**EGC**

Frontend de Resultados

Grupo 8

Cabrera Coronilla, Manuel

Cumplido Díaz, Raquel Mª

Fernández Bueno, José Antonio

Fernández García, Álvaro

Fernández Romero, Daniel

González Castro, Adrián

Ocaña Almagro, José Pablo

Rodríguez Palomar, Alejandro

# Resumen

Este documento contiene toda la información relativa al desarrollo del subsistema Frontend de Resultados de la aplicación web Agora@US. Agora@US es un sistema de información que ofrece la posibilidad de realizar votaciones por Internet. Ésta se utilizará para aprender a llevar a cabo la gestión de la configuración en un proyecto, y constituye el trabajo cuatrimestral de la asignatura Evolución y Gestión de la Configuración, perteneciente a la titulación de Grado en Ingeniería del Software.

La organización del trabajo consiste en la división del sistema Agora@US en subsistemas más pequeños, cuyo desarrollo es asignado a distintos grupos de trabajo conformados por los alumnos. En nuestro caso, el grupo se encarga de desarrollar el subsistema de Frontend de Resultados, el cual se encarga de comunicarse con los subsistemas de Recuento y Modificación de Resultados para obtener datos que posteriormente son enviados al subsistema de Visualización de Resultados.

Para realizar el trabajo se han usado, entre otras, las siguientes herramientas:

* Gestión del proyecto: *Redmine*
* Gestión del código fuente: *Git*
* Gestión de la construcción: *Maven*
* Gestión de la integración continua: *Jenkins*
* Gestión del despliegue: *Openshift*
* Gestión de la calidad: *SonarQube*

# Historial de versiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Versión** | **Fecha** | **Contenido** | **Autor** |
| 1.0 | 17/12/2014 | Formato y estructura del documento. | Adrián González Castro |
| 1.1 | 17/12/2014 | Redacción de introducción y resumen. | Adrián González Castro |
| 1.2 | 17/12/2014 | Redacción de gestión de la construcción y la integración continua. | Daniel Fernández Romero, Raquel María Cumplido Díaz |
| 1.3 | 18/12/2014 | Redacción de gestión del cambio, incidencias y depuración. | José Antonio Fernández Bueno |
| 1.4 | 18/12/2014 | Redacción de la gestión del código fuente. | Álvaro Fernández García |
| 1.5 | 20/12/2014 | Redacción del mapa de herramientas. | Manuel Cabrera Coronilla |
| 2.0 | 20/12/2014 | Redacción de las conclusiones. | Raquel Cumplido Díaz |
| 2.1 | 15/01/15 | Mejora del apartado de gestión del código fuente | Raquel Cumplido,  José Pablo Ocaña, Álvaro Fernández y Adrián González |
| 2.15 | 17/01/15 | Reestructuración de la memoria, mejora de índices, mejora de introducción y resumen | José Pablo Ocaña Almagro |
| 2.2 | 21/01/15 | Introducción del apartado de gestión de calidad | Daniel Fernández Romero |
| 2.3 | 25/01/15 | Introducción del apartado de gestión de la construcción e integración continua | José Antonio Fernández y José Pablo Ocaña |
| 2.4 | 27/01/15 | Mejora del apartado de gestión del cambio, incidencias y depuración | Alejandro Rodríguez |
| 2.5 | 30/01/15 | Introducción del apartado de gestión de la construcción y despliegue | Manuel Cabrera, Álvaro Fernández |
| 2.55 | 31/01/15 | Ampliación del apartado del mapa de herramientas | José Pablo Ocaña |
| 2.6 | 02/02/15 | Introducción del apartado de integración de todos los subsistemas | José Antonio Fernández y José Pablo Ocaña |
| 2.65 | 02/02/15 | Redacción del glosario de términos, los anexos y la bibliografía | José Pablo Ocaña Almagro, Álvaro Fernández y José Antonio Fernández |
| 2.7 | 03/02/15 | Se rehacen las conclusiones | Daniel Fernández Romero |
| 3.0 | 05/02/15 | Organización del documento completo, corrección y revisión completa | José Pablo Ocaña Almagro, Daniel Fernández Romero |

1. Índice

[Resumen 3](#_Toc410870766)

[Historial de versiones 3](#_Toc410870767)

[1. Índice 5](#_Toc410870768)

[2. Índice de figuras 7](#_Toc410870769)

[3. Roles 8](#_Toc410870770)

[4. Introducción 8](#_Toc410870771)

[5. Gestión del código fuente 11](#_Toc410870772)

[**5.1.** **Sintaxis del código fuente** 11](#_Toc410870773)

[**5.2.** **Gestión de ramas** 12](#_Toc410870774)

[**5.2.1.** **Gestión interna** 12](#_Toc410870775)

[**5.2.2.** **Gestión externa** 14](#_Toc410870776)

[**5.3.** **Creación y aplicación de *patch*** 15](#_Toc410870777)

[**5.4.** **Roles** 16](#_Toc410870778)

[**5.5.** **Conflictos** 16](#_Toc410870779)

[**5.6.** **Migración de SVN a Git** 17](#_Toc410870780)

[**5.7.** **Herramientas similares** 17](#_Toc410870781)

[**5.8.** **Ejercicios** 17](#_Toc410870782)

[**Ejercicio 1: Añadir nueva característica haciendo uso de ramas** 17](#_Toc410870783)

[**Ejercicio 2: Resolución de conflictos** 20](#_Toc410870784)

[**Ejercicio 3: Creando y aplicando un parche** 24](#_Toc410870785)

[6. Gestión de la construcción y despliegue 26](#_Toc410870786)

[**6.1.** **Maven** 26](#_Toc410870787)

[**6.2.** **Openshift** 27](#_Toc410870788)

[**6.3.** **Ejercicios** 27](#_Toc410870789)

[7. Gestión de la construcción e integración continua 27](#_Toc410870790)

[**7.1.** **Jenkins** 28](#_Toc410870791)

[**7.1.1.** **Herramientas necesarias** 28](#_Toc410870792)

[**7.1.2.** **Tareas realizadas** 33](#_Toc410870793)

[**7.2.** **Ejercicio** 38](#_Toc410870794)

[8. Gestión de la calidad 38](#_Toc410870795)

[**8.1.** **Herramientas utilizadas** 39](#_Toc410870796)

[**8.2.** **Proceso de instalación de SonarQube y componentes asociados** 41](#_Toc410870797)

[**8.3.** **Integración de SonarQube con Jenkins** 46](#_Toc410870798)

[**8.4.** **Interpretación de los resultados obtenidos** 51](#_Toc410870799)

[**8.5.** **Tareas abiertas** 55](#_Toc410870800)

[**8.6.** **Conclusiones** 56](#_Toc410870801)

[**8.7.** **Ejercicio** 57](#_Toc410870802)

[9. Gestión del cambio, incidencias y depuración 61](#_Toc410870803)

[**9.1.** **Gestión del cambio** 62](#_Toc410870804)

[**9.2.** **Gestión de las incidencias** 64](#_Toc410870805)

[**9.3.** **Depuración** 65](#_Toc410870806)

[**9.4.** **Ejercicios** 65](#_Toc410870807)

[**Ejercicio 1: Ampliación de la funcionalidad** 65](#_Toc410870808)

[**Ejercicio 2: Incidencia y depuración** 67](#_Toc410870809)

[9. Mapa de herramientas 70](#_Toc410870810)

[**9.1.** **Lista de herramientas** 70](#_Toc410870811)

[**9.2.** **Visión interna del mapa de herramientas** 72](#_Toc410870812)

[**9.3.** **Visión global del mapa de herramientas** 73](#_Toc410870813)

[10. Integración de todos los subsistemas 74](#_Toc410870814)

[**10.1.** **Gestión del código** 74](#_Toc410870815)

[**10.2.** **Estrategia de integración** 74](#_Toc410870816)

[**10.3.** **Herramientas** 74](#_Toc410870817)

[**10.4.** **Automatización de la construcción** 74](#_Toc410870818)

[11. Conclusiones 75](#_Toc410870819)

[Glosario de términos 76](#_Toc410870820)

[Bibliografía 77](#_Toc410870821)

[**Gestión de la construcción y despliegue** 78](#_Toc410870822)

[**Gestión de la construcción e integración continua** 78](#_Toc410870823)

[**Gestión de la calidad** 78](#_Toc410870824)

[**Gestión del cambio, incidencias y depuración** 78](#_Toc410870825)

[**Mapa de herramientas** 78](#_Toc410870826)

[Anexos 78](#_Toc410870827)

1. Índice de figuras

**Figura 1: Arquitectura de Agora@US........................................................................................7**

**Figura 2: Esquema de jerarquización de ramas..................................................................12**

**Figura 3: Gestión de ramas en Frontend de Resultados...................................................13**

**Figura X: Ejercicio Git 1: Uso de ramas 1.................................................................................X**

**Figura X: Ejercicio Git 1: Uso de ramas 2.................................................................................X**

**Figura X: Ejercicio Git 1: Uso de ramas 3.................................................................................X**

**Figura X: Ejercicio Git 1: Uso de ramas 4.................................................................................X**

**Figura X: Ejercicio Git 2: Resolución de conflictos 1............................................................X**

**Figura X: Ejercicio Git 2: Resolución de conflictos 2............................................................X**

**Figura X: Ejercicio Git 2: Resolución de conflictos 3............................................................X**

**Figura X: Ejercicio Git 3: Patch....................................................................................................X**

**Figura X: Mapa de herramientas de SonarQube...................................................................X**

**Figura X: Ubicación de SonarQube en el sistema operativo..............................................X**

**Figura X: Verificación de la creación de la base de datos de Sonar.................................X**

**Figura X: Configuración de Sonar...............................................................................................X**

**Figura X: Arranque de la aplicación Sonar..............................................................................X**

**Figura X: Menú principal de Sonar.............................................................................................X**

**Figura X: Configuración en Jenkins de Sonar Plugin............................................................X**

**Figura X: Configuración en Jenkins del analizador de Sonar............................................X**

**Figura X:** **Configuración en Jenkins del proceso post-construcción de Sonar.............X**

**Figura X:** **Configuración del fichero de Sonar en el proyecto............................................X**

**Figura X:** **Salida de la construcción del proyecto en Jenkins.............................................X**

**Figura X:** **Métricas del proyecto tras el análisis del código................................................X**

**Figura X:** **Lista de incidencias críticas en el código del subsistema................................X**

**Figura X:** **Descripción de una incidencia en el código..........................................................X**

**Figura X:** **Habilitación de envío de notificaciones.................................................................X**

**Figura X:** **Notificación recibida para solucionar una incidencia......................................X**

**Figura X:** **Resumen general de los análisis aplicados al código........................................X**

**Figura X:** **Registro de nuevo usuario en Sonar.......................................................................X**

**Figura X:** **Modificación del e-mail de un nuevo usuario......................................................X**

**Figura X:** **Notificación de la incidencia asignada...................................................................X**

**Figura X:** **Incidencia de mayor grado a solucionar...............................................................X**

**Figura X:** **Estado de la deuda técnica e incidencias tras resolver una de ellas............X**

**Figura X:** **Comparativa entre el análisis actual y anterior (I)............................................X**

**Figura X:** **Comparativa entre el análisis actual y anterior (II)..........................................X**

**Figura X:** **..........................................X**

**Figura X:** **..........................................X**

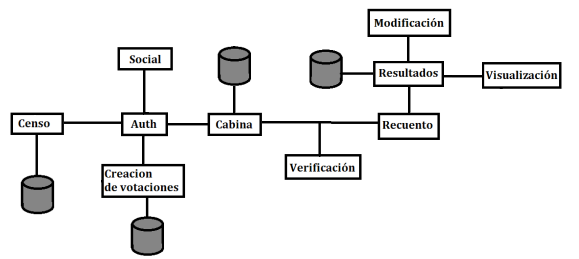
**Figura X:** **..........................................X**

# Roles

|  |  |
| --- | --- |
| **Nombre y apellidos** | **Rol** |
| Manuel Cabrera Coronilla | Gestor de la configuración |
| Raquel Cumplido Díaz | Gestor de la configuración |
| José Antonio Fernández Bueno | Jefe de proyecto |
| Álvaro Fernández García | Gestor de la configuración |
| Daniel Fernández Romero | Gestor de la configuración |
| Adrián González Castro | Gestor de la configuración |
| José Pablo Ocaña Almagro | Gestor de la configuración |
| Alejandro Rodríguez Palomar | Gestor de la configuración |

# Introducción

En esta sección se presenta la arquitectura de la aplicación en su totalidad y se describe, en resumidas líneas, el funcionamiento del subsistema Frontend de Resultados, así como las distintas etapas por las que ha pasado el desarrollo del proyecto en general.



*Figura 1: Arquitectura de Agora@US*

Para comenzar, se define el propósito principal del subsistema Frontend de Resultados. Éste se encarga de recibir los recuentos de los votos de las distintas votaciones o referéndums para almacenarlos en una base de datos propia y, posteriormente, atender las peticiones del subsistema de Visualización de resultados. Dichos recuentos serán recibidos de los subsistemas Modificación de Resultados y Recuento de Votos, que se ponen de acuerdo entre ellos para ofrecer la información correcta a través del subsistema de Recuento de Votos.

Desde las primeras etapas del proyecto, entre los grupos de Recuento, Modificación, Frontend y Visualización de Resultados se llegó al acuerdo de que la comunicación entre estos subsistemas se realizaría mediante el uso de APIs REST, por lo que los datos serían enviados como un JSON con formatos distintos, dependiendo de la API en cuestión.

Por otro lado, en lo referente a la tecnología utilizada, en primer lugar se tomó la decisión de utilizar el *framework* GWT (*Google Web Toolkit*) para desarrollar el subsistema y crear la API. No obstante, tras investigar y desarrollar la parte de la aplicación relacionada con la comunicación con la base de datos, en este caso HSQLDB (*HyperSQL DataBase*), unido a la nefasta experiencia sufrida en la primera sesión de integración, se decidió migrar el desarrollo para utilizar el framework *Spring* e *Hibernate* para que la comunicación se realizase con una base de datos MySQL. Una causa más de esta migración fue la incapacidad de HSQLDB para soportar operaciones concurrentes.

A partir de este momento, se desarrolla un segundo prototipo de la aplicación que consume la API del subsistema de Recuento de Resultados, instanciando los tipos Voto y Votación a partir del JSON emitido por dicha API, persistiéndolos en la base de datos. Por otro lado, estos datos son exportados a través de una API al subsistema de Visualización de Resultados adecuándose al formato utilizado por estos últimos. El funcionamiento global es el antes descrito: si la votación que necesita Visualización está en la base de datos es devuelta, y en caso contrario se realiza la petición a la API de Recuento de Votos. Este segundo prototipo fue el que se utilizó durante la segunda sesión de integración, logrando llevar a cabo la integración con el subsistema de Visualización de Resultados.

Tras esta segunda sesión de integración se refina este mismo prototipo y se consigue establecer la correcta comunicación con el subsistema de Recuento de Votos y con el subsistema de Modificación de Resultados. En este momento, el subsistema realiza las consultas al subsistema de Recuento de Votos o al de Modificación de Resultados, dependiendo de si el subsistema de Visualización de Resultados necesita el recuento de votos original o el modificado.

Sin embargo, poco antes de la fecha de la tercera sesión de integración, el formato de los JSON que se consumían y el de los JSON que necesita el subsistema de Visualización fue cambiado, por lo que hubo que adecuar la aplicación a estas nuevas circunstancias. Se llega así a la versión final del subsistema, cuya estructura se expone a continuación.

En primer lugar, las clases del dominio de la aplicación son la clase Propuesta y la clase ReferendumRecuento. La clase Propuesta contiene tres atributos que modelan una pregunta, el número de votos que han votado “SÍ” para esa propuesta y el número de votos que han votado “NO” para dicha propuesta.

Por otro lado, la clase ReferendumRecuento tiene tres atributos: **idVotaciónRecuento**, que es el identificador de la votación de la cual se desea obtener la información contenida en el subsistema de Recuento de Votos, **idVotaciónModificación**, que juega el mismo papel pero para la información que se quiere obtener del subsistema de Modificación de Resultados y, por último, una **colección de propuestas**.

La verdadera funcionalidad del subsistema se encuentra en el único controlador de la aplicación, definido en el archivo PropuestaController.java. Dicho controlador consta de los siguientes métodos:

* Object[] getVotación(@RequestParam Integer idVotación): este método se encarga de recibir el identificador de una votación que quiera ser consultada desde el subsistema de Visualización de Resultados para, posteriormente, devolver un JSON para que lo pueda consumir dicho subsistema. En este caso, la consulta se realiza al subsistema de Recuento de Votos.
* Object[] getModificacion(@RequestParam Integer idVotación): este método se encarga de recibir el identificador de una votación que quiera ser consultada desde el subsistema de Visualización con el fin de devolver un JSON para que lo pueda consumir dicho subsistema. En este caso, la consulta se realiza al subsistema de Modificación de Resultados.

En ambos casos se trata de métodos bastante sencillos que realizan la comunicación con la base de datos para comprobar si pueden devolver las votaciones directamente o deben pedírselas a los subsistemas correspondientes.

# Gestión del código fuente

A la hora de gestionar el código fuente, existen diferentes opciones y herramientas. En primer lugar, para elegir una herramienta que ayude al equipo a realizar esta labor, se sometieron a estudio dos opciones: Redmine y Project.net. En lo que respecta a la primera, se observaron una serie de ventajas e inconvenientes:

* Ventajas:
  + No precisa de instalación en equipos, sino que se trata de un servicio web.
  + Se pueden añadir y configurar elementos como *plugins* o temas visuales.
  + Se asemeja bastante a la herramienta ProjETSII, la cual ha sido ampliamente usada por todos los miembros del grupo.
* Inconvenientes:
  + Aunque tenga una gran cantidad de extensiones, no es demasiado personalizable.
  + Los lectores RSS no pueden leer correctamente noticias RSS de wikis.

Por otro lado, observando Project.net, cabe destacar:

* Ventajas:
  + Ofrece servicios de seguimiento para solucionar problemas en el proyecto.
  + Permite utilizar wikis o blogs como medio de comunicación dentro del propio proyecto.
* Inconvenientes:
  + Necesita el uso de Apache Tomcat como servidor de aplicaciones y bases de datos Oracle.
  + La versión web parece ser que sólo puede ser utilizada durante un tiempo limitado.

Después de realizar este estudio, se llegó al acuerdo de utilizar Redmine, debido a su similitud con ProjETSII, unido a que se trata de un servicio web.

Por otro lado, en lo que se refiere a la herramienta para gestionar el código fuente, en un principio se comenzó a usar SVN, ya que era la herramienta con la que estaban familiarizados los componentes del equipo de trabajo. No obstante, cuando se explicaron las ventajas de Git sobre SVN, se decidió migrar el proyecto. Más adelante se mostrará cómo se llevó a cabo esta migración para conseguir mantener el histórico de cambios.

* 1. **Sintaxis del código fuente**

En este apartado se detalla la política que se ha seguido a la hora de escribir el código fuente, con el fin de homogeneizar la sintaxis utilizada.

En primer lugar, los nombres de todas las clases empiezan por mayúscula, y en caso de tener más de una palabra, cada una de las que siguen a la primera también empezarán con mayúscula. En el caso de las variables, por el contrario, no se empieza por mayúsculas.

Por otro lado, los servicios y los repositorios relacionados con las entidades del dominio se denominarán siguiendo el esquema NombreEntidadService y NombreEntidadRepository respectivamente.

En lo que respecta a los métodos de cada clase, éstos comienzan con minúscula, pero en el caso de que tengan más de una palabra, de la segunda en adelante comienzan con mayúscula.

Por último, para la estructura general del código se ha seguido el esquema de programación indentada usando tabulaciones.

* 1. **Gestión de ramas**
     1. **Gestión interna**

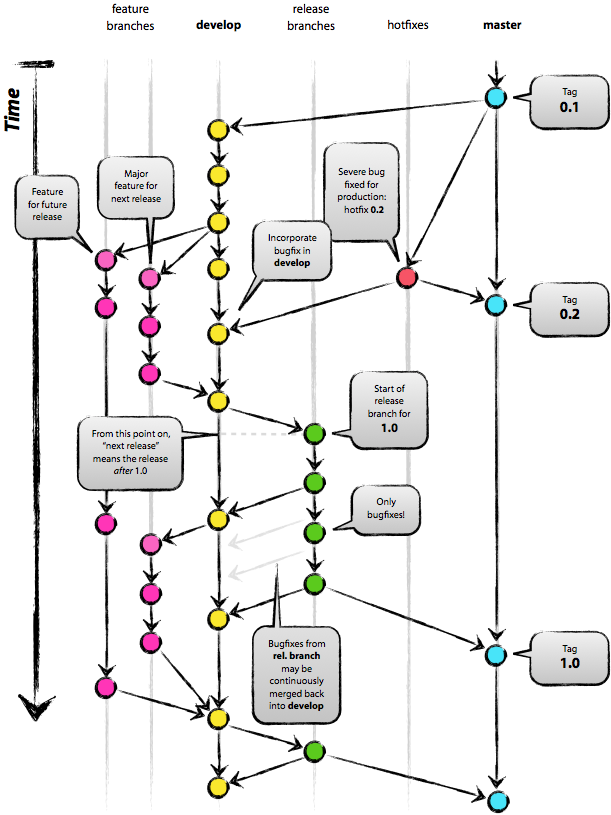
Al comenzar el proyecto, el equipo de trabajo estableció que, independientemente del sistema de control de versiones que se utilizase, SVN o GIT, se utilizarán diferentes ramas. Esto se debe a que gracias al uso de ramas se consigue:

* Organizar el trabajo de manera más eficiente.
* Llevar un control de versiones estables del proyecto.
* Desarrollar nuevas funcionalidades sin afectar a la versión estable del proyecto mientras se desarrolla.
* Realizar un *hotfix* en una versión que ya ha salido a producción.

Existen algunas políticas y recomendaciones generales a la hora de jerarquizar las distintas ramas que se tienen dentro de un proyecto. Las más habituales constan de las ramas:

* **Master.** Se recogen las versiones estables del proyecto.
* **Develop**. Se realiza el desarrollo del proyecto.
* **Features.** Ramificaciones para la adición de nuevas funcionalidades.
* **Release.** Rama de liberación de versiones, en la cual se comprueba la existencia de bugs, en caso de que no existan se encarga de pasar a la rama master la nueva versión del proyecto.
* **Hotfix.** Se solucionan los bugs detectados en las versiones de la rama master.

A continuación se presenta un esquema visual de esta jerarquización, en el cual se observa cómo avanza en el tiempo:



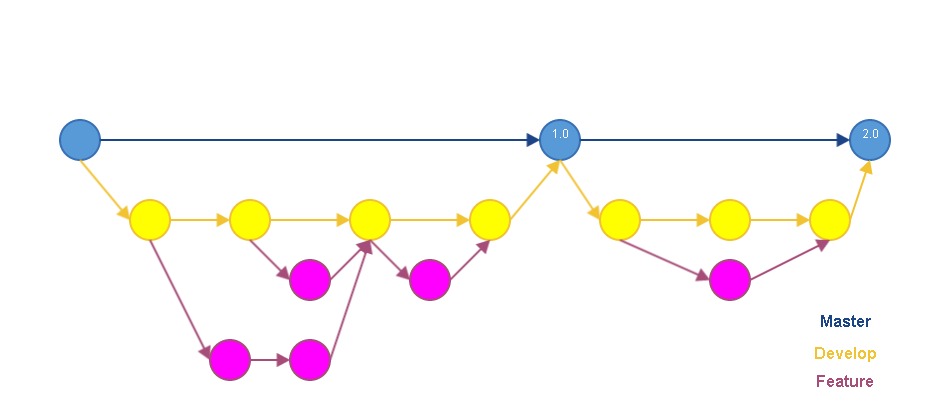
*Figura 2: Esquema de jerarquización de ramas*

En el caso concreto del subsistema de Gestión de Resultados, debido a su envergadura, se incluyeron la rama *master*, la rama *develop* y las ramas de *features*. La función que cumple cada una de estas ramas no difiere de lo anteriormente explicado, es decir, la rama *master* recoge las versiones estables del sistema. Para que esto sea posible, esta rama sólo se modifica en el caso de poseer en *develop* una versión totalmente estable y validada del sistema.

También en*develop* se recogen las versiones estables del proyecto, con las nuevas funcionalidades incluidas desde las ramas *feature*. Al igual que en la rama *master*, el *merge* se realiza cuando la nueva funcionalidad ha sido validada en la rama *feature*.

Por último, las ramas *feature* son aquellas donde se desarrollan las nuevas funcionalidades. Cuando se desea añadir una nueva funcionalidad al sistema, se crea una nueva *feature*en el repositorio. Esto implica que se crean tantas como funcionalidades nuevas hay, por lo que no hay un número fijo ni limitado de éstas.

Así pues, el esquema utilizado para este subsistema se trata de una simplificación del anterior, como puede observarse en la siguiente figura:



*Figura 3: Gestión de ramas en Frontend de Resultados*

* + 1. **Gestión externa**

En lo que se refiere a la administración del código fuente del sistema completo, en un principio se realizaba mediante Archivos de Aplicación Web (WAR), pero este método se desestimó debido a que era demasiado lento y costoso. Finalmente, la opción aprobada por todos los subsistemas fue la de crear un repositorio de Git común para todos los subsistemas, conocido por el nombre de Repvoting, y gestionar el conjunto de las versiones estables de cada uno de ellos desde ahí. Esta decisión hizo que la comunicación entre los distintos subsistemas fuese más fácil y rápida, además que todos los grupos tuvieran a su alcance la última versión actualizada del resto de los subsistemas.

En este repositorio cada subsistema se desarrolló en una rama diferente, y la rama *master* se usó para albergar las versiones estables del sistema Agora@US, formado por la unión de los distintos subsistemas. La decisión de que cada subsistema se desarrollara en una rama diferente fue tomada para que cada grupo fuera capaz de desarrollar su subsistema sin influir al resto, permitiendo así un desarrollo del mismo sin influir en la versión estable del proyecto situada en la rama *master*.

Cada subsistema aloja su código fuente en una carpeta dentro del repositorio, lo cual tiene como objetivo mejorar la organización del código entre los distintos subsistemas, y minimizar la posibilidad de alterar un subsistema diferente al tuyo.

En el caso del subsistema de Frontend de Resultados, el nombre que recibe la carpeta situada en el repositorio compartido es “results-frontend”, y la gestión de ramas dentro de esta carpeta sigue las pautas explicadas en la sección de gestión interna.

* 1. **Creación y aplicación de *patch***

En ocasiones se necesita exportar los cambios realizados en el código fuente, ya sea para enviárselo a un compañero por correo electrónico, o para dejarlo a un lado cuando no se está trabajando en él. Al fichero que contiene ese cambio o conjunto de cambios se le conoce como *patch*,o parche. La generación de un *patch* usando Git se realiza normalmente a través de la ventana de comandos. En primer lugar hay que navegar hasta el directorio donde se encuentra el proyecto, y situarse en la rama de la que se quiera exportar el cambio. Una vez hecho esto, se genera el *patch*, lo cual se puede hacer de dos formas:

* **%** git format-patch -1 HEAD 0001-ultimo-cambio-aplicado.patch

Este comando exporta el último cambio aplicado.

* **%** git format-patch -N COMMIT-ID

Donde 'N' se reemplaza por el número de cambios a exportar. El *COMMIT-ID* indica el primer cambio a exportar, es decir, el cambio de referencia.

Para aplicar la diferencia incremental que se ha recibido de otro desarrollador, al igual que en el caso anterior se utilizara una ventana de comandos en la cual habrá que dirigirse al directorio del proyecto y a la rama en caso de que fuera necesario. Finalmente, se utilizara el siguiente comando:

* % git am 0001-cambio.patch

Esto valida que el fichero contenga una diferencia incremental y la aplica al repositorio local.

* % git am –-abort

Se debe ejecutar este comando en caso de que se produzca algún error a la hora de importar el fichero.

Con el fin de ilustrar y dejar aún más claro cómo y por qué se lleva a cabo esta exportación e importación de cambios, en la sección de ejercicios se añade un ejemplo en el que surge la necesidad de usar lo descrito en este apartado.

* 1. **Roles**

Para llevar a cabo la gestión del código fuente del subsistema Frontend de Resultados, se diferencian dos roles principales:

* Desarrollador: Persona que se encarga de realizar y completar las partes del código fuente que le han sido asignadas, haciendo *push* de manera regular a la rama en la que esté trabajando. En el caso en el considere oportuno realizar un *merge*, debe ponerse en contacto con el administrador del repositorio.
* Administrador: Persona con la capacidad de autorizar los *merge* entre las distintas ramas del repositorio, además de poseer todos los privilegios como desarrollador.

Todos los integrantes del subsistema tienen el rol de desarrollador, excepto José Antonio Fernández Bueno, que ejerce de administrador.

El procedimiento para realizar un *merge* es bastante simple: el desarrollador o conjunto de desarrolladores que desee realizar esta acción, debe en primer lugar hacer *push* a la rama sobre la que está trabajando. Una vez hecho esto, le envía un correo electrónico al administrador, informándole de que desea que esa rama sea unida con la que se encuentra en el nivel superior. Entonces, el administrador realiza un *pull*, para validar el estado de dicha rama y decidir si está en condiciones de ser unida con la rama superior o no. En caso afirmativo, el administrador hace el *merge* de la rama tras responder positivamente al solicitante. Si por el contrario no se acepta la solicitud, el administrador envía al solicitante una respuesta negativa, añadiendo el motivo por el cual se ha rechazado. En el caso excepcional de que se autorice el *merge*, y al realizarlo se produzca un conflicto, el administrador sería el encargado de solucionarlo, ya que ha sido el que se ha equivocado al validar dicho *merge*.

* 1. **Conflictos**

En lo que respecta a la gestión de conflictos, al comenzar el proyecto se implantó una política para gestionarlos y evitar, en la medida de lo posible, que se produjesen, ya que al ser una gran cantidad de personas trabajando en un proyecto con muy pocas líneas de código, existía una probabilidad alta de colisionar.

En primer lugar, se definió un procedimiento base para la interacción con el repositorio, que consta de los siguientes procesos:

1. Realizar un ***git clone*** sobre el proyecto de GitHub. Esto se realiza la primera vez, cuando no se tiene aún una copia del repositorio en local.
2. Modificar los ficheros oportunos.
3. Realizar un ***git pull*** para actualizar la versión local.
4. Realizar un ***git add*** con todos los ficheros que se quieran incluir en el *commit*.
5. Realizar un ***git commit*** añadiendo título y descripción para guardar los cambios en el repositorio local.
6. Realizar un ***git push*** de la rama sobre la que esté trabajando.

Además de este procedimiento, se tomaron otras medidas, como la mejora en la comunicación entre los integrantes. Para ello se llegó al acuerdo de que en el momento de empezar a trabajar sobre una parte del código, esto se comunicaría a través de WhatsApp al equipo, para minimizar la probabilidad de trabajar sobre el mismo aspecto de manera concurrente. Del mismo modo, se acordó informar de la creación de nuevas ramas, así como del contenido de la misma.

En lo que respecta a los *checkouts*, éstos se fijaron como no reservados, ya que al estar informando continuamente de la parte en la que se está trabajando, se aceptó que más de una persona pudiese modificar partes del mismo archivo.

No obstante, si aun así se produjese algún conflicto, se acordó que el desarrollador causante de dicho conflicto, es decir, la persona a la que le aparece el error a la hora de hacer *push,* sería el encargado de resolverlo línea a línea de forma que se conserve la integridad y funcionalidad de la aplicación. Un ejemplo de esto podrá verse más adelante, en la sección de ejercicios.

* 1. **Migración de SVN a Git**

[Álvaro]

* 1. **Herramientas similares**

[Adrián]

* 1. **Ejercicios**

A continuación se enuncian ejercicios donde se ponen en práctica las decisiones tomadas por el grupo en cuanto a la gestión del código fuente antes expuestas y la interacción de los integrantes del grupo con el repositorio donde se guarda dicho código. Están centrados en Git y dentro de la resolución de los mismos a veces se han realizado ciertas operaciones como los merge para ilustrar la solución directamente en vez de tener que cambiar al usuario del administrador. Aún así estos casos se especifican en las resoluciones de los mismos ejercicios. Además, se utilizarán ramas del tipo *feature* para realizar los ejercicios, exceptuando el de la aplicación del parche.

**Ejercicio 1: Añadir nueva característica haciendo uso de ramas**

Se desea implementar una página de bienvenida que muestre un mensaje que especifique el nombre del subsistema y el sistema al que pertenece (Agora@US). Obviamente, los cambios realizados en la aplicación deben quedar registrados en el sistema de control de versiones donde se almacena el código fuente de la aplicación.

**Resolución**

En primer lugar, si no se dispone del repositorio en la máquina donde se vayan a realizar las labores de codificación, se deben obtener realizando un *git clone* a la dirección del repositorio en GitHub, *git@github.com:EGC-FrontEnd-Resultados/code.git* usando SSH o *https://github.com/EGC-FrontEnd-Resultados/code.git* en el caso de HTTPS.

En el caso de disponer de dicho repositorio, se realizará un *git pull* para asegurar que el repositorio local esté actualizado con los últimos cambios del repositorio en línea.

Una vez obtenida la copia del repositorio en la máquina local lo primero que se debe realizar es crear una rama llamada “Welcome\_Dev”, que será sobre la que se trabajará mientras se esté implementando esta característica. Tras esto, el desarrollador debe situarse en dicha rama y debe enviarla al servidor.

A continuación, se trabaja normalmente con el entorno de desarrollo. Se deben marcar los ficheros que vayan a ser modificados mediante el uso de *git add*, tras lo cual se realizarán los *git commits* pertinentes cuyos comentarios tendrán la estructura especificada en la sección de gestión del cambio. Si los ficheros ya se encontraban bajo el control de versiones no es obligatorio utilizar el comando *git add*.

Tras esto, se debe realizar un *git push* para enviar los cambios que hayan sido realizados en el repositorio local.

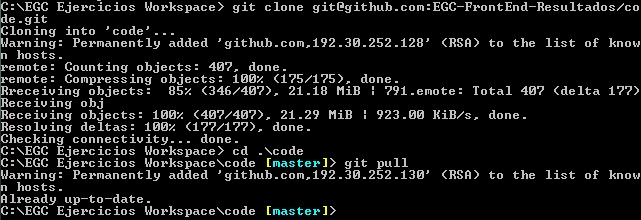
Cuando se haya implementado completamente la nueva funcionalidad y se haya notificado al administrador, este mismo será el encargado de hacer un *git merge* de la rama “Welcome\_Dev” con la rama “master” del repositorio (en caso de que haya dado el visto bueno), tras lo cual debe realizar un último *git push* para que los cambios se apliquen al repositorio.

**Secuencia de comandos**

A continuación se puede observar cuál es la secuencia exacta de los comandos de git que se deben utilizar para realizar este ejercicio (algunos comandos como *git log* o *git status* mostrados en las capturas no son necesarios):

***git clone*** *git@github.com:EGC-FrontEnd-Resultados/code.git*

***git pull***

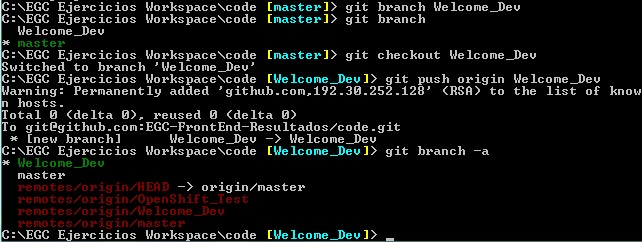
******

*Figura X: Ejercicio Git 1: Uso de ramas 1*

***git branch*** *Welcome\_Dev*

***git checkout*** *Welcome\_Dev*

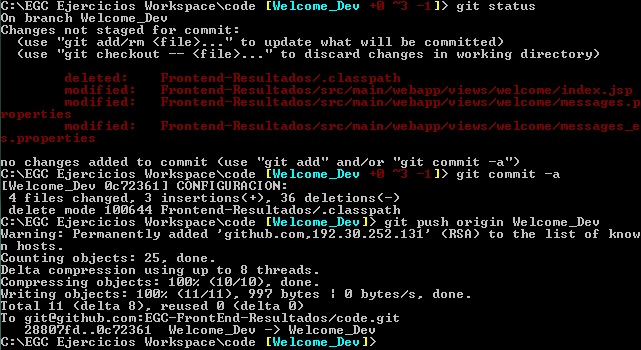
***git push*** *origin Welcome\_Dev*

******

*Figura X: Ejercicio Git 1: Uso de ramas 2*

***git commit*** *–a* (Tras realizar todos los cambios necesarios)

***git push*** *origin Welcome\_Dev*

******

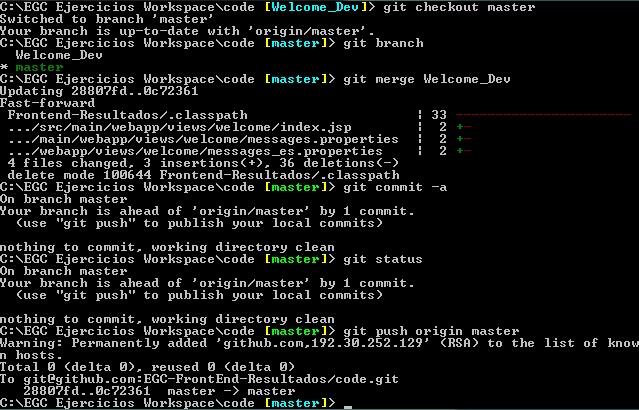
*Figura X: Ejercicio Git 1: Uso de ramas 3*

***git checkout*** *master*

***git merge*** *Welcome\_Dev*

***git commit*** *–a*

***git push*** *origin master*

******

*Figura X: Ejercicio Git 1: Uso de ramas 4*

**Ejercicio 2: Resolución de conflictos**

Este ejercicio parte del supuesto en el que otro de los desarrolladores del grupo se dispone a implementar la característica descrita en el siguiente enunciado al mismo tiempo, por lo que se producirá un conflicto cuando se realice el último merge. En nuestro caso, durante el desarrollo del ejercicio forzaremos que se produzca el conflicto utilizando dos ramas distintas para implementar la misma funcionalidad variando un poco el código. El enunciado es el siguiente:

Se desea implementar en el subsistema varios enlaces dirigidos a las páginas de bienvenida de los subsistemas censo y modificación, y a la wiki de la asignatura. Esto debe realizarse en la página de bienvenida del subsistema. Por supuesto, los cambios realizados en la aplicación deben quedar registrados en el sistema de control de versiones donde se almacena el código fuente de la aplicación.

Las URL de dichos subsistemas son: “/ADMCensus”, “/Modificacion”

URL de la wiki de la asignatura de Evolución y Gestión de la Configuración: <https://1984.lsi.us.es/wiki-egc/index.php/P%C3%A1gina_Principal>

**Resolución**

En primer lugar, si no se dispone del repositorio en la máquina donde se vayan a realizar las labores de codificación, se deben obtener realizando un *git clone* a la dirección del repositorio en GitHub, *git@github.com:EGC-FrontEnd-Resultados/code.git* usando SSH o *https://github.com/EGC-FrontEnd-Resultados/code.git* en el caso de HTTPS.

En el caso de disponer de dicho repositorio, se realizará un *git pull* para asegurar que el repositorio local esté actualizado con los últimos cambios del repositorio en línea.

Una vez obtenida la copia del repositorio en la máquina local lo primero que se debe realizar es crear una rama llamada “Links\_Dev1”, sobre la que se trabajará para implementar esta nueva característica. Tras esto, el desarrollador debe situarse en dicha rama y debe enviarla al servidor.

Por otro lado, debe crearse otra rama llamada “Links\_Dev2”, que simulará la rama que habría usado el segundo programador.

A continuación, se trabaja normalmente con el entorno de desarrollo tal y como se ha descrito en el ejercicio anterior (nos encontramos en este momento en la rama “Links\_Dev1”).

Tras esto, se debe realizar un *git push* para enviar los cambios que hayan sido realizados en el repositorio local.

Cuando se haya implementado completamente la nueva funcionalidad se realizará el merge entre las ramas master y “Links\_Dev1”.

Ahora, nos situamos en la rama “Links\_Dev2” y repetimos el proceso cambiando el orden de los links en las líneas de código que fueron modificadas en “Links\_Dev1”. A la hora de realizar el merge entre “master” y “Links\_Dev2” tendremos un conflicto.

Para resolver el conflicto simplemente se deberán editar los ficheros que estén en conflicto, añadiendo a la versión final del archivo los cambios realizados que antes no estaban, es decir, en el resultado final se podrá ver la descripción del ejercicio anterior y los links añadidos durante este ejercicio en la página de bienvenida de la aplicación.

Siguiendo la política establecida por el grupo, estos últimos pasos a partir del *git merge* deberán ser realizados por el administrador de la manera anteriormente descrita.

**Secuencia de comandos**

A continuación se puede observar cuál es la secuencia exacta de los comandos de git que se deben utilizar para realizar este ejercicio:

***git pull***

***git branch*** *Links\_Dev1*

***git checkout*** *Links\_Dev1*

***git push*** *origin Links\_Dev1*

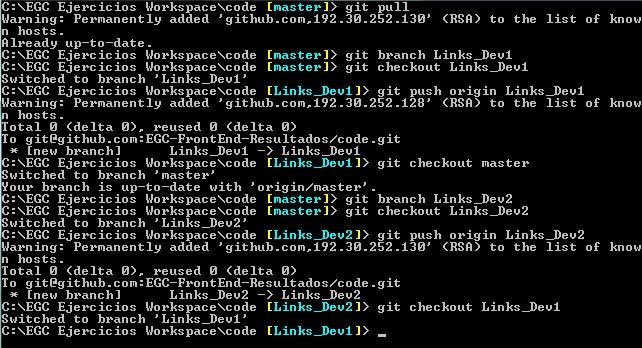
***git checkout*** *master*

***git branch*** *Links\_Dev2*

***git ckeckout*** *Links\_Dev2*

***git push*** *origin Links\_Dev2*

***git checkout*** *Links\_Dev1*

******

*Figura X: Ejercicio Git 2: Resolución de conflictos 1*

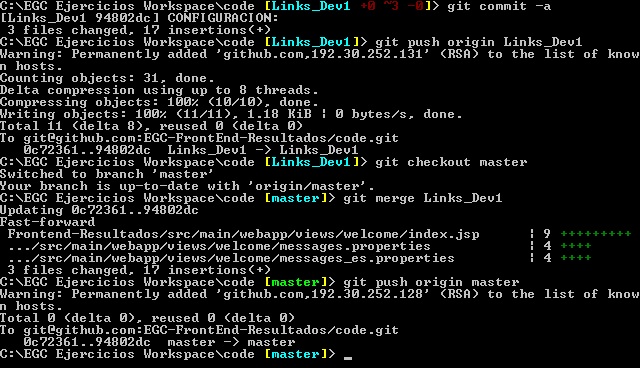
***git commit*** *–a* (Tras realizar todos los cambios en Links\_Dev1)

***git push*** *origin Links\_Dev1*

***git checkout***  *master*

***git merge*** *Links\_Dev1*

***git push*** *origin master*

******

*Figura X: Ejercicio Git 2: Resolución de conflictos 2*

***git checkout*** *Links\_Dev2*

***git commit*** *–a*

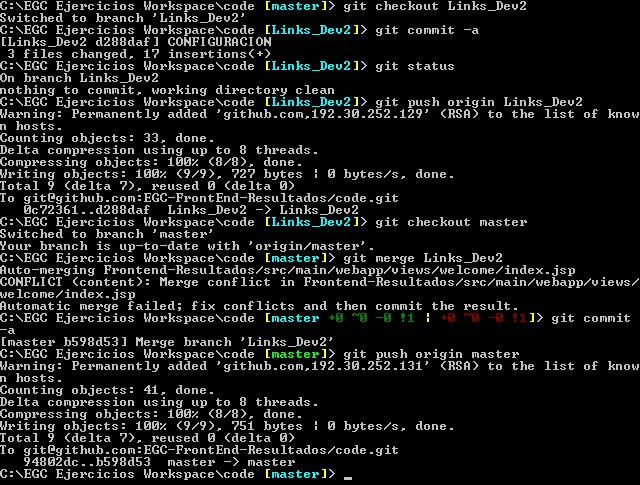
***git push*** *origin Links\_Dev2*

***git checkout*** *master*

***git merge*** *Links\_Dev2*

***git commit*** *–a* (Tras resolver el conflicto)

***git push*** *origin master*

******

*Figura X: Ejercicio Git 2: Resolución de conflictos 3*

**Ejercicio 3: Creando y aplicando un parche**

Uno de los miembros del equipo se ha quedado sin internet en casa, y necesita importarse los cambios realizados en el repositorio. En este caso el cambio consiste en la adición de un enlace que redirige a la página web de la escuela de Ingeniería Informática. Así pues, otro miembro debe crear un *patch* que contenga la diferencia incremental deseada, y el receptor deberá aplicarlo. Con el fin de agilizar el proceso, para la resolución del ejercicio se trabajará en dos ramas diferentes en el mismo ordenador, ya que la manera de hacerlo no variaría si realmente se tuviesen dos ordenadores diferentes.

**Resolución**

En primer lugar, si no se dispone del repositorio en la máquina donde se vayan a realizar las labores de codificación, se deben obtener realizando un *git clone* a la dirección del repositorio en GitHub, *git@github.com:EGC-FrontEnd-Resultados/code.git* usando SSH o *https://github.com/EGC-FrontEnd-Resultados/code.git* en el caso de HTTPS.

En el caso de disponer de dicho repositorio, se realizará un *git pull* para asegurar que el repositorio local esté actualizado con los últimos cambios del repositorio en línea.

A continuación, lo primero que se hará será crear una rama donde se realicen los cambios en el proyecto que pasarán a formar el parche. Tras esto, el desarrollador debe situarse sobre esta rama.

Ya en la nueva rama se realizan los cambios deseados en el proyecto y se agregan utilizando *git add* y *git commit*.

Se pueden comprobar los cambios que se han realizado haciendo uso del comando *git diff*, tras lo cual se procederá a crear el parche haciendo uso del comando *git format-patch*.

Para aplicar el parche y comprobar que funciona realmente, el desarrollador debe posicionarse en la rama master del repositorio y borrar la rama de la cual se ha realizado el parche (eso último simplemente es para asegurar que los cambios introducidos con la aplicación del parche no “existen” dentro de ningún sitio del repositorio pero sí en el parche).

A continuación, es buena idea revisar cuáles son los cambios que se realizarán con la aplicación del parche y si el aplicarlo puede conllevar la generación de algún error, esto se realiza con el comando *git apply.*

Por último aplicamos el parche, haciendo uso nuevamente del comando *git apply*. En el caso de que se quiera revertir la aplicación del parche habría que usar el comando *git apply -R*.

En el contexto de nuestro grupo, el parche generado sería enviado al administrador que sería el encargado de aplicarlo después de decidir si es o no válido. En la resolución del ejercicio se ha hecho directamente para ilustrar directamente la creación y aplicación de un parche en Git. Además, a diferencia del tipo de parche descrito anteriormente, en este ejercicio se ha usado otro tipo de parche que aglutina todos los commits realizados hasta la generación del parche.

**Secuencia de comandos**

***git pull***

***git branch*** *Parche\_Dev*

***git checkout*** *Parche\_Dev*

***git commit*** *–a*

***git format-patch*** *master --stdout > parche.patch*

***git checkout*** *master*

***git branch*** *–D Parche\_Dev*

***git apply*** *--stat parche.patch*

***git apply*** *--check parche.patch*

***git apply*** *parche.patch*

***git commit*** *–a*

***git push*** *origin master*

******

*Figura X: Ejercicio Git 3: Patch*

# Gestión de la construcción y despliegue

En lo que se refiere a la construcción y al despliegue, existe una serie de herramientas que automatizan estos procesos, como son Maven, Rake o Apache Ant. Si esto se quiere realizar en la nube, construyendo y desplegando directamente la aplicación en un servidor gratuito, una buena opción es OpenShift. Para el caso del subsistema de Frontend de Resultados, se ha optado por Maven y OpenShift, y su uso y características se explican en los siguientes apartados.

* 1. **Maven**

[Álvaro: mirar grupos 1 y 9]

* 1. **Openshift**

[Waly: recuerda explicar primero en general para qué sirve y qué ofrece, y luego en nuestro caso los pasos que se han realizado para conseguir desplegarlo]

* 1. **Ejercicios**

[Álvaro: mirar grupos 9 y 1 (ejercicio de Maven, no de OpenShift)]

# Gestión de la construcción e integración continua

A la hora de llevar a cabo la compilación y ejecución de pruebas del proyecto, es recomendable que esto se realice de manera automática y periódica, para detectar posibles errores introducidos en el código fuente lo antes posible. A este proceso de descargar el código fuente desde el control de versiones, compilarlo, ejecutar pruebas y generar informes se le denomina integración continua.

Para ayudar en la pronta detección de errores y en el ahorro de una gran cantidad de tiempo, las herramientas de integración continua utilizan el correo electrónico para notificar a los desarrolladores cuando se producen errores en alguna prueba o proceso.

Algunos autores propugnan una serie de buenas prácticas para conseguir realizar la integración continua. Entre ellas encontramos:

* Tener un único repositorio de código.
* Automatizar la construcción, es decir, que un solo comando sea capaz de construir el sistema.
* Después de cada construcción se debe comprobar que ésta se ha realizado correctamente.
* Toda persona que esté trabajando en el sistema debe hacer *commit* a la rama *master* cada día, con el fin de minimizar los conflictos.
* Después de cada *commit* se debe construir el sistema verificando la correcta integración del cambio.
* Automatizar el despliegue, consiguiendo así probar el sistema en un servidor de prueba. El siguiente paso de esta práctica sería el despliegue continuo, que despliega de manera periódica el sistema detectando errores.

Aunque para este proyecto se ha usado Jenkins para llevar a cabo la integración continua, existen otras opciones, como Bamboo, Continuum, o Hudson.

Para el caso concreto del subsistema Frontend de Resultados, se observa que la envergadura del mismo es bastante reducida, lo cual tienta a que su desarrollo se realice de un modo más rudimentario y sin control. Sin embargo, se han seguido todas las buenas prácticas anteriormente expuestas, como se podrá observar a continuación, en la sección específica sobre el uso de Jenkins.

* 1. **Jenkins**

Como se ha mencionado anteriormente, Jenkins es un *software* *open source* de integración continua desarrollado en Java. Se trata de una herramienta muy popular, por lo que existen multitud de *plugins* para incorporarle, además de una extensa documentación de la misma. Todos estos aspectos, unidos a las prácticas realizadas sobre Jenkins en la asignatura, han sido la causa principal por la cual se ha decidido darle uso. A continuación, se expone paso a paso cómo se ha configurado, así como las tareas que han sido automatizadas para el subsistema de Frontend de Resultados.

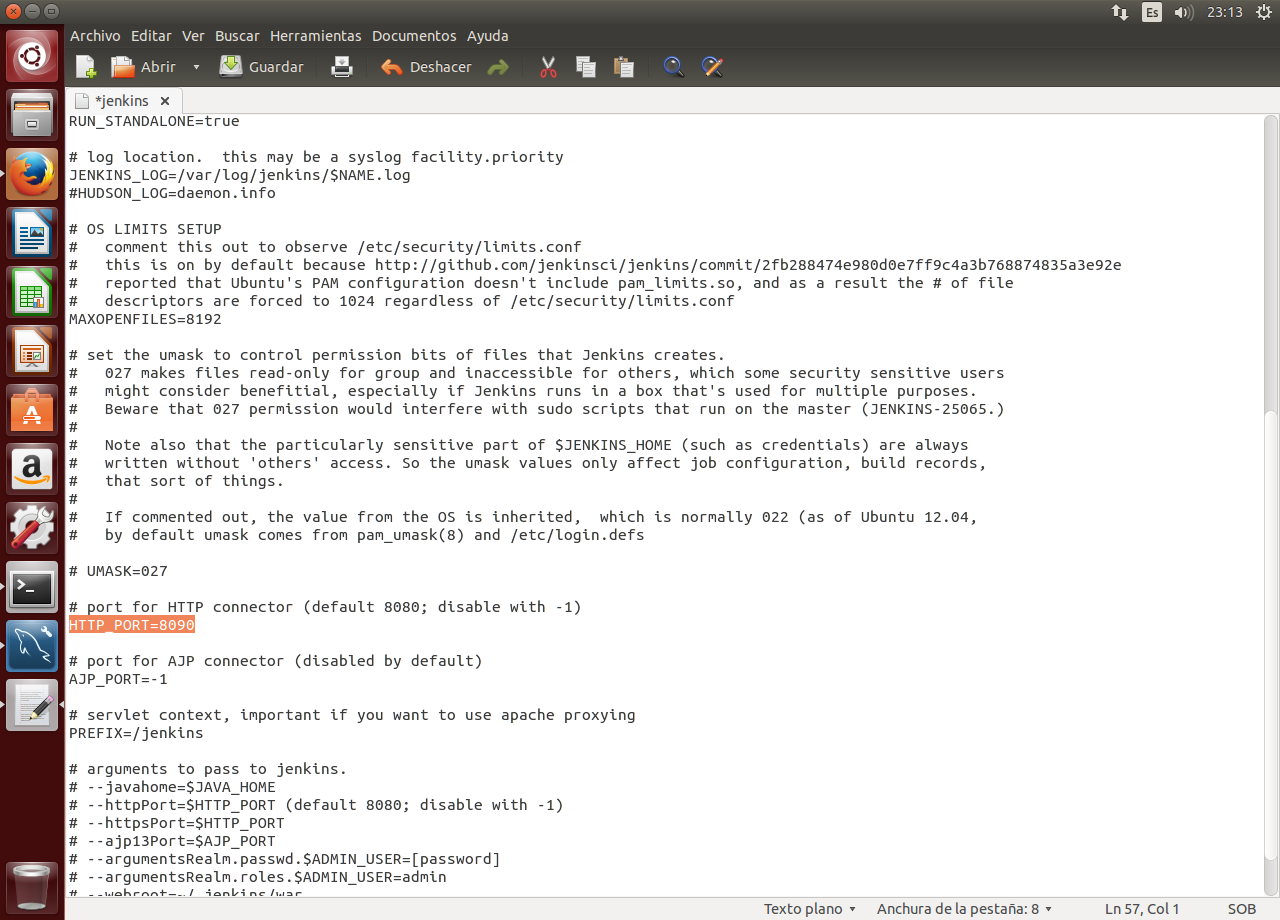
* + 1. **Herramientas necesarias**

Todo lo que se explica a continuación se ha realizado sobre el sistema operativo Ubuntu 14.04 con MySQL preinstalado. La instalación de MySQL se explica en la sección de gestión de la calidad, en la subsección “Proceso de instalación de SonarQube y componentes asociados”.

En primer lugar, es necesario instalar Jenkins en el equipo. Para ello, únicamente se necesita acceder una ventana de terminal e introducir la siguiente secuencia de comandos (todos los comandos de aquí en adelante se deben ejecutar como root):

* **wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -**
* **sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.d/jenkins.list'**
* **apt-get update**
* **apt-get install jenkins**

Una vez hecho esto, se procede a configurar Jenkins en un puerto diferente al que usa Tomcat. Para modificar el puerto, se debe acceder al fichero llamado “Jenkins” ubicado en la ruta etc/default y cambiar la línea “HTTP\_PORT=8080” por “HTTP\_PORT=8090”, como puede verse en la siguiente captura:



*Figura X: Modificación del puerto para Jenkins*

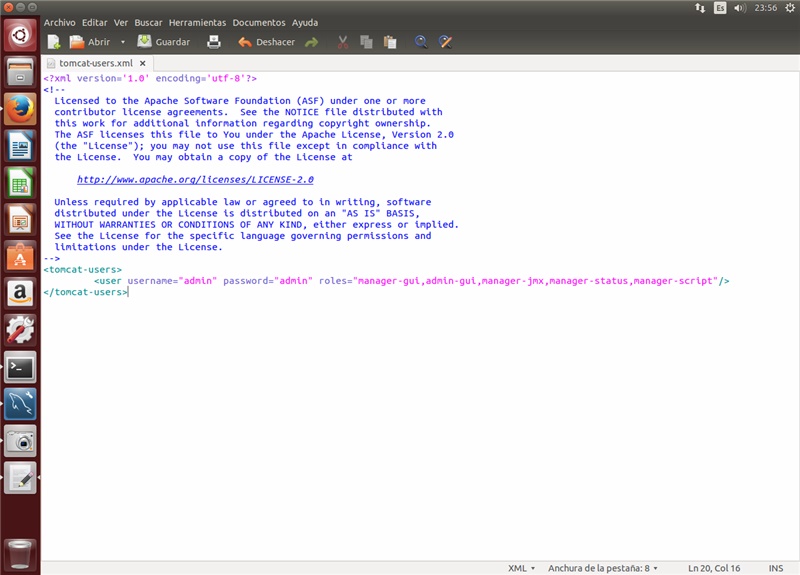
Una vez reiniciado Jenkins, este cambio se hará efectivo. Para reiniciarlo, se puede usar el siguiente comando:

* **service jenkins restart**

Lo siguiente que hay que hacer es instalar Tomcat7, introduciendo los comandos:

* **apt-get install tomcat7**
* **apt-get install tomcat7-docs tomcat7-admin tomcat7-examples**

Una vez instalado, hay que configurar el usuario con el que Jenkins desplegará los proyectos. El archivo de configuración a modificar se encuentra en /etc/tomcat7 y su nombre es “tomcat-users.xml”. Dicho fichero debe quedar como se ve en la siguiente captura de pantalla:



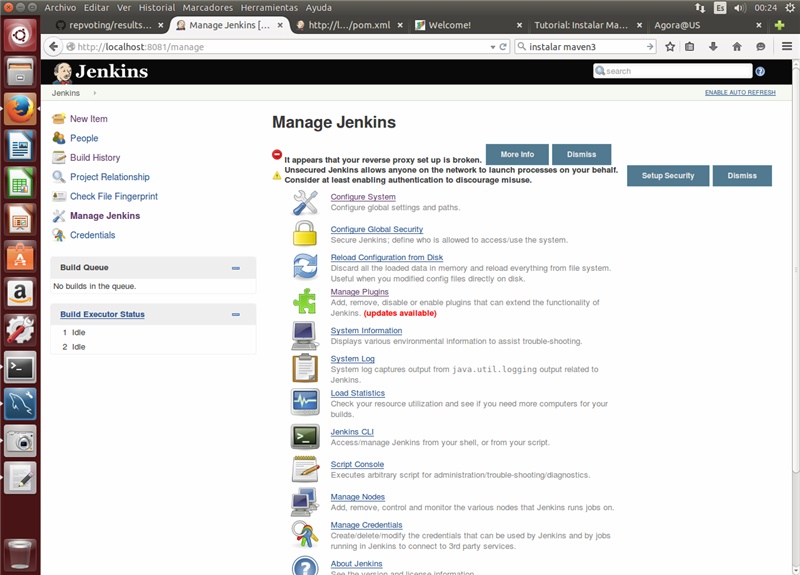
*Figura X: Modificación del fichero de configuración de Tomcat*

El siguiente paso consiste en la instalación de Maven, lo cual se realiza con un solo comando:

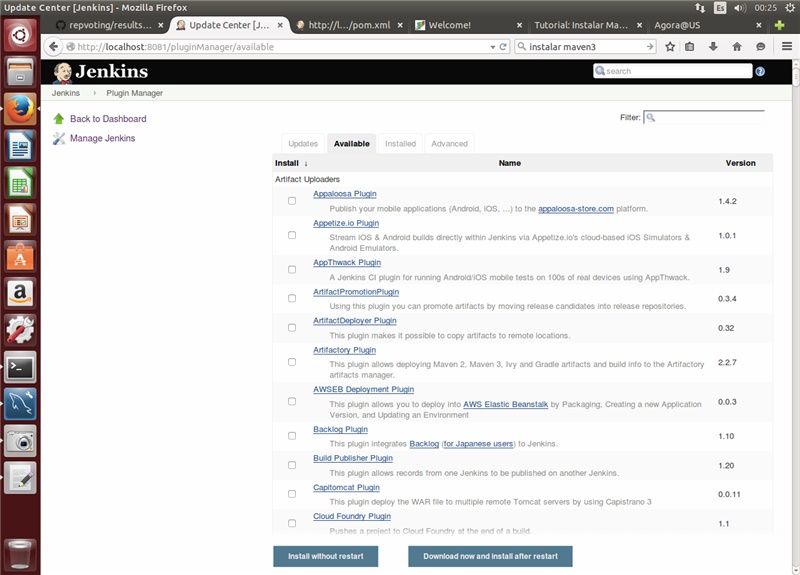
* **apt-get install maven**

Tras realizar todos estos pasos, ya quedaría configurado el entorno, a falta de instalar los siguientes *plugins* de Jenkins: *GitHub plugin, Deploy plugin*

La instalación de plugins en Jenkins es bastante simple e intuitiva, ya que se realiza desde la propia interfaz gráfica. Basta con hacer click en “*Manage Jenkins*” y posteriormente en “*Manage Plugins*”.

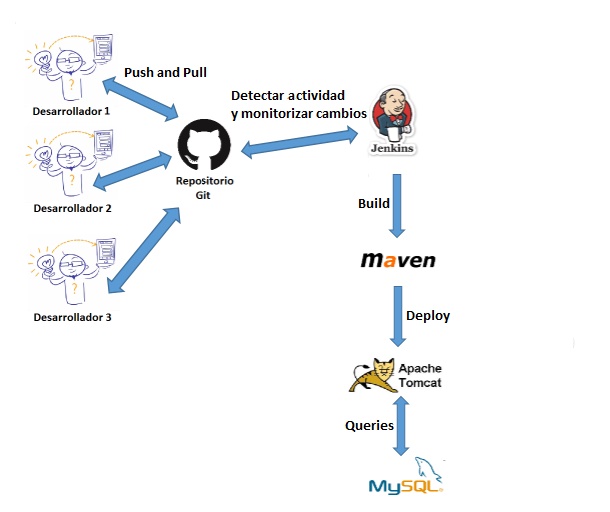


*Figura X: Vista tras pulsar “Manage Jenkins”*



*Figura X: Lista de plugins de Jenkins*

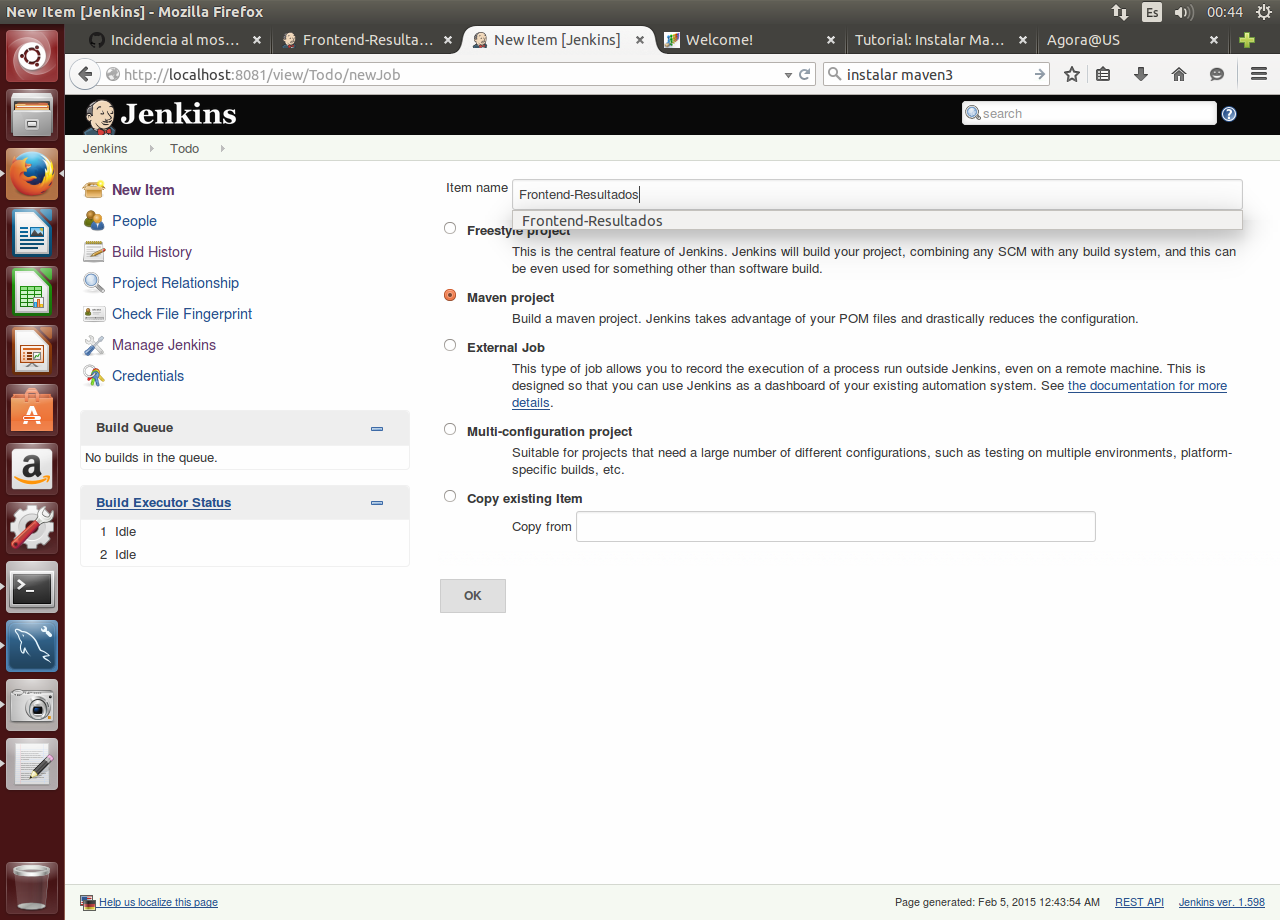
Una vez instaladas todas estas herramientas se puede comenzar a usar Jenkins con normalidad. La arquitectura resultante se muestra a continuación:



*Figura X: Arquitectura resultante tras la instalación de Jenkins*

* + 1. **Tareas realizadas**

Jenkins ofrece la posibilidad de automatizar un conjunto de pruebas o comprobaciones, recibiendo el nombre de tareas. Para crear tareas en Jenkins hay que hacer click en la opción “New Item”, tras lo cual aparecerá una ventana como esta, en la que hay que seleccionar el tipo de tarea que se desee crear:



*Figura X: Pantalla de elección del tipo de tarea*

Tras esto, aparece una ventana en la que se configuran las características de la tarea. Lo primero que hay que especificar es la ubicación en la que se encuentra alojada el proyecto en cuestión, como se puede observar en la siguiente captura.



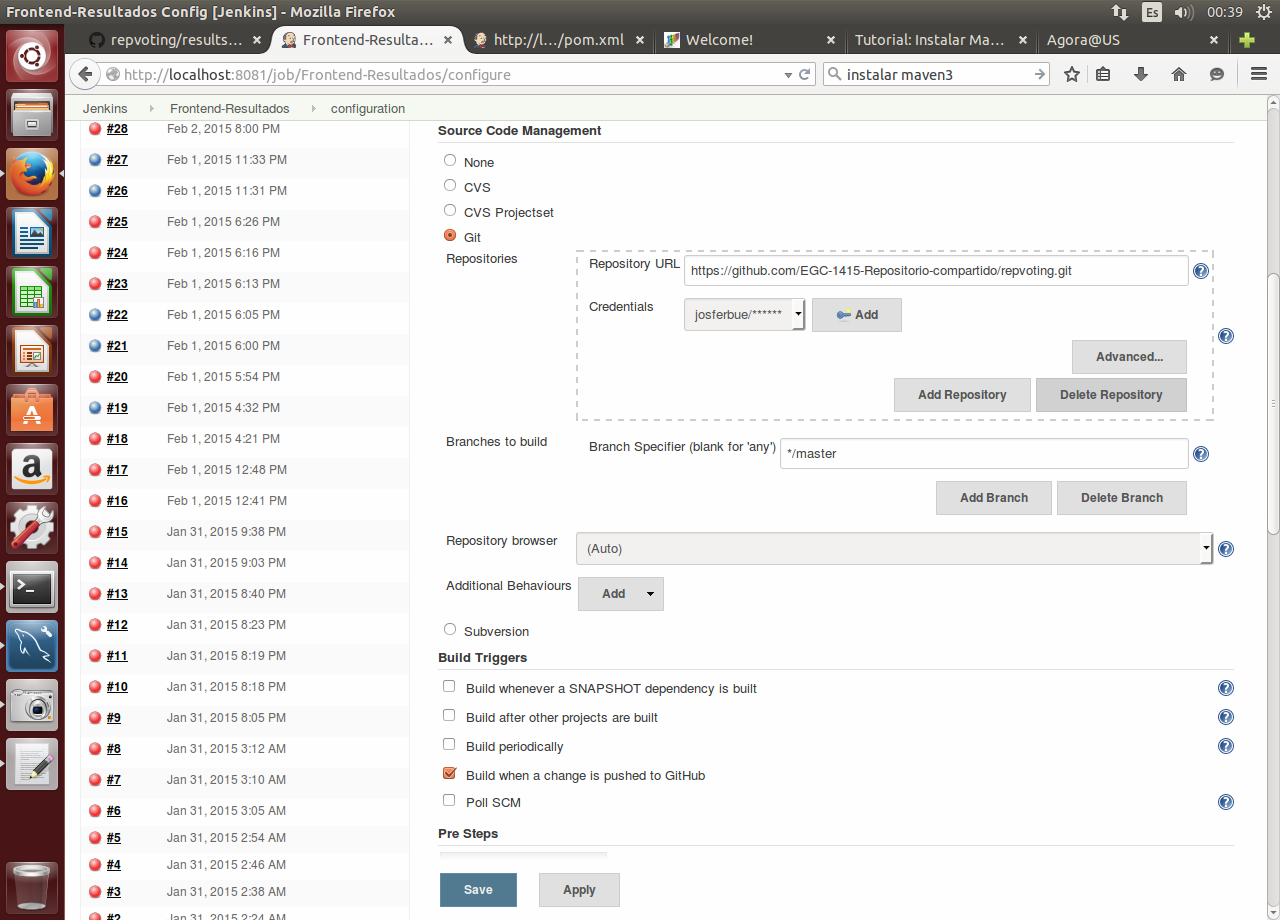
*Figura X: Configuración de tareas en Jenkins (I)*

A continuación, se introduce la dirección HTTPS del repositorio de Git y las credenciales para acceder a él, si fueran necesarios.



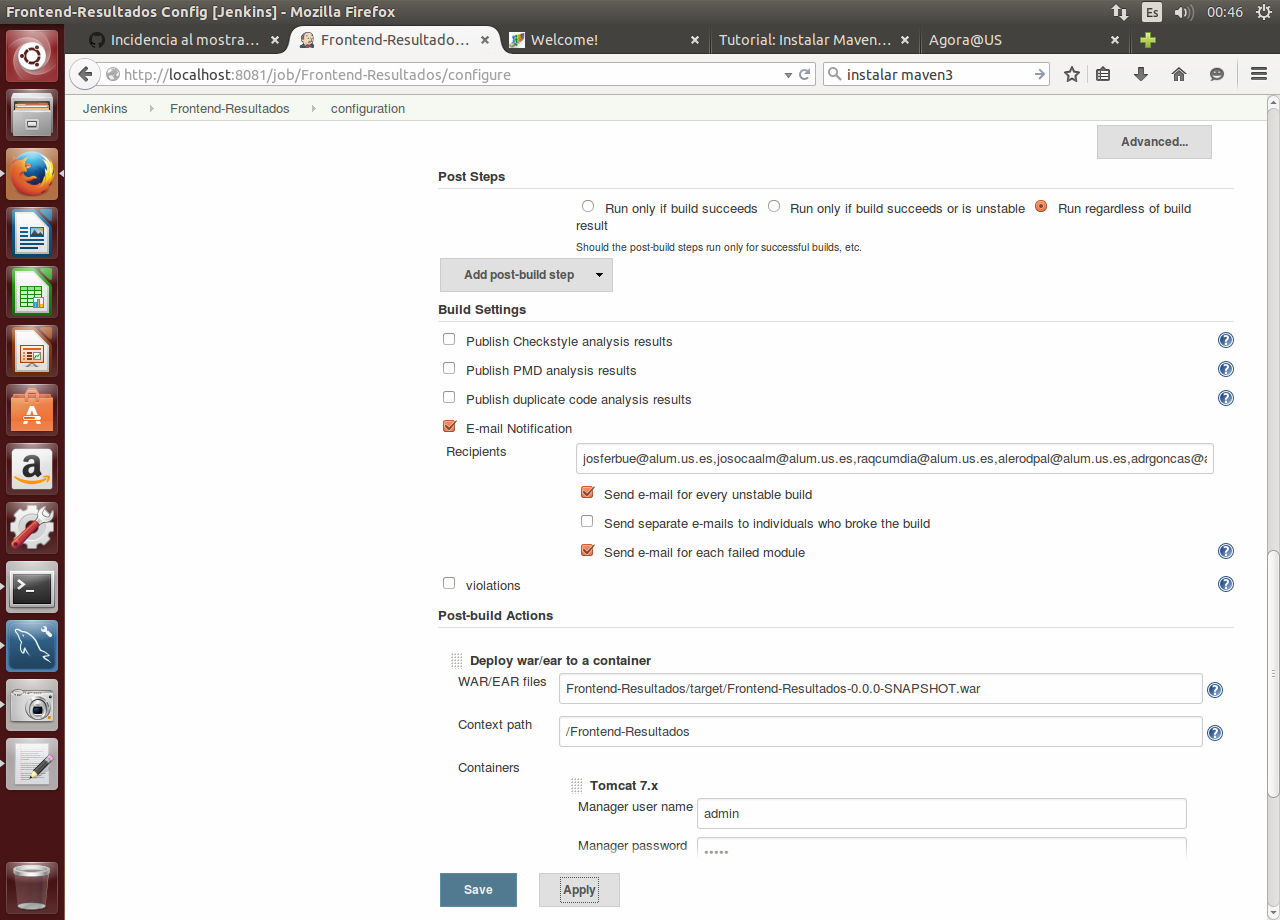
*Figura X: Configuración de tareas en Jenkins (II)*

Tras esto, se especifica con qué frecuencia se desea que sea ejecutada la construcción del proyecto. En este caso, se realizará cada vez que se haga un *push* al repositorio.



*Figura X: Configuración de tareas en Jenkins (III)*

Lo siguiente que hay que configurar son las opciones de construcción y las acciones posteriores a la misma. Como se puede observar, cuando se produzca alguna irregularidad a la hora de construir el proyecto, se informará por correo electrónico a las direcciones especificadas. En cuanto a las acciones posteriores a la construcción, en este caso de desea que se despliegue el proyecto en Tomcat. El primer parámetro indica la ruta del archivo WAR, mientras que en el segundo se introduce la ruta de Tomcat donde se desplegará el WAR.

**

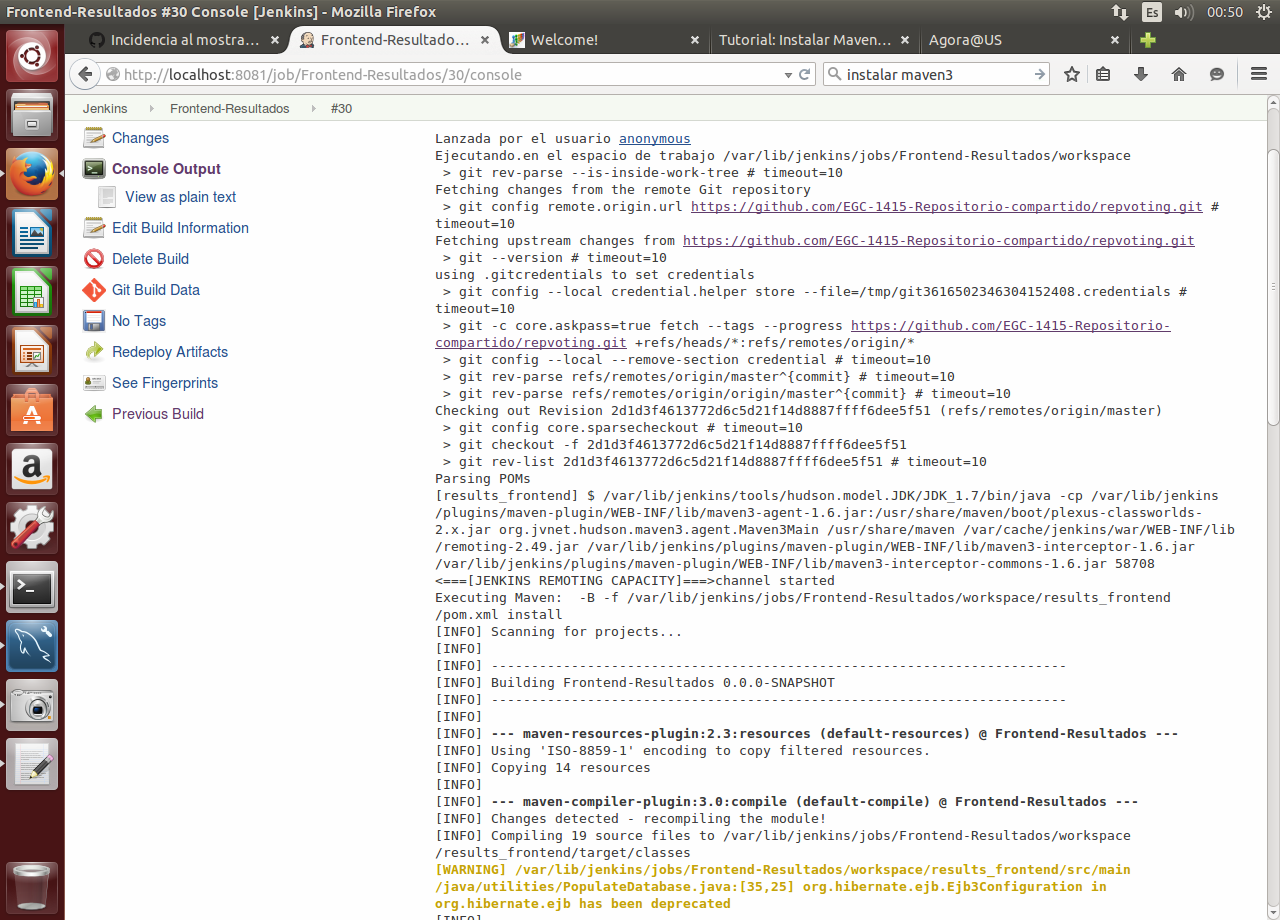
*Figura X: Configuración de tareas en Jenkins (IV)*

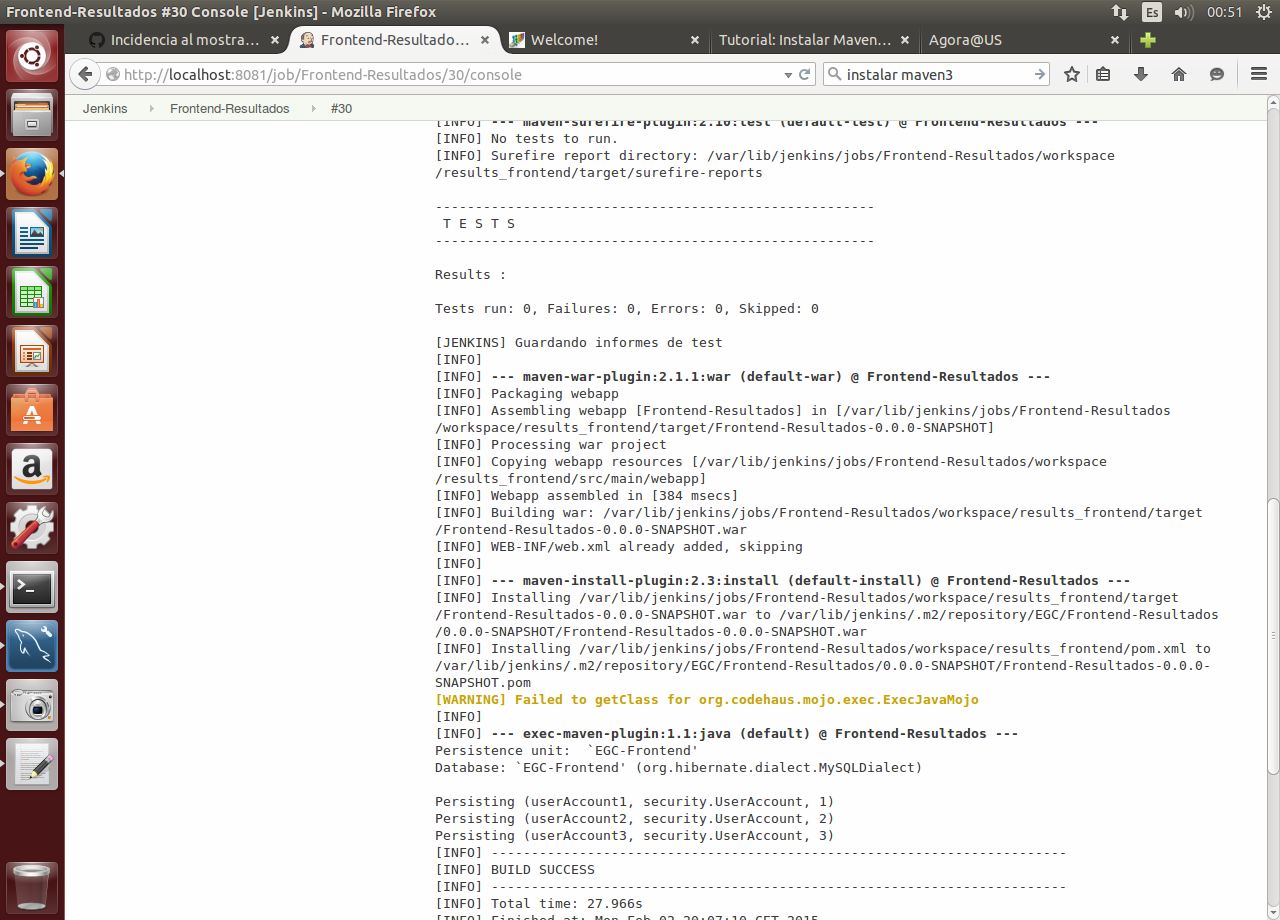
Una vez realizados todos estos pasos, quedaría terminada la creación y configuración de la tarea.

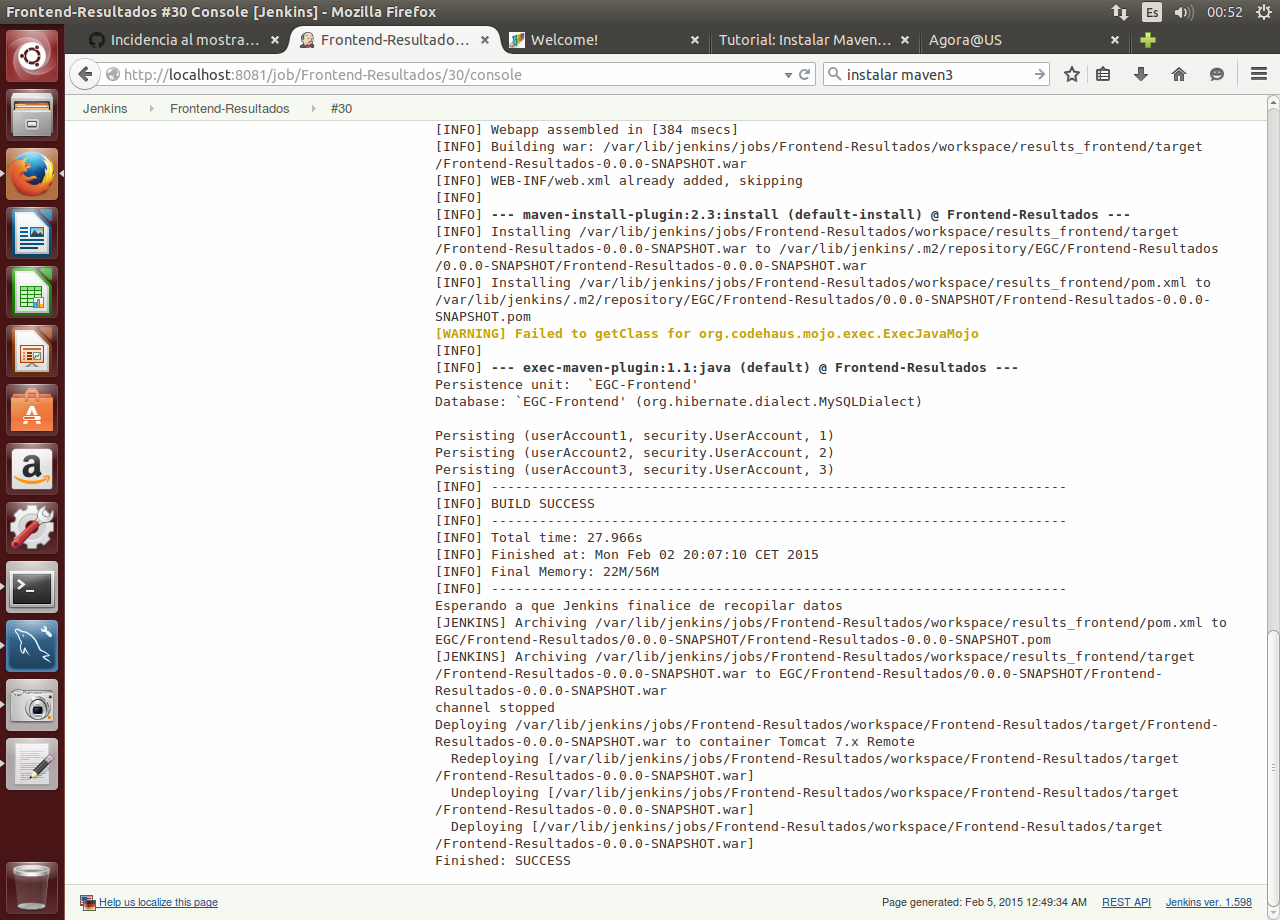
En lo que respecta al subsistema de Frontend de Resultados, la tarea que se realiza incluye las siguientes acciones:

* Descarga del código desde el repositorio de Git
* Descarga de las dependencias de Maven
* Ejecución de los test que haya dentro del proyecto
* Se popula la base de datos de MySQL
* Generación del fichero WAR
* Despliegue del proyecto en Tomcat

A continuación se añaden tres capturas de pantalla con los resultados obtenidos tras ejecutar esta tarea:







*Figura X: Resultado de ejecución de la tarea en Jenkins*

Con esto se daría por finalizada la explicación general del uso de Jenkins, así como la aplicación particular para el subsistema de Frontend de Resultados. Cabe destacar que esta configuración concreta es válida para cualquier proyecto Maven.

* 1. **Ejercicio**

El contenido de este sub apartado ha sido trasladado a la sección de integración con todos los subsistemas, en concreto al punto “Integración continua”, en el cual se explica cómo se han configurado todos y cada uno de los subsistemas en Jenkins, lo cual se considera suficiente como ejercicio para esta parte.

# Gestión de la calidad

La gestión de la calidad hace referencia a las estrategias y procedimientos seguidos para verificar que se siguen una serie de buenas prácticas a la hora de desarrollar el código. Es importante considerar estos aspectos de calidad con el fin de garantizar que el código desarrollado posee la calidad que se espera.

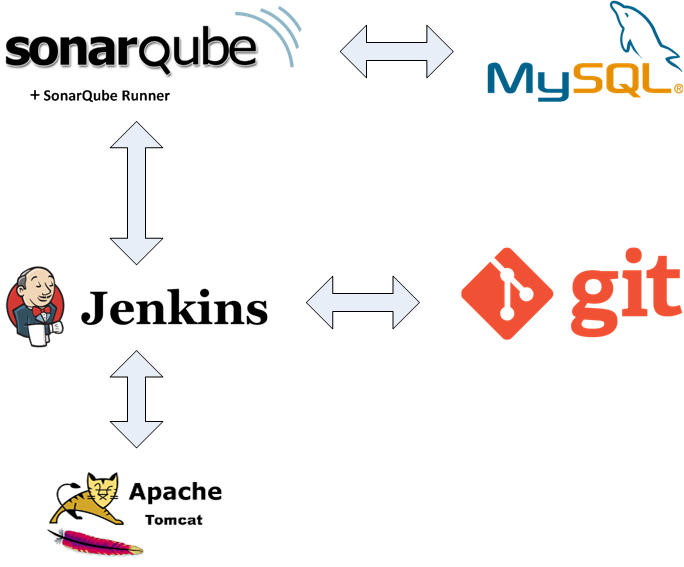
Para facilitar esta tarea se utilizará una herramienta especializada en análisis de código estático. Dicha herramienta es **SonarQube**, aunque normalmente se la conoce como **Sonar**. Con el objeto de realizar el análisis del código, Sonar utiliza un conjunto de reglas definidas para comprobar que el código de determinado proyecto ha sido desarrollado siguiendo las buenas prácticas ya mencionadas. Si cierto fragmento de código no cumple una o más reglas, Sonar generará un listado en el cual se especificará qué reglas ha incumplido cada uno de dichos fragmentos, además de especificar cuánto tiempo supondría realizar los cambios oportunos en cada fragmento con el fin de que las reglas sean cumplidas. Adicionalmente, puede asignarse cada una de estas tareas a uno de los miembros involucrados en el proyecto. Por último, y no menos importante, también se especifica la deuda técnica total del proyecto, la cual indica cuánto tiempo deberá emplearse en subsanar las incidencias de código señaladas por Sonar para que todas las reglas sean cumplidas.

En el caso del subsistema Frontend de Resultados, desarrollado por el equipo de trabajo, resulta más interesante lograr la integración de Sonar con Jenkins, puesto que se elimina la ejecución manual del análisis del código del subsistema y sólo es necesario emplear tiempo en interpretar las métricas o resultados obtenidos. Más adelante se profundizará un poco más en el caso particular del subsistema, realizando un análisis de código e interpretando los resultados.

Para llevar a cabo la instalación de Sonar, es necesario utilizar software auxiliar que dé soporte a dicha herramienta. A continuación, se presentan los componentes software necesarios y la relación entre cada uno de ellos.

* 1. **Herramientas utilizadas**

Como ya se ha mencionado, el análisis de código estático se realiza a través de Sonar. No obstante, Sonar necesita ciertas herramientas que le den soporte y conlleven a una ejecución exitosa del mismo. En el siguiente gráfico puede apreciarse la relación entre cada una de las herramientas software utilizadas en la gestión de la calidad del código fuente.



*Figura X: Mapa de herramientas de SonarQube*

A continuación, se detalla el papel que juega cada una de las herramientas en el análisis de código estático llevado a cabo por Sonar:

* ***Java Development Kit***. Es un requisito fundamental y necesario para la ejecución de SonarQube. Es, por tanto, obligatorio llevar a cabo su instalación, tal y como se indica en la documentación oficial de SonarQube. Muy probablemente la herramienta esté basada en Java, de ahí la necesidad de utilizar Java JDK. Se recomienda utilizar Java JDK 1.6 o inferior, debido a problemas de incompatibilidad con Java JDK 1.7. A pesar de que la herramienta no aparece en el mapa de herramientas anterior, esta herramienta tanto por SonarQube como por Apache Tomcat y Jenkins.
* **SonarQube.** Representa el servidor de análisis de código estático. Se encarga de llevar a cabo el análisis de proyectos para obtener datos que reflejen la calidad del código y de coordinar las interacciones con el servidor de bases de datos para solicitar o registrar los resultados de cada uno de estos análisis. Destacar que el análisis es realizado por un componente adicional denominado SonarQube Runner.
* **SonarQube Runner**. Se trata del componente analítico de SonarQube. Permite analizar proyectos codificados en diferentes lenguajes de programación. Por defecto, SonarQube Runner sólo analiza código desarrollado en Java, aunque puede analizar código desarrollado en otros lenguajes de programación añadiendo los *plug-ins* pertinentes.
* **MySQL Server**. Es el servidor de bases de datos en el que se almacenarán los resultados obtenidos a partir del análisis realizado por SonarQube. Como ya se comentó anteriormente, el servidor de análisis de código estático se comunicará con el servidor de bases de datos para almacenar los resultados. También es necesario instalar el cliente correspondiente a MySQL, con el fin de facilitar dicha comunicación entre ambos servidores.
* **Jenkins**. Esta herramienta representa el servidor de integración continua. Realmente no es necesario para llevar a cabo los análisis de código estático, pero el hecho de poder automatizar dichos análisis la convierte en una herramienta extremadamente útil. Será necesario utilizar un *plug-in* que facilite la comunicación entre el servidor de integración continua y el servidor de análisis de código estático. En este caso, hablamos del *plug-in* Sonar Plugin.
* **GIT**. Para poder automatizar la construcción y análisis del proyecto de la asignatura, el cual está almacenado en un repositorio en GitHub, será necesario instalar la herramienta GIT en el sistema operativo y el *plug-in* GitHub Plugin en Jenkins. Esto permitirá que el proyecto sea tanto construido por Jenkins como analizado por SonarQube cada vez que se estime oportuno, dependiendo de cómo esté configurado el proyecto en Jenkins.
* **Apache Tomcat.** Por último, será necesario un servidor de aplicaciones Apache Tomcat en el que desplegar el fichero **.war** correspondiente a la herramienta Jenkins, el cual está disponible para su descarga gratuita en la página web oficial de Jenkins.

Una vez presentadas las herramientas a utilizar durante el proceso de gestión de la calidad, se muestran los pasos que se han seguido a lo largo del desarrollo del proyecto de la asignatura para poder realizar la instalación de la herramienta, llegando así a la integración y el análisis continuos del subsistema a través de Jenkins.

* 1. **Proceso de instalación de SonarQube y componentes asociados**

Antes de mostrar el proceso de instalación de todas las herramientas necesarias para realizar análisis de código estático, es necesario mencionar que esta serie de operaciones se harán bajo una máquina virtual cuyo sistema operativo está basado en Linux, concretamente la distribución Ubuntu de 64 bits. Por otra parte, el proyecto que se utilizará para llevar a cabo los análisis será el subsistema Frontend de Resultados. Es recomendable ejecutar el comando **sudo su** en una consola de comandos para ser usuario *root* (totalmente privilegiado) durante todo el proceso de instalación.

En primer lugar, será necesario instalar el servidor de bases de datos. Han de ser instalados tanto el servidor como el cliente de MySQL, utilizando las siguientes instrucciones en una consola de comandos.

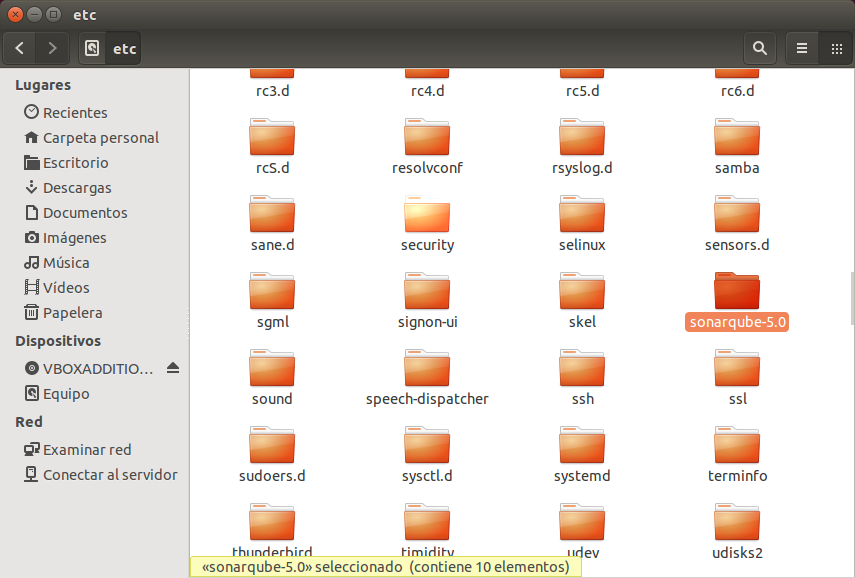
* **sudo apt-get install mysql-server**
* **sudo apt-get install mysql-client**

En el caso del servidor de MySQL, al introducir el comando la contraseña del usuario *root* de dicho servidor será solicitada. En este proceso se ha establecido dicha contraseña como **“practica”**.

Como ya se comentó, es necesario llevar a cabo la instalación de Java JDK 1.6 antes de instalar SonarQube, por lo que se debe ejecutar el siguiente comando:

* **sudo apt-get install openjdk-6-jre**

Una vez instalado el servidor de bases de datos y Java JDK 1.6, se descarga la última versión de la herramienta SonarQube desde [aquí](http://dist.sonar.codehaus.org/sonarqube-5.0.zip). Basta con descomprimir el fichero **.zip** descargado e insertar el directorio resultante en la ruta **/etc**.



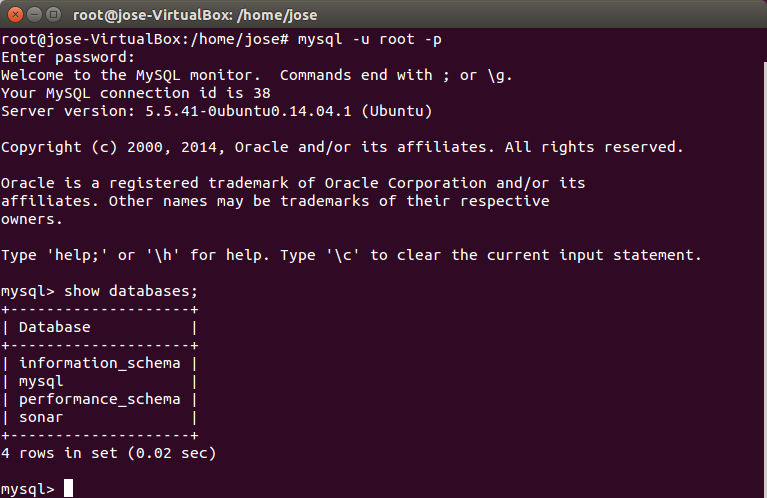
*Figura X: Ubicación de SonarQube en el sistema operativo*

A continuación, se creará una base de datos MySQL cuyo nombre será **sonar**. En esta base de datos serán almacenados los proyectos, sus configuraciones y los resultados de los análisis aplicados a cada uno de estos proyectos. Para crear la base de datos, se ejecutan los siguientes comandos:

* **CREATE DATABASE sonar CHARACTER SET utf8 COLLATE utf8\_general\_ci;**
* **grant all privileges on sonar.\* to ‘sonar’@‘localhost’ identified by ‘admin’;**
* **flush all privileges;**

La primera instrucción creará una base de datos cuyo set de caracteres sea el correspondiente a UTF-8, con el fin de evitar problemas con caracteres especiales. La segunda instrucción creará un usuario con todos los privilegios cuyo usuario y contraseña son **sonar:admin**. Por último, la tercera instrucción sirve para eliminar posibles usuarios privilegiados anteriores al que se acaba de crear o, al menos, retirar los privilegios de dichos usuarios.

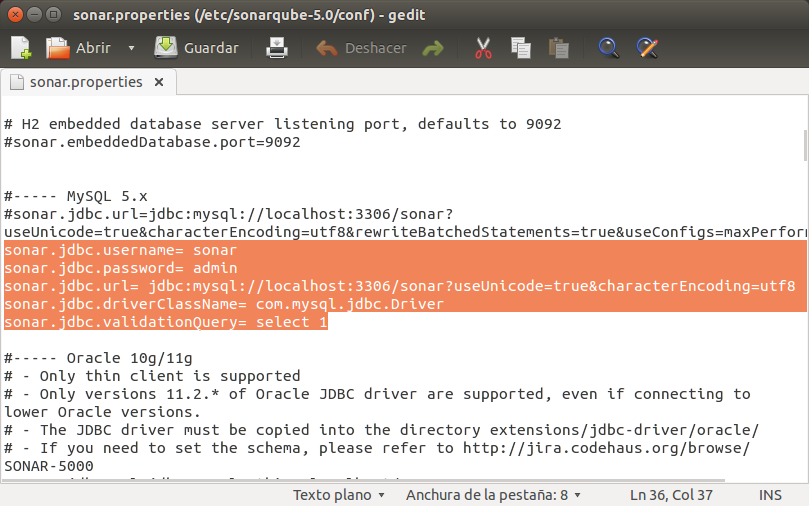
Para verificar que la base de datos se ha creado correctamente, podemos entrar en la consola de MySQL y utilizar el comando **show databases**:



*Figura X: Verificación de la creación de la base de datos de Sonar*

Confirmamos así que la base de datos **sonar** se creó correctamente.

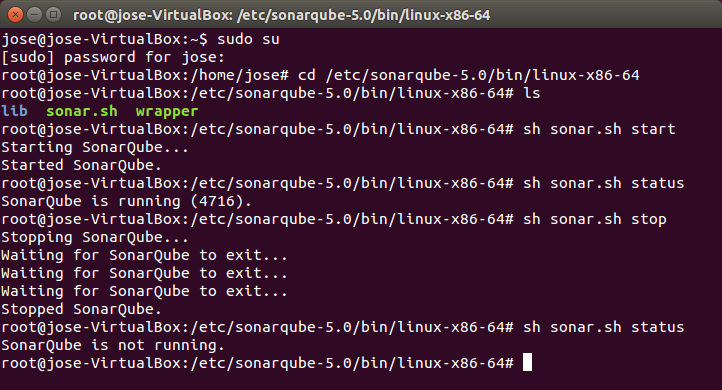
El siguiente paso lógico es llevar a cabo la configuración de Sonar para su correcto funcionamiento y comunicación con la base de datos creada. Para ello, nos situamos en el directorio **/etc/sonarqube-5.0/conf** y abrimos con permisos de edición el fichero **sonar.properties**. El fichero debe quedar tal que así:



*Figura X: Configuración de Sonar*

La primera y segunda línea indican el nombre de usuario y contraseña del usuario privilegiado de la base de datos creada en el paso anterior, respectivamente. La tercera línea representa la dirección que deberá utilizar Sonar para comunicarse con la base de datos **sonar**. La cuarta línea especifica el *driver* utilizado para facilitar las comunicaciones entre SonarQube y la base de datos. La quinta línea es una consulta simple de validación que comprueba que la base de datos está operativa.

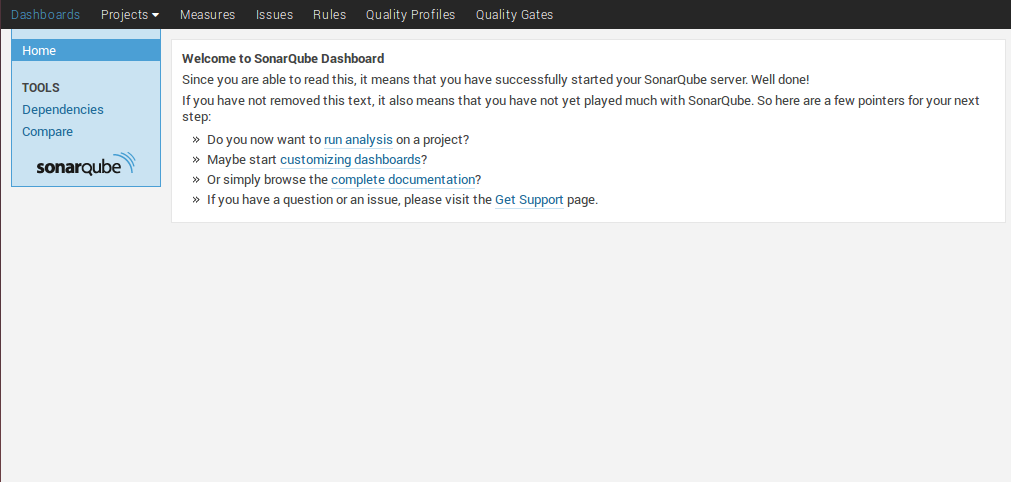
Para finalizar con la instalación de SonarQube, es conveniente realizar una prueba con el objeto de comprobar su correcto funcionamiento. Nos situamos en el directorio  **/etc/sonarqube-5.0/bin**. Dependiendo del sistema operativo utilizado, deberá accederse a un directorio u otro. En el caso de este proceso de instalación, se ha utilizado Ubuntu de 64 bits, por lo que se accede al directorio **/linux-x86-64**. Dentro de este directorio existe un fichero denominado **sonar.sh**. Desde la terminal, se introducen los siguientes comandos para realizar una prueba de ejecución de SonarQube.



*Figura X: Arranque de la aplicación Sonar*

El comando **sh sonar.sh start** arranca la aplicación SonarQube y establece las comunicaciones con la base de datos de manera automática. Para verificar que la aplicación está en funcionamiento, basta con utilizar el comando **sh sonar.sh status**, lo cual nos indicará, en caso de estar en funcionamiento, el PID (*Process ID*) de la aplicación. Por último, puede detenerse dicho proceso aplicando el comando **sh sonar.sh stop**. Se recomienda anteponer **sudo** a todos estos comandos, pues es posible que el comando no cause el efecto deseado sin permisos de usuario *root*.

Una vez arrancada la aplicación, podemos acceder a la misma situándonos en un navegador y visitando la dirección **http://localhost:9000/**. Se presupone que el puerto utilizado por SonarQube para escuchar peticiones es el configurado por defecto (puerto 9000). Debería mostrarse la siguiente interfaz de usuario:



*Figura X: Menú principal de Sonar*

Como es obvio, aún no existe ningún proyecto en la aplicación, por lo que es imposible realizar análisis de código estático.

Tras haber instalado SonarQube, debe instalarse el complemento **SonarQube Runner** para que la realización de análisis sea posible. La instalación consiste en descargar la última versión de SonarQube Runner de [aquí](http://repo1.maven.org/maven2/org/codehaus/sonar/runner/sonar-runner-dist/2.4/sonar-runner-dist-2.4.zip), descomprimir el fichero descargado y situar el directorio resultante dentro del directorio **/etc/sonarqube-5.0**.

Aunque es posible analizar un proyecto de manera manual, se decidió no optar por esta vía. En su lugar, el grupo de trabajo acordó utilizar Sonar de forma automatizada, es decir, a través de Jenkins. La idea es que, cada vez que se lleve a cabo la construcción del proyecto, Jenkins también lance SonarQube para realizar un análisis del código del subsistema Frontend de Resultados. De este modo, se pueden saber las deficiencias de calidad que presenta el subsistema y pueden ser subsanadas a la mayor brevedad posible. Este también es el principal motivo por el cual no se lleva a cabo una explicación en este documento de cómo analizar el código de un proyecto manualmente.

En el siguiente apartado se explicará de qué manera puede integrarse SonarQube con Jenkins.

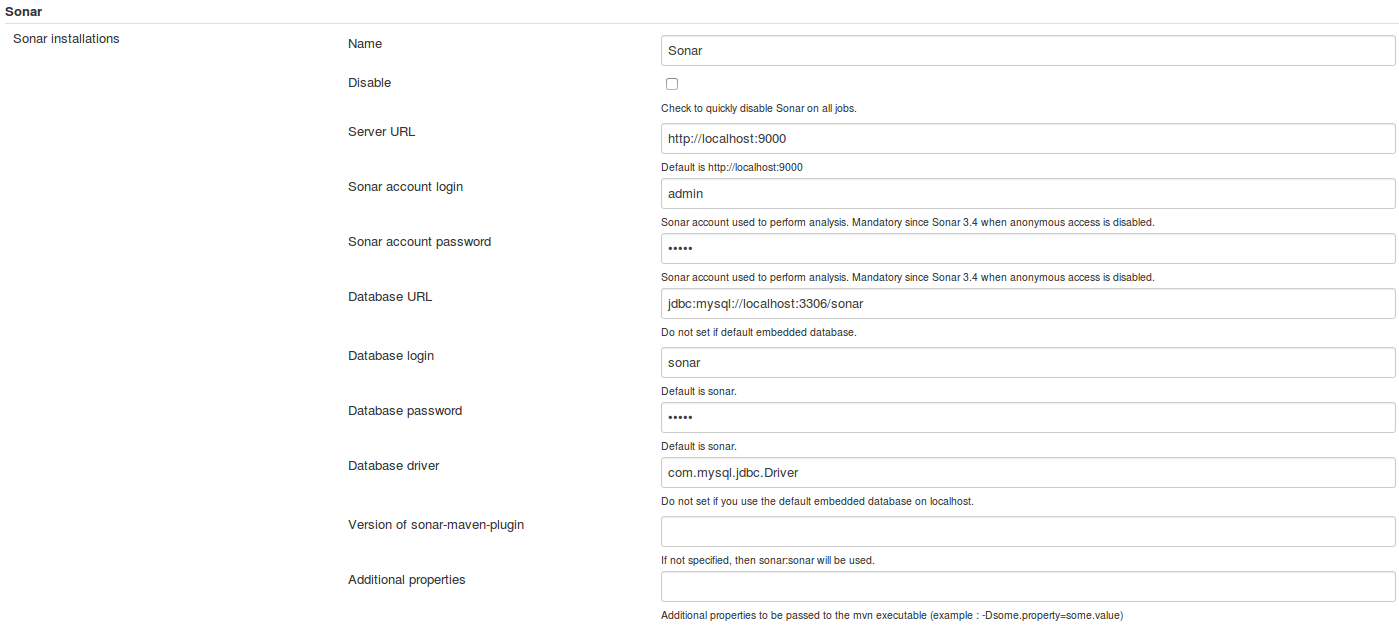
* 1. **Integración de SonarQube con Jenkins**

Previamente a dar paso a la integración de SonarQube con Jenkins, mencionar que no se explicará cómo instalar Jenkins y sus componentes asociados (Apache Tomcat y GIT). Esto se debe a que su instalación ya ha sido explicada en otra sección del documento y no es necesario volver a repetir todo el proceso.

Ya en el proceso de integración de ambas herramientas, en primer lugar es necesario situarse en la interfaz de usuario de Jenkins accediendo a la dirección **http://localhost:8080/**.

Una vez dentro, y desde el menú principal, se navega hasta **Manage Jenkins > Manage Plugins**. A continuación se selecciona la pestaña **Available** y en el campo de búsqueda insertamos **Sonar Plugin**. Basta con marcar la única opción existente y solicitar su instalación, además de exigir que Jenkins se reinicie tras la misma.

El siguiente paso es configurar tanto la aplicación SonarQube como su complemento SonarQube Runner. Para configurar SonarQube, se regresa al menú principal de Jenkins y se navega hasta **Manage Jenkins > Configure System**. Situándose en el apartado **Sonar**, establecer las siguientes configuraciones:

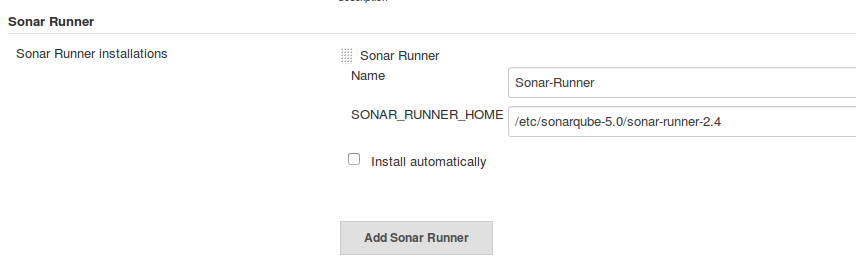


*Figura X: Configuración en Jenkins de Sonar Plugin*

Se procede a explicar el significado de cada uno de los campos:

* **Name**. Es el nombre de la instancia del servidor de análisis de código estático a crear.
* **Server URL**. Hace referencia a la dirección en la que SonarQube se encuentra ejecutándose. Nótese que el puerto es el que se establece por defecto al instalar SonarQube. Si es modificado, debe ser modificado aquí también.
* **Sonar account login**. Es el nombre de usuario del administrador de la aplicación SonarQube. Por defecto, el valor del campo es **admin**.
* **Sonar account password**. Contraseña asociada al administrador de SonarQube. Su valor también es **admin** por defecto.
* **Database URL**. Dirección a la que SonarQube accede para comunicarse con la base de datos en la que almacena los datos resultantes de los análisis. Aunque fue configurada durante la instalación de SonarQube, también debe especificarse en Jenkins.
* **Database login**. Nombre de usuario *root* establecido anteriormente para la base de datos **sonar**. En este caso, el usuario fue denominado **sonar**.
* **Database password**. Contraseña asociada al usuario *root* de la base de datos **sonar**. Su valor es **admin**.
* **Database driver**. *Driver* que facilitará la comunicación entre SonarQube y la base de datos, ya especificado en el fichero de configuración de SonarQube. Su valor es **com.mysql.jdbc.Driver**.

Una vez se ha configurado la instancia del servidor de análisis de código estático, se aplican los cambios y se guardan. A continuación, se accede al apartado **Sonar Runner** y se crea una nueva instancia. Debe quedar configurado tal que así:

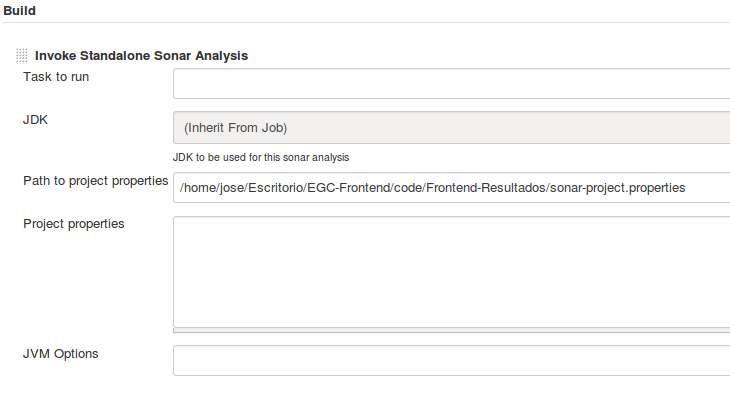


*Figura X: Configuración en Jenkins del analizador de Sonar*

El campo **Name** especifica el nombre de dicha instancia, y el campo **SONAR\_RUNNER\_HOME** indica la ruta del sistema operativo en la que se encuentran los ficheros binarios, configuraciones y librerías de Sonar Runner. Nuevamente, se aplican los cambios y se guardan.

El siguiente paso lógico es crear un nuevo proyecto en Jenkins a través de la opción **New Item**. Se especifica el tipo de proyecto tal que sea **Freestyle Project** y, una vez creado, se accede al mismo y se accede al apartado de configuración del proyecto. No se explicará la configuración de GIT en Jenkins, aunque sí es de suma importancia mencionar que la URI a utilizar para clonar el proyecto de GitHub es <https://github.com/EGC-FrontEnd-Resultados/code.git>. Dicho proyecto es el ya mencionado subsistema Frontend de Resultados, el cual se utilizará para ser analizado una vez que SonarQube esté totalmente configurado.

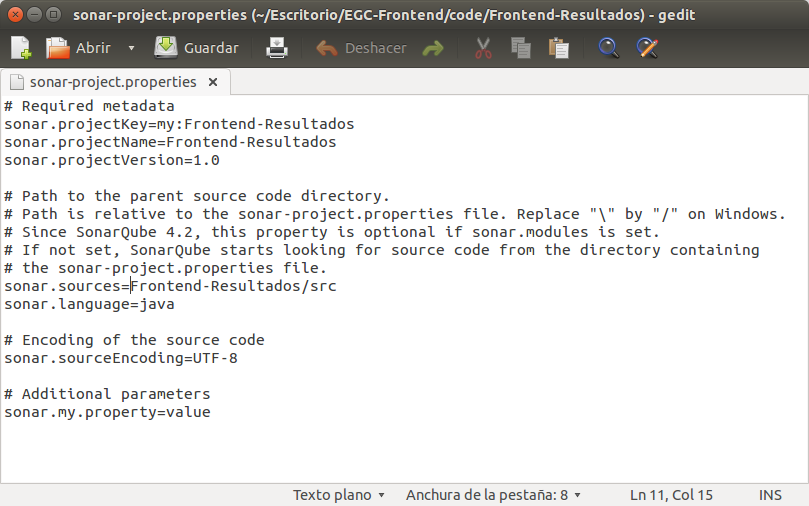
Tras esta pequeña aclaración, se accede al apartado **Build** y se añade una nueva opción de construcción denominada **Add Standalone Sonar Analysis.** Tras esto, surgirán una serie de campos a completar como se ilustra a continuación:



*Figura X: Configuración en Jenkins del proceso post-construcción de Sonar*

Sólo es necesario configurar el campo **Path to project properties**, en el cual se especifica la ruta de un fichero necesario para que Sonar pueda analizar el subsistema. Dicho fichero ha de llamarse obligatoriamente **sonar-project.properties**, y debe estar ubicado en la raíz del proyecto. Tras esto, se aplican y se guardan los cambios.

La creación del fichero **sonar-project.properties** es el último paso antes de poder lanzar el análisis automatizado del subsistema. Este fichero debe crearse en la raíz del proyecto **descargado desde GitHub** y debe ser configurado de la siguiente forma:

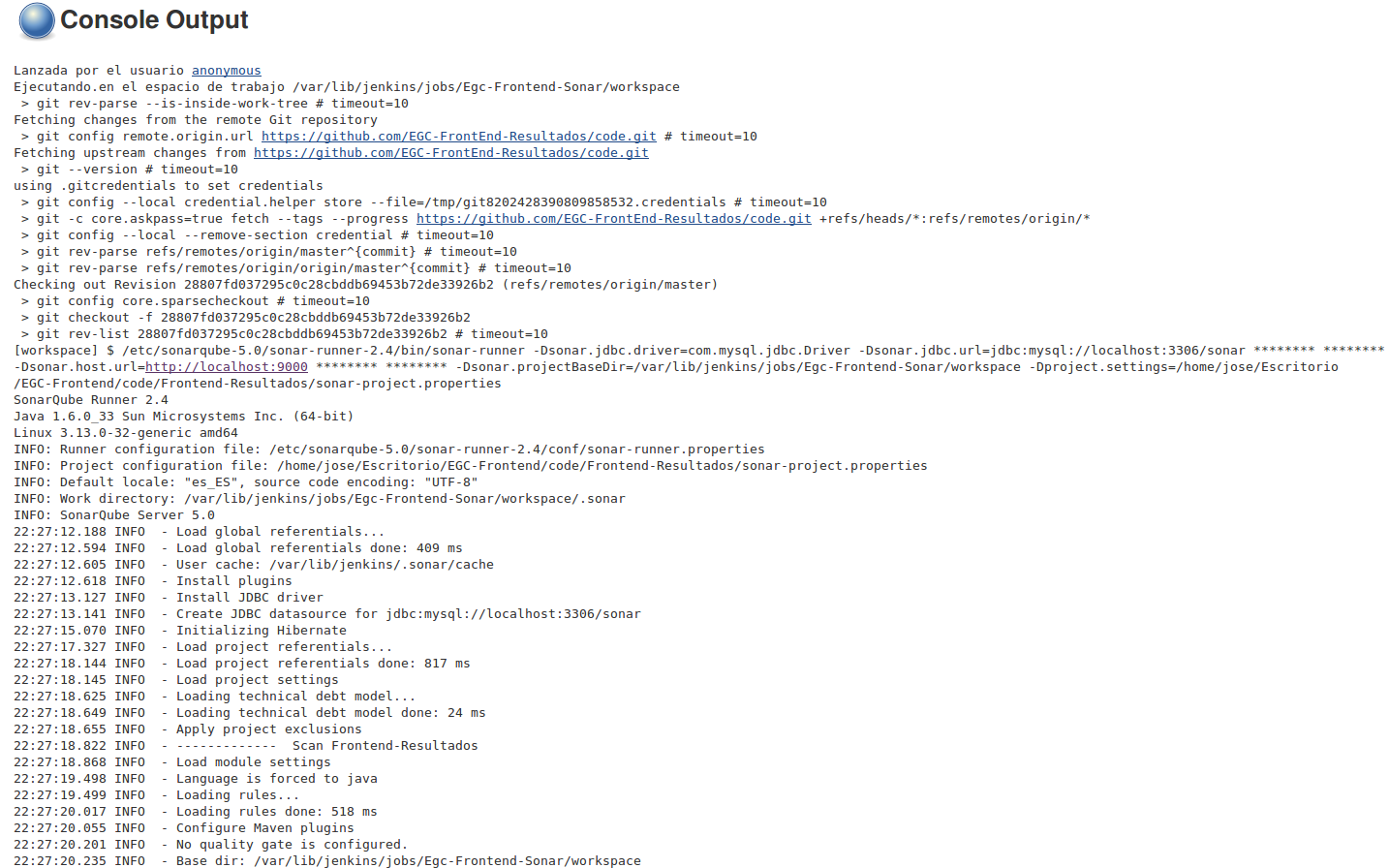


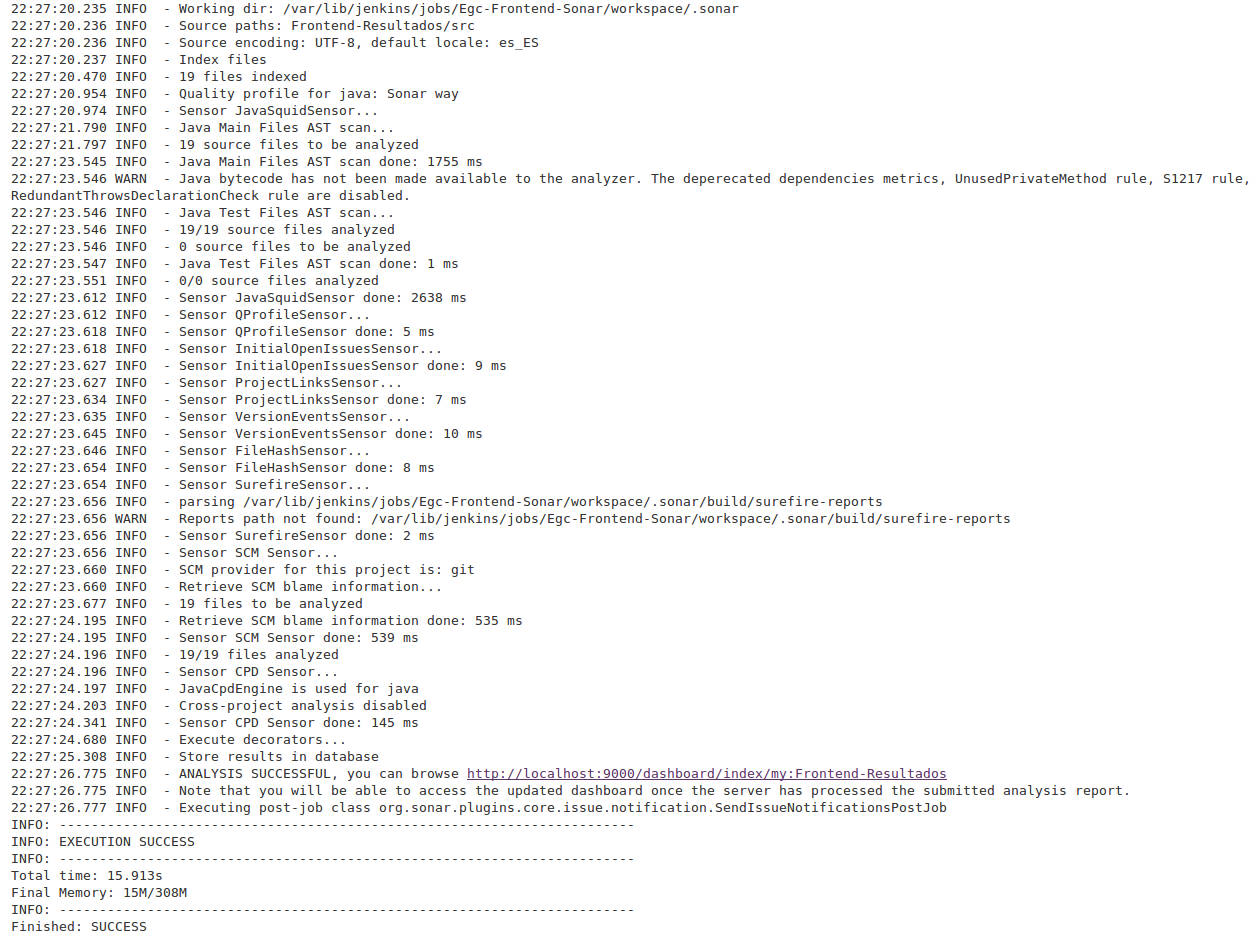
*Figura X: Configuración del fichero de Sonar en el proyecto*

Se detalla cada una de las líneas a continuación:

* **sonar.projectKey**: establece la ruta en la que se encuentra este fichero de configuración y le asocia un valor clave. En este caso, dicho valor es **Frontend-Resultados.**
* **sonar.projectName**: nombre representativo del proyecto. En este caso ha sido denominado **Frontend-Resultados**.
* **sonar.projectVersion**: versión del fichero de configuración. No es necesario alterar esta línea.
* **sonar**.**sources**: indica dónde se encuentran los ficheros de código que van a ser analizados por SonarQube Runner. Basta con definirlo como el valor clave antes especificado seguido del directorio **src** típico de la mayoría de proyectos (al menos, en proyectos Java).
* **sonar.language**: hace referencia al lenguaje de programación del código que se va a analizar. En el caso del subsistema Frontend de Resultados, es necesario definirlo como **java**. No obstante, pueden usarse otros como **php** o **py**, siempre y cuando se instalen los complementos necesarios desde la aplicación de SonarQube.
* **sonar.sourceEncoding**: codificación de los caracteres del código fuente.
* **sonar.my.property**: línea utilizada para añadir parámetros adicionales desde Jenkins.

Se guarda el estado del fichero de configuración y, ahora sí, se lanza en Jenkins la construcción del proyecto y, si todo ha ido bien, la salida por consola de esta construcción debería mostrar lo siguiente:





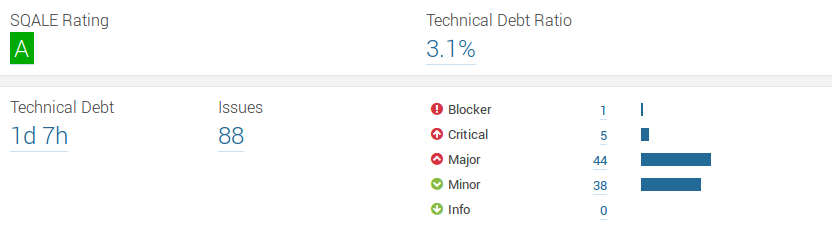
*Figura X: Salida de la construcción del proyecto en Jenkins*

Si justo al final de la salida se muestra que el análisis fue llevado a cabo con éxito (**SUCCESS**), puede accederse a la aplicación SonarQube para ver los resultados del análisis. Se procede a continuación a llevar a cabo dicha tarea y a interpretar las métricas.

* 1. **Interpretación de los resultados obtenidos**

Una vez dentro de la aplicación SonarQube, puede verse que ha aparecido el proyecto Frontend-Resultados como parte del conjunto de proyectos analizados por esta herramienta. Accediendo a dicho proyecto, se muestran los siguientes resultados del análisis realizado:





*Figura X: Métricas del proyecto tras el análisis del código*

SonarQube nos muestra algunos detalles del subsistema Frontend de Resultados. A la vista está, tal y como se ha visto a lo largo de la asignatura, que el proyecto no es demasiado grande, pues presenta tan sólo mil ocho líneas de código y unas sesenta y tres funciones, de las cuales la mayoría son funciones que ya existían en la plantilla base utilizada para desarrollar el subsistema.

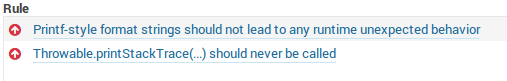
Si se observa la sección de complejidad, puede apreciarse que existen pocas funciones con una complejidad elevada. De hecho, a medida que se avanza en el gráfico de barras, el ratio de complejidad por número de funciones va decreciendo progresivamente hasta hacerse cero. Es algo lógico, teniendo en cuenta la baja complejidad de este subsistema frente a otros del sistema Agora@US.

Pero realmente, la parte más interesante es la que especifica la deuda técnica total existente. El concepto de deuda técnica expresa el coste total asociado a las tareas necesarias a aplicar para subsanar todas las incidencias reportadas durante el análisis realizado por SonarQube. Dicho coste es puramente temporal, aunque es posible añadir el coste en unidades monetarias por hora (um/h) que supondría subsanar estas incidencias. En el caso del subsistema Frontend de Resultados, el ratio de deuda técnica es de 0.031 (3.1%), lo cual indica que la calidad del proyecto es, en general, excelente. Este indicador de calidad viene dado por el valor del **SQALE Rating** (*Software Quality Assessment based on Lifecycle Expectations*, Valoración de la Calidad del Software basada en las Expectativas del Ciclo de Vida), el cual puede tomar valores desde la letra **A** (de excelente calidad) hasta la letra **E** (calidad pésima). Ya que el subsistema desarrollado por este equipo de trabajo posee la mayor calificación SQALE existente, podría decirse que cumple con los requisitos de calidad del código exigidos en cualquier entorno de trabajo.

A pesar de que el coste temporal de la deuda técnica puede parecer muy elevado, realmente no lo es. Teniendo en cuenta que la deuda técnica total es de unas treinta y una horas, es posible deducir cuánto tiempo debe emplear cada miembro del equipo para que todas las incidencias sean resueltas. Sabiendo que el grupo se compone de ocho miembros, el coste temporal por miembro sería de, aproximadamente, unas cuatro horas.

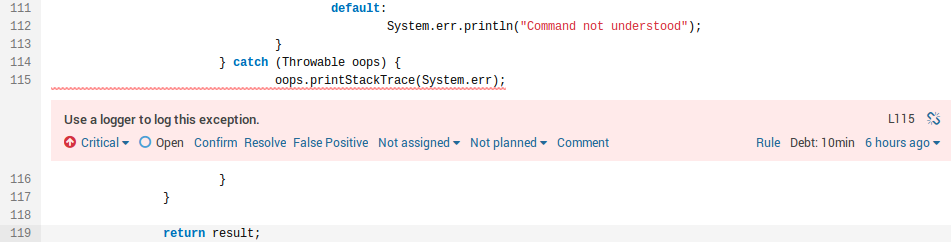
Si se echa un vistazo a las incidencias registradas por Sonar, se obtiene que la mayor parte de ellas son incidencias de mayor y menor grado. Existen muy pocas incidencias de carácter crítico y tan sólo una incidencia en el código que puede llevar a un bloqueo. La manera en que Sonar cataloga estas incidencias puede ser alterada modificando los umbrales de calidad para cada una de estas categorías. En el caso de este proyecto, se ha decidido no alterar dichos parámetros, pues se ha considerado que el modo en que SonarQube cataloga las incidencias es correcto.

Visitando las incidencias críticas, se muestra lo siguiente por pantalla:



*Figura X: Lista de incidencias críticas en el código del subsistema*

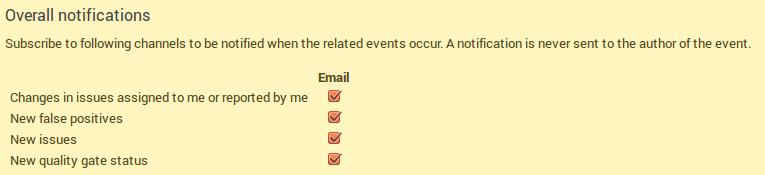
Aquí entran en juego lo que SonarQube denomina **reglas**. Una regla es un patrón que se espera que el código analizado siga. Si sigue el patrón, no se reporta el hecho de que presente una incidencia. En caso contrario, SonarQube reporta la incidencia y le asigna una categoría dependiendo de la gravedad de la misma. Por ejemplo, en el caso del subsistema Frontend de Resultados existe una incidencia crítica debido a que se está imprimiendo por pantalla la traza de una excepción, lo cual puede contener datos sensibles. Esto debe ser modificado, y SonarQube ofrece la opción de usar un registro o *log* en lugar de imprimir la traza:



*Figura X: Descripción de una incidencia en el código*

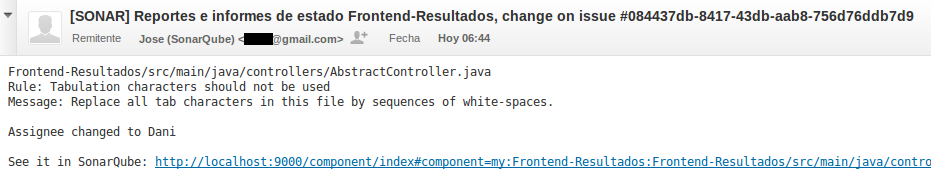
SonarQube ha señalado la línea en la cual se encuentra la incidencia y, además, ha especificado el tiempo necesario para subsanarla. En este caso, con tal sólo diez minutos es posible modificar el código de forma que la regla se cumpla para el fichero explorado (**QueryDatabase.java**). Adicionalmente, Sonar permite marcar la incidencia como resuelta, marcarla como falso positivo (es decir, que parece que no cumple una regla pero sí lo hace o que no supone una incidencia grave), asignarla a un miembro del proyecto para que sea resuelta por él y dejar comentarios, entre otras.

Sin duda, la opción más interesante es la de asignar la incidencia al miembro del equipo deseado para que la resuelva, lo cual ha sido adoptado en el propio equipo de desarrollo del subsistema. Para ello, cada uno de los miembros se registró en la aplicación instalada en uno de los ocho ordenadores disponibles. Posteriormente, cada miembro configuró su perfil de manera que pudiese recibir correos electrónicos ante la aparición de cambios en las incidencias en las que está involucrado, de nuevas incidencias, nuevos falsos positivos y actualizaciones de los umbrales de calidad.

****

*Figura X: Habilitación de envío de notificaciones*

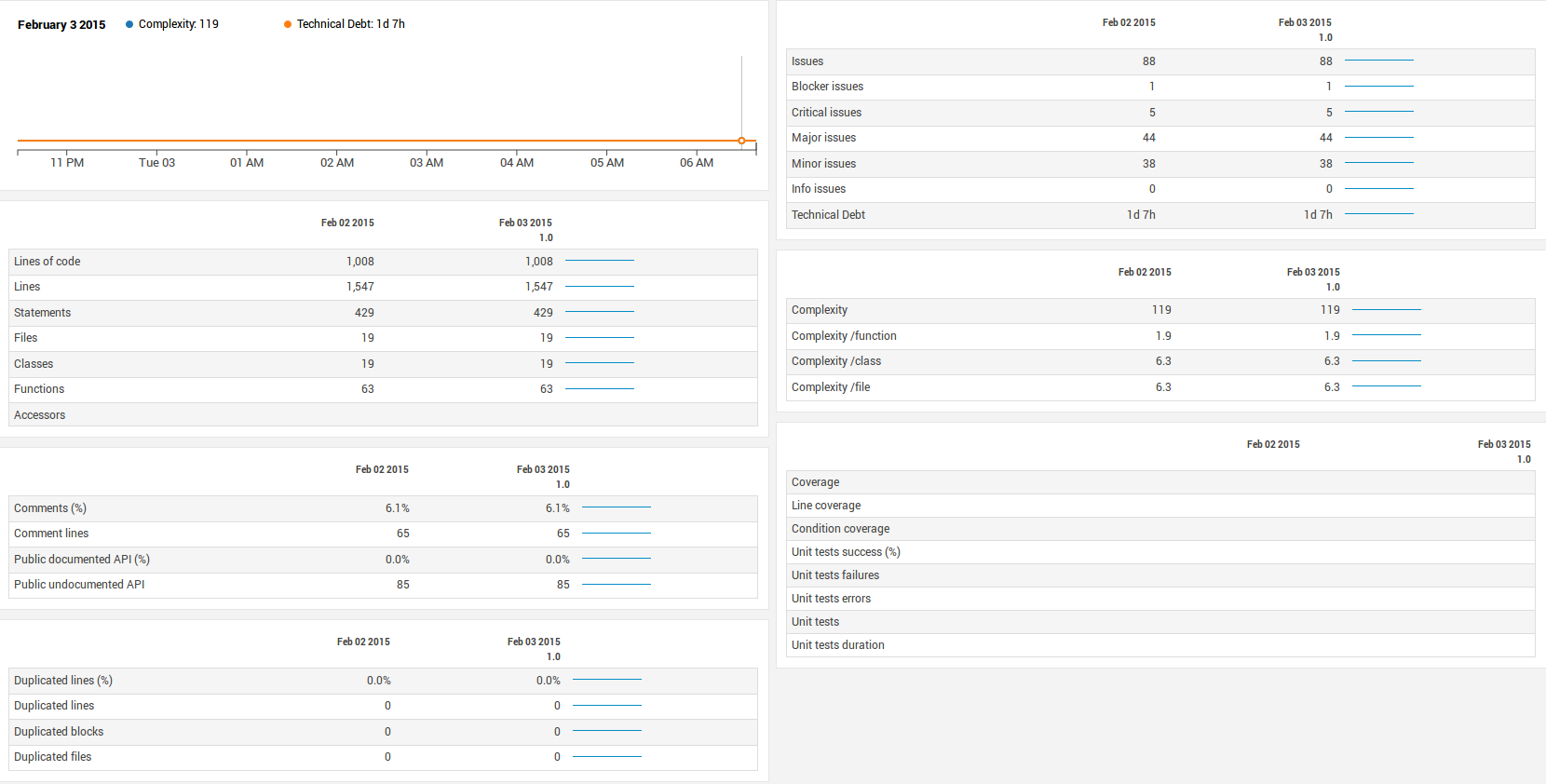
Cuando un miembro asigne a otro la resolución de una incidencia, este último será notificado por correo electrónico, indicándole el fichero en el que está presente la incidencia, la regla incumplida y un comentario sobre cómo subsanar la incidencia:

****

*Figura X: Notificación recibida para solucionar una incidencia*

El equipo consideró que era una manera muy eficiente de reportar las incidencias a resolver sin la necesidad de llevar a cabo el proceso manualmente. El envío de notificaciones podía automatizarse y era relativamente sencillo llevar a cabo la configuración para habilitar dichas notificaciones.

Para terminar con esta sección, es importante mencionar el apartado ***Time Machine*** de SonarQube. En cierto modo, es muy parecido al concepto de *Time Machine* estudiado en la asignatura y utilizado durante el desarrollo del subsistema. SonarQube permite ver un histórico del proyecto desde el punto de vista analítico, mostrando una comparativa entre las métricas de los análisis más recientes. Esto permite al equipo realizar un estudio de la calidad del código a nivel de métricas. Estas métricas pueden ser el porcentaje de líneas comentadas, la complejidad, las incidencias totales y clasificadas, porcentaje de cobertura de los tests unitarios sobre todo el subsistema, etc.



*Figura X: Resumen general de los análisis aplicados al código*

Sin duda alguna, SonarQube es una aplicación verdaderamente útil. Es cierto que no se ha podido explorar una gran parte de las funcionalidades que posee esta aplicación, pero sí se ha procurado utilizar las que se han considerado de mayor importancia. En el siguiente apartado se muestran algunas de las funcionalidades que hubiese sido interesante utilizar pero que fueron desestimadas por falta de tiempo u otros factores.

* 1. **Tareas abiertas**

Existen una serie de tareas que el equipo de trabajo no pudo llevar a cabo y que podrían haber sido muy beneficiosas para la gestión de las incidencias a nivel de grupo y a nivel global.

La primera de ellas es el hecho de no haber introducido todos los subsistemas para ser analizados por SonarQube. Esto habría sido una muy buena opción de cara a encontrar *bugs* y problemas menores. Desplegar el servidor de análisis de código estático en algún servicio de *hosting* gratuito en Internet para que todos los grupos tuviesen acceso al mismo es un uso mucho más eficiente que el que se le ha dado a la aplicación. Hubiese bastado con crear diversos grupos de trabajo, asignar a cada grupo su subsistema y que cada cual realizase los análisis y gestiones oportunas. No obstante, no es fácil desplegar este servidor sin unos costes de mantenimiento asociados.

La siguiente tarea sin realizar deriva de la anterior. Aunque no se registraron todos los grupos de trabajo existentes, podrían haberse registrado, al menos, los grupos que están en contacto directo con el subsistema Frontend de Resultados. Dichos grupos son los de Recuento, Visualización y Modificación de Resultados. Esto se explica porque en ocasiones se utilizan ficheros de subsistemas adyacentes al desarrollado por este grupo, y pueden encontrarse incidencias que no pueden ser reportadas al no estar dados de alta en la aplicación de Sonar. Aun así, sigue existiendo la problemática de mantener un servidor de análisis de código más allá de un entorno local, así que difícilmente puede ser llevada a cabo esta posibilidad.

Por último, existe un problema intrínseco al propio subsistema, más que a los grupos de trabajo. Dado el reducido tamaño y, por tanto, complejidad del subsistema Frontend de Resultados, es evidente que el desarrollo y aplicación de tests funcionales y de rendimiento no resulta demasiado productivo. Es por esto que no se ha podido explotar una de las características más importantes de Sonar: la cobertura de los tests unitarios sobre todo el subsistema. Con esta opción se puede determinar si los tests aplicados están cubriendo y contemplando todas las posibilidades en el código, o al menos gran parte de ellas. Quizá puedan desarrollarse algunos tests para probar esta funcionalidad, pero se ha estimado innecesario dada la naturaleza del subsistema.

* 1. **Conclusiones**

Sonar es, indudablemente, una herramienta muy potente que facilita la tarea de encontrar *bugs* y determinar si se están utilizando buenas prácticas a la hora de desarrollar el código.

No obstante, la aplicación presenta problemas de gestión de usuarios que pueden dificultar el reporte de incidencias. Desde la versión 3.0 de SonarQube, es imposible establecer el e-mail de cada usuario desde la propia aplicación. La única forma de modificar este parámetro es modificando la base de datos a mano e introduciendo para cada usuario la dirección de correo electrónico a la que enviar las notificaciones correspondientes a la asignación o identificación de incidencias.

Por otra parte, la configuración por parte del administrador del servicio de notificaciones también es una tarea tediosa. En el caso de este subsistema, se utilizó como dirección origen de estas notificaciones una cuenta de Gmail. Para otorgar permisos de envío de correos electrónicos a la aplicación, es necesario realizar una serie de ajustes también en la cuenta de correo electrónico utilizada. Por tanto, esta característica no sólo depende de la aplicación en sí, sino que precisa de la configuración de aplicaciones externas para su correcto funcionamiento.

Pero, a pesar de estos inconvenientes, utilizar la aplicación sigue siendo viable. El uso de la misma ha supuesto un gran avance en la gestión de la calidad del subsistema por parte de los miembros del equipo de trabajo. Abarca tantos aspectos que resulta difícil dejar escapar detalles que puedan empeorar la calidad del subsistema por lo que, de modo general, se considera que se ha dado el uso adecuado a la misma.

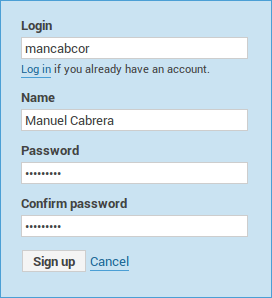
* 1. **Ejercicio**

Suponga que entra a formar parte del equipo de trabajo del subsistema Frontend de Resultados y que, acto seguido, recibe un correo electrónico informándole de que se le ha asignado una incidencia en el código descubierta por Sonar que deberá resolver a la mayor brevedad posible, con el fin de mejorar la calidad del código y de reducir la deuda técnica total del proyecto.

Resuelva la incidencia asignada y asegúrese de que la construcción del proyecto fuerce a Sonar a realizar un nuevo análisis que mejore los resultados de análisis anteriores. Comente los resultados obtenidos.

**Resolución**

En primer lugar, es necesario crear un nuevo usuario en la aplicación de SonarQube con el fin de que sea incorporado al equipo de trabajo. Para ello, se accede a la dirección **http://localhost:9000/users/new** y se rellenan los campos del formulario:

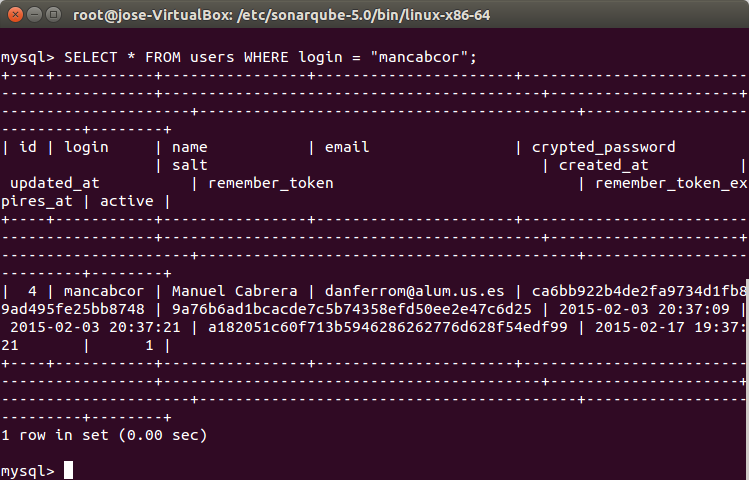


*Figura X: Registro de nuevo usuario en Sonar*

Una vez registrado, se habilita el envío de notificaciones por correo electrónico en el perfil del usuario. Tras esto, es necesario introducir manualmente el correo electrónico del usuario en la base de datos MySQL, debido a carencias en la gestión de usuarios por parte de SonarQube. Con el fin de lograr esto, se accede a una terminal y se aplican los siguientes comandos:

* **mysql –u root – p** [Insertar la contraseña solicitada: **practica**]
* **use sonar;**
* **UPDATE users SET email = “danferrom@alum.us.es” WHERE login = “mancabcor”;**
* **SELECT \* FROM users WHERE login = “mancabcor”;**

Esto modificará el campo del usuario correspondiente a la dirección de correo electrónico. La última sentencia sirve para comprobar que, efectivamente, el cambio se aplicó con éxito:



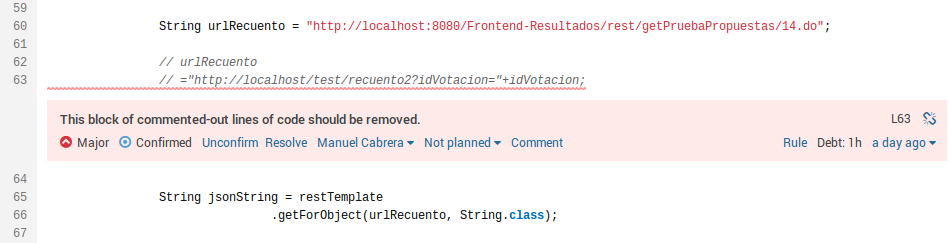
*Figura X: Modificación del e-mail de un nuevo usuario*

El siguiente paso lógico tras esta comprobación es revisar la bandeja de entrada del correo electrónico en busca de la notificación indicada.



*Figura X: Notificación de la incidencia asignada*

Accediendo al enlace proporcionado, se observa lo siguiente:

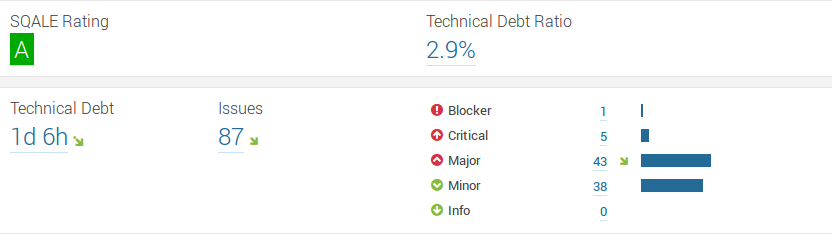


*Figura X: Incidencia de mayor grado a solucionar*

Utilizando el entorno de desarrollo deseado, se elimina el bloque de líneas comentadas para que la regla se cumpla y se realiza la subida correspondiente del proyecto al repositorio de GitHub.

Para finalizar, se marca la incidencia como resuelta y se inicia el proceso de construcción del proyecto desde Jenkins. Si todo ha ido correctamente, la construcción habrá sido un éxito y, por tanto, el análisis de SonarQube también.

Visitando la aplicación de SonarQube, aparece el siguiente cambio apreciable a simple vista:



*Figura X: Estado de la deuda técnica e incidencias tras resolver una de ellas*

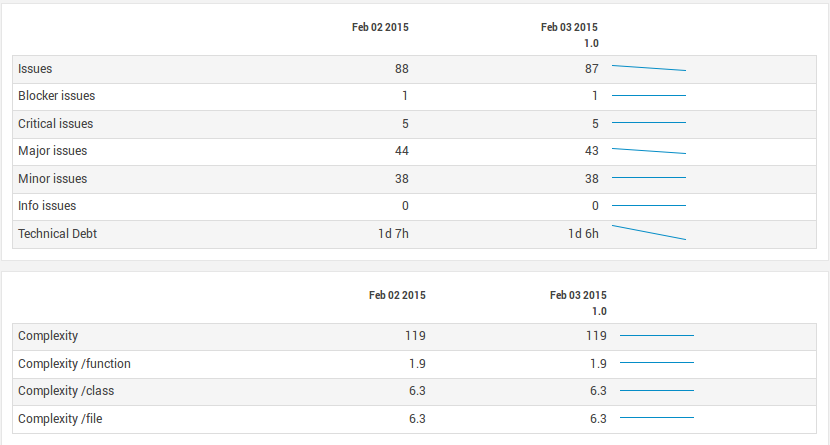
Es evidente que la deuda técnica del proyecto se ha reducido. Para ser exactos, la reducción de la misma ha sido de una hora, que es el tiempo estimado por SonarQube para subsanar la incidencia asignada. Como consecuencia, el ratio de deuda técnica también se ha visto reducido en un 0.2% y, evidentemente, el número de incidencias ha disminuido al haberse solucionado una de ellas.

Si se accede al apartado ***Time Machine***, se puede ver de manera cuantitativa cuánto ha mejorado la calidad del código:



*Figura X: Comparativa entre el análisis actual y anterior (I)*

Tal y como se observa en la gráfica, la complejidad del código se ha mantenido constante desde el anterior análisis hasta el actual. No obstante, la deuda técnica sí se ha visto reducida como consecuencia de la incidencia resuelta. Respecto al número de líneas de código, tan sólo ha disminuido en una línea, pues sólo se ha borrado un comentario. El porcentaje de comentarios ha disminuido un 0.2% y el número de líneas comentadas en tan sólo dos.



*Figura X: Comparativa entre el análisis actual y anterior (II)*

El número de incidencias ha disminuido en una unidad debido a la incidencia recientemente resuelta. Concretamente, se ha resuelto una incidencia de mayor grado.

A pesar de haberse resuelto una incidencia muy simple, sí es cierto que la deuda técnica del proyecto puede verse afectada de manera brusca. Esto se debe a las estimaciones hechas por Sonar en lo que a esfuerzo imputado en la resolución de las incidencias se refiere.

# Gestión del cambio, incidencias y depuración

Los cambios son inevitables cuando se construye software, éstos pueden darse en cualquier momento, por lo que debemos estar preparados para afrontarlos.

Nuestro proyecto tiene un alto riesgo de aparición de incidencias, por diversos motivos. Ejemplo de ello son la elevada división en subsistemas del proyecto o la falta de experiencia de los miembros del equipo, a la hora de llevar a cabo una integración de tantos subsistemas.

Para reducir en la mayor medida posible los impactos producidos, por los cambios e incidencias, es vital llevar una correcta gestión de éstos a lo largo de todo el proyecto, es decir, hay que prestar especial atención a la gestión de la configuración. Un sólo cambio en uno de los subsistemas, puede tener repercusiones en el resto. Los desarrolladores deben poder acceder de forma sencilla a un registro de incidencias y cambios, donde consultar un histórico de cambios, y tomar medidas al respecto si hay una nueva incidencia o un nuevo cambio.

La necesidad de gestionar la configuración surge del hecho de que el software evoluciona en el tiempo:

* Durante el desarrollo
  + El desarrollo del software siempre es progresivo, incluso en el ciclo de vida en cascada.
  + El desarrollo evolutivo consiste, precisamente, en una evolución controlada (ciclo de vida espiral, prototipos evolutivos).
* Durante la explotación
  + Durante la fase de mantenimiento se realizan modificaciones sucesivas del producto.
* En todos los casos
  + Suele ser necesario recuperar versiones antiguas, aunque sea sólo para consultar.
  + Para ello hace falta tener organizado el almacenamiento de versiones anteriores.

Establecemos una serie de pautas para homogeneizar los commits en GIT:

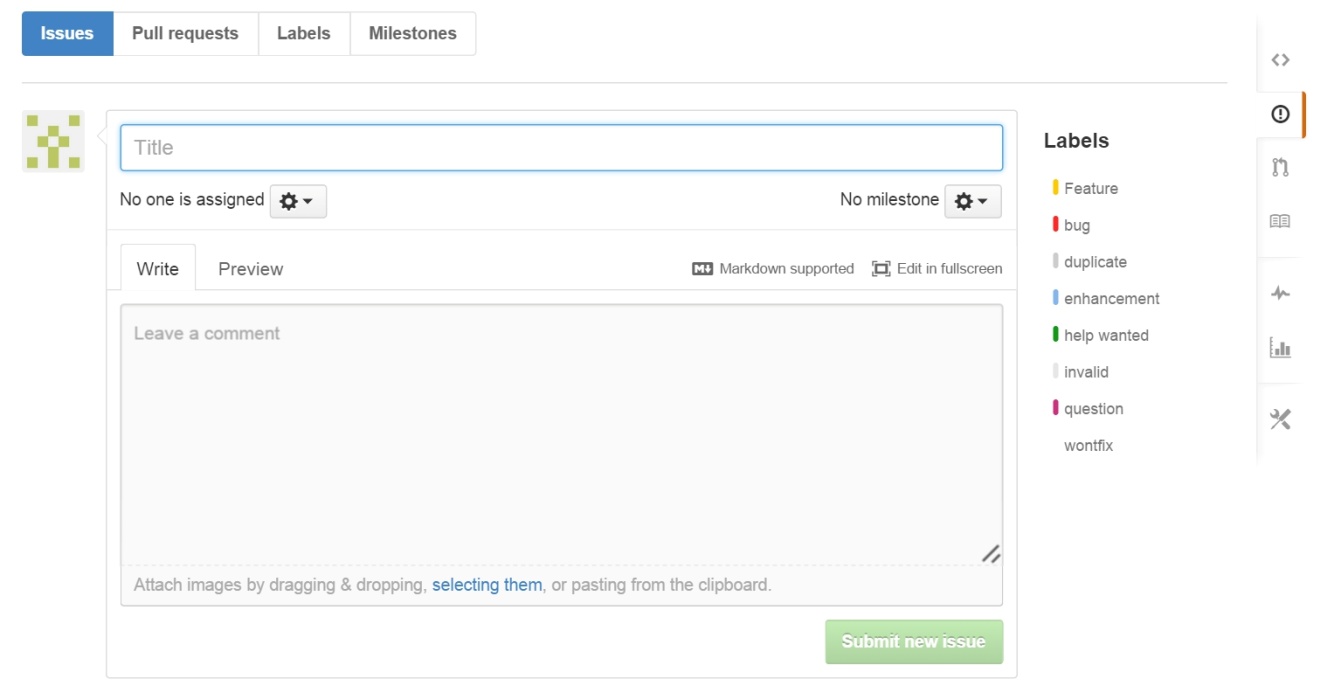
* En primer lugar, el título comenzará con una palabra clave en mayúsculas y en español. Esta palabra hará referencia al tipo de cambio del que se trata:
  + ADICIÓN: [título del commit]
  + CORRECCIÓN: [título del commit]
  + APIGET: [título del commit]
  + PERSISTENCIA: [título del commit]
  + CONFIGURACIÓN: [título del commit]
  + DESPLIEGUE: [título del commit]
* Una vez especificado el tipo de cambio, en la parte [título del commit] se expondrá más de detalle el cambio lógico que supone dicho commit, sin superar los 80 caracteres.
* Por último, se añadirá una descripción detallada del cambio realizado.
  1. **Gestión del cambio**

Los cambios provocan el aumento del grado de confusión entre los ingenieros de software que trabajan en un proyecto. La confusión surge cuando los cambios:

* No se analizan antes de realizarlos.
* No se registran antes de implementarlos
* No se reportan a quienes deben conocerlos.
* No se controlan de forma que mejore la calidad y reduzca el error.

Para evitar estas situaciones, elaboramos un plan para llevar acabo la gestión de los cambios:

Lo primero es elegir una manera con la que poder comunicar los cambios al resto del equipo y que queden grabados a modo de histórico, de tal forma que los integrantes del equipo puedan acceder a ellos siempre. Usaremos la funcionalidad de “Issues” que nos brinda la plataforma GitHub, la elegimos por dos motivos de peso: el primero es que cada subsistema tiene su propio repositorio creado en la plataforma y el segundo es que GitHub notifica por correo electrónico los issue, de modo que no hay que estar pendiente de mirar el listado de nuevos issues.



*Figura X: Esquema creación issue*

Los issues llevan un título **obligatorio**, luego opcionalmente podemos añadirle:

* Una **descripción**.
* Un **responsable:** Persona encargada de llevar el cambio, o el control sobre este.
* Un **hito**.
* **Etiquetas**, una etiqueta es una palabra clave que identifica el tipo de issue, también nos permiten realizar búsquedas filtradas en el listado de issues. Hay algunas predefinidas, aunque podemos crear las nuestras propias, de hecho nosotros hemos añadido la etiqueta “feature”, que será la elegida cuando queramos realizar un cambio destinado a añadir una nueva funcionalidad en el proyecto.

El procedimiento a seguir cuando haya que realizar un cambio en nuestro subsistema será el siguiente:

El miembro del equipo que propone un cambio, creará un nuevo issue, el miembro puede ser de nuestro propio subsistema o de otro. Las comunicaciones se realizaran mediante comentarios sobre el propio issue.

Una vez notificado el cambio, será analizado por nuestro equipo, de tal forma que podamos ver si dicho cambio afecta realmente nuestro subsistema. En caso negativo se comunicará que dicho cambio está fuera de nuestra competencia, y se procederá a cerrar el issue.

En caso afirmativo, lo primero será evaluarlo y hacer un balance del impacto negativo y positivo fruto de llevarlo a cabo. Si tras la evaluación se decide seguir adelante, se creará un nuevo comentario en el issue indicando que comenzamos a aplicar dicho cambio.

Acto seguido el equipo de desarrollo creará una nueva rama en el repositorio llamada “feature”, donde se realizarán las tareas necesarias para satisfacer el cambio.

Una vez finalizado correctamente, se volverá a unir a la rama master.

Por último, se notificará con un nuevo comentario que el cambio se realizó correctamente, y se procederá a cerrar dicho issue.

* 1. **Gestión de las incidencias**

Para la gestión de incidencias, también usaremos la funcionalidad “issues” de GitHub, puesto que cumple con todos los requisitos que vimos oportunos para llevar un registro de éstas. Así evitamos usar distintos métodos para la gestión de incidencias y cambios, que a menudo están relacionados. De modo que dotamos de mayor facilidad al equipo a la hora de gestionar la configuración del proyecto (gestión de cambios e incidencias).

El procedimiento inicial, será similar al de gestión de cambio, es decir, el miembro que quiera reportar una incidencia referente a nuestro subsistema, creará un issue donde se indique y describa la incidencia detectada.

Una vez sea la incidencia sea notificada al equipo se procederá a evaluar del siguiente modo:

Primero se analizará si dicha incidencia realmente compete a nuestro subsistema, en caso contrario se notificaría al autor del issue, mediante un comentario sobre este mismo, y acto seguido se procederá al cierre de dicho issue.

Si por el contrario realmente tiene que ver con nuestro subsistema, se procedería a resolver, previa notificación al autor del issue, donde se indicará que se ha comenzado a realizar las tareas pertinentes para resolver dicha incidencia.

Para llevar a cabo la resolución el equipo de desarrollo creará una nueva rama llamada “hotfix”, y seguirá las prácticas descritas en el siguiente apartado.

Una vez corregida la incidencia, se testeará de nuevo toda la aplicación, para poder observar que no hayan surgido nuevas incidencias a raíz de la reparación de ésta, y finalmente en caso de no surgir ningún problema, se unirá a la rama master, se comunicará la resolución y se cerrará el issue.

* 1. **Depuración**

La depuración la usaremos para poder hacer frente a las incidencias producidas en nuestro subsistema, mostramos a continuación una serie de buenas prácticas a seguir, a la hora de resolver una incidencia.

Llegados a este punto, por el diagnóstico de una incidencia en nuestro subsistema, lo primero que haremos como se indicaba en la gestión de incidencias, es crearnos una nueva rama del repositorio llamada “hotfix”.

Lo siguiente será reproducir el error descrito en la incidencia, así el equipo de desarrollo puede analizarla, si se tratase de un error de ejecución, generalmente se seguiría la traza de errores, mostradas por consola de nuestro entorno de desarrollo (Eclipse), siguiéndole la pista al error. Si no consiguiésemos concretarlo o se tratase de un error lógico, pasaríamos a usar el depurador de java, para ir reduciendo la localización de error mediante: puntos de ruptura, ejecutándolo paso a paso, observando el valor que toman las variables...

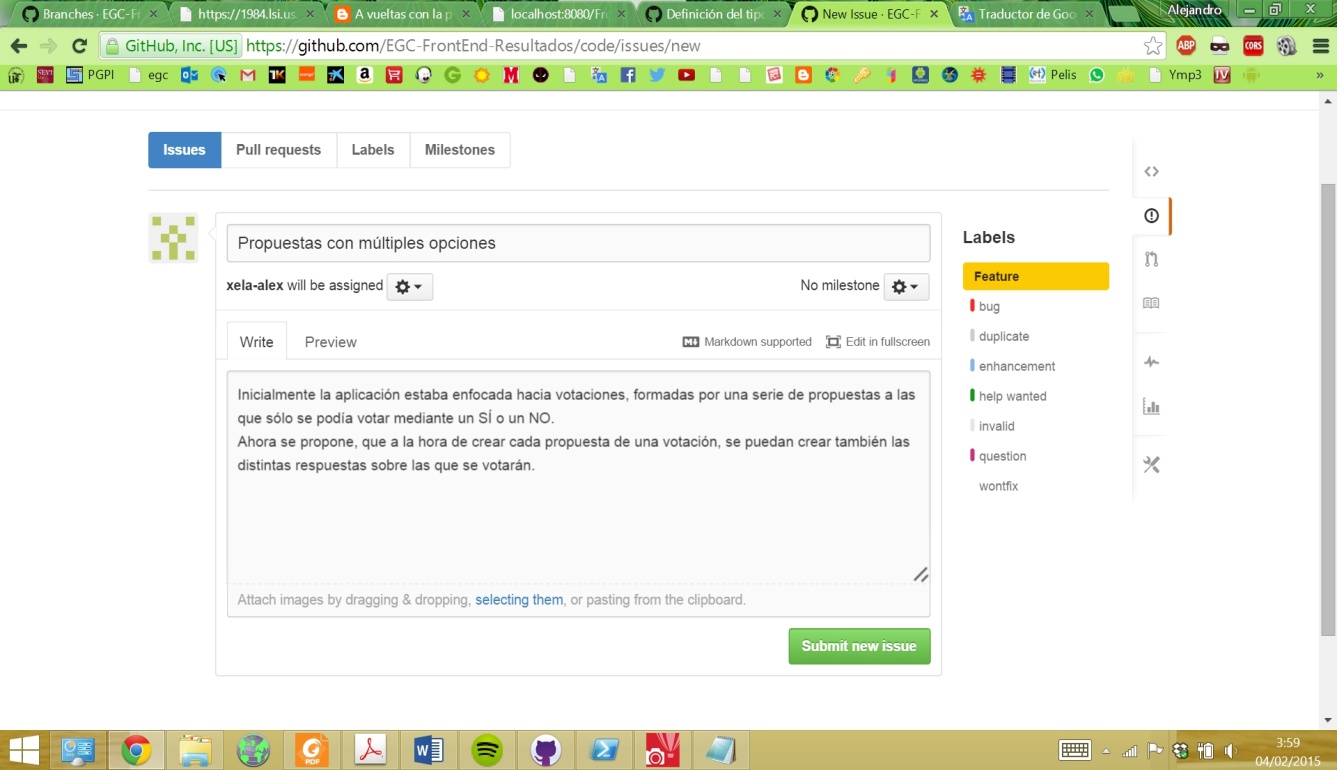
Una vez localizado el error y solucionado, se prueba nuevamente, para poder comprobar que se ha solucionado. Por último como buena práctica se deben volver a ejecutar todos los test del proyecto global, con el fin de detectar nuevos errores creados al solucionar la incidencia.

* 1. **Ejercicios**

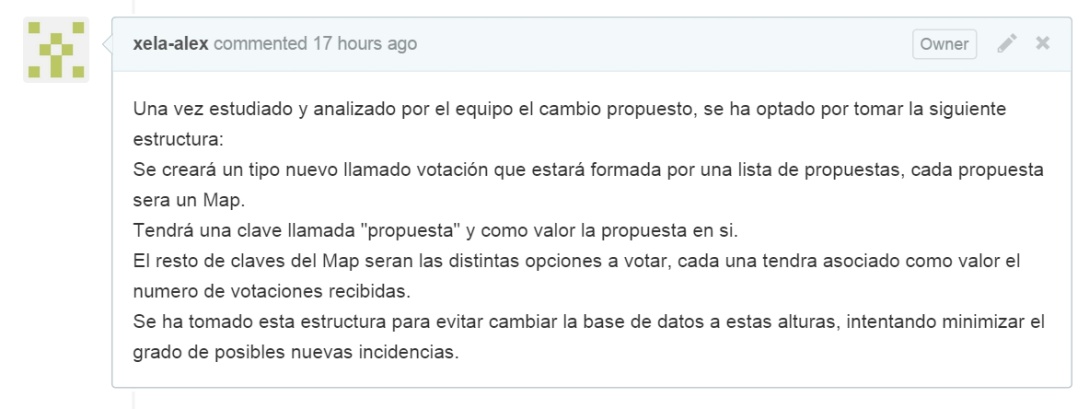
**Ejercicio 1: Ampliación de la funcionalidad**

Se ha decidido añadir una nueva funcionalidad al sistema, un usuario que cree nuevas propuestas en una votación, debe poder crear las distintas opciones para votar sobre esas propuestas (anteriormente sólo se tenían votos a favor y en contra, es decir se trataba referéndums).

**Resolución**

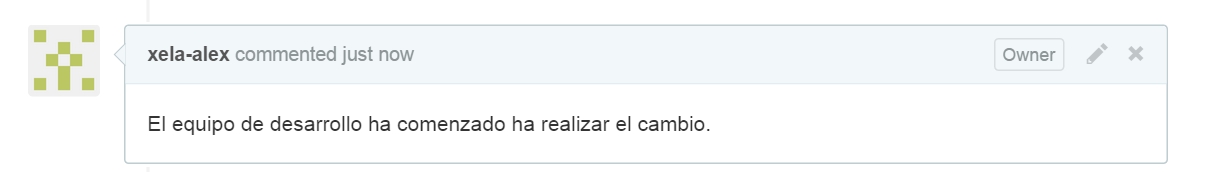
El primer paso es notificar este cambio, como indicamos en el apartado de la gestión de cambios, para ello creamos un nuevo issue que lo describa, le asignamos la etiqueta “Feature” y un responsable dentro d los miembros del equipo.

*Figura X: Creación issue ejercicio X*

Una vez analizado el cambio y su impacto, se marca el camino que debe seguir el equipo de desarrollo, en este caso hemos optado por no modificar la estructura anterior de la aplicación, es decir lo que estaba funcionado no se tocará, para evitar volver a crear incidencias resueltas a la hora de integrar los subsistemas. Esto se registra mediante un nuevo comentario.

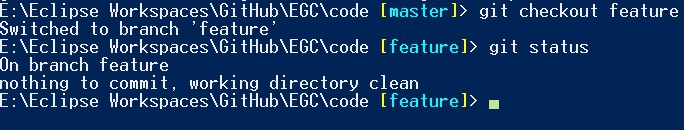
*Figura X: Comentario 1 issue ejercicio X*

Acto seguido notificamos que comenzamos a realizar el cambio:



*Figura X: Comentario 2 issue ejercicio X*

Lo siguiente es crear una nueva rama “feature”, sobre la trabajará el equipo de desarrollo:



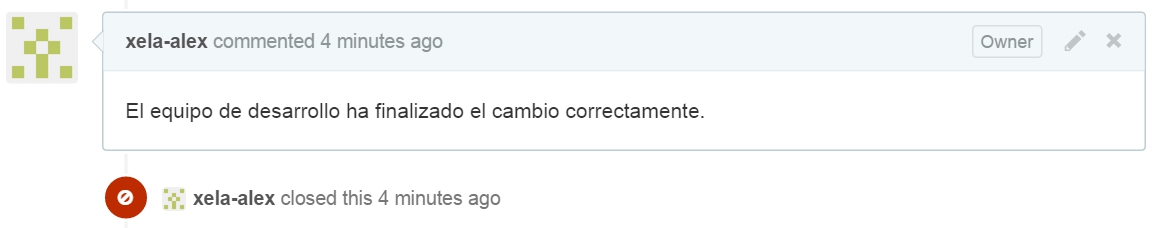
*Figura X: Creación rama feature ejercicio X*

Una vez finalizado el cambio y testeado, volveremos a ejecutar todos los test anteriores, de modo que comprobemos que no se han producido nuevas incidencias con este cambio.

Si no hay ninguna incidencia, y todo está correcto, la rama “feature” se unirá con la rama master:



*Figura X: Merge rama feature ejercicio X*

Finalmente, se notificará que el cambio ha finalizado y se procederá a cerrar el issue:

*Figura X: Comentario 3 issue ejercicio X*

**Ejercicio 2: Incidencia y depuración**

El equipo detecta una incidencia. Si las votaciones que ya han sido mostradas se actualizan, al volver a consultarse, muestran los datos de la primera consulta no los actualizados.

**Resolución**

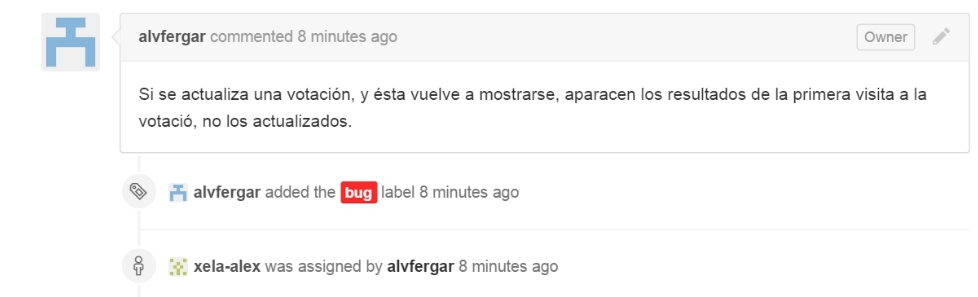
Al ver que es un problema al obtener los datos, y sabiendo que la función de nuestro subsistema radica en obtener datos del subsistema de Modificación para pasárselos al de Visualización, pensamos que es una incidencia de Modificación.

Nos dirigimos a crear un issue que haga referencia al subsistema de modificación, indicándole que realizando pruebas en la aplicación, hemos podido observar el error citado en el enunciado.

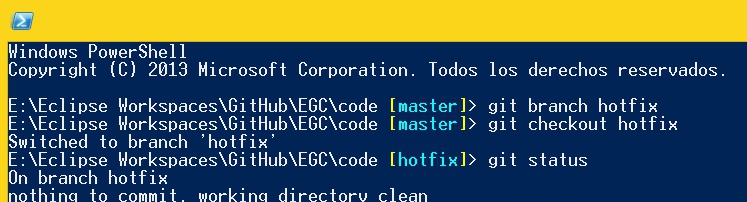
Una vez notificado, el equipo de modificación previo análisis, nos responde, que tras haber realizado una depuración concisa pueden afirmar, que no se trata de un error suyo, verifican que envían los resultados actualizados correctamente. De tal forma que, proceden a cerrar dicho issue.

Una vez obtenida respuesta de Modificación, podemos confirmar que el error viene después de su subsistema, los siguientes subsistemas en la comunicación son Frontend-Resultado (nosotros) o visualización. Con casi total probabilidad de ser el nuestro el culpable de la incidencia, puesto que visualización sólo muestra datos que le enviamos.

Como se trata de una incidencia, observada por nuestro propio equipo, no nos llega una notificación a GitHub con la incidencia. En este caso lo que debemos hacer es crear es crear nosotros mismo el issue en GitHub a modo de registro con la etiqueta “bug” que lo identifique como una incidencia, y le asignamos un responsable.

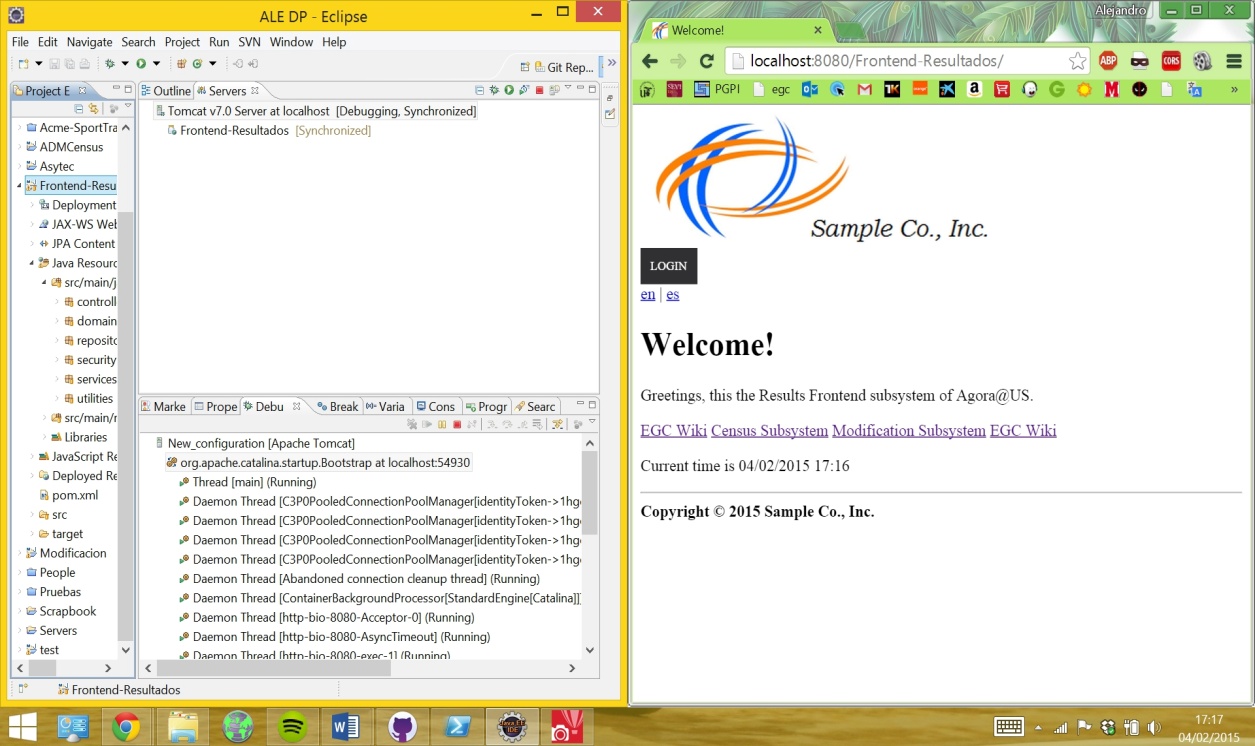


*Figura X: issue incidencia ejercicio X*

Una vez creado el registro, se crea una rama “hotFix”, sobre la que trabajara el equipo de desarrollo:

*Figura X: Creación rama hotfix ejercicio X*

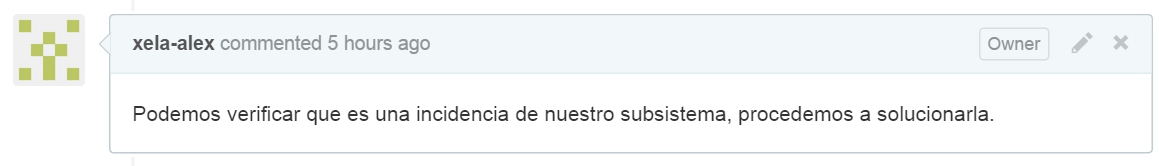
No es un error de sintaxis, ni de ejecución por lo tanto no nos aparece ninguna traza de error en la consola acerca de él. Se trata de un error lógico, es el tipo de error más difícil de detectar y arreglar. Lo primero que haríamos será proceder a la depuración de la aplicación, para ello iniciamos la aplicación en modo debug desde eclipse.



*Figura X: captura depuración ejercicio X*

Como no sabemos de dónde puede venir el error, colocamos un punto de ruptura en el método del controlador que se encarga de la petición que nos realiza visualización. A partir del punto de ruptura iremos siguiendo la ejecución del programa paso a paso, observando el valor que van tomando las variables.

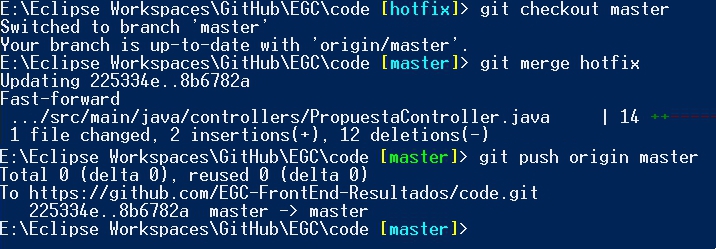
Tras un minucioso estudio de la traza que sigue el programa, podemos detectar el error, resulta que nosotros para evitar sobrecarga de peticiones a los sistemas de modificación y recuento, habíamos añadido una base de datos a modo de “memoria cache” que almacena datos de las votaciones ya consultadas, para que cada vez que las solicitase nuevamente visualización no hubiese que volver a comunicarse con los dos subsistemas citados. De esta forma las obtendríamos directamente desde nuestra base de datos, pero claro si éstas se actualizan, nosotros volvemos a mandar la que teníamos en base de datos fruto de la primera visita.



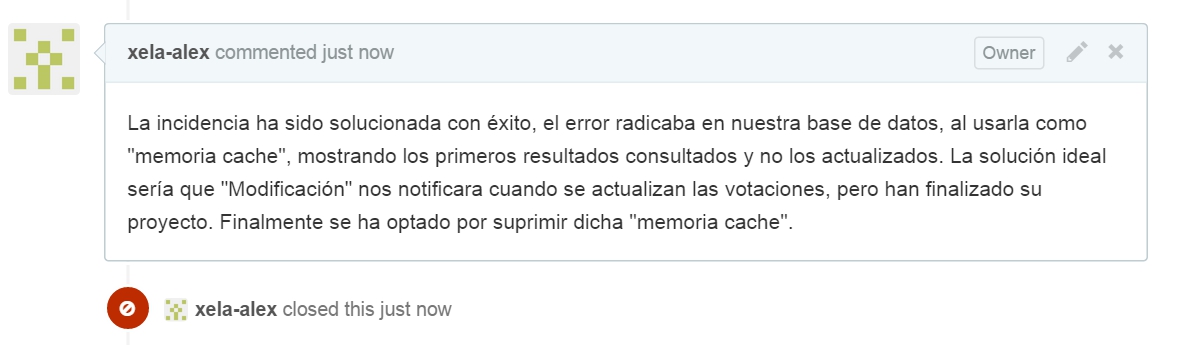
*Figura X: Comentario 1 issue ejercicio X*

La solución ideal sería comunicarle al subsistema de Modificación que nos indicase de algún modo cuando alguna votación de las ya consultadas es actualizada para proceder en ese caso a actualizar nuestra base de datos, pero el subsistema de Modificación ya finalizó su proyecto. Por lo que optamos por otra opción, suprimir nuestra “memoria cache” de forma que toda petición por parte de visualización se comunicará a Modificación, aunque esto implique eliminar nuestra “memoria cache” y realizar más peticiones al subsistema de Modificación.

Una vez solucionada la incidencia, y comprobar de nuevo que no surgen nuevas a raíz de corregir esta, unimos la rama “hotfix” a la master:



*Figura X: merge rama hotfix ejercicio X*

Por último comunicamos que se ha corregido la incidencia y cerramos el issue de GitHub:

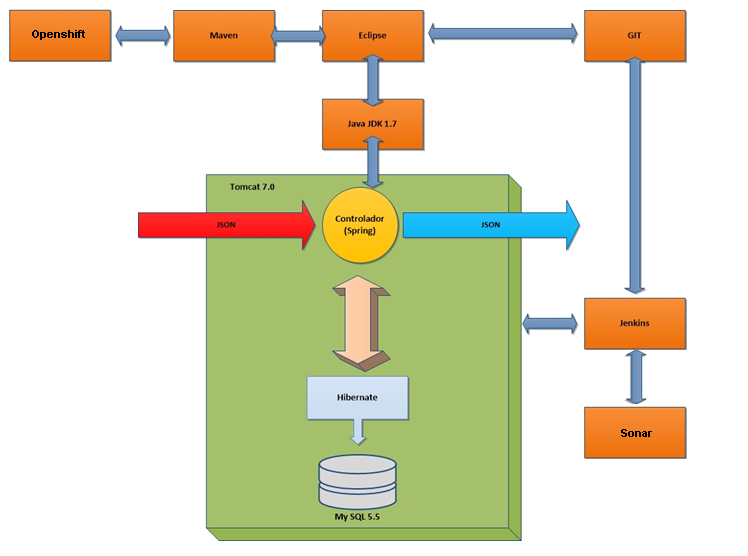
*Figura X: Comentario 2 issue ejercicio X*

# Mapa de herramientas

* 1. **Lista de herramientas**

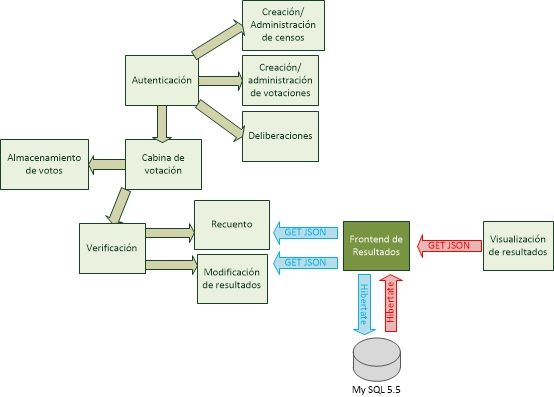
|  |  |
| --- | --- |
| * **Java**   Será el lenguaje de programación que se utiliza para implementar el subsistema, ya que está bastante extendido y se tiene bastante experiencia con él.  Es un lenguaje de programación de propósito general orientado a objetos, derivado en gran medida de C y C++. | **C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\java_tech.jpg** |
| * **JDK 1.7**   Java Development Kit es el conjunto de herramientas que proporciona todo lo necesario para crear, compilar e interpretar el código programado en Java. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\jdk-java.png |
| * **Eclipse**   Será el IDE o marco de trabajo en el cual se desarrolla el subsistema. Entre sus características destaca que es de código abierto, y dispone de multitud de *plug-ins* y permite la integración con Hibernate. | **C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Eclipse-logo.png** |
| * **Maven**   Maven es una herramienta de software para la gestión y construcción, ya que está diseñada para Java. Ofrece un modelo de construcción bastante simple basado en XML. | **C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\maven.png** |
| Maven utiliza un Project Object Model (POM) para describir el proyecto, y está listo para usarse en red, y descargarse las librerías de un repositorio de forma automática. | |
| * **Spring**   Este framework de código abierto está bastante extendido en la plataforma Java.  Sus características principales son la inyección de dependencias y la programación orientada a aspectos. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Spring.png |
| * **Hibernate**   Es una herramienta de software libre que permite el mapeo objeto-relación. Para su configuración se usan archivos XML y anotaciones en los objetos que permiten establecer relaciones. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\hibernate.png |
| * **GIT**   Diseñado por Linus Torvard, es una herramienta de control de versiones.  Entre sus características, cabe destacar la gestión distribuida, fuente de apoyo al desarrollo no lineal y gestión eficiente de grandes proyectos. | **C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\git.jpg** |
| * **Jenkins**   Software escrito en Java. De código abierto, gestor de integración continua y permite la integración con GIT. Esto facilita el uso de baterías de pruebas para cada *push* realizado sobre el repositorio en GitHub. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\jenkins.png |
| * **JSON**   Es un formato ligero para el intercambio de datos. Usaremos una librería Java para codificar nuestros datos e intercambiarlos con los subsistemas correspondientes. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\json_logo-555px.png |
| * **MySQL**   Como sistema gestor de base de datos usaremos MySQL para almacenar los datos de nuestro subsistema. Usaremos la versión GNU GPL. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\mysql_logo.png |
| * **Apache Tomcat**   Para la ejecución de nuestro subsistema necesitaremos un servidor Tomcat, puesto que ofrece soporte como contenedor web, de servlets y JSP. En definitiva, es un servidor web adaptado. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tomcat.jpg |
| * **Python**   Otros subsistemas usaran el lenguaje de programación Python para el desarrollo de su subsistema. | C:\Users\Manuel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\python-logo-master-v3-TM.PNG |
| * **SonarQube**   Se trata de una plataforma *open source* para la comprobación continua de la calidad del código fuente. | http://javierac.biz/wp-content/uploads/2014/05/SonarQube.png |
| * **OpenShift**   Para conseguir realizar el despliegue del subsistema en la nube se utiliza este producto, que ofrece un servicio de *hosting* gratuito, junto a otros, como la integración con Jenkins. | http://upload.wikimedia.org/wikipedia/en/thumb/3/3a/OpenShift-LogoType.svg/1024px-OpenShift-LogoType.svg.png |

* 1. **Visión interna del mapa de herramientas**

****

*Figura X: Visión interna del mapa de herramientas*

* 1. **Visión global del mapa de herramientas**



*Figura X: Visión global del mapa de herramientas*

# Integración de todos los subsistemas

[Intro: Xory y JP]

* 1. **Gestión del código**

**[Xory y JP]**

* 1. **Estrategia de integración**

**[Xory y JP]**

* 1. **Herramientas**

**[Xory y JP]**

* 1. **Automatización de la construcción**

**[Xory y JP]**

# Conclusiones

Durante el desarrollo de este proyecto, se han llegado a una serie de conclusiones con respecto a aspectos como respecto a la forma de trabajar, al aprendizaje y a la comunicación a nivel de grupo y a nivel global.

Es evidente que la forma de trabajar se ha visto enormemente mejorada. Se han conocido y aprendido varias herramientas y conceptos que acercan aún más a un entorno profesional de desarrollo.

El concepto de integración continua ha sido adoptado como algo fundamental en cualquier proyecto de desarrollo software, pues asegura la automatización de muchas tareas que se repiten asiduamente durante todo el ciclo de vida del proyecto y también que la aplicación sea susceptible a ser construida en todo momento. En el caso de este subsistema, se ha utilizado la herramienta Jenkins como servidor de integración continua para asegurar la misma tanto a nivel local (Frontend de Resultados) como a nivel global (todos los subsistemas de Agora@US).

La gestión del código fuente ya era algo conocido por todos los miembros del equipo desde mucho antes, pero no ha sido hasta el desarrollo de este subsistema cuando se ha conocido otro modelo de control de versiones que difiere bastante del utilizado habitualmente, que no es otro que Subversion. En su lugar, GIT ofrecía la posibilidad de realizar cambios en el subsistema y de subirlos a un repositorio local hasta que se considerase oportuno realizar la subida al repositorio global. En este subsistema se comenzó utilizando Subversion, pues era el sistema de control de versiones común y que siempre se había utilizado. No obstante, a medida que el subsistema tomaba forma y se avanzaba en los contenidos de la asignatura, se consideró muy beneficioso el hecho de migrar todo lo desarrollado a GIT y establecerlo como el SCV a utilizar.

Existen otros aspectos que también fueron mejorados gracias a herramientas como Redmine, para la gestión de proyectos, o Sonar, para el análisis de código estático con el fin de mejorar la calidad del mismo.

En lo que respecta a la comunicación a nivel de grupo, no se destacan demasiadas novedades con respecto a otros proyectos. La comunicación siempre se ha mantenido estable y con un tono cordial, sin trabas que imposibilitasen el desarrollo del subsistema. Pero sí es cierto que la comunicación a nivel global ha resultado bastante compleja por diversos motivos.

El primero de estos motivos es el mutuo acuerdo entre los grupos de trabajo con respecto a sus subsistemas. En ocasiones se producían conflictos con respecto al formato de los datos intercambiados entre subsistemas adyacentes o a la estructura general de Agora@US. Resultaba difícil que uno de los grupos cediese y aceptase la solución de otro grupo, o una solución intermedia a la propuesta por ambos. Afortunadamente, las tensiones se aliviaron a medida que el proyecto avanzaba y esto dio lugar a que se pudiese llevar a cabo la tan ansiada integración de todos los subsistemas, que era el verdadero objetivo de la comunicación entre todos los grupos. Esta comunicación seguía siendo complicada incluso con los subsistemas de Visualización, Recuento y Modificación, los cuales son adyacentes a este subsistema.

Pero, en definitiva, desarrollar este subsistema ha supuesto una grata experiencia. Se ha experimentado lo que ocurre en entornos de desarrollo reales, con lo cual todo el equipo ha cambiado un poco la mentalidad a la hora de entrar parte en un proyecto y a tener en cuenta que en un proyecto no sólo importa el código, sino que también es necesario considerar muchos otros factores.

# Glosario de términos

|  |  |
| --- | --- |
| Término | Definición |
| ANÁLISIS DE CÓDIGO ESTÁTICO | Es un tipo de análisis que se realiza sin ejecutar el programa. |
| API | Es el conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software. |
| API-REST | Interfaz web simple que utiliza XML y HTTP, para ofrecer operaciones CRUD. Cada recurso es direccionable únicamente a través de su URI. |
| BASE DE DATOS | Es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso. |
| BUGS | Es un error o fallo en un programa de computador o sistema de software. |
| CLASES | Abstracción que define un tipo de objeto especificando qué propiedades (atributos) y operaciones disponibles va a tener. |
| CONTROLADOR | Orquestador que maneja las solicitudes de las acciones realizadas por el usuario en la vista, actualiza el modelo y dirige a los usuarios la vista apropiada basándose en el resultado de la ejecución. |
| COMMIT | Consigna un conjunto de cambios sobre un repositorio Git local. |
| COMPILACIÓN | Traducir con un compilador un programa en lenguaje de alto nivel a lenguaje de la máquina. |
| CRUD | Crear, Obtener, Actualizar y Borrar |
| DESPLIEGUE | Es la acción de generar un .war y un script creación de base de datos a fin de pasar una aplicación en desarrollo a estado de producción. |
| DRIVER | Es un controlador que sirve para mantener una comunicación con la base de datos para poder realizar operaciones CRUD sobre ella. |
| FRAMEWORK | Es una estructura tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que sirve de base para la organización y desarrollo de software. |
| GITHUB | Es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git. |
| HOSTING | Es el servicio que provee a los usuarios de Internet un sistema para poder almacenar cualquier tipo de información vía web. |
| HTTPS | Protocolo seguro de transferencia de hipertexto, es un protocolo de aplicación basado en HTTP. |
| MERGE | Es una fusión de dos ramas en un repositorio de código. |
| MÉTODOS | Es un conjunto de instrucciones definidas dentro de una clase |
| RAMA | Es el conjunto de commits y pushes a lo largo del tiempo que guardan una relación incremental y que persiguen un determinado objetivo de principio a fin. |
| PLUGINS | Es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica. |
| PULL | Es el comando encargado de recibir los cambios que se alojan en el servidor remoto Git. |
| PUSH | Es el comando encargado de enviar los cambios al servidor remoto Git. |
| REPOSITORIO | Es el lugar en el que se almacenan los datos actualizados e históricos de cambios. |
| SSH | Intérprete de órdenes segura, sirve para acceder a máquinas remotas a través de una red. |
| TIME MACHINE | Es un histórico sobre el que se guardan datos y que posteriormente se utilizarán para inferir conclusiones sobre el conjunto de ellos. |
| WAR | Es un archivo JAR utilizado para distribuir una colección de JavaServer Pages, servlets, clases Java, archivos XML, librerías de tags y páginas web estáticas (HTML y archivos relacionados) que juntos constituyen una aplicación web. |

# Bibliografía

**Gestión del código fuente**

* Tutorial en AdictosAlTrabajo:

<http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=git-branch-bash#3.%20Branch%20en%20Git>

* AprendeGit: <http://aprendegit.com/git-flow-hotfix-branches/>
* Blog Endpoint: <http://blog.endpoint.com/2014/05/git-workflows-that-work.html>
* Wiki de EGC: <https://1984.lsi.us.es/wiki-egc/index.php/Pr%C3%A1ctica_2_14-15>

**Gestión de la construcción y despliegue**

**Gestión de la construcción e integración continua**

**Gestión de la calidad**

* Comunidad de desarrolladores StackOverflow: <http://www.stackoverflow.com>
* Documentación oficial de SonarQube:

<http://docs.sonarqube.org/display/SONAR/Documentation>

* G Developer - Instalación y configuración de SonarQube:

<http://alejosd.blogspot.com.es/2014/10/instalacion-y-configuracion-de-sonarqube.html>

* G Developer - Instalación y configuración Jenkins con SonarQube:

<http://alejosd.blogspot.com.es/2014/10/instalacion-y-configuracion-jenkins-con.html>

* SQALE - Artículo de Wikipedia: <http://en.wikipedia.org/wiki/SQALE>

**Gestión del cambio, incidencias y depuración**

**Mapa de herramientas**

* Wikipedia - Java: <https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)>
* Wikipedia - Eclipse: <http://es.wikipedia.org/wiki/Eclipse_%28software%29>
* Wikipedia - Maven: <http://es.wikipedia.org/wiki/Maven>
* Wikipedia - Spring: <http://es.wikipedia.org/wiki/Spring_Framework>
* Wikipedia - Hibernate: <http://es.wikipedia.org/wiki/Hibernate>
* Wikipedia - Git: <http://es.wikipedia.org/wiki/Git>
* Wikipedia - Jenkins: <http://es.wikipedia.org/wiki/Jenkins>
* Wikipedia - SonarQube: <http://en.wikipedia.org/wiki/SonarQube>
* Wikipedia - OpenShift: <http://es.wikipedia.org/wiki/OpenShift>

**Integración de todos los subsistemas**

**Glosario de términos**

* Conociendo GitHub: <http://conociendogithub.readthedocs.org/en/latest/data/introduccion/>
* Wikipedia: <https://es.wikipedia.org>
* Git – Book: <http://git-scm.com/book/es/v1>

# Anexos

[Xory y JP: Instalación de las máquinas virtuales]