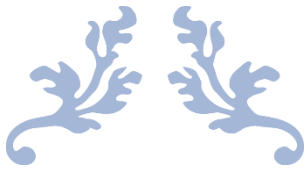


TRABAJO FINAL

Integración Continua en el Desarrollo Ágil

25 DE FEBRERO DE 2024

GONZALO ROMERO ESCRIBANO
Máster Universitario en Desarrollo Ágil de Software para la Web

Índice

Introducción 2
Cambios en el workflow del proyecto 3
Nuevo sprint 4
Programación de los issues 5
QA 5
REQ-1 5
PU 6
REQ-2 9
PF-A 13
PF-B 14
Conclusiones 17

Introducción

Este documento corresponde a la memoria de la práctica final de la asignatura Integración Continua en el Desarrollo Ágil para la Web, perteneciente al Máster Universitario en Desarrollo Ágil para la Web.

El objetivo de este trabajo es realizar un desarrollo ágil con integración continua sobre la aplicación web creada en la práctica guiada del tema 5 de la asignatura, para generar una segunda versión.

Antes de comenzar a trabajar sobre la aplicación, se realizó la práctica guiada de la sesión 5 de la asignatura al completo, de forma que se adquirieran los conceptos necesarios para trabajar con CI/CD mediante Github, así como realizar los despliegues sobre Azure.

Este proceso no está documentado, ya que simplemente se siguió el guion de la práctica. Se tuvieron algunos problemas al realizar esta, ya que se tuvo que volver a configurar el contenedor con Ubuntu varias veces. Esto fue debido que los puertos que se utilizan en la práctica para conectar con la BBDD son 3306:3306, pero, al tener en la máquina local ya un servidor mysql, Docker no permitía el uso del puerto 3306, por lo que se optó por utilizar 3307:3306 como puertos, teniendo que repetir la configuración varias veces.

Con estos conceptos adquiridos y la práctica guiada terminada, se puede comenzar a trabajar en esta práctica final.

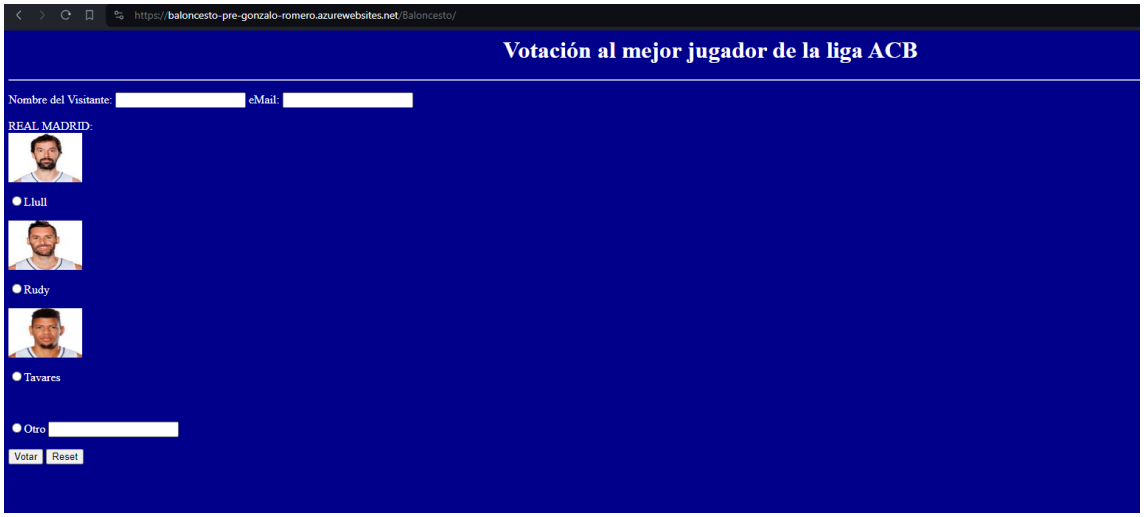
Cambios en el workflow del proyecto

Como pide el enunciado, se añadió al workflow un nuevo trabajo, “stage”. Este se hizo basándose en el trabajo “deploy” realizado para la práctica del tema 5. Se colocó entre los trabajos “qa” y “deploy”. Su código es el siguiente:

```yaml
stage:
  runs-on: ubuntu-latest
  needs: qa
  if: github.ref == 'refs/heads/main'
  steps:
    - name: Descargar repositorio
      uses: actions/checkout@v3
    - name: Crear el archivo .war
      run: |
        mvn package -DskipTests=true
    - name: Aprobación manual
      uses: trstringer/manual-approval@v1
      with:
        secret: ${{ secrets.TOKEN }}
        approvers: Gonza190
    - name: Desplegar en Azure
      uses: Azure/webapps-deploy@v2
      with:
        app-name: baloncesto-pre-gonzalo-romero
        publish-profile: ${{ secrets.AZURE_WEBAPP_PRE_PUBLISH_PROFILE }}
        package: target/*.war
```

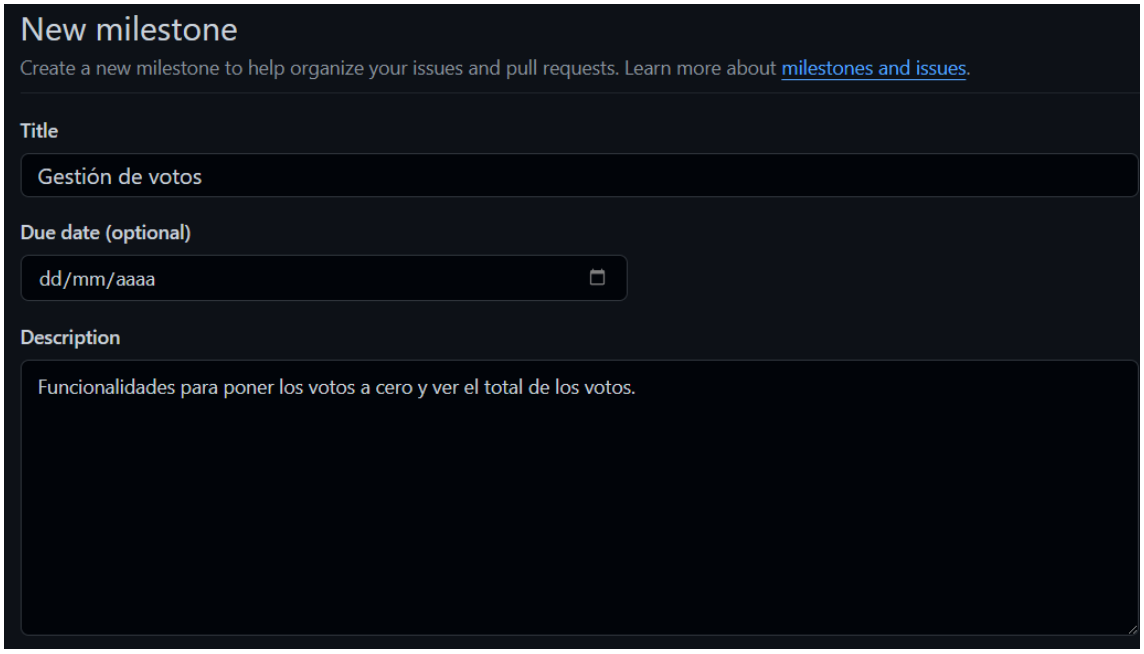
Para el despliegue en preproducción, se creó una nueva aplicación en Azure, de la misma manera que se hizo para la web del despliegue final. El secreto para el perfil de publicación de Azure también se configuró en GitHub de la misma manera.

El enlace de la nueva aplicación, en preproducción, es: https://baloncesto-pre-gonzalo-romero.azurewebsites.net/Baloncesto

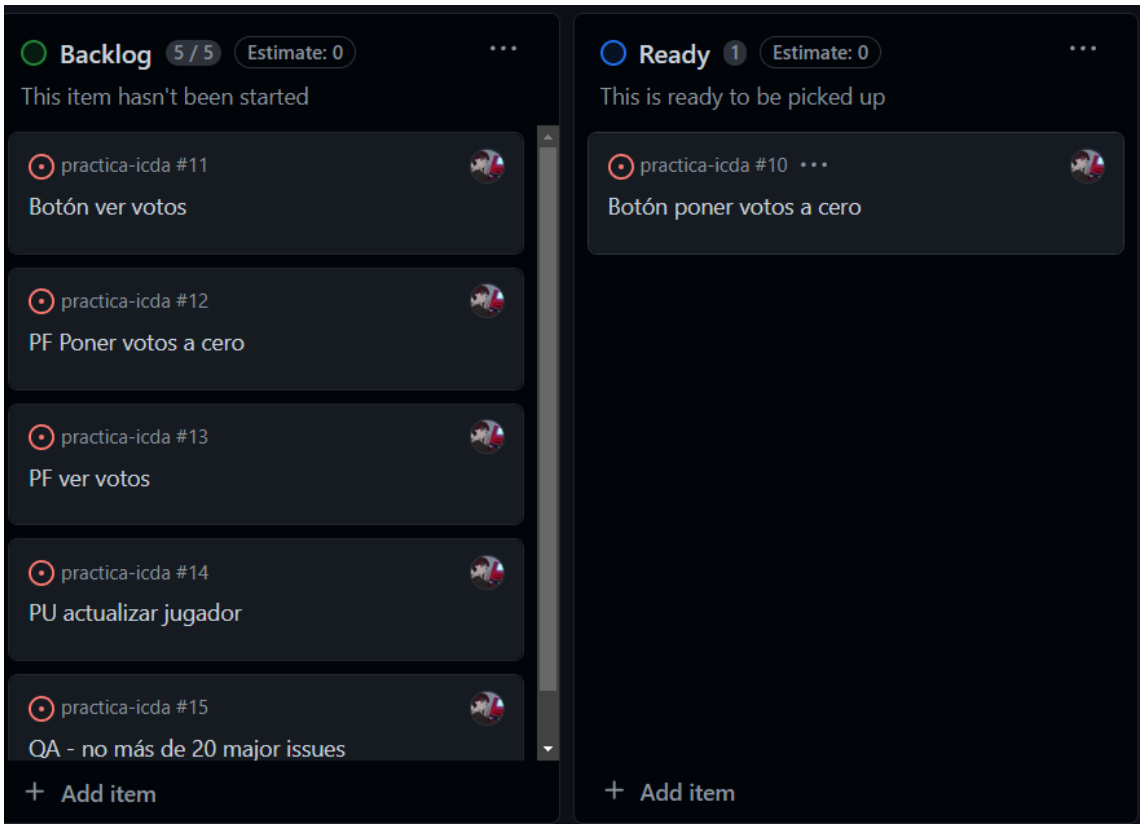


Nuevo sprint

En esta sección se va a preparar el sprint, para añadir las funcionalidades de la nueva versión del proyecto. Como en la práctica guiada, creamos un nuevo milestone (sprint):



También se han creado las 6 issues especificadas en el enunciado de la práctica final, y se han incluido en el proyecto Baloncesto:



Programación de los issues

QA

Para garantizar que la issue QA-no más de 20 major issues, se utilizó Sonarqube. Se cambió la Quality Gate de Sonarqube de forma que, si hay más de 20 major issues en el código, no se pueda continuar y el trabajo en el workflow falle:

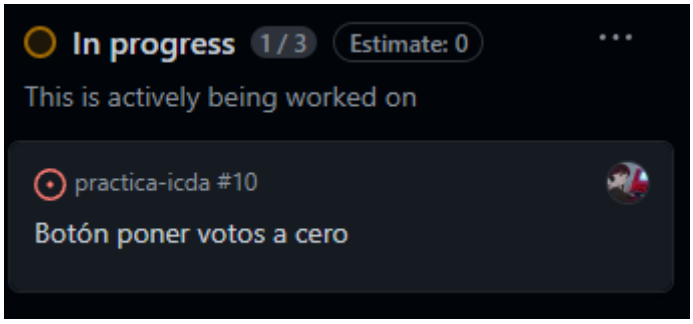
Conditions on Overall Code

Metric	Operator	Value
Major Issues	is greater than	20

Para bajar el número de issues en el código, se siguieron las indicaciones de Sonarqube. Esto se ha vigilado durante todo el proyecto, y hasta que no se hayan terminado todas las demás issues y el código no tenga menos de 20 major issues, no se dará por terminada esta issue.

REQ-1

Como indica el enunciado, primero se programó el REQ-1, poner los votos a cero, y después REQ-2, ver votos.



Para ello, se crearon dos ramas, mediante el comando git branch:

```
$ git branch poner-votos-a-cero
```

```
$ git branch ver-votos
```

```
$ git branch
  añadir-estilos-pagina-principal
  añadir-estilos-pagina-resultado
* main
  poner-votos-a-cero
  ver-votos
```

Cambiamos a la rama poner-votos-a-cero y comenzamos a trabajar en el REQ-1. Cambiamos la rama:

```
$ git checkout poner-votos-a-cero
Switched to branch 'poner-votos-a-cero'
```

Y comenzamos a programar. El REQ-1 indica que al pulsar el botón “Poner votos a cero” de la aplicación, los votos de todos los jugadores en la base de datos se deben poner a cero. Para ello, se realizaron varios cambios en el código.

Primero, programamos el método “ponerVotosACero” en “ModeloDatos”, que accede a la base de datos y cambia los votos de todos los jugadores a cero:

```
    public void ponerVotosACero() {
        try {
            set = con.createStatement();
            set.executeUpdate(sql:"UPDATE Jugadores SET votos=0");
            rs.close();
            set.close();
        } catch (Exception e) {
            logger.severe(msg:"No se modiican los votos");
            logger.severe(errorMsg + e.getMessage());
        }
    }
```

En “index”, creamos un botón para que ejecute el método:

```
    <p>
      <input type="submit" name="cero" value="Poner votos a cero" />
    </p>
```

Y detectamos en “Acb” si se ha pulsado dicho botón, haciendo que se recargue la página:

```
// poner votos a cero
if (req.getParameter("cero") != null) {
    bd.ponerVotosACero();
    res.sendRedirect(res.encodeRedirectURL("index.html"));
    return;
}
```

PU

La prueba unitaria PU, se programó junto al requisito REQ-1 en la misma rama. Para esta prueba unitaria, se creó una prueba para “actualizarJugador”, en “ModeloDatosTest”:

```
@Test
public void testActualizarJugador() {    Remove this 'public' modifier.

    System.out.println(x:"Prueba de actualizarJugador");

    String nombre = "Llull";
    ModeloDatos instance = new ModeloDatos();
    instance.abrirConexion();

    int antesDeActualizar = instance.cuantosVotosJugador(nombre);
    instance.actualizarJugador(nombre);
    int despuesDeActualizar = instance.cuantosVotosJugador(nombre);

    // Se aumenta en 1 tras actualizar
    assertEquals(antesDeActualizar + 1, despuesDeActualizar);
}
```

En el método se crea una instancia del objeto “ModeloDatos”, abrimos la conexión con la base de datos, guardamos el número de votos antes y después de actualizar el jugador, y comparamos sus resultados. Si el método actualizar jugador funciona, después de actualizar el jugador una vez, el número antes de actualizarlo debería uno menos que el de después de actualizarlo, de ahí que en la prueba unitaria comparemos el valor de “antes + 1” con el de “después”.

Par contar los votos, se creó un método en “ModeloDatos”, “cuantosVotosJugador”, que obtiene de la base de datos los votos que tiene en ese momento el jugador cuyo nombre se pasa como parámetro, y los devuelve:

```java
public int cuantosVotosJugador(String nombre) {
    int votos = 0;
    try {
        set = con.createStatement();
        rs = set.executeQuery("SELECT votos FROM Jugadores WHERE nombre " + " LIKE '%" + nombre + "%'");
        while (rs.next()) {
            votos = rs.getInt(columnLabel:"votos");
        }
        rs.close();
        set.close();
    } catch (Exception e) {
        // No lee de la tabla
        logger.severe(msg:"No lee los votos del jugador");
        logger.severe(errorMsg + e.getMessage());
    }
    return votos;
}
```

También se añadió al trabajo de “build” del workflow lo siguiente, de manera que se use nuestro runner local, y se prepare la base de datos de prueba en este momento, ya que es indispensable para realizar las pruebas:

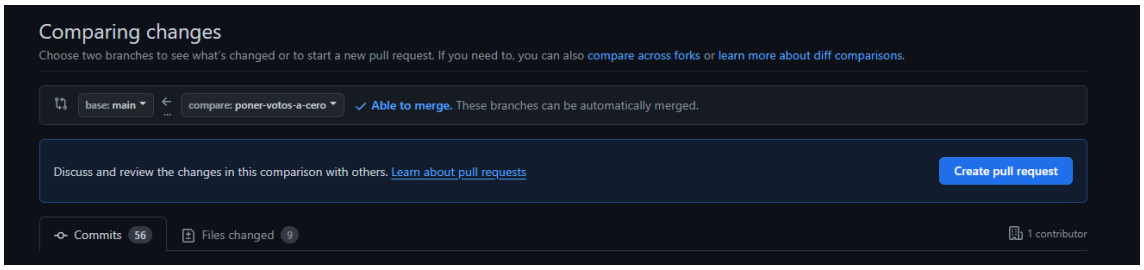
```yaml
build:
  runs-on: self-hosted
  steps:
    - name: descargar repostorio
      uses: actions/checkout@v3
    - name: preparar base de datos de prueba
      run: |
        mysql -u root < db/baloncesto.sql
    - name: pruebas-unitarias
      run: mvn test
    - name: compilar la aplicación sin repetir pruebas
      run: mvn package -DskipTests=true
```

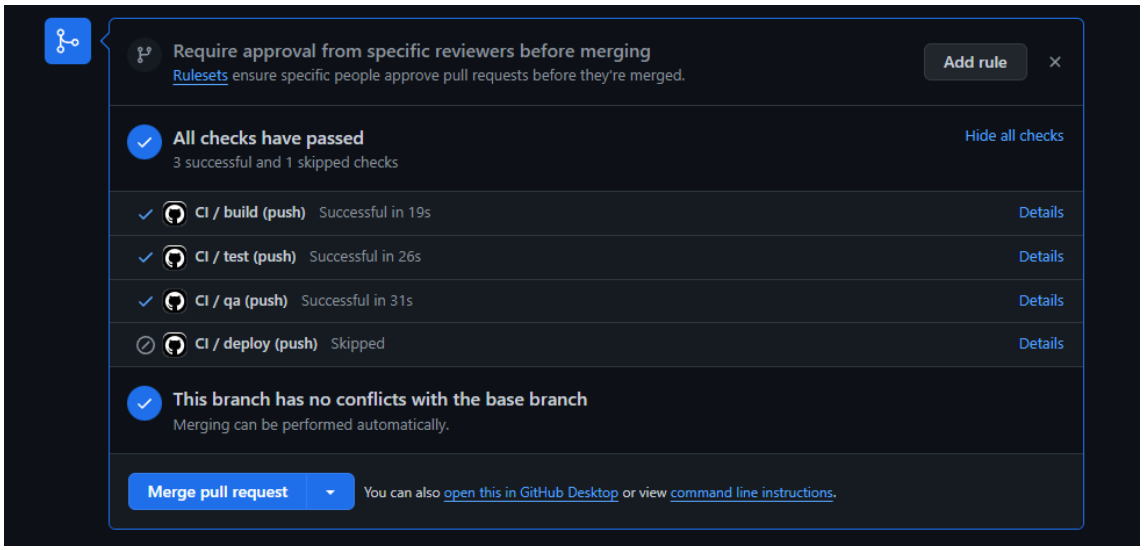
Para poder realizar la prueba unitaria, se tuvo que cambiar también el método de “abrirConexion”, debido a que, al ejecutarlo en la fase de pruebas, las variables de la base de datos no estaban definidas, y no se conectaba con la base de datos del contenedor.

```java
public void abrirConexion() {

    try {
        Class.forName(className:"com.mysql.cj.jdbc.Driver");

        // Con variables de entorno
        // String dbHost = System.getenv().get("DATABASE_HOST");
        // String dbPort = System.getenv().get("DATABASE_PORT");
        // String dbName = System.getenv().get("DATABASE_NAME");
        // String dbUser = System.getenv().get("DATABASE_USER");
        // String dbPass = System.getenv().get("DATABASE_PASS");

        String dbHost = "jdbc:mysql://localhost";
        String dbPort = "3306";
        String dbName = "baloncesto";
        String dbUser = "usuario";
        String dbPass = "clave";

        String url = dbHost + ":" + dbPort + "/" + dbName;
        con = DriverManager.getConnection(url, dbUser, dbPass);

    } catch (Exception e) {
        // No se ha conectado
        logger.severe(msg:"No se ha podido conectar");
        logger.severe(errorMsg + e.getMessage());
    }
}
```

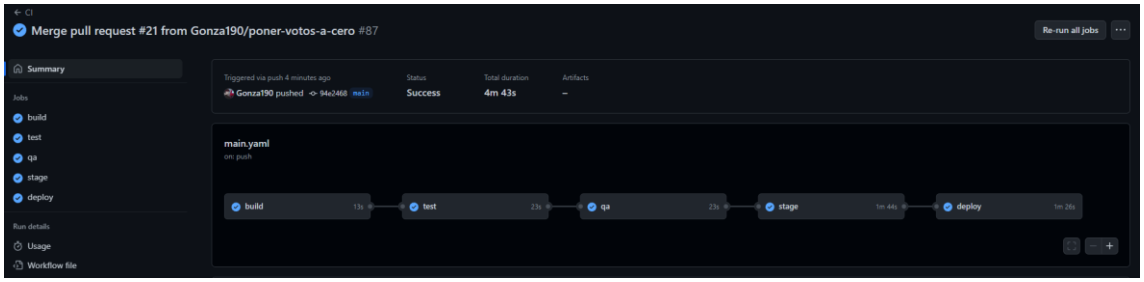
Aunque escribir las variables de entorno en el código no es muy seguro, ya que las podría ver cualquier usuario que entre en nuestro repositorio, las mismas variables se incluyen en el yaml del workflow, también público, por lo que se ha considerado escribirlas directamente en el código en lugar de acceder a ellas. Además, en el momento en el que se realizan las pruebas unitarias en la fase de build, las variables de entorno no están aún definidas.

Una vez realizadas tanto el REQ-1 como la PU, se realizó un pull request:

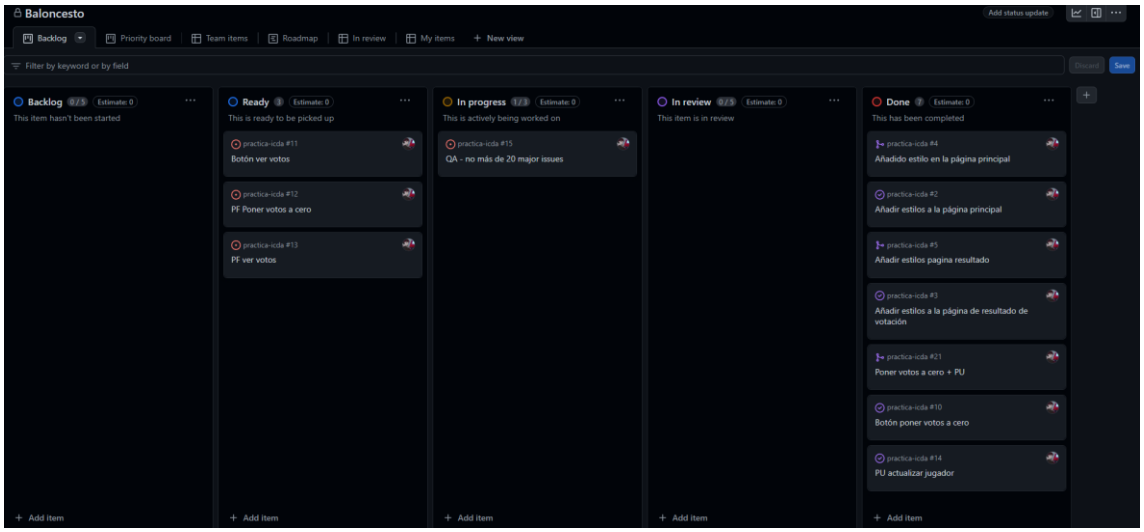




Al hacer el pull request, se comienza a realizar el workflow de nuevo, incluido el nuevo trabajo de stage creado en apartados anteriores.



En el proyecto de GitHub, cerramos las issues del REQ-1 y la PU y las movemos a Done, y ya podemos poner en Ready las tres issues restantes (REQ-2, PF-A y PF-B). QA sigue en proceso, ya que se ha considerado que es un trabajo constante de todo el proyecto, y que mantener las major issues por debajo de 20 se ha controlado durante todo el desarrollo.



Ahora sí, podemos empezar a trabajar en las issues restantes, ahora en la otra rama.

REQ-2

Para comenzar a trabajar en las issues, se ha cambiado de rama a ver-votos:

```
gonza@DESKTOP-KP3PBTC MINGW64 /d/master-web-proyectos/practica-icda (poner-votos
-a-cero)
$ git branch
  añadir-estilos-pagina-principal
  añadir-estilos-pagina-resultado
  main
* poner-votos-a-cero
  ver-votos

gonza@DESKTOP-KP3PBTC MINGW64 /d/master-web-proyectos/practica-icda (poner-votos
-a-cero)
$ git checkout ver-votos
Switched to branch 'ver-votos'
```

Sin embargo, para realizar las tareas restantes, se necesitaba código de la rama main, por lo que se hizo un pull request. Hay un conflicto en “ModeloDatos”, ya que se editó un comentario en ver-votos, lo que genera un conflicto en los archivos que se quieren copiar desde main. Se soluciona el conflicto manualmente desde el editor de código de Github.

Aun así, GitHub obliga a crear una nueva rama secundaria para mantener la seguridad de la rama main al hacer pull request, a la que llamamos ver-votos-secundaria. Después de realizar esto, y hacer pull, podemos trabajar en ver-votos con el código desarrollado en main.

Para resolver el REQ-2, se llevaron a cabo varias acciones.

La primera fue crear el botón que redirija a la página con la tabla de votos en index:

```
<p>
  <input type="submit" name="verVotos" value="Ver votos" />
</p>
```

Después, se creó un método en “ModeloDatos”, que accediera a la base de datos y recuperara el nombre y número de votos de cada jugador:

```
public List<Jugador> obtenerJugadores() {

    List<Jugador> jugadores = new ArrayList();    Type safety:
    try {
        set = con.createStatement();
        rs = set.executeQuery(sql:"SELECT * FROM Jugadores");
        while (rs.next()) {
            jugadores.add(new Jugador(
                    rs.getString(columnLabel:"nombre"),
                    rs.getInt(columnLabel:"votos")));
        }
        rs.close();
        set.close();
    } catch (Exception e) {
        // No lee de la tabla
        logger.severe(msg:"No lee los jugadores");
        logger.severe(ERROR_MSG + e.getMessage());
    }
    return jugadores;
}
```

Para almacenar los datos de cada jugador en la lista que se devuelve, se codificó una clase sencilla, “Jugador”:

```java
package jugador;

//Se ha creado un paquete para esta única clase debido a
//que al tratar de importar esta clase desde src/main/java
//donde están las demás clases .java, el JSP no compila

public class Jugador {

    private String nombre;
    private int votos;

    public Jugador(String nombre, int votos) {
        this.nombre = nombre;
        this.votos = votos;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getVotos() {
        return votos;
    }

    public void setVotos(int votos) {
        this.votos = votos;
    }

}
```

Como se indica en el comentario, se ha tenido que crear un paquete jugador dentro de scr/main/java, debido a que el JSP que se utiliza para representar la tabla no permitía importar clases dentro de ese paquete. Como no se podían importar clases de ese paquete, la llamada al método “obtenerJugadores” de “ModeloDatos” se realizó desde “Acb”:

```java
// ir a VerVotos.jsp
if (req.getParameter("verVotos") != null) {

    // al importar cualquier archivo del paquete src/main/java al
    // JSP da error, por lo que se introducen los jugadores en la request

    req.setAttribute("jugadores", bd.obtenerJugadores());
    req.getRequestDispatcher("VerVotos.jsp").forward(req, res);
    return;
}
```

El JSP “VerVotos” es el siguiente. Únicamente contiene un título y una tabla, en la que sus valores se iteran desde el valor de la request:

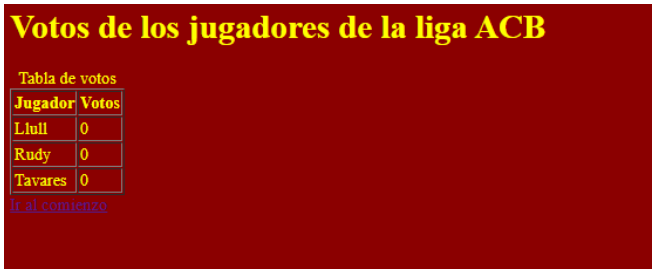
```jsp
<%
//obtengo los jugadores del request
List<Jugador> jugadores = (List<Jugador>) request.getAttribute("jugadores");
%>
```

```
<table border="1" >
<caption>Tabla de votos</caption>
  <tr>
    <th>Jugador</th>
    <th>Votos</th>
  </tr>
  <%
  //itero los jugadores en una tabla, si no son nulos
  if(jugadores!=null){
    for (Jugador jugador : jugadores) { %>
    <tr>
      <td><%= jugador.getNombre() %></td>
      <td><%= jugador.getVotos() %></td>
    </tr>
  <% }} %>
</table>
```

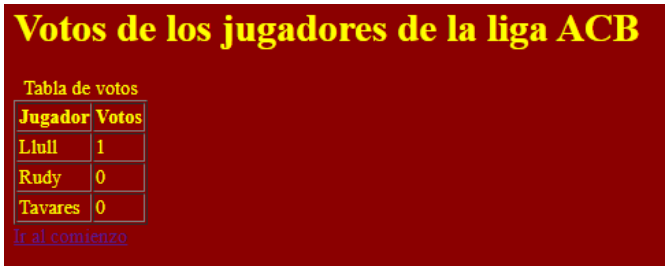
También se corrigieron ciertos errores que indica la extensión Sonarlint de Visual Studio Code, refactorizando así el código antes de las pruebas funcionales:

```
<html>    Add "lang" and/or "xml:lang" attributes to this "<html>" element
```

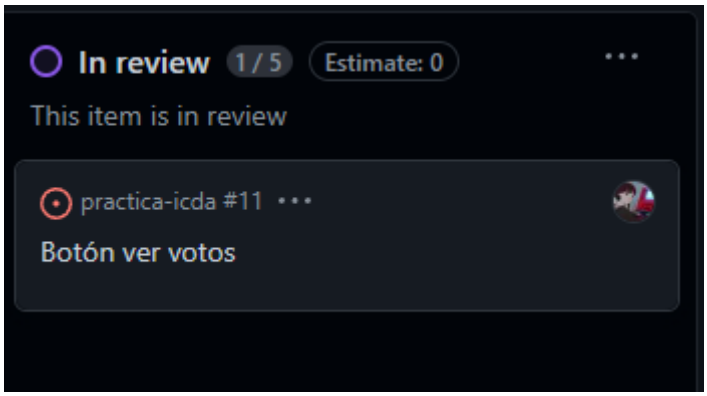
Esta es la vista:



Si votamos un jugador, la tabla se modifica (solamente en local con Tomcat):



Con la vista terminada, se puede pasar a programar las dos pruebas funcionales. En el tablero Kanban, colocamos el REQ-2 como “In review”, ya que se quiere comprobar que también funciona con las pruebas funcionales.



PF-A

Para esta prueba, se pretende simular un usuario al pulsar “Poner votos a cero” y “Ver votos”, de forma que todos los votos que aparezcan en la tabla sean cero.

Para ello, se programó la siguiente prueba funcional:

```
// se pulsa en el boton "Poner votos a cero"
driver.findElement(By.id("cero-btn")).click();

// se pulsa en el boton "Ver votos"
driver.findElement(By.id("ver-votos-btn")).click();

// se busca la tabla
WebElement tabla = driver.findElement(By.tagName("table"));

// se busca cada fila y se introduce en una lista
List<WebElement> filas = tabla.findElements(By.tagName("tr"));

boolean todasCero = true;

// se itera sobre la lista de filas
for (int i = 1; i < filas.size(); i++) {
        List<WebElement> columnas = filas.get(i).findElements(By.tagName("td"));
        // si la segunda columna no es igual a cero, error
        if (!columnas.get(index:1).getText().equals(anObject:"0")) {
                todasCero = false;
                break;
        }
}

// si todasCero es true, es porque todas las filas tienen cero votos
assertEquals(true, todasCero);
```

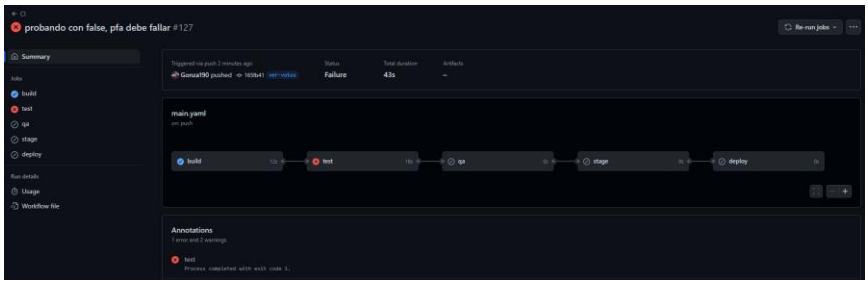
Se tuvieron que especificar ids para los dos botones creados en index. Se buscan los botones, se pulsan, y se itera sobre la tabla. Si para alguna fila, el número de votos no es igual a cero, la variable “todasCero” sería false, y el assert no sería correcto y daría error.

Tras ejecutar la prueba en el workflow, no da error, por lo que se puede concluir que la aplicación pasa la prueba funcional.

Además, si cambiamos el assert a false, la prueba falla, lo que demuestra que todasCero se mantiene como verdadera para toda la prueba:

```
assertEquals(false, todasCero);
```

Al fallar la prueba, el workflow también falla:



Damos por buena la prueba, y la ponemos en la columna “In review” hasta terminar la segunda prueba funcional.

Ya se puede desarrollar la última prueba funcional.

PF-B

Para desarrollar esta prueba, se debe simular que se vota a un jugador que no está en la base de datos seleccionando “Otro”, y comprobar si en la tabla de votos aparece con 1 solo voto. Para ello, se programó la siguiente prueba:

```
// nombre del jugador con el que probaremos
String nombre = "Gasol";

// se busca el select y el campo de texto para rellenarlos
driver.findElement(By.id("select-otro")).click();
driver.findElement(By.id("text-otro")).sendKeys(nombre);

// se pulsa en el boton "Votar"
driver.findElement(By.id("votar")).click();

// se vuelve al index
driver.navigate().to("http://localhost:8080/Baloncesto/");

// se pulsa en el boton "Ver votos"
driver.findElement(By.id("ver-votos-btn")).click();

// se busca la tabla
WebElement tabla = driver.findElement(By.tagName("table"));

// se busca cada fila y se introduce en una lista
List<WebElement> filas = tabla.findElements(By.tagName("tr"));

// buscamos solo la columna 4 porque se inserta al final de la tabla
boolean votado = false;
List<WebElement> columna = filas.get(index:4).findElements(By.tagName("td"));
```

```java
// si el jugador se inserta y en la columna de votos
// tiene un 1 es porque funciona
if (columna.get(index:1).getText().equals(anObject:"1") &&
            columna.get(index:0).getText().equals(nombre)) {
        votado = true;
}

// Prints para ver desde workflow si se inserta bien la informacion
System.out.println("JUGADOR: " + columna.get(index:0).getText());
System.out.println("VOTOS: " + columna.get(index:1).getText());

// si votado es true, es porque se ha introducido correctamente
assertEquals(true, votado);

driver.close();
driver.quit();
```

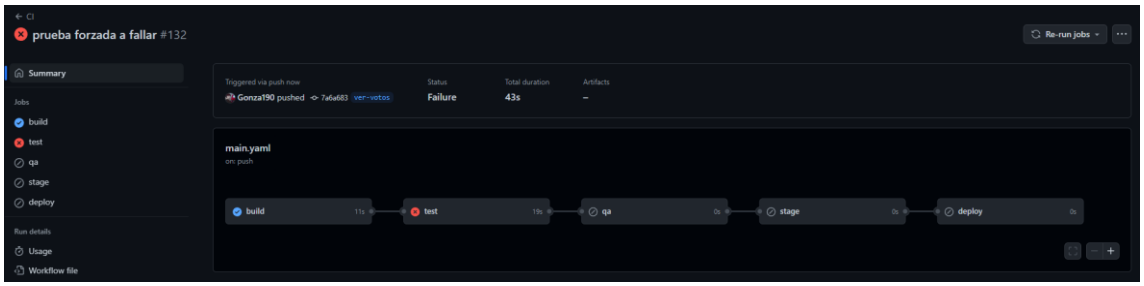
Lo que se hace en la prueba es buscar los campos para votar a otro jugador mediante sus ids (se cambiaron en el index), pulsar sobre el botón “Votar”, volver al index, pulsar en “Ver votos”, y comprobar que el jugador se encuentra en la tabla, y que tiene un voto.

Se usa el índice 4 para buscar en la columna 4 de la tabla (el header tiene índice 0), ya que en la base de datos se tienen solo tres jugadores, y al insertar el nuevo serían 4 en total.

También se forzó el error de la prueba, esta vez, cambiando el nombre puesto por otro:

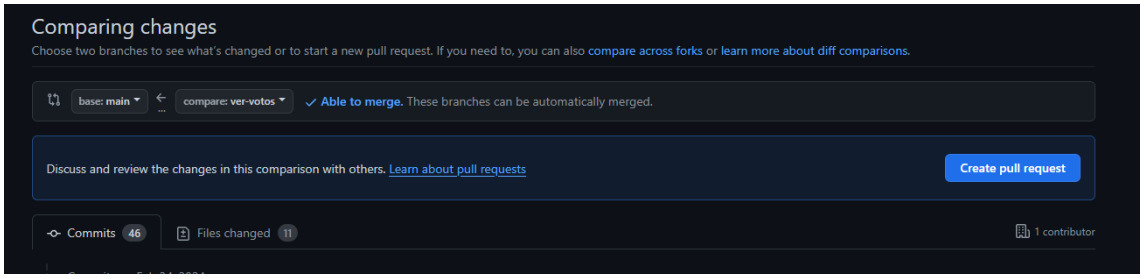
```java
if (columna.get(index:1).getText().equals(anObject:"1") &&
            columna.get(index:0).getText().equals(anObject:"Fallo")) {
        votado = true;
}
```

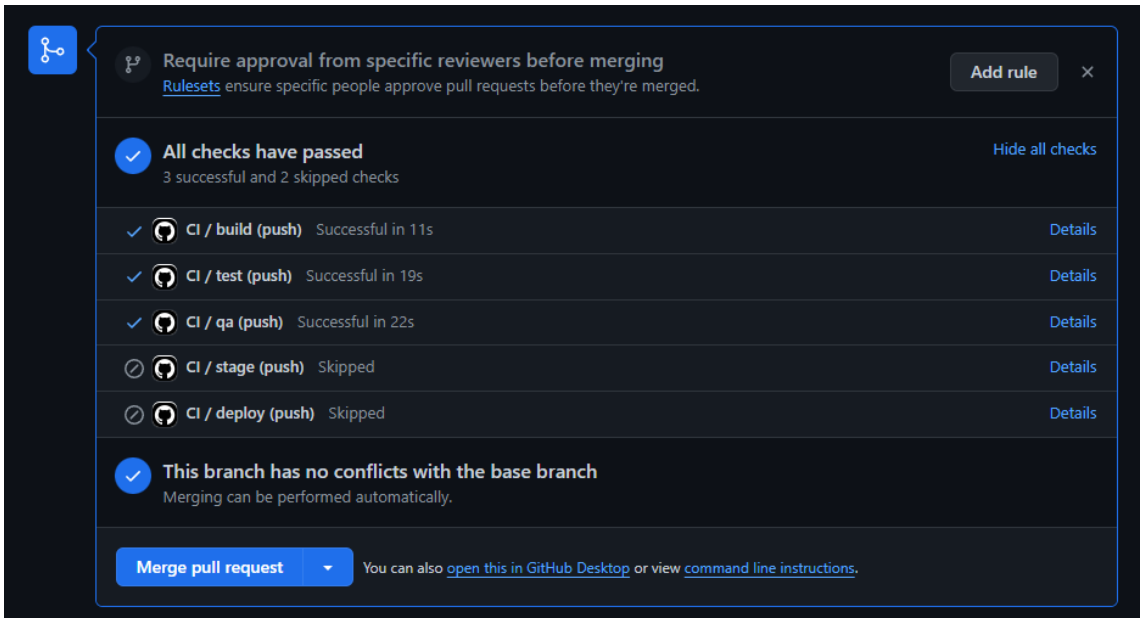
La prueba falla cuando hacemos esto:



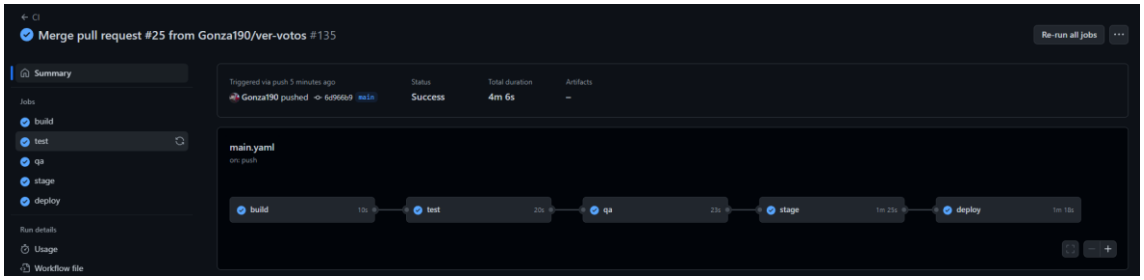

```
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 6.728 s <<< FAILURE! -- in PruebasPhantomjsIT
[ERROR] PruebasPhantomjsIT.otroJugadorTest -- Time elapsed: 2.703 s <<< FAILURE!
```

Por lo que podemos concluir que la prueba se ha realizado correctamente. Con todas las pruebas pasadas, el desarrollo de REQ-2, PF-A y PF-B está terminado y es correcto, por lo que se puede realizar el pull request y pasar el código a main para realizar el despliegue. Esto se realizó de la misma manera que para la rama anterior:



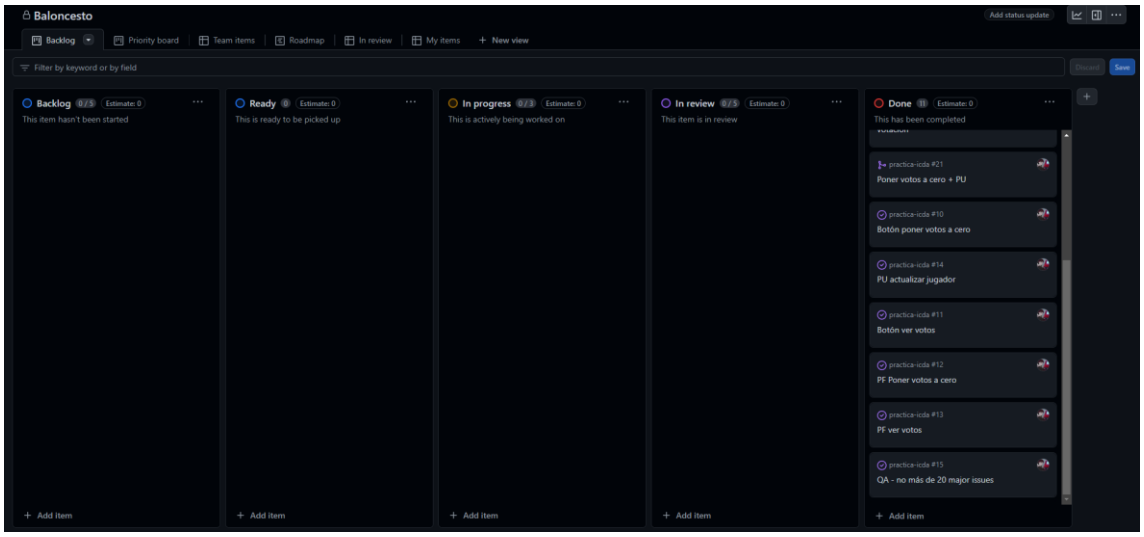


Una vez hecho el merge, sin conflictos, comenzó el workflow en main, está vez con el despliegue en preproducción y el final:



No ha habido errores al hacer el pull request, por lo que podemos poner las tres issues que teníamos en Done y cerrarlas. Como nuestro código tampoco tenía más de 20 major issues al finalizar el desarrollo, también podemos dar por finalizada la issue QA, moverla a Done en el tablero, y cerrarla.

El tablero final quedaría así:



Al realizar el último pull request, se olvidó poner que pertenecía al Milestone creado, y se puso después, por lo que no aparece en el tablero como los demás.

Conclusiones

Tras haber trabajado con CI/CD durante este trabajo, se han sacado algunas conclusiones:

- Trabajar con CI/CD es muy cómodo a la hora de desplegar, ya que se hace automáticamente, aunque es complicado de configurar.
- Trabajar con el contenedor a veces era tedioso, ya que cada vez que se hacía push de algo sobre el repositorio había que volver a encender el Tomcat para probar en local. Además, se ha perdido mucho tiempo por fallos que eran más difíciles de investigar con el Tomcat (conexiones a la base de datos en REQ-1 y problemas con los imports en REQ-2).
- El uso de ramas y pull requests facilitan el desarrollo, aunque para esta práctica no ha sido tan evidente debido a que solamente había una persona desarrollando el código y no varias a la vez.