

Práctica guiada

Desarrollo ágil de una aplicación web con
integración continua

Contenido

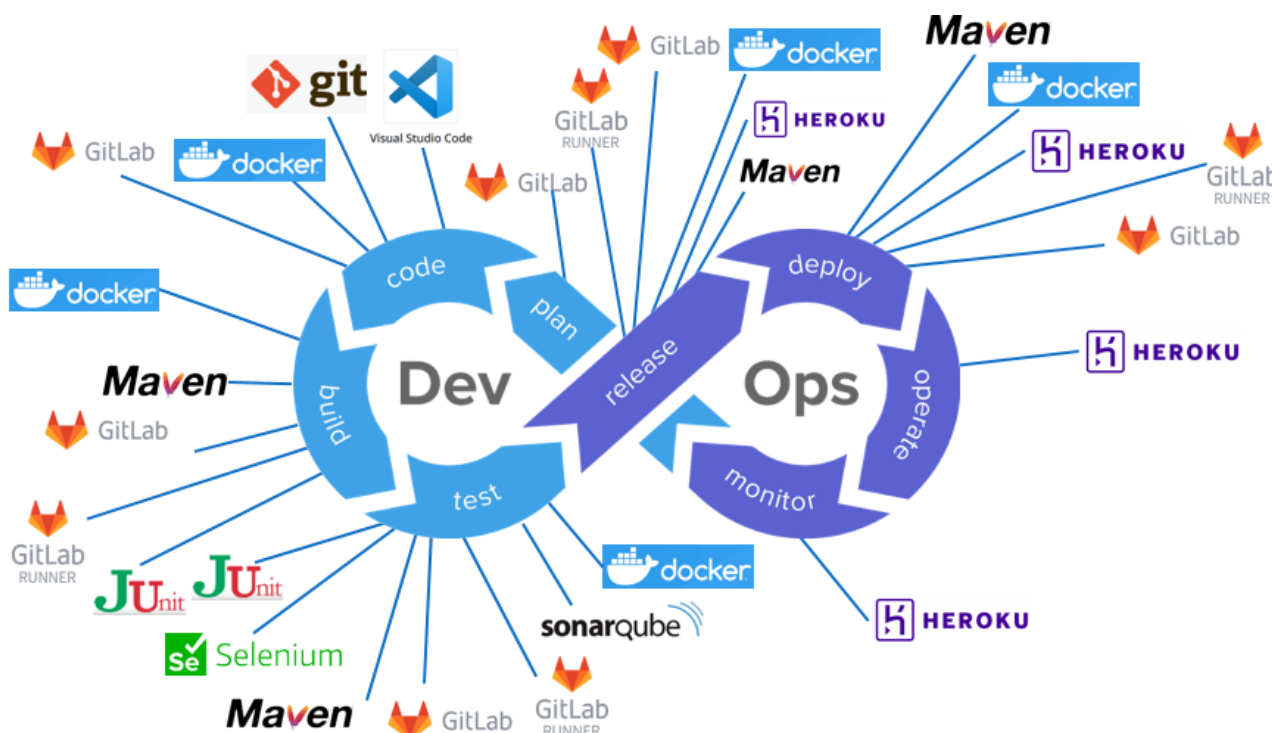
Objetivo de la práctica	4
1. Instalar la infraestructura necesaria	5
1.1 Crear una cuenta gratuita en gitlab.com	6
1.2 Crear una cuenta gratuita en heroku.com.....	7
1.3 Instalar docker.....	8
1.4 Instalar git.....	8
1.5 Instalar visual studio code	9
1.6 Instalar el contenedor ubutu-ci	9
1.7 Instalar el contenedor sonarqube	12
1.8 Entorno para pruebas locales en el ordenador de cada desarrollador	13
2. Crear repositorio local y proyecto en gitlab.com.....	14
2.1 Crear repositorio git local.....	14
2.2 Crear repositorio remoto y proyecto en GitLab	15
2.3 Activar gitlab-runner en el contenedor ubuntu-ci.....	17
3. Diseñar el flujo de trabajos para la integración continua (pipeline).....	21
3.1 Fase (stage) build: Trabajo (job) empaquetar	24
3.1.1 Compilar con un error forzado.....	25
3.1.2 Empaquetar sin errores.....	26
3.2 Fase build: Trabajo pruebas-unitarias	27
3.2.1 Realizar una prueba fallida	28
3.2.2 Realizar una prueba sin errores	30
3.3 Fase test: Trabajo pruebas-funcionales.....	31
3.3.1 Realizar una prueba fallida	32
3.3.2 Realizar una prueba sin errores	34
3.4 Fase qa: Trabajo calidad-codigo	36
3.4.1 Aumentar la exigencia de calidad	38
3.5 Fase deploy: Trabajo despliegue-host-heroku.....	42
4. Realizar una segunda versión de la aplicación	46
4.1 Planificación ágil: crear historias de usuario (issues).....	46
4.2 Crear una rama con git para cada issue (feature).....	51
4.2.1 Programar issue “Añadir estilo a la página principal”	52

4.2.2 Programar issue “Añadir estilo a la página de resultado de votación” y resolver conflicto	
.....	59

Objetivo de la práctica

Con esta práctica vamos a simular la realización de un proyecto de desarrollo de una aplicación web Java aplicando los principios de la Integración Continua y el Desarrollo Ágil.

Se utilizarán las siguientes herramientas: Visual Studio Code, Maven, Git, GitLab, GitLab Runner, Docker, JUnit, Selenium, Sonarqube y Heroku, habituales en el contexto DevOps, en el que se enmarca este trabajo.



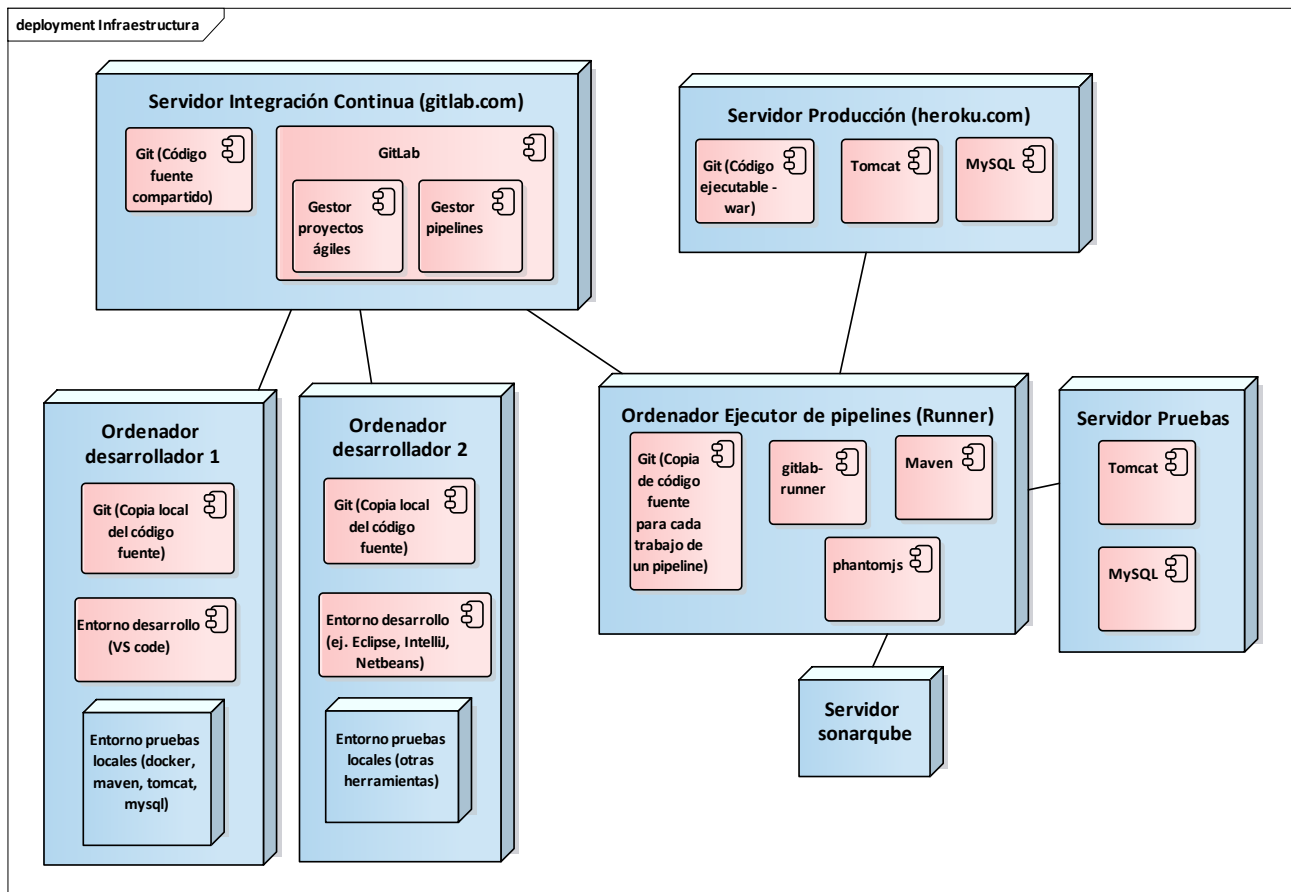
También se utilizará un servidor Tomcat para instalar la aplicación y un servidor MySQL con la base de datos utilizada por la aplicación, que estarán pre-instalados en un contenedor Tomcat.

Se seguirán los siguientes pasos, cada uno de los cuales se explica en el apartado del documento, que tiene el mismo nombre:

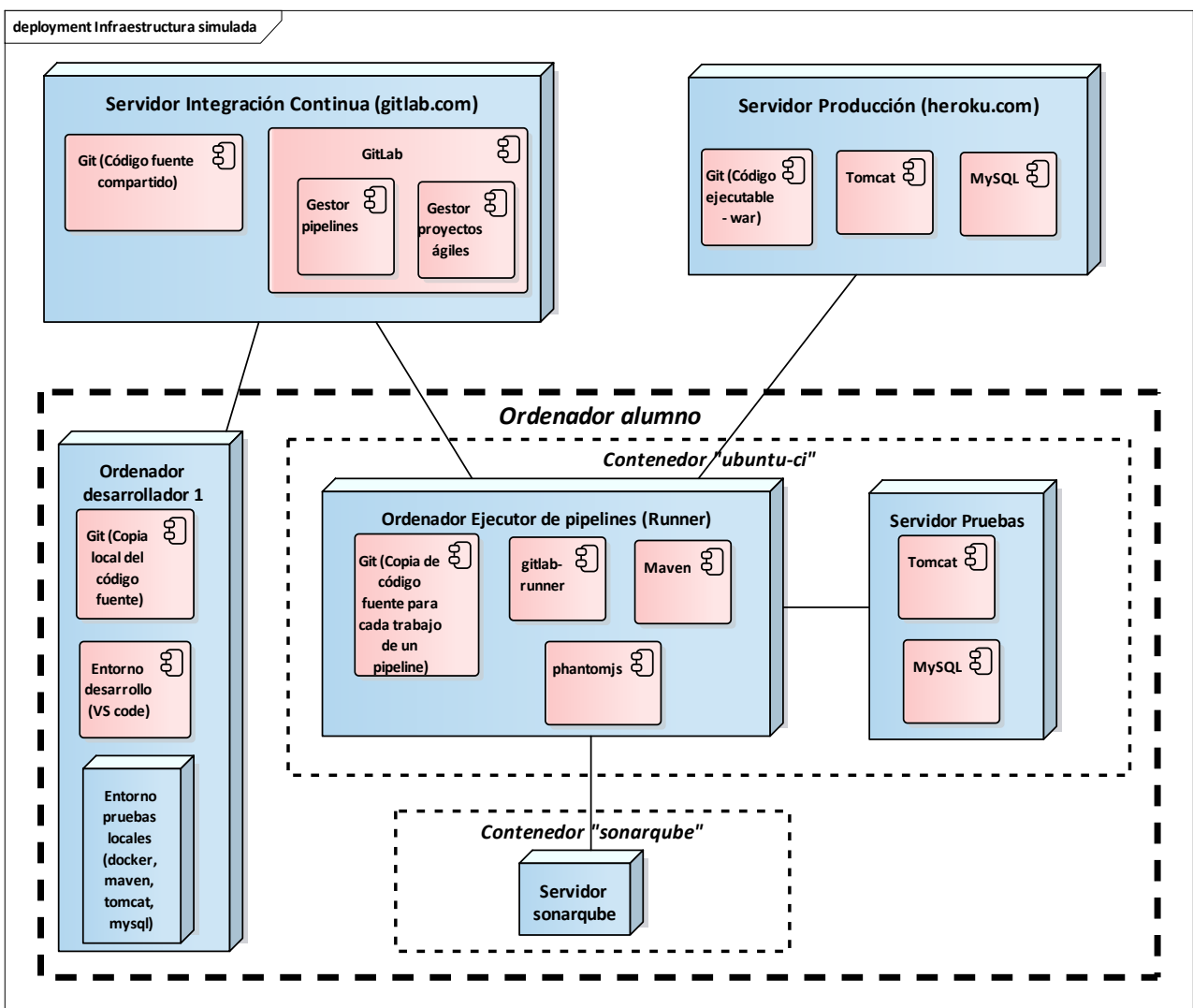
- 1) Instalar la infraestructura necesaria
- 2) Crear repositorio local y proyecto en gitlab.com
- 3) Diseñar el flujo de trabajos para la integración continua (pipelines)
- 4) Realizar una segunda versión de la aplicación

1. Instalar la infraestructura necesaria

En un entorno real, seguramente se utilizaría una infraestructura como la de la siguiente imagen. Es un diagrama de despliegue de UML, los cubos representan máquinas (reales o virtuales) y los otros elementos son componentes software.



Sin embargo, en esta práctica, suponemos que sólo contamos con el ordenador del alumno y acceso a internet. Por lo que simularemos las diferentes máquinas mediante contenedores, resultando la siguiente infraestructura.



Utilizaremos dos servidores externos: gitlab.com y heroku.com. El resto estará instalado en nuestro ordenador.

En los siguientes subapartados se explica cómo instalar y la razón de usar cada elemento.

1.1 Crear una cuenta gratuita en gitlab.com

En esta práctica se utilizará gitlab.com, porque es una herramienta que ofrece tres funcionalidades en una única máquina:

- Servidor de integración continua
- Repositorio git para almacenar el código fuente del proyecto
- Tableros Kanban para la gestión ágil de un proyecto, aplicando una metodología como Scrum.

Es una herramienta open source, y se podría haber instalado en local en una máquina virtual Linux, simplemente creando dicha máquina y creando un contenedor a partir de la imagen oficial disponible en <https://hub.docker.com/r/gitlab/gitlab-ce/>.

En esta práctica se utilizará gitlab.com porque ofrece prácticamente toda la funcionalidad en la nube de forma gratuita si un proyecto es público. Y porque de esta forma el profesor podrá acceder al proyecto y evaluar el trabajo realizado por el alumno.

Si se utilizara una máquina virtual, toda la práctica sería igual a lo que se verá en los siguientes apartados, y sólo habría que cambiar la URL gitlab.com por la IP de la máquina virtual en la que se estuviera ejecutando gitlab.

Para crear una cuenta en gitlab.com hay que acceder a <https://gitlab.com> y elegir Login > Register now.

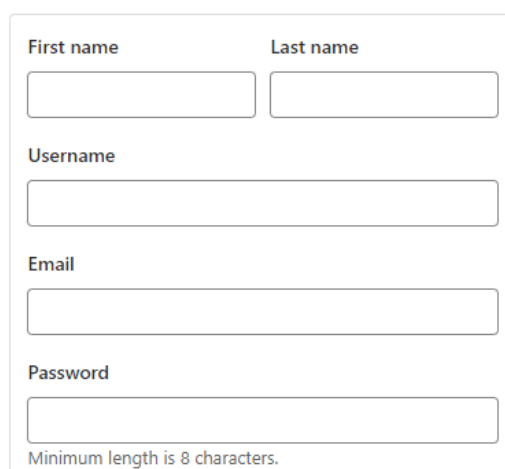
GitLab.com

GitLab.com offers free unlimited (private) repositories and unlimited collaborators.

- [Explore projects on GitLab.com](#) (no login needed)
- [More information about GitLab.com](#)
- [GitLab Community Forum](#)
- [GitLab Homepage](#)

By signing up for and by signing in to this service you accept our:

- [Privacy policy](#)
- [GitLab.com Terms](#).

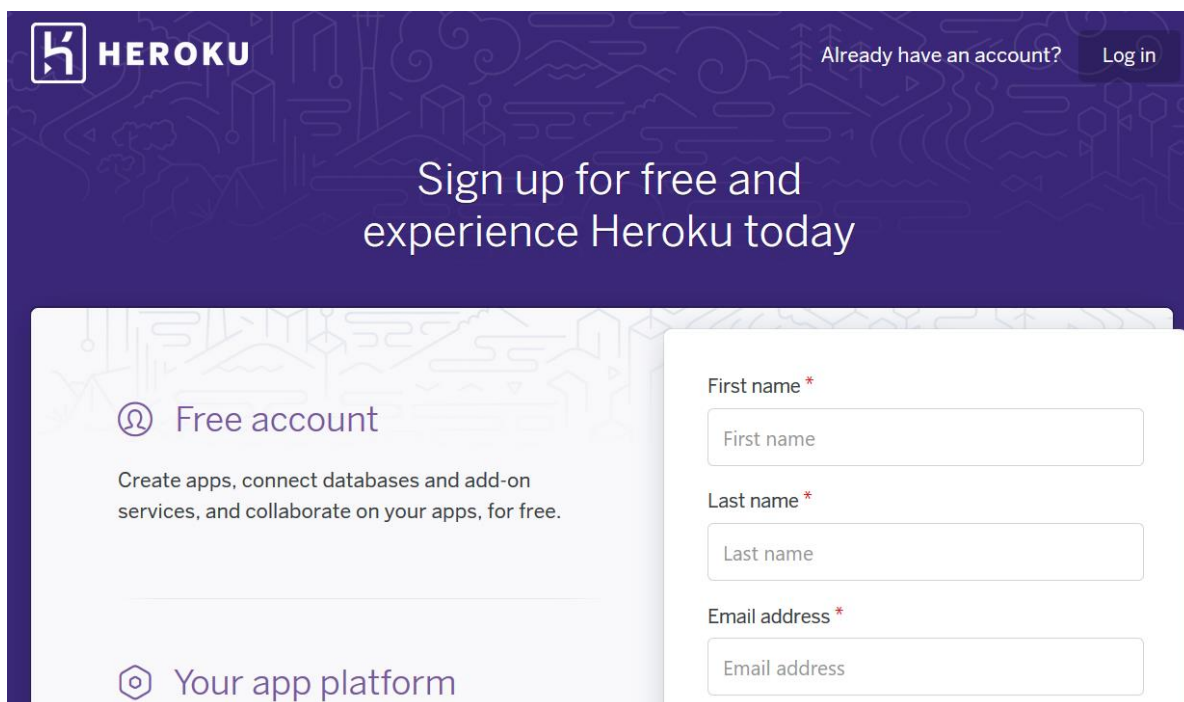


The image shows a registration form for GitLab.com. It includes input fields for 'First name', 'Last name', 'Username', 'Email', and 'Password'. Below the password field, it states 'Minimum length is 8 characters.'

1.2 Crear una cuenta gratuita en heroku.com

Para simular un entorno real de producción en el que quede instalada la aplicación web que se desarrolle, se puede utilizar un servicio de hosting en la nube que admita el despliegue mediante comandos desde una consola Linux. En esta práctica se utilizará heroku.com, porque permite desplegar aplicaciones web Java y esa gratuito instalar hasta 3 aplicaciones.

Para crear una cuenta, hay que acceder a <https://signup.heroku.com/> y registrarse:



1.3 Instalar docker

Se vio en la asignatura anterior.

Descargar de <https://docs.docker.com/get-docker/> para el SO que corresponda. En los ejemplos de esta práctica, se supone que se trabaja en un ordenador Windows, pero funcionaría igual en un ordenador Linux.

Comprobar que está instalado, desde la consola con el comando:

```
C:\> docker --version
Docker version 20.10.6, build 370c289
```

1.4 Instalar git

Git se necesita para crear y gestionar el repositorio de código del proyecto, así como sus versiones. Es necesario para subir nuevas versiones del código al servidor gitlab.com, que también tiene instalado git, y donde se almacena el repositorio remoto compartido por todos los desarrolladores. Git también se necesita en local para descargar desde gitlab.com las nuevas versiones del código del proyecto que haya modificado otro desarrollador.

Descargar e instalar desde <https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Instalaci%C3%B3n-de-Git>, eligiendo el sistema que corresponda. Para Windows: <https://git-scm.com/download/win>

Comprobar que está instalado, desde la consola con el comando:

```
C:\> git --version
git version 2.5.2.windows.2
```

1.5 Instalar visual studio code

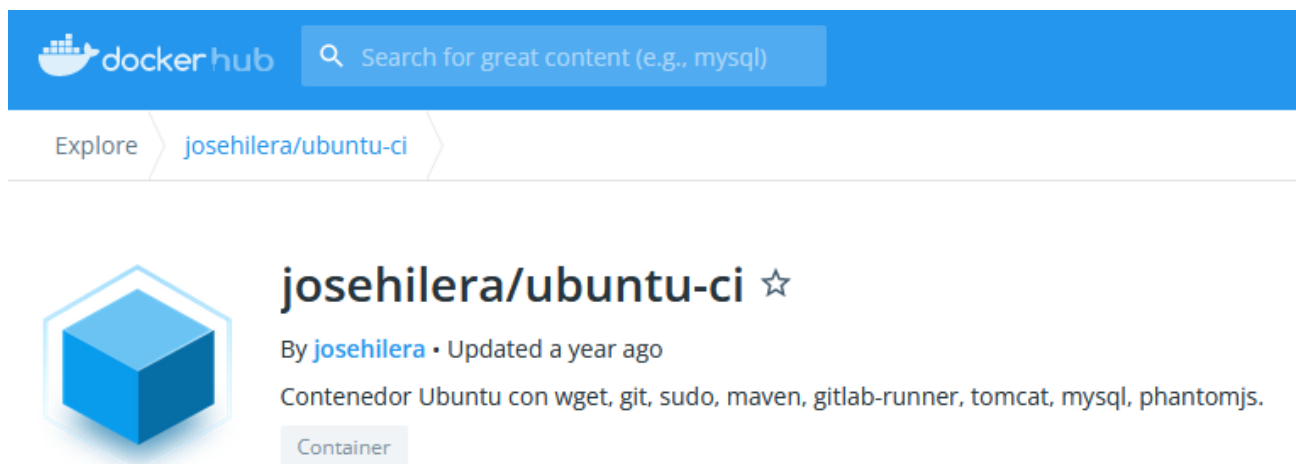
Aunque se puede utilizar cualquier entorno de desarrollo para Java, en esta práctica se asume que el desarrollador va a utilizar Visual Studio Code para hacer cambios en la aplicación web. Esta herramienta tiene la ventaja de ofrecer editores para todos los tipos de archivos que habrá que editar en la práctica, y además se integra con git, de tal forma que el desarrollador sabe en cada momento en que rama está trabajando, y los cambios pendientes de actualizar en el repositorio remoto.

Se descarga desde:

<https://code.visualstudio.com/download>

1.6 Instalar el contenedor ubuntu-ci

Descargar la imagen <https://hub.docker.com/r/josehilera/ubuntu-ci>.



Mediante el comando:

```
C:\> docker pull josehilera/ubuntu-ci
```

Al finalizar, puede comprobarse que se ha descargado, mediante el comando:

```
C:\> docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED
josehilera/ubuntu-ci	latest	8bee474ca146	5 days ago
1.08GB			

Esta imagen permite crear el contenedor ubuntu-ci que aparece en la figura de la infraestructura del apartado 1, que contiene ya instalados: git, maven¹, gitlab-runner, tomcat, mysql, y phantomjs. Estas herramientas serán necesarias durante la práctica.

Para crear el contenedor hay que ejecutar el comando:

```
C:\> docker run -it --name ubuntu-ci -p 8080:8080 -p 3306:3306 josehilera/ubuntu-ci

warning: cannot change directory to /nonexistent: No such file or directory
Starting MySQL database server mysqld

[ OK ]

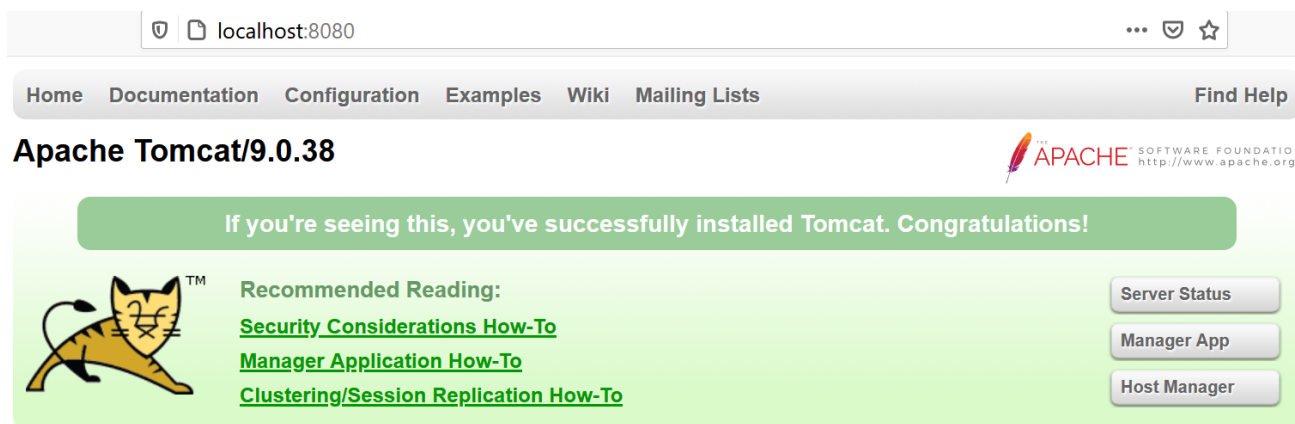
Using CATALINA_BASE:   /usr/local/tomcat
Using CATALINA_HOME:   /usr/local/tomcat
Using CATALINA_TMPDIR: /usr/local/tomcat/temp
Using JRE_HOME:        /usr
Using CLASSPATH:
/usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
root@ee734bd0faf8:/#
```

El contenedor está funcionando, es una máquina Linux (Ubuntu) que ha arrancado un servidor Tomcat en el Puerto 8080 y un servidor MySQL en el puerto 3306, u después deja abierta la consola para que podamos introducir comandos de Ubuntu.

En la dirección <https://hub.docker.com/r/josehilera/ubuntu-ci> puede verse lo que se ha ejecutado durante la creación del contenedor, en el apartado Overview, donde está el contenido del archivo Dockerfile que se usó para definir la imagen a partir de la que se ha creado el contenedor.

Puede comprobarse que el servidor Tomcat, está arrancado, accediendo a la URL <http://localhost:8080> desde un navegador:

¹ No ha hecho falta instalar Maven en el ordenador local porque en esta práctica se usará desde este contenedor ubuntu-ci, en el que ya está instalado.



En el caso de mysql, puede accederse al servidor mediante el comando mysql dentro de la consola del contenedor ubuntu-ci:

```
# mysql
Welcome to the MySQL monitor...

> mysql> show databases;
+-----+
| Database                |
+-----+
| information_schema      |
| mysql                   |
| performance_schema      |
| sys                     |
+-----+
4 rows in set (0.00 sec)
```

Para ver el estado del contenedor, sin cerrar la ventana de comandos, se puede abrir otra de Windows y ejecutar:

```
C:\> docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
ee734bd0faf8	josehilera/ubuntu-ci	"/bin/sh -c 'sudo se..."	17 minutes ago
Up 17 minutes		0.0.0.0:3306->3306/tcp, 0.0.0.0:8080->8080/tcp	

```
ubuntu-ci
```

Si en algún momento se detiene el contenedor, se puede arrancar de nuevo mediante:

```
C:\> docker start -ai ubuntu-ci
```

Además si queremos tener abierta otra consola del contenedor, se puede hacer mediante:

```
C:\> docker exec -it ubuntu-ci /bin/bash
```

1.7 Instalar el contenedor sonarqube

Como se indicaba en la figura de la infraestructura, también necesitaremos en la práctica un contenedor con un servidor sonarqube.

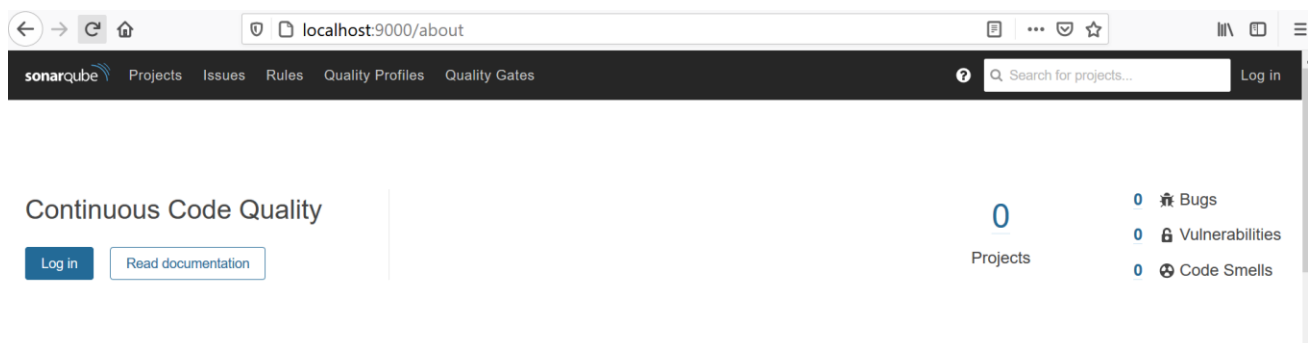
Para crearlo ejecutamos el comando:

```
C:\> docker run -d --name sonarqube -p 9000:9000 sonarqube
```

Puede comprobarse que está arrancado con el comando docker ps:

```
C:\> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS        PORTS         NAMES
bb3030a82d6d   sonarqube     "bin/run.sh bin/sona..." About a
minute ago    Up About a minute    0.0.0.0:9000->9000/tcp
sonarqube
ee734bd0faf8   josehilera/ubuntu-ci  "/bin/sh -c 'sudo se..." 27 minutes
ago          Up 27 minutes    0.0.0.0:3306->3306/tcp, 0.0.0.0:8080->8080/tcp
ubuntu-ci
```

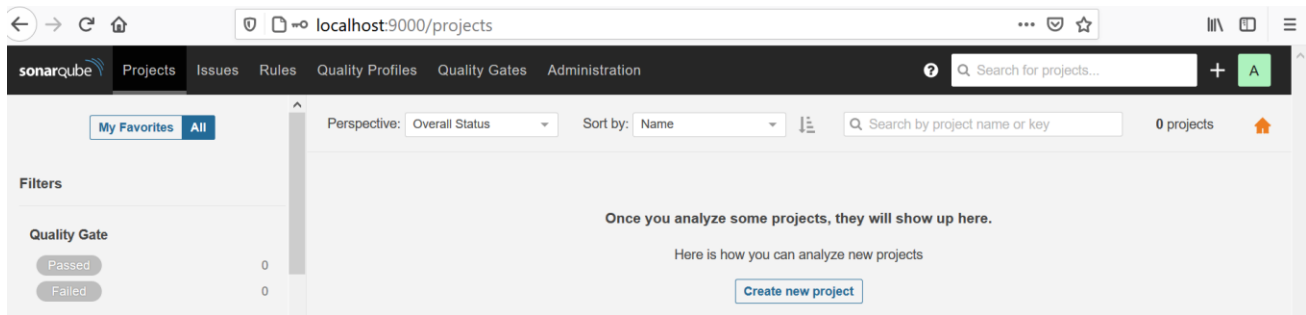
Ahora puede accederse al servidor con un navegador, en la URL localhost:9000



Por defecto existe un usuario “admin” con clave “admin”.

Log In to SonarQube

Log in Cancel



1.8 Entorno para pruebas locales en el ordenador de cada desarrollador

Para no alargar la práctica, no se usará un entorno para pruebas locales en el ordenador de cada desarrollador, pero hay que tener en cuenta que cada desarrollador trabaja con una copia local del código fuente de la aplicación en su máquina, y que cuando hace cambios en la aplicación, antes de enviarlos al repositorio remoto compartido, debe compilarlos y probarlos previamente en su máquina y asegurarse de que funcionan al menos a nivel local, para lo cual necesitará instalarse un entorno de desarrollo, maven, y también su propio servidor web y de base de datos.

Es importante resaltar que estos servidores locales sólo los usa el desarrollador en su máquina para sus pruebas y no están compartidos con otros desarrolladores, que probablemente tendrán algo parecido en sus respectivos ordenadores. Es habitual que cada miembro del equipo utilice una infraestructura local similar o acordada por el equipo, pero lo importante en integración continua es que todos compartan el repositorio remoto de código, donde se harán las integraciones de los cambios de la aplicación.

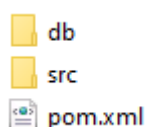
2. Crear repositorio local y proyecto en gitlab.com

Partimos del código fuente de la aplicación Baloncesto disponible en el campus virtual en el archivo Baloncesto.zip.

Hay que descomprimirlo, obteniéndose el mínimo código para compilar y ejecutar la aplicación.

Para los ejemplos de esta práctica se descomprimirá en ordenador Windows, en la raíz de un disco C:, y como programa de comandos se usará cmd de Windows. Se puede utilizar otro lugar del disco, otro sistema operativo y otro intérprete de comandos, como bash, disponible para todos los sistemas.

Al descomprimir se obtiene la carpeta (directorio) “Baloncesto” con la estructura:

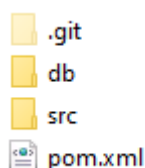


2.1 Crear repositorio git local

Para convertir la carpeta Baloncesto en un repositorio git para el control de versiones, y que permita subir (push) y bajar (pull) cambios en el código al repositorio remoto compartido por todos los desarrolladores del proyecto en el servidor de integración continua gitlab.com, se debe ejecutar el comando git init dentro de la carpeta:

```
C:\Baloncesto> git init
Initialized empty Git repository in D:/Baloncesto/.git/
```

Se ha creado una carpeta .git oculta con los archivos que necesita git para gestionar versiones del repositorio.



Ahora hay que crear en la carpeta del proyecto, con un editor de texto, un archivo con el nombre .gitignore (IMPORTANTE: sin extensión y con punto delante del nombre), en el que se indicarán los archivos y carpetas completas que no deberán subirse al repositorio remoto cuando se hagan envíos con push, ya que son archivos de uso particular del desarrollador, que han de ser ignorados por git, y no subirlos al remoto.

Por ejemplo, hay que incluir la carpeta /target/, ya que contiene los resultados de la compilación de la aplicación que haga el desarrollador para sus pruebas locales, pero que no hay que subir al repositorio remoto compartido, en que sólo debe almacenarse el código fuente de la aplicación, no los archivos resultantes de la compilación ni ejecutables. El servidor de integración continua

gitlab.com se encargará de recompilar la aplicación si se aceptan los cambios que envíen los desarrolladores.

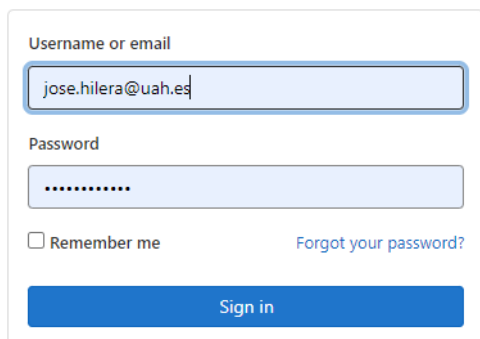
Otros archivos que hay que incluir en .gitignore son aquellos que genere el entorno de programación que utilice el desarrollador. Puede ocurrir que cada desarrollador de un mismo proyecto utilice en su ordenador un entorno de desarrollo diferente (Visual Studio Code, Netbeans, Eclipse, IntelliJ, etc.), que crea archivos en la carpeta del proyecto que no hay que subir al repositorio remoto, en el que el código compartido por todos los desarrolladores debe ser independiente de cualquier entorno de desarrollo. Por ejemplo, si se desarrolla localmente con Netbeans, existirá una carpeta nbproject, en el caso de usar VS Code, existirá la carpeta .vscode, etc, que habrá que ignorar.

Para evitar que se suban al repositorio remoto, el archivo .gitignore debería ser el siguiente, indicando en cada fila un archivo o carpeta que git debe ignorar al hacer push hacia el repositorio remoto.

Archivo C:\Baloncesto\gitignore
/target/ /nbproject/ /.vscode/

2.2 Crear repositorio remoto y proyecto en GitLab

Hay que tener una cuenta en gitlab.com, en los ejemplos de esta práctica se utilizará la cuenta jose.hilera. Cada alumno debe utilizar la suya.



Una vez dentro de GitLab, para crear un proyecto se selecciona el botón New Project > Create blank project, indicando como nombre “Baloncesto”, visibilidad “Public”, no marcar “Initialize repository with a README”.

New project › Create blank project

Project name

Balconcesto

Project URL

https://gitlab.com/

josehilera

Project slug

baloncesto

Want to house several dependent projects under the same namespace? [Create a group.](#)

Project description (optional)

Description format

Visibility Level [?](#)

☐ Private

Project access must be granted explicitly to each user. If this project is part of a group,

☒ Public

The project can be accessed without any authentication

Project Configuration

☐ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up

Analyze your source code for known security vulnerabilities. [Learn more.](#)

Create project

Cancel

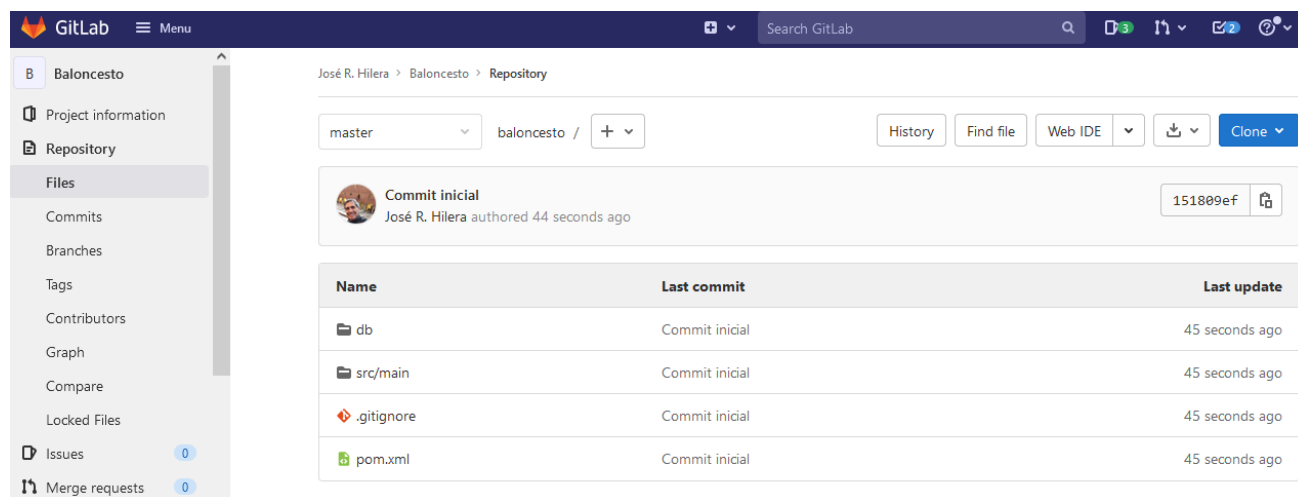
Al pulsar el botón para crear el proyecto aparecen sugerencias de comandos para subir un repositorio existente: "Push an existing folder". Que podremos ir copiando y pegando en la consola de nuestro ordenador local.

Volvemos a la consola de nuestro ordenador y dentro de la carpeta Baloncesto ejecutamos los siguientes comandos:

```
C:\Baloncesto> git remote add origin https://gitlab.com/josehilera/baloncesto.git
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Commit inicial"
C:\Baloncesto> git push -u origin master
Username for 'https://gitlab.com': jose.hilera@uah.es
^Password ...
```

² Si no queremos tener que introducir usuario y contraseña cada vez que hagamos git push, se pueden ejecutar los siguientes comandos:

Si volvemos a gitlab.com, vemos en la sección Repository que se ha subido una réplica del repositorio local, excepto la carpeta oculta .git, y los archivos y subcarpetas indicados en .gitignore.



2.3 Activar gitlab-runner en el contenedor ubuntu-ci

Gitlab.com utiliza el servicio externo [gitlab-runner](#) para ejecutar las órdenes de los pipelines del proyecto. Ese servicio puede estar instalado en un ordenador Linux, Windows o Mac con conexión a internet, y se recomienda que esté en una máquina diferente a la del servidor gitlab.com. En esta práctica se ha instalado en el contenedor ubuntu-ci que se ha creado en un apartado anterior.

Pero para que quede en funcionamiento y conectado al servidor gitlab.com para que le ordene la ejecución de trabajos, primero hay que registrarlo y luego hay que ejecutarlo.

Hay que tener arrancado el contenedor ubuntu-ci, como se indicó en el apartado 1.7. Si estuviera parado, se puede reactivar con el comando:

```
C:\> docker start -ai ubuntu-ci
```

Y aparece la consola de comandos de ubuntu-ci:

```
git config --global user.name "nombre de usuario"
git config --global user.password "contraseña"
```

```
C:\>docker start -ai ubuntu-ci
* Starting MySQL database server mysqld
su: warning: cannot change directory to /nonexi

Using CATALINA_BASE:   /usr/local/tomcat
Using CATALINA_HOME:   /usr/local/tomcat
Using CATALINA_TMPDIR: /usr/local/tomcat/temp
Using JRE_HOME:        /usr
Using CLASSPATH:        /usr/local/tomcat/bin/bo
Using CATALINA_OPTS:
Tomcat started.
root@dfa42fca94ec:/#
```

Debemos ejecutar dentro del contenedor el siguiente comando para registrar gitlab-runner:

```
# gitlab-runner register
Runtime platform arch=amd64 os=linux pid=328 revision=e95f89a0 version=13.4.1
Running in system-mode.
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
```

Nos pide la URL del servidor gitlab al que nos queremos conectar. Esta información y la que nos pedirá después, se encuentran en el proyecto que hemos creado en gitlab.com, en la sección settings > CD/CI > Runners.

Lo primero que haremos es desactivar en la sección “Shared runners” el botón “Enable shared runners for this project” porque vamos a utilizar un runner propio. Estos runners compartidos tienen un límite de tiempo de uso gratuito, a partir del cual hay que pagar por su uso.

Shared runners

These runners are shared across this GitLab instance.

Shared Runners on GitLab.com run in [autoscale mode](#) and are powered by Google Cloud Platform. Autoscaling means reduced wait times to spin up builds, and isolated VMs for each project, thus maximizing security.

They're free to use for public open source projects and limited to 400 CI minutes per month per group for private projects. Read about all [GitLab.com plans](#).

Enable shared runners for this project

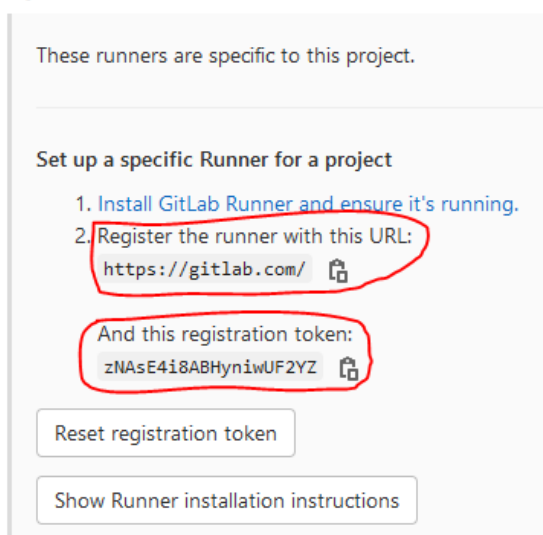


Available shared runners: 42

En la sección “Specific runners”, podemos ir copiando los valores que se nos pide en el registro del runner. Primero copiamos la URL, que es `https://gitlab.com/`, y la pegamos en la consola de ubuntu-ci:

Please enter the gitlab-ci coordinator URL (e.g. `https://gitlab.com/`):
`https://gitlab.com/`

Specific runners



Después gitlab-runner nos pide un token que copiamos de gitlab.com:

Please enter the gitlab-ci token for this runner:
`zNAse4i8ABHyniwUF2YZ`

NOTA: Cada alumno debe copiar el token que aparezca en su proyecto en gitlab.com.

A continuación nos pide un nombre para el runner. Podemos escribir cualquiera, por ejemplo “ubuntu-ci”.

Please enter the gitlab-ci description for this runner:
[dfa42fca94ec]: **`ubuntu-ci`**

Y después nos pide una etiqueta (tag), que debemos dejar en blanco, pulsando tecla intro.

Please enter the gitlab-ci tags for this runner (comma separated):

Y a continuación nos pide el tipo de ejecución que realizará el runner, en nuestro caso escribiremos “shell”, porque será un runner que aceptará comandos de la Shell de Ubuntu.

```
Registering runner... succeeded runner=zNAse4i8
Please enter the executor: docker-ssh, ssh, virtualbox, docker+machine, docker-ssh+machine, kubernetes, custom, docker, parallels, shell:
shell
```

Una vez registrado, hay que arrancar el runner, mediante el comando:

```
# gitlab-runner run
```

Si ha ido todo bien, podemos comprobar en gitlab.com, refrescando la página, que el runner está conectado y funcionando, porque aparece el nombre que dimos al runner y un círculo en verde.

Specific runners

These runners are specific to this project.

Set up a specific Runner for a project

1. Install [GitLab Runner](#) and ensure it's running.
2. Register the runner with this URL:
<https://gitlab.com/>

And this registration token:
`zNA5E4i8ABHyniwUF2YZ`

[Reset registration token](#)

[Show Runner installation instructions](#)

Available specific runners

● #12817745 (rkXoxhyV) [Remove runner](#)

ubuntu-ci

No debemos cerrar la consola de ubuntu-ci porque si no se detendría el runner. Si queremos trabajar con ubuntu-ci, podemos abrir otra consola con el siguiente comando de docker desde una nueva ventana de comandos de Windows:

```
C:\> docker exec -it ubuntu-ci /bin/bash
```

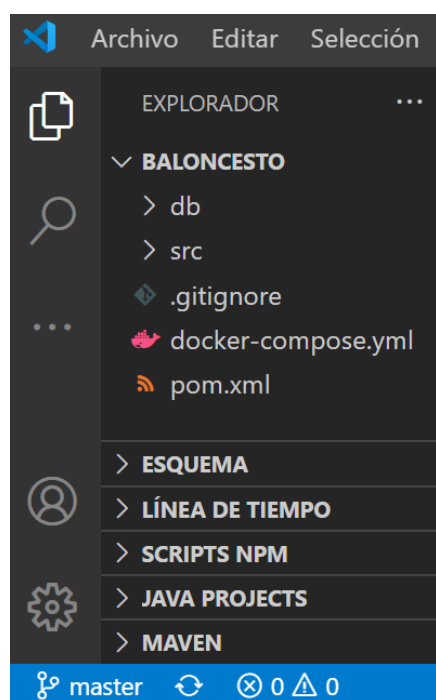
3. Diseñar el flujo de trabajos para la integración continua (pipeline)

GitLab actúa como servidor de integración continua, y en el repositorio creado todos los miembros de proyecto irán subiendo cambios.

Cada vez que se sube un cambio hay que compilar, probar y, en su caso, desplegar la aplicación a producción. Un principio de la integración continua es que el repositorio tenga todo lo necesario para recompilar y ejecutar la aplicación, que ha podido funcionar en el ordenador local, pero se trata de que funcione en uno o varios servidores web externos, bajo el control del servidor de integración.

Para que cada vez que se suba un cambio se recompile la aplicación, hay que preparar un flujo de trabajos (pipeline) en un archivo con el nombre `.gitlab-ci.yml` (**IMPORTANTE: debe empezar por punto**).

Este archivo lo creamos en el repositorio local con un editor de texto o un entorno de desarrollo. Se recomienda usar Visual Studio Code, pues ofrece utilidades para sincronizarse con el repositorio remoto, y nos indica en la parte inferior izquierda en que rama del proyecto estamos trabajando en cada momento. En este caso, si abrimos la carpeta Baloncesto con este editor, podemos ver que estamos en la rama master:



Para añadir un archivo `.gitlab-ci.yml` a la carpeta seleccionamos Archivo > Nuevo archivo.

Escribimos en el archivo el siguiente contenido, estableciendo cuatro fases o **stages** (build, test, qa, deploy). Con dos trabajos o **jobs** denominados pruebas-unitarias y empaquetar dentro de la fase build, un trabajo pruebas-funcionales dentro de la fase test, un trabajo calidad-codigo dentro de la fase de qa, y un trabajo despliegue-host-heroku dentro de la fase deploy.

Archivo C:\Baloncesto\gitlab-ci.yml

```
stages:
  - build
  - test
  - qa
  - deploy

empaquetar:
  stage: build
  script:
    - echo "empaquetar"

pruebas-unitarias:
  stage: build
  script:
    - echo "pruebas-unitarias"

pruebas-funcionales:
  stage: test
  script:
    - echo "pruebas-funcionales"

calidad-codigo:
  stage: qa
  script:
    - echo "calidad-codigo"

despliegue-host-heroku:
  stage: deploy
  script:
    - echo "despliegue-host-heroku"
```

Los trabajos consisten en comandos que ejecutará la máquina en la que se ha instalado el ejecutor gitlab-runner, en nuestro caso simulada mediante el contenedor ubuntu-ci. Estos comandos se indican en la sección script de cada trabajo.

Al empezar cada trabajo, lo primero que hará gitlab.com es enviar y copiar en el directorio raíz de la máquina el contenido completo de nuestro repositorio, por eso después los comandos script se aplicarán sobre los archivos del proyecto.

De momento en el archivo gitlab-ci.yml sólo hemos definido en los trabajos un comando echo que muestre el nombre del trabajo. En los siguientes apartados los sustituiremos por comandos que realicen de verdad el trabajo previsto en cada caso.

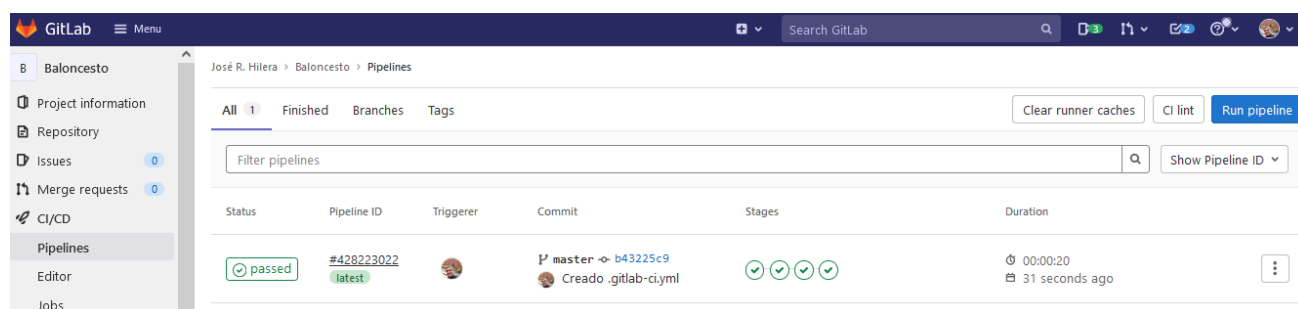
NOTA: Se puede validar si es correcta la sintaxis del archivo gitlab-ci.yml desde CI/CD > Pipelines > CI Lint, pegando el código en la ventana y pulsando Validate.

Los trabajos definidos en .gitlab-ci.yml se ejecutarán de forma automática cada vez que un desarrollador envíe cambios al repositorio remoto desde su repositorio local mediante un push.

Como acabamos de modificar el repositorio local añadiendo este nuevo archivo, podemos hacer un push contra el repositorio remoto para actualizar los cambios, y comprobaremos que se ejecutan los trabajos definidos en el archivo.

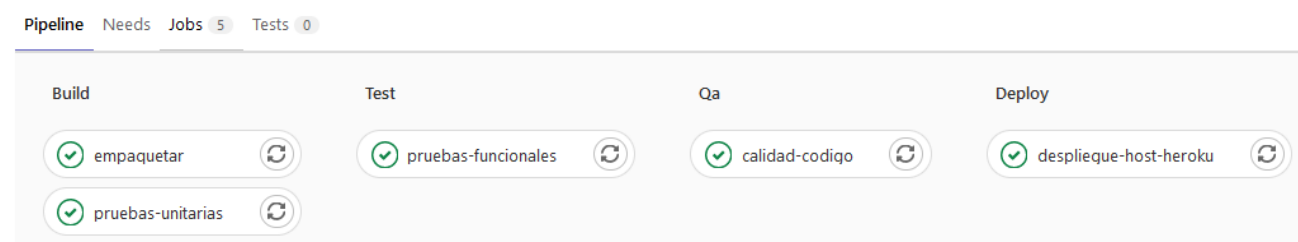
```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Creado .gitlab-ci.yml"
C:\Baloncesto>
```

En la sección CI/CD > Pipelines de gitlab.com podemos ver las ejecuciones de los trabajos definidos en el archivo .gitlab-ci.yml, cada pipeline es la ejecución de todos los trabajos como consecuencia de un push, y el nombre del pipeline es el del commit correspondiente a ese push. En este caso "Creado .gitlab-ci.yml".



Puede comprobarse que la ejecución del pipeline ha sido un éxito porque en Status se indica en verde "passed".

Si seleccionamos el identificador en la columna Pipeline, vemos un diagrama con las fases y trabajos realizados:



Todos están en verde indicando que han pasado con éxito. Si seleccionamos uno de los trabajos, por ejemplo "empaquetar", podemos ver la consola del contenedor ubuntu-ci en el que está el runner vinculado al proyecto que creamos en el apartado anterior, y lo que ha ido ocurriendo.

passed Job **empaquetar** triggered 5 minutes ago by José R. Hilera

```

1 Running with gitlab-runner 13.4.1 (e95f89a0)
2 on ubuntu-ci rkXxhyV
3 Resolving secrets
4
5 Preparing the "shell" executor
6 Using Shell executor...
7
8 Preparing environment
9 Running on dfa42fca94ec...
10
11 Getting source from Git repository
12 Fetching changes with git depth set to 50...
13 Initialized empty Git repository in /builds/rkXxhyV/0/josehilera/baloncesto/.git/
14 Created fresh repository.
15 Checking out b43225c9 as master...
16 Skipping Git submodules setup
17
18 Executing "step_script" stage of the job script
19 $ echo "empaquetar"
20 empaquetar
21
22 Cleaning up file based variables
23
24 Job succeeded

```

empaquetar

Retry

Elapsed time: 6 seconds

Timeout: 1h (from project)

Runner: #12817745 (rkXxhyV) ubuntu-ci

Commit [b43225c9](#)

Creado .gitlab-ci.yml

✓ Pipeline #428223022 for master

build

→ ✓ empaquetar✓ pruebas-unitarias

Podemos ver que primero se conectó al runner, después cargó una copia del repositorio del proyecto y finalmente ejecuta el comando echo que había en la sección script de este trabajo. También vemos a la derecha que el trabajo se ha ejecutado en un segundo.

Hay que tener en cuenta que el trabajo no se ha ejecutado en el servidor gitlab.com sino en el ordenador en el que está el runner instalado, y puede ocurrir que haya trabajos muy largos que tarden mucho tiempo en ejecutarse, por eso si utilizamos runners compartidos que ofrece gitlab.com, hay un límite de tiempo, a partir del cual su uso tiene un coste. No es nuestro caso, porque estamos usando un runner propio instalado en nuestro ordenador.

En los siguientes apartados vamos a ir definiendo el detalle de los trabajos a realizar, modificando el archivo .gitlab-ci.yml.

3.1 Fase (stage) build: Trabajo (job) empaquetar

Vamos a definir el trabajo “empaquetar” para el proyecto Baloncesto. Se trata de que se compile y se empaque la aplicación en un archivo baloncesto.war. Lo más sencillo es utilizar maven, pues est se puede hacer con un único comando. Por lo que modificamos el archivo .gitlab-ci.yml cambiando el comando echo por un comando mvn.

Archivo C:\Baloncesto\gitlab-ci.yml

```

empaquetar:
  stage: build
  script:
    - mvn package -DskipTests=true
  artifacts:
    paths:
      - target/

```


En el comando `mvn` indicamos la opción `DskipTest` a `true`, para que no se realicen las pruebas unitarias, pues no las hemos programado todavía, y además queremos que las pruebas unitarias se realicen en un trabajo que definiremos en otro apartado.

Además añadimos una sección `artifacts`, indicando con `path=target/`, que queremos que la carpeta `target` que se habrá generado en el contenedor `ubuntu-ci` al compilar sea enviada al servidor `gitlab.com` cuando termine de realizarse el trabajo “empaquetar”. Esto nos permitirá pasar esta carpeta, que contiene el archivo ejecutable `baloncesto.war`, a otros trabajos posteriores, como el de despliegue.

Un artefacto puede ser una carpeta o un archivo. Si sólo nos interesa el archivo `war`, se podría haber escrito el nombre del archivo o bien `*.war`:

```
paths:
  - target/Baloncesto.war
```

Hay que tener en cuenta que cuando finaliza un trabajo, se borra la copia del repositorio que había en el contenedor `ubuntu-ci`, incluida la carpeta `target` generada, y que cuando empiece el siguiente trabajo, se repite el proceso, copiando el repositorio de `gitlab.com`, que sólo almacena el código fuente, no los archivos compilados ni ejecutables.

Todavía no hacemos `push` contra el repositorio remoto, lo haremos en el siguiente apartado.

3.1.1 Compilar con un error forzado

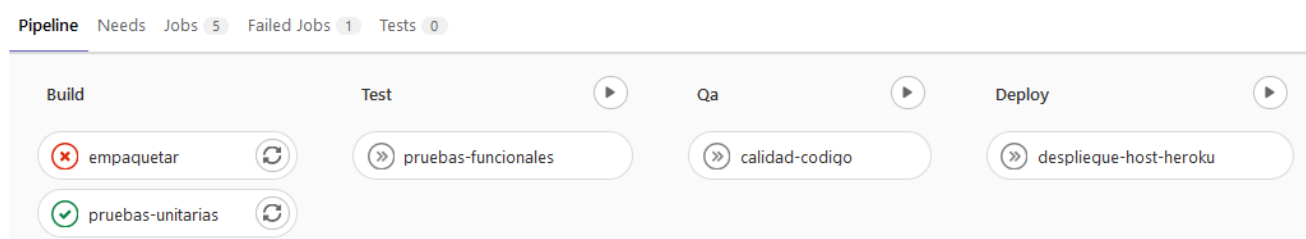
Vamos a ver cómo falla el trabajo de empaquetado forzando un error de compilación. Para ello modificamos con el editor el archivo en `src\main\java\Acb.java` en nuestro repositorio local, comentando la primera línea de código para forzar un error de compilación.

```
//import java.io.*;
```

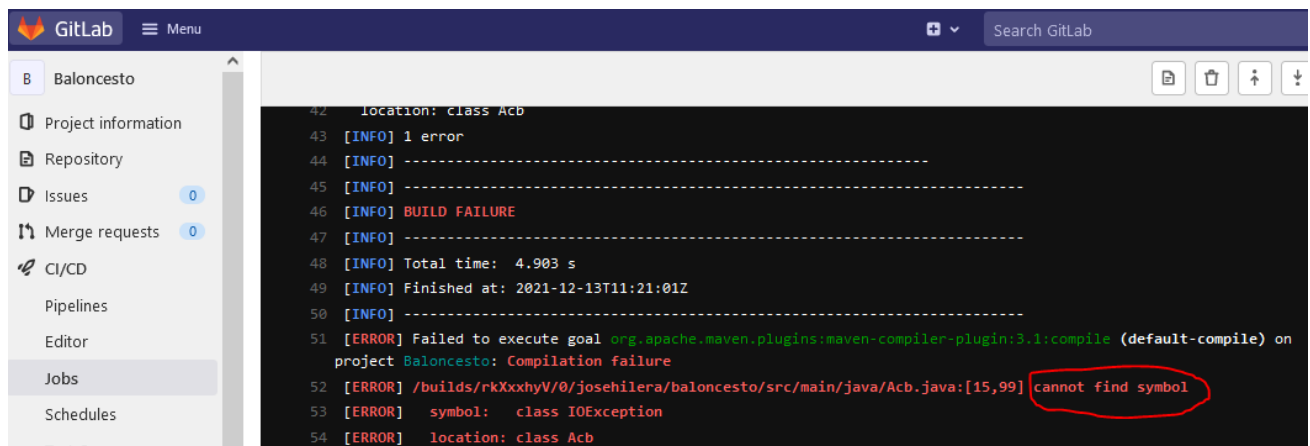
Salvamos todo y lo subimos al repositorio remoto:

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Error de compilación forzado"
C:\Baloncesto> git push
```

Vemos en `gitlab.com`, en la sección `CI/CD > pipelines`, que se ha creado un nuevo pipeline, y que se ha producido un error en el trabajo “empaquetar”:



Ello supone que habrá que revisar los errores y corregirlos. En la consola del trabajo empaquetar se puede ver cuál ha sido el error:



```
42 location: class Acb
43 [INFO] 1 error
44 [INFO] -----
45 [INFO] -----
46 [INFO] BUILD FAILURE
47 [INFO] -----
48 [INFO] Total time: 4.903 s
49 [INFO] Finished at: 2021-12-13T11:21:01Z
50 [INFO] -----
51 [ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:compile (default-compile) on
project Baloncesto: Compilation failure
52 [ERROR] /builds/rkXxhyV/0/josehilera/baloncesto/src/main/java/Acb.java:[15,99] cannot find symbol
53 [ERROR] symbol: class IOException
54 [ERROR] location: class Acb
```

3.1.2 Empaquetar sin errores

Para corregir el error, editamos de nuevo en local quitando el comentario de Acb.java:



















```
import java.io.*;
```

Guardamos y lo volvemos a subir:

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Error de compilación corregido"
C:\Baloncesto> git push
```

Comprobamos que se arranca un nuevo pipeline y que ahora no hay ningún error.

José R. Hilera > Baloncesto > Pipelines

All 3 Finished Branches Tags				
Filter pipelines				
Status	Pipeline ID	Triggerer	Commit	Stages
passed	#428232617 latest		P master -> 528a6c7f  Error de compilación cor...	   
failed	#428230009		P master -> 7da5b275  Error de compilación for...	   
passed	#428223022		P master -> b43225c9  Creado .gitlab-ci.yml	   

Como indicamos en `.gitlab-ci.yml` que queríamos recuperar como artefacto toda la carpeta `/target/` generada en el contenedor donde se ejecute el runner antes de que se borre al finalizar el trabajo empaquetar, podemos comprobar que ese artefacto se encuentra en la página del trabajo. A la derecha aparece una sección “Job artifacts”, donde podemos seleccionar Browse y podemos acceder a los artefactos generados en ese trabajo.

José R. Hilera > Baloncesto > Jobs > #1877614587 > Artifacts

✓ passed Job #1877614587 in pipeline #428232617 for 528a6c7f from master by José R. Hilera 8 minutes ago

Artifacts / target

Name	Size
..	
Baloncesto	
classes	
generated-sources	
maven-archiver	
maven-status	
Baloncesto.war	3.88 MB

Más adelante en la práctica veremos cómo se puede usar un artefacto en otro trabajo de una fase posterior de un pipeline.

3.2 Fase build: Trabajo pruebas-unitarias

Como parte de la fase de construcción de la aplicación podemos incluir un trabajo de realización de las pruebas unitarias. Si lo incluimos en la misma fase, gitlab.com considera que se puede realizar en paralelo con el trabajo de empaquetar que está en la misma fase.

El diseñador del flujo de trabajos debe decidir cómo organizar las fases y trabajos. En esta práctica se han incluido estos dos trabajos en la misma fase, sólo para ver que se ejecutan en paralelo, pero seguramente sería más lógico crear una fase previa para las pruebas unitarias y después el empaquetado. Para hacer pruebas unitarias no hace falta compilar toda la aplicación. Se anima al alumno a modificar el diseño del flujo de trabajos.

Como es un proyecto maven, el comando para ejecutar las pruebas es tan sencillo como `mvn test`, y así habrá que escribirlo en el archivo `.gitlab-ci.yml`, sustituyendo el comando `echo` anterior:

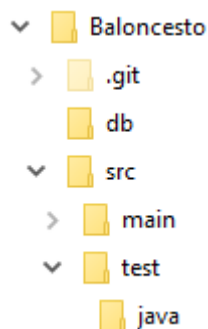
Archivo C:\Baloncesto\gitlab-ci.yml
<pre>pruebas-unitarias: stage: build script: - mvn test</pre>

Para ejecutar las pruebas unitarias hay que decidir qué framework se utilizará, en este caso usaremos JUnit 5, por lo que hay que modificar el archivo pom.xml del repositorio local para añadir las dependencias necesarias, que son las siguientes:

Archivo C:\Baloncesto\pom.xml
<pre><dependency> <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-api</artifactId> <version>5.3.1</version> <scope>test</scope> </dependency> <dependency> <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-params</artifactId> <version>5.3.1</version> <scope>test</scope> </dependency> <dependency> <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-engine</artifactId> <version>5.3.1</version> <scope>test</scope> </dependency></pre>

3.2.1 Realizar una prueba fallida

Como es un proyecto con estructura maven, los test deben programarse en la carpeta src\test\java\, que hay que crear.



Vamos a crear un archivo `ModeloDatosTest.java` en dicha carpeta, con el siguiente código.

Archivo C:\Baloncesto\src\test\java\ModeloDatosTest.java

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class ModeloDatosTest {

    @Test
    public void testExisteJugador() {
        System.out.println("Prueba de existeJugador");
        String nombre = "";
        ModeloDatos instance = new ModeloDatos();
        boolean expResult = false;
        boolean result = instance.existeJugador(nombre);
        //assertEquals(expResult, result);
        fail("Fallo forzado.");
    }
}
```

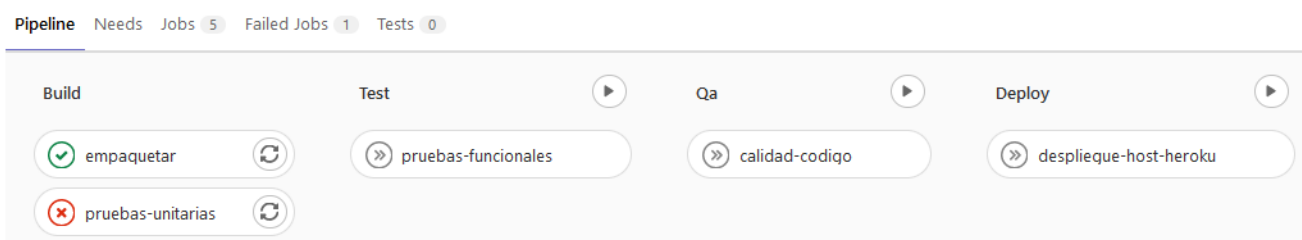
Se trata de una prueba del método `existeJugador()`. Es una prueba ficticia sólo para probar el funcionamiento de la integración continua. Se recomienda que se prepare una prueba real.

En este caso se ha incluido una sentencia “fail” para forzar un error y comprobar que gitlab lo detecta cuando se ejecute el pipeline.

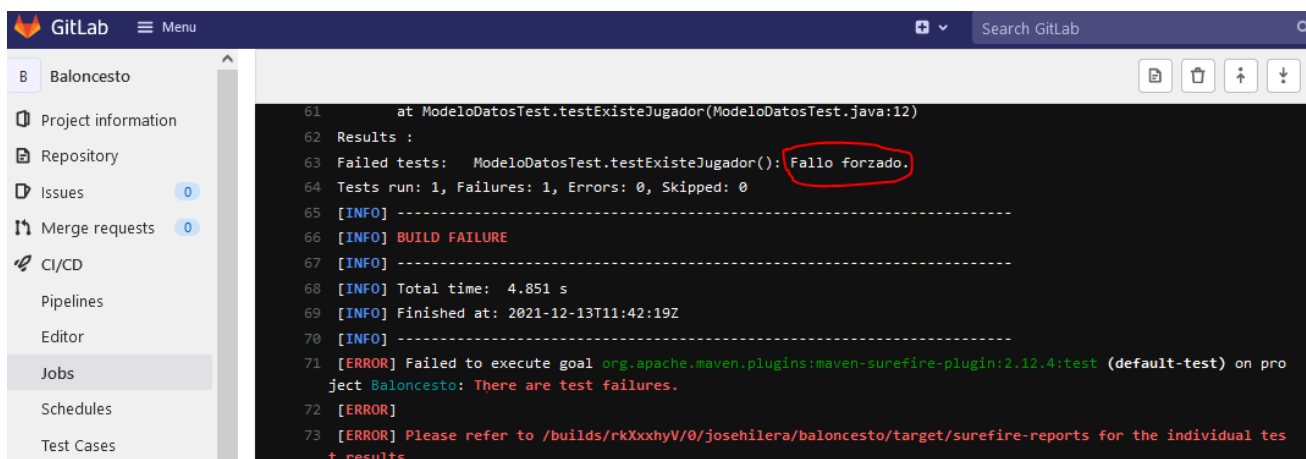
Una vez guardados los tres archivos modificados: `.gitlab-ci.yml`, `pom.xml` y `ModeloDatosTest.java`, hacemos push al repositorio remoto, para ver que al desencadenarse de nuevo la pipeline, se interrumpe el trabajo prueba-unitaria.

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Fallo forzado en prueba unitaria"
C:\Baloncesto> git push
```

Puede comprobarse en el nuevo pipeline que ahora falla el trabajo pruebas-unitarias:



Si accedemos al detalle del trabajo pruebas-unitarias, puede verse que ha fallado la prueba `testExisteJugador`, debido a que habíamos forzado la ejecución de la sentencia `fail`.



3.2.2 Realizar una prueba sin errores

Para que no falle la prueba, hay que borrar o comentar la sentencia `fail` en el archivo `\src\test\java\ModeloDatosTest.java`, y quitar el comentario de la sentencia `assertEquals`:

```
assertEquals(expResult, result);
//fail("Fallo forzado.");
```

Guardar el archivo y volver a subir al repositorio remoto:

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Prueba unitaria corregida"
C:\Baloncesto> git push
```

Puede comprobarse que este segundo pipeline se realiza sin problemas:

Status	Pipeline ID	Triggerer	Commit	Stages	Duration
 passed	#428248922 latest		master  fa538cc6  Prueba unitaria corregida	   	 00:00:45  55 seconds ago
 failed	#428244115		master  825459b4  Fallo forzado en prueba ...	   	 00:00:27  10 minutes ago

3.3 Fase test: Trabajo pruebas-funcionales

La siguiente fase definida en el pipeline es la de test, que se realiza una vez empaquetada la aplicación y superadas las pruebas unitarias. En esta fase hay un solo trabajo denominado pruebas-funcionales.

En este trabajo, hay que preparar los comandos necesarios para de forma automática:

- Instalar la aplicación Baloncesto, que se empaquetó en la fase anterior, en un servidor Tomcat
- Crear una base de datos “baloncesto” en un servidor MySQL
- Realizar pruebas del funcionamiento de la aplicación simulando su uso por parte de un usuario que utiliza un navegador web simulado, para lo cual se utilizarán las librerías Selenium y Phantomjs.

Hay que modificar el archivo .gitlab-ci.yml cambiando el trabajo pruebas-funcionales por el código siguiente:

Archivo C:\Baloncesto\gitlab-ci.yml
<pre>pruebas-funcionales: stage: test dependencies: - empaquetar script: - cp -r target/Baloncesto /usr/local/tomcat/webapps - mysql -u root < db/baloncesto.sql - export DATABASE_HOST="jdbc:mysql://localhost" - export DATABASE_PORT="3306" - export DATABASE_NAME="baloncesto" - export DATABASE_USER="usuario" - export DATABASE_PASS="clave" - sh /usr/local/tomcat/bin/catalina.sh stop - sh /usr/local/tomcat/bin/catalina.sh start - mvn failsafe:integration-test failsafe:verify</pre>

Hay que tener en cuenta que estos comandos se ejecutarán en la máquina (contenedor) ubuntu-ci, en la que esté corriendo el runner creado para el proyecto (ver apartado 2.3 de esta práctica). Y en ese contenedor ya están instalados y arrancados un servidor Tomcat y un servidor MySQL (ver apartado 1.6).

Por ello la instalación de la aplicación es tan sencillo como copiar la carpeta target/Baloncesto en la carpeta webapps de Tomcat. Ya que como se ha dicho, cuando empieza el trabajo, desde gitlab.com se ha enviado a la máquina donde está el runner, una copia del repositorio, y los comandos se ejecutarán desde el directorio en la que está esa copia.

A continuación se explica el contenido de .gitlab-ci.yml:

- **Sección “dependencies”:** En el repositorio copiado a la máquina del runner no están los ejecutables de la aplicación, pero gitlab permite utilizar la sección dependencies para indicar que un trabajo depende de otro anterior, y lo que hace es copiar en la máquina del runner los artefactos que generó aquel otro trabajo anterior. En este caso depende del trabajo

“empaquetar”, que generó como artefacto la carpeta target completa, por lo que está disponible en la máquina del runner, que en esta práctica es el contenedor ubuntu-ci.

- **Comando para instalar la aplicación web:** Como sabemos que al arrancar el trabajo, en el contenedor ubuntu-ci estará la carpeta target con la aplicación generada, podemos ejecutar el comando `cp -r target/Baloncesto /usr/local/tomcat/webapps`, para copiar la subcarpeta Baloncesto en tomcat, y ya estaría instalada la aplicación.
- **Comando para crear la base de datos:** Como al arrancar el trabajo se copia el repositorio del proyecto en el contenedor ubuntu-ci, en la subcarpeta db tenemos el archivo baloncesto.sql cpo el script de creación de la base de daos, por lo que sólo hay que ordenar su ejecución desde la consola de ubuntu-ci con el comando `mysql -u root < db/baloncesto.sql`. Hay que tener en cuenta que el usuario root por defecto no tiene clave, por eso no es necesaria en este comando.
- **Comandos export para las variables de entorno:** En el código de ModeoDatos.java, los datos de conexión se obtienen de variables de entorno. En los cinco comandos export se da valores a dichas variables.
- **Comandos de parada y re-arranque de Tomcat:** Para que el servidor Tomcat cargue las variables de entorno hay que pararlo y volverlo a arrancar.
- **Comando para ejecutar las pruebas funcionales:** Al utilizar maven, es tan secillo como ejecutar el comando `mvn failsafe:integration-test`. Este comando localiza los archivos cuyo nombre termina en “IT” (Integration Test) dentro de la carpeta src/test/java y los ejecuta. Se supone que en esos archivos están programadas las pruebas de integración de la aplicación, entre las que se encuentran las pruebas funcionales, también llamadas end-to-end o de interfaz de usuario.

La prueba vamos a programar en el siguiente apartado utiliza las librerías Selenium y phantomjs, por lo que hay que añadir dos nuevas dependencias al archivo pom.xml del proyecto:

Archivo C:\Baloncesto\pom.xml
<pre><dependency> <groupId>org.seleniumhq.selenium</groupId> <artifactId>selenium-java</artifactId> <version>2.41.0</version> </dependency> <dependency> <groupId>com.github.detro.ghostdriver</groupId> <artifactId>phantomjsdriver</artifactId> <version>1.1.0</version> </dependency></pre>

3.3.1 Realizar una prueba fallida

Vamos a programar una prueba funcional en un archivo que llamaremos PruebasPhantomjsIT.java, ubicado en la carpeta `src\test\java\`.

Cuando se programan pruebas de este tipo hay que preparar pruebas que simulen diferentes navegadores web, con la limitación de que el contenedor ubuntu-ci no tiene interfaz gráfica de usuario, por lo que hay que usar algún navegador sin interfaz de usuario (“headless browser”).

Existen versiones headless para Chrome, Firefox, etc., y se anima al alumno a crear la misma prueba para ellos. Por sencillez, en esta práctica usaremos Phantomjs.

El código del archivo para las pruebas es el siguiente:

Archivo C:\Baloncesto\src\test\java\PruebasPhantomjsIT.java

```
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriverService;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class PruebasPhantomjsIT
{
    private static WebDriver driver=null;

    @Test
    public void tituloIndexTest()
    {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setJavascriptEnabled(true);

        caps.setCapability(PhantomJSDriverService.PHANTOMJS_EXECUTABLE_PATH_PROPERTY, "/usr/bin/phantomjs");
        caps.setCapability(PhantomJSDriverService.PHANTOMJS_CLI_ARGS, new String[] {"--web-security=no", "--ignore-ssl-errors=yes"});
        driver = new PhantomJSDriver(caps);
        driver.navigate().to("http://localhost:8080/Baloncesto/");
        assertEquals("Votacion mejor jugador liga ACB", driver.getTitle(),
"El titulo no es correcto");
        System.out.println(driver.getTitle());
        driver.close();
        driver.quit();
    }
}
```

Esta prueba lo único que hace es abrir la aplicación Baloncesto previamente instalada en Tomcat, y comprueba el contenido de la etiqueta <title> del archivo index.html de la aplicación, que es el que se abre por defecto en la URL <http://localhost:8080/Baloncesto/>. Comprueba si el título es “Votacion mejor jugador liga ACB”.

Como en la versión actual del archivo index.html está bien, vamos a forzar un error cambiando “ACB” por “NBA” en la etiqueta <title>:

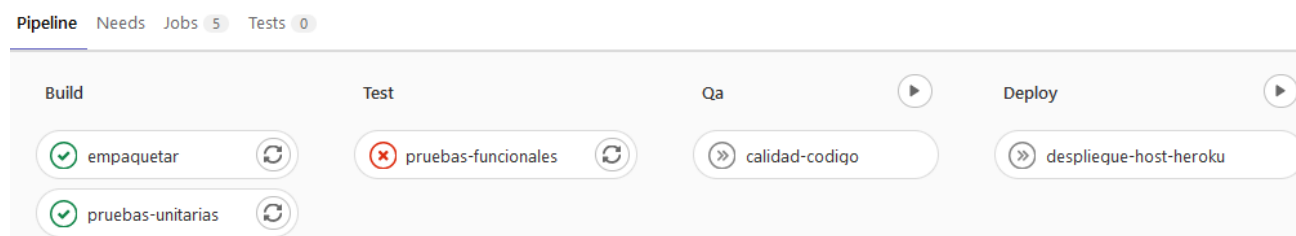
```
<title>Votacion mejor jugador liga NBA</title>
```

Debido a esto deberá generarse un error, porque el contenido de la etiqueta <title> esperado sería “Votacion mejor jugador liga ACB”.

Guardamos los cuatro archivos modificados: .gitlab-ci.yml, pom.xml, index.html y PruebasPhantomjsIT.java, hacemos push al repositorio remoto, para ver que al desencadenarse de nuevo la pipeline, se interrumpe el trabajo pruebas-funcionales.

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Fallo forzado en prueba funcional"
C:\Baloncesto> git push
```

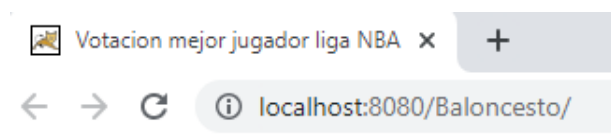
Puede comprobarse en el nuevo pipeline que ahora falla el trabajo pruebas-funcionales:



Si accedemos al detalle del trabajo pruebas-unitarias, puede verse que ha fallado la prueba por no coincidir el título esperado.

```
65 [INFO - 2021-12-13T12:03:20.471Z] SessionManagerReqHand - _postNewSessionCommand - New Session Created: aa157a
66 [ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 9.238 s <<< FAILURE! - in PruebasPhantomjsIT
67 [ERROR] PruebasPhantomjsIT.tituloIndexTest Time elapsed: 9.148 s <<< FAILURE!
68 org.opentest4j.AssertionFailedError: El título no es correcto ==> expected: <Votacion mejor jugador liga ACB> but was: <Votacion mejor jugador liga NBA>
69 at PruebasPhantomjsIT.tituloIndexTest(PruebasPhantomjsIT.java:20)
70 [INFO]
71 [INFO] Results:
72 [INFO]
73 [ERROR] Failures:
74 [ERROR] PruebasPhantomjsIT.tituloIndexTest:20 El título no es correcto ==> expected: <Votacion mejor jugador liga ACB> but was: <Votacion mejor jugador liga NBA>
75 [INFO]
76 [ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
```

Como el error se ha generado después de instalar la aplicación en el servidor Tomcat de prueba del el contenedor ubuntu-ci donde está gitlab-runner, podemos abrirla con un navegador en <http://localhost:8080/Baloncesto/> y comprobar en el título de la ventana que en efecto aparece “NBA”:



3.3.2 Realizar una prueba sin errores

Para que no falle la prueba, corregimos el archivo index.html:

```
<title>Votacion mejor jugador liga NBA</title>
```

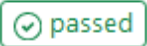





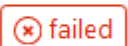





Cambiando NBA por ACB:

```
<title>Votacion mejor jugador liga ACB</title>
```

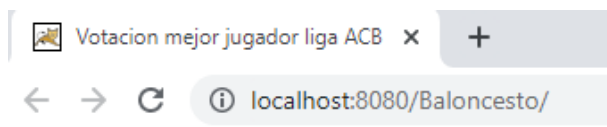
Guardamos el archivo y volvemos a subir al repositorio remoto:

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Prueba funcional corregida"
C:\Baloncesto> git push
```

Puede comprobarse que este segundo pipeline se realiza sin problemas:

Status	Pipeline ID	Triggerer	Commit	Stages
	#428266476 latest		master → b6184cd3 Prueba funcional corri...	   
	#428257488		master → 5f5c70c3 Fallo forzado en prueba ...	   

Como en el apartado anterior la aplicación queda instalada en el servidor Tomcat de pruebas del contenedor ubuntu-ci donde está gitlab-runner, podemos abrirla con un navegador en <http://localhost:8080/Baloncesto/> y comprobar en el título de la ventana que ya aparece el “ACB”:



Podemos incluso utilizar la aplicación en el servidor de pruebas. En esta aplicación se puede votar a jugadores y las votaciones quedan registradas en una tabla “Jugadores” de la base de datos “baloncesto” en el servidor MySQL de pruebas también instalado en el contenedor ubuntu-ci. Por ejemplo podemos votar al primer jugador:

Votación al mejor jugador de la liga ACB

Nombre del Visitante: eMail:

REAL MADRID:

☒ Carroll

☐ Llull

☐ Rudy

☐ Otro

Y comprobar que esa votación se ha registrado en la base de datos. Para lo cual abrimos una consola del contenedor ubuntu-ci:

```
C:\Baloncesto> docker exec -it ubuntu-ci /bin/bash
```

Y ejecutamos en la consola de ubuntu-ci el comando mysql para abrir el cliente de la base de datos:

```
# mysql
mysql> use baloncesto
mysql> select * from Jugadores;
+----+-----+-----+
| id | nombre | votos |
+----+-----+-----+
| 1  | Carroll | 1     |
| 2  | Llull   | 0     |
| 3  | Rudy    | 0     |
+----+-----+-----+
3 rows in set (0.00 sec)
```

Y vemos que el voto ha quedado registrado.

3.4 Fase qa: Trabajo calidad-codigo

Para poder realizar este trabajo, necesitamos instalar un servidor sonarqube en alguna máquina. Lo más sencillo en este caso es simularlo utilizando un contenedor, a partir de la imagen oficial sonarqube disponible en docker hub: https://hub.docker.com/_/sonarqube

Para ello, debemos crear un contenedor que se llamará sonarqube con el comando:

```
C:\> docker run -d --name sonarqube -p 9000:9000 sonarqube
```

Podemos probar que el servidor sonar está arrancado, abriendo en un navegador la dirección:

<http://localhost:9000>

Los datos de acceso por defecto son admin/admin.

Necesitamos que el contenedor ubuntu-ci se pueda conectar con éste contenedor sonarqube, por lo que habrá que volver a crearlo, utilizando la opción link en el comando run de docker. Pero antes hay que parar y eliminar el contenedor anterior con el comando rm de docker.

```
C:\> docker stop ubuntu-ci
```

```
C:\> docker rm ubuntu-ci
```

```
C:\> docker run -it --name ubuntu-ci -p 8080:8080 -p 3306:3306 --link sonarqube  
josehilera/ubuntu-ci
```

Desde dentro del contenedor ubuntu-ci, la URL que hay que usar para acceder al servidor sonarqube, no es localhost, sino el nombre del contenedor: <http://sonarqube:9000>

Una vez hechos estos cambios, ya podemos definir el trabajo en el archivo .gitlab-ci.yml.

Archivo C:\.gitlab-ci.yml
calidad-codigo: stage: qa dependencies: - empaquetar script: - mvn sonar:sonar -Dsonar.host.url=http://sonarqube:9000 -Dsonar.qualitygate.wait=true -Dsonar.login=admin -Dsonar.password=admin allow_failure: true

Hay que incluir la dependencia, porque SonarQube cuando detecta que hay archivos .java en un proyecto, [exige que también estén los archivos .class](#), en otro caso se genera un error.

En el comando mvn se han indicado como usuario y password “admin”, si se han modificado en SonarQube hay que cambiarlos en el comando.

En el archivo pom.xml del proyecto hay que añadir el plugin de sonarqube.

Archivo C:\pom.xml
<plugin> <groupId>org.sonarsource.scanner.maven</groupId> <artifactId>sonar-maven-plugin</artifactId> <version>3.6.0.1398</version> </plugin>

Una vez guardados los cambios hay que hacer push sobre el repositorio remoto. Pero antes hay que recordar volver a registrar y ejecutar gitb-runner en el contenedor ubuntu-ci, ya que este contenedor se ha creado de nuevo para conectarse con el contenedor sonarqube:

```
# gitlab-runner register  
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):  
https://gitlab.com/
```

Please enter the gitlab-ci token for this runner:

zNAsE4i8ABHyniwUF2YZ → BUSCAR TOKEN EN gitlab.com > settings > CD/CI > Runners > Specific runners

Please enter the gitlab-ci description for this runner:

[dfa42fca94ec]: **ubuntu-ci**

Please enter the gitlab-ci tags for this runner (comma separated):

NO ESCRIBIR NADA → PULSAR INTRO

Registering runner... succeeded runner=zNAsE4i8

Please enter the executor: docker-ssh, ssh, virtualbox, docker+machine, docker-ssh+machine, kubernetes, custom, docker, parallels, shell:

shell

```
# gitlab-runner run
```

Una vez hecho esto, podemos hacer push al repositorio remoto:

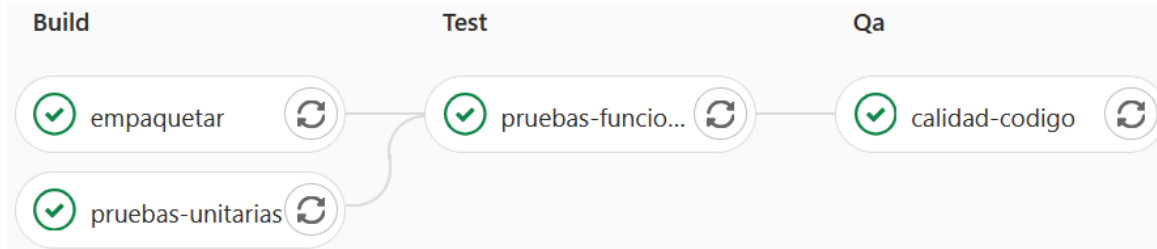
```
C:\Baloncesto> git add .
```

```
C:\Baloncesto> git status
```

```
C:\Baloncesto> git commit -m "Creado trabajo calidad-codigo"
```

```
C:\Baloncesto> git push
```

Al ejecutarse el trabajo calidad-codigo, puede comprobarse que pasa sin problemas.



3.4.1 Aumentar la exigencia de calidad

Si accedemos al servidor sonarqube <http://localhost:9000> (admin/admin), en la sección Projects > Baloncesto > Issues > Major podemos ver que hay 24 fallos de calidad importantes (major issues).

The screenshot shows the SonarQube 'Issues' page for the 'Baloncesto' project. In the left sidebar, under 'Filters', the 'Severity' filter is set to 'MAJOR', showing 24 issues. The main area displays a list of issues. The first three issues are 'Code Smell' with 'Major' severity, titled 'Add the "@Override" annotation above this method signature'. The fourth issue is also a 'Code Smell' with 'Major' severity, titled 'Replace this use of System.out or System.err by a logger.'

Podemos hacer que sea más exigente, y definir una regla para que no pase la prueba de calidad del código si el número de fallos de calidad importantes es mayor de una cantidad. Esto se hace en la sección:

Quality Gates > Create > Name = "control calidad" > Add Condition > On Overall code > Quality Gate fails when > Major issues is greater than 5

Add Condition

☐ On New Code ☒ On Overall Code

Quality Gate fails when

Major Issues

Operator

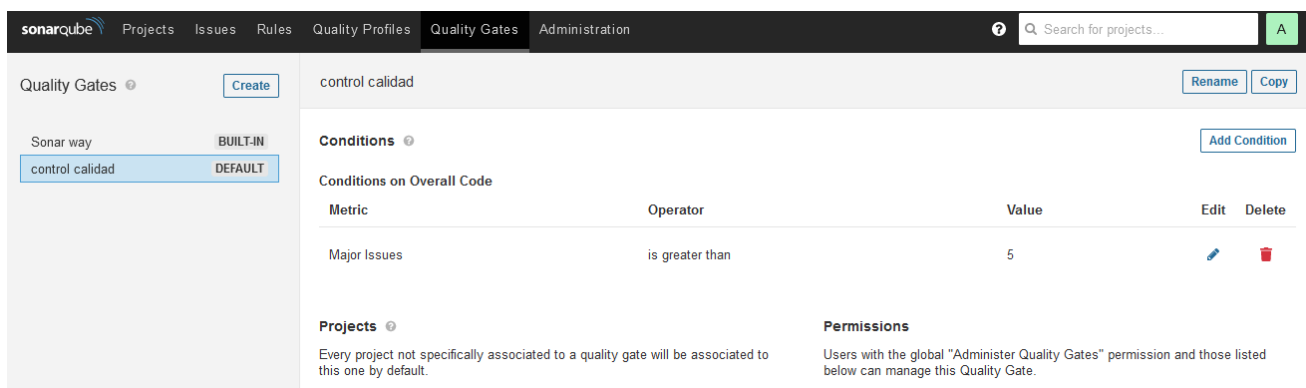
is greater than

Value

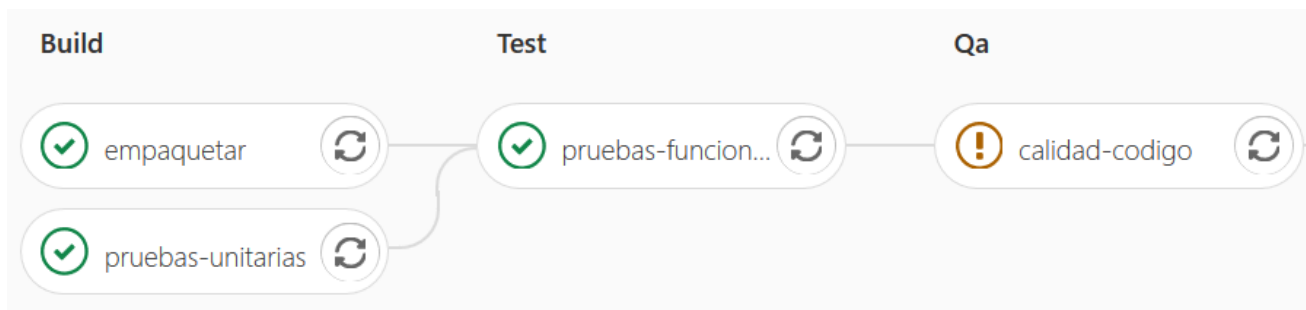
5

Add Condition Cancel

Después hay que entrar en la ventana de la condición "control calidad" y activar el botón "Set as default", para que se aplique a todos los proyectos.



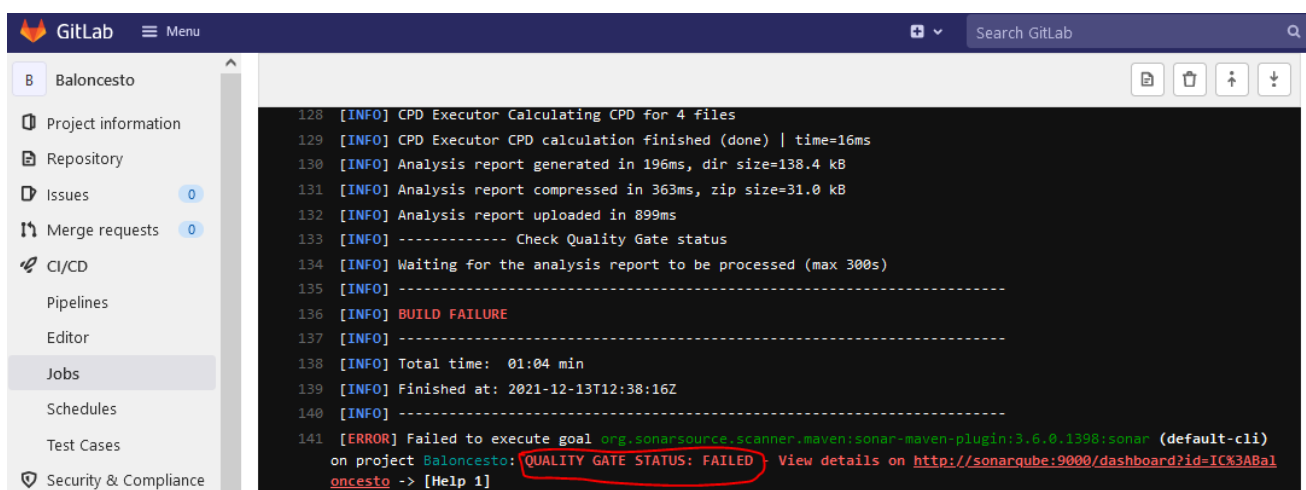
Si volvemos al pipeline, y seleccionamos que se repita (flechas en círculo junto al nombre del trabajo para refrescar), veremos que en este caso, también pasa pero con un aviso:



No se aborta el pipeline porque en gitlab-ci.yml hemos indicado `allow_failure: true`.

Esto es habitual, que la calidad de código sea importante y se revise, pero que no bloquee el proceso de entrega de la aplicación, es decir, que se intente mejorar en siguientes versiones, pero que no detenga la integración o el despliegue de la aplicación.

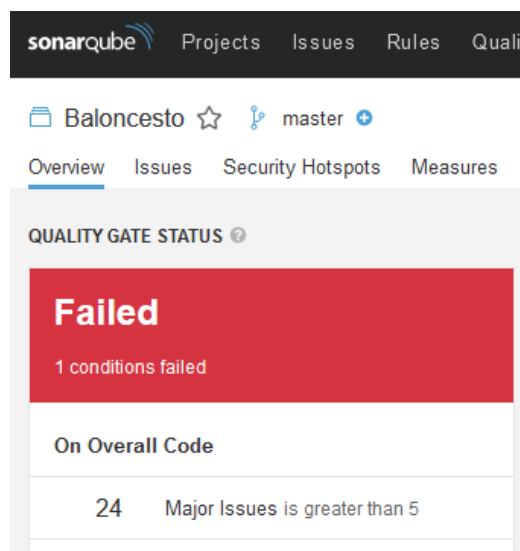
En la ventana del trabajo nos dice que ha habido un error:



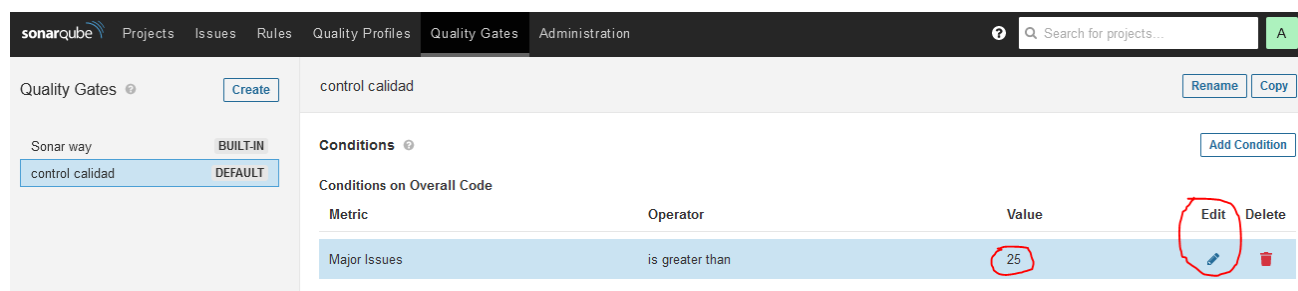
Y que el detalle está en:

<http://sonarqube:9000/dashboard?id=IC%3ABaloncesto>

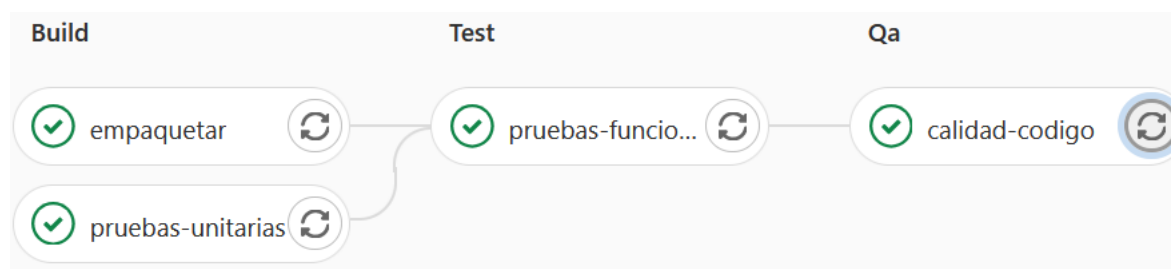
Como se ha dicho antes, esa URL es interna en el contenedor ubuntu-ci. Desde fuera del contenedor sería <http://localhost:9000/dashboard?id=IC%3ABalconcesto>. Si accedemos con usuario y contraseña, vemos que en efecto ha habido 24 errores de calidad importantes, que es mayor que el límite de 5 que habíamos definido.



Si en el servidor sonarqube aumentamos el número de fallos de calidad importantes permitidos a 25 en la condición de la Quality Gate que habíamos creado llamada “control calidad”:



Y refrescamos de nuevo el trabajo calidad-código en el pipeline, deja de aparecer el aviso de no superación del trabajo ya que el nivel de exigencia ahora es menor, ya que permitimos hasta 25 fallos de calidad importantes.



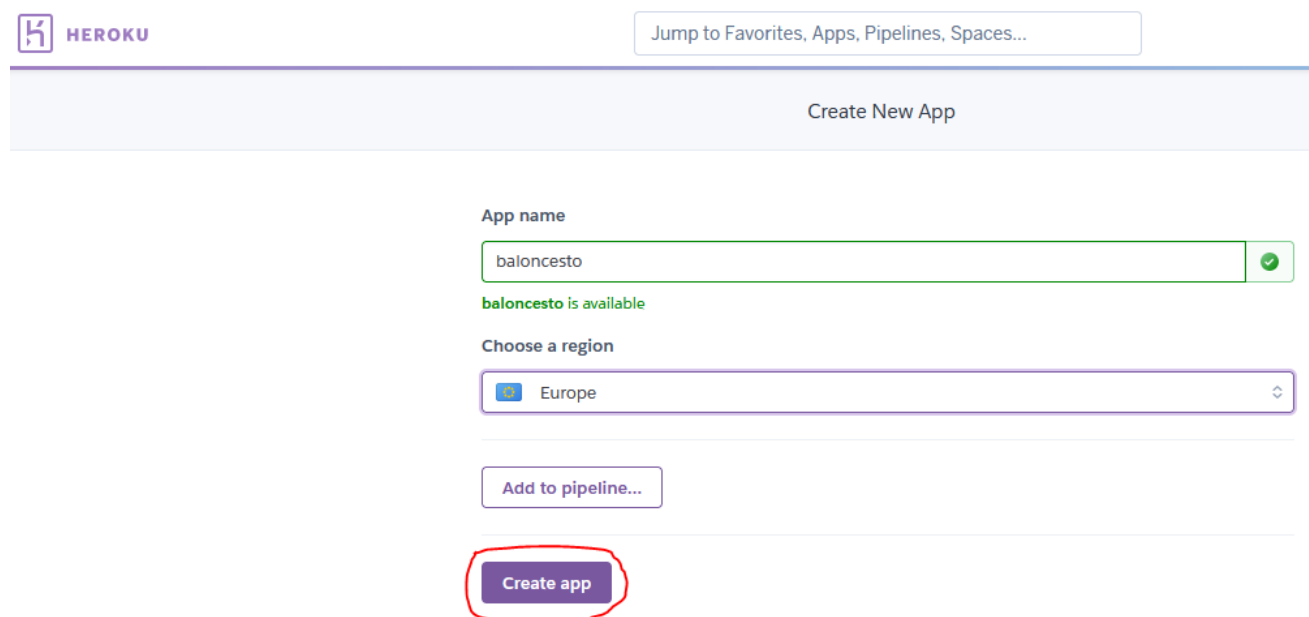
3.5 Fase deploy: Trabajo despliegue-host-heroku

Se utilizará el servicio de hosting gratuito Heroku, que permite instalar hasta 3 aplicaciones de forma gratuita.

Una vez identificados, hay que crear una aplicación vacía en:

<https://dashboard.heroku.com/apps>

Seleccionando New > Create new app. Con el nombre baloncesto.



HEROKU

Jump to Favorites, Apps, Pipelines, Spaces...

Create New App

App name

baloncesto

baloncesto is available

Choose a region

Europe

Add to pipeline...

Create app

En el archivo .gitlab-ci.yml añadimos los comandos para el despliegue usando maven.

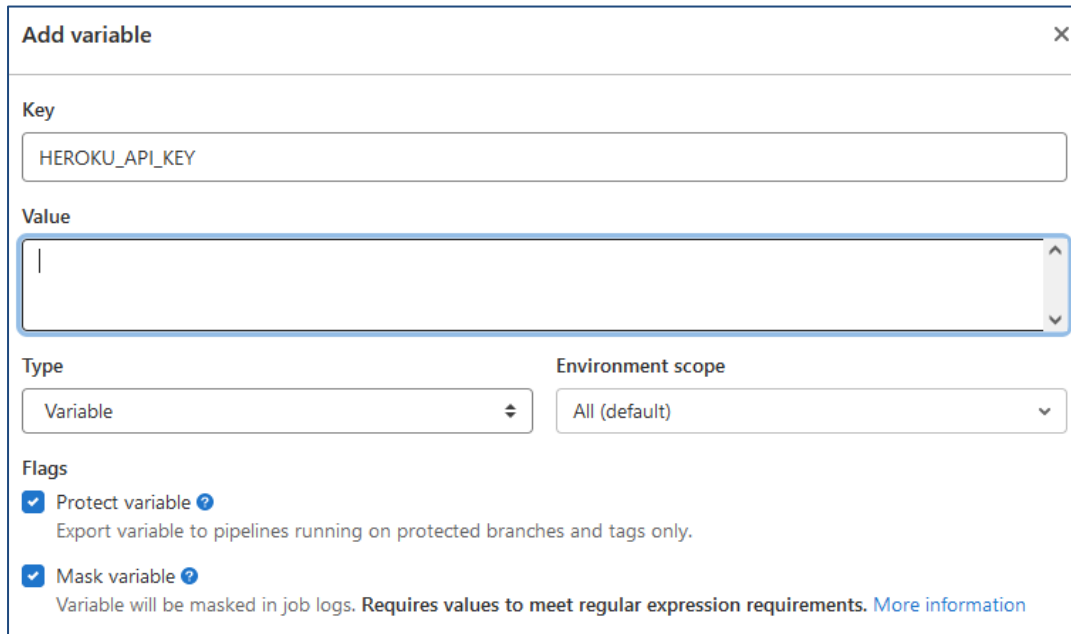
Archivo C:\.gitlab-ci.yml
<pre>despliegue-host-heroku: stage: deploy dependencies: - empaquetar script: - export HEROKU_API_KEY=\$HEROKU_API_KEY - mvn heroku:deploy-war -Dheroku.appName=baloncesto -DskipTests=true when: manual only: - master</pre>

Para desplegar en Heroku debemos utilizar una clave (API Key) que podemos encontrar en los ajustes de nuestra cuenta de Heroku:

Account settings > API Key > Reveal

Para evitar que se vea esa clave, en el archivo .gitlab-ci.yml hacemos referencia a una variable de gitlab.com, que permite crear variables cuyo valor es oculto.

Para crear esa variable, en gitlab debemos acceder a Settings > CI/CD > Variables > Expand > Add Variable. Y creamos una con el nombre HEROKU_API_KEY y como valor el de la API Key que tenemos en Heroku.



Add variable [X]

Key
HEROKU_API_KEY

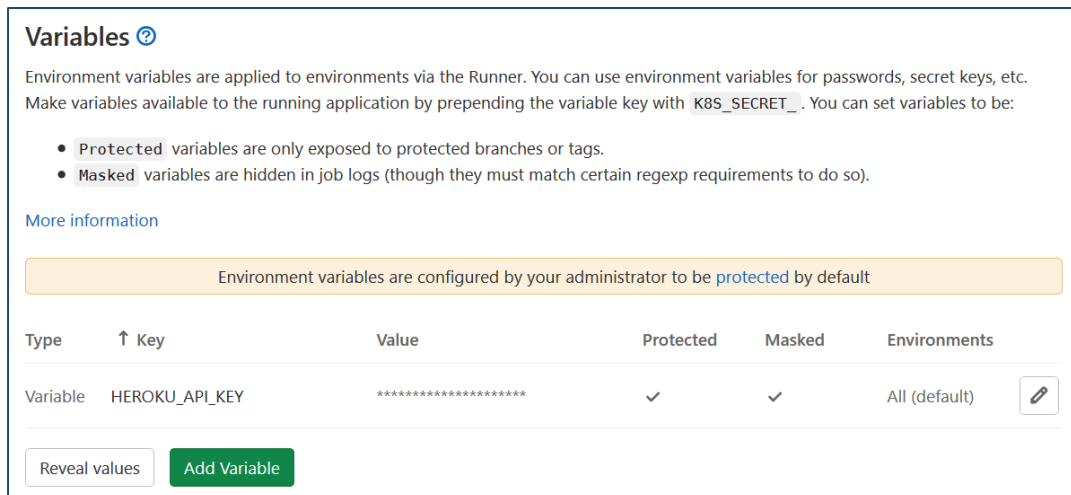
Value
[Empty text area]

Type: Variable [Dropdown]
Environment scope: All (default) [Dropdown]

Flags

- ☒ Protect variable ⓘ
Export variable to pipelines running on protected branches and tags only.
- ☒ Mask variable ⓘ
Variable will be masked in job logs. Requires values to meet regular expression requirements. [More information](#)

Pulsamos “Add variable” y aparece en la lista de variables del proyecto:




Variables ⓘ

Environment variables are applied to environments via the Runner. You can use environment variables for passwords, secret keys, etc. Make variables available to the running application by prepending the variable key with `K8S_SECRET_`. You can set variables to be:

- Protected** variables are only exposed to protected branches or tags.
- Masked** variables are hidden in job logs (though they must match certain regexp requirements to do so).

[More information](#)

Environment variables are configured by your administrator to be **protected** by default

Type	↑ Key	Value	Protected	Masked	Environments	
Variable	HEROKU_API_KEY	*****	✓	✓	All (default)	

[Reveal values](#) [Add Variable](#)

En el archivo .gitlab-ci.yml podemos hacer referencia a las variables con el símbolo \$.

Con el primer comando se crea en el contenedor ubuntu-ci la variable de entorno que necesita Heroku para autorizar el despliegue.

```
export HEROKU_API_KEY=$HEROKU_API_KEY
```

Con el segundo comando se hace el despliegue, siguiendo las indicaciones de <https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin>.

Para que funcione, hay que añadir dos plugins al archivo pom.xml del proyecto:

Archivo C:\pom.xml
<pre><plugin> <groupId>com.heroku.sdk</groupId> <artifactId>heroku-maven-plugin</artifactId> <version>3.0.3</version> </plugin> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-dependency-plugin</artifactId> <executions> <execution> <phase>package</phase> <goals> <goal>copy</goal> </goals> <configuration> <artifactItems> <artifactItem> <groupId>com.heroku</groupId> <artifactId>webapp-runner</artifactId> <version>9.0.30.0</version> <destFileName>webapp-runner.jar</destFileName> </artifactItem> </artifactItems> </configuration> </execution> </executions> </plugin></pre>

Se ha indicado en el archivo .gitlab-ci.yml que este trabajo sólo se ejecutará cuando manualmente el responsable active el trabajo, que aparece en gitlab.com con un icono en forma de flecha, para arrancar la ejecución. Al ser el despliegue a producción, suele hacerse de esta forma.

Si se quitara la orden manual y se dejara automático como el resto de trabajos, se podría decir que se está haciendo **despliegue continuo**.

Se guarda en el repositorio local y se hace push al remoto, y se comprueba que se despliega correctamente:

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Creado trabajo despliegue-host-heroku"
C:\Baloncesto> git push
```

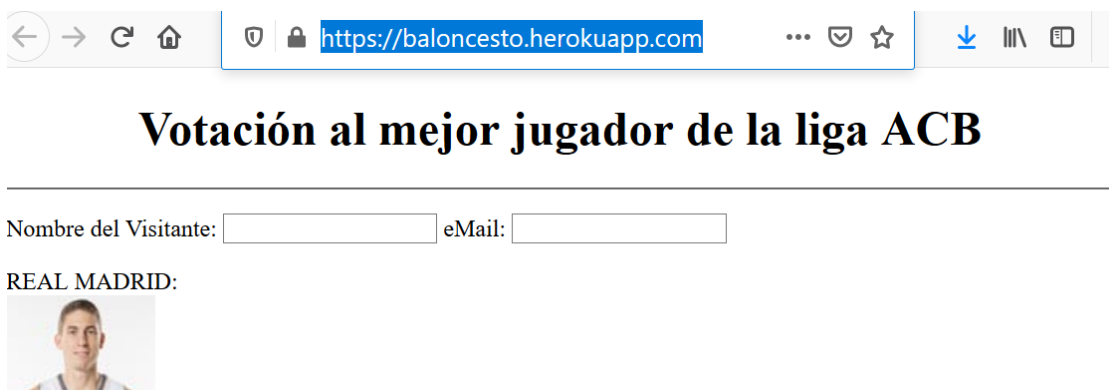
Cuando se han ejecutado todos los demás trabajos, como el trabajo de despliegue es manual, debemos seleccionar la flecha que aparece junto al nombre del trabajo para que se ejecute.



Si toda ha ido bien, aparece que le trabajo ha sido un éxito.



Si abrimos la aplicación en Heroku en la URL que corresponda a nuestra aplicación, en este ejemplo en <https://baloncesto.herokuapp.com/>, vemos que funciona:



NOTA: Para simplificar la práctica no se ha instalado en Heroku la base de datos. Se sugiere que el alumno que esté interesado en utilizar en el futuro Heroku como hosting gratuito, investigue cómo [instalar una base de datos MySQL en Heroku](#).

4. Realizar una segunda versión de la aplicación

4.1 Planificación ágil: crear historias de usuario (issues)

Se puede gestionar un proyecto ágil basado en Scrum con GitLab, teniendo en cuenta que la terminología no coincide exactamente con la que se maneja en Scrum. En esta tabla se muestra la equivalencia:

Agile (Scrum)	GitLab
User story	Issue
Sprint	Milestone
Story point	Weight
Product backlog	Issue list
Burndown chart	Burndown chart
Board	Board
Epic	Epic (versión pago)

En el proyecto se va a crear un nuevo sprint (milestone) llamado “Aplicación con estilos” para la segunda versión de la aplicación, a la que se añadirán estilos css que no había en la primera versión. Se van a crear 2 historias de usuario (issues), con su correspondiente estimación: 10 puntos por historia. El peso total previsto del sprint es por tanto de 20 puntos.

Los issues son los siguientes:

- Añadir estilos a la página principal: Se trata de poner color de fondo azul y color de texto blanco en la página index.html.
- Añadir estilos a la página de resultado de votación: Se trata de poner color de fondo rojo y color de texto amarillo en la página TablaVotos.jsp.

Para poder crear un milestone hay que tener categoría de Developer o Maintainer.

Para crear el milestone, en la web del proyecto en gitlab.com, en el menú principal de la izquierda hay que acceder a Issues > Milestones > botón “New milestone” > Title = “Aplicación con estilos” > Description = “Se van a añadir estilos a todas las páginas de la aplicación”.

Se puede fijar una fecha de inicio y otra de final: ponemos la fecha de hoy como inicio y como final una semana después. Y pulsamos Create milestone.

En la sección Issues > Milestones del menú principal puede verse la lista de milestones del proyecto:

Si seleccionamos el milestone creado, se abre una ventana de detalle con los diagramas burndown y burnup:



Ahora hay que crear los dos issues que forman el milestone, para ello se selecciona la opción Issues en el menú de la izquierda de Gitlab. Y en la pantalla el botón New issue. Aparece un formulario de

creación del issue, y ponemos como título “Añadir estilos a la página principal”. En la sección Assignee se selecciona Assign to me. En milestone “Aplicación con estilos”. Y en Weight el peso que representa el esfuerzo previsto para el issue, por ejemplo 10 puntos. Y al final se pulsa Create issue.

José R. Hilera > Baloncesto > Issues > New

New Issue

Title

Add [description templates](#) to help your contributors to communicate effectively!

Type ?

Description

Write Preview

B **I** **”** **</>**

Se trata de poner color de fondo azul y color de texto blanco en la página index.html.

[Markdown](#) and [quick actions](#) are supported

☐ This issue is confidential and should only be visible to team members with at least Reporter access.

Assignees

Weight

Milestone





Due date


Se procede de la misma forma para crear el otro issue con el título “Añadir estilos a la página de resultado de votación”.

En la página de Issues > List está la lista de issues del proyecto. Si aplicamos Scrum, representaría el product backlog o pila de producto, siendo los issues las historias de usuario. En este caso como todas las historias son del mismo sprint, representaría realmente la pila del sprint.

José R. Hilera > Baloncesto > Issues

Open 2 Closed 0 All 2


    Edit issues New issue



Añadir estilos a la página principal

#1 · created 1 minute ago by José R. Hilera · Aplicación con estilos · 10


updated 1 minute ago

 0

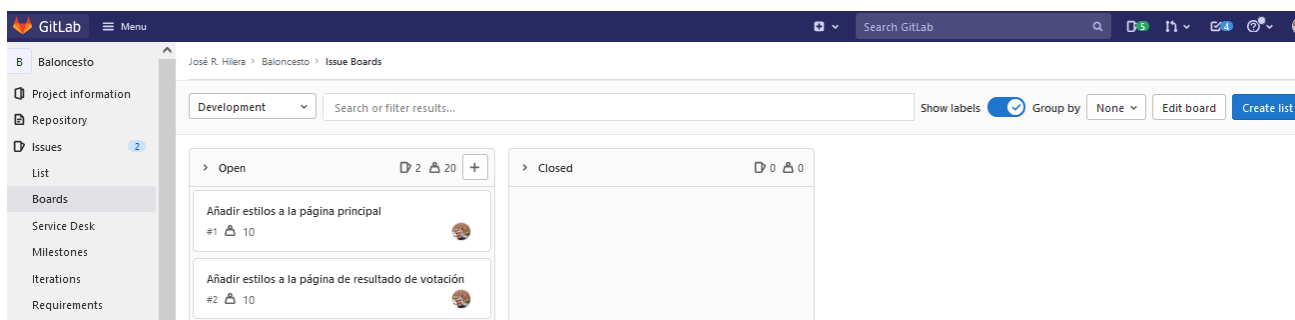
Añadir estilos a la página de resultado de votación

#2 · created just now by José R. Hilera · Aplicación con estilos · 10

updated just now

 0

En la sección Issues > Boards se puede ver el tablero Kaban del proyecto.

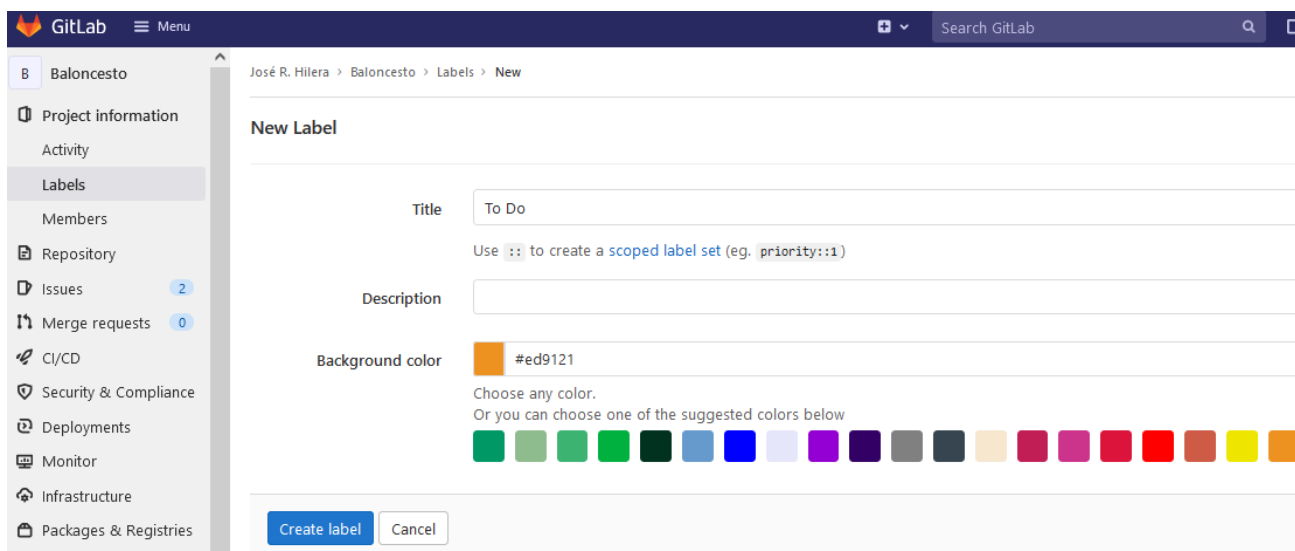


En el tablero que existe por defecto, que se llama “Development” sólo hay dos columnas: Open y Closed. Inicialmente las dos historias de usuario o issues están en la columna Open (pila de producto).

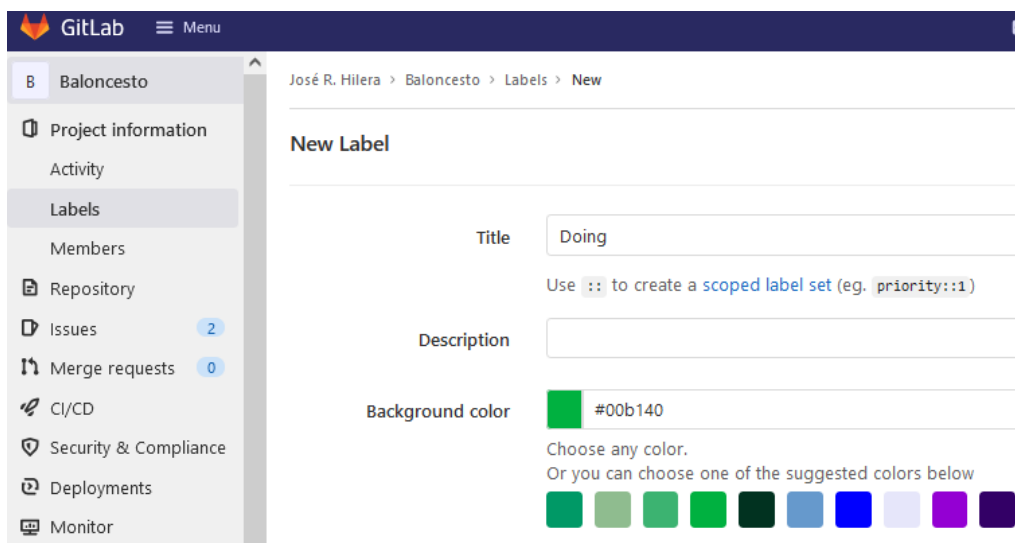
Normalmente en los proyectos reales se utilizan más columnas. Vamos a crear dos nuevas columnas entre las existentes, que se denominen “To Do” y “Doing” para que la primera ellas actúe como pila del sprint con los issues que hay que implementar en el spring, y la segunda con los que se están implementando el cada momento.

Antes de crear las columnas, hay que crear dos etiquetas (labels) en el proyecto con el nombre “To Do” y “Doing”, ya que al crear una columna nos pedirá como nombre una etiqueta de las que existen en el proyecto.

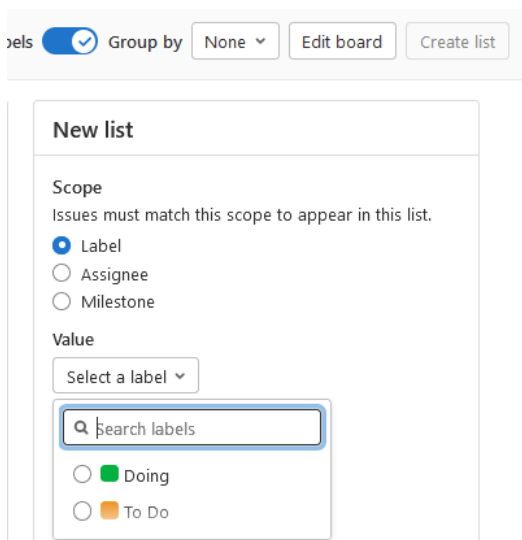
Las etiquetas se crean desde la sección Project information > Labels > New label > Title = “To Do”. Elijiendo un color, por ejemplo naranja.



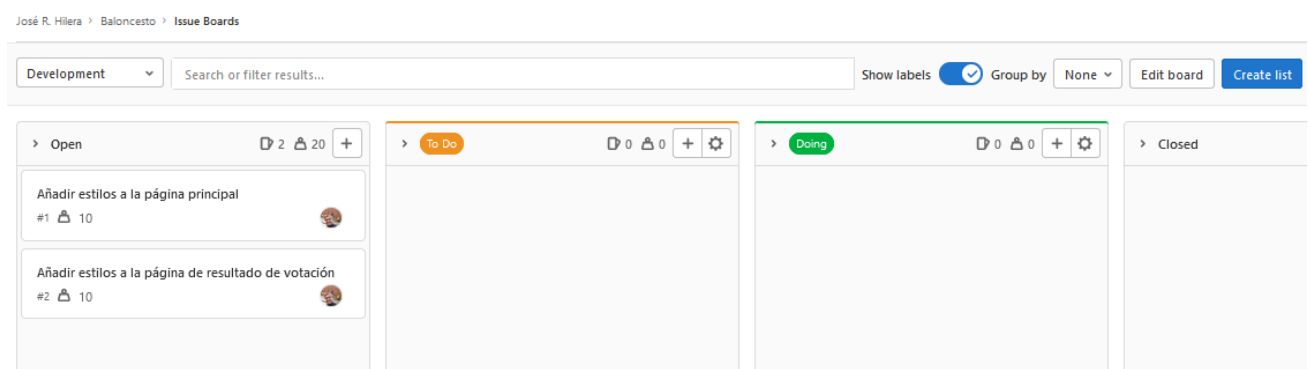
La etiqueta “Doing” se crearía de la misma forma, eligiendo otro color, por ejemplo verde.



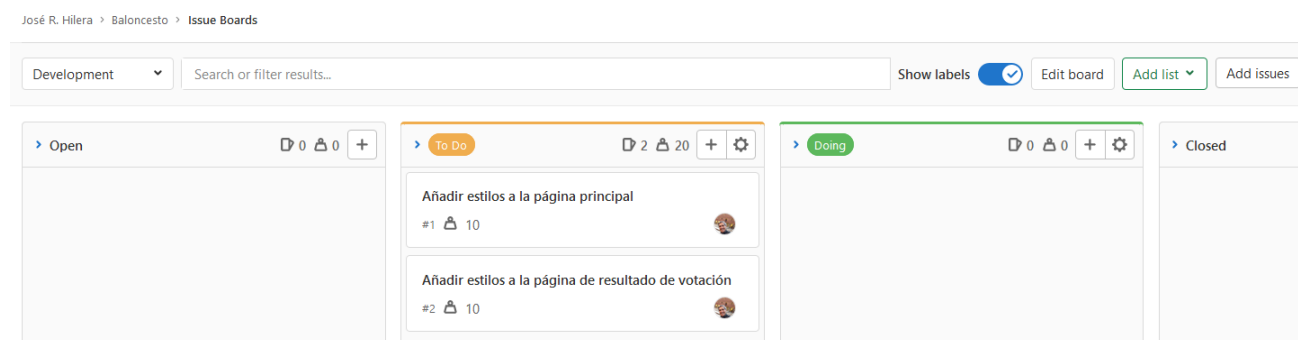
Para crear la columna “To Do” hay que volver a Issues > Boards, y seleccionar botón Create list > Select a label > Elegir la etiqueta “To Do” > Add to board.



Y lo mismo para crear la otra columna “Doing”. Así ahora en el tablero hay cuatro columnas o listas:

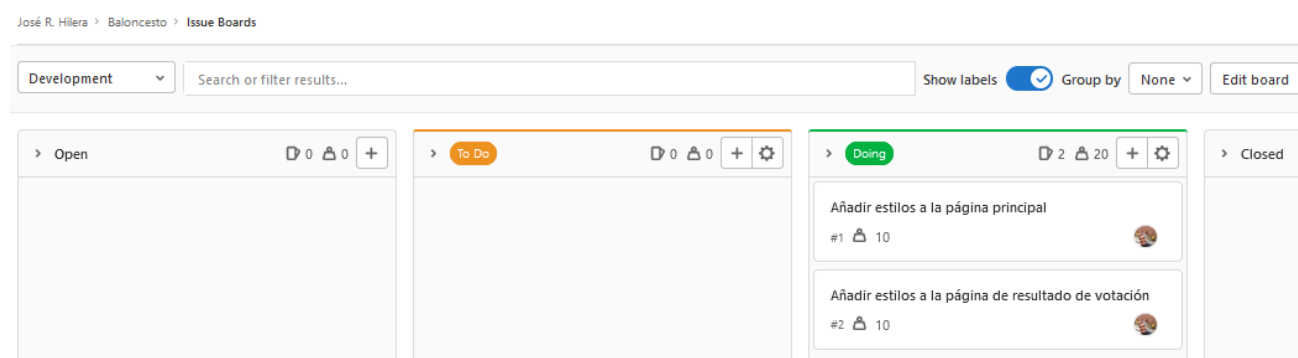


Como se van a implementar los dos issues en el sprint, se pueden arrastrar y soltar en la columna To Do, que puede considerarse la pila del sprint.



Según vaya a comenzar en desarrollo de cada issue, hay que mover la ficha correspondiente a la columna Doing. Es importante hacerlo cuando de verdad comience, para que el diagrama burndown y las métricas de seguimiento reflejen la evolución real del proyecto.

Para simular un conflicto en el desarrollo porque se hagan cambios en el código fuente del proyecto incompatibles entre sí, vamos a suponer que los dos issues comienzan al mismo tiempo en este momento, y los movemos a la columna Doing.



Vamos a simular ahora cómo trabajarían dos desarrolladores diferentes con el mismo repositorio compartido. Al haber sólo un desarrollador, lo que se hará a continuación puede parecer que no tiene mucho sentido. Si se prefiere, el alumno puede crear otro usuario de Gitlab con otro email e invitarle a ser miembro del proyecto desde la opción: Members > Invite member, asignándole el rol Developer. Y asignar uno de los issues al nuevo miembro, desde la ficha del issue, sección Assignee.

Si suponemos que solo hay un desarrollador, será quien se encargue de los dos issues, pero los va a desarrollar simultáneamente.

4.2 Crear una rama con git para cada issue (feature)

Cada historia de usuario o issue implica añadir una nueva característica o feature al código de la aplicación que se está desarrollando. Existen varias políticas de gestión de ramas. En este proyecto seguiremos la más sencilla, que consiste y es crear una rama partiendo de la rama máster para cada nueva feature, y cuando esté terminada, integrar (merge) los cambios de esa rama en la rama máster.

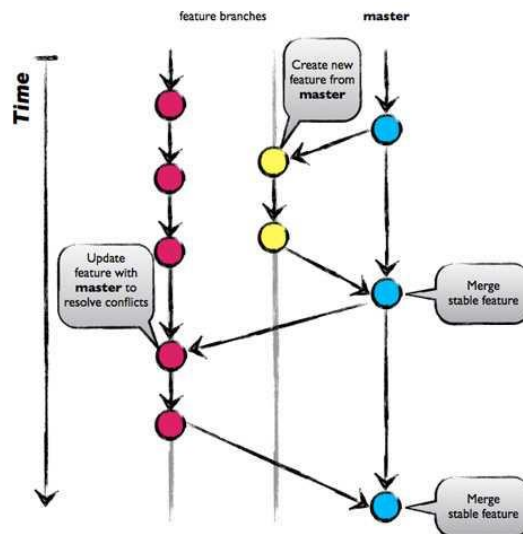


Imagen: [nicoespeon](#)

Hay que trabajar en las dos features de los dos issues, que llamaremos:

- añadir-estilos-pagina-principal
- añadir-estilos-pagina-resultado

Desde nuestro ordenador local como desarrollador encargado de la primera feature, crearemos una rama para la misma con el comando siguiente:

```
C:\Baloncesto> git branch añadir-estilos-pagina-principal
```

Pero como también vamos a desarrollar la otra issue creamos la otra rama:

```
C:\Baloncesto> git branch añadir-estilos-pagina-resultado
```

Tenemos por tanto tres ramas. Con el comando `git branch` puede comprobarse, indicando con `*` en la que nos encontramos trabajando:

```
C:\Baloncesto> git branch
  añadir-estilos-pagina-principal
  añadir-estilos-pagina-resultado
* master
```

Si usamos Visual Studio Code como editor, podemos instalar la extensión Git Graph para ver las ramas: <https://marketplace.visualstudio.com/items?itemName=mhutchie.git-graph>

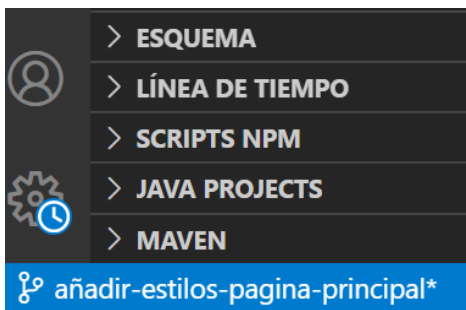
4.2.1 Programar issue “Añadir estilo a la página principal”

Vamos a pasarnos a la primera rama creada para que el desarrollador trabaje en la nueva característica (feature) de añadir estilos a la página principal. Para lo cual hay que cambiarse a dicha rama con el comando:

```
C:\Baloncesto> git checkout añadir-estilos-pagina-principal
Switched to branch 'añadir-estilos-pagina-principal'
```

Para saber en qué rama estamos podemos ejecutar `git branch`, se muestra con `*` la rama actual.

Si trabajamos con Visual Studio Code, en la parte inferior izquierda siempre se indica la rama en la que trabajamos, en este caso estamos en `añadir-estilos-pagina-principal`:



Estando en esa rama, vamos a crear un archivo `estilos.css` en la carpeta `C:\Baloncesto\src\main\webapp\`.

Archivo C:\Baloncesto\src\main\webapp\estilos.css
<pre>body { color: white; background-color: darkblue; }</pre>

Y modificamos el archivo `index.html` de la misma carpeta, haciendo referencia al archivo de estilos desde la sección `<head>`, para que la página principal tenga fondo azul y texto blanco.

Archivo C:\Baloncesto\src\main\webapp\index.html
<pre><head> <title>Votacion mejor jugador liga ACB</title> <link href="estilos.css" rel="stylesheet" type="text/css" /> </head></pre>

Con esto terminan los cambios en la rama `añadir-estilos-pagina-principal` y se deben consolidar en el repositorio local antes de subirlos al repositorio remoto.

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Añadido estilo en página principal"
```

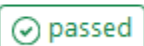






Cuando se suban los cambios al repositorio compartido en GitLab, se lanzarán las pruebas y si todo ha ido bien, se podrán integrar los cambios en la rama `máster` del repositorio compartido y a continuación pasar el issue del tablero del proyecto de la columna `Doing` a la columna `Closed`.

Para subir los cambios locales al repositorio compartido ejecutamos:

```
C:\Baloncesto> git push origin añadir-estilos-pagina-principal
```


En el servidor de integración continua Gitlab se crea la rama añadir-estilos-pagina-principal con el código fuente modificado, y se lanza la ejecución para esa rama de los trabajos de todas las fases del pipeline del proyecto menos deploy, ya que en el archivo gitlab-ci.yml se indicaba que el trabajo de la fase deploy llamado despliegue-host-heroku sólo se tenía que ejecutar cuando hubiera cambios en la rama máster.


Se puede consultar el resultado en la sección CI/CD > pipelines



Status	Pipeline ID	Triggerer	Commit	Stages
 passed	#428444939 latest		 añadir-esti... f5cd1341  Añadido estilo en página...	  


Si se selecciona la pipeline se ve el detalle:

Añadido estilo en página principal

 4 jobs for [añadir-estilos-pagina-principal](#) in 3 minutes and 59 seconds (queued for 45 seconds)



 latest



 [f5cd1341](#) 

 No related merge requests found.



Pipeline Needs Jobs 4 Tests 0

Build



 empaquetar 

 pruebas-unitarias 

Test

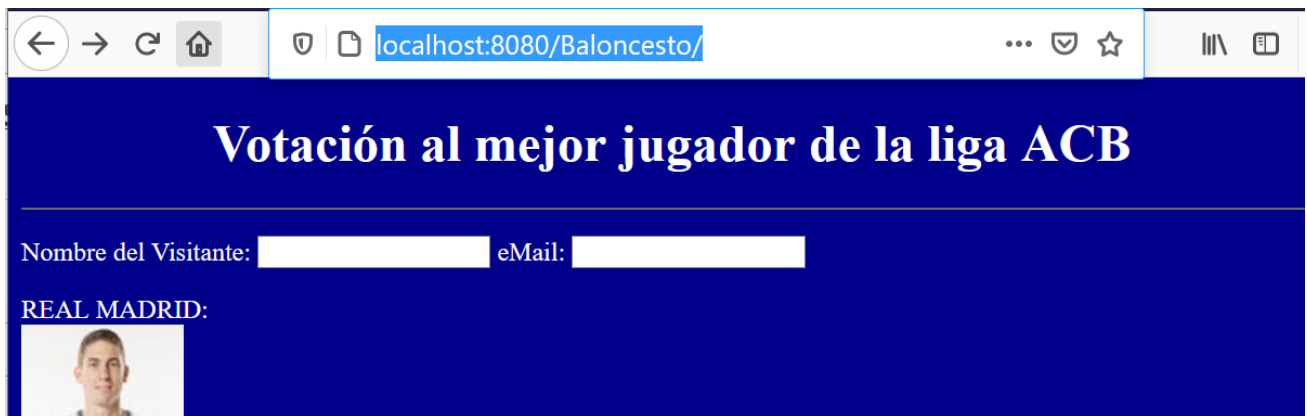
 pruebas-funcionales 

Qa

 calidad-codigo 

No aparece el trabajo despliegue-host-heroku porque en el archivo .gitlab-ci.yml indicamos que este trabajo sólo se ejecutase cuando hubiera cambios en la rama master.

Como en el trabajo pruebas-funcionales de la fase Test se despliega aplicación modificada en el servidor web Tomcat del contenedor ubuntu-ci que se utiliza para las pruebas, podemos comprobar que ha cambiado el color de la página principal, accediendo a la aplicación en el servidor web de pruebas: <http://localhost:8080/Baloncesto/>.



Una vez pasado el pipeline con éxito, se puede proceder a integrar los cambios en la rama máster.

Para ello hay que crear una solicitud de integración (merge request) desde la sección del menú principal izquierdo de Gitlab llamada Merge Request > botón Create merge request.

Se trata de hacer un merge (integración) desde la rama añadir-estilos-pagina-principal a la rama master.

En la pantalla de creación del merge request se puede dejar el nombre por defecto. Se puede seleccionar el milestone al que pertenece y la persona del proyecto que lo tiene asignado o dejarlo en blanco. Se pueden definir diferentes reglas de aprobación, en este caso lo dejamos en blanco.

Desmarcar “Delete source branch when merge request is accepted”, para poder revisar el historial de ramas en el futuro.

New merge request

From `añadir-estilos-pagina-principal` into `master` [Change branches](#)

Title

[Start the title with `Draft:`](#) to prevent a merge request draft from merging to `main`.
Add [description templates](#) to help your contributors to communicate effectively.

Description

Write Preview

Describe the goal of the changes and what reviewers should be aware of.

[Markdown](#) and [quick actions](#) are supported

Assignees

Reviewers

Approvals are optional.

[Approval rules](#)

Milestone

Se debe desmarcar la opción “Delete source branch when merge request is accepted”.

Merge options ☒ Delete source branch when merge request is accepted.
☐ Squash commits when merge request is accepted. [?](#)


Create merge request

Cancel

Y se pulsa el botón Create merge request.

Añadido estilo en página principal

Overview 0 Commits 1 Pipelines 1 Changes 2


 Request to merge añadir-estilos-pagina... into master

Open in Web IDE

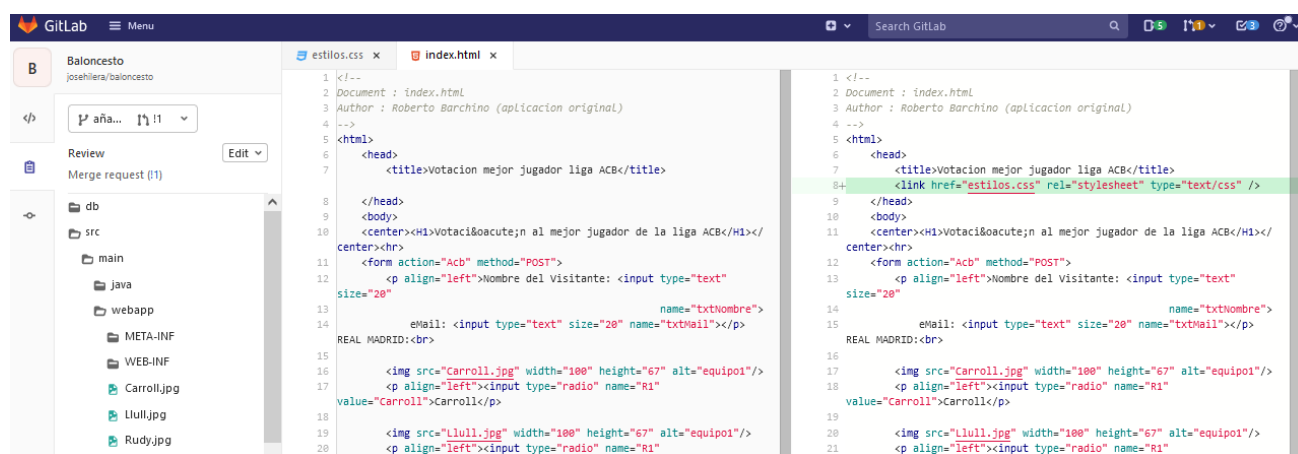
 Pipeline #428444939 passed for f5cd1341 on añadir-estilos-pagina... 6 minutes ago

 Approval is optional

> [View eligible approvers](#)



El miembro del equipo que esté designado para aceptar los cambios (por ejemplo el product owner o quien haya decidido el equipo) puede pulsar el botón “Open in Web IDE” para ver los cambios en los archivos del repositorio. En este caso, se comprueba que el archivo index.html tiene una línea nueva, para incluir la hoja de estilos.



El responsable puede editar los archivos si decide que alguno de los cambios no es aceptable, o bien aceptarlos y proceder a fusionarlos en la rama master, pulsando el botón Merge.

Si no está activado el botón Merge, debe seleccionar primero “Ready to merge”, y entonces quedará activo.

Seleccionando Merge, los cambios se integran en la rama master, y se dispara la ejecución de todos los trabajos del pipeline definido en gitlab-ci.yml, pero ahora para la rama máster, por lo que se ejecutan todas las fases. En el caso del despliegue al servidor público Heroku, se indicó que fuera manual, por lo que requiere que se active manualmente.

✓ passed Pipeline #428467773 triggered 11 minutes ago by José R. Hilera

Merge branch 'añadir-estilos-pagina-principal' into 'master'

Añadido estilo en página principal

See merge request !1

🕒 5 jobs for master in 3 minutes and 59 seconds (queued for 52 seconds)

📄 latest

🔗 fb48d602

🔍 No related merge requests found.

Pipeline Needs Jobs 5 Tests 0

Build

✓ empaquetar

✓ pruebas-unitarias

Test

✓ pruebas-funcionales

Qa

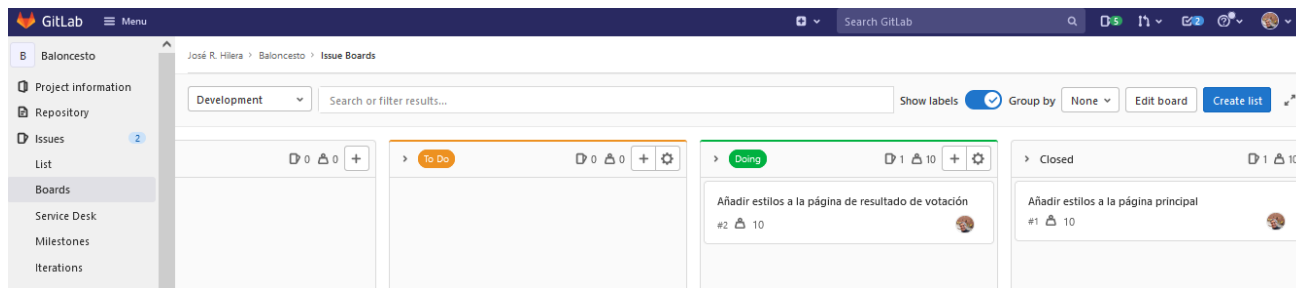
✓ calidad-codigo

Deploy

⚙️ despliegue-host-heroku

En este momento no tiene sentido desplegar a producción, pues todavía no ha finalizado el sprint, ya que falta implementar el otro issue para aplicar estilos a la otra página de la aplicación web.

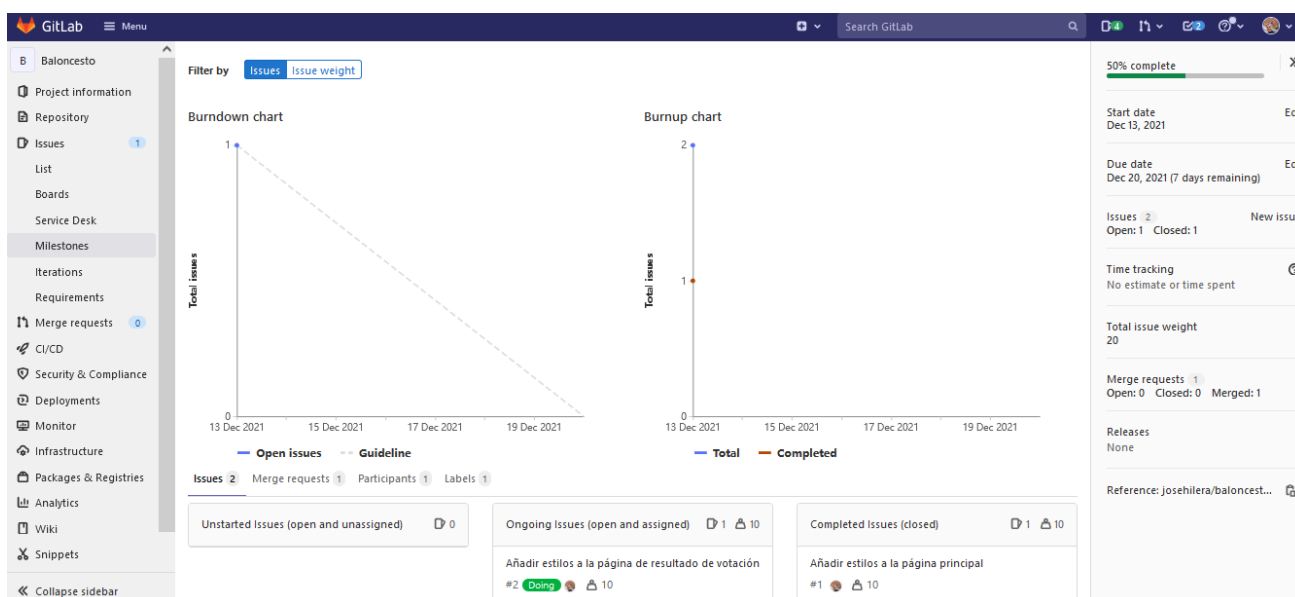
Como ha finalizado el issue, hay que proceder a cerrarlo en el tablero Kanban, arrastrándolo desde la columna Doing a la columna Closed.



Si accedemos al diagrama burndown en Issues > Milestones > Aplicación con estilos, vemos que se ha contabilizado la terminación del issue, y que sólo queda un issue pendiente para finalizar el sprint o milestone.

Si hemos empezado el proyecto el mismo día que hemos terminado el issue, no se puede apreciar en el diagrama, pues la apertura y cierre aparecen en la misma vertical.

Este diagrama se puede ver también con unidades de peso (weight).



Vemos que el peso total previsto del sprint era de 20 puntos (asignamos 10 puntos a cada historia de usuario o issue cuando las creamos), y que quedan 10 puntos para finalizar (los del issue pendiente).

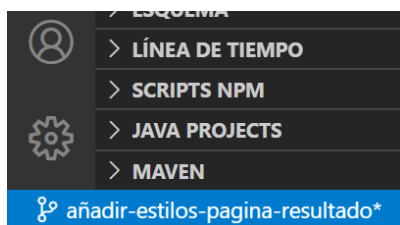
4.2.2 Programar issue “Añadir estilo a la página de resultado de votación” y resolver conflicto

La otra historia de usuario (issue) para el sprint (milestone) “Aplicación con estilos” era la titulada “Añadir estilos a la página de resultado de votación”, para la cual ya se había creado una rama llamada añadir-estilos-pagina-resultado.

Para trabajar en esa rama, hay que ejecutar el comando:

```
C:\Baloncesto> git checkout añadir-estilos-pagina-resultado
Switched to branch 'añadir-estilos-pagina-resultado'
```

Si se trabaja con Visual Studio Code, puede verificarse en la parte inferior izquierda, que estamos en esa rama:



También puede comprobarse que en esa rama no existe el archivo estilos.css, porque ese archivo se creó en la otra rama (añadir-estilos-pagina-principal).

En este caso el desarrollador encargado de añadir estilos en la tabla de resultado decide crear también un archivo estilos.css:

Archivo C:\Baloncesto\src\main\webapp\estilos.css
<pre>body { color: yellow; background-color: darkred; }</pre>

Y añadir la referencia a dicho archivo en la página de resultado de la votación TablaVotos.jsp.

Archivo C:\Baloncesto\src\main\webapp\TablaVotos.jsp
<pre><head> <title>Votaci&oacute;n mejor jugador liga ACB</title> <link href="estilos.css" rel="stylesheet" type="text/css" /> </head></pre>

Con esto terminan los cambios en la rama añadir-estilos-pagina-resultado y se deben consolidar en el repositorio local antes de subirlos al repositorio remoto.

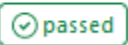






```
C:\Baloncesto> git add .
C:\Baloncesto> git status
C:\Baloncesto> git commit -m "Añadido estilo en página de resultado"
```

Una vez hecho, desde el ordenador local se envían los cambios a esa rama del repositorio compartido en Gitlab:

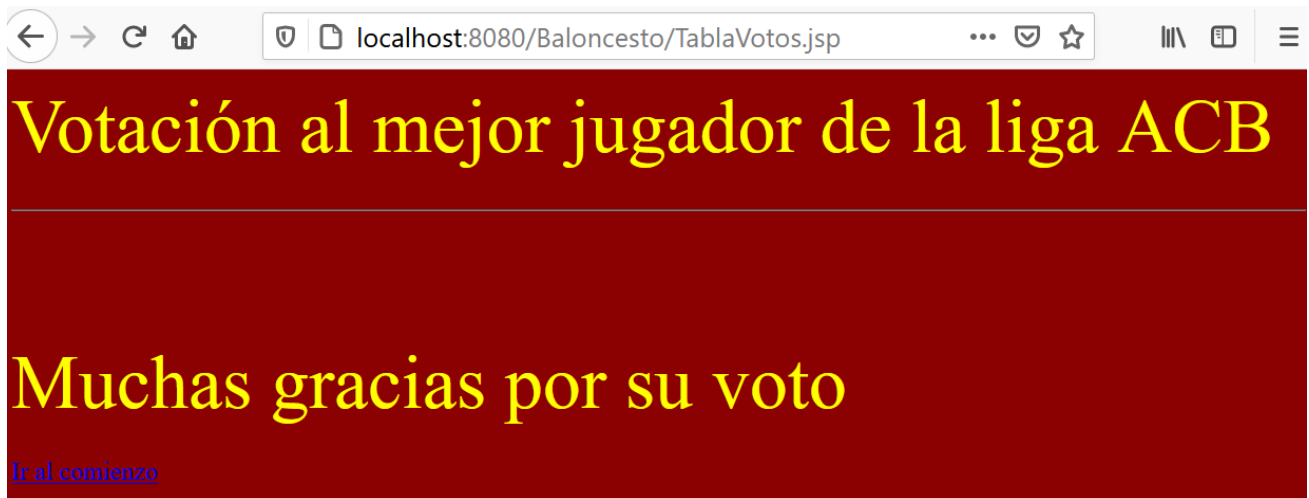
```
C:\Baloncesto> git push origin añadir-estilos-pagina-resultado
```

Se ha creado la rama añadir-estilos-pagina-resultado en el repositorio compartido y se ha lanzado para esa rama la ejecución de los trabajos de las fases build, test y qa del pipeline del proyecto.

Si todo ha ido bien, se indica en CI/CD > Pipelines:

Status	Pipeline ID	Triggerer	Commit	Stages
	#428493257 latest		 añadir-estil... f90f3fb6  Añadido estilo en página...	  

Se puede comprobar en el servidor web de pruebas que el estilo de la página de resultado de votación es el correcto: <http://localhost:8080/Baloncesto/TablaVotos.jsp>



Ahora es el momento de solicitar su integración en la rama máster del proyecto, desde Merge Request > botón Create merge request, como se hizo en el apartado anterior.

En la página de creación del merge request, se avisa de que hay conflictos y no se puede continuar hasta que se resuelvan.

Añadido estilo en página de resultado

Overview 0 Commits 1 Pipelines 1 Changes 2

Lo que ha ocurrido es que el desarrollador encargado de este segundo issue ha creado un archivo estilos.css definiendo un estilo para body. Pero el desarrollador del issue del apartado anterior también creó ese archivo con un estilo diferente para body. Este archivo ya está integrado en la rama master, pues se hizo en el apartado anterior.

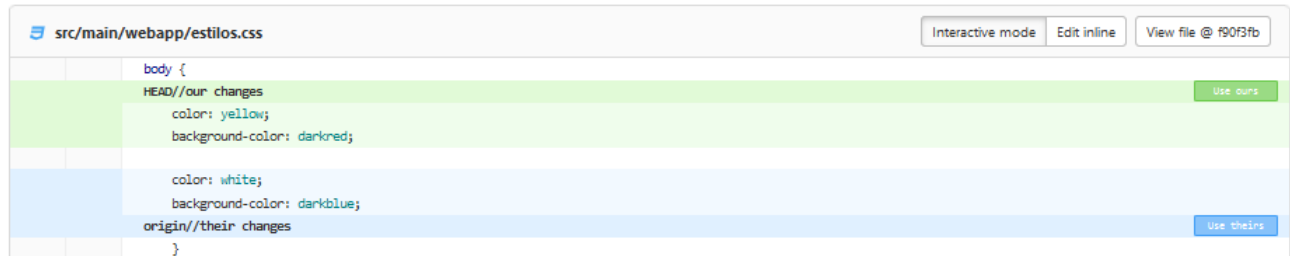
Es el desarrollador del nuevo issue el que debe hacer algo para solucionar el problema y poder integrar sus cambios.

En Gitlab se puede ver el detalle del conflicto pulsando el botón Resolve conflicts:

Añadido estilo en página de resultado

Showing 1 conflict between añadir-estilos-pagina-resultado and master

Inline Side-by-side



Los conflictos se pueden resolver en el servidor, con los botones “Use ours” o “Use theirs”, pero en general no es lo más conveniente. Es recomendable que los desarrolladores tengan en su ordenador local el último estado del repositorio compartido antes de enviar sus cambios. Por eso en la pantalla de Gitlab también hay un botón “Merge locally”, que al pulsarlo nos recomienda que refresquemos nuestro repositorio local, que resolvamos los conflictos y que volvamos a subir los cambios.



Pero es más fácil actualizar nuestro repositorio local usando el comando pull, que combina en un solo comando fetch y merge, de la siguiente forma:

```
C:\Baloncesto> git pull origin master
```

```
C:\Baloncesto>git pull origin master
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 288 bytes | 3.00 KiB/s, done.
From https://gitlab.com/josehilera/baloncesto
* branch          master      -> FETCH_HEAD
   e2a53a8..fb48d60 master     -> origin/master
CONFLICT (add/add): Merge conflict in src/main/webapp/estilos.css
Auto-merging src/main/webapp/estilos.css
Automatic merge failed; fix conflicts and then commit the result.
```

Nos avisa de que ha habido problemas. Si abrimos el archivo problemático, podemos ver los conflictos y borrar manualmente la sección que decidamos que hay que cambiar.

En el caso del archivo estilos.css, si lo abrimos con Visual Studio Code, nos ofrece aceptar el cambio entrante:

```
src > main > webapp > # estilos.css > background-color
1  body {
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
2  <<<<<< HEAD (Current Change)
3      color: yellow;
4      background-color: darkred;
5  =====
6      color: white;
7      background-color: darkblue;
8  >>>>>> fb48d60255b75480ccf17d136010f41bea11cb3e (Incoming Change)
9  }
```

Realmente nos interesan ambos estilos, pero hay que cambiar el nombre del nuevo estilo para que no haya conflictos. Aceptamos ambos cambios y cambiamos el nombre del estilo de texto amarillo y fondo rojo.

Por lo que el archivo final sería:

Archivo C:\Baloncesto\src\main\webapp\estilos.css
<pre>body { color: white; background-color: darkblue; } .resultado { color: yellow; background-color: darkred; }</pre>

Ahora hay que modificar el archivo TablaVotos.jsp para vincular el nuevo estilo “resultado” a la etiqueta body:

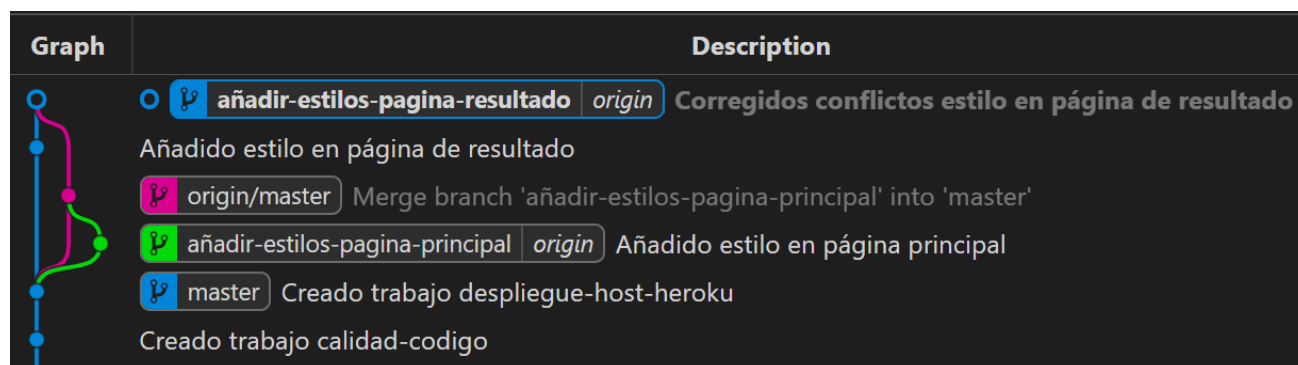
Archivo C:\Baloncesto\src\main\webapp\TablaVotos.jsp
<pre><html> <head> <title>Votaci&oacute;n mejor jugador liga ACB</title> <link href="estilos.css" rel="stylesheet" type="text/css" /> </head> <body class="resultado"> ... </pre>

Estos cambios se deben enviar al repositorio compartido.

```
C:\Baloncesto> git add .
C:\Baloncesto> git status
```

```
C:\Baloncesto> git commit -m "Corregidos conflictos estilo en página de resultado"
C:\Baloncesto> git push origin añadir-estilos-pagina-resultado
```

Si hemos instalado en Visual Studio Code la extensión Git Graph, podemos ver gráficamente cómo han evolucionado las ramas del proyecto. La palabra “origin” se refiere al repositorio remoto.



Si volvemos a Gitlab y refrescamos la pantalla del merge request, cuando se termina de ejecutar el pipeline correspondiente, ya aparece el botón “Merge” habilitado.

Añadido estilo en página de resultado

Overview 0 Commits 2 Pipelines 2 Changes 2

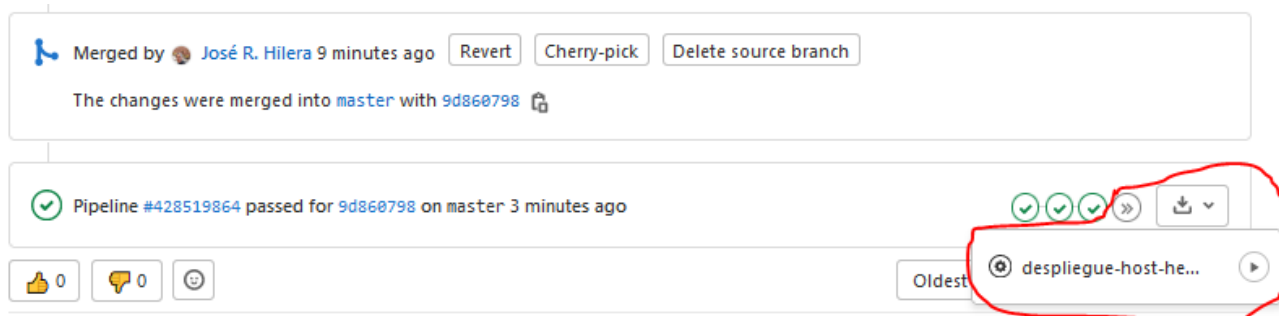
Request to merge **añadir-estilos-pagina...** into **master** Open in Web IDE

Pipeline #428511787 passed for 04541208 on añadir-estilos-pagina... 1 minute ago

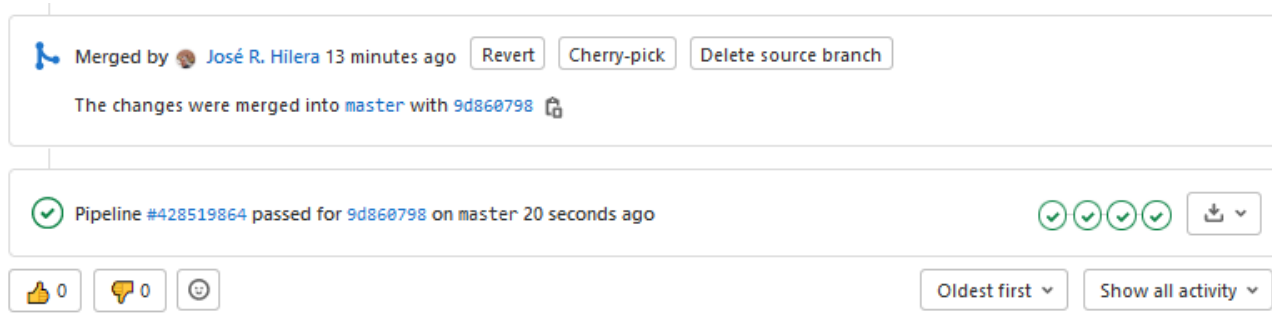
Approval is optional
[View eligible approvers](#)

Merge ☐ Delete source branch ☐ Squash commits

Si pulsamos el botón Merge se producirá la integración en la rama máster y se lanzará de nuevo el pipeline para la rama máster. En este caso se ejecutan los trabajos del pipeline, incluido el despliegue a producción (Heroku), que lo debemos activamos manualmente cuando el pipeline llegue a este último trabajo, ya que en el archivo gitlab-ci.yml se indicó que este trabajo era manual.



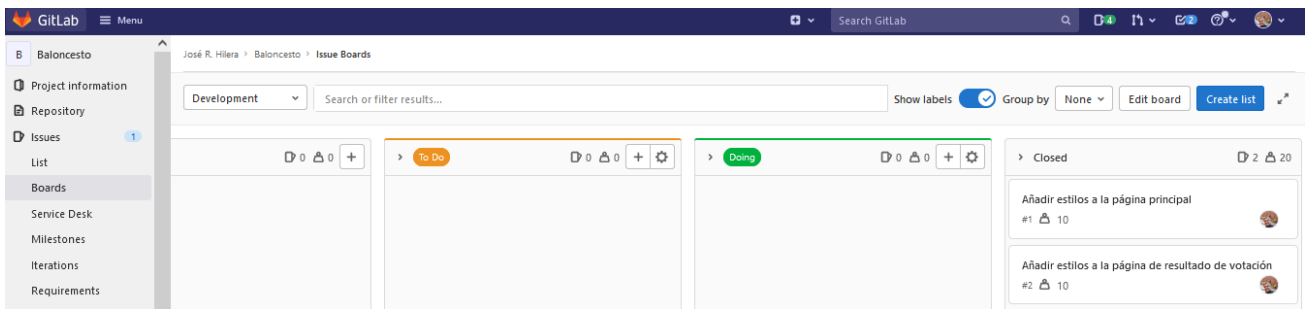
Cuando termina el despliegue, aparece en verde indicando que ha finalizado con éxito.



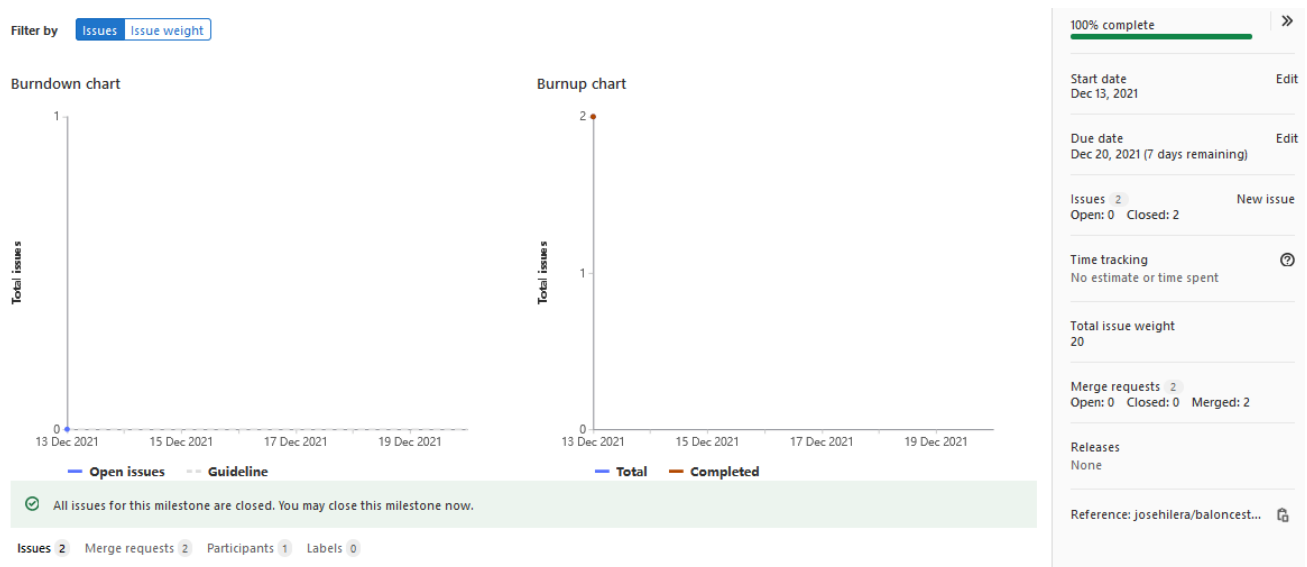
Si accedemos a <https://baloncesto.herokuapp.com>, vemos que los estilos de las dos páginas son correctos.



Con esto hemos terminado el issue “Añadir estilos a la página de resultado de votación”, por lo que debemos indicarlo en el tablero del proyecto, en Issues > Boards, arrastrando la ficha del issue de la columna Doing a la columna Closed.



Si accedemos al diagrama burndown en Issues > Milestones > Aplicación con estilos, vemos que se ha contabilizado la terminación del issue.



Debajo del diagrama se avisa de que todas las historias de usuario (issues) de este sprint (milestone) están terminadas y que podemos cerrar el sprint. Para ello vamos a la opción Milestones del menú principal de la izquierda y pulsamos el botón Close Milestone.

