**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 | Fecha de mentira | Bla | Bla |
| 2.2 | Fecha de mentira | Bla | Bla |

1. Índice

[Resumen 3](#_Toc410778989)

[Historial de versiones 3](#_Toc410778990)

[1. Índice 5](#_Toc410778991)

[2. Índice de figuras 6](#_Toc410778992)

[3. Roles 7](#_Toc410778993)

[4. Introducción 7](#_Toc410778994)

[5. Gestión del código fuente 10](#_Toc410778995)

[**5.1.** **Sintaxis del código fuente** 10](#_Toc410778996)

[**5.2.** **Gestión de ramas** 11](#_Toc410778997)

[**5.2.1.** **Gestión interna** 11](#_Toc410778998)

[**5.2.2.** **Gestión externa** 13](#_Toc410778999)

[**5.3.** **Creación y aplicación de *patch*** 14](#_Toc410779000)

[**5.4.** **Roles** 15](#_Toc410779001)

[**5.5.** **Conflictos** 15](#_Toc410779002)

[**5.6.** **Migración de SVN a Git** 16](#_Toc410779003)

[**5.7.** **Herramientas similares** 16](#_Toc410779004)

[**5.8.** **Ejercicios** 16](#_Toc410779005)

[6. Gestión de la construcción y despliegue 16](#_Toc410779006)

[**6.1.** **Maven** 16](#_Toc410779007)

[**6.2.** **Openshift** 16](#_Toc410779008)

[**6.3.** **Ejercicios** 17](#_Toc410779009)

[7. Gestión de la construcción e integración continua 17](#_Toc410779010)

[**7.1.** **Jenkins** 17](#_Toc410779011)

[**7.2.** **Ejercicios** 18](#_Toc410779012)

[8. Gestión de la calidad 18](#_Toc410779013)

[9. Gestión del cambio, incidencias y depuración 18](#_Toc410779014)

[**9.1.** **Gestión del cambio** 18](#_Toc410779015)

[**9.2.** **Gestión de las incidencias** 18](#_Toc410779016)

[**9.3.** **Depuración** 18](#_Toc410779017)

[**9.4.** **Ejercicios** 18](#_Toc410779018)

[10. Mapa de herramientas 18](#_Toc410779019)

[**10.1.** **Lista de herramientas** 18](#_Toc410779020)

[**10.2.** **Visión interna del mapa de herramientas** 20](#_Toc410779021)

[**10.3.** **Visión global del mapa de herramientas** 21](#_Toc410779022)

[11. Integración de todos los subsistemas 22](#_Toc410779023)

[**11.1.** **Gestión del código** 22](#_Toc410779024)

[**11.2.** **Estrategia de integración** 22](#_Toc410779025)

[**11.3.** **Herramientas** 22](#_Toc410779026)

[**11.4.** **Automatización de la construcción** 22](#_Toc410779027)

[12. Conclusiones 23](#_Toc410779028)

[Bibliografía 23](#_Toc410779029)

[**Gestión de la construcción y despliegue** 23](#_Toc410779030)

[**Gestión de la construcción e integración continua** 23](#_Toc410779031)

[**Gestión de la calidad** 23](#_Toc410779032)

[**Gestión del cambio, incidencias y depuración** 23](#_Toc410779033)

[**Mapa de herramientas** 23](#_Toc410779034)

[**Integración de todos los subsistemas** 23](#_Toc410779035)

[Glosario de términos 23](#_Toc410779036)

[Anexos 23](#_Toc410779037)

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 4: Esquema de jerarquización de ramas....................................................................X**

**Figura 5: Esquema de jerarquización de ramas....................................................................X**

**Figura 6: Esquema de jerarquización de ramas....................................................................X**

**Figura 7: Esquema de jerarquización de ramas....................................................................X**

**Figura X: Visión interna del mapa de herramientas............................................................X**

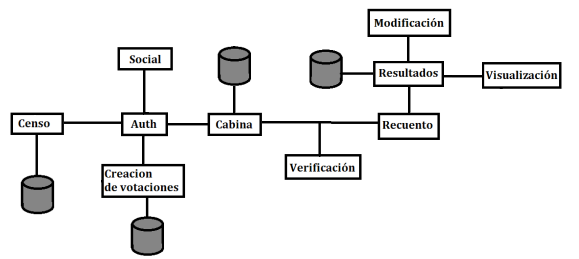
**Figura X: Visión global del mapa de herramientas...............................................................X**

# 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:



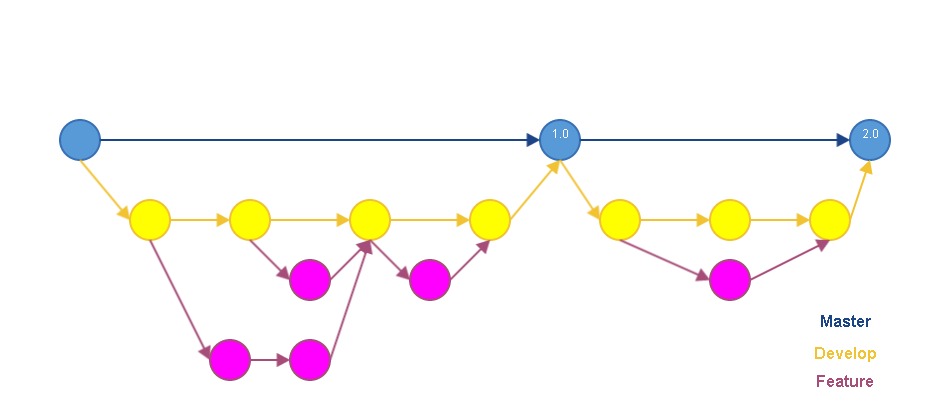
*Figura 2: Esquema de jerarquización de ramas*

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 una serie de ejercicios donde se ponen en práctica las políticas y decisiones anteriormente expuestas en relación a la gestión del código fuente, y la interacción de los integrantes del grupo con el repositorio donde se almacena dicho código. Estos ejercicios se centran en el uso del sistema de control de versiones Git y no en SVN, puesto que ha sido Git el sistema de control de versiones que se ha decidido usar finalmente.

### 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.

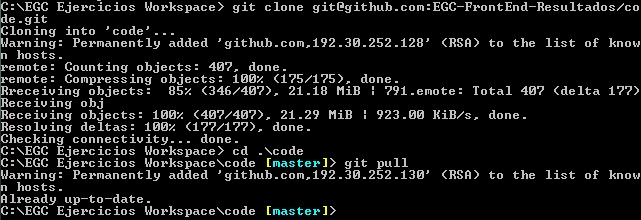
Cuando se haya implementado completamente la nueva funcionalidad y se haya notificado al resto del grupo, el desarrollador deberá hacer un *git merge* de la rama “Welcome\_Dev” con la rama “master” del repositorio, tras lo cual se realiza un último *git push* al repositorio.

**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***

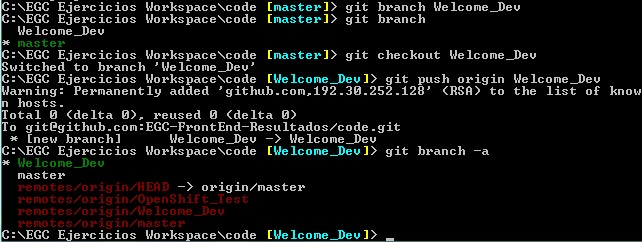
******

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

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

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

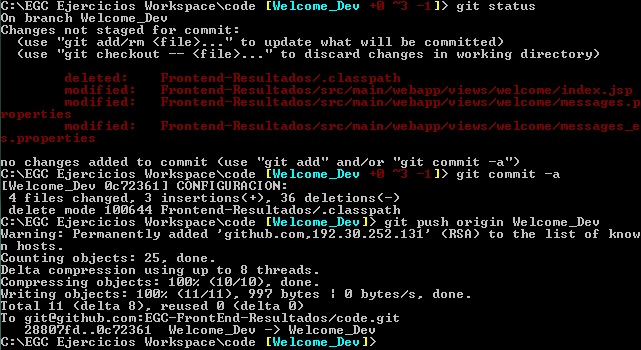
***git push*** *origin Welcome\_Dev*

******

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

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

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

******

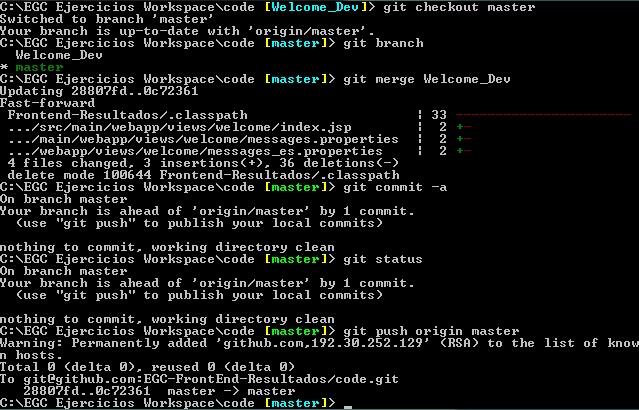
*Figura X: Ejercicio Git 1: Uso de ramas 3*

***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 dos de los desarrolladores del grupo se disponen a implementar la característica descrita en el siguiente enunciado al mismo tiempo, por lo que se producirá un conflicto cuando el último que termine envíe sus cambios al repositorio. 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.

**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*

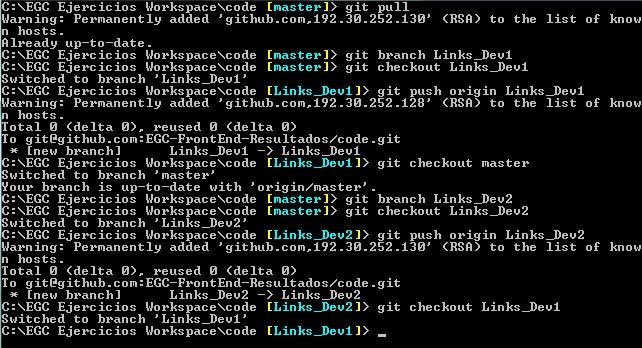
***git checkout*** *master*

***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*

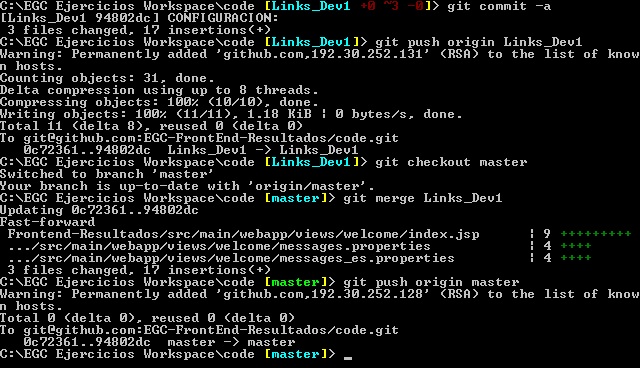
***git commit*** *–a* (Tras realizar todos los cambios en Links\_Dev1)

***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*

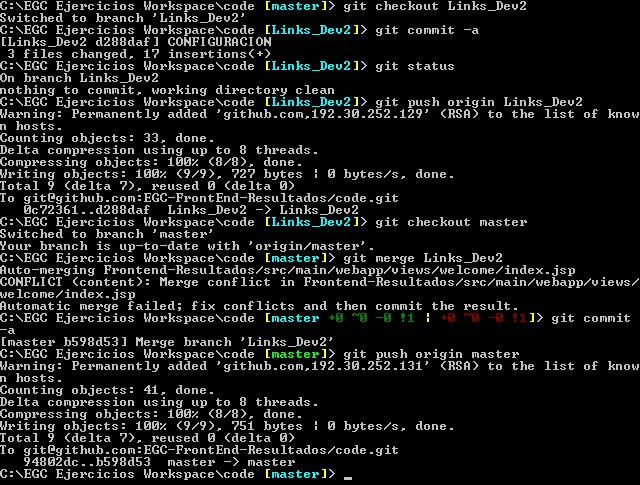
***git push*** *origin Links\_Dev2*

***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*.

**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*

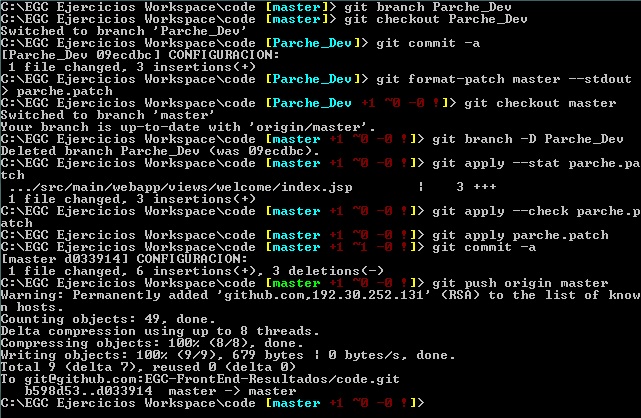
***git apply*** *--stat parche.patch*

***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**

[Xory y yo]

* 1. **Ejercicios**

[Xory y yo]

# Gestión de la calidad

[Daniel: un parrafito y luego las secciones que quieras (al final recuerda ejercicios)]

# Gestión del cambio, incidencias y depuración

[Ale]

* 1. **Gestión del cambio**

[Ale]

* 1. **Gestión de las incidencias**

[Ale]

* 1. **Depuración**

[Ale]

* 1. **Ejercicios**

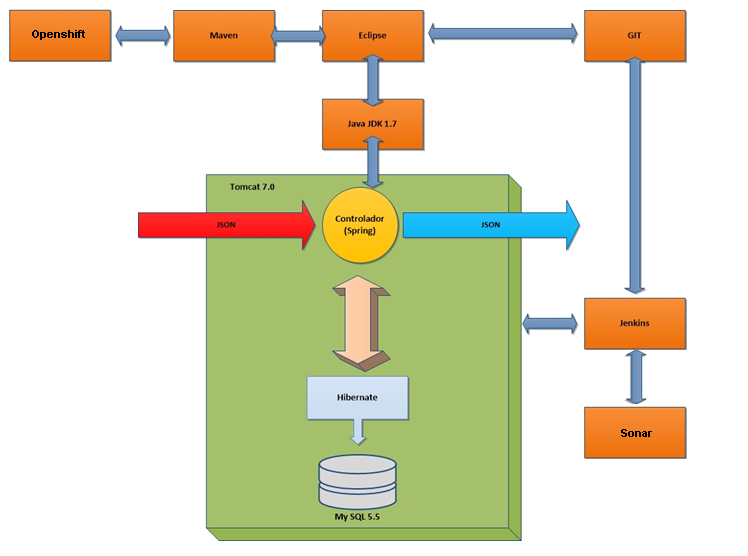
[Ale: un ejercicio que incluya un ejemplo de las tres gestiones (añadiendo funcionalidad de que las votaciones puedas ser de otra forma)]

# Mapa de herramientas

* 1. **Lista de herramientas**

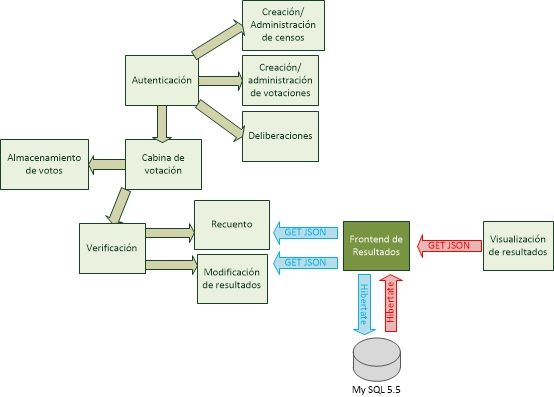
|  |  |
| --- | --- |
| * **Java**   Será el lenguaje de programación que utilizaremos para implementar nuestro subsistema, ya que está bastante extendido y tenemos bastante experiencia con él.  Es un lenguaje de programación de propósito general orientado a objetos, derivado en gran medida de C y C++. | **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 nos provee todo lo necesario para crear, compilar e interpretar nuestro 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 desarrollaremos nuestro 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

[JP: rehacerlas enteras y con más cosas]

# Bibliografía

**Gestión del código fuente**

* <http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=git-branch-bash#3.%20Branch%20en%20Git>
* <http://aprendegit.com/git-flow-hotfix-branches/>
* <http://blog.endpoint.com/2014/05/git-workflows-that-work.html>
* <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**

**Gestión del cambio, incidencias y depuración**

**Mapa de herramientas**

**Integración de todos los subsistemas**

# Glosario de términos

[Álvaro: mirar el grupo 7 (**pero no copypaste**) y que tenga los términos de esta memoria]

# Anexos

[Xory y JP: Instalación de las máquinas virtuales]