10/01/17

Miembros

Doncel Ramírez, Andrés

Gavilán Ruiz, José

Jiménez Ríos, Andrés Miguel

Menéndez Montes, Eva

Ramos González, José Renato

Frontend y visualización de resultados

Documentación del proyecto

Índice

[1. Control de versiones 3](#_Toc471555421)

[2. Resumen 4](#_Toc471555422)

[3. Introducción y contexto 4](#_Toc471555423)

[4. Descripción del sistema 6](#_Toc471555424)

[4.1. Descripción de los componentes e integración con otros subsistemas 6](#_Toc471555425)

[4.2. Cambios desarrollados 6](#_Toc471555426)

[4.3. Planificación del trabajo 7](#_Toc471555427)

[5. Elementos de control 8](#_Toc471555428)

[5.1. Elementos de configuración: 8](#_Toc471555429)

[5.2. Artefactos: 8](#_Toc471555430)

[5.3. Gestión para el cambio de artefactos 8](#_Toc471555431)

[6. Entorno de desarrollo 9](#_Toc471555432)

[6.1. Máquina virtual 9](#_Toc471555433)

[6.2. NodeJS 9](#_Toc471555434)

[6.3. Atom 10](#_Toc471555435)

[6.4. Git 10](#_Toc471555436)

[7. Gestión del código fuente 11](#_Toc471555437)

[7.1. Herramientas 11](#_Toc471555438)

[7.2. Gestión de ramas 11](#_Toc471555439)

[7.2.1. Ejemplo 1 12](#_Toc471555440)

[7.2.2. Ejemplo 2 13](#_Toc471555441)

[7.3. Roles 14](#_Toc471555442)

[7.4. Gestión de parches 14](#_Toc471555443)

[7.4.1. Ejemplo 14](#_Toc471555444)

[7.5. Aprobación de cambios 15](#_Toc471555445)

[7.6. Políticas de nombre y estilo 15](#_Toc471555446)

[8. Gestión de la construcción e integración continua 16](#_Toc471555447)

[8.1. Docker 19](#_Toc471555448)

[8.2. Travis 19](#_Toc471555449)

[9. Gestión del cambio, incidencias y depuración 20](#_Toc471555450)

[9.1. Mecanismos de depuración 20](#_Toc471555451)

[9.2. Procesos de gestión 20](#_Toc471555452)

[9.2.1. Incidencia interna 20](#_Toc471555453)

[9.2.2. Ejemplo 20](#_Toc471555454)

[9.2.3. Incidencia externa 21](#_Toc471555455)

[9.2.4. Ejemplo 22](#_Toc471555456)

[9.2.5. Auto asignación 24](#_Toc471555457)

[9.2.6. Ejemplo 24](#_Toc471555458)

[9.3. Roles 25](#_Toc471555459)

[9.4. Estados 26](#_Toc471555460)

[9.4.1. Prioridad y estado de incidencias 26](#_Toc471555461)

[9.4.2. Ejemplo 27](#_Toc471555462)

[9.5. Políticas para descartar, fomentar o retardar un cambio 28](#_Toc471555463)

[10. Gestión de liberaciones y despliegue 29](#_Toc471555464)

[11. Mapa de herramientas 30](#_Toc471555465)

[11.1. Lista de herramientas 30](#_Toc471555466)

[11.2. Mapa de herramientas 32](#_Toc471555467)

[12. Conclusiones 32](#_Toc471555468)

[12.1. Cambios propuestos 33](#_Toc471555469)

# Control de versiones

|  |  |  |  |
| --- | --- | --- | --- |
| Versión | Autor | Fecha | Cambios |
| 1.0.0 | José Renato Ramos González | 05/01/2017 | Esquema del documento, apartados principales cohesionados |
| 1.0.1 | José Renato Ramos González | 05/01/2017 | Erratas y faltas de ortografía |
| 1.1.0 | José Renato Ramos González | 05/01/2017 | Añadido comentario sobre la integración de Travis CI |
| 1.1.1 | Andrés Doncel Ramírez | 05/01/2017 | Eliminados párrafos duplicados |
| 1.2.0 | Andrés Doncel Ramírez | 06/01/2017 | Eliminadas duplicidades, primeras conclusiones |
| 1.3.0 | Andrés Miguel Jiménez Ríos | 07/01/2017 | Detallado etiquetado y proceso de liberación de entregables |
| 1.3.1 | Andrés Doncel Ramírez | 07/01/2017 | Control de versiones, adaptado el formato de la nueva versión |
| 1.3.2 | Eva Menéndez Montes | 07/01/2017 | Añadidas conclusiones al resumen. |

# Resumen

El objetivo principal de este trabajo consiste en la realización de una mejora del proyecto Agora US, que es una aplicación encargada de gestionar un sistema de votaciones, y que se divide en varios subsistemas. En este caso, el grupo será el responsable del módulo Frontend y Visualización de resultados, que se encarga de visualizar los resultados de las distintas votaciones, siendo necesaria una integración con otro de los subsistemas existentes, el de Recuento y Modificación de Resultados.

Para llevar a cabo esta tarea, hemos usado las siguientes herramientas:

* Para la gestión del código fuente, hemos utilizado Git.
* Para la gestión de incidencias, utilizamos las issues de GitHub.
* Para la automatización de la construcción utilizamos Jenkins, configurado en un entorno adecuado en Docker. Además, también se encarga del despliegue, la entrega y la integración del sistema.
* Para la automatización de pruebas, utilizamos Travis.
* Como sistema de comunicación entre los miembros del grupo, usamos Telegram.

Por último, las mejoras que se han llevado a cabo, resumidas muy brevemente, han sido las siguientes:

* Cambio de Spring a NodeJS.
* Opción a cambio de idioma.
* Correcciones en el diseño.
* Integración continua con el resto de subsistemas.
* Integración con el módulo de Recuento y Modificación de resultados.
* Refactorización del código.
* Realización de pruebas con los framework Karma y Jasmine.

Entre las conclusiones más importantes que hemos sacado del trabajo realizado en este proyecto, se encuentran la necesidad de familiarizarnos lo antes posible con los procesos de gestión de código y de incidencias, con el fin de facilitarnos el trabajo, la utilidad de la realización de un diario semanal para tener un mayor control de las tareas realizadas por cada integrante del grupo, y la importancia de una buena comunicación tanto entre los miembros del equipo como con otras partes implicadas, como pueden ser los responsables de otros módulos.

# Introducción y contexto

En esta sección se va a proceder lo que se ha realizado en el proyecto desde que se heredó el código hasta que se ha finalizado el proyecto de forma escueta.

Agora US es un sistema de votaciones online creado por alumnos de la escuela de informática de la Universidad de Sevilla. Este proyecto está dividido en numerosos módulos, una de las cuales es Frontend y visualización de resultados, de la cual se encarga de mejorar este grupo.



Ilustración 1

Cuando se heredó dicho módulo, el código estaba compuesto por la parte de lógica, hecha en Spring y la parte de vistas, hecha en Angular. Lo primero que hizo nuestro grupo con respecto a lo heredado fue pasar toda la parte de Spring a NodeJS, y hacerla compatible con la parte de Angular que habíamos heredado.

Además de eso, realizamos la integración con el módulo de recuento, el cual nos provee de los datos que nuestro módulo muestra y con el cual hemos estado en constante comunicación para que la integración se llevase a cabo con éxito. Esta integración se basa en llamadas a la API del módulo de recuento, tras la cual recibe un JSON con la información requerida.

También se ha añadido soporte a añadir idiomas al sistema (de momento español e inglés) mediante Angular Translate, funcionalidad que podía ser aportada con Spring (aunque no había sido llevada a cabo por el grupo del que heredamos el proyecto).

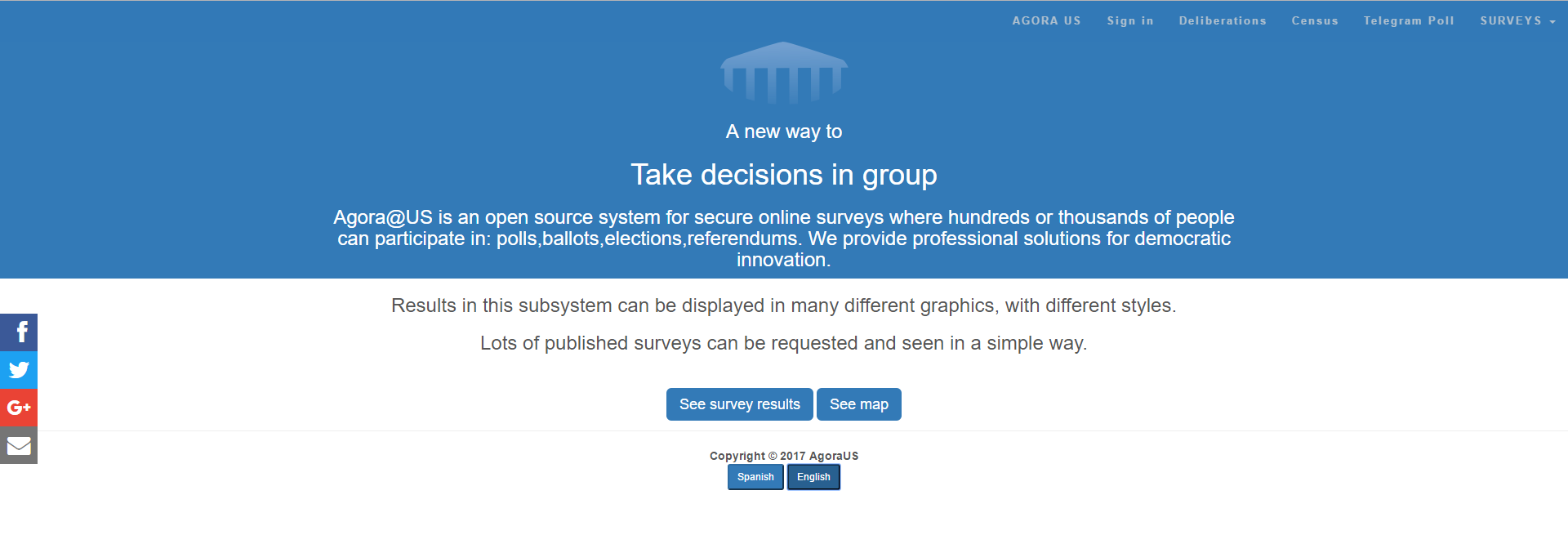


Ilustración 2

Por último, se han añadido tests al sistema para probar su buen funcionamiento, utilizando Karma y Jasmine, los cuales son entornos que facilitan la realización y ejecución de los tests.

Además de todo lo añadido al módulo en sí, también se ha logrado la integración continua del mismo utilizando Jenkins para tener el módulo siempre actualizado en la versión desplegada, y utilizando Travis para comprobar que se pasan los tests y el sistema sigue funcionando.

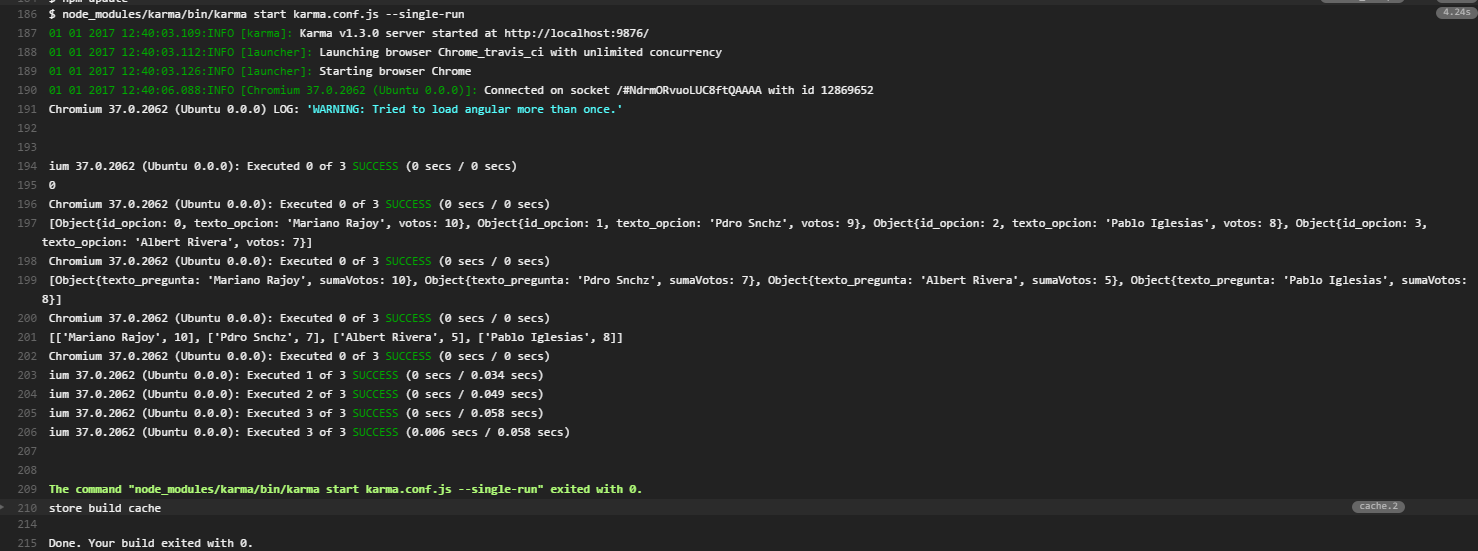


Ilustración 3

# Descripción del sistema

## Descripción de los componentes e integración con otros subsistemas

Como se ha explicado en los apartados anteriores, nuestro grupo es el encargado del subsistema de Frontend y Visualización, que tiene como objetivo ofrecer al usuario la visualización de los resultados de las votaciones. Esta información se muestra en forma de gráficas, con un diseño que se ha intentado que sea lo más atractivo posible.

Nuestro módulo recibe el recuento de votos de otro de los subsistemas de Agora US, el de Recuento y Modificación de resultados. Esta comunicación que tiene lugar entre los dos módulos se lleva a cabo mediante el uso de una API, enviando datos como objetos JSON.

Describiendo nuestro sistema desde un punto de vista más técnico, podemos destacar la migración realizada a NodeJS que explicamos más adelante, y el uso de los framework Jasmine y Karma para los tests, así como Angular para las vistas.

## Cambios desarrollados

Las mejoras realizadas en nuestro proyecto han sido las siguientes:

* Se ha cambiado el backend de Spring a NodeJS, así como el controlador encargado de realizar la simulación de conectarse con Recuento, este último con fines de testing, pudiendo funcionar nuestro proyecto aun en el supuesto de que el módulo del que dependemos no funcionara correctamente.
* Se ha añadido la opción de cambio de idioma, estando nuestra aplicación disponible tanto en español como en inglés.
* Se han llevado a cabo correcciones en el diseño.
* Con el objetivo de facilitar el desarrollo y la integración de los distintos subsistemas, se ha integrado nuestro proyecto con un servidor de integración continua.
* Nuestro proyecto se encuentra completamente integrado con el módulo de Recuento y Modificación de resultados.
* Se ha realizado una refactorización del código, eliminando librerías CSS y JavaScript que se incluían en el proyecto heredado que ya no tenían uso y unificando módulos. El mismo proceso se ha realizado con las imágenes.
* Se han añadido los framework de pruebas Karma y Jasmine, con el fin de comprobar los controladores en Angular.

## Planificación del trabajo

La planificación del trabajo que hemos realizado y el reparto de todas las tareas se muestran en la siguiente tabla:

|  |  |
| --- | --- |
| Tarea | Encargado |
| Migración a Node.js | José Renato Ramos González |
| Integración con Recuento | José Gavilán Ruiz |
| Internacionalización | Andrés Doncel Ramírez |
| Integración con redes sociales | Andrés Miguel Jiménez Ríos |
| Refactorización de código | Eva Menéndez Montes |
| Integración continua | Andrés Miguel Jiménez Ríos |
| Cambios en el back-end | José Renato Ramos González |
| Realización de pruebas | Andrés Doncel Ramírez |
| Creación máquina virtual | José Gavilán Ruiz |
| Cambios en el diseño | Andrés Miguel Jiménez Ríos |
| Documentación | Todos |
| Diario | Eva Menéndez Montes |

Tabla 1

De forma que todos los integrantes del grupo han participado tanto en el desarrollo del proyecto como en la documentación, para poder poner en práctica todos los conocimientos adquiridos en la asignatura.

# Elementos de control

## Elementos de configuración:

* Memoria del proyecto
* Máquina virtual de desarrollo
* Código fuente
* Dependencias NPM

## Artefactos:

La nomenclatura utilizada es la estándar para paquetes NPM.

|  |  |
| --- | --- |
| Artefactos | Versión |
| express | 4.14.0 |
| mongoose | 4.7.5 |
| nodemon | 1.11.0 |
| jasmine-core | 2.5.2 |
| angular | 1.6.1 |
| angular-route | 1.6.1 |
| angular-mocks | 1.6.1 |
| angular-ui-router | 0.3.2 |
| karma | 1.3.0 |
| karma-cli | 1.0.1 |
| karma-jasmine | 1.1.0 |
| karma-chrome-launcher | 2.0.0 |

## Gestión para el cambio de artefactos

En el caso de que sea necesario añadir o modificar un componente se deberá:

1. Realizar pruebas con la versión deseada para comprobar y verificar que todo funciona como antes, esto incluye la ejecución de todos los tests.
2. En el caso de incompatibilidad, buscar otra versión (nunca más antigua de la versión inicial si se trata de una actualización) y volver al paso anterior, o abortar el proceso.
3. Documentar los cambios realizados.

# Entorno de desarrollo

A continuación, se van a detallar las herramientas que se han utilizado para desarrollar en el grupo de Visualización y Frontend:

## Máquina virtual

Como programa de virtualización hemos utilizado Virtual Box, ya que es uno de los más conocidos, utilizado previamente por el grupo y bastante fiable. El hecho de utilizar una máquina virtual ayuda en la instalación del entorno tanto a nuestro grupo, que trabajará con entornos exactamente iguales, como al grupo que utilice nuestro entregable como heredado, ya que no necesitará investigar en exceso como instalar todos los componentes. El equipo ha trabajado con diferentes versiones de Virtual Box al tenerlo instalado de asignaturas anteriores sin problemas.

<https://www.virtualbox.org/wiki/Downloads>



En cuanto a la máquina virtual en sí, se ha utilizado Ubuntu en su versión 16.04 LTS, a la cual se le otorgaron 2 procesadores y 2GB de RAM, con lo que, en principio, se ha obtenido un rendimiento bastante aceptable, además de ser una configuración asequible para prácticamente cualquier equipo.

<https://www.ubuntu.com/download/desktop>

## NodeJS

Para NodeJS hemos utilizado la versión 6.9.2 LTS, ya que es la última versión estable de Node. Para instalar paquetes de Node se ha utilizado npm, el cual es un instalador bastante sencillo de utilizar. Algunos paquetes utilizados son nodemon (para que el módulo se ejecute en fase de pruebas directamente después de realizar cambios sin reiniciar a mano el servidor), mongoose (para modelar los datos almacenados en mongoDB en el servidor de pruebas), Jasmine y Karma para los tests y Angular para las vistas.



## Atom

Hemos utilizado Atom como editor para desarrollar el módulo. Es un editor bastante simple y a la vez muy personalizable gracias a la cantidad de plugins que se pueden añadir. Además, como está desarrollado por Github cuenta con funciones muy interesantes, como un indicador de las líneas que han sido editadas, o marcas con colores de los ficheros que han sido modificados o añadidos con colores, de forma que se pueda ver fácilmente.

<https://nodejs.org/en/download/>

## Git

Como hemos utilizado GitHub para compartir el código, es necesario el uso de Git para mantener el control de versiones y el código compartido entre todos los miembros del grupo, además de tener siempre el sistema desplegado lo más actualizado posible.

<https://git-scm.com/downloads>

# Gestión del código fuente

## Herramientas

Para la gestión del código se usará la herramienta de control de versiones Git[[1]](#footnote-1), así como la plataforma GitHub[[2]](#footnote-2) para la coordinación de todos los miembros del equipo. En lugar de utilizar un plugin para nuestro IDE (Atom), hemos decidido usar la consola de comandos para realizar las operaciones de gestión de código, dado que, una vez acostumbrados a ello, sentimos que la eficiencia del equipo sería mayor.

## Gestión de ramas

La gestión de ramas en el repositorio remoto y sobre el que se articulará el trabajo del equipo son dos: “master” y “dev”.

Existe una tercera rama “legacy” en la que mantenemos el código utilizado por el equipo del año pasado, que utilizó una tecnología diferente para desarrollar el sistema. Nosotros hemos cogido los ficheros necesarios de esa rama para trabajar en la mejora del sistema.

Las ramas principales tienen dos objetivos muy claros, que describimos a continuación:

* Rama master: aquí se mantendrán las versiones estables del sistema. La última versión de esta rama será nuestra línea base a la hora de desarrollar el código. Esta rama solo se conecta con la rama dev, es decir, solo se pueden obtener modificaciones de esta rama, y no de cualquier forma, sino siguiendo un procedimiento de control de cambios.
* Rama dev: los desarrolladores deben trabajar y subir sus cambios en esta rama, antes de que puedan pasar formalmente a la siguiente versión estable. A la hora de subir los cambios al repositorio remoto, se pueden hacer directamente sobre esta rama, siempre que no deje al sistema inservible. Esto es, bajo ningún concepto está permitido subir código con errores de compilación, si fuera el caso, o, si produce un error en el despliegue o en la integración continua, deben deshacerse los cambios de forma inmediata. En este último caso, deberá seguirse el proceso de rama auxiliar que describiremos más adelante. Aunque sea en desarrollo, se tratará que esta versión esté funcional el mayor tiempo posible, dado que ofreceremos al resto de subsistemas la oportunidad de acceder a la versión en desarrollo para las nuevas funcionalidades que puedan requerir, antes de que sean liberadas.

También existe una rama docs, donde se almacenan todos los documentos generados sobre el proyecto, a fin de mantenerlos y actualizarlos para ofrecer una documentación profesional.

De forma local, se puede trabajar directamente en la rama dev para cambios menores, siempre que, antes de subirlo a remoto, no se incumplan las restricciones descritas antes. Para el resto de cambios, se define el siguiente procedimiento de gestión de código, basado en la creación de ramas funcionales:

* Crear una rama nueva a partir de la última versión en dev, con un nombre que sea lo suficientemente descriptivo del cambio que se quiere implementar (corrección de bugs, tests, funcionalidades, etc.).
* Si la tarea está asignada a varios desarrolladores, esta rama debe ser subida al repositorio remoto, y debe ser traída a su repositorio local por todos los implicados.
* Trabajar en esa rama normalmente, tratando de implementar un cambio lógico y funcional con cada commit. Se debe hacer referencia a la incidencia, si la hubiere. Antes de comenzar a trabajar, siempre se deben traer los cambios más recientes de la rama, para facilitar y acelerar la resolución de conflictos.
* Si las modificaciones realizadas no han supuesto la implementación completa del cambio existen varios posibles cursos de acción:
  + Si se está trabajando con otros compañeros en esa rama, se deben subir los cambios al repositorio remoto en esa rama (tratando de no romper el software, aunque no es imprescindible en las ramas auxiliares).
  + Si sólo hay una persona asignada, pero se pide la verificación de algún cambio implementado, la ayuda para la implementación del mismo, o simplemente un cambio en la asignación, se debe realizar el mismo paso que en el caso anterior (subiendo la rama si no se hizo previamente).
  + Si se está trabajando sólo en ese cambio, y se va a continuar trabajando más adelante, no es necesario subir la rama al repositorio remoto.
* Si las modificaciones realizadas suponen la total implementación del cambio en cuestión, se debe hacer lo siguiente:
  + El commit que suponga el fin del cambio debe hacer referencia a la resolución de la incidencia, si la hubiere.
  + Si se está trabajando en solitario en ese cambio y no fue necesario subir la rama auxiliar, se puede integrar directamente en la rama dev y subir esta.
  + Si, por el contrario, la rama está subida, todos los miembros asignados deben comprobar la validez del cambio. Una vez aceptado, se pueden unir los cambios con la rama dev.

Usando las herramientas comentadas anteriormente, el proceso es apoyado por una variedad de comandos útiles que se corresponden de forma casi directa con el propio proceso. Algunos de ellos, los más utilizados:

* Gestión de ramas: comandos branch, merge, checkout
* Repositorios local y remoto: comandos pull, push, fetch

### Ejemplo 1

En una incidencia se ha propuesto añadir interacción del sistema con las redes sociales. El cambio ha sido aprobado y asignado a un único compañero y va a comenzar a trabajar. La gestión de la incidencia se va desarrollando conforme a lo establecido en su correspondiente sección.

1. El desarrollador trae los últimos cambios en dev a su repositorio local.
2. Crea una rama llamada redes sociales.
3. Empieza a trabajar en esa rama y va haciendo commits: uno por cada red social con la que se integra. Hoy hace dos integraciones: con Facebook y Twitter.
4. Como sólo él trabaja en este incremento y no tiene problemas en el desarrollo, no necesita subir la rama a GitHub.
5. Al día siguiente, hace la integración con Google+ y da por finalizado el incremento. El commit especifica que completa la incidencia según lo descrito en el apartado de Gestión de Incidencias.
6. Integra la rama redes sociales con la rama dev y la sube a GitHub.
7. Como este cambio no ha sido verificado debidamente por otro miembro del equipo, el coordinador debe hacerlo antes de que la versión pase a la rama master.

### Ejemplo 2

En una incidencia se ha propuesto el uso de un framework de pruebas de los controladores de Angular para utilizar la integración continua y automatizar las pruebas en el Frontend. El cambio ha sido aprobado y asignado a varios compañeros y van a comenzar a trabajar. La gestión de la incidencia se va desarrollando conforme a lo establecido en su correspondiente sección.

1. Uno de los desarrolladores (el primero asignado, lo llamaremos responsable) trae los últimos cambios en dev a su repositorio local
2. Crea una rama llamada tests a partir de dev y la sube al repositorio.
3. El resto de desarrolladores bajan esa rama y trabajan en ella.
4. Cada desarrollador va haciendo commits locales con cada cambio lógico realizado (una prueba añadida, cambios en los ficheros de configuración de integración continua, etc.).
5. Cuando terminan de trabajar, deben subir sus cambios al repositorio remoto. El último commit realizado debe referenciar a la incidencia anterior. Si hay conflictos, estos deben ser resueltos por la persona que los ha encontrado (probablemente, el último en subir los cambios). Es obligatorio que cada miembro haya subido un commit funcional al dejar de trabajar.
6. Al día siguiente, cada miembro repite el paso anterior y completan la implementación de las pruebas.
7. Se deben subir todos los cambios a la rama tests y hacer las pruebas de integración continua. El responsable debe, una vez verificado, indicar con el último commit que la incidencia está resuelta.
8. Los compañeros deben verificar los resultados de la integración y confirmar el paso a dev.
9. El responsable integra la rama tests con la rama dev y la sube a GitHub.
10. Como este cambio ha sido verificado debidamente por otros miembros del equipo, el coordinador no tiene que hacerlo cuando se produzca la nueva entrega.

En general, no hemos tenido demasiados problemas a la hora de gestionar el código con este proceso. Los conflictos han sido escasos a la hora de juntar las ramas, en parte porque cada miembro del equipo se ha ido ocupando de una parte del sistema, de manera que se minimizaban el trabajo en los mismos ficheros concurrentemente.

Además, dado que las funcionalidades de calado deben ser probadas en una rama aparte, se evitan problemas en la versión de desarrollo y en la estable. Por otro lado, los cambios menores, que no requerían de una rama nueva, se podían implementar sobre la versión de desarrollo directamente, evitando así sobrecargar la gestión con ramas que aportan poco valor al desarrollo.

## Roles

Existen varios roles en la gestión del código que hemos definido, que se encargan de diferentes partes del proceso. Estos roles, en general, son asignados cuando una incidencia es aprobada, salvo en casos particulares:

* Desarrollador: miembros del equipo que producen incrementos en el subsistema de Frontend. Todos los miembros son desarrolladores. Los desarrolladores asignados a un incremento deben comprobar, en grupo o individualmente, que el incremento está funcionando y cubierto con las pruebas necesarias. Se necesita la aprobación de al menos otro desarrollador, a parte del responsable, para poder pasar el incremento a la rama de desarrollo.
* Responsable de un incremento: Una vez aprobado un cambio a través de una incidencia, se asignan uno o varios desarrolladores a producir el incremento. El primero de ellos es el responsable del incremento. El responsable debe, según lo definido en el proceso anterior, crear la rama auxiliar y mantener esa rama hasta su unión con la rama de desarrollo. También puede delegar esta tarea a uno de los desarrolladores, de manera que quede constancia de tal aprobación. El responsable también es un desarrollador.
* Jefe de proyecto: este rol es fijo durante el desarrollo del proyecto. También es un desarrollador. El jefe del proyecto es el encargado de revisar los cambios menores de la rama de desarrollo, así como de autorizar y realizar el paso a una versión estable, usando la rama master.

En la práctica, esta gestión de roles se nos ha probado suficiente para las labores que hemos tenido que desempeñar de cara a la gestión del código. Si hubiéramos tenido algún problema, tal vez se podría añadir un nuevo rol, por ejemplo, un encargado de resolver los conflictos del código, aunque la manera de resolver los que hemos tenido ha sido principalmente hablando entre los desarrolladores implicados o, simplemente, guardando los cambios de ambas versiones.

## Gestión de parches

En general, se debe trabajar siempre con la última versión en desarrollo. Para evitar romper la rama de desarrollo en caso de resolución de conflictos, cada desarrollador debe traerse la última versión de dev y de la rama del incremento, y juntarlos en esta última. Esto debe realizarse siempre, antes de empezar a trabajar. De esta manera, cualquier conflicto se resolverá siempre en la rama auxiliar, dejando la de desarrollo lo más estable posible (sin serlo totalmente) para que otros incrementos puedan partir de ella.

En definitiva, cada vez que haya un cambio en la rama de desarrollo, se debe aplicar un parche en la rama en que se esté trabajando.

### Ejemplo

Se va a continuar a trabajar en el incremento de automatización de pruebas. El desarrollador, antes de comenzar a escribir nuevo código, debe traer la última versión de la rama tests y la rama dev. Por último, debe juntar ambas en la rama tests para asegurarse de que el incremento no será incompatible con lo desarrollado. En caso de conflicto, debe resolverse en la rama tests, y pasarse a dev cuando el incremento esté completo.

## Aprobación de cambios

La aprobación de los cambios en el código se realiza, a nivel general, en dos fases: a nivel de aceptación del incremento y a nivel de estabilidad.

* El proceso definido anteriormente para la gestión del código especifica que los cambios de cierta magnitud deben ser desarrollados en una rama separada a partir de la última versión de desarrollo. Los desarrolladores asignados a un incremento deben validar de forma cruzada los cambios implementados, siendo el responsable del incremento el encargado de pasarlo a la versión en desarrollo. Una aplicación práctica de esto se lleva a cabo en el apartado anterior.
* Cuando se va a llevar a cabo una liberación del entregable, el jefe de desarrollo debe comprobar las incidencias que se van a cerrar en esta nueva versión. Los incrementos que no provengan de una rama subida al repositorio remoto deben ser verificados para garantizar la estabilidad del nuevo entregable.

En la práctica, la aprobación de los cambios se hacía de manera informal, en el sentido de que se pedía a los otros miembros del desarrollo que comprobaran el incremento, y estos dieran su aprobación. Esta aprobación venía frecuentemente a través de mensajería instantánea, por lo que apenas queda constancia de los mismos, y siendo poco transparente de cara a la persona que creó la incidencia (aunque se resuelve parcialmente mediante la vinculación de incidencias y commits, explicada en el apartado de gestión de incidencias).

Por eso, pensamos que podría ser más eficiente si la aprobación quedara patente en la propia incidencia. De alguna manera, conseguimos paliar esto permitiendo que el responsable pudiera delegar el paso a dev a uno de los desarrolladores, dejando constancia de que al menos uno si habría realizado la verificación pertinente.

## Políticas de nombre y estilo

Hemos llevado a cabo una política de nombres siguiendo el estilo *lowerCamelCase,* acordado entre todos los miembros del equipo. Este estilo es utilizado para nombrar los ficheros (recuentoController.js, encuestaGrafica.html), así como para las funciones y variables. Creemos que es el estilo más frecuente y utilizado, además de ser uno al que estamos muy acostumbrados.

Para los nombres de atributos y clases, nos hemos puesto de acuerdo con los subsistemas con los que nos relacionamos (Recuento y Modificación, Creación y Administración de Votaciones), y hemos adoptado la nomenclatura que ellos usan en sus APIs, facilitando así la integración a todos los niveles (encuesta, pregunta, texto\_opcion, id\_pregunta). Por tanto, no siempre tienen que ser consistentes con nuestro estilo.

Es recomendable que, para próximos cursos, haya un consenso común en el nombre y atributos de las entidades comunes a todos, pues mucho facilita el tratamiento de los datos recibidos, especialmente si se tienen que combinar datos de dos fuentes distintas.

Utilizamos el vocabulario (encuesta, votación, votos) del dominio del sistema en castellano, mientras que empleamos el inglés para los términos propios de la construcción del sistema (controller, tests, router, server). Por tanto, no es raro encontrar en el código variables con combinaciones de términos en distintos idiomas. Aunque pueda parecer extraño, permite al desarrollador familiarizado con la terminología, comprender rápidamente el significado de ciertas variables o funciones.

Es necesario comentar el código que no quede suficientemente claro. Cualquier miembro del equipo puede exigir a otro que realice los comentarios pertinentes, si no puede entender el código. Por supuesto, se recomienda que sea el propio desarrollador el que tome la iniciativa y mantenga su código claro e indentado.

# Gestión de la construcción e integración continua

Para llevar a cabo la gestión de la construcción y la gestión de la integración utilizamos la idea de la integración continua, nacida de Martin Fowler. Todo lo que cada uno va desarrollando y lo que el conjunto desarrolla se sube a un servidor y se compila automáticamente. Dentro de nuestra estructura de proyecto, esta parte se encuentra en la rama 'dev'. Es aquí donde el servidor de integración continua, una vez que detecta un commit nuevo en GitHub en la rama 'dev', coge todo el código y se lo lleva al servidor, donde se compila automáticamente.

La idea es tener un sistema de despliegue e integración continua durante el desarrollo de los proyectos con el fin de facilitar tanto el desarrollo como la integración de los subsistemas. Para ello se ha pensado que dicha integración constará de 3 partes usando la tecnología Jenkins dentro de la estructura propia de nuestro proyecto:

1. Fase make. En esta fase se descarga el código tras una modificación y se prepara para ser lanzado. En ocasiones podrían ejecutarse test para comprobar su integridad antes del despliegue.

a) En nuestro caso, no se hace nada, ya los test los ejecutamos en otra máquina independiente y con otra tecnología: Travis, que se explicará más adelante.

b) Coge, como dijimos anteriormente, el código de la rama 'dev' de Github. Al ser Jenkins la tecnología que utiliza, esto no deja de ser una tarea.

c) Este se configura en un archivo de script bash que ejecuta todo lo afirmado anteriormente y que se recoge más abajo.

d) Para poder verlo, se puede acceder a la página del servidor de integración continua de todos los proyectos de AgoraUS: https://jenkins.egc.duckdns.org/

El archivo consiste en estas cuatro instrucciones que eliminan el anterior contenedor y lo vuelve a crear:

#!/bin/bash

rm -rf $JENKINS\_HOME/builds/$JOB\_NAME/

mkdir -p $JENKINS\_HOME/builds/$JOB\_NAME/

cp -r \* $JENKINS\_HOME/builds/$JOB\_NAME/

ls -lah $JENKINS\_HOME/builds/$JOB\_NAME/

2. Fase beta. Esta fase se ejecuta automáticamente tras la finalización de la fase make. Se elimina la aplicación ya desplegada y se lanza la compilada en la fase make.

a) Aquí es donde tiene lugar la fase de construcción e integración. En nuestro proyecto, gracias a la enorme sencillez de NodeJS, esta fase consiste en dos instrucciones: npm install y npm start.

b) Se compila dentro del mismo servidor, pero en un contenedor Docker, por su simplicidad, ya que este no deja de ser un script sobre un contendor base. Esto te quita toda la complicación que pueda tener instalar los componentes en el servidor a mano. El nuestro es uno de NodeJS que se puede encontrar aquí en el foro de Docker.

c) El build, en nuestro caso, es una aplicación web que, al estar en un servidor en la web, se puede acceder desde internet. El link es el siguiente: https://beta.frontend.agoraus1.egc.duckdns.org/

Este archivo es más complejo, y por ello está comentado:

#!/bin/bash

ENV\_NAME="AgoraUS-G10-Frontend"

URL\_VIRTUAL\_HOST="beta.frontend.agoraus1.egc.duckdns.org"

BRANCH="beta"

PROJECT\_JENKINS\_NAME="AgoraUS-G10-Frontend\_make"

PATH\_ROOT="/var/jenkins\_home"

PATH\_ROOT\_HOST="/home/egcuser/jenkins\_home"

echo "Eliminando contenedores antiguos"

ContainerId2=`docker ps -qa --filter "name=$ENV\_NAME-$BRANCH-nodejs"`

if [ -n "$ContainerId2" ]

then

echo "Stopping and removing existing $ENV\_NAME-$BRANCH-nodejs container"

docker stop $ContainerId2

docker rm -v $ContainerId2

fi

echo "Preparando archivos para despliegue"

rm -r "$PATH\_ROOT/deploys/$ENV\_NAME/$BRANCH/"

mkdir -p "$PATH\_ROOT/deploys/$ENV\_NAME/$BRANCH/"

# NODEJS FOLDER

cp -r $PATH\_ROOT/builds/$PROJECT\_JENKINS\_NAME/\*

$PATH\_ROOT/deploys/$ENV\_NAME/$BRANCH/

echo "Desplegando contenedores para $ENV\_NAME"

docker run -d --name $ENV\_NAME-$BRANCH-nodejs \

-v "$PATH\_ROOT\_HOST/deploys/$ENV\_NAME/$BRANCH/":/myapp \

-w /myapp \

--add-host recuento.agoraus1.egc.duckdns.org:192.168.20.84 \

--add-host beta.recuento.agoraus1.egc.duckdns.org:192.168.20.84 \

--restart=always \

-e VIRTUAL\_HOST="$URL\_VIRTUAL\_HOST" \

-e VIRTUAL\_PORT=8080 \

-e "LETSENCRYPT\_HOST=$URL\_VIRTUAL\_HOST" \

-e "LETSENCRYPT\_EMAIL=annonymous@alum.us.es" \

--expose=8080 \

anapsix/nodejs

echo "Aplicación desplegada en https://$URL\_VIRTUAL\_HOST"

3. Fase stable. Esta fase se ejecuta manualmente por la administración del Jenkins. Se diferencia de la fase beta en la estabilidad, algo necesario para la interacción por parte de los otros subsistemas con él. El código ejecutado en esta fase debe ser el mismo que el de la fase beta para corroborar su estabilidad antes de ejecutar este despliegue.

a) Aquí es donde tiene lugar la fase de liberación y despliegue, que es explicará más tarde.

El fichero es casi idéntico al anterior, solo que este coge el código de beta, y no de make, como el anterior, y que este lo guarda en una carpeta llamada builds:

#!/bin/bash

ENV\_NAME="AgoraUS-G10-Frontend"

URL\_VIRTUAL\_HOST="frontend.agoraus1.egc.duckdns.org"

BRANCH="stable"

PATH\_ROOT="/var/jenkins\_home"

PATH\_ROOT\_HOST="/home/egcuser/jenkins\_home"

echo "Eliminando contenedores antiguos"

ContainerId2=`docker ps -qa --filter "name=$ENV\_NAME-$BRANCH-nodejs"`

if [ -n "$ContainerId2" ]

then

echo "Stopping and removing existing $ENV\_NAME-$BRANCH-nodejs container"

docker stop $ContainerId2

docker rm -v $ContainerId2

fi

echo "Preparando archivos para despliegue"

rm -r "$PATH\_ROOT/deploys/$ENV\_NAME/$BRANCH/"

mkdir -p "$PATH\_ROOT/deploys/$ENV\_NAME/$BRANCH/"

cp -r $PATH\_ROOT/deploys/$ENV\_NAME/beta/\*

$PATH\_ROOT/deploys/$ENV\_NAME/$BRANCH/

echo "Desplegando contenedores para $ENV\_NAME"

docker run -d --name $ENV\_NAME-$BRANCH-nodejs \

-v "$PATH\_ROOT\_HOST/deploys/$ENV\_NAME/$BRANCH/":/myapp \

-w /myapp \

--add-host recuento.agoraus1.egc.duckdns.org:192.168.20.84 \

--add-host beta.recuento.agoraus1.egc.duckdns.org:192.168.20.84 \

--restart=always \

-e VIRTUAL\_HOST="$URL\_VIRTUAL\_HOST" \

-e VIRTUAL\_PORT=8080 \

-e "LETSENCRYPT\_HOST=$URL\_VIRTUAL\_HOST" \

-e "LETSENCRYPT\_EMAIL=annonymous@alum.us.es" \

--expose=8080 \

anapsix/nodejs

echo "Aplicación desplegada en https://$URL\_VIRTUAL\_HOST"

## Docker

Aunque ya hemos explicado qué es Docker, creo que es necesario recalcar más algunos conceptos. Para la configurar Jenkins, en concreto la construcción y el despliegue, nos hace falta utilizar un contenedor de código con la configuración de nuestro proyecto: NodeJS. Para eso, se ha utilizado la tecnología Docker, que permite, a través de comandos bash, construir el sistema con una configuración inicial dada. Nosotros hemos utilizado una imagen que se puede encontrar en el foro de la empresa. La imagen elegida está en <https://hub.docker.com/r/anapsix/nodejs/> ya que incluye todo lo que nos hace falta: NodeJS y sus comandos iniciales: 'npm install', que instala todos los componentes que se encuentren en el package.json y 'npm start', que ejecuta el servidor, en nuestro caso, el archivo server.js. Los comandos para ejecutar en local el código y probar el contenedor son los siguientes:

git clone <https://github.com/AgoraUS-G1-1617/Frontend.git>

cd Frontend

git pull --all

sudo apt-get install docker.io

sudo docker pull anapsix/nodejs

sudo docker run -ti --expose=8080 -p 8081:8080 -v /home/usuario/Frontend/:/app/ -w /app anapsix/nodejs

Si funciona correctamente en localhost:8080, se puede ya configurar en el servidor, por medio de un archivo de script.

## Travis

Las pruebas se realizarán con los framework Jasmine y Karma, que testean los controladores de AngularJS. Para la automatización de las mismas se utilizará Travis, dado su fácil integración con Karma. Igual que anteriormente, para configurarlo, es necesario solo un fichero, llamado travis.yml:

language: node\_js

node\_js:

- "6.9.1"

script: node\_modules/karma/bin/karma start karma.conf.js --single-run

before\_install:

- export CHROME\_BIN=chromium-browser

- export DISPLAY=:99.0

- sh -e /etc/init.d/xvfb start

before\_script:

- npm update

cache:

directories:

- node\_modules

En este, es donde se indican algunas instrucciones, de las cuales la más importante es el ‘karma start’, la necesaria para que las pruebas, configuradas en otro archivo, se ejecuten automáticamente. Al igual que en Jenkins, estas pruebas se realizan por cada commit en la rama 'dev'. Estas ejecuciones están visibles en: <https://travis-ci.org/AgoraUS-G1-1617/Frontend>.

# Gestión del cambio, incidencias y depuración

## Mecanismos de depuración

El primer paso para la gestión de la depuración es reportar el bug encontrado. Para ello usaremos el gestor de incidencias que ofrece el propio GitHub, tanto para las encontradas internamente como para registrar las incidencias encontradas por compañeros de otros subsistemas.

Cuando un miembro desarrollador esté implicado en una incidencia, debe poder recibir las actualizaciones de la misma en su correo electrónico, para que esté al corriente de su avance y de la interacción con el resto de participantes.

## Procesos de gestión

### Incidencia interna

Si un miembro del equipo encuentra un error en la aplicación, debe crear una nueva incidencia en GitHub indicando al menos la siguiente información:

* Una introducción general al contexto donde se produce el fallo.
* Pasos a ejecutar: será necesario que describa, con más o menos detalle, los pasos que producen el error en el sistema, así puedan otros miembros del equipo reproducirlo y confirmar su existencia.
* Resultado esperado de realizar los pasos anteriores, que se considera como funcionamiento correcto.
* Resultado obtenido y que se interpreta como un fallo en el funcionamiento de la aplicación

No es necesario que se desglosen los apartados anteriores en párrafos separados, pero sí que debe quedar claro cada parte de la información. Por supuesto, cualquier dato adicional que sea añadido, como la versión del navegador, capturas de pantalla o cualquier otro intento del miembro para acotar la búsqueda del problema puede y debe ser añadido si procede.

Si, en cambio, quiere proponer una nueva funcionalidad, será necesario que realice una descripción lo más detallada posible de lo que desea que se implemente.

Una vez registrada la incidencia, el miembro que la creó debe comunicarlo al resto de desarrolladores para decidir si se aprueba, así como dar inicio al proceso de gestión de la misma mediante etiquetas y tipado, explicado más adelante.

### Ejemplo

Título: Automatización de pruebas con Karma

Como parte del proceso de automatización de la gestión del software, vamos a tratar de diseñar las pruebas con Jasmine y Karma. ¿Qué es eso? pues son frameworks de pruebas para Angular, que utilizan Chrome para ejecutarlas de forma automática en local.

En breve subiré una rama tests que implementa el fichero de configuración de Karma y dos tests de prueba con Jasmine sobre los controladores de Angular.

La tarea es la siguiente: integrar este framework de pruebas con nuestro automatizador de la construcción y despliegue (¿Jenkins? ¿Docker?)

O, en su defecto, encontrar un framework que, USANDO JASMINE, pueda automatizar la ejecución de las pruebas al desplegar.

Aquí os dejo más información, espero que sea un tutorial que os sirva.  
<https://karma-runner.github.io/0.8/plus/Jenkins-CI.html>

### Incidencia externa

Cuando un participante externo deja una incidencia, el proceso difiere ligeramente. En primer lugar, la comunicación con el mismo se vuelve mucho más relevante. Por tanto, dejamos en segundo plano la mensajería instantánea para concentrarnos en la comunicación a través de la herramienta de gestión de incidencias.

Lo primero que debemos hacer es identificar el problema. Para ello, uno o varios de los desarrolladores deberán recopilar la información necesaria aportada por el creador de la incidencia.

Una vez obtenida la información, los desarrolladores (ahora implicados en la incidencia), procederán a reproducir el error detectado. Si esto no fuera así, se repetiría el proceso de recopilación de información. Si no se encuentra el bug o si se es incapaz de reproducirlo, se puede debatir sobre invalidar la incidencia o asignar a otro desarrollador. Todo ello debe etiquetarse correctamente.

Una vez que se ha identificado el fallo del sistema, se puede proceder al etiquetado de prioridades, a asignar a un miembro del equipo y a comenzar a resolver la incidencia. Este proceso es exactamente igual al de una incidencia interna, y debe consensuarse en el grupo.

Cuando se resuelve la incidencia, se debe referenciar la corrección a la misma y añadir la etiqueta “fixed”. Se debe pedir explícitamente al participante que verifique la correcta resolución de la incidencia. En caso negativo, se debe volver a marcar como “started” y repetir el proceso desde aquí.

El tiempo máximo que tiene el participante para verificar la incidencia es hasta la próxima liberación de un entregable. Aquí, el jefe del proyecto debe realizar su propia verificación, y su aprobación se considerará como válida, cerrando cualquier incidencia que tengan un commit que lo resuelva.

Si, en cambio, la incidencia supone un incremento en la funcionalidad, el proceso es similar al de una incidencia interna. Esto es, el equipo debate el impacto y el alcance del incremento y se notifica al creador de la incidencia su aprobación o rechazo (con justificación). A partir de entonces, se procede al etiquetado y a la resolución de la incidencia, según lo definido en el apartado sobre estados, tipos y prioridades y en el proceso de gestión de código.

### Ejemplo[[3]](#footnote-3)

Título: Visualización incorrecta pie de página

Buenos días chicos,  
llevo bastante tiempo viendo que el pie de página no se muestra correctamente. En el único navegador que parece que va bien es en chrome/chromium.

Os dejo capturas de pantalla y más detalles:

[](https://cloud.githubusercontent.com/assets/12049827/21423939/0338c4f0-c840-11e6-9f98-144df705bc76.png)

Ilustración 4

Navegadores:

Chrome/Chromium OK

Vivaldi Error

FIrefox Error

Mi sistema operativo es Linux

**Respuesta:**

Buenas, he estado tratando de reproducir el error que has comentado para poder solucionar el problema viendo si cambiando cosas consigo que se solucione. Sin embargo, he instalado adblock tanto en Firefox como en Vivaldi, que son los casos que mencionas, en Linux y, como puedes ver, no me aparece el error, ya que tanto las fechas como el texto de los botones aparece sin problemas:

[](https://cloud.githubusercontent.com/assets/22834907/21425784/6cd58550-c84b-11e6-96ae-61ca13f281d9.png)

**Respuesta:**

Puede que sea por el tema de las listas, en tu navegador veo que no bloquea nada, en el mío me dice que bloquea 3 elementos.

Esta es mi configuración de AdBlockPlus:  
[](https://cloud.githubusercontent.com/assets/12049827/21428618/384ce05c-c85b-11e6-8085-bbd6b2b9ff54.png)

y por las pruebas que realicé la lista que produce el fallo es "Fanboy's Social Blocking List"

¿Quizás lo hayáis puesto como algo social en lugar de un botón normal y eso causa el error?

Si hay incidencias externas con más de una semana sin resolver o sin actividad, el jefe del proyecto debe asignar a un desarrollador para que la resuelva, dando preferencia a las prioritarias.

Cuando una incidencia está resuelta, se debe indicar apropiadamente, tanto por escrito en el sistema de gestión de incidencias como a través del cambio introducido. Es obligatorio empezar los commits con una referencia a la incidencia. Teniendo en cuenta si es una parte del incremento o la culminación de este, un commit debe comenzar de la siguiente manera:

* Para cambios relacionados: [issue] #nº de incidencia: título del commit. El título del commit debe ser no superior a 80 caracteres, y debe describir el cambio lógico introducido.
* Para incrementos completos: fixed #nº de incidencia: título del commit. Cuando el cambio es pasado a la rama estable, esta nomenclatura permite cerrar la incidencia automáticamente. De este modo, aunque se haya etiquetado como “fixed”, existe un periodo en el cual se puede verificar por parte de cualquier participante que, efectivamente, el cambio introducido resuelve la incidencia.

### Auto asignación

Hay veces en las que un miembro del equipo detecta una incidencia y puede ponerse a solucionarla rápidamente. Para evitar saturar el sistema con este tipo de incidencias, o de retrasar su resolución con el proceso anterior, hemos definido un proceso simplificado de auto asignación.

Cuando un desarrollador vaya a registrar y resolver una incidencia, se permite que la descripción de la misma sea más escueta, siempre que quede claro cuál es el problema y qué debe arreglarse.

A continuación, puede proceder a etiquetar la incidencia según lo definido en los apartados siguientes, con la salvedad de que puede escoger la prioridad del mismo de forma individual (de forma general será alta prioridad, aunque puede variar). Una vez asignada la incidencia a sí mismo, puede empezar a trabajar en ella.

### Ejemplo

Un miembro del equipo ha comprobado que la página web no se adapta bien a dispositivos móviles. Dado que tiene conocimientos previos en el área de la responsividad, cree que puede hacerse cargo rápidamente del incremento. Por tanto, lo primero que hace es registrar la incidencia[[4]](#footnote-4).

A continuación, dado que va a comenzar a trabajar inmediatamente en la solución, realiza el etiquetado de la incidencia y se la asigna a sí mismo. A partir de ahí, puede desarrollar el incremento según lo especificado en el proceso de gestión de código.

El commit que resuelve la incidencia debe hacer referencia a la misma con “Fixed #nº de incidencia”. Cuando se realice la entrega de la nueva versión, esta se cerrará automáticamente.

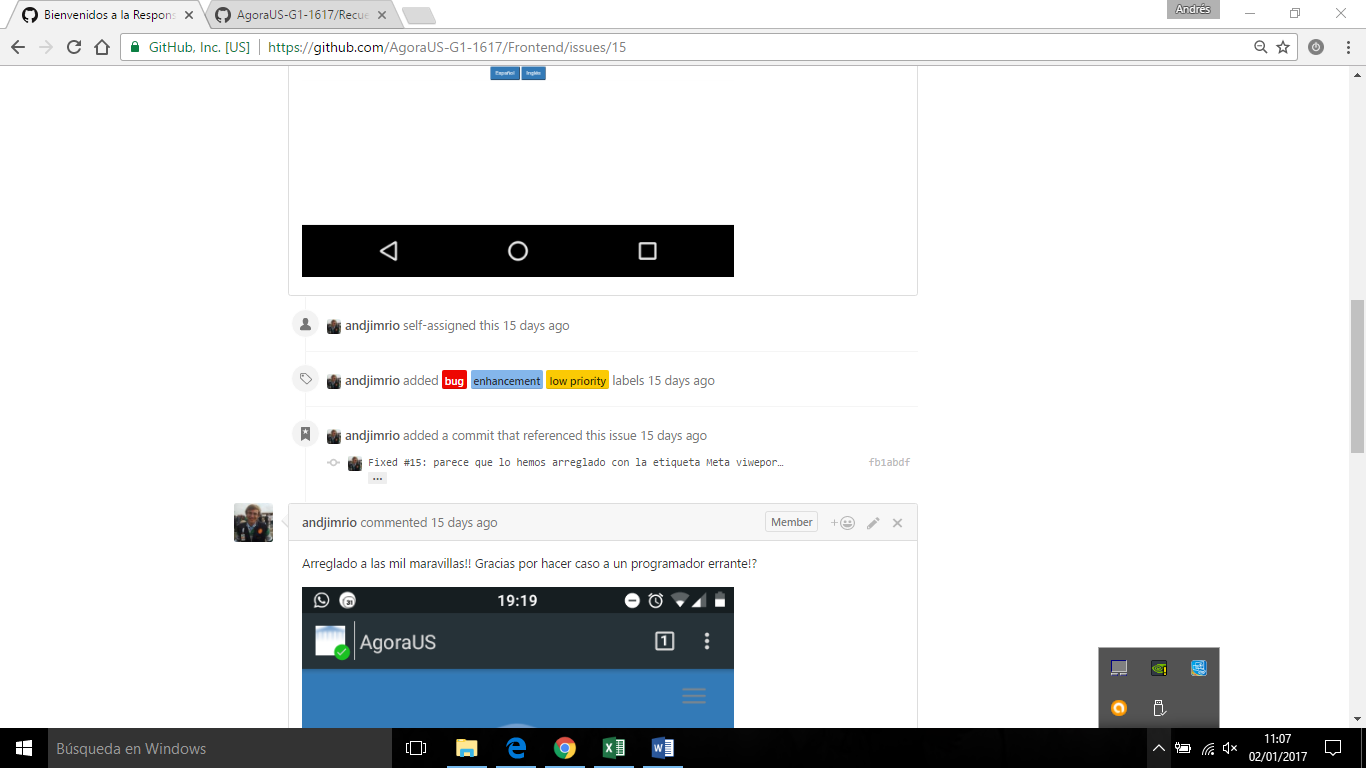


Ilustración 5

Antes de la entrega, un compañero verifica que la incidencia está correctamente resuelta y la etiqueta como “fixed”.



Ilustración 6

## Roles

En general, existen tres tipos de roles, muy relacionados con los roles definidos para la gestión de código, dado que los cambios en uno deben reflejarse en el otro, y viceversa.

* Encargado de las incidencias: no tiene por qué ser el jefe del proyecto. Se trata de un miembro del equipo que se encarga de revisar periódicamente el estado de las incidencias, especialmente de dar soporte a las nuevas que se reciban desde el exterior. El encargado de las incidencias debe comunicar las actualizaciones al resto del equipo para realizar las labores de identificación y etiquetado, asignación o rechazo de la incidencia.

Esto no exime al resto del equipo de conocer los mecanismos de gestión y de tramitación de incidencias, pero el encargado prestará una mayor atención al registro de las mismas. El objetivo de este rol, en definitiva, es mantener al equipo lo más informado posible de cualquier actualización en las incidencias, para poner en marcha el proceso de gestión.

Para la comunicación se usará, en la mayoría de los casos, la mensajería instantánea.

* Implicado en una incidencia: cuando una incidencia externa es creada, los miembros involucrados en ella deben informar al creador, tratar de recopilar toda la información posible para tomar una decisión en cuanto a su aprobación, prioridad y asignación.

Para este tipo de incidencias, la comunicación debe realizarse a través de la herramienta de gestión de incidencias. En cambio, las incidencias internas pueden debatirse adicionalmente mediante mensajería instantánea.

* Responsable de una incidencia: tiene el mismo papel que el responsable de un incremento, pero orientado a la gestión de incidencias. Debe asegurarse de que los commits relacionados referencian la incidencia en concreto.

## Estados

Una vez creada una incidencia, se ha de definir el proceso por el que pasa a ser llevada a cabo. Clasificaremos las incidencias en varios tipos para identificar fácilmente su contenido.

* bug: para informar de un error en el funcionamiento del código.
* enhancement: para proponer una nueva funcionalidad o ampliar una existente.
* help wanted y question: para dudas sobre el funcionamiento en general, sea de organización, de la arquitectura del sistema, de la configuración del entorno de desarrollo, etc.
* example: para ejemplos de cómo resolver incidencias.

En general, las dos primeras son las que más hemos utilizado, por abarcar la mayoría de situaciones de incidencias. El resto fueron necesarias al principio del desarrollo, para poder configurar el entorno o facilitar el adaptarse al proceso definido.

### Prioridad y estado de incidencias[[5]](#footnote-5)

Hay tres etiquetas de alta, media y baja prioridad. La prioridad implica la urgencia con que debe resolverse una incidencia, y es, en general, decidida en consenso por los miembros del grupo. La prioridad se puede cambiar en cualquier momento si así se considera necesario, por el mismo método que se decidió la prioridad original.

El entregable al que está destinado también influye en la prioridad de la incidencia. Debe consensuarse el entregable en el que estará disponible y especificarlo en la incidencia.

A priori, existen dos etiquetas para indicar el avance de una incidencia:

* started: indica que se ha tenido en cuenta la incidencia y se está trabajando para resolverla.
* fixed: indica que la incidencia ha sido resuelta y verificada por varios miembros del equipo asignado.

También existen tres etiquetas que indican una cancelación de la incidencia, por diferentes motivos:

* duplicate: la incidencia ya ha sido notificada por otro usuario o miembro.
* invalid: la incidencia no es tal, sea porque el comportamiento erróneo detectado es realmente el esperado, o porque la funcionalidad ya está siendo implementada o corregida.
* wontfix: el alcance del cambio es inviable y no está en los planes abarcarlo

La asignación de estas etiquetas debe ir acompañadas de una justificación, que la incidencia no es válida y no se va a resolver. Se procederá a cerrar la incidencia tras un periodo máximo de dos días, en los que se pueden hacer las intervenciones necesarias.

Se espera confirmación del usuario que indicó la incidencia para que compruebe que realmente el problema está resuelto. Si es así, se puede cerrar la incidencia. En caso contrario, se debe volver a marcar la etiqueta started y repetir el proceso.

No se puede cerrar una incidencia que esté en estado started, sino que debe pasar a fixed, duplicate, invalid o wontfix (siempre con explicación previa) antes de poder cerrarse.

El cierre de una incidencia resuelta con éxito se dará durante la próxima liberación o entrega. Una vez el cambio con la corrección/mejora ha sido subido al repositorio remoto, y está relacionada con la incidencia en concreto, el jefe del proyecto procederá a la verificación del mismo, si no se hizo previamente. Una vez los cambios se pasan a la rama master, las incidencias se cierran automáticamente. Los usuarios tienen hasta ese entonces para verificar el cambio realizado.

### Ejemplo

Aquí veremos un ejemplo de gestión de incidencia[[6]](#footnote-6) interno, cómo se han ido gestionando las etiquetas durante el desarrollo del incremento. En primer lugar, el equipo discutió y aprobó la incidencia, y es etiquetada y definida para un entregable.

Más tarde, al ver que no se estaba cumpliendo el objetivo, el equipo decidió bajar la prioridad dado que el valor que aportaba era mucho menor que cuando se creó la incidencia. También se discutió la posibilidad de rechazarla, pero finalmente se optó por aplazarla.



Ilustración 7

Una vez se resolvieron las incidencias prioritarias, se decidió asignar a un miembro del equipo para que trabajara en ella. Se produce la asignación y se añade la etiqueta “started”

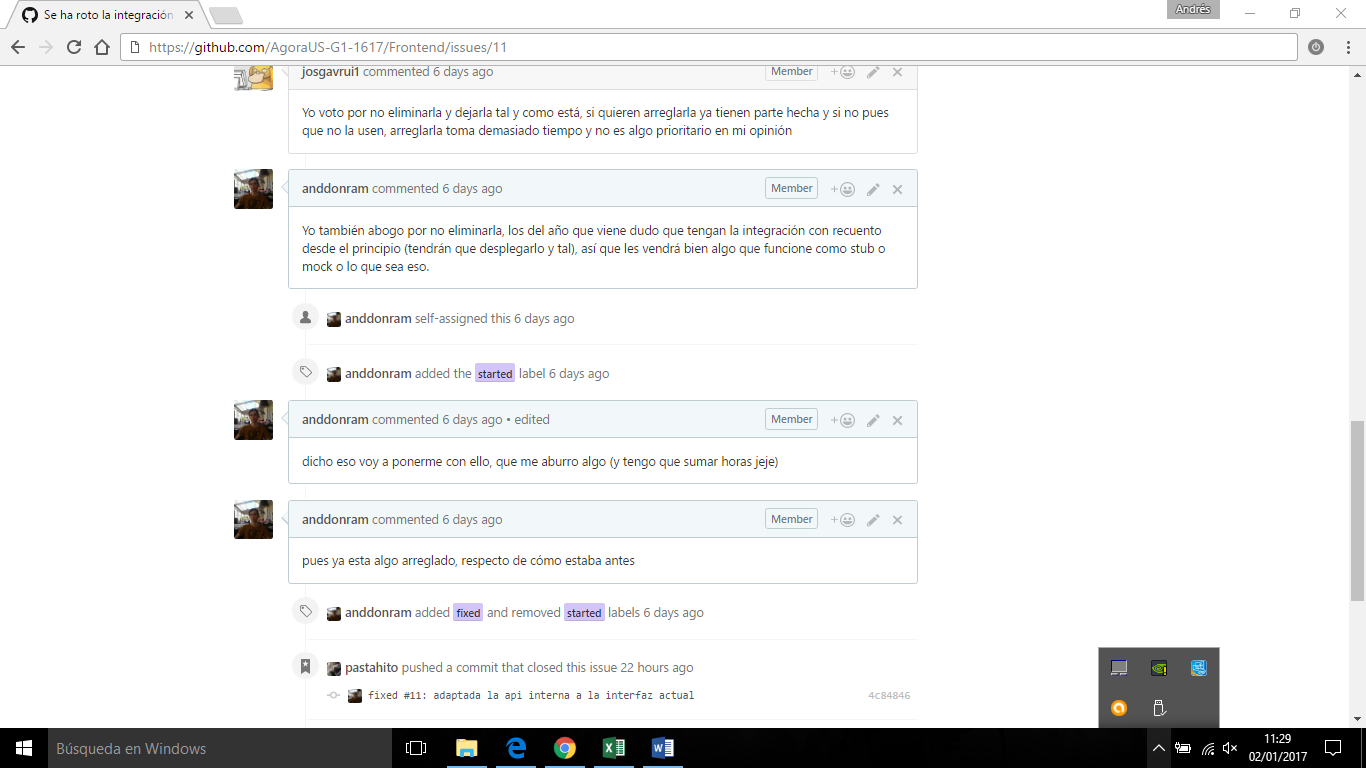


Ilustración 8

Cuando se completó el incremento, el responsable subió el commit con el encabezado “fixed #11”. También cambia la etiqueta “started” por “fixed”. Dado que no hubo una rama en el equipo remoto y no se verificó de forma cruzada el incremento, según nuestro proceso de gestión de código, esta referencia no se muestra hasta que el jefe del proyecto realiza la verificación del incremento en el nuevo entregable. Una vez hecho esto, se muestra el commit que resuelve la incidencia. Automáticamente, GitHub sabe que, con el paso a master, la incidencia debe cerrarse.

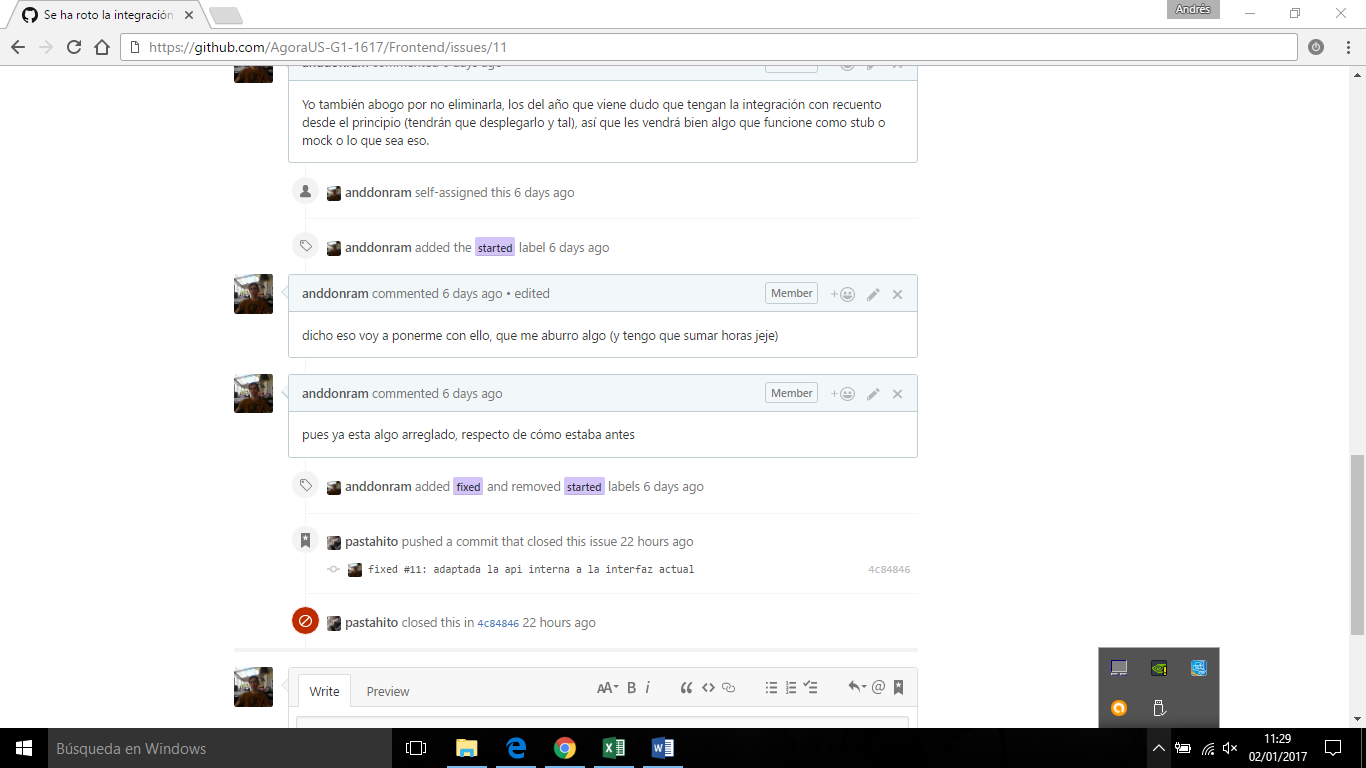


Ilustración 9

## Políticas para descartar, fomentar o retardar un cambio

La utilización de las etiquetas descritas anteriormente provoca inherentemente un cambio en las prioridades del proyecto. Si una incidencia tiene una entrega más temprana, su prioridad será mayor. Lo mismo ocurre con cambios que contengan un alto valor para el software a desarrollar.

En cambio, las etiquetas que rechazan el cambio propuesto deben ir justificadas apropiadamente, y se deja un periodo menor para las argumentaciones y rectificaciones.

Por otro lado, si vemos que un cambio que consideramos prioritario se está prolongando en el tiempo, y su valor real es menor de lo que se percibió al principio, o en relación al resto de incidencias, se puede bajar la prioridad. Esto indicará que los desarrolladores pueden dedicarse a completar otras incidencias comenzadas, si tienen mayor prioridad.

Por ejemplo, en el ejemplo anterior, al comenzar la integración con el subsistema de Recuento, necesitamos hacer cambios que rompieron la funcionalidad de nuestra API interna, que simulaba ese subsistema. Abrimos una incidencia para ese problema5, y consideramos que sería de una prioridad media, y un objetivo a más o menos largo plazo. Sin embargo, dado la complejidad de rehacer ese módulo que fuimos observando, tuvimos que bajar la prioridad frente a otros cambios que aportaban más valor, ya que la interacción con la verdadera API estaba hecha.

Por tanto, la incidencia se aplazó hasta que el resto de incidencias fueron resueltas.

# Gestión de liberaciones y despliegue

La gestión de liberaciones se hará por versiones del software. Cada versión se etiquetará como *vX.Y.Z*, siendo X la versión mayor, Y la menor, y Z la revisión. Las versiones candidatas se marcarán con "c" (de *candidate*) tras la versión.

Solo cuando la funcionalidad de una versión candidata esté probada por al menos dos desarrolladores en "dev" se unirá con la rama "master", marcando esa versión como "r" (de *release*). Con esta metodología se ha llegado hasta la versión 3.0.1.

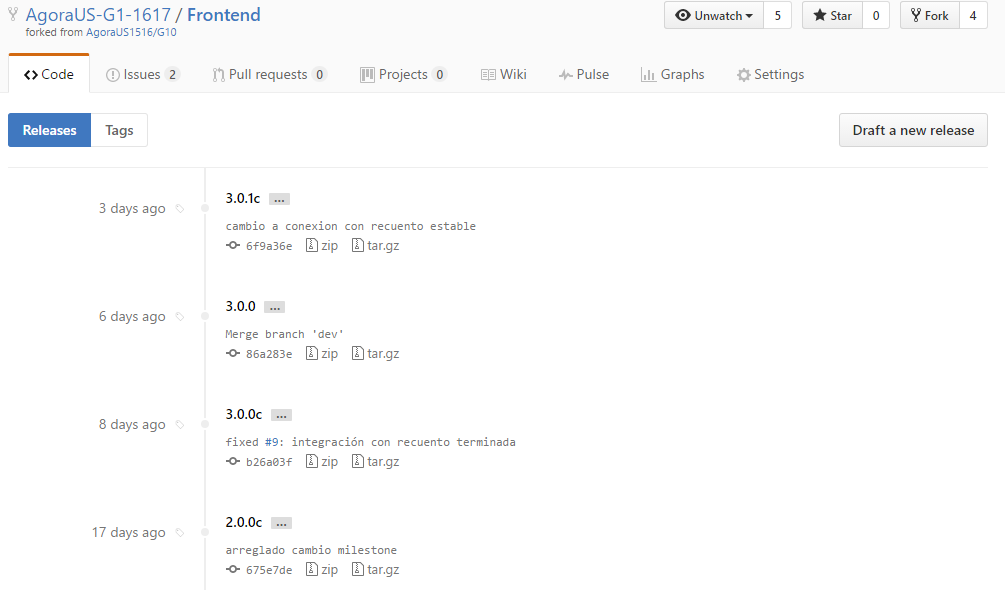


Ilustración 10

Serán estas versiones unidas a master las que se desplegarán en el entorno de producción. Esto es, propiamente, la gestión del despliegue.

La gestión del despliegue funciona de igual manera que la gestión construcción e integración explicado más arriba: máquina con Jenkins que coge el código de Github y lo compila. Solo que, para las liberaciones de las versiones, y el despliegue en general, lo hacemos en otra url.

Como se explicó antes, había una fase en Jenkins llamaba 'stable' que consistía en coger el código de la fase 'beta' y ejecutarlo en otro sitio. La url de fase estable es esta: <https://frontend.agoraus1.egc.duckdns.org/>

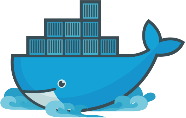
Esta metodología nos permite que probemos siempre los cambios durante el desarrollo del proyecto (fase 'beta'), pero que contemos, igualmente, con la aplicación corriendo para los clientes de manera segura y fiable (fase 'stable'). Además, una vez testeado en desarrollo, pasar a estable es muy sencillo: ejecutar la tarea en Jenkins y ya se encarga el servidor de hacer todo lo demás.

# Mapa de herramientas

## Lista de herramientas

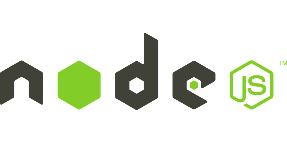
|  |  |
| --- | --- |
| C:\Users\pasta\AppData\Local\Microsoft\Windows\INetCacheContent.Word\node.png | **Node**  Entorno en tiempo de ejecución multiplataforma escrito en C++ y ECMAScript para una arquitectura basada en eventos. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como servidores web. |
| npm | **NPM**  Es gestor de paquetes de Node. Es el ecosistema más grande de librerías de código abierto en el mundo. |
| jenkins | **Jenkins**  Es un software de integración continua open source escrito en Java. Es un sistema corriendo en un servidor que es un contenedor de servlets. |
| travis | **Travis**  Software de integración continua escrito en Ruby. Brinda un servicio web para ejecutar pruebas en proyectos alojados en GitHub. |
| angular | **Angular JS**  Es un framework de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. |
| bootstrap | **Bootstrap**  Conjunto de herramientas HTML y CSS de código abierto que contienen tipografía, formularios, botones, cuadros, menús de navegación, entre otros, así como extensiones de JavaScript opcionales adicionales. |
| git | **Git**  Es un software de control de versiones distribuido de código libre diseñado para manejar desde pequeños hasta grandes proyectos con velocidad y eficiencia. |
| github | **GitHub**  Servicio de alojamiento de proyectos git escrito en Ruby on Rails, gratis para repositorios públicos. Ofrece navegador de código fuente, edición en línea, wikis y gestor de incidencias, entre otros servicios. |
| mongo | **MongoDB**  Es un sistema de base de datos NoSQL orientado a documentos. |
| karma | **Karma JS**  Herramienta de ejecución de tests para Angular que te permite ejecutar pruebas en múltiples navegadores. |
| jasmine | **Jasmine JS**  Framework de pruebas unitarias para código Javascript que no depende de navegadores ni del DOM. |
| C:\Users\pasta\AppData\Local\Microsoft\Windows\INetCacheContent.Word\docker.png | **Docker**  Proyecto de código abierto para automatizar el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción del SO. |
| telegram | **Telegram**  Servicio de mensajería instantánea por internet. |

## Mapa de herramientas



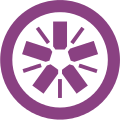
Cliente

Servidor



Pruebas

Gestores



# Conclusiones

* Establecer el proceso de gestión de código y de incidencias lo antes posible, y acostumbrarse a usarlo por parte de todos los miembros del equipo.
* Llevar a cabo un registro semanal del trabajo realizado, a fin de determinar la implicación de cada miembro en el proyecto, lo cual repercute en la evaluación del trabajo.
* Consensuar el formato de las APIs con el resto de subsistemas, especialmente con aquellos con los que hay dependencia directa. Especial atención a las clases, los atributos que se van a necesitar y sus nombres, para evitar muchos cambios a la hora de la integración. Tratar de acordar una nomenclatura, por ejemplo, para las Encuestas (Survey/Votación/Encuesta). Escoger un nombre y definir a qué se refiere, si va a haber varios tipos de encuesta, etc., y qué información se va a ofrecer a través de las APIs.

# Cambios propuestos

* Desarrollar pruebas para el servidor de NodeJS. Las pruebas desarrolladas hasta ahora solo cubren los controladores de la parte front-end, que usa la tecnología AngularJS. Sin embargo, el servidor que sirve de conexión con el resto de subsistemas no tiene un conjunto de pruebas.
* Integrar la funcionalidad de listado de votaciones con el subsistema Creación y Administración de votaciones. Dado que este año ese subsistema no ha estado listo para la integración durante los hitos correspondientes, la integración ha resultado imposible. Para paliar esto, hemos diseñado una API interna auxiliar que simula y recupera el listado de votaciones de una base en MongoDB. Al mismo tiempo, el subsistema de Recuento y Modificación de resultados ha realizado una API que permite crear votaciones y depositar un voto. Nosotros hemos terminado integrándonos con esa API, pero el cambio entre esa y la interna tan sencilla como cambiar una línea de código.

1. <https://git-scm.com/> [↑](#footnote-ref-1)
2. <https://github.com/> [↑](#footnote-ref-2)
3. Para ver detalladamente el proceso de recopilación de información, la reproducción del bug y la resolución, visite <https://github.com/AgoraUS-G1-1617/Frontend/issues/17> [↑](#footnote-ref-3)
4. <https://github.com/AgoraUS-G1-1617/Frontend/issues/15> [↑](#footnote-ref-4)
5. Para ver ejemplos de diferentes configuraciones y asignaciones, visite <https://github.com/AgoraUS-G1-1617/Frontend/issues?q=is%3Aissue+is%3Aclosed> [↑](#footnote-ref-5)
6. <https://github.com/AgoraUS-G1-1617/Frontend/issues/11> [↑](#footnote-ref-6)