

Curso de desarrollo de software

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:

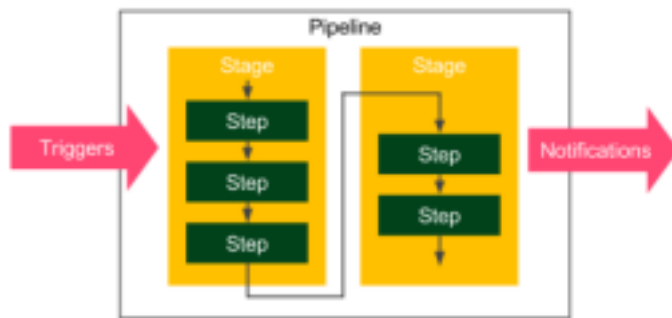
- Agrupación de operaciones: las operaciones se agrupan en etapas (también conocidas como puertas o puertas de calidad¹) que introducen una estructura en un proceso y definen claramente una regla: si una etapa falla, no se ejecutan más etapas.
- Visibilidad: se visualizan todos los aspectos de un proceso, 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:

¹ quality gates



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, ampliamos 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'
      }
    }
  }
}
```

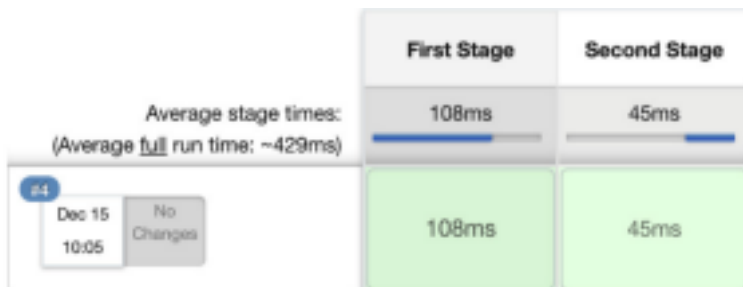
Usando Jenkins

Pipeline script

Script ?

```
1 pipeline{
2   agent any
3   stages{
4     stage('First Stage'){
5       steps{
6         echo 'Step1.hola mundo!'
7       }
8     }
9     stage('Second Stage'){
10      steps{
11        echo 'Step2.Segunda vez Hola'
12        echo 'Step3.Tercera vez Hola'
13      }
14    }
15  }
16 }
```

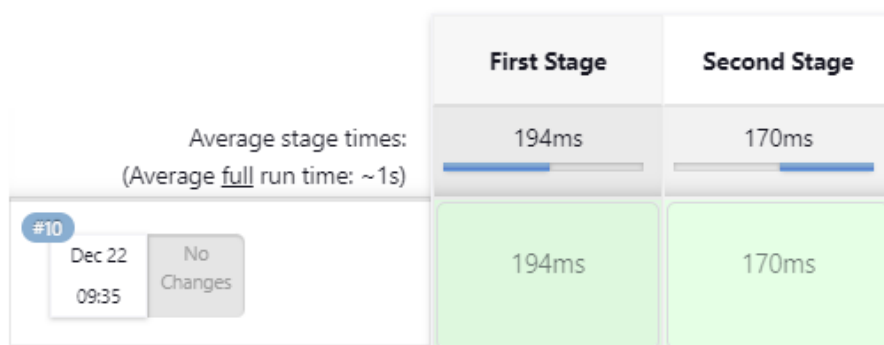
El pipeline no tiene requisitos especiales en términos de entorno y ejecuta tres pasos dentro de dos etapas. Cuando hacemos clic en Build Now, deberíamos ver una representación visual.



Comprueba estos resultados.

Comprobando los resultados

Stage View



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' } }
}
```

Usando Jenkins

Script ?

```
1 pipeline {
2   agent any
3   triggers { cron('* * * * *') }
4   options { timeout(time: 5) }
5   parameters {
6     booleanParam(name: 'DEBUG_BUILD', defaultValue: true,
7       description: 'construimos la depuracion?')
8   }
9   stages {
10    stage('Example') {
11      environment { NAME = 'Checha' }
12      when { expression { return params.DEBUG_BUILD } }
13    }
14    steps {
15      echo "Hola $NAME"
16      script {
17        def browsers = ['chrome', 'firefox']
18        for (int i = 0; i < browsers.size(); ++i) {
19          echo "Prueba el ${browsers[i]} browser."
20        }
21      }
22    }
23  }
24 }
25 post {
26   always {
27     echo 'Hola muchas gracias'
28   }
29 }
30 }
31 }
32 }
```

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.

Salida Por Consola

```

Started by timer
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /home/breiner/.jenkins/workspace/Hello World
[Pipeline] {
[Pipeline] timeout
Timeout set to expire in 5 min 0 sec
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Example)
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
Hola Checha
[Pipeline] script
[Pipeline] {
[Pipeline] echo
Prueba el chrome browser.
[Pipeline] echo

```

```

Prueba el firefox browser.
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Hola muchas gracias
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // timeout
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

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.
- Unite 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.

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'
      }
    }
  }
}
```

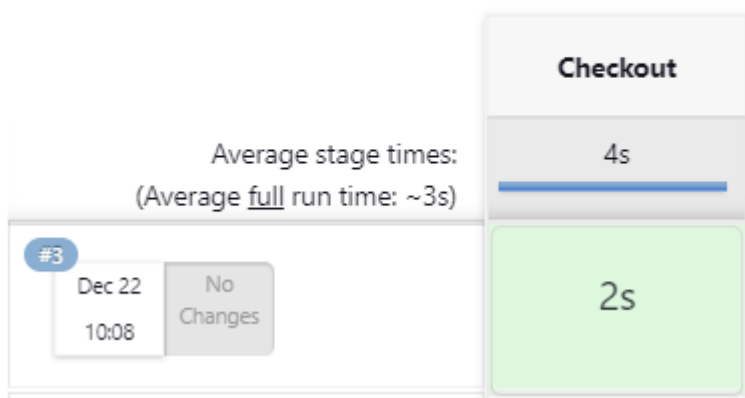

Script ?

```
1 pipeline{
2   agent any
3   stages{
4     stage("Checkout"){
5       steps{
6         git url:'https://github.com/BreinerCatalinoMorales/CC3S2.git',branch: 'master'
7       }
8     }
9   }
10 }
```

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.

Como se observa, se ejecuto con exito

Stage View



Ahora vemos la salida por consola

```

Started by user Breiner Catalino Morales
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /home/breiner/.jenkins/workspace/Calculador
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Checkout)
[Pipeline] git
The recommended git tool is: NONE
No credentials specified
> git rev-parse --resolve-git-dir /home/breiner/.jenkins/workspace/Calculador/.git # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/BreinerCatalinoMorales/CC3S2.git # timeout=10
Fetching upstream changes from https://github.com/BreinerCatalinoMorales/CC3S2.git
> git --version # timeout=10
> git --version # 'git version 2.34.1'
> git fetch --tags --force --progress -- https://github.com/BreinerCatalinoMorales/CC3S2.git +refs/heads/*:refs/remotes/origin/*
# timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision 541d2beb4a1e12e4f3014c437438b24ad54a41b3 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 541d2beb4a1e12e4f3014c437438b24ad54a41b3 # timeout=10
> git branch -a -v --no-abbrev # timeout=10
> git checkout -b master 541d2beb4a1e12e4f3014c437438b24ad54a41b3 # timeout=10

> git checkout -b master 541d2beb4a1e12e4f3014c437438b24ad54a41b3 # timeout=10
Commit message: "Update README.md"
First time build. Skipping changelog.
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

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.

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

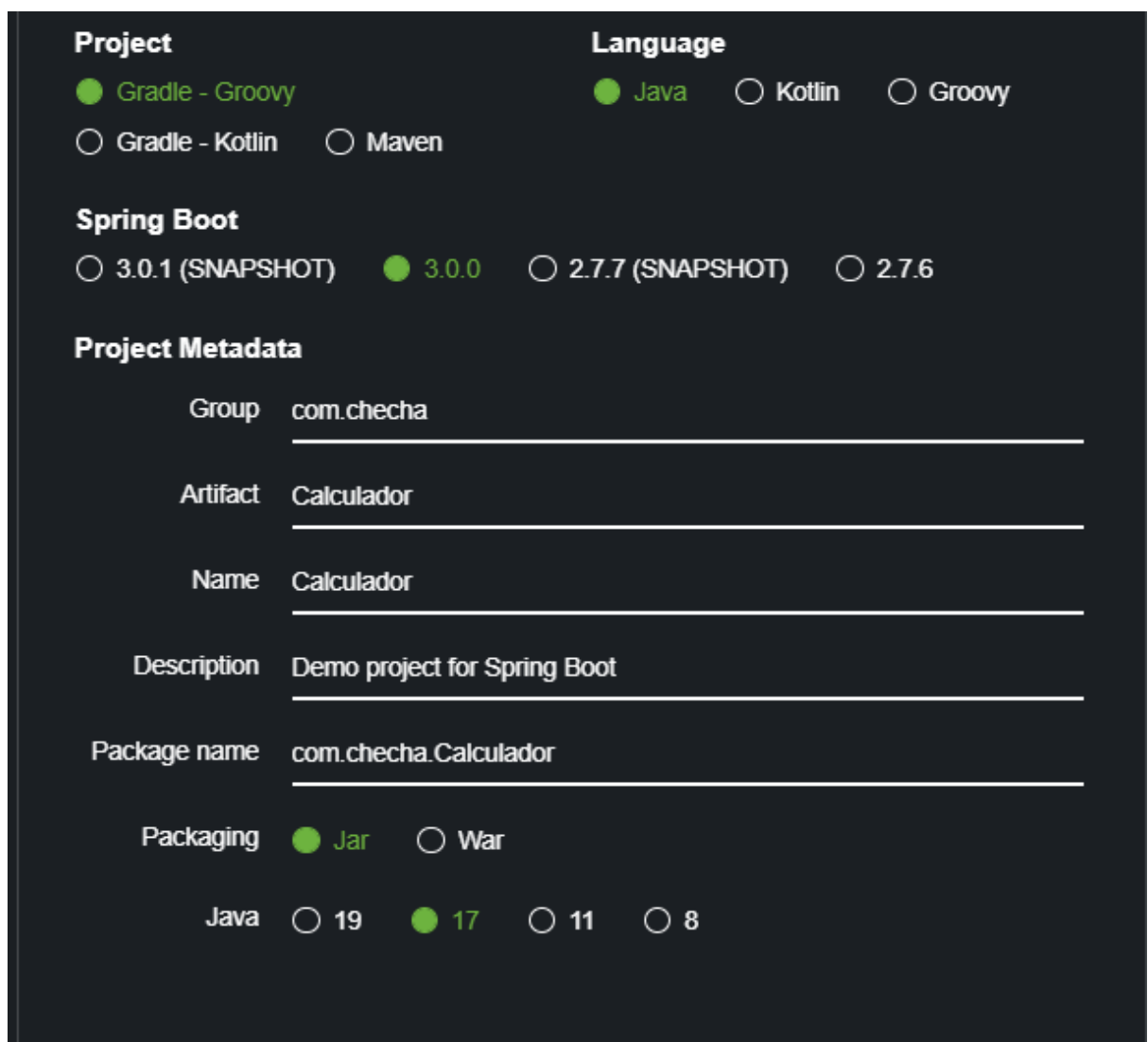
Vamos a crear un proyecto Java muy simple utilizando el marco Spring Boot creado por Gradle.

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.



The screenshot shows the Spring Boot project generator interface. It is configured for a Gradle project in Java. The Spring Boot version is set to 3.0.0. The project metadata includes a group of 'com.checha', an artifact of 'Calculador', and a package name of 'com.checha.Calculador'. The packaging is set to 'Jar' and the Java version is '17'.

Project		Language	
<input checked="" type="radio"/> Gradle - Groovy	<input type="radio"/> Gradle - Kotlin	<input checked="" type="radio"/> Java	<input type="radio"/> Kotlin
<input type="radio"/> Maven		<input type="radio"/> Groovy	

Spring Boot			
<input type="radio"/> 3.0.1 (SNAPSHOT)	<input checked="" type="radio"/> 3.0.0	<input type="radio"/> 2.7.7 (SNAPSHOT)	<input type="radio"/> 2.7.6

Project Metadata	
Group	com.checha
Artifact	Calculador
Name	Calculador
Description	Demo project for Spring Boot
Package name	com.checha.Calculador

Packaging	Java
<input checked="" type="radio"/> Jar	<input type="radio"/> 19
<input type="radio"/> War	<input checked="" type="radio"/> 17
	<input type="radio"/> 11
	<input type="radio"/> 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Reactive Web WEB
Build reactive web applications with Spring WebFlux and Netty.

Spring for GraphQL WEB
Build GraphQL applications with Spring for GraphQL and GraphQL Java.

Rest Repositories WEB
Exposing Spring Data repositories over REST via Spring Data REST.

Spring Session WEB
Provides an API and implementations for managing user session information.

Rest Repositories HAL Explorer WEB
Browsing Spring Data REST repositories in your browser.

Spring HATEOAS WEB
Eases the creation of RESTful APIs that follow the HATEOAS principle when working with Spring / Spring MVC.

Spring Web Services WEB
Facilitates contract-first SOAP development. Allows for the creation of flexible web services using one of the many ways to manipulate XML payloads.

Jersey WEB
Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.

Se genero el .zip llamado Calcudaor

 Calculador22/12/2022 10:16Carpeta comprimi...65 KB

Después de crear el proyecto, podemos hacer push en el repositorio de GitHub.

Pushing el código a GitHub

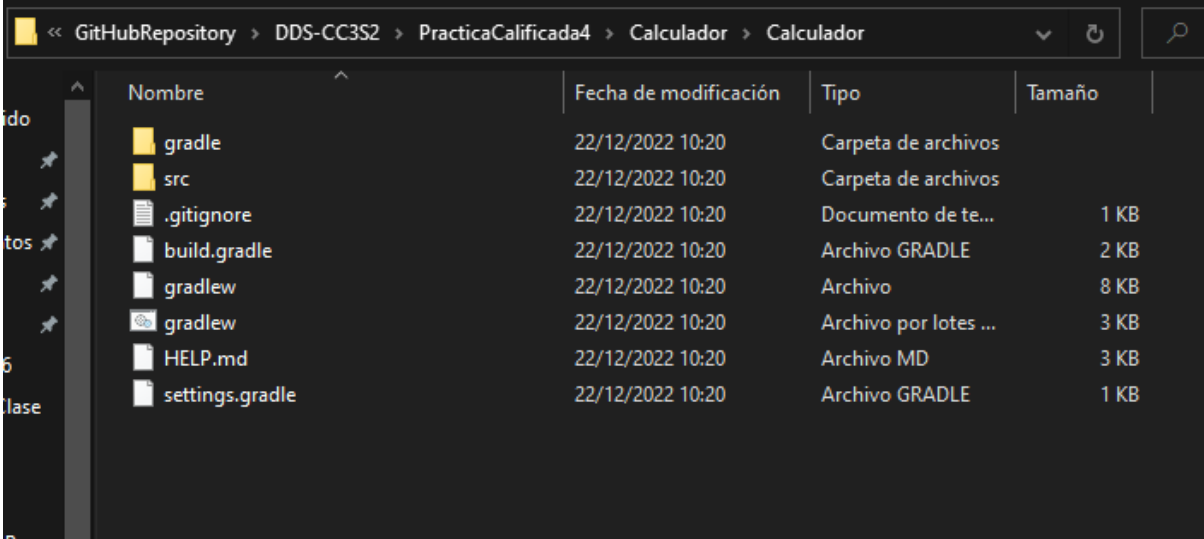
Usaremos la herramienta Git para realizar las operaciones de commit y push.

Primero clonamos el repositorio en el sistema de archivos:

```
$ git clone https://github.com/<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.



Nombre	Fecha de modificación	Tipo	Tamaño
gradle	22/12/2022 10:20	Carpeta de archivos	
src	22/12/2022 10:20	Carpeta de archivos	
.gitignore	22/12/2022 10:20	Documento de te...	1 KB
build.gradle	22/12/2022 10:20	Archivo GRADLE	2 KB
gradlew	22/12/2022 10:20	Archivo	8 KB
gradlew	22/12/2022 10:20	Archivo por lotes ...	3 KB
HELP.md	22/12/2022 10:20	Archivo MD	3 KB
settings.gradle	22/12/2022 10:20	Archivo GRADLE	1 KB

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
```

```
brein@DESKTOP-AMRA9N7 MINGW64 ~/desktop/GitHubRepository/DDS-CC3S2/PracticaCalif  
icada4/Calculador (master)  
$ ls  
HELP.md build.gradle gradle/ gradlew* gradlew.bat 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`.

```
brein@DESKTOP-AMRA9N7 MINGW64 ~/desktop/GitHubRepository/DDS-CC3S2/PracticaCalificadora4/Calculador (master)
$ ./gradlew compileJava
Downloading https://services.gradle.org/distributions/gradle-7.6-bin.zip
.....10%.....20%.....30%.....40%.....50%.....
.....60%.....70%.....80%.....90%.....100%

Welcome to Gradle 7.6!

Here are the highlights of this release:
- Added support for Java 19.
- Introduced '--rerun' flag for individual task rerun.
- Improved dependency block for test suites to be strongly typed.
- Added a pluggable system for Java toolchains provisioning.

For more details see https://docs.gradle.org/7.6/release-notes.html

Starting a Gradle Daemon (subsequent builds will be faster)

BUILD SUCCESSFUL in 1m 56s
1 actionable task: 1 executed
```

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.

```
brein@DESKTOP-AMRA9N7 MINGW64 ~/desktop/GitHubRepository (master)
$ git add .
warning: in the working copy of 'tatus', LF will be replaced by CRLF the next time Git touches it

brein@DESKTOP-AMRA9N7 MINGW64 ~/desktop/GitHubRepository (master)
$ git commit -m "Agrega plantilla Spring Boot"
[master c7c87f9] Agrega plantilla Spring Boot
1 file changed, 157 insertions(+)
create mode 100644 tatus

brein@DESKTOP-AMRA9N7 MINGW64 ~/desktop/GitHubRepository (master)
$ git push -u origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 1.63 KiB | 1.63 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/BreinerCatalinoMorales/CC3S2.git
33eebb3..c7c87f9 master -> master
branch 'master' set up to track 'origin/master'.

brein@DESKTOP-AMRA9N7 MINGW64 ~/desktop/GitHubRepository (master)
$ |
```