

Curso de desarrollo de software

Actividad individual

En las siguientes secciones, veremos qué mejoras se pueden hacer para mejorar aún más el commit pipeline .

Etapas de calidad de código

Podemos extender los tres pasos clásicos de integración continua con pasos adicionales. Los más populares son la cobertura de código y el análisis estático. Veamos cada uno de ellos.

Cobertura de código

Tienes un proceso de integración continua bien configurado, sin embargo, nadie en tu proyecto escribe pruebas unitarias.

Pasa todas las construcciones, pero no significa que el código funcione como se esperaba. ¿Qué hacemos entonces? ¿Cómo nos aseguramos de que el código sea probado?

La solución es agregar una herramienta de cobertura de código que ejecute todas las pruebas y verifique qué partes del código se han ejecutado. Luego, puedes crear un informe que muestre las secciones no probadas. Además, podemos hacer que la construcción falle cuando hay demasiado código sin probar.

Hay muchas herramientas disponibles para realizar el análisis de cobertura de prueba para Java, las más populares son JaCoCo, OpenClover y Cobertura.

Usemos JaCoCo y mostremos cómo funciona la verificación de cobertura. Para ello, debemos realizar los siguientes pasos:

1. Agrega JaCoCo a la configuración de Gradle.
2. Agrega la etapa de cobertura de código al pipeline.
3. Opcionalmente, publica informes de JaCoCo en Jenkins.

Veamos estos pasos en detalle.

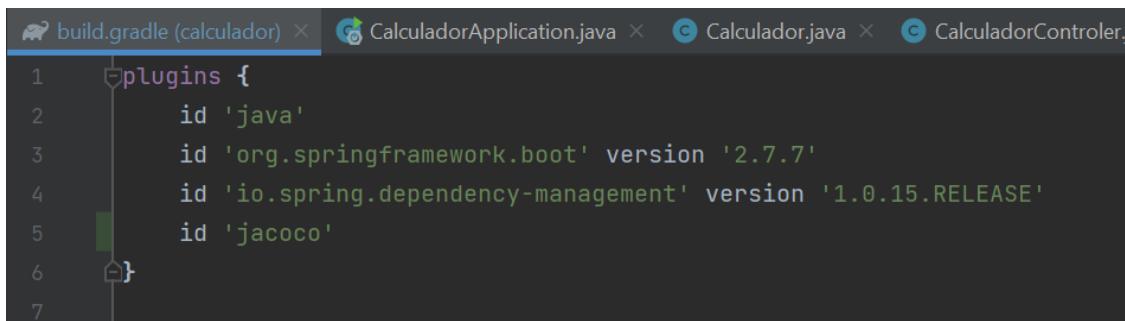
Agregar JaCoCo a Gradle

Para ejecutar JaCoCo desde Gradle, debemos agregar el complemento jacoco al archivo build. El archivo gradle insertando la siguiente línea:

```
plugins {
```

```
    ...
    id 'jacoco'
}
```

Aquí se muestra la parte de plugins del archivo gradle.build agregando el plugin 'jacoco' para usar la herramienta de cobertura JaCoCo en nuestro proyecto:



```
build.gradle (calculador) ✘ CalculadoraApplication.java ✘ Calculador.java ✘ CalculadorController.java
1  plugins {
2      id 'java'
3      id 'org.springframework.boot' version '2.7.7'
4      id 'io.spring.dependency-management' version '1.0.15.RELEASE'
5      id 'jacoco'
6  }
7
```

A continuación, si queremos que Gradle falle en caso de baja cobertura de código, podemos agregar la siguiente configuración al archivo build.gradle:

```
jacocoTestCoverageVerification {
```

```

violationRules{
    rule{
        limit{
            minimum = 0.2
        }
    }
}

```

Se muestra la sección de jacocoTestCoverageVerification en el archivo build.gradle de nuestro proyecto, esto establece un límite mínimo de cobertura de 20%, en caso no cumpla ello, el proyecto fallará:

```

build.gradle (calculador) × CalculadoraApplication.java × Calculador.java × CalculadorController.java ×
20
21
22
23 > jacocoTestCoverageVerification {
24     violationRules { JacocoViolationRulesContainer it ->
25         rule { JacocoViolationRule it ->
26             limit { JacocoLimit it ->
27                 minimum = 0.2
28             }
29         }
30     }
31 }

```

Esta configuración establece la cobertura de código mínima en un 20 %. Podemos ejecutarlo con el siguiente comando:

```
$ ./gradlew test jacocoTestCoverageVerification
```

El comando puede ser reemplazado por gradle build test jacocoTestCoverageVerification para la consola de Windows, tal como funciona a continuación:

```
C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build test jacocoTestCoverageVerification
BUILD SUCCESSFUL in 6s
8 actionable tasks: 2 executed, 6 up-to-date
```

La construcción ha resultado exitosa, entonces podemos afirmar que la cobertura ha pasado del 20% como mínimo, pero podemos establecer otras condiciones más estrictas para poner a prueba su calidad, como por ejemplo podemos determinar a **minimum = 0.8**, así fallará al no cumplir una cobertura del 80%:

```
C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build test jacocoTestCoverageVerification
> Task :jacocoTestCoverageVerification FAILED
[ant:jacocoReport] Rule violated for bundle calculador: instructions covered ratio is 0.2, but expected minimum is 0.8

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':jacocoTestCoverageVerification'.
> Rule violated for bundle calculador: instructions covered ratio is 0.2, but expected minimum is 0.8

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 1s
8 actionable tasks: 1 executed, 7 up-to-date
```

Podemos ver que la construcción ha fallado con una excepción y describe que el índice de instrucciones cubiertas es 0.2, pero el mínimo esperado es 0.8, tal como lo habíamos establecido.

Como el radio es 0.2, pensé en probar la construcción cambiando el valor de **minimum** con valores entre 0.20 hasta 0.30 y las construcciones hasta llegar al valor de 0.25 fueron exitosas, pero al momento de tomar el valor de 0.26 la construcción falla e indica que el índice de instrucciones cubiertas es 0.25 como se exhibe a continuación:

```
C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build test jacocoTestCoverageVerification
> Task :jacocoTestCoverageVerification FAILED
[ant:jacocoReport] Rule violated for bundle calculador: instructions covered ratio is 0.25, but expected minimum is 0.26
FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':jacocoTestCoverageVerification'.
> Rule violated for bundle calculador: instructions covered ratio is 0.25, but expected minimum is 0.26

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 1s
8 actionable tasks: 1 executed, 7 up-to-date

C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build test jacocoTestCoverageVerification
```

Este comando verifica si la cobertura del código es al menos del 20%. Puedes jugar con el valor mínimo para ver el nivel en el que falla la construcción. También podemos generar un informe de cobertura de prueba usando el siguiente comando:

```
$ ./gradlew test jacocoTestReport
```

Dejaremos el valor de **minimum** con 0.25 y ahora generaremos el informe de cobertura con el comando gradle build test jacocoTestReport

```
C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build test jacocoTestReport
BUILD SUCCESSFUL in 1s
8 actionable tasks: 1 executed, 7 up-to-date
```

Puede consultar el informe de cobertura completo en el archivo build/reports/jacoco/test/html/index.html:

Podremos comprobar que, en efecto, la construcción ha sido exitosa al acceder al path:

C:\Users\Nicole\Documents\GitHub\calculadorActividad21\build\reports\jacoco\test\html, en mi caso. Esta es la pantalla que se muestra inicialmente al acceder a index.html:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes				
com.checha.calculador	25%	n/a	4	6	5	7	4	6	2	3		
Total	20 of 27	25%	0 of 0	n/a	4	6	5	7	4	6	2	3

Presenciamos que la cobertura es del 25%, que coincide justamente con lo que probamos anteriormente, ahora accedemos al elemento com.checha.calculador para poder tener más detalles

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes				
CalculadorController	0%	n/a	2	2	2	2	2	1	1			
CalculadorApplication	0%	n/a	2	2	3	3	2	2	1	1		
Calculador	100%	n/a	0	2	0	2	0	2	0	1		
Total	20 of 27	25%	0 of 0	n/a	4	6	5	7	4	6	2	3

Podemos acceder a cada uno de los elementos para tener más detalles todavía, pero con lo que tenemos, podemos decir que solo se ha probado la clase Calculador, lo cual es congruente con lo que hemos implementado, ya que solo hicimos una prueba testSum para la función sum de la clase Calculador

Ahora agregamos la etapa de cobertura en el pipeline.

Agregando una etapa de cobertura de código

Agregar una etapa de cobertura de código al pipeline es tan simple como las etapas anteriores:

```
stage("Code coverage") {  
    steps {  
        sh "./gradlew jacocoTestReport"  
        sh "./gradlew jacocoTestCoverageVerification"  
    }  
}
```

A continuación, podemos agregar tranquilamente el stage Code coverage, en el que se agregan los comandos a ser ejecutados para generar el reporte y su verificación de cobertura en el pipeline de nuestro Jenkinsfile:

```
14 |         stage("Code coverage") {  
15 |             steps {  
16 |                 sh "./gradlew jacocoTestReport"  
17 |                 sh "./gradlew jacocoTestCoverageVerification"  
18 |             }  
19 |         }
```

Después de agregar esta etapa, si alguien hace commit en el código que no está bien cubierto con las pruebas, la construcción fallará.

Ya que al final terminamos definiendo **minimum = 0.25**, podemos presenciar que al construir se detectan los commits y todos los pasos, incluyendo al último implementado, han sido ejecutados con éxito.

The screenshot shows the Jenkins UI for the 'calculator' pipeline. On the left, there's a sidebar with options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, Rename, and Pipeline Syntax. Below that is the Build History section, which lists two builds: #21 (Dec 25, 2022, 10:39 PM) and #20 (Dec 25, 01:18 AM). The main area is titled 'Pipeline calculator' and shows the 'Stage View' table. The table has four columns: Declarative: Checkout SCM, Compile, Unit test, and Code coverage. For build #21, the times are 2s, 37s, 9s, and 15s respectively. For build #20, the times are 2s, 43s, 19s, and 15s. The table also includes average stage times: 2s, 37s, 9s, and 15s. A note at the top of the table says 'Average stage times: (Average full run time: ~2min 17s)'.

Declarative: Checkout SCM	Compile	Unit test	Code coverage
2s	37s	9s	15s
#21 Dec 25 22:39 2 commits	2s	43s	19s
#20 Dec 25 01:18 1 commit	2s	2min 17s	25s

Publicación del informe de cobertura de código

Podemos ejecutar Gradle localmente y generar el informe de cobertura, sin embargo, es más conveniente que Jenkins nos muestre el informe.

Para publicar el informe de cobertura de código en Jenkins, requerimos la siguiente definición de etapa:

```
stage("Code coverage") {
    steps {
        sh "./gradlew jacocoTestReport"
        publishHTML (target: [
            reportDir: 'build/reports/jacoco/test/html',
            reportFiles: 'index.html',
            reportName:"JaCoCo Report"
        ])
        sh "./gradlew jacocoTestCoverageVerification"
    }
}
```

También agregamos al stage anterior la sección de publishHTML, en el pipeline de nuestro Jenkinsfile, para lograr que Jenkins nos muestre el informe generado,:



The screenshot shows a Jenkinsfile editor with several tabs at the top: Jenkinsfile (selected), CalculadoraApplication.java, Calculador.java, and CalculadorController.java. The Jenkinsfile content is as follows:

```
11 |         sh "./gradlew test"
12 |     }
13 | }
14 | stage("Code coverage") {
15 |     steps {
16 |         sh "./gradlew jacocoTestReport"
17 |         publishHTML (target: [
18 |             reportDir: 'build/reports/jacoco/test/html',
19 |             reportFiles: 'index.html',
20 |             reportName:"JaCoCo Report"
21 |         ])
22 |         sh "./gradlew jacocoTestCoverageVerification"
23 |
24 |     }
25 |
26 | }
27 |
28 }
```

Esta etapa copia el informe JaCoCo generado en la salida de Jenkins. Cuando volvamos a ejecutar la construcción, deberíamos ver un enlace a los informes de cobertura de código (en el menú del lado izquierdo, debajo de Build Noe).

Ahora pongamos a prueba el funcionamiento de la construcción en la imagen a continuación.

En esta se muestra que al llegar al stage Code coverage, el pipeline presenta un error y la ejecución ha resultado fallida, el error será reparado posteriormente.

Información

Para realizar el paso de publicación HTML, debes tener instalado el complemento HTML Publisher en Jenkins. Puede leer más sobre el complemento en <https://www.jenkins.io/doc/pipeline/steps/htmlpublisher/>. Ten en cuenta también que si el informe se genera pero no se muestra correctamente en Jenkins, es posible que deba configurar Jenkins Security, como se describe aquí: <https://www.jenkins.io/doc/book/security/configuring-content-security-policy/>.

Hemos creado la etapa de cobertura de código, que muestra el código que no se prueba y, por lo tanto, es vulnerable a errores. Veamos qué más se puede hacer para mejorar la calidad del código.

Para reparar el error que presentamos en la construcción anterior, accederemos a la sección de manage jenkins para manejar los plugins y vemos, efectivamente, que el complemento HTML publisher no se encuentra y es necesario instalarlo para publicar nuestro index.html en el que se mostrará el reporte de cobertura, así que procedemos con su instalación.

Ahora, podemos comprobar si es que con este cambio la construcción resulta exitosa

The screenshot shows the Jenkins Pipeline calculator stage view. On the left, there's a sidebar with options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, Rename, Pipeline Syntax, and Build History. The Build History section shows two builds: #23 (Dec 25, 2022, 10:51 PM) which succeeded, and #22 (Dec 25, 2022, 10:47 PM) which failed. The main area displays a Stage View grid with four columns: Declarative: Checkout SCM, Compile, Unit test, and Code coverage. Each column has a horizontal bar indicating average stage times: 2s, 32s, 8s, and 10s respectively. Below the bars are individual stage boxes. The 'Code coverage' box for build #22 is highlighted in red and labeled 'failed'.

Nota: Para poder visualizar el reporte generado en el menú del lado izquierdo es necesario recargar la página, yo no me percaté de ello hasta pasos posteriores, pero se muestra cómo se ve a continuación:

The screenshot shows the Jenkins Pipeline calculator sidebar. It includes options like Dashboard, Configure, Delete Pipeline, Full Stage View, JaCoCo Report (which is highlighted with a purple arrow), Rename, and Pipeline Syntax. A purple arrow points from the text above to the 'JaCoCo Report' option in the sidebar.

Y al acceder podemos ver lo siguiente, que es justamente el reporte de cobertura generado:

The screenshot shows the generated JaCoCo report for the 'calculador' pipeline. The report title is 'calculador'. It contains a table with columns for Element, Missed Instructions, Cov., Missed Branches, Cov., Missed, Cxty, Missed, Lines, Missed, Methods, and Missed Classes. The first row shows data for 'com.checha.calculador': Missed Instructions: 25%, Cov.: 75%, Missed Branches: n/a, Cov.: 0 of 0, Missed: 4, Cxty: 6, Missed: 5, Lines: 7, Missed: 4, Methods: 6, Missed: 2, Classes: 3. The total row shows 20 of 27, 25%, 0 of 0, n/a, 4, 6, 5, 7, 4, 6, 2, 3. At the bottom, it says 'Created with JaCoCo 0.8.8.202204050719'.

Información

Si necesitas una cobertura de código más estricta, puedes consultar el concepto de prueba de mutación y agregar la etapa del framework PIT al pipeline. Lee más en <http://pitest.org/>.

Análisis de código estático

El análisis de código estático es un proceso automático de verificación de código sin ejecutarlo realmente. En la mayoría de los casos, implica verificar una serie de reglas en el código fuente. Estas reglas pueden aplicarse a

una amplia gama de aspectos, por ejemplo, todas las clases públicas deben tener un comentario de Javadoc, la longitud máxima de una línea es de 120 caracteres, o si una clase define el método equals(), también debe definir el método hashCode().

Las herramientas más populares para realizar análisis estáticos en código Java son Checkstyle, FindBugs y PMD. Veamos un ejemplo y agreguemos la etapa de análisis de código estático usando Checkstyle. Lo haremos en tres pasos:

1. Agregamos la configuración de Checkstyle
2. Agregamos la etapa Checkstyle
3. Opcionalmente, publicamos el informe Checkstyle en Jenkins

Pasaremos por cada uno de ellos.

Agregamos la configuración de Checkstyle

Para agregar la configuración de Checkstyle, necesitamos definir las reglas contra las cuales se verifica el código. Podemos hacer esto especificando el archivo config/checkstyle/checkstyle.xml:

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
"-//Puppy Crawl//DTD Check Configuration 1.2//EN"
"http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
<module name="TreeWalker">
<module name="ConstantName"/>
</module>
</module>
```

La configuración contiene solo una regla: checking si todas las constantes de Java siguen la convención de nomenclatura y consisten solo en caracteres en mayúsculas.

Agregamos lo necesario para la configuración de Checkstyle



```
Jenkinsfile × checkstyle.xml × CalculadorApplication.java × Calculador.java × CalculadorApplicationTest.java ×
1  <?xml version="1.0"?>
2  <!DOCTYPE module PUBLIC
3      "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
4      "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
5  <module name="Checker">
6      <module name="TreeWalker">
7          <module name="ConstantName"/>
8      </module>
9  </module>
```

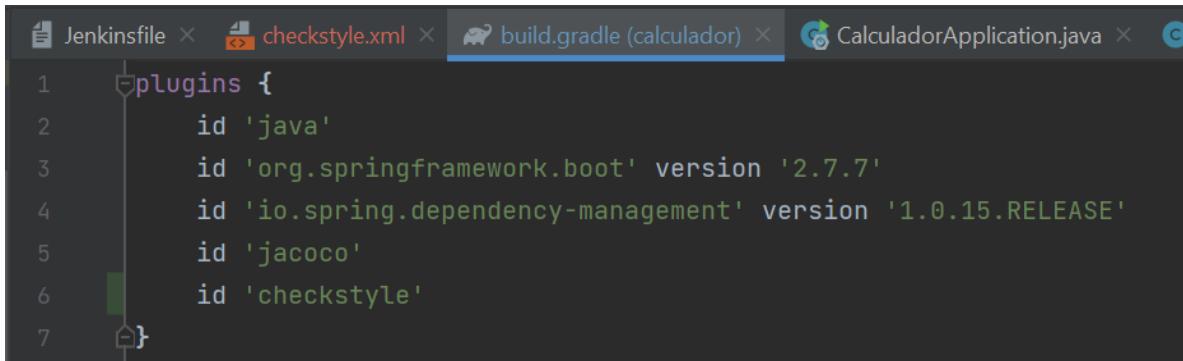
Información

La descripción completa de Checkstyle se puede encontrar en:
<https://checkstyle.sourceforge.io/config.html>.

También necesitamos agregar el complemento checkstyle al archivo build.gradle:

```
plugins {
    ...
    id 'checkstyle'
```

Agregamos el plugin en la sección respectiva del build.gradle:



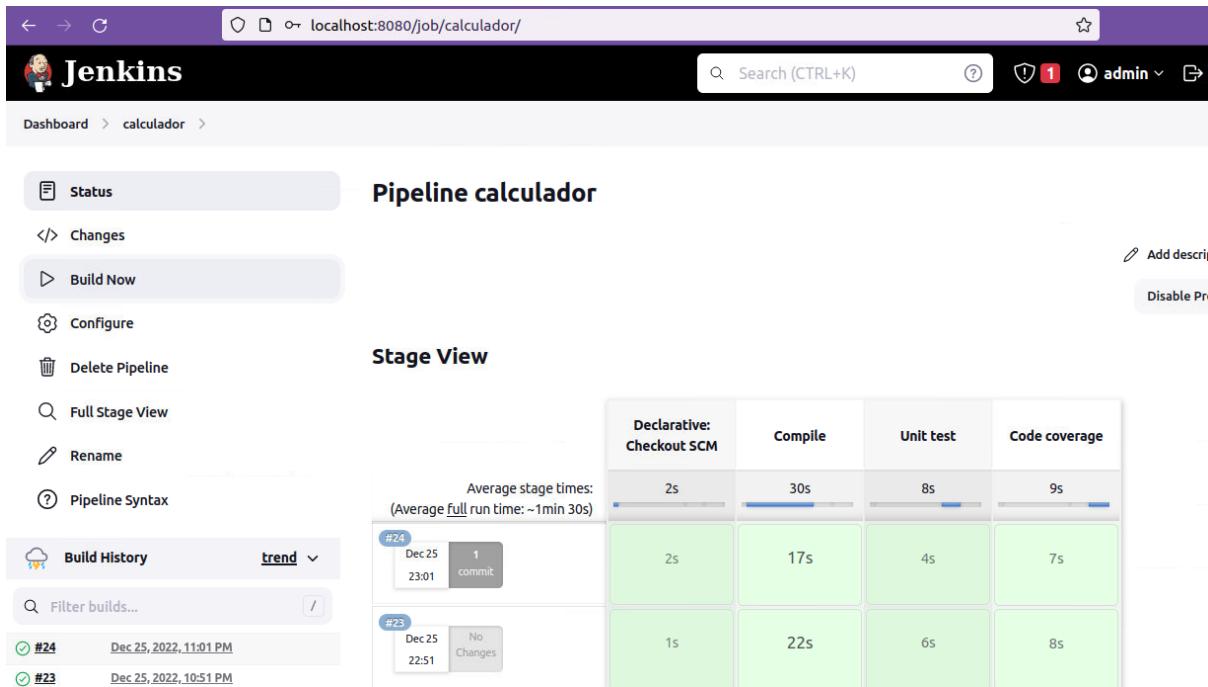
```
plugins {
    id 'java'
    id 'org.springframework.boot' version '2.7.7'
    id 'io.spring.dependency-management' version '1.0.15.RELEASE'
    id 'jacoco'
    id 'checkstyle'
}
```

Luego, podemos ejecutar checkstyle con el siguiente comando:
\$./gradlew checkstyleMain

Se muestra la consola al lograr una construcción exitosa del checkstyleMain

```
C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build checkstyleMain
BUILD SUCCESSFUL in 17s
9 actionable tasks: 2 executed, 7 up-to-date
```

También podemos verificar que la construcción se completa con éxito en Jenkins, por lo que podemos decir que hasta ahora todo está funcionando adecuadamente



En el caso de la actividad, este comando debería completarse con éxito porque hasta ahora no usamos ninguna constante. Sin embargo, puedes intentar agregar una constante con el nombre incorrecto y verificar si la construcción falla. Por ejemplo, si agregas la siguiente constante al archivo src/main/java/com/<tu nombre>/calculador/CalculadorAplicacion.java, checkstyle falla:

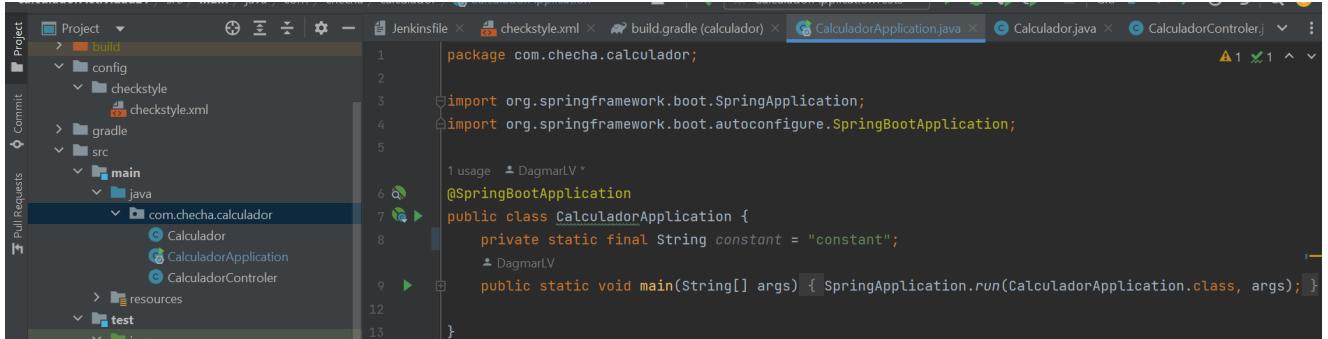
@SpringBootApplication

```

public class CalculatorApplication {
    private static final String constant = "constant";
    public static void main(String[] args) {
        SpringApplication.run(CalculatorApplication.class,
        args);
    }
}

```

Ahora agregamos una variable constante definida incorrectamente en la clase CalculadorApplication para probar la falla del checkstyle



Justamente al construir con el mismo comando, como se hizo anteriormente, este resultará con error

```

C:\Users\Nicole\Documents\GitHub\calculadorActividad21>gradle build checkstyleMain

> Task :checkstyleMain FAILED
[ant:checkstyle] [ERROR] C:\Users\Nicole\Documents\GitHub\calculadorActividad21\src\main\java\com\checha\calculador\CalculadorApplication.java:8:37: El nombre 'constant' debe coincidir con el patrón '[A-Z][A-Z0-9]*[_A-Z0-9]+*$'. [ConstantName]

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':checkstyleMain'.
> A failure occurred while executing org.gradle.api.plugins.quality.internal.CheckstyleAction
  > Checkstyle rule violations were found. See the report at: file:///C:/Users/Nicole/Documents/GitHub/calculadorActividad21/build/reports/checkstyle/main.html
    Checkstyle files with violations: 1
    Checkstyle violations by severity: [error:1]

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 4s
9 actionable tasks: 7 executed, 2 up-to-date

```

Agregando una etapa de análisis de código estático

Podemos agregar una etapa Static code analysis al pipeline:

```

stage("Static code analysis") {
    steps {
        sh "./gradlew checkstyleMain"
    }
}

```

Ahora agregaremos el stage al pipeline para indicar que se ejecute la construcción de checkstyleMain

```

17     publishHTML (target: [
18         reportDir: 'build/reports/jacoco/test/html',
19         reportFiles: 'index.html',
20         reportName:"JaCoCo Report"
21     ])
22     sh "./gradlew jacocoTestCoverageVerification"
23
24 }
25
26
27 stage("Static code analysis") {
28     steps {
29         sh "./gradlew checkstyleMain"
30     }
31 }
32
33
34 }
35

```

Ahora, si alguien hace commit cualquier código que no siga la convención de nomenclatura constante de Java, la construcción fallará.

Ahora lo comprobamos en el Jenkins y podemos presenciar que todo el pipeline se ha ejecutado exitosamente:

	Declarative: Checkout SCM	Compile	Unit test	Code coverage	Static code analysis
#26 Dec 25 23:13 1 commit	2s	28s	8s	10s	29s
#25 Dec 25 23:08 1 commit	1s	20s	9s	10s	29s
#24	3s	21s	9s	13s	

Publicación de informes de análisis de código estático

Muy similar a JaCoCo, podemos agregar el informe Checkstyle a Jenkins:

```

publishHTML (target: [
    reportDir: 'build/reports/checkstyle/',
    reportFiles: 'main.html',
    reportName:"Checkstyle Report"
])

```

Aca agregamos el coso eso

```
1 Jenkinsfile
22
23
24
25
26
27     stage("Static code analysis") {
28         steps {
29             sh "./gradlew checkstyleMain"
30             publishHTML (target: [
31                 reportDir: 'build/reports/checkstyle/',
32                 reportFiles: 'main.html',
33                 reportName:"Checkstyle Report"
34             ])
35         }
36     }
37
38
39 }
40 }
```

Esto genera un enlace al informe Checkstyle.

Ahora hemos agregado la etapa de análisis de código estático, que puede ayudar a encontrar errores y estandarizar el estilo de código dentro de un equipo u organización.

Ahora podemos ver el informe de checkstyle al momento de construir y al recargar la pagina aparecerá en la parte izquierda el Checkstyle Report

The screenshot shows the Jenkins Pipeline calculator job dashboard. On the left, there's a sidebar with options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, JaCoCo Report, Checkstyle Report, Rename, Pipeline Syntax, Build History (with a trend dropdown), and a Filter builds... search bar. The main area is titled "Pipeline calculator". It has a "Stage View" section with a table showing average stage times for three builds (#27, #26, #25). The columns represent stages: Declarative: Checkout SCM, Compile, Unit test, Code coverage, and Static code analysis. The table shows times ranging from 1s to 30s. Below the table, there's a "Checkstyle Report" link. The URL in the browser bar is "localhost:8080/job/calculador/".

Al acceder en Checkstyle Report podemos visualizar lo siguiente:

The screenshot shows a Jenkins job's Checkstyle report page. At the top, the URL is `localhost:8080/job/calculador/Checkstyle_20Report/`. Below the header, there are links for "Back to calculador" and "main". The main title is "CheckStyle Audit". A note says "Designed for use with [CheckStyle](#) and [Ant](#)".

Summary

Files	Errors
3	0

Files

Name	Errors
/var/jenkins_home/workspace/calculador/src/main/java/com/checha/calculador/CalculadorController.java	0
/var/jenkins_home/workspace/calculador/src/main/java/com/checha/calculador/Calculador.java	0
/var/jenkins_home/workspace/calculador/src/main/java/com/checha/calculador/CalculadorApplication.java	0

File /var/jenkins_home/workspace/calculador/src/main/java/com/checha/calculador/CalculadorController.java

Error Description	Line
-------------------	------

File /var/jenkins_home/workspace/calculador/src/main/java/com/checha/calculador/Calculador.java

Error Description	Line
-------------------	------

[Back to top](#)

SonarQube

SonarQube es la herramienta de gestión de calidad de código fuente más extendida. Es compatible con varios lenguajes de programación y se puede tratar como una alternativa a los pasos de análisis de código estático y cobertura de código que vimos. En realidad, es un servicio separado que agrega diferentes framework de análisis de código, como Checkstyle, FindBugs y JaCoCo. Tiene sus propios dashboards y se integra bien con Jenkins.

En lugar de agregar pasos de calidad de código al pipeline, podemos instalar SonarQube, agregar complementos allí y agregar una etapa sonar al pipeline. La ventaja de esta solución es que SonarQube proporciona una interfaz web fácil de usar para configurar reglas y mostrar vulnerabilidades de código.

Información

Puedes leer más sobre SonarQube en su página oficial en <https://www.sonarqube.org/>.

Triggers y notificaciones

En esta sección, veremos cómo mejorar el proceso para que el pipeline se inicie automáticamente y, cuando se complete, notifique a los miembros del equipo sobre su estado.

Triggers

Una acción automática para iniciar la construcción se denomina pipeline trigger. En Jenkins, hay muchas opciones para elegir, sin embargo, todos se reducen a tres tipos:

- External
- Source Control Management(SCM)
- Construcciones programadas

Echemos un vistazo a cada uno de ellos.

External

Los triggers externos son fáciles de entender. Significan que Jenkins inicia la construcción después de que lo llame el notificador (notifier), que puede ser otra construcción de un pipeline, el sistema SCM (por ejemplo, GitHub) o cualquier secuencia de comandos remota.

Para configurar el sistema de esta manera, necesitamos los siguientes pasos de configuración:

1. Instala el complemento de GitHub en Jenkins.
2. Genera una clave secreta para Jenkins.
3. Configura el webhook de GitHub y especifica la dirección y la clave de Jenkins.

En el caso de los proveedores de SCM más populares, siempre se proporcionan complementos dedicados de Jenkins.

También hay una forma más genérica de activar Jenkins a través de la llamada REST al punto final <jenkins_url>/job/<job_name>/build?token=<token>. Por razones de seguridad, se requiere configurar el token en Jenkins y luego usarlo en el script remoto.

Información

Jenkins debe ser accesible desde el servidor SCM. En otras palabras, si usamos el repositorio público de GitHub para activar Jenkins, el servidor Jenkins también debe ser público. Esto también se aplica a la solución de llamada REST, en cuyo caso, la dirección <jenkins_url> debe ser accesible desde el script que la activa.

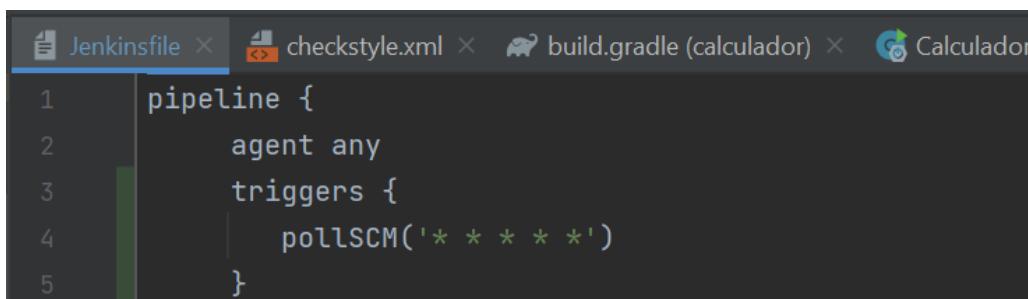
SCM

SCM es un poco menos intuitivo.

La configuración de pollSCM también es de alguna manera más simple porque la forma de conectarse de Jenkins a GitHub ya está configurada (Jenkins verifica el código de GitHub, por lo que sabe cómo acceder a él). En el caso de la actividad del proyecto del calculador, podemos configurar un triggers automático agregando la declaración triggers (justo después del agente) al pipeline:

```
triggers{  
    pollSCM('* * * * *')  
}
```

Agregamos el trigger para que comience las construcciones automáticamente al recibir un nuevo commit



The screenshot shows a code editor with several tabs at the top: 'Jenkinsfile' (which is the active tab), 'checkstyle.xml', 'build.gradle (calculador)', and 'Calculador'. The Jenkinsfile content is as follows:

```
1 pipeline {  
2     agent any  
3     triggers {  
4         pollSCM('* * * * *')  
5     }  
6 }
```

Después de ejecutar el pipeline manualmente por primera vez, se establece el triggers automático. Luego, verifica GitHub cada minuto y para nuevos commits, comienza una construcción. Para probar que funciona como se esperaba, puedes realizar commits y push de cualquier cosa al repositorio de GitHub y ver que comienza la construcción.

Lo construimos y verificamos que la construcción no presente problemas:

The screenshot shows the Jenkins Pipeline calculator status page. On the left, there's a sidebar with options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, JaCoCo Report, Checkstyle Report, Rename, Pipeline Syntax, and Build History. The main area is titled "Pipeline calculador". It displays a "Stage View" grid with columns for Declarative: Checkout SCM (2s), Compile (28s), Unit test (8s), Code coverage (9s), and Static code analysis (12s). Below the grid, two builds are listed: #28 (Dec 25, 23:33, 1 commit) and #27 (Dec 25, 23:23, 1 commit). The stage view for build #28 shows the following times: Declarative: Checkout SCM (2s), Compile (1s), Unit test (15s), Code coverage (5s), and Static code analysis (6s). The stage view for build #27 shows: Declarative: Checkout SCM (2s), Compile (2s), Unit test (17s), Code coverage (5s), and Static code analysis (8s).

Y ahora realizaremos una construcción y agregaremos una línea de comentario para probar que la construcción sea automática al hacer un nuevo commit

```

usage -- DagmarV
@SpringBootApplication
public class CalculadoraApplication {
    //Este es un comentario para probar que la construcción es automática
    ▲ DagmarV
    public static void main(String[] args) { SpringApplication.run(CalculadoraApplication.class, args);
}

```

Justo como se esperaba, la construcción fue automática y se realizó sin necesidad de hacer click al build now, esta también resulta exitosa como la anterior construcción

The screenshot shows the Jenkins Pipeline calculator status page again. The sidebar and main title are identical. The build history now includes three entries: #29 (Dec 25, 23:37, 1 commit) at the top, followed by #28 (Dec 25, 23:23, 1 commit) and #27 (Dec 25, 23:23, 1 commit) below it. The stage view for build #29 shows: Declarative: Checkout SCM (2s), Compile (26s), Unit test (8s), Code coverage (9s), and Static code analysis (12s). The stage view for build #28 shows: Declarative: Checkout SCM (2s), Compile (2s), Unit test (3s), Code coverage (6s), and Static code analysis (4s). The stage view for build #27 remains the same as in the previous screenshot.

Usamos * * * * * como argumento para pollSCM. Esto especifica la frecuencia con la que Jenkins debe verificar si hay nuevos cambios en la fuente y se expresa en el formato de cadena de estilo cron.

Información

El formato de la cadena cron se describe (junto con la herramienta cron) en <https://en.wikipedia.org/wiki/Cron>.

Construcciones programadas

El trigger programado significa que Jenkins ejecuta la construcción periódicamente, independientemente de si hubo commits en el repositorio.

La implementación de Scheduled build es exactamente igual que el SCM. La única diferencia es que se utiliza la palabra clave cron en lugar de pollSCM. Este método rara vez se usa para el commit pipeline, pero se aplica bien a las nightly builds (por ejemplo, pruebas de integración complejas ejecutadas por la noche).

Notificaciones

Jenkins proporciona muchas formas de anunciar tu estado de construcción. Además, como todo en Jenkins, se pueden agregar nuevos tipos de notificaciones mediante complementos.

Repasemos los tipos más populares para que puedas elegir el que se ajuste a tus necesidades.

Correo electrónico

La forma más clásica de notificar a los usuarios sobre el estado de construcción de Jenkins es enviar correos electrónicos.

La configuración de la notificación por correo electrónico es muy sencilla:

1. Tener configurado el servidor SMTP (Simple Mail Transfer Protocol).
2. Configura tus detalles en Jenkins (en Manage Jenkins | Configure system).
3. Utiliza el correo para la instrucción en el pipeline.

La configuración del pipeline puede ser la siguiente:

```
post {
    always {
        mail to: 'team@company.com',
        subject:"Completed Pipeline: ${currentBuild.
            fullDisplayName}",
        body:"Your build completed, please check: ${env. BUILD_URL}"
    }
}
```

Agregamos al pipeline lo siguiente para que se envíe un correo a mi cuenta institucional cada que se ejecute una construcción, por ello el always y muestra un poco más de información respecto a la construcción.

```
42      post {
43          always {
44              mail to: 'dagmar.lezama.v@uni.com',
45              subject:"Completed Pipeline: ${currentBuild.fullDisplayName}",
46              body:"Your build completed, please check: ${env. BUILD_URL}"
47          }
48      }
```

Es importante configurar bien esta sección para que no hayan problemas:

localhost:8080/manage/configure

E-mail Notification

SMTP server: smtp.gmail.com

Default user e-mail suffix:

Use SMTP Authentication: User Name: dagmar.lezama.v@uni.pe, Password:
 Use SSL
 Use TLS

Save **Apply**

Yo presenté ciertos problemas al configurarlo, por eso tengo construcciones finalizadas sin éxito como se podrá observar más adelante, para completar bien esta sección me fue necesario averiguar sobre la configuración de SMTP y sus puertos, así como también activar la contraseña de aplicaciones de dónde Jenkins enviará el correo.

Seguridad

Contactos y compartir
Pagos y suscripciones
Información general

Iniciar sesión en Google



Contraseña Última modificación: 22 nov >

Verificación en dos pasos Activada >

Contraseñas de aplicaciones 1 contraseña >

←

Para evitar tener más construcciones sin éxito, es recomendable usar el Test configuration para comprobar que la configuración puesta es efectivamente correcta

localhost:8080/manage/configure

SMTP Port: 587

Reply-To Address:

Charset: UTF-8

Test configuration by sending test e-mail
Test e-mail recipient: dagmar.lezama.v@uni.pe
Email was successfully sent

Test configuration ←

Save **Apply**

Después de tantas pruebas, obtuve éxito en la #35, como se muestra a continuación, un mensaje enviado por Jenkins:

This screenshot shows an email from Jenkins. The subject is "Test email #35". The recipient is "dagmar.lezama.v@uni.pe" (with "para mí" selected). The message body says "This is test email #35 sent from Jenkins". Below the message are two buttons: "Responder" and "Reenviar".

Ahora podemos comprobar que la construcción será exitosa al seleccionar Build now y tras esperar un tiempo recibí el correo mostrado a continuación, tal como lo describimos en el pipeline:

This screenshot shows an email from Jenkins. The subject is "Completed Pipeline: calculador #33". The recipient is "dagmar.lezama.v@uni.pe" (with "para mí" selected). The message body says "Your build completed, please check: <http://localhost:8080/job/calculador/33/>". Below the message are two buttons: "Responder" and "Reenviar".

Aquí se muestra la pantalla de etapas de nuestro pipeline en Jenkins, siendo el número #33 la última construcción que ha resultado con éxito.

This screenshot shows the Jenkins Pipeline calculator stage view. On the left, there is a sidebar with various options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, JaCoCo Report, Checkstyle Report, Rename, Pipeline Syntax, Git Polling Log, Build History, and a Filter builds... search bar. The main area is titled "Pipeline calculador" and shows a "Stage View" grid. The grid has columns for "Declarative: Checkout SCM", "Compile", "Unit test", "Code coverage", "Static code analysis", and "Declarative: Post Actions". There are three rows of data, each representing a build: #33 (Dec 26 15:45, No Changes), #32 (Dec 26 14:03, No Changes), and #31 (Dec 25, 1). Each row contains six time values: 2s, 27s, 6s, 8s, 5s, and 4s respectively. Above the grid, it says "Average stage times: (Average full run time: ~53s)".

Chats grupales

Si un chat grupal (por ejemplo, Slack) es el primer método de comunicación en tu equipo, vale la pena considerar agregar las notificaciones de construcción automáticas allí. Independientemente de la herramienta que utilices, el procedimiento para configurarla es siempre el mismo:

1. Busca e instala el complemento para tu herramienta de chat grupal (por ejemplo, el complemento Slack Notification).
2. Configura el complemento (la URL del servidor, el canal, el token de autorización, etc.).
3. Agrega la instrucción de envío al pipeline.

Veamos una configuración de pipeline de muestra para que Slack envíe notificaciones después de que falla la construcción:

```
post {
    failure {
        slackSend channel: '#dragons-team',
        color: 'danger',
        message:"The pipeline ${currentBuild.displayName} failed."
    }
}
```

Se muestra el pipeline (el cambio de channel por '#notificaciones' será explicado más adelante)

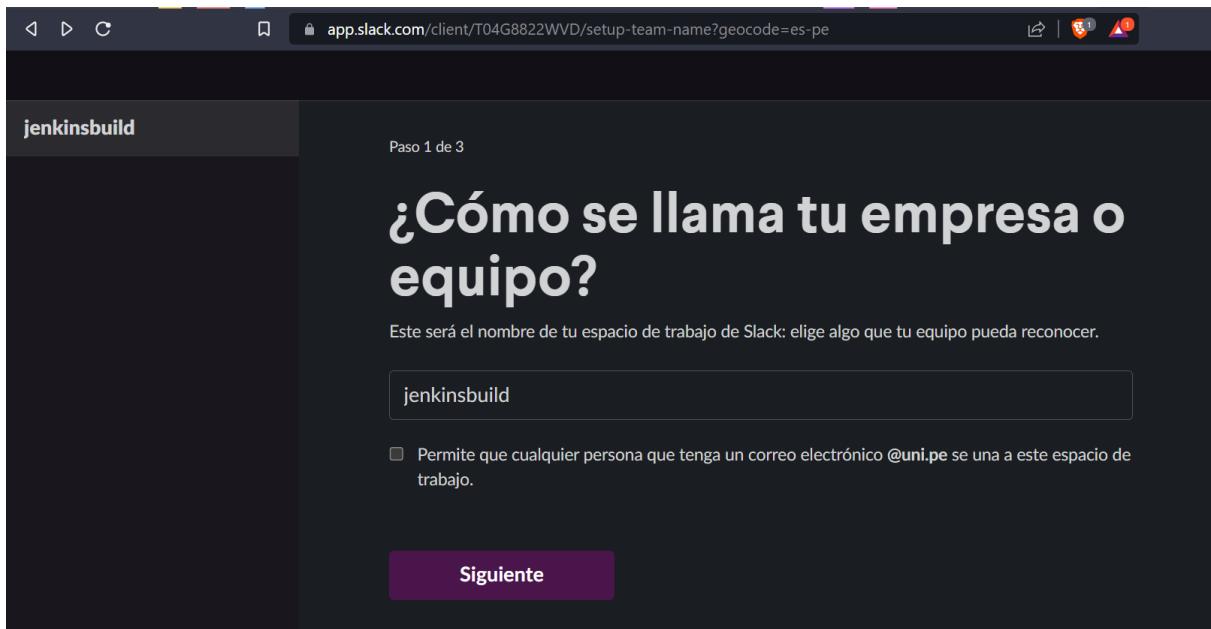
```
50      post {
51          failure {
52              slackSend channel: '#notificaciones',
53              color: 'danger',
54              message:"The pipeline ${currentBuild.displayName} failed."
55          }
56      }
```

Procedemos con la instalación del plug in Slack Notification para poder proseguir con su configuración

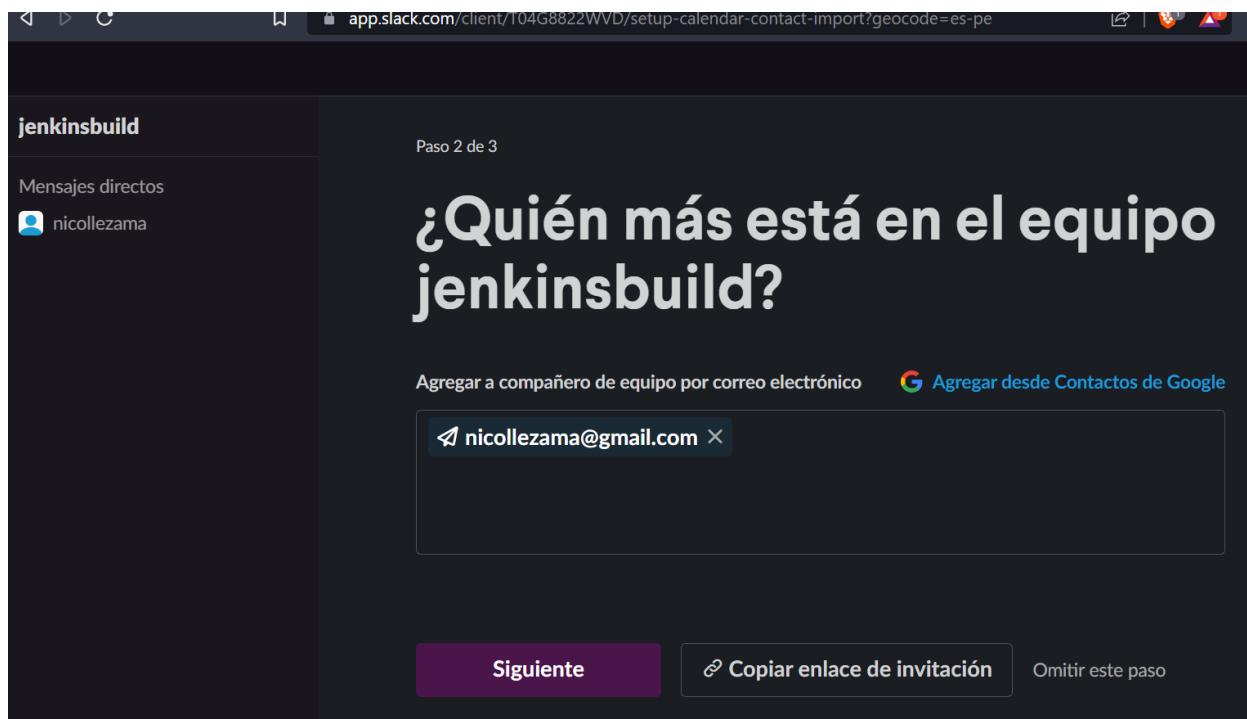
The screenshot shows the Jenkins Plugin Manager interface. On the left, there's a sidebar with links for Updates, Available plugins (which is selected), Installed plugins, and Advanced settings. The main area is titled "Plugins" and has a search bar with the word "slack". Below the search bar, there's a table listing available plugins. The first item in the list is "Slack Notification" version 631.v40deea_40323b, which is checked for installation. The table includes columns for "Name", "Version", and "Released". Other visible items include "Global Slack Notifier" and "Build Notifications". At the bottom of the table are buttons for "Install without restart" and "Download now and install after restart".

Y tenemos que tener una cuenta en slack, para ello accedemos a su página y terminamos completando lo siguiente tras asociarlo a una cuenta:

Paso 1: Designación del espacio de trabajo como jenkinsbuild



Paso 2: Agregué otra cuenta asociada a mi persona como compañero de equipo para hacer la prueba



Paso 3: Asignación al proyecto como Jenkinsprueba

Jenkinsbuild

Paso 3 de 3

Mensajes directos

nicollezama

¿En qué está trabajando tu equipo en este momento?

Puede ser cualquier cosa: un proyecto, una campaña, un evento o el acuerdo que intentas cerrar.

Jenkinsprueba| 67

Siguiente

Ya en el área de trabajo, creamos un nuevo canal dirigido exclusivamente para recibir las notificaciones de jenkins:

jenkinsbuild

Explorar Slack

general

jenkinsprueba

notificaciones

varios

+ Agregar canales

Mensajes directos

DAGMAR NICOLE LEZAMA ...

nicollezama

+ Invitar a tus compañeros de ...

Buscar en jenkinsbuild

Agregar gente

Agregar a todos los miembros de jenkinsbuild

Agregar personas específicas

Solo los administradores pueden ver este ajuste

Agregar automáticamente a cualquier persona que se une a jenkinsbuild

Listo

Ahora en el menú izquierdo, accedemos a explorar Slack y seleccionamos aplicaciones para buscar jenkins CI y agregarlo, así podremos vincularlos tranquilamente

jenkinsbuild

Explorar Slack

general

jenkinsprueba

notificaciones

varios

+ Agregar canales

Mensajes directos

DAGMAR NICOLE LEZAMA ...

nicollezama

+ Invitar a tus compañeros de ...

Buscar en jenkinsbuild

Aplicaciones

Conecta herramientas

Más información sobre las apps

Instalar Google Calendar

Instalar Google Drive

Q jenki

Jenkins CI

Un servidor de integración continua de código abierto.

Agregar

Tras definir el canal donde se publicarán las notificaciones de Jenkins, seleccionamos agregar

The screenshot shows the Jenkins CI integration setup page in the Slack App Directory. At the top, there's a header with the Slack logo and navigation links for 'Explorar', 'Administrar', and 'Compilar'. A dropdown menu shows 'jenkinsbuild'. Below the header, the page title is 'Explorar aplicaciones > Jenkins CI > Nueva configuración'. The main content area features the Jenkins CI icon and a brief description: 'Un servidor de integración continua de código abierto.' It states that Jenkins CI is a customizable server with over 600 add-ons. A note says, 'Esta integración publicará las notificaciones de compilaciones en un canal de Slack.' A section titled 'Publicar en el canal' allows selecting a channel and specifying the number of notifications (dropdown: '# notificaciones'). A link 'o crear un nuevo canal' is also present. A large green button at the bottom right says 'Agregar integración con Jenkins CI'.

Ahora, proseguiremos con los pasos de guía que nos ofrecen a continuación:

The screenshot shows the Jenkins CI integration configuration page in the Slack App Directory. The top navigation and dropdown are identical to the previous screenshot. The main content shows the Jenkins CI app card, which was added by DAGMAR NICOLE LEZAMA VERASTEGUI on December 26, 2022. A 'Desactivar' (Disable) and 'Eliminar' (Delete) button are visible. The description reiterates that Jenkins CI is a customizable server with many add-ons. A note about notifications is present. A large box titled 'Instrucciones de configuración' (Configuration instructions) contains steps for configuration. Step 1: 'En el panel de Jenkins, haz clic en Manage Jenkins (Administrar Jenkins), en la navegación izquierda.' (In the Jenkins panel, click on Manage Jenkins (Administer Jenkins) in the left navigation.) Below this is a small image of a Jenkins interface.

Tras completar todos los pasos adecuadamente, nos dirigimos a Jenkins para configurar todo lo relacionado a Slack. Lo completado en Jenkins es en base a la guía de la página misma de slack en la configuración de Jenkins

Tras terminar con la configuración, lo probaremos con Test Connection, en la pantalla que se muestra a continuación veremos que la prueba ha resultado satisfactoria:

The screenshot shows the Jenkins configuration interface for Slack. The 'Workspace' field contains 'jenkinsbuilddespacio'. The 'Credential' dropdown is set to 'slack notificaciones'. The 'Default channel / member id' field contains '#notificaciones'. There is an unchecked checkbox for 'Custom slack app bot user'. Below the fields are 'Save' and 'Apply' buttons. To the right of the 'Test Connection' button, a purple arrow points to the Slack interface.

Aquí se puede evidenciar el mensaje de la prueba de conexión:

The screenshot shows a Slack conversation in the '#notificaciones' channel. A message from 'jenkins' at 12:42 says: 'Slack/Jenkins plugin: you're all set on http://localhost:8080/'. This message is highlighted with a purple arrow pointing from the Jenkins configuration screen above.

Ahora hacemos push del pipeline agregado en la parte inicial de esta sección y podremos ver que la construcción es generada automáticamente.

En esta construcción se presenta un error, debido a que, como indica hay una excepción: 'Multiple occurrences of the post section', esto es causado a que agregué ese post section y no eliminé el anterior post section, entonces habían varias ocurrencias en una misma instrucción, tras reparar esto, la construcción será exitosa, pero el pipeline no pasará por el post section, ya que solo pasará por ese stage cuando detecte una falla.

```

Dashboard > calculador > #36
Git Build Data
Replay
Pipeline Steps
Workspaces
Previous Build
Next Build

Fetching changes from the remote Git repository
> 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 ae06237b89c2d4279ad8afb742ace668bfc84f5e (refs/remotes/origin/main)
> git config core.sparsecheckout # timeout=10
> git checkout -f ae06237b89c2d4279ad8afb742ace668bfc84f5e # timeout=10
Commit message: "Add slack notification"
> git rev-list --no-walk ae06237b89c2d4279ad8afb742ace668bfc84f5e # timeout=10
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 50: Multiple occurrences of the post section @ line 50, column 6.
    post {
    ^
1 error

at org.codehaus.groovy.control.ErrorCollector.failIfErrors(ErrorCollector.java:309)
at org.codehaus.groovy.control.CompilationUnit.applyToPrimaryClassNodes(CompilationUnit.java:1107)
at org.codehaus.groovy.control.CompilationUnit.doPhaseOperation(CompilationUnit.java:624)
at org.codehaus.groovy.control.CompilationUnit.processPhaseOperations(CompilationUnit.java:602)
at org.codehaus.groovy.control.CompilationUnit.compile(CompilationUnit.java:579)
at groovy.lang.GroovyClassLoader.doParseClass(GroovyClassLoader.java:232)

```

Para poder probar el mensaje de Slack podríamos cambiar el pipeline de la siguiente manera:

```

42     post {
43         success {
44             slackSend channel: '#notificaciones',
45             color: 'success',
46             message:"The pipeline ${currentBuild.displayName} ha resultado exitosa."
47         }
48     }

```

De esta manera, cada que haya una construcción exitosa, nos notificará en slack. Ahora al hacer push, Jenkins detecta el commit y automáticamente se realiza la construcción. En esta oportunidad, logramos presenciar que el pipeline ha resultado exitoso y se ha logrado completar cada paso satisfactoriamente

Status

Pipeline calculator

Stage View

	Declarative: Checkout SCM	Compile	Unit test	Code coverage	Static code analysis	Declarative: Post Actions
Average stage times: (Average full run time: ~50s)	2s	21s	5s	7s	5s	2s
#39 Dec 26 18:29 1 commit	1s	16s	3s	7s	3s	1s
#38 Dec 26 18:25 1 commit	1s	18s	6s	6s	3s	1s

Ahora revisaremos el Slack y podremos observar la última notificación cuyo mensaje describe que el pipeline #39 (el último construido) ha sido exitosa.

The screenshot shows a Slack interface with the following details:

- Channel:** # notificaciones
- Messages:**
 - DAGMAR NICOLE LEZAMA VERASTEGUI 12:17: se unió a #notificaciones.
 - jenkins Aplicación 12:42: Slack/Jenkins plugin: you're all set on <http://localhost:8080/>
 - jenkins Aplicación 13:03: Slack/Jenkins plugin: you're all set on <http://localhost:8080/>
 - jenkins Aplicación 13:25: The pipeline calculador #38 failed.
 - The pipeline calculador #39 ha resultado exitosa.

Nota: El mensaje “the pipeline calculador #38 failed”, es causado a que no cambié el mensaje del post al resultar un success inicialmente, por eso existe la construcción #39, para mostrar un mensaje más congruente con lo que sucede, de todas maneras, de ambas construcciones resaltamos que lo importante es que hemos probado que las notificaciones funcionan correctamente.

Espacios de equipo

Junto con la cultura ágil surgió la idea de que es mejor que todo suceda en un espacio de equipo. En lugar de escribir correos electrónicos, reúnanse, en lugar de mensajes en línea, ven y habla, en lugar de herramientas de seguimiento de tareas, ten una pizarra. La misma idea vino a la entrega continua y a Jenkins.

Dado que los desarrolladores son seres creativos, inventaron muchas otras ideas que juegan el mismo papel. Algunos equipos cuelgan altavoces grandes que emiten un pitido cuando falla el pipeline. Otros tienen juguetes que parpadean cuando se termina la construcción. Uno muy conocido es Pipeline State UFO, que se proporciona como un proyecto de código abierto en GitHub.

Estrategias de desarrollo del equipo

Hemos cubierto todo lo relacionado con cómo debería verse el pipeline de integración continua. Sin embargo, ¿cuándo exactamente debería ejecutarse? Por supuesto, se activa después del commit en el repositorio, pero ¿después del commit en qué rama? ¿Solo en el master o en cada rama? ¿O tal vez debería ejecutarse antes, no después del commit para que el repositorio siempre esté en buen estado? O, ¿de no tener ramas?

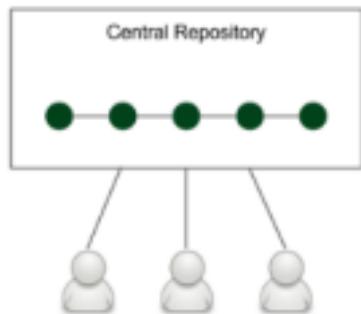
No hay una sola mejor respuesta a estas preguntas. En realidad, la forma en que utiliza el proceso de integración continua depende del flujo de trabajo de desarrollo de tu equipo.

Flujos de trabajo de desarrollo

Un flujo de trabajo de desarrollo es la forma en que tu equipo coloca el código en el repositorio. Depende, por supuesto, de muchos factores, como la herramienta SCM, las especificaciones del proyecto y el tamaño del equipo. Como resultado, cada equipo desarrolla el código de una manera ligeramente diferente. Sin embargo, podemos clasificarlos en tres tipos: trunk-based workflow, branching workflow y forking workflow.

Trunk-based workflow

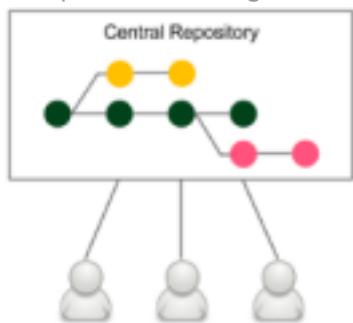
Trunk-based workflow es la estrategia más simple posible. Se presenta en el siguiente diagrama:



Hay un repositorio central con una sola entrada para todos los cambios en el proyecto, que se denomina trunk o master. Cada miembro del equipo clona el repositorio central para tener sus propias copias locales. Los cambios se envían directamente al repositorio central.

The branching workflow

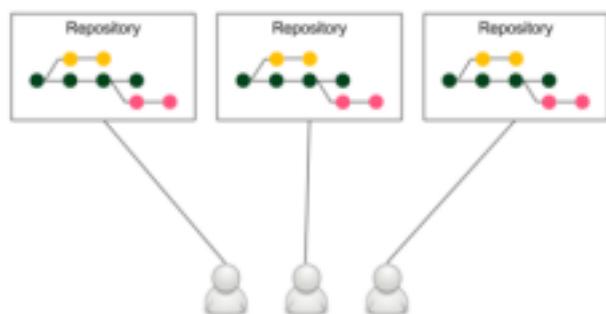
Branching workflow, como sugiere su nombre, significa que el código se guarda en muchas ramas diferentes. La idea se presenta en el siguiente diagrama:



Cuando los desarrolladores comienzan a trabajar en una nueva característica, crean una rama dedicada desde el troncal y realizan allí todos los cambios relacionados con la característica. Esto facilita que varios desarrolladores trabajen en una característica sin romper la base del código principal.

Forking workflow

Forking workflow es muy popular entre las comunidades de código abierto. Se presenta en el siguiente diagrama:



Cada desarrollador tiene su propio repositorio del lado del servidor. Puede que sea o no el repositorio oficial, pero técnicamente, cada repositorio es exactamente igual.

Forking significa literalmente crear un nuevo repositorio a partir de otro repositorio. Los desarrolladores

envían a sus propios repositorios, y cuando quieren integrar código, crean un pull request para el otro repositorio.

La principal ventaja del forking workflow es que la integración no se realiza necesariamente a través de un repositorio central. También ayuda con la propiedad porque permite la aceptación de pull request de otros sin darles acceso de escritura.

Adoptar la integración continua

Hemos descrito diferentes flujos de trabajo de desarrollo, pero ¿cómo influyen en la configuración de integración continua?

Estrategias de ramificación

Cada flujo de trabajo de desarrollo implica un enfoque de integración continua diferente:

- Trunk-based workflow: Esto implica luchar constantemente contra un pipeline roto. Si todo el mundo realiza commits con la base de código principal, el pipeline suele fallar. En este caso, la antigua regla de integración continua dice: si la construcción no funciona, el equipo de desarrollo deja de hacer lo que esté haciendo y soluciona el problema de inmediato.
- Branching workflow: esto resuelve el problema broken trunk pero introduce otro: si cada uno se desarrolla en sus propias ramas, ¿dónde está la integración? Una característica suele tardar semanas o meses en desarrollarse y, durante todo este tiempo, la rama no está integrada en el código principal. Por lo tanto, en realidad no puede llamarse integración continua, sin mencionar que existe una necesidad constante de fusionarse y resolver conflictos.
- Forking workflow: Esto implica administrar el proceso de integración continua por parte de cada propietario del repositorio, lo que no suele ser un problema. Sin embargo, comparten los mismos problemas que el branching workflow.

Alternancias de características

La alternancia de características es una técnica que es una alternativa al mantenimiento de múltiples ramas de código fuente, de modo que la función se puede probar antes de que se complete y esté lista para su lanzamiento. Se utiliza para deshabilitar la característica para los usuarios, pero habilitarla para los desarrolladores durante la prueba. La alternancia de características son esencialmente variables que se utilizan en declaraciones condicionales.

La implementación más simple de alternancia de características son las banderas y las declaraciones if. Un desarrollo que utiliza alternancias de características, en lugar de un desarrollo de ramas de características, aparece de la siguiente manera:

1. Se debe implementar una nueva característica.
2. Crea una nueva bandera o una propiedad de configuración: feature_toggle (en lugar de la rama feature).
3. Todo el código relacionado con la característica se agrega dentro de la instrucción if (en lugar de un commit con la rama feature, como el siguiente:

```
if (feature_toggle) {
    // algo...
}
```
4. Durante el desarrollo de características, ocurre lo siguiente:
 - La codificación se realiza en el master con feature_toggle = true
 - La liberación se realiza desde el master con feature_toggle = false.
5. Cuando se completa el desarrollo de la característica, se eliminan todas las declaraciones if y se elimina feature_toggle de la configuración.

El beneficio de alternar características es que todo el desarrollo se realiza en el troncal, lo que facilita una integración continua real y mitiga los problemas con la fusión del código.

Multi-Ramas en Jenkins

Si decides utilizar ramas de cualquier forma, ya sea las ramas de características largas o las ramas de corta duración recomendadas, es conveniente saber que el código está en buen estado antes de fusionarlo con el master.

Para usar multi-ramas en el proyecto calculador, procedamos con los siguientes pasos:

1. Abre la página principal de Jenkins.
2. Haga clic en New Item.
3. Ingresa calculador-ramas como el nombre del elemento, selecciona Multibranch Pipeline, y haga clic en OK.
4. En la sección Branch Sources, haga clic en Add source y seleccione Git.
5. Introduce la dirección del repositorio en el campo Project Repository.
6. Marca Periodically if not otherwise run y establece 1 minuto como intervalo.
7. Haga clic en Save.

Cada minuto, esta configuración verifica si se agregaron (o eliminaron) ramas y crea (o elimina) el pipeline dedicado definido por Jenkinsfile.

Podemos crear una nueva rama y ver cómo funciona. Vamos a crear una nueva rama llamada caracteristica y enviarla al repositorio:

```
$ git checkout -b caracteristica  
$ git push origin caracteristica
```

Después de un momento, debería ver un nuevo pipeline de rama creado y ejecutado automáticamente.

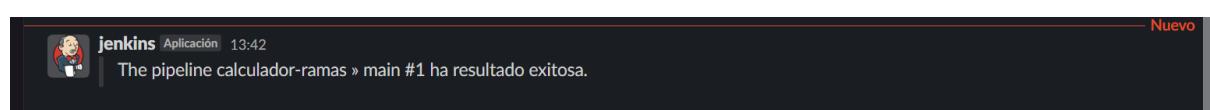
Comprueba esto.

Crearemos el calculador-ramas como Multibranch Pipeline, tal como indican las instrucciones y al hacerlo veremos lo siguiente:

The screenshot shows the Jenkins interface for a 'calculador-ramas' job. On the left, there's a sidebar with various Jenkins management links like Status, Configure, Scan Multibranch Pipeline Now, Scan Multibranch Pipeline Log (which is selected), View as plain text, Multibranch Pipeline Events, Delete Multibranch Pipeline, People, Build History, Rename, Pipeline Syntax, and Credentials. The main content area is titled 'Scan Multibranch Pipeline Log'. It displays the command-line output of the indexing process:

```
Started [Mon Dec 26 18:40:59 UTC 2022] Starting branch indexing...  
> git --version # timeout=10  
> git --version # 'git version 2.30.2'  
> git ls-remote --symref .. https://github.com/DagmarLV/calcuadorActividad21 # timeout=10  
> git rev-parse --resolve-git-dir /var/jenkins_home/.caches/git-8e61ea7f3a98d3fa625e358c9e946022/.git # timeout=10  
Setting origin to https://github.com/DagmarLV/calcuadorActividad21  
> git config remote.origin.url https://github.com/DagmarLV/calcuadorActividad21 # timeout=10  
Fetching & pruning origin...  
Listing remote references...  
> git config --get remote.origin.url # timeout=10  
> git -version # timeout=10  
> git --version # 'git version 2.30.2'  
> git ls-remote -h .. https://github.com/DagmarLV/calcuadorActividad21 # timeout=10  
Fetching upstream changes from origin  
> git config --get remote.origin.url # timeout=10  
> git fetch --tags --force --progress --prune --origin +refs/heads/*:refs/remotes/origin/* # timeout=10  
Checking branches...  
Checking branch main  
'Jenkinsfile' found  
Met criteria  
Scheduled build for branch: main  
Processed 1 branches  
[Mon Dec 26 18:41:03 UTC 2022] Finished branch indexing. Indexing took 4.3 sec  
Finished: SUCCESS
```

El Slack nos mostrará la siguiente notificación. Podremos afirmar que el pipeline fue satisfactorio.

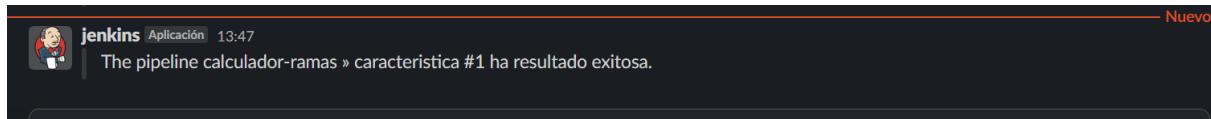


Probemos qué sucede al crear una nueva rama llamada caracteristica y la enviamos al repositorio

```
Nicole@LAPTOP-M3SQF64H MINGW64 ~/Documents/GitHub/calculadorActividad21 (main)
$git checkout -b caracteristica
Switched to a new branch 'caracteristica'

Nicole@LAPTOP-M3SQF64H MINGW64 ~/Documents/GitHub/calculadorActividad21 (caracteristica)
$ git push origin caracteristica
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'caracteristica' on GitHub by visiting:
remote:     https://github.com/DagmarLV/calculadorActividad21/pull/new/caracteristica
remote:
To https://github.com/DagmarLV/calculadorActividad21.git
 * [new branch]      caracteristica -> caracteristica
```

El Slack nos muestra el mensaje informándonos que el pipeline resultó exitoso



Si seleccionamos Status, podremos ver las ramas de nuestro proyecto, como se muestra en la siguiente imagen

A screenshot of the Jenkins dashboard for the 'calculador-ramas' project. The left sidebar shows options like Status, Configure, Scan Multibranch Pipeline Now, Scan Multibranch Pipeline Log, Multibranch Pipeline Events, Delete Multibranch Pipeline, People, Build History, Rename, Pipeline Syntax, and Credentials. The main area shows the 'Status' tab selected. A table lists 'Branches (2)'. The table has columns: S, W, Name, Last Success, Last Failure, and Last Duration. There are two rows: one for 'caracteristica' (green checkmark, 1 min 5 sec ago, N/A, 51 sec) and one for 'main' (green checkmark, 6 min 15 sec ago, N/A, 54 sec). At the bottom, there are links for Atom feed for all, Atom feed for failures, and Atom feed for just latest builds.

Un enfoque muy similar es construir un pipeline por pull request en lugar de un pipeline por rama, lo que da el mismo resultado: la base de código principal siempre está en buen estado.

Requisitos no técnicos

Por último, pero no menos importante, la integración continua no se trata solo de tecnología. Por el contrario, la tecnología viene en segundo lugar.

La idea es un poco simplificada y las herramientas automatizadas son útiles; sin embargo, el mensaje principal es que sin el compromiso de cada miembro del equipo, incluso las mejores herramientas no ayudarán.