Universidad Zaragoza

Trabajo Fin de Grado

ADIDAS GSD-MP: Portal de Métricas de Desarrollo de Software de ADIDAS GSD

ADIDAS Global Software Development Metric Portal

Autor

Héctor Anadón León

Director

Javier Pelayo Gil

Ponente

Francisco Javier López Pellicer

Grado en Ingeniería Informática

Septiembre 2016

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Héctor Anadón León,

con nº de DNI 77215279R en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Ingeniería Informática, (Título del Trabajo) ADIDAS GSD-MP: Portal de Métricas de Desarrollo de Software de ADIDAS GSD,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 21-06-2016

Héctor

Fdo: Héctor Anadón León

Resumen ejecutivo

Adidas tiene una cantidad de proyectos software y por tanto, de desarrolladores, superior a cien, tanto internos como externos. Se requiere saber en qué estado se encuentra cada proyecto y, sobre todo, controlar su evolución.

La calidad de los proyectos iniciales no es muy buena porque no se puso el esfuerzo necesario en su momento, sin embargo, con los cambios en la dirección en el departamento de IT en los últimos años, se está poniendo empeño en mejorar su calidad y su mantenibilidad principalmente. Será imprescindible monitorizar su progreso y que éste siempre vaya en ascenso.

Sobre los proyectos creados recientemente se exige una calidad excepcional. Conociendo los errores del pasado se intenta no volver a cometerlos, para ello es necesario controlar estos proyectos exhaustivamente para tener desde el principio un código legible, mantenible y testeado. El coste de mantener un código de calidad mientras está siendo desarrollado es inferior a mejorarlo una vez finalizado.

Así mismo, se dispone de distintas empresas externas que proveen desarrolladores que no trabajan en el mismo espacio geográfico. Adidas busca obtener los mejores y que estén cualificados para hacer el trabajo que les corresponde. Debido al gran número de personal es necesario monitorizar su rendimiento así como su evolución dentro de los proyectos.

Actualmente no existe una herramienta unificada de análisis que, en un vistazo, se pueda observar esta información tan crucial para que Adidas pueda controlar la calidad y mantener la gran cantidad de código que genera, siendo competitivos frente a otras empresas del mismo ámbito en el mundo IT.

Índice de contenidos

Índice de figuras

1 Introducción

1.1 Contexto

Adidas es una compañía multinacional fabricante de calzado, ropa deportiva y otros productos relacionados con el deporte. Su sede principal se encuentra en Herzogenaurach, Alemania. La compañía se divide internamente por países y a su vez, por departamentos como ventas, marketing, recursos humanos, etc. **Global IT** es el departamento de **Adidas** que da a la compañía soporte mundial relacionado con la informática. Éste está dividido en distintos grupos que aparecen en la Figura 1.

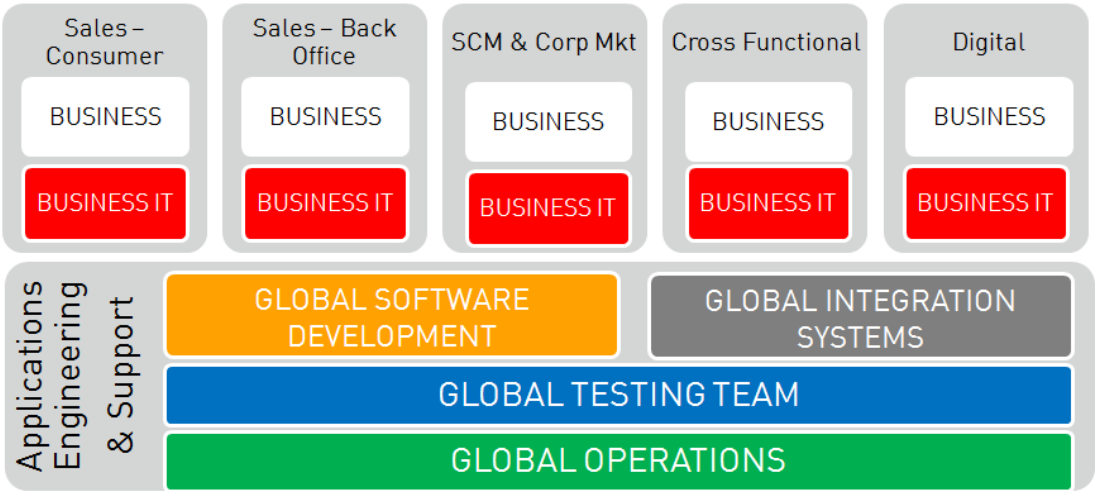


Figura 1: Grupos de GSD

A destacar de esta organización encontramos dos tipos de grupos, los grupos verticales que se dedican a proporcionar funcionalidad para un área determinada de negocio y los grupos horizontales que dan apoyo a todos los grupos.

El TFG se ha realizado dentro de **GSD** (Global Software Development), un grupo horizontal que da soporte a los demás, dedicado a desarrollar y mantener software de calidad, tanto aplicaciones internas de la compañía como aplicaciones *core* que son las consideradas causantes de un beneficio directo como la web de ventas.

Adidas tiene una gran cantidad de proyectos software y por tanto, de desarrolladores, superior a cien, tanto internos como externos. En **GSD** se captura una gran cantidad de datos sobre la calidad de este código que puede ser calificada como Big Data a través de herramientas como **Jira [1]** (control de tareas), **Sonar [2]** (análisis de código), y **Stash [3]** (almacén de repositorios git). Esta información es crucial para evaluar tanto a programadores internos como a empresas proveedoras de software externas. Actualmente no existe una herramienta unificada de análisis que, en un vistazo, se pueda ver el estado de un proyecto y su evolución con respecto al tiempo.

1.2 Objetivo del proyecto

El objetivo principal del proyecto consiste en el desarrollo de una herramienta que permita la evaluación visual de la calidad de código de los distintos proyectos software en los que Adidas participa. Actualmente existen herramientas para ver datos de calidad de código como Jira y Sonar, sin embargo, se busca una herramienta unificada, tanto de las distintas informaciones de evaluación de código, como de los distintos proyectos. Además, ésta deberá ser configurable dependiendo del tipo de proyecto que sea. No se evaluarán de la misma forma los proyectos antiguos cuya calidad de código no se ha podido controlar desde el inicio, como los proyectos recientes de los que se espera una calidad superior.

Se quiere observar el rendimiento de los distintos proyectos propietarios de Adidas para poder comparar la calidad de las distintas compañías externas contratadas, así como su equipo de programadores. Se espera ver que, gracias a la inclusión en el proyecto de GSD, tanto desarrollando como revisando código, se ve una sustancial mejora en el estado general del proyecto en número de *issues*, en velocidad de crecimiento, legibilidad y mantenibilidad del código.

Para conseguir dicho objetivo se realizará una aplicación Web pesada con Modelo-Vista-Controlador que accederá a estos datos en tiempo real y proporcionará herramientas visuales avanzadas de análisis. Esta aplicación será utilizada tanto por los jefes de proyecto como por los propios programadores para controlar el estado general de un proyecto, su evolución, su velocidad de desarrollo y sus principales debilidades. Desde una única herramienta será posible acceder a todos los proyectos y a su información de una forma gráfica e intuitiva.

En el apartado 2.1.2 se explica detalladamente las distintas métricas que se obtienen para su posterior visualización.

1.3 Marco tecnológico

En este capítulo se pretende ofrecer un resumen sobre las tecnologías que han sido utilizadas para realizar esta aplicación de cliente Web. Se hace especial hincapié en el apartado 1.3.4 que trata sobre las librerías utilizadas para mostrar los distintos gráficos y como éstas fueron elegidas y el funcionamiento interno del *framework* seleccionado (AngularJS) cuyo conocimiento es imprescindible para un diseño y un desarrollo eficiente y de calidad.

1.3.1 Lenguaje y framework

El lenguaje utilizado ha sido **JavaScript**, es un lenguaje de programación interpretado estandarizado en la especificación del lenguaje ECMAScript [4]. El lenguaje soporta estilos de programación basada en prototipos, imperativa y funcional. Es también un lenguaje débilmente tipado y dinámico. Su uso principalmente es client-side implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas. Tiene soporte en todos los navegadores importantes [5]. Apoyándose en JavaScript se ha utilizado el

framework AngularJS explicado más adelante. En este proyecto se ha utilizado ECMAScript5, debido a su amplio soporte.

En cuanto a tecnologías utilizadas, la más destacada es **AngularJS**. Este *framework* nos permite modificar el DOM (representación del código HTML) en tiempo real dando lugar al desarrollo de webs dinámicas. Da la capacidad de Modelo Vista Controlador, donde a cada vista se le asigna un "controlador", es decir, un fichero cargado con la lógica encargada de aplicar el dinamismo de la web gracias a la funcionalidad de *data binding* bidireccional. Permite sincronizar variables del DOM con variables dentro del controlador. Se concreta sobre su funcionamiento interno a bajo nivel en el anexo 9.

Algunas de las principales funcionalidades que nos da AngularJS:

- Al estar la lógica del DOM en un fichero controlador mejora la capacidad de testear el código (más información en el apartado 2.4).
- El desarrollo de la aplicación *frontend* es independiente del estado del *backend* ya que permite un desarrollo en paralelo gracias a la posibilidad de simular los datos que se recibirán AngularJS-Mock [6].
- El código es abierto y además está mantenido por Google.

1.3.2 Estilos

Gracias a **Bootstrap** [7] es posible crear una aplicación multiplataforma, tanto para móvil, tablet y ordenador, que además nos ofrece estilos básicos.

Este *framework* posee una gran comunidad que junto a Google ofrece soporte y gran cantidad de documentación. Ésta es la principal razón por la que se ha utilizado este en vez de **ReactJS**. Éste es un *framework* similar dirigido más concretamente a componentes, tiene un rendimiento ligeramente mejor que Angulares, ya que no refresca el DOM completo al detectar un cambio, sino sólo la parte que ha cambiado. Sin embargo, AngularJS es suficiente en cuanto al rendimiento para los requisitos recibidos.

Para estilar la página web se va a utilizar el preprocesador de código CSS llamado **Less**. Éste permite añadir funcionalidades sobre CSS, como la existencia de variables, funciones y otras técnicas que permiten hacer que el código CSS sea más legible, mantenible y extendible.

Less deriva de Sass, muy parecido a Less, sin embargo, se ha decidido elegir Less por su mayor sencillez ya que no se van a necesitar estilizaciones complejas, puesto que las librerías de gráficos se caracterizan por poder personalizarse libremente.

1.3.3 Alojamiento

Para alojar la aplicación se va a utilizar un servidor HTTP de Node.js [8]. Es un entorno multiplataforma de código abierto para el desarrollo de aplicaciones *backend*. Se ha utilizado esta herramienta por su sencillez para desplegar un servidor web, ya que no será necesaria la escalabilidad de la aplicación, al menos en un futuro cercano.

Se utilizará la librería: http-server[9] cuyo protocolo del servidor (Figura 2) es el siguiente:

1. El cliente web conecta con el servidor y acuerdan un protocolo de comunicación (como por ejemplo WebSockets, XMLHttpRequest…).
2. El cliente web (explorador) envía un evento al servidor de Node.js a través de JavaScript.
3. El servidor captura el evento.
4. El servidor responde al cliente con el fichero requerido.

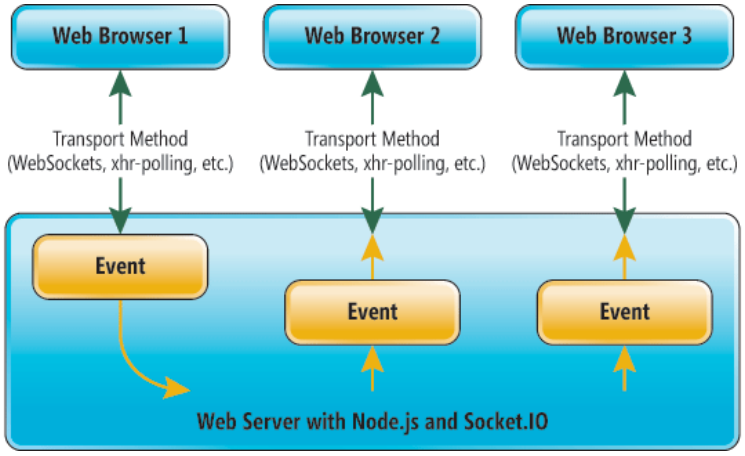


Figura 2: Comunicación entre Web Browser y Servidor de Node.js HTTP

1.3.4 Tecnología de gráficos

Para cumplir los objetivos del proyector se busca una librería JavaScript para gráficos estadísticos que permita mostrar gráficos personalizables de barras, de líneas, de sectores…

Deben ser responsivos en cuanto a la anchura de la pantalla y deben poder actualizarse en tiempo real (requisito que no viene desde el negocio sin embargo, como decisión personal de diseño, se decide añadir este requisito).

Tecnología de *render*: HTML 5 Canvas vs SVG

Existen dos posibles aproximaciones para generar los gráficos o *render*: el elemento *canvas* de HTML5 y el formato SVG.

El elemento **Canvas [10]** permite dibujar un mapa de bits sobre una superficie, no tiene capas y no se tiene conocimiento de lo que está dibujado dentro de éste, lo que implica gran dificultad para interactuar con el gráfico. Es decir, se podría comparar con una imagen plana. Sin embargo, el elemento *canvas* es muy potente en animaciones altamente activas y es capaz de dibujar una gran cantidad de elementos.

SVG [11] es un formato XML de imágenes vectoriales soportado por todos los navegadores modernos que permite crear gráficos vectoriales bidimensionales, poseen capas y existen en el DOM, lo que hace que sea fácil escuchar eventos de interacción con las distintas partes del gráfico. Permite conocer la posición relativa de los elementos y permite manipularlos usando JavaScript y CSS. Con SVG la generación de gráficos es un poco más lenta debido a su complejidad. El rendimiento empieza a ser malo cuando hay más de 1000 elementos en la misma pantalla, un caso de uso que no se espera.

Aunque ambas soluciones son soportadas en los principales exploradores modernos, para cumplir los objetivos se va a usar gráficos SVG porque son mucho más fáciles de controlar y manipular. Como no van a existir animaciones de gran nivel de complejidad y el número de objetos no va a llegar al límite de rendimiento, no habrá problemas de estas características.

Librerías JavaScript SVG

Una vez elegida la tecnología de *render* hay que elegir qué librería se puede utilizar en el lado del cliente. Hay diferentes alternativas:

- **D3:** es una librería exhaustiva que permite enlazar datos a elementos. Está basada en los estándar web actuales, HTML5 y CSS. Y lo más importante, es muy popular con una gran comunidad que da soporte. Tiene diferentes librerías derivadas que reutilizan gráficos y componentes de D3:
  - **NVD3:** Bastante personalizable y con una comunidad bastante grande.
  - **N3, C3**: Demasiado simple y poco personalizable.
- **Raphael:** Parecido a D3, fácil de usar, sin embargo es poco personalizable.
- **Google Graphics:** muy simple y con unos estilos muy fijados.
- **HighCharts, FusionCharts:** no son libres, a cambio se recibe apoyo.

Para el Proyecto se ha decidido utilizar D3 para los gráficos más complejos porque esta librería permite un grado muy alto de personalización. Además es posible introducirlo en una directiva de Angular para que pueda adaptarse a la anchura de la pantalla y sea posible actualizarla en tiempo real. La gran curva de dificultad que implica el dominio de esta librería sería su factor negativo.

Para gráficos más sencillos se utilizará NVD3 que son desarrollados con mucha más rapidez gracias a los gráficos predefinidos que provee esta librería. Existe librería con gran compatibilidad con respecto a Angular utilizando objetos JSON para su personalización.

1.3.5 Problema de Big Data

Desde la aplicación Web se consumirá un volumen considerable de datos. Estos estarán alojados en MongoDB con lo que se espera dar una solución al problema de Big Data. Es una base de datos orientada a documentos, no relacional, no SQL, multiplataforma, gratis y de código libre, además incorpora datos estadísticos sobre la eficiencia de la base de datos en tiempo real.

Big Data [12] se refiere al uso de tecnologías e iniciativas que incumbe datos de estructura diversa, que cambian constantemente durante el tiempo o grandes volúmenes de datos. Resumiendo, el volumen, la variedad y la velocidad de cambio de los datos son significativos.

En concreto, en este proyecto, el volumen de datos no es muy grande. Crece en orden de 1 KByte por métrica, por número de métricas por proyecto. Actualmente existen diez tipos de métricas dando soporte a veinte proyectos. Puede aumentar sin causar gran impacto en el tamaño.

Sin embargo, la variedad de datos es grande ya que para cada métrica, así como la configuración del proyecto, tienen una estructura de datos distinta. Además existe un caso de uso en el que la configuración de un proyecto se cambia, y se reindexa toda la información de todos los días del proyecto desde su fecha de inicio hasta la actual (más información en 2.2), por lo que hay momentos en que gran cantidad de datos varían.

1.4 Marco metodológico

1.4.1 Enfoque

Desde la empresa se decidió seguir un enfoque *agile* [13] frente al enfoque tradicional o desarrollo en cascada [14]. Para justificar esta elección antes de empezar este proyecto se ha hecho un estudio sobre los diferentes enfoques metodológicos. Para ello se han de tener en cuenta los siguientes aspectos:

- Los requisitos no están definidos totalmente e irán evolucionando conforme se vayan consultando a distintos jefes de proyecto de las distintas áreas de Adidas.
- En el análisis inicial del proyecto se han documentado algunas métricas deseadas, sin embargo, la forma de su visualización no lo está.
- La REST API, que será accedida a través de la aplicación para obtener los datos, no está desarrollada ni totalmente documentada, por lo que es posible que el modelo y la estructura de datos vaya cambiando respecto al tiempo.
- Nuevas métricas podrán ser añadidas según nuevos requisitos aparezcan.
- El coste en tiempo será fijo, el correspondiente al trabajo fin de grado.

En los siguientes apartados se van a comparar los dos enfoques y analizar cuál de ellos es el más adecuado para cumplir los aspectos especificados anteriormente.

1.4.2 Desarrollo tradicional

Uno de los principales problemas del desarrollo tradicional es su secuencialidad. Se divide en cuatro partes que se ejecutan una detrás de otra. Éstas son la planificación, el diseño, el desarrollo de código y el testeo, como se puede ver en la Figura 3. La ejecución de estas cuatro fases constará como la ejecución del proyecto (que costará más o menos tiempo dependiendo del tamaño de éste), lo que propiciará que no se tenga nada que enseñar al cliente hasta su finalización.

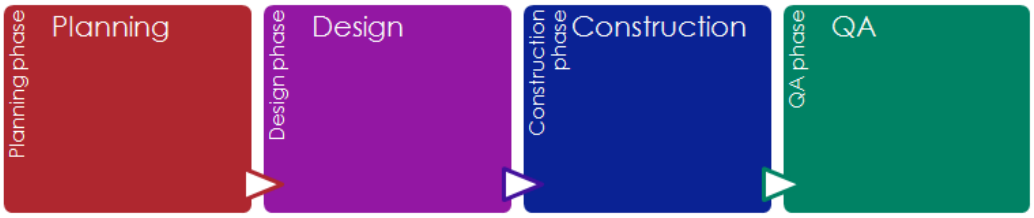


Figura 3: Secuencia en el desarrollo tradicional

Esto repercute en uno de los siguientes problemas: si el proyecto estaba mal definido o insuficientemente documentado **se puede entregar algo que no era lo que se había pedido,** sino lo que se creía que se había pedido. Como se puede ver en la Figura 4, si el proyecto estaba algo mejor definido es posible entregar lo que el cliente aceptaría pero en muy pocos casos o casi ninguno se podrá entregar lo que el cliente necesita de verdad porque **en muchos casos ni siquiera él lo sabe.**

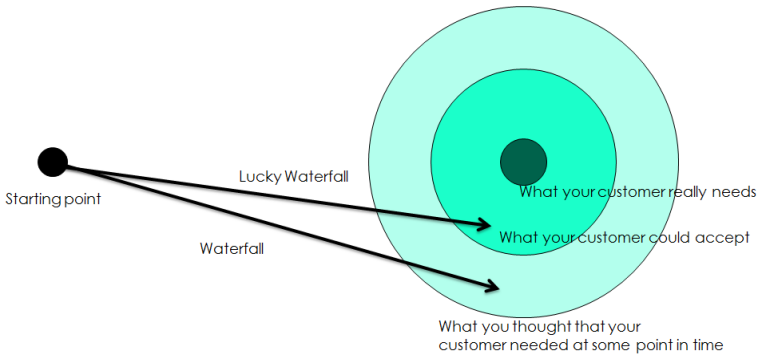


Figura 4: Finalización desarrollo tradicional

Finalmente, otro de los mayores problemas del desarrollo tradicional es que el proyecto está fijado desde el principio, el "scope" es invariable y se estima el tiempo y el presupuesto que podrán variar como podemos ver en la Figura 5. Si durante su desarrollo cambian los requisitos (algo no contemplado durante el desarrollo tradicional) el esfuerzo que debe realizarse para adaptar el proyecto será muy grande o incluso inviable al estar planeado y diseñado desde el principio con los requisitos iniciales.

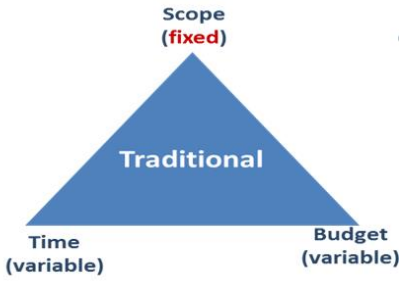


Figura 5: Scope en el desarrollo tradicional

1.4.3 Metodologías ágiles

El desarrollo *Agile* es mucho más flexible que el desarrollo tradicional. Se caracteriza principalmente porque los requisitos y las especificaciones del proyecto (scope) son variables (Figura 6), por lo que éste no se tiene que decidir al principio y puede definirse conforme se va teniendo mayor conocimiento sobre lo que el cliente desea. Podemos encontrar sus bases en el apartado 6.5.

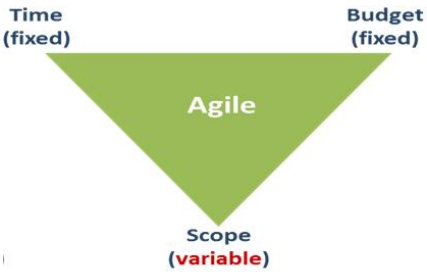


Figura 6: Scope en el desarrollo Agile

Como se ha dicho anteriormente, el cambio de requisitos genera problemas, ya que para desarrollar la aplicación es necesario planificar y diseñar según éstos con anterioridad. Por ello, *Agile* deja de ser secuencial, y el proceso realizado en el modelo tradicional (planificación, diseño, desarrollo de código y testing) se realiza n veces en unidades de tiempo constantes (ej. dos semanas) llamadas “sprint” donde se define el scope (Figura 7).

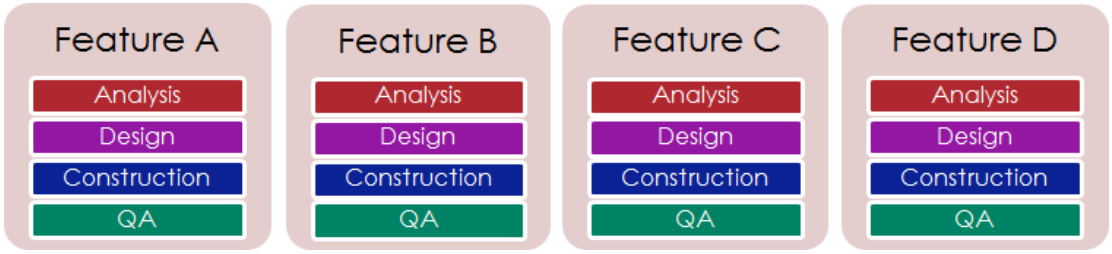


Figura 7: Desarrollo Agile no secuencial

Posibilitando el cambio de requisitos a lo largo del tiempo, permite al cliente obtener un producto que se ajuste a sus necesidades. Sin embargo, y como se puede ver en la Figura 8, muchas veces se toman decisiones de requisitos que luego se desechan por lo que lleva frecuentemente a tomar un camino más largo, necesitando más tiempo y más recursos. Sin embargo, el resultado es mejor que en el desarrollo tradicional.

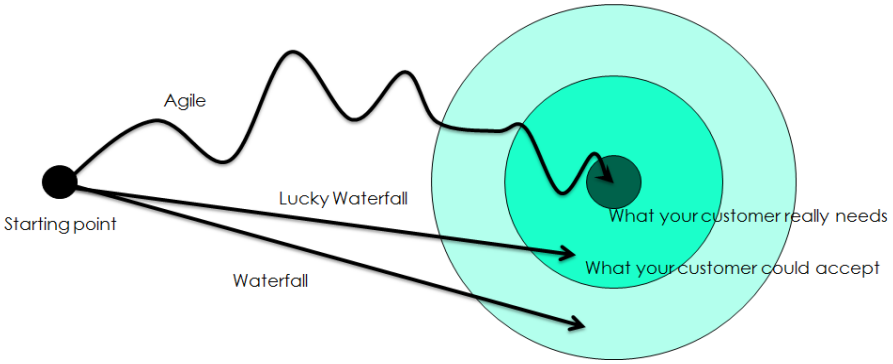


Figura 8: Desarrollo Agile vs desarrollo tradicional

1.4.4 Conclusión:

Siguiendo las ventajas y desventajas explicadas en el anexo 6 de ambos enfoques, debido a que el tradicional es muy inflexible y los requisitos del cliente, en este caso el departamento Global IT de Adidas, no están bien definidos y se irán añadiendo durante el desarrollo, se certifica la elección del enfoque *Agile*. La metodología utilizada está detallada en el apartado 3.1 donde se explica qué acciones se han tomado siguiendo el enfoque seleccionado.

1.5 Herramientas de desarrollo

Como editor principal se ha utilizado **Sublime Text 3**, multiplataforma que soporta la mayoría de lenguajes de programación, en este caso JavaScript. Como funcionalidad principal a destacar es la posibilidad de añadir distintos *plugins* creados por el usuario (ej. eliminar espacios innecesarios, convertir saltos de línea…). Además es muy ligero, por lo que su tiempo de carga es rápido.

La primera herramienta utilizada fue un generador de **Yeoman** [15]. Es una herramienta de desarrollo en el lado del cliente de código libre que provee herramientas para ayudar a desarrollar aplicaciones web. Está basado en Node.js y combina distintas funcionalidades como generar una estructura de código de base, incluir dependencias y correr test unitarios de forma automática a través de la línea de comandos. En este proyecto se ha usado principalmente para generar la estructura de ficheros y los ficheros de test unitarios.

Bower [16]**:** es un sistema de administración de paquetes o librería para el lado del cliente en el desarrollo Web. Depende de Node.js y npm. Funciona con repositorios de git. En este proyecto se utiliza Bower para obtener todas las librerías que utiliza la aplicación.

Npm [17]**:** basado en la misma tecnología que Bower. Además de las librerías para el lado del cliente posee todo tipo de librerías JavaScript con el objetivo de reutilizar y difundir código.

Desde Npm obtenemos **Grunt** [18] es un lanzador automático de tareas que agiliza en gran medida el testeo, servir la aplicación, crear un paquete de despliegue (más información en el apartado 2.5), lanzar un servidor de mock data, generar documentación JSDOC (ver anexo 19 de documentación de código), compilar Less en CSS... Además es posible crear tareas personalizadas según las necesidades del usuario. En resumen, permite agilizar tareas que si no, el desarrollador tendría que realizar manualmente.

Como tarea a destacar de Grunt se encuentra el **linter** (en este caso se usa eslint)**:** es un programa que comprueba estilos y errores de código. En Adidas se usan de manera asidua para asegurarse que se siguen sus estilos y que la calidad de éste es óptima con el objetivo de que sea legible.

Existe Gulp [19], una herramienta con la misma funcionalidad que Grunt, sin embargo se ha elegido esta segunda opción ya que existe mucha más documentación, la comunidad es más grande, tiene tareas oficiales mantenidas y el código para crear tareas personalizadas es mucho más sencillo. No se necesita gran rendimiento en automatización de tareas que serán sólo usadas en desarrollo.

1.6 Estructura de la memoria

En esta sección se ha descrito el ámbito del proyecto, sus objetivos y las tecnologías y herramientas utilizadas.

En la sección 2 se explicará el trabajo realizado. Se empezará con la fase de análisis donde se explicará el problema y se abordarán los requisitos, especialmente las métricas a representar. Posteriormente se detallará la fase de diseño donde se especificará la arquitectura del sistema y de la aplicación y el diseño de las gráficas. A continuación, la implementación entrando en detalle sobre la generación de gráficas, la integración del sistema y la seguridad. Hay que tener en cuenta que se sigue una metodología *Agile*, por lo que el proceso de análisis, diseño e implementación es iterativo e incremental. Finalmente, se hablará de las distintas pruebas realizadas para asegurar la funcionalidad y calidad del código, la utilización de la integración continua y las claves sobre el rendimiento de la aplicación.

En la sección 3 se explicará la metodología utilizada, Scrum.

La memoria se cerrará con el resultado de la aplicación, el trabajo futuro y una opinión personal en la sección 4.

Tras la memoria se adjuntan los siguientes anexos:

1. Diagrama de casos de uso
2. Diagramas de secuencia
3. Evolución de la arquitectura del sistema.
4. Mapa de navegación
5. Formación de los usuarios
6. Cascada vs *Agile*, ventajas y desventajas
7. Principios *Agile*
8. Gestión de tiempo
9. Funcionamiento interno de AngularJS
10. Estructura de JWT
11. Gráficas generadas con mayor detalle
12. Capturas de pantalla de Jira
13. Capturas de pantalla de Sonar
14. Configuración de TeamCity
15. Capturas de los requisitos de negocio obtenidos en Confluence
16. Obtención de la métrica Documented APIs
17. Capturas de la configuración de la aplicación
18. Rendimiento de la aplicación
19. JSDOC

2 Trabajo realizado

En este capítulo se pretende explicar el trabajo de ingeniería que ha sido realizado para alcanzar el objetivo planeado. Hay que tener en cuenta, que al seguir un modelo *Agile* se ha seguido un proceso iterativo e incremental de análisis, diseño e implementación como se aprecia en la Figura 7. En esta sección se va a explicar el resultado final en cada uno de estos procesos. También se especificará en aquellos casos en los que se tomó una decisión y posteriormente se rectificó.

2.1 Análisis

En este apartado se va a abordar cuál es el problema a solucionar y los requisitos que obtenemos desde negocio a través de épicas, especialmente los tipos de métricas.

2.1.1 El problema

Desde la primera iteración se estableció con claridad que el problema a resolver era el diseño e implementación de una aplicación web pesada que se ejecutaría en un navegador, la cual ofrecería al usuario mecanismos para conocer el estado de los distintos proyectos desarrollados en ADIDAS. Para acceder a la información de estos proyectos, esta aplicación se conectaría a una REST API que obtiene, a través de servicios externos, una serie de métricas de los proyectos. Esas métricas son las utilizadas por la aplicación para mostrar los distintos gráficos, que permitirán al usuario obtener una idea general del estado del proyecto e información más detallada en determinados aspectos.

2.1.2 Tipos de métricas

Como se ha dicho anteriormente, la aplicación de métricas reúne datos de distintas herramientas (Sonar, Jira y Stash), en esta sección se va a entrar en más detalle en qué datos se muestran y cómo se organizan.

Sonar, también llamado SonarQube es una herramienta de código libre que permite el análisis de fragmentos de código. Soporta todos los lenguajes más comunes como Java, C, C++, JavaScript, Python… Lo más importante es que todos los lenguajes en los que Adidas tiene proyectos están soportados. Se pueden observar capturas de pantalla de los resultados que Sonar obtiene del proyecto en el anexo 13. De todas las métricas que Sonar obtiene se extraen las siguientes:

- ***Code violations*:** Número de problemas de calidad encontrados en el código. No son problemas funcionales sino de mantenibilidad, legibilidad, buenas prácticas o que pueden generar comportamientos inesperados. Hay distintos niveles de severidad:
    - ***Blocker*:** Riesgo de seguridad o fallo funcional. El problema puede derivar en un comportamiento inesperado de la aplicación (ej. NullPointerException).
    - ***Critical*:** Riesgo de seguridad o fallo funcional. El problema puede producir inestabilidad en la aplicación (ej. olvidar cerrar un socket).
    - ***Major*:** Riesgo mayor en la productividad y mantenibilidad (ej. funciones demasiado complejas).
    - ***Minor*:** Riesgo menor en la productividad y mantenibilidad (ej. convención de nombre de variables)

    - ***Info*:** Comentarios a los desarrolladores, no causa impacto negativo.
- ***Documented APIs*:** Número de líneas con comentarios significativos (Anexo 16 *Documented APIs*). Se calcula un porcentaje con respecto a la cantidad de funciones expuestas públicamente.
- ***Unit test coverage*:** Número de líneas de código ejecutable que son testeadas con *unit test* (ejecutada durante los test). Una vez se obtiene este valor se divide entre el número total de líneas de código ejecutable. El resultado es un porcentaje
- ***Duplications*:** Número de bloques de líneas duplicados en el código. Se considera un bloque a aquél con al menos cien caracteres duplicados seguidos. El sangrado es ignorado.
- ***Code complexity*:** Se obtiene el resultado de la complejidad media tanto de las funciones, clases (no existen en AngularJS) y ficheros. Cada vez que el flujo de ejecución se divide (ej. en un *if*) la complejidad aumenta en uno.

Jira expone una API que provee información sobre *sprints*, desarrolladores, valor de las tareas en *story points* y *bugs* encontrados. Se pueden observar capturas de pantalla de la aplicación web de Jira en el proyecto y más información en el anexo 12. Las métricas que se obtienen a través de Jira son las siguientes:

- ***Defects*:** Corresponden al número de bugs encontrados en proceso de desarrollo actualmente abiertos en un proyecto. Tanto si están en progreso, siendo revisado o todavía no están asignados.
- ***GTT Defects*:** Mismas características que *defects*, pero estos bugs han sido encontrados en proceso de *testing*.
- ***Old Defects*:** Corresponde al número de bugs abiertos durante más de una semana.

Estos bugs tienen los mismos niveles de severidad explicados en *code violations*.

Stash es un software de la empresa Atlassian de repositorios Git. Provee una interfaz web así como una REST API para acceder a algunos recursos. Correspondiente a Stash y cruzando datos con Sonar y Jira obtenemos las siguientes métricas:

- ***Defects per developer*:** el número de defectos de cualquier tipo que tiene asignado cada desarrollador.
- ***Velocity per Sprint*:** El número de *story points* quemados durante un *sprint*.
- ***Velocity / Capacity*:** La velocidad con la que se queman los *story points* que tiene cada desarrollador como capacidad asignada.

Se han seleccionado estas métricas pero se pueden añadir más, y en el futuro se espera que se haga como se explica en el apartado 4.2.

2.1.3 Configuración de los proyectos

Adidas tiene gran cantidad de proyectos desarrollados por distintos equipos, por lo que su visualización en métricas uno por uno sería un caos. Para ello, se agrupan en lo que se denomina área: conjunto de proyectos que siguen una estrategia común o pertenecen a la misma área de negocio. Sin ir más lejos, el área a la que el proyecto pertenece está formada por dos proyectos, GMP-backend y GMP-frontend.

Por tanto, los datos a mostrar se realizarán por áreas. El cálculo de *code violations,* como de todos los tipos de *defects,* será aditivo. Sin embargo, todos los datos porcentuales (*documented APIs*, *unit test coverage* y *duplications*) y *code complexity* serán ponderados según el número de líneas

de cada proyecto. Por consiguiente, tendrá mayor impacto que un proyecto de 10.000 líneas de código tenga un cero por ciento de *unit testing coverage,* que uno de 1.000.

Además de las métricas, en cada área se configurará manualmente umbrales dados por el contrato con el que se ha firmado cada proyecto. Cada métrica (con su correspondiente nivel de severidad) tendrá estos umbrales como se puede observar en la Figura 9. Serán configurables a través de la página web siempre y cuando el usuario esté autenticado como administrador del área a modificar. El color que adquiera la gráfica corresponderá con el umbral al que el dato pertenezca.

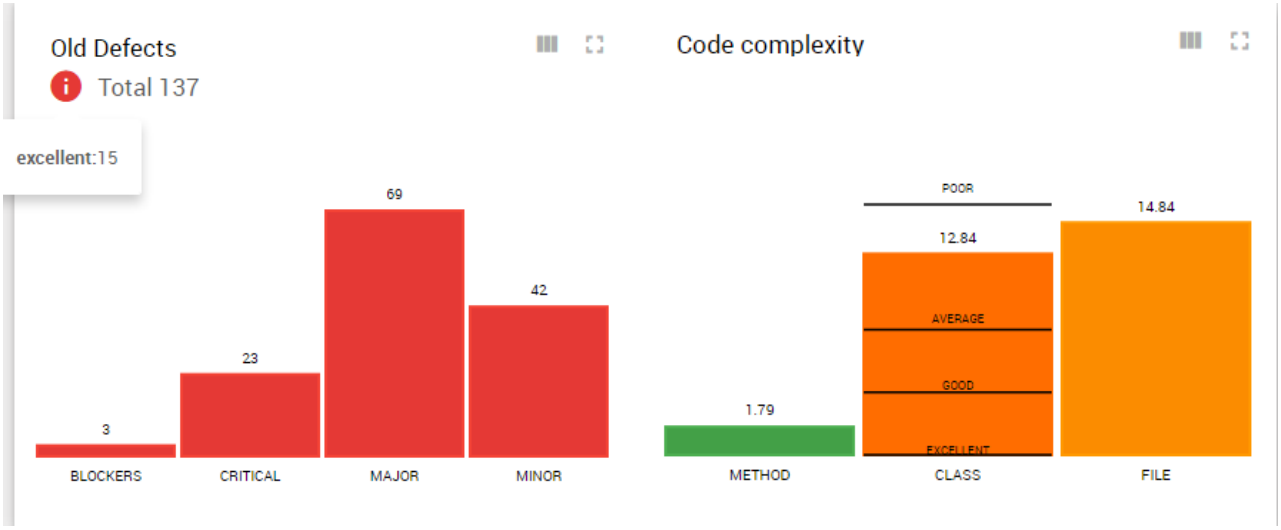


Figura 9: Umbral de Old Defects totales y de complejidad de clase

Como se puede ver en la Figura 10 y en el anexo 17 de las imágenes de configuración, aparte de los umbrales, es posible configurar distintos apartados a destacar:

- **Administradores:** Usuarios capaces de modificar la configuración de un área en concreto.
- ***Team users*:** miembros del equipo de desarrollo.
- ***Sprints*:** Considerado como un periodo de tiempo en el que se deben cumplir ciertos objetivos con respecto al área, como sería por ejemplo bajar el número de violaciones de código o alcanzar cierta velocidad de desarrollo.

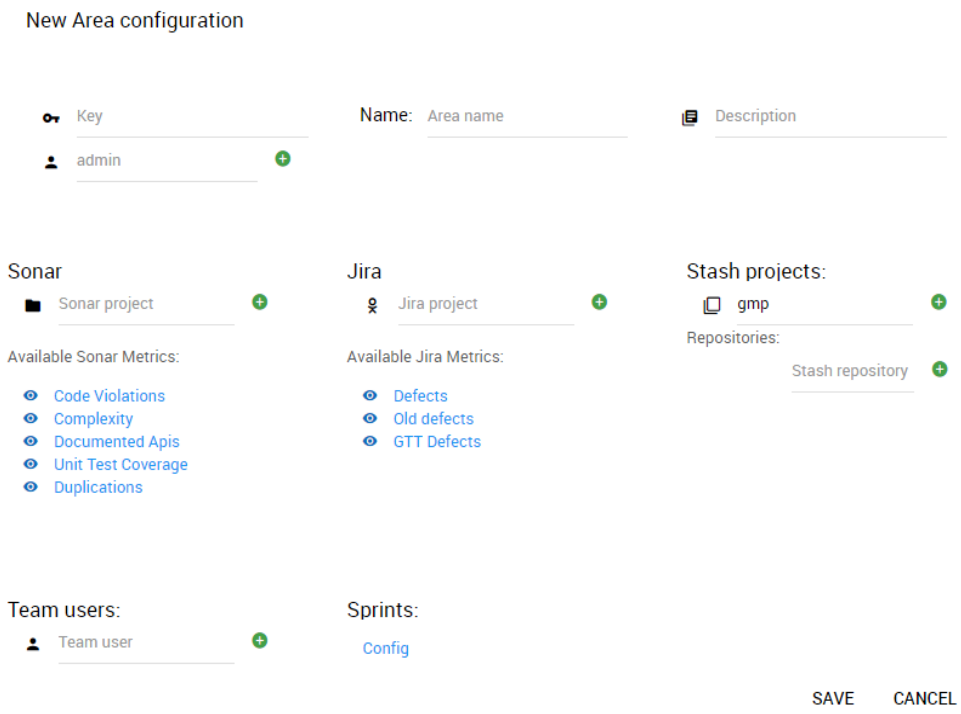


Figura 10: Página principal de configuración

2.1.4 Casos de uso

Gracias a las épicas recibidas (requisitos desde negocio de las métricas a mostrar) se obtienen unos requisitos iniciales. Con ellos se ha hecho un diagrama de casos de uso mostrado en el Anexo 1. Como se ha mencionado con anterioridad, se obtienen requisitos de forma incremental. Se empezó con *Code Violations* y *Defects*, posteriormente se añadieron los requisitos de *Documented APIs, Unit Test Coverage* y *Duplications*. Los últimos requisitos añadidos fueron *Code Complexity* y *Defects per developer*.

2.2 Diseño

En este apartado se hará hincapié en diseño de la arquitectura (tanto del sistema como de la aplicación), las vistas y las gráficas. Como se ha mencionado anteriormente su ejecución ha sido incremental.

2.2.1 Arquitectura del sistema

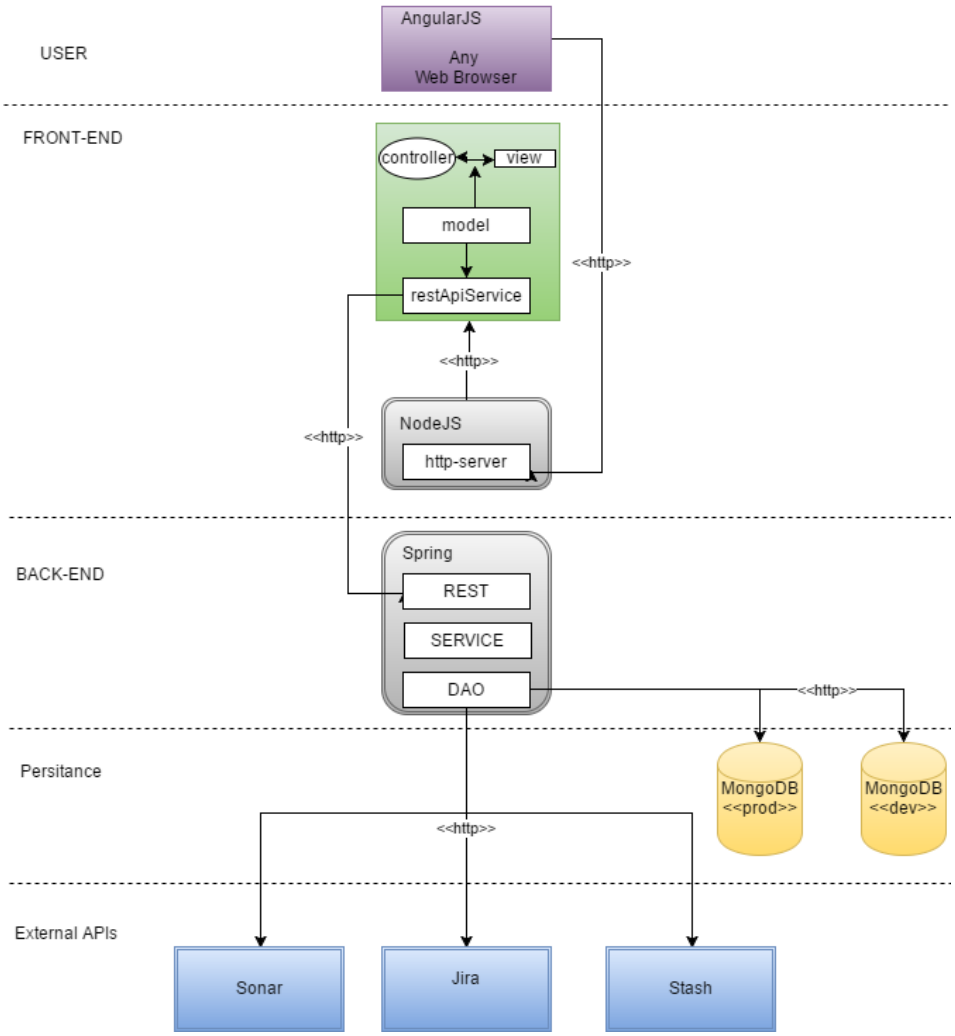


Figura 11: Versión definitiva de arquitectura del Sistema

Como se puede ver en la Figura 11, la aplicación Web (cuya arquitectura es explicada en detalle en el apartado 2.2.2) es alojada en un servidor *frontend,* al cual los usuarios accederán a través de un explorador de Internet. La aplicación se comunicará con un servidor *backend* provisto por Adidas. Este servidor será el encargado de pedir los recursos tanto a Sonar, Jira y a la base de datos. Esta máquina se convierte en un *gateway* (un intermediario) para acceder a los recursos de Sonar, Jira, Stash y de la base de datos. Su principal objetivo es permitir la persistencia de datos históricos, los cuales, cada día se guardarán de forma periódica en MongoDB.

Sin embargo, ésta no es la versión, debido al proceso iterativo seguido. La evolución de esta arquitectura está disponible en el anexo 3.

2.2.2 Arquitectura de la aplicación web

En la Figura 12 se representa la arquitectura interna de la aplicación desarrollada. Se divide principalmente en tres grandes bloques que corresponden a dos grandes vistas (vista de estado y vista de tendencia, explicado en siguiente apartado) y a la configuración de proyectos (apartado 2.1.3).

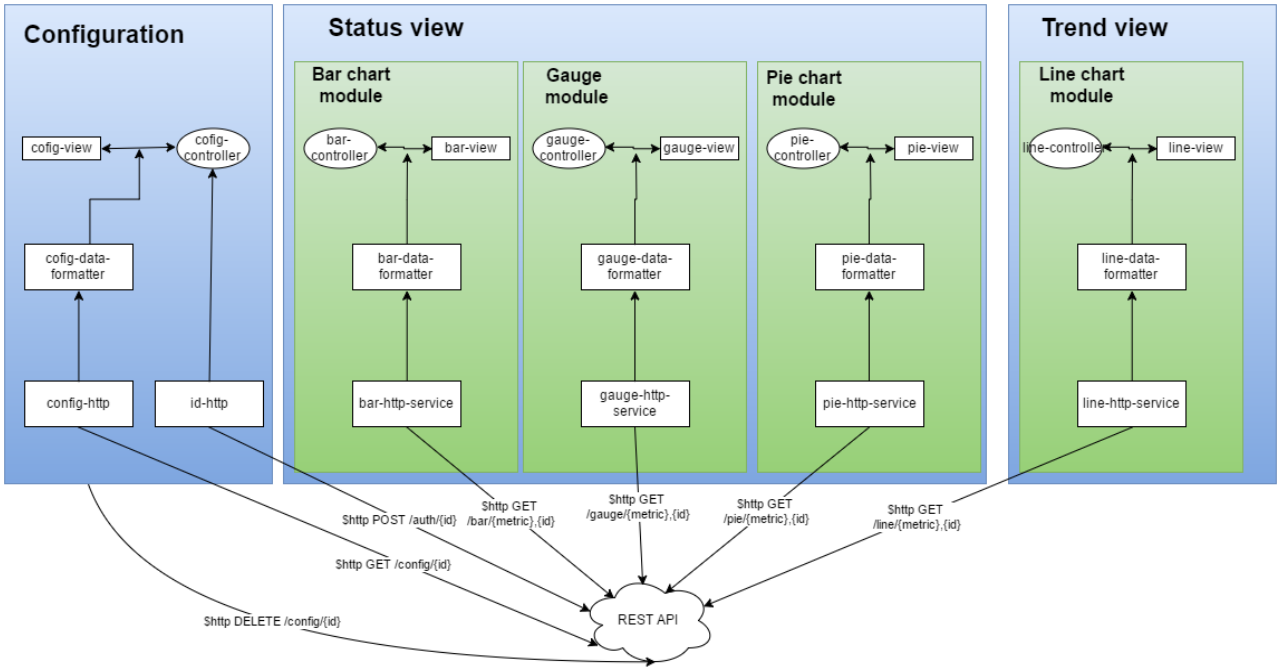


Figura 12: Arquitectura interna

En cada una de las dos vistas hay un módulo para cada tipo de gráfica (que no para cada métrica). Ya que aquellas métricas que se representen de la misma forma (ej. gráfica de barras) utilizarán el mismo módulo para su visualización. Cada módulo sigue la misma estructura; un servicio para lanzar la llamada adecuada a la REST API (en nuestro caso el servidor de Adidas) utilizando HTTP [20] y otro servicio para formatear esos datos, de forma que sean representables a través de la librería correspondiente. Más información en el apartado 2.3.1. Una vez se disponen los datos se entra en el Modelo-Vista-Controlador. Donde el modelo son los datos recibidos, la vista corresponderá con el *template,* se mostrará la gráfica y el controlador será el encargado de ejercer los cambios correspondientes cuando el usuario interactúe con la aplicación.

El bloque de la configuración corresponderá a todo el funcionamiento de configuración. Será necesario estar autenticado y tener los permisos adecuados para acceder a esta información, tanto en lectura como en escritura. Una vez confirmada la autenticidad del usuario, éste podrá acceder utilizando el mismo modelo descrito con anterioridad. En este caso se extiende la funcionalidad del servicio HTTP permitiendo hacer las operaciones POST y PUT para guardar nuevas configuraciones y modificar existentes respectivamente.

En esta arquitectura de módulos, éstos se han ido añadiendo de forma incremental, empezando con el de estado, al que se le fueron introduciendo las distintas gráficas a mostrar. Posteriormente el de tendencia y finalmente el de configuración. Gracias al este diseño modular esta aplicación es completamente escalable en cuanto a la implementación. Se pueden añadir cuantos tipos de gráficas se desee siguiendo la arquitectura previamente descrita.

Servicio de control de módulos

Hemos visto la unicidad de los distintos componentes de la aplicación, sin embargo esta funcionalidad tiene que ser controlada desde un servicio centralizado para dirigir que módulos deben representarse y con qué modelos de datos en las distintas vistas.

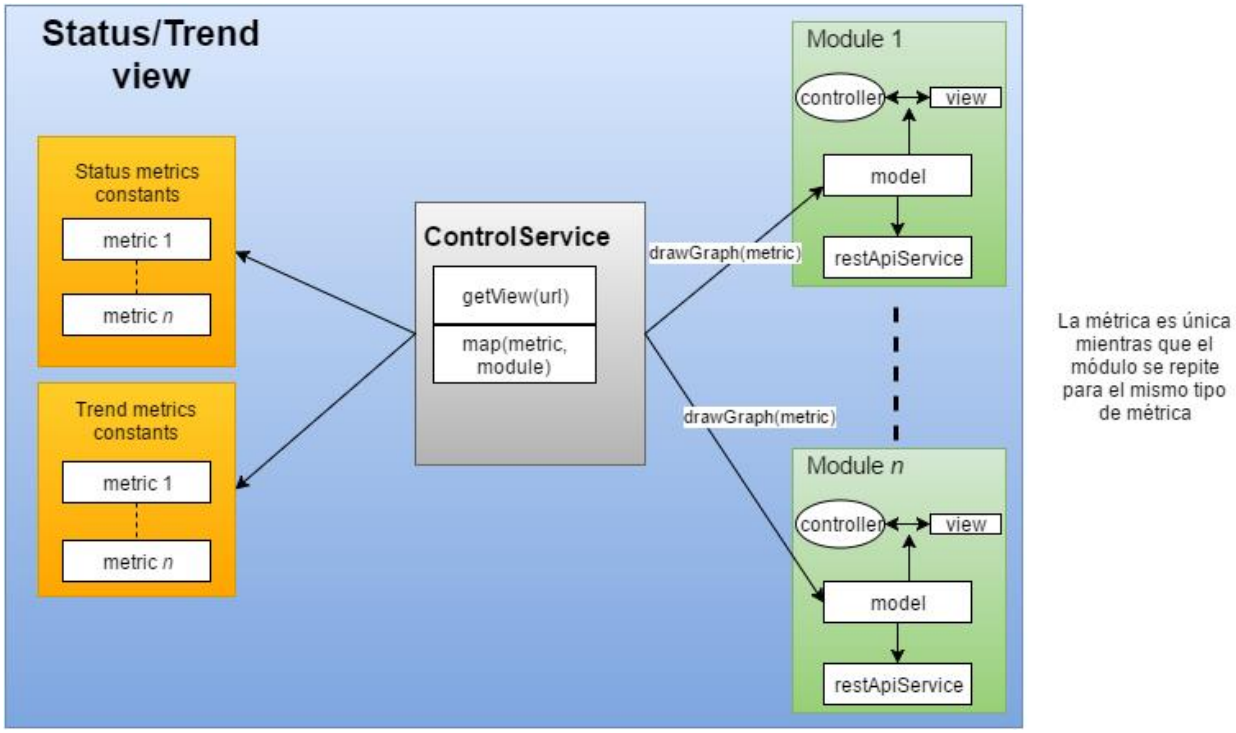


Figura 13: Arquitectura del servicio de control

Dentro de la vista de estado y de tendencia existe el servicio encargado de controlar qué métricas deben ser mostradas en la correspondiente vista representada en la Figura 13. Para ello se dispone de varias constantes listando las distintas métricas a representar de cada vista. Gracias a éstas genera un modelo de datos para su posterior representación. Este modelo es una lista ordenada por orden de aparición dentro de la vista. El servicio de control será el encargado de actualizar estos índices en dos pasos:

1. Cuando se hayan recibido lo datos sin procesar. Http-service será el encargado de realizar está acción.
2. Cuando se hayan procesado los datos obteniendo aquellos que representar. Data-formatter será el encargado de llevar acabo esta acción.

Posteriormente, estos datos pasarán al Modelo Vista Controlador donde el servicio de control habrá acabado ya su tarea.

Además, este servicio es el encargado de detectar si ha habido un error durante el proceso de carga o formateo de datos mostrando el correspondiente mensaje de error de cara al usuario. También es el encargado de controlar que el *spinner* de carga desaparezca cuando la actualización del índice correspondiente haya sido validada en sus dos pasos.

Localización de acceso al *backend*

La obtención del punto de acceso al *backend* es modular y dinámica. Se cargará a través de una llamada HTTP a un fichero de configuración introducido durante la fase de despliegue (explicada en el apartado 2.5).

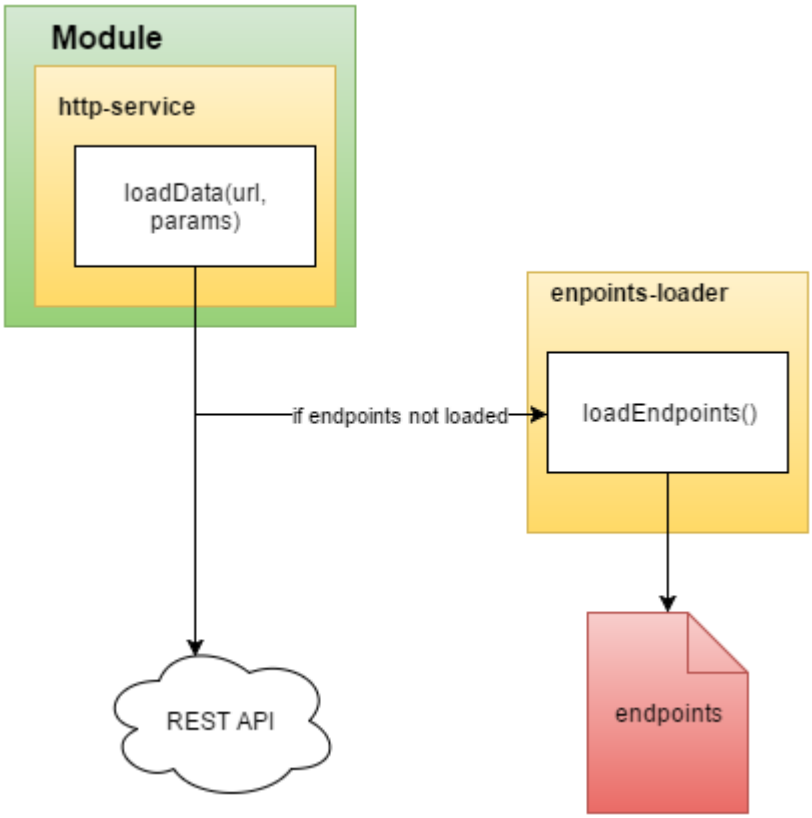


Figura 14: Carga de punto de acceso

Como se observa en la Figura 14, en cada llamada a la REST API se hará una comprobación si el punto de acceso ha sido previamente cargado, de lo contrario éste será cargado de un fichero propio del proyecto de configuración. Aun así, debido a la arquitectura de AngularJS, esta llamada seguirá siendo $http. Una vez ésta se ha completado se procederá a pedir el recurso a la dirección correspondiente de la REST API. Como punto negativo a esta modularidad, en la primera llamada accediendo al servicio *backend* se perderán unos milisegundos (en torno a 20), que no causarán un gran impacto en el rendimiento de la aplicación.

Esta funcionalidad nos permite desplegar el servidor en cualquier máquina sin necesidad de realizar ningún cambio en el código ni en el paquete creado tras el proceso de empaquetado (explicado en el apartado 2.5).

2.2.3 Diseño de las vistas

Una vista es un conjunto de datos representados numéricamente o a través de gráficas. Éstas han sido diseñadas para distinguir la forma en la que se muestra la información:

- De forma general como la vista de áreas y la principal. con el objetivo de que el usuario obtenga de un vistazo unas primeras impresiones sobre las áreas.
- De forma más concreta como la vista del estado actual y de tendencia. Éstas están formadas por las mismas métricas representadas a través de gráficas perteneciendo a una única área.

Para transmitir la información al usuario sobre un área existen cuatro vistas principales:

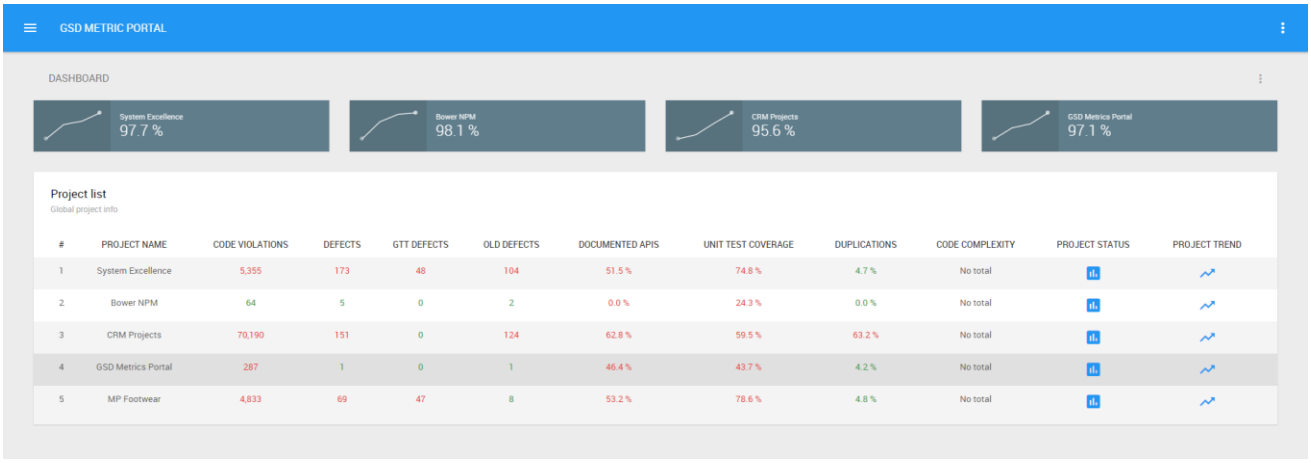


Figura 15: Página principal

VISTA GENERAL DE ÁREAS

Esta vista aparece en la parte superior de la página de inicio que se puede apreciar en la Figura 15. Por defecto se ven las cuatro mejores áreas, sin embargo, el usuario tiene la posibilidad de configurarlo y tener predefinido las que quiere visualizar una vez esté autenticado en la aplicación.

Esta visión general permite al usuario ver la evolución de un área a lo largo de un año. El dato que se muestra es porcentual, y es introducido por un administrador de la propia área mensualmente. No es un valor autogenerado derivado de la información obtenida, al menos de momento.

VISTA PRINCIPAL

Como se puede observar en la Figura 15, en la página de inicio se muestra una tabla con los proyectos disponibles para ver su información. Desde ella será posible acceder a la vista de estado actual o de tendencia (explicadas más adelante). Además se muestran numéricamente las distintas métricas existentes con una marca de color indicando el estado actual total.

ESTADO ACTUAL DE PROYECTO

En primera instancia se diseñó un *mock up* de cómo esta vista debería ser, como se puede observar en la Figura 16.

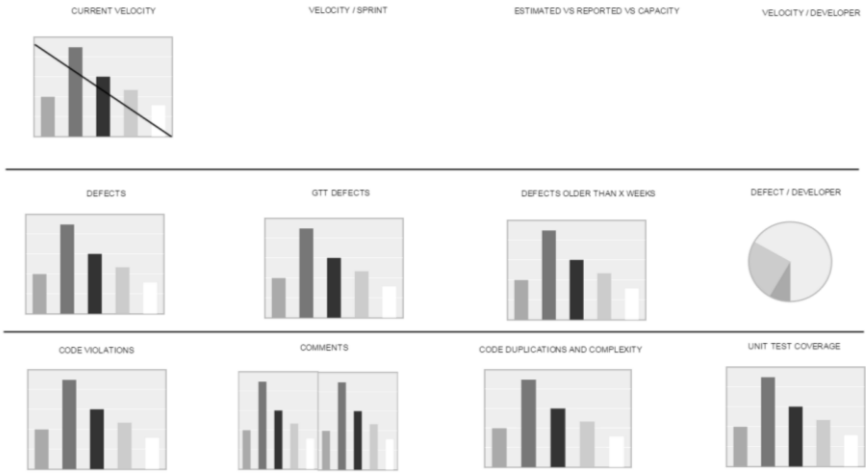


Figura 16: Mock up de la vista de estado actual

En esta vista se muestran todas las métricas (si existen) en un momento concreto del proyecto. Así el usuario es capaz de ver el estado concreto de un proyecto. Es posible seleccionar la fecha en la que se desea verlo.

TENDENCIA DE ÁREA

Al igual que en la vista de estado actual, se diseñó un *mock up* con la vista de tendencia, como se observa en la Figura 17.

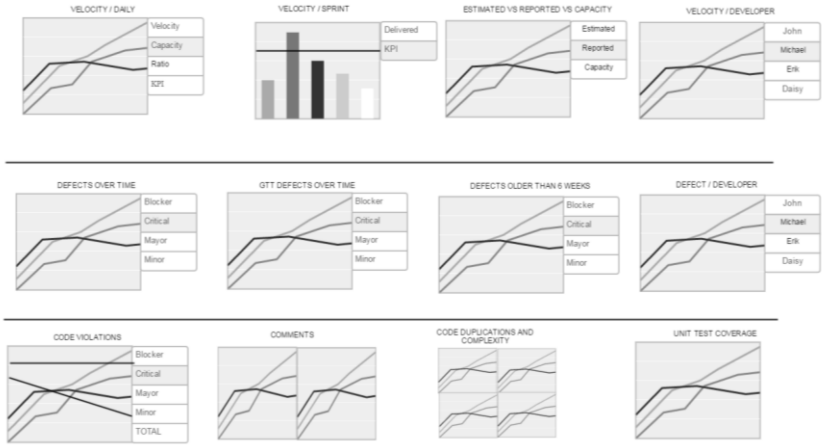


Figura 17: Mock up de vista de tendencia

En esta vista cuyo resultado está en la Figura 18, se aprecia cómo ha evolucionado cada métrica en concreto a lo largo del tiempo. Al igual que en la vista de estado se puede elegir un rango de fechas que analizar. Por defecto serán los últimos tres meses a partir de la fecha actual.

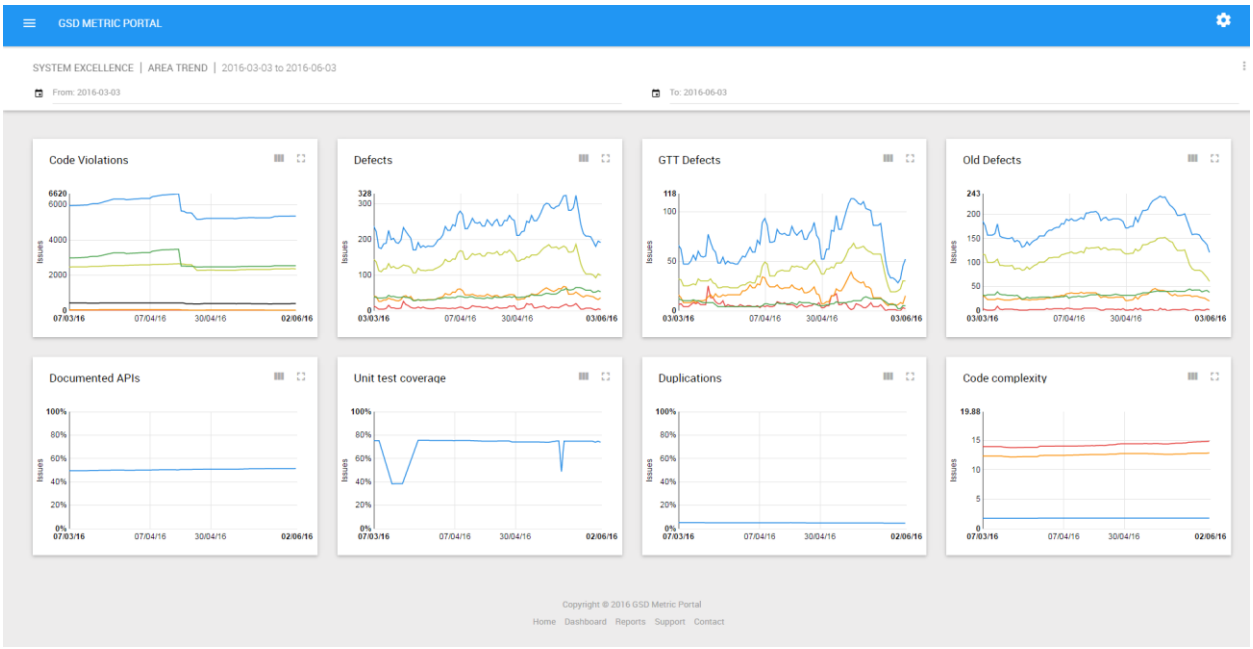


Figura 18: Resultado de área de tendencia

Tanto en estado como en tendencia (aunque su funcionalidad es principalmente útil en tendencia) es posible aumentar el tamaño de gráfico de una métrica, de esta forma se pueden observar las modificaciones a lo largo del tiempo con mayor precisión.

Todas las vistas que dependen de una fecha (tanto estado como tendencia de área) son dependientes de la URI, así, si un usuario desea compartir la vista actual, puede compartir el link actual y el destinatario será capaz de visualizar la misma información que el emisor estaba visualizando.

Para la vista de tendencia existen dos parámetros de la URI (denominados *query strings* [21]) que indican la fecha de inicio y la fecha de fin. Para la vista de estado sólo existe uno.

2.2.4 Mapa de navegación

Se diseñó una primera versión del mapa de navegación que se puede observar en la Figura 19. A partir de ahí se implementó en la aplicación siguiendo la misma lógica, como se puede ver en el Anexo 4. Se observa que la navegación se realiza de forma intuitiva, siendo posible desplazarse entre vistas y fechas de forma que el usuario no necesite un entrenamiento previo para su uso.

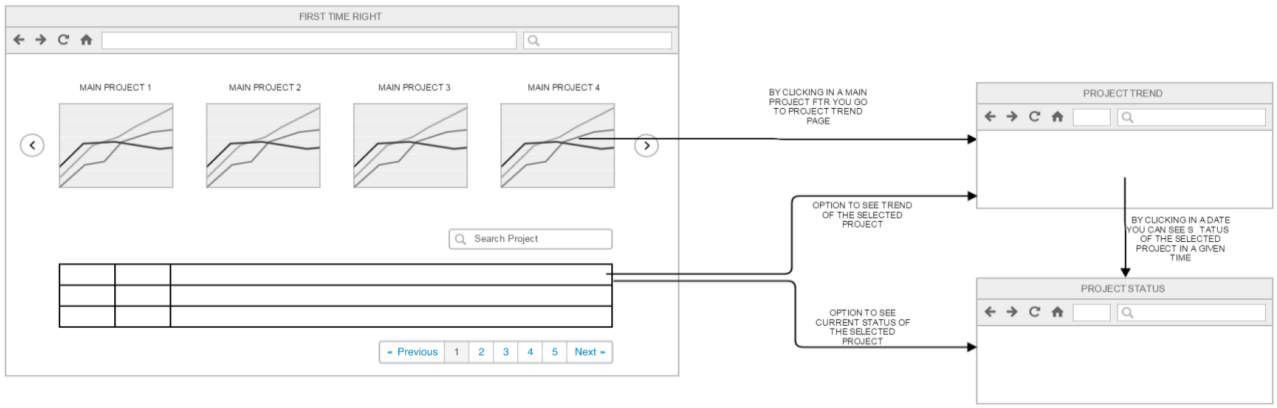


Figura 19: Mock up de navegación

2.2.5 Diagramas de secuencia

Análogo al mapa de navegación se expone un diagrama de secuencia en el Anexo 2, en el que se observa las interacciones necesarias con la aplicación que el usuario debe realizar para completar una tarea. También se muestra la secuencia de interacciones entre la aplicación y el servidor.

2.2.6 Diseño de gráficas

Desde negocio, a través de la herramienta Confluence (anexo 15), se especificó como requisito las métricas a representar, sin embargo la gráfica a mostrar fue diseñada personalmente. En esta fase se concreta los tipos de gráficas que se van a utilizar para representar las distintas métricas.

Graficas de barras

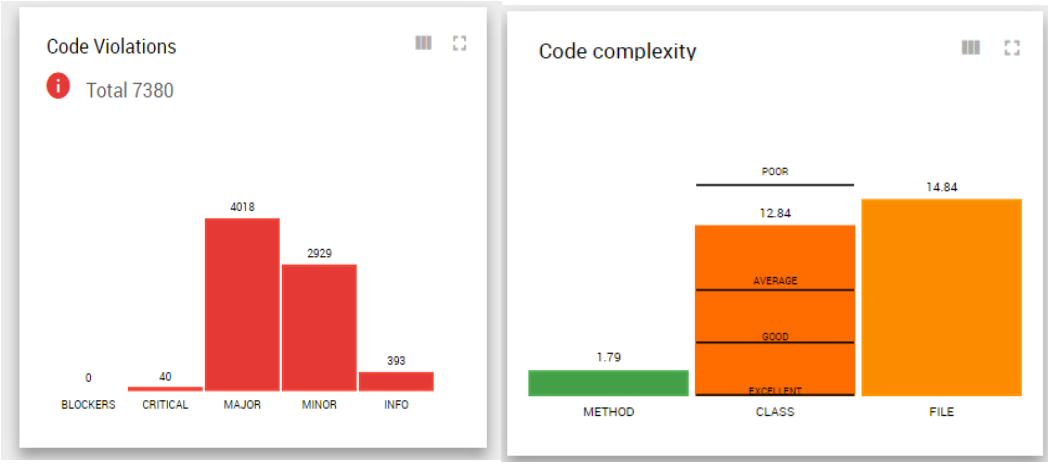


Figura 20: Gráfica de barras para Code Violations y Code Complexity

Con este tipo de gráfica se representan las métricas de Code Violations, Defects, Old Defects, GTT Defects y Code Complexity. Existen tres razones principales:

- ✓ Representan un número absoluto (no es un porcentaje o un rango)
- ✓ Dentro de cada una de estas métricas hay distintas categorías. No tendría sentido un gráfico de barras con una única columna.
- ✓ Representa un estado actual, no requiere una progresión en el tiempo.

Como se puede ver en la gráfica derecha de la Figura 20, cuando el usuario pasa por encima el ratón (*hover*) se muestran los umbrales de la correspondiente columna. Así el usuario, de un vistazo, puede observar cuánto debe mejorar en ese aspecto para alcanzar un umbral mejor.

Velocímetro

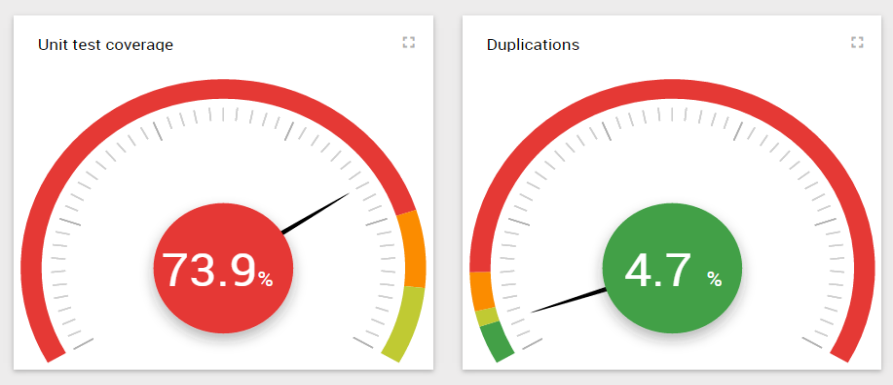


Figura 21: Velocímetro para Unit Test Coverage y Duplications

Esta gráfica representa las métricas medidas en porcentajes: Documented APIs, Unit Test Coverage y Duplications. Para representar una métrica en la que el valor a tomar está dentro de un rango [0-100] se necesita una gráfica acotada, como en este caso es este velocímetro. Permite transmitir al usuario que hay un valor máximo y mínimo.

El aro exterior transmite en qué estado (o umbral) se encuentra la métrica. Como se puede ver en la Figura 21, los umbrales de duplicaciones de líneas de códigos son inversos a Unit Test Coverage, ya que cuantas menos duplicaciones existan en un proyecto es mejor, mientras que sucede lo contrario con los test unitario y el código documentado, que cuanto más, mejor.

Gráfico de líneas

Para la vista de tendencias se ha utilizado gráficas de líneas, ya que representan una evolución temporal. Se ha utilizado el mismo tipo de gráfica en todas las métricas para mantener la consistencia y porque lo que se representa es lo mismo: las variaciones que ha habido de una métrica a lo largo del tiempo. Para las gráficas de porcentajes los umbrales siempre estarán acotados entre 0% y 100%.

Es posible desplegar una leyenda para activar y desactivar líneas, permitiendo al usuario ver las líneas deseadas. Por defecto, la leyenda no está desplegada porque hace que los gráficos sean más chatos y que se requiera una altura mayor. Sin embargo, cuando el gráfico está maximizado la leyenda está habilitada por defecto.

Gráfico de línea minimalista

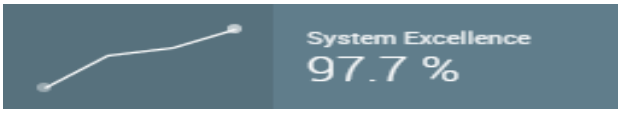


Figura 22: Gráficos de visión general de área

Para representar la visión general de área se ha utilizado un gráfico sencillo y minimalista para no sobrecargar la página principal. Además, es una representación con un único valor que cambia a lo largo del tiempo por lo que no exige gran complejidad. Se ha utilizado una librería externa que permite la renderización de este tipo de gráficos.

Gráfico circular de sectores

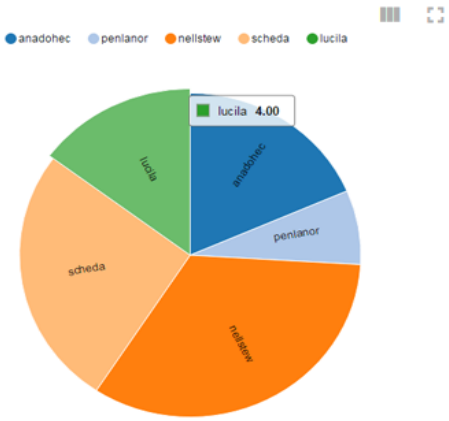


Figura 23: Gráfico circular de sectores

Como se puede ver en la Figura 23 se utilizará un gráfico circular de sectores para representar el número de *Defects* que un desarrollador tiene asignado, así, de un simple vistazo, se puede observar la carga de trabajo de cada uno.

En la vista de tendencia se podrá ver el número de defectos que resuelve cada desarrollador con respecto del tiempo y cuándo esos defectos le han sido asignados.

2.2.7 Seguridad

La aplicación no necesitará una seguridad exhaustiva ya que será de uso abierto para todos aquellos que estén en la red de Adidas. Sin embargo, requiere autentificación porque los administradores de cada área serán los encargados de realizar modificaciones a éstas, como se ha explicado en la sección 2.1.3, en la parte de configuración de proyectos.

La autentificación se envía al servidor provisto por Adidas, como el resto de llamadas. Sin embargo, esta autentificación no se envía allí sino que se redirige al servidor LDAP. Allí se guardan todas las credenciales de Adidas, y éste será el encargado de garantizar o no el permiso. No será necesaria la funcionalidad de registrar usuarios ya que solo se pueden usar estas credenciales. Se habilitará la función de *remember me* que, utilizando los *cookies* del navegador, será posible mantener la sesión abierta durante un periodo determinado de tiempo.

2.3 Implementación

En esta sección se pretende dar los detalles de implementación más característicos y de mayor importancia. Se presta gran atención a como las gráficas han sido generadas, la integración del sistema con la aplicación de *backend* y la ejecución de la seguridad.

El generador Yeoman proporcionó una estructura de archivos muy conveniente para desarrollar la aplicación Web basada en AngularJS, aun así se refinó un poco para cumplir con las convenciones de nombres de Adidas. En el directorio raíz tenemos varios ficheros y su posterior división:

- **App:** Se encuentra el código de la aplicación web. Dentro de esta carpeta hay subcarpetas para los distintos tipos de funciones que existen en AngularJS: controladores, directivas, servicios, constantes… También hay carpetas para las librerías y los ficheros HTML.
- **Dist:** Directorio donde se alojará el paquete creado en proceso de compilación para ser desplegado en el entorno de producción.
- **Build:** Directorio donde todas las tareas de automatización de Grunt se alojan.
- **Doc:** Directorio donde se generará la documentación automáticamente.
- **Test:** Aquí están implementados los test unitarios. Más información en el apartado 2.4.1.

En esta aplicación existe un “mock server”. Es un servidor (en este caso creado con Express, librería de NodeJS), el cuál simula los recursos que la aplicación recibirá.

Esta funcionalidad permite que el proceso de desarrollo de software del *frontend* sea independiente de un servidor *backend* o de servicios externos como pueden ser Sonar o Jira. Además, permite forzar fallos y distintos tiempos de respuesta para que la aplicación web soporte errores inesperados.

2.3.1 Generación de gráficas

Gráficas de barras

Estas gráficas han sido desarrolladas desde cero con D3.js. Una de las mayores dificultades fue dar un estilo adecuado con el resto de la página web. Esta gráfica ha ido evolucionando y mejorándose a lo largo del tiempo, como se puede observar comparando la primera versión mostrada en la Figura 24 con la versión definitiva.

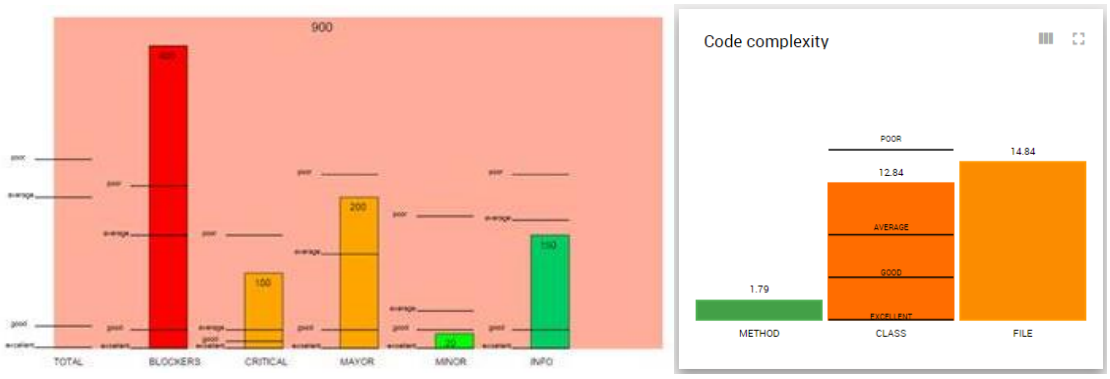


Figura 24: Gráfica de barras inicial vs versión definitiva

Además, existe la funcionalidad, como se puede ver en la Figura 25, de visualizar o no determinadas columnas. Al realizar esta acción la proporción entre las columnas sigue siendo la misma. Esta funcionalidad no era requerida desde negocio, sin embargo, se añadió al haber casos como en la propia Figura 25, en el que los valores de alguna columna ascienden por encima de mil y debido a esto, la visualización de los umbrales en otras columnas de gran importancia (en este caso Critical), era imposible. Otra decisión que se probó para solventar este problema fue hacer que la altura de las columnas sea con proporción logarítmica. Con ello se solventaba el problema de los umbrales, pero aparecía otro más grave, que provocaba al usuario confusión a la hora de entender la gráfica. La diferencia entre una columna y otra no era muy significativa, mientras que en realidad sí debería variar en gran medida.

Figura 25: Despliegue de selección de columnas

Velocímetro

Al igual que la gráfica anterior, esta gráfica porcentual ha sufrido modificaciones en cuanto a implementación a lo largo del tiempo. Su desarrollo se puede apreciar en la Figura 26.

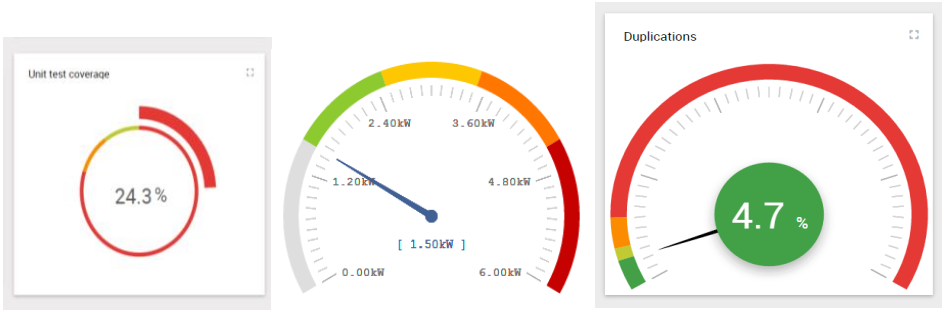


Figura 26:Evolución del velocímetro

Para esta gráfica se utilizó una librería externa (ngRadialGauge [22]) que se personalizó. Se le añadió un tooltip, se borraron los números y se añadió un círculo central para marcar el estado de la métrica. En la Figura 26 se puede ver la evolución sufrida del gráfico a lo largo del tiempo.

Una de las principales razones de la evolución de este gráfico fue debido a que la gráfica no era intuitiva y su comprensión era ambigua. Los umbrales corresponden al círculo interior y el valor será el círculo exterior. Sin embargo, esta gráfica presentaba ciertos problemas:

- Debido a que Duplications sigue unos umbrales inversos, confundía en gran medida a los usuarios.
- Como se puede ver en la gráfica de la izquierda de la Figura 27, hay un 0% de Documented APIs, lo que es muy negativo, sin embargo, no se aprecia visualmente.

Por estas razones se buscó una alternativa que sigue respetando los estilos y los usuarios la entienden con mayor facilidad.

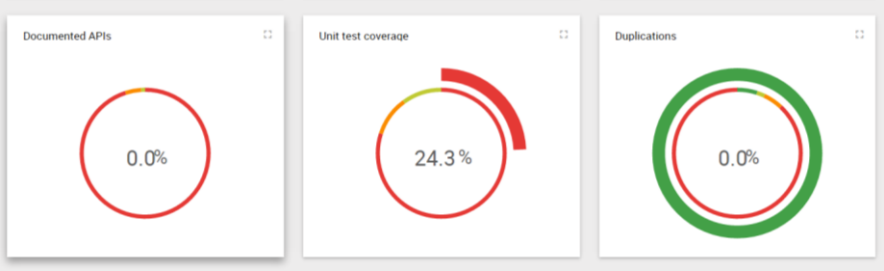


Figura 27: Primera versión de la gráfica de porcentajes

Gráfico de líneas

Para estos gráficos, como el de la Figura 28, se ha utilizado la librería **angular-nvd3**. Ésta permite la creación de diagramas utilizando un sencillo objeto de configuración. En esa misma figura se observa el JSON modelo.

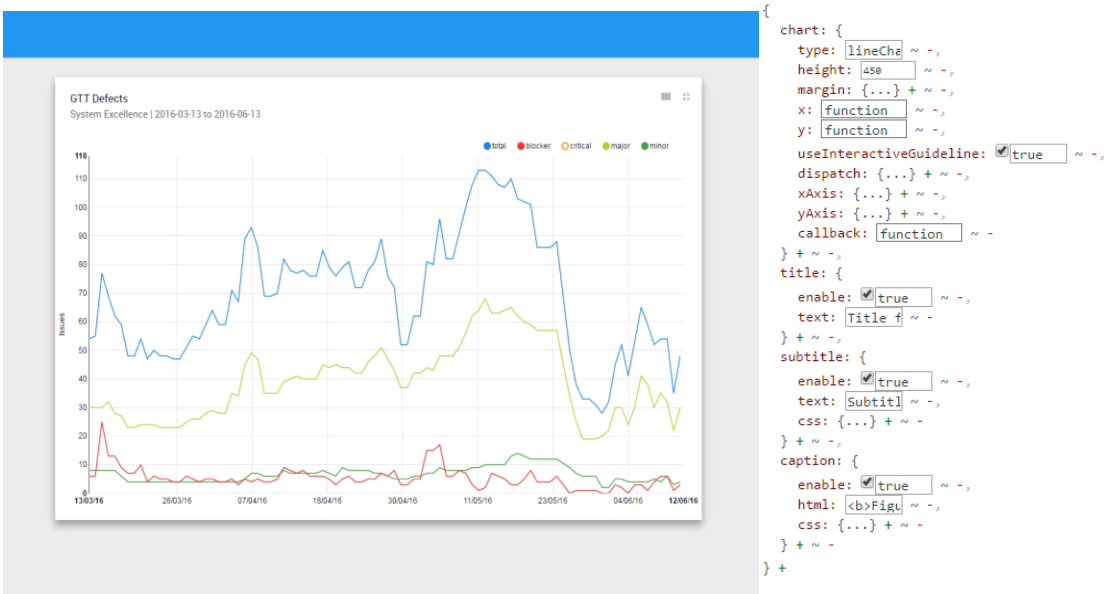


Figura 28: Gráfico de líneas

El principal problema con estos gráficos es el rendimiento. Se están intentando representar 1Mbyte de información por gráfico, en algunos casos más, si el rango de fechas es mayor. Debido a esto, el explorador está procesando toda esta información y requiere un tiempo mayor de carga que para la vista de estado. Sin embargo, este tiempo de espera no es lo suficientemente grande para que el usuario pierda interés y se distraiga.

Gráfico circular de sectores

Implementada también con angular-nvd3. Una vez conocida la librería no supuso ningún problema su implementación e integración gracias a la modularidad de la arquitectura utilizada.

2.3.2 Integración del sistema

Los pasos a seguir para generar una gráfica son comunes. Como se puede ver en la Figura 29, primero se recibe información desde el servidor, ésta (según su tipo) es formateada desde la aplicación web de forma que se puede aplicar a los distintos programas y librerías que utilizamos para su representación. Cada gráfica tiene su propio controlador AngularJS, el cual contiene la información de éstas y maneja las diversas funcionalidades que tiene cada gráfica (esconder columnas, mostrar leyenda, ampliar, etc.).

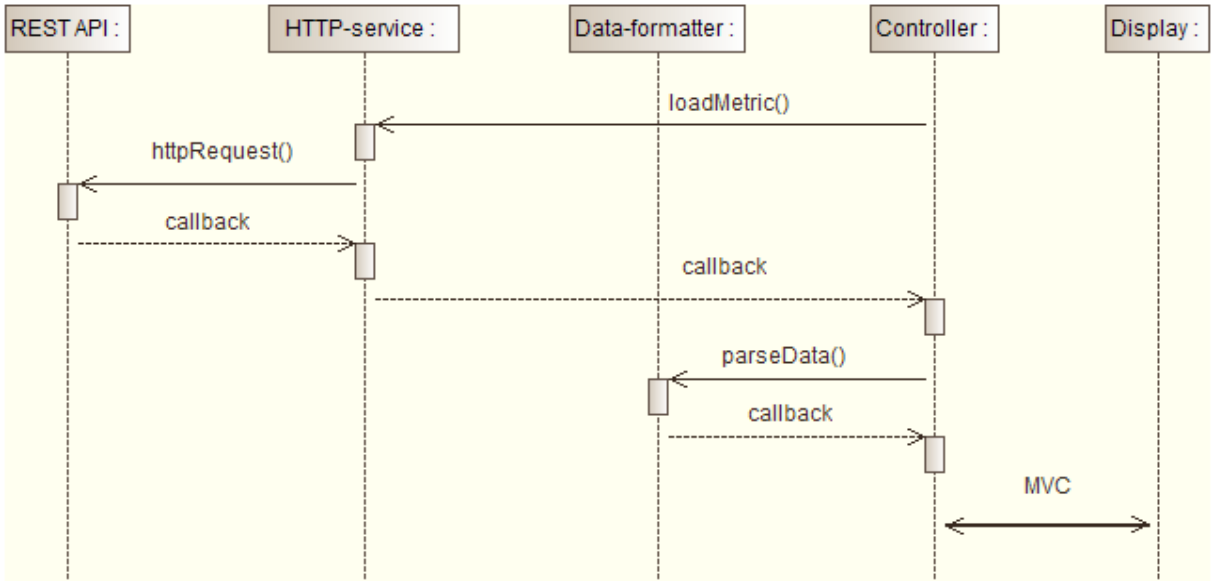


Figura 29: Flujo para mostrar una gráfica

Como se ha explicado en la arquitectura, este sistema empieza accediendo directamente a las APIs que ofrecen Sonar y Jira. Posteriormente se añade un servicio de *backend*, por lo que se necesitará acoplar estos dos sistemas.

Se llegó a un acuerdo entre ambas partes para definir las direcciones Web a las que la aplicación iba a acceder para obtener el recurso. Es decir, la REST API que el servidor *backend* iba a exponer.

También hubo consenso sobre qué estructura iban a tener la información provista por cada "endpoint", es decir por cada dirección web.

Figura 30: Estructura estándar de transmisión de datos

Como se puede ver en la Figura 30 el objeto recibido es un objeto de tipo JSON, un tipo de datos estándar en las comunicaciones web.

Su estructura es clara. Un campo para comprobar si hay un error y un contenido en el cual, de forma genérica, se recibe la información principal como campo obligatorio. Como campo opcional se reciben los umbrales calificativos de esta información. Gracias a la consistencia de esta información es posible recibir información de distintas métricas y añadir nuevas métricas con facilidad y sin tener que hacer grandes cambios al tratamiento de datos.

Finalmente se llegó a un acuerdo de versionado de recursos, tanto de direcciones web como de estructura de información. Así, con un acuerdo previo entre *backend* y *frontend,* era posible acceder a una configuración previa compatible entre ambos.

2.3.3 Seguridad

Para la autentificación de usuarios se utilizará JSON Web Token (**JWT** [23]), es un estándar abierto (RFC 7519) que define una forma compacta y contenida de asegurar la transmisión de información entre distintas partes a través de objetos JSON [24]. Esta información puede ser verificada y de confianza porque está firmada digitalmente. Estos Tokens se pueden firmar usando una clave secreta (utilizando un algoritmo HMAC) o con una clave pública/privada utilizando RSA.

Las principales características de JWT son las siguientes:

- **Compacto:** gracias a su pequeño tamaño, JWT puede ser enviado a través de un parámetro URL, POST o en un header HTTP. Además, su transmisión será más rápida al ocupar menos.
- **Auto-contenido:** en el JWT está contenida toda la información relevante sobre el usuario, por lo que evita hacer varias búsquedas dentro de la base de datos.

Se van a utilizar JWTs solamente para la autentificación del usuario (uso más común), aunque también se pueden utilizar para el intercambio de información. Una vez que el usuario es autenticado, en cada llamada al *backend* será incluido un JWT. Así el *backend* podrá verificar si ese usuario tiene permisos para realizar la acción requerida. La única acción que requerirá esta verificación será la actualización de la configuración de un área o la creación de una nueva.

Se puede encontrar más información sobre la estructura de JWT en el anexo 10.

2.4 Pruebas del sistema

En este apartado se va a explicar los distintos test que se han realizado para asegurar el funcionamiento completo de la aplicación Web. Estos son indispensables para asegurar el desarrollo de una aplicación de calidad como predican los estándares actuales de Adidas.

2.4.1 Enfoque

Se puede ver en la pirámide izquierda de la Figura 31 lo que sería el enfoque tradicional, el cual consiste en un gran porcentaje de pruebas funcionales, en torno al diez por ciento de integración (los sistemas se acoplan unos con otros correctamente) y apenas test unitarios. Todas estas pruebas están hechas por personal dedicado esencialmente a la prueba de aplicaciones.

Como se puede ver en la pirámide derecha de la citada figura, el enfoque ideal es aquel en el que todas las pruebas están automatizadas. Se priorizan los test unitarios sobre los demás. Gracias al mayor número de éstos, los test funcionales son menos importantes. Los test de integración se dividen en componentes y APIs, lo que permite abstraerlos en mayor medida.

Como se ha explicado en los apartados 2.4.1 y 2.4.2, se sigue el modelo de la pirámide ideal ya que, tanto los test unitarios como los funcionales, están automatizados. Sin embargo, no se han automatizado los test de integración que se han hecho manualmente. Automatizar éstos sería una funcionalidad requerida para el futuro.

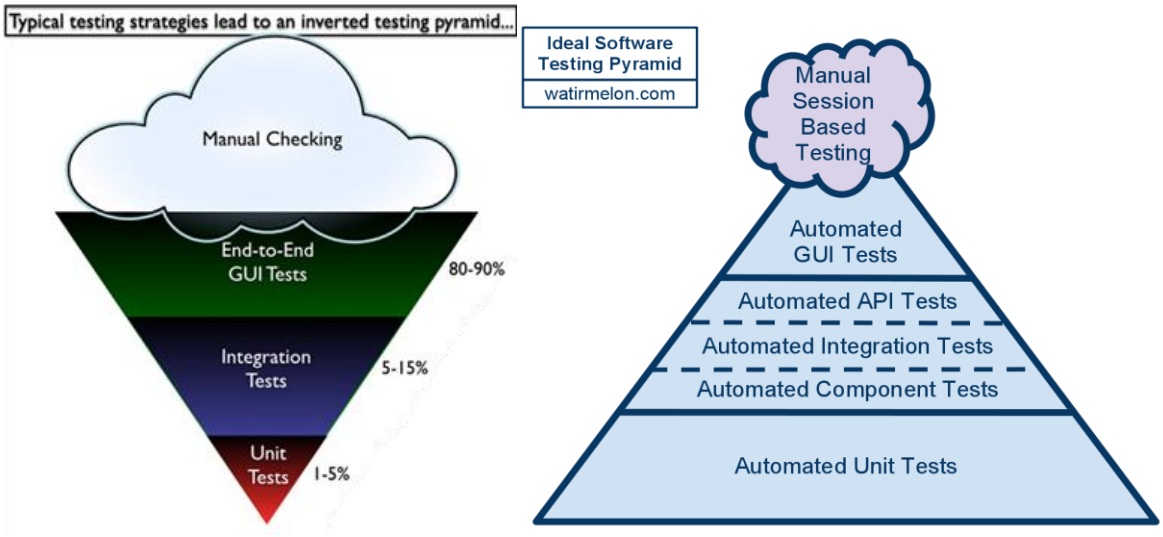


Figura 31: Pirámide tradicional vs pirámide ideal de testing

2.4.2 Test unitarios

Como prueba de sistema se aplican test unitarios. Es una forma de comprobar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Se deben probar todos los casos en los que el método se pueda llamar, incluido los casos de error. Para los casos en los que se requiera una llamada a un recurso externo se utilizará la librería AngularJS-Mock [4] para asumir que la llamada se ha realizado

Para los test unitarios se ha utilizado Mocha: es una herramienta que permite correr de forma asíncrona test unitarios en Node.js y en distintos exploradores. El utilizado es PhantomJS [25] un explorador sin interfaz gráfica de usuario y con soporte nativo para distintos estándares Web como manejo del DOM, selección de CSS, JSON y lo más importante para la aplicación, SVG.

2.4.3 Pruebas funcionales

Los test funcionales son un proceso de control de calidad. Son una caja negra que basa sus casos de prueba en las especificaciones de software que se deben cumplir. Estos test se realizan introduciendo al programa una entrada y observando una salida. Describen lo que el sistema hace, no comprueban la funcionalidad de un método o clase, sino una parte de la funcionalidad del sistema completo.

Aplicándose a este proyecto consistiría en pulsar un botón y observar que la salida es correcta. Concretamente, al pulsar en un botón de esconder una columna del gráfico de barras se observa que, en efecto, desaparece una.

Se pueden automatizar comprobando la salida a través de cambios en el DOM.

Se ha utilizado la herramienta “Protractor” que permite esta automatización tanto generando la entrada como comprobando la salida. **Protractor** [26] es un sistema de test para aplicaciones AngularJS, sin necesitar complejas configuraciones. Ejecuta estos test contra la propia aplicación en navegadores de internet reales haciendo la misma interacción que un usuario haría utilizando eventos. Como funcionalidad a destacar, Protractor espera a que el test anterior haya acabado para empezar el siguiente, permitiendo al usuario no tener que encargarse de programar este comportamiento.

2.4.4 Pruebas de aceptación

Gracias a la metodología usada, Scrum (explicada en el apartado 3.1), al final de cada *sprint*, en la reunión correspondiente el Project Owner (el cliente) revisa la aplicación en una demostración comprobando que todos los requisitos son cumplidos. Estas pruebas de aceptación no están automatizadas pero se hacen periódicamente.

2.5 Despliegue, arranque y aceptación

Este proyecto va a seguir el modelo de ***Continous Integration*** (CI) o integración continua. Este proceso consiste en hacer integraciones automáticas de un proyecto lo más frecuentemente posible para así detectar fallos cuanto antes y además tener la aplicación en producción lo más actualizada posible. La herramienta utilizada en el proyecto es TeamCity, la cual facilita un conjunto de agentes distribuidos que proceden a hacer distintas tareas computacionales. Se puede encontrar su configuración y más información en el anexo 14.

A continuación se va a explicar el proceso de integración continua seguido en este proyecto y representado en la Figura 32:

1. Cuando una tarea de desarrollo se finaliza se crea una "pull request" en Stash donde será revisada.
2. Cuando esta revisión sea completada con éxito, la tarea será aprobada y es cuando empieza el proceso de integración. Además, se lanzará un servicio Sonar que analizará la última versión del código.
3. La pull request será incorporada a la rama de desarrollo. Un disparador será lanzado en la herramienta de TeamCity.
4. TeamCity asignará un agente que realizará las siguientes tareas:
    a. Ejecutar la tarea *linter*.
    b. Lanza los test unitarios.
    c. Llama a un cliente Sonar que evaluará el estado de la aplicación.
    d. Empieza el proceso de *build* o empaquetado. En este proceso, el código de la aplicación Web se introduce en un paquete lo más compacto posible para que su rendimiento dentro de un explorador de internet sea lo más eficiente posible. A continuación se detallan qué tareas Grunt se ejecutan.
        i. Compilar Less en CSS.
        ii. Mueve las imágenes, fuentes y traducciones al directorio "dist".
        iii. Convierte los ficheros HTML en JavaScript para poder ser introducidos en cache.
        iv. Minimiza [27] tanto ficheros CSS como JavaScript para que ocupen el menor tamaño posible. Para ello se eliminarán espacios, se cambiarán nombres de variables para acortarlos…
        v. Cambia el nombre de los ficheros para que no sean cacheados en el servidor de destino.
        vi. Todos los ficheros generados serán desplazados al directorio "dist".
5. Si todo ha ido como es esperado se copiará el paquete generado a través del protocolo SFTP al servidor *frontend*.
6. Dependiendo del entorno en el que se esté desplegando, se copiará un fichero "endpoints" con las direcciones IP a las que la aplicación web accederá para obtener los recursos.

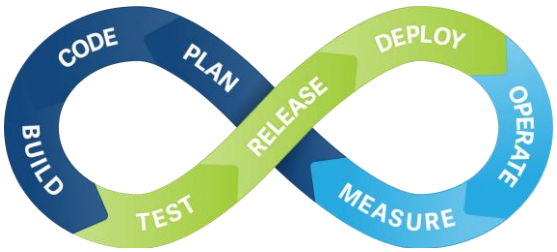


Figura 32: Flujo de integración continua

2.6 Rendimiento y mejoras

La aplicación presenta un rendimiento aceptable en cuanto a la carga y visualización de datos gracias a las mejoras aplicadas. Si un dato está tardando más de lo esperado en cargar un *spinner* aparecerá, mientras tanto, la aplicación sigue siendo funcional gracias a las llamadas asíncronas. El tratamiento de éstas consiste en enviar la petición. La aplicación funcionará con normalidad y recibirá una alerta cuando la respuesta de la petición llegue y ésta será tratada. Los exploradores permiten unas seis llamadas paralelas simultáneas [28]. Además, el formateo de datos será también asíncrono evitando que la aplicación se bloquee cuando ésta reciba los datos, que en algunos casos puede ser de manera simultánea.

Es importante tener esto en cuenta porque se hace una llamada HTTP asíncrona para cada métrica. En las vistas de estado actual y tendencia no hay problema porque se hacen unas diez llamadas, sin embargo, es un ligero problema en la página de inicio donde el número de llamadas es de diez por proyecto. Por cada área de proyectos se hará una llamada por métrica configurada. En un futuro cercano se espera unificar en una única llamada, al menos para la página de inicio específicamente. Además para mejorar el rendimiento de cada petición se ha hecho en el lado del servidor una compresión gzip [29], que consiste en minimizar todo lo posible el tamaño de cada respuesta para reducir así el tiempo de envío a través de la red. Se ha llegado a conseguir un segundo y medio en el tiempo de espera al recibir los datos de un área en concreto.

Otra de las mejoras introducidas fue habilitar la *cache* de las llamadas XMLHttpRequest para la obtención de los objetos de las métricas. Así, si los datos requeridos en la vista de estado no serán llamados a través de la red de nuevo puesto que estarán en la memoria *cache,* ya que eran necesarios en la vista principal. Esta funcionalidad permite una navegación mucho más eficiente ya que dispone de todos los datos de antemano.

Finalmente una de las últimas mejoras en introducirse que agilizó en gran medida el tiempo de carga fue la forma de actualizar el modelo de datos en memoria. El modelo anterior actualizaba la lista completa de métricas cada vez, mientras que el nuevo sólo actualiza aquellos índices que habían cambiado permitiendo no volver a dibujar de nuevo las gráficas cuyas métricas no han sido modificadas.

En las siguientes imágenes se observa el **tiempo de carga** de las vistas principales. Como se detalla aquí [30] se debe poner especial interés en las dos barras verticales:

- Carga del DOM (azul): Indica cuando los elementos del DOM han sido cargados. A partir de este momento el usuario puede interactuar con la aplicación a pesar que los datos aún no han sido cargados.
- Load (roja): Indica que se han cargado los recursos como imágenes, fuentes, estilos…

Las líneas horizontales indican las solicitudes realizadas. Cada segmento horizontal es una. Como se puede observar hay un máximo de seis paralelas (el máximo que permite Chrome que es el explorador donde se han realizado las pruebas). Los datos (métricas) habrán cargado cuando todas las peticiones hayan sido procesadas.

Las peticiones tienen distintos colores, los cuales indican distintos sucesos:

- Esperando (gris): tiempo en el cual la petición ha esperado a ser enviada.
- DNS (azul verdoso): Si la dirección no está cacheada no aparece.
- Conectando (naranja): Establecimiento de la conexión con el servidor.
- Enviando y esperando la respuesta (verde).
- Descargando contenido (azul).

En estas llamadas se vuelven a cargar todos los recursos. Una vez cargados, cuando se navega por la aplicación, éstos están en memoria y sólo se cargarán los datos de las métricas para ser mostradas, reduciendo en gran medida el tiempo de carga, como se puede ver en la Figura 36.

Como se puede ver en las siguientes imágenes y en el anexo 18, esta aplicación Web, a pesar del gran tamaño de los datos que tiene que cargar, presenta unos tiempos de respuesta más que aceptables haciendo que su manejo sea ágil.

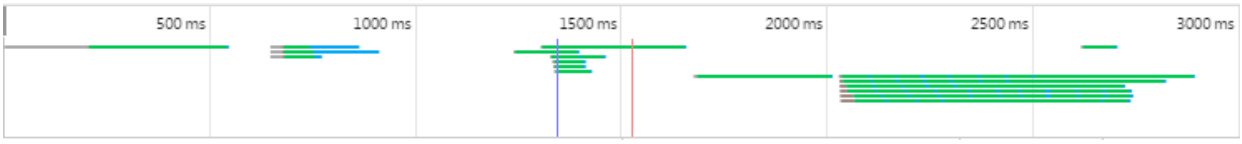


Figura 33: Tiempo de carga página principal (58 request)

Figura 34: Tiempo de carga vista de estado actual (29 request)

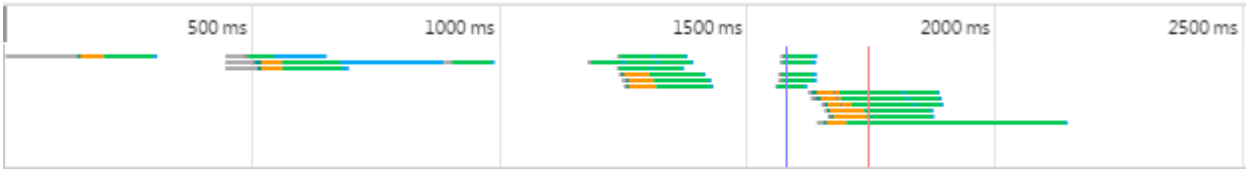


Figura 35: Tiempo de carga vista de tendencia (29 request)

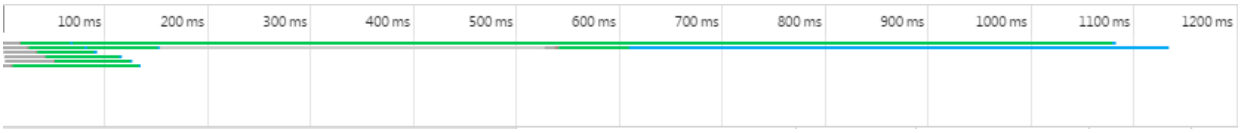


Figura 36: Carga de datos sólo (9 request)

3 Gestión del proyecto

Dentro del enfoque *Agile* explicado en el apartado 1.4.3 se ha seguido la metodología **Scrum** [31] ya que es un estándar que la compañía está empezando a utilizar en sus proyectos más novedosos. En el anexo 8 se puede ver el tiempo y esfuerzo empleado a lo largo del proyecto.

En este proyecto se ha puesto especial interés en esta metodología, la cual se ha seguido con la mayor precisión posible. Dentro de la empresa Adidas, este proyecto está siendo uno de los primeros que utiliza Scrum, sirviendo como precedente y asentando las bases. Algunas veces esta exhaustividad repercute en cuanto al empleo de mayor tiempo del deseado en las reuniones pero se obtiene una mejor organización y gestión además de ser un precedente para otros proyectos.

3.1 Metodología

Scrum es un proceso de administración y control que siguiendo el enfoque *Agile*, acaba con la necesidad de tener unos requisitos iniciales completos. Los equipos son capaces de recibir requisitos a lo largo del tiempo y proveer software de calidad y funcional cada ciertos periodos de tiempo.

La metodología Scrum permite una colaboración efectiva entre miembros de un equipo en proyectos software complejos.

Para entender esta metodología vamos a dividirla en tres partes: artefactos, roles, y reuniones.

3.1.1 Artefactos

En la Figura 37 podemos ver un *backlog* que es una acumulación de tareas incompletas esperando para realizarse. Su principal función es llevar un seguimiento de cada una de las tareas, si está en desarrollo, si se está siendo revisada, si ya está hecha o si todavía no se ha empezado.

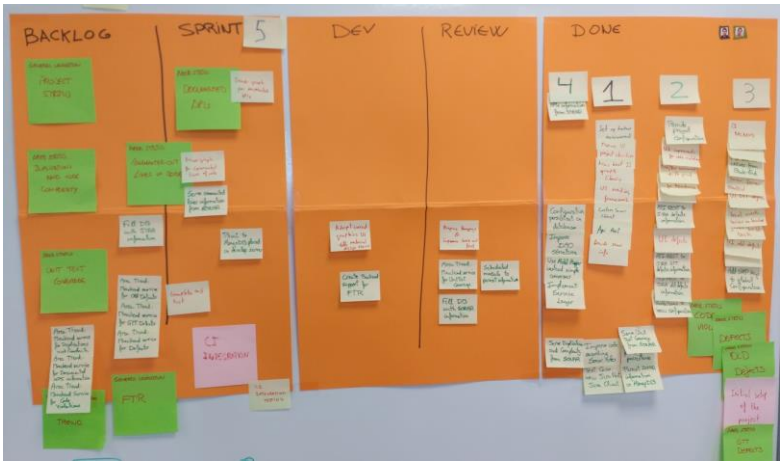


Figura 37: Backlog del sprint 5

En esta metodología habrá dos tipos de backlog

- Product backlog: Es el backlog con todas las tareas por hacer que han sido documentadas. Al ser *Agile*, se puede añadir tareas, ya que el scope es variable. Estas tareas se pueden clasificar en tres tipos.
    - Épica: Esta tarea viene definida por el cliente. Es una tarea a grandes rasgos en la que no está definido nada técnico, sino el objetivo a conseguir a alto nivel. A partir de una épica se crearán distintas User stories.
    - User story: Son tareas derivadas de una épica. Son consideradas el requerimiento técnico mínimo. Según la complejidad son estimadas con los denominados "story points". La resolución de todas las user stories asignadas a una épica indicará que ésta ha sido finalizada.
    - Tarea: Considerada como las tareas de deuda técnica que no añaden nueva funcionalidad, sino que mejoran partes del código previamente desarrolladas. La refactorización, los test unitarios y la corrección de bugs entrarían en esta categoría.
- Sprint backlog: Es el backlog con las tareas a realizar durante el intervalo de tiempo designado. El número de tareas introducidas en este backlog dependerá de la capacidad del equipo encargado en desarrollo. Esta capacidad es medida en "story points" y es variable. Aumentará con la experiencia de los desarrolladores y el número de éstos.
- Product increment: son la lista de tareas que han sido completadas durante el sprint actual y todos los sprints anteriores.

3.1.2 Roles

- **Product Owner (PO):** Es la persona de contacto con el cliente.
- **Scrum Master (ScM):** Es la persona encargada de ser mediador de las reuniones. Es el punto de unión entre el equipo de desarrollo y el PO.
- **Equipo de desarrollo:** Equipo encargado de desarrollar la aplicación. En la metodología Scrum no hay un jefe de proyecto como en el desarrollo tradicional, al contrario, cada miembro del equipo es el encargado de crear y asignar tareas correspondientes a una épica creada por el PO, por lo que los integrantes tienen una mayor percepción de propiedad del producto que están desarrollando.

3.1.3 Reuniones

Scrum es una metodología en la que se realizan reuniones con bastante frecuencia. Todas estas reuniones están prefijadas en el tiempo para promover que se aborden los aspectos importantes y no se entre en materia de implementación.

Debido a la asiduidad de las reuniones es recomendable que el equipo Scrum esté centralizado geográficamente.

Hay distintos tipos de reuniones cuya sucesión se puede ver en la Figura 38:

- **Daily scrum:** Se realiza diariamente, con una duración aproximada de 15 minutos, una reunión en la que cada desarrollador comenta al resto qué tareas ha realizado, la tarea que está en proceso de desarrollo y la próxima tarea que tomará. Son reuniones que se deberían hacer de pie, ya que favorece la agilidad de éstas y en las que no se deben tomar decisiones importantes.

- **Sprint planning - Parte 1:** Antes de comenzar el sprint. Duración de una hora. El equipo se reúne con el PO, el cual explicará si ha habido nuevos requisitos o un cambio de los anteriores.
- **Sprint planning - Parte 2:** Justo después de la parte 1, con los requisitos dados por el PO, el equipo se dispone a crear User Stories para cumplirlos. Se deberán estimar estas tareas según su complejidad con story points por parte de todo el equipo. Finalmente, se decidirá qué tareas serán incluidas en el sprint teniendo en cuenta la capacidad del equipo. La duración debe ser de una hora.
- **Review meetings:** Al cierre del sprint, el equipo se reúne con el PO y le enseñan una demo sobre lo realizado durante el sprint. El PO da su opinión sobre la demo e indica si se han cumplido con los requisitos. Además comenta al equipo que es lo que más le gusta y qué mejoras o cambios aplicaría.
- **Retrospective meetings:** Después de la demo, el equipo se junta y se hace crítica sobre la aplicación de la metodología scrum en el sprint. Se da la opinión sobre lo que ha podido afectar negativamente al desarrollo, como por ejemplo que las tareas han sido mal estimadas y por eso no se ha podido entregar todo lo que fue propuesto en el sprint planning.

Figura 38: Ciclo de vida de Scrum

En el proyecto realizado se ha seguido esta metodología de forma intensiva. Donde el PO corresponde con el equipo de GSD de Adidas Group, quien era el encargado de establecer los requisitos a alto nivel (ej. información a representar) así como dar su opinión y posibles mejoras al cierre del sprint.

Mis tareas en cuanto a metodología, consistían, a partir de los requisitos a alto nivel (creados como épicas en el backlog), en crear las user story correspondientes para alcanzar los objetivos y estimarlas junto con el equipo.

La estimación (en story points) se ha hecho siguiendo la serie de Fibonacci ya que las tareas menos complejas se pueden estimar con más precisión que las que son de complejidad mayor (normalmente se estima entre 1 y 8).

Todas las tareas son guardadas en el backlog para permitir su seguimiento. Aparte de un backlog en papel se ha utilizado la herramienta Jira que pertenece a la empresa Atlassian. Es muy potente y permite guardar las distintas tareas con las distintas jerarquías que existen en Scrum. Además permite dar prioridad a éstas y asignarlas a distintos desarrolladores. Finalmente, provee información sobre el rendimiento del proyecto, así como un estado general. Se pueden encontrar capturas de pantalla sobre Jira en el anexo 12 tanto sobre rendimiento como estado actual del proyecto. Por supuesto, Jira permite la funcionalidad básica de mostrar el estado actual de cada tarea dentro del sprint (En desarrollo, revisando y hecha).

Una vez creadas las tareas y estimadas, según mi capacidad, se asignan al sprint. Los sprints son de dos semanas. Todas las tareas creadas que no son incluidas al sprint se quedan en el product backlog. Durante el sprint, es posible crear nuevas tareas durante el análisis de cada tarea. Éstas se añadirán al product backlog automáticamente, podrán ser incluidas en el sprint si es necesario aunque no es buena práctica, ya que se han incluido las tareas justas para ser completadas según la capacidad del desarrollador.

La capacidad es variable respecto al tiempo, ya que cada desarrollador obtiene experiencia del proyecto. Empecé con una capacidad de 6 story points y actualmente la he duplicado.

El **ciclo de vida de una tarea** comienza en el backlog al ser creada. Posteriormente se estima junto con todo el equipo. Una vez introducida en un sprint se procede a su análisis y posterior desarrollo. Una vez terminado éste, se crea una pull request en git donde será revisada (más información en el apartado 2.5). Se solucionarán los problemas encontrados en la revisión si los hay y se calificará la tarea como hecha una vez que se ha mergeado. Finalmente la tarea se archivará cuando el PO dé el visto bueno al cumplir los requisitos.

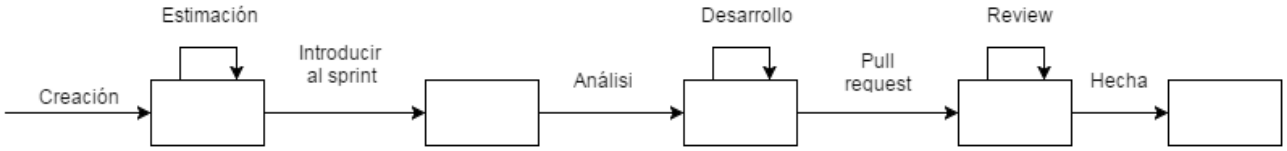


Figura 39: Ciclo de vida estándar de una tarea

En la Figura 39 se observa el ciclo normal de una tarea. Sin embargo, pueden darse otros casos en los que una tarea sea incluida a mitad de un sprint o salga, ya que éste ha sido estimado de forma errónea. También se puede dar el caso de la aparición de un *bug* con prioridad que deberá introducirse en el *sprint*.

4 Conclusiones

Después de cuatro meses de trabajo a tiempo completo dedicados al análisis, diseño e implementación de la aplicación se ha podido llegar a una versión estable, que ya está siendo configurada para algunos proyectos de GSD con expectativas de alcanzar otros departamentos. El balance es positivo, ya que se ha cumplido el principal objetivo de desarrollar una herramienta capaz de controlar la calidad de código de distintos proyectos en los plazos fijados.

4.1 Resultados

Como se ha visto a lo largo de esta memoria, se ha realizado una aplicación web cumpliendo con los requisitos dados por negocio, en este caso el grupo de GSD de Adidas. Gracias a esta aplicación, se facilita el control de todos los proyectos de este equipo. Además se espera extenderla a otros grupos de la compañía para controlar sus propios proyectos. La web ya está en funcionamiento en fase beta donde ha sido introducida en los proyectos más recientes y se ha incluido de base en aquellos nuevos.

La información mostrada sobre desarrolladores es limitada pero suficiente para observar una visión general sobre el estado actual de cada uno y su evolución.

Las opiniones recibidas de los jefes de proyecto son positivas ya que las métricas mostradas son precisas y permiten ver los avances de un proyecto de un vistazo. Además la configuración es intuitiva ya que tienen amplios conocimientos de Sonar y Jira.

Los desarrolladores también han transmitido su aprobación ya que les permite tener una visión general del proyecto que durante el desarrollo es difícil tener. Además, gracias a esto, pueden comprobar de primera mano que los esfuerzos realizados están teniendo sus frutos fomentando la percepción del trabajo producido.

Próximamente la aplicación será presentada al CEO de la rama de IT, donde se espera que cause un buen impacto y se impulse su utilización en distintos ámbitos de monitorización, así como ponerla en grandes pantallas que están distribuidas a lo largo de los departamentos.

Es una aplicación propia de Adidas, sin embargo se puede aplicar a cualquier proyecto software que use Jira como gestor de tareas y Sonar como medidor de calidad de código. Además no importa el estado de madurez, ya que la aplicación permitirá acceder a datos antiguos de las herramientas de métricas.

Se puede observar claramente la aplicación en el apartado de navegación y en el anexo 11 de gráficas. Se ha puesto detalle en que los estilos queden consistentes unos con otros.

4.2 Trabajo futuro

A corto plazo se espera incluir métricas relacionadas principalmente sobre los desarrolladores. La versión dos tiene como fecha de entrega finales de agosto y se piensan entregar las siguientes métricas con sus correspondientes gráficas:

- **Estimación / Realidad:** Se mostrará la relación entre la estimación de una tarea y el tiempo real empleado en resolverla. De esta forma se podrá calificar la calidad de la estimación.
- **Velocidad / Desarrollador:** Consistirá en la cantidad de tareas proporcionales a su estimación que cada desarrollador ha finalizado.
- **Líneas de código / Capacidad:** Proporcionalidad entre las líneas de código que un desarrollador escribe con respecto a su capacidad asignada.
- **Rendimiento desarrollador:** Contando con distintas métricas como la velocidad, las líneas de código y la estimación se generará un porcentaje correspondiente a su rendimiento.
- **Calidad de pull request:** Derivado del número de comentarios y análisis de sonar se obtendrá un porcentaje.
- **Revisión de pull request:** Según la participación del desarrollador en cuanto a comentarios y velocidad en evaluar una pull request se obtendrá un porcentaje.

Un aspecto a tener en cuenta es que la aplicación es totalmente escalable y puede tener *n* métricas. Para dar este soporte deberá ser necesario habilitar funcionalidad de no mostrar las gráficas no deseadas, la cual no será de gran extensión.

La automatización de test de integración sería recomendable para mejorar el modelo de integración continua, aunque sería una tarea de baja prioridad.

Finalmente, a largo plazo se podría añadir distintas funcionalidades de predicción utilizando técnicas de aprendizaje automático. Esta aplicación dispone de gran cantidad de datos que podrían utilizarse para alimentar una red neuronal para predecir, gracias al entrenamiento previo realizado, la evolución esperada de un proyecto, dada una tendencia o la curva de rendimiento esperada de un desarrollador, gracias a sus estadísticas previas.

Estas nuevas tecnologías siguen en desarrollo pero ya empiezan a aparecer en distintos ámbitos. Existen librerías en JavaScript que nos permiten realizar estos entrenamientos incluso disponible en el explorador como es el ejemplo de **convnets** [32].

Con estas predicciones sería más sencillo estimar la deuda técnica de un proyecto o el número de desarrolladores necesarios para cumplir ciertas tareas.

4.3 Lecciones aprendidas

Este proyecto me ha introducido en el mundo del desarrollo software. Sobre todo he aprendido como funciona una empresa internamente, algo que considero de gran valor.

Lo que me ha parecido más interesante es la metodología Scrum (se puede ver un seguimiento en el anexo 8 de gestión de tiempo) para desarrollar proyectos en el mundo moderno, en el cual los requisitos cambian constantemente y en algunos casos es imposible definir unos requisitos al principio. Es difícil imaginar cómo sería entregar un proyecto de grandes dimensiones sin que el cliente lo haya revisado periódicamente. Se puede encontrar en esta memoria que la metodología está altamente documentada, siendo una de las cosas a las que no me importaría dedicarme en un futuro con la correspondiente experiencia.

Otra cosa importante, que va ligado a la metodología *Agile*, es la integración continua. Algo que en mi opinión considero vital para el desarrollo adecuando de un proyecto y cuya configuración es compleja, sin embargo, una vez realizada, agiliza los procesos de test y de despliegues para el resto de desarrollo del proyecto.

Testing ha sido una de las habilidades que más me ha costado dominar y que más tediosa me ha parecido. Sin embargo, su existencia es clave para el desarrollo de calidad de un proyecto.

Finalmente y no menos importante, el uso de las librerías de gráficos, en concreto D3. Una librería muy potente que permite a bajo nivel, dibujar cualquier cosa en el navegador. Su curva de dificultad es prominente pero una vez dominado te permite manejar el DOM a tu antojo creando cualquier figura.

4.4 Conclusión personal

Lo más valioso que este proyecto me ha proporcionado ha sido el aprendizaje a través de su realización. Los inicios fueron difíciles, con una gran carga de lectura y de tutoriales. Sin embargo, conforme se va cogiendo experiencia y conocimiento en el proyecto, el avance ha sido notorio. Por supuesto, siempre existen problemas a lo largo del recorrido que se han superado con menor o mayor dificultad.

También considero muy valiosa la experiencia en un ámbito profesional en el que se cuidan los detalles en cuanto a la ejecución, donde todo es revisado dando resultado a proyectos de buena calidad.

Además, trabajar para un proyecto con aplicaciones y uso inmediato me ha ayudado a obtener una gran motivación sabiendo que será útil en un futuro y que servirá para mejorar la calidad de los proyectos software en Adidas.

5 Bibliografía

1. Jira https://www.atlassian.com/software/jira
2. Sonar http://www.sonarqube.org/
3. Stash https://www.atlassian.com/software/bitbucket/server
4. ECMAScript 5 http://speakingjs.com/es5/ch25.html
5. Javascript browser support http://www.javascripter.net/faq/browsers.htm
6. Angular mock https://docs.angularjs.org/api/ngMock
7. Bootstrap http://getbootstrap.com/
8. NodeJS HTTP https://nodejs.org/api/http.html
9. HTTP-server https://www.npmjs.com/package/http-server
10. Canvas http://html5doctor.com/an-introduction-to-the-canvas-2d-api/
11. SVG https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
12. Big Data https://www.mongodb.com/big-data-explained
13. Agile http://agilemanifesto.org/
14. Waterfall https://en.wikipedia.org/wiki/Waterfall_model
15. Generador Yeoman http://yeoman.io/generators/
16. Bower https://bower.io/
17. Npm https://docs.npmjs.com/getting-started/what-is-npm
18. Grunt http://gruntjs.com/
19. Gulp http://gulpjs.com/
20. HTTP AngularJS https://docs.angularjs.org/api/ng/service/$http
21. Query parameters https://perishablepress.com/how-to-write-valid-url-query-string-parameters/
22. Librería ngRadialGauge https://github.com/stherrienaspnet/ngRadialGauge
23. Seguridad utilizada https://jwt.io/
24. JSON http://www.json.org/
25. Explorador de testing http://phantomjs.org/
26. Protractor http://www.protractortest.org/#/
27. Minimizar https://github.com/gruntjs/grunt-contrib-uglify
28. Request paralelas http://www.browserscope.org/?category=network.
29. GZIP https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer?hl=en
30. Tiempos de la red https://developers.google.com/web/tools/chrome-devtools/profile/network-performance/resource-loading#view-network-timing-details-for-a-specific-resource
31. Scrum https://www.scrum.org/Resources
32. ConventJS http://cs.stanford.edu/people/karpathy/convnetjs/

6 Anexos

6.1 Diagrama de casos de uso

En este diagrama se muestran las distintas acciones que puede cumplir el usuario. Se observa que es necesario seleccionar una fecha para observar las distintas métricas para estado y tendencia y que se necesitan permisos de administrador para modificar la configuración de un área. Además, para crear o borrar un área es necesario ser súper administrador.

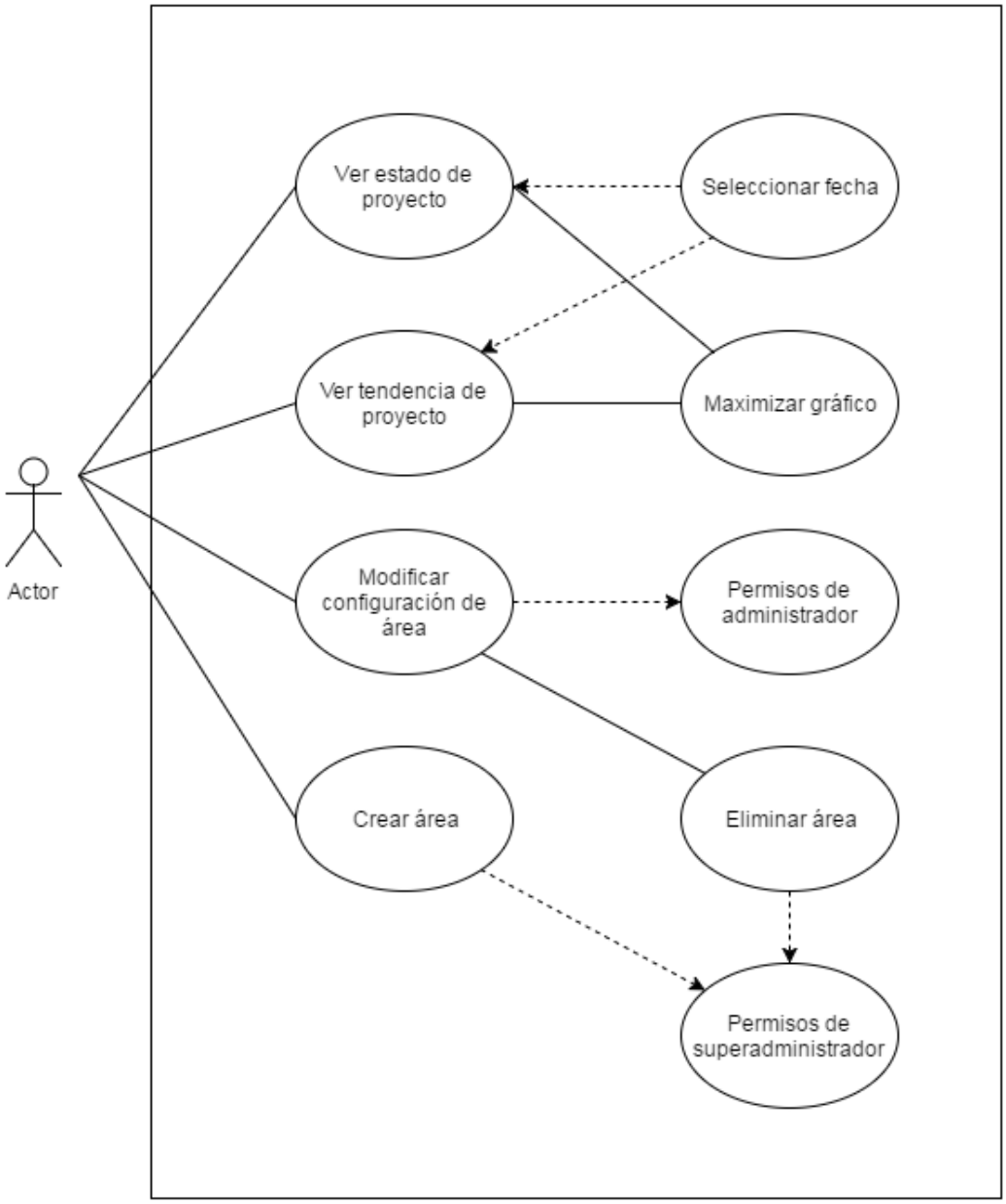


Figura 40: Diagrama de casos de uso

6.2 Diagramas de secuencia

En este anexo se va a documentar a través de diagramas las secuencias de acciones necesarias que se deben suceder para cumplir los distintos casos de uso. Tres agentes interactúan, el usuario, la aplicación *frontend* y el servidor *backend*. Las acciones internas del servidor han sido omitidas ya que no pertenecen al ámbito del proyecto.

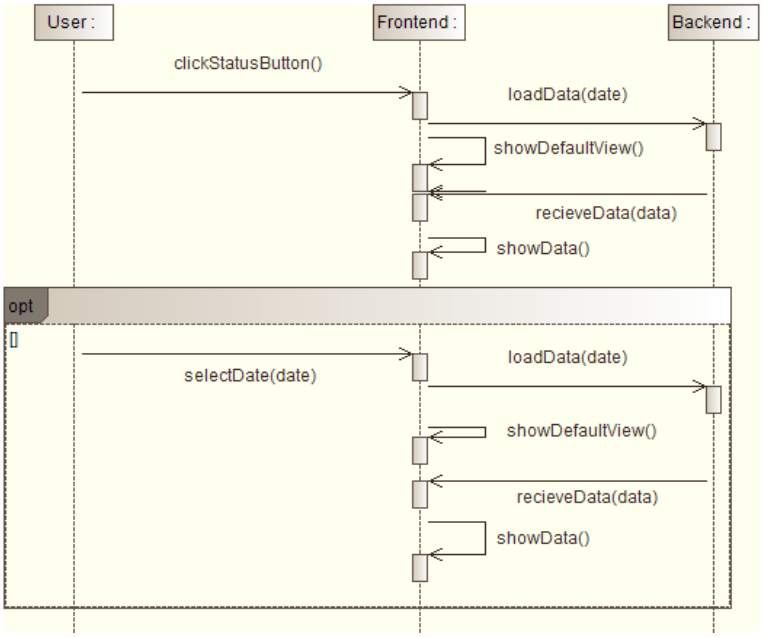


Figura 41: Diagrama de secuencias - Ver estado del área

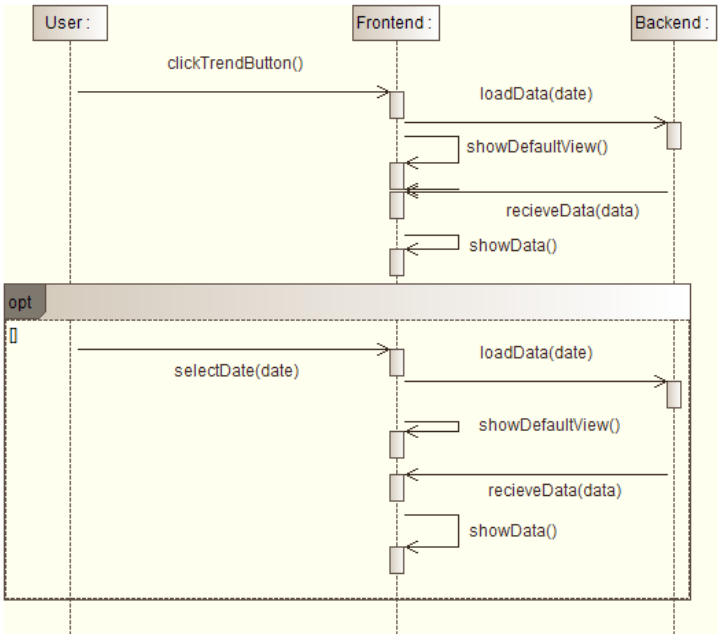


Figura 42: Diagrama de secuencias - Ver tendencia del área

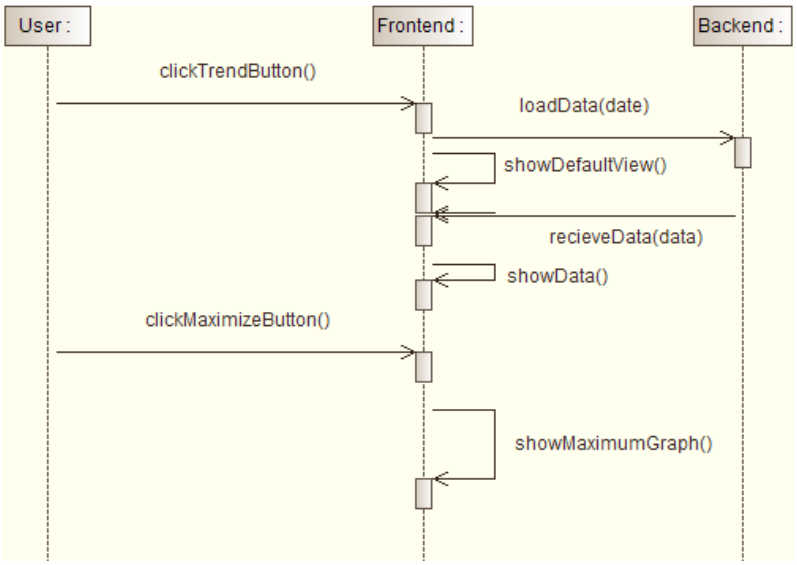


Figura 43: Diagrama de secuencias - Maximizar gráfico

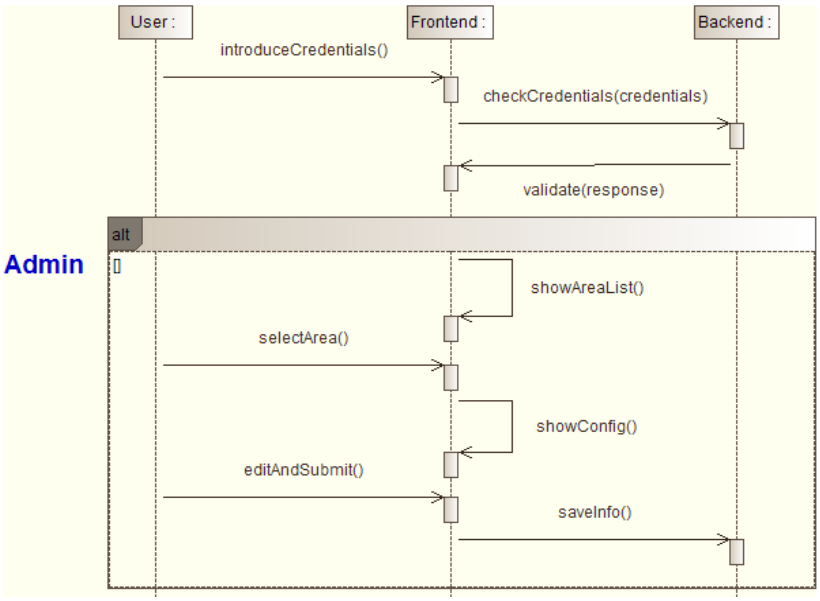


Figura 44: Diagrama de secuencias - Modificar configuración de área

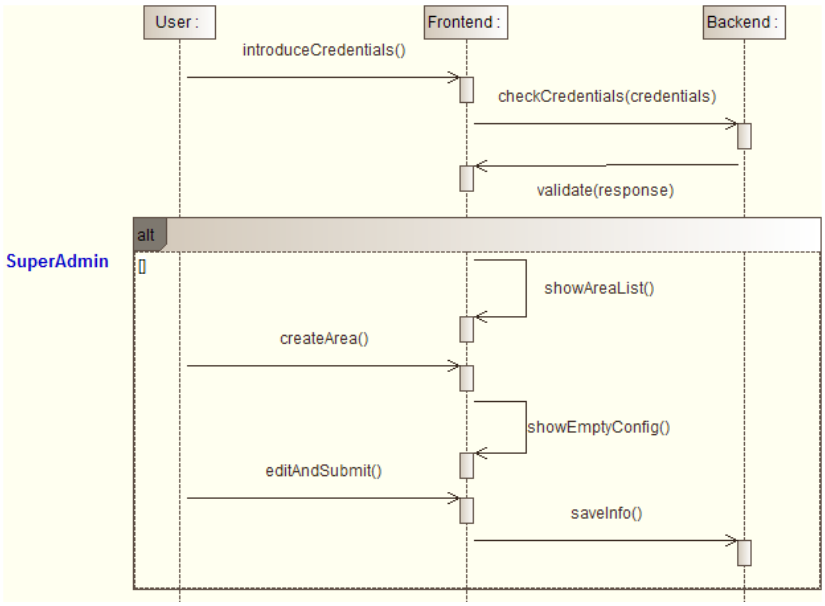


Figura 45: Diagrama de secuencias - Crear área

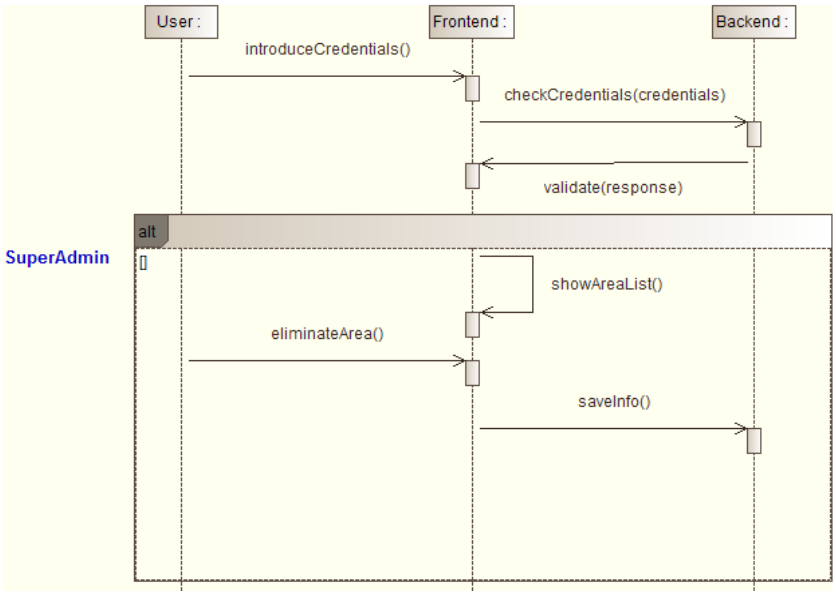


Figura 46: Diagrama de secuencias - Eliminar área

6.3 Evolución de la arquitectura del sistema

En este anexo se va a detallar la evolución de la arquitectura del sistema siendo el resultado final detallado en el apartado 2.2.1.

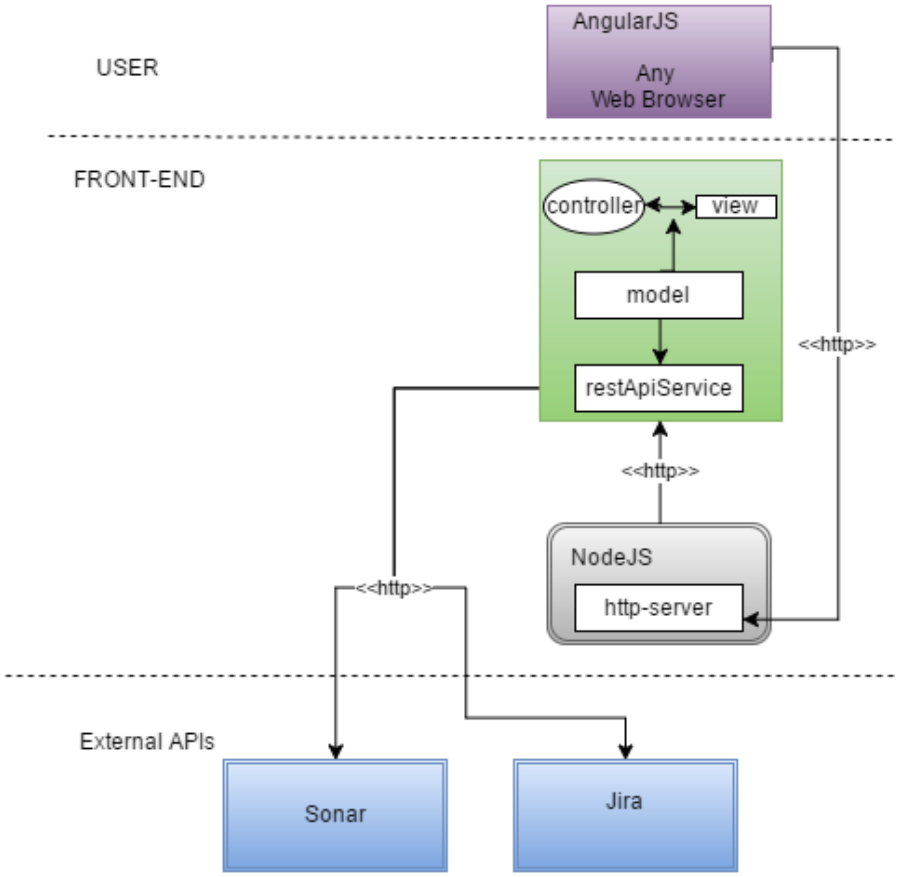


Figura 47: Primera versión de la arquitectura funcional

Se empezó, como se puede ver en la Figura 47, accediendo directamente a las *APIs* expuestas de Sonar y Jira.

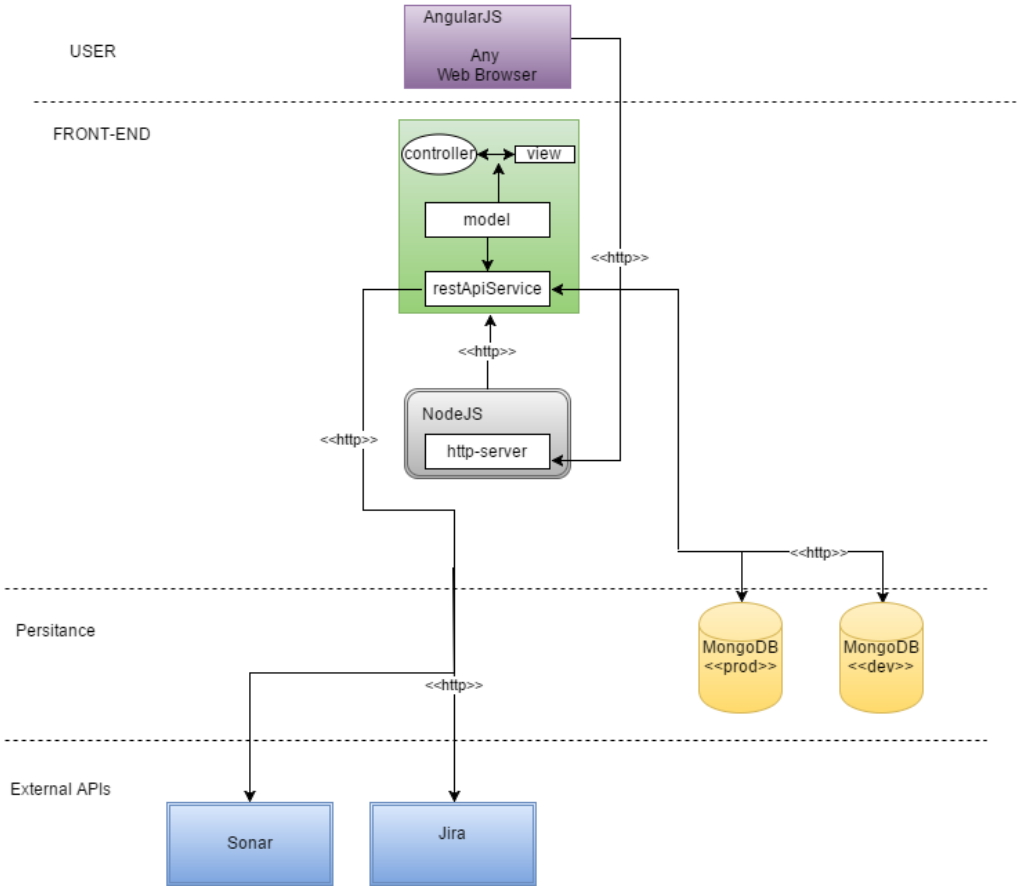


Figura 48: Segunda versión de la arquitectura funcional

A la primera versión se le añade una base de datos MongoDB para añadir control de usuarios a la aplicación y así permitir modificaciones en cuanto a proyectos y umbrales.

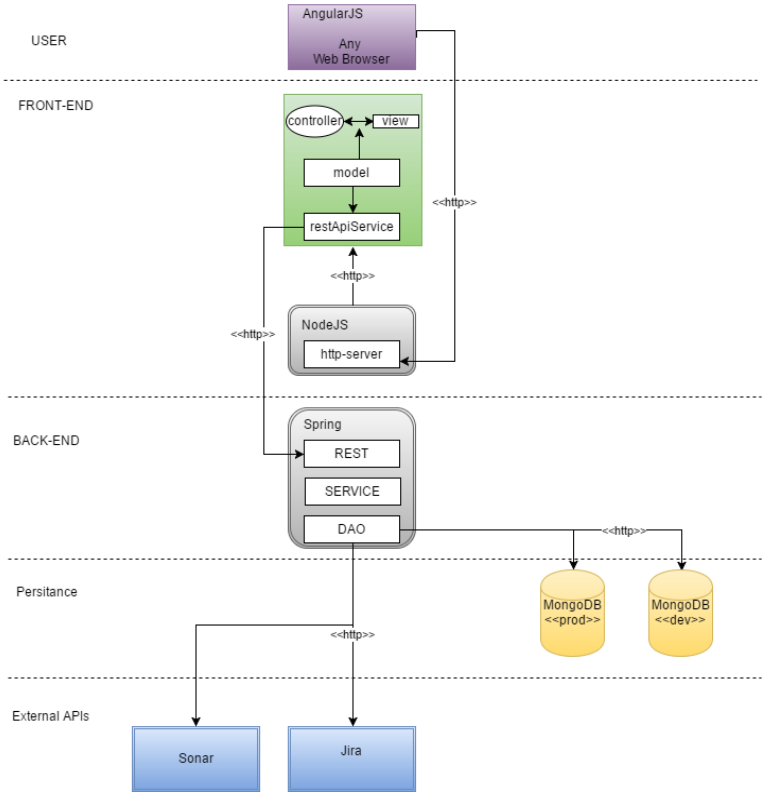


Figura 49: Tercera versión de la arquitectura funcional

En la tercera versión se añade el servidor provisto por Adidas. Sólo se diferencia de la última versión en que Stash no está integrado por el momento.

6.4 Mapa de navegación

En la siguiente sucesión de imágenes se muestra el mapa de navegación de la aplicación. Se caracteriza por ser intuitivo, con una curva de aprendizaje pequeña siendo innecesaria la formación de los usuarios.

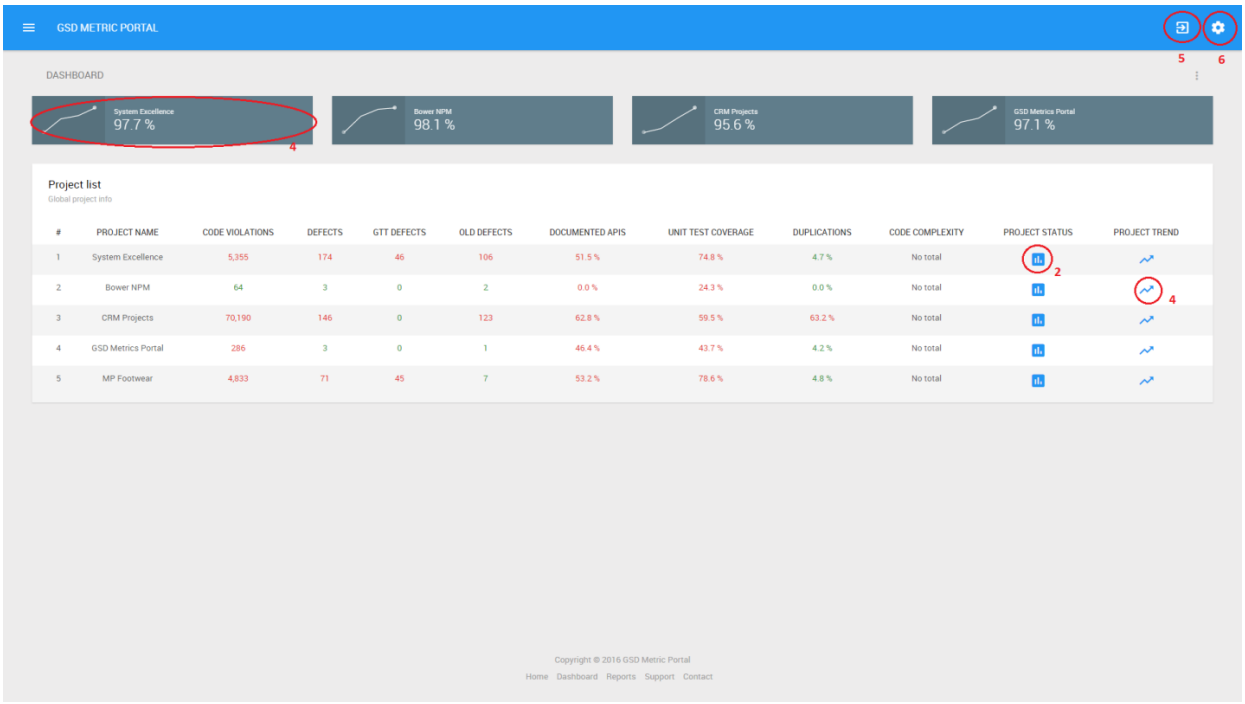


Figura 50: Mapa de navegación 1 - Página principal

Figura 51: Mapa de navegación 2 - Vista de estado

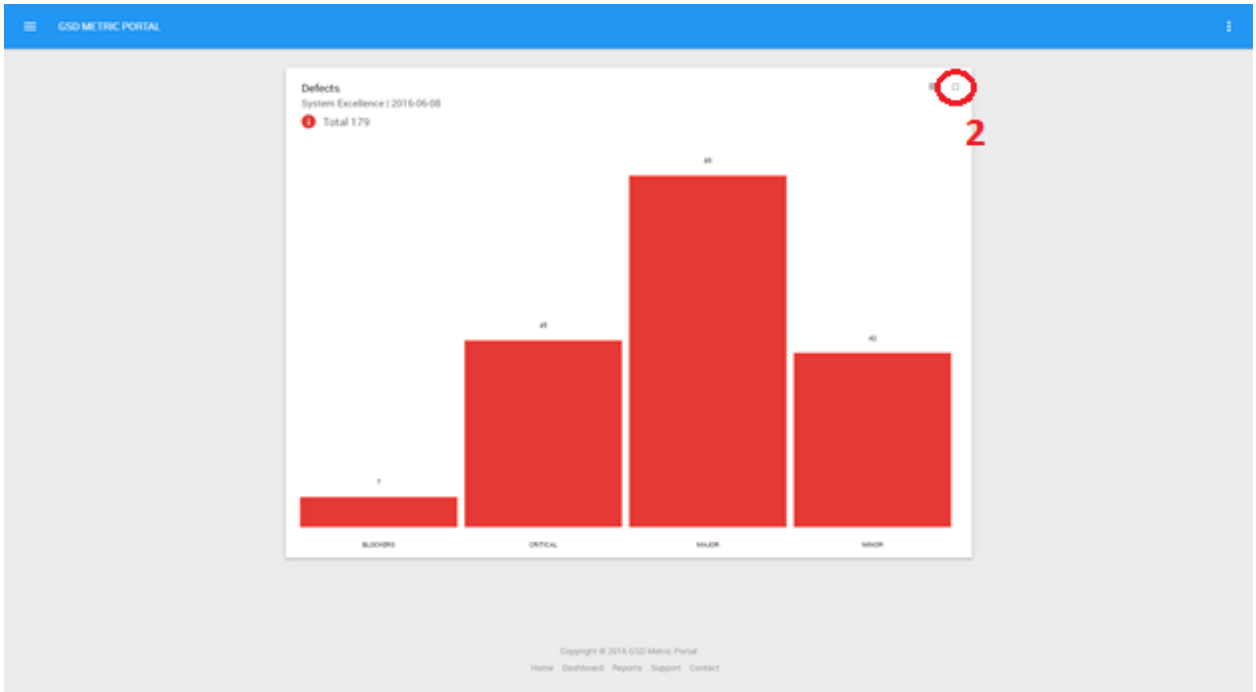


Figura 52: Mapa de navegación 3 - Gráfica maximizada

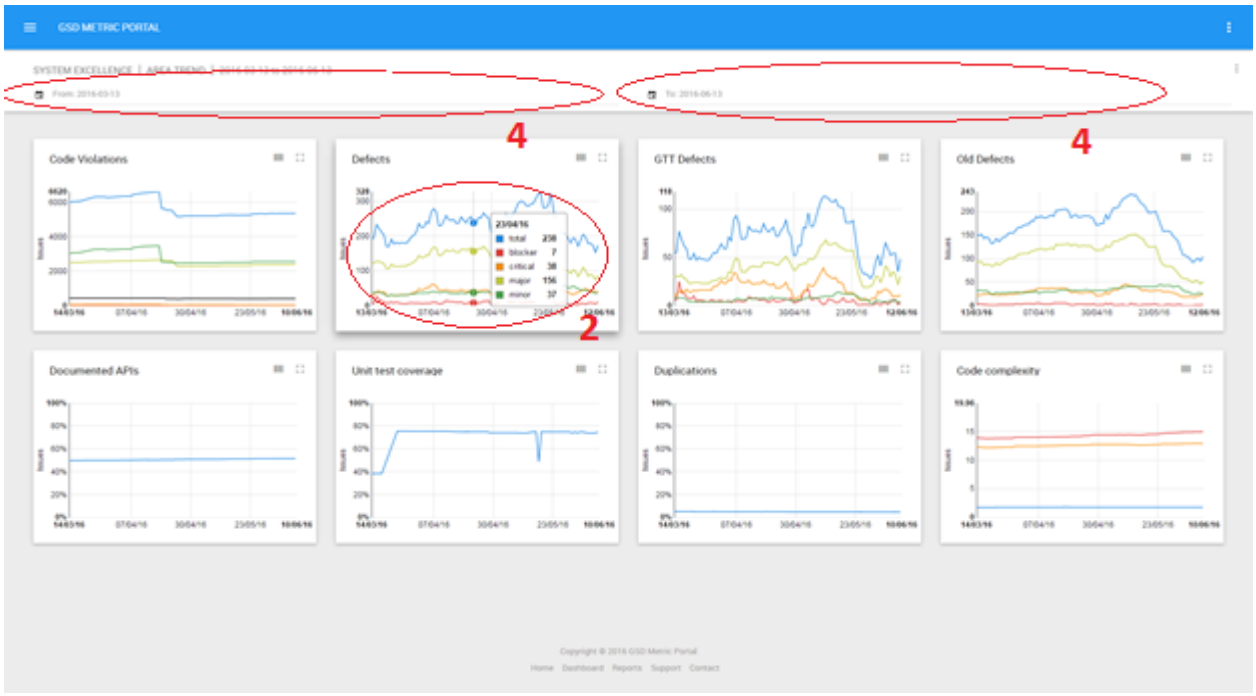


Figura 53: Mapa de navegación 4 - Vista de tendencia

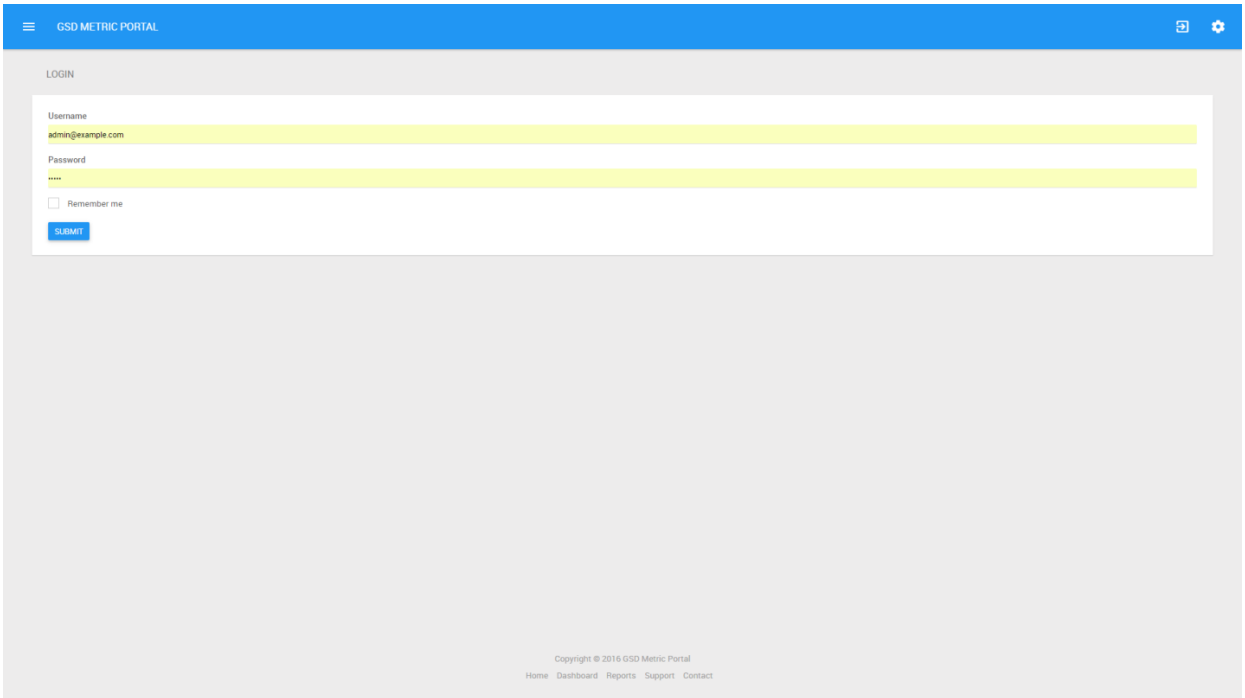


Figura 54: Mapa de navegación 5 - Login

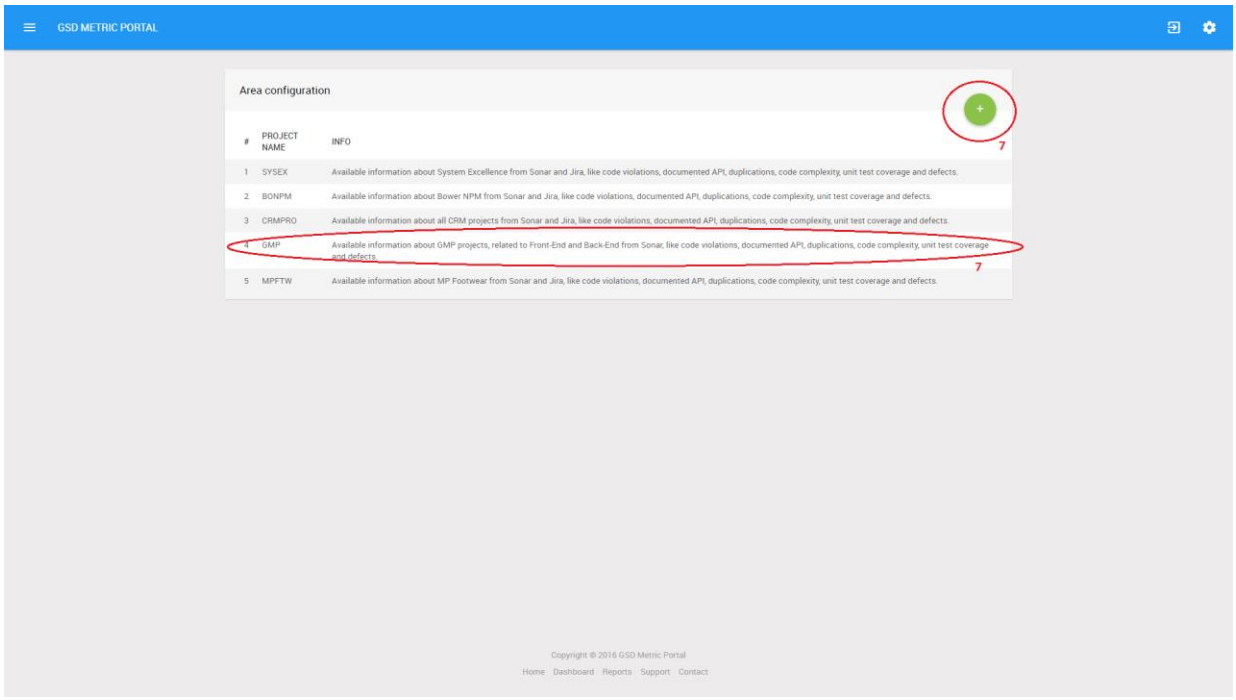


Figura 55: Mapa de navegación 6 - Configuración - Lista de áreas

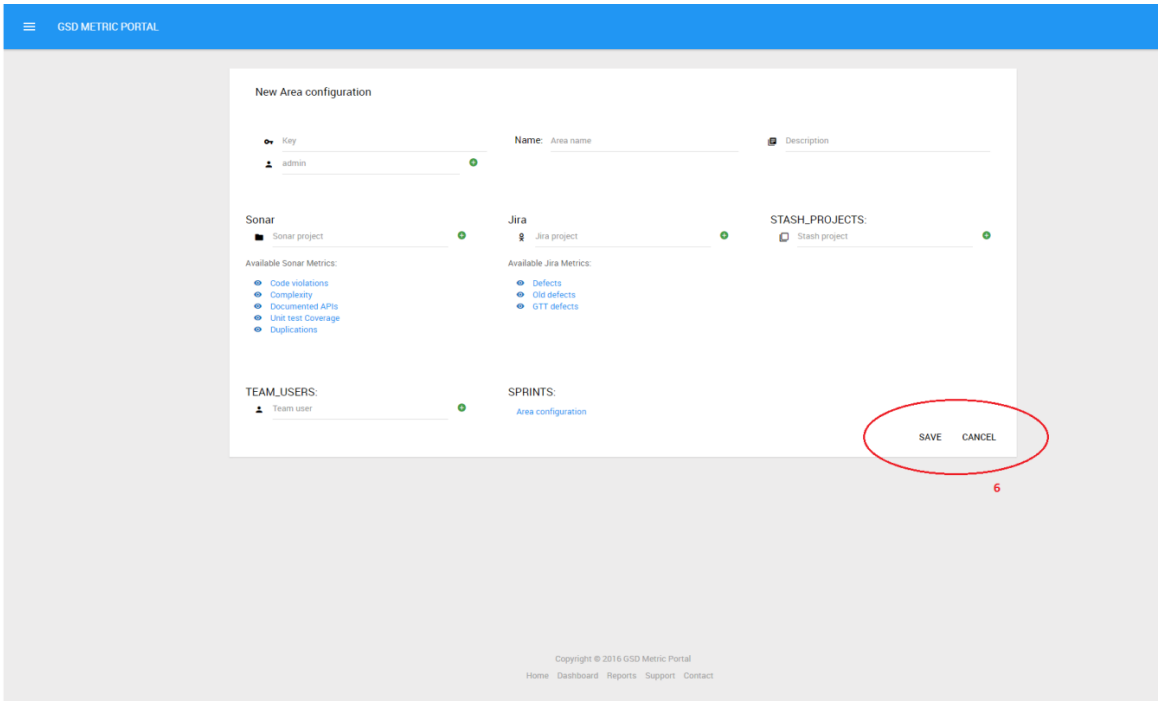


Figura 56: Mapa de navegación 7 - Configuración de área

6.5 Formación de los usuarios

No es necesaria una formación exhaustiva para los usuarios que van a utilizar la aplicación. Sin embargo, es bastante recomendable tener conocimientos de Sonar, Jira y Stash, esto suele ser bastante común ya que los desarrolladores de Adidas, que son los usuarios potenciales de esta aplicación, trabajan de continuo con estas herramientas.

Además la aplicación se ha dado a probar entre ciertos usuarios, tanto jefes de proyecto como desarrolladores, y los comentarios han sido positivos.

6.6 Cascada vs *Agile*, ventajas y desventajas

Hay numerosas diferencias entre la enfoque tradicional o en cascada y *Agile*. En este anexo se especifican esquemáticamente las ventajas y desventajas de cada una de ellos:

Enfoque tradicional o cascada

Ventajas

1. La idea de coste y duración en el tiempo está fijada y suele ser bastante precisa.
2. En el caso de que un miembro del proyecto lo deje, gracias a la documentación, no causará un impacto muy grande.

Desventajas

1. Una vez que un paso ha sido completado, no es posible volver a un estado anterior y realizar cambios.
2. Lo requisitos iniciales deber estar bien definidos para asegurar un buen resultado.
3. Si un requisito cambia, o se encuentra un error en uno de ellos, el proyecto debe empezar desde el principio.
4. El testeo del producto se realiza al final. Si un fallo se ha producido al principio del desarrollo y se detecta al final, puede afectar a cómo se ha desarrollado el código a partir de éste.

Enfoque *Agile*

Ventajas

1. Permite cambios después del planeamiento inicial.
2. Permite añadir funcionalidad para mantener actualizado el producto con respecto al resto de la industria.
3. Al final de cada “sprint” el proyecto es evaluado. Esto permite al cliente dar su opinión y así obtener un producto que se ajuste más a sus necesidades.
4. La fase de test en cada “sprint” asegura que los bugs son detectados en un estado temprano y no afectarán al desarrollo final.
5. El proyecto podría ser lanzado a desarrollo al final de cada “sprint”.

Desventajas

1. Al no estar definido el proyecto desde un inicio, se pueden realizar tareas que luego serán desechadas, por lo que se puede tomar caminos más largos para llegar a un objetivo.

6.7 Principios *Agile*

6.7.1 Manifiesto

- Individuos e interacciones sobre procesos y herramientas
- Software funcionando sobre documentación extensiva
- Colaboración con el cliente sobre negociación contractual
- Respuesta ante el cambio sobre seguir un plan

6.7.2 Principios

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

6.8 Gestión de tiempo

De un total de 8 sprints de dos semanas cada uno (excepto un sprint de deuda técnica de una semana) suman un total de 15 semanas a tiempo completo de trabajo, aproximadamente unas 600 horas. Posteriormente se muestra una lista detallada de cuándo se ha realizado cada tarea.

La línea roja indica la cantidad de story points del sprint en un momento del tiempo. Si baja es que una tarea ha sido completada. Si sube significa que una tarea ha sido incluido a mitad (mala práctica según la metodología Scrum). Conforme se ha ganado experiencia más cantidad de story points han sido incluidos en los sprints ya que la velocidad aumenta, como se puede ver en el último gráfico correspondiente a velocidad. También con más conocimiento del proyecto se mejora en la estimación como se observa a lo largo de los sprints.

(El primer sprint es un mal ejemplo debido a insuficientes conocimientos a la metodología Scrum)

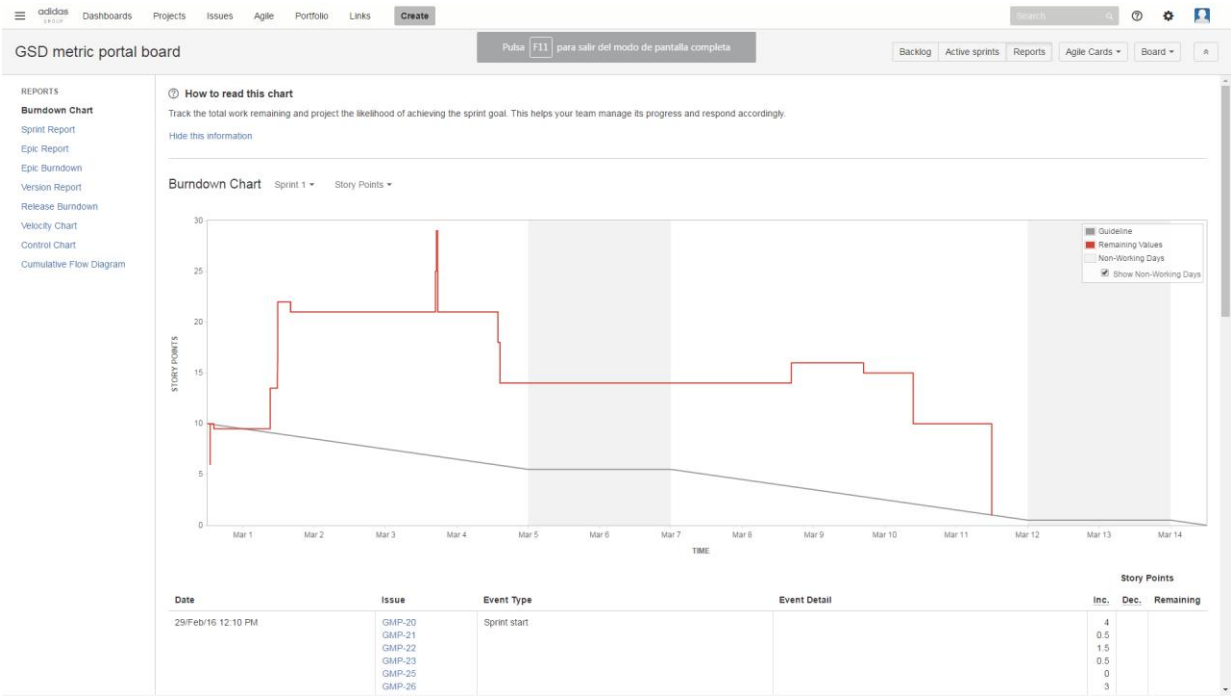


Figura 57: Sprint 1

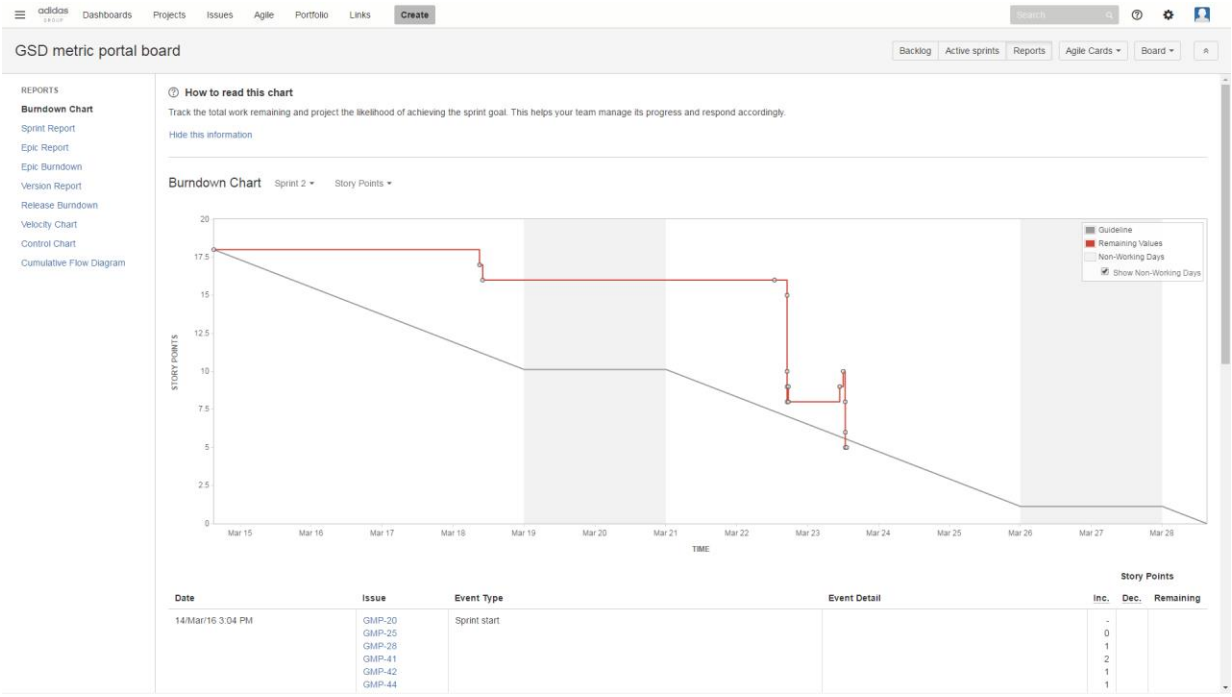


Figura 58: Sprint 2

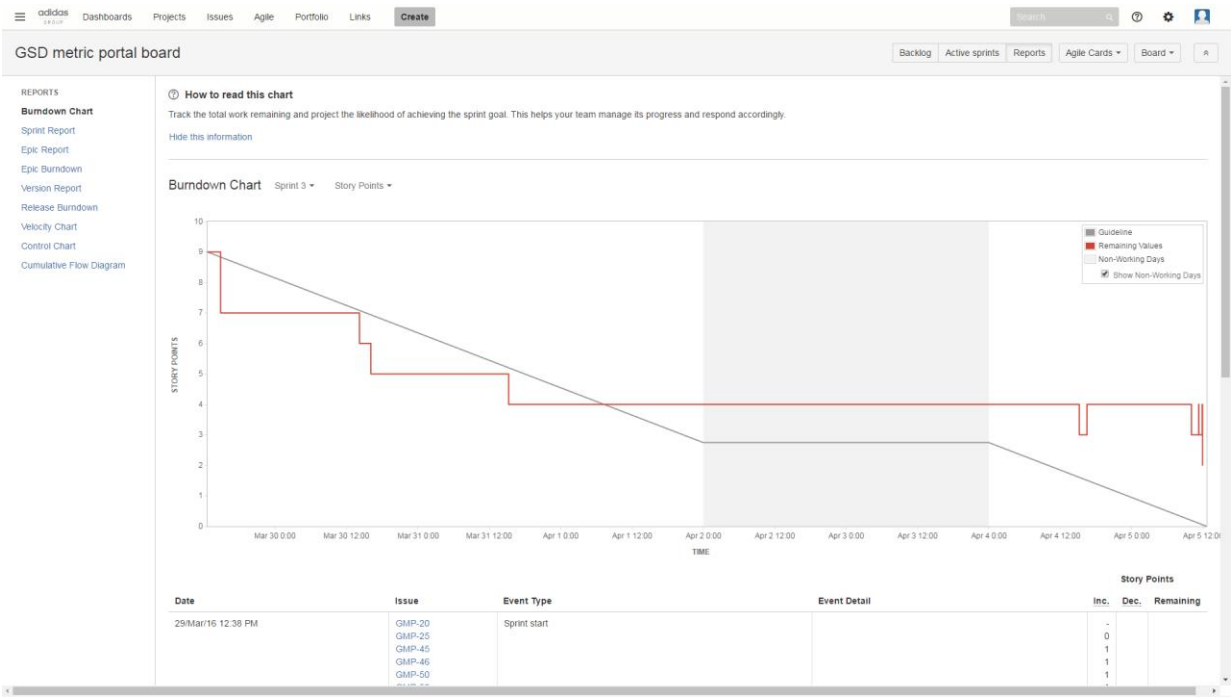


Figura 59: Sprint 3

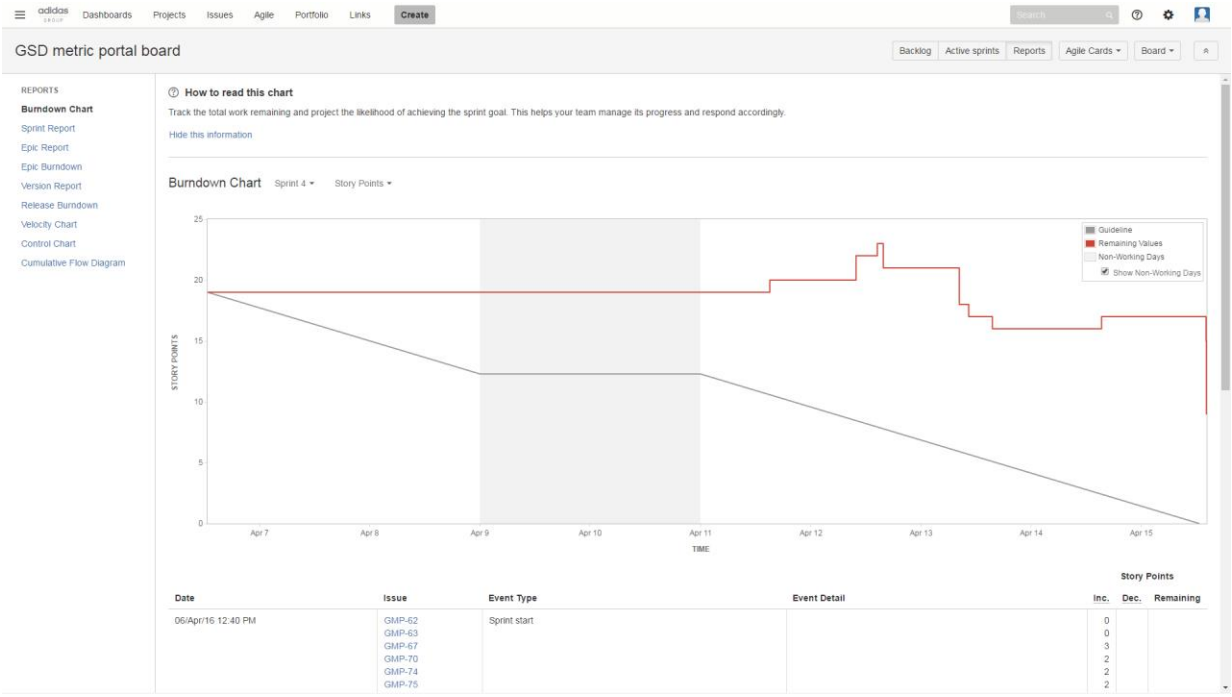


Figura 60: Sprint 4 - Deuda técnica

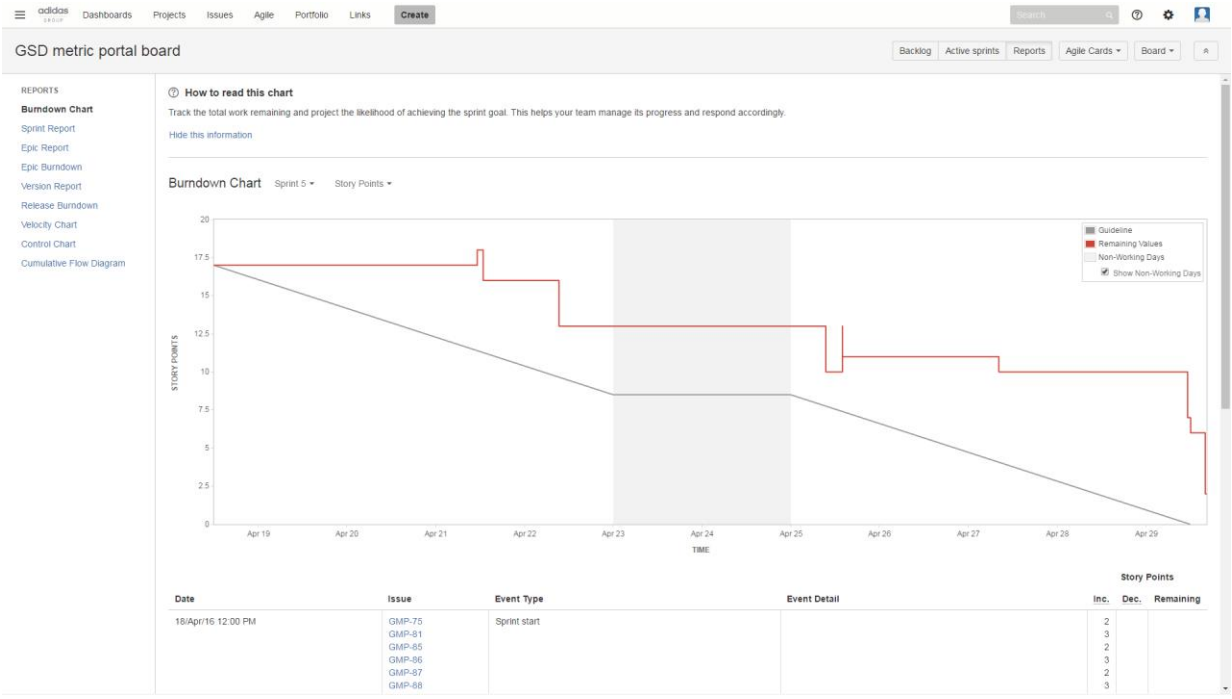


Figura 61: Sprint 5 - Ejemplo de buen sprint

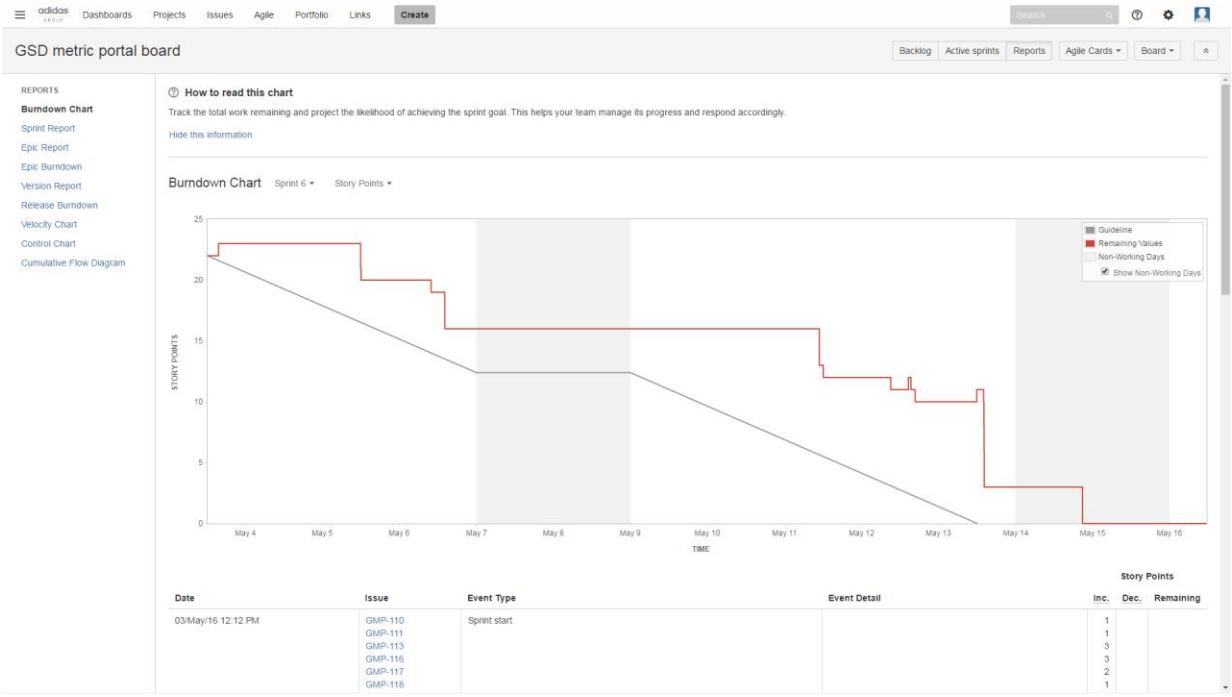


Figura 62: Sprint 6 - Ejemplo de buen sprint

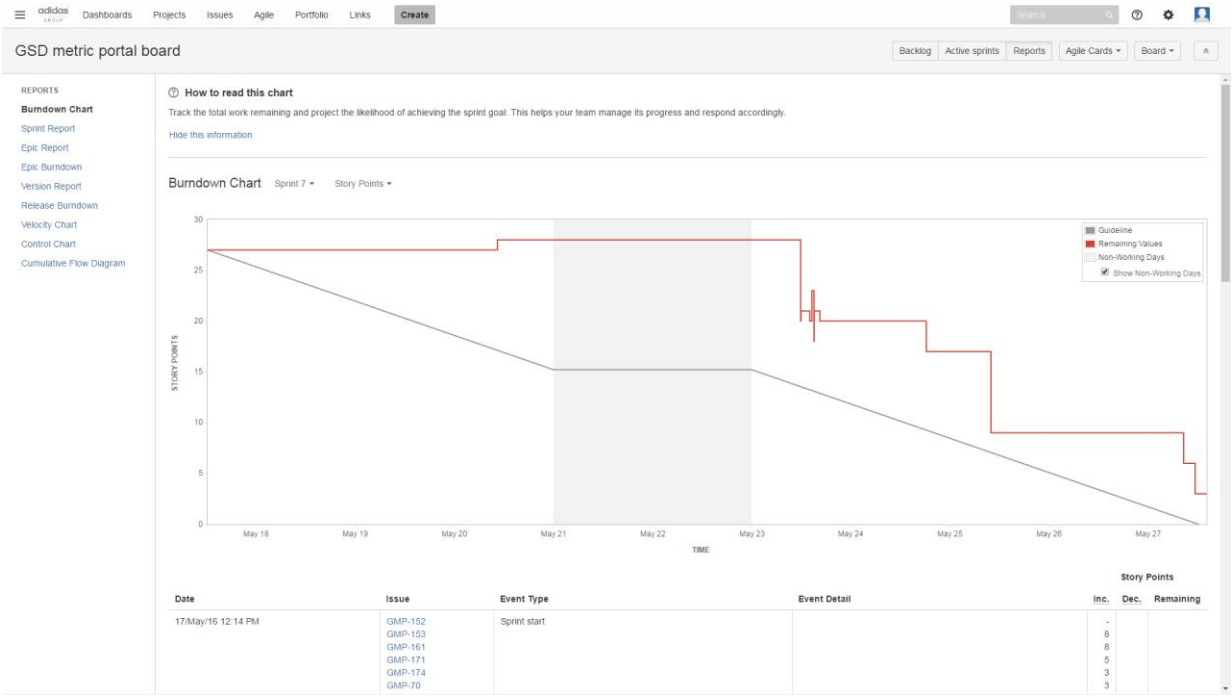


Figura 63: Sprint 7 - Ejemplo de buen sprint

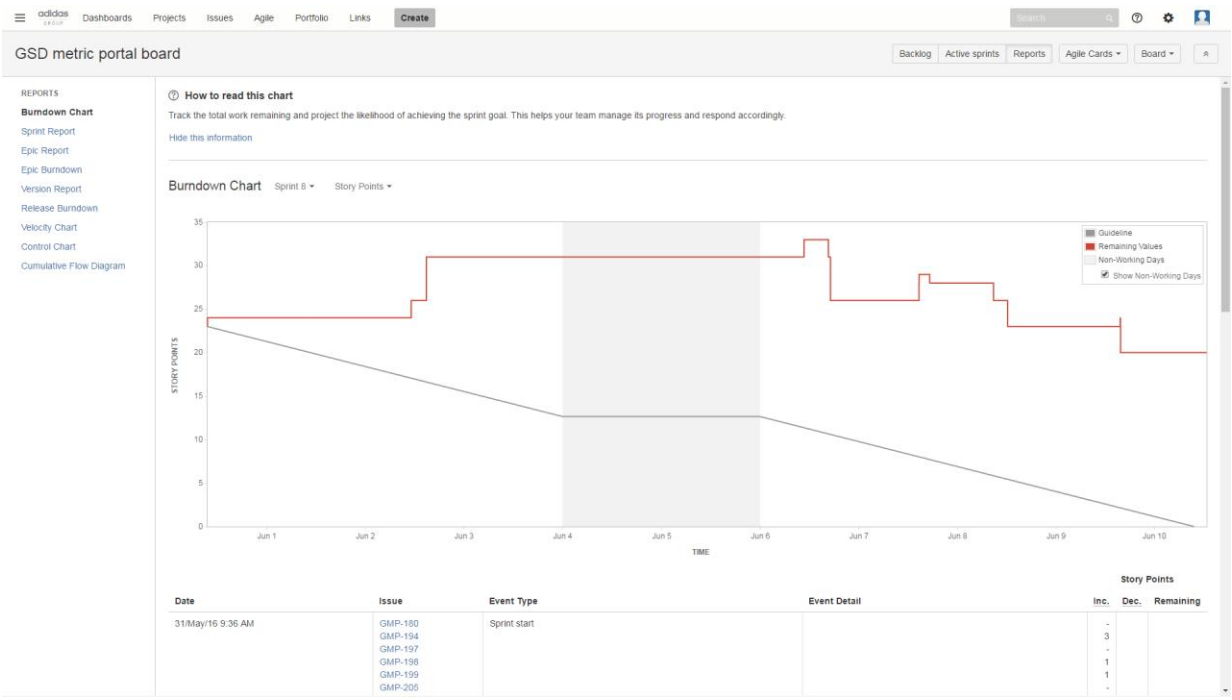


Figura 64: Sprint 8 - Sprint mal estimado

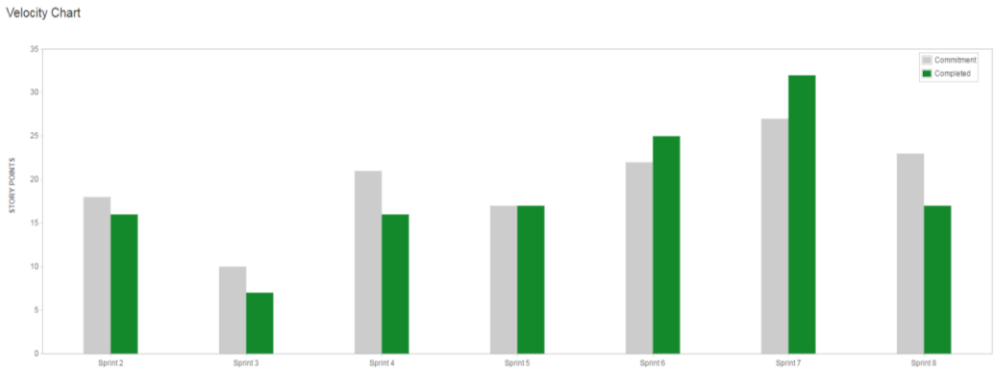


Figura 65: Velocidad de los sprints (Gris estimado, verde finalizado)

6.9 Funcionamiento interno de AngularJS

En la Figura 66 se puede observar como AngularJS funciona a bajo nivel y logra la capacidad del Modelo Vista Controlador. Esta funcionalidad se alcanza a través de *derty-checking*, en otras palabras, comprobar si una variable ha sido modificada. Este proceso se produce en el bucle *$digest* donde el motor dispone de una lista de las variables que debe chequear (*$watch list)*. Si alguna de estas variables ha sido modificada, entraremos en el *Event Loop* el cual se ejecutará hasta actualizar todas las variables modificadas tanto en el DOM como en la aplicación JavaScript. Si este bucle se da más de diez veces seguidas saltará una excepción y este terminará.

Nótese que si una variable es modificada fuera del control de AngularJS será necesario forzar este evento (*$apply*) por lo que el conocimiento del funcionamiento interno del modelo es muy importante tanto para desarrollar como para diseñar una aplicación.

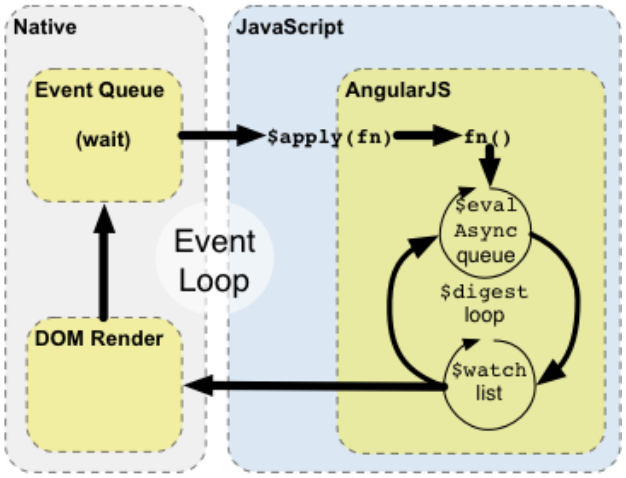


Figura 66: Ciclo de actualización de AngularJS

6.10 Estructura de JWT

En la Figura 67 se puede ver el proceso que sigue la creación de un JWT y su posterior uso.

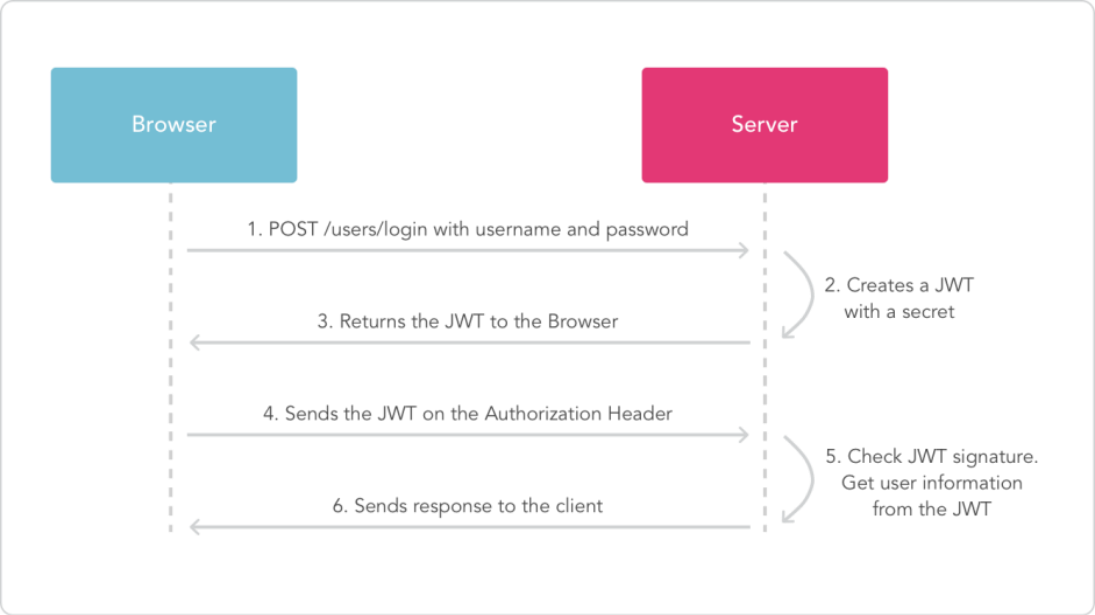


Figura 67: Flujo de proceso de JWT

Los JSON Web Tokens tienen la siguiente estructura: “xxxxx.yyyyy.zzzzz”. Serado por puntos el *header*, contenido y firma:

- **Header:** contiene el tipo (JWT) y el algoritmo hash.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Figura 68: Header JWT

- **Contenido:** codificado en Base64Url contiene los requerimientos, hay de tres tipos:
    - **Reservados:** Aquellos campos recomendados pero no obligatorios:
        - **iss:** Editor
        - **exp:** Tiempo de expiración
        - **sub:** Sujeto
        - **aud:** Audiencia

- **Públicos:** Definidos por los usuarios de Jets
- **Privados:** Necesario que ambas partes estén de acuerdo.

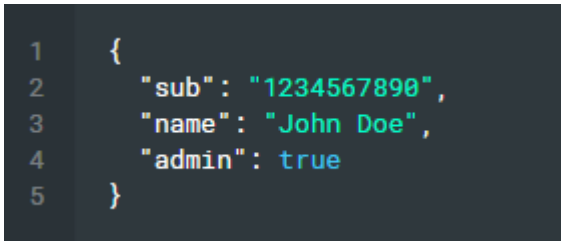


Figura 69: Contenido de JWT

- **Firma:** Se usa para verificar que el emisor del JWT es quién dice ser y para asegurarse que el mensaje no ha cambiado mientras se estaba enviando. Para crear la firma es necesario codificar el header, el contenido y junto con una contraseña utilizando el algoritmo especificado en el header codificarlo.

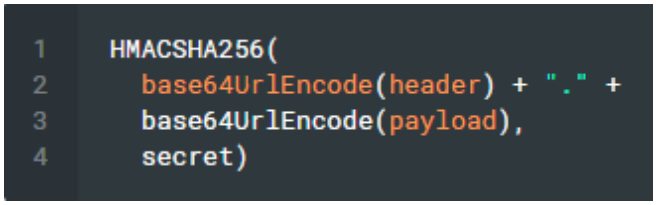


Figura 70: Firma usando HMACSHA256

6.11 Graficas en detalle

En las siguientes figuras se muestran los resultados finales de los distintos tipos de gráficas: de barras, velocímetro, de líneas y de sectores.

Figura 71: Gráfico de barras final

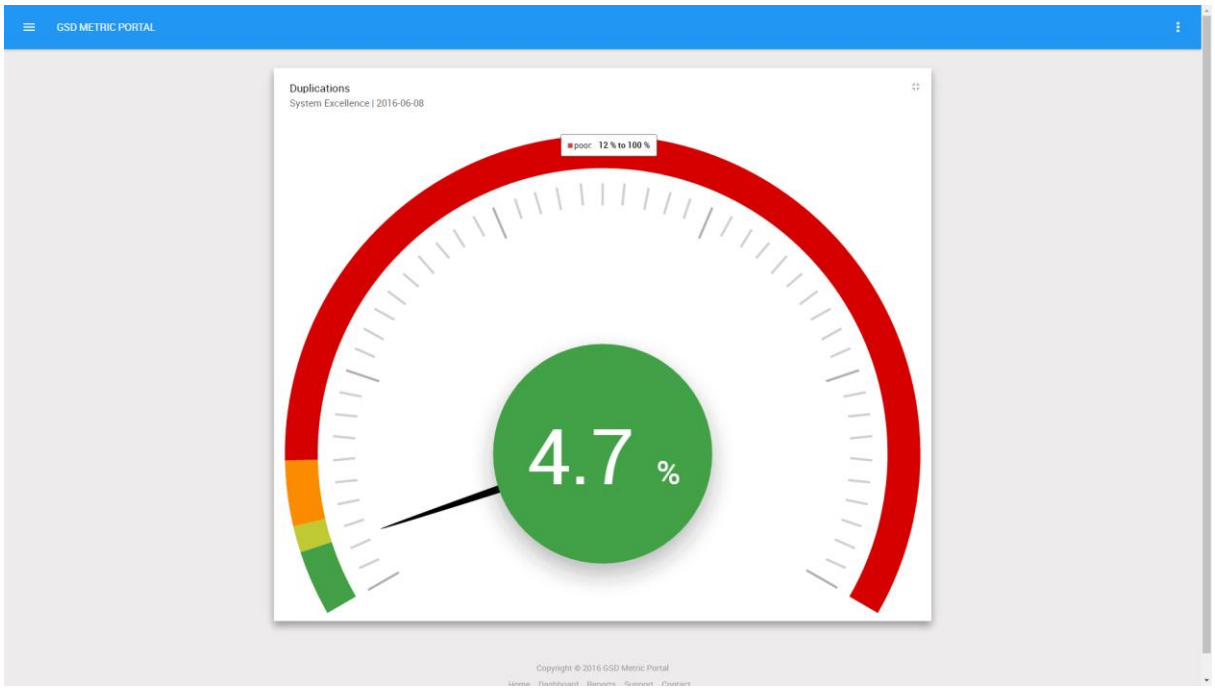


Figura 72: Velocímetro final

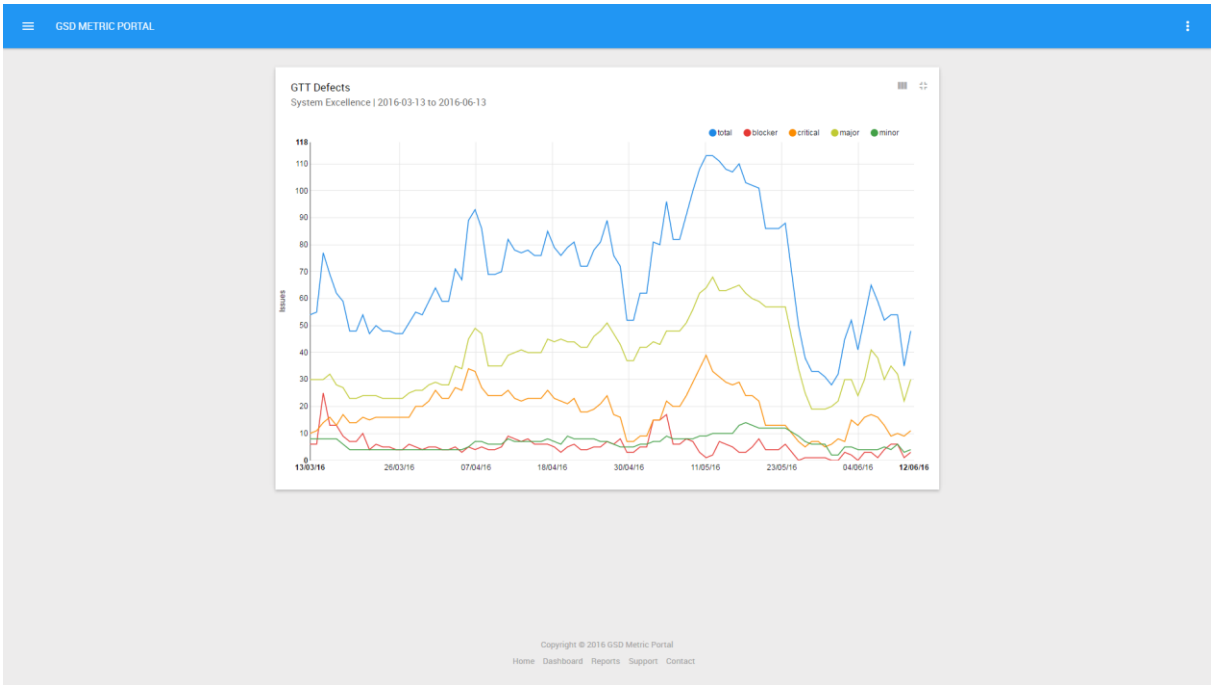


Figura 73: Gráfico de líneas final

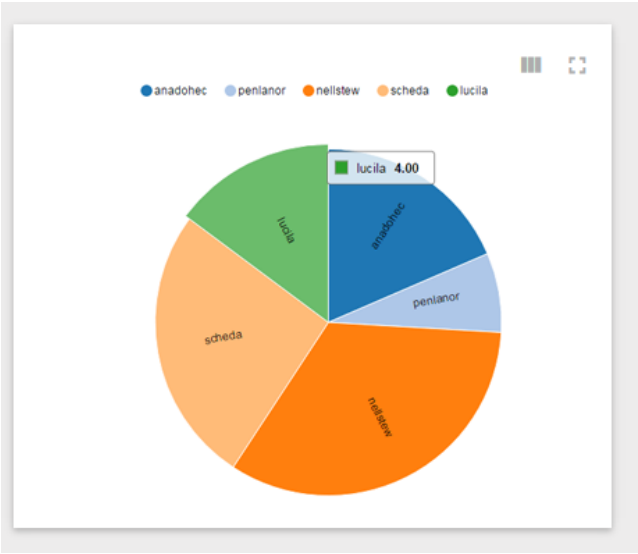


Figura 74: Gráfico de sectores final

6.12 Jira

Es una herramienta que pertenece a la empresa Atlassian. Es muy potente y permite guardar las distintas tareas con las distintas jerarquías que existen en Scrum. Además permite dar prioridad a éstas y asignarlas a distintos desarrolladores. Finalmente, provee información sobre el rendimiento del proyecto así como un estado general. Dispone de gráficas tanto sobre rendimiento como estado actual del proyecto. Por supuesto, Jira permite la funcionalidad básica de mostrar el estado actual de cada tarea dentro del sprint (En desarrollo, revisando y hecha).

En la Figura 75 y la Figura 76 se puede observar, por orden de aparición el backlog general del proyecto, el backlog del sprint.

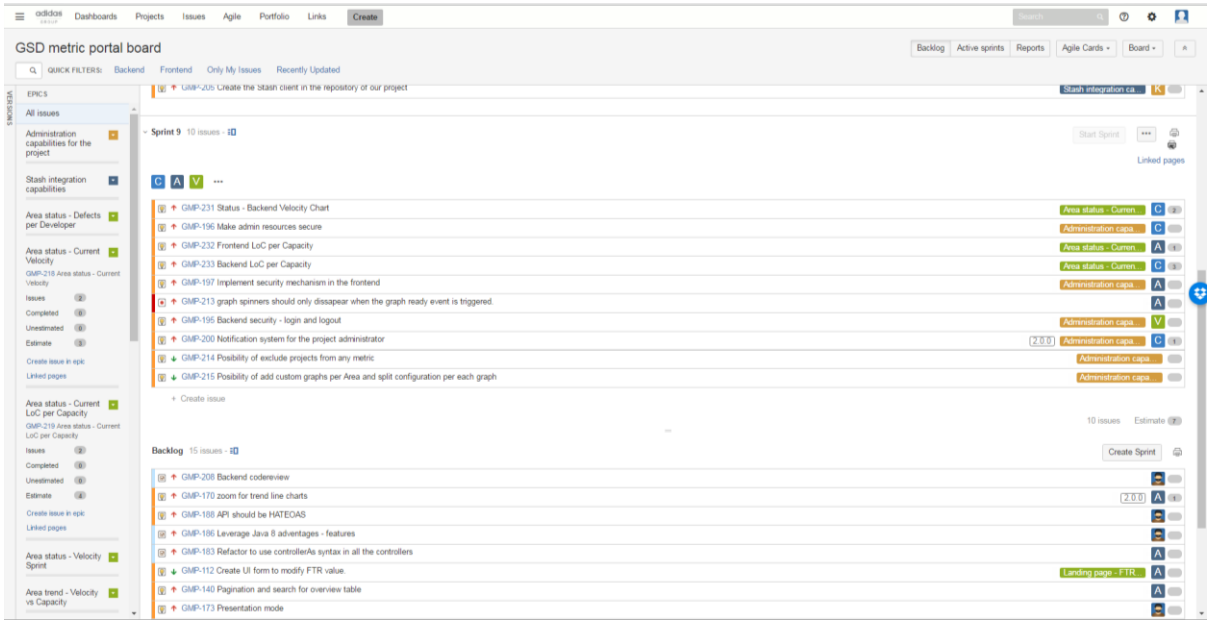


Figura 75: Product backlog

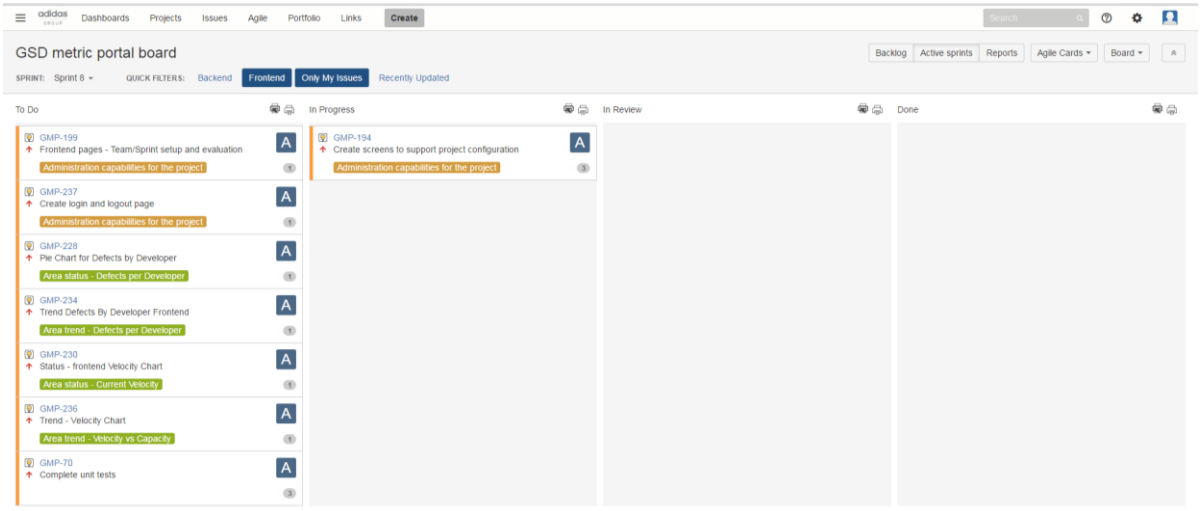


Figura 76: Sprint backlog

6.13 Sonar

Es una herramienta de código libre que permite el análisis de un código asignado. En la Figura 77 se puede ver el estado actual del proyecto. Destaca por el porcentaje de código duplicado con un 0%. Como punto negativo, la complejidad de algunos ficheros está por encima del punto óptimo, lo que requeriría una refactorización que la herramienta Sonar estima que se realizaría en 2h.

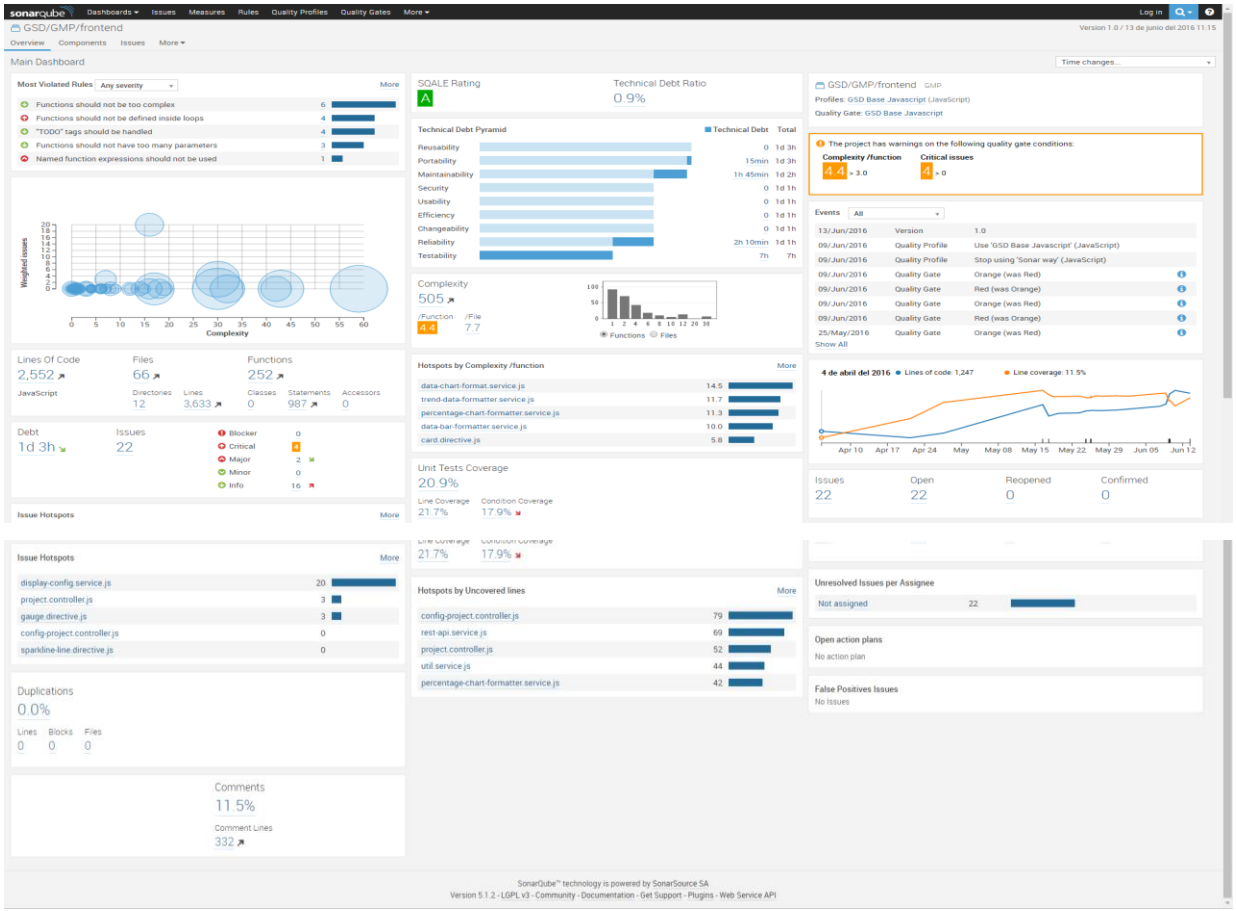


Figura 77: Resultados de Sonar par GMP

6.14 TeamCity

Consiste en un servicio compuesto por un conjunto de agentes que proceden a hacer distintas tareas computacionales. En la Figura 78 se puede ver la interfaz para ejecutar las tareas que tenemos configuradas. La primera de ellas es realizar el proceso de *build* explicado en la Figura 79 y la segunda consiste en correr los test unitarios para confirmar el funcionamiento de la aplicación como es explicado en la Figura 80.

Figura 78: Tareas de TeamCity

Build Steps (5) edit »

Step 1: **Install NPM dependencies.**
- Runner type: **Node.js NPM** (Starts NPM)
- Execute: **If all previous steps finished successfully**
- Run targets: view
- NPM: **<default>**
- Working directory: **same as checkout directory**
- Additional command line arguments: **<empty>**

Step 2: **Install Bower dependencies**
- Runner type: **Command Line** (Simple command execution)
- Execute: **If all previous steps finished successfully**
- Working directory: **same as checkout directory**
- Custom script: view script content
- Deploy artifacts to Artifactory: **disabled**

Step 3: **Grunt build**
- Runner type: **Command Line** (Simple command execution)
- Execute: **If all previous steps finished successfully**
- Working directory: **same as checkout directory**
- Custom script: view script content
- Deploy artifacts to Artifactory: **disabled**

Step 4: **App Deployment**
- Runner type: **SSH Deployer** (Runner able to deploy build artifacts via SSH)
- Execute: **If all previous steps finished successfully**
- Target host: **%deploymentpath%**
- Target port: **default**
- Username: **alzorign**
- Transport: **SFTP**
- Source: **dist/**/* => ./**

Step 5: **Configuration Deployment**
- Runner type: **SSH Deployer** (Runner able to deploy build artifacts via SSH)
- Execute: **If all previous steps finished successfully**
- Target host: **%deploymentpath%**
- Target port: **default**
- Username: **alzorign**
- Transport: **SFTP**
- Source: **frontend/%staging properties%/endpoints.properties => ./**

Figura 79: Pasos de Build and Deploy

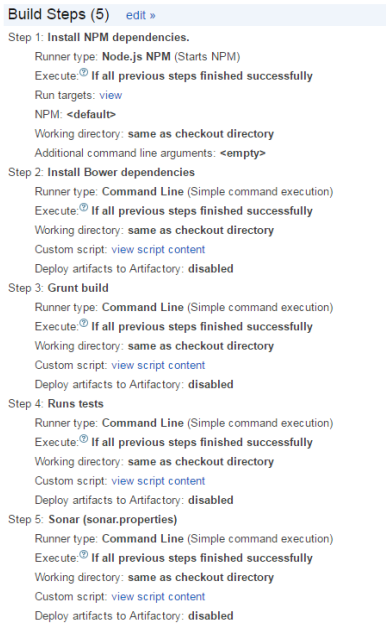


Figura 80: Pasos de Testing

6.15 Confluence

Es una herramienta web en la que se documentará los requisitos de negocio. En las siguientes imágenes se pueden ver algunos de los requisitos que se recibieron desde negocio a través de *mock ups*.

En la Figura 81 se muestra el mapa de navegación con las distintas ventanas y redirecciones que la aplicación debe soportar.

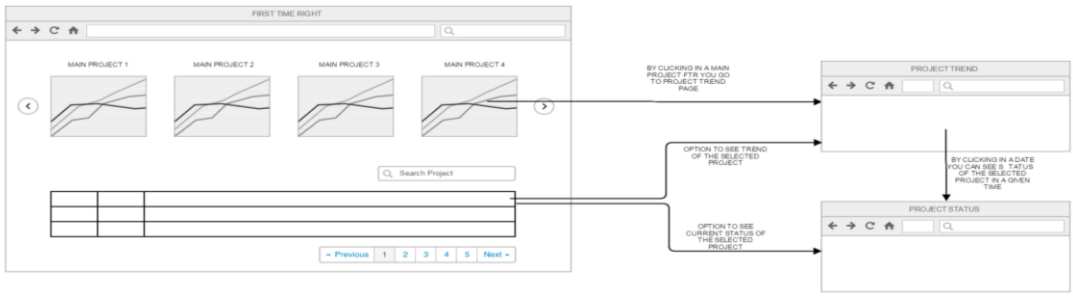


Figura 81: Mock up de mapa de navegación

En la Figura 82 y la Figura 83 se muestran las distintas ventanas de métricas que habrá con su distribución. Las gráficas mostradas son solo intuitivas.

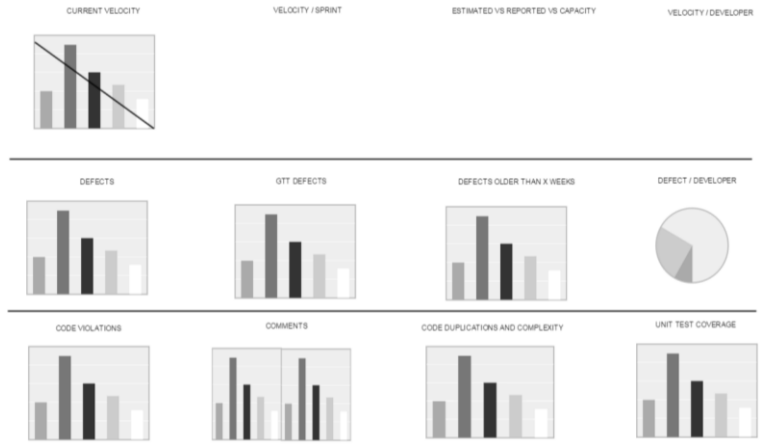


Figura 82: Mock up de área de estado

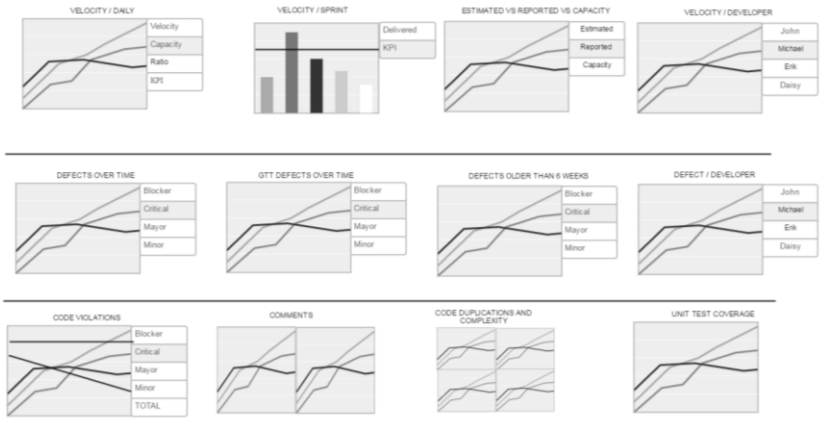


Figura 83: Mock up de área de tendencia

En la Figura 84, la Figura 85 y la Figura 86 se muestra el primer análisis de las métricas a mostrar. Es una guía, no es definiitivo ya que *Unit test coverage* no fue un gráfico de barras finalmente.

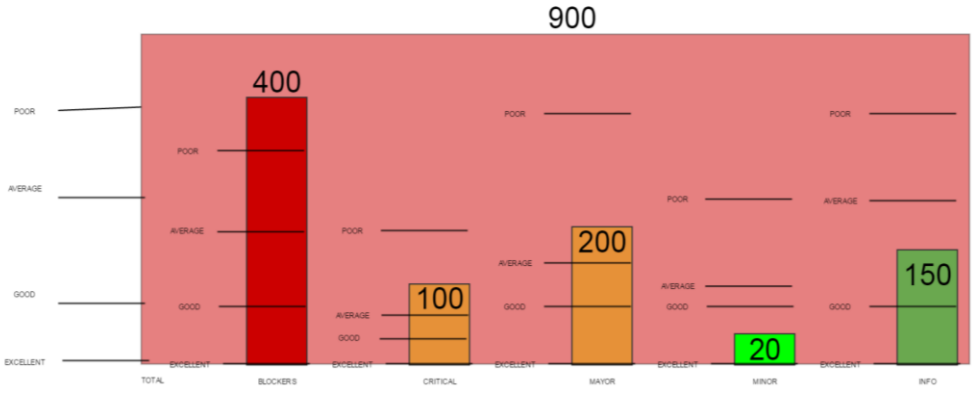


Figura 84: Mock up de Code violations

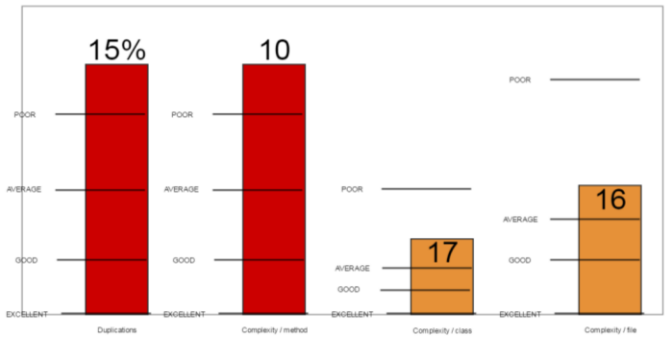


Figura 85: Mock up de Duplications y Code complexity

Figura 86: Mock up de Unit test coverage

6.16 Documented APIs

Ejemplo de cómo se calcula el porcentaje de código documentado.

El siguiente extracto de código contiene 9 líneas relevantes de comentarios:

```
/**                              +0 => empty comment line
 *                               +0 => empty comment line
 * This is my documentation          +1 => significant comment
 * although I don't                  +1 => significant comment
 * have much                         +1 => significant comment
 * to say                          +1 => significant comment
 *                               +0 => empty comment line
 ****************************          +0 => non-significant comment
 *                               +0 => empty comment line
 * blabla...                       +1 => significant comment
```

```
 */                          +0 => empty comment line

/**                          +0 => empty comment line
 * public String foo() {          +1 => commented-out code
 *   System.out.println(message);      +1 => commented-out code
 *   return message;                +1 => commented-out code
 * }                          +1 => commented-out code
 */                          +0 => empty comment line
```

6.17 Página de configuración

En las siguientes imágenes se aprecia la posibilidad de configurar distintos umbrales y opciones. La duración de un sprint y los umbrales de *Code Violations*.

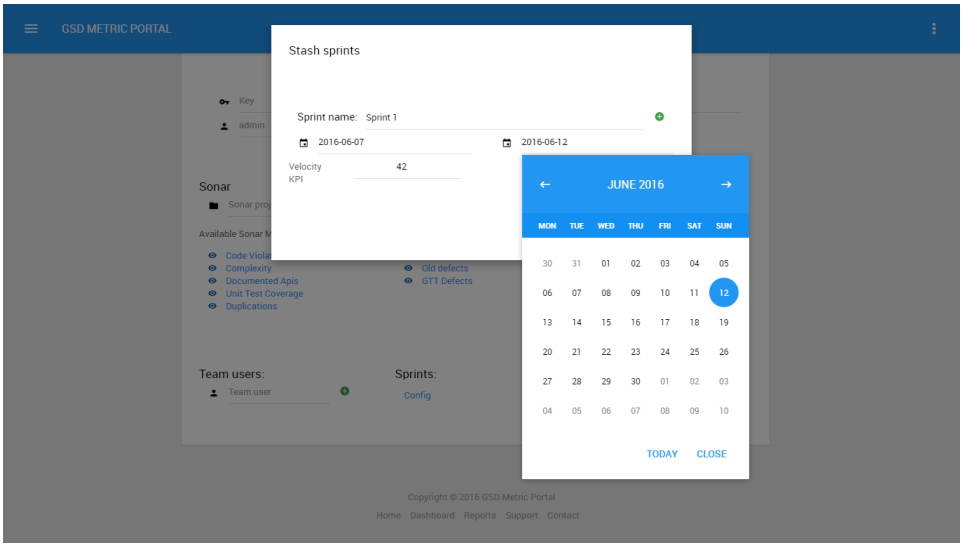


Figura 87: Configuración de spritns

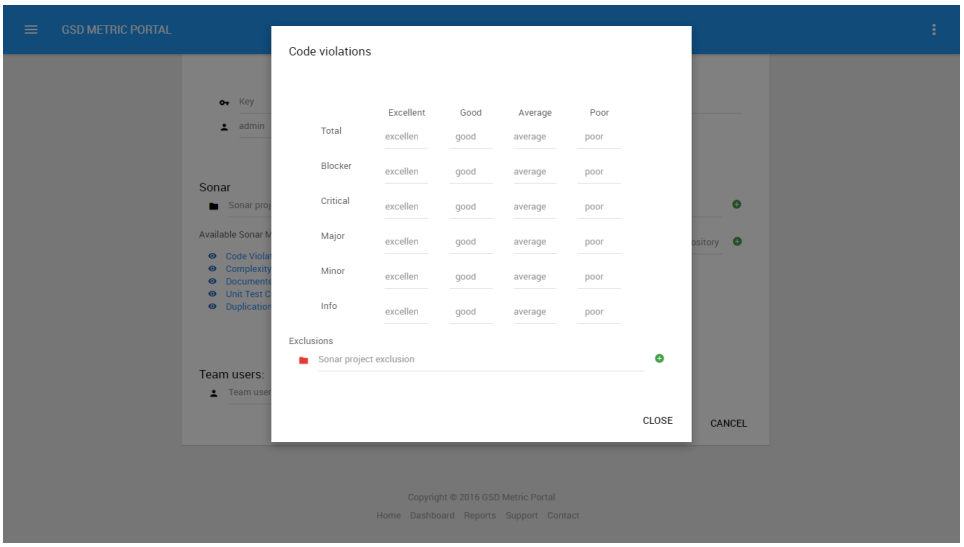


Figura 88: Configuración de quality gates

6.18 Rendimiento de la aplicación

Para comprobar el rendimiento de la aplicación se ha utilizado las herramientas de desarrollo de Google Chrome https://developer.chrome.com/extensions/experimental_devtools_audits.

Estas contienen un apartado de auditoría. En este, comprueban las llamadas que el navegador realiza y como las maneja con el objetivo de encontrar los puntos donde la aplicación puede fallar o se puede mejorar. Los fallos encontrados fueron mínimos (se pueden ver en la Figura 89) y se resolvieron habilitando la cache en el servidor que sirve los ficheros de la aplicación y habilitando la compresión gzip explicada previamente.

Enable gzip compression (5)

Compressing the following resources with gzip could reduce their transfer size by about two thirds (~1017 KB):

deheremap6993/could save ~679 B
style.4abd38f46b7eaa4b.csscould save ~208 KB
vendor.508f1eb31f94a682.jscould save ~701 KB
app.5f1e5609548e26db.jscould save ~105 KB
:9002/languages/en.jsoncould save ~1.7 KB

Leverage browser caching (8)

The following cacheable resources have a short freshness lifetime:

deheremap6993/
style.4abd38f46b7eaa4b.css
vendor.508f1eb31f94a682.js
app.5f1e5609548e26db.js
:9002/languages/en.json
:9002/endpoints.properties
profile-menu.png
Material-Design-Iconic-Font.woff2

Minimize cookie size

The average cookie size for all requests on this page is 102 B

Figura 89: Auditoría Chrome dev tools

A continuación, se muestra los resultados obtenidos a través del plugin de Chrome "*Performance-Analyser*". A destacar, el tiempo total de carga de llamadas es de 2415ms sin embargo la aplicación empieza a ser usable cuando se ha cargado el DOM a los 825ms consiguiendo una gran agilidad a pesar de la pesadez de la carga de datos.

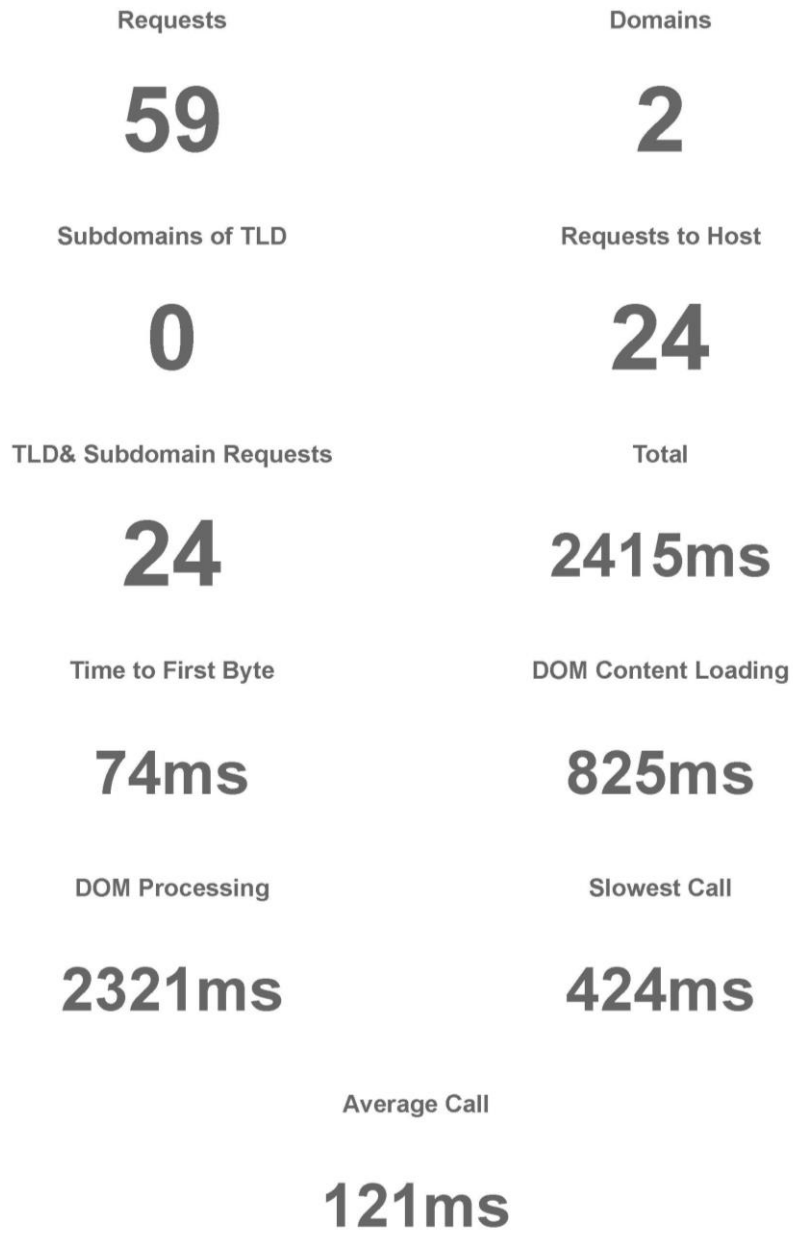
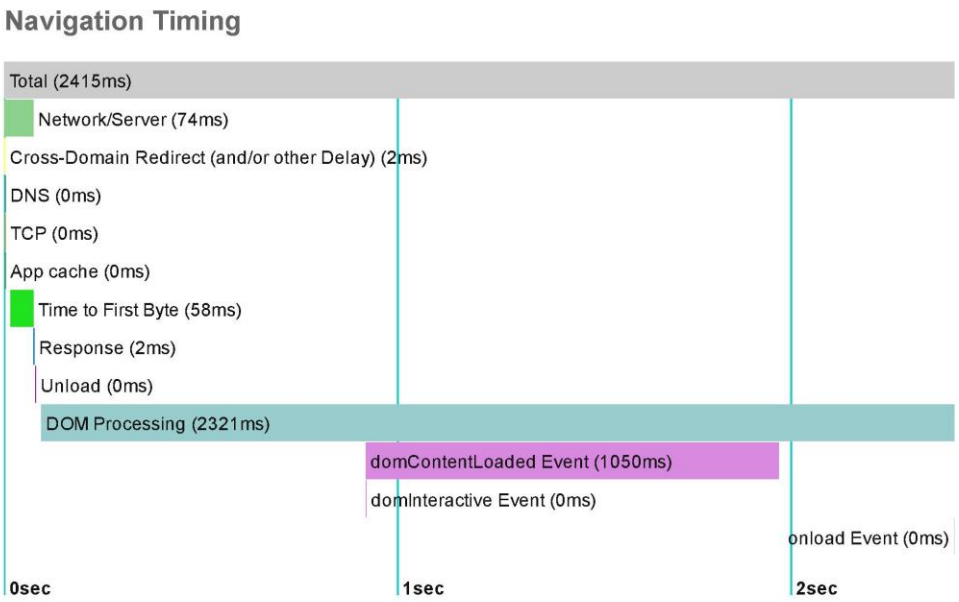
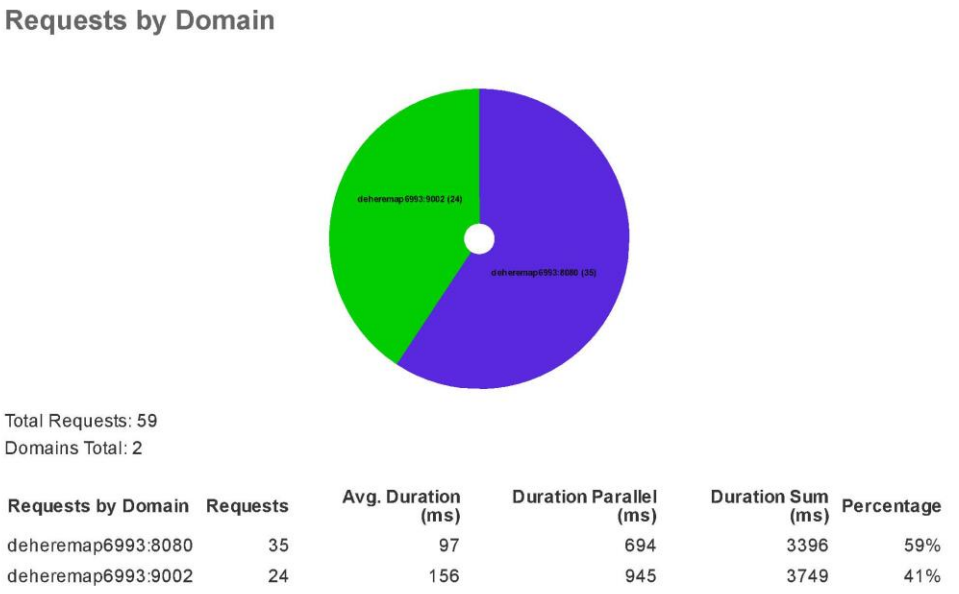


Figura 90: Tiempos de carga

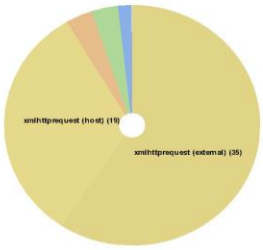




Requests by Domain	Requests	Avg. Duration (ms)	Duration Parallel (ms)	Duration Sum (ms)	Percentage
deheremap6993:8080	35	97	694	3396	59%
deheremap6993:9002	24	156	945	3749	41%

Figura 91: Tiempos de navegación

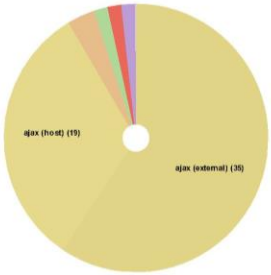
Requests by Initiator Type (host/external domain)



Total Requests: 59
Requests to Host: 24
Host: deheremap6993:9002

Requests by Initiator Type (host/external domain)	Requests	Percentage
xmlhttprequest (external)	35	59%
xmlhttprequest (host)	19	32%
script (host)	2	3.4%
css (host)	2	3.4%
link (host)	1	1.7%

Requests by File Type (host/external domain)



Total Requests: 59
Requests to Host: 24
Host: deheremap6993:9002

Requests by File Type (host/external domain)	Requests	Percentage
ajax (external)	35	59%
ajax (host)	19	32%
js (host)	2	3.4%
css (host)	1	1.7%
font (host)	1	1.7%
image (host)	1	1.7%

Request FileTypes & Initiators

FileType	Count	Count Internal	Count External	Initiator Type	Count by Initiator Type	Initiator Type Internal	Initiator Type External
css	1	1		link	1	1	
js	2	2		script	2	2	
ajax	54	19	35	xmlhttprequest	54	19	35
font	1	1		css	1	1	
image	1	1		css	1	1	

Figura 92: Clasificación de llamadas

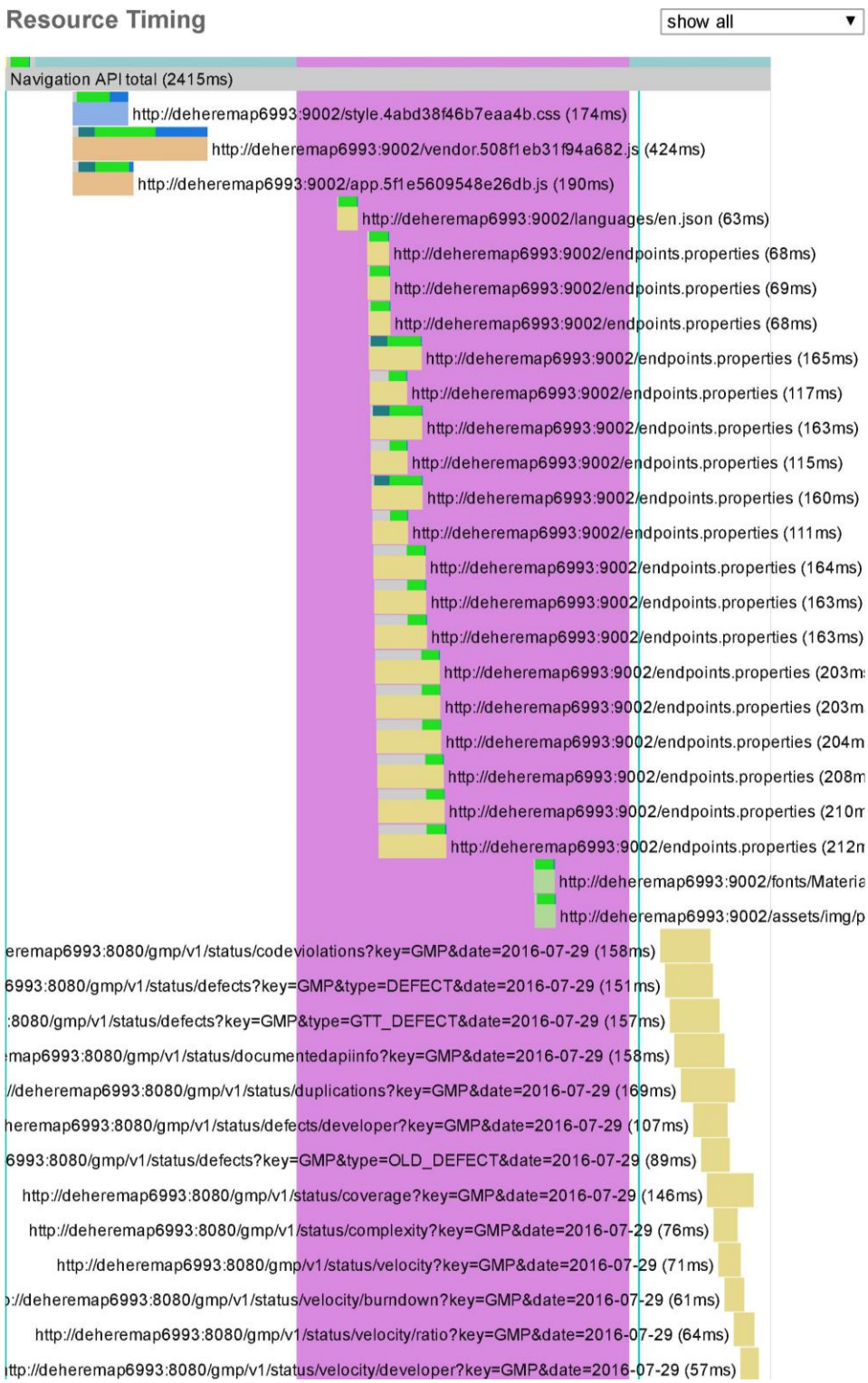


Figura 93: Tiempos de carga de los recursos

6.19 JSDOC

En este proyecto, gracias a la ayuda de “grunt” se genera documentación JSDOC automáticamente.

A continuación un ejemplo del servicio REST encargado de obtener recursos de fuentes externas:

Class: restApiService

.metricPortalFrontend. restApiService

Load resources from rest api.

Constructor

```
new restApiService($http, PATHS, endpointsService, utilService, ngDialog)
```

Parameters:

Name	Type	Description
$http	Object	Project key
PATHS	Object	Status to show
endpointsService	Object	Backend url
utilService	Object	utilService methods
ngDialog	Object	Popup dialogs

Source: services/rest-api.service.js, line 1

Methods

```
(static) getBaseUrl(trend, date, projectStatusType) → {String}
```

Return base url

Parameters:

Name	Type	Description
trend	boolean	Is trend?
date	object	Object date
projectStatusType	string	Status to show

Source: services/rest-api.service.js, line 107

Returns:

Return base url

Type
String

(static) getPath(projectStatusType) → {String}

Load existing project keys

Parameters:

Name	Type	Description
projectStatusType	string	Type of status

Source: services/rest-api.service.js, line 58

Returns:

Relative path to status type

Type
String

(static) loadEndpoints(success)

Load endpoint url if not loaded

Parameters:

Name	Type	Description
success	function	Callback function

Source: services/rest-api.service.js, line 17

(static) loadProject(key, projectStatusType, success, error, trend, date, dateEnd)

Load a selected project

Parameters:

Name	Type	Description
key	string	Project key
projectStatusType	string	Status to show
success	function	Success callback
error	function	Error callback
trend	boolean	Is trend
date	string	Starting date
dateEnd	string	End date

Source: services/rest-api.service.js, line 125

(static) loadProjectKeys(success)

Load existing project keys

Parameters:

Name	Type	Description
success	function	Callback function

Source: services/rest-api.service.js, line 41

Documentation generated by JSDoc 3.3.3 on Mon Jun 13 2016 14:56:58 GMT+0200 (Romance Daylight Time)

Figura 94: JSDOC de RestApiService