

Curso de desarrollo de software

Nota: Lo realizado se encuentra en otro color más notorio que el de la actividad, para resaltar la evidencia.

Actividad individual: 100 minutos

Presentar en clase el proceso entregado

Pipeline de Integración Continua

Ya sabemos cómo configurar Jenkins. En esta actividad, veremos cómo usarlo de manera efectiva, centrándonos en la función que se encuentra en el corazón de Jenkins: los pipelines. Al construir un proceso completo de integración continua desde cero, describiremos todos los aspectos del desarrollo de código moderno orientado al equipo.

Requerimientos técnicos

Para completar esta actividad, necesitarás el siguiente software:

- Jenkins
- Java JDK 8

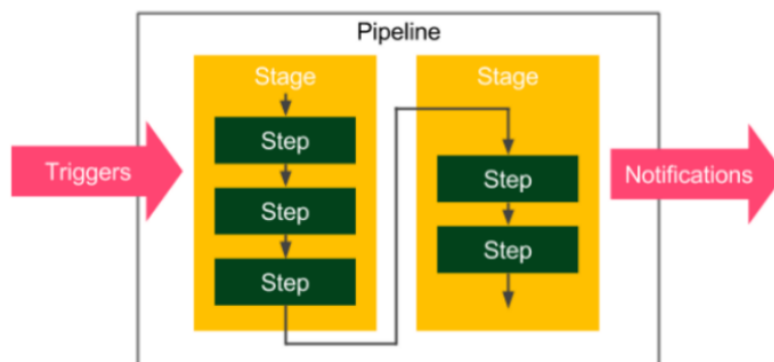
Introducción a los pipelines

Un pipeline es una secuencia de operaciones automatizadas que normalmente representa una parte del proceso de entrega y control de calidad del software. Pueden verse como una cadena de scripts que brindan los siguientes beneficios adicionales:

- Agrupación de operaciones: las operaciones se agrupan en etapas (también conocidas como puertas o puertas de calidad) que introducen una estructura en un proceso y definen claramente una regla: si una etapa falla, no se ejecutan más etapas.
- Visibilidad: se visualizan todos los aspectos de un proceso, lo que ayuda en el análisis rápido de fallas y promueve la colaboración en equipo.
- Feedback: los miembros del equipo aprenden acerca de los problemas tan pronto como ocurren para que puedan reaccionar rápidamente.

Primero describamos la estructura de un pipeline de Jenkins y luego cómo funciona. **La estructura de un pipeline**

Un pipeline de Jenkins consta de dos tipos de elementos: una etapa (stage) y un paso (step). El siguiente diagrama muestra cómo se utilizan:



Los siguientes son los elementos básicos de la pipeline:

- Step: una sola operación que le dice a Jenkins qué hacer; por ejemplo, verifica el código del repositorio y ejecuta un script.
- Stage: una separación lógica de pasos que agrupa secuencias de pasos conceptualmente distintas, por

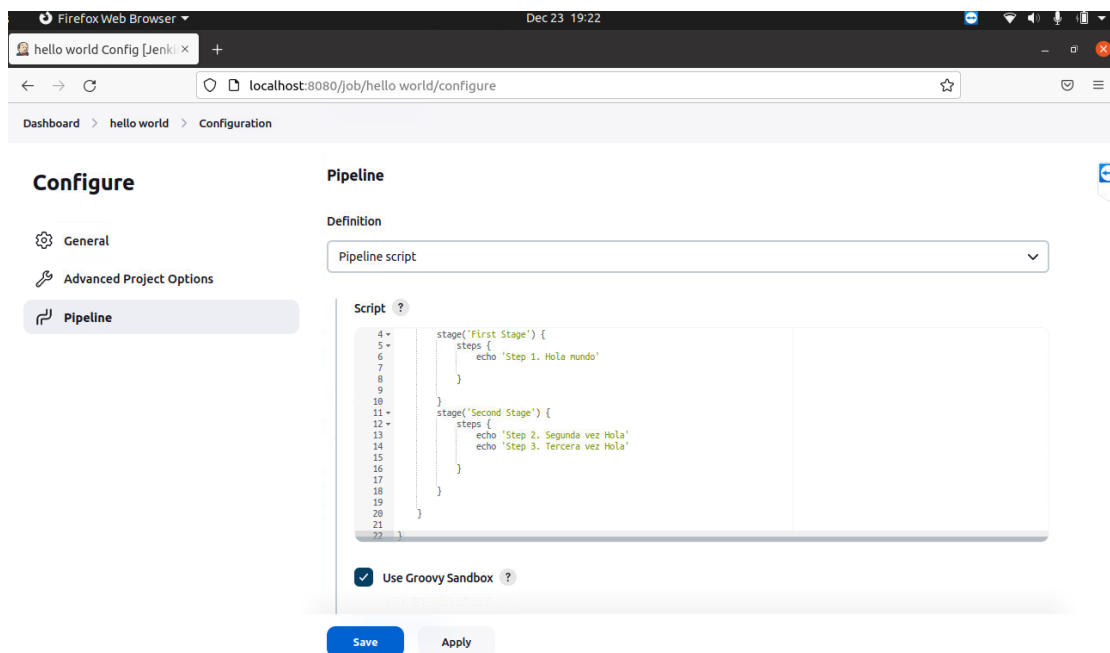
ejemplo, build, test y deploy, que se usa para visualizar el progreso del pipeline de Jenkins.

Un Hello World de varias etapas

Como ejemplo, ampliemos la pipeline Hello World para que contenga dos etapas:

```
pipeline {
  agent any
  stages {
    stage('First Stage') {
      steps {
        echo 'Step 1. Hola mundo'
      }
    }
    stage('Second Stage') {
      steps {
        echo 'Step 2. Segunda vez Hola'
        echo 'Step 3. Tercera vez Hola'
      }
    }
  }
}
```

Aquí se puede presenciar la modificación del pipeline según lo presentado en la sección anterior:

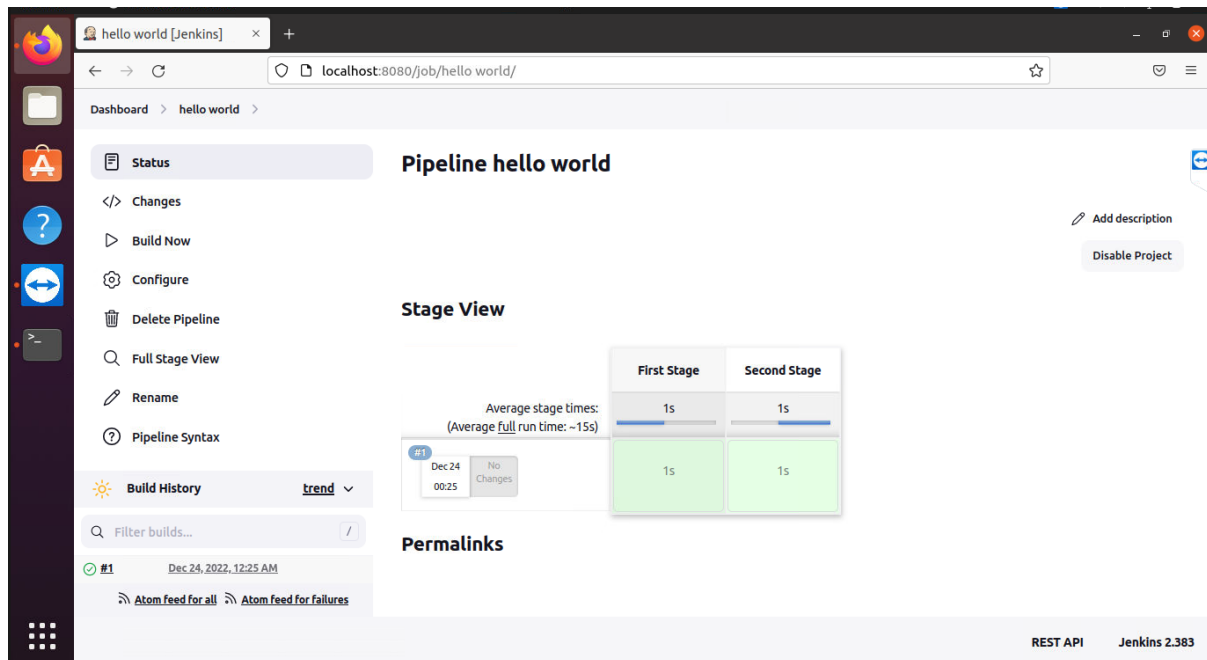


El pipeline no tiene requisitos especiales en términos de entorno y ejecuta tres pasos dentro de dos etapas. Cuando hacemos clic en Build Now, deberíamos ver una representación visual.



Comprueba estos resultados.

Los resultados se muestran de forma análoga, como se puede ver a continuación, Gracias a esto podemos decir que el pipeline ha resultado ejecutarse con éxito.



El pipeline tuvo éxito y podemos ver los detalles de ejecución haciendo clic en la consola. Si alguno de los pasos fallaba, el procesamiento se detendría y no se ejecutaban más pasos.

La sintaxis del pipeline

Usamos la sintaxis declarativa que se recomienda para todos los proyectos nuevos. Las otras opciones son un DSL basado en Groovy y (antes de Jenkins 2) XML (creado a través de la interfaz web).

La sintaxis declarativa se diseñó para simplificar al máximo la comprensión del pipeline, incluso para personas que no escriben código a diario. Es por eso que la sintaxis se limita solo a las palabras clave más importantes.

Probemos un experimento (escribe en un editor de texto el pipeline), pero antes de describir todos los detalles, lee la siguiente definición de pipeline e intenta adivinar qué hace:

```
pipeline {
  agent any
  triggers { cron('* * * * *') }
  options { timeout(time: 5) }
  parameters {
    booleanParam(name: 'DEBUG_BUILD', defaultValue: true,
      description: 'construimos la depuracion?')
  }
  stages {
    stage('Example') {
      environment { NAME = 'Checha' }
      when { expression { return params.DEBUG_BUILD } }
      steps {
        echo "Hola $NAME"
        script {
```

```

        def browsers = ['chrome', 'firefox']
        for (int i = 0; i < browsers.size(); ++i) {
            echo "Prueba el ${browsers[i]}
            browser."
        }
    }
}
}
}
}
post { Siempre { echo 'Hola muchas gracias' } }
}

```

Tengo entendido que gracias a la opción `timeout`, su duración ocupa un tiempo máximo de ejecución de 5 minutos, luego hay una sección de parámetros en la que define uno del tipo booleano que tiene un valor predeterminado `true` y es nombrado como `'DEBUG_BUILD'`, ahora cuando pase al Stage designado como `'Example'` se usará este valor para determinar cuándo ejecutar que será cuando este sea verdadero, en ese Stage también se define una variable de entorno llamada `NAME` que es designada con `'Checha'`, luego imprimirá un mensaje saludando a nuestra variable de entorno y luego un mensaje de Prueba el ____ browser, donde el espacio será ocupado por los elementos de un conjunto definido como `browsers`: `'chrome'` y `'firefox'`, y ya por lo último, siempre mostrará un mensaje de Hola muchas gracias.

El pipeline es bastante complejo. En realidad, es tan complejo que contiene la mayoría de las instrucciones de Jenkins disponibles. Para responder al acertijo del experimento, veamos qué hace el pipeline, instrucción por instrucción:

1. Utiliza cualquier agente disponible
2. Se ejecuta automáticamente cada minuto
3. Se detiene si la ejecución tarda más de 5 minutos
4. Solicita el parámetro de entrada booleano antes de comenzar
5. Establece `Checha` como la variable de entorno `NAME`
6. Hace lo siguiente, solo en el caso del parámetro de entrada `True`:
 - Imprime Hola Checha
 - Imprime Prueba el browser firefox
 - Imprime Prueba el browser chrome
7. Imprime ¡Siempre Hola muchas gracias, independientemente de que haya algún error durante la ejecución.

Un pipeline declarativo siempre se especifica dentro del bloque del pipeline y contiene secciones, directivas y pasos. Pasaremos por cada uno de ellos.

Secciones

Las secciones definen la estructura del pipeline y normalmente contienen una o más directivas o pasos. Se definen con las siguientes palabras clave:

- **Stages**: Define una serie de una o más directivas de etapa.
- **Steps**: Esto define una serie de instrucciones de uno o más pasos.
- **Post**: Esto define una serie de instrucciones de uno o más pasos que se ejecutan al final de la construcción del pipeline; están marcados con una condición (por ejemplo, `always`, `success` o `failure`) y generalmente se usan para enviar notificaciones después de la construcción del pipeline
- **Agent**: Esto especifica dónde tiene lugar la ejecución y puede definir `label` para que coincida con los agentes igualmente etiquetados, o `docker` para especificar un contenedor que se aprovisiona dinámicamente para proporcionar un entorno para la ejecución del pipeline.

Directivas

Las directivas expresan la configuración de un pipeline o de sus partes:

- **Triggers**: Esto define formas automatizadas de activar el pipeline y puedes usar `cron` para establecer la

planificación basada en el tiempo, o pollSCM para comprobar si hay cambios en el repositorio .

- Options: Esto especifica las opciones específicas del pipeline, por ejemplo, timeout (el tiempo máximo de ejecución de un pipeline) o retry (la cantidad de veces que se debe volver a ejecutar el pipeline después de una falla).
- Environment: Esto define un conjunto de valores clave utilizados como variables de entorno durante la construcción.
- Parameters: Esto define una lista de parámetros de entrada del usuario.
- Stage: Esto permite la agrupación lógica de pasos.
- When: Determina si se debe ejecutar la etapa, dependiendo de una condición dada.
- Tools: Esto define las herramientas a instalar y usar el PATH.
- Input: Nos permite solicitar los parámetros de entrada.
- Parallel: Esto nos permite especificar etapas que se ejecutan en paralelo.
- Matrix: Esto nos permite especificar combinaciones de parámetros para los cuales las etapas dadas corren en paralelo.

Steps

Los pasos son la parte más fundamental del pipeline. Definen las operaciones que se ejecutan, por lo que en realidad le dicen a Jenkins qué hacer:

- sh: Esto ejecuta el comando de shell, en realidad, es posible definir casi cualquier operación usando sh.
- custom: Jenkins ofrece muchas operaciones que se pueden usar como pasos (por ejemplo, echo), muchos de ellos son simplemente envoltorios sobre el comando sh que se usa por conveniencia. Los complementos también pueden definir sus propias operaciones.
- scripts: esto ejecuta un bloque de código basado en Groovy que se puede usar para algunos escenarios no triviales donde se necesita control de flujo.

Ten en cuenta que la sintaxis de un pipeline es muy genérica y, técnicamente, pueden usarse para casi cualquier proceso de automatización. Es por esto que el pipeline debe ser tratado como un método de estructuración y visualización. Sin embargo, el caso de uso más común es implementar el servidor de integración continua, que veremos en la siguiente sección.

Commit pipeline

El proceso de integración continua más básico se denomina commit pipeline. Esta fase clásica, como su nombre lo indica, comienza con el commit (o push en Git) en el repositorio principal y da como resultado un informe sobre el éxito o el fracaso de la construcción. Dado que se ejecuta después de cada cambio en el código, la construcción no debería demorar más de 5 minutos y debería consumir una cantidad razonable de recursos.

La fase de commit es siempre el punto de partida del proceso de entrega continua y proporciona el ciclo de retroalimentación más importante en el proceso de desarrollo.

La fase de commit funciona de la siguiente manera: un desarrollador registra el código en el repositorio, el servidor de integración continua detecta el cambio y comienza la construcción. El pipeline commit más fundamental contiene tres etapas:

- Checkout: Esta etapa descarga el código fuente del repositorio.
- Compile: esta etapa compila el código fuente.
- Unit test: esta etapa ejecuta un conjunto de pruebas unitarias.

Vamos a crear un proyecto de muestra y ver cómo implementar el commit pipeline.

Información Este es un ejemplo de pipeline para un proyecto que usa tecnologías como Git, Java, Gradle y Spring Boot. Sin embargo, los mismos principios se aplican a cualquier otra tecnología.

Checkout

Checkout el código del repositorio siempre es la primera operación en cualquier pipeline. Para ver esto, necesitamos tener un repositorio. Entonces, podemos crear una pipeline.

Creación de un repositorio de GitHub

La creación de un repositorio en el servidor de GitHub requiere solo unos pocos pasos:

1. Vaya a <https://github.com/>.
 2. Crea una cuenta si aún no tienes una.
 3. Haga clic en New, junto a Repositories.
 4. Dale un nombre: calculador o el nombre de la actividad.
 5. Marca Initialize this repository with a README.
 6. Haga clic Create repository.
- Ahora, debería ver la dirección del repositorio.

La creación es evidenciada al acceder al siguiente enlace <https://github.com/DagmarLV/calculadorActividad21>

Creación de una etapa checkout

Podemos crear un nuevo pipeline llamado calculador, y como es un script de pipeline, colocar el código con una etapa llamada Checkout:

```
pipeline {
  agent any
  stages {
    stage("Checkout"){
      steps {
        git url: 'https://github.com/<tu cuenta>/calculador.git', branch: 'main'
      }
    }
  }
}
```

Aquí se muestra el pipeline en nuestro Jenkins:

Script ?



```
1 pipeline {
2   agent any
3   stages {
4     stage("Checkout") {
5       steps {
6         git url: 'https://github.com/DagmarLV/calculadorActividad21'
7       }
8     }
9   }
10 }
11
12
13
14
```

El pipeline se puede ejecutar en cualquiera de los agentes, y su único paso no es más que descargar código del repositorio. Podemos hacer clic en Build Now para ver si se ejecutó con éxito.

Información: El kit de herramientas de Git debe instalarse en el nodo donde se ejecuta la construcción. Cuando tenemos el checkout, estamos listos para la segunda etapa.

Tenemos que tener git en el nodo donde se ejecuta la construcción, para tener git podemos instalarlo directamente en el container mediante el uso de un comando de instalación de paquetes o simplemente podríamos usar un plugin de Jenkins para instalar Git en el container de Jenkins y configurar su uso en nuestros trabajos con Jenkins.

Optamos por esta última opción y al revisar los plugins podemos darnos cuenta de que el GitPlugin ya se encuentra habilitado como se muestra a continuación:

Plugins

Name ↓	Enabled
Git client plugin 3.13.1 Utility plugin for Git support in Jenkins Report an issue with this plugin	<input checked="" type="checkbox"/>
Git plugin 4.14.3 This plugin integrates Git with Jenkins. Report an issue with this plugin	<input checked="" type="checkbox"/>
GitHub API Plugin 1.303-400.v35c2d8258028 This plugin provides GitHub API for other plugins. Report an issue with this plugin	<input checked="" type="checkbox"/>
GitHub Branch Source Plugin 1696.v3a_7603564d04 Multibranch projects and organization folders from GitHub. Maintained by CloudBees, Inc.	<input checked="" type="checkbox"/>

Ahora si podemos construir el pipeline y verificar que se puede ejecutar en cualquiera de los agentes y como decía en la parte superior, descarga el código de nuestro repositorio exitosamente, por lo que se muestra su ejecución con éxito.

The screenshot shows the Jenkins web interface for a pipeline named 'calculador'. The left sidebar contains navigation links: Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, Rename, and Pipeline Syntax. The main content area is titled 'Pipeline calculador' and shows a 'Stage View' with a 'Checkout' stage taking 5s. Below the stage view, there's a 'Permalinks' section and a 'Build History' table showing a successful build #1 on Dec 24, 2022, at 5:19 PM. The top navigation bar shows the user is logged in as 'admin'.

Compile

Para compilar un proyecto, necesitamos hacer lo siguiente:

1. Crea un proyecto con el código fuente.
2. Push al repositorio.
3. Agrega la etapa compile al pipeline.

Veamos estos pasos en detalle.

Creación de un proyecto Java Spring Boot

Vamos a crear un proyecto Java muy simple utilizando el marco Spring Boot creado por Gradle.

Información: Spring Boot es un framework de Java que simplifica la creación de aplicaciones empresariales.

Gradle es un sistema de automatización de construcción que se basa en los conceptos de Apache Maven. La forma más sencilla de crear un proyecto Spring Boot es realizar los siguientes pasos:

1. Vaya a <http://start.spring.io/>.
2. Selecciona Gradle Project en lugar del Proyecto Maven (puedes elegir Maven si lo prefieres a Gradle).
3. Completa Group y Artifact (por ejemplo, com.checha y calculador).
4. Agregar Web a Dependencies.
5. Haga clic en Generate.
6. El proyecto de esqueleto generado debe descargarse (el archivo calculador.zip).

Mostramos la pantalla <http://start.spring.io/> y los datos dados.

Aquí se evidencia la pantalla justo antes de darle click a generate con todos los datos brindados

Nota: En la sección de Spring Boot, opté por la versión 2.7.7 ya que al seleccionarla no me aparece ningún error relacionado a ello en los pasos posteriores como me sucedió con la versión por defecto.

Después de crear el proyecto, podemos hacer push en el repositorio de GitHub. **Pushing el código a GitHub**
Usaremos la herramienta Git para realizar las operaciones de commit y push.

Primero clonamos el repositorio en el sistema de archivos:

```
$ git clone https://github.com/l<tu nombre>/calculador.git
```

Extraiga el proyecto descargado de <http://start.spring.io/> en el directorio creado por Git.

Sugerencia: Si prefieres, puedes importar el proyecto a IntelliJ, Visual Studio Code, Eclipse o tu herramienta IDE favorita.

Como resultado, el directorio calculador debería tener los siguientes archivos:

```
$ ls-la
```

```
... build.gradle .git .gitignore gradle gradlew gradlew.bat
```

```
HELP.md README.md settings.gradle src
```

Información: Para realizar las operaciones de Gradle localmente, debes tener instalado Java JDK.

Podemos compilar el proyecto localmente usando el siguiente código:

```
$ ./gradlew compileJava
```

En el caso de Maven, puedes ejecutar `./mvnw compile`. Tanto Gradle como Maven compilan las clases de Java ubicadas en el directorio `src`.

Ahora, podemos hacer commit y push al repositorio de GitHub:

```
$ git add .
```

```
$ git commit -m "Agrega plantilla Spring Boot"
```



```
$ git push -u origin main
```

El código ya está en el repositorio de GitHub. Si quieres comprobarlo, puedes ir a la página de GitHub y ver los archivos.

El archivos en el repositorio se muestran de la siguiente manera:



Creación de una etapa Compile

Podemos agregar una etapa Compile al pipeline usando el siguiente código:

```
stage("Compile") {  
    steps {  
        sh "./gradlew compileJava"  
    }  
}
```

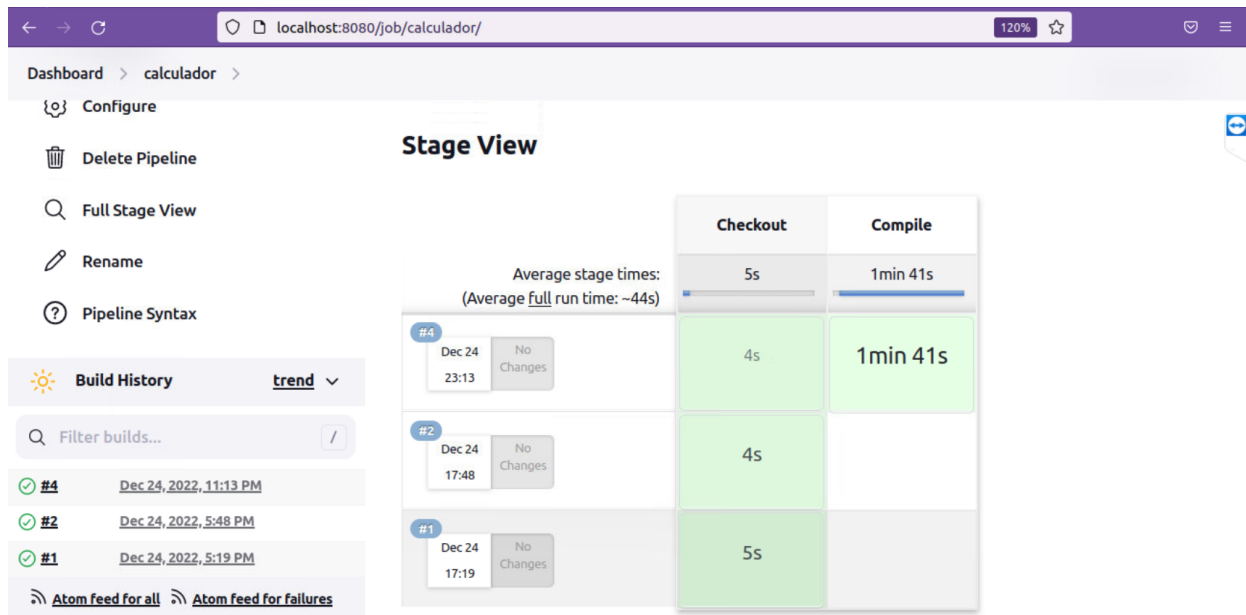
A continuación se muestra el pipeline agregando el Stage "Compile", que como su nombre indica, su sección de steps permiten la compilación mediante gradle en el lenguaje de programación Java.

Script ?

```
1 pipeline {  
2     agent any  
3     stages {  
4         stage("Checkout") {  
5             steps {  
6                 git url: 'https://github.com/DagmarLV/calculadorActividad21'  
7             }  
8         }  
9     }  
10  
11     stage("Compile") {  
12         steps {  
13             sh "./gradlew compileJava"  
14         }  
15     }  
16 }  
17  
18  
19  
20
```

Ten en cuenta que usamos exactamente el mismo comando localmente y en el pipeline de Jenkins, lo cual es una muy buena señal porque el proceso de desarrollo local es coherente con el entorno de integración continua. Después de ejecutar la construcción, deberías ver dos cuadros verdes. También puedes verificar que el proyecto se construyó correctamente en el registro de la consola.

Al construir el pipeline obtenemos lo siguiente y podemos evidenciar que ha resultado con éxito todos los Stages, así como cuando solo tenía el Stage de Checkout.



Pruebas unitarias

Es hora de agregar la última etapa, que es las pruebas unitarias, comprobar si el código hace lo que esperamos que haga. Tenemos que hacer lo siguiente:

1. Agrega el código fuente para la lógica del calculador.
2. Escribe una prueba unitaria para el código.
3. Agrega una etapa de Jenkins para ejecutar la prueba unitaria.

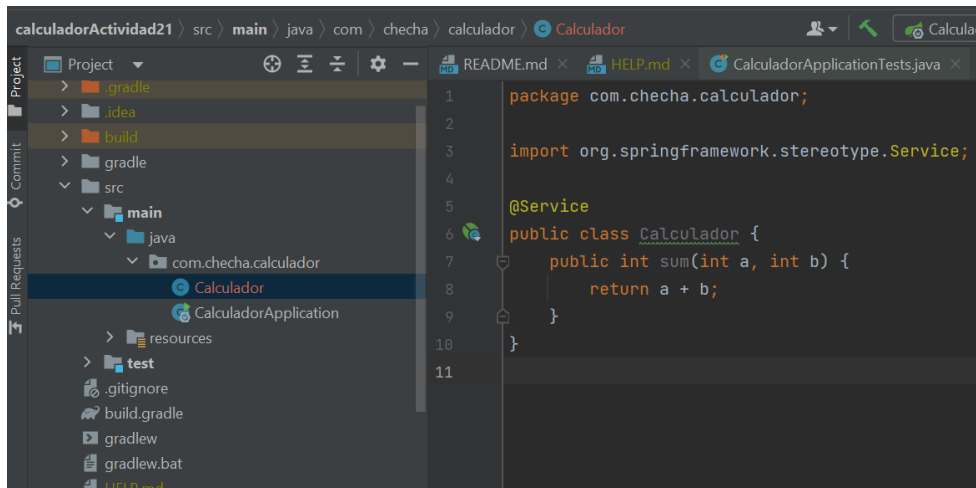
Vamos a elaborar más sobre estos pasos a continuación.

Crear la lógica de negocios

La primera versión del calculador podrá sumar dos números. Agreguemos la lógica empresarial como una clase en el archivo `src/main/java/com/<tu nombre>/calculator/Calculator.java`:

```
package ...
import org.springframework.stereotype.Service;
@Service
public class Calculador {
    public int sum(int a, int b) {
        return a + b;
    }
}
```

A continuación, se muestra el código implementado para la clase `Calculador` utilizando el IDE IntelliJ IDEA, esta clase tiene una función `sum` que recibe dos enteros como parámetros y simplemente retorna a la suma de ambos números enteros:



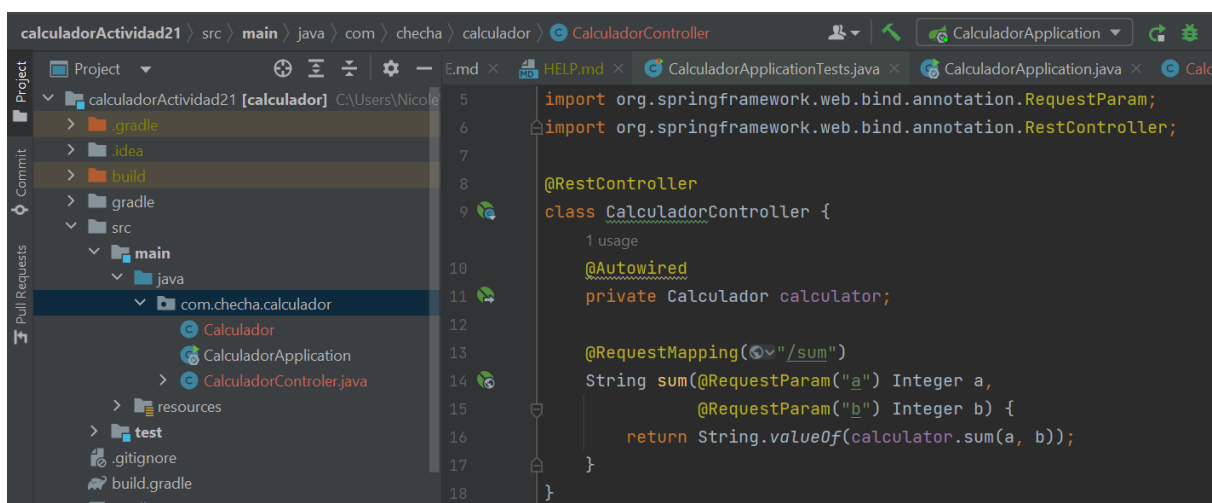
Para ejecutar la lógica empresarial, también debemos agregar el controlador del servicio web en un archivo separado: `src/main/java/com/<tu nombre>/calculador/CalculadorController.java`:

```

package ...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
class CalculadorController {
    @Autowired
    private Calculador calculador;
    @RequestMapping("/sum")
    String sum(@RequestParam("a") Integer a,
    @RequestParam("b") Integer b) {
    return String.valueOf(calculador.sum(a, b));
    }
}

```

Se muestra el código implementado para la clase `CalculadorController`, así exponemos la lógica empresarial como un servicio web:

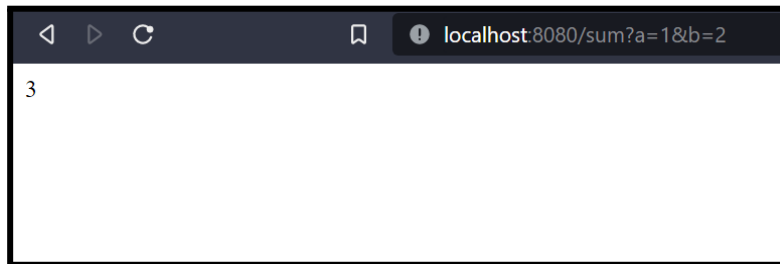


Esta clase expone la lógica empresarial como un servicio web. Podemos ejecutar la aplicación y ver cómo

funciona: `$./gradlew bootRun`

Esto debería iniciar el servicio web y podemos comprobar que funciona navegando hasta el navegador y abriendo `http://localhost:8080/sum?a=1&b=2`. Esto debería sumar dos números (1 y 2) y mostrar 3 en el navegador.

Podemos verificar que funciona, justo como menciona, al acceder al <http://localhost:8080/sum?a=1&b=2>, que gracias a la última clase implementada, toma como parámetros a dos enteros designados como a y b y muestra el valor de la suma de esos dos enteros gracias a que usa la función `sum` de la instancia de la clase `Calculador` con esos parámetros.



Escribiendo una prueba unitaria

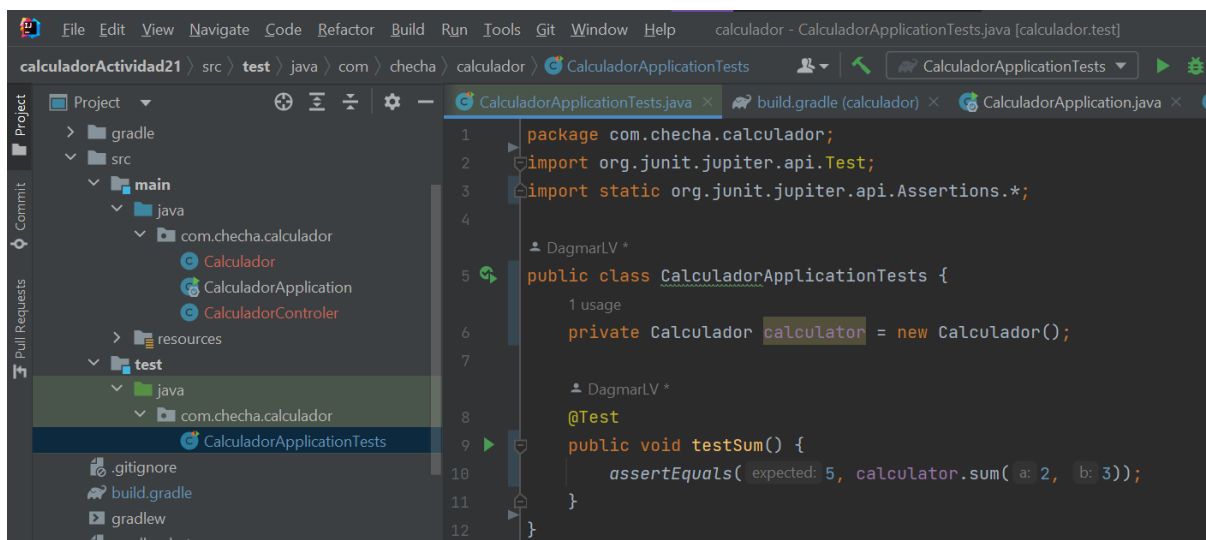
Ya tenemos la aplicación de trabajo. ¿Cómo podemos asegurarnos de que la lógica funcione como se espera? Lo intentamos una vez, pero para saber que funcionará de manera consistente, necesitamos una prueba unitaria. En este caso, será trivial, quizás incluso innecesario, sin embargo, en proyectos reales, las pruebas unitarias pueden salvarte de errores y fallas del sistema.

Vamos a crear una prueba unitaria en el archivo `src/test/java/com/<tu nombre>/calculador/CalculadorTest.java`:

```
package ...
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class CalculadorTest {
    private Calculador calculador = new Calculador();
    @Test
    public void testSum() {
        assertEquals(5, calculador.sum(2, 3));
    }
}
```

Aquí se puede ver la implementación completa del Test que prueba la función `sum` de un objeto de la clase `Calculador` tal como describe el nombre de la prueba asociada `testSum`.

La captura mostrada ha sido captada cuando ya había sido editada la sección de dependencias, por lo que no se presentan errores como se puede evidenciar.



La prueba usa la librería JUnit, por lo que debemos agregarla como una dependencia en el archivo build.gradle:

```
dependencies {  
    ...  
    testImplementation 'junit:junit:...' → completa  
}
```

Aquí se muestra lo que se añadió en la sección de *dependencies* en el archivo build.gradle que se encuentra en el repositorio. En este caso, yo añadí junit 5.9.0, también podríamos usar otras versiones por temas de compatibilidad, pero con esta versión no he presentado ningún inconveniente.

```
15 dependencies {  
16     implementation 'org.springframework.boot:spring-boot-starter-web'  
17     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.9.0'  
18     testImplementation 'org.springframework.boot:spring-boot-starter-test'  
19  
20 }
```

Podemos ejecutar la prueba localmente usando el comando `./gradlew test`. Después, hacemos commit al código y realizamos push al repositorio:

```
$ git add .  
$ git commit -m "Agrega suma logica, controlador y prueba unitaria"  
$ git push
```

Creación de una etapa de prueba unitaria

Ahora, podemos agregar una etapa Unit test al pipeline:

```
stage("Unit test") {  
    steps {  
        sh "./gradlew test"  
    }  
}
```

Ahora podemos ver la parte final del pipeline en el que se ha agregado el Stage "Unit test" que es justamente para compilar los test agregados en nuestro proyecto, en este caso nuestro test: testSum

Pipeline script

Script ?

```
9  
10  
11 stage("Compile") {  
12     steps {  
13         sh "./gradlew compileJava"  
14     }  
15  
16  
17  
18 stage("Unit test") {  
19     steps {  
20         sh "./gradlew test"  
21     }  
22  
23  
24  
25 }  
26  
27
```

Observación: En el caso de Maven, usa el comando `./mvnw test` en su lugar.

Cuando volvamos a construir el pipeline, deberíamos ver tres cuadros, lo que significa que hemos completado el pipeline de integración continua

Verifica este caso en el desarrollo de la actividad.

	Checkout	Compile	Unit test
Average stage times: (Average full run time: ~22s)	579ms	25s	12s
#3 Dec 15 21:48 1 commit	623ms	5s	12s

Ahora que tenemos el pipeline preparado, veamos cómo lograr exactamente el mismo resultado usando Jenkinsfile.

Al construir, podemos presenciar que ha resultado con éxito nuevamente, como se muestra a continuación:

Stage View

	Checkout	Compile	Unit test
Average stage times: (Average full run time: ~46s)	4s	57s	31s
#5 Dec 24 23:26 No Changes	2s	13s	31s
#4 Dec 24 23:13 No Changes	4s	1min 41s	
#2 Dec 24 17:48 No Changes	4s		
#1 Dec 24 17:19 No Changes	5s		

Permalinks

Jenkinsfile

Hasta ahora, hemos creado todo el código de pipeline directamente en Jenkins. Sin embargo, esta no es la única opción. También podemos colocar la definición del pipeline dentro de un archivo llamado Jenkinsfile y enviarlo al repositorio, junto con el código fuente. Este método es aún más consistente porque la apariencia de un pipeline está estrictamente relacionada con el proyecto en sí.

Por ejemplo, si no necesitas la construcción del código porque tu lenguaje de programación se interpreta (y no se compila), no tendrás la etapa Compile. Las herramientas que utilizas también difieren, según el entorno. Usamos Gradle/Maven porque creamos un proyecto Java, sin embargo, en el caso de un proyecto escrito en Python, puedes usar **PyBuilder**. Esto lleva a la idea de que los pipelines deben ser creados por las mismas personas que escriben el código: los desarrolladores.

Además, la definición de pipeline debe estar junto con el código, en el repositorio. Este enfoque trae beneficios inmediatos, como sigue:

- En el caso de una falla de Jenkins, la definición de pipeline no se pierde (porque está almacenado en el repositorio de código, no en Jenkins).
- Se almacena el historial de cambios en el pipeline.
- Los cambios en el pipeline pasan por el proceso de desarrollo de código estándar (por ejemplo, están sujetos a revisiones de código).
- El acceso a los cambios en el pipeline está restringido exactamente de la misma manera que el acceso al

código fuente.

Veamos cómo se ve todo en la práctica creando un archivo Jenkinsfile.

Creando el archivo Jenkins

Podemos crear el archivo Jenkinsfile y enviarlo al repositorio de GitHub. Su contenido es casi el mismo que el commit pipeline que escribimos.

La única diferencia es que la etapa checkout se vuelve redundante porque Jenkins primero tiene que verificar el código (junto con Jenkinsfile) y luego leer la estructura del pipeline (de Jenkinsfile). Esta es la razón por la que Jenkins necesita conocer la dirección del repositorio antes de leer el archivo Jenkins.

Vamos a crear un archivo llamado Jenkinsfile en el directorio raíz de nuestro proyecto:

```
pipeline {
  agent any
  stages {
    stage("Compile") {
      steps {
        sh "./gradlew compileJava"
      }
    }
    stage("Unit test") {
      steps {
        sh "./gradlew test"
      }
    }
  }
}
```

Ahora podemos hacer commit los archivos agregados y push al repositorio de GitHub:

```
$ git add Jenkinsfile
$ git commit -m "Add Jenkinsfile"
$ git push
```

La creación del Jenkinsfile puede evidenciarse en el repositorio, también es sugerible ver los commits para verificar que se ha estado trabajando justo como los pasos han ido indicando: <https://github.com/DagmarLV/calculadorActividad21>

Ejecutar el pipeline desde el Jenkinsfile

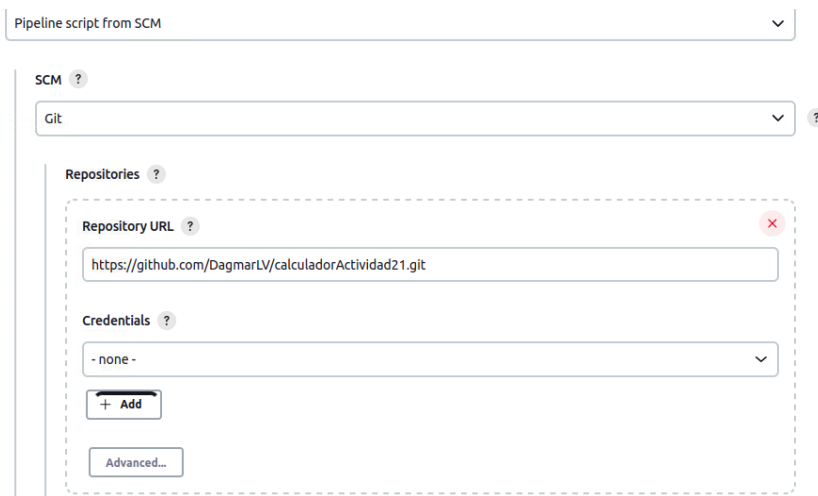
Cuando el Jenkinsfile está en el repositorio, todo lo que tenemos que hacer es abrir la configuración del pipeline y hacer lo siguiente en la sección Pipeline:

1. Cambia Definition desde Pipeline script a Pipeline script from SCM.
2. Seleccione Git en SCM.
3. Coloca <https://github.com/<tu nombre>/calculador.git> en la URL del repositorio.
4. Utiliza */main como Branch Specifier

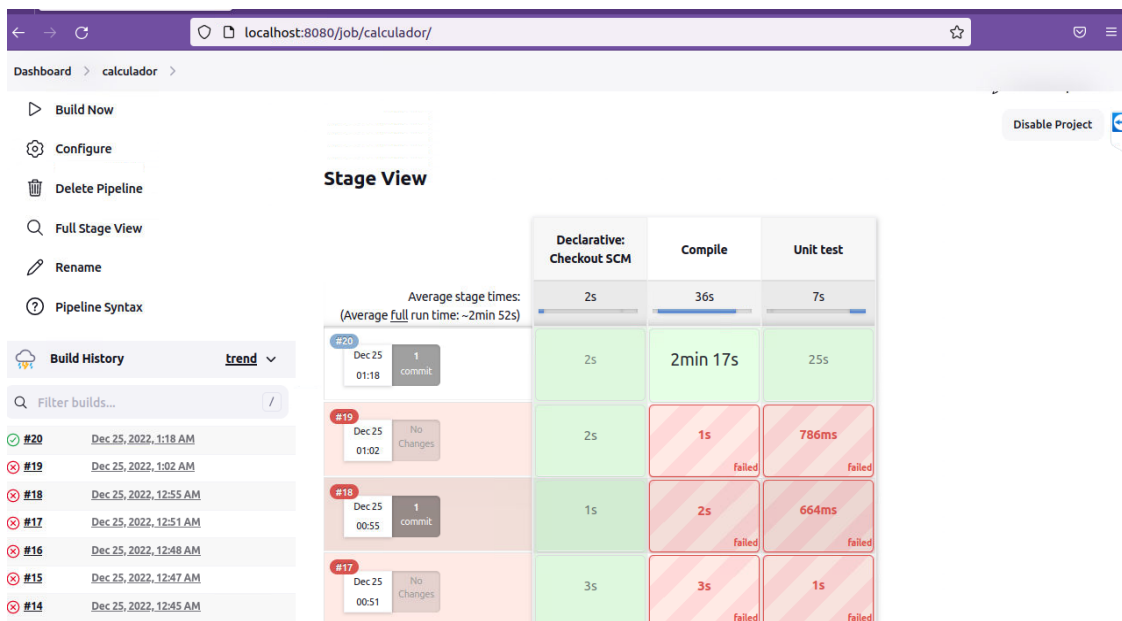
Después de guardar, la construcción siempre se ejecutará desde la versión actual de Jenkinsfile en el repositorio.

Hemos creado con éxito el primer commit pipeline completo. Puede ser tratado como un producto mínimo viable, y de hecho, en muchos casos, esto es suficiente como el proceso de integración continua.

Aquí tenemos la configuración del pipeline modificando las secciones correspondientes



Vemos que cuenta con un URL a nuestro repositorio, al guardarlo ya podemos construir nuestro proyecto correctamente con éxito como se muestra a continuación:



Nota: Como se puede observar han habido varias ejecuciones previas a la correcta que han resultado fallidas, esto es debido a que por alguna razón no se creó un archivo gradlew cuando se creó el proyecto, lo cual generaba conflictos, tal como se puede presenciar a continuación en el Console output de una de las salidas anteriores (Ver imagen más adelante).

Tras encontrar y evaluar el problema, se le dió solución mediante los siguientes comandos mostrados también en una de las imágenes en la parte inferior:

A considerar:

Todo lo correspondiente al manejo de GitHub y el uso del IDE para editar el código de prueba, de producción o demás archivos importantes para el proyecto, fueron manejados en el sistema operativo de Windows10 en mi máquina personal donde uso mi cuenta de github única y exclusivamente. El manejo correspondiente a docker (necesario, en mi caso, para levantar jenkins) y jenkins fue usado en un sistema operativo Linux:Ubuntu 20:04 en una máquina prestada para evitar posibles problemas o inconvenientes durante la prueba, cabe resaltar que no me presté la máquina de nadie del entorno del curso correspondiente.

Console output de las construcciones fallidas:

```
← → ↺ localhost:8080/job/calculador/19/console

Dashboard > calculador > #19

> git config remote.origin.url https://github.com/DagmarLV/calculadorActividad21.git # timeout=10
Fetching upstream changes from https://github.com/DagmarLV/calculadorActividad21.git
> git --version # timeout=10
> git --version # 'git version 2.30.2'
> git fetch --tags --force --progress -- https://github.com/DagmarLV/calculadorActividad21.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git rev-parse refs/remotes/origin/main^{commit} # timeout=10
Checking out Revision b0053067a4e863bc46968cd369a11517bb8e82ac (refs/remotes/origin/main)
> git config core.sparsecheckout # timeout=10
> git checkout -f b0053067a4e863bc46968cd369a11517bb8e82ac # timeout=10
Commit message: "Update Jenkinsfile"
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Compile)
[Pipeline] sh
+ ./gradlew compileJava
/var/jenkins_home/workspace/calculador/tmp/durable-bd7b6335/script.sh: 1: ./gradlew: Permission denied
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Unit test)
Stage "Unit test" skipped due to earlier failure(s)
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
```

Lo usado para repararlo:

```
Nicole@LAPTOP-M3SQF64H MINGW64 ~/Documents/GitHub/calculadorActividad21 (main)
$ git update-index --chmod=+x gradlew

Nicole@LAPTOP-M3SQF64H MINGW64 ~/Documents/GitHub/calculadorActividad21 (main)
$ git add .

Nicole@LAPTOP-M3SQF64H MINGW64 ~/Documents/GitHub/calculadorActividad21 (main)
$ git commit -m "Changing permission of gradlew"
[main 7e898c2] Changing permission of gradlew
1 file changed, 0 insertions(+), 0 deletions(-)
mode change 100644 => 100755 gradlew

Nicole@LAPTOP-M3SQF64H MINGW64 ~/Documents/GitHub/calculadorActividad21 (main)
$ git push
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 247 bytes | 247.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/DagmarLV/calculadorActividad21.git
b005306..7e898c2 main -> main
```

Console Output al conseguir una construcción exitosa:

```
← → ↺ localhost:8080/job/calculador/20/console

Dashboard > calculador > #20

BUILD SUCCESSFUL in 2m
1 actionable task: 1 executed
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Unit test)
[Pipeline] sh
+ ./gradlew test
> Task :compileJava UP-TO-DATE
> Task :processResources
> Task :classes
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test

BUILD SUCCESSFUL in 24s
4 actionable tasks: 3 executed, 1 up-to-date
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```