قبله أنشأنا Shared Library فيها سكريبت Groovy اسمه TrivyScan.groovy،
  وجواه دالتين (methods) :
  1. Vulnerability() ← بتعمل فحص أمني لصورة Docker باستخدام أداة اسمها Trivy
  2. reportsConverter() ← بتحوّل نتايج الفحص (الـ JSON) إلى تقارير HTML و XML علشان نقدر نعرضها)
- ❑ في الجزء دا، هانشوف إزاي:
  - نحمل الـ library دي جوّا الـ Jenkins pipeline
  - نستدعي الدالتين اللي جوّاها من مرحلة (stage) معينة في الـ pipeline
  - ونتعامل مع الأخطاء اللي ممكن تحصل لما نستخدمها في declarative pipeline

---

🧩 الشرح التفصيلي خطوة بخطوة

1 إعداد الـ Shared Library

- ➢ هو بيبدأ بالشرح إن عنده Shared Library على فرع (branch) اسمه feature/TrivyScan
  وجواه سكريبت TrivyScan.groovy فيه دالتين:
  - vulnerability()
  - reportsConverter()
- ➢ الهدف دلوقتي إنه يربط الـ pipeline بالفرع دا علشان يقدر يستخدم الدالتين.

---

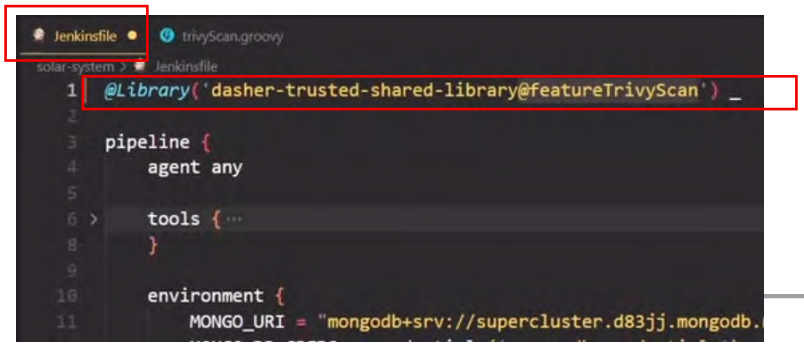
2 إعداد Jenkins Global Pipeline Library

- ➢ يروح على Jenkins > Manage Jenkins > Configure System > Global Pipeline Libraries
  وهناك بيشوف إن الـ default version = main
- ➢ لكن علشان نقدر نستخدم فرع تاني (زي feature/TrivyScan) ← لازم نفعّل خيار اسمه:
  ✅ "Allow default version to be overridden"
- ➢ ده معناه إن أي مطور يقدر يكتب في الـ Jenkinsfile اسم الفرع اللي عايز يستخدم منه الـ library بدل ما يفضل ثابت على "main"

---

3 استدعاء الـ Library داخل الـ Jenkinsfile

- ➢ في أول الـ Jenkinsfile ، قبل أي stages عموما، بنضيف سطر زي كده :
- ➢ ◆ الكود دا معناه:
  - حمّل الـ Shared Library اللي موجودة في فرع اسمه feature/TrivyScan
- ➢ بكده، أي سكريبت Groovy داخل فولدر vars في الـ Library دي هيبقى متاح نستخدمه هنا،
  زي مثلاً TrivyScan.groovy أو SlackNotification.groovy



4️⃣ استدعاء الـ Function من الـ Library

❑ بعد كده بدل ما أكتب كل أوامر Trivy جوّا الـ stage بتاعته اللي في الـ Jenkinsfile كل مرة ، أنا هخلي الـ stage بتاعة Trivy دي تستخدم الـ function الجاهزة اللي جوّا Shared Library

➢ وبالتالي ففي Stage('Trivy Vulnerability Scanner') .. هانكتب زي مافي الصورة كدا:

✅ الشرح:

- TrivyScan ← هو اسم ملف Groovy اللي جوّا الـ vars folder
- Vulnerability() ← هو اسم الميثود اللي جوّاه.
- reportsConverter() ← اسم الميثود جواه بردو .. ودي بتاع تحويل التقارير
- ('siddharth67/solar-system:$GIT_COMMIT') ← دي القيمة اللي هاتتمرر وتتبعت مباشرة للـ parameter اللي اسمه ImageName اللي جوا الـ function (راجع ملف TrivyScan.groovy)

ودي عموما طريقة استدعاء method جوّا Shared Library في declarative pipeline

```
Jenkinsfile ×   trivyScan.groovy   slackNotification.groovy
solar-system > Jenkinsfile
            sh 'docker build -t siddharth67/solar-system:$GIT_COMMIT .'
        }
    }

    stage('Trivy Vulnerability Scanner') {
        steps {
            script {
                trivyScan.vulnerability("siddharth67/solar-system:$GIT_COMMIT")
            }
        }
        post {
            always {
                script {
                    trivyScan.reportsConverter()
                }

                publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',

                publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: './',
```

Note that:
The way you call any method is :
Name of script.method(‘’)
But:
If it was a simple call() method, then the way you call it is :
the name of the script

You can put here any input parameter (based on your scenario) as if it was required & if not, let it empty as in reportConverter()

الميثود دي مش محتاجة parameters ، هي بس بتشغّل أوامر تحويل الـ JSON إلى HTML/XML

6️⃣ ⚠️ المشكلة اللي حصلت (Error)

Extending with Shared Libraries

➢ لما شغّل الـ build ، حصل خطأ:

"method calls on objects not allowed outside the script block"

➢ يعني Jenkins مش بيسمح تستدعي methods من Shared Library مباشرة جوّا declarative pipeline إلا لو كانت جوّا script {} block

🧠 السبب:

- Jenkins declarative syntax بتتعامل مع stages بشكل declarative (إعلاني)
- لكن Shared Library methods تعتبر Groovy code، وده محتاج script context علشان يتنفّذ.

📌 الحل:

- نحط أي استدعاء لمثود من Shared Library جوّا script block، زي ما شفنا فوق في الكود.

7️⃣ بعد التعديل

➢ بعد ما أضاف script {} حوالين كل method ، عمل commit جديد:

o git add Jenkinsfile → git commit -m "Loading Trivy Scan library" → git push

بعد كده Jenkins شغّل build رقم 8 .. والمرة دي اشتغل بنجاح. ✅

---

8 التأكد من التنفيذ

لما راح على Console Output، شاف الآتي:

- Jenkins فعلاً حمّل الـ library من الفرع feature/TrivyScan
- ظهر echo للـ ImageName (اللي بيتم تمريره للدالة)
- أوامر Trivy اتنفذت على الصورة فعلاً.
- تم إنشاء تقارير الفحص.
- وظهر Slack Notification إن الـ build نجح 🎉

```
echo image - siddharth67/solar-system:c9dc5eb9b28147642a87fb0e2c8f9f2845fa4b1d trivy image siddharth67/solar-system:c9dc5eb9b28147642a87fb0e2c8f9f2845fa4b1d --severity LOW,MEDIU...
+ echo image - siddharth67/solar-system:c9dc5eb9b28147642a87fb0e2c8f9f2845fa4b1d
image - siddharth67/solar-system:c9dc5eb9b28147642a87fb0e2c8f9f2845fa4b1d
+ trivy image siddharth67/solar-system:c9dc5eb9b28147642a87fb0e2c8f9f2845fa4b1d --severity LOW,MEDIUM,HIGH --exit-code 0 --quiet --format json -o trivy-image-MEDIUM-results.json
+ trivy image siddharth67/solar-system:c9dc5eb9b28147642a87fb0e2c8f9f2845fa4b1d --severity CRITICAL --exit-code 1 --quiet --format json -o trivy-image-CRITICAL-results.json
trivy convert --format template --template "@/usr/local/share/trivy/templates/html.tpl" --output trivy-image-MEDIUM-results.html trivy-image-MEDIUM-results.json trivy convert --format template --t...
+ trivy convert --format template --template @/usr/local/share/trivy/templates/html.tpl --output trivy-image-MEDIUM-results.html trivy-image-MEDIUM-results.json
+ trivy convert --format template --template @/usr/local/share/trivy/templates/html.tpl --output trivy-image-CRITICAL-results.html trivy-image-CRITICAL-results.json
+ trivy convert --format template --template @/usr/local/share/trivy/templates/junit.tpl --output trivy-image-MEDIUM-results.xml trivy-image-MEDIUM-results.json
+ trivy convert --format template --template @/usr/local/share/trivy/templates/junit.tpl --output trivy-image-CRITICAL-results.xml trivy-image-CRITICAL-results.json
```

🧠 الملخص : أوامر وأكواد مهمة

⬆️ لتحميل مكتبة معينة من فرع معين. ← @Library('feature/TrivyScan') _

⬆️ خطوات إنشاء فرع جديد ورفع التعديلات عليه.

git checkout -b feature/TrivyScan → git add . →
git commit -m "Added Trivy Scan shared library" → git push origin feature/TrivyScan

---

Best Practices 💡

1. ✅ دايماً فعّل "Allow default version to be overridden"
   علشان تقدر تستخدم فروع مختلفة أثناء التطوير والتجربة.
2. ✅ دايماً استخدم {} script لما تستدعي methods من Shared Library داخل declarative pipeline
3. ⚙️ نظّم كل function في Shared Library حسب مهمتها (مثلاً Trivy ، Slack، Deployment)
4. 🔒 استخدم branches مستقلة للتجارب على Shared Libraries
   وما تدمجهاش في main إلا لما تتأكد إنها شغالة تمام.
5. 📄 احتفظ بالتقارير في Artifacts عشان تراجعها بعد كل build

🧭 الملخص النهائي

- الجزء ده بيورينا إزاي نستخدم Shared Library جوّا Jenkins declarative pipeline
- الفكرة إننا بنعيد استخدام كود جاهز بدل ما نكرره في كل Jenkinsfile
- أول خطوة: نفعّل الـ feature اللي تسمح نغير الـ branch
- ثاني خطوة: نحمل الـ library باستخدام @Library('branch')
- ثالث خطوة: نستدعي الميثودز من الـ library جوّا script {} block
- أخيراً: نشغّل الـ build ونتأكد من التنفيذ والتقارير والـ Slack notification

what&why&create Library Resource

🎯 الفكرة العامة

- ❑ الدرس ده بيتكلم عن Library Resources في Jenkins Shared Library
- ❑ الهدف منه هو إزاي تخزن ملفات ثابتة زي سكربتات Shell أو ملفات YAML داخل مكتبتك وتستخدمها بعد كده في الـ Jenkins Pipeline أو في Groovy Scripts بدل ما تكتب كل حاجة هارد كود.
- ❑ 📦 يعني بدل ما نكتب أوامر Trivy أو Configs جوّا الكود نفسه، نخزنها كملفات مستقلة داخل فولدر اسمه resources ونخلي Jenkins يحمّلها وقت التنفيذ.

🧩 الشرح التفصيلي خطوة بخطوة

1. ◆ المقدمة : What are Library Resources ؟

- ➢ الـ Library Resources بتسمح إنك تحط ملفات مش Groovy (زي sh, .yaml, .json) جوه مجلد اسمه resources/ داخل الـ Shared Library بتاعتك.
- ➢ بعد كده تقدر تقرأها أو تشغلها من أي سكريبت Groovy تاني داخل المكتبة.
  - وده بيفيدك لما تكون عندك أوامر كتير أو Configs عايز تحتفظ بيها برا الكود.

---

2. ◆ المشكلة اللي بيحلها الدرس

- ➢ في الدرس السابق (trivyScan.groovy) ، كان حاطط أوامر Trivy Scan بشكل hardcoded زي كده:

```
def vulnerability(String imageName) {

    sh """
        echo image - ${imageName}

        trivy image ${imageName} \
            --severity LOW,MEDIUM,HIGH \
            --exit-code 0 \
            --quiet \
            --format json -o trivy-image-MEDIUM-results.json

        trivy image ${imageName} \
            --severity CRITICAL \
            --exit-code 1 \
            --quiet \
            --format json -o trivy-image-CRITICAL-results.json
    """
}
```

- ➢ المشكلة هنا 👇
  كل مرة عايز تغيّر الإعدادات
  (زي الـ severity أو اسم الصورة أو exit code) ..
  هتضطر تعدّل الكود نفسه وتعمل commit جديد

➕ وده مش عملي، خصوصًا لو كل مطور عايز يستخدم إعدادات مختلفة.

---

3. ◆ الحل — استخدام Shell Script داخل resources

- ➢ الحل هو:
  نعمل (trivy.sh) static shell script ونحطه جوّه الـ resources folder ، ويكون بيشغّل أمر Trivy Scan لكن باستخدام parameters ديناميكية بدل القيم الثابتة.
- ➢ بمعنى:
  - بدل ما أكتب image_name و HIGH جوّا الكود،
  - أكتب placeholders زي $1, $2, $3 ، والـ Jenkins Pipeline يمرر القيم دي وقت التنفيذ.

---

4. ◆ إنشاء هيكل الـ folders

- ➢ داخل الـ Shared Library ، نعمل المجلدات دي:

```
(root)
├── vars/
└── resources/
    └── scripts/
        └── trivy.sh
```

➕ السكربت trivy.sh هيتكتب جوّه resources/scripts/

5. ◆ كتابة سكربت trivy.sh

➢ السكربت بيحتوي على أمر Trivy Scan ، لكن بدلاً من القيم الثابتة، بيستخدم placeholders :

```
Jenkinsfile   trivyScan.groovy   trivy.sh U   slackNotification.groovy
shared-libraries > resources > scripts > trivy.sh
#!/bin/bash

echo imageName - $1
echo severity - $2
echo exitCode - $3

trivy image $1 \
    --severity $2 \
    --exit-code $3 \
    --format json -o trivy-image-$2-results.json
```

✅ الشرح:

- $1 = مستوى الخطورة (مثلاً HIGH, CRITICAL)
- Exit code = $2
- $3 = اسم الصورة (image name)

➕ بكده أي حد يقدر يمرّر القيم اللي عايزها من الـ pipeline بدل ما يغيّر الكود.

➕ طاب السؤال بقا .. ازاي نستخدم الـ shell script ده في الـ shared library ؟؟

للعلم بالشئ : كنا ممكن نشتغل عادي في نفس الـ trivyScan.groovy اللي شغالين فيه من ساعتها .. بس غالبا هو حب يتحنيك شوية وعمل فايل جديد اسمه trivyScanScript.groovy

6. ◆ إنشاء سكريبت Groovy جديد (TrivyScanScript.groovy)

➢ بعد كده بيعمل ملف جديد داخل vars اسمه : trivyScanScript.groovy

➢ وده اللي هيستدعي سكربت الـ shell اللي عملناه.

➢ جواه هتكون في method اسمها vulnerability(config)

- الـ config هنا عبارة عن Map فيها key-value pairs للخيارات اللي المستخدم هيبعته.
- مثلاً:

```
Jenkinsfile   trivyScan.groovy   trivyScanScript.groovy U   trivy.sh U   slackNotification.groovy
shared-libraries > vars > trivyScanScript.groovy
def vulnerability(Map config = [:]) {
    sh "./trivy.sh ${config.imageName} ${config.severity} ${config.exitCode}"
}
```

طاب ازاي بقا الـ Libraries دي هاتـ access الـ trivy.sh ؟!؟!؟!
ملاحظة بسيطة .. لسا هانضيف خطوات في الفايل ده

✅ الشرح:

- استخدم Map config علشان يسمح بمرونة في تمرير عدد مختلف من المتغيرات.
- config = [:] ← معناها لو محدش بعث حاجة، الكود مش هايبوظ.

7. ◆ مفهوم الـ Map Parameters

➢ الـ Map دي بنستخدمها علشان المطور اللي هيستدعي الـ library من Jenkinsfile يقدر يحدد القيم بسهولة.

➢ في الـ Jenkinsfile ، بدل ما نكتب كده:

o TrivyScan.vulnerability("myimage", "HIGH", 1)

➢ نكتب كده بشكل أوضح:

```
TrivyScanScript.vulnerability([
    imageName: "myimage",
    severity: "HIGH",
    exitCode: 1
])
```

Using map :
Provide flexible input to the functions or the methods which you are defined in shared library
Also you can use the map parameter which can allow the caller to pass different set of KEY-VALUE pair configurations without requiring a fixed set of arguments.

➕ وده يخلي الكود أسهل في القراءة والصيانة.

8. ◆ تحميل سكربت الـ Shell داخل الـ Library (load script)

الحل : هانعمل library تانية اسمها loadscript
ونديها الاسم بتاع الـ file اللي بنحاول نأكسسه

- عمل سكريبت جديد اسمه : **loadScript.groovy**
  - وده هدفه الوحيد إنه يقرأ سكربت trivy.sh من resources/scripts/ ويكتبه داخل الـ workspace وقت التنفيذ.
- عمل كده باستخدام دالة اسمها: **libraryResource("scripts/${config.name}")**

```
def call(Map config = [:]) {
    def scriptData = libraryResource "scripts/${config.name}"
    writeFile file: "${config.name}", text: scriptData
    sh "chmod +x ./${config.name}"
}
```

طاب ليه استخدمنا ${config.name}
ومستخدمناش اسم الفايل علطول؟؟!

قالك علشان احنا في الـ
trivyScanScript.groovy أضفنا
.. loadScript(name: trivy.sh)
اللي هو فيما معناه يعني اننا باصينا الـ
input argument name اللي بدوره
هايُستخدم في الـ loadScript.groovy

```
def vulnerability(Map config = [:]) {
    loadScript(name: 'trivy.sh')
    sh "./trivy.sh ${config.imageName} ${config.severity} ${config.exitCode}"
}
```

❑ شرح الكود اللي في الـ loadScript.groovy :

1. libraryResource() ✅ ← بيحمّل الملف trivy.sh كـ string من resources
2. writeFile ← بيكتب المحتوى جوّه workspace
3. chmod +x ← بيخلي الملف قابل للتنفيذ.

بكده بقى عندنا نسخة من trivy.sh جوه الـ Jenkins workspace ممكن نشغلها بأمان.

❑ شرح الكود اللي في الـ trivyScanScript.groovy :

➢ ربط trivyScanScript بالـ load script .. بمعني :

- السكربت الأول بيحمّل الملف ويجهزه ← loadScript(name: trivy.sh)
- السكربت التاني بيشغّله فعليًا بالـ parameters اللي جايه من الـ pipeline

📘 libraryResource دي هي المفتاح الأساسي للدرس:

- بتدور على الملف داخل resources/
- وترجعه على شكل string (مش كملف قابل للتنفيذ)

---

11. ◆ ملخص العملية الكاملة

1. trivyScanScript.groovy ← بيتكلم مع Jenkinsfile
2. بيطلب من loadScript.groovy يحمّل الملف trivy.sh
3. loadScript بيستخدم libraryResource علشان يقرأ الملف ويكتبه في workspace
4. بيعمل له chmod +x
5. بيرجع التحكم لـ TrivyScanScript اللي بيشغّل الـ shell script فعليًا بأوامر Trivy المطلوبة.

Best Practices 💡

1. ✅ استخدم resources/ لتخزين أي ملفات ثابتة (Shell, YAML, JSON)
2. ✅ دايمًا استخدم libraryResource() لتحميل الملفات بدل hardcoding paths
3. ✅ استخدم map parameters لمرونة أكبر في الدوال.
4. ⚙️ استخدم سكربت منفصل زي loadScript.groovy لإعادة الاستخدام في مشاريع تانية.
5. 🔒 خليك حذر من تعديل permissions جوه Jenkins .. والأفضل تستخدمها فقط مع سكربتات آمنة.

🧭 الملخص النهائي

➢ الدرس ده بيعلّمنا:

- إزاي نستخدم Library Resources علشان نحمل وننفذ ملفات shell ثابتة داخل Jenkins Shared Library

➢ الترتيب كده 👇

1. أنشأنا فولدر resources/scripts/
2. كتبنا trivy.sh فيه.
3. عملنا loadScript.groovy لتحميل الملف في runtime
4. عملنا trivyScanScript.groovy لاستدعائه وتشغيله بالـ parameters
5. استخدمنا map config لمرونة أكتر.
6. وفي النهاية بقينا نقدر نغير إعدادات Trivy Scan من الـ pipeline بسهولة بدون تعديل الكود الأساسي.

Load Trivy Scan Script Library in Jenkins Pipeline

🎯 الفكرة العامة

- ❑ الجزء ده بيكمل على نفس الدرس اللي قبله، وبيشرح إزاي نستخدم مكتبة الـ TrivyScanScript اللي أنشأناها في الـ Jenkinsfile فعليًا، ونتعامل مع الأخطاء اللي بتحصل أثناء التشغيل.
- ❑ هانشرح إزاي نحمل الـ Groovy Script الجديد اللي عملناه (trivyScanScript.groovy) من الـ Shared Library جوه الـ Jenkinsfile ، ونستخدمه بدل السكريبت القديم اللي كان اسمه Trivy.scan
- ❑ السكربت الجديد بيسمح إننا نمرر متغيرات زي:
  - اسم الصورة (image name)
  - مستوى الخطورة (severity)
  - exit code .. بدل ما تكون القيم دي مكتوبة جوه الكود نفسه.

🧩 الشرح خطوة بخطوة

1. ◆ تحميل المكتبة الجديدة داخل Jenkinsfile

```
stage('Trivy Vulnerability Scanner') {
    steps {
        script {
            trivyScan.vulnerability("siddharth67/solar-system:$GIT_COMMIT")
        }
```

- ➢ أول حاجة هانرجع للـ Jenkinsfile علشان نستدعي السكربت الجديد .
  - قبل كده كنا بنستخدم: Trivy.scan()
  - دلوقتي بنغيرها إلى: trivyScanScript.vulnerability()

```
stage('Trivy Vulnerability Scanner') {
    steps {
        script {
            trivyScanScript.vulnerability(imageName:"siddharth67/solar-system:$GIT_COMMIT", severity:"LOW", exitCode:"0")
        }
    }
```

- ➢ 📘 السبب:

لأن اسم الملف داخل vars هو TrivyScanScript.groovy ، واسم الـ function جوّاه هو vulnerability() ، وبالتالي ده الشكل الصحيح للاستدعاء.

---

2. ◆ تمرير المتغيرات المطلوبة

- ➢ الدالة vulnerability() محتاجة 3 متغيرات:
  1. اسم الصورة (image name)
  2. مستوى الخطورة (severity)
  3. Exit code
- ➢ وبالتالي كتبنا الكود كده داخل الـ Jenkinsfile (زي مافي الصورة السابقة)

🧠 بس هنا لازم نخلي بالنا إن المتغيرات بتتفصل بـ فواصل (commas) ← علشان Groovy تقرأهم كـ parameters

---

3. ◆ تكرار الخطوات لعدة مستويات خطورة

- ➢ بعد كده حب يختبر أكتر من مستوى خطورة، فقرر يكرر نفس السطر 4 مرات لتجربة كل نوع من أنواع الـ severity :

```
ript {
  trivyScanScript.vulnerability(imageName:"siddharth67/solar-system:$GIT_COMMIT", severity:"LOW", exitCode:"0")
  trivyScanScript.vulnerability(imageName:"siddharth67/solar-system:$GIT_COMMIT", severity:"MEDIUM", exitCode:"0"
  trivyScanScript.vulnerability(imageName:"siddharth67/solar-system:$GIT_COMMIT", severity:"HIGH", exitCode:"0")
  trivyScanScript.vulnerability(imageName:"siddharth67/solar-system:$GIT_COMMIT", severity:"CRITICAL", exitCode:"1")
```

✅ الملاحظات:

- الصورة (image name) ثابتة.
- الـ severity بيتغير في كل مرة.
- الـ exit code بقى 1 في الحالة الأخيرة علشان تعتبر فشل في حالة وجود مشاكل حرجة (Critical)

---

4. ◆ الحفاظ على باقي المراحل زي ما هي

❖ وضّح إنه مش هيغير خطوة الـ "report conversion" .. لأنها بتستخدم مكتبة قديمة وبتشتغل تمام، فقال إنه هيسيبها زي ما هي علشان مش محتاج يعدّل عليها.

---

5. ◆ حفظ التعديلات وتشغيل Build جديد

➢ بعد ما خلّص التعديلات، عمل commit وكتب رسالة: Loaded new lib

➢ وبعدين Push للتغييرات على الفرع (feature branch) ، وطبعًا Jenkins شاف التغيير، فبدأ Build رقم 9 تلقائيًا.

---

6. ◆ ظهور خطأ 😳 MissingPropertyException

➢ شاف إن الـ pipeline فشل، ولما فتح اللوجز لقى الرسالة دي:

MissingPropertyException: No such property: Trivy for class: trivyScanScript

➢ 📘 المعنى:

- Groovy بيحاول يلاقي خاصية أو متغير اسمه Trivy داخل TrivyScanScript، بس مش موجود.
  وده معناه إن في خطأ في كتابة الكود (Typo)

---

7. ◆ تصحيح الخطأ

➢ لما رجع للـ trivyScanScript.groovy، اكتشف إن السطر اللي بيحمّل السكربت كان كده:

o loadScript(name: trivy.sh)

o وده خطأ ❌ لأن اسم الملف لازم يكون بين علامات اقتباس (' ') علشان Groovy يتعامل معاه كـ string

➢ ✅ الصحيح:

o loadScript(name: 'trivy.sh')

➢ بعد كده حفظ التعديل، كتب رسالة commit : Typo

➢ وعمل push جديد.

---

8. ◆ تشغيل Build يدوي بعد الإصلاح

➢ المدرب دخل على Jenkins ← Branches ← واختار الفرع feature-advanced-demo ← وعمل Trigger build يدوي.

➢ بدأ Build رقم 10 .. و استنى شوية لحد ما خلص بنجاح ✅

---

9. ◆ التحقق من النتائج بعد نجاح الـ Build

➢ دلوقتي لما راح على اللوجز شاف التالي:

- مرحلة Trivy vulnerability scanner اشتغلت بنجاح 🎉
- Jenkins شغّل 4 أوامر مختلفة للـ severities : LOW → MEDIUM → HIGH → CRITICAL
- كمان بعدهم شغّل مرحلة تحويل التقارير(report conversion)

لما فتح اللوجز، شاف كل أوامر Trivy اشتغلت فعلاً واحدة ورا التانية والـ reports اتولدت بنجاح.

10. ◆ التحقق من الـ Library Resource في الـ Logs

➢ المدرب كمان حب يتأكد إن Jenkins فعلاً استخدم الـ library resource .. فبحث في الـ log عن كلمة LibraryResource ولقى الآتي:

```
[Pipeline] libraryResource
[Pipeline] writeFile
[Pipeline] sh
+ chmod +x trivy.sh
```

✅ وده بيأكد إن Jenkins :

1. حمّل ملف trivy.sh من resources
2. كتب نسخة منه داخل workspace
3. خلى الملف executable
4. شغّله 4 مرات بالقيم المختلفة اللي حددها في Jenkinsfile

🧠 الفكرة التقنية اللي اتطبقت هنا

➢ الدرس ده بيوضح قوة الفصل بين الكود الثابت والكود الديناميكي:

- ملفات الـ shell (زي trivy.sh) بتتحفظ جوه resources وتستخدم كـ assets ثابتة.
- ملفات Groovy (زي TrivyScanScript.groovy) بتتولى تشغيلها وتمرير المتغيرات.
- Jenkinsfile هو اللي بيستدعيها وبيحدد الـ parameters وقت التشغيل.

💡 Best Practices

1. ✅ دايمًا استخدم strings بين quotes في أي call فيه parameters نصية.
2. ✅ استخدم build logs علشان تتأكد من الـ stages اللي اشتغلت فعلاً.
3. ✅ لو عندك نفس الكود بيتكرر بقيم مختلفة ← استخدم looping أو parameterization بدل التكرار.
4. ✅ احرص إن الـ libraryResource يتحمل جوه workspace قبل التنفيذ.
5. ⚙️ سيب المراحل اللي شغالة مظبوطة زي ما هي، متغيرهاش إلا لو محتاج.

🧭 الملخص النهائي

➢ الدرس ده شرح:

إزاي نحمل ونستخدم مكتبة Groovy جديدة (TrivyScanScript) داخل Jenkinsfile ، ونمرر لها متغيرات مختلفة لتشغيل سكربت shell ثابت (trivy.sh) .. وكل ده باستخدام LibraryResource

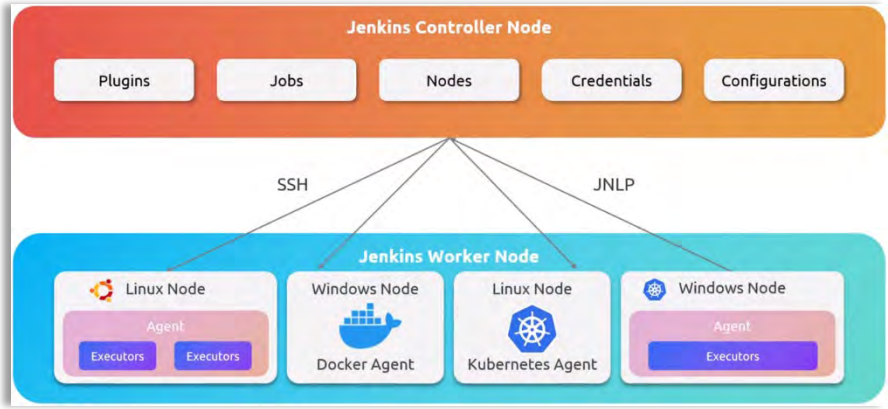
➢ الخطوات كانت كده 👇

1. استدعاء TrivyScanScript من Jenkinsfile
2. تمرير القيم (image name, severity, exit code)
3. تكرارها لأكتر من مستوى خطورة.
4. عمل commit و push وتشغيل build جديد.
5. تصحيح خطأ بسيط في string parameter
6. تشغيل build تاني بنجاح.
7. التحقق من تنفيذ أوامر Trivy والـ libraryResource فعلاً من الـ logs

Types of Agents

المقدمة:

- الجزء دا بيتكلم عن أنواع الـ Agents في Jenkins وازاي بيتربطوا بالـ controller ، وكمان بيشرح أمثلة عملية على طريقة تعريفهم جوه الـ Jenkinsfile
- فالفكرة إن الـ Agent هو اللي بينفذ الـ Jobs ، يعني هو العامل أو الـ worker اللي بياخد الأوامر من الـ controller ويبدأ يشغل خطوات الـ pipeline
- وده ممكن يكون جهاز فعلي (physical server) ، أو virtual machine ، أو حتى container



أولاً: يعني إيه Agent بالضبط؟

- الـ Agent هو ببساطة جزء من Jenkins مسؤول عن تشغيل الـ jobs اللي انت بتبنيها في الـ pipeline
- وكمان هو اللي بينفذ كل Stage من Stages الـ CI/CD pipeline

تخيل إن Jenkins controller هو "المدير"، والـ Agents هما "العمال" اللي بيشتغلوا على المهام اللي المدير بيوزعها.

- كل Agent بيستخدم executor، وده اللي بينفذ فعلاً الأوامر.

أنواع الـ Agents من ناحية الاتصال:

الـ Agent بيرتبط بالـ controller بطرق مختلفة حسب البروتوكول المستخدم:

- JNLP Agent ← بيستخدم Java Web Start (غالبًا لما الجهاز يكون وراء firewall)
- SSH Agent ← بيربط الـ agent بالـ controller عن طريق الـ SSH protocol

وفي الحالتين، الـ agent بيحتاج يكون عليه الأدوات (tools) اللي هيستخدمها الـ pipeline ، زي:

- Git – Maven – Node.js
- أو أي dependency تانية.

Agents مبنية على Docker أو Kubernetes :

- لو انت شغال في بيئة modern ، بدل ما تفضل تنزل أدوات على السيرفرات، فممكن تستخدم Docker Agents
  - يعني كل build job هيشتغل في container نظيف ومخصص له، ومليان بالأدوات اللي محتاجها فقط.
- فبدل ما تثبت Node.js أو Python على كل worker node ، بتعمل image فيها الأدوات دي وتستخدمها كـ agent
  - الميزة هنا:
    - كل build شغال في بيئة isolated (مفيش تضارب بين المشاريع)
    - Consistency بين البيئات.
    - لو build بوّظ البيئة بتاعته ← الـ container بيتحذف ويتبني من جديد.
- وبرضه ممكن تستخدم Kubernetes agents، وده لما Jenkins يشغل Pod مؤقت في Cluster مخصوص للـ build ، ولما يخلص الـ job .. بيحذفه.

💪 أنواع الـ Agents الأساسية في Jenkins :

1️⃣ Permanent Agents (الثابتة) :

- دي السيرفرات الدائمة اللي دايمًا جاهزة للشغل.
- زي إنك عندك 3 سيرفرات دايمًا أونلاين، كل واحد مجهز بأدوات معينة.
- مثلاً:
  - واحد عليه JDK ← للبروجكتات الجافا
  - واحد عليه Node.js ← للبروجكتات الجافاسكريبت
- ✅ مميزاتها:
  - جاهزة دايمًا.
  - بيئاتها ثابتة.
- ⚠️ عيبها:
  - بتستهلك موارد طول الوقت حتى لو مفيش build شغال.
  - لازم تصينها دايمًا عشان تكون كويسة.

2️⃣ Docker Agents 🐳

- زي ما قلنا، دي temporary containers بتشتغل وقت الـ build بس.
  - تقدر تعتبرها زي "مقاول مؤقت" بيخلص الشغل ويمشي.
- مثال عملي:

بدل ما تسطب Node.js على سيرفر حقيقي، بتستخدم image فيها Node ، زي:

```
agent {
  docker {
    image 'node:latest'
  }
}
```

- لما Jenkins يبدأ build :
  - بيشغل container جديد من image node:latest
  - ينفذ الكود
  - وبعدين يمسح الـ container .. يعني نظافة وثبات ومرونة 💪

3️⃣ Cloud-based Agents ☁️

- دول بقى اللي شغالين على Cloud (زي AWS ، Azure، أو GCP)
  - يعني Jenkins بيروح يستدعي instance جديدة لما يبقى فيه build ، ولما يخلص يقفلها.

🔥 الميزة:

- Scalability ممتازة.
- بتدفع بس لما يكون فيه build شغال.
- مناسبة جدًا للمشاريع الكبيرة أو اللي الـ load فيها متغير.

➕ ولو بتستخدم Kubernetes ، Jenkins ممكن ينشئ Pod مخصوص للـ job ويقفله بعدين.

4️⃣ Label-based Agents 🏷️

- هنا Jenkins بيستخدم Labels بدل ما تحدد جهاز معين.
  - يعني كل Agent بياخد label يعبر عن قدراته.
- مثلاً:
  - agent عليه Java tools ← label : java
  - agent عليه Windows tools ← label : windows

وبعدين في Jenkinsfile ، تقول : agent { label 'java' } .. ساعتها Jenkins هيختار أي agent عليه نفس الـ label

➢ الميزة:

- مرونة عالية جدًا.
- تقدر تشغل jobs على أي agent مناسب من غير ما تحدد جهاز بعينه.

🧠 أمثلة Jenkinsfile :

🧩 Example 1

❖ agent any

◆ يعني Jenkins ممكن يشغل الـ pipeline على أي agent متاح .

```
Terminal
$ cat Jenkinsfile
pipeline {
  agent any  // Use any available agent
  stages {
    stage('Build') {
      steps {
        sh 'echo Running on ${NODE_NAME}'        }
    }
  }
}
```

🧩 Example 2

❖ agent { label 'my-agent' }

◆ هنا Jenkins هيختار agent معين عليه label اسمه my-agent ولاااازم تكون ضايف label دا من إعدادات Jenkins للـ node

```
Terminal
$ cat Jenkinsfile
pipeline {
  agent {
    label 'my-agent'  // Run on agent with label "my-agent"
  }
  stages {
    stage('Build') {
      steps {
        sh 'echo Running on ${NODE_NAME}'
      }
    }
  }
}
```

◆ إعداد Agent :

1. Dashboard → Manage Jenkins → Nodes → New Node
2. اكتب الاسم : my-agent
3. اختار Permanent Agent
4. في خانة Labels اكتب : my-agent
5. كمل الإعدادات

(الـRemote root directory ، والـ connection type سواء SSH أو JNLP)

🧩 Example 3

◆ هنا Jenkins هيشغل الـ job جوه Docker container فيه Node.js

◆ كمان عامل volume mount علشان يحتفظ بالـ npm cache

📊 النتيجة:

- Jenkins سحب image node:latest من Docker Hub
- شغّل container مؤقت.
- نفذ الـ stage جوه الـ container
- بعد انتهاء build ← حذف الـ container

💬 التفسير:

كده كل job بتشتغل في بيئة نظيفة فيها Node جاهز، وده يمنع أي تضارب بين المشاريع.

```
Terminal
$ cat Jenkinsfile
pipeline {
  agent {
    docker {
      image 'node:latest'  // Use a Docker image with Node.js
      args '-v $HOME/.npm:/root/.npm'  // Optional: Mount volume
    }
  }
  stages {
    stage('Build') {
      steps {
        ........
```

Example 4 🧩

- 🔹 هنا الـ pipeline كله بيستخدم agent اسمه my-agent،
- 🔹 لكن الـ stage بتاعة Build فقط بتستخدم agent تاني اسمه nodejs-agent
- 🔹 يعني تقدر تخصص agent مختلف لكل stage حسب احتياجك.

📊 النتيجة:

- Stage "Prepare" هتشتغل على my-agent
- Stage "Build" هتشتغل على agent تاني اسمه nodejs-agent

💬 الفكرة:

ده بيخليك تخصص Stage معينة ببيئة مختلفة (زي بيئة فيها Node أو Python)

```
Terminal
$ cat Jenkinsfile
pipeline {
  agent {
    label 'my-agent'  // Run on agent with label "my-agent"
  }
  stages {
    stage('Build') {
      agent { label 'nodejs-agent'}
      steps {
        sh 'echo Running on ${NODE_NAME}'
      }
    }
  }
```

🎯 الخلاصة:

النوع	المميزات	العيوب	الاستخدام
Permanent	ثابت، جاهز دايمًا	بيستهلك موارد	بيئات ثابتة ومشاريع دائمة
Docker	Clean وisolated	محتاج setup للـ images	Builds متعددة وisolated
Cloud	Scalable جدًا	يعتمد على الإنترنت والتكلفة	مشاريع كبيرة ومتغيرة
Label-based	مرن جدًا	محتاج تنظيم للـ labels	لما يكون عندك أنواع مختلفة من Agents

Create and Configure Node

💡 المقدمة:

- ❑ الشرح ده بيتكلم عن إزاي نربط Jenkins Controller بواحد أو أكتر من Jenkins Agents (Nodes) علشان نوزّع تنفيذ الـ Jobs بدل ما كلها تتنفذ على نفس المكنة.
- ❑ ده بيدي مرونة أكتر في الـ CI/CD Pipelines وبيخلي كل Agent متخصص في نوع معين من الـ builds (مثلاً Node.js أو Java أو Docker)

🧩 الجزء الأول: فكرة الـ Distributed Builds

- Jenkins ممكن يشتغل بطريقتين:
  1. Standalone Mode : كل حاجة (الـ Controller + الـ Build Execution) على نفس السيرفر.
  2. Distributed Mode : عندك Controller بيدير، و Agents/Nodes بينفذوا الـ Jobs
- الهدف: توسيع القدرة على التنفيذ + توزيع الحمل + فصل البيئات (مثلاً كل Node يكون مجهز بلغات وأدوات مختلفة)

⚙️ الجزء الثاني: إنشاء Agent جديد

➢ أنشأنا VM جديدة باسم: Ubuntu-Docker-JDK17-Node20
وهي اللي هتشتغل كـ Agent

🪜 الخطوات داخل Jenkins Dashboard :

1. روح إلى: Manage Jenkins → Nodes and Clouds → New Node
2. ادّيها اسم (مثلًا): Ubuntu-Agent
3. اختار النوع: Permanent Agent
4. هتفتح صفحة إعدادات الـ Node الجديدة.

🧠 الجزء الثالث: إعداد الـ Node بالتفصيل

1️⃣ عدد الـ Executors

- ده بيحدد كام Build ممكن يتنفذ في نفس الوقت على نفس الـ Node
- مثال:
  - o لو عندك 2 Executors ← ممكن تشغل 2 Build في نفس الوقت.
  - o لو 1 Executor ← Jenkins هيستنى أول Build يخلص قبل ما يبدأ التاني.

✅ Best Practice : خليه = عدد الـ CPU cores للمكنة.

2️⃣ Root Directory

- ده المكان اللي Jenkins بيخزن فيه كل حاجة على الـ Node (Logs, Artifacts, Configs...)
- مثلاً : /home/jenkins-agent
  - o أو ممكن تسيبه /home لو التجارب بسيطة.

✅ نصيحة : دايمًا اختار Partition فيه مساحة كافية للـ builds

3️⃣ Label

- الـ Labels بتستخدم لتجميع Agents تحت تصنيف معين.
  مثلًا : Label = Ubuntu-Docker-JDK17-Node20
- بعد كده في Jenkinsfile تقدر تقول: agent { label ' Ubuntu-Docker-JDK17-Node20' }
- في الحالة دي Jenkins هيختار أي Agent عنده نفس الـ Label ده.

✅ Use Case : لما يكون عندك أكتر من Agent بخصائص مختلفة (Linux / Windows / Java / Node.js)

4️⃣ Usage

- Jenkins بيسألك: "إزاي تستخدم الـ Node ؟"
  - Use this node as much as possible : بالتالي Jenkins يقدر يستخدمه لأي Job
  - Only build jobs with label expressions : استخدمه بس للـ Jobs اللي الـ label بتاعها مطابق.

✅ نصيحة : دايمًا اربط كل Node بـ Label واضح علشان مايحصلش تعارض في استخدام الموارد.

5️⃣ Launch Method

- نختار: Launch agent by connecting it to the controller
  - يعني الـ Agent هو اللي هيتصل بالـ Controller (Inbound connection)

✅ نصيحة : دي الطريقة الآمنة الشائعة لأنها بتقلل مخاطر فتح بورتات من السيرفر الرئيسي.

6️⃣ Availability

اختار: Keep this agent online as much as possible
علشان يفضل متاح دايمًا.

7️⃣ Node Properties

➢ هنا ممكن تضيف:
- Environment Variables
- Disk space monitoring
- Tool locations

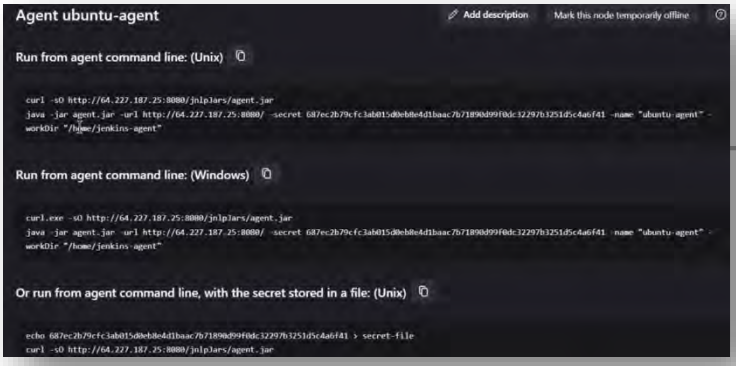
➢ في الفيديو، فعّل بس: Monitor disk space threshold
- وده مفيد علشان Jenkins يعرف لو الـ Node قرب يتملي Disk Space

---

💻 الجزء الرابع : تشغيل الـ Agent وربطه فعليًا

➢ بعد ما تضغط Save، هتلاقي إن الـ Node باين كـ "Offline" لأنه لسا مش connected
➢ ولو ضغطت عليه ← Jenkins هيطلعلك أوامر لتشغيله يدويًا .. ولاااحظ اختلاف الأوامر حسب نظام التشغيل:
- Unix/Linux
- Windows

وفيه خيارات بـ secrets أو بدونها .



الخطوات داخل الـ Node (VM) :

1️⃣ ادخل على السيرفر: ssh ubuntu@<node_ip>

2️⃣ روح على الـ home directory : cd /home

3️⃣ نزّل ملف الـ agent jar : curl -sO http://<controller-ip>:8080/jnlpJars/agent.jar

4️⃣ شغّل الـ agent باستخدام Java :

- java -jar agent.jar -jnlpUrl http://<controller-ip>:8080/computer/Ubuntu-Agent/jenkins-agent.jnlp -secret <secret-key> -workDir "/home"

🔍 ملاحظات مهمة:

- لازم يكون الـ JDK مثبت بنفس الإصدار الموجود على الـ Controller
- الـ secret key بيتولد تلقائي من Jenkins لما تعمل Add Node
- استخدم نفس Port اللي Jenkins بيستخدمه (عادة 8080 أو TCP خاص للـ Agents)

⚠️ الجزء الخامس: حل مشكلة Port 404 Not Found

➢ لما شغّل الأمر، طلع Error : Port 404 not found - TCP slave agent listener not enabled

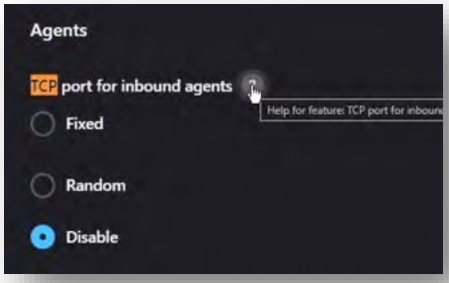
➢ ده معناه إن Jenkins قافل الاتصال عبر TCP لأسباب أمنية.

الحل:

1. ارجع لـ Dashboard
2. افتح : Manage Jenkins → Security → TCP port for inbound agents
3. بدل ما هو Disabled، اختار: Fixed 50000 OR Random
4. اضغط Apply and Save
5. رجع تاني على الـ Node وشغّل نفس الأمر تاني.

هتلاقي في اللوج : INFO: Connected

🎯 يعني الـ Agent اشتغل واتربط بنجاح .

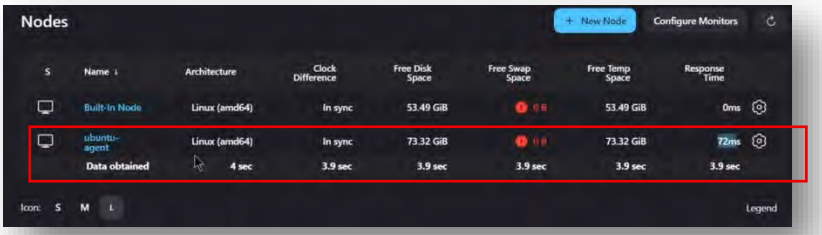


🔍 الجزء السادس : التأكد من الاتصال

في Dashboard → Nodes

هتلاقي:

- الحالة Connected ✅
- الوقت متزامن (Clock in sync)
- Response time ≈ 72 ms (ده طبيعي لأنها VM منفصلة)



🧠 الجزء السابع: التحقق من التفاصيل

من صفحة الـ Node تقدر تشوف:

- Load Statistics
- System Info
- Agent Logs
- Environment Variables
- Config History
- Monitoring Reports

وكل تعديل في إعدادات الـ Node بيتسجّل عشان تقدر ترجع لأي إصدار قديم.

🗂️ الجزء الأخير: التحقق من ملفات الـ Node

- على السيرفر:
  - cd /home/
  - ls
  - cd Jenkins-agent/
- هتلاقي مجلد : /remoting
  - وده بيتخزن فيه ملفات الاتصال.
- بعد ما تشغل أول Build باستخدام الـ Node ده، هتبدأ تظهر مجلدات جديدة جوه /home لكل Job

---

📋 الملخص النهائي

المرحلة	الفكرة	الأوامر / الأدوات
إنشاء Node	تعريف Agent جديد داخل Jenkins	Manage Jenkins → Nodes → New Node
الإعداد	Executors, Root dir, Labels	واجهة Jenkins GUI
الاتصال	تشغيل agent.jar	`... java -jar agent.jar`
حل المشكلة	تمكين TCP inbound port	Manage Jenkins → Security
التحقق	Logs + Sync + Latency	Dashboard → Node info
المخرجات	مجلد remoting + Build dirs	داخل `home/` في الـ Agent

🎯 النتيجة النهائية:

- الـ Node بقى متصل بـ Controller ، جاهز ينفذ Stages معينة من الـ Pipelines ،
- وده بيدي مرونة كبيرة جدًا في إدارة الـ CI/CD environment على مستوى الإنتاج الحقيقي.

Utilize Agents in Jobs

🎯 الفكرة الأساسية

- ❑ ده هو الجزء العملي المكمل للي شرحناه قبل كده،
- ❑ يعني بعد ما أنشأنا الـ Jenkins Agent (Node) وربطناه بالـ Controller ، دلوقتي هنشوف إزاي نستفيد منه فعليًا داخل الـ Jobs سواء كانت Freestyle أو Pipeline
- ❑ يعني ببساطة الجزء ده بيشرح إزاي تشغل الـ Jobs (Freestyle و Pipeline) على Agent خارجي بدل ما تشتغل على السيرفر الرئيسي (Controller)
- ❑ وده بيفيدنا لما يكون عندنا:
  - أكثر من بيئة (مثلاً Windows, Ubuntu, CentOS)
  - أو لما نحب نفصل الـ Build عن الـ Controller علشان نحافظ على الأداء والأمان.

🧩 الجزء الأول : استخدام الـ Agent في Freestyle Job

1 التأكد من إن الـ Agent شغال

هايرجع أولًا للـ Jenkins Dashboard وبيشوف إن الـ Agent اللي عملناه (مثلاً : Ubuntu-Agent)

حالته online ✅ ← وده معناه إنه جاهز لاستقبال Jobs

---

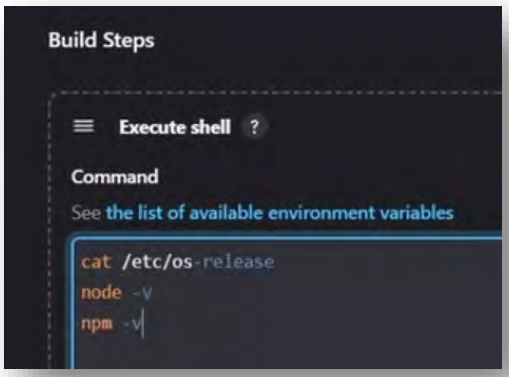
2 إنشاء Job جديدة من نوع Freestyle

من الـ Dashboard :

- نضغط على New Item
- نختار اسم مثلاً : freestyle-external-agent
- ونختار النوع Freestyle Project .. ثم نضغط OK

---

3 إضافة Build Step



في إعدادات الـ Job :

- ننزل على قسم Build Steps
- نختار Execute shell
- ونكتب أوامر بسيطة علشان نعرف نتاكد إن الـ Job فعلاً اتنفذت على الـ Agent ده:

💡 أوامر دي هتطبع :

- إصدار نظام التشغيل (Linux version)
- إصدار Node.js
- إصدار npm

وده هيساعدنا نعرف الفرق بين الجهاز الرئيسي (Controller) والـ Agent

---

4 ربط الـ Job بالـ Agent

- ➢ علشان نخلّي الـ Job تشتغل على الـ Agent ده بالذات، نروح لقسم Restrict where this project can be run ✅ ونفعّله . (هتلاقيه في الـ : General ← configuration ← job)
  - هايظهرلك خانة اسمها Label Expression، هنكتب فيها الـ Label اللي كنا محددينه للـ Agent لما أنشأناه (مثلاً : Ubuntu-Docker-JDK17-Node20)
- ➢ الفكرة هنا إن Jenkins بيشوف: (هل فيه Node عنده نفس الـ Label ده ؟) .. لو آه ← يشغّل الـ Job عليه.

5 حفظ وتشغيل الـ Job

- نضغط Apply ← Save
- وبعدين Build Now

هتبدأ الـ Job تشتغل، ولو فتحت الـ console OutPut هتلاقي مكتوب: Building remotely on Ubuntu-Agent

وده معناه إن Jenkins فعلاً شغّلها على الـ Agent الخارجي مش على السيرفر الرئيسي 👏

---

6 التحقق من النتيجة

في نفس اللوج:

هتلاقي:

- /etc/os-release بيعرض تفاصيل نظام تشغيل Ubuntu 22.04 مثلًا :
  - node -v بيعرض Node.js 22

وده دليل فعلي إن التنفيذ حصل على الجهاز الخارجي مش الـ Controller

---

🚀 الجزء الثاني : تشغيل Job على Agent داخل Pipeline

هنا بننقل نفس الفكرة لكن بطريقة الـ Pipelines

---

1 إنشاء Pipeline جديدة

- نرجع للـ Dashboard
- نضغط New Item → Pipeline
- نكتب اسمها : pipeline-external-agent
- ونضغط OK

---

2 تحديد مصدر السكريبت (SCM)

بدل ما نكتب الـ Pipeline Script يدوي داخل Jenkins ، هنخليه يقرأه من GitHub Repository

- نختار Pipeline script from SCM
- نحدد نوع الـ Repository : Git
- ونحط لينك الريبو اللي فيه ملف Jenkinsfile

jenkins-kk-demo/exploring-agents

---

3 محتوى ملف Jenkinsfile

الملف ده فيه مرحلتين (Stages)

25 lines | 378 B | Groovy

```
pipeline {
  agent any

  stages {
    stage('S1-Any Agent') {
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
      }
    }

    stage('S2-Ubuntu Agent') {
      agent {
          label 'ubuntu-docker-jdk17-node20'
      }
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
      }
    }
```

🧠 الشرح:

- في أول السطر عندنا agent any
  - يعني بشكل عام، أي Stage ممكن تشتغل على أي Agent متاح.
- لكن في Stage 2 حددنا

  agent { label ' Ubuntu-Docker-JDK17-Node20 ' }
  - يعني المرحلة دي لازم تشتغل على Agent معين فقط.

4️⃣ تشغيل الـ Job وملاحظة التنفيذ

بعد ما نحط لينك الريبو ونضغط Build Now :

- Jenkins هيعمل أولًا Checkout للـ Repository
- بعد كده يشغّل الـ Stage 1 ← على الـ Controller
- ثم Stage 2 ← على الـ Ubuntu Agent

5️⃣ التأكد من النتيجة

من اللوج:

- Stage 1 هتشوف إنها بتستخدم Workspace في:
  - /var/lib/jenkins/workspace/external-agent-pipeline-job
  - وبتعرض Node.js version = 20
- Stage 2 هتشوف إنها بتستخدم Workspace في:
  - /home/jenkins/workspace/external-agent-pipeline-job
  - وبتعرض Node.js version = 22

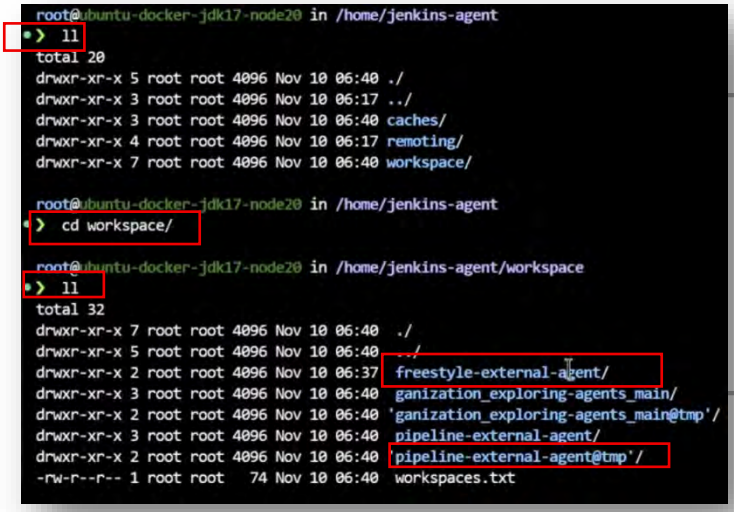
يعني فعلاً كل Stage اتنفذت على جهاز مختلف 💪

6️⃣ فتح Blue Ocean UI

المدرب فتح Blue Ocean علشان يعرض الشكل الرسومي:

- مرحلتين (S1 و S2)
- كل واحدة بعلامة Agent مختلفة
- النتيجة Successful ✅

7️⃣ التحقق من الـ Workspace على الـ Agent



وده دليل إن الـ Jobs فعلاً بتتنفذ من Jenkins Controller على الـ Agent الخارجي.

🧠 الخلاصة

- Jenkins يسمحلك توزّع المهام على أكتر من Agent
- ممكن تستخدم نفس الـ Agent سواء في Freestyle أو Pipeline Jobs
- باستخدام الـ Labels ، تقدر تتحكم مين يشتغل فين.
- كل Agent ليه workspace منفصل وبيئة مختلفة.
- ده بيساعد في:
  - الأداء
  - الأمان
  - الفصل بين البيئات

Utilize Docker Image Agent

🎯 الفكرة الأساسية

- ❑ ده الجزء اللي بيتكلم عن تشغيل Jenkins Pipeline stages داخل Docker Containers
- ❑ وده من أهم وأقوى أجزاء الـ DevOps لأنك هنا بتدمج بين Jenkins + Docker مع بعض 🔥
- ❑ بمعني انه بيشرح إزاي تخلي Jenkins يشغّل جزء من الـ Pipeline داخل Docker Container بدل ما يشغله على الـ Agent نفسه.
- ❑ بمعنى تاني:
- ❑ بدل ما Jenkins يشغّل الكود على جهاز Ubuntu Agent مباشرة، هو هيعمل Container مؤقت من Docker Image معينة، ويشغّل فيها الخطوات (Steps) المطلوبة، وبعد ما يخلص... يقفل ويشيل الـ Container.

🧩 الجزء الأول: الفكرة النظرية

➢ 💡 لماذا نستخدم Docker داخل Jenkins ؟
لأن Docker بيسمحلك تشغّل كل Stage في بيئة مستقلة ومعزولة.

➢ وده مفيد جدًا لما:

- تحب تستخدم بيئات مختلفة (Node.js ، Python، Java)
- مش حابب تبوّظ إعدادات النظام الأساسي
- أو عايز تبني Image مخصصة للـ Builds

🧰 المتطلبات الأساسية

علشان Jenkins يقدر يستخدم Docker في الـ Pipelines :

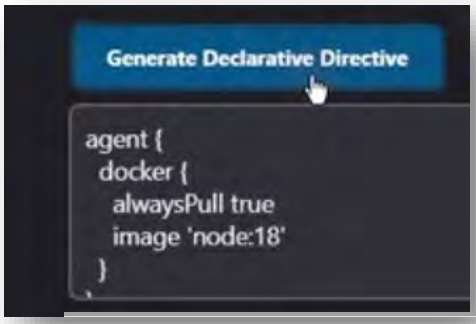
1. لازم يكون عندك Docker Pipeline Plugin (وده غالبًا متسطب بالفعل في Jenkins)
2. لازم الـ Agent اللي هيشغّل الـ Job .. يكون عليه Docker engine متسطب ومشتغل.

⚙️ الجزء الثاني: إعداد الـ Agent باستخدام Docker

➢ علشان يولد الكود .. هايروح علي :

- o pipeline syntax → Declarative Directive Generator → Sample Directive → agent: Agent
- o Agent → docker: Run inside a Docker Container
- o Image → node:18 (بتدخل unique name للـ image)
- o label ← لو عايز تختار على أي Node يتنفذ
- o registryUrl و credentialsId ← لو الصورة خاصة ومحتاجة login
- o Always Pull Image ✅ (علشان كل مرة يسحب أحدث نسخة من الصورة)

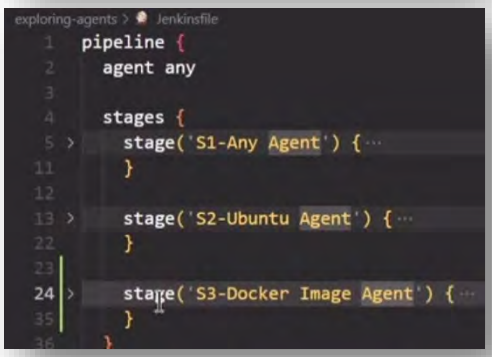
وده معناه:
شغّل المرحلة دي داخل Container معمول من الصورة node:18-alpine



مثال عملي

jenkins-kk-demo/exploring-agents

➢ استخدم نفس الـ Pipeline السابق (exploring-agents)
اللي كان فيه مرحلتين + ضاف عليهم مرحلة جديدة S3 باسم:
stage('S3 Docker image agent')



```
Jenkinsfile
exploring-agents > Jenkinsfile
pipeline {
  agent any

  stages {
    stage('S1-Any Agent') {…
    }

    stage('S2-Ubuntu Agent') {…
    }

    stage('S3-Docker Image Agent') {
      agent {
        docker {
          image 'node:18-alpine'
          label 'ubuntu-docker-jdk17-node20'
        }
      }
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
      }
    }
  }
}
```

تعريف المرحلة الجديدة داخل Jenkinsfile

شرح السطور :

1 stage('S3 Docker image agent')

- يعني دي مرحلة (Stage) اسمها S3 ودي مرحلة جديدة في الـ Pipeline

2 agent { docker { image 'node:18-alpine' ... } }

- ده معناه:
  - شغّل المرحلة دي جوه Docker Container معمول من الصورة "node:18-alpine
- الصورة دي (Image) فيها:
  - نظام تشغيل بسيط اسمه Alpine
  - ومعاه Node.js الإصدار 18
- يعني Jenkins هيشغّل الخطوات دي جوه Container فيه Node جاهز ✅

3 label 'ubuntu-docker-jdk17-node20'

- يعني المرحلة دي تشتغل على Agent اللي الـ label بتاعه اسمه 'ubuntu-docker-jdk17-node20'
  (يعني السيرفر اللي عليه Docker متسطب)
  - ولاااااحظ اني لو مكنتش كتبتله الـ label .. فبسبب agent any اللي في أول الـ pipeline ← كان هايشتغل علي الـ Jenkins controller (حيث ان دا هوا لمستخدم في الـ agent any)

4 steps { ... }

➢ الخطوات اللي جوه الـ stage :

sh 'cat /etc/os-release'

- ده أمر بيطبع نوع نظام التشغيل.
- هيطلع له إن النظام اسمه Alpine Linux (مش Ubuntu)
  يعني فعلاً الشغل جوه الـ Container مش على السيرفر.

sh 'node -v'`

- بيطبع نسخة Node.js (وهي 18 x.x.جايه من الـ Image)

sh 'npm -v'`

- بيطبع نسخة npm اللي جايه مع Node.

🚀 الجزء الثالث: التشغيل ومتابعة النتيجة

1 Jenkins يبدأ التنفيذ

أول حاجة بيعملها Jenkins لما توصل للـ Stage S3 :

- بيشوف هل Image node:18-alpine موجودة؟
  - لو لأ ← يعمل docker pull node: 18-alpine
- بعد كده ← ينفذ docker run وينشئ Container مؤقت
- بعدين ينفذ الـ Steps جوه الـ Container ده
  (زي node -v , cat /etc/os-release)

2 النتيجة في اللوج

```
Pulling docker image node:18-alpine ...
Using docker image node:18-alpine ...
[Pipeline] sh
+ cat /etc/os-release
NAME="Alpine Linux"
```

- في الـ Console Output هتلاقي:
- وده دليل إن التنفيذ فعلاً جوه Container مبني على Alpine Linux مش Ubuntu زي الـ Agent نفسه
- وكمان:

node -v → v18.x.x

npm -v → 9.x.x

فكل حاجة اتنفذت جوه الـ Container بنجاح ✅

---

3 ملاحظات مهمة

- بعد ما الـ Stage تخلص ← Jenkins بيعمل stop للـ Container ويمسحه تلقائيًا.
  والدليل انه لما جرب يعمل: docker ps ملقاش أي Containers شغالة.
- لكن لما كتب: docker images

لقى:

node 18-alpine created 4 months ago لأن الـ Image نفسها لسه موجودة.

🧠 نظرة أعمق: اللي بيحصل فعليًا

Jenkins لما يوصل لمرحلة Docker بيعمل:

1. يسحب الـ Image من Docker Hub : docker pull node:18-alpine
2. يشغل منها Container مؤقت : docker run -d -v /workspace:/workspace node:18-alpine
   - ينشئ Container مؤقت ويربطه بـ Workspace
3. ينسخ الملفات المطلوبة داخل الـ Container
4. ينفذ الأوامر (cat, node -v, npm -v)
5. لما الـ Stage تخلص ← يعمل docker stop و docker rm للـ Container ويشيله نهائيا علشان ميستهلكش مساحة

الخلاصة

العنصر	التوضيح
الهدف	تشغيل Pipeline Stage داخل Docker Container
الأداة	Docker Pipeline Plugin
مثال	`agent { docker { image 'node:18-alpine' } }`
نظام التشغيل	Alpine Linux (من داخل الصورة)
النتيجة	Jenkins بيعمل Pull + Run + Execute + Remove
بعد التنفيذ	الـ Container بيتم مسحه تلقائيًا
الفائدة	بيئة معزولة وقابلة للتكرار لأي Stage

Utilize Dockerfile Agent

🎯 أولاً: الفكرة العامة

- ❑ الجزء ده بيشرح إزاي Jenkins يستخدم Dockerfile علشان يبني Image مخصصة وفيها كل الحاجات اللي محتاجها عشان يشغل مرحلة معينة (Stage) في الـ Pipeline
- ❑ بمعنى تاني:

بدل ما تستخدم Docker Image جاهزة (زي node:18-alpine) وتكتشف إنها ناقصة Tools أو Libraries ،

اعمل Image مخصصة بنفسك عن طريق Dockerfile 👇

🧱 ثانياً: البداية – إنشاء Stage جديدة

- راح على الـ Jenkinsfile وعمل Stage جديدة اسمها : stage('S4 Dockerfile agent') {
  - دي المرحلة رقم 4، وهي اللي هيجرب فيها Dockerfile Agent

⚙️ ثالثاً: الـ Agent Configuration

- في الأول، استخدم نفس إعدادات المرحلة اللي قبلها:

```
Jenkinsfile
exploring-agents > Jenkinsfile
agent any

stages {
  stage('S1-Any Agent') {…
  }

  stage('S2-Ubuntu Agent') {…
  }

  stage('S3-Docker Image Agent') {…
  }

  stage('S4-Dockerfile Agent') {
    agent {
      docker {
        image 'node:18-alpine'
        label 'ubuntu-docker-jdk17-node20'
      }
    }
    steps {
      sh 'node -v'
      sh 'npm -v'
      sh 'cowsay -f dragon This is running on Docker Container'
    }
  }
```

يعني:

- هيشغل المرحلة على Agent اسمه ubuntu-agent
- وهيستخدم Image جاهزة اسمها node:18

➢ لكن المختلف هنا مش في الـ Agent ... ولللكن

الاختلاف في الـ Steps (الأوامر اللي هتنفذ جوه الـ Container) 👇

📋 رابعاً: الـ Steps اللي جوه الـ Stage

- أول أمرين هيطبعوا نسخ Node و npm (الموجودين أصلاً في الـ Image)
- لكن الثالث (cawsay) دي Tool اسمها cawsay CLI ووظيفتها انها بتطبع رسومات ASCII من نصوص، يعني حاجات شكلها جمالي مش أساسي

❌ خامساً: أول مشكلة – الـ Build فشل

- الـ Jenkins شغل الـ Stage .. وظهر إن Node و npm شغالين تمام ✅ .. لكن cawsay فشل ❌
- السبب واضح:

الـ node:18 Docker image مافيهاش الـ cawsay library

يعني الـ Container مش عارف يعني إيه الأمر cawsay

💡 سادساً: الحل – نعمل Image مخصصة

➢ في حالات كتير هتحتاج تعمل Image فيها كل الـ Tools المطلوبة لخطواتك.

➢ فقرر يبني Dockerfile جديد بنفسه.

- وده هيكون Base على node:18-alpine .. لكن يضيف عليه حاجات تانية زي git, perl , cawsay CLI

سابعاً: إنشاء الـ Dockerfile

- عمل ملف جديد جوا نفس الـ Pipeline (exploring-agents) .. وسماه : **Dockerfile.cawsay**
  - ممكن تسميه أي اسم، مش لازم Dockerfile بالضبط، طالما هتشيرله صح في Jenkinsfile
- وجواه كتب:

```
Jenkinsfile    Dockerfile.cowsay U
exploring-agents > Dockerfile.cowsay
FROM node:18-alpine

RUN apk update && \
    apk add --no-cache git perl && \
    cd /tmp && \
    git clone https://github.com/jasonm23/cowsay.git  && \
    cd cowsay ; ./install.sh /usr/local
```

- شرح الكود :

السطر	الشرح
`FROM node:18-alpine`	استخدم Image فيها Node 18 كنقطة بداية.
`apk update`	حدّث الـ repositories (عشان يعرف يحّمل البرامج).
`apk add git perl`	نزّل أدوات مطلوبة لبناء **cawsay**
`... git clone`	نزّل كود الـ **cawsay** من GitHub.
`make install`	نزّل الأداة جوه الـ Image نفسها.

النتيجة = Image جديدة فيها Node + npm + Kause ✅

ثامناً: تعديل Jenkinsfile لاستخدام Dockerfile

- pipeline syntax → Declarative Directive Generator → Sample Directive → agent: Agent
  - Agent → dockerfile: Build a Dockerfile and run in a container using that image
  - Filename: Dockerfile.cawsay

- لاحظ الفرق:
  - بدل { ... } docker { image
  - استخدم { ... } dockerfile { filename
- وده بيقول لـ Jenkins :

"ابني الـ Image من Dockerfile ده الأول، وبعدها شغّل المرحلة عليه".

```
stage('S4-Dockerfile Agent') {
  agent {
    docker {
      image 'node:18-alpine'
      label 'ubuntu-docker-jdk17-node20'
    }
```

غيرها إلي

```
stage('S4-Dockerfile Agent') {
  agent {
    dockerfile {
      filename 'Dockerfile.cowsay'
      label 'ubuntu-docker-jdk17-node20'
    }
```

🚀 تاسعاً : الـ Build نجح

المرة التانية Jenkins نفذ الخطوات التالية:

1. قرأ Dockerfile من الـ workspace
2. بدأ يبني Image جديدة : `docker build -t jenkins-agent-xyz -f Dockerfile.k8s .`
3. خلص البناء وطلّع Image ID جديدة.
4. شغّل Container من الـ Image دي.
5. نفّذ أوامر Node و npm و Kause ✅

النتيجة:

- Node 18 شغال.
- npm شغال.
- cawsay كمان اشتغل بنجاح 💪

🧠 الحتة المهمة هنا

- إنك تقدر تبني Docker Image خاصة بيك فيها كل الأدوات المطلوبة لأي Stage في Jenkins
- وده بيديلك مرونة رهيبة، لأن:
  - مش لازم تثبّت tools على كل Agent
  - كل Stage تشتغل في بيئة نظيفة.
  - الـ Builds تبقى متكررة ومضمونة (reproducible)

✅ خلاصة الدرس:

النقطة	الشرح
🎯 الهدف	استخدام Dockerfile مخصص في Jenkins stage
🧱 الـ Base Image	`node:18-alpine`
🧰 الإضافات	git + perl + Kause CLI
🧠 الميزة	كل حاجة تبنيها حسب احتياجك بدقة
⚙️ Jenkins يعمل إيه	يبني Image → يشغل Container → ينفذ الأوامر
💪 النتيجة	بيئة مخصصة، نظيفة، فيها كل الأدوات المطلوبة

Utilize NewContainerPerStage

المقدمة – الفكرة العامة

- الشرح ده بيتكلم عن Jenkins Declarative Pipeline، وبالتحديد عن الفرق بين تشغيل كل المراحل (Stages) في نفس الـ container مقابل تشغيل كل مرحلة في container مختلف باستخدام الـ directive اللي اسمه
  👉 newContainerPerStage
- الفكرة ببساطة إنك لما تستخدم Jenkins مع Docker agent ، بيتم إنشاء container لتشغيل البايبلاين.
- افتراضيًا، كل الـ stages بتشتغل جوه نفس الـ container .. لكن أحيانًا في بيئات العمل (real-world pipelines) ، ممكن تحتاج كل stage تشتغل في container جديد ونظيف .. وهنا بيجي دور newContainerPerStage

```
pipeline {
  agent {
    dockerfile {
      filename 'Dockerfile.cowsay'
    }
  }

  stages {
    stage('Stage-1') {
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
        sh 'echo $((RANDOM)) > /tmp/imp-file-$BUILD_ID'
        sh 'ls -ltr /tmp/imp-file-$BUILD_ID'
        sh 'cat /tmp/imp-file-$BUILD_ID'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
      }
    }
    stage('Stage-2') {
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
        sh 'ls -ltr /tmp/imp-file-$BUILD_ID'
        sh 'cat /tmp/imp-file-$BUILD_ID'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
      }
    }
    stage('Stage-3') {
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
        sh 'ls -ltr /tmp/imp-file-$BUILD_ID'
        sh 'cat /tmp/imp-file-$BUILD_ID'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
      }
    }
    stage('Stage-4') {
      steps {
        sh 'node -v'
        sh 'npm -v'
        sh 'cowsay -f dragon This is running on Docker Container'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
        sh 'ls -ltr /tmp/imp-file-$BUILD_ID'
        sh 'cat /tmp/imp-file-$BUILD_ID'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
        sh 'sleep 60s'
    }
  }
}
```

For Demo Purpose

⚙️ الشرح التفصيلي خطوة بخطوة

🧩 المرحلة 1 : الوضع العادي (بدون newContainerPerStage)

1. الـ pipeline عنده global agent من نوع Dockerfile
   يعني Jenkins بيستخدم Dockerfile واحد يبني منه (image) ويشغل منها container واحد.
2. كل الـ 4 stages بيشتغلوا داخل نفس الـ container بالتالي :
   - كلهم بيستخدموا نفس الـ workspace
   - أي ملفات بتتولد في stage 1 تفضل موجودة في stage 2, 3, 4 + بتبقي متشافة من كل الـ Stages

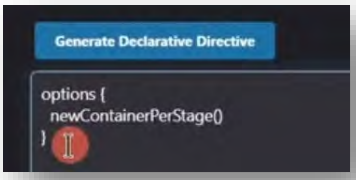
🎯 التجربة :

➢ في stage 1 اتعمل ملف في /tmp اسمه مثلًا:
tmp/imp-file-${BUILD_ID}/

- واتكتب جواه رقم عشوائي (random number)
- بعدين باقي الـ stages قرأوا نفس الملف، وفعلاً ظهر نفس الرقم في كل المراحل.
- ده دليل إنهم بيشتغلوا في نفس الـ container ونفس الـ filesystem

الاوامر اللي استخدمه للتأكيد من فكرة التجربة السابقة :

- **docker ps -a** ← علشان يشوف الـ containers اللي شغالة
  (وهتلاحظ انهم running in Jenkins controller ودا طبيعي لأننا عاملين Global Agent)
- **docker exec -it <container> bash** ←
  بيدخله على الـ container أثناء التنفيذ .. وبعدين نفذ جواه أمر **cat /tmp/imp/file** فطلعله نفس الرقم .. (وبكدا التجربة ناجحة وبالدليل)
- **docker stop / docker rm** ←
  بيتنفذ تلقائي بعد نهاية البايبلاين عشان يمسح الـ containers
- **docker build -t <image-id>** ← يبني image جديدة.
- **docker run** ← ينشئ container جديد.



المرحلة 2 : إضافة الـ directive newContainerPerStage

- بعد ما تأكد إن كل المراحل بيستخدموا نفس الـ container ، أضاف السطر ده في الـ pipeline :
- لاااحظ انه جابه من :

pipeline syntax → Declarative Directive Generator → Sample Directive → option: Options then Add → newContainerPerStage

```
pipeline {
  agent {
    dockerfile {
      filename 'Dockerfile.cowsay'
    }
  }

  options {
    newContainerPerStage()
  }

  stages {
    stage('Stage-1') {
      steps {
        sh 'cat /etc/os-release'
        sh 'node -v'
        sh 'npm -v'
        echo "#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
        sh 'echo $((RANDOM)) > /tmp/imp-file-$B
```

📌 المعنى:

كل stage هيشتغل في container جديد خاص بيه، مبني من نفس Dockerfile ، لكن معزول تمامًا عن الباقيين.

🎯 النتيجة :

- Stage 1 أنشأ ملف في /tmp وكتب رقم جواه.
- لما Stage 2 حاول يقرأ الملف ← فشل لأن الملف مش موجود في الـ container التاني. ❌
- Jenkins logs أظهرت إن كل stage عمل docker run جديد ← يعني فعلاً كل واحدة في container مستقل.

💡 ملاحظات عملية (Best Practices + Real Use Cases + Tricks)

🔍 Best Practices :

- استخدم newContainerPerStage() فقط لما فعلاً تحتاج عزل تام بين المراحل.
  مثلاً:
  - مرحلة بتبني التطبيق.
  - مرحلة تانية بتعمل deploy
  - مرحلة ثالثة بتختبر التطبيق.

  كل واحدة عايزة بيئة جديدة مفيهاش مخلفات المرحلة اللي قبلها.
- لو المراحل متتابعة وبتشارك ملفات (artifacts) ، فالأفضل تخليهم في نفس container لتوفير الوقت والمساحة.
- لو استخدمت newContainerPerStage()، خليك عارف إن Jenkins هيعمل build و run جديد في كل stage ← ده معناه زمن أطول واستهلاك موارد أكبر.

⚙️ Real Use Case :

في الشركات الكبيرة اللي بتطبق CI/CD على microservices :

- مرحلة الـ *Build* بتستخدم container فيه tools زي Maven أو npm
- مرحلة الـ *Test* بتحتاج بيئة نظيفة لتشغيل unit tests
- مرحلة الـ *Deploy* بتحتاج container فيه kubectl أو helm

يبقى منطقي إنك تستخدم newContainerPerStage() لتفصل كل مرحلة ببيئة مخصصة.

🧠 Tricks :

- تقدر تستخدم shared volume بين الـ containers لو عايز تحتفظ بملفات معينة رغم newContainerPerStage
- لو عندك stages بتحتاج نفس الـ dependencies ، استخدم pre-built Docker image بدل ما Jenkins يبنيها كل مرة.
- دايمًا راقب الـ logs في Classic UI عشان تشوف مراحل build/run لكل container ← دي بتساعد جدًا في الـ debugging

---

🧩 الملخص

الحالة	عدد الـ Containers	الملفات بين الـ stages	الأداء	أفضل استخدام
بدون `newContainerPerStage`	1	مشتركة ✅	أسرع ⚡	مراحل تعتمد على نفس الملفات
مع `newContainerPerStage`	لكل stage واحد مستقل	معزولة ❌	أبطأ ⏱️	مراحل تحتاج بيئات منفصلة

NewContainerPerStage In More Details

1. مقدمة — الفكرة العامة (إيه المشكلة وإيه الحل)

- ❑ في Jenkins بنقدر ننفذ كل Stage داخل حاوية Docker بدل ما ننفذها على نفس الـ node أو الـ controller
- ❑ ده بيفيد لأنك بتحصل على بيئة متسقة (كل مرة نفس الأدوات/الإصدارات) ومنع تداخل الاعتمادات بين المشاريع
- ❑ عندك طريقتين رئيسيتين:
  - **agent { docker { image '...' } }** ← تشغّل الخطوات في حاوية مبنية من صورة موجودة مسبقًا.
  - **agent { dockerfile { filename 'Dockerfile.X' } }** ← Jenkins بيبني صورة من Dockerfile الموجود في الـ workspace ثم يشغّل الحاوية من الصورة دي.
- ❑ سلوك مهم : لو عندك agent على مستوى أعلى (global) واستخدمت Dockerfile ، فبشكل افتراضي:
  - Jenkins هايبني صورة واحدة ويستخدم نفس الحاوية لجميع المراحل داخل الـ pipeline
  - لكن لو حابب كل Stage يتنفّذ في حاوية جديدة مع بيئة نظيفة ← تستخدم الخيار:
    - **options { newContainerPerStage() }**
    - وده بيخلّي كل Stage ينطلق في container منفصل.

---

2. شرح كل جزئية خطوة-بخطوة مع الأوامر والأدوات

2.1 المتطلبات الأساسية (Environment)

أدوات لازم تكون جاهزة:

- Jenkins مُثبت مع Docker Pipeline Plugin
- Node/Agent اللي هيشغّل الحاويات لازم يكون مثبت عليه Docker daemon
- صلاحيات للوصول إلى Docker socket (/var/run/docker.sock) أو يكون الـ agent شغّال كـ Docker-enabled user

أوامر تحقق : **docker –version** AND **docker info**

- لو عايز تشوف الحاويات أو الصور: **docker images** AND **docker ps -a**

2.2 كتابة Dockerfile (لو محتاج أدوات خاصة)

لو الصورة الرسمية ملهاش الأدوات اللي محتاجها (مثلاً kubectl, trivy, cawsay) .. ابني صورة مخصصة:

مثال Dockerfile.cawsay :

```
FROM node:18-alpine

# تثبيت أدوات أساسية
RUN apk add --no-cache bash curl git

# تثبيت kubectl (كمثال)
RUN curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl" \
 && install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

# لو تحب شغل كـ non-root
USER node
WORKDIR /home/node/app
```

➢ أوامر لبناء/اختبار محلي:

- بناء الصورة محليًا : **docker build -t myimage:cawsay -f Dockerfile.cawsay .**
- تشغيل حاوية تفاعلية للتحقق : **docker run --rm -it myimage:cawsay sh**

2.3 Jenkinsfile : استخدام agent { dockerfile { ... } } على مستوى global

مثال Jenkinsfile :

```
pipeline {
  agent {
    dockerfile {
      filename 'Dockerfile.k8s'
      label 'ubuntu-docker-node'   // يُشير إلى agent اللي عليه Docker
      args '--pull --rm'           // اختياري
    }
  }

  stages {
    stage('Stage 1 - Build') {
      steps {
        sh 'node --version'
      }
    }
    stage('Stage 2 - Test') {
      steps {
        sh 'kubectl version --client'
      }
    }
  }
}
```

- ايه اللي بيحصل وايه الأوامر اللي Jenkins ينفذها تلقائيًا:
  1. Jenkins يبني الصورة:

     docker build -t jenkins-temp-image -f Dockerfile.cawsay .
  2. Jenkins ينشئ حاوية واحدة من الصورة دي ويشغّل كل المراحل داخلها:
     - **docker run --name jenkins-temp-container -v <workspace>:/workspace ... jenkins-temp-image sh -c '<stage steps>'**
  3. بعد الانتهاء، Jenkins يوقف ويحذف الحاوية.

2.4 طاب لو عايز كل Stage في container جديد؟ هاتستخدم newContainerPerStage()

لو ضفت:

```
options {
  newContainerPerStage()
}
```

- كدا الـ Jenkins هيشغّل كل stage في حاوية جديدة
- النتيجة: لا مشاركة ملفية بين المراحل عبر ملف نظام الحاوية.

أمثلة سريعة:

- بدون newContainerPerStage ← Stage1 يولّد ملف /tmp/ .. و Stage2 يقرأ /tmp/ بنجاح.
- مع newContainerPerStage ← Stage2 مش هيلاقي /tmp/ إلا لو استخدمت stash/unstash أو artifact

---

2.5 مشاركة الملفات بين مراحل مع newContainerPerStage

- لو عايز تنقل ملفات بين مراحل مع newContainerPerStage استخدم stash / unstash :

مثال:

```
stage('Create') {
  steps {
    sh 'echo hello > build-id.txt'
    stash includes: 'build-id.txt', name: 'id'
  }
}
stage('Use') {
  steps {
    unstash 'id'
    sh 'cat build-id.txt'
  }
}
```

---

3 ملاحظات عملية مهمّة Best Practices / Real Use Cases / Tricks

Best Practices

1. صغّر أحجام الصور : استخدم alpine أو اجمع الأوامر في RUN واحد لتقليل الطبقات.
2. شغّل كـ non-root داخل الصورة (استخدم USER) خصوصًا بالـ production
3. ضع الأدوات في Dockerfile بدل تثبيتها أثناء pipeline (أسرع & مضمون)
4. استخدم label مناسب على الـ agent علشان تتأكد إن Docker متاح على الماكينة الصحيحة.
5. تجنّب الوصول المباشر لـ /var/run/docker.sock إن أمكن)

Real Use Cases

- CI لبناء صور Docker : استخدم {} dockerfile لبناء الصورة المطلوبة وتشغيل اختبارات داخل نفس الصورة.
- اختبارات متوازية ومعزولة : لكل pipeline تفعيل newContainerPerStage() لبيئة نظيفة لكل مرحلة اختبار.
- أدوات خاصة (kubectl/trivy/helm) : ضفها في Dockerfile بدل تركيبها وقت التشغيل.

Tricks

- لو بناء Dockerfile بطيء : ادفع الصورة إلى registry محلي بعد أول بنية واستخدمها لاحقًا
  بـ agent { docker { image 'registry/myimage:tag' } }
- لو محتاج تشغّل أو تدخل للحاوية وقت التنفيذ : ضع sleep مؤقت في آخر stage وادخل
  بـ docker exec -it <id> /bin/sh (فقط في dev)
- استخدم --pull في args علشان تضمن أحدث نسخة من الـ base image

---

4 Troubleshooting مشاكل شائعة وحلولها

- Docker : command not found ← الـ agent ليس لديه Docker ؛ ثبت Docker أو غيّر الـ label
- permission denied to /var/run/docker.sock ← عدّل صلاحيات أو شغّل الـ agent بيوزر يملك صلاحية الوصول.
- أدوات ناقصة داخل الصورة (مثلاً kubectl) ← عدل Dockerfile وأضف تثبيت الأداة)
- ملفات مفقودة بين مراحل مع newContainerPerStage ← استخدم stash/unstash أو artifact server

---

5 مثال عملي متكامل (سلسلة أوامر منطقية)

1. أنشئ Dockerfile.cawsay (مثل المثال أعلاه)
2. تحقق محليًا:
   - docker build -t myimage:cawsay -f Dockerfile.cawsay .
   - docker run --rm -it myimage:cawsay sh -c 'node --version && kubectl version --client'
3. ارفع الملفات إلى repo
4. Jenkinsfile (باستخدام stash لنقل ملفات بين مراحل لو فعلت newContainerPerStage)

```
pipeline {
  agent {
    dockerfile {
      filename 'Dockerfile.k8s'
      label 'ubuntu-docker-node'
    }
  }
  options { newContainerPerStage() } // أو شيله لو عايز نفس الحاوية
  stages {
    stage('Make file') {
      steps {
        sh 'echo $(date +%s) > build-id.txt'
        stash includes: 'build-id.txt', name: 'id'
      }
    }
    stage('Consume file') {
      steps {
        unstash 'id'
        sh 'cat build-id.txt'
      }
    }
  }
}
```

---

6 الملخص الشامل — ربط كل الأفكار

- تشغيل مراحل Jenkins داخل Docker يوفّر بيئة متناسقة
- agent { docker { image... } } يستخدم image جاهز؛
  agent { dockerfile { filename... } } يبني image من Dockerfile
- بدون newContainerPerStage()، Jenkins يبني الصورة ويشغّل حاوية واحدة لجميع المراحل ← المراحل تتشارك الـ filesystem
- مع newContainerPerStage()، كل Stage يُشغّل في حاوية جديدة ← عزلة تامة ولكن يفقد مشاركة الملفات بين المراحل ما لم تستخدم stash/unstash
- Best practice : ابني صورة صغيرة، شغّل كـ non-root ، استعمل stash للملفات بين الحاويات، واحفظ صورك في registry لو البناء مكلف زمنيًا.

Configure Cloud Instance-K8S

🧠 المقدمة – الفكرة العامة

- ❑ الجزء ده بيشرح إزاي نقدر نخلّي Jenkins يستخدم Kubernetes كـ Cloud Agent Provider
- ❑ الفكرة ببساطة:

بدل ما Jenkins يخصّص VM أو Docker محلي لتشغيل الـ builds ، نقدر نستخدم Kubernetes Pods كـ Agents مؤقتين (ephemeral agents)

- ❑ كل Job بتتشغّل ← Jenkins يطلب من Kubernetes ينشئ Pod مخصوص ← يشتغل عليه ← وبعدها يتحذف.

وده بيدّي:

- مرونة عالية (scalable builds)
- عزل تام لكل Job
- تكامل طبيعي مع البيئات الـ cloud-native

⚙️ الشرح خطوة بخطوة بالتفصيل

Kubernetes | Jenkins plugin

1. 💙 تثبيت الـ Plugin الخاص بـ Kubernetes

الخطوات:

- ➢ من واجهة Jenkins ← نروح لـ Manage Jenkins ← Clouds
  - o Jenkins هيقول مفيش أي Cloud plugins متسطبة.
  - o اضغط على Install a Plugin
  - o هتتفتح صفحة الـ Available Plugins
  - o دور على Kubernetes Plugin
  - o اختار نسخة قديمة (4.2.9.5) بدل الأحدث، علشان النسخة الأخيرة فيها تغييرات غير مستقرة.
  - o نزلها يدويًا، وركّبها من (.hpi) Advanced → Upload plugin file which you downloaded
  - o اعمل restart للـ Jenkins (ضروري)

💡 Trick :

دايمًا في بيئة production ، استخدم stable plugin version، مش أحدث واحدة، لأن الجديدة ممكن تكون فيها breaking changes

---

2. 💙 ربط Jenkins بـ Kubernetes Cluster لأول مرة

الخطوات:

1. بعد تثبيت الـ plugin ← ارجع إلى Manage Jenkins ← Clouds
2. هتلاقي Cloud جديدة اسمها Kubernetes
3. اضغط على ➕ لإنشاء Cloud جديدة.
4. سمّيها مثلاً : Dasher-prod-K8s-us-east
5. Jenkins هيسألك عن:
   - o Kubernetes URL
   - o Namespace
   - o طريقة الاتصال (manual أو kubeconfig)

الطريقة الأولى kubeconfig file :

- استخدم ملف الـ kubeconfig من الكلاستر ← kubectl config view --raw
- نسخ محتواه كلله وحفظه في file.yaml عالجهاز عادي
- بعدين راح يضيفه داخل Jenkins Credentials عن طريق :

o Credentials → add → Domain: Global Credentials (unrestricted) → Kind: secret file → Scope: Global → File: “choose file.yaml” → ID: kube-config-us-east → Description: same as ID → ADD

- بعدين راح يكمل ربط Jenkins بالـ K8S cluster وفي خانة الـ credentials ← هايضيف الـ credential اللي لسا عاملينه حيث هايستخدمه لاختبار الاتصال.

تحذيير:

- o خلي بالك ان الـ credential دي فيها صلاحيات admin كاملة (access للـ cluster كله) وده غير آمن، لأن Jenkins المفروض يستخدم صلاحيات محدودة.
- o بس هو عمل كدا لغرض الـ Demo فقط .. لكن لا ينصح بالـ admin privilege في الـ credential عموما والافضل تخليك دايما عالـ At Least Privilege

---

الطريقة الثانية : إنشاء Namespace مخصصة لـ Jenkins

الخطوات:

- على Kubernetes : kubectl create namespace jenkins
- بعدها بدأ يربط Jenkins بالـ namespace دي فقط بدل الـ default
- بس لما حاول يربط Jenkins باستخدام:
  - o Kubernetes URL (endpoint) (اللي موجودة في ملف الـ kubeconfig file)
  - o Namespace = Jenkins (اللي لسا عاملينها)
  - o بدون kubeconfig

النتيجة:

- o فشل الاتصال برسالة : PKIX path building failed
- o يعني الشهادة (SSL certificate) مش معروفة .

الحل المؤقت:

- o Disable HTTPS certificate check ✅
- o أو استخدم الشهادة الفعلية للـ API Server (بتحطها في مربع الـ Kubernetes server certificate key)

النتيجة:

- o بردو فشل الإتصال .. لكن بسبب مختلف وهو 403 Forbidden
- o ودا معناه ان الـ user مش authorized to retrieve any information from Jenkins namespace

الحل (مؤقت بردو) :

- o هاتحتاج تعمل service account + تديلها الـ permissions المناسبة

⚙️ إنشاء Service Account لـ Jenkins داخل Namespace

- الهدف : Jenkins يربط بالكلاستر بصلاحيات محدودة.
- الخطوات:

1. إنشاء Service Account :
   - kubectl -n jenkins create serviceaccount jenkins-service-account
   - kubectl -n Jenkins get secret ↓

(لاحظ انك مش هتلاقي secrets .. يعني مبتتولدش by default ولازم انت اللي تولدها يدويا أو تولد token وخلاص)

2. إنشاء Token مرتبط بيه:
   - kubectl -n jenkins create token jenkins-service-account --duration=9999999s
   - ده بيولّد JWT token فهاننسخه ونستخدمه في Jenkins كـ Secret Text Credential
     (يعني هانولد credential جديدة وننسخه فيها .. وتحديدا في مربع الـ Secret)
   - وهايكون صالح فقط لمدة 115 days (9999999s)
   - وخلي بالك ان الـ token دا خاص بالـ namespace دي فقط .. ولو غيرت الـ namespace فلازم تعمل token جديد
3. أضف الكريدينشال في Jenkins :
   - Kind : Secret Text
   - ID: k8s-jenkins-sa-token
4. جرّب الاتصال من Jenkins بالـ token

📦 النتيجة:

- فشل الاتصال بسبب ان لسه فيه 403 Forbidden ← لأن مفيش صلاحيات (Role Binding)

🧠 الحل النهائي :

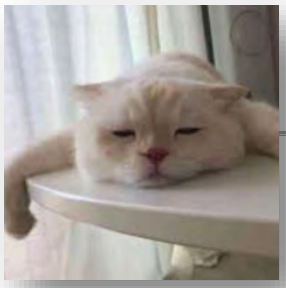
تعمل Role Binding للـ Service Account

---

5. 💙 إنشاء Role Binding للـ Service Account

- هانستخدم الـ Admin Role ولكننا هانقتصره فقط علي الـ Jenkins service account اللي في الـ Jenkins namespace
- الخطوات:

1. استخدم أمر:
   - kubectl -n jenkins create rolebinding jenkins-admin-binding \
     --clusterrole=admin --serviceaccount=jenkins:jenkins-service-account
2. كده الـ service account بقى ليه صلاحيات admin داخل Jenkins namespace دي فقط.
3. ارجع Jenkins ← اضغط Test Connection← نجحت. ✅



---

6. 💙 ضبط إعدادات الاتصال (Connection Settings)

الخيارات المهمة:

الخيار	المعنى	الملاحظة
Jenkins URL	الرابط اللي الـ Pods هتستخدمه للاتصال بالـ controller	لو TCP ports مقفولة، استخدم WebSockets
Connection Type	TCP أو WebSocket	WebSocket مفيد في بيئات مقفولة
Pod Labels	Labels بتتضاف تلقائيًا للـ Pods	مثلاً: `organization=KodeKloudAzureArc`
Pod Retention	إمتى الـ Pod يتحذف بعد الـ build	`Never` / `OnFailure` / `Always`

📦 أفضل إعدادات في بيئة Production :

- Pod Retention = OnFailure
- Connection = WebSocket (لو environment مقفولة)
- استخدم Label ثابت علشان تقدر تتابع الـ Pods بسهولة.

💡 ملاحظات عملية (Best Practices & Real Use Cases)

1 🔹 Least Privilege Access

- متديش Jenkins صلاحيات admin على الكلاستر كله.
- استخدم Namespace مخصصة + Service Account محدودة.

2 🔹 Token Lifetime

- لو بتولّد token يدوي، راقب مدة صلاحيته (--duration) أو استخدم automated rotation

3 🔹 Certificates

- يفضل دايمًا استخدام شهادة معتمدة بدل ما تلغي HTTPS checks

4 🔹 Debugging Builds

- خلي الـ Pod retention على "OnFailure" علشان تقدر تفتح الـ Pod وتشوف logs وقت الفشل.

5 🔹 Scaling Jenkins Agents

- الطريقة دي بتخلي Jenkins يـ auto-scale agents على حسب عدد الـ jobs بدون ما تضيف VMs جديدة.

📄 الملخص الشامل

المرحلة	الهدف	الأوامر / الأدوات
تثبيت Plugin	تمكين Jenkins من دعم Kubernetes	واجهة Jenkins GUI
إعداد kubeconfig	اختبار الاتصال السريع	`kubectl config view --raw`
إنشاء Namespace	تخصيص بيئة آمنة لـ Jenkins	`kubectl create namespace jenkins`
إنشاء ServiceAccount	تعريف هوية Jenkins على الكلاستر	`kubectl create serviceaccount`
إنشاء Token	Authentication بين Jenkins و K8s	`kubectl create token`
RoleBinding	إعطاء صلاحيات محدودة	`kubectl create rolebinding`
Test Connection	تأكيد الربط بين Jenkins و K8s	Jenkins GUI

✅ الخلاصة:

- Jenkins لما يتكامل مع Kubernetes ، بيقدر يشغّل كل Build في Pod منفصل داخل الكلاستر.

ده بيديلك:

- 💪 Scalability
- 🔒 Isolation
- وإدارة موارد ذكية 🌐

وفي الدرس الجاي هانبدأ نستخدم الـ Kubernetes Cloud فعليًا لتشغيل الـ Jobs داخل الـ Pods

Utilize K8S Pod As Agent

المقدمة:

- الجزء دا بيتكلم عن تشغيل Jenkins build agents داخل Kubernetes pods بدل ما تستخدم الـ master node أو static agents
- الفكرة إن Jenkins هيبعت الـ jobs كـ containers جوا Kubernetes cluster ، وكل build هيشتغل في pod مؤقتة (ephemeral) تتخلق وقت التنفيذ وتتقفل بعده.
- الموضوع دا مفيد جدًا لما تكون عاوز بيئة clean لكل build ، وبتتعامل مع scalable infrastructure بدون ما تضيف manual nodes

---

الشرح التفصيلي خطوة بخطوة:

الخطوة 1: إنشاء الـ pipeline job

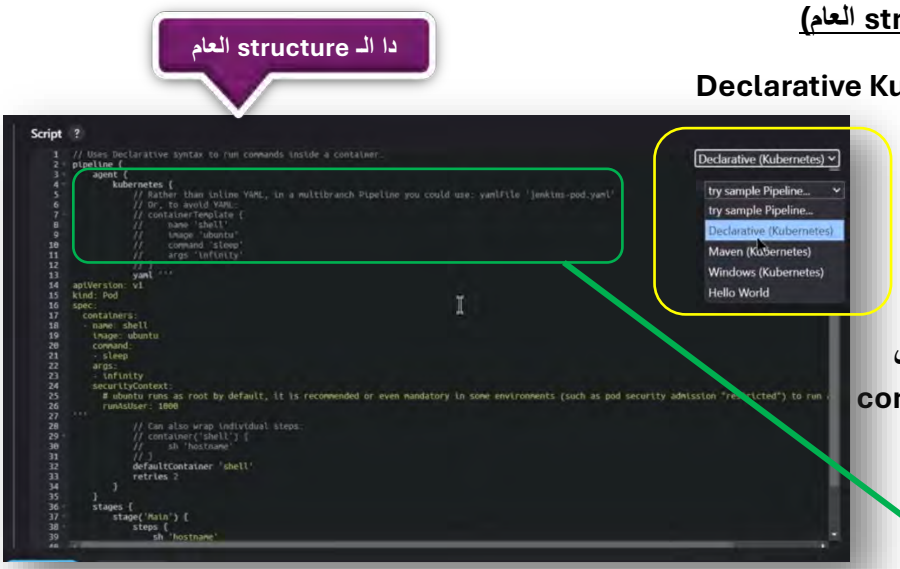
- بيروح على Dashboard → New Item → Pipeline
- بيسميها مثلًا : k8s-cloud-agent-demo
- يختار النوع Pipeline

الغرض : هي الـ job اللي هنستخدم فيها Kubernetes كـ agent لتشغيل الـ build steps

---

الخطوة 2: كتابة الـ Jenkinsfile (الـ structure العام)

- بيستخدم Try Sample وبيختار Declarative Kubernetes
- الكود بينتج structure فيه:




```
agent {
    kubernetes {}
```

معناها إن كل الـ stages اللي جوه الـ pipeline هتشتغل جوا Pod على الـ K8s cluster اللي اتعملها configured Jenkins Cloud

الفكرة:

فيه (3) طرق ممكن تحدد بيها شكل الـ Pod اللي Jenkins هيستخدمه:

1. Inline YAML داخل Jenkinsfile *دا المستخدم في الشرح*
   - يعني تكتب تعريف الـ Pod جوه الـ pipeline نفسه زي ماهو موضح في الصورة السابقة
2. استخدام ملف YAML خارجي (في مشروع multibranch) *هاتتشرح قدام في صفحة 368*
   - يعني تحط ملف YAML اسمه مثلاً podTemplate.yaml جوه الـ repo، و Jenkins يقرأ منه شكل الـ pod.
3. تعريف Pod template في إعدادات Jenkins Cloud Configuration *فكرته هاتتشرح قدام في صفحة 399 *
   - تروح من الواجهة: Manage Jenkins → Configure Clouds → Kubernetes
   - وتضيف هناك template ثابت يستخدمه Jenkins لما يحتاج Pod.

Kubernetes | Jenkins plugin

Script ?

```
// Uses Declarative syntax to run commands inside a container.
pipeline {
    agent {
        kubernetes {}
    }
    stages {
        stage('Print Hostname') {
            steps {
                sh 'hostname'
                sh 'sleep 120s'
            }
        }
    }
}
```

◆ الخطوة 3: تعديل الـ stages في الـ structure اللي ولدناه :

- بيعمل stage واحدة اسمها print-hostname
- جوهها بيشغل :
- hostname
- sleep 120

💡 الهدف :

يطبع اسم الـ pod (اللي هو نفسه الـ hostname) وبعدين
يعمل sleep لمدة 120 ثانية علشان يقدر يشوف الـ pod وهي شغالة في Kubernetes قبل ما تتقفل.

Script ?

```
// Uses Declarative syntax to run commands inside a container.
pipeline {
    agent {
        kubernetes {
            // Rather than inline YAML, in a multibranch Pipeline you could use: yamlFile 'jenkins-pod.yaml'
            yaml '''
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: ubuntu-container
    image: ubuntu
    command:
    - sleep
    args:
    - infinity
'''
            // Can also wrap individual steps:
            // container('shell') {
            //     sh 'hostname'
            // }
            defaultContainer 'ubuntu-container'
            retries 2
        }
    }
    stages {
        stage('Print Hostname') {
            steps {
                sh 'hostname'
                sh 'sleep 120s'
            }
        }
    }
}
```

◆ الخطوة 4: إعداد الـ Kubernetes Agent

في Jenkinsfile بيعرّف الـ YAML الخاص بالـ Pod inline كالتالي:

💬 شرح:

- بيعمل Pod فيها container واحد اسمه ubuntu-container
- Image: Ubuntu
- Sleep: infinitly (هايفضل alive)
  - وممكن بدالها تستخدم cat & tty: true ودول بيخلوا الـ container يفضل شغال بردو (هايستخدمها في المثال الجاي)

➢ 💡 السؤال بقا هو ازاي فعليا هاتستخدم الـ pod ده علشان تعمل run للـ stages ؟؟

- هانستخدم حاجة اسمها default container (خلي بالك من الكتابة في الـ yamlfile انها علي نفس مستوي الـ agent)
- وبعدين هاتديها اسم الـ container اللي انت هاتستخدمه .. وبالتالي كل الـ stages هايتعملها run في الـ container ده

➢ وفيه كمان اوبشن :

- انك لو مش عاوز تستخدم الـ default container ده
- فممكن تستخدم container command في كل step عادي (مشروح في المثال الثاني)

🧠 Best Practice :

يفضل دايمًا تحط الـ container في وضع "alive" أثناء الاختبار علشان Jenkins يقدر يشغّل الأوامر جوه.

◆ الخطوة 5: تشغيل الـ pipeline ومتابعة النتائج

بعد ما يشغّل الـ build :

- Jenkins بيبعت YAML لـ Kubernetes
- Kubernetes بيخلق Pod جديد (تقدر تشوفه بـ kubectl -n jenkins get pods)
- Pod دا هيحتوي على 2 containers :
  - ubuntu-container (اللي انت عرفته)
  - JNLP (اللي Jenkins بيستخدمه للتواصل مع الـmaster)

🔍 أوامر بتستخدمها:

- kubectl get pods
- kubectl -n jenkins describe pod <container-name>

📘 Real Case Insight :

كل مرة Jenkins job بتشتغل، بيتخلق Pod جديد، ينفّذ الخطوات، وبعدين يتم حذفه (حسب الـ retention policy)
وده مفيد جدًا في البيئات CI/CD اللي فيها builds كتير وبتحتاج scalability

◆ الخطوة 6 : ملاحظة مهمة (Pod Retention)

➢ في الكود كان عامل : **podRetention: never**

- يعني الـ pod هتتحذف تلقائيًا بعد ما الـ job يخلص.
- علشان كده استخدم **sleep 120** بس علشان يشوف الـ pod قبل ما تتقفل.

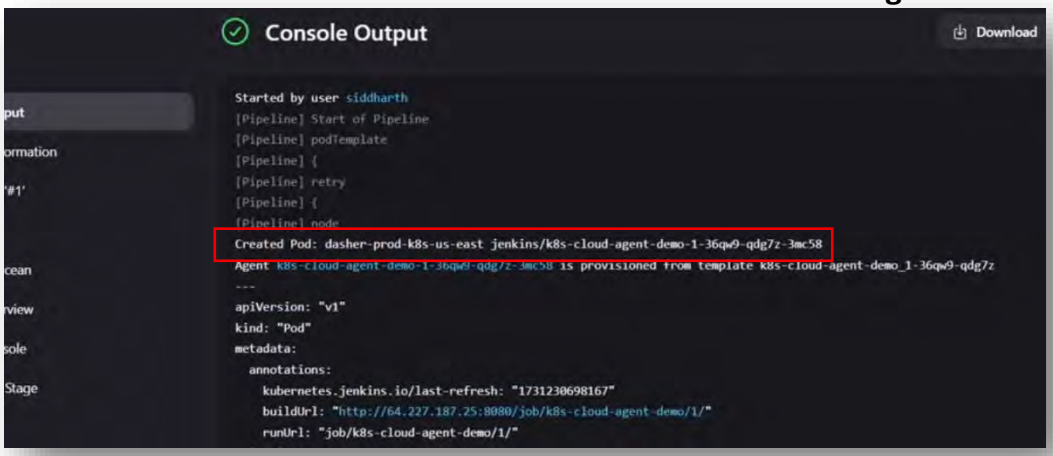
📌 **Trick :**

❖ لو عايز تراجع pods القديمة، استخدم: **kubectl -n jenkins get events**

o ده بيجيبلك history عن الـ pods اللي اتخلقت واتقفلت.

◆ ملاااحظة مهمة جدا (How pod connected with Config file?!)

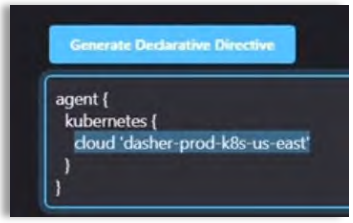
➢ لو شوفت الـ logs (من الـ Console output) بتاع المثال السابق .. هتلاقي ان by default الـ pod اتعملها create بناء علي الـ configuration اللي في الـ cloud configuration file



➢ 💡 زمانك هتسأل طاب هي الـ pod دي اصلا استخدمت الـ config file واتصلت بيه ازاي بالرغم من اننا ممنشنناش ولا حطينا اسم الـ config file في اي حتة في الـ yaml اللي عملناه في الـ jenkinsfile ؟!؟!؟!

o قالك انها اتصلت بيه implicity (يعني ضمنيا) والسبب هو انك لو روحت عالـ ↓

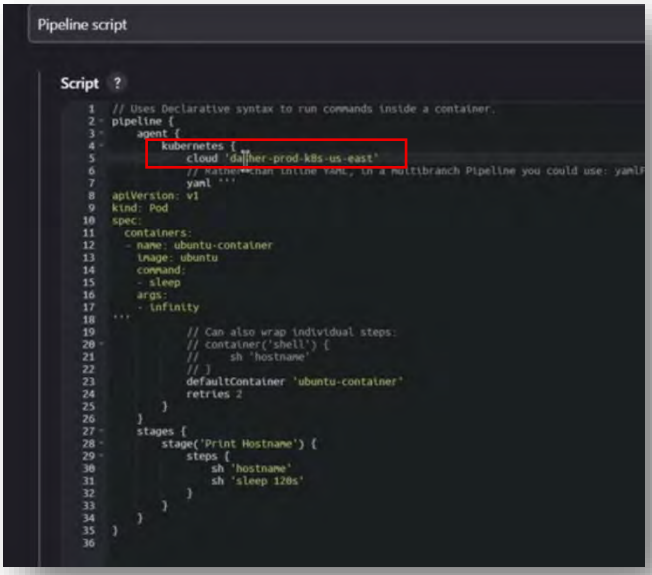
pipeline syntax → Declarative Directive Generator → sample Directive → agent: Agent → Kubernetes: Run inside a Kubernetes pod

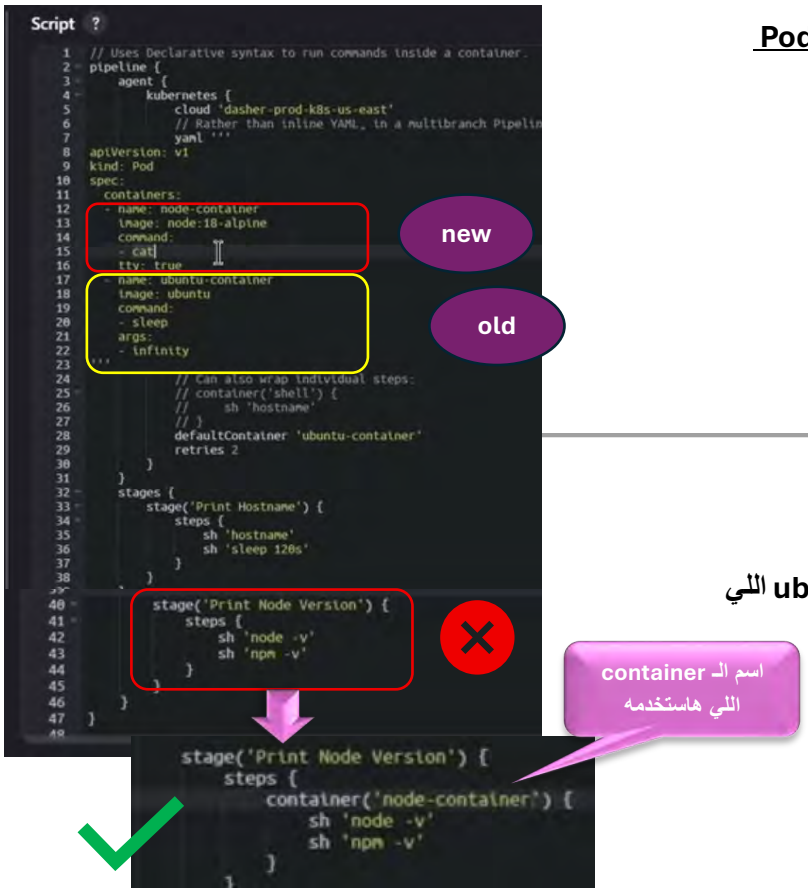


فهتلاحظ ان في خيار Cloud to use مختارين –any– ودي معناها انك لو معرفتلوش اي kubernetes cloud config عشان يستخدمه .... فهو (ضمنيا) هايروح يستخدم الـ first available kubernetes cloud config

o وبما اننا معندناش الا config file يتيم .. فهو استخدمه ضمنيا.

➢ هايرجع بقا عالسكريبت ويضيف الحتة دي عشان يبقا شغال explicity (يعني بوضوح) ودا هو الـ Best Practise





◆ الخطوة 7: إضافة Multiple Containers في نفس الـ Pod

الخطوة اللي بعد كده بيعمل تجربة جديدة:

- بيضيف container تاني :

🧠 الهدف:

يعني دلوقتي كل pod هيبقى فيها 3 containers :

1. ubuntu-container
2. node-container
3. JNLP (auto added by Jenkins)

◆ الخطوة 8: تشغيل Stage جديد في Container معين

➢ بيضيف stage تانية:

➢ لكنها هاتفشل لأن الـ default container هو ubuntu اللي مفيهوش Node.js

➢ 👨‍🔧 الحل :

- اننا هانستخدم اوبشن الـ container command

💡 Real Use Case :

دي طريقة ممتازة لو كل stage محتاج بيئة مختلفة (مثلاً واحدة فيها NodeJS ، التانية فيها Maven أو Python) وهكذا

◆ الخطوة 9 : تشغيل الـ build بعد التعديل

بعد التعديل، بيشغّل الـ pipeline تاني وبيلاحظ:

- أول stage (print-hostname) بيشتغل جوه ubuntu-container
- التاني (Print Node Version) بيشتغل جوه node-container

✅ النتيجة: كل حاجة شغالة تمام، وكل Container بيعمل دوره المحدد.

🧩 Best Practices & Tips :

1. خلي كل stage isolated جوه container مخصص.
2. استخدم sleep أو podRetention: always لما تكون بتجرب (عشان يفضل شغال)
3. فعل logging في Jenkins علشان تقدر تتتبع pod lifecycle
4. لو عندك plugins كتير بتحتاج tools مختلفة ← استخدم multi-container pod
5. استخدم : cloud: "kubernetes-name" في agent لتحديد cluster معين لو عندك أكتر من واحد.

📋 الملخص النهائي

❖ الجزء دا بيشرح إزاي Jenkins يقدر يستخدم Kubernetes كـ build agent :

- بيخلق Pod مؤقت (ephemeral pod) لكل build
- تقدر تعرّف الـ containers جوه الـ Jenkinsfile inline أو في Cloud config
- ممكن تشغل كل stages في container واحد أو تفصلها بين أكتر من container حسب احتياجك.
- Jenkins بيتواصل مع الـ pod عن طريق container خاص اسمه JNLP
- بعد انتهاء الـ job ، الـ pod بتتحذف تلقائيًا لو retention = never

🚀 وكده بقا عندك بيئة CI/CD مرنة جدًا، scalable، و isolated لكل build، بدون أي manual configuration على agents

Sharing File Between Containers

المقدمة

- الجزء ده بيتكلم عن مفهوم مشاركة الملفات بين الـ containers داخل نفس الـ Pod في Kubernetes ،
  وبيوضح ده من خلال مثال عملي باستخدام Jenkins pipeline
- الفكرة ببساطة إن:
  - لو عندك Pod فيها أكتر من Container ، كل الـ Containers دي بتشارك نفس الـ(volume) filesystem ،
    فبالتالي أي ملف بيتكتب في Container ممكن التاني يشوفه ويستخدمه.

وده بيكون مفيد جدًا لما يكون عندك stages مختلفة في Jenkins pipeline بتشتغل على نفس Pod

---

الشرح خطوة بخطوة

الخطوة 1 – الفكرة العامة للتجربة

هنا عمل Pipeline فيها مرحلتين (stages) :

1. Stage 1 : بيشغّل Ubuntu container
   - بيكتب ملف جديد اسمه ubuntu-build-id.txt فيه بعض البيانات .
   - بيعمل list للملفات علشان يتأكد إنه اتكتب.
2. Stage 2 : بيشغّل Container تاني (Node container)
   - بيحاول يقرأ نفس الملف اللي اتعمل في الـ Ubuntu container

والهدف من التجربة : يثبت إن الملف متاح بين الـ containers لأنهم جوه نفس الـ Pod

```
Script ?
agent {
    kubernetes {
        cloud 'dasher-prod-k8s-us-east'
        // Rather than inline YAML, in a multibr
        yaml '''
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: node-container
    image: node:18-alpine
    command:
    - cat
    tty: true
  - name: ubuntu-container
    image: ubuntu
    command:
    - sleep
    args:
    - infinity
'''
        // Can also wrap individual steps:
        // container('shell') {
        //     sh 'hostname'
        // }
        defaultContainer 'ubuntu-container'
        retries 2
    }
}
stages {
    stage('Print Hostname') {
        steps {
            sh 'hostname'
            echo "##################################################"
            sh 'echo important_UBUNTU_data  > ubuntu-$BUILD_ID.txt'
            sh 'ls -ltr ubuntu-$BUILD_ID.txt'
            echo "##################################################"
        //    sh 'sleep 60s'
        }
    }

    stage('Print Node Version') {
        steps {
            container('node-container') {
                sh 'node -v'
                sh 'npm -v'
                echo "##################################################"
                sh 'ls -ltr ubuntu-$BUILD_ID.txt'
                sh 'cat ubuntu-$BUILD_ID.txt'
                echo "##################################################"
            }
        }
    }
}
```

Same pod

1st container

2nd container

طلاما مفيش container commad تبقي شغالة علـ default container اللي هو ubuntu-container

الخطوة 2 – الكود المستخدم داخل Jenkins

ضاف الأوامر دي داخل الـ Jenkinsfile (كمثال مبسط) :

شرح الكود:

- أنشأنا Pod فيها 2 containers (Ubuntu + Node)
- كل Stage بيستخدم Container مختلف .
- الملف اللي اتكتب في الـ Ubuntu container بيتقرأ من الـ Node container لأنهم بيشاركوا نفس الـ volume

الخطوة 3 – التنفيذ (Build & Logs)

بعد ما شغّل الـ pipeline :

- في مرحلة Ubuntu ظهر في الـ log :
  - echo "This is a test file" > ubuntu-build-id.txt
  - ls

  ✅ الملف اتنشأ بنجاح.
- في مرحلة Node :
  - cat ubuntu-build-id.txt

  ✅ نفس الملف اتقري بنجاح، وده يثبت إنهم بيشاركوا نفس التخزين داخل الـ Pod

💡 Best Practices / Real Use Cases / Tricks

✅ Best Practices :

1. لو محتاج الـ containers تشارك بيانات، استخدم Shared Volumes (مثل emptyDir)
2. لو مش عايز مشاركة، استخدم Pods منفصلة لكل stage
3. دايمًا نظم أسماء الملفات والمسارات لتجنب الـ overwriting بين stages

🧰 Real Use Case :

في pipelines حقيقية ممكن تستخدم الطريقة دي لما:

- Container يعمل build أو compile للـ code
- Container تاني ياخد نفس الملفات ويعمل testing أو deployment

🪄 Trick :

ممكن تستخدم نفس المبدأ علشان تمرر ملفات credentials أو config بين مراحل متعددة بدون ما تستخدم external storage (زي S3 أو NFS)

📋 الملخص النهائي

العنصر	الوصف
🎯 الفكرة	Containers داخل نفس Pod بيشاركوا نفس filesystem.
🧱 التنفيذ	Stage 1 يكتب ملف في Container – Stage 2 يقرأه من Container تاني.
⚙️ الأداة	Jenkins pipeline مع Kubernetes Pod Template.
💡 الفايدة	مشاركة الملفات بسهولة بين خطوات الـ pipeline بدون external storage.
🧠 نصيحة عملية	استخدم shared volumes بحذر وراقب الـ permissions والـ cleanup بعد انتهاء الـ job.

Refactoring Solar System Pipeline

المقدمة

- الموضوع هنا بيتكلم عن:
  إزاي نربط Jenkins مع Kubernetes علشان نخلي الـ Pipelines تشتغل داخل Pods بدل ما تستهلك موارد الـ Jenkins نفسه.
- يعني بدل ما كل Stage من الـ pipeline تشتغل على الـ controller أو VM ثابتة،
  الـ Jenkins هيطلب من Kubernetes إنه ينشئ له Pod مؤقت فيه الـ containers اللي محتاجها (زي Node.js أو Python) بحيث ينفّذ فيها الـ stages دي ، وبعدها الـ Pod يتم مسحه تلقائيًا.
- هنا هو قرر يعمل تعديل (Refactor) على Pipeline اسمها Solar System Pipeline علشان:
  - يخلي بعض الـ stages تشتغل على Kubernetes agent
  - ويستخدم ملف YAML خارجي بدل ما يكتب تعريف الـ Pod جوه الكود وبعدين يشير إليه جوا الـ jenkinsfile

---

الشرح خطوة بخطوة

الخطوة 1 – تجهيز الـ YAML file الخاص بالـ Kubernetes Agent

- بدل ما يكتب تعريف الـ Pod داخل Jenkinsfile ، أنشأ ملف خارجي اسمه : K8s-agent.yaml
- وفيه تعريف بسيط للـ Pod فيه 2 Containers :
  - node-18 (نسخة Node.js 18)
  - node-19 (نسخة Node.js 19)
- مثال بسيط:

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: node-18
    image: node:18-alpine
    command:
    - cat
    tty: true
  - name: node-19
    image: node:19-alpine
    command:
    - cat
    tty: true
```

الفكرة هنا:
بدل ما نكرر الكود ده في Jenkinsfile ، نحفظه في ملف مستقل ونربطه داخل الـ pipeline

---

الخطوة 2 – تعديل الـ Jenkinsfile لاستخدام الـ YAML file الخاص بالـ Kubernetes Agent

- هايروح علي pipeline syntax وهايظبط الت configuration كالتالي :
  - Agent: Run inside a kubernetes pod
  - Cloud to use: dasher-prod-k8s-us-east
  - Default Container: node-18
  - وبعدين اضغط generate فهايولدلك الكود اللي هاتكتبه في الـ jenkinsfile

Generate Declarative Directive

```
agent {
  kubernetes {
    cloud 'dasher-prod-k8s-us-east'
    defaultContainer 'node-18'
  }
}
```

- بدل ما كان الكود بيبدأ كده : agent any
- بقى كده:

```
@Library('dasher-trusted-shared-library@featureTrivyScan') _

pipeline {
    agent {
        kubernetes {
            cloud 'dasher-prod-k8s-us-east'
            yamlFile 'k8s-agent.yaml'
            defaultContainer 'node-18'
        }
    }

    tools {...
    }

    environment {
        MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net
        MONGO_DB_CREDS = credentials('mongo-db-credentials')
        MONGO_USERNAME = credentials('mongo-db-username')
```

لاحظ إننا أضفنا هنا اسم الـ yamlfile اللي عملناه

📘 شرح الجزء ده:

- yamlFile : بيقول لـ Jenkins يجيب تعريف الـ Pod من الملف اللي على الـ repo
- defaultContainer : بيحدد الـ container الأساسي اللي الأوامر هتتنفذ جواه (هنا Node 18)
- الـ Jenkins هيروح ينشئ Pod جديد في الـ cluster بناءً على الملف ده.

🧱 الخطوة 3 – تحديد أي stages تشتغل على Kubernetes Agent

➢ في المشروع فيه stages كتير
(npm install, dependency scan, test, code coverage, build image, trivy scan...)

➢ قرر يخلي:

- Stages الخاصة بـ Node.js تشتغل على الـ Kubernetes agent
(زي npm install, dependency scan, unit testing, code coverage)
- والباقي (زي build docker image أو Trivy scan) تشتغل على الـ controller عادي.

➢ فالكود بقى بالشكل ده مثلًا:

```
stage('Build Docker Image') {
    agent any
    steps {
        sh 'printenv'
        sh 'docker build -t siddharth67/solar-system:$GIT_COMMIT .'
    }
}

stage('Trivy Vulnerability Scanner') {
    agent any
    steps {
        script {
            trivyScanScript.vulnerability(imageName:"siddharth67/solar-sy
```

لاحظ انه في الـ stages اللي هايشغلها عالـ controller .. أضاف فيها agent any طاب ليه وايه الفكرة اصلا ؟!

لأن انت محددله في الصورة السابقة (في أول الـ jenkinsfile) الـ agent (اللي هو الـ kubernetes agent يعني) وبالتالي دا هاينطبق علي كل الـ stages ... طاب افرض فيه stages انت عاوز تشغلها عالـ controller ؟؟
هاتقوم ضايف فيها الـ agent any زي ماهو عمل كدا

وبكدا تبقي كدا خليت الـ stages تشتغل علي agents مختلفة وفي نفس الـ jenkinsfile

---

🧰 الخطوة 4 – تشغيل الـ Pipeline ومراجعة Logs

➢ بعد ما عمل Commit & Push، الـ Jenkins أوتوماتيك بدأ يبني الـ pipeline الجديدة.

➢ في الـ console output ظهر التالي:

- Jenkins بيحمّل الملف K8s-agent.yaml
- بينشئ Pod جديد في الـ cluster
- بيعرض الـ spec الكامل للـ Pod (volumes, env vars, node selector...)
- لاحظ ان الـ volume المستخدم نوعه emptyDir ← بيتشارك بين كل الـ containers داخل الـ Pod

📦 النتيجة في التنفيذ:

- أول Stage (npm install) شغالة على Node 18
- نفس الـ container استخدم 4 مرات في stages مختلفة (install, scan, test, coverage)
- كلهم بيشوفوا نفس الـ dependencies اللي اتعملت install أول مرة.

✅ مفيش حاجة اتكررت أو اتهدرت لأنهم بيشاركوا نفس الـ volume

💡 Best Practices / Real Use Cases / Tricks

✅ Best Practices :

1. دايمًا افصل تعريف الـ Pod في ملف YAML مستقل (سهل الصيانة وإعادة الاستخدام).
2. استخدم shared emptyDir volumes لو محتاج الـ stages تشارك نفس الملفات.
3. استخدم labels على الـ Pods (زي node selectors) علشان تتحكم وين تتنفذ الـ jobs
4. لو الـ stages مش محتاجة تكون parallel ، حافظ على Pod واحد لتقليل الـ overhead

🧰 Real Use Case :

شركة كبيرة ممكن تعمل:

- Stage build & test داخل Kubernetes Pod
- وبعدها الـ deploy يشتغل على agent تاني مخصوص له access على production cluster

🪄 Trick :

لو عندك stages كتير بتستخدم نفس البيئة، استخدم defaultContainer بدل ما تعرّف كل مرة container جديد لأن ده بيوفر وقت وكود.

📃 الملخص النهائي

العنصر	الشرح
🎯 الهدف	تشغيل Jenkins stages داخل Kubernetes Pods بدل الـ static agents.
🧱 الخطوات	إنشاء YAML file → تعديل Jenkinsfile → تشغيل الـ pipeline.
⚙️ الأداة	Jenkins Kubernetes Plugin + yamlFile definition.
💾 الفايدة	تشغيل مؤقت، مشاركة بيانات بين الـ containers، وسرعة CI/CD.
💡 الممارسات الجيدة	فصل الـ pod spec في YAML مستقل، استخدام shared volume، تحديد container افتراضي.
🧠 النتيجة	كل stages الخاصة بـ Node.js استخدمت Pod واحد بنفس الـ environment بدون تكرار.

Refactoring Unit test Stage

المقدمة

- الجزء ده بيتكلم عن تشغيل Unit Tests على أكثر من نسخة Node.js داخل Jenkins pipeline
- الهدف هنا إننا نتحقق إن الكود بتاعنا شغال تمام على نسخ مختلفة من Node (18, 19, 20) ، وده بيساعد في الـ compatibility testing لأي تطبيق Node.js
- وبما إننا أصلاً مشغلين الـ Jenkins pipeline داخل Kubernetes Pod (فيه أكثر من container) فده معناه إننا نقدر نعمل stages متوازية (parallel stages) علشان نختبر كل نسخة Node.js في نفس الوقت.

الشرح خطوة بخطوة

الخطوة 1 – الـ Pipeline الأصلية

- الـ stage الأساسية قبل التعديل كانت بالشكل ده:

```
stage('Unit Test') {
  steps {
    sh 'node -v'
    sh 'npm test'
  }
}
```

- بتشتغل على Node 18 (الـ default container)
- فيها خطوات بسيطة : بتطبع الـ version ، وتشغل الـ tests

الخطوة 2 – إضافة Stages متوازية (Parallel Stages)

- هنا عدّل الكود علشان يعمل 3 stages متوازية:

```
stage('Unit Testing') {
    parallel {
        stage('NodeJS 18') {
            options { retry(2) }
            steps {
                sh 'node -v'
                sh 'npm test'
            }
        }
    stage('NodeJS 19') {
        options { retry(2) }
        steps {
            container('node-19') {
                sh 'sleep 10s'
                sh 'node -v'
                sh 'npm test'
            }
        }
    }
stage('NodeJS 20') {
    agent {
        docker {
            image 'node:20-alpine'
        }
    }
    options { retry(2) }
    steps {
        sh 'node -v'
        sh 'npm test'
    }
```

هاتشتغل عالـ default container

شرح الكود:

- استخدمنا parallel block علشان نشغّل الـ stages مع بعض .
- Stage 18 & 19 بيشتغلوا داخل نفس الـ Kubernetes Pod لكن كل واحد في container مختلف.
- Stage 20 بتشتغل داخل Docker container مستقل (مش على الـ Pod)

الخطوة 3 – تجنّب Port Conflict

- في مرحلة Node 19 ، أضاف سطر : sleep 10
- السبب:
  - الاتنين Node 18 & 19 شغالين داخل نفس الـ Pod على نفس الـ volume فلو التطبيق بيستخدم نفس الـ port (زي 3000 مثلًا)، ممكن يحصل conflict
  - فالسطر ده بيأخّر تنفيذ المرحلة 10 ثواني علشان ما يتصادموش .

هو عارف إن ده ضد فكرة الـ parallel processing شوية، بس عملها بس علشان يتجنّب الـ conflict في demo الشرح .

الخطوة 4 – تشغيل الـ Pipeline

بعد التعديل، عمل:

- git add .
- git commit -m "Added parallel unit test stages"
- git push

فـ Jenkins اشتغل build رقم 13، وظهر إن Unit Test Stage فيها 3 parallel stages :

- Node 18 ✅ ← نجح
- Node 19 ✅ ← نجح
- Node 20 ❌ ← فشل

🧩 الخطوة 5 – تحليل الخطأ في Node 20

➢ في الـ logs ظهر: 'Error: Cannot find module 'mocha

➢ 📍 السبب الحقيقي:

- الـ npm install اتنفذ قبل كده داخل Kubernetes container (اللي فيه الـ node_modules)
- لكن Stage Node 20 شغالة داخل Docker container مستقل.

فبالتالي الـ node_modules مش موجودة هناك، علشان كده الاختبارات فشلت.

```
stage('NodeJS 20') {
    agent {
        docker {
            image 'node:20-alpine'
        }
    }
    options { retry(2) }
    steps {
        sh 'npm install'
        sh 'node -v'
        sh 'npm test'
    }
}
}
```

مش أحسن حل .. والأفضل تستخدم Sequential Stages

🧩 الخطوة 6 – اقتراح الحل

قالك إن فيه حلين:

1️⃣ تحط سطر npm install داخل Stage Node 20

- ده هيخليها تنجح، بس هيزود التكرار.

2️⃣ أحسن حل ← تعمل Stage جديدة مخصوصة لـ dependencies وتخليها تشتغل ضمن الـ parallel stages دي باستخدام sequential stages (هاتتشرح الدرس الجاي)

💡 Best Practices / Real Use Cases / Tricks

✅ Best Practices :

- استخدم parallel stages لما تحب تختبر أكوادك على نسخ مختلفة من runtime (Node, Python, Java...)
- لو عندك containers بتشارك نفس الـ volume ، راقب الـ port usage لتجنّب conflicts
- خليك دايمًا consistent في مكان تثبيت dependencies .. يا إما داخل نفس Pod ، يا إما استخدم stage خاصة للـ npm install
- استخدم sequential stages لو محتاج تتابع steps جوا parallel block

🧰 Real Use Case :

شركة كبيرة بتعمل CI/CD لتطبيق Node.js بتختبر الكود على:

- Node 18 (Production env)
- Node 20 (Upcoming LTS)
- Node 21 (Experimental)

وبتستخدم parallel stages علشان تشوف الكود ثابت على كل نسخة.

🪄 Trick :

ممكن تحفظ ملف package-lock.json كـ artifact بين stages علشان كل container يعيد استخدام نفس dependencies بدون npm install جديد كل مرة .

📋 الملخص النهائي

العنصر	الشرح
🎯 الهدف	اختبار الكود على نسخ متعددة من Node.js باستخدام parallel stages
⚙️ الأداة	Jenkins Pipeline + Kubernetes Pod + Docker
🧩 الخطوات	تعديل stage الوحدة → إضافة parallel stages لـ 18 و 19 و 20
⚠️ المشكلة	اختلاف الـ environment خلّى Stage Node 20 تفشل (مش شايفة node_modules)
💡 الحل المقترح	Stage خاصة لـ dependencies + استخدام sequential stages
🧠 النتيجة	Parallel testing بين نسخ Node مختلفة لتحسين compatibility وتقليل وقت الـ CI

Sequential Stages

المقدمة

- ❑ في الدرس السابق اتكلمنا عن parallel stages .. يعني تشغيل أكثر من stage في نفس الوقت داخل Jenkins pipeline (زي اختبار الكود على Node 18 و19 و20)
- ❑ لكن دلوقتي بنتكلم عن Sequential Stages
  - 💡 وهي ببساطة : مجموعة stages بتشتغل ورا بعض (تتابعيًا) داخل كل فرع parallel
- ❑ يعني لو عندك فرع parallel مخصوص لـ Node 19 ، ممكن جوّاه تعمل:
  - Stage 1: Install Dependencies
  - Stage 2: Run Unit Tests
- ❑ الاتنين يشتغلوا ورا بعض، بس الفرع كله (Node 19) هيشتغل بالتوازي مع الفروع التانية (Node 18, Node 20)
  وده بيديك وضوح أكتر في واجهة Jenkins UI .. لأنك بتشوف مراحل التنفيذ منفصلة بدل ما تدور عليها في الـ logs

هانشغل Stages توالي جوا Stage كبيرة (فرع) موجودة في حالة توازي مع Stages تانية (فروع أخري)

الشرح خطوة بخطوة

الخطوة 1 – فكرة الـ Sequential Stages

- ➢ قبل الميزة دي، كان ممكن تعمل stages متوازية فقط،
- ➢ لكن دلوقتي تقدر تعمل stages جوا كل فرع parallel ، يعني مثلاً:

```
Parallel Branch (Node 19)
├── Stage: Install Dependencies
└── Stage: Run Tests
```

- الاتنين بيشتغلوا بالتتابع (Sequentially) داخل الفرع ده فقط بس في نفس الوقت الفرع دا ككل في حالة توازي مع فروع تانية

الخطوة 2 – الحالة العملية (Use Case)

- ➢ كان فيه مشكلة في الدرس السابق:
  Stage Node 20 فشلت لأنها مش لاقية dependencies
  (ملف node_modules)
- ➢ فالحل الطبيعي هو:
  - نضيف Stage جديدة قبل الـ Test علشان نعمل npm install داخل نفس الفرع.
    - يعني بدل ما نكتب كل حاجة في Stage واحدة، نخليها اتنين متتابعين:
      - Install Dependencies
      - Run Unit Tests

```jenkinsfile
stage('NodeJS 20') {
    agent {
        docker {
            image 'node:20-alpine'
        }
    }
    stages {
        stage ('Install Dependencies') {
            options { retry(2) }
            steps {
                sh 'node -v'
                sh 'npm install --no-audit'
            }
        }
    stages {
        stage ('Testing') {
            options { retry(2) }
            steps {
                sh 'node -v'
                sh 'npm test'
            }
        }
    }
}
```

Main stage, In Parallel with other stages

First stage in the main stage, In Series with the second stage

The second stage that in series with first stage.

الخطوة 3 – التعديل في Jenkinsfile

- ➢ هو عدّل الكود بالشكل التالي :

📘 شرح الكود:

- استخدمنا stages داخل stage ← يعني Sequential stages
- المرحلة الأولى بتعمل npm install --no-audit علشان تثبّت الباكدجات.
- المرحلة التانية بتشغل الاختبارات.
- الاتنين بيتنفذوا بالتتابع، بس الفرع Node 20 بيشتغل بالتوازي مع الفروع التانية.

الخطوة 4 – تشغيل الـ Pipeline

- بعد ما حفظ الملف عمل:

- `git add .`
- `git commit -m "Added sequential stages"`
- `git push`

- فـ Jenkins شغّل Build رقم 14.
- في الواجهة شاف إن فيه فرع جديد Parallel (Node 20) جواه 2 stages متتابعين:
  - Install Dependencies
  - Testing

لكن الـ pipeline فشلت في المرحلة الأولى.

الخطوة 5 – تحليل الخطأ

- الخطأ كان : Error: No such file or directory
- وبعدين في الـ logs ظهر: Your cache folder contains root-owned files due to a bug in previous versions.
- السبب:

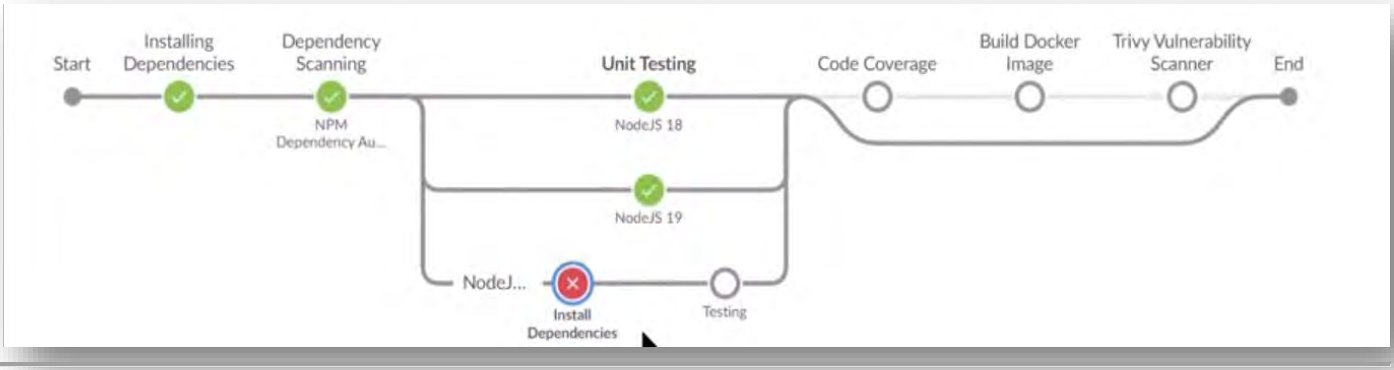
مجلد الكاش بتاع npm (~/.npm) كان فيه ملفات مملوكة للمستخدم root بسبب Bug في نسخ npm القديمة.

الخطوة 6 – محاولة الحل

- قال هيجرّب يخلي npm يخزن الكاش في المسار الحالي بدل الافتراضي،
- فعدّل الأمر كده: `npm install --no-audit --cache .`
- وسجّل التعديل:

- `git commit -m "use current dir for npm cache"`
- `git push`

- لكن لما شغّل الـ pipeline رقم 15، لسه المرحلة بتفشل في نفس النقطة. فقال إن المشكلة دي مش من Jenkins ، لكن من Node.js / npm نفسها، ومش هيكمّل فيها دلوقتي . (انطفي ← فااهمك عموما .. بس هايحلها الدرس الجاي)



الخطوة 7 – النتيجة

رغم إن المرحلة فشلت، الشرح وضّح إن:

- Sequential stages اشتغلت فعلاً.
- Jenkins بقى يعرض مراحل الفرع بوضوح في UI
- Testing stage ما اشتغلتش لأن Install Dependencies فشلت (وهي متتابعة معاها).

لكن عموما دي بالضبط طريقة عمل Sequential stages :

أي مرحلة تفشل ← المراحل اللي بعدها في نفس الفرع مش هتشتغل.

💡 Best Practices / Real Use Cases / Tricks

✅ Best Practices :

- استخدم Sequential stages لو عايز مراحل فرعية منظمة داخل فرع parallel (زي build → test → deploy)
- دايمًا افصل Install عن Testing علشان تقدر تعيد تشغيل كل واحدة لوحدها بسهولة.
- لو عندك مشاكل في npm permissions أو cache داخل Jenkins agent ، استخدم:
  - npm install --no-audit --unsafe-perm=true
  - لكن بحذر، ويفضل في بيئة test فقط

🧰 Real Use Case :

- في شركات كبيرة بيستخدموا Jenkins pipeline كده:
- كل فرع parallel بيمثل نسخة مختلفة من الـ runtime ، وجواه مراحل متتابعة sequentially

```
Parallel:
 ├─ Node 18:
 │    ├─ Install Dependencies
 │    └─ Run Tests
 ├─ Node 20:
 │    ├─ Install Dependencies
 │    └─ Run Tests
 └─ Node 22:
      ├─ Install Dependencies
      └─ Run Tests
```

🪄 Trick :

لو عندك ملفات dependencies أو artifacts محتاج تنقلها بين stages في نفس الفرع،
ممكن تستخدم Jenkins feature اسمها : stash / unstash

📜 الملخص النهائي (Summary)

العنصر	الشرح
🎯 الهدف	تشغيل أكثر من Stage متتابع داخل فرع Parallel واحد
⚙️ الوظيفة	Install Dependencies ثم Run Tests في ترتيب معين
🧩 الأوامر الأساسية	`npm install --no-audit` و `npm test`
⚠️ المشكلة	npm cache ownership error (Node.js issue)
💡 الحل النظري	تغيير cache directory أو استخدام permission flag
🧠 النتيجة	فهمنا إزاي نستخدم Sequential stages بداخل Parallel branches
➡️ الخطوة الجاية	تعلم كيفية نقل الملفات بين stages باستخدام `stash/unstash`

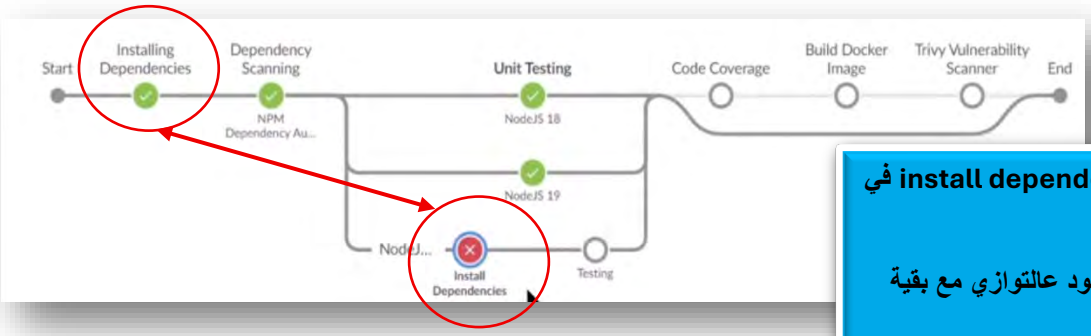
Stash and Unstash

🧠 أولًا: الفكرة العامة

- ❑ في الجزء ده بيتكلم عن مفهومين مهمين جدًا في Jenkins pipelines وهما Stash و Unstash
- ❑ الفكرة إنك في Jenkins ممكن تعمل stages مختلفة بتشتغل على Nodes مختلفة
  (زي agent أو container أو Kubernetes pod)
- ❑ لكن في بعض الحالات بتحتاج تنقل ملفات أو dependencies من stage إلى تانية داخل نفس pipeline run
  ودي بقى وظيفة stash/unstash

- 🔹 Stash = بيخزن (يحفظ) مجموعة من الملفات علشان تستخدمها لاحقًا في نفس الـ pipeline
- 🔹 Unstash = بيسترجع الملفات اللي اتعملها stash في stage تانية.

⚙️ ثانيًا: الشرح خطوة بخطوة

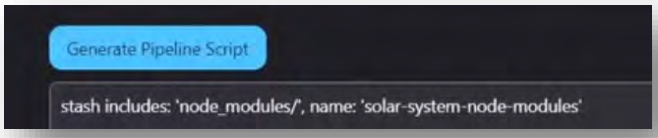


لو هتلاحظ فهتلاقي اننا اصلا كنا عاملين install dependencies في تاني stage وكانت ناجحة

بس لما جينا عملناها عالتوالي في الفرع اللي موجود عالتوازي مع بقية الفروع ففشلت

فأنا هاستخدم الكونسيبت بتاع الـ stash/unstash بحيث ان الـ stage اللي فشلت دي تقدر تشوف وتستخدم نفس الـ node_module بتاع الـ stage اللي نجحت

1. 🧩 إعداد الفكرة واستخدام stash



🎯 الهدف:

نخزن ملفات معينة (زي node_modules) – بعد ما نعمل npm install– علشان نستخدمها بعدين في stage تانية بدون ما نكرر نفس العملية.

هانروح علي : Pipeline syntax → snippet Generator → sample step → stach: stash some files

- o Name: solar-system-node-modules (لازم يكون الاسم unique)
- o Include: node_modules/ (بيتيحلك تحددله هو هايحتفظ بإيه علشان يستخدمه في بقية الـ stages ..
  هو هنا حددله الفولدر بتاع الـ node_modules)
- o ممكن كمان تستخدم excludes لو عايز تستثني حاجات معينة.

📍 المكان:

الـ stash بيتكتب داخل stage بعد الخطوات اللي خلصت الملفات (مثلا بعد npm install)

```
stages {
    stage('Installing Dependencies') {
        options { timestamps() }
        steps {
            sh 'node -v'
            sh 'npm install --no-audit'
            stash(includes: 'node_modules/', name: 'solar-system-node-modules')
        }
    }
```

💡 ملاحظات عملية (Best Practices) :

- استخدم اسم واضح للـ stash يعبر عن محتواه (زي frontend-build أو backend-modules)
- خلي الـ stash يحصل بعد الخطوة اللي بتنتج الملفات مباشرة.
- لو الـ pipeline كبيرة، ممكن تضيف timestamps في الاسم لو بتعمل أكتر من stash

---

2. 🧩 استخدام unstash في Stage تانية

🎯 **الهدف:**

نسترجع الملفات اللي اتخزنت باستخدام stash في Stage مختلفة (حتى لو على Node تاني)
بنستخدمه في الـ stages اللي هاتحتاج تستخدم ملفات من stages تانية

```
stage('NodeJS 20') {
    agent {...
    }
    stages {
        stage ('Install Dependencies') {
            options { retry(2) }
            steps {
                sh 'node -v'
                //sh 'npm install --no-audit --cache .'
                unstash 'solar-system-node-modules'
            }
        }
```

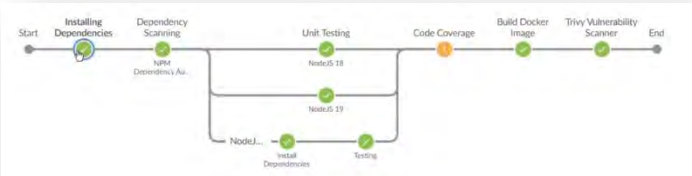
📍 **المكان:**

بنستخدمه عادة (بس مش شرط) في stage بعديه، زي مرحلة testing أو build

- هنا بنستدعي stash اللي اسمه SolarSystemNodeModules
- Jenkins هيعمل extract للملفات دي في workspace الحالي.

💡 **ملاحظات:**

- لو تتذكر كان عندنا قبل كدا مشكلة في مرحلة npm install داخل Kubernetes Pod (stage تانية)
- فبدل ما نعمل npm install من الأول، استخدمنا unstash علشان نسترجع الـ node_modules اللي اتعملها install قبل كده.
- كده وفرنا وقت وبنفس الوقت حلّينا مشكلة environment مختلفة.



3. 🧩 النتيجة

- بعد ما أضاف stash بعد الـ npm install في أول stage
  - Jenkins قال : Stashed 4898 files successfully
- بعدين في الـ sequential stage ، استخدم unstash وقال: Restored files previously stashed successfully

وبكده الـ pipeline اشتغلت كلها بنجاح، حتى الـ stages اللي كانت بتفشل قبل كده بسبب Node.js issue بقت تمام.

🧠 ثالثًا : Best Practices و Tricks

✅ **استخدم stash/unstash لما تكون :**

- عايز تنقل ملفات بين stages بدون commit في Git
- عندك agents مختلفة (Windows, Linux, Docker, Kubernetes)
- بتبني project فيه multi-platform build/test matrix

⚠️ **Avoid :**

- تخزين ملفات ضخمة جدًا (هيتقل الـ master node)
- استخدام stash بين pipelines مختلفة (يعني هو بس بيشتغل داخل نفس pipeline run)
- تخزين أسرار أو مفاتيح داخل stash

💡 **Trick واقعية:**

لو بتحتاج تستخدم الـ stash في rerun بعد فشل pipeline ، استخدم الخيار : options { preserveStashes() }
ده بيخلي Jenkins يحتفظ بالـ stashes حتى بعد نهاية الـ pipeline

---

📋 الملخص

المرحلة	الفكرة	الأداة / الأمر	الهدف
1	إنشاء stash	`stash name: 'X', includes: '**/'path`	حفظ ملفات لاستخدامها لاحقًا
2	استرجاع stash	`unstash 'X'`	استرجاع الملفات في stage تانية
3	Preserve stashes	`options { preserveStashes() }`	الاحتفاظ بالملفات بعد انتهاء pipeline
💡	Use Cases	npm modules – build artifacts – compiled binaries	توفير وقت وتفادي مشاكل البيئات المختلفة

Pipeline Caching

🧠 أولًا: المقدمة

- ❑ الجزء ده بيتكلم عن Dependency Caching داخل Jenkins Pipeline
- ❑ الفكرة ببساطة إنك لما تيجي تبني (build) مشروعك كل مرة، Jenkins بينزل الـ dependencies (زي مكتبات npm أو pip أو maven) من الأول، وده بياخد وقت.
- ❑ فعلشان نوفر وقت وموارد، بنستخدم الـ caching – يعني نخزّن النسخ اللي اتبنت قبل كده، ولما Jenkins يشتغل تاني، فبدل ما يبدأ من الصفر، يستخدم الكاش اللي موجود.

🚀 وده مفيد جدًا خصوصًا لو Jenkins بيشتغل داخل containers (اللي دايمًا بتبدأ من zero state)

🧩 ثانيًا: الشرح التفصيلي خطوة بخطوة

Job Cacher | Jenkins plugin

1️⃣ الفكرة الأساسية داخل Jenkins

- ➢ Jenkins مش بيعمل cache لوحده ولكنه محتاج plugin اسمه : **Job Caching Plugin**

الـ plugin ده بيسمحلك إنك تخزّن dependencies (زي node_modules) أو build artifacts (زي ملفات compiled)

📌 **Best Practice : استخدمه خصوصًا لو عندك stages بتشتغل داخل Docker أو Kubernetes pods**

2️⃣ مرحلة "Install Dependencies"

- ➢ مرحلة install dependencies بتاخد حوالي 6 ثواني بس لأن المشروع صغير،
- ➢ لكن في مشروعات ضخمة فيها آلاف الـ dependencies ، ممكن تاخد دقايق كاملة، وهنا الكاش بيعمل فرق ضخم.

🎯 الهدف: نعمل cache للـ node_modules علشان Jenkins ما يعيدش تحميلها كل مرة.

3️⃣ تثبيت الـ Plugin

الأوامر الأساسية هنا:

- تروح لـ Jenkins → Manage Jenkins → Plugins → Available
- وتبحث عن Job Caching Plugin وتعمل Install
- بعد التثبيت، Jenkins لازم يعمل Restart

📌 **Real Tip : دايمًا اعمل restart بعد أي plugin جديد علشان التغييرات تتفعل.**

4️⃣ إنشاء الكاش من Snippet Generator

- ➢ تروح علي Jenkinsfile → Pipeline Syntax → Snippet Generator → Step: cache
- ➢ هيفتحلك فورم تملاه زي كده:
  - Path: node_modules
  - Cache name: npm-dependency-cache
  - Cache validity file: package-lock.json
  - Compression: tar or zip
  - Max cache size: 550 MB (المفروض تكون كفاية)

📌 **Trick :**

package-lock.json مهم جدًا لأنه بيتغير لما تضيف أو تشيل dependency ، وده اللي بيخلي Jenkins يعرف إن الكاش بقى قديم فيعمل refresh تلقائي

Generate Pipeline Script

```
cache(caches: [arbitraryFileCache(cacheName: 'npm-dependecny-cache', cacheValidityDecidingFile: 'package-lock.json', excludes: '', includes: '**/*', path: 'node_modules')],
defaultBranch: '', maxCacheSize: 550) {
  // some block
}
```

5 إضافة الكاش داخل Jenkinsfile

- تروح لملف الـ pipeline الأساسي، وتضيف الـ cache block داخل الـ step الخاص بالـ stage ('Install dependencies')
- هيبقى الشكل كده:

```
stages {
    stage('Installing Dependencies') {
        agent any
        options { timestamps() }
        steps {
            cache(maxCacheSize: 550, caches: [
                arbitraryFileCache(
                    cacheName: 'npm-dependecny-cache',
                    cacheValidityDecidingFile: 'package-lock.json',
                    includes: '**/*',
                    path: 'node_modules')
                ] ) {
                    sh 'node -v'
                    sh 'npm install --no-audit'
                    stash(includes: 'node_modules/', name: 'solar-system-node-modules')
                }
        }
    }
```

Run it on jenkins controller

لاحظ انه عمله جوا الـ step وكمان أضاف الـ commands جوا الـ cache

📌 ملاحظة مهمة:

- **الكاش بيتحفظ داخل Jenkins نفسه (في المسار الخاص بالـ job)**
- **وفي الـ build الجاي، Jenkins هيدور على كاش مطابق للـ hash بتاع package-lock.json ويستخدمه بدل ما يعيد التثبيت.**

---

6 التعامل مع stages تانية

- كمان وضّح إنك لازم تعمل : unstash 'node_modules' في الـ stages اللي بتحتاج الـ dependencies دي (زي test أو code coverage) خصوصًا لو بتشغلهم في Kubernetes Pods منفصلة . (agents مختلفة يعني)

📌 Best Practice :

استخدم unstash و stash لو بتفصل stages في بيئات مختلفة (controllers / agents)

---

7 تشغيل الـ Pipeline

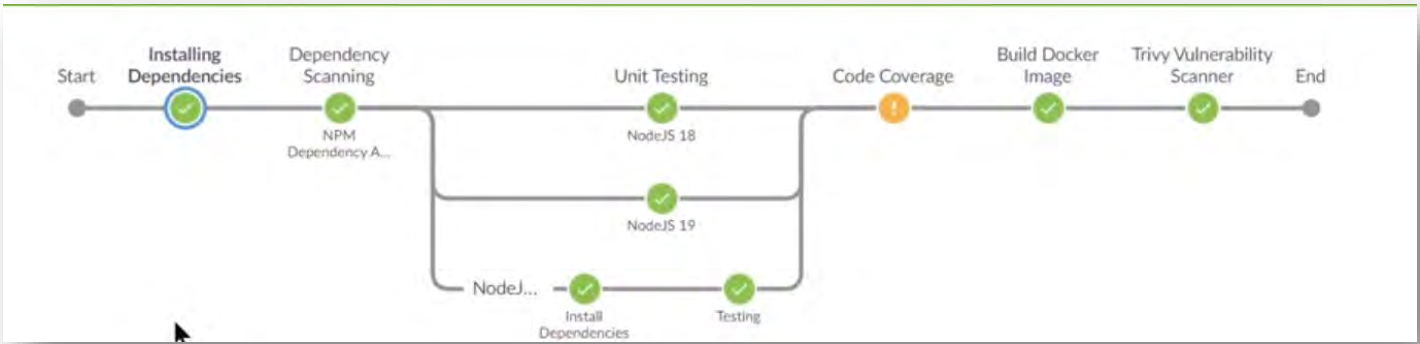
- عمل commit & push ← Jenkins شغّل pipeline جديدة ← راح يراجع logs
- في أول build ، Jenkins قال : "skip restoring cache as no up-to-date cache exists"
- يعني مفيش كاش جاهز، فاشتغل طبيعي ونفّذ npm install وبعدها خزّن النتيجة كـ cache جديد.

---

8 التحقق في الـ Logs

- في build رقم 18 (التاني)، لما عمل rerun ، Jenkins كتب:
  - "restoring cache"
    "cache is up-to-date, skipping npm install"

💡 يعني الكاش اشتغل صح، وتم استخدام النسخة القديمة بدل التثبيت الجديد — فالتنفيذ أسرع بكتير.



💎 ثالثًا: ملاحظات وBest Practices

✅ استخدم package-lock.json أو requirements.txt كـ cache key
✅ راقب حجم الكاش (550MB أو أكتر حسب المشروع)
✅ استخدم كاش لكل نوع dependency (npm / pip / maven / gradle)
✅ دايمًا جرّب أول build manual علشان تتأكد الكاش شغال.
✅ استخدم Blue Ocean UI للـ stages visualization ، والـ Classic UI للـ log details

---

📄 رابعًا: الملخص النهائي

الفكرة	الهدف	الأداة
Dependency Caching	تسريع build pipeline	Job Caching Plugin
Snippet Generator	لإنشاء الكاش بسهولة	cache step
package-lock.json	لتحديد صلاحية الكاش	cache key
npm install	مرحلة التنفيذ داخل الكاش	step command
unstash/stash	لنقل الكاش بين stages	Jenkins built-ins

💡 في النهاية:
الكاش بيخلّي Jenkins أسرع، أوفر في الموارد، وأكتر استقرارًا، خصوصًا لو عندك microservices أو مشاريع كبيرة بتتبني كل شوية.

Invalidate Cache

🧠 المقدمة

- ❑ الدرس ده بيتكلم عن Cache Invalidation .. يعني إزاي تخلي Jenkins أو أي CI/CD system يعرف إن الكاش (اللي خزن dependencies قبل كده) بقى قديم، فلازم يعمله refresh أو update
- ❑ الموضوع ده مهم جدًا في المشاريع اللي فيها Node.js أو أي dependency manager (زي npm أو pip أو Maven)
  - لأننا مش عايزين كل Build يعيد تحميل كل الـ packages من الإنترنت، فبنستخدم Cache علشان نسرّع العملية.
- ❑ لكن لو الـ dependencies اتغيّرت (زي لما تضيف أو تشيل package في package.json)
  - لازم Jenkins يعرف إن الكاش القديم خلاص مش صالح.
- ❑ الدرس ده بيشرح بالظبط إزاي بيحصل الكلام ده باستخدام :
  - o Job Cache Plugin
  - o وملف package-lock.json كـ "cache validity deciding file"

🧩 الشرح خطوة بخطوة

📘 الخطوة 1: المقدمة – إنشاء Cache سابقًا

- ➢ سابقا كنا أنشأنا Cache عشان نحفظ الـ dependencies اللي بننزلها npm .. وده بيوفر وقت كبير في الـ build التالي بدل ما يعيد التحميل من الصفر.
- ➢ الخطوة الجاية هي : إزاي نعمل invalidate للكاش ده؟
  - يعني نخليه يتجاهل الكاش القديم ويعمل Cache جديد لما يحصل تغيير في الـ dependencies

---

⚙️ الخطوة 2 : تعديل في الـ Dependencies

- ➢ دخل على المشروع (Solar System repository) وبدأ يضيف Dependency جديدة عشان يشوف الكاش هيشتغل إزاي بعد التغيير.
- ➢ الأمر المستخدم: npm install localtunnel
- 🔹 ده بيضيف مكتبة جديدة (localtunnel) في المشروع، وبالتالي بيتم تحديث ملفين مهمين:
  - package.json
  - package-lock.json

💡 معلومة مهمة:

ملف package-lock.json هو اللي بيحتوي على الـ exact versions لكل dependency ، فعشان كده بنستخدمه لتحديد صلاحية الكاش (hash value)

📦 الخطوة 3: تأثير التغيير على الكاش

- ➢ بعد ما أضاف dependency جديدة، بالتالي package-lock.json اتغير، وده يعني إن الـ hash value بتاعه كمان اتغير.
- ➢ الفكرة بقا ان لما Jenkins ييجي يبني المشروع تاني، الـ Job Cache Plugin هيحسب الـ hash الجديد، ويقارنه بالقديم.
  - إذا اختلف ← Jenkins هيعمل invalidate للكاش القديم وهيبدأ يحمله من جديد.

🧪 الخطوة 4: تشغيل Pipeline جديدة

- بعد التعديل عمل Commit و Push عشان الـ pipeline تشتغل تاني تلقائيًا.

- لما الـ pipeline تشتغل، أول مرحلة بنبص عليها هي مرحلة تثبيت الـ dependencies (install dependencies stage)
- في الـ logs ، بدأ يدور على كلمة "cache" .. ولقى إن الـ hash بتاع package-lock.json اتغير:
  - القديم كان بيبدأ بـ A47
  - الجديد بقى 5115

يبقى واضح إن Jenkins اكتشف إن الكاش بقى outdated

```
17:01:01  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Searching cache in job specific caches...
17:01:01  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] got hash a47b9ef02dbc79db72ab6385105e0142 for
cacheValidityDecidingFile(s) - actual file(s): /var/lib/jenkins/workspace/lar-system_feature_advanced-demo/package-lock.json
17:01:01  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Found cache in job specific caches
17:01:01  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Restoring cache...
17:01:02  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Cache restored in 771ms
[Pipeline] {
[Pipeline] sh
17:01:02  + node -v
17:01:02  v22.6.0
[Pipeline] sh
17:01:02  + npm install --no-audit
17:01:04
17:01:04  up to date in 1s
```

دا الـ cache القديم .. ولاحظ الـ hash

```
[Pipeline] cache
17:06:25  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Searching cache in job specific caches...
17:06:25  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] got hash 5a15d94c8bab08a6882fddf4b8ef16c2 for
cacheValidityDecidingFile(s) - actual file(s): /var/lib/jenkins/workspace/lar-system_feature_advanced-demo/package-lock.json
17:06:25  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] cacheValidityDecidingFile configured, but previous
hash does not match - cache outdated
17:06:25  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Searching cache in default caches...
17:06:25  [Cache for node_modules (npm-dependecny-cache) with id 3ec03583f8eaec275cb2183db769ff47] Skip restoring cache as no up-to-date cache exists
[Pipeline] {
[Pipeline] sh
17:06:25  + node -v
17:06:25  v22.6.0
[Pipeline] sh
17:06:26  + npm install --no-audit
17:06:27
17:06:27  added 7 packages in 2s
17:06:27
```

دا الـ cache الجديد .. ولاحظ ان الـ hash اتغير .. فإكتشف انه مش متطابق مع القديم .. فراح يضيف الـ package ويعملها install تاني

الخطوة 5: كيف يتعامل Jenkins مع الكاش الجديد؟

- Jenkins يقول: "Cache is outdated. Skipping restoring cache as no up-to-date cache exists."
  - يعني مش هيستخدم الكاش القديم.
  - بعد كده بيبدأ يعمل: npm install من الأول.

بعد ما يخلص التثبيت، Jenkins هيحسب الـ hash الجديد ويعمل Cache جديد بناءً عليه.

🧠 Best Practices & Real Use Cases

✅ دايمًا استخدم package-lock.json كـ"Cache Validity File"
لأنه أكتر ملف بيمثل الحالة الفعلية للـ dependencies

✅ لو المشروع كبير، caching ممكن يقلل زمن build من 10 دقايق ← 2 دقيقتين بس.

✅ تجنب إنك تعمل Cache للـ node_modules نفسه يدويًا،
الأفضل تسيب Jenkins يديره عن طريق الـ plugin ، لأن أي اختلاف بسيط في الـ lock file ممكن يعمل تعارضات (dependency mismatch)

✅ دايمًا بعد أي تغيير في الـ dependencies ، اعمل Commit للـ lock file ، لأنه هو اللي بيخلي Jenkins يعرف إن فيه تغيير حصل.

🧩 ملخص شامل

- الـ Cache بيوفر وقت في الـ builds لأنه بيخزن الـ dependencies اللي تم تثبيتها.
- لكن لازم نعرف إمتى الكاش بقى قديم (invalid)
- لما نضيف أو نحذف dependency ، بيتغير ملف package-lock.json، وده بيغيّر الـ hash اللي Jenkins بيستخدمه للتحقق من صلاحية الكاش.
- لو الـ hash اتغير ← Jenkins يعرف إن الكاش قديم ← يعمل Cache جديد.
- الـ Job Cache Plugin في Jenkins بيسهل العملية دي جدًا عن طريق خيار: "Cache validity deciding file"
- وبكده تقدر توازن بين السرعة والدقة في إدارة الـ builds داخل Jenkins

المرحلة	الفكرة	الأداة / الأمر
إنشاء الكاش	نحفظ الـ dependencies لتقليل وقت الـ build	Jenkins Cache Plugin
التغيير في dependencies	npm install أو npm remove	`npm install localtunnel`
اكتشاف التغيير	Jenkins يحسب hash جديد لـ package-lock.json	Built-in hash logic
إبطال الكاش	إذا الـ hash اختلف → الكاش يتلغي	Jenkins auto-detect
إنشاء كاش جديد	بعد التثبيت الجديد	Jenkins job cache re-creation
إعادة الاستخدام	في الـ builds التالية طالما الـ hash ثابت	Cache restore

Declarative vs Scripted Pipeline

🧠 المقدمة

- ❑ الجزء ده بيتكلم عن الفرق بين نوعين من Jenkins Pipelines :
  1. Declarative Pipeline
  2. Scripted Pipeline
- ❑ الـ Pipelines دي هي الأساس اللي بنبني عليه الـ CI/CD automation داخل Jenkins
- ❑ الاتنين بيتم كتابتهم في ملف اسمه Jenkinsfile، لكن الفرق الأساسي بينهم في طريقة الكتابة (syntax) ودرجة المرونة والتحكم اللي كل واحد بيوفرها.

🚨 الفهم الجيد للفرق ده مهم جدًا لأي DevOps Engineer، علشان تختار النوع المناسب بناءً على طبيعة المشروع — هل محتاج مرونة كاملة ولا سهولة وسرعة في الكتابة.

⚙️ الشرح التفصيلي خطوة بخطوة

1 🧩 تعريف الأنواع

يبقى عندنا نوعين فقط من Pipelines :

- Declarative Pipeline ← صيغة أبسط وسهلة القراءة.
- Scripted Pipeline ← صيغة مرنة ومعقدة تعتمد على الكود.

الاتنين بيتنفذوا بنفس الآلية الداخلية في Jenkins (نفس الـ Pipeline subsystem) وبيستخدموا نفس اللغة الأساسية: Apache Groovy 🧠

---

2 🧩 القواسم المشتركة

مهما كان النوع اللي تختاره:

- الملف اسمه دايمًا Jenkinsfile
- بيتنفذ بنفس المبدأ داخل Jenkins
- ممكن تستخدم فيهم Shared Libraries (مكتبات Jenkins reusable code)

📌 Shared Libraries :

عبارة عن أكواد جاهزة reusable بتسهل إنك تستخدم نفس steps أو functions في كذا project من غير ما تعيد كتابتهم.

---

3 🧩 Scripted Pipeline

ده النوع الأقدم من الـ Pipelines ، ومعتمد على كتابة كود Groovy بالكامل.

يعني كأنك بتبرمج فعلاً، مش مجرد بتوصف خطوات.

🔹 المزايا:

- 🔧 تحكم كامل : تقدر تعمل أي منطق (logic) عايزه، loops, if conditions, exceptions، إلخ.
- 🧩 مرونة تامة : مناسب للـ Workflows المعقدة جدًا.
- 🧠 استخدام كامل لـ Groovy : ممكن تستخدم متغيرات، دوال، واستيراد مكتبات خارجية.

🔹 العيوب:

- 🧗 صعب التعلم شوية : لازم تكون فاهم Groovy أو عندك خلفية برمجية.
- 😵 Syntax معقد : أكواد طويلة وصعبة الفهم للي مش مبرمج.

📄 مثال بسيط على Scripted Pipeline :

```
node {
    stage('Build') {
        echo 'Building the project...'
    }
    stage('Test') {
        echo 'Running tests...'
    }
    stage('Deploy') {
        echo 'Deploying application...'
    }
}
```

🧠 Real Trick :

في المشاريع الكبيرة جدًا اللي فيها logic متغير
(زي conditions حسب نوع الـ branch أو deployment region)
دايمًا يفضل إنك تستخدم Scripted لأنه بيديك تحكم أدق في الـ flow

---

🧩 4️⃣ Declarative Pipeline

ده النوع الأحدث والأكثر انتشارًا حاليًا في Jenkins
الهدف منه : سهولة القراءة والكتابة لأي DevOps أو حتى Developer مش خبير في Groovy

🔹 المزايا:

- ✨ سهل وواضح : قريب جدًا من اللغة الطبيعية.
- 🚀 سريع في الإعداد perfect : لو عايز تبني pipeline بسرعة.
- 💡 أقل أخطاء : لأن الـ syntax محدود وواضح.

🔹 العيوب:

- ❌ مرونة أقل : مش مناسب للـ Workflows المعقدة جدًا.
- 🚫 مش بيقبل scripting حر بالكامل :لازم تلتزم بالبنية المحددة (structure) اللي بيقبلها Jenkins

📄 مثال على Declarative Pipeline :

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building the project...'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying application...'
            }
        }
    }
}
```

🧠 Note :

ده النوع اللي Jenkins بيشجع عليه حاليًا لأنه أسهل في الفهم والمشاركة بين الفرق.

5️⃣ 🧩 امتى تختار كل نوع

الحالة	النوع المناسب	السبب
مشروع بسيط أو متوسط	Declarative	سهل الكتابة والفهم، أقل كود
فريق في بداية التعامل مع Jenkins	Declarative	مثالي للمبتدئين
Workflow معقد جدًا	Scripted	بيدي مرونة كاملة
محتاج منطق شرطي (if, loop, function)	Scripted	يدعم Groovy logic بالكامل
مشاريع سريعة الإعداد	Declarative	setup أسرع وسهل الصيانة

```
stage('Check Env') {
  steps {
    script {
      if (env.BRANCH_NAME == 'main') {
        echo 'Deploying to production...'
      } else {
        echo 'Deploying to staging...'
      }
    }
  }
}
```

💡 ملاحظات عملية (Best Practices / Real Use Cases / Tricks)

✅ Best Practice #1 :

ابدأ دايمًا بـ Declarative Pipeline، ولو احتجت logic معقد جوه stage معينة، استخدم script block جواها (زي ماتعلمنا ف الشروحات السابقة)

✅ Best Practice #2 :

احفظ Jenkinsfile في root المشروع داخل Git علشان Jenkins يقدر يلقطه أوتوماتيك.

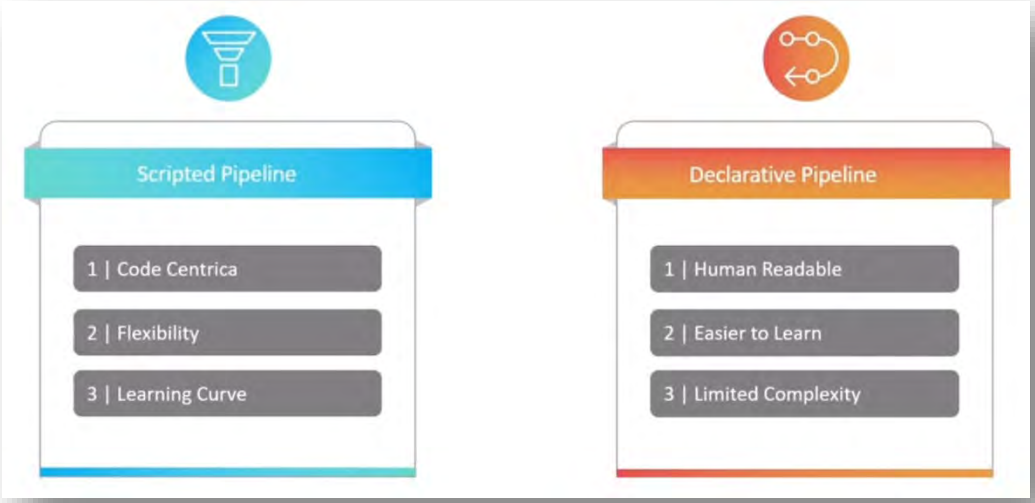
✅ Real Case :

شركات زي Google و IBM بتستخدم Declarative Pipelines كأساس للـ CI/CD templates ، لكن بيضيفوا جواها Scripted Blocks علشان يعملوا logic ديناميكي.

✅ Trick :

لو عندك كود Jenkins مكرر في كذا مشروع، حوّله لـ Shared Library reusable .. لأن ده بيشتغل في Declarative و Scripted الاتنين.

📜 الملخص الشامل



العنصر	Declarative Pipeline	Scripted Pipeline
السهولة	سهل وبسيط	معقد ويتطلب برمجة
المرونة	محدودة نسبيًا	عالية جدًا
اللغة	صيغة Jenkins محددة	Groovy كامل
مناسب لمين	المبتدئين والمشاريع البسيطة	الخبراء والمشاريع المعقدة
ملف التعريف	Jenkinsfile	Jenkinsfile
Shared Libraries	✅ مدعومة	✅ مدعومة
أمثلة الاستخدام	build/test/deploy البسيط	workflows متعددة الخطوات والمنطق

🎯 الخلاصة:

- لو عايز سرعة وسهولة ← استخدم Declarative
- لو محتاج تحكم ومرونة قصوى ← استخدم Scripted
- ولو عايز تجمع الاتنين ← استخدم Declarative وفيه script blocks

Demo Declarative vs Scripted Pipeline

cd-org/declarative-vs-scripted-pipeline

🧠 المقدمة

- ❑ في Jenkins عندك طريقتين رئيسيتين تكتب بيهم الـ Pipeline :
  1. Declarative Pipeline
     بتكون منظمة أكتر وسهلة القراءة، وبتستخدم بناء محدد (Structured Syntax)
  2. Scripted Pipeline
     بتعتمد على لغة Groovy ، مرنة جدًا ومفتوحة لأي logic أو condition
- ❑ هنا هتلاقيه عامل Repository فيه اتنين Jenkinsfile :
  - واحد اسمه Jenkinsfile.Declarative
  - والتاني Jenkinsfile.Scripted
- ❑ هايجربهم على نفس الـ job علشان يوضح الفرق العملي بينهم.

⚙️ الشرح خطوة بخطوة

🧩 1️⃣ Declarative Pipeline

```
pipeline {
    agent any

    stages {
        stage('Echo Message') {
            steps {
                echo 'This is executed within a DECLARATIVE Pipeline'
            }
        }
    }
    post {
        always {
            sh 'rm -rf *'
        }
    }
}
```

- pipeline {} ←
  هو البلوك الأساسي اللي كل declarative pipeline بتبدأ بيه.
- agent any ←
  معناها Jenkins يقدر يشغّل الـ job على أي Agent متاح.
- stage('Echo Message') ←
  عندك مرحلة واحدة بتعمل echo
- post { always { ... } } ←
  بيتنفذ دايمًا بعد انتهاء الـ pipeline (زي cleanup)

📦 لما شغّل الـ job :
ظهر عنده 3 stages بدل Stage واحدة:

1. Checkout SCM : Declarative ✅
   دي Jenkins بيعملها تلقائيًا في الـ Declarative Pipeline ، علشان يسحب الكود من الـ Git repo
2. Echo Message
3. Post Actions (always)

📌 المعلومة المهمة:
في Declarative Pipeline ، Jenkins بيعمل checkout تلقائي بدون ما تكتبه بنفسك.
وده بيوفر عليك خطوة كتابة checkout scm

🧩 2️⃣ Scripted Pipeline

```
node {
    try {
        stage('Echo Message') {
            sh 'echo This is executed within a SCRIPTED Pipeline'
        }
    } catch (err) {
        echo "Failed: ${err}"
    } finally {
        echo 'This will always run'
        //sh 'rm -r *'
    }
}
```

📖 الشرح:

- node {} ←
  هو الـ container اللي بيحتوي كل الـ stages
- try/catch/finally ←
  للتحكم في الأخطاء وتنفيذ أوامر بعد الانتهاء.
- لااحظ ان مفيش checkout scm هنا،
  لأن Jenkins ما بيعملش checkout تلقائي في الـ Scripted Pipeline

🎬 لما شغّل الـ job :

- Jenkins ما لاقاش ملفات في workspace لأنه مفيش checkout
- اشتغل الـ echo بس.
- بعد كده نفّذ الكود اللي في finally

📌 الفرق هنا:

لازم تضيف السطر ده يدويًا لو عايز Jenkins يسحب الكود من الـ repo : checkout scm

---

3️⃣ 🔧 تعديل Scripted Pipeline

بعد ما لاحظ إن الـ scripted pipeline ما عملش checkout ، أضاف :

```
node {
    try {
        stage('Echo Message') {
            checkout scm
            sh 'ls -ltr'
            sh 'echo This is executed within a SCRIPTED Pipeline'
        }
    } catch (err) {
        echo "Failed: ${err}"
    } finally {
        sh 'echo "This will always run"'
        sh 'rm -r *'
    }
}
```

📖 النتيجة:

- الـ Scripted Pipeline بقى بيعمل checkout فعلاً.
- تقدر تشوف في الـ logs إنه جاب الملفات من GitHub
- كده النتيجة بقت قريبة من الـ Declarative ، بس يدويًا.

💡 ملاحظات عملية (Best Practices & Real Use Cases)

➢ ✅ Automatic Checkout

- Declarative بيعمل checkout لوحده، Scripted محتاجك تضيف checkout scm

➢ ✅ Restart from Stage

- ميزة مهمة جدًا:
  - تقدر في Declarative Pipeline تعيد تشغيل الـ job من أي stage فشلت فيها.
  - لكن في Scripted Pipeline مش ممكن تعمل restart من مرحلة معينة.

➢ ✅ Error Handling

- Declarative بتستخدم post { failure { ... } }
- Scripted بتستخدم try/catch

➢ ✅ Real Use Case

- في شركات زي E& Egypt أو Orange :
  - بيستخدموا Declarative في الـ standard CI/CD pipelines
  - ولو فيه logic خاص (زي dynamic deploy أو matrix builds)
    بيضيفوا blocks script {} جوه declarative

➢ ✅ Cleanup & Post Actions

دايمًا خلّي cleanup في:

- post { always { sh 'rm -rf *' } } ← لو Declarative
- finally { sh 'rm -rf *' } ← لو Scripted

📋 الملخص الشامل

المقارنة	Declarative Pipeline	Scripted Pipeline
البداية	`{} pipeline`	`{} node`
Checkout	تلقائي	لازم تكتبه يدويًا
Error Handling	`post` blocks	`try/catch/finally`
Restart from Stage	✅ متاح	❌ غير متاح
سهولة القراءة	أعلى وأسهل	أصعب وأكتر مرونة
Best Use	Pipelines قياسية وثابتة	Pipelines فيها logic متقدم

🎯 الخلاصة:

- **لو عايز تشتغل بسرعة وتكتب Pipeline واضح وسهل ← استخدم Declarative**
- **لو عايز حرية كاملة وتتحكم في كل تفصيلة ← استخدم Scripted**
- **وأحسن حل عملي في الشركات الكبيرة:**

Declarative base + Script blocks جوه stages معينة لما تحتاج logic متقدم . ✅

Post Section in Declarative Pipelines vs. Finally Block in Scripted Pipelines

🧠 المقدمة

- ❑ كل Pipeline سواء Declarative أو Scripted : لازم يكون عنده طريقة للتعامل مع الحالات اللي بتحصل بعد تنفيذ المراحل (stages) :
  زي مثلاً:
  - تنظيف الملفات 🧹 (cleanup)
  - إرسال إشعار بالنجاح أو الفشل 📢
  - حفظ الـ artifacts أو logs 💾
- ❑ في Jenkins ، فيه طريقتين أساسيين نعمل بيهم الكلام ده:
  - الاتنين بيحققوا نفس الهدف العام (التنفيذ بعد انتهاء الـ stages)
  - بس الاختلاف في الـ syntax وطريقة التعامل مع حالات النجاح أو الفشل.

النوع	الأداة المستخدمة
Declarative Pipeline	`{ ... } post`
Scripted Pipeline	`{ ... } finally`

⚙️ الشرح التفصيلي

1️⃣ 🧩 في Declarative Pipeline ← post Section

📖 الشرح خطوة بخطوة:

- post ← دي الـ block اللي بتتنفذ بعد ما الـ stages كلها تخلص.
- جواها ممكن تحط أكتر من حالة:
  - success {} ← لما الـ pipeline تم بنجاح.
  - failure {} ← لو حصل error في أي stage.
  - always {} ← دايمًا بتتنفذ سواء فشل أو نجح.

📌 معلومة مهمة جدًا:
Jenkins بيتعامل مع كل نوع من الحالات بشكل منفصل،
يعني ممكن تضيف خطوات مختلفة لكل واحدة حسب السيناريو.

🧠 مثال عملي من الواقع:
في شركات زي Giza Systems أو e& Egypt، دايمًا بتحط:

- في success : deploy notification على Slack أو Teams
- في failure : إرسال email بالتفاصيل.
- في always : مسح workspace لتفادي تكدس الملفات القديمة.

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo "Building the project..."
            }
        }
    }

    post {
        success {
            echo "✅ Build completed successfully!"
        }
        failure {
            echo "❌ Build failed!"
        }
        always {
            echo "🧹 Cleaning up workspace..."
            sh 'rm -rf *'
        }
    }
}
```

2️⃣ 🧩 في Scripted Pipeline ← finally Block

📖 الشرح خطوة بخطوة:

- هنا ما فيش post
  بدلها بنستخدم try / catch / finally زي لغة Groovy العادية.
- try ← بيحتوي على الكود الأساسي.
- catch (err) ← بيتنفذ لو حصل error
- finally ← بيتنفذ دائمًا في كل الحالات، سواء نجح أو فشل.

```
node {
    try {
        stage('Build') {
            echo "Building project..."
            sh 'exit 1' // هيفشل عمداً علشان نوضح الفرق
        }
    } catch (err) {
        echo "❌ Error detected: ${err}"
    } finally {
        echo "🧹 Cleaning up workspace..."
        sh 'rm -rf *'
    }
}
```

📌 معلومة مهمة:

finally في Scripted Pipeline = always في Declarative Pipeline لكن الفرق إنك تتحكم يدويًا في كل حاجة (مافيش جاهز زي success/failure)

💡 ملاحظات عملية (Best Practices - Real Cases - Tricks)

1. ✅ لما تستخدم Declarative Pipeline :
   - خليك دايمًا تستخدم post بدل try/catch لأن Jenkins هيتعامل مع الحالات تلقائيًا.
   - استخدم post { always { ... } } لتنظيف workspace
   - استخدم post { failure { ... } } علشان الإشعارات.

2. ✅ لما تستخدم Scripted Pipeline :
   - لازم تستخدم try/catch/finally بنفسك.
   - لو نسيت finally ، مش هيتنظف الـ workspace بعد الفشل.
   - لو عندك stages كتير، حاول تحيط كل Stage بـ try/catch صغير عشان تشوف فين المشكلة بالضبط.

3 ✅ Trick (من الواقع العملي) :

في بيئات الإنتاج (Production) ، بيستخدموا في Declarative Pipeline :

```
post {
    always {
        cleanWs()
    }
}
```

Rather Than

```
'* sh 'rm -rf
```

لأن cleanWs() هي دالة built-in من Jenkins plugin بتنضف الـ workspace بطريقة آمنة وذكية أكتر.

📋 الملخص الشامل

العنصر	Declarative Pipeline	Scripted Pipeline
طريقة التنفيذ بعد الانتهاء	`post { ... }`	`finally { ... }`
الأنواع المتاحة	`success` , `failure` , `always` , `unstable`	يدويًا باستخدام `try/catch`
سهولة الاستخدام	أسهل وبتدعم حالات كتير	مرنة بس لازم تكتبها بنفسك
أنسب استخدام	CI/CD Standard Jobs	Advanced Custom Logic
Cleanup الموصى به	`post { always { cleanWs() } }`	`finally { cleanWs() }`

reversing Stash In Declarative Pipeline

Explained in p.375

اكيد بتسأل هو بيشرحها تاني ليه ؟!
علشان فيه حتة زيادة في الاخر
وأكيد الحتة دي هاتجاوب علي سؤال
جه في دماغك حتي لو من بعيد

🧠 الفكرة الأساسية

- ❑ الجزء ده بيتكلم عن الـ Stash في Jenkins .. يعني إزاي نحفظ الملفات اللي بتتولد في Stage معينة ونستخدمها في Stage تانية حتى لو كانوا بيشتغلوا على Agents مختلفين،
- ❑ وكمان إزاي نحافظ عليها بعد إعادة تشغيل الـ Pipeline من مرحلة معينة (Restart from Stage)
- ❑ بمعني انك لما بتقسم الـ Pipeline بتاعك إلى stages ، أحيانًا:
  - Stage 1 بتولد ملف (زي quote.txt)
  - Stage 2 بتحتاج تستخدم الملف ده.
- ❑ لكن المشكلة إن كل Stage ممكن تشتغل على Agent مختلف (server مختلف أو container مختلف)
  وده معناه إن الملفات اللي اتولدت في Stage 1 مش موجودة على الجهاز بتاع Stage 2

👉 هنا بييجي دور الـ stash و unstash

⚙️ الشرح خطوة بخطوة

🔹 المرحلة الأولى : إنشاء Pipeline بسيط

هو عمل Pipeline فيها مرحلتين:

- o أول stage هاتعمل
  Generate a motivational quote
- o ثاني stage هاتعمل
  Display that motivational quote

لاحظ ان الـ 2 stages شغالين عالـ jenkins controller

Script ?

```
pipeline {
    agent any

    stages {
        stage('Generate Motivational Quote') {
            //  agent { label 'ubuntu-docker-jdk17-node20' }
            steps {
                script {
                    def motivationalQuotes = [
                        "Believe you can and you're halfway there.",
                        "The only way to do great work is to love what you do.",
                        "Don't watch the clock; do what it does. Keep going.",
                        "Success is not the key to happiness. Happiness is the key to success.",
                        "The only limit to our realization of tomorrow is our doubts of today."
                    ]

                    def randomIndex = new Random().nextInt(motivationalQuotes.size())
                    def quote = motivationalQuotes[randomIndex]
                    writeFile file: 'quote.txt', text: quote
                }

                sh '''
                    echo Generate Motivational Quote Success!
                    cat quote.txt
                '''
            }
        }

        stage('Display Motivational Quote') {
            steps {
                sh '''
                    cat quote.txt
                    rm quote.txt
                '''
            }
        }
    }
}
```

📌 النتيجة:

- الـ Pipeline بيشتغل عادي جدًا لأن كل المراحل شغالة على نفس الـ Agent (نفس الجهاز)
- الملف quote.txt موجود ومتقرى في المرحلتين.

🔹 المرحلة الثانية : تشغيل Stage على Agent مختلف

➢ هو قرر يشغل أول Stage على Agent اسمه : Ubuntu-agent
هايديله Label : Ubuntu Docker, JDK 17, Node 20

➢ فبقى الكود كده:

Script ?

```
pipeline {
    agent any

    stages {
        stage('Generate Motivational Quote') {
            agent { label 'ubuntu-docker-jdk17-node20' }
            steps {
                script {
                    def motivationalQuotes = [
                        "Believe you can and you're halfway there.",
                        "The only way to do great work is to love what you do.",
                        "Don't watch the clock; do what it does. Keep going.",
                        "Success is not the key to happiness. Happiness is the key to success.",
                        "The only limit to our realization of tomorrow is our doubts of today."
                    ]

                    def randomIndex = new Random().nextInt(motivationalQuotes.size())
                    def quote = motivationalQuotes[randomIndex]
                    writeFile file: 'quote.txt', text: quote
                }

                sh '''
                    echo Generate Motivational Quote Success!
                    cat quote.txt
                '''
            }
        }

        stage('Display Motivational Quote') {
            steps {
                sh '''
```

📍 المشكلة:

- الملف quote.txt اتولد على الـ Ubuntu Agent ،
  لكن المرحلة التانية شغالة على الـ Controller Agent
- بالتالي مش هتلاقي الملف ← الـ Stage 2 فشلت. ❌

```
Script ?
stages {
    stage('Generate Motivational Quote') {
        agent { label 'ubuntu-docker-jdk17-node20' }
        steps {
            script {
                def motivationalQuotes = [
                    "Believe you can and you're halfway there.",
                    "The only way to do great work is to love what you do.",
                    "Don't watch the clock; do what it does. Keep going.",
                    "Success is not the key to happiness. Happiness is the key to success.",
                    "The only limit to our realization of tomorrow is our doubts of today."
                ]

                def randomIndex = new Random().nextInt(motivationalQuotes.size())
                def quote = motivationalQuotes[randomIndex]
                writeFile file: 'quote.txt', text: quote
            }

            sh '''
                echo Generate Motivational Quote Success!
                cat quote.txt
            '''
            stash 'quote.txt'
        }
    }

    stage('Display Motivational Quote') {
        steps {
            unstash 'quote.txt'
            sh '''
                cat quote.txt
                rm quote.txt
            '''
        }
```

◆ المرحلة الثالثة : حل المشكلة باستخدام stash وunstash

Jenkins بيديك أوامر مخصوصة:

- Stash : تحفظ ملفات معينة من مرحلة.
- Unstash : تسترجع الملفات دي في مرحلة تانية.

✅ النتيجة:

- الـ file بيتنقل أوتوماتيك من الـ Ubuntu agent إلى الـ Controller عن طريق Jenkins Master
- فكده المراحل تشتغل تمام. 🎯

◆ المرحلة الرابعة :المشكلة الجديدة (Restart from Stage)

➢ Jenkins بيسمحلك تعمل : "Restart from Stage"

- يعني تشغل الـ Pipeline من منتصفه (مثلاً من Stage 2 بس) بدل ما تبدأ من الأول.

➢ المشكلة هنا 😬 :

لما تشغل Stage 2 بس،

فالـ Stage 1 مش هتتنفذ ← لأن الـ stash متطبقش أصلًا (لانك معملتش stage 1) ← وبالتالي Stage 2 هتفشل تاني.

◆ المرحلة الخامسة : الحل باستخدام preserveStashes()

➢ هانروح علي :

Pipeline syntax → Declarative Directive Generator → Sample Step → options: Options

- Add: preserveStashes
- ✅ Preserve stashes from completed builds
- Keep this many of the most recent builds' stashes : 5

(عاوز تطبق الت stash علي اخر كام build انت عملته)

```
Generate Declarative Directive
options {
  preserveStashes buildCount: 5
}
```

➢ وعشان نحافظ على الـ stashes القديمة من آخر كام Build ، بنستخدم الـ option ده في بداية الـ Pipeline :

```
Script ?
1  pipeline {
2      agent any
3
4  options {
5      preserveStashes buildCount: 5
6  }
7
8      stages {
9          stage('Generate Motivational Quote') {
10             agent { label 'ubuntu-docker-jdk17-node20' }
11             steps {
12                 script {
13                     def motivationalQuotes = [
14                         "Believe you can and you're halfway
```

📌 النتيجة النهائية:

- لما تشغل Stage 2 بس بعدين ← هتشتغل بنجاح ✅
- لأن Jenkins حافظ نسخة من الـ stash من الـ build اللي فات.

📋 الملخص الشامل

العنصر	المعنى	مثال
`stash`	يحفظ ملفات من مرحلة معينة مؤقتًا	`stash includes: 'quote.txt', name: ''quoteFile`
`unstash`	يسترجع الملفات دي في مرحلة تانية	`'unstash 'quoteFile`
`preserveStashes(buildCount : X)`	يحافظ على stashes من آخر X builds عشان تستخدمها لو عملت Restart من Stage	`options { preserveStashes(buildCount: 5) }`
المشكلة اللي بتحلها	اختلاف الـ Agents أو Restart من Stage بيخلي الملفات مش متاحة	✅ تتحل باستخدام stash + preserveStashes

---

🎯 الخلاصة:

- استخدم **stash/unstash لما تنقل ملفات بين stages على Agents مختلفة.**
- استخدم **preserveStashes() لما تحب تحتفظ بالملفات لو عملت Restart من Stage معينة.**
- **Jenkins بيتعامل مع الملفات دي مؤقتًا ويخزنها داخل الـ Master**

Scripted Pipeline Initalize

أولاً: الفكرة العامة

- ❑ في Jenkins عندنا نوعين من الـ Pipeline :
  1. Declarative Pipeline ← هو الأسهل، ليه Syntax ثابت ومنظم.
  2. Scripted Pipeline ← بيعتمد على Groovy script ، وبيديك حرية أكتر لكن أصعب شوية.
- ❑ الشرح دا بيوريك إزاي تكتب نفس السكريبت اللي كان Declarative (stage install dependencies مثلًا) لكن باستخدام Scripted Pipeline

---

الخطوات اللي اتعملت:

1 إنشاء فرع جديد (Branch)

- ❖ عمل git checkout -b pipeline/scripted
  عشان يكتب ملف جديد اسمه Jenkinsfile فيه السكريبت الجديد.

---

2 تجهيز ملفات Jenkinsfile

- غيّر اسم الملف القديم لـ jenkins.declarative (كـ نسخة احتياطية)
- عمل Jenkinsfile جديد فاضي عشان يكتب فيه الكود الـ Scripted

انا قسمت كل خطوة هو حولها بشكل منفصل للتبسيط والشرح .. بس في الاخر هتلاقي الـ script كامل

---

3 كتابة الـ Scripted Pipeline

```
pipeline {
    agent any
    stages {
        stage('Install') {
            steps { ... }
        }
    }
}
```

```
node {
    // code goes here
}
```

يعني بدل كلمة pipeline و agent any، بنستخدم node {}

---

4 تعريف الأدوات (Tools)

NodeJS | Jenkins plugin

كان عايز يستخدم Node.js tool ، فجاب المثال من Documentation :

Example of the use of buildwrapper Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                nodejs(nodeJSInstallationName: 'Node 6.x', configId: '<config-file-provider-id>') {
                    sh 'npm config ls'
                }
            }
        }
    }
}
```

Example of use tools in Jenkinsfile (Scripted Pipeline)

```
node {
    env.NODEJS_HOME = "${tool 'Node 6.x'}"
    // on linux / mac
    env.PATH="${env.NODEJS_HOME}/bin:${env.PATH}"
    // on windows
    env.PATH="${env.NODEJS_HOME};${env.PATH}"
    sh 'npm --version'
}
```

```
tools {
    nodejs 'nodejs-22-6-0'
}
```

```
env.NODEJS_HOME = "${tool 'nodejs-22-6-0'}"
env.PATH="${env.NODEJS_HOME}/bin:${env.PATH}"
```

Pipeline Syntax

5 إضافة خصائص (Properties)

```
options {
    disableResume()
    disableConcurrentBuilds abortPrevious: true
}
```

```
properties([
    disableConcurrentBuilds(abortPrevious: true)
])
```

---

6 عمل Stage للـ Checkout (علشان يسحب من SCM)

في Declarative Jenkins بيعمل Checkout تلقائي،
لكن في Scripted لازم نعملها يدوي:

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

---

7 عمل Stage لتثبيت Dependencies

```
stage('Installing Dependencies') {
    agent any
    options { timestamps() }
    steps {
        cache(maxCacheSize: 550, caches: [
            arbitraryFileCache(
                cacheName: 'npm-dependecny-cache
                cacheValidityDecidingFile: 'pack
                includes: '**/*',
                path: 'node_modules')
                ] ) {
                    sh 'node -v'
                    sh 'npm install --no-audit'
                    stash(includes: 'node_module
                }
    }
```

deleted

```
wrap([$class: 'TimestamperBuildWrapper']) {
    stage('Installing Dependencies') {
        cache(maxCacheSize: 550, caches: [
            arbitraryFileCache(
                cacheName: 'npm-dependecny-cache
                cacheValidityDecidingFile: 'pack
                includes: '**/*',
                path: 'node_modules')
                ] ) {
                    sh 'node -v'
                    sh 'npm install --no-audit'
                    stash(includes: 'node_module
                }
    }
}
```

- الفرق هنا إننا استخدمنا wrap عشان نضيف timestamps بدل options { timestamps() } اللي كنا بنستخدمها في Declarative
- مسحنا الـ agent any

---

8 النتيجة في Jenkins

بعد ما عمل Commit وPush، Jenkins عمل Detect للفرع الجديد،
وشغّل الـ pipeline بنجاح:

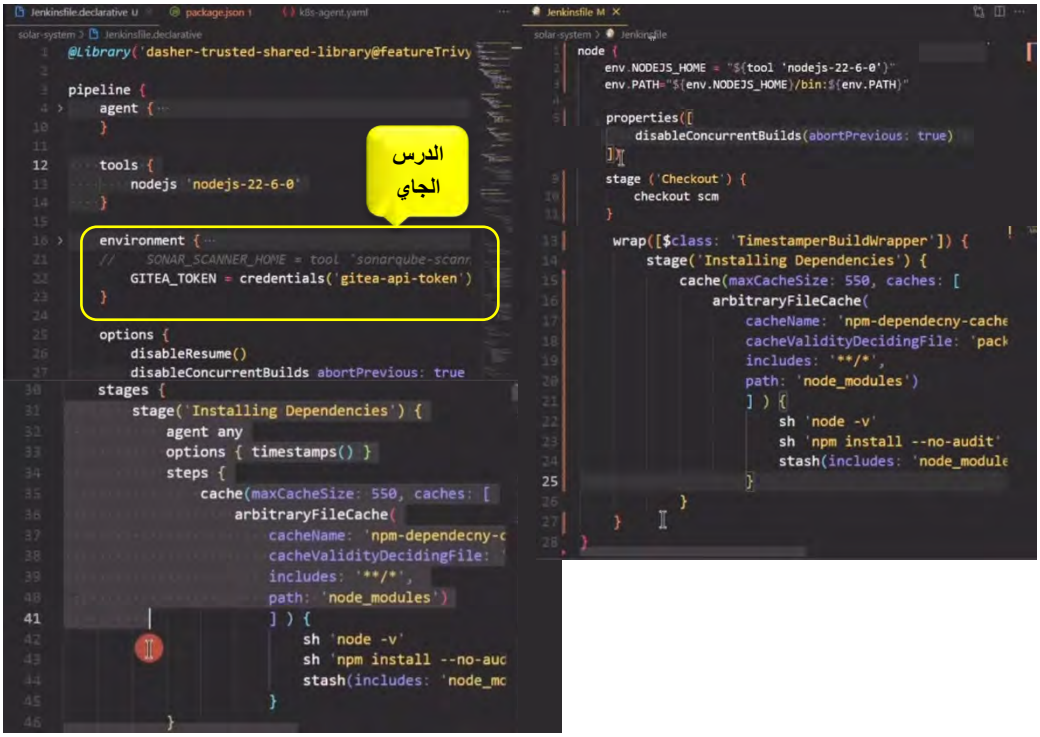
- Stage "Checkout" شغالة.
- Stage "Install Dependencies" فيها timestamps
- Cache اشتغل، و Node.js tool اتضاف صح.

---

9 جرب اختبار Disable Concurrent Builds

- لما شغّل Build مرتين، التاني تم إلغاؤه تلقائيًا،
- والـ log كتب إن التاني "superseded by build #3" وده دليل إن disableConcurrentBuilds اشتغلت صح.

✅ الخلاصة:



العنصر	في Declarative	في Scripted
البداية	`{} pipeline`	`{} node`
Agent	`agent any`	بيتضمن ضمنيًا داخل node
Options	`{} options`	`([])properties`
Steps	`{} steps`	مباشرة داخل stage
Timestamps	`options { timestamps() }`	`{} wrap([$class: 'TimestamperBuildWrapper'])`
Checkout	تلقائي	يدوي `checkout scm`

Scripted Pipeline-Static Agent

- الشرح ده هو تكملة للِّي قبلُه، وبيشرح إزاي تخلي الـ Scripted Pipeline اللي عملناه قبل كده يشتغل علي Static Agent بدل الـ Controller
- وكمان إزاي تضيف Unit Test stage وتتعامل مع MongoDB credentials

---

أولًا: تشغيل الـ Scripted Pipeline على Static Agent

- لما كتبنا الـ Scripted Pipeline قبل كده، كان بيشتغل على الـ Jenkins Controller node (يعني السيرفر الأساسي) وده حصل لأننا ماحددناش أي agent
- علشان نخليه يشتغل على Agent معين (Static Agent) ، بنكتب كده في أول السكريبت:
-

```
node('ubuntu-docker-jdk17-node20') {
    env.NODEJS_HOME = "${tool 'nodejs-22-6-0'}"
    env.PATH="${env.NODEJS_HOME}/bin:${env.PATH}"

    properties([ ...
```

- الجملة دي 'Ubuntu-Docker-JDK-17-node20' هي label للـ Agent اللي عايز تشغّل عليه البيلد، وبتتكتب بين single quotes
- وبكده كل الـ stages في السكريبت هتشتغل على الـ Agent ده بدل الـ controller

---

ثانيًا: إضافة Stage جديدة للـ Unit Test

- كان عندنا Stage واحدة اسمها "Install Dependencies" ، ودلوقت عاوز يضيف Stage جديدة اسمها "Unit Test"

للتذكير ← شكل الـ stage('Unit Test') اللي كانت في الـ Declarative ←

```
stage('Unit Testing') {
    parallel {
        stage('NodeJS 18') {
            options { retry(2) }
            steps {
                sh 'node -v'
                unstash 'solar-system-node-m
                sh 'npm test'
            }
        }
```

- لكن هنا فيه مشكلة 👇

الـ npm test محتاج MongoDB connection (URI + username + password) وده بيتطلب إننا نمرر credentials و environment variables جوه الـ pipeline

- ولاحظ اننا في الـ Declarative كنا مباصينها في الـ environment بالشكل ده ↓

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83j
    MONGO_DB_CREDS = credentials('mongo-db-crede
    MONGO_USERNAME = credentials('mongo-db-usern
    MONGO_PASSWORD = credentials('mongo-db-passw
//    SONAR_SCANNER_HOME = tool 'sonarqube-scann
    GITEA_TOKEN = credentials('gitea-api-token')
}
```

ثالثًا: الحل || تعريف المتغيرات الخاصة بـ MongoDB

- في الـ Declarative Pipeline كنا بنكتب مثلًا:

```
environment {
    MONGO_URI = "mongodb+srv://supercluster.d83j
```

→

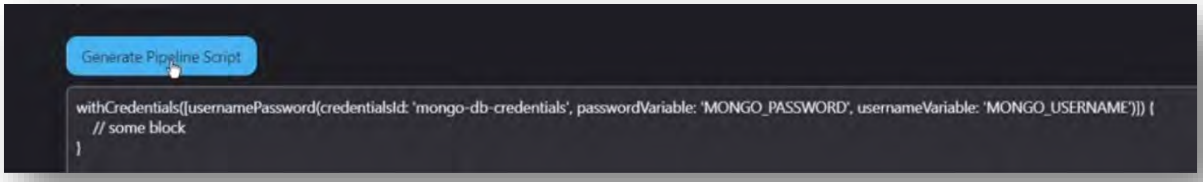
```
env.MONGO_URI = "mongodb+srv://supercluster.d83j
```

🧩 رابعًا: استخدام withCredentials في Scripted Pipeline

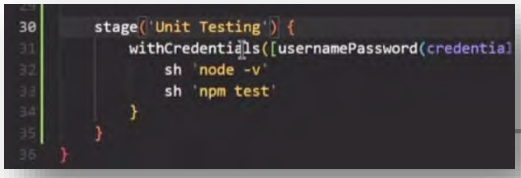
- في الـ Declarative ممكن نستخدم environment عادي، لكن في Scripted لازم نستخدم step اسمه withCredentials
- هاتروح علي

Pipeline Syntax Generator → snippet generator → sample step → withCredentials: Bind credentials to vars

- Credentials: Username and Password (seperated)
- Username Variable: MONGO_USERNAME
- Password Variable: MONGO_OASSWORD
- Credentials: (add credentials you made earlier)

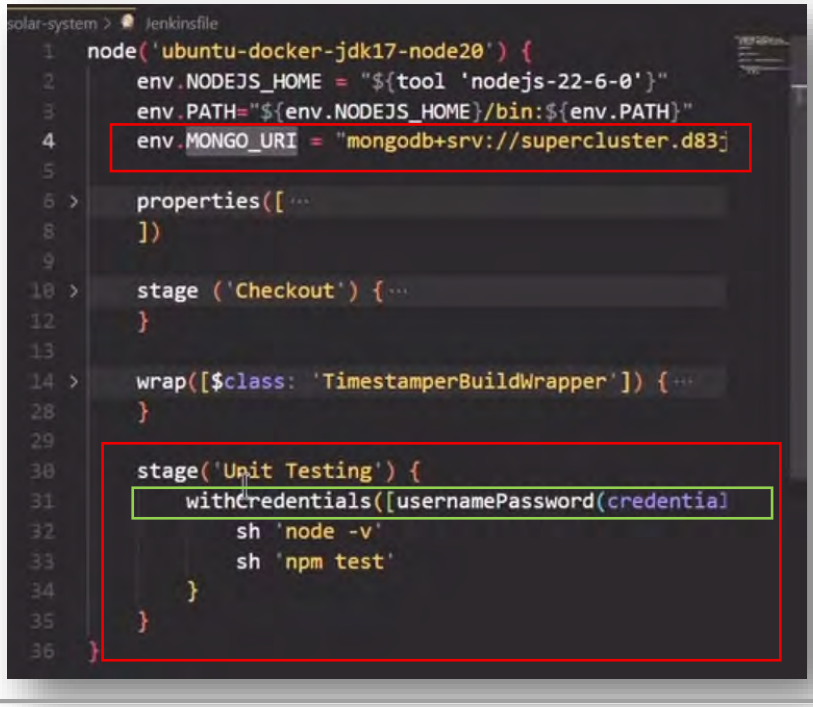


- credentialsId = اسم الـ Credential اللي خزّناه في Jenkins
- mongo_username و mongo_password = أسماء المتغيرات اللي هتستخدمها جوه السكريبت.
- جوه الـ withCredentials بنحط أي أوامر (زي npm test) بتحتاج توصل للداتا بيز.



- بعدها هايروح يضيفخ في الـ stage("Unit Test")

💡 النتيجة النهائية للـ Scripted Pipeline



✅ في النهاية:

- كل الـ stages اتنفذت على الـ Ubuntu Agent (Static Agent)
- الـ MongoDB credentials اتجابت صح عن طريق withCredentials
- الـ npm test اشتغل واتصل بـ MongoDB وطلع ناجح.
- البيلد كله خلص بنجاح.

Scripted Pipeline-K8S Agent

أولًا: المقدمة – الفكرة العامة

- ❑ الجزء ده بقى بيشرح أصعب وأهم نقطة في Jenkins pipelines :
  - إزاي تشغّل Scripted Pipeline جوه Kubernetes Pods بدل ما تشتغل على static agent أو على controller
- ❑ ده مهم جدًا في بيئات الإنتاج الكبيرة اللي كل حاجة فيها بتشتغل داخل containers على Kubernetes
- ❑ في الشرح اللي فات، كنا شغالين على Scripted Pipeline بيشتغل على static agent (Ubuntu)
  لكن في الشركات الكبيرة، بنحب نخلي كل الـ stages تشتغل داخل Kubernetes Pods علشان:
  - نقدر نعزل كل مرحلة في بيئة خاصة بيها.
  - نحافظ على scalability وسهولة الـcleanup.
  - نقدر نتحكم في الـ containers (الإصدار – الصلاحيات – الـ tools)

```
pipeline {
    agent {
        kubernetes {
            cloud 'dasher-prod-k8s-us-east'
            yamlFile 'k8s-agent.yaml'
            defaultContainer 'node-18'
        }
    }
```

- ❑ في Declarative Pipeline كنا بنكتب:
  وده كان سهل جدًا.
- ❑ لكن في Scripted Pipeline ← بما ان مفيش agent block
  يبقى لازم نعملها بطريقة تانية:
  وهي إننا نستخدم حاجة اسمها **Pod Templates** 👉 من خلال Kubernetes plugin في Jenkins

---

ثانيًا: شرح المفهوم Pod Templates و Container Templates

Kubernetes | Jenkins plugin

- 🔹 Pod Template = تعريف pod كامل (اسم، namespace، containers، إلخ)
- 🔹 Container Template = تعريف كل container داخل الـ pod (اسم - image – command – args)

ممكن تعرفهم:

- يا إما في إعدادات Jenkins نفسها (Cloud Level) (اللي هو dasher-prod-k8s-us-east – pod template)
- يا إما جوه الـ Jenkinsfile (Pipeline Level)

🚨 هنا هو بيبدأ بشرح الفكرتين وبيختار إنه يـ "يكتبهم يدويًا داخل الـ Jenkinsfile" علشان التجربة تبقى أوضح.

ثالثًا: إنشاء Pod Template داخل الـ Jenkinsfile

هاتروح علي Pipeline Syntax Generator → snippet generator → sample step → podTemplate
ومن هناك، بدأ يملأ القيم:

- Cloud : اسم الـ Cloud اللي Jenkins متصل بيها (موجود في إعدادات Jenkins) (dasher-prod-k8s-us-east)
- Label : اسم pod label اللي Jenkins هيستخدمها لتحديد الـ node
  (هو هنا هايديها اسم nodejs-pod)
- Usage: only build jobs with label expressions

➢ containerTemplate : يحدد الـ image اللي هنشغل عليه الكود.

- Name: node-18
- Docker Image: node:18-alpine
- Command to run: sleep (علشان يخلو الـ container شغال طول الوقت)
- Arguments to pass the command: 9999999
  - ttyEnabled : يخلي الـ container interactive ✅
- Run in privilege mode ✅

Generate Pipeline Script

```
podTemplate(cloud: 'dasher-prod-k8s-us-east', containers: [containerTemplate(args: '9999999', command: 'sleep', image: 'node:18-alpine', livenessProbe:
containerLivenessProbe(execArgs: '', failureThreshold: 0, initialDelaySeconds: 0, periodSeconds: 0, successThreshold: 0, timeoutSeconds: 0), name: 'node-18', privileged: true,
resourceLimitCpu: '', resourceLimitEphemeralStorage: '', resourceLimitMemory: '', resourceRequestCpu: '', resourceRequestEphemeralStorage: '', resourceRequestMemory: '',
ttyEnabled: true)], label: 'nodejs-pod') {
  // some block
```

بعدا ماعدل وشال اللي مش محتاجه في الـ use case دي

Raw YAML for the Pod ?

```
podTemplate(
cloud: 'dasher-prod-k8s-us-east', label: 'nodejs-pod', containers: [
containerTemplate(args: '9999999', command: 'sleep', image: 'node:18-alpine', name: 'node-18', privileged: true, ttyEnabled: true)]) {
  // some block
}
```

Show raw yaml in console ?

🧠 **Best Practice** : خليك بتستخدم images خفيفة زي alpine علشان توفر وقت ومساحة.

🧩 مكانه في الـ jenkinsfile :

بيتم وضع الكود اللي ولدته في أول الـ jenkinsfile بتاعك :

```
podTemplate(
    cloud: 'dasher-prod-k8s-us-east',
    label: 'nodejs-pod',
    containers: [
        containerTemplate(
            args: '9999999',
            command: 'sleep',
            image: 'node:18-alpine',
            name: 'node-18',
            privileged: true,
            ttyEnabled: true)])
{
    node('ubuntu-docker-jdk17-node20') {
        env.NODEJS_HOME = "${tool 'nodejs-22-6-0'}"
        env.PATH="${env.NODEJS_HOME}/bin:${env.PATH}"
        env.MONGO_URI = "mongodb+srv://supercluster.d83jj.mongodb.net/superData"

        properties([...
        ])

        stage ('Checkout') {...
        }

        wrap([$class: 'TimestamperBuildWrapper']) {...
        }

        stage('Unit Testing') {
            withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVariable: 'MONGO_PA
                sh 'node -v'
                sh 'npm test'
            }
        }
    }
```

دا في حالة لو انت عاوز تعرف Multiple Container فعاتبقي بالشكل ده وبقية الكود زي ماهو

هانشوفهم حكايتهم في الصفحة الجاية علشان فيه 2 errors هايحصلوا بسببهم فخليك فاكر

```
        containerTemplate(
            args: '9999999',
            command: 'sleep',
            image: 'node:18-alpine',
            name: 'node-18',
            privileged: true,
            ttyEnabled: true),
        containerTemplate(
            args: '9999999',
            command: 'sleep',
            image: 'node:19-alpine',
            name: 'node-19',
            privileged: true,
            ttyEnabled: true)])
```

🔹 وبكدا احنا عندنا 2 container template & 1 pod tempalt

🧩 رابعًا: ربط الـ Stage بالـ Pod tempate والـ Container tempalates

علشان تشغل stage معينة جوه pod/container ، لازم تعمل:

1. node(label) ← علشان تختار الـ pod
2. container(name) ← علشان تختار الـ container داخل الـ pod

وكل دا هايكون جوا الـ stage اللي عاوز تربطها

مثال:

```
stage('Unit Testing') {
    node('nodejs-pod') {
        container('node-18') {
            withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwor
                sh 'node -v'
                sh 'npm test'
            }
        }
    }
}
```

خامسًا: مشاكل ظهرت أثناء التنفيذ (Real Case)

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

Only accessible to the (Ubuntu-Docker) static Virtual Machine
وبالتالي فالـ stages اللي بتشتغل في الـ static agent هي بس اللي عندها قابلية الوصول الي الفايلات اللي في الريبوو .. وبالتبعية فالـ stage("unit testing") مش قادرة توصل للريبو لأنها شغالة على الـ kubernetes pod

الحل : بالتالي هاتحتاج تضيف checkout option في stage('Unit Testing')
لأن دي scripted pipeline .. انما لو كانت Declarative فكان زمانها اتعملت

```
wrap([$class: 'TimestamperBuildWrapper']) {
    stage('Installing Dependencies') {
        cache(maxCacheSize: 550, caches: [
            arbitraryFileCache(
                cacheName: 'npm-dependecny-cache',
                cacheValidityDecidingFile: 'package-lock.json',
                includes: '**/*',
                path: 'node_modules')
                ] ) {
                    sh 'node -v'
                    sh 'npm install --no-audit'
                    stash(includes: 'node_modules/', name: 'solar-system-node-modules')
                }
```

Running on
node ('ubuntu-docker-jdk17-node20')

While the stage('Unit Testing') is running on the kubernetes pod
والـ stage التانية محتاجة ملف الـ node_module اللي بيطلع من الـ stage دي

الحل النهائي

الحل : استخدم stash و unstash
- نعمل stash بعد install dependencies
- ونستخدم unstash قبل npm test

وبكدا نكون استرجعنا الملفات اللي كانت متخزنة من stage تانية (install dependencies)

```
wrap([$class: 'TimestamperBuildWrapper']) {
    stage('Installing Dependencies') {
        cache(maxCacheSize: 550, caches: [
            arbitraryFileCache(
                cacheName: 'npm-dependecny-cache',
                cacheValidityDecidingFile: 'package-lock.json',
                includes: '**/*',
                path: 'node_modules')
                ] ) {
                    sh 'node -v'
                    sh 'npm install --no-audit'
                    stash(includes: 'node_modules/', name: 'solar-system-node-modules')
                }
        }
}

stage('Unit Testing') {
    node('nodejs-pod') {
        container('node-18') {
            checkout scm
            unstash 'solar-system-node-modules'
            withCredentials([usernamePassword(credentialsId: 'mongo-db-credentials', passwordVa
                sh 'node -v'
                sh 'npm test'
            }
```

Best Practices + Real Use Cases

- استخدم Labels واضحة لكل Pod Template (زي nodejs-test-pod)
- حافظ على صور خفيفة (Alpine)
- دايمًا أعمل checkout لكل بيئة مختلفة.
- استخدم stash/unstash لمشاركة الملفات بين stages

Real Case :

في الشركات الكبيرة (زي GitLab أو E& أو Orange) بيستخدموا Kubernetes pods في Jenkins عشان كل pipeline step تشتغل في container مخصص (nodejs للـ frontend – maven للـ backend – terraform للـ infra)

سادسًا: ملخص شامل

العنصر	Declarative Pipeline	Scripted Pipeline
تعريف Pod	داخل `agent { kubernetes { ... } }`	باستخدام `podTemplate()`
تعريف Container	في ملف YAML خارجي	داخل `containerTemplate()`
ربط Stage بـ Container	تلقائي	لازم تستخدم `container` + `node()`
مشاركة البيانات	تلقائي	لازم تستخدم `unstash` / `stash`
السهولة	أسهل جدًا	مرن لكن أعقد

Best Practice for Scripted Pipelines

🎯 المقدمة: الفكرة العامة

- ❑ الجزء دا بيتكلم عن أفضل الممارسات (Best Practices) في Jenkins Pipelines ، سواء كانت Declarative أو Scripted Pipelines
- ❑ الهدف هو إنك تعرف إزاي تكتب Pipeline نظيف، سريع، سهل الصيانة، وما يحملش Jenkins Controller فوق طاقته.

⚙️ الشرح التفصيلي خطوة بخطوة

🔹 1. Effective Log Management

➢ الفكرة:

بدل ما تخلي Jenkins يطبع كل اللوجز (logs) جوه الكونسول، خليه يكتبها في ملف على الـ Agent نفسه، وبعدين تعمل compress & archive كـ artifact

➢ الأمر المستخدم:

- o sh 'your_command > build.log'
- o archiveArtifacts artifacts: 'build.log.gz', fingerprint: true

➢ الفائدة:

- أداء Jenkins يبقى أسرع (مش بيغرق في كمية لوجز ضخمة).
- تحليل اللوجز بعدين أسهل.

🧠 Best Practice :

خلي Jenkins يحفظ اللوجز فقط في الـ Artifact ، واستخدم أدوات خارجية لتحليلها (زي ELK أو Splunk لو عندك setup كبير)

🔹 2. Leverage Jenkins Plugins

➢ الفكرة:

استغل الـ plugins الموجودة بدل ما تعيد اختراع العجلة.

➢ أمثلة:

- Source Control ← git plugin
- Artifact Management ← artifactDeployer plugin
- Deployment ← ssh agent plugin, ansible plugin
- System Automation ← pipeline-utility-steps plugin

🧠 Trick عملي:

قبل ما تكتب سكريبت جديد، ابحث في الـ Plugin Manager .. في الغالب فيه Plugin جاهز يحل مشكلتك.

🔹 3. Keep Pipeline Concise (Steps < 300)

➢ الفكرة:

ما تعملش مراحل كتير جدًا داخل Jenkinsfile ، لو عندك عمليات متتابعة كتير، اجمعها في سكريبت خارجي.

➢ أمر مثال : sh './scripts/build_and_test.sh'

➢ الفائدة:

- سرعة تنفيذ أعلى.
- Jenkinsfile أنظف وأسهل قراءة.
- تقدر تعيد استخدام السكريبت في مشاريع تانية.

🔹 4. Use Command Line Tools

- الفكرة :

لو عندك عمليات معقدة (زي parsing JSON أو التعامل مع APIs)
استخدم أدوات CLI بدل ما تعملها داخل Groovy

- أمثلة:

  - jq '.key' file.json
  - xmlint --xpath "//element" file.xml

🧠 Real Use Case :

بدل ما تكتب جمل Groovy معقدة استخدم :

- curl أو wget ← للتعامل مع APIs
- jq ← لتحليل JSON
- xmllint ← لتحليل XML
- grep, awk, sed ← لمعالجة النصوص

🔹 5. Offload Heavy Work to Agents

- الفكرة:

خلي العمليات التقيلة (زي build أو data processing) تشتغل على Agents، مش على Jenkins Controller

- كود مثال:

```
node('linux-agent') {
    sh 'mvn clean install'
}
```

- الفائدة:

  - الـ Controller يفضل خفيف وسريع.
  - Agents ممكن تكون scaled horizontally حسب الحمل.

🔹 6. Minimize Pipeline Code

- الفكرة:

خلي Jenkinsfile هو "المنسق" بين الأدوات فقط. وما تكتبش logic أو حسابات كبيرة فيه.

- مثال سيئ: ❌

```
def list = []
for (i in 1..1000) {
   list.add(i * 2)
}
```

- مثال صحيح: ✅

```
sh './process_data.sh'
```

🧠 Best Practice :

خلي Jenkins هو الـ "glue" بين الأدوات فقط (Git + Docker + Kubernetes)

💡 نصائح خاصة بالـ Scripted Pipeline

🚫 1. Avoid Complex Groovy Syntax

- استخدم Groovy بسيط، ومتقعدش تستخدم Features معقدة زي closures أو meta-programming
- الهدف إن أي حد يقرأ الكود يفهمه بسهولة.

🚫 2. Don't Call Internal Jenkins APIs

- متستخدمش أوامر جوهرية (Internal APIs) للـ Jenkins من الـ pipeline
- لو محتاج تتعامل مع Jenkins internals ← اعمل Custom Pipeline Step أو Plugin مخصص.

⚙️ 3. Parse XML/JSON with CLI Tools

- متستخدمش Groovy في parsing للبيانات.
- استخدم أدوات زي:
  - jq .data response.json
  - xmllint --xpath "//node" file.xml

💡 Reason :

ده بيقلل استهلاك الذاكرة ويحافظ على أداء السيرفر.

🚫 4. Avoid Direct Network or Disk Operations

- متحاولش تعمل عمليات زي:
  - URL.openConnection()
  - new File().write()
- واستخدم Jenkins steps زي:
  - sh 'curl -O https://example.com/file.zip'
  - writeFile file: 'data.txt', text: 'hello'

💡 Reason :

العمليات دي لو حصل فيها خطأ ممكن توقف الـ controller أو تسبب تسريب في الذاكرة.

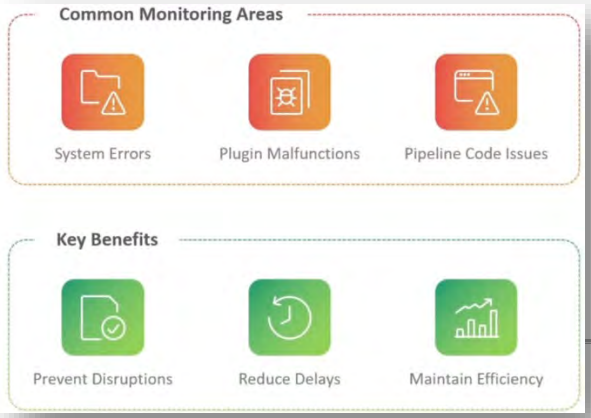
🧩 الملخص الشامل

المحور	التوصية
Logs	خزن الـ logs كملفات واحتفظ بيها كـ artifacts
Plugins	استخدم الـ plugins الجاهزة بدل ما تعيد اختراع العجلة
Pipeline Size	خليك تحت 300 خطوة واستخدم سكربتات خارجية
CLI Tools	استخدم أدوات CLI بدل Groovy المعقد
Agents	شغّل العمليات الثقيلة على agents مش على controller
Script Simplicity	خليك في Groovy بسيط وسهل
No Direct Jenkins Calls	متتعاملش مع Jenkins APIs مباشرة
No Heavy Logic Inside	خلي Jenkinsfile منظم وبسيط

Jenkins Supervision

المقدمة – فكرة عامة

- الموضوع بيتكلم عن مراقبة (Monitoring & Auditing) Jenkins
- الفكرة إن أي نظام CI/CD (زي Jenkins) لازم يكون فيه متابعة مستمرة علشان تقدر:
  - تكتشف المشاكل قبل ما تحصل.
  - تعرف أداء السيرفر وحال الـ pipelines
  - تتابع أنشطة المستخدمين علشان الأمان والمساءلة (Auditing)
- بالتالي، المراقبة هنا بتنقسم لثلاث محاور رئيسية:
  - Logs (السجلات) : علشان تشوف كل حاجة بتحصل في Jenkins
  - Monitoring Tools : علشان تتابع أداء النظام والـ resources
  - Auditing Plugins : علشان تعرف مين عمل إيه وإمتى .



الشرح التفصيلي خطوة بخطوة

1. Logs في Jenkins

- الفكرة:

الـ Logs بتديك صورة كاملة عن نشاط Jenkins – سواء مشاكل في plugins ، أو أخطاء في الكود، أو مشاكل سيستم.

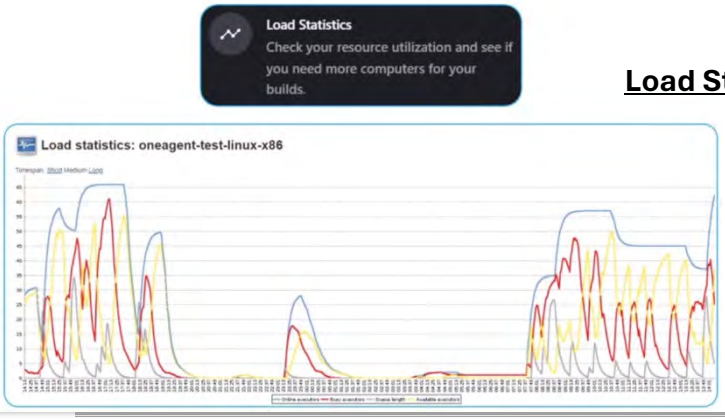
- أماكن الـ Logs حسب طريقة التشغيل:

طريقة التشغيل	مكان الـ Logs	ملاحظات
تشغيل يدوي باستخدام `java -jar jenkins.war`	بتظهر في الـ console مباشرة	يفضل تحويلها إلى ملف `log.` لتسهيل التحليل
نظام Linux `$ cat /var/log/jenkins/jenkins.log`	`var/log/jenkins/`	المكان الافتراضي
Debian-based `$ vi /etc/default/jenkins`	إعداد من `etc/default/jenkins/`	ممكن تعدل الـ path من هنا
RedHat-based `$ vi /etc/sysconfig/jenkins`	إعداد من `etc/sysconfig/jenkins/`	نفس الفكرة
Windows Installer `$ cat %JENKINS_HOME%/jenkins.out`	داخل `JENKINS_HOME`	ويمكن تغييره من `jenkins.xml`
Docker Container `$ docker logs containerId`	استخدم الأمر:	

Best Practice :

- كمان ممكن من الواجهة: Manage Jenkins → System Log تشوف الـ logs مباشرة من الـ Dashboard
- استخدم ← log rotation علشان تمنع امتلاء القرص.
- فعّل external log forwarding ← لو عندك ELK أو Splunk
- دايمًا احفظ نسخة من الـ logs في أرشيف build artifacts لتحليل المشاكل.

2. Monitoring Jenkins Performance 📈



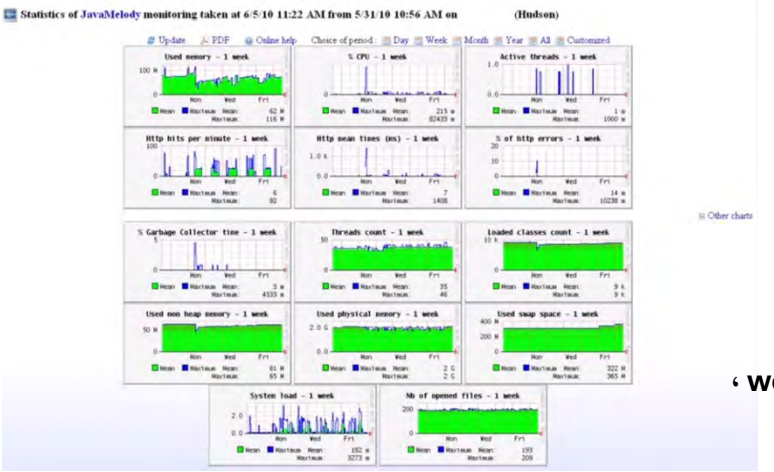
- الأدوات المدمجة (Built-in Tools) مثل : Load Statistics Page

بتعرض رسم بياني فيه 4 مؤشرات:

- عدد executors المتاحة (ready)
- عدد executors اللي شغالة فعليًا.
- عدد الـ jobs اللي مستنية دورها.
- الحمل العام على السيرفر (Overall Load)

📍 تقدر تستخدمها من : Manage Jenkins → Load Statistics

- Plugins مهمة للـ Monitoring :



i. Monitoring Plugin (Java Melody)

يعرض رسومات عن:

- CPU / Memory Usage
- Response Time
- HTTP Sessions
- Errors & Garbage Collection

ii. Disk Usage Plugin

يوضح مساحة التخزين المستخدمة لكل job أو workspace ،
ويعمل تتبع للتغير في المساحة مع الوقت.

iii. Build Monitor Plugin

يعرض حالة الـ jobs (ناجحة / فاشلة) .. ويقدر يوضح مين المسؤول عن الـ failed build (Culprit)

Best Practices 💡

- راقب باستمرار CPU/RAM/Disk
- استخدم تنبيهات (alerts) لما الموارد تتخطى thresholds
- فعّل الـ Build Monitor في الشاشات العامة للفريق (Dashboard TV)

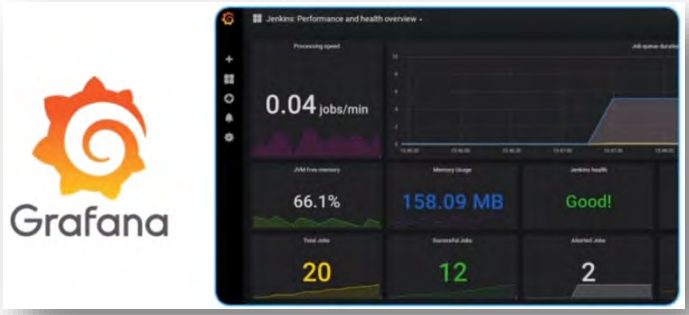
- أدوات خارجية متقدمة (External Monitoring)

i. Datadog / New Relic Plugins

تقدر تبعت الـ metrics ليهم تلقائيًا .

ii. Prometheus + Grafana

- حل مفتوح المصدر لمتابعة Jenkins بصورة تفاعلية جدًا:
- مثال بسيط لجمع metrics : curl http://<jenkins-url>/prometheus/
  - بعد كده تربط Grafana بـ Prometheus لعمل dashboard



🔐 3. Auditing Jenkins (تتبع أنشطة المستخدمين)

- ليه مهم؟

علشان تعرف مين غيّر إعدادات حساسة أو حذف job أو نفّذ build بدون إذن.

- أهم Plugins :

i. Audit Trail Plugin

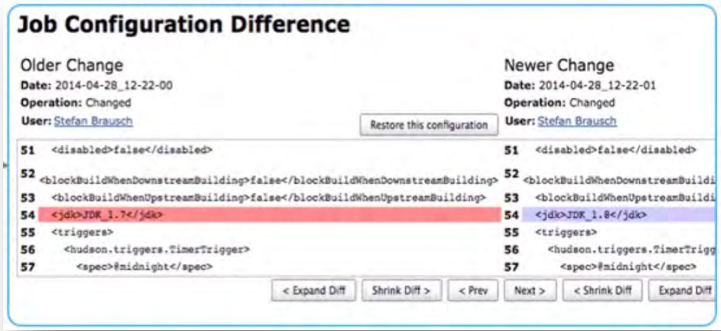
- بيضيف قسم جديد في الإعدادات: Manage Jenkins → Configure System → Audit Trail
- بيسجل كل actions حصلت (مين عمل إيه ومتى).
- بيوفر 4 أنواع من الـ loggers :

النوع	الوظيفة
File Logger	يخزن logs في ملفات دوّارة (rotating files)
Syslog Logger	يرسلها لسيرفر syslog مركزي
Console Logger	للـ debugging فقط
Elasticsearch Logger	يرسلها لسيرفر Elasticsearch لتحليل متقدم

➕ في الشركات الكبيرة، دايمًا بيفضّلوا الـ Elasticsearch Logger مع Kibana dashboards

ii. Job Config History Plugin

- بيخزن نسخ من كل تعديل في الـ jobs والـ folders
- يحفظ نسخ من ملفات config.xml
- واجهته تتيح مقارنة التعديلات بين نسختين (diff view)
- تقدر ترجع إعداد قديم بضغطة زر.



⚠️ ملحوظة:

بيحتفظ فقط بتغييرات الإعدادات، مش بتنفيذ الـ jobs نفسها.

💡 Best Practices :

- فعّل الـ plugins 2 معًا:

Audit Trail (لمتابعة مين عمل إيه) & Job Config History (لمتابعة التغييرات نفسها)

- خزّن سجلات المراجعة في مكان مركزي آمن.
- راقب الأنشطة الإدارية بانتظام.

🧭 الملخص الشامل

العنصر	الوظيفة	الأدوات/الأوامر	أفضل الممارسات
Logs	متابعة نشاط Jenkins وأخطاء النظام	`var/log/jenkins`, `docker logs` /, `System Log`	استخدم rotation، وصدّر logs لـ ELK
Performance Monitoring	متابعة الأداء واستخدام الموارد	Load Stats, Monitoring Plugin, Grafana	راقب CPU/RAM باستمرار، استخدم تنبيهات
Auditing	تتبع المستخدمين والتعديلات	Audit Trail, Job Config History	فعّل الاثنين معًا، خزّن السجلات مركزيًا

Monitoring with Prometheus & Grafana

🎯 الهدف:

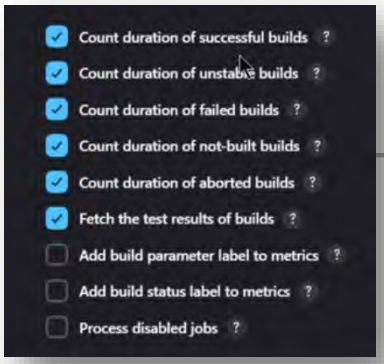
- ❑ إننا نعمل Monitoring Dashboard لـ Jenkins باستخدام Grafana & Prometheus علشان نتابع حالته، زي:
  - عدد الـ jobs
  - استخدام الـ CPU و الـ Memory
  - حالة الـ Nodes
  - عدد الـ plugins ....إلخ.

🧩 الخطوات الأساسية:

1 تثبيت Plugin في Jenkins :

- ➢ روح على Manage Jenkins → Plugins → Available plugins .. دور على **Prometheus metrics plugin** وثبّته.
- ➢ بعد التثبيت، اعمل Restart لـ Jenkins
- 💡 المهمة الأساسية للـ Plugin :

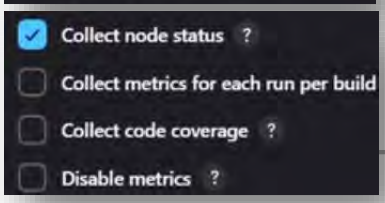
بيفتح Endpoint على /prometheus فيه كل الـ metrics اللي Prometheus يقدر يجمعها.

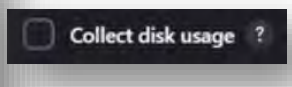


---

2 إعداد Jenkins Prometheus Plugin :

- ➢ بعد ما تفتح Jenkins تاني، روح لـ Manage Jenkins → System → Prometheus
- ➢ غيّر الإعدادات:
  - Collect period : من 120 ثانية لـ 10 ثواني (علشان التحديث أسرع).
  - اختياري : فعّل أو عطّل أنواع الـ metrics اللي محتاجها .
  - access the metrics on : /Prometheus/Endpoint





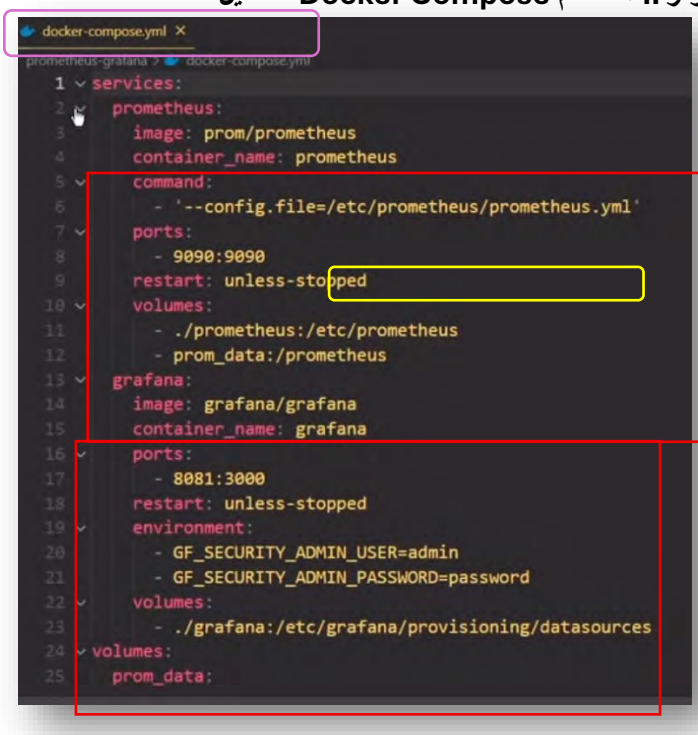
---

3 إعداد Prometheus نفسه:

- ➢ ازاي هاتسطب الـ PrometheusServer ؟؟
- بدل مايركب الـ Prometheus or Grafana يدويا عالجهاز او السيرفر .. استخدم Docker Compose لتشغيل Prometheus و Grafana بسهولة كـ containers.
- وضاف جواه الـ prometheus.yaml لتفاصيل الـ targets

📄 مثال: docker-compose.yml ←

- ملف **prometheus.yml** (فيه تفاصيل الـ Targets اللي Prometheus هيسحب منها البيانات)
- Ports: 9090 (واجهة الـ prometheus)
- Volumes: prom_data (بيحفظ بيانات الـ Prometheus علشان متضيعش بعد الـ restar)
- environment variables: as in image (بيبعت للـ container متغيرين وهما اسم المستخدم والباسورد)



```yaml
services:
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
    ports:
      - 9090:9090
    restart: unless-stopped
    volumes:
      - ./prometheus:/etc/prometheus
      - prom_data:/prometheus
  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - 8081:3000
    restart: unless-stopped
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=password
    volumes:
      - ./grafana:/etc/grafana/provisioning/datasources
volumes:
  prom_data:
```

نبذة عن الـ docker-compose في حالة لو متعرفوش لسا
ببساطة docker-compose.yaml هو أداة لتجميع وتشغيل أكثر من container مترابطين سوا (Stack) .. وهايكون مسئول عن :

- تشغيلهم مع بعض.
- توصيلهم في نفس الـ network
- عمل volume لحفظ البيانات.
- إعادة التشغيل التلقائى لو وقعوا

مثال: Prometheus.yaml ←

```yaml
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

scrape_configs:
- job_name: prometheus
  metrics_path: /metrics
  scheme: http
  static_configs:
  - targets:
    - localhost:9090

- job_name: jenkins
  metrics_path: <<replace-me>> #ideally it would be /prometheus/
  scheme: http        #if target endpoint schema is https, change schema https
  static_configs:
  - targets:
    - <<replace-me>> #jenkins url and port if applicable
      - 64.227.187.25:8080 #jenkins url and port if applicable
```

هاتشغل بقا كل الليلة دي من CLI بـ :

docker-compose up -d

```
root@jenkins-controller-1 in prometheus-grafana on main [!] on (us-east-2)
❯ docker compose up -d
[+] Running 4/4
 ✓ Network prometheus-grafana_default      Created
 ✓ Volume "prometheus-grafana_prom_data"   Created
 ✓ Container prometheus                    Started
 ✓ Container grafana                       Started
```

5 التحقق من Prometheus :

- افتح المتصفح على **http://<vm_ip>:9090**
  - روح لـ Status → Targets
  - لازم تلاقي Jenkins ظاهر كـ "up" ✅
- روح علي الـ Graph وجرب Expressions زي :
  - [jenkins_job_count] - [jenkins_plugins_active] - [jenkins_plugins_with_update]

6 إعداد Grafana :

- افتح **http://<vm_ip>:8081**
  - دخل (user: admin / pass: password)
  - روح علي connections → Add Data Source → Prometheus
    واكتب URL : http://prometheus:9090

7 استيراد Dashboard جاهز لـ Jenkins :

Grafana dashboards | Grafana Labs

- روح لموقع Grafana Dashboards
- دور على Dashboard رقم 9964 (Jenkins performance and health overview)
- ارجع لـ Grafana ← new ← new dashboard ← Import Dashboard ← اكتب 9964
  - اختار الـ data source (Prometheus) واضغط Import

8 تجربة البيانات:

- ارجع لـ Jenkins واعمل Job جديدة اسمها مثلًا **monitor-jenkins**
  - جوّاها Pipeline بسيط :

```
Script ?
node {
    stage('Echo Message') {
        sh 'sleep 2s'
    }
}
node('ubuntu-docker-jdk17-node28') {
    stage('Echo Message2') {
        sh 'echo 1s'
    }
}
```

  - شغّلها كذا مرة (15-20 مرة).
- دلوقتي لما ترجع لـ Grafana Dashboard ، هتبدأ تشوف البيانات بتتحدث:
  - Memory usage - Queue rate - Job success count - CPU usage - Executors health

ملاحظات مهمة:

- كل ما تشغّل Jobs أكتر، البيانات تبقى أدق.
- ممكن تعدّل في الرسوم (تخليها Gauge بدل Graph مثلًا)
- الـ Prometheus Plugin لازم يفضل مفعّل علشان يحدّث البيانات.

Log Recorder

الفكرة العامة

- الجزء دا بيشرح إزاي نقدر نعمل Logger داخل Jenkins علشان نحل مشاكل الـ plugins أو نعرف تفاصيل أكتر وقت الـ Debugging
- الفكرة ببساطة : بدل ما Jenkins يورينا Error بسيط في الـ UI ، بنقدر نشوف كل التفاصيل اللي بتحصل في الخلفية زي الـ API Calls والـ Responses .. وطبعا ده بيساعدنا نحدد السبب الحقيقي لأي مشكلة.

الجزء الأول: قبل ما تبدأ (Understanding the Concept)

Configure Loggers for Jenkins and CloudBees

- Jenkins بيقدم 5 طرق تقدر بيها تعمل Logging :
  - من خلال واجهة Jenkins (UI) ← ودي أسهل وأكتر طريقة مستخدمة. (دي اللي هاتتشرح)
  - تشغيل Groovy script وقت الـ initialization
  - إنشاء ملف logging levels وتخزينه داخل Jenkins Home
  - إضافة Logger يدويًا من الـ file system
  - ضبط default levels من الـ UI (بس بتتمسح بعد الـ restart)

ملاحظة مهمة جدًا:

- لما تغيّر Log Level لأي package وتخليه مثلًا (ALL أو FINEST) .. Jenkins ← هيبدأ يكتب تفاصيل كتير جدًا.
  - ده ممكن يزود الـ I/O Load على الـ controller ، فلذلك يفضل استخدامه بس وقت الـ Debug ، وبعدها ارجع للوضع العادي أو احذف الـ Logger

---

الجزء الثاني: نبدأ نعمل Logger من الـ UI

- الخطوات من داخل Jenkins :
  - روح لـ Manage Jenkins → System Log
    - هتلاقي Logs عامة (زي system warnings و errors)
  - بس إحنا محتاجين Logs تفصيلية أكتر (debug level)
    - فاضغط على "Add new log recorder"
  - هيسألك عن الاسم ← اكتبه مثلاً: K8s Logs

علشان في المثال هنا إحنا بنحل مشكلة في Kubernetes Cloud Plugin

---

الجزء الثالث: ضبط تفاصيل الـ Logger

- بعد ما تسميه:
  - اضغط على "Add Logger"
  - هتظهرلك خانتين:
    - Loger : لتحديد الـ Package Name (ابحث عن الباكدج اللي فيها المشكلة، زي في المثال ده: io.fabric8.kubernetes .. ودي الباكدج الخاصة بالـ Kubernetes Plugin)
    - Log Level : All (وده علشان تشوف كل التفاصيل HTTP Calls, Responses, Exceptions ... إلخ)
- بعد كده اضغط Save

🧠 الجزء الرابع : نستخدم الـ Logger فعليًا

❖ دلوقتي هنجرب حاجة عملية :

- ➢ افتح إعدادات Kubernetes Cloud في Jenkins واضغط على "Test Connection"
- ➢ لو الإعدادات فيها مشكلة ، Jenkins هيعرض Error بسيط في الواجهة، لكن لما تروح تفتح الـ Logger اللي أنشأته وتعمل Refresh، هتلاقي معلومات كتير جدًا زي:
  - o 🌐 الـ HTTP Request اللي اتبعت للـ API
  - o 📜 الـ JSON Response اللي رجع من الـ Kubernetes
  - o 🔐 الـ Token المستخدم في المصادقة.
  - o ⚙️ الـ Headers اللي استخدمها Jenkins
  - o ❌ الأكواد اللي رجعت (زي 403 Forbidden أو 200 OK)

ودي كلها حاجات مش بتظهر في الواجهة العادية.

🧩 الجزء الخامس : مثال عملي واضح

- ➢ لو الإعدادات فيها خطأ (مثلاً Token غلط) .. هتشوف في الـ Logger سطر زي:

403 Forbidden - Invalid credentials

- ➢ ولو الإعدادات صح، هتلاقي Jenkins كتب:

Connected to Kubernetes, version 1.23

- ➢ وفي الخلفية هتشوف كمان الـ HTTP Request كامل في شكل JSON

🚨 يعني Jenkins بيطلب الـ **/api/v1/namespaces** وبيستقبل Response ، وده كله بيتسجل عندك في الـ Logger

---

🧹 الجزء السادس: تنظيف بعد الـ Debug

❖ بعد ما تخلص من حل المشكلة:

- ارجع إلى System Logs
- امسح محتوى الـ Log (Clear log)
- احذف الـ Log Recorder بالكامل (Delete log recorder)

✳️ دي خطوة مهمة جدًا لأن الـ Logger لو فضل شغال هيفضل يولد Logs ضخمة وده ممكن يبطأ الـ Controller أو يستهلك مساحة التخزين.

🧠 نصائح و Best Practices :

- ➢ استخدم الـ UI method لأنها الأسهل والأوضح.
- ➢ دايمًا خليك محدد في اختيار الـ Package (مثلاً docker, git, kubernetes)
- ➢ متفعّلش Logging على كل حاجة مرة واحدة.
- ➢ بعد الـ Debug ، ارجع تاني للوضع الافتراضي (INFO أو WARNING)
- ➢ لو محتاج Debug مستمر، اعتمد على External log management tools (زي ELK أو Splunk)

🧩 الجزء الأخير Real-World Example :

❖ في بيئة Production ، لو Jenkins بيعمل Deploy على Kubernetes وبيفشل من غير ما يوضح السبب، فممكن تفعل Logger للـ plugin علشان تشوف مثلًا:

- هل Jenkins قدر يعمل Auth مع الـAPI ؟
- هل الـ Namespace موجود فعلًا؟
- هل الـ Pod Creation فشل في مرحلة معينة؟

كل دي تفاصيل بتبان بس في الـ Logger مش في الواجهة.

📋 الملخص النهائي

العنصر	التفاصيل
🎯 الهدف	مراقبة تفاصيل Jenkins وقت الـ Debug
🧩 الأداة	System Log → Add New Log Recorder
🧠 الباكدج المستخدمة	io.fabric8.kubernetes
⚙️ مستوى الـ Log	ALL (مؤقتًا فقط)
⚠️ التحذير	ممكن يبطأ الـ Controller لو فضل مفتوح
🧹 بعد الانتهاء	امسح أو احذف الـ Logger
💡 استخدامات عملية	Kubernetes, Docker, Git, أو أي Plugin

Install & Configure Audit Trail

🎯 الفكرة العامة

- ❑ الجزء ده بيتكلم عن إزاي تتابع وتراقب الأنشطة اللي بتحصل جوه Jenkins – يعني مين عمل إيه، إمتى، ومنين.
- ❑ ده بيتحقق عن طريق Plugin اسمها Audit Trail ، واللي تعتبر مهمة جدًا في بيئات العمل اللي فيها أكتر من مهندس بيعدلوا في Jenkins
- ❑ الفكرة إنك تخلي Jenkins يسجل كل حاجة بتحصل في Logs تقدر ترجع لها في أي وقت.

⚙️ الشرح خطوة بخطوة :

Audit Trail | Jenkins plugin

🧩 الخطوة 1: تثبيت Audit Trail Plugin

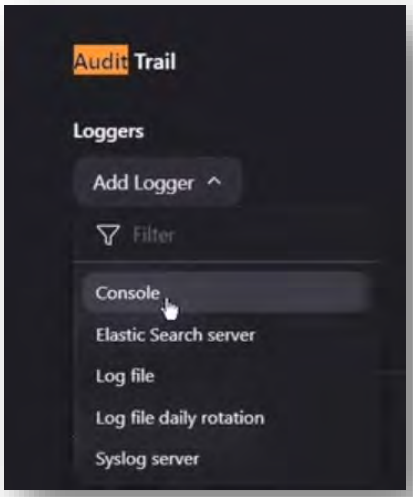
- بدأ يروح على Manage Jenkins → Manage Plugins → Available
- كتب في البحث Audit Trail
- بعدين عمل Install

💡 الملاحظة:

الـ Audit Trail plugin دي بتخليك تسجل كل العمليات اللي بتحصل زي (تعديل job ، تشغيل build ، أو تغيير في إعدادات Jenkins)

🚤 ودي بتفيدك لما تحب تعرف "مين لمس إيه" لو حصل مشكلة أو تعديل غير متوقع.

🧩 الخطوة 2 : فتح إعدادات Audit Trail



بعد ما خلص التثبيت،

- راح على Manage Jenkins → System Configuration
- كتب في البحث "Audit" .. فظهرت له صفحة فيها إعدادات Audit Trail

هنا بقى عندك 5 طرق مختلفة لتخزين الـ Logs :

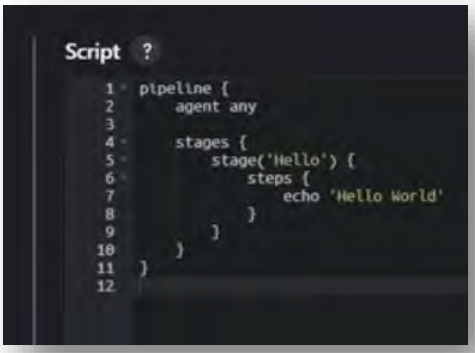
1. Console
2. Elasticsearch Server
3. Log File
4. Log File Daily Rotation
5. Syslog Server

هو اختار Log File Daily Rotation

- o يعني Jenkins هيكتب اللوجات في ملف بيتجدد كل يوم، علشان مايبقاش ضخم جدًا.
- حدد المكان اللي هيتحفظ فيه الملف: /var/log/jenkins/custom_audit-%g.log
- o وكمان استخدم pattern %G علشان لو اللوج بيتلفّ يوميًا (rotate) ، Jenkins يرقم الملفات القديمة بدل ما يكتب فوقها (نفس فكرة log rotate في لينكس)

بعد كده حفظ الإعدادات وقالك إن كل حاجة تمام ✅

🧩 الخطوة 4 : اختبار الإعدادات



قالك دلوقتي نختبر هل الـ Audit Trail شغال فعلاً ولا لأ.

- راح على Dashboard ← Job اسمها Monitor Jenkins
- دخل على Configure
- غيّر السكريبت اللي جواها (مثلاً بدّل حاجة معينة بـ "echo "Hello World")
- عمل Apply → Save → Build Now

الخطوة 5: التحقق من اللوجات

- فتح التيرمنال وكتب:
  - cd /var/log/jenkins
  - ls

```
root@jenkins-controller-1 in /var/log/jenkins on (us-east-2)
❯ ls
custom-audit-0.log-2024-11-10  custom-audit-0.log-2024-11-10.lck
```

- لقى فعلاً ملفين : واحد للوج الأساسي، والتاني نسخة متلففة (rotated)

- بعد كده قرأ الملف : cat custom_audit.log
  - ولقى فيه السطور دي مثلًا:

```
root@jenkins-controller-1 in /var/log/jenkins on (us-east-2)
❯ cat custom-audit-0.log-2024-11-10
Nov 10, 2024 2:29:30,662 PM/job/monitor-jenkins/configSubmit by siddharth from 124.123.186.17
Nov 10, 2024 2:29:34,672 PMjob/monitor-jenkins/ #29 Started by user siddharth, Parameters:[]
Nov 10, 2024 2:29:39,049 PMmonitor-jenkins #29 Started by user siddharth, Parameters:[] on node #unknown# started at 2024-11-10T14:29:34Z completed
ted: SUCCESS
```

- يعني Jenkins فعلاً سجل مين عدّل، ومين شغّل الـ job ، وامتى، ومن أي IP
  وده دليل إن البلجن شغالة 100%.

الخطوة 6: تعديل تاني للتأكيد

- قالك خلينا نعمل تعديل إضافي للتأكد.
  - رجع على System Configuration تاني وغيّر قيمة اسمها : Log File Count = 10
  - وبعدها عمل Save
  - رجع يقرأ اللوج تاني، ولما عمل cat custom_audit.log
    شاف إن السطر الجديد تسجل برضو:

```
root@jenkins-controller-1 in /var/log/jenkins on (us-east-2)
❯ cat custom-audit-0.log-2024-11-10
Nov 10, 2024 2:29:30,662 PM/job/monitor-jenkins/configSubmit by siddharth from 124.123.186.17
Nov 10, 2024 2:29:34,672 PMjob/monitor-jenkins/ #29 Started by user siddharth, Parameters:[]
Nov 10, 2024 2:29:39,049 PMmonitor-jenkins #29 Started by user siddharth, Parameters:[] on node #unknown# started at 2024-11-10T14:29:34Z completed
ted: SUCCESS
Nov 10, 2024 2:30:45,042 PM/manage/configSubmit by siddharth from 124.123.186.17
Nov 10, 2024 2:30:50,232 PM/manage/configSubmit by siddharth from 124.123.186.17
```

- يعني كل تعديل جديد بيتسجل لحظيًا.

الملاحظات المهمة

- كل حاجة بتتسجل باسم المستخدم اللي عاملها (User ID) + Timestamp + IP
- ممكن تبعت نفس اللوج ده لـ Elasticsearch أو Syslog Server لو عندك نظام مركزي لمراقبة الأحداث.
- الأفضل تخلي اللوجات دي دايمًا في مكان آمن ومحدش عنده صلاحيات يعدّل عليها.

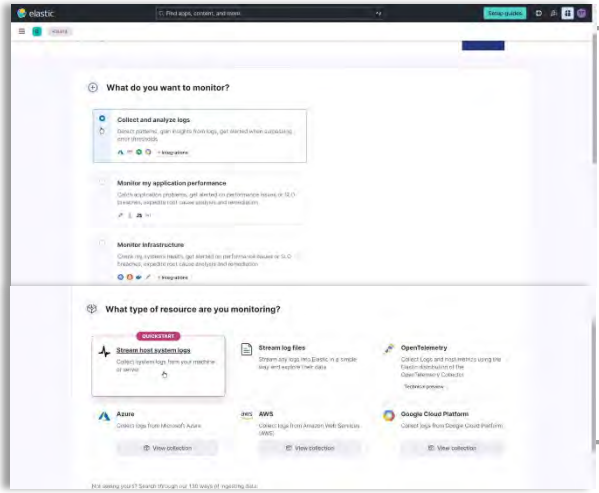
الخلاصة

- الهدف : مراقبة من بيغير إيه جوه Jenkins
- الأداة Audit Trail Plugin
- النتيجة : كل تعديل أو تنفيذ بيتسجل تلقائيًا في ملف لوج، تقدر ترجعه وقت التحقيق أو المراجعة.
- نقطة القوة : تقلل الأخطاء وتزود الـ accountability في الفريق.

Forward Audit Logs to External Server like Elastic Cloud

المقدمة

- ❑ في الدرس السابق، كنا عملنا إعداد لـ Audit Trail Plugin في Jenkins علشان نسجل كل الأحداث (زي مين شغّل Job ، إمتى، وإيه اللي حصل)
- ❑ لكن التسجيل ده كان بيحصل محليًا على الـ Jenkins Controller .. يعني جوه السيرفر نفسه فقط.
- ❑ المشكلة؟
  - لو السيرفر ده حصل فيه Crash أو اتقفل، ممكن نفقد اللوجات المهمة.
  - الحل؟ نبعثها (Forward) لسيرفر خارجي لحفظها ومراقبتها في الوقت الحقيقي.
- ❑ في الجزء ده، هاستخدم Elastic Cloud (Elasticsearch SaaS) علشان يجمع اللوجات دي من Jenkins ويحللها.

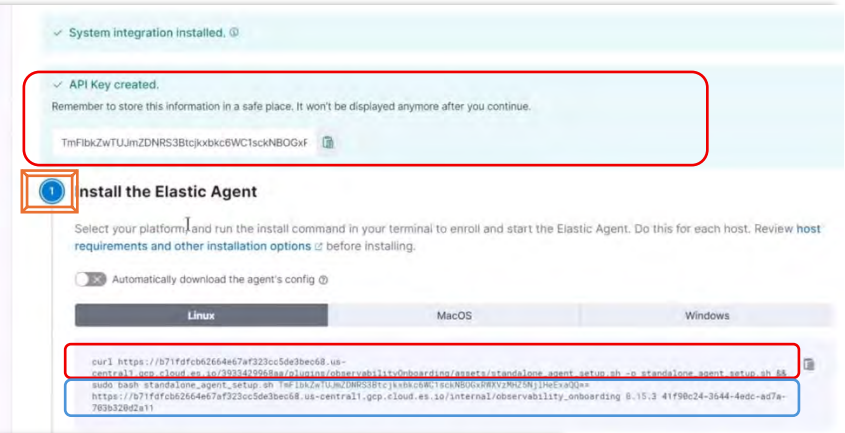


الشرح خطوة بخطوة

الخطوة 1: الدخول على Elastic Cloud

- فتح حساب Trial على Elastic Cloud
- دخل على قسم Observability Data
- هناك اختار: Collect and Analyze Logs → Host system logs
  وده علشان يضيف Agent هيجمع اللوجات من الجهاز اللي عليه Jenkins (في حالتنا، Jenkins Controller).

الخطوة 2: تنزيل الـ Elastic Agent



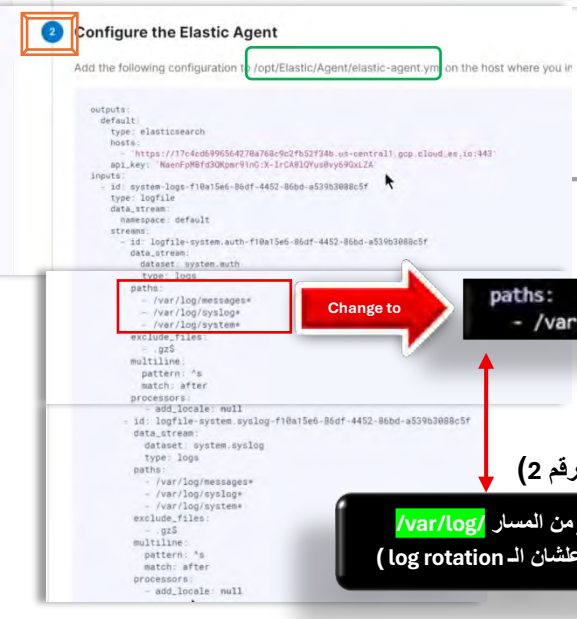
بعد ما تختار نوع اللوجز، Elastic بيظهرلك:

- (Command) جاهز لتنزيل الـ Agent
- API Key خاصة بالحساب.
- Endpoint
  (رابط الخادم السحابي بتاع Elastic اللي هنبعتله الـ logs)
- ملف : /opt/elastic-agent/elastic-agent.yaml
  (بيخزن فيه كل الإعدادات)

شرح الأمر:

- curl ← لتحميل سكريبت التثبيت من الـ CLI
- بعدين(sudo bash ..) ← بيشغّل السكريبت ويمررله الـ API Key + (endpoint) رابط السيرفر.

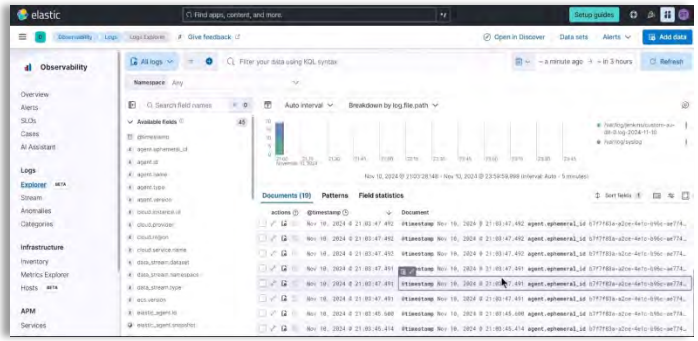
السكريبت بينزل الـ Agent ، يفك ضغطه، ويبدأ يشتغل كخدمة (Service)



الخطوة 3: إعداد الـ YAML لتوجيه اللوجات

- دلوقتي الـ Agent جاهز، فبقى محتاج نعرفه :
  - "منين تاخد اللوجز اللي هتبعتها؟"
- ففتح الملف: vi /opt/elastic-agent/elastic-agent.yaml
  - حذف كل اللي فيه .
  - وبعدين نسخ الـ configuration yaml file (اللي كان أوريدي متولد في الخطوة رقم 2) في الملف اللي فتحه وعدل الـ Path

يعني : خُد أي ملف يبدأ بـ jenkins-custom من المسار /var/log/
(الـ * معناها "أي ملف يبدأ بالاسم ده" — علشان الـ log rotation)



🚀 الخطوة 4: حفظ الملف وتشغيل الخدمة

بعد ما حفظ الإعدادات:

- الـ Agent بدأ يقرأ اللوجات تلقائيًا.
- وبعد ثوانٍ، ظهرت اللوجات في Elastic Cloud Dashboard

✨ هناك تقدر تشوف كل التفاصيل:

- مين شغّل الجوب.
- وقت التشغيل .
- الحالة (نجح أو فشل).
- نفس المعلومات اللي فيJenkins controller ، لكنها دلوقتي في مكان آمن ومركزي.

🧠 ملاحظات عملية (Best Practices & Real Use Cases)

1. 🔹 الأمان مهم جدًا:

الـ API Key دي بتدي صلاحية إرسال الداتا، فخليها سرية جدًا.

2. 🔹 Real use case :

الشركات الكبيرة زي البنوك أو شركات الاتصالات بتستخدم Elastic أو Splunk علشان:

- تتابع أي تغييرات أو محاولات Logins في Jenkins
- تكتشف أي تصرف مش طبيعي (مثلاً تنفيذ Job خارج وقت العمل)

3. 🔹 الـ Hash Trick :

استخدام `/var/log/jenkins-custom*` مفيد لما Jenkins بيعمل Log Rotation (يحوّل اللوجات القديمة لملفات جديدة كل فترة)

4. 🔹 Alternatives :

ممكن تستخدم بدل Elastic :

- Splunk
- Graylog
- Fluentd + Grafana Loki

كلها حلول لمراقبة اللوجات.

📃 الملخص النهائي

🎯 الهدف كان:

نرسل Jenkins audit logs إلى Elastic Cloud بدل ما تبقى محلية.

🧩 الخطوات الأساسية:

1. إعداد حساب Elastic Cloud
2. تحميل وتثبيت الـ Elastic Agent على السيرفر اللي فيه Jenkins
3. تعديل ملف الإعداد `/opt/elastic-agent/elastic-agent.yaml` وتحديد مسار اللوجز.
4. حفظ وتشغيل الخدمة.
5. التأكد من ظهور اللوجات في الـ Dashboard

🧠 النتيجة:

الـ Jenkins logs بقيت بتتجمع أوتوماتيكيًا في Elastic – تقدر تحللها، تبحث فيها، وتبني Dashboards جميلة منها.

Groovy Sandbox and In-Process Script Approval

المقدّمة: الفكرة العامة

- ❑ الجزء ده بيتكلم عن Groovy Sandbox و In-Process Script Approval في Jenkins
- ❑ الاتنين دول يعتبروا جزء أساسي من الـ Security System بتاع Jenkins ، والهدف إنك تقدر تشغّل سكريبتات Groovy بأمان، من غير ما تسمح لأي كود خبيث يأثر على الـ Jenkins environment نفسها.
- ❑ الفكرة ببساطة:
  - ❖ Jenkins بيسمحلك تكتب سكريبتات Groovy داخل الـ pipeline ، بس مش أي كود يتنفّذ كده وخلاص.
  - ❖ فيه sandbox بيحمي النظام، ولو السكريبت حاول يستخدم method مش مصرح بيها، Jenkins بيوقفه ويستنى approval من الـ admin (sandbox is something like : هي زريبة ياحبيبي)

أولاً: الـ Groovy Sandbox .. يعني إيه؟

- لما تفتح Jenkins وتكتب سكريبت pipeline ، هتلاقي تحت علامة تبويب الـ Script فيه checkbox مكتوب عليه: "Use Groovy Sandbox" ✅
- دي مش مجرد checkbox شكلها حلو، دي فعلاً ميزة أمنية قوية جدًا.

💡 الفكرة

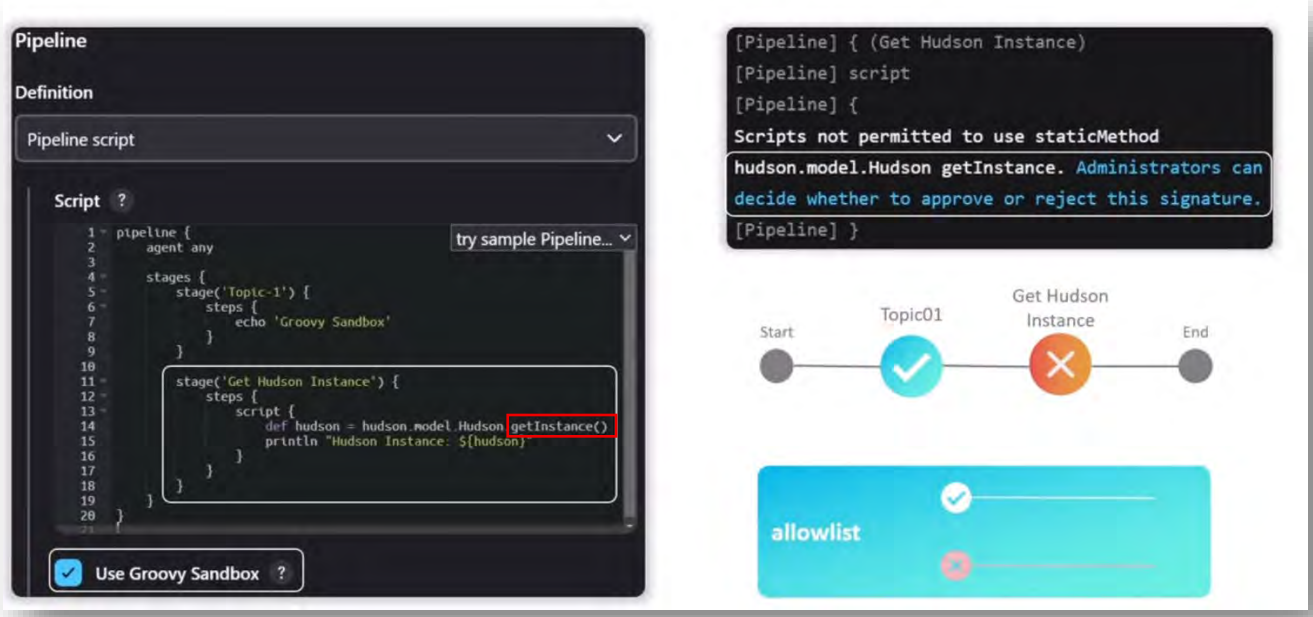
- ❖ الـ sandbox هو بيئة تنفيذ محدودة الصلاحيات .. يعني الكود بيتنفّذ في مساحة آمنة مش ممكن تأذي Jenkins
- ❖ لما تكون الـ Sandbox مفعّلة:
  - أي حد يقدر يجرب سكريبت جديد.
  - Jenkins بيشيّك على كل method call وأي access لأي object
  - لو الـ operation دي مسموح بيها ← السكريبت يكمل.
  - لو مش مسموح بيها ← Jenkins يوقف السكريبت فورًا.

🧱 وده كله بيتم بفضل Plugin اسمه: Script Security Plugin .. وده بيتنزل تلقائي أثناء الـ setup الأولي لـ Jenkins

---

ثانيًا: لما يحصل Violation – (يعني سكريبت حاول يعمل حاجة ممنوعة)

- ❖ مثال عملي 👇
  - لو سكريبت حاول يستخدم method زي: Jenkins.getInstance()
    - o ده معناه إنه بيحاول ياخد instance من الـ Jenkins class نفسه — ودي حاجة مش مسموح بيها افتراضيًا.
- ❖ النتيجة : Jenkins يوقف السكريبت ويكتب في الـ logs إن الكود حاول يستخدم method غير معتمدة.



👮‍♂️ ثالثًا: هنا بقى بيظهر دور الـ Admin

❖ Jenkins مش هيمنع السكريبت ويسيبك كده!

- هو بيضيف الـ method اللي حصل معاها violation في قائمة انتظار approval
- ومن هنا الـ admin عنده 3 اختيارات لما يروح للصفحة دي:
  - من 🧭 : Manage Jenkins → Security → In-Process Script Approval
  - الاختيارات الثلاثة 📃:

1. ✅ Approve
   - معناها: اسمح للسكريبت دايمًا يستخدم الـ method دي.
   - ⚠️ مخاطرة كبيرة لو الكود فيه أي حاجة ممكن تستغلها جهة خبيثة.
2. ❌ Deny
   - معناها: امنعها للأبد، ومفيش سكريبت يستخدمها.
3. 🛡️ Approve assuming permission check
   - دي أأمن واحدة.
   - معناها: اسمح، بس بشرط اللي بيشغل السكريبت يكون عنده صلاحية أصلاً (permission كافية)
   - يعني Jenkins هيتأكد إن الشخص ده Admin أو عنده الحق إنه يستخدم الميثود دي.

🧰 رابعًا: بعد الـ Approval

❖ بمجرد ما الـ Admin يعمل"Approve" ، Jenkins بيضيف الـ method دي إلى قائمة اسمها:
Approved Signatures List

❖ أي سكريبت بعد كده يستخدم نفس الميثود ← هيتنفذ عادي من غير error

🖍️ بس لو الـ Admin شالها من الـ list ، أو عطّل الـ sandbox ← هترجع المشاكل تظهر تاني.

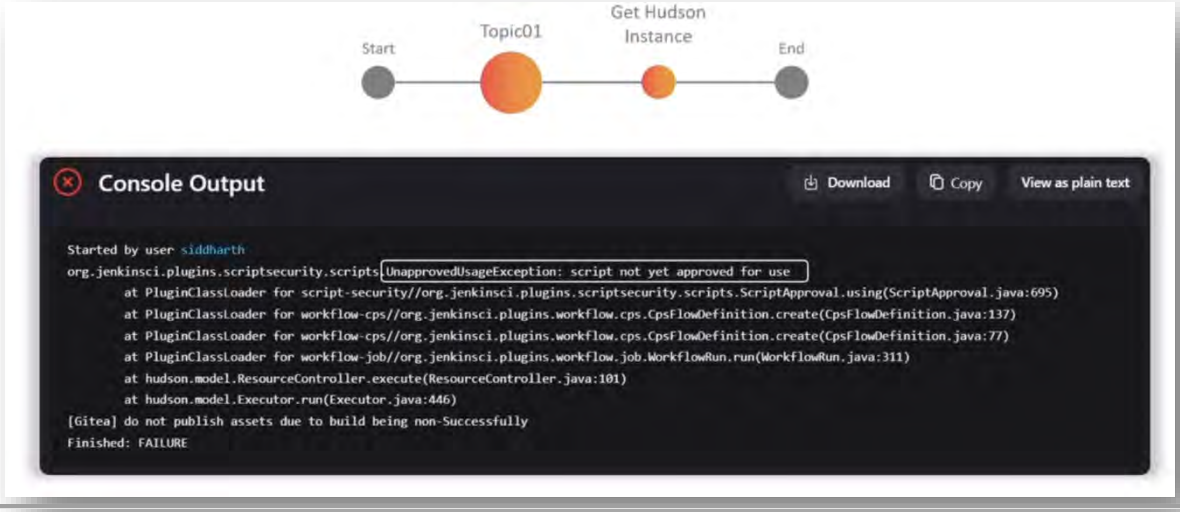
🧩 خامسًا: لما الـ Sandbox يتعطّل

لو الـ checkbox بتاع Groovy Sandbox مش متفعّل (unchecked)

- لو الـ user مش admin ← هيظهر له تنبيه يقول إن السكريبت محتاج موافقة admin
- لو الـ user is admin ← يقدر يوافق من نفس الصفحة على طول.

لكن لو ماوافقش ← أي build يحاول يشغل السكريبت ده هيفشل برسالة : "Script not yet approved for use".

- يعني ولا أي stage من الـ pipeline هيتنفذ.



🧠 سادسًا Best Practices & Tips :

1. ما تعطّلش الـ sandbox أبدًا إلا لو مضطر جدًا .لان الخطر كبير جدًا لأنك بتفتح الباب لأي كود يعمل حاجة في الـ Jenkins core
2. في Declarative pipelines الوضع أأمن أصلًا، لأنك مش بتكتب Groovy حر، فمش دايمًا بتحتاج Sandbox قوي زي الـ Scripted pipelines
3. في Scripted pipelines لازم تكون فاهم إنت بتعمل إيه، لأن كل method call ممكن تحتاج موافقة جديدة — خصوصًا لما تشتغل على مشاريع جديدة.
4. حاول تقلل الـ approvals للي فعلاً لازم تستخدمه فقط.
5. أي method بتعدّل بيانات محفوظة (زي setProperty() أو execute()) الأفضل ماتتحطش في الـ allow list
6. Methods (زي getInstance() أو getSomething()) عادة harmless ، وممكن يَسمح بها لأنها بتجيب بيانات فقط ومش بتعدّلها.

📄 الخلاصة

- Jenkins بيستخدم Groovy Sandbox كـ "درع أمني" يمنع السكريبتات الغير موثوقة من إنها تغيّر أو تبوّظ حاجة.
- أي سكريبت بيحاول يعمل حاجة مش معتمدة ← بيتوقف لحد ما الـ admin يراجعها في In-Process Script Approval
- الـ admin يقدر يوافق، يرفض، أو يوافق بشرط الـ permissions
- كل حاجة approved بتتسجل في Approved Signature List وبتتسمح تلقائي بعد كده.
- أهم حاجة : ما تطفّيش الـsandbox ، لأنه هو اللي واقف بينك وبين disaster أمني في Jenkins

Demo Groovy Sandbox and In-Process Script Approval (Part1)

🎯 أولًا: المقدمة – الفكرة العامة

- ❑ الجزء ده بيتكلم عن الـ Groovy Sandbox و In-Process Script Approval في Jenkins
- ❑ دي ببساطة تعتبر آلية أمان (Security Mechanism) جوا Jenkins بتضمن إن أي سكريبت Groovy يتنفّذ في بيئة آمنة ومحدودة.
- ❑ بمعنى تاني:
  - Jenkins مش بيشغّل أي كود Groovy كده على طول، لكنه بيشغّله داخل "Sandbox" ودا مكان محمي بيمنع أي كود ممكن يأذي النظام.
- ❑ والـ Script Approval هو الجزء اللي بيسمح للإدمن إنه يراجع ويوافق على السكريبتات اللي فيها أوامر غير معتمدة.

🧩 ثانيًا: مكونات الحماية في Jenkins

In-process Script Approval

- ➢ Jenkins بيعتمد على Plugin اسمه : Script Security Plugin 🧱
  - وده بيتنزل تلقائيًا وقت ما Jenkins يتثبت أول مرة (مش محتاج تنزّله بنفسك).
- ➢ البلجن ده بيقسّم عملية الأمان لحاجتين:

1. Groovy Sandbox ← بتشتغل افتراضيًا (Enabled by default)
   وتشغّل السكريبتات (سواء declarative أو scripted) في بيئة آمنة من غير تدخل إدمن.
2. Script Approval ← بتسمح للإدمن إنه يراجع أي سكريبت أو method مش مسموح بيها
   ويقرر هل يوافق عليها ولا يرفضها.

🧰 ثالثًا: تجربة عملية

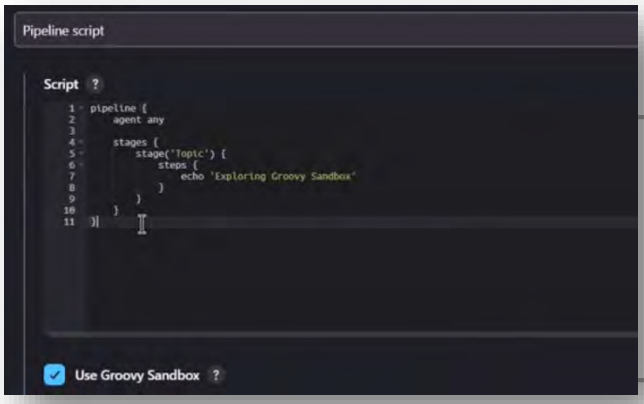
◆ الخطوة 1 – التأكد من وجود الـ Plugin

- راح على Manage Jenkins → Plugins → Installed
- ولقي فعلاً إن الـ Script Security Plugin متسطب افتراضي.

يعني Jenkins محمّيه من أول يوم.

◆ الخطوة 2 – إنشاء Pipeline جديد

- ➢ أنشأ job جديدة باسم : Groovy-Sandbox-Test
  - واختار نوعها Pipeline، وبعدين كتب سكريبت بسيط جدًا فيه Stage واحد بيعمل print لقيمة معينة (يعني مجرد اختبار).

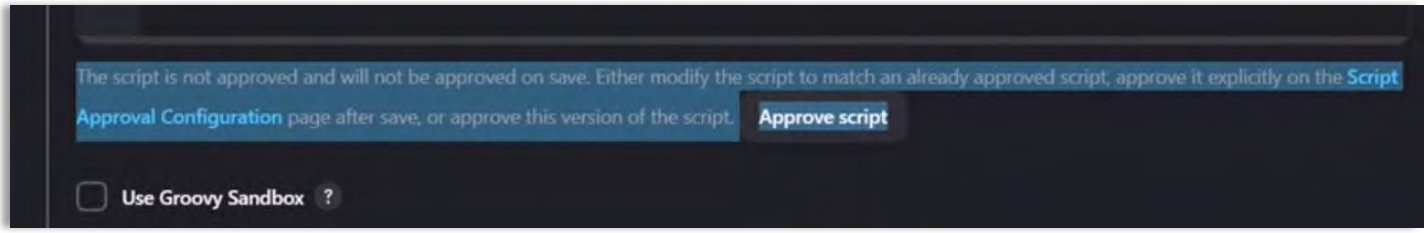


◆ الخطوة 3 – ملاحظة الـ Checkbox بتاع الـ Sandbox

- ➢ في أسفل صفحة السكريبت دايمًا هتلاقي : [✓] Use Groovy Sandbox
  - لما تكون متفعلة : ✅
    - o السكريبت هيتشغّل في بيئة آمنة (restricted mode)

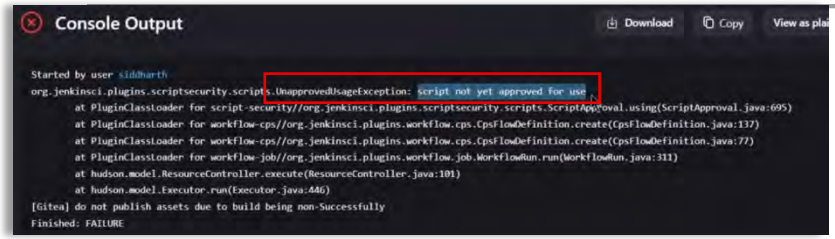
◆ الخطوة 4 – إيه اللي بيحصل لو عطلتها؟

- هو جرّب يشيل العلامة دي ❌ (يعني عطّل الـ Sandbox)
- مجرد ما عمل كده، Jenkins كتبله رسالة تحذير:



  - يعني Jenkins بيقوله:
    "السكريبت ده مش معتمد، ومش هيتشغّل إلا لما الإدمن يوافق عليه".
- لكن بما إنه كان داخل بحساب Admin ، Jenkins إداله خيارين:
  - يا إما يوافق على السكريبت من نفس الصفحة.
  - يا إما يعدّله بحيث يتوافق مع سكريبتات معتمدة . (موجودة في In-Process Script Approval)

هو اختار ما يوافقش دلوقتي، عشان يشوف هيحصل إيه لما يشغّل الـ job كده.



◆ الخطوة 5 – تشغيل الـ Job

- عمل : Build Now
- وطبعًا النتيجة كانت Failed ❌ :

لأن Jenkins وقف السكريبت وقال في الـ logs :

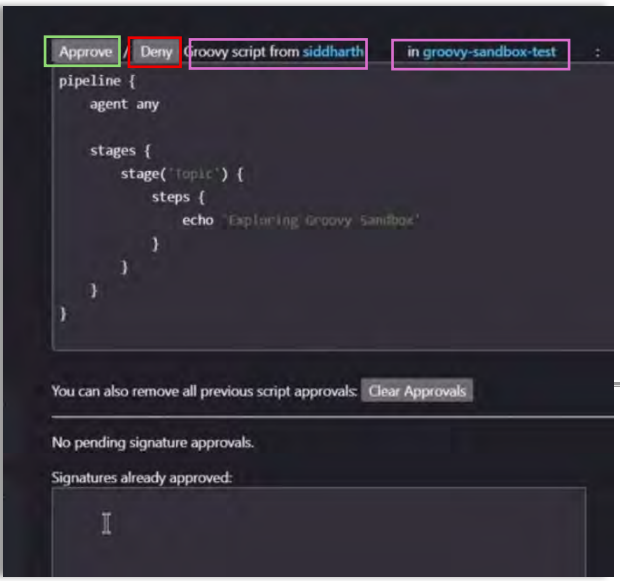
  - يعني Jenkins رفض يشغّله قبل الـ Admin ما يراجع الكود.

👨‍💼 رابعًا: هنا بييجي دور الـ Admin

➢ الإدمن يروح على المسار : Manage Jenkins → In-Process Script Approval

- وهتلاقي هناك إشعار (Notification) بيقول: "1 Script Pending for Approval"
- دخل عليه، ولقي السكريبت اللي محتاج الموافقة ومعاه:
  - مين اللي رفعه (اسم الـuser)
  - واسم الـ job اللي استخدم السكريبت ده
- وهنا عنده زرارين:
  - Approve ✅
  - Deny ❌
- هو اختار Approve

وبكدا Jenkins أضاف السكريبت إلى قائمة السكريبتات المعتمدة (approved list)



◆ الخطوة 6 – إعادة تشغيل الـ Job بعد الموافقة

➢ رجع تاني لـ "Groovy Sandbox Test" job ، وضغط Build Now تاني.

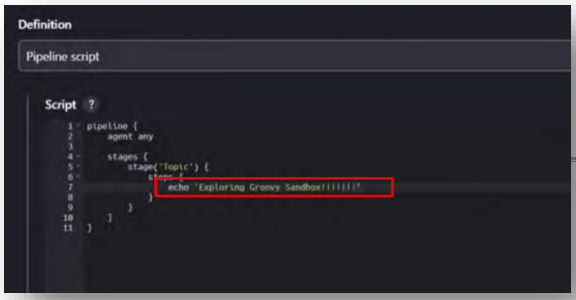
➢ النتيجة:

✅ الـ Job اشتغلت بنجاح وطبعت النتيجة.

◆ الخطوة 7 – تجربة تعديل السكريبت

- هو عدّل السكريبت شوية (مثلاً غير قيمة أو أمر) .. وبمجرد ما حفظ، Jenkins فهم إن ده سكريبت جديد. فرجع يطلب موافقة تاني:

"Script modified — needs new approval"



- الإدمن المرة دي وافق من نفس الصفحة (بضغطة زر "Approve Script")
وبعدها الـ Job اشتغلت تاني عادي جدًا.

🔒 خامسًا: نقطة مهمة جدًا — التحكم في الـ Sandbox

❖ في الحالة العادية، أي مستخدم يقدر:

- يفعّل ✅ أو يعطّل ❌ الـ Groovy Sandbox من checkbox في الـ UI

❖ لكن ده ممكن يبقى خطر، خصوصًا في شركات كبيرة.

❖ علشان كده في السيشن الجاية هنشوف إزاي نقدر نمنع المستخدمين من تعطيل الـ Sandbox نهائيًا
بحيث تفضل دايمًا مفعّلة ومحدش يقدر يعبث بيها.

💡 Best Practices & Real Notes

1. سيب الـ Sandbox مفعّل دايمًا .
   - تعطيله بيعرّض Jenkins لأي كود ضار ممكن يعمل access على الـ filesystem أو network
2. استخدم Script Approval بحكمة.
   - ماتوافقش على أي سكريبت إلا لو فاهم كل سطر فيه.
3. كل تعديل جديد في السكريبت = Approval جديد.
   - ده طبيعي جدًا لأن Jenkins بيعتبره "نسخة جديدة" من الكود.
4. في بيئات Production ، اقفل خيار Disable Sandbox
   - علشان المستخدمين العاديين مايلعبوش في إعدادات الأمان.

📜 الخلاصة

- Jenkins بيستخدم Groovy Sandbox كطبقة حماية أولى.
- وبيستخدم Script Approval كخط دفاع ثاني بيخلي الإدمن يراجع السكريبتات الغريبة.
- أي سكريبت جديد أو متعدل محتاج موافقة إدمن.
- الـ Plugin المسؤول عن كل ده هو Script Security Plugin
- ودايمًا خليك فاكر:

  ✅ الـ Sandbox = أمانك الأول.

  ❌ تعطيله = مخاطرة كبيرة جدًا.

Demo Groovy Sandbox and In-Process Script Approval (Part2)

المقدمة

- ❑ في الجزء ده، هانكمل الشرح اللي بدأه قبل كده عن الـ Groovy Sandbox وموضوع Script Approval في Jenkins
- ❑ الفكرة هنا إن الـ admin عايز يمنع المستخدمين العاديين إنهم يلعبوا في الإعداد اللي اسمه "Use Groovy Sandbox"
(يعني مايبقاش كل واحد يقدر يفعل أو يعطل الـ Sandbox زي ما يحب)

الخطوات

أول حاجة:

- ❖ راح على إعدادات Security داخل Jenkins من : Manage Jenkins → Security Tab
- ❖ وبعدين بدأ يدور على حاجة اسمها Sandbox Configuration
- ❖ لقى إن فيه اختيارين مهمين:

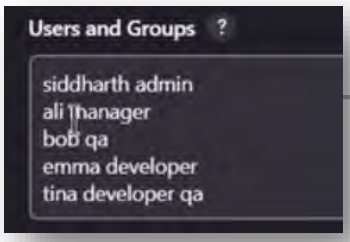
1. ✅ Force the use of Sandbox globally ← بيخلي الـ sandbox شغال إجباري على كل الـ pipelines
2. 👁 Hide the Sandbox checkbox ← يخفي الـ checkbox ده تمامًا من واجهة المستخدم.

- ❖ فعّل الاتنين مع بعض:
  - فعّل "Force Sandbox" علشان يمنع تشغيل سكريبتات غير آمنة.
  - وفعّل "Hide Checkbox" علشان المستخدمين ما يقدروش يغيروا الإعداد ده.

بعد كده:

ضغط على Apply علشان يحفظ التغييرات.

- لاحظ انه في اللحظة دي هو داخل بال (user admin) فكدا كدا الـ checkbox هايبقي ظاهر



بعدين جرب يدخل كمستخدم تاني (اسمه Ali)

- ➢ مبدئيا : أضاف ali في الـ manager group
- ➢ بعدين اداله permission: configuration

- ❖ وعلشان يتأكد إن المستخدم العادي (Ali) مش شايف الـ checkbox
سجل الدخول في jenkins بـ Ali
  - لكن في الأول، لسه الخيار ظاهر عند Ali
  - فعرف إن Jenkins محتاج Restart علشان التغيير يشتغل بشكل فعلي.

بعد الـ Restart :

- ❖ رجع يدخل تاني كمستخدم Ali وفتح أي job (زي DevOps pipeline أو Hello World)
- ❖ لما دخل على Configure وفتح جزء الـ Pipeline script...
✅ فعلاً ما لقاش الـ checkbox بتاع "Use Groovy Sandbox" موجود.

- يعني المستخدمين العاديين بقوا متقيدين بالـ sandbox

⚠️ لكن خد بالك:

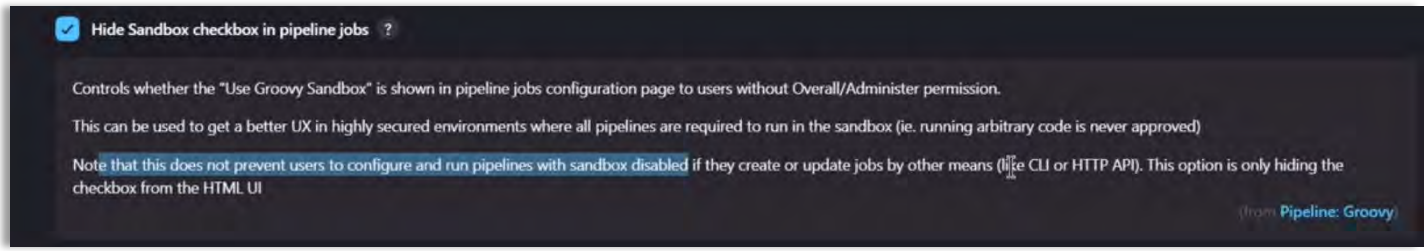
❖ نبه على نقطة مهمة جدًا 👇

- الإخفاء ده بيمنع المستخدم يشوف أو يغير الإعداد من الـ UI فقط (واجهة Jenkins)
- لكن لو المستخدم عنده صلاحية يستخدم CLI أو HTTP API ممكن لسه يقدر يغير الإعداد من هناك.

❖ بمعنى تاني:

"Hide the checkbox" مش حماية كاملة، هي بس شكلية في الواجهة الرسومية.

🔒 الحماية الحقيقية بتيجي من Force sandbox globally



📘 الخلاصة

- Jenkins بيستخدم Groovy Sandbox كطبقة أمان تشغل السكريبتات في بيئة محدودة.
- الـ Admin يقدر يفرض استخدام الـ Sandbox على كل الـ pipelines
- كمان ممكن يخفي الـ checkbox من واجهة المستخدم علشان ما يتلعبش فيه.
- التغيير ده بيتطلب Restart علشان يشتغل.
- إخفاء الـ checkbox ما يمنعش التعديل من CLI أوAPI ، بس يمنعه من الـ UI

Demo Groovy Sandbox and In-Process Script Approval (Part3)

المقدمة:

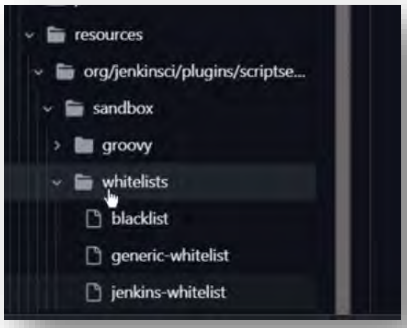
- ❑ الجزء ده بيكمل السلسلة اللي اتكلمنا فيها قبل كده عن Groovy Sandbox في Jenkins
- ❑ المرّة دي التركيز على:
- ❖ "إيه هي الـ methods (الدوال) والـ signatures اللي مسموح بيها أو الممنوعة داخل الـ Groovy Sandbox ؟"
- ❖ بمعنى تاني، Jenkins بيمنع بعض الدوال الحساسة من إنها تتنفّذ داخل الـ pipeline إلا لما الـ admin يوافق عليها يدويًا، علشان يمنع أي سكريبت ضار أو غير آمن من التأثير على النظام.

الخطوة 1: معرفة مصدر الـ Allowed / Blocked Methods

script-security-plugin/src at master · jenkinsci/script-security-plugin

- بدأ بأنه فتح GitHub repo الرسمي بتاع Jenkins Script Security Plugin (اللي بيتحكم في أمان السكريبتات)
- المسار داخل الكود كان كده 👇

src/main/resources/org/jenkinsci/plugins/scriptsecurity/sandbox/



- جوه هناك، لقى فولدر اسمه whitelist، وفيه:
  - ملف اسمه jenkins-whitelist ← بيحتوي على الـ API methods المسموح بيها.
  - ملف اسمه generic-whitelist ← بيعرض كل الـ signatures المسموح بيها.
  - وفي الآخر، فيه blacklist file ← فيه الدوال والـ signatures اللي ممنوعة تمامًا.

دي اللي بنسميها blocked / restricted methods

الخطوة 2 : تجربة دوال ممنوعة عمليًا

- ➢ رجع لـ Jenkins وفتح مشروع اسمه "Groovy Sandbox Test Project"
- ➢ دخل على Configure وعدّل السكريبت علشان يضيف 3 stages :

Script ?

```
pipeline {
    agent any

    stages {
        stage('Topic') {
            steps {
                echo 'Exploring Groovy Sandbox'
            }
        }
        stage('Get Hudson Instance') {
            steps {
                script {
                    def hudson = hudson.model.Hudson.getInstance()
                    println "Hudson Instance: ${hudson}"
                }
            }
        }

        stage('Get System Property') {
            steps {
                script {
                    def myProperty = java.lang.System.getProperty("user.name")
                    println "System Property: ${myProperty}"
                }
            }
        }
```

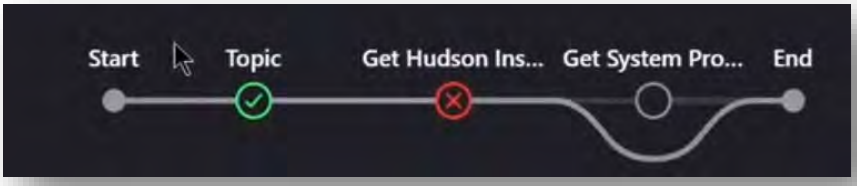
محتوى الـ stages 👇

1. Stage 1 — Simple Echo
   - دي بسيطة، بس بتطبع رسالة .
2. Stage 2 — GetHudsonInstance
   - هنا بيحاول يجيب الـ instance الرئيسي من Jenkins
   - لكن الـ method getInstance() ممنوعة في الـ sandbox
   - وده واضح من قائمة الـ blacklist اللي شافها في GitHub :

any variant of Jenkins.getInstance is blocked

3. Stage 3 — GetSystemProperty
   - هنا بيحاول يجيب اسم المستخدم من system properties
   - بس برضو getProperty() ممنوعة لأنها ممكن تسرّب بيانات حساسة من النظام.

🧱 الخطوة 3: تجربة تشغيل السكريبت

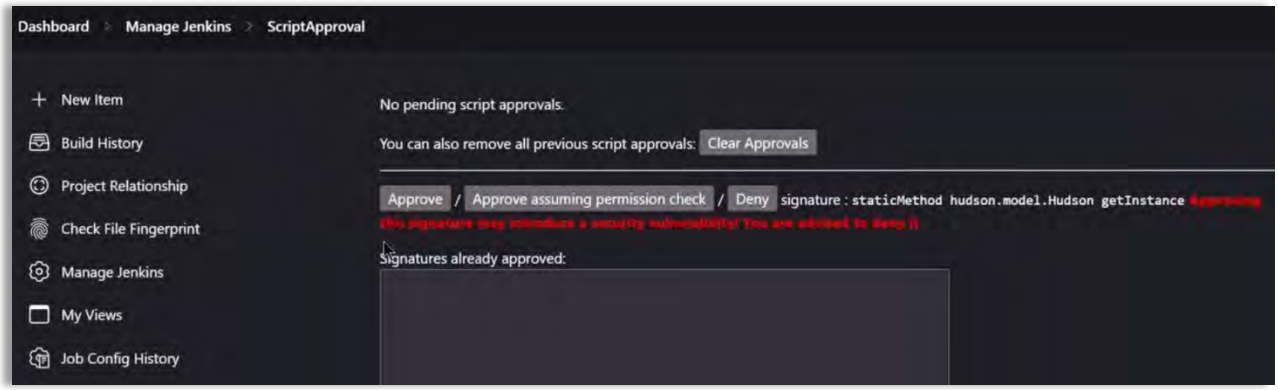


- فعل خيار "Use Groovy Sandbox" ✅
- ثم ضغط Save → Build Now
- النتيجة:
  - ✅ المرحلة الأولى نجحت (echo)
  - ❌ المرحلة الثانية فشلت (GetHudsonInstance)
- في اللوج:

```
[Pipeline] stage
[Pipeline] { (Get Hudson Instance)
[Pipeline] script
[Pipeline] {
Scripts not permitted to use staticMethod hudson.model.Hudson getInstance. Administrators can decide whether to
approve or reject this signature.
[Pipeline] }
[Pipeline] // script
[Pipeline] }
```

- وده معناه إن Jenkins رفض تشغيلها لأنها على الـ blacklist

- ولو ضغطت عالكلام (اللي باللون الأزرق دا) ← هيأخدك علي الـ Script Approval Page
(ممكن تروحلها بردو من خلال : Manage Jenkins → In-process Script Approval)
- ولو عملت Refresh هايوريك لو فيه أي signatures في حالة Pending ومستنية الـ approval بتاعك كأدمن.



🧠 النقطة الأهم

اللي حصل هنا بيوضح إن:

- أي method مش على الـ "allow list" بتحتاج موافقة من الـ admin
- Groovy Sandbox بيراقب كل عملية (method call / field access)
- لما حاجة مش آمنة تتنفّذ، Jenkins يوقف السكريبت ويطلب موافقة.
- الـ admin يقدر يقرر إذا كانت تتحط في الـ "approved list" ولا تفضل ممنوعة.

🔐 الخلاصة:

- الـ Groovy Sandbox بيحمي Jenkins من تشغيل أكواد ضارة.
- بعض الـ methods زي:
  - Jenkins.getInstance()
  - System.getProperty()

  بتكون محظورة افتراضيًا لأنها ممكن تكشف أو تتحكم في النظام.
- الـ admin يقدر يسمح بيها يدويًا من Script Approval page
- بعد الموافقة، Jenkins بيضيفها في approved signatures list وبتشتغل عادي في أي pipeline

Migrate Jenkins to Another Node

🎬 المقدمة

- ❑ النهارده هنشرح عملية نقل Jenkins من Node إلى Node تاني
- ❑ بمعنى تاني، إزاي الـ Jenkins Server تهاجر من سيرفر قديم لسيرفر جديد من غير ما تضيع أي إعدادات، Jobs، Plugins أو Data
- ❑ الجزء ده بيتكلم عن طريقة يدوية بس موثوقة وسهلة، تعتمد على نسخ مجلد Jenkins Home من الماكينة القديمة إلى الجديدة.

🧩 أولاً: الفكرة العامة

Migrating a Jenkins instance to a new machine

فكرة العملية بسيطة جدًا:

1. نوقف Jenkins على السيرفر القديم.
2. نعمل Backup (tar.gz) لمجلد JENKINS_HOME
3. ننقل الملف للسيرفر الجديد باستخدام scp
4. نفك الضغط عن الملف ونبدأ Jenkins من جديد.

بس قبل ده كله، لازم نراعي حاجتين مهمين جدًا 👇

- إصدار Jenkins على السيرفر الجديد = نفس الإصدار اللي على السيرفر القديم.
- إصدار JDK (Java) يكون متطابق على الاتنين.

⚙️ ثانيًا: الخطوات العملية بالتفصيل

1. 🖥️ التحقق من البيئة

- السيرفر القديم عليه Jenkins شغال، وليكن IP: 64.227.x.x
- السيرفر الجديد عليه Jenkins لسه متسطب بس مش متكوّن (Getting Started Page) ، وليكن IP: 165.322.x.x

---

2. 🧱 تحديد مكان Jenkins Home

- ➢ في السيرفر القديم، المكان الافتراضي دايمًا بيكون : /var/lib/Jenkins
- ➢ نقدر نتأكد بأمر: ls -l /var/lib
- ➢ هنلاقي مجلد اسمه jenkins ومالكه هو المستخدم Jenkins

---

3. 🛑 إيقاف Jenkins قبل النسخ

عشان نمنع أي تغييرات أثناء النسخ، نوقف السيرفس:

- o sudo systemctl stop jenkins
- o sudo systemctl disable jenkins

---

4. 📦 عمل Backup (نسخة مضغوطة)

نضغط المجلد بالكامل في ملف tarball : sudo tar -zcfv jenkins-backup.tar.gz /var/lib/jenkins

🟡 ملاحظة مهمة : لو عندك Jenkins كبير (كثير الـ jobs أو plugins) ← الخطوة دي ممكن تاخد وقت طويل جدًا.

---

5. 💻 إعداد السيرفر الجديد

- ندخل عليه: ssh user@165.322.x.x
- نتأكد إن Jenkins متسطب فعلاً: sudo systemctl status Jenkins
- هنشوف إنه شغال لكن مش configured لسا ، فهنوقفه برضو:
  - sudo systemctl stop jenkins
  - sudo systemctl disable jenkins

---

6. 🚀 نقل النسخة إلى السيرفر الجديد

- نستخدم أمر scp : scp jenkins-backup.tar.gz user@165.322.x.x:/tmp/jenkins-backup.tar.gz
- بعد ما يطلب الباسورد وينتهي النقل، هيظهر الملف في /tmp على السيرفر الجديد.

---

7. 🗂️ استبدال المجلد القديم بالـ Backup وفك ضغطه (دلوقت احنا في السيرفر الجديد)

- عشان تتأكد ان الخطوة رقم (6) تمام والدنيا فل :
  - ll /tmp/jenkins-scripted-backup.tar.gz
  - هايجيبلك انه اتنقل والدنيا تمام

```
-rw-r--r-- 1 root root 2194140482 Nov 10 15:53
```

- بعدها هاينقله علي /var/lib لأن زي ماتفقنا ان المسار الافتراضي بتاع الـ Jenkins بيبقي في الفولدر ده
  - mv /tmp/jenkins-scripted-backup.tar.gz /var/lib/

🚨 خلي باااااااالك علشان الحتة دي وقعت أبطال ☻

- طبعا انت كدا خلاص نقلت الـ zip. في الـ var/lib وهاتجري تعمله exract .. بس المشكلة انك لاااااااازم تحذف مجلد الـ Jenkins الحالي (اللي لسا فاضي) وموجود في الـ var/lib/ بتاع السيرفر الجديد
- وللـ Best practice ..
  - خدلك backup الاول : tar -zcfv jenkins.bak jenkins
  - وبعدين امسحه : sudo rm -rf jenkins

- دلوقت بقا مبقاش عندك أي Jenkins Directory في الـ var/lib/ وبالتالي اعمل الـ extract بتاعك في سلام
  - sudo tar -zxvf /tmp/jenkins-backup.tar.gz -C /var/lib/

---

8. 👤 تأكد من الـ Permissions

- بعد الاستخراج : ls -ld /var/lib/Jenkins
- لو المالك مش Jenkins ، عدلها كده : sudo chown -R jenkins:jenkins /var/lib/jenkins

---

9. ▶️ تشغيل Jenkins مرة تانية

شغّله وفعّله:

- sudo systemctl enable jenkins
- sudo systemctl start jenkins
- sudo systemctl status jenkins

لما تشوف الحالة active (running)، كده السيرفس اشتغل بنجاح.

---

10. 🌐 التحقق من المتصفح

- افتح المتصفح على IP السيرفر الجديد: http://165.322.x.x:8080
- هتلاحظ إن واجهة Jenkins اللي كانت على السيرفر القديم ظهرت بالضبط نفسها .. كل الـJobs ، Plugins، Configurationsاتنقلت.

⚠️ تنبيه: ماينفعش يكون Jenkins شغال على السيرفرين في نفس الوقت بنفس الإعدادات (لتفادي تضارب البيانات).

🧭 الخلاصة

إحنا هنا عملنا Full Migration لـ Jenkins من Node إلى Node جديد باتباع الخطوات التالية:

المرحلة	الخطوة الأساسية
1	إيقاف Jenkins القديم
2	عمل tar.gz للـ Jenkins Home
3	نقل الملف بالسيرفر الجديد
4	حذف Jenkins الفاضي
5	استخراج النسخة القديمة
6	تصحيح Permissions
7	تشغيل Jenkins الجديد

النتيجة النهائية:

- نفس Jenkins القديم اشتغل على السيرفر الجديد.
- كل البيانات محفوظة 100%
- العملية آمنة وقابلة للتكرار في أي بيئة (Development, Staging, Production)

GitHub Action Basics

🎯 المقدمة – الفكرة العامة

- ❑ الجزء ده بيتكلم عن GitHub Actions، وهي خدمة مقدّمة من GitHub بتوفر منصة Automation و CI/CD داخل نفس الريبو، بدون الحاجة لأي أدوات خارجية زي Jenkins أو GitLab CI
- ❑ الفكرة إنك تقدر تكتب ملف YAML داخل فولدر .github/workflows، ومن خلاله تحدد إيه اللي يحصل تلقائيًا لما يحصل حدث معين في الريبو (زي push أو pull request)
- ❑ GitHub هو اللي بيتكفّل بتشغيل الكود ده على virtual machines (runners) سواء Ubuntu أو Windows أو macOS

⚙️ الشرح التفصيلي خطوة بخطوة

1. 🧩 ما هو GitHub Actions

- GitHub Actions = Automation Platform داخل GitHub
- بتسمحلك تعمل مهام تلقائية من جوه الـ repository نفسه.
  - يعني بدل ما تعمل عمليات زي Build – Test – Deploy يدويًا، فتقدر تكتب ملف واحد (YAML file) وGitHub هينفذ كل ده تلقائيًا لما يحصل حدث معين (event).
- مكتوبة بـ YAML وموجودة في .github/workflows/

🛠️ أوامر أو أدوات رئيسية:

- On : لتحديد الـ event (زي push, pull_request)
- Jobs : لتحديد الـ tasks
- runs-on : لتحديد نظام التشغيل للـ runner (Ubuntu, Windows, macOS)

💡 Best Practice :

- استخدم YAML validation tool زي yamllint لتجنب الأخطاء في المسافات (indentation errors)

---

2. 🧩 مميزات GitHub Actions

- ➢ Manage the infrastructure
  - Including (setting up servers – Scalling Resources – Managing execution environment
- ➢ وظيفتك انك تكتب الـ workflow configurations in YAML files .. وبعدين هي هاتهندل الدنيا .. وده بيشمل :
  - Task in virtual environments
  - Caching necessary dependencies
  - Providing reports on the outcomes
- ➢ بالنسبة للـ Automation فهي بتساعدك في :
  - Streamline Development
  - Reduce Manual Errors
  - Increase the Efficiency of your software development workflow

---

🧠 ليه الشركات بتستخدمه؟

- ❖ لأن لو الشركة أصلًا بتستخدم GitHub كمكان لحفظ الكود، فـ GitHub Actions بيكون الحل الأنسب للـ Automation ودا لأنه متكامل جوه المنصة نفسها (يعني مش هاتحتاج أدوات خارجية زي Jenkins أو GitLab CI)

⚙️ آلية عمل GitHub Actions :

كل Automation في GitHub Actions بيتكون من 4 مستويات رئيسية:

المستوى	المعنى
Workflow	هو العملية الكاملة (زي "Run Tests on Push") — ملف YAML واحد
Job	كل Workflow فيه Job(s)، كل Job = مجموعة مهام تنفذ مع بعض
Step	الخطوات داخل الـ Job (زي: "checkout repo", "install Node.js", "run tests")
Runner	الجهاز (VM) اللي بيشغّل الـ Jobs فعليًا

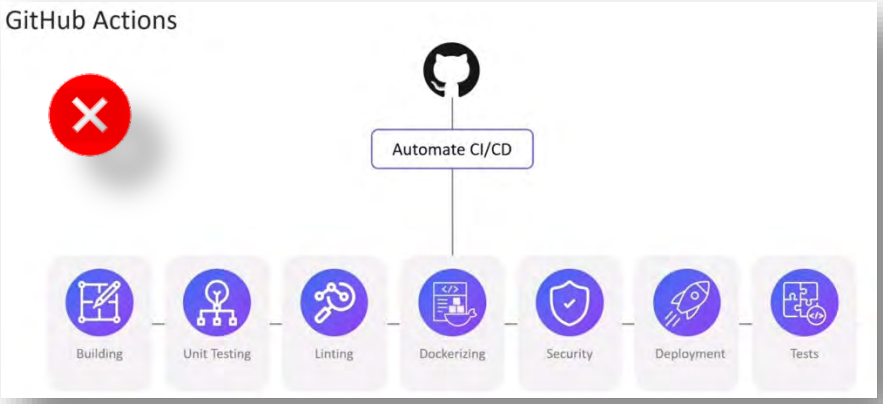
📁 مكان ملفات الـ Workflow

- كل الملفات دي بتتحط في : **.github/workflows/**
- ودي بتكون ملفات YAML زي: **.github/workflows/test.yml**

---

3. 🧩 GitHub Actions vs. Jenkins

❑ هل GitHub Actions بيتستخدم فقط لتطبيق CI/CD (يعني build - test - deploy) ؟

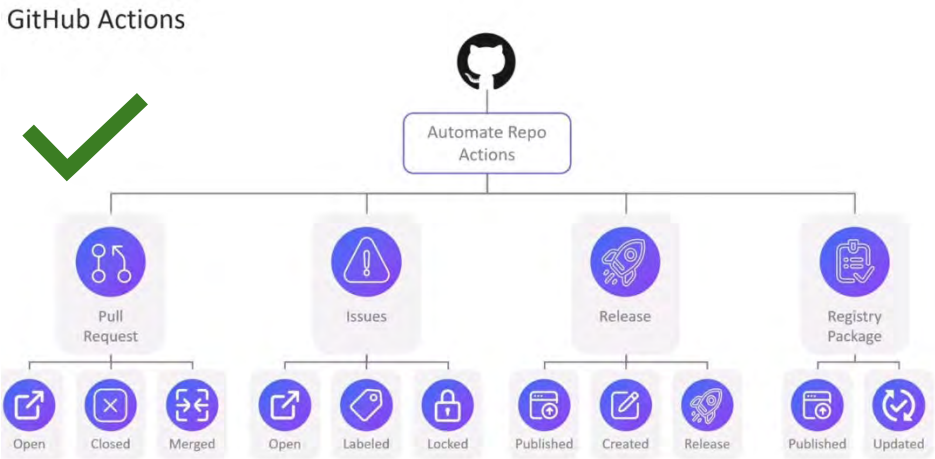


❖ الاجابة وبشكل قاطع هي لأ،
GitHub Actions مش بس للـ CI/CD

❑ لأن GitHub Actions هي ميزة داخل GitHub بتخليك تعمل Automation لأي حاجة بتحصل داخل الـ repository بتاعك.
ومش بس لما ترفع كود أو تبني مشروع، لكن كمان لأي حدث (event) بيحصل جوه الـ repo.

❖ وبالتالي تقدر تستخدمها علشان :

- تنظّم الـ issues
- تدير الـ pull requests
- تبعت رسائل Slack أو Emails
- تمسح أو ترتّب ملفات معينة.
- تعمل Tasks مجدولة تلقائيًا .



💻 أمثلة على الأحداث (Triggers)

GitHub Actions ممكن تتفاعل مع أحداث كتيرة، زي مثلاً:

1. **push event**
لما تعمل push لكود جديد، ممكن GitHub Actions تشغّل workflow يختبر الكود أو يبني المشروع.
2. **pull request event**
لما حد يعمل Pull Request أو يعدّل فيه، ممكن تشغّل workflow يراجع الكود أو يضيف تعليق تلقائي.
3. **issue event**
لما تتفتح Issue جديدة أو تتحدث، ممكن GitHub Actions تبعت Notification أو تصنّفها تلقائيًا (labeling)
4. **package event (GitHub Registry)**
لما ترفع Package جديدة أو تحديث، ممكن تشغّل عملية نشر أو إخطار لفريقك.
5. **schedule (time-based trigger)**
ممكن تخلي workflow يشتغل تلقائيًا كل يوم أو أسبوع(زي cron job)

---

4. 🧩 مفاهيم مهم تكون عارفها :

1. 📁 الـ Workflow

- كل workflow عبارة عن تسلسل من العمليات التلقائية.
- مكانها في المسار /.github/workflows.
- ممكن يكون عندك أكتر من workflow في نفس الريبو.

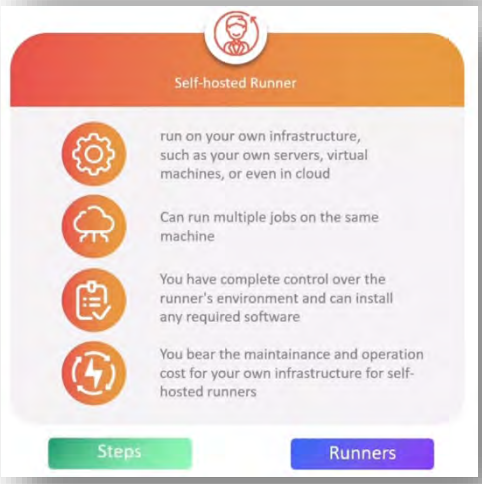
2. ⚡ الـ Jobs والـ Steps

- كل Job بيتنفذ على runner (VM)
- كل Job بيتكون من مجموعة steps
- Steps بتتنفذ تتابعيًا داخل الـ job ، لكن jobs ذات نفسها ممكن تشتغل بالتوازي.

💡 Trick :

لو عايز تشغّل نفس الـ step على أكتر من OS ، استخدم Matrix strategy :

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
runs-on: ${{ matrix.os }}
```



3. 🧠 الـ Runners

- عبارة عن VM ومسئول عن execute your workflow upon triggering ..
- وكمان GitHub تلقائيا بتوفر runners لكل Job عندي .. وفيه منه نوعين:

I. GitHub-hosted runners
   - GitHub هو اللي بيوفرهم ويشغّلهم.
   - بتشتغل على Azure (Linux/Windows) أو macOS cloud
   - بتبدأ من VM نظيفة لكل Job

II. Self-hosted runners
   - إنت اللي بتوفّر السيرفر أو الماشين اللي هيشتغل عليها.
   - عندك تحكم كامل في الـ environment
   - لازم تعمل setup و maintenance بنفسك.

المقارنة	GitHub-Hosted	Self-Hosted
الإدارة	Managed by GitHub	Managed by you
التخصيص	محدود	كامل
الأداء	ثابت	يعتمد على السيرفر
التكلفة	ضمن الاشتراك	على حسابك
عدد الـ Jobs	Clean VM لكل Job	ممكن تشغل أكتر من Job على نفس السيرفر

💸 ملاحظات عملية:

- GitHub-hosted = convenient لكن محدود بساعات تشغيل مجانية.
- Self-hosted = مناسب للشركات الكبيرة أو لو محتاج hardware معين.

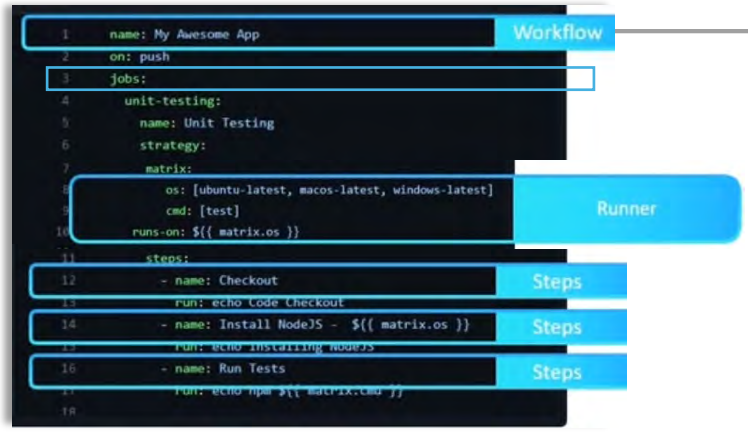
4. Logs و Artifacts 📄

- تقدر تتابع logs من Actions tab داخل GitHub UI
- تشوف الـ steps اللي اتنفذت ونتيجتها.
- ممكن كمان تعمل Download لأي artifacts (زي نتائج build أو test reports)

💡 Best Practice :

احفظ الـ test reports أو build files كـ artifacts عشان تقدر تراجعها بعدين بدون إعادة التشغيل.

⚡ مثال عملي :

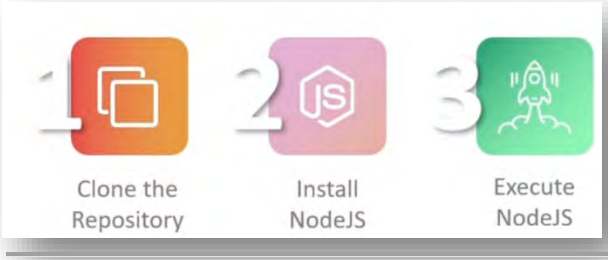


❖ في مشروع Node.js ، كل مرة تعمل push للكود،
GitHub Actions هتعمل build و run للـ tests تلقائيًا .

➢ جواه فيه job اسمه Unit Test Stage،
وده بيشتغل على 3 أنظمة تشغيل:

- Windows
- Ubuntu
- macOS

➢ يعني GitHub هينشئ 3 virtual machines في نفس الوقت
(واحدة لكل OS) وكمان هايشغل نفس الـ steps على التلاتة بالتوازي (parallel)



🧩 خطوات التنفيذ داخل الـ Job :

1. Clone repository (يسحب الكود من GitHub)
2. Install Node.js (يجهز البيئة)
3. Run Node.js tests (يشغل الاختبارات)

➕ الـ 3 runners هيعملوا نفس الخطوات في نفس الوقت .

📄 بعد التنفيذ:

- تقدر تشوف كل الـ logs والنتائج من داخل GitHub → Actions tab
- ممكن كمان تشوف Artifacts (زي ملفات ناتجة من الـ build أو التقارير)
- كده تقدر تتابع كل خطوة وتشوف فين حصل خطأ لو حصل.

🧩 الملخص الشامل

- GitHub Actions = منصة Automation مدمجة داخل GitHub لتنفيذ CI/CD
- بتكتب ملفات YAML جوه .github/workflows/ لتحديد إيه اللي يحصل لما يحصل (event معين push, PR, ... إلخ)
- كل workflow بيتكون من jobs وsteps، وبيتنفذ على runners (GitHub-hosted أو self-hosted)
- GitHub-hosted runners بتتوفر جاهزة، بينما self-hosted بتديك تحكم كامل.
- تقدر تتابع الـ logs والـ results من واجهة GitHub مباشرة.

📘 في النهاية:

GitHub Actions بتوفر طريقة بسيطة، متكاملة، وقابلة للتخصيص لأي مشروع، وبتقلل اعتمادك على أدوات خارجية زي Jenkins أو GitLab CI .. ولذلك هي الأنسب لما يكون الريبو الأساسي بتاعك على GitHub وعايز تعمل Automation بدون Setup إضافي.

Migrating Jenkins Pipeline to GitHub Action

في الجزء ده، هانشرح خطوة بخطوة إزاي تقدر تنقل (migrate) الـ Jenkins jobs اللي عندك إلى GitHub Actions workflows باستخدام أداة جاهزة اسمها GitHub Actions Importer

Migrating from Jenkins with GitHub Actions

🧩 أولاً: مقدمة عن الفكرة

- ❑ بدل ما تعيد بناء الـ pipelines بتاعتك من الأول في GitHub Actions ، تقدر تستخدم tool من GitHub اسمها GitHub Actions Importer 👉
- ❑ ودي بتعمل تحويل تلقائي (automated migration) من Jenkins إلى GitHub Actions
- ❑ لكن الأداة ليها شوية شروط مسبقة (Prerequisites) لازم تكون موجودة قبل ما تبدأ.

⚙️ ثانياً: المتطلبات الأساسية (Prerequisites)

قبل ما تقدر تشغّل الـ importer ، لازم تكون مجهّز الآتي:

1. Jenkins account أو Organization فيها الـ pipelines اللي عايز تنقلها.
2. Personal API Token من Jenkins علشان الأداة تقدر توصل للـ jobs
3. في البيئة اللي هتشغّل منها importer ، لازم يكون عندك:
   - Docker (الأداة بتشتغل جوه container)
   - GitHub CLI (gh)

💡 ملحوظة مهمة:

خلي باااالك ان الأداة دي ما تقدرش تعمل Migrate لـ scripted pipelines أو secrets أو plugins مش معروفة.

🧰 ثالثاً: تثبيت GitHub CLI

cli/docs/install_linux.md at trunk · cli/cli

➢ أول خطوة عملية هي تنزيل وتثبيت GitHub CLI :

المدرب راح على الموقع الرسمي، دخل على صفحة CLI ، واختار نسخة Linux (Ubuntu)

ثم نفّذ أوامر التثبيت اللي كانت بالشكل ده :

Debian, Ubuntu Linux, Raspberry Pi OS (apt)

Install:

```
(type -p wget >/dev/null || (sudo apt update && sudo apt-get install wget -y)) \
        && sudo mkdir -p -m 755 /etc/apt/keyrings \
        && wget -qO- https://cli.github.com/packages/githubcli-archive-keyring.gpg | sudo tee /etc/apt/keyrings/githubcli-archi
        && sudo chmod go+r /etc/apt/keyrings/githubcli-archive-keyring.gpg \
        && echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/githubcli-archive-keyring.gpg] https://cli.
        && sudo apt update \
        && sudo apt install gh -y
```

بعد التثبيت، تأكد إنها اشتغلت بـ : gh --version

🧩 رابعاً: تثبيت GitHub Actions Importer Extension

الـ Importer مش جزء من CLI نفسه، فبتحتاج تضيفه كـ extension :

- gh extension install github/gh-actions-importer

🔐 خامساً : تسجيل الدخول على GitHub CLI

➢ علشان الأداة تقدر تشتغل مع حسابك في GitHub ، لازم تعمل login : **gh auth login**

وبعدين هيسألك:

- هل عايز تستخدم github.com ← قل له Yes
- طريقة الاتصال ← HTTPS
- طريقة المصادقة ← Login with a web browser

➢ هتاخد كود وتفتحه في المتصفح، وتكتبه في صفحة GitHub علشان تربط CLI بحسابك.

بعد كده هيرجعلك CLI ويأكد إنك Logged in بنجاح .

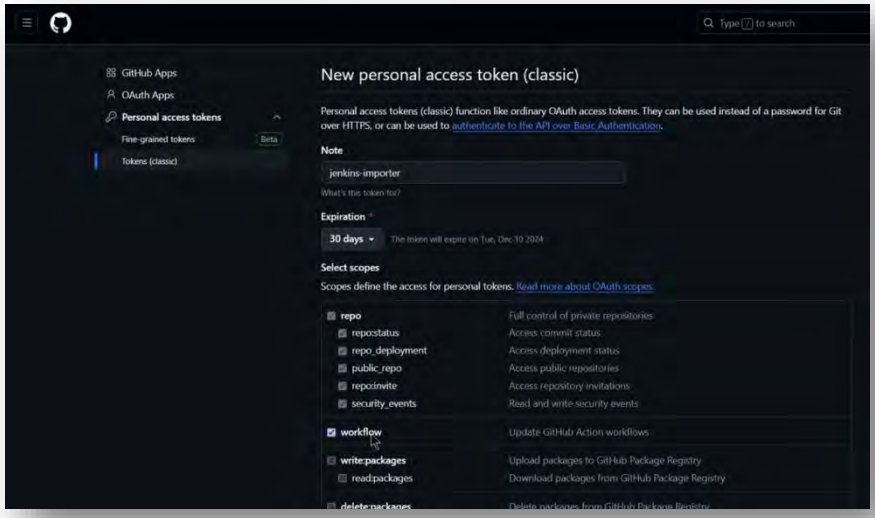
```
root@jenkins-controller-1 in ~ on ☁ (us-east-2)
❯ gh auth login
? Where do you use GitHub? GitHub.com
? What is your preferred protocol for Git operations on this host? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI? Login with a web browser

! First copy your one-time code: 143F-AE59
Press Enter to open https://github.com/login/device in your browser...
```

📄 سادساً: إعداد الـ Tokens (GitHub + Jenkins)

❖ بعد كده نبدأ إعداد البيانات اللي الأداة هتستخدمها:

1. GitHub Personal Access Token (Classic)

- تروح على GitHub → Settings → Developer Settings → Personal Access Tokens
- تختار "Generate new token (classic)"
- تديه اسم زي: Jenkins Importer
- تختار الصلاحيات:
  - ✅ workflow
  - ✅ repo
- تعمل Generate وتنسخ الـ Token



2. Jenkins API Token

- تروح على Jenkins → User → Configure
- تحت Tokens ← تضيف Token جديد اسمه GitHub Importer
- Generate وتنسخه قبل ما تسيب الصفحة.

⚙️ سابعاً: تشغيل أمر التهيئة (Configure)

هتستخدم الأمر ده علشان تسجل الـ tokens في الأداة : **gh actions-importer configure**

هيسألك:

- نوع الـ CI ← تختار Jenkins
- تدخل GitHub token + Jenkins URL + Jenkins token + Jenkins username

بعدها الأداة هتحفظ الإعدادات .

```
root@jenkins-controller-1 in ~ on ☁ (us-east-2) took 7s
❯ gh actions-importer configure

? Which CI providers are you configuring?: [Hit <space> to select, <ctrl+a> to toggle all, <ctrl+i> to invert selection
› ◯ Azure DevOps
  ◯ Bamboo
  ◯ CircleCI
  ◯ GitLab CI
  ◯ Jenkins
  ◯ Travis CI
  ◯ Bitbucket
```

```
✓ Which CI providers are you configuring?: Jenkins
Enter the following values (leave empty to omit):
✓ Personal access token for GitHub: ****************************************
✓ Base url of the GitHub instance: https://github.com
✓ Personal access token for Jenkins: ****************************************
✓ Username of Jenkins user: siddharth
✓ Base url of the Jenkins instance: http://64.227.187.25:8080/
Environment variables successfully updated.
```

🧱 ثامناً: اختبار الاتصال (Update command)

- قبل ما تبدأ التحويل، تأكد إن الأداة شغالة وآخر إصدار : **gh actions-importer update**
  - ده بيعمل Pull لأحدث container image من GitHub Package Registry

---

📊 تاسعاً: عمل Audit (تحليل قبل التحويل)

❖ الأمر ده بيعمل الآتي : **gh actions-importer audit jenkins --output-dir ./audit**

- يجمع كل الـ pipelines اللي موجودة على سيرفر Jenkins.
- بعدين بيحوّل كل pipeline منهم إلى الـ equivalent بتاعها في GitHub Actions (بشكل مبدئي).
- بعد كده بيولّد تقرير report بيوضح : (بيطلب منك تحددله مجلد يحفظ فيه كل الملفات الناتجة من عملية الـ Audit)
  - هل ممكن فعلاً تهاجر كل pipeline ولا لأ.
  - درجة التعقيد (يعني pipeline دي معقدة ولا بسيطة).
  - نسبة الاكتمال (قد إيه من الخطوات تقدر الأداة تهاجرها تلقائيًا).

➢ 🧠 يعني هو كأنه بيديّك صورة تحليلية قبل ما تبدأ النقل الفعلي، عشان تفهم المخاطر والصعوبات ..كمان بيديلك خيار اضافي وهو :

❖ 📊 **Forecast Build Runner Usage**

➢ بعد الـ audit ، GitHub بيديك اختيار إضافي وهو انك تعمل forecast
(يعني تقدير مسبق لاستهلاك الموارد (build runner usage))

🧩 الفكرة هنا:

- لما تنقل pipelines إلى GitHub Actions ، هتحتاج تعرف تقريبًا كل Pipeline بتستهلك وقت قد إيه أثناء التشغيل عشان تقدر تحسب تكلفة الـ runners أو تحدد الحجم المطلوب.

⚠️ لكن المشكلة : Jenkins مفيهوش REST API جاهز بيقدّم معلومات دقيقة عن مدة كل build

💡 الحل :

➢ GitHub بيقترح إنك تثبّت Plugin في Jenkins
الـ Plugin ده:

- بيجمع بيانات الـ builds
- وبيديها للأداة (GitHub Actions Importer)
- بالتالي تقدر تعمل توقع دقيق للوقت والموارد اللي هتحتاجها بعد الهجرة.

➕ لكنه قال إنه هيتخطى المرحلة دي، وهيروح مباشرةً للـ dry run

المرحلة	الهدف	الناتج
Audit	تحليل وضع الـ Jenkins pipelines قبل النقل	تقرير عن مدى قابلية النقل ومدى التعقيد
Forecast	حساب استهلاك الوقت والموارد بعد النقل	تقرير عن الـ build runner usage
Dry Run / Migrate	تجربة النقل أو التنفيذ الفعلي	GitHub Actions workflows جديدة

🧪 عاشراً Dry Run : (تجربة تحويل بدون تنفيذ فعلي)

❖ الـ Dry Run بيحول Pipeline واحدة مؤقتًا ويولّد لك ملف GitHub Workflow عشان تشوف شكله قبل التنفيذ.

- gh actions-importer dry-run jenkins --source-url https://jenkins.example/job/demo \ --output-dir ./output

```
root@jenkins-controller-1 in ~ on ☁ (us-east-2) took 16s
❯ gh actions-importer dry-run jenkins --source-url http://64.227.187.25:8080/job/Generate%20ASCII%20Artwork/ --output-dir tmp/dry-run

[2024-11-10 16:19:30] Logs: 'tmp/dry-run/log/valet-20241110-161930.log'
[2024-11-10 16:19:30] Output file(s):
[2024-11-10 16:19:30]  tmp/dry-run/Generate_ASCII_Artwork/.github/workflows/generate_ascii_artwork.yml
```

display it using cat command

- بعد التنفيذ، بيطلع ملف yml. في المجلد اللي اخترته.
- الملف شكله كده:

```
root@jenkins-controller-1 in ~ on ☁ (us-east-2) took 8s
❯ cat tmp/dry-run/Generate_ASCII_Artwork/.github/workflows/generate_ascii_artwork.yml
name: Generate_ASCII_Artwork
on:
  workflow_dispatch:
env:
#   # TimestamperBuildWrapper was not converted because the behavior is available by default in GitHub Actions and/or it is not configurable
jobs:
  build:
    runs-on:
      - ubuntu-latest
    steps:
    - name: checkout
      uses: actions/checkout@v4.1.0
    - name: run command
      shell: bash
      run: |-
        # Build a message by invoking ADVICESLIP API
        curl -s https://api.adviceslip.com/advice > advice.json
        cat advice.json
        # Test to make sure the advice message has more than 5 words.
        cat advice.json | jq -r .slip.advice > advice.message
        [ $(wc -w < advice.message) -gt 5 ] && echo "Advice has more than 5 words" || (echo "Advice - $(cat advice.message)  has 5 words or less" && exit 1)
        # Deploy
        sudo apt-get install cowsay -y
        echo $PATH
        export PATH="$PATH:/usr/games:/usr/local/games"
        cat advice.message | cowsay -f $(ls /usr/share/cowsay/cows | shuf -n 1)
```

وده فعليًا نسخة GitHub Actions من الـ Jenkins Job

⚠️ (تذكر ان انا كدا لسا منفذتش الـ migrate .. انا بس مجرد بشوف شكل الـ migrate هايبقي عامل ازاي)

🚀 الحادي عشر: تنفيذ التحويل الفعلي (Migrate)

بعد ما يتأكد إن الـ dry run تمام، ينفذ التحويل الحقيقي :

```
gh actions-importer migrate jenkins --target-url https://github.com/:owner/:repo --output-dir tmp/migrate --source-url my-jenkins-project
```

الشكل العام

```
root@jenkins-controller-1 in ~ on ☁ (us-east-2)
❯ gh actions-importer migrate jenkins --target-url https://github.com/jenkins-kk-demo/jenkins-to-actions --output-dir tmp/migrate --source-url http://64.227.187.25
:8080/job/Generate%20ASCII%20Artwork/
[2024-11-10 16:22:17] Logs: 'tmp/migrate/log/valet-20241110-162217.log'
[2024-11-10 16:22:21] Pull request: 'https://github.com/jenkins-kk-demo/jenkins-to-actions/pull/1'
```

طريقة كتابته في الـ CLI

الأداة هنا:

- بتخلق branch جديدة.
- بتحط فيها ملف .github/workflows/*.yml
- وبتعمل Pull Request في GitHub تلقائيًا.

➕ بعد كده تقدر تدخل GitHub وتشوف الـ PR وتعمل Merge

🧩 الثاني عشر: تشغيل الـ Workflow

❖ بعد الدمج، تروح إلى تبويب Actions في GitHub وتشغل الـ workflow يدويًا
(لأن trigger كان workflow_dispatch) ودا عمل error في البداية بسبب environment

```
Annotations
1 error
Invalid workflow file: .github/workflows/generate_ascii_artwork.yml#L4
The workflow is not valid. .github/workflows/generate_ascii_artwork.yml (Line: 4, Col: 5): Unexpected value ''
```

➢ عدلنا الملف كده:

```yaml
1  name: Generate_ASCII_Artwork
2  on:
3    workflow_dispatch:
4  env:
5  #   # TimestamperBuildWrapper was not converted
6  jobs:
7    build:
```

```yaml
1  name: Generate_ASCII_Artwork
2  on:
3    push:
4    workflow_dispatch:
5  #env:
6  #   # TimestamperBuildWrapper was not converted
7  jobs:
8    build:
```

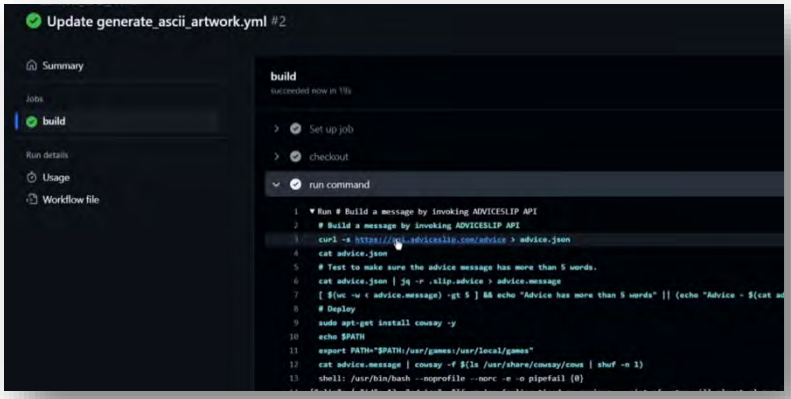
❖ بعد كده commit جديد ← GitHub شغّل الـ pipeline أوتوماتيك.

🖥️ النتيجة

الـ workflow اشتغل فعلاً ومر بمراحل:

- Checkout الكود
- Run script
- استخدم curl + jq + cowsay علشان يولد ASCII Art

وفي الآخر، نفذ نفس اللي كان بيعمله Jenkins Job الأصلي 💪



✅ الخلاصة

- استخدمنا GitHub Actions Importer لتحويل Jenkins pipelines ← GitHub Actions
- جهزنا الأدوات (CLI + Docker + Tokens)
- عملنا Dry Run ← وبعدها Migrate حقيقي.
- في الآخر، تم توليد Workflow جاهز في GitHub وتشغيله بنجاح.

Backing-Up & Restoring Jenkins

- ❑ في الجزء ده، هانشرح بالتفصيل كل الطرق اللي ممكن تستخدمها علشان تعمل Backup لبيئة Jenkins بتاعتك،
- ❑ فياتري ليه ده مهم، وإيه الملفات الأساسية اللي لازم تكون دايمًا ضمن أي Backup plan ؟؟

🧩 أولًا: مقدمة — ليه نعمل Backup لـ Jenkins ؟

- ➢ لازم تعمل نسخ احتياطي دوري علشان تقدر:
  - ترجع Jenkins لو حصل كوارث أو تلف في السيرفر.
  - تحافظ على سلامة البيانات (data integrity)
  - تستعيد ملفات أو إعدادات اتحذفت بالغلط.
- ➢ 💡 ملحووظة مهمة:
  - الـ Jenkins server دايمًا بيخزن كل حاجة (jobs, plugins, secrets) داخل ملفات عادية على النظام
  - فلو السيرفر وقع أو الملف الرئيسي اتلف، كل شغلك ممكن يضيع.

⚙️ ثانيًا: طرق عمل Backup في Jenkins

❖ فيه أكثر من طريقة تعمل بيها نسخ احتياطي، وكل طريقة ليها مميزاتها وعيوبها.

1 Snapshot للـ File System

- ➢ دي أكتر طريقة موثوقة (consistent) للنسخ الاحتياطي.
- ➢ بتستخدم أدوات زي:
  - LVM snapshots على Linux
  - أدوات النسخ في Cloud platforms (زي AWS أو Azure)
  - أو أجهزة التخزين اللي بتدعم snapshot feature

🧠 الفكرة:

الـ Snapshot بياخد صورة كاملة من النظام في لحظة معينة، يعني بيرجع Jenkins بالحالة اللي كان عليها بالضبط وقت ما خدت النسخة.

---

2 Backup Plugins في Jenkins

- ➢ Jenkins عنده Plugins بتساعدك تعمل Backup أوتوماتيكي،
- ➢ بس خلي باالك وحتة من الدهاليز 👇
  - معظم Plugins بتاع الـ Backup بطلت تتحدّث أو فيها مشاكل
  - لكن Thin Backup لسه شغال ومستقر، فلو هتستخدم Plugin ← استخدم ده فقط.

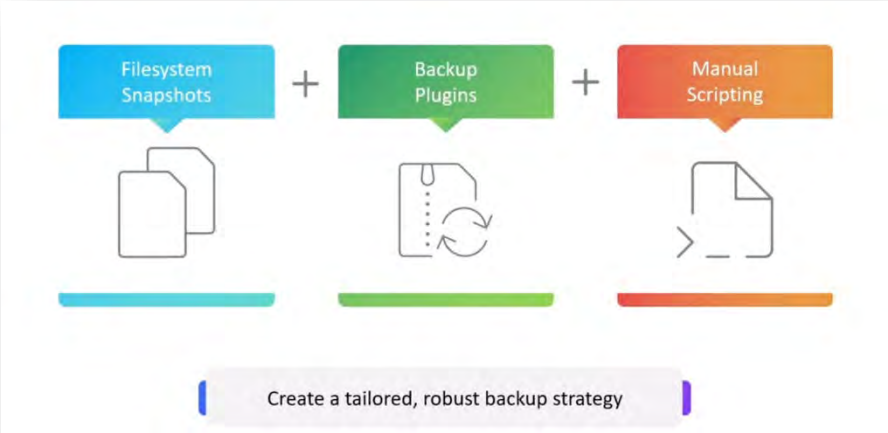
---

3 Custom Backup Script

- ➢ تكتب سكريبت مخصص بنفسك بحيث ينسخ الملفات المهمة من مجلد Jenkins إلى مكان آمن.
  - مثلاً: `cp -r /var/lib/jenkins/config.xml /backup/jenkins/`
- ➢ 💡 نصيحة :
  - اعمل Backup محلي الأول (locally) وبعدين انقل النسخة لمكان بعيد (Remote location) زي S3 Bucket أو NAS storage علشان تأمين إضافي.

💪 Combining Methods

في الآخر ممكن تجمع بين الطرق دي كلها علشان تعمل استراتيجية Backup قوية ومتكاملة تناسب احتياجاتك.



📁 ثالثًا : إيه الملفات اللي لازم أعملها Backup ؟

- قلب Jenkins كله موجود في مجلد واحد : /var/lib/Jenkins
- ده اسمه JENKINS_HOME directory وبيحتوي على كل البيانات، الإعدادات، الـ jobs ، والـ plugins

```
Terminal
$ tree /var/lib/jenkins
├── builds
│   └── [BUILD_ID]
│       ├── build.xml  (build result summary)
│       └── changelog.xml  (change log)
├── config.xml  (Jenkins root configuration file)
├── *.xml  (other site-wide configuration files)
├── fingerprints  (stores fingerprint records, if any)
├── identity.key.enc  (RSA key pair that identifies an instance)
├── jobs
│   └── [JOBNAME]
│       └── config.xml  (job configuration file)
├── plugins  (root directory for all Jenkins plugins)
│   └── [PLUGIN].jpi  (.jpi or .hpi file for the plugin)
├── secrets  (root directory for the secret+key for credential decryption)
│   ├── hudson.util.Secret  (used for encrypting some Jenkins data)
│   ├── master.key  (used for encrypting the hudson.util.Secret key)
│   └── InstanceIdentity.KEY  (used to identify this instance)
├── userContent  (files served under your https://server/userContent/)
└── workspace  (working directory for the version control system)
```

📌 لكن خد بالك:
مش لازم تعمل Backup لكل حاجة جواه،
لأن ده بيكبر الحجم جدًا.

الملفات والمجلدات الأساسية اللي لازم تعملها Backup :

العنصر	الوصف	هل ضروري؟
config.xml	الإعدادات العامة الأساسية لـ Jenkins	✅ ضروري جدًا
/jobs	كل الـ pipelines و build history	✅ ضروري
/plugins	الـ Plugins المثبتة	⚠️ ممكن تعيد تنزيلها لاحقًا
/secrets	مفاتيح SSH وبيانات حساسة	✅ ضروري جدًا
/users	بيانات المستخدمين وصلاحياتهم	✅ مهم
credentials.xml	لو موجود – بيحوي بيانات الدخول المشفرة	✅ جدًا

🧠 ملحوظة من الزمن الجميل :

- بما ان كل ملفات الإعدادات في Jenkins بتكون بامتداد .xml
  - فممكن تعمل أمر بسيط ينسخ كل ملفات الـ XML مرة واحدة زي: `cp /var/lib/jenkins/*.xml /backup/jenkins/`

❌ ملفات مش لازم تعملها Backup :

- workspace/ ← ملفات مؤقتة من كل build
- logs/ ← ممكن تتولد تاني
- caches/
- tools/

السبب إن الملفات دي مؤقتة أو ممكن تتولد بسهولة أثناء الـ restore

---

🧩 رابعًا: نصائح ختامية

- خليك دايمًا تعمل Backup بشكل دوري (weekly أو daily)
- خزن نسخة في مكان بعيد (offsite / remote)
- ركّز على الملفات المهمة بس علشان ما تهدرش مساحة تخزين.
- استخدم Script أو Tool أو Plugin مناسب حسب بيئتك.

✅ الخلاصة النهائية

- Jenkins Backup مش مجرد رفاهية — ده جزء أساسي من إدارة Jenkins بشكل احترافي.
- لو حصلت أي مشكلة:
  - الـ config.xml بيرجع إعدادات السيرفر.
  - مجلد jobs بيرجع الـ Pipelines
  - secrets بيرجع بيانات الأمان.

🔄 وبكده تقدر ترجع Jenkins للحالة الأصلية في دقائق بدل ما تعيد كل حاجة من الأول

Demo Backing-Up & Restoring Jenkins (Thin Backup Plugin)

- ❑ في الجزء ده، هانشرح طريقة عملية ومضمونة لعمل Backup و Restore لبيئة Jenkins باستخدام Plugin بسيط لكنه قوي جدًا اسمه **Thin Backup**
- ❑ وده يعتبر الحل الرسمي الموصى به بدل ما تكتب سكريبتات يدوية معقّدة.

🧩 أولًا: مقدمة

- إزاي تنزل الـ plugin
- إزاي تعمله إعداد.
- إزاي تعمل Backup
- وبعدين إزاي ترجّع الملفات لو حصلت مشكلة (Restore)

⚙️ ثانيًا: تثبيت Thin Backup Plugin

ThinBackup | Jenkins plugin

- ➢ الخطوة الأولى : زي أي Plugin في Jenkins ← روح إلى:
  Manage Jenkins → Manage Plugins → Available
- ➢ ودوّر على: Thin Backup
- ➢ وبعدها اضغط Install without restart

💡 الهدف من الـ Plugin :

- بيعمل Backup للـ global configuration (الإعدادات العامة)
- وكمان للـ job-specific configurations (إعدادات كل Job أو Pipeline)
- تقدر تعمل له جدولة تلقائية (schedule)
- ومش بيخزن حاجات زيادة — بس أهم الملفات الضرورية.

⚙️ ثالثًا: إعداد الـ Plugin بعد التثبيت

- ➢ بعد التثبيت، تروح إلى : Manage Jenkins → System Configuration
- ➢ وتكتب في خانة البحث Thin :
  - هتلاقي إعداد اسمه: Thin Backup configuration

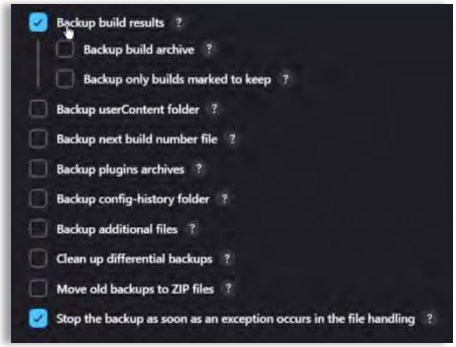
📁 احنا هنا استخدمنا المسار ده كمكان لحفظ النسخ الاحتياطية: **var/lib/jenkins/JENKINS_BACKUP/**

🔹 **ملاحظة مهمة:**

لو المجلد ده مش موجود، الـ Plugin هيعمله تلقائي أول مرة بيشغّل فيها Backup

---

🕒 رابعًا: إعدادات إضافية مهمة



الإعداد	المعنى
Schedule (Cron)	تحدد كل قد إيه تعمل Backup (مثلاً كل يوم أو كل أسبوع).
Differential Backup	يعمل نسخة فيها بس التغييرات الجديدة من آخر Backup — علشان توفّر مساحة.
Max Number of Backup Sets	تحدد أقصى عدد نسخ يحتفظ بيها قبل ما يمسح القديمة.
Exclude Files (Regex)	تستبعد ملفات معينة من النسخ الاحتياطي.

🧠 يعني تقدر تتحكم في كل تفصيلة حسب بيئتك واحتياجاتك.

💾 خامسًا: تنفيذ أول Backup يدوي

- ➢ بعد حفظ الإعدادات، نروح إلى: Manage Jenkins → Tools and Actions → Thin Backup
- ➢ هتلاقي هناك:
  - Backup Now 🔹
  - Restore 🔹

🚀 نضغط على Backup Now

- ➢ هتلاحظ ان العملية دي مش بتعرض Logs جوه الصفحة نفسها، لكن تقدر تتابعها من :
  Dashboard → Manage Jenkins → System Logs → All Jenkins Logs
- وهناك بتشوف Log بيقول مثلًا:

```
Loaded all jobs
Nov 10, 2024 4:34:21 PM INFO jenkins.InitReactorRunner$1 onAttained
Configuration for all jobs updated
Nov 10, 2024 4:34:21 PM INFO jenkins.InitReactorRunner$1 onAttained
Completed initialization
Nov 10, 2024 4:34:21 PM INFO hudson.model.UpdateCenter$CompleteBatchJob run
Completed installation of 1 plugins in 2.8 sec
Nov 10, 2024 4:37:03 PM INFO org.jvnet.hudson.plugins.thinbackup.ThinBackupMgmtLink doBackupManual
Starting manual backup.
Nov 10, 2024 4:37:03 PM INFO org.jvnet.hudson.plugins.thinbackup.backup.HudsonBackup
No previous full backup found, thus creating one.
Nov 10, 2024 4:37:03 PM INFO org.jvnet.hudson.plugins.thinbackup.backup.HudsonBackup backupJobsDirectory
Found 16 jobs in /var/lib/jenkins/jobs to back up.
Nov 10, 2024 4:37:03 PM INFO org.jvnet.hudson.plugins.thinbackup.backup.HudsonBackup backupJobsDirectory
Found 3 jobs in /var/lib/jenkins/jobs/Gitea-Organization/jobs to back up.
Nov 10, 2024 4:37:21 PM INFO org.jvnet.hudson.plugins.thinbackup.ThinBackupPeriodicWork backupNow
Backup process finished successfully.
```

يعني كل الـ Jobs والـ Folders تم نسخهم بنجاح ✅

📂 سادسًا: التحقق من ملفات النسخ الاحتياطي

- ➢ بعدها راح على الـ VM وكتب:
  - cd /var/lib/jenkins
  - ls
- ➢ ولقى مجلد اسمه: jenkins_backup/
  - جواه ملفات كتير جدًا زي:
    - config.xml
    - jobs/
    - plugins/
    - users/

وده دليل إن الـ Plugin فعلاً أنشأ النسخة الاحتياطية الكاملة للـ Jenkins instance

---

🧩 سابعًا : تجربة الـ Restore عمليًا

❖ لو حبيت تختبر هل الـ Restore فعّال فعلًا.

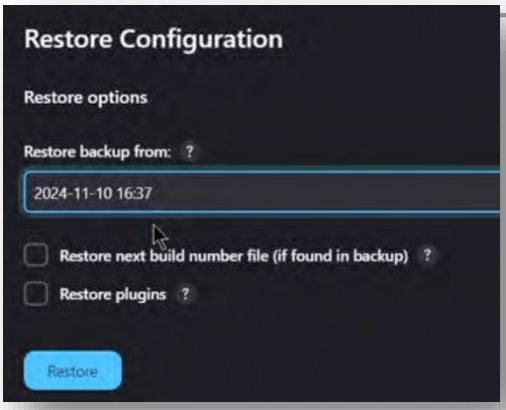
- ➢ عمل حركة بسيطة جدًا:
  1. راح على Jenkins Dashboard
  2. اختار Job اسمه Monitor Jenkins
  3. لاحظ إنه فيه حوالي 38 Build history
  4. شاف إعداداته (كانت مجرد Pipeline "Hello World")
  5. بعدين حذف الـ Job دي بالكامل

🔄 ثامنًا: استرجاع النسخة (Restore)

- ➢ رجع تاني إلى : Manage Jenkins → Thin Backup → Restore
  هناك بيظهرلك كل الـ Backups اللي اتعملت.
- ➢ تختار النسخة اللي عايز ترجعها وتضغط على Restore

🧠 تقدر تختار ترجع الـ Plugins كمان، بس ساب الخيار ده مقفول .

🚀 بعد ماتعمل Restore ، لازم تعمل Restart Jenkins server علشان التغييرات تتطبّق.



🚀 تاسعًا: بعد الـ Restart

➢ بعد ما Jenkins رجع، عمل Login وشاف المفاجأة 👇
الـ Job اللي كان اسمه Monitor Jenkins رجع تاني بكامل البيانات والتاريخ.
يعني:
✅ رجعت كل الإعدادات.
✅ رجعت كل الـ Builds القديمة.

⚖️ عاشرًا: الفرق بين Backup كامل و Backup جزئي

❖ الفرق هو نقطة مهمة جدًا في الإعدادات:

- لو فعّلت خيار Backup Build Results ← هيتنسخ كل نتائج الـ Builds والـ Artifacts (وده بياخد مساحة كبيرة جدًا)
- لو قفلته ← هيتنسخ بس الـ Job configuration (يعني السكريبتات والإعدادات بدون نتائج التنفيذ)

🧠 الأفضل تستخدم النسخ الجزئي لو المساحة محدودة أو النسخ اليومي متكرر.

📄 أخيرًا: ملخص العملية

❖ في الآخر رجع على صفحة الـ Thin Backup

- وقالك إنك دلوقتي تقدر:
  - تعمل Backup جديد يدوي.
  - تعمل Restore لأي نسخة قديمة.
  - وكل ده من واجهة Jenkins بدون أوامر خارجية.

✅ الخلاصة النهائية

❖ 📦 Thin Backup Plugin هو أداة بسيطة لكنها قوية جدًا لإدارة النسخ الاحتياطية في Jenkins
❖ بتساعدك تحافظ على بيئتك في أمان، وتقدر ترجع كل حاجة بضغطة زر واحدة.

🔹 مميزاته:

- سهل التثبيت والإعداد.
- بيدعم الجدولة الآلية.
- بيسمح بالـ full أو differential backup
- بيقدر يرجّع كل حاجة حتى الـ Build history

🔹 نصيحة احترافية:

اعمل Full Backup أسبوعي + Differential Backup يومي، وخلي النسخ تتحفظ في Storage خارجي أو Cloud (زي AWS S3)

Validating Backup with Only Linux

أولاً: مقدمة

- الفكرة هنا إننا مش هنستخدم أي Plugin أو أداة خارجية
- هنعتمد فقط على Linux Commands علشان ناخد نسخة احتياطية من فولدر Jenkins الرئيسي— وده اللي بيتخزن فيه كل الإعدادات، الـ Jobs ، الـ Plugins ، وكل حاجة تخص Jenkins
- المكان الأساسي للـ Jenkins Home هو : var/lib/Jenkins/
  - لو الفولدر ده راح أو تلف، السيرفر كله مش هيقوم تاني.
  - علشان كده لازم نعمل منه Backup دوري.

ثانيًا: خطوات التنفيذ خطوة بخطوة

```
root /var/lib
❯ ll | grep -i jenkins
drwxr-xr-x 28 jenkins   jenkins        12288 Feb  7 07:05 jenkins/
-rw-r--r--  1 root      root      2209844982 Feb  3 09:00 jenkins-EOF.tar.gz
drwxr-xr-x 10 jenkins   jenkins         4096 Feb  3 08:59 jenkins.bak/
```

الخطوة 1: تحديد مكان الـ Jenkins Home

- في البداية عمل : ls | grep -i jenkins
  - الهدف من الأمر ده هو التأكد من مكان فولدر Jenkins الفعلي .
- طلع عنده 3 فولدرات فيهم الاسم"jenkins" ،
  لكن المهم هو : var/lib/Jenkins/ وده الفولدر اللي فيه كل حاجة متعلقة بـ Jenkins

---

الخطوة 2: إنشاء نسخة احتياطية (Backup)

- قرر يستخدم أمر tar علشان يعمل أرشيف مضغوط لكل محتويات Jenkins :
  - tar -czvf /tmp/jenkins_home_$(date +%F).tar.gz /var/lib/jenkins

```
root /var/lib
❯ tar zcvf /tmp/jenkins_home-$(date +"%Y%m%d%H%M").tar.gz jenkins
```

شرح الأوامر:

- tar ← الأداة الخاصة بإنشاء أرشيفات.
- -c ← create archive
- -z ← ضغط باستخدام gzip
- -v ← verbose (يعرض تقدم العملية)
- -f ← تحديد اسم الملف الناتج.
- /tmp/jenkins_home_$(date +%F).tar.gz ← اسم ملف النسخة الاحتياطية + بيضيف التاريخ في الاسم.
- /var/lib/jenkins ← هو المصدر اللي بننسخه.

ملاحظة مهمة (Best Practice) :

- قبل ما تبدأ النسخ، لازم توقف Jenkins مؤقتًا: sudo systemctl stop Jenkins
- ده يمنع أي ملفات من التغيير أثناء النسخ، علشان النسخة تطلع سليمة ومفيهاش فساد (corruption)

الخطوة 3: التحقق من النسخة الاحتياطية

- بعد حوالي دقيقتين ونص، الملف اتولد في المسار /tmp
  - تحقق منه بالأمر: ls -lh /tmp | grep jenkins

وظهر ملف اسمه مثلاً: jenkins_home_2025-10-23.tar.gz

```
root /var/lib took 2m36s
❯ ll /tmp/jenkins_home-202502070714.tar.gz
-rw-r--r-- 1 root root 2172715434 Feb  7 07:17 /tmp/jenkins_home-202502070714.tar.gz
```

كده النسخة اتعملت بنجاح .

الخطوة 4: عمل نسخة مطابقة (Copy) بدل من الضغط (طريقة اخرى تعمل بيها الـ Backup)

- فيه طريقة تانية — بدل ما تعمل Tar ، ممكن تعمل Copy مباشر للفولدر بالكامل (لو عندك مساحة كفاية):
  - `cp -r /var/lib/jenkins /tmp/jenkins_backup`

الـ -r معناها recursive ← يعني انسخ كل الملفات والفولدرات الفرعية.

المهمة دي خدت شوية وقت، بس في النهاية لما دخل على /tmp/jenkins_backup لقى كل الملفات متنسخة فعلاً.

الخطوة 5: إعداد بيئة اختبار للنسخة (Validation) (عمل الخطوة دي علشان يتحقق من صحة النسخة الاحتياطية)

- دلوقتي نبدأ نختبر هل الباك أب ده فعلاً شغال ولا لأ.
- الفكرة ببساطة:

هنشغل Jenkins جديد مؤقتًا، ونخليه يقرأ من فولدر النسخة الاحتياطية بدل الفولدر الأصلي.

أول خطوة: تعريف متغير الـ Jenkins Home

- `export JENKINS_HOME=/tmp/jenkins_backup`

ده بيقول لأي عملية Jenkins بعد كده إنها تستخدم /tmp/jenkins_backup كأنها الـ home الأساسي.

الخطوة 6: تشغيل Jenkins باستخدام WAR file

- كنا نزلنا قبل كده ملف Jenkins WAR (من موقع Jenkins الرسمي) .. وده ملف بيشغلك Jenkins من غير تثبيت فعلي.
- اتأكد إنه موجود: `ls | grep jenkins.war`
- وبعدين شغّله كده: `java -jar jenkins.war --httpPort=99999`

شرح سريع:

- `java -jar jenkins.war` ← بيشغّل Jenkins كمشروع Java
- `--httpPort=99999` ← يحدد البورت اللي السيرفر يشتغل عليه.

هيستخدم تلقائيًا متغير البيئة $JENKINS_HOME اللي عرفناه فوق.

Jenkins دلوقتي بيبدأ من النسخة الاحتياطية اللي في /tmp/jenkins_backup

---

الخطوة 7: التحقق من خلال الـ Browser

- بعد التشغيل، اكتب في المتصفح : http://localhost:99999 وظهرله نفس صفحة تسجيل الدخول الأصلية
- دخل بنفس الـ credentials القديمة، ولقى كل الـ Jobs والـ Pipelines اللي كانت موجودة!

ده تأكيد إن النسخة الاحتياطية فعلاً شغالة وسليمة 100%.

ملاحظات عملية (Best Practices + Tricks)

1. أوقف Jenkins قبل الـ backup :
   علشان تتجنب أي ملفات تتغير أثناء النسخ.
2. جدولة النسخ تلقائيًا:
   ممكن تعمل Cron job :
   - `0 2 * * * tar -czf /tmp/jenkins_home_$(date +\%F).tar.gz /var/lib/jenkins`
     - دي بتعمل باك أب يومي الساعة 2 صباحًا.
3. خزّن الباك أب في مكان خارجي (Remote Storage) :
   زي AWS S3 أو NAS أو Google Drive

4. ⚙️ استخدم Jenkins Home Variable دايمًا لما تشغل نسخة تجريبية.
5. 🧪 جرّب النسخة الاحتياطية كل فترة:
علشان تتأكد إنها بتشتغل ومفيهاش corruption

🧩 ملخص شامل

- استخدمنا tar أو cp لعمل نسخة احتياطية من /var/lib/Jenkins
- أوقفنا Jenkins قبل النسخ لتجنّب تلف البيانات.
- جرّبنا النسخة الاحتياطية باستخدام متغير البيئة $JENKINS_HOME
- شغّلنا Jenkins جديد باستخدام ملف jenkins.war على بورت مختلف.
- تأكدنا من أن كل الـ Jobs و الإعدادات رجعت سليمة.

🎯 الخلاصة:

- الطريقة دي بسيطة، سريعة، ومناسبة لأي بيئة حتى لو مفيهاش Plugins
- ومجرد إنك قدرت تشغّل Jenkins من النسخة الاحتياطية، فده دليل إن عملية الباك أب ناجحة تمامًا 💪

Upgrading Jenkins

- ❑ الجزء ده بيتكلم عن إزاي تعمل Upgrade للـ Jenkins Instance بتاعتك بطريقة آمنة ومنظمة.
- ❑ وده مهم جدًا لأي System Administrator أو DevOps Engineer عشان تحديث Jenkins بدون ما تسبب downtime أو تلف في الـ Jobs

---

🧭 أولاً: مقدمة

الـ Jenkins بيصدر تحديثات دورية لضمان:

- معالجة الثغرات الأمنية (Security Fixes)
- إضافة تحسينات ومميزات جديدة (Enhancements)
- إصلاح الـ Bugs والمشاكل القديمة (Bug Fixes)

فيه نوعين من الإصدارات:

1. LTS (Long-Term Support) : تصدر كل 12 أسبوع، أكثر استقرارًا ومناسبة للإنتاج.
2. Weekly Releases : تصدر كل أسبوع لإصلاح سريع أو تحديثات جديدة.

الجزء ده بيشرح إزاي:

1. تعرف إن فيه نسخة جديدة متاحة.
2. تعمل Upgrade بشكل آمن.
3. تمنع أي مشاكل أثناء العملية.

⚙️ ثانيًا: خطوات التنفيذ (Step-by-Step Execution)

📋 الخطوة 1: معرفة الإصدار الحالي والجديد

- ➢ أول حاجة بتفتح Jenkins Dashboard → Manage Jenkins
  - هتلاقي تنبيه بيقولك إن فيه نسخة جديدة متاحة.
- ➢ الإصدار الحالي : 2.462.1
- ➢ الإصدار الجديد: 2.492.1
- ➢ بعد كده اضغط على Change Log علشان تشوف التغييرات اللي اتضافت زي الـ bug fixes أو الـ security patches .. وكمان تتأكد إن التحديث آمن قبل ما تبدأ.

📋 الخطوة 2: تحميل نسخة الـ WAR الجديدة

- ➢ Jenkins بيتثبّت أساسًا من ملف WAR فبالتالي كل اللي هتعمله إنك تنزل النسخة الجديدة من الموقع الرسمي.
- ➢ ولو انت مش متأكد اذا كان الفايل اصلا عندك ولا لأ .. او ناسي مكانه ← فمن الكونسول:

```
root ~ via
❯ ps aux | grep jenkins
jenkins    16561 12.8  7.7 10876676 1260320 ?    Ssl  07:27   0:45 /usr/bin/java -Xms1G -Xmx1G -jar /usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war -
ttpPort=8080
root       18153  0.0  0.0   4088  1976 pts/6    S+   07:33   0:00 grep --color=auto jenkins
```

- o cd /usr/share/java
- o ls | grep jenkins

- هتلاقي الملف: jenkins.war
- ➢ هنحذف القديم وننزّل الجديد : (قبل ماتعمل الخطوة دي لاااازم تنفذ الخطوات رقم (4&3))

- o sudo rm -f /usr/share/java/jenkins.war
- o sudo wget https://get.jenkins.io/war-stable/2.492.1/jenkins.war

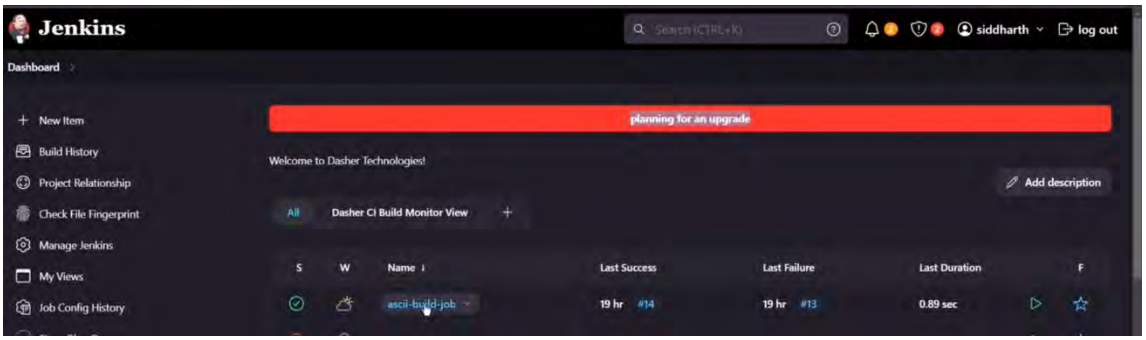
💡 نصيحة : دايمًا نزل نسخة الـ LTS إلا لو أنت بتجرب أو بتختبر تحديث.

الخطوة 3: تقليل التأثير على المستخدمين (Prepare for Shutdown)

- قبل ما تعمل Restart أوUpgrade ، لازم تبلغ المستخدمين وتمنع أي Build جديد من التشغيل.
- ادخل على : Manage Jenkins → Prepare for Shutdown
- واكتب رسالة زي:

"Planning for an upgraded"

- اللي بيحصل دلوقتي إن:
  - الـ Jobs اللي شغالة بتكمل عادي.
  - الـ Jobs الجديدة بتفضل "Pending" ومش هتبدأ.
  - المستخدمين هيشوفوا Banner في كل الصفحات إن النظام في وضع التحديث.



وده بيوفر أمان كامل للبيئة الإنتاجية أثناء التحديث.

الخطوة 4: إيقاف Jenkins مؤقتًا

- الخطوة دي ضرورية جدًا قبل استبدال ملفات النظام : sudo systemctl stop jenkins
  - الهدف منها هو التأكد إن Jenkins مش شغال أثناء استبدال ملف الـ WAR علشان مايحصلش file corruption

---

الخطوة 5: استبدال ملف WAR القديم بالجديد

بعد ما وقفنا Jenkins ، نحذف الملف القديم ونضيف الجديد:

- sudo rm -f /usr/share/java/jenkins.war
- sudo cp ~/Downloads/jenkins.war /usr/share/java/

أو لو نزلته مباشرة، هيكون بالفعل في مكانه الصحيح.

الخطوة 6: تشغيل Jenkins من جديد

- نشغّل الخدمة تاني : sudo systemctl start jenkins
- ونتأكد إنها اشتغلت: sudo systemctl status jenkins
- لو كل حاجة تمام، ادخل المتصفح: http://<your-server>:8080
- وسجّل الدخول، وبعدها روح: Manage Jenkins → About Jenkins
- هتلاقي الإصدار الجديد ظاهر ✅
  مثلاً: Jenkins 2.492.1

الخطوة 7: التحقق بعد التحديث (Post-Upgrade Validation)

✅ تأكد إن كل الـ Jobs موجودة وسليمة.

✅ شوف إن Plugins شغالة ومفيش warning messages

✅ لو فيه أي Plugins فيها تحديثات، اعمل لها upgrade برضو من : Manage Jenkins → Manage Plugins → Updates

💡 نصائح وملاحظات عملية (Best Practices + Real Use Cases)

1. 📄 اعمل Backup قبل أي Upgrade
   - استخدم Tar أو ThinBackup Plugin علشان تقدر ترجع لو حصل مشكلة.
2. 🧱 اختبر التحديث على Test Jenkins قبل الإنتاج
   - جرب النسخة الجديدة على سيرفر تجريبي قبل ما تحدث الرئيسي.
3. 🕐 اختار وقت التحديث بحكمة
   - اعمله في وقت الـ Low Traffic أو خارج ساعات العمل لتقليل التأثير.
4. 🔐 تابع صفحة الأمن الرسمية لـ Jenkins
   - https://www.jenkins.io/security/advisories/
     علشان تعرف أي ثغرات أو تحديثات مهمة.
5. 💉 راجع Logs بعد التشغيل
   - استخدم: sudo journalctl -u jenkins -f
   - عشان تتأكد إن السيرفر اشتغل بدون Errors

🧩 الملخص الشامل

- 🔹 عرفنا إزاي Jenkins بيصدر تحديثات دورية (LTS / Weekly)
- 🔹 شفنا طريقة التأكد من وجود إصدار جديد عن طريق Dashboard
- 🔹 استخدمنا خيار Prepare for Shutdown لتجهيز النظام بأمان.
- 🔹 أوقفنا Jenkins ، استبدلنا ملف WAR ، وشغلناه من جديد.
- 🔹 تأكدنا من الإصدار الجديد بعد الـ Restart
- 🔹 واتكلمنا عن أهم الـ Best Practices قبل وبعد الـ Upgrade

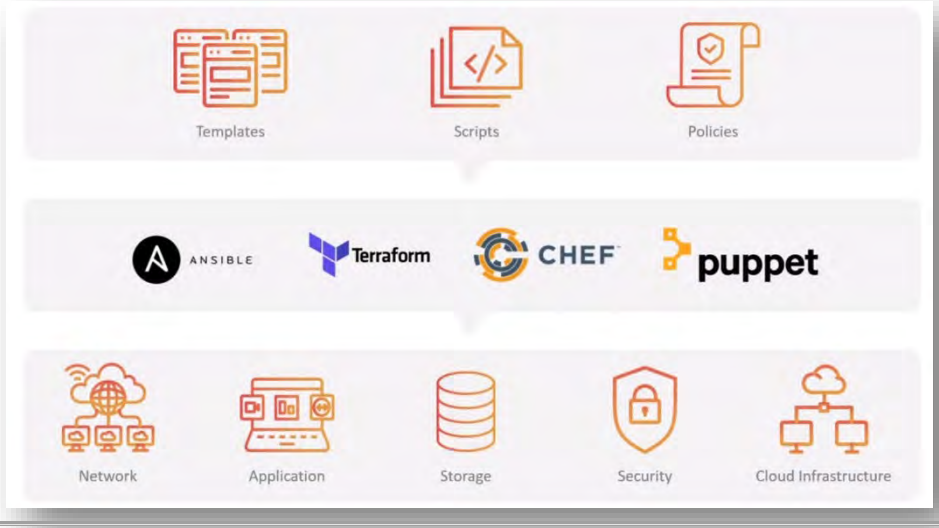
🎯 الخلاصة:

- ❖ تحديث Jenkins عملية سهلة لكنها حساسة.
- ❖ المفتاح هو التخطيط المسبق، التجهيز الآمن، والتحقق بعد التنفيذ.
- ❖ لو مشيت على الخطوات دي، تقدر تحدث Jenkins بثقة بدون ما توقف الإنتاج لحظة واحدة 💪

Jenkins Configure as code (JCasC)

🎯 أولاً: الفكرة العامة

- ❑ قبل ما ندخل Jenkins Configuration as Code، لازم نفهم أصل الفكرة اللي جاي منها:
  الـ Infrastructure as Code (IaC) دي ببساطة فكرة إنك تدير البنية التحتية بتاعتك زي السيرفرات، الشبكات، الـ VMs ... عن طريق كود بدل ما تعمل كل حاجة يدوي.
- ❑ يعني بدل ما تفتح لوحة تحكم وتضيف VM أو user أو network ، تكتب كود في Terraform أو Ansible أو Puppet
- ❑ الكود ده لما تشغله بيبني لك نفس الـ setup بنفس الشكل كل مرة .. وده بيخلي البيئة repeatable، reliable، scalable



⚙️ بعد كده بنطبق نفس الفكرة على Jenkins

- ➢ زي ما الـ IaC بينفذ بنية تحتية بالكود، Jenkins كمان عنده حاجة اسمها Configuration as Code (JCasC) اللي بتخليك تدير إعدادات Jenkins بالكامل عن طريق ملف YAML
- ➢ الفكرة هنا:

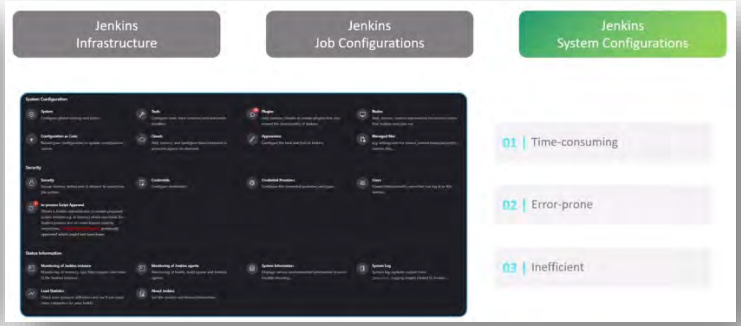
بدل ما تفتح Jenkins UI وتدخل manually تعمل Configurations أو تضيف Plugins أو Credentials أو Users ، فممكن تعمل كل ده في ملف واحد YAML .. ولما Jenkins يقرأ الملف ده، يكوّن نفسه تلقائياً من غير أي تدخل منك.

🧩 الأدوات اللي Jenkins بيديهالك عشان تدير الإعدادات بالكود:

- Jenkins CLI ← تتحكم في Jenkins من التيرمنال.
- REST API ← تقدر تتكامل مع Jenkins من تطبيقات أو سكريبتات.
- Client Libraries (زي Java/Python/Go SDKs)
- Integration with IaC Tools ← تقدر تستخدم Ansible أو Chef تبني Jenkins environment بالكامل بالكود.
- Docker Image ← تقدر تبني Jenkins كـ Container portable وسهل النقل.

🧱 المشكلة اللي بيحلّها JCasC :

- Jenkins UI بيكون عملي في الأول... لكن لما عدد الـ Jobs يكبر، كل تعديل بسيط في الإعدادات بيبقى مجهد جدًا.
  - مثلا عندك 100 Job ؟ لازم تعدل كل واحد فيهم بإيدك؟ كارثة 😅 !
- عشان كده بدأوا حلول زي:
  - Job DSL Plugin ← تكتب الـ jobs بالكود Groovy
  - Job Builder (YAML/JSON) ← ملفات بسيطة تنشئ وتدير بيها Jobs
  - Pipeline as Code ← الـ Jenkinsfile اللي بنكتب فيه stages
  - Multibranch Pipeline ← تبني الـ Jobs تلقائيًا بناءً على تغييرات في Git branches
- ودي كلها كانت مقدمة للفكرة الكبرى : Jenkins Configuration as Code Plugin



⚠️ قبل JCasC :

- ❖ الناس كانت بتستخدم Groovy init scripts عشان تعمل Configurations تلقائياً.
- ❖ كانت قوية فعلاً، بس مشكلتها انها كانت معقدة جدًا لأنك محتاج تفهم الـ Jenkins API بعمق وتتعامل مع كود Groovy

يعني لو مش خبير Jenkins Internals ، صعب تعمل Automation للإعدادات كلها صح.

💡 الحل Jenkins Configuration as Code (JCasC) :

- ❖ لما نزل الـ Plugin ده سنة 2018 في DevOps World Conference، غيّر اللعبة كلها.
  - بتكتب إعدادات Jenkins في ملف YAML
  - نفس الإعدادات اللي كنت بتعملها من UI
  - والملف ده Jenkins بيقراه ويطبقه تلقائيًا من غير ما تفتح أي صفحة Web

---

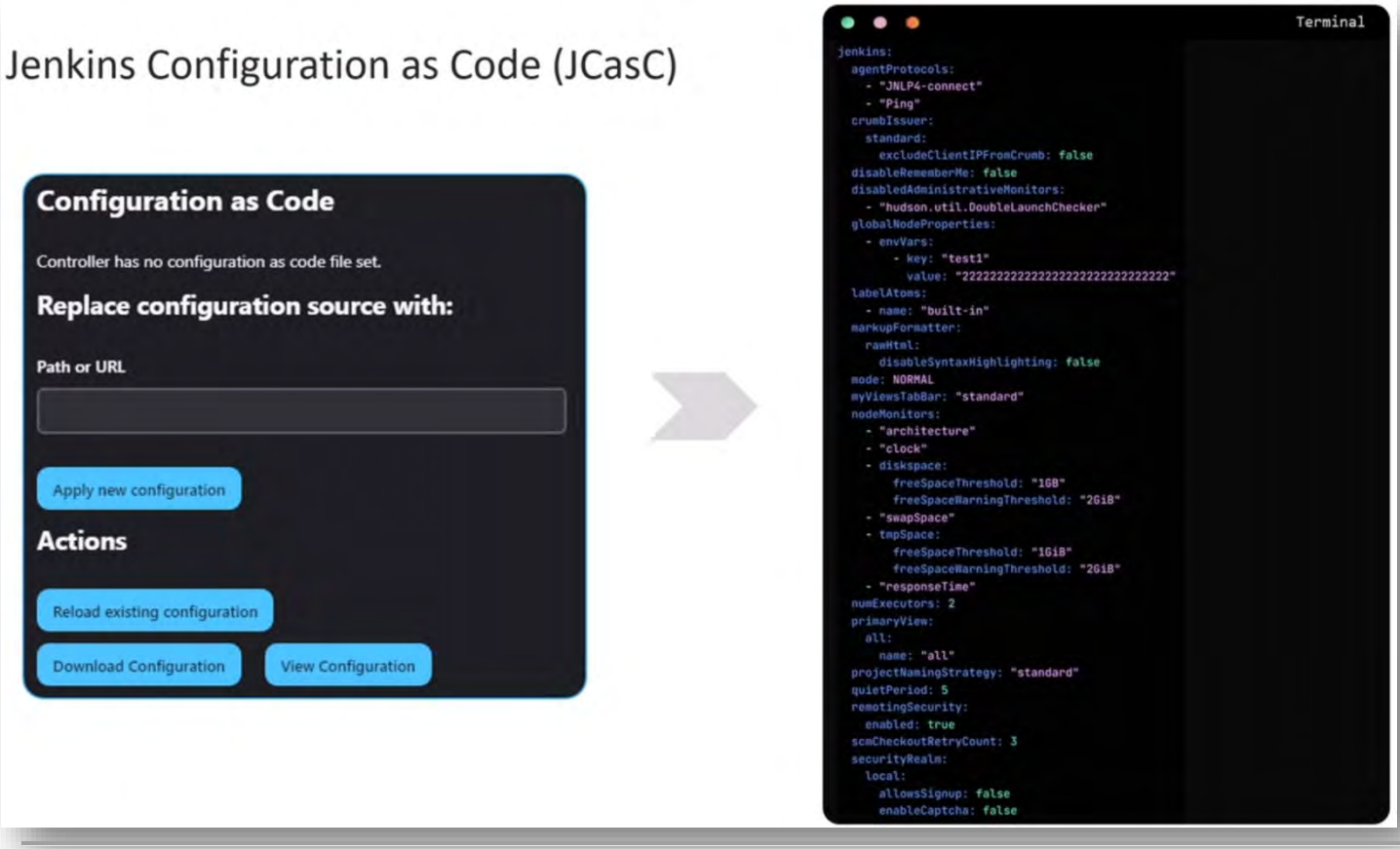
🔧 مميزات Jenkins Configuration as Code :

1. Version Control Integration ← تقدر تحفظ ملف الإعدادات في Git ، وتعمل Rollback أو Track للتعديلات.
2. Consistency ← أي Jenkins جديد تقدر تبنيه بنفس الإعدادات بمجرد نسخ ملف YAML
3. No manual clicks ← كل حاجة بالكود، مفيش UI errors
4. No restart needed ← تقدر تطبّق التعديلات بدون Restart
5. Human error أقل ← الكود ثابت، قابل للمراجعة، سهل التصحيح.

📦 بعد تثبيت الـ Plugin :

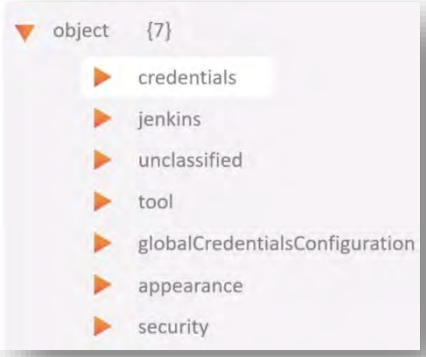
- ❖ هتلاقي تحت Manage Jenkins → Configuration as Code
  زرار اسمه "View Configuration"
- ❖ لما تضغط عليه Jenkins هيولّد لك ملف YAML كامل فيه كل الإعدادات الحالية.
  تقدر:
  - تحفظه في Git كنسخة backup
  - أو تعدله وتعيد رفعه عشان يعمل Update للإعدادات.

📌 دا طبعا بيبقي مفيد بعد كدا كمرجع أو في عملية الـ rollbacks



📑 محتويات ملف الـ YAML :

- ❑ مبدئيا كدا ممكن تعتبر ان الـ Jenkins Configuration as Code YAML file بمثابة الـ blueprint بتاع الـ Jenkins Configuration بتاعك



1. Jenkins :
   - o الإعدادات العامة (زي الـ System Message ، Global Configurations)
2. Tool :
   - o تعريف الـ Tools (زي Maven أو JDK paths)
3. Unclassified :
   - o إعدادات Plugins
4. Credentials :
   - o كل بيانات الدخول والتوكنز.
5. Views:
   - o ممكن تعمل Views في الـ Dashboard للـ Jobs (زي "Active Jobs" أو "All Jobs")

🧠 مثال عملي:

❖ في ملف YAML تقدر تكتب مثلاً:

❖ يعني بمجرد Jenkins يقرأ الملف ده، هيعمل الاتي :

- يضيف Message في الـ Dashboard
- يعرف Tool اسمه Maven3
- ويعمل View فيها الـ Jobs دي
  - o واحدة بتعرض الـ Active Jobs
  - o التانية بتعرض كل الـ Jobs

🚨 كل ده من غير ما تفتح صفحة واحدة في الـ UI ! 🔥

🧩 ملاحظات مهمة:

- ملف YAML ممكن يحتوي على Sections إضافية حسب Plugins اللي عندك.
- Jenkins بيقرأ الملف ده وقت التشغيل أو لما تعمل Apply من الـ UI
- تقدر تحطه في الـ GitHub وتخليه جزء من CI/CD بتاعك.

✅ الخلاصة:

➢ Jenkins Configuration as Code (JCasC) = الطريقة الحديثة لإدارة Jenkins بالكامل بالكود.

➢ بدل ما تعمل Configuration manually :

- اكتبها في YAML
- احفظها في Git
- خلي Jenkins يبني نفسه أوتوماتيك.

النتيجة؟

💪 بيئة Jenkins ثابتة، قابلة للتكرار، وسهلة الاسترجاع في أي وقت.

Demo Configure and Explore (JCasC)

🧠 أولاً: المقدمة — ليه بنستخدم Configuration as Code فيJenkins ؟

- ❑ Jenkins دايمًا بيحتاج إعدادات كتير جدًا سواء في النظام نفسه أو في الـ plugins
- ❑ اللي اشتغل على Jenkins UI قبل كده عارف إن كل حاجة فيها parameters كتير جدًا
  Users, Nodes, Plugins, Credentials, Cloud Connections ....إلخ.
- ❑ فده بيخلي عملية الإعداد مرهقة جدًا وممكن يحصل فيها أخطاء بشرية.
- ❑ عشان كده اتعمل الـ Configuration as Code Plugin (JCasC) وده ببساطة Plugin بيخليك تعمل إعداد Jenkins بالكامل من خلال ملف YAML واحد readable وسهل جدًا.
- ❖ يعني بدل ما تفتح UI وتظبط كل حاجة manually ، بتكتبها في YAML و Jenkins يقرّاه ويطبّق الإعدادات كلها أوتوماتيك.

⚙️ ثانياً: تثبيت الـ Plugin

Configuration as Code | Jenkins plugin

1. تروح على Manage Jenkins → Plugins → Available Plugins
2. تكتب Configuration as Code :
3. تعمل Install
4. Jenkins هيسألك تعمله Restart ← وافق، وبعد ما يعمل Restart ارجع وسجّل الدخول تاني.

◆ معلومة مهمة:

الـ Plugin عنده Documentation ممتازة جدًا، وكمان GitHub Repository فيه Examples جاهزة لأي نوع إعداد عايزه

🧩 بعد التثبيت

- ➢ لما ترجع على Manage Jenkins → System Configuration
- ➢ هتلاقي اختيار جديد اسمه Configuration as Code لما تفتحه 👇
- ➢ هتلاقي عندك 3 اختيارات رئيسية:

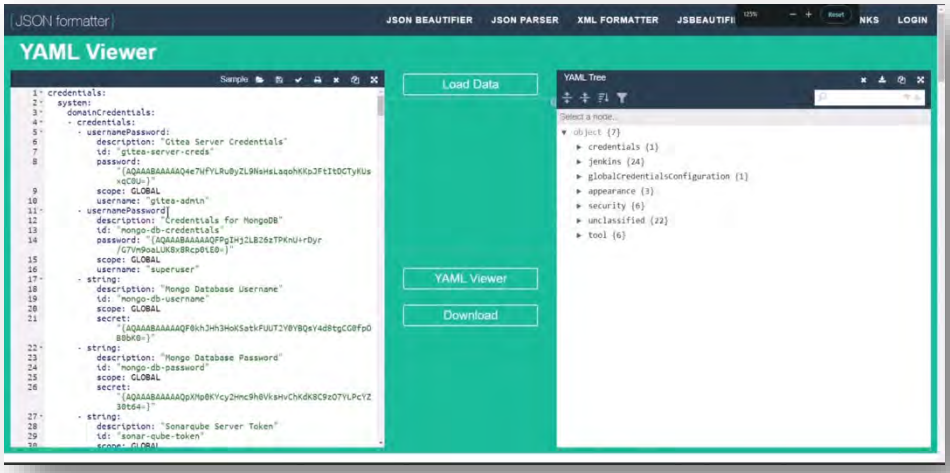
1. Replace Configuration ← تستخدمها لما تحب ترفع ملف YAML من الجهاز أو من GitHub
2. View Configuration ← تعرض الإعدادات الحالية اللي Jenkins بيستخدمها (في شكل YAML)
3. Documentation & Reload Options ← لمراجعة الملفات أو إعادة تحميلها.

📜 ثالثاً: عرض الإعدادات الحالية كـ YAML

- ❖ لما تختار View Configuration
  Jenkins بيولّد ملف YAML ضخم جدًا فيه كل إعدادات الـ instance الحالية بتاعتك.
- ❖ نسخ الملف كله وفتحه في YAML Viewer علشان يشوف الـ structure

Best YAML Viewer Online

- الملف بيحتوي على 7 Objects رئيسية:



🧱 رابعاً: تحليل الملف خطوة بخطوة

- **Credentials**
  - فيه كل الـ credentials بأنواعها المختلفة (AWS access keys, Kube config, MongoDB, SonarQube)
  - الـ secrets مش باينة لأنها بتكون hashed (متشفّرة)
- **Jenkins**
  - فيه إعدادات زي:
    - agent protocols (JNLP, connect, ping)
    - authorization strategy (مين ليه access و على إيه)
    - تعريف الـ nodes (عدد executors ، labels، mode، cloud configs)
    - الـ system message اللي بتظهر على الـ dashboard
- **appearance**
  - بتتحكم في الثيمات (dark/light mode) والـ UI look
- **unclassified**
  - فيها إعدادات الـ Plugins كلها (زي Audit Trail, Slack Notifier, Prometheus, SonarQube)
- **tools**
  - بتحدد أماكن الأدوات زي:
    - Maven
    - Git
    - Node.js
    - SonarQube Scanner
    - Dependency Check Tools

Object	المعنى
credentials	كل الـ credentials والـ secrets (AWS, GitHub, SonarQube...)
jenkins	إعدادات النظام نفسه (agents, nodes, security, system message...)
globalCredentialConfiguration	إعدادات عامة للـ credentials
appearance	الثيمات والألوان والمظهر العام
security	إعدادات الأمان والـ tokens
unclassified	إعدادات الـ plugins
tools	الأدوات اللي Jenkins بيستخدمها (Maven, Git, Node.js...)

💡 خامساً: حفظ الملف كنسخة احتياطية أو لتعديل الإعدادات

الخطوات اللي عملها:

1. نسخ كل محتوى الـ YAML
2. فتح الـ Jenkins controller terminal
3. راح على مجلد ← cd /var/lib/jenkins/JENKINS_BACKUP/
4. أنشأ ملف اسمه : jenkins-casc.yaml ← vi jenkins-casc.yaml
5. لصق كل المحتوى فيه.

🧰 سادساً: عمل تعديل بسيط كمثال عملي

- عدّل system message اللي بتظهر على الـ Dashboard
  - كانت مكتوبة : "Welcome to Dashboard Controller"
  - غيّرها إلى: "Loading data from Jenkins Configuration as Code just for demo purpose"
    وسجّل الملف بعد التعديل.

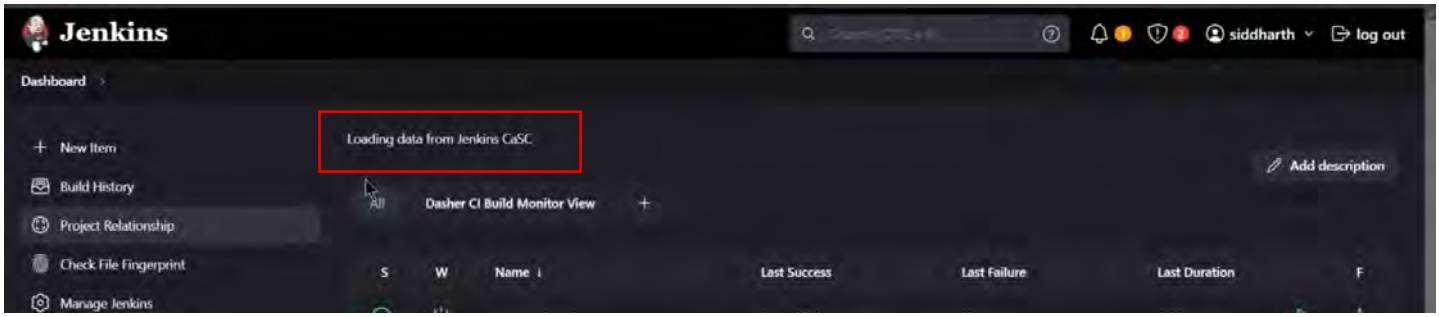
خلي باالك انها كدا لسا معرفتلهوش مسار الملف اللي احنا لسا عاملينه .. (تابع الخطوة الجاية)

🔗 سابعاً: تطبيق الملف الجديد داخل Jenkins

- ❖ رجع تاني إلى Manage Jenkins → Configuration as Code
وكتب مسار الملف : /var/lib/jenkins/JENKINS_BACKUP/jenkins-casc.yaml
- ❖ ثم ضغط: Apply new configuration
- ❖ لو الملف مكتوب صح (valid YAML syntax) ← Jenkins هيقولك :

✅ "Configuration loaded successfully from YAML file."

- ❖ بعد كده عمل Refresh للـ Dashboard
وظهر فعلاً الـ message الجديدة اللي كتبها في الـ YAML



🧠 ثامناً: خيارات إضافية بعد التحميل

- Download configuration ← تنزّل نسخة من الـ YAML الحالي.
- Reload existing configuration ← تعمل Reload من الملف الحالي لو عدّلته.
- Documentation & JSON Schema ← تقدر ترجع ليهم عشان تشوف كل المفاتيح اللي ممكن تستخدمها جوه YAML وتكتبها صح.

✅ الخلاصة

Jenkins Configuration as Code Plugin (JCasC) هو طريقة قوية جدًا تخليك تتحكم في كل إعدادات Jenkins من خلال ملف YAML واحد بدل من UI

🎯 الفوايد:

- سهل، منظم، readable
- تقدر تحطه في Git عشان تعمل Version Control
- تقدر تعمل نفس الـ Jenkins setup لأي environment بسهولة.
- تقلل الأخطاء البشرية وتخلي الـ setup reproducible في ثواني.

Real Example on JCasC (creating a job) & how to write It's YAML

🎯 الهدف من الشرح

في الشرح ده هنتعلم:

Example 1

1. نعرف إزاي نضيف Node.js Tool عن طريق YAML
2. نعرف إزاي نضيف Pipeline Job تلقائيًا.
3. نشرح سبب الحاجة لـ JobDSL Plugin

Example 2

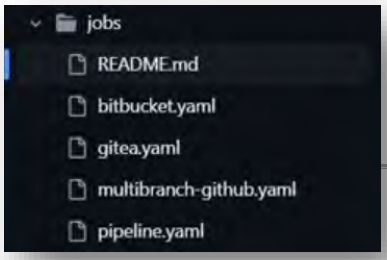
4. إزاي نكتب ملف YAML بسيط نستخدمه مع Configuration as Code Plugin
5. إزاي نحطه في المسار الصح
6. إزاي نخلي Jenkins يقرأ الملف ده ويطبّق الإعدادات الجديدة

---

configuration-as-code-plugin/demos at master

المثال الأول : 🧱 الخطوة 1: إنشاء ملف YAML جديد

- نفتح السيرفر اللي عليه Jenkins ونروح لمسار الجينكنز الرئيسي : cd /var/lib/jenkins
- وبعدين نفتح أي محرر زي nano أو vi وننشئ ملف اسمه: nano jenkins-casc.yaml
- وبعدين ننسخ فيه الاعدادات (الـ jobs مثلا) اللي جيبناها من الريبو
  - لااحظ ان ممكن وانت بتنسخ الاعدادات (مثلا الـ jobs) (من فولدر الـ demos اللي في الريبو) .. انها ذات نفسها تكون محتاجة plugins معينة تتسطب علشان تشتغل .. وبالتالي متنساش تبص عالـ README.md اللي موجود في السكشن بتاع الاعداد (الـ jobs) اللي انت عاوزه.
- ومتنسااش تعرفله مسار الملف var/lib/jenkins/jenkins-casc.yaml/
  في Manage Jenkins → Configuration as Code

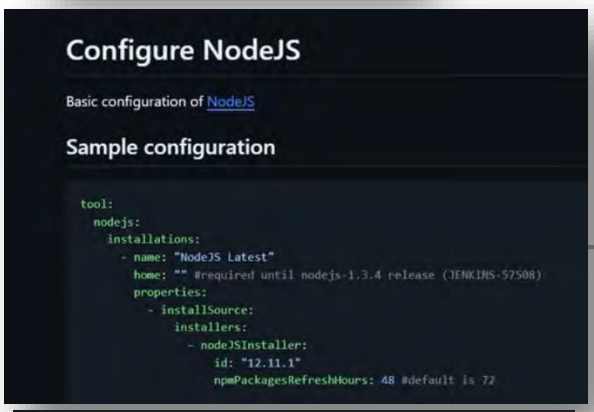


🧱 الخطوة 2: تجهيز YAML

هنعرف الـ tools و jobs في الملف اللي عملناه وننسخ :

✅ شرح سريع:

- tool: nodejs: ← بيحدد أدوات Node.js اللي هنستخدمها.
- jobs: ← بنضيف Pipeline Job مباشر من YAML
- script: ← هنا بنكتب الـ Jenkins Pipeline Script



🧩 الخطوة 2: تثبيت JobDSL Plugin

- السبب: لو عايزين نعرف jobs من YAML ، لازم يكون عندنا JobDSL Plugin عشان JCasC يعرف يقرأ الـ jobs
- طريقة التثبيت:

1. Jenkins → Dashboard → Manage Jenkins → Plugins → Available
2. ابحث عن JobDSL
3. Install → Restart (لو مطلوب)

```
jobs:
  - script: >
      folder('testjobs')
  - script: >
      pipelineJob('testjobs/default-agent') {
        definition {
          cps {
            script("""\
              pipeline {
                agent any
                stages {
                  stage ('test') {
                    steps {
                      echo "hello"
                    }
                  }
                }
              }"""stripIndent())
          }
        }
      }
```

🧠 الخطوة 3: تطبيق الإعدادات

1. ادخل Jenkins → Manage Jenkins → Configuration as Code
2. في خانة Path/URL ، حط : /var/lib/jenkins/jenkins-casc.yaml
3. اضغط Apply new configuration
4. لو YAML صح، Jenkins هيطبق كل الإعدادات:
   - Node.js tool هيظهر في Tools
   - Job جديدة هتظهر على Dashboard تحت folder test jobs

🔍 الخطوة 4: التأكد من النتيجة

- Dashboard ← Tools ← Node.js : هتشوف node12 ونسخة تانية لو موجودة
- Dashboard ← Jobs : هتشوف Pipeline Job باسم Test اسمه Stage مع default agent يطبع Hello from Pipeline Job!

💡 الملاحظات المهمة

1. Jobs لا يمكن أن تكون root element بدون JobDSL Plugin
2. YAML يمكن يحتوي على:
   - jenkins: إعدادات عامة
   - tool: أدوات
   - jobs: Pipeline/Multibranch jobs
3. التعديل على YAML ← reload ← مباشرة يتم تطبيق الإعدادات بدون restart

---

✅ الخلاصة

دلوقتي تقدر:

- تضيف أدوات Jenkins (Node.js, Maven, ...) من YAML
- تنشئ Pipeline Jobs تلقائيًا بدون UI
- تستخدم JCasC + JobDSL لعمل Infrastructure + Jenkins Jobs as Code

---

✍️ المثال الثاني :

هنكتب إعداد بسيط جدًا فيه حاجتين:

- systemMessage : رسالة بتظهر على الـ Dashboard
- numExecutors : عدد الـ executors على الـ controller

وده مثال عملي 👇

```yaml
jenkins:
  systemMessage: "🚀 This Jenkins is configured using Configuration as Code (JCasC)!"
  numExecutors: 2
  mode: NORMAL
  securityRealm:
    local:
      allowsSignup: false
      users:
        - id: "admin"
          password: "admin123"
  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false
```

🧠 شرح الملف سطر بسطر:

السطر	المعنى
`:jenkins`	بداية تعريف إعدادات Jenkins
`:systemMessage`	رسالة بتظهر في الـ Dashboard (زي الـ banner message)
`:numExecutors`	عدد الـ builds اللي ممكن Jenkins يشغلها في نفس الوقت على الـ master node
`mode: NORMAL`	معناها Jenkins يقدر يشغل jobs على الـ master نفسه
`:securityRealm`	بيحدد نوع تسجيل الدخول (هنا local users)
`:users`	بننشئ مستخدم admin محلي
`:authorizationStrategy`	بنحدد صلاحيات المستخدمين (هنا فقط الـ logged-in users يقدروا يشوفوا ويشتغلوا)

📃 النتيجة بعد التطبيق:

- ✅ Jenkins هيقرا الملف ويقولك:

"Configuration loaded successfully from /var/lib/jenkins/jenkins-casc.yaml"

- ولو رجعت للـ Dashboard هتشوف الرسالة دي فوق:

🚀 This Jenkins is configured using Configuration as Code (JCasC)!

- ولو دخلت على Manage Jenkins → Nodes هتلاقي إن عدد الـ executors بقى 2.
- وكمان لو دخلت على Users هتلاقي المستخدم "admin" اتضاف فعلاً.

💡 الخطوة 5: لو حبيت تعيد تحميل الإعدادات بعد التعديل

- لو عدّلت الملف بعدين، مش لازم تعمل restart .. بس روح على نفس الصفحة واضغط: "Reload existing configuration"
- Jenkins هيقرأ الملف من جديد ويطبّق التغييرات فورًا.

🧠 الخلاصة

- JCasC = Infrastructure as Code + Jenkins Configs
- 📦 بدل ما تضبط Jenkins بـ GUI كل مرة، تقدر تكتب الإعدادات كلها في ملف YAML واحد، تحفظه في Git ، وتعمل setup لأي بيئة جديدة بثانية واحدة بس.
- الفوايد الحقيقية:
  - Reproducible setups (تقدر تبني Jenkins تاني بنفس الإعدادات بالظبط)
  - سهل في الـ Automation وCI/CD
  - مناسب جدًا للـ DevOps Teams اللي بيحبوا كل حاجة كـ "Code"

Pipeline Durability

مقدمة – فكرة Pipeline Durability

- ❑ في Jenkins ، الفرق الأساسي بين Pipeline Job و Freestyle Job هو إن Pipeline قابل للاستئناف بعد أي restart غير متوقع للـ Jenkins
  - ده بيحصل لأن Pipeline بيكتب بيانات مؤقتة على الديسك باستمرار، عشان لو حصل crash أو restart ، يقدر يكمل من حيث توقف.
- ❑ لكن المشكلة: لو عندك Pipeline فيه مئات الخطوات، ده ممكن يسبب بطء في الأداء بسبب كثرة عمليات الـ I/O على الديسك.
- ❑ عشان كده عندنا خيارات التحمل (Durability Settings) لتوازن الأداء والاستمرارية.

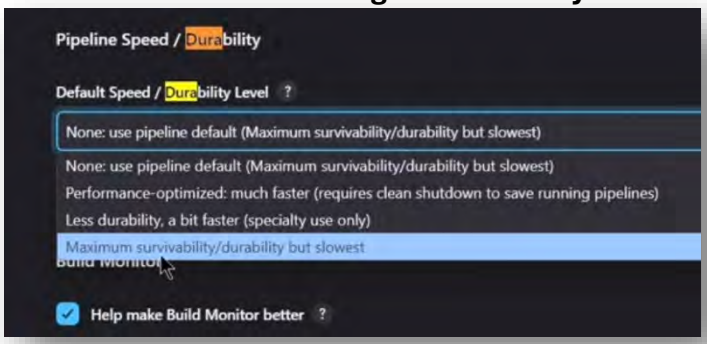
أنواع Durability Settings

Scaling Pipelines

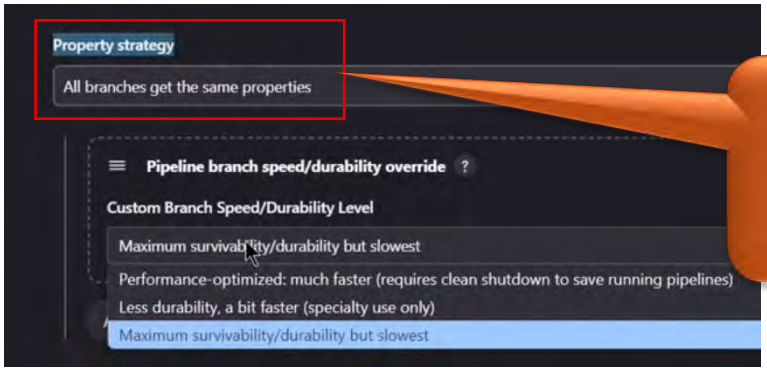
1. Performance Optimized
   - يقلل جدًا من عمليات الكتابة على الديسك.
   - لو Jenkins اتقفل بشكل غير نظيف، ممكن تفقد البيانات حيث يعامل الـ Job كـ Freestyle
   - ملاحظات عملية : ممتاز للـ pipelines غير الحرجة أو اللي performance فيها مهم جدًا.
2. Max Survivability
   - أبطأ لأنه بيكتب كل حاجة على الديسك.
   - تقدر تستأنف Pipeline بعد أي restart أو crash
   - ملاحظات عملية : استخدمه للـ production deployment أو أي pipeline بيعدل على critical infrastructure
3. Survivable Non-Atomic
   - خيار وسط، قليل الاستخدام لكن موجود كخيار إضافي.

أماكن تعديل الإعدادات

1. على مستوى Jenkins كله (Global)
   - اذهب إلى Manage Jenkins → System → Pipeline Speed / Durability Settings
   - Default: Max Survivability
   - يمكن تغييره لأي خيار حسب احتياجاتك .



2. على مستوى Pipeline معين
   - عند إنشاء أو تعديل Pipeline Job ، يوجد خيار Pipeline Speed / Durability



بيتيحلك تطبق الـ Durability علي كل الـ branches
اللي عندك أو لو عاوز تضيف Exception
وتخصص الـ Durability لكل branch

وبكدا انت شغال علي مستوي الـ Multi-Branch

3. على مستوى Multi-Branch Project
   - تقدر تخصص durability لكل فرع:
     - مثلاً : الفرع main → Max Survivability
     باقي الفروع ← Performance Optimized

مثال عملي Max Survivability :

1 إعداد Pipeline

- Pipeline فيه stage واحدة تعمل:
- الإعداد : Max Survivability

```
Script ?
pipeline {
    agent any
    stages {
        stage('For Loop - Durability Test') {
            steps {
                script {
                    sh 'touch numbers.txt'
                    for (int i = 0; i < 600; i++) {
                        sh "echo ${i + 1} >> numbers.txt"
                        sleep 1
                    }
                }
            }
        }
    }
}
```

2 تجربة Restart

- نبحث عن PID لـ Jenkins :
  - ps aux | grep -i jenkins.war
  - kill -9 <PID>

```
root@jenkins-controller-1 in ~ on ☁ (us-east-2)
❯ ps aux | grep -i jenkins.war
jenkins  1048864  9.7 25.2 3768468 1012872 ?    Ssl  16:47   3:22 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --webroot=/var/cache/
ins/war --httpPort=8080
root     1051824  0.0  0.0   7084  2176 pts/5   S+   17:22   0:00 grep --color=auto -i jenkins.war

root@jenkins-controller-1 in ~ on ☁ (us-east-2)
❯ kill -9 1048864
```

- بعد Restart من الـ GUI عادي، نفتح الـ Pipeline Console :
  - Jenkins يستأنف التشغيل من الرقم اللي توقف عنده، بدون فقد بيانات.

```
[Pipeline] sleep
Sleeping for 1 sec
[Pipeline] sh
+ echo 39
[Pipeline] sleep
Sleeping for 1 sec
Resuming build at Sun Nov 10 17:22:47 UTC 2024 after Jenkins restart
No need to sleep any longer
Ready to run at Sun Nov 10 17:22:48 UTC 2024
[Pipeline] sh
+ echo 40
[Pipeline] sleep
Sleeping for 1 sec
```

ملاحظات عملية : هذا مثال حي على الاستمرارية العالية عند Pipelines المهمة.

مثال عملي Performance Optimized :

- نغير إعداد Pipeline إلى Performance Optimized، ونكرر التجربة بنفس الكود :
- Kill Jenkins أثناء التنفيذ.
- بعد Restart :
  - Pipeline يفشل ويعتبر كأنه Freestyle Job
  - السبب : Performance Optimized يقلل كتابة البيانات على الديسك، لذا لا يمكن الاستئناف بعد crash

```
[Pipeline] sleep
Sleeping for 1 sec
[Pipeline] sh
Creating placeholder flownodes because failed loading originals.
ERROR: Cannot resume build because FlowNode 42 for FlowHead 1 could not be loaded. This is expected to happen when using the PERFORMANCE_OPTIMIZED
durability setting and Jenkins is not shut down cleanly. Consider investigating to understand if Jenkins was not shut down cleanly or switching to
the MAX_SURVIVABILITY durability setting which should prevent this issue in most cases.
[Gitea] do not publish assets due to build being non-Successfully
Finished: FAILURE
```

Best Practices & Tips

1. استخدم Max Survivability للـ Production أو أي عمليات حرجة.
2. استخدم Performance Optimized للـ Dev/Testing pipelines لتسريع الأداء.
3. في Multi-Branch Projects :
   - Main branch → Max Survivability
   - Feature branches → Performance Optimized
4. دايمًا اختبر الـ Pipeline مع Durability مختلفة قبل الإطلاق الفعلي.

◆ ملخص شامل

Setting	Resume after Crash	Speed / Performance	Use Case
Max Survivability	Yes ✅	Slow	Critical pipelines, Production
Performance Optimized	No ❌	Fast	Dev/Test pipelines
Survivable Non-Atomic	Partial	Medium	Rare use cases

❑ Pipeline Durability في Jenkins هي وسيلة للتحكم بين الأداء والقدرة على الاستئناف بعد restart أوcrash ، ويمكن ضبطها على مستوى Global / Pipeline / Branch حسب أهمية المشروع أو الفرع.