



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Álvaro Mompó Camarasa

Tutor: Sergio Sáez Barona

[2016/2017]

Resumen

El presente documento se centra en el establecimiento de una nueva filosofía de trabajo para la empresa Everis, tomando como base el uso herramientas de control de versiones que ayuden a la implantación de la integración continua. El objetivo principal es el de construir una nueva forma de desarrollar *software* capaz de instaurarse como un estándar en la empresa, con la finalidad de mejorar la eficiencia y rendimiento dentro de ella, incluyendo también una rápida respuesta ante el cliente. Para ello, se ha desarrollado una nueva arquitectura de corte ligero, capaz de establecer un ecosistema para la fácil compartición de componentes. Para demostrar su viabilidad, se ha desarrollado una prueba piloto de un proyecto real sobre una aplicación web que haga uso de sus bondades. Del mismo modo, se han desarrollado los test necesarios que aseguren su correcto funcionamiento. Para finalizar, se ha establecido un sistema de integración continua, junto a una herramienta de control de versiones, que permite la automatización de los test y despliegues.

Palabras clave: arquitectura, microservicio, test, integración, automatización.

Resum

El present document es centra en establir una nova filosofia de treball per a l'empresa Everis, agafant com a base l'ús de ferramentes de control de versions que ajuden a la implantació de la integració continua. L'objectiu principal es el de construir una nova forma de desenvolupar *software* capaç d'instaurar-se com a estàndard en l'empresa, amb la finalitat de millorar l'eficiència y rendiment dins d'ella, incloguin també una ràpida resposta davant del client. Per a això , s'ha desenvolupat una nova arquitectura de cort lleuger , capaç d'establir un ecosistema par a la fàcil compartició de components. Per a demostrar la seua viabilitat, s'ha produït una proba pilot d'un projecte real sobre una aplicació web que faja us de les sues bondats. De la mateixa manera, s'ha establert els test necessaris que assegurin el seu correcte funcionament. Per a finalitzar, instruït un sistema d'integració continua , junt a una ferramenta de de control de versions, que permet l'automatització dels test y desplegaments.

Paraules clau: arquitectura, microservici, test, integració, automatització.

Abstract

The following document is focused in the description of a brand-new work philosophy intended to be established in the company called Everis, using version control tools that help to develop a continuous integration implementation. The main goal is to build a new way of developing software capable of founding as a standard for the company, causing improvements in the efficiency and performance in addition to an agile reaction when dealing with the customer. In order to accomplish those goals, a new light architecture has been developed which is capable of offering an ecosystem that enables an easy way of sharing components. In the contemplation of proving its viability, a pilot test about a real web application project has been developed using all its characteristics. In the same vein, a set of test have been elaborated in favor to ensure the integrity of the application. To conclude, a continuous integration system along with a version control tool have been provided as the mechanism for test and deployment automation.

Keywords: architecture, microservice, test, integration, automation.

Tabla de contenidos

Contenido

1.	Introducción	11
2.	Especificación de requisitos	12
2.1	Introducción	12
2.2	Descripción general	13
2.3	Requisitos específicos	14
3.	Análisis	19
3.1	Diagrama de clases	19
3.2	Casos de uso	20
4.	Diseño	35
4.1	Arquitectura	35
4.2	Test	43
4.3	Capa de presentación	48
4.4	Capa de persistencia	53
5.	Detalles de implementación	55
5.1	Tecnologías utilizadas	55
5.2	Estructura de ficheros y directorios	67
6.	Testeo	71
6.1	Jasmine y Karma	71
6.2	Mocha y Chai	72
6.3	Protractor	73
7.	Pruebas	74
7.1	Validación	74
7.2	Pruebas de uso	75
8.	Integración continua	84
8.1	Introducción	84
8.2	Prácticas	85
8.3	Herramientas	87
8.4	Tecnologías	90
8.5	Funcionamiento	92
9.	Estructura de trabajo	94



Estudio de sistema de control de versiones y uso de los mismos en proyectos de
integración continua

10.	Conclusiones	97
11.	Bibliografía	101

Índice de figuras

1. Diagrama de clases básico.....	19
2. Diagrama de casos de uso: acceso.....	22
3. Diagrama de caso de uso: dashboard	24
4. Diagrama de caso de uso: dispositivos.....	27
5. Diagrama de casos de uso: categorías	30
6. Diagrama de casos de uso: alarmas y notificaciones	34
7. Arquitectura monolítica.....	36
8. Arquitectura de microservicios	39
9. Boceto de arquitectura general.....	41
10. Tecnologías de la arquitectura.....	42
11. V-Model	46
12. Login Mock-up.....	49
13. Sign up Mock-up.....	49
14. Dashboard superior Mock-up.....	50
15. Dashboard inferior Mock-up.....	51
16. Opciones Dashboard Mock-up.....	51
17. Dispositivos Mock-up	52
18. Categorías Mock-up	52
19. Alarmas Mock-up.....	53
20. Diagrama de clases final	54
21. Estructura de un documento MongoDB de la colección Sensors.....	54
22. Estructura de la arquitectura con las tecnologías	55
23. Estructura de la arquitectura con las tecnologías test.....	58
24. Jasmine y Karma	58
25. Mocha y Chai	59
26. Protractor	60
27. CVCS	62
28. DCVS	64
29. Bitbucket	65
30. Estructura de repositorios de la parte back-end.....	68
31. Estructura de repositorios de la parte front-end	69
32. Visualización de los test Karma y Jasmine en consola	71
33. Visualización de los test Karma y Jasmine en el navegador	72



34. Visualización de test Mocha y Chai en la consola	73
35. Versión de smartphone.....	74
36. Login	75
37. Sign up	75
38. Dashboard superior	76
39. Calidad de aire diaria.....	76
40. Calendario	77
41. Notificaciones de cambio de estado de alarmas	77
42. Notificaciones de alarmas activas	78
43. Gráfico de calidad del aire	78
44. Gráfico de la media de los contaminantes OMS	79
45. Dashboard inferior	79
46. Mapa de calor y gráfica temporal de la temperatura	80
47. Tabla de dispositivos.....	80
48. Tabla de categorías.....	81
49. Categoría añadida a las opciones del Dashboard	81
50. Tabla de alarmas	81
51. Filtrado de alarmas PM 2'5.....	82
52. Creación de una nueva alarma	82
53. Tabla con la nueva alarma creada	82
54. Sign out	83
55. Jenkins.....	87
56. Bamboo	89
57. Docker	90
58. Máquinas virtuales vs contenedores.....	91
59. Representación del funcionamiento de Ngrok	92

Índice de tablas

1. Requisitos funcionales.....	16
2. Requisitos de rendimiento.....	17
3. Caso de uso 1: iniciar sesión.....	20
4. Caso de uso 2: registrarse como usuario.....	20
5. Caso de uso 3: cerrar sesión.....	21
6. Caso de uso 4: refrescar datos del Dashboard.....	22
7. Caso de uso 5: escoger umbrales de contaminantes para el gráfico de donut.....	23
8. Caso de uso 6: escoger parámetro para el mapa de calor y la gráfica temporal.....	23
9. Caso de uso 7: crear dispositivo.....	24
10. Caso de uso 8: filtrar dispositivo.....	25
11. Caso de uso 9: editar dispositivo.....	25
12. Caso de uso 10: eliminar dispositivo.....	26
13. Caso de uso 11: crear categoría.....	27
14. Caso de uso 12: filtrar categoría.....	28
15. Caso de uso 13: editar categoría.....	28
16. Caso de uso 14: eliminar categoría.....	29
17. Caso de uso 15: crear alarma.....	30
18. Caso de uso 16: filtrar alarma.....	31
19. Caso de uso 17: editar alarma.....	31
20. Caso de uso 18: eliminar alarma.....	32
21. Caso de uso 19: mostrar notificaciones.....	33
22. Caso de uso 20: mostrar notificaciones de alarmas activas.....	33
23. Arquitectura monolítica vs arquitectura de microservicios.....	40
24. Navegadores.....	74
25. Resoluciones.....	74
26. Jenkins vs Bamboo.....	89
27. Sprint 1.....	94
28. Sprint 2.....	94
29. Sprint 3.....	95
30. Sprint 4.....	95



Glosario

- **ETSINF:** Escuela Técnica Superior de Informática
- **UPV:** Universitat Politècnica de València
- **MVC:** *Model-View-Controller* / Modelo-Vista-Controlador
- **HTTP:** *Hypertext Transfer Protocol* / Protocolo de Transferencia de Hipertexto
- **REST:** Representational State Transfer / Transferencia de Estado Representacional
- **JSON:** *JavaScript Object Notation*
- **ICA:** Índice de Calidad del Aire
- **OMS:** Organización Mundial de la Salud
- **WHO:** *World Health Organization*
- **SQL:** *Structured Query Language*
- **AMQP:** *Advanced Message Queuing Protocol*
- **CRUD:** *Create, Read, Update and Delete* / Crear, Leer, Actualizar y Borrar
- **TDD:** *Test-Driven Development* / Desarrollo guiado por pruebas de *software*
- **E2E:** *End-To-End*
- **BDD:** *Behaviours-Driven Development* / Desarrollo guiado por comportamiento
- **CVS:** *Concurrent Version System*
- **CVCS:** *Centralized Version Control System* / Sistema de Control de Versiones Centralizado
- **VPN:** *Virtual Private Network* / Red Privada Virtual
- **DVCS:** *Distributed Version Control System* / Sistema de Control de Versiones Distribuido
- **CSS:** *Cascading Style Sheets* / Hoja de Estilo en Cascada
- **VM:** *Virtual Machine* / Máquina Virtual

1. Introducción

El actual documento describe el diseño de una nueva filosofía de desarrollo *software*, para la empresa Everis, mediante una nueva arquitectura de microservicios junto a herramientas de integración continua y control de versiones.

Como alumno de la ETSINF en la UPV, solicité realizar prácticas de empresa para conocer mejor el mundo laboral y empezar a ganar experiencia como futuro ingeniero informático. Se me concedió la oportunidad de realizarlas en la empresa llamada Everis y fui asignado al sector de industria donde trabajan con tecnologías como *AngularJS*, *C++*, *Subversion*, *JavaScript*, etc. Durante mi periodo de prácticas, he detectado algunas irregularidades a la hora de mantener un flujo constante en el desarrollo de los proyectos que ocasionan pérdidas de tiempo y eficiencia que podrían resolverse con un mejor nivel de gestión. Debido esto, se me ha ocurrido proponer el uso de una arquitectura diferente a la actual, capaz de potenciar la compartición de código para conseguir una mayor reutilización. Al mismo tiempo, implantar una herramienta de integración continua capaz de automatizar tareas tan repetitivas como la realización de test y despliegues. Este documento recoge todas las ideas necesarias para establecer este nuevo modelo de desarrollo de proyectos, así como una prueba general que implementa todo el flujo de trabajo.

Con el fin de poner a prueba la nueva arquitectura, se va a desarrollar una aplicación web piloto sobre un proyecto real de Everis, en la que aplicar la automatización de test y despliegues. El primer punto de todos es la especificación de requisitos, que ayuda a poner en contexto el ámbito de esta prueba y determinar los pasos que se deben tener en cuenta a la hora de la implementación. Una vez descritos todos los requisitos, se establece un análisis mediante la estructura de datos que maneja la aplicación y los casos de uso que deben ser incluidos. A continuación, para la etapa de diseño, se realiza una descripción del papel de las tecnologías y pruebas test, discutiendo los diferentes tipos y cuáles son los más adecuados para este desarrollo. Del mismo modo, se presentan los bocetos y estructura final de datos que se utilizan en la implementación de la prueba piloto. Más adelante, se exponen las diferentes tecnologías y herramientas que se usan en la nueva arquitectura, ya sea para el desarrollo web, test o control de versiones. Después de esto, se ofrece una descripción de la realización de test sobre la aplicación web, así como algunos ejemplos de su ejecución. Con todos los elementos para el desarrollo de la aplicación en escena, se expone una demo de ella a través de capturas realizadas sobre la versión final, mostrando todas las funcionalidades que esta ofrece. Como objetivo final, se realiza un estudio en relación a la integración continua y las herramientas que permiten llevarla a cabo, incluyendo una pequeña descripción del proceso aplicado en este proyecto. Para acabar, se expone la organización de trabajo seguida durante el desarrollo de este trabajo de fin de grado así como las conclusiones que se han sacado de este.

2. Especificación de requisitos

El capítulo dos presenta la especificación de requisitos para la aplicación web piloto, haciendo usos de la nueva arquitectura propuesta para la empresa Everis. En él, se analizan los diferentes aspectos a considerar para el desarrollo de la aplicación desde el ámbito de esta, hasta los diferentes requisitos que se deben tener en cuenta para su implementación.

2.1 Introducción

Antes de comenzar con la especificación de requisitos, es importante saber a qué a hacen referencia estos. Como se ha explicado durante en punto número uno de este documento, el proyecto consiste en la implementación de una nueva arquitectura estándar para el desarrollo de proyectos, sobre la que aplicar un sistema de integración continua que permita la automatización de test y despliegue en entornos de desarrollo y producción, todo eso haciendo uso de un sistema de control de versiones que ayude a la gestión de código.

Para demostrar su funcionamiento, se ha desarrollado una aplicación web conceptual que haga uso de la arquitectura propuesta y permita el desarrollo de test y su posterior automatización mediante el uso de una herramienta de integración continua. Debido a esto, es crucial tener en cuenta que la especificación de requisitos descrita a continuación hace referencia a dicha aplicación web respecto a la nueva arquitectura y no a esta en sí.

2.1.1 Propósito

Esta especificación de requisitos aborda la definición de una aplicación web piloto capaz de satisfacer los requisitos de la arquitectura propuesta para el desarrollo *software* dentro de la compañía Everis. Esta nueva arquitectura, descrita en profundidad más a delante en este documento, hace hincapié en ofrecer una alta escalabilidad mediante la modularidad de sus partes. Los requisitos descritos para la web están sujetos a los principios marcados por dicha arquitectura.

2.1.2 Ámbito

El ámbito de este este producto se sitúa dentro de lo relacionado con el medioambiente y la contaminación. Es un hecho que nuestro paso sobre la tierra está dejando una huella de contaminación nunca antes vista. Las grandes ciudades se caracterizan por ser perfectos focos de emisión de gases nocivos que pueden resultar perjudiciales para la salud de sus habitantes, ocasionando malestares y enfermedades que no sería posible contraer si no se diesen este tipo de condiciones.

Para saber a qué nos enfrentamos y averiguar nuestro nivel de impacto, es posible contar con sensores que se encarguen de captar las sustancias perjudiciales que emitimos a nuestro entorno con el fin de estudiarlas y poder aplicar soluciones que las mitiguen. Estos sensores únicamente actúan como receptores de datos por lo que, posteriormente, es necesario una gestión de los datos recogidos y una visualización cómoda de ellos que permita llevar un registro exhaustivo dando la posibilidad de actuar sobre la áreas afectadas.

2.1.3 Acrónimos

- **HTTP:** *Hypertext Transfer Protocol* / Protocolo de Transferencia de Hipertexto
- **IOPS:** *Input/Output Operations Per Second*
- **ICA:** Índice de Calidad del Aire
- **OMS:** Organización Mundial de la Salud
- **AMQP:** *Advanced Message Queuing Protocol*

2.1.4 Visión de conjunto

El propósito de esta aplicación web es el de ofrecer una visualización gráfica de los datos recibidos por sensores medioambientales repartidos por las ciudades. Al mismo tiempo, ofrece una capa de personalización para el usuario de manera que este pueda configurar nuevos dispositivos, controlar las ciudades sobre las que desea visualizar información y contar con un sistema de alarmas a la carta que le notifiquen acorde con sus intereses. Todo esto debe hacer uso de la nueva arquitectura genérica propuesta para la futura creación de proyectos. El nombre decidido para el proyecto web es *On Air*.

2.2 Descripción general

2.2.1 Perspectiva del producto

La experiencia dentro de la aplicación web viene condicionada por el tipo de dispositivo encargado de enviar la información y el número de sensores que se encuentren distribuidos por la ciudad. Existen dispositivos capaces de medir un distinto número de contaminantes y condiciones medioambientales como la temperatura, humedad o presión. Otros, se encuentran más limitados y solo recogen información de un pequeño grupo de parámetros.

También afecta la zona en la que se sitúen los sensores. Existe un uso que consiste en repartir un gran número de sensores por distintos puntos de la ciudad, estableciéndolos como su lugar fijo de medición. Es fundamental estudiar las zonas en las que se deba colocar cada sensor ya que, si por ejemplo se instalan principalmente en zonas verdes, estos emitirán valores que afectarán a la media real de contaminación. Otro método es el de acoplar el sensor a bicicletas

que se dediquen a realizar diferentes rutas por la ciudad, recogiendo datos cada cierta distancia. También se debe tener en cuenta la frecuencia con la que los sensores envían información nueva.

2.2.2 Características del usuario

El usuario al que está destinada esta aplicación es de un perfil científico/tecnológico o de interés público. Por un lado permite la visualización de datos a aquellos que se dediquen al estudio del medioambiente y contaminación. Por el otro, se encuentran aquellos usuarios que cuenten con un dispositivo de medición de contaminantes y necesiten una plataforma sobre la que ver los resultados recogidos por estos. De la misma manera, instituciones como los ayuntamientos pueden utilizar este servicio para establecer un control de la situación medioambiental en la que se encuentra su ciudad o pueblo.

2.2.3 Dependencias

Las dependencias vienen dadas básicamente por el tipo de dispositivo que esté enviando datos, ya que la información que este reciba marca los resultados finales que se observen en la web.

A parte de los dispositivos, la interpretación de datos también está limitada a los umbrales y formulas empleados para el tratamiento de dicha información. Por ejemplo, a la hora de calcular los niveles registrados para cada contaminante, existen un modelo de medidas establecido por Bruselas y otro por la OMS. Es trascendental tener en cuenta ambos umbrales ya que su uso puede variar en referencia al estudio que se desee realizar.

2.3 Requisitos específicos

2.3.1 Requisitos de interfaz externa

Interactividad del usuario:

- El sistema debe proveer una interfaz amigable y semejante a las aplicaciones web de hoy en día.
- Las acciones a realizar no deben necesitar más que unos pocos pasos para su ejecución.
- La navegación entre ventanas debe ser clara y simple.

Interfaz del usuario:

- Acorde con las bases establecidas y proporcionadas, el sistema debe ofrecer una interfaz de navegador tanto para ordenadores como para dispositivos *Smartphone*.
- Los *layouts* y elementos que la compongan deben responder dinámicamente ante los cambios de tamaño que sufra la ventana.
- El diseño de la interfaz debe ser claro y limpio, sin demasiados elementos en pantalla al mismo tiempo.

2.3.2 Requisitos funcionales

Req#	Requisito	Prioridad	Comentario
LOGIN_01	El usuario debe ser capaz de iniciar sesión con los datos introducidos durante el registro	1	
SIGN_UP_02	El usuario puede crear un nuevo perfil con sus datos y, de esa forma, tener acceso a la aplicación	1	
ICA_MÁXIMO_DIARIO_03	El sistema debe calcular y refrescar diariamente el índice de calidad máxima del aire	2	El índice de calidad del aire máximo viene dado por el máximo correspondiente a uno de los contaminantes
CÁLCULO_DE_ICA_04	El sistema debe calcular la calidad de aire correspondiente a cada contaminante y mostrarlo en una gráfica de araña	1	
CÁLCULO_DE_LA_MEDIA_CANTIDAD_PARA_CADA_CONTAMINANTE_05	El sistema debe calcular la cantidad media para cada contaminante en base a los umbrales y mostrarlo en una gráfica <i>Gauge</i>	1	Existen dos tipos de umbrales, los ofrecidos por Bruselas y los de la OMS
UMBRALES_DE_BRUSELAS_O_LA_OMS	El usuario debe ser capaz de alternar entre los valores de los umbrales establecidos por Bruselas o la OMS	1	

MAPA_DE_CALOR_06	El sistema debe mostrar sobre un mapa los valores de cada parámetro con su correspondiente intensidad, para cada sensor instalado en dicha categoría	1	La intensidad de cada punto representado en el mapa es proporcional al valor del parámetro en dicho sensor
VALORES_DE_CONCENTRACION_DEL_TIEMPO_7	Le sistema debe mostrar los valores para para el parámetro seleccionado en una gráfica lineal temporal	1	
SELECCIÓN_DEL_PARÁMETRO_08	El usuario debe ser capaz de seleccionar el parámetro sobre el que desea visualizar información en el mapa de calor y gráfica temporal	1	
GESTIÓN_DE_DISPOSITIVOS_09	El usuario debe ser capaz de crear, buscar, editar o borrar dispositivos	1	
GESTIÓN_DE_CATEGORIAS_10	El usuario debe ser capaz de crear, buscar, editar o borrar categorías	1	
GESTIÓN_DE_ALARMAS_11	El usuario debe ser capaz de crear, buscar, editar o borrar alarmas.	1	
HISTORIAL_DE NOTIFICACIONES_12	El sistema debe comprobar el estado de las alarmas y enviar una notificación cada vez que se active o desactive una alarma	1	
NOTIFICACIONES_DE_ALARMAS_ACTIVAS_13	El sistema debe mantener un contador de notificaciones que alerte al usuario únicamente sobre aquellas alarmas que se encuentren actualmente activas	1	

1. Requisitos funcionales

2.3.3 Requisitos de rendimiento

Para entender los requisitos de rendimiento de la aplicación sobre la nueva arquitectura es vital entender el cambio de mentalidad necesario. Esta arquitectura está pensada para utilizar *hosting cloud*, lo que implica una modificación del paradigma original en el que para mejorar el rendimiento se aumentan los recursos *hardware* de la pieza que lo solicite. En mi caso, el

rendimiento viene determinado por el número de llamadas que se realicen, siendo el proveedor de servicios *cloud* el encargado de gestionar el tema *hardware*. Con este tipo de arquitectura la máquina no importa, lo que determina el rendimiento es el servicio ofrecido por el proveedor *cloud*.

Componente	Especificaciones	Características
Front-end	1 procesador de 1 <i>Core</i> con 1,75GB de RAM (B1 de <i>Azure</i>)	Soportar entre 30.000 y 50.000 llamadas por segundo
Back-end	1 procesador de 1 <i>Core</i> con 1,75GB de RAM (B1 de <i>Azure</i>)	Soportar entre 30.000 y 50.000 llamadas por segundo
lot Hub	El S1 tiene una capacidad de 400.000 mensajes al día	Puede consumir 1111 KB/minuto (ingesta de datos)
Base de datos	10GB que es lo mínimo que proporciona <i>Azure</i> , siendo suficiente para <i>MongoDB</i>	1000 IOPS por colección (más o menos)

2. Requisitos de rendimiento

2.3.4 Atributos de sistema software

2.3.4.1 Disponibilidad

- La aplicación debe estar en funcionamiento las veinticuatro horas del día los siete días de la semana.
- La disponibilidad de nuevos datos viene dada por la tasa de refresco que ofrezca cada dispositivo.
- Las áreas cubiertas por la aplicación son aquellas que cuentan con algún dispositivo registrado en el sistema. Ejemplo: si un ayuntamiento desea darle uso en su ciudad, previamente deberá instalar sensores en los puntos que desee y dar de alta en el sistema el tipo de dispositivo usado.
- El impacto por fallo viene determinado por el origen de este. Si se encuentra en la aplicación web, es posible que no se puedan visualizar los datos recibidos por los sensores. En caso contrario, si el fallo se origina en el sensor, la web debe seguir en funcionamiento solo que los datos enviados por este sensor no aparecerán.



2.3.4.2 Seguridad

En el tema de seguridad necesario numerar varios puntos:

- El registro de dispositivos para el uso de sus datos en la página web requiere de un servidor *IoT Hub* para validar y dar de alta dispositivos mediante el uso de *tokens*.
- Uso de conexión no cifrada AMQP en vez de HTTPS.
- Uso de *oauth2 basic* para el *login* inicial de usuarios en la parte *front-end*.

2.3.4.3 Mantenimiento

Debido a la naturaleza de la arquitectura y el desacoplamiento que ofrece entre sus elementos, el mantenimiento del sistema resulta bastante sencillo. En caso de que algún elemento falle, el resto de la web no debería verse afectada y tendría que continuar en ejecución mostrando el resto de funcionalidades mientras se arregla el fallo que lo haya originado. Del mismo modo, la identificación y tratamiento de errores se convierte en una tarea más simple dado que la estructura del código permite detectar con facilidad la fuente del error.

2.3.4.4 Portabilidad

La portabilidad de la aplicación es algo innato ofrecido por la nueva arquitectura. Una única implementación permite la ejecución de la web en distintos navegadores y dispositivos con diferentes tamaños y resoluciones.

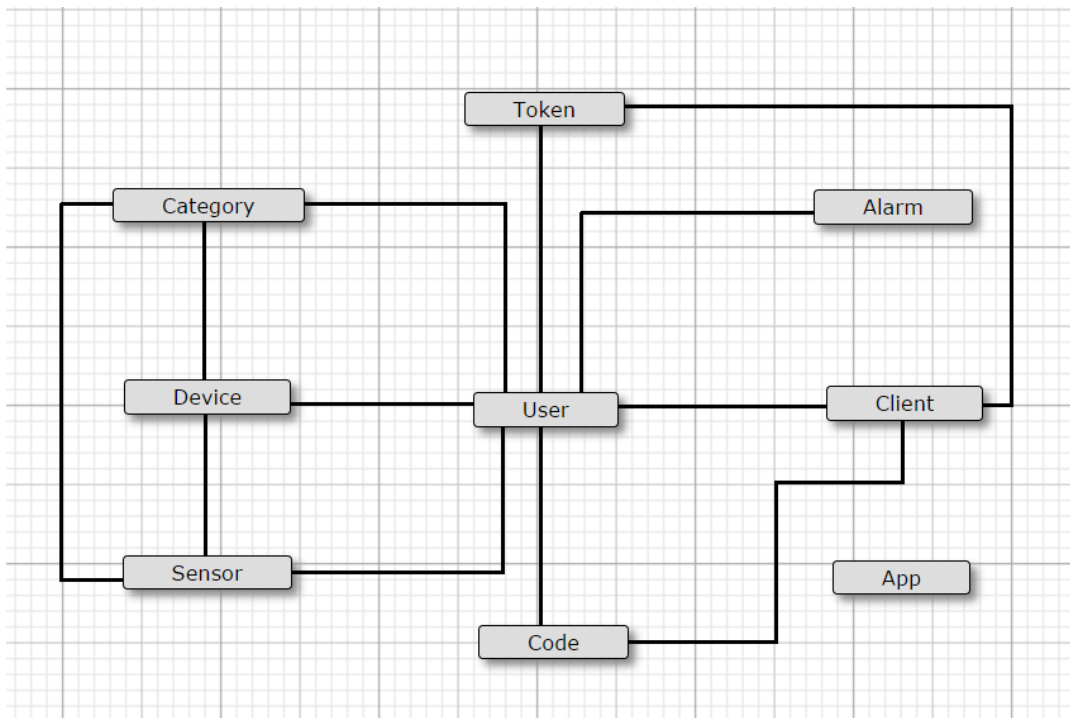
3. Análisis

En el tercer capítulo se expone el panorama general pensado para la disposición de datos de la aplicación web mediante un diagrama de clases con algunos matices de cara a la arquitectura elegida. Más tarde se describen los casos de uso a implementar en la prueba piloto, clasificados por temas y junto a su representación en diagramas de uso.

3.1 Diagrama de clases

El tratamiento de datos se lleva a cabo mediante una base de datos *MongoDB* de tipo *NoSQL*. Este tipo bases de datos elimina el uso de tablas de registros relacionales mediante las cuales se establecen relaciones que restringen su interacción. En sustitución, *MongoDB* trabaja sobre el manejo de colecciones de documentos. Estas colecciones reemplazan a las entidades encontradas en bases de datos relacionales mientras que los documentos de cada una representan los registros que conforman las entidades.

Es importante tener claro que *MongoDB* plantea relaciones entre colecciones, pero no existen entidades de relación con restricciones como en las bases de datos *SQL*. El diagrama inferior representa las colecciones necesarias para la parte de persistencia de la aplicación *On Air*. Para ayudar a su visualización, se han establecido uniones entre las diferentes colecciones pero se debe tener en cuenta que estas no existen como tal. Estas relaciones son una mera representación para el entendimiento de los datos, que no aplican ninguna restricción a la hora de crear nuevas colecciones o registros.



1. Diagrama de clases básico



3.2 Casos de uso

A continuación se describen los casos de uso a implementar en la web piloto, haciendo uso de las bases establecidas por la arquitectura nueva y teniendo siempre en cuenta las características de esta. La web consta de un total de veinte casos agrupados en diferentes temáticas.

3.2.1 Acceso

Casos de uso que representan las vías de acceso a la aplicación web.

CU-1	Iniciar sesión	
Descripción	El usuario inicia sesión introduciendo el email y contraseña con los que está registrado	
Actores	Usuario registrado	
Precondiciones	Constar como usuario registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario introduce su correo electrónico 2. El usuario introduce su contraseña 3. El usuario pulsa <i>Log in</i> 	
Postcondiciones	Se muestra la ventana de <i>Dashboard</i> . Los datos se muestran en base a las categorías que tenga asociadas. El usuario tiene acceso a todas las funcionalidades de la web	
Excepciones	Paso	Acción
	1	El usuario no introduce el email o este es erróneo
		E.1
	2	El usuario no introduce el email o este es erróneo
		E.1

3. Caso de uso 1: iniciar sesión

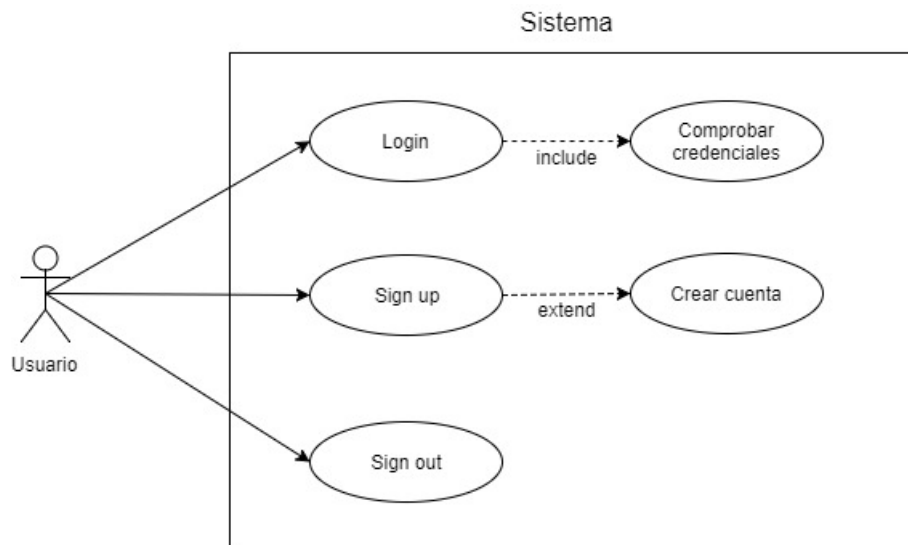
CU-2	Registrarse como usuario
Descripción	El usuario se registra en la aplicación con sus datos personales

Actores	Usuario no registrado	
Precondiciones	No poseer una cuenta con las mismas credenciales que la que se pretende crear	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa la opción <i>Sign up</i> 2. El usuario introduce su nombre 3. El usuario introduce su apellido/s 4. El usuario introduce su email 5. El usuario introduce su contraseña 6. El usuario confirma su contraseña 7. El usuario pulsa la opción de <i>Sign up</i> 	
Postcondiciones	Se muestra la ventana de <i>Login</i> . El usuario queda registrado. Los datos se muestran en base a las categorías que tenga asociadas. El usuario tiene acceso a todas las funcionalidades de la web	
Excepciones	Paso	Acción
	4	El campo email no contiene un email
		E.1
	5 ó 6	Las contraseñas no coinciden
		E.1

4. Caso de uso 2: registrarse como usuario

CU-3	Cerrar session
Descripción	El usuario cierra su sesión
Actores	Usuario
Precondiciones	Haber iniciado sesión en la aplicación
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa sobre su nombre 2. El sistema despliega un menú 3. El usuario pulsa la opción <i>Sign out</i>
Postcondiciones	Se cierra la sesión actual del usuario y se muestra la ventana de <i>Login</i>

5. Caso de uso 3: cerrar sesión



2. Diagrama de casos de uso: acceso

3.2.2 Dashboard

Casos de uso que representan las acciones que se pueden realizar dentro de la ventana del *Dashboard* o principal, teniendo en cuenta la interacción con los datos recogidos por los sensores.

CU-4	Refrescar datos del <i>Dashboard</i>	
Descripción	El usuario selecciona las fechas y ciudad sobre las que desea obtener información. El sistema se encarga de refrescar los datos en base a los parámetros seleccionados	
Actores	Usuario	
Precondiciones	Estar registrado y tener al menos una categoría asignada	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa sobre el campo calendario 2. El usuario selecciona alguno de los periodos predefinidos o elige uno propio marcando los días en el calendario. 3. El usuario marca la categoría que desee 4. El usuario pulsa la opción de <i>Refresh</i> 5. El sistema actualiza la información del <i>Dashboard</i> con los parámetros seleccionados 	
Postcondiciones	Se muestra la ventana de <i>Dashboard</i> con los datos actualizados acorde a las opciones seleccionadas por el usuario	
Excepciones	Paso	Acción
	2	El usuario selecciona un período en el que no hay datos

		E.1	El sistema muestra los gráficos vacíos
	2		El usuario no tiene ninguna categoría asignada
		E.1	El sistema muestra los gráficos vacíos

6. Caso de uso 4: refrescar datos del Dashboard

CU-5	Escoger umbrales de contaminantes para el gráfico de donut		
Descripción	El usuario alterna entre los umbrales establecidos por la OMS o Bruselas		
Actores	Usuario		
Precondiciones	Estar registrado, tener al menos una categoría asignada y haber seleccionado un rango de fechas con datos		
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa la opción <i>WHO</i> 2. El sistema refresca el gráfico aplicando los umbrales establecidos por la OMS 3. El usuario pulsa la opción <i>Brussels</i> 4. El sistema refresca el gráfico aplicando los umbrales establecidos por Bruselas 		
Postcondiciones	Se muestra el gráfico con los datos seleccionados		
Excepciones	Paso	Acción	
	2	No existen datos para las fechas seleccionadas	
		E.1	El sistema muestra el gráfico vacío
	4	No existen datos para las fechas seleccionadas	
		E.1	El sistema muestra el gráfico vacío

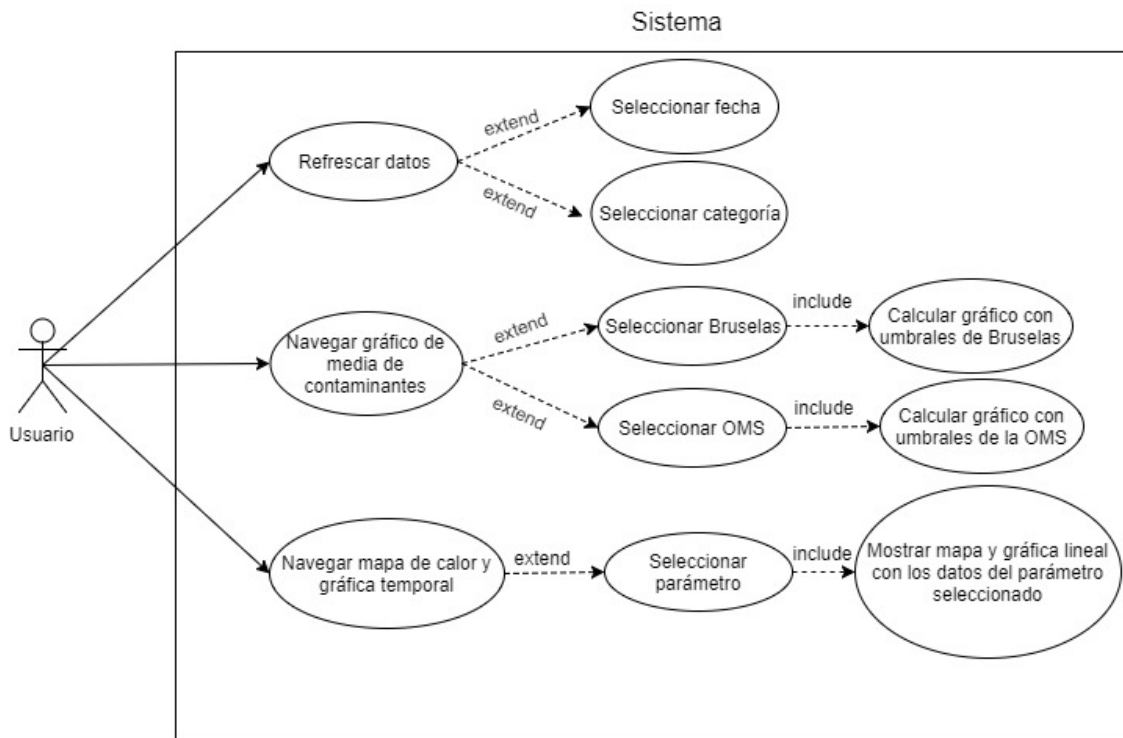
7. Caso de uso 5: escoger umbrales de contaminantes para el gráfico de donut

CU-6	Escoger parámetro para el mapa de calor y la gráfica temporal		
Descripción	El usuario alterna entre los parámetros de datos ofrecidos por el sensor: PM 2'5, CO, NO2, O3, <i>Humidity</i> , <i>Temperature</i> y <i>Pressure</i>		
Actores	Usuario		
Precondiciones	Estar registrado, tener al menos una categoría asignada y haber seleccionado un rango de fechas con datos		
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario pulsa cualquier parámetro 2. El sistema refresca el mapa de calor y gráfica temporal con los datos referentes al parámetro pulsado 		



Postcondiciones	Se muestra el gráfico con los datos seleccionados	
Excepciones	Paso	Acción
	2	No existen datos para las fechas seleccionadas
	E.1	El sistema muestra el gráfico vacío

8. Caso de uso 6: escoger parámetro para el mapa de calor y la gráfica temporal



3. Diagrama de caso de uso: dashboard

3.2.3 Dispositivos

Casos de uso que representan las operaciones CRUD a realizar sobre los dispositivos.

CU-7	Crear dispositivo
Descripción	El usuario da de alta un nuevo dispositivo
Actores	Usuario
Precondiciones	Estar registrado
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Devices</i> del menú lateral 2. El sistema abre la ventana de los dispositivos 3. El usuario pulsa la opción de <i>Añadir</i> 4. El usuario rellena los parámetros del dispositivo

	<ol style="list-style-type: none"> 5. El usuario pulsa la opción de <i>tick</i> 6. El sistema añade el nuevo dispositivo a la tabla y la base de datos
Postcondiciones	Se actualiza la tabla de <i>Devices</i> con una nueva fila que contiene los datos del nuevo dispositivo

9. Caso de uso 7: crear dispositivo

CU-8	Filtrar dispositivo	
Descripción	El usuario filtra entre los dispositivos existentes	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Devices</i> del menú lateral 2. El sistema abre la ventana de los dispositivos 3. El usuario introduce la información en el campo o campos superiores correspondientes a cada atributo del dispositivo 4. El sistema actualiza la tabla acorde con la búsqueda 	
Postcondiciones	Cuando el usuario borra los datos de los campos de búsqueda, se muestra la tabla original	
Excepciones	Paso	Acción
	2	No existe ningún registro en la tabla <i>Devices</i>
		E.1 El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "
	3	No existe ningún dispositivo que coincida con la información dada
E.1 El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "		

10. Caso de uso 8: filtrar dispositivo

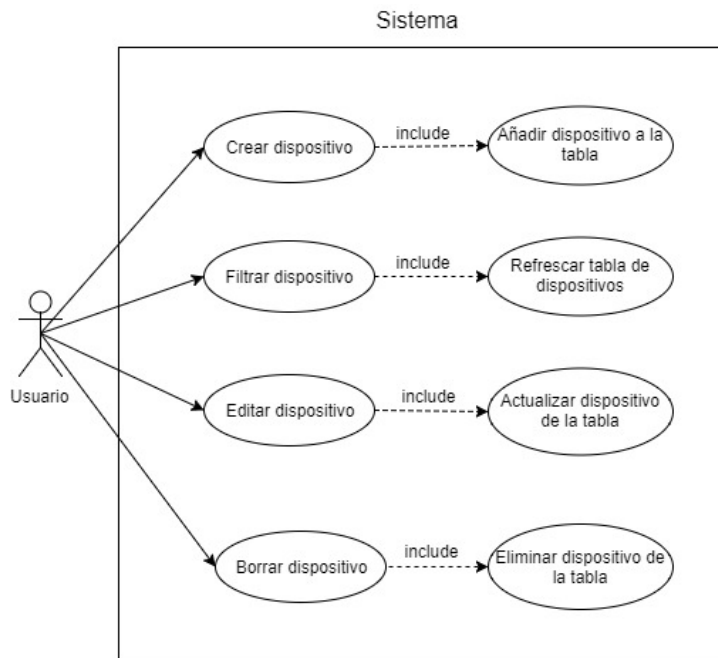
CU-9	Editar dispositivo
Descripción	El usuario edita un dispositivo ya existente
Actores	Usuario
Precondiciones	Estar registrado

Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Devices</i> del menú lateral 2. El sistema abre la ventana de los dispositivos 3. El usuario pulsa el icono de edición del dispositivo que desee modificar 4. El usuario introduce las modificaciones deseadas 5. El usuario pulsa la opción de <i>tick</i> 6. El sistema actualiza el dispositivo modificado 	
Postcondiciones	La tabla muestra el dispositivo modificado	
Excepciones	Paso	Acción
	1	No existe ningún registro en la tabla <i>Devices</i>
	E.1	El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

11. Caso de uso 9: editar dispositivo

CU-10	Eliminar dispositivo	
Descripción	El usuario elimina un dispositivo	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Devices</i> del menú lateral 2. El sistema abre la ventana de los dispositivos 3. El usuario pulsa el icono de borrado 4. El sistema pregunta por una confirmación 5. El usuario pulsa la opción de <i>Aceptar</i> 6. El sistema borra el dispositivo seleccionado 	
Postcondiciones	La tabla se muestra sin el dispositivo seleccionado	
Excepciones	Paso	Acción
	2	No existe ningún registro en la tabla <i>Devices</i>
	E.1	El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

12. Caso de uso 10: eliminar dispositivo



4. Diagrama de caso de uso: dispositivos

3.2.4 Categorías

Casos de uso que representan las operaciones CRUD a realizar sobre las categorías.

CU-11	Crear categoría
Descripción	El usuario da de alta una nueva categoría
Actores	Usuario
Precondiciones	Estar registrado
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Categorías</i> del menú lateral 2. El sistema abre la ventana de las categorías 3. El usuario pulsa la opción de <i>Añadir</i> 4. El usuario rellena los parámetros de la categoría 5. El usuario pulsa la opción de <i>tick</i> 6. El sistema añade la nueva categoría a la tabla y la base de datos
Postcondiciones	Se actualiza la tabla de <i>Categorías</i> con una nueva fila que contiene los datos de la nueva categoría

13. Caso de uso 11: crear categoría

CU-12	Filtrar categoría	
Descripción	El usuario filtra entre las categorías existentes	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Categories</i> del menú lateral 2. El sistema abre la ventana de las categorías 3. El usuario introduce la información en el campo o campos superiores correspondientes a cada atributo de la categoría 4. El sistema actualiza la tabla acorde con la búsqueda 	
Postcondiciones	Cuando el usuario borra los datos de los campos de búsqueda, se muestra la tabla original	
Excepciones	Paso	Acción
	2	No existe ningún registro en la tabla <i>Categories</i>
		E.1
	3	No existe ninguna categoría que coincida con la información dada
E.1		El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

14. Caso de uso 12: filtrar categoría

CU-13	Editar categoría	
Descripción	El usuario edita una categoría ya existente	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Categories</i> del menú lateral 2. El sistema abre la ventana de las categorías 3. El usuario pulsa el icono de edición de la categoría que desee modificar 4. El usuario introduce las modificaciones deseadas 5. El usuario pulsa la opción de <i>tick</i> 6. El sistema actualiza la categoría modificada 	
Postcondiciones	La tabla muestra la categoría modificada	

Excepciones	Paso	Acción
	1	
E.1		El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

15. Caso de uso 13: editar categoría

CU-14	Eliminar categoría	
Descripción	El usuario elimina una categoría	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Categories</i> del menú lateral 2. El sistema abre la ventana de las categorías 3. El usuario pulsa el icono de borrado 4. El sistema pregunta por una confirmación 5. El usuario pulsa la opción de <i>Aceptar</i> 6. El sistema borra la categoría seleccionada 	
Postcondiciones	La tabla se muestra sin la categoría seleccionada	
Excepciones	Paso	Acción
	2	
E.1		El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

16. Caso de uso 14: eliminar categoría



5. Diagrama de casos de uso: categorías

3.2.5 Alarmas

Casos de uso que representan las operaciones CRUD a realizar sobre las alarmas.

CU-15	Crear alarma
Descripción	El usuario da de alta una nueva alarma
Actores	Usuario
Precondiciones	Estar registrado
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Alarms</i> del menú lateral 2. El sistema abre la ventana de las alarmas 3. El usuario pulsa la opción de <i>Añadir</i> 4. El usuario rellena los parámetros de la alarma 5. El usuario pulsa la opción de <i>tick</i> 6. El sistema añade la nueva alarma a la tabla y la base de datos

Postcondiciones	Se actualiza la tabla de <i>Alarms</i> con una nueva fila que contiene los datos de la nueva alarma
------------------------	---

17. Caso de uso 15: crear alarma

CU-16	Filtrar alarma	
Descripción	El usuario filtra entre las alarmas existentes	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Alarms</i> del menú lateral 2. El sistema abre la ventana de las alarmas 3. El usuario introduce la información en el campo o campos superiores correspondientes a cada atributo de la alarma 4. El sistema actualiza la tabla acorde con la búsqueda 	
Postcondiciones	Cuando el usuario borra los datos de los campos de búsqueda, se muestra la tabla original	
Excepciones	Paso	Acción
	2	No existe ningún registro en la tabla <i>Alarms</i>
		E.1
	3	No existe ninguna alarma que coincida con la información dada
E.1		El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

18. Caso de uso 16: filtrar alarma

CU-17	Editar alarma
Descripción	El usuario edita una alarma ya existente
Actores	Usuario
Precondiciones	Estar registrado

Secuencia normal	<ol style="list-style-type: none"> 1. El usuario selecciona la opción <i>Alarms</i> del menú lateral 2. El sistema abre la ventana de las alarmas 3. El usuario pulsa el icono de edición de la alarma que desee modificar 4. El usuario introduce las modificaciones deseadas 5. El usuario pulsa la opción de <i>tick</i> 6. El sistema actualiza la alarma modificada 	
Postcondiciones	La tabla muestra la alarma modificada	
Excepciones	Paso	Acción
	1	No existe ningún registro en la tabla <i>Alarms</i>
	E.1	El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

19. Caso de uso 17: editar alarma

CU-18	Eliminar alarma	
Descripción	El usuario elimina una alarma	
Actores	Usuario	
Precondiciones	Estar registrado	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la opción <i>Alarms</i> del menú lateral
	2	El sistema abre la ventana de las alarmas
	3	El usuario pulsa el icono de borrado
	4	El sistema pregunta por una confirmación
	5	El usuario pulsa la opción de <i>Aceptar</i>
	6	El sistema borra la alarma seleccionada
Postcondiciones	La tabla se muestra sin la alarma seleccionada	
Excepciones	Paso	Acción
	2	No existe ningún registro en la tabla <i>Alarms</i>
	E.1	El sistema muestra una única fila en la tabla con el mensaje " <i>Data not found</i> "

20. Caso de uso 18: eliminar alarma

3.2.5 Notificaciones

Casos de uso que representan la gestión de notificaciones que realiza el sistema en base a las alarmas creadas por el usuario.

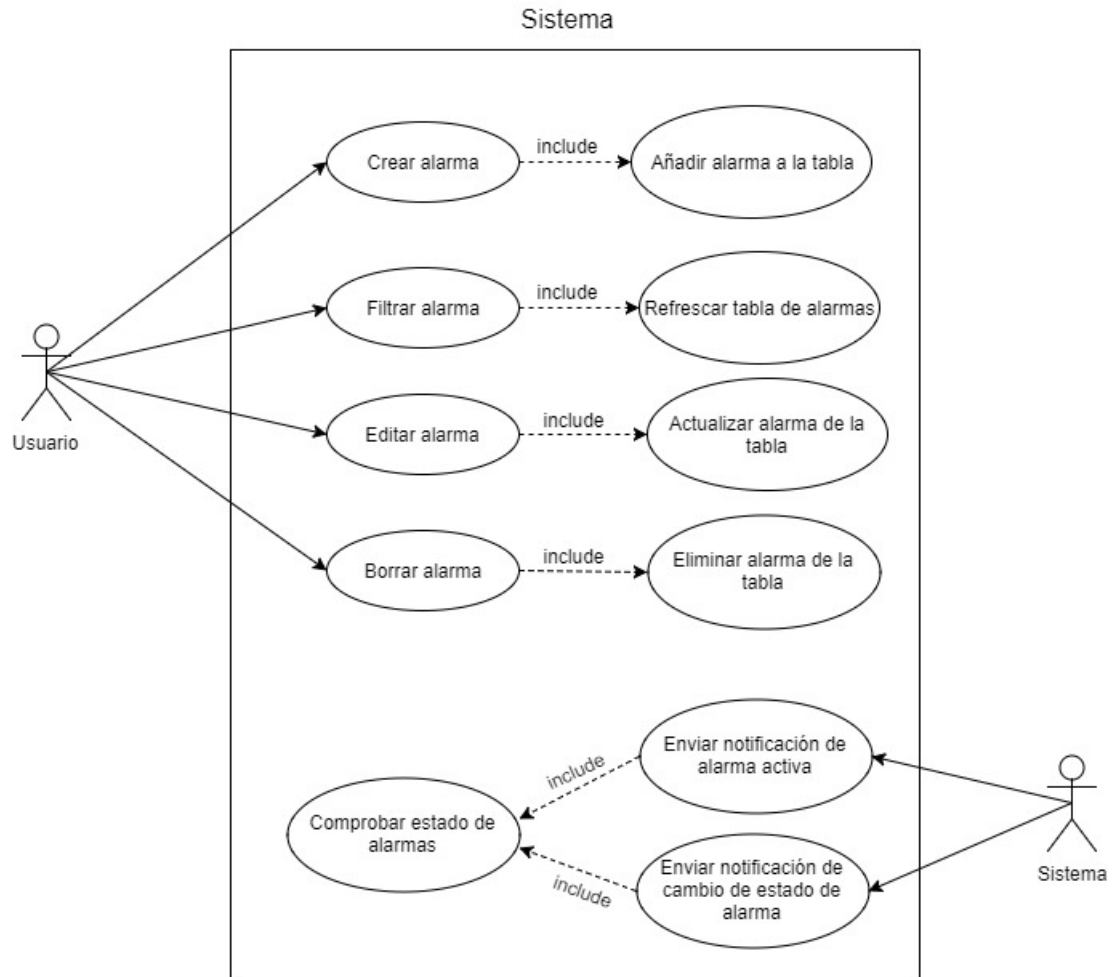
CU-19	Mostrar notificaciones	
Descripción	El sistema envía una notificación cada vez que una alarma se ha activa o desactiva	
Actores	Sistema	
Precondiciones	Tener alguna alarma creada	
Secuencia normal	<ol style="list-style-type: none"> 1. El sistema comprueba las alarmas creadas 2. El sistema envía una notificación de activación o desactivación 	
Postcondiciones	El contador de notificaciones se incrementa y el usuario puede ver la actividad de sus alarmas	
Excepciones	Paso	Acción
	2	No se cumple la condición de la alarma
	E.1	El sistema no envía ninguna notificación
	E.2	El sistema muestra el mensaje <i>"You have no notifications"</i>

21. Caso de uso 19: mostrar notificaciones

CU-20	Mostrar notificaciones de alarmas activas	
Descripción	El sistema mantiene un registro de alarmas activas	
Actores	Sistema	
Precondiciones	Tener alguna alarma creada	
Secuencia normal	<ol style="list-style-type: none"> 1. El sistema comprueba las alarmas creadas 2. El sistema mantiene una notificación por cada alarma activa y elimina aquellas correspondientes a alarmas que se desactiven 	
Postcondiciones	El contador de notificaciones se incrementa y el usuario puede sus alarmas activas	
Excepciones	Paso	Acción
	2	No hay alarmas activas

	E.1	El sistema no muestra ninguna notificación de alarma activa
	E.2	El sistema muestra el mensaje <i>"You have no active alarms"</i>

22. Caso de uso 20: mostrar notificaciones de alarmas activas



6. Diagrama de casos de uso: alarmas y notificaciones

4. Diseño

El cuarto capítulo comienza con la introducción del concepto de arquitectura dentro del mundo del *software*, dando a conocer su papel e importancia en cualquier proyecto de este ámbito. Al mismo tiempo, se presentan dos de las arquitecturas más influyentes: la monolítica, algo más antigua y los microservicios, arquitectura emergente y clave dentro de este proyecto. Más adelante se incluye la presentación del *framework* elegido para dar soporte a esta arquitectura de microservicios y la prueba piloto que pretende demostrar su viabilidad, así como su metodología e infraestructura. Después se explica otro de los componentes cruciales del proyecto, el desarrollo de test. Se expone la importancia de aplicar estas pruebas a las arquitecturas de microservicios y aquellas que han sido seleccionadas como las más adecuadas para cubrir la implementación de cualquier aplicación que haga uso de la nueva arquitectura. Para finalizar, se ofrece una visualización de las capas de presentación y persistencia.

4.1 Arquitectura

La arquitectura es un arte en el que entra en juego el diseño del esqueletos, desde elementos tangibles como pueden ser los edificios en los que vivimos y trabajamos, hasta cosas imperceptibles a la vista o tacto como lo son los productos *software*. En el ámbito de la computación y desarrollo *software*, a este concepto se le puede definir como la estructura o estructuras de un sistema entre los cuales se incluyen los elementos *software* que la conforman, así como sus responsabilidades y propiedades, y la manera en que interactúan y se relacionan dentro de ella (Varvana Myllärniemi, 2015).

Normalmente, se conoce como arquitectura de *software*, aquellos aspectos en relación al diseño de los cuales se tiene la percepción de ser reticentes a cambios y cuya modificación, una vez iniciado el proceso de desarrollo, puede acarrear quebraderos de cabeza llegándose a traducir en un incremento del tiempo y presupuesto respecto al original. Para ello, el diseño de una arquitectura debe focalizarse en aquellos elementos y decisiones que sea arquitecturalmente significantes. Se dice que algo es arquitecturalmente significativo cuando implica un gran riesgo y dificultad en el diseño, que debe ser definido de manera concisa y sin ambigüedades debido a su dificultad de modificación.

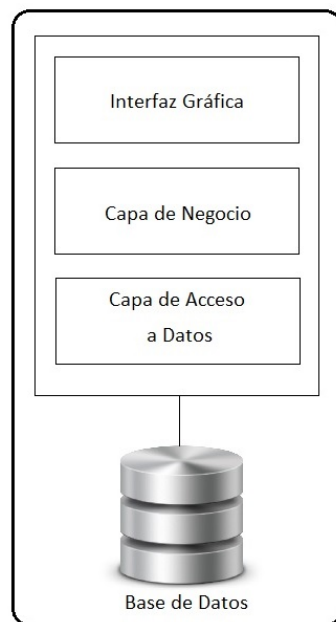
4.1.1 Tipos de arquitecturas

A lo largo de los años las arquitecturas para desarrollo de *software* han ido evolucionando para adaptarse, cada vez más, al clima cambiante de los proyectos *software*. La aparición de las metodologías ágiles y su posterior éxito, han colaborado a la modernización de dichas estructuras con el fin de dotar a los desarrolladores de una mayor flexibilidad a la hora de trabajar y cumplir los requisitos establecidos por el cliente. Las arquitecturas a las que me estoy refiriendo son: monolítica y microservicios.

A lo largo de los siguientes puntos me dedicaré a exponer la filosofía y forma de trabajo utilizadas en cada arquitectura, amén de las ventajas y desventajas que ello conlleva.

4.1.1.1 Arquitectura monolítica

Las arquitecturas monolíticas son conocidas como la máxima expresión de simpleza dentro del mundo de arquitecturas *software*, debido a su sencilla composición. Como su nombre bien indica, la estructura seguida es la de una torre o monolito en la que todas las partes de una aplicación estas apiladas unas encima de otras, enlazando de esta manera las funciones llevadas a cabo por cada una de sus capas.



7. Arquitectura monolítica

Como se puede apreciar en el diagrama superior, una arquitectura monolítica supone un diseño de aplicación en capas bien definidas, de las cuales cada una tiene unas funcionalidades que requieren de sus vecinas para su correcta actividad. Desde la parte visual hasta la de datos están dispuestas en capas que juntas se consolidan para formar una única aplicación. Estas capas tienen un alto nivel de dependencia ya que, si por ejemplo, se sustrae la capa del controlador en una aplicación MVC monolítica, la parte visual quedaría aislada de la parte de datos lo cual impediría el correcto funcionamiento de todo el conjunto.

Una manera más sencilla de verlo es como un crucero de lujo que navega por el océano. Igual que en una aplicación monolítica, el crucero está dividido en pisos con diferentes propósitos: la parte más baja alberga la zona del motor que permite la movilidad del barco, un poco más arriba tenemos los almacenes donde se guardan los productos alimenticios que serán consumidos por aquellos que se encuentren a bordo, luego nos encontramos con los

camarotes donde descansan la tripulación y los viajeros, por encima de eso se encuentran los restaurantes y cocinas que proveen de alimentos y en la parte de la terraza nos encontramos con las tumbonas y la piscina. Si alguna de estas partes es eliminada, este crucero de lujo dejaría de ser lo que es. Si se quita la parte de los motores la mayor distancia que recorrerá el crucero será la que consigan al empujarlo desde la orilla del puerto, de la misma manera que si eliminamos la planta de almacenaje de alimentos los clientes no podrán disfrutar de un exquisito plato de caviar iraní junto a una copa del mejor vino francés. Si quitamos los camarotes, nadie podrá descansar durante la noche y sin tomar el sol o darse un baño en una piscina de agua dulce en mitad del mar, estas vacaciones no se sentirían de la misma forma. Este ejemplo representa el concepto de una estructura fuertemente ligada en la que todas sus partes dependen unas de otras.

Como Nadalin (2015) explica en su artículo, este tipo de arquitectura funciona sin ningún problema en la mayoría de las aplicaciones, manteniendo la complejidad en su mínimo exponente. Esta idea de simplicidad puede ser muy suculenta a primera vista pero implica un alto grado de desventajas en relación con otros aspectos a tener en cuenta numeradas por Badola (2015):

- La complejidad de una arquitectura monolítica es directamente proporcional al tamaño de esta misma, es decir, cuanto más crezca una aplicación de este tipo más dificultades y problemas surgirán durante el desarrollo. Este tipo de situaciones son causantes de una bajada en la producción de los programadores debido a que, conforme crece el proyecto, se va destinando más y más tiempo al mantenimiento de la arquitectura que a la propia programación de los requisitos especificados por el cliente.
- Introducir modificaciones y *refactorizar* una aplicación con tal estructura supone un quebradero de cabeza al intentar mantener todas las piezas unidas. En caso de que se demande una actualización sobre alguna de las funcionalidades, es necesario llevar a cabo un minucioso análisis de todas las partes que interactúan para llevar a cabo dicha tarea. De esa manera, es posible controlar las dependencias y evitar los errores con otras capas.
- Un único fallo en la aplicación es capaz de inutilizarla por completo. Al tratarse de un bloque único intercomunicado entre capas, un error en cualquiera de sus partes puede cortar el flujo entero de esta, incapacitando su uso hasta que sea resuelto. Esto puede resultar engañoso ya que, un error minúsculo que cause el fallo total puede ser interpretado como algo de mayor calado y distraer la atención de los desarrolladores.
- El llevar a cabo la tarea de escalar dicha arquitectura no es un trabajo nada sencillo. Esta estructura únicamente permite el escalado en horizontal, dejando solo como opción el despliegue del sistema entero en diferentes servidores. Dicha escalabilidad supone una gran limitación en cuanto a ofrecer una mayor eficiencia y empleo del tiempo. No es posible el despliegue por separado debido a su condición de pieza única, lo que repercute en la respuesta ofrecida al cliente. Como resultado, la integración en



la nube de aplicaciones con semejantes características resulta una tarea casi imposible de llevar a cabo.

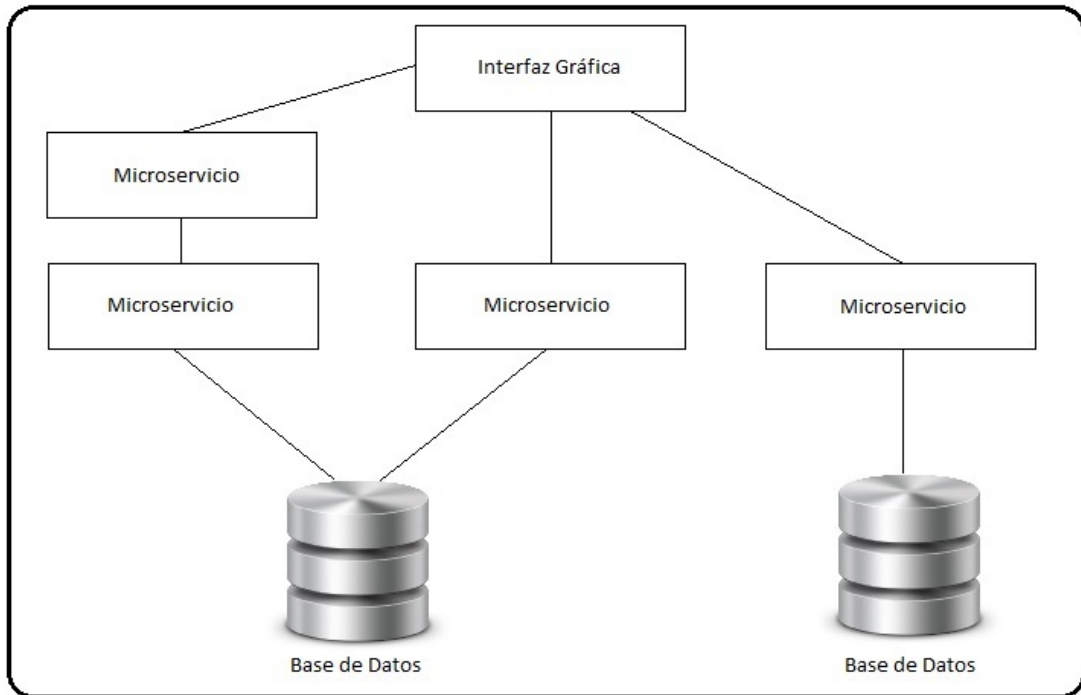
- Dentro de un proyecto los roles de los desarrolladores son separados en tareas específicas, limitando de esta manera la colaboración entre equipos. Este tipo de clasificación recuerda a la metodología de desarrollo en cascada (*waterfall*) en la que cada equipo era separado por fases y apenas existe interacción entre diferentes profesionalizaciones.

4.1.1.2 Arquitectura de microservicios

En el polo opuesto encontramos la arquitectura basada en microservicios. Este método de desarrollo de aplicaciones *software* está fundado en la partición de la aplicación en múltiples servicios, abordando el problemas de diseño que acaban dando lugar a arquitecturas demasiado complejas (Badola, 2015). Cada una de estas piezas es encargada de realizar una tarea en concreto, de manera que se adopta una postura de independencia y modularidad respecto al resto de servicios que componen el conjunto. Estas características proporcionan total individualidad a la hora de desplegar, permitiendo que solo aquella parte que haya sido modificada sea desplegada sin necesidad de incluir al resto de elementos en este.

Gracias a la simplificación de los servicios en partes tan pequeñas, el desarrollo del producto aumenta en agilidad y productividad (cjrequena, 2016). Mientras que en las arquitecturas monolíticas se trabaja con grandes porciones de código, la modularidad de los microservicios reduce el tamaño de este a niveles mucho menores, permitiendo realizar una gestión más efectiva sobre el conjunto en general. Del mismo modo, esta disminución de tamaño conlleva una reducción notable en la complejidad ligada a la codificación. Al contar con piezas más desacopladas y ligeras, la introducción de modificaciones resulta mucho más sencilla, ofreciendo una mayor respuesta ante el cliente mediante la rápida respuesta y entregas dinámicas con opciones a la carta.

En cuanto a temas de escalabilidad, una vez más el reducido tamaño de los microservicios ofrece importantes ventajas. Esta deja de ser general a la aplicación completa, siendo posible el escalado individual de aquellos componentes que realmente lo requieran. De este modo es posible establecer un ahorro de recursos que repercuta positivamente en el proyecto. Como ejemplo se puede tomar una función que reciba una mayor carga de trabajo o tráfico, mientras que en una arquitectura monolítica sería necesario replicar la aplicación entera, los microservicios cuentan con la opción de escalar individualmente el componente que cuente con una mayor carga.



8. Arquitectura de microservicios

La identificación y resolución de errores también supone un claro avance en lo que a la gestión del tiempo se refiere. Al tratarse de piezas *software* de tamaño tan reducido, el identificar errores no da lugar a la ambigüedad. Aquellos microservicios en los que se produzcan fallos o sean defectuosos, pueden ser aislados del conjunto general y proporcionarles un trato individual, todo esto sin que la ejecución de la aplicación quede parada por completo. No obstante el trabajo con microservicios no es un camino de rosas. Puede parecer que el optar por una arquitectura basada en microservicios sea la opción óptima, la bala de plata que acabe con la bestia que supone el encontrar la mejor arquitectura posible y así desarrollar cualquier aplicación *software* sin ningún tipo de preocupación, pero no todo es tan simple como parece.

Al tratarse de muchas componentes con diferentes propósitos, es necesaria la definición de un mecanismo de comunicación claro y conciso que permita una comunicación eficiente y organizada entre los distintos micorservicios. El desarrollo de un sistema distribuido siempre acarrea una gran complejidad en cuanto a que debe haber un sistema claro de comunicación entre las distintas partes que gestione con precaución las conversaciones entre todos sus servicios. Esto tiene especial importancia en aplicaciones con un alto número de llamadas remotas o asíncronas entre sus componentes, representando un problema si no se establece una gestión adecuada.

4.1.1.3 Comparativa

Factor	Arquitectura Monolítica	Arquitectura de Microservicios
Forma	Disposición de pila unificada en capas	Piezas separadas con funciones únicas
Implementación	Fácil debido a su estructura simple	Complicada red de comunicación entre servicios
Complejidad	Aumenta proporcionalmente al tamaño de la aplicación	Solventa los problemas de diseño en arquitecturas grandes
Modificación y refactorización	Introducir una modificación supone revisar la estructura completa	Introducir una modificación atañe únicamente al servicio al que se le aplica
Escalabilidad	Complejo, duplicado horizontal	Sencillo, descomposición funcional
Control de errores	Un único error puede incapacitar el funcionamiento de la aplicación entera	Un error afecta solamente a aquel componente en el que ocurre
Integración en la nube	Difícil debido a su estructura unificada	Simple
Despliegue en producción	Desplegar en producción implica la subida de todas y cada una de las partes del proyecto	Se despliegan únicamente aquellos módulos que han sido añadidos o modificados

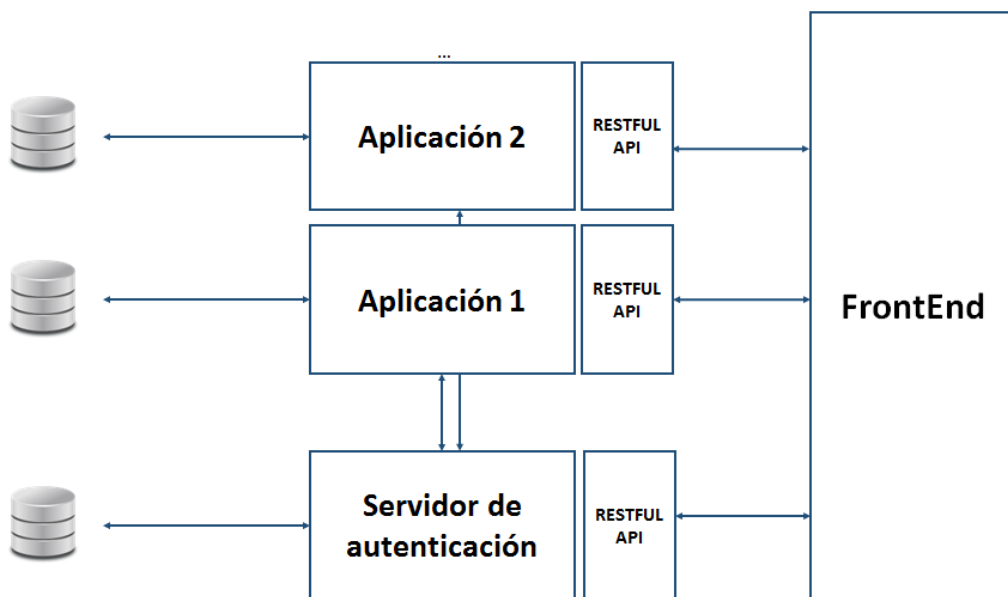
23. Arquitectura monolítica vs arquitectura de microservicios

4.1.2 Framework

En el anterior punto se han mencionado dos tipos de arquitectura utilizados en desarrollos de proyectos *software* (monolítica y microservicios). Teniendo en cuenta las posibilidades que ofrecen cada una y el estado actual de los desarrollos de la empresa, se ha decidido optar por una arquitectura de microservicios que sirva como base para futuros proyectos. Los puntos

clave que se desean atacar para producir una mejora sustancial son los siguientes: estructura modular que facilite la creación de componentes o servicios, su compartición de estos entre los diferentes proyectos y ramas con la finalidad de conseguir un mejor empleo del tiempo y cohesión entre desarrollos, mayor implementación de test y de una forma más sencilla y facilitar la tarea de aplicación de test y despliegue mediante labores de integración continua y automatización.

En líneas generales, la arquitectura de microservicios que se propone usa APIs REST para la comunicación y un servidor de autenticación independiente. Al desarrollar sobre este tipo de servidor no es necesario volver a implementar desde cero la gestión interna de usuarios para las futuras aplicaciones. De esta manera la flexibilidad de cada proyecto no se vería comprometida respecto al añadir nuevas funcionalidades o modificar las ya existentes. De la misma manera se facilita la integración de servicios ofrecidos por terceros de forma transparente y fiable, así como la integración de nuevos desarrollos con tecnologías diferentes a las que se describen en este proyecto y que no han sido contempladas en un primer pensamiento. Todas estas características satisfacen las bondades sobre los microservicios descritas en el anterior punto.



9. Boceto de arquitectura general

Como anteriormente se ha mencionado, las arquitecturas de microservicios ofrecen una alta tolerancia a la integración de componentes usando múltiples lenguajes de programación o tecnologías, permitiendo dotar a los proyectos de un alto grado de flexibilidad. No obstante, el propósito de esta arquitectura en este escenario lleva a moldear este aspecto y adaptarlo a las necesidades. Para ello, y con la finalidad de cumplir las metas propuestas, se ha decidido establecer unas tecnologías por defecto con el objetivo de unificar criterios y dar la posibilidad para poder reaprovechar código en futuros proyectos. Este aspecto no trunca los orígenes de la arquitectura ni la convierte en otra diferente, simplemente se adapta para crear un ecosistema común dentro de Everis, así como evitar conflictos y limitaciones entre proyectos como los que surgen actualmente.

Las tecnología estándar escogida tiene como nombre *MEAN2 stack* y consta de lo siguiente: *Angular, NodeJS, ExpressJS* y *MongoDB*.



10. Tecnologías de la arquitectura

Más adelante se detalla en qué consiste y qué aporta cada tecnología elegida pero primero es necesario centrarse en las principales ventajas que proporcionan en conjunto:

- Uso de un único lenguaje para todo el proyecto: la elección correspondiente en este caso es *JavaScript* ya que es un lenguaje bastante portable y ligero para el desarrollo web. Esto implica que la aplicación estará desarrollada íntegramente con este lenguaje, de arriba a abajo, tanto el *front-end* como el *back-end*. El aporte principal que proporciona esta decisión es la facilidad a la hora de intercambiar roles en el equipo de desarrollo, reduciendo ampliamente la curva de aprendizaje y el proceso de adaptación a nuevos proyectos. A todo esto se suma la opción de generar aplicaciones de escritorio, mediante el uso de la herramienta *Electron* (<https://electron.atom.io/>), con el mismo código *JavaScript* utilizado para el desarrollo web.
- Establecimiento de un único formato de datos para todo el proyecto: el formato elegido para el tratamiento de datos en toda la plataforma es *JSON* con lo cual no es necesaria ninguna conversión de un tipo a otro. Este formato, creado por Douglas Crockford, fue diseñado específicamente como manera de intercambio de datos en aplicaciones escritas en *JavaScript*, lo cual transforma la tarea de manipulación de datos en algo inherente al lenguaje usado. Tanto la parte *back-end* desarrollada en *NodeJS* y *ExpressJS*, como la parte *front-end* desarrollada en *Angular*, así como la *API*, trabajan con datos en formato *JSON*.

4.1.3 Metodología de desarrollo

Para aplicar la arquitectura propuesta en los puntos anteriores se requiere establecer una metodología de desarrollo TDD que permita el desarrollo de test, tanto unitarios como E2E durante el desarrollo de cada proyecto. Este último tipo de test son de gran relevancia en arquitecturas de microservicios como la explicada aquí, permitiendo la comprobación de la integridad de la aplicación al completo después de la modificación de alguna de sus partes. Es

decir, asegurar que un cambio realizado en algún componente no afecta a ninguna funcionalidad implementada por el resto.

A este punto se le ha de sumar la combinación con herramientas de integración continua y gestión de repositorios con el fin de evitar los tiempos empleados en pruebas y puesta en producción debido a que cualquier subida de código habrá sido probada de forma automática y el despliegue en producción estará automatizado desde antes de empezar el desarrollo.

Estos puntos mencionados corresponden a etapas más avanzadas del proyecto en cuestión por lo que se desarrollan más ampliamente a lo largo del documento.

4.1.4 Infraestructura

Para la infraestructura se ha optado por servicios *Cloud*, concretamente *Azure*. Este tipo de infraestructura ofrece un gran número de posibilidades y ventajas respecto a la arquitectura descrita, pero a su mismo tiempo hay algunos puntos negativos a tener en cuenta:

- **Ventajas:** con el fin de lograr la máxima escalabilidad y flexibilidad posible en la infraestructura, se ha optado por una configuración de servicios independientes de la aplicación, es decir, evitar la configuración de un único servidor *Linux* o *Windows* con todos los servicios ejecutándose allí como ocurre en las arquitecturas monolíticas. *Azure* como aplicación de *hosting-cloud* ofrece la posibilidad de levantar servicios y escalar de forma automática.
- **Desventajas:** el *hosting* de aplicaciones *Cloud* no tiene fijado un precio como tal y sus servicios se cobran con respecto al uso que se necesite. Esta variabilidad puede resultar difícil a la hora de controlar o cuantificar por lo que se necesita la realización minuciosa de un estudio previo del uso que se le vaya a dar, que incluya también el *Tier* de pago que se desee utilizar y unas medidas de control, en caso de costes extras, bien clarificadas.

Un desarrollo de estas características debe afrontar costes extras que no surgirían de realizarse sobre servidores propios, por lo que conlleva una negociación previa con los proveedores de dichos servicios.

4.2 Test

Los sistemas o aplicaciones *software* han evolucionado a lo largo de los años, aumentando en tamaño y complejidad. Como cualquier producto de negocio, es de gran importancia desarrollar, implementar y analizar pruebas que determinen el correcto funcionamiento y el cumplimiento de las metas establecidas a principios del proceso. En el mundo de la informática a este tipo de pruebas se las denomina *test*.

Un test es la comprobación de que aquello que se ha desarrollado cumple el propósito para el que se había diseñado. Acorde con Itakonen (2015) las metas principales de diseño de test deben cumplir con dos objetivos: verificar y validar. El primero de ellos, la verificación, supone el cumplimiento por parte del *software* de satisfacer los requisitos y especificaciones acordadas, así como cumplir con el diseño previsto. Por otro lado, la validación se encarga de que el *software* desarrollado satisfaga las necesidades y expectativas por parte de clientes y usuarios a los que vaya dirigido. Como añadido, testear también ofrece ayudas como la detección de defectos y problemas que puedan repercutir sobre la calidad final del producto, así como la investigación de dichos riesgos con el fin de que no vuelvan a ocurrir. De esta manera, aplicando test en fases de desarrollo tempranas es posible el control de errores a un nivel más bajo que, de otra manera, podrían desencadenarse después de su entrega.

La realización de test requiere tiempo y personal dedicado para completarse con éxito. Para ello, y con la finalidad de que el proceso sea óptimo y beneficioso, es de vital importancia el entendimiento del *software* sobre el que se van a aplicar las pruebas. Es necesaria la identificación de aquellas partes que necesitan ser testeadas y aquellas que no lo requieren, de la misma forma que debe haber una priorización dentro de aquellas que sí resulten imprescindibles.

4.2.1 Testeo en arquitecturas de microservicios

Como se ha expuesto durante anteriormente en este documento, la arquitectura elegida para el desarrollo de este proyecto se corresponde con la de microservicios de manera que se hace énfasis principalmente en el testeo de este tipo de arquitectura.

El testeo de arquitecturas de microservicios frente a las monolíticas ha originado la aparición de nuevas estrategias y técnicas para abordar dicha práctica que han tenido que ser replanteadas para su adaptación (Celemson, 2014). No es posible la reutilización completa de aquellas técnicas que sirvieron durante años para el testeo de aplicaciones de carácter monolítico ya que la distribución interna de los microservicios varía drásticamente. Esta nueva estructura basada en la modularidad del *software* demanda un nuevo pensamiento a la hora de comprobar su correcto funcionamiento. Los microservicios ofrecen muchas bondades a la hora de desarrollar una aplicación o sistema pero, al mismo tiempo, puede resultar enredado el hecho de realizar test sobre ellos. Como bien se ha mencionado anteriormente, dentro de un *software* de microservicios existen muchos módulos independientes unos de otros que se comunican entre sí de diversas manera y empleando múltiples canales.

Con el fin de atacar estas complicaciones, Roslonek (2016) ofrece como una primera aproximación la técnica de realización de test aislados. Los test aislados proporcionan información sobre el servicio específico sobre el que se están realizando, de manera que este no cuenta con ningún tipo de comunicación con el resto de los servicios que conforman la aplicación. No obstante, es necesario realizar pruebas que comprueben el funcionamiento de la aplicación en conjunto, controlando la interacción de los diferentes servicios así como sus comunicaciones. Al tratarse de una gran cantidad de módulos con múltiples conexiones, comunicándose de formas distintas, el objetivo de recorrer todos los caminos posibles como si

de una aplicación monolítica se tratase queda lejos de hacerse realidad. Dicha tarea podría llevarse a cabo pero el tiempo, presupuesto y esfuerzo invertidos serían tan elevados que en lugar de producir beneficios, causarían problemas al desarrollo.

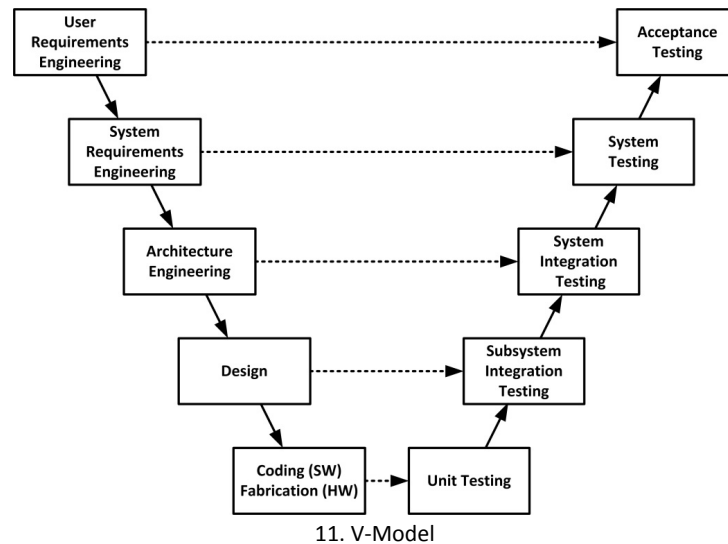
Con la finalidad de no caer en un agujero negro de test por resolver, es posible utilizar prácticas como la regla *80/20* o *Pareto Principle*. Como describe Ragnar (n.d.) en su artículo *When should I start load testing?* es posible testear una aplicación o sistema y abarcar aquellas funcionalidades o partes de código cruciales para obtener un funcionamiento correcto, sin necesidad de sumirse en una historia sin fin recorriendo todos los caminos que el usuario final podría visitar. Esta regla predica que el 80% de los efectos causados en una aplicación o sistema, provienen o son causados por el 20% de las acciones que se realizan en ella. La búsqueda del test perfecto suena más a película fantástica de *Hollywood* que a una realidad posible, cierto es que se puede llegar a una descomposición del comportamiento de los usuarios que diese lugar a la prueba definitiva pero, una vez más, todo los elementos que conllevan llegar hasta este punto no compensan con sus resultados. Es por esto que Ragnar (n.d.) apoya el uso de la regla *80/20* con la que es posible el desarrollo de un mayor número de test, pudiendo ser implementados en las fases más tempranas del desarrollo. Pero el aplicar esta regla no implica la desaparición de las complicaciones, es necesario un fuerte entendimiento del contexto para cada parte a testear, así como la complejidad que alberga.

Dicha práctica puede dar lugar a contratos con el cliente. Aquí, el cliente es el encargado de realizar un estudio sobre la aplicación y proporcionar a los desarrolladores aquellos test que considere indispensables para el cumplimiento de sus requisitos, reflejando la interacción del usuario que se espera.

De esta manera, el testeado sobre aplicaciones de microservicios requiere de un cambio de mentalidad si uno proviene de desarrollos con arquitecturas monolíticas.

4.2.2 Tipos de test

A la hora de comenzar con la aplicación de test, es posible encontrar una amplia variedad y tipos con unos objetivos y formas de uso bien definidas. Acorde con el la figura 11, correspondiente al *V-Model*, es posible visualizar los diferentes tipos de test, organizados de nivel más bajo a más alto, que se realizan durante cada fase del proyecto.



En este caso, y para el alcance de este proyecto, se ha utilizar test unitarios y de integración dado que ambos cubren todo aquello que necesario por comprobar para la prueba piloto y las partes que la componen. De esta manera, el documento se centra en describir únicamente estos dos tipos y explicar qué son y en qué consiste, así como sus ventajas y desventajas.

4.2.2.1 Test unitario

Cuando se habla de test unitarios, se hace referencia a la unidad mínima en la que una aplicación o sistema puede ser particionados para poder aplicar pruebas de control sobre su funcionamiento y que estas proporcionen información relevante que refleje la calidad de dicho *software*. Este proceso de desarrollo básico para el testeo a nivel más bajo consiste en la división de una aplicación *software* en pequeñas partes o unidades que ofrezcan un funcionamiento autónomo e independiente al resto del código. Al individualizar las distintas partes es posible realizar pruebas sobre ellas de forma aislada, sin que ninguna otra unidad de código pueda afectar al resultado final.

El alcance que puede tener cada unidad depende de la funcionalidad que ofrezca pero, normalmente, deben estar correctamente delimitados. Dicha división en partes con propósitos bien definidos recuerda a la distribución dentro de una aplicación con arquitectura de microservicios, es por eso que la elección de este tipo de test se acopla casi de forma natural a la estructura ofrecida por estos, llegando incluso a influir durante la fase de diseño. Semejante acotación permite que, en caso de surgir algún error en los test, los desarrolladores tenga la capacidad de identificarlo y solventarlo de forma rápida y eficiente sin que este llegue a afectar al conjunto entero.

Como bien describe Rouse (n.d.), los test unitarios pertenece a un proceso de desarrollo *software* denominado TDD que, como bien indica su nombre, está basado en una forma de trabajo en la que continuamente se somete el código a una serie de test con el fin de mejorar aquellas parte que puedan flaquear. Rouse (n.d.) entiende este proceso de la siguiente forma: en primer lugar, los desarrolladores deben encargarse desarrollar los test unitarios necesarios para el proyecto en cuestión, antes siquiera de haber implementado cualquier funcionalidad.

Teniendo esto en mente, el desarrollo comienza con normalidad pero con el objetivo de programar respecto a los resultados esperados por los test. Cuando un desarrollador completa alguna de las funcionalidades, esta es sometida a los test que fueron diseñados durante las fases más tempranas del proyecto y, en caso de error, el código es *refactorizado* para cumplir con las exigencias marcadas. Aplicando test desde los inicios de un proyecto ayuda a la detección de errores en situaciones donde es posible establecer un control y aplicar la solución adecuada sin que implique una repercusión importante en la agenda de desarrollo, evitando así escenarios catastróficos como los que serían encontrar un fallo a pocos días de una entrega o que el cliente mismo experimente dichos problemas. El resultado de todos estos pasos produce un código que resulta más reusable, ayudando a la reducción de costes destinados a la realización de test en futuros proyectos.

No obstante, el optar por el uso de test unitarios no es tan sencillo como pueda hacer creer a simple vista. Igual que a la hora de definir microservicios, es necesaria un conocimiento previo de la aplicación (*white box test*) con el fin de dividir la aplicación en unidades lo suficientemente grandes como para ofrecer información relevante sobre el estado del *software* y que a la vez no resulten ser tan pequeñas como para desestimar su comprobación. Al mismo tiempo, la automatización de dichas pruebas es un hecho que se repite con más y más frecuencia, lo que conlleva el estudio de uso de herramientas que permitan el despliegue de test de forma instantánea. Este último punto representa una de las piezas principales para los objetivos planteados en este proyecto y es una de las razones por la que se ha elegido la realización de test unitarios.

4.2.2.1 Test End-to-End

Los test *End-to-End*, también conocidos como test de integración, son los encargados aplicar pruebas de principio a fin uniendo todas las piezas que conforman un aplicación *software*. Comprueban que todos los componentes necesarios para ejecutar cada funcionalidad se comunican y responden con éxito, ofreciendo resultados comparables a la interacción con un usuario real.

Estos test deben ser realizados una vez se ha completado con buenos resultados la fase de test unitarios ya que, teniendo la seguridad de que el *software* funciona correctamente para el nivel más bajo, facilita la tarea a la hora de implementarlos y depurarlos.

En el artículo *Why End to End Testing is Necessary and How to Perform?* ofrecido por *Software Testing Help* (20017) se describe una serie de consejos que sirven de ayuda para el desarrollo de este tipo de test:

- **Perspectiva de usuario:** es importante tener un conocimiento previo de tipo de usuario al que va dirigida la aplicación, así como el propósito de esta y el entorno en el que está destinada a ser usada. De esta manera es posible obtener la implementación de pruebas de integración que sigan los pasos adecuados para cada funcionalidad, actuando sobre todos los servicios necesarios.

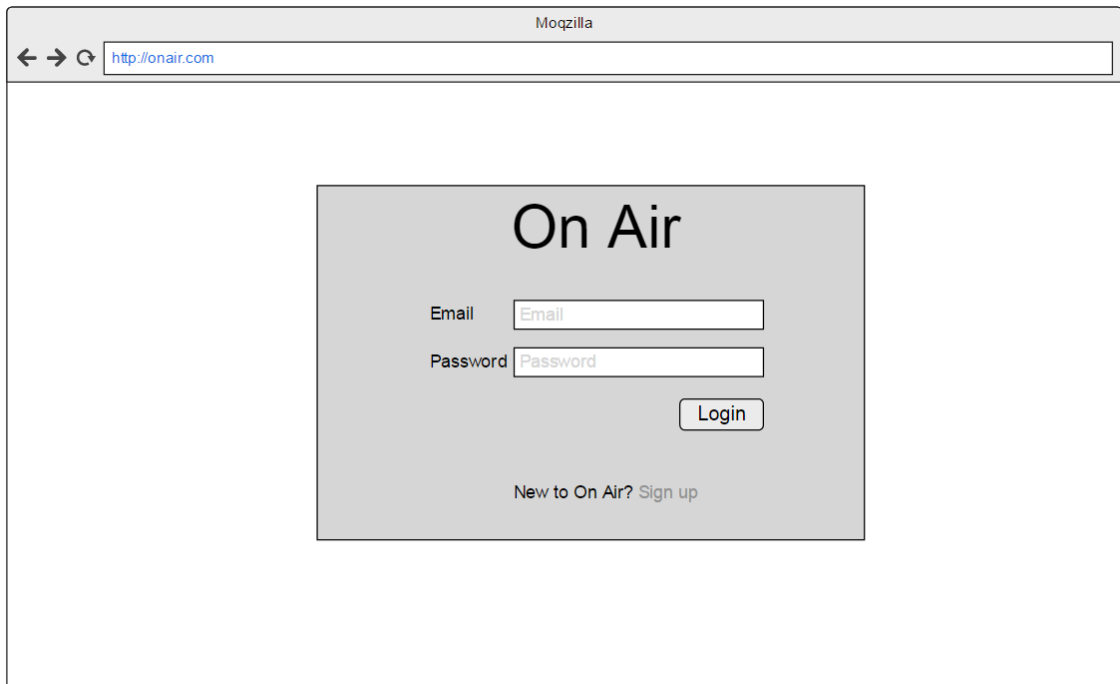


- **Testear funcionalidades existentes:** a la hora de seleccionar aquellas acciones que requieran ser comprobadas, es conveniente elegir las acorde a su estado de implementación dentro del sistema actual. Tiene sentido aplicar test *E2E* a los procedimientos que cuenten con todos los componentes vitales para su funcionamiento ya que, el hecho de hacerlo sobre aquellos que se encuentren incompletos producirían resultados sin lógica ni valor alguno para el desarrollo.
- **Múltiples escenarios con test específicos para cada uno:** debido a la naturaleza incierta del ser humano, usuarios de una misma aplicación pueden interactuar de formas distintas y contemplar caminos que al otro nunca se le hubiesen ocurrido. Es por eso que para ofrecer un muestreo con más peso, se debe procurar el considerar más de un usuario para dichas pruebas. De la misma manera, cada escenario elegido tiene que contar con test que se adapten a él.

Como punto a favor, destaca su versatilidad para poder acoplarse a métodos *White Box* como *Black Box*. Para los primeros, es necesario un conocimiento del flujo interno de la aplicación y así diseñar test que se adapten a él mientras que en la segunda es necesario conocer aquellas acciones que se deseen testear y los resultados que se esperan obtener. A ello se suma la confianza de la que se ve dotado el producto al contar con una cobertura completa (Demiss, 2016).

4.3 Capa de presentación

La primera toma de contacto con la aplicación debe ser con una ventana tradicional de *Login* en la que el usuario pueda tener introducir sus datos para acceder a al contenido de la aplicación. Debe quedar claro la aplicación en sí requiere de un usuario para poder usar sus funcionalidades, no existe la posibilidad de acceder como anónimo.



12. Login Mock-up

En caso de que sea nuestra primera vez, necesitaremos registrarnos pulsando la opción de *Sign Up* en la ventana de *Login*. De este modo seremos redirigidos a la ventana de *Sign up* para rellenar nuestros datos y efectuar el registro en la aplicación.



13. Sign up Mock-up

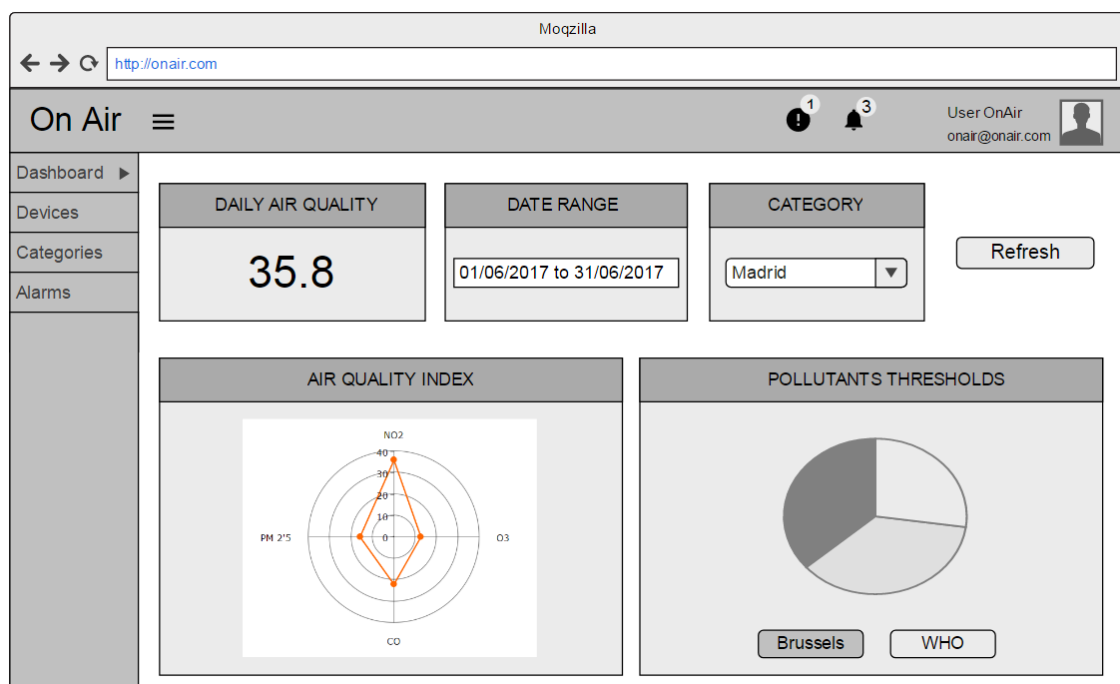
Una vez tengamos nuestro usuario creado, es hora de entrar y contemplar el contenido de la página web.

Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Lo primero que nos encontramos al acceder con nuestro usuario es la parte superior del *Dashboard*. Esta cuenta con una barra superior que contiene el nombre de la aplicación, un icono que permite esconder o mostrar el menú lateral, otro icono (exclamación) que muestra el número de alarmas activas y un tercer icono (campana) que muestra el número de notificaciones recibidas en base a la activación o desactivación de las alarmas. También es posible observar el nombre y correo con el que nos hayamos registrado. En el lateral izquierdo se puede observar un menú desde el que se puede navegar por las distintas vistas que conforman la aplicación.

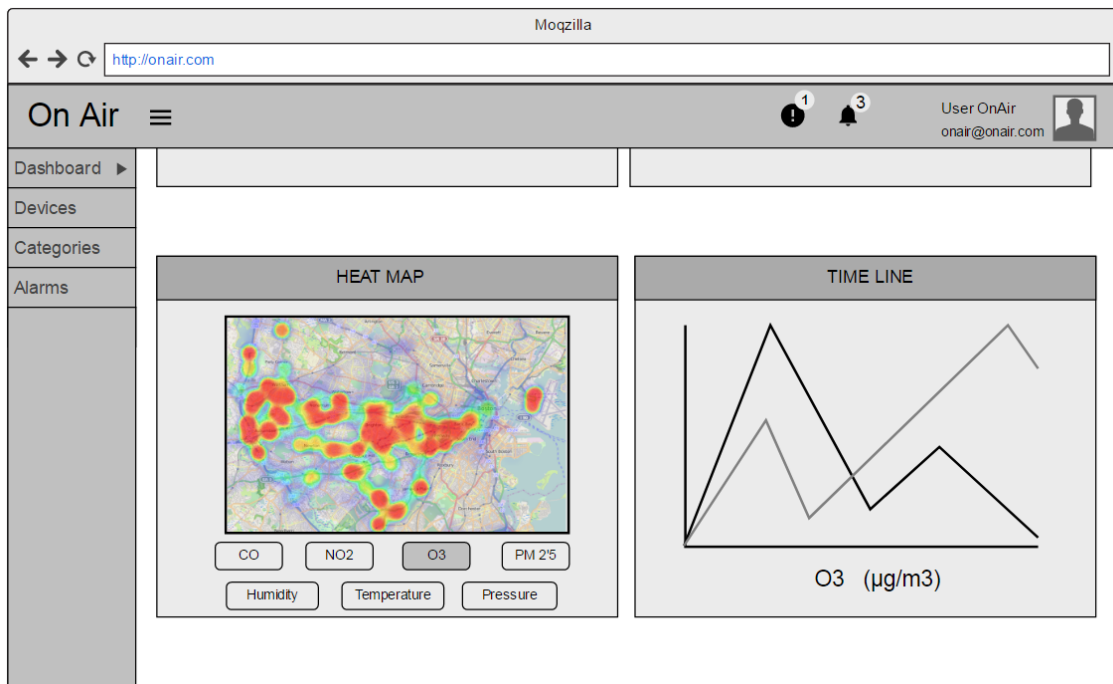
En la zona central es posible ver cinco componentes. El primero de ellos *Daily air quality* únicamente se debe encargar de mostrar el índice de calidad de aire más alto que se haya registrado para cada día. Este ICA viene dado por el cálculo de índice más alto entre los contaminantes. A su derecha se observa el componente *Data Range*, que permite al usuario seleccionar las fechas entre las que desea consultar los datos. El tercero de ellos, un *combobox*, permite elegir la ciudad que se desea visualizar. Para hacer efectivo los parámetros seleccionados, es necesario que el usuario pulse el botón *Refresh*, de manera que se refrescará todo el *Dashboard* reflejando los datos en base a las fechas y ciudad seleccionadas.

Los siguientes componentes son los encargados de interpretar y mostrar los datos recibidos por el sensor. En el *Dashboard* superior es posible encontrar dos de ellos: el gráfico de calidad de aire y el de la media de contaminantes. El primero de ellos *Air Quality Index* es un gráfico de tipo radar en el que se muestra una comparativa de los valores de calidad de aire calculados para cada contaminante. El segundo, *Pollutants Thresholds*, permite visualizar la cantidad media registrada para cada contaminante y saber si se encuentra dentro de los límites aceptados. A la hora de determinar los límites, existen dos fuentes de información distintas, una ofrecida por Bruselas y otra por la OMS. El usuario puede seleccionar una u otra dependiendo de su interés.

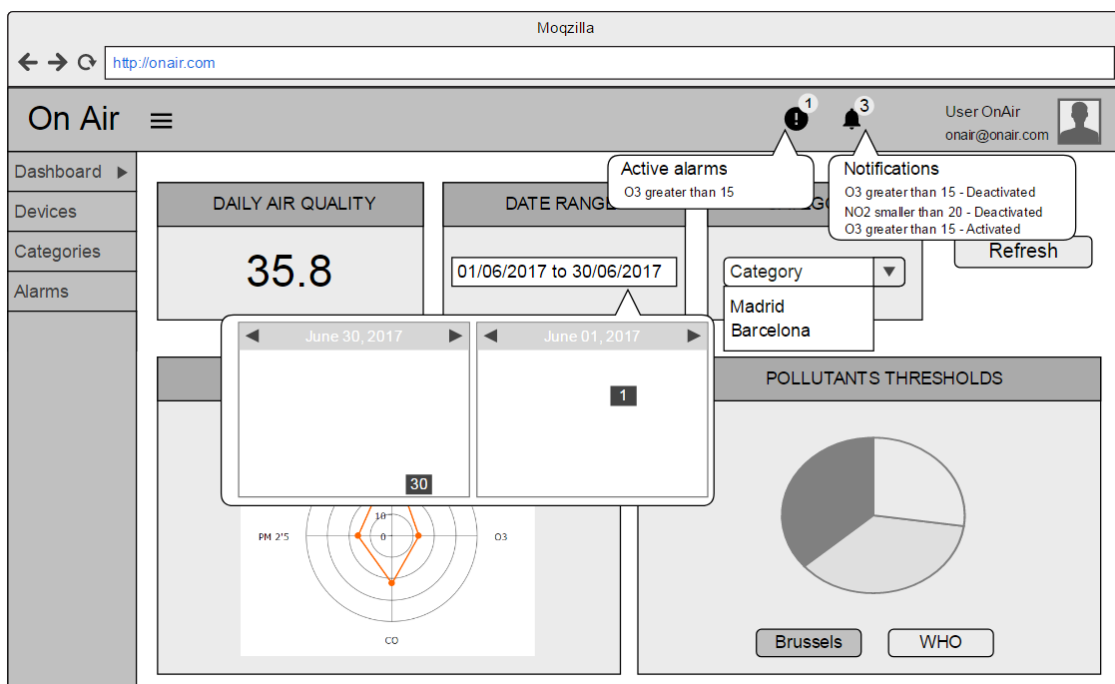


14. Dashboard superior Mock-up

Justo debajo, se encuentran los dos últimos componentes que conforman el Dashboard. En primer lugar tenemos un *Heat Map*, compuesto de un *Google Maps* con una capa para la representación de puntos de calor. Cada uno de ellos se corresponde con la posición actual de un sensor y los datos que este recoge. Los botones de bajo indican los parámetros captados por el sensor y entre los que puede navegar el usuario. Al pulsar alguno de ellos, la información de mapa cambia. El gráfico lineal de su derecha muestra los datos de forma temporal para el parámetro pulsado en el componente *Heat Map*.



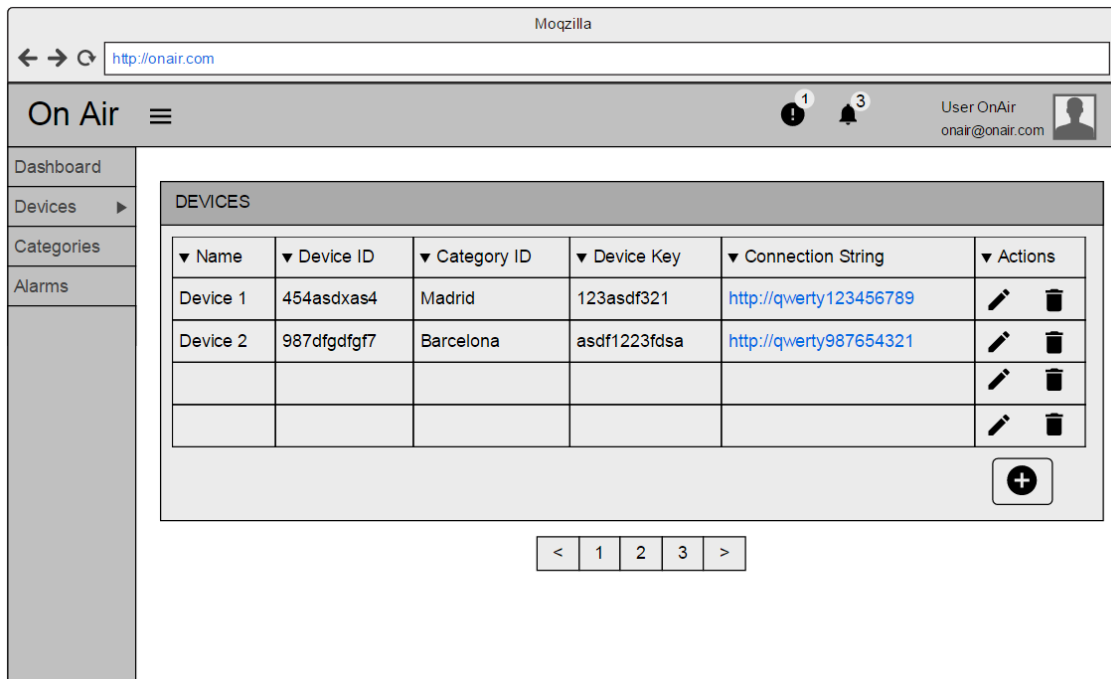
15. Dashboard inferior Mock-up



16. Opciones Dashboard Mock-up

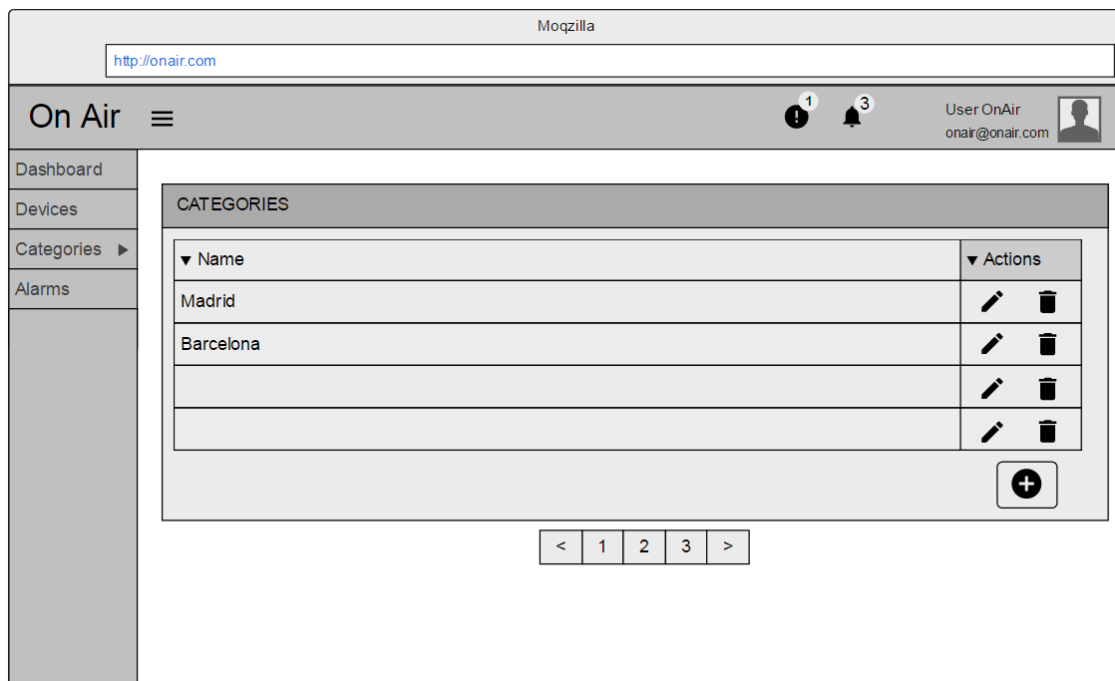
Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Las siguientes ventanas se corresponden a tablas de visualización y edición de información. La primera de ellas, *Devices*, muestra los dispositivos creados. Sobre ellos es posible realizar acciones de modificación o borrado, de igual manera que la creación un dispositivo nuevo.



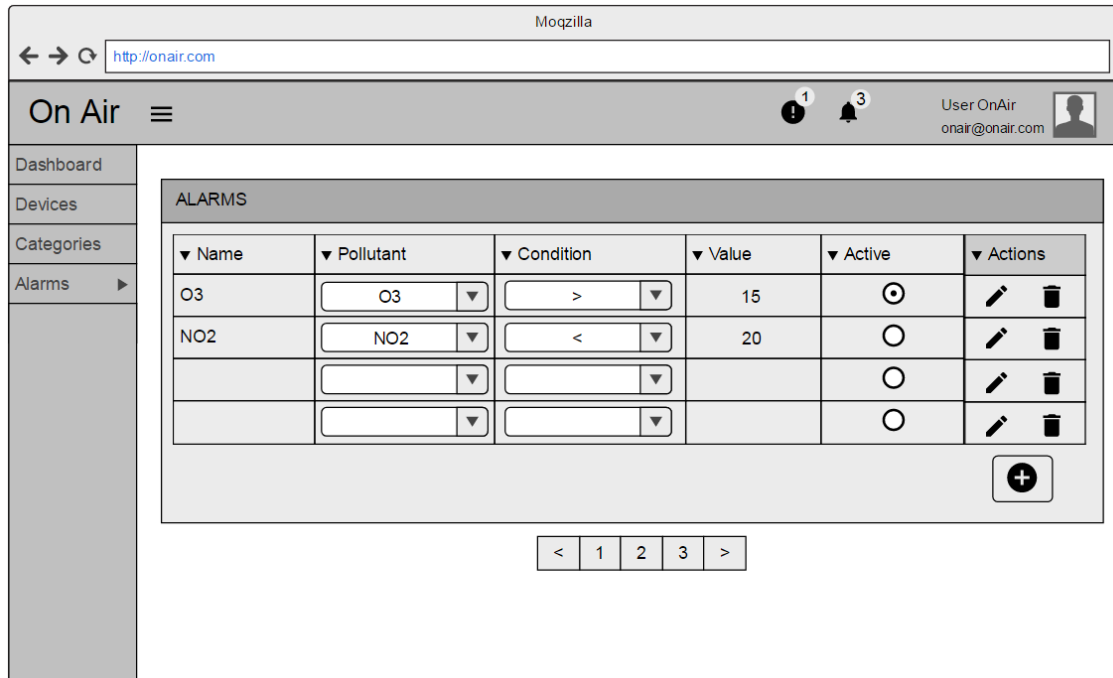
17. Dispositivos Mock-up

La ventana de *Categories* determina las ciudades sobre las que el usuario puede consultar información. Las categorías creadas aquí son añadidas al *Dashboard* de forma automática. También es posible editar y borrar categorías.



18. Categorías Mock-up

La cuarta y última ventana contiene una tabla con la información referente a las alarmas. El usuario tiene la posibilidad de personalizar sus alarmas acorde con el interés que tenga para cada contaminante. La aplicación se encarga de comprobar las alarmas definidas por el usuario y ofrecer notificaciones en base a estas.



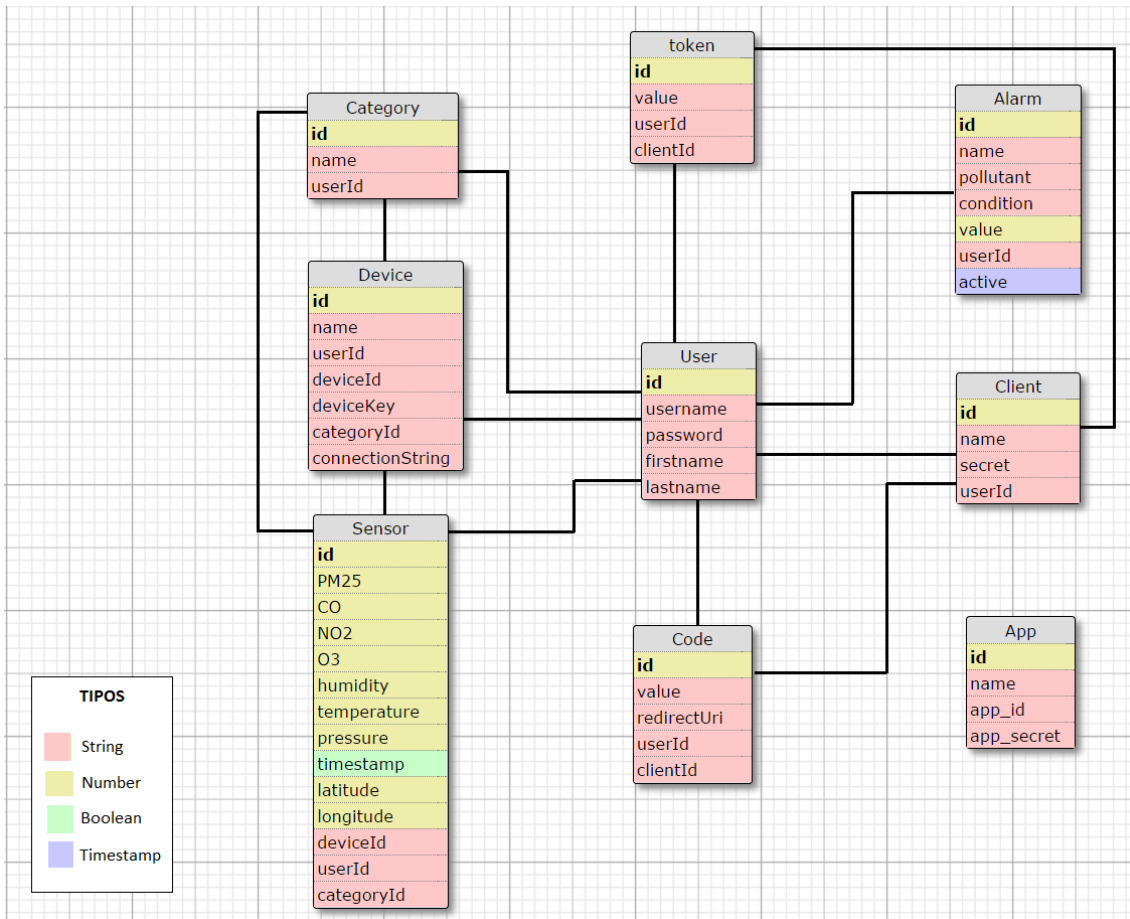
19. Alarmas Mock-up

Para la aplicación final se ha decidido utilizar la plantilla *ng2-admin* de *akveo* (<https://www.akveo.com/>) junto a la librería *amCharts* (<https://www.amcharts.com/>) para los gráficos que aparecen en el *Dashboard*.

4.4 Capa de persistencia

El esquema de datos final empleado por la aplicación viene representado por los siguientes elementos. Es necesario recordar que este diagrama y sus relaciones únicamente representan una visualización conceptual de la estructura de datos manejada para la aplicación web. *MongoDB* se clasifica dentro de las bases de datos no relacionales, lo que significa que los lazos entre los elementos son simplemente orientativos, ya que su existencia no es real. Se ha elegido una representación clásica de diagrama de clases para bases de datos *SQL* para poder otorgar algo de luz sobre la gestión de información mediante este tipo de base de datos *NoSQL*.





20. Diagrama de clases final

Como se menciona en el punto 3.1, cada “clase” del diagrama representa una colección de documentos. Cada uno de estos documentos contiene una estructura con los parámetros definidos en cada colección. En la siguiente imagen es posible observar la estructura de un documento de datos perteneciente a la colección *Sensors*. Este tipo de formato permite obtener una base de datos rápida y ligera, capaz de manejar una gran cantidad de datos.

```

{
  "_id" : ObjectId("5939be54a6b5de4948ad04bd"),
  "categoryId" : "5936d67c734d1d2b1e933198",
  "userId" : "58ca7e87f067e22dd08f962d",
  "deviceId" : "libelium",
  "longitude" : -3.71619,
  "latitude" : 40.31045,
  "timestamp" : ISODate("2017-06-02T13:00:00.000Z"),
  "pressure" : 94392,
  "temperature" : 34,
  "humidity" : 25,
  "O3" : 57,
  "NO2" : 41,
  "CO" : 2,
  "PM25" : 20,
  "__v" : 0
}
    
```

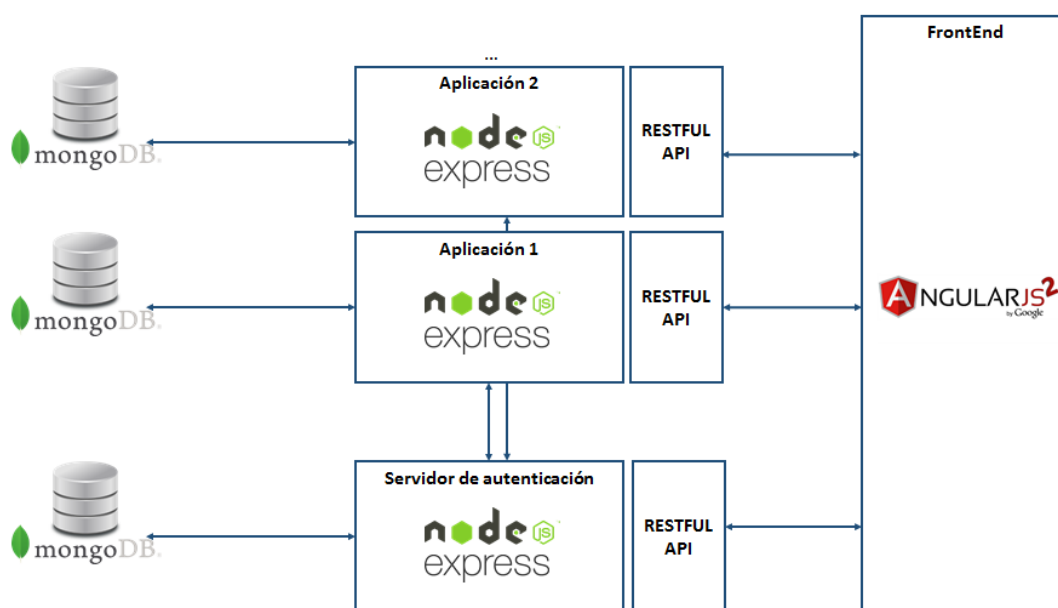
21. Estructura de un documento MongoDB de la colección Sensors

5. Detalles de implementación

El quinto capítulo explica en profundidad las tecnologías representantes de la nueva arquitectura, usadas para la implementación de la aplicación web piloto. Primeramente, se exponen las tecnologías primarias para el desarrollo, acorde con la capa que representen: *front-end*, *back-end* o base de datos. A continuación se describen aquellas necesarias para la creación de test, siempre acorde con su tipo y capa de aplicación. Para finalizar se presenta un estudio de las herramientas de control de versiones así como las clases que existen de estas, las más populares hasta el momento y la elegida para dar soporte a este proyecto.

5.1 Tecnologías utilizadas

Para el desarrollo de la aplicación web piloto se utilizan las tecnologías correspondientes a la arquitectura propuesta: *Angular*, *NodeJS*, *Express* y *MongoDB*.



22. Estructura de la arquitectura con las tecnologías

5.1.1 Front-end

5.1.1.1 Angular

Angular es un *framework* de *JavaScript* cuya filosofía facilita la creación y reutilización de componentes a lo que se le suma la capacidad de realizar llamadas asíncronas a diferentes servidores. Existen dos revisiones de este *framework*: *AngularJS* es la primera de ellas y la segunda, denominada de forma oficial como *Angular*, es el ampliamente conocido *Angular 2*.

La principal ventaja de estar basado en la creación de componentes es su alto nivel de reutilización. Cada vez que se programa un nuevo componente, este contiene una funcionalidad bien definida que puede ser requerida más adelante en el desarrollo del mismo proyecto o ser compartida e implementada en otro nuevo. Para realizar esta integración de componentes ya creados tan solo es necesario pasar los datos requeridos en el formato específico de cada uno para que este tenga todo lo necesario para pintarlo de forma automática y manteniendo todas sus funcionalidades. Adicionalmente, y como las arquitecturas de microservicios permiten, *Angular* es capaz de manejar e integrar datos provenientes de más de un *back-end* convirtiéndolo en un *framework* perfecto con dotes suficientes para desarrollar aplicaciones dinámicas y escalables.

Ahondando todavía más en el tema de escalabilidad con *Angular*, existen herramientas como *NativeScript* que permiten la compilación de aplicaciones desarrolladas utilizando este *framework* a lenguajes nativos para aplicaciones móviles como *Android* o *iOS*.

5.1.2 Back-end

5.1.2.1 NodeJS

NodeJS se diseñó con un modelo orientado a eventos que le permite tener como meta la creación de aplicaciones con un alto nivel de escalabilidad dando lugar a la creación de decenas de miles de conexiones simultáneas en un mismo servidor. Es más, debido a su arquitectura interna, ofrece un rendimiento superior a otros servidores de aplicaciones clásicos, sobre todo en servidores de APIs. Estas características lo distancian de entornos más comunes hoy en día como pueden serlo *Apache + PHP* o *TOMCAT + JAVA*.

La principal diferencia con los servidores nombrados previamente es el modelo de funcionamiento, en el que para *NodeJS* únicamente existe un solo hilo de ejecución. Cada conexión recibida, lanza un evento disparador dentro del motor de ejecución. En el caso específico en el que produjese una operación bloqueante en la aplicación, un nuevo hilo de ejecución sería creado, permitiendo tener un gran número de operaciones concurrentes con un consumo de memoria menor que en otras opciones como lo podría ser *Apache*.

5.1.3 Base de datos

5.1.3.1 MongoDB

MongoDB, como base de datos *NoSQL*, se ha convertido en una opción muy bien valorada respecto a las bases de datos relacionales. Se la define como una base de datos documental,

ya que en vez de guardar los datos en registros como las relacionales, estos son almacenados en documentos (Fernández, 2014).

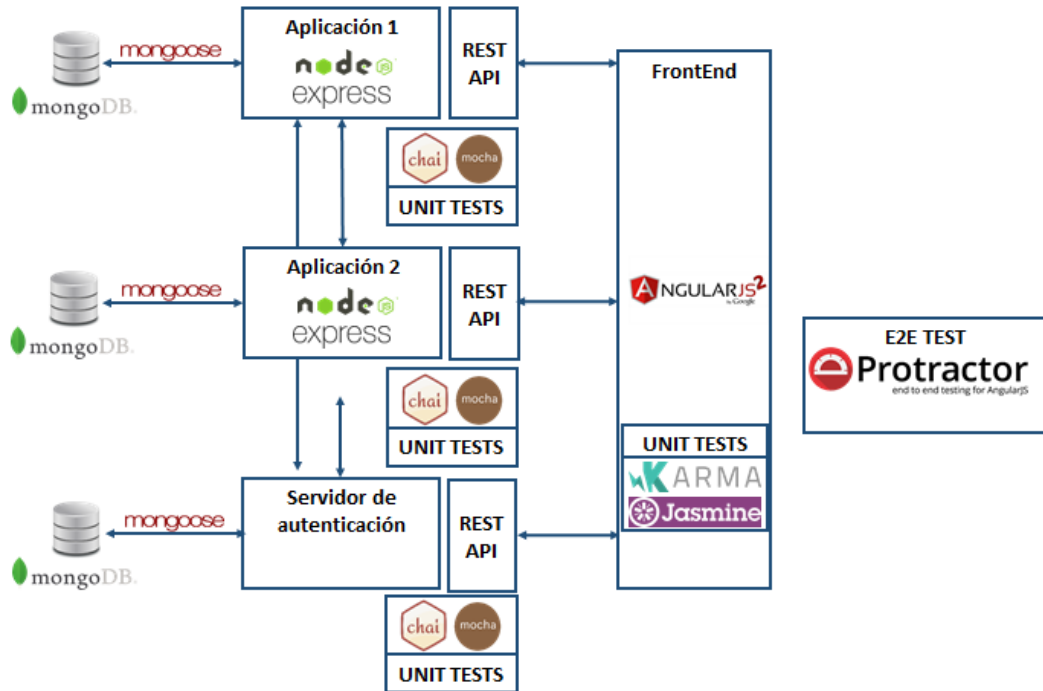
Se ha podido observar que *MongoDB* ofrece un rendimiento más eficiente en operaciones *CRUD* de relaciones únicas frente a otras opciones como *Oracle* y *SQL Server* en el ámbito de bases de datos relacionales. Este aspecto guarda relación con el mayor número de verificaciones de consistencia e integridad, así como la gestión de transacciones que deben realizar este último tipo de bases de datos. Por lo contrario, su eficiencia baja cuando se trata de operaciones binarias, aquellas que involucran relaciones o colecciones. No obstante, a pesar de no estar al nivel de las relacionales en este punto, es una de las cualidades bloqueantes de características de estas últimas. Por esta razón, y pese a que las bases de datos relacionales superan a las *NoSQL* en este aspecto, es necesario evitar este tipo de consultas si se quiere conseguir una mayor escalabilidad y eficiencia.

En mi caso, utilizando *NodeJS*, existe un módulo de *MongoDB* llamado *mongoose* que proporciona funcionalidades tan interesantes como las de *aggregate()* y *populate()*. Estas funciones que se nos presentan permiten la división de las consultas en dos: se construye una consulta que recupera los datos de una colección y para cada uno de los *IDs* que hacen referencia a otra colección (*aggregate*) se hace la llamada a otra con la finalidad de obtener los datos e integrarlos en el *JSON* final (*populate*) que conforma nuestro objeto en base de datos. Siguiendo esta lógica es posible separar operaciones únicas de manera que puedan ser ejecutadas de forma secuencial.

5.1.4 Test

La implementación de test sobre la aplicación web desarrollada en base a la arquitectura propuesta corre a cargo de: *Jasmine*, *Karma*, *Mocha*, *Chai* y *Protractor*.





23. Estructura de la arquitectura con las tecnologías test

5.1.4.1 Test unitarios

5.1.4.1.1 Jasmine y Karma

El desarrollo de test unitarios para la parte de *front-end* con *Angular* y *TypeScript/JavaScript* se realiza con la ayuda de *Jasmine* y *Karma*.

Jasmine es un *framework* simple que permite la programación de pruebas para entornos *JavaScript* de forma síncrona o asíncrona (Burguess, 2011). Los test resultan muy visuales a la hora de escribirlos e interpretarlos debido a la proximidad de sus comandos con el lenguaje escrito. A pesar de que su utilización es posible en procesos desarrollo *software* como lo es TDD, esta herramienta se encapsula dentro de los BDD, los cuales están más focalizados en los aspectos de comportamiento que en los puramente técnicos. Como indican *Dan North & Associates* (2006), las técnicas BDD han ido evolucionando con la aparición de las metodologías ágiles, facilitando las tareas de análisis y automatización de test para este tipo de proyectos.



24. Jasmine y Karma

Por el otro lado, *Karma* es el *test runner* (Braithwaite, 2015) encargado de ofrecer un ámbito de ejecución en el que se puedan lanzar los test unitarios implementados con *Jasmine*. Mediante un fichero de configuración, necesario para su funcionamiento, es posible especificar múltiples opciones para la personalización de: el navegador sobre el que se ejecutan los test, aquellos ficheros de test que se deseen llevar a cabo, un *watcher* encargado de vigilar cualquier cambio realizado, etc. Los resultados pueden ser estudiados desde la consola donde han sido lanzados o mediante un entorno HTML más visual que facilita el control de los test mediante *plugins* tan interesantes como *karma-jasmine-html-reporter*.

5.1.4.1.2 Mocha y Chai

De la misma manera que en el *front-end*, el desarrollo de test unitarios para el *back-end* cuenta con dos tecnologías: *Chai* y *Mocha*.

Chai, igual que *Jasmine*, es una librería BDD/TDD para el desarrollo de test unitarios. La diferencia entre ellos es que, mientras *Jasmine* se encarga de la parte del *front-end*, *Chai* está diseñado para desarrollar test unitarios para *back-end* con *Node.JS* en *JavaScript*. La tipografía utilizada en *Chai* es muy similar a la empleada por *Jasmine*, por lo que es posible el compartir conceptos y facilitar su aprendizaje.



25. Mocha y Chai

Mocha está catalogado como un *framework* en *NodeJS* encargado de la ejecución de test en la parte de *back-end*.

5.1.4.2 Test de integración

5.1.4.2.1 Protractor

Protractor es un *framework* para test *E2E* diseñado originalmente para la primera versión de *Angular*, es decir, *AngularJS*. Su uso permite la automatización de las funcionalidades de la web, así como la interacción con la interfaz, como si se tratase de un usuario real.



26. Protractor

5.1.5 Sistema de control de versiones

Los sistemas de control de versiones proporcionan el poder tomar las riendas del código que conforma un proyecto, así como los archivos que lo componen, aumentando el nivel de productividad dentro del equipo.

El uso de tecnologías para el control de versiones no es una práctica obligatoria para el desarrollo de un proyecto. Antes de que estos existiesen la implementación de aplicaciones *software* ya era vigente, sin embargo, existían situaciones que dificultaban la coordinación entre miembros de un mismo equipo, originando conflictos que afectaban tanto a la calidad como a la agenda del producto en desarrollo. En este tipo de proyectos, es posible observar un nivel bastante pobre de comunicación, aspecto vital si la implementación se lleva a cabo por un grupo de personas. Es posible encontrar casos en el que el canal principal de comunicación reside en el *email*, donde miembros de un mismo equipo comparten ficheros referentes al *software* en cuestión y discuten las modificaciones y conflictos que van surgiendo. Esta forma rudimentaria de compartición de código, imposibilitando el establecimiento de un registro de cambios y origina una situación de caos interno que puede llegar a repercutir en el producto final entregado al cliente.

Al no existir un banco de modificaciones con las versiones sobre las que se han realizado, la acción de sobrescribir un fichero o borrarlo por accidente suponen un de riesgo que somete al equipo a preocupaciones que deberían estar controladas de antemano. En el caso de que alguien modifique la última versión y estos cambios desencadenen un error fatal para la aplicación, la recuperación del estado anterior de los ficheros modificados se convierte en tarea imposible. Con la finalidad de solventar estos y muchos otros obstáculos, surgieron los sistemas de control de versiones.

La finalidad principal de estas herramientas es el establecer un directorio consistente y común a todos los miembros de un mismo equipo. Este directorio se encarga de guardar los cambios realizados, registrando la persona que los realizó y la razón de ser. De este modo, se construye poco a poco un histórico de todas las modificaciones realizadas a lo largo de la vida del proyecto, permitiendo a los desarrolladores acceder a cualquiera de las anteriores versiones en caso de experimentación de errores.

La comunicación y compartición de ficheros no requiere de herramientas externas como el email ya que su naturaleza permite a los desarrolladores la resolución de conflictos o el intercambio de ficheros a través de la misma herramienta.

Para entender mejor el ecosistema de los sistemas de control de versiones, Neagle (n.d.) recomienda el entendimiento de los siguientes conceptos, cruciales para el correcto aprendizaje de su uso:

- **Trazabilidad de cambios:** el objetivo principal que da lugar a la existencia de controladores de versiones es el de mantener una trazabilidad con respecto a todo cambio que se haya realizado dentro de los directorios y archivos de un mismo proyecto. Para ello, estos sistemas permiten ser configurados por el usuario con el fin de especificar que partes quieren ser escuchadas a la espera de cambios. Estos directorios, a la espera de reportar cualquier actualización, están catalogados como repositorios.
- **Committing:** cuando se han realizado todos los cambios pertinentes considerados aptos para ser dados de alta, los desarrolladores llevan a cabo la acción de *commit*. Los controladores de versiones están siempre a la escucha de nuevas modificaciones: creación y borrado de ficheros o directorios, cambios de nombre, cambios internos a cada fichero, etc., pero estos son lo suficientemente inteligentes como para no actualizar a una nueva versión a cada cambio realizado. Sería exhaustivo el hecho de actualizar las versiones por aspectos tan sencillos como pueden ser la rectificación de un punto y coma o la adición de un nuevo parámetro. Para ello, los controladores de versiones mantienen un registro de cambios hasta que el desarrollador considere que hay suficientes modificaciones como para ser añadidas mediante la acción de *commit*.
- **Revisiones:** al hacer *commit* de los cambios, se crea una referencia dicha versión. Cada versión es única a cada revisión y esta, a la vez, contiene datos sobre su realizador y los motivos por los que se ha llevado a cabo. De este modo es posible mantener un historial organizado con todos los cambios realizados por cada miembro del equipo sobre el proyecto.

En caso de un posible fallo en alguno de los *commits*, siempre existe la opción de inspeccionar el historial, identificar la última revisión funcional y revertir los últimos cambios realizados. Con esta ayuda, los desarrolladores pueden modificar sus versiones sin ningún miedo a perder el trabajo previo. El nivel de repercusión en caso de error vendrá dado por la frecuencia de *commits* realizada.

- **Actualizaciones:** cuando algún participante del el equipo incorpora una nueva revisión, es aconsejable actualizar la versión local para cada uno lo antes posible. La posibilidad de conflicto entre versiones se ve reducida proporcionalmente al tiempo que transcurre desde que se actualiza el repositorio común hasta que se descargan los cambios en local.
- **Control de cambios:** como se ha descrito más arriba, cada *commit* realizado guarda consigo información referente a los cambios realizados. Existe la opción de explorar entre los archivos modificados y observar las líneas de código que han sido modificadas para cara directorio o fichero.



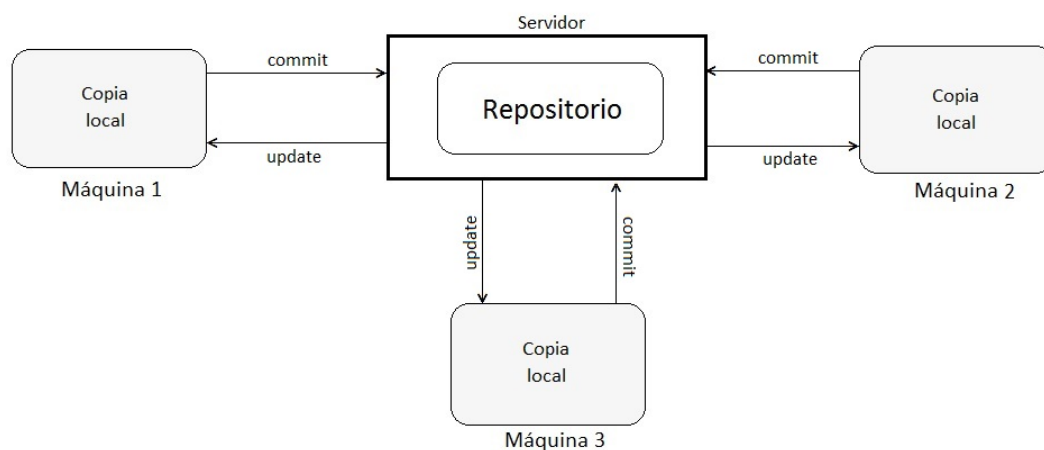
- **Branching y merging:** los sistemas de control de versiones ofrecen nuevas formas de trabajar que facilitan la tarea a los desarrolladores sin temor a romper el la aplicación. Estos ofrecen opciones tan interesantes como la creación de ramas o *branching*, cuya finalidad es la de crear una copia paralela de la rama principal para poder realizar modificaciones sobre ella sin que los errores que puedan llegar a darse afecten a esta. Contando con semejante ayuda, integrar nuevas funcionalidades o cambios no supone una tarea aterradora en la que, si algo sale mal, el desarrollo del producto final puede verse comprometido.

Del mismo modo que se crean ramas que se separan dela principal para efectuar cambios sobre ella, estas necesitan ser integradas de nuevo para poder aplicar las modificaciones de forma definitiva. Para ello, los sistemas de control de versiones ofrecen la opción de *merging*, donde se realiza un proceso automático encargado de detectar los ficheros que han sido alterados o creados y compararlos con los existentes para hacerlos efectivos.

A continuación se exponen los tipos de sistemas de control de versiones que existen: centralizados y distribuidos.

5.1.5.1 Sistemas centralizados

Los sistemas de control centralizados, como su nombre indica, están dispuestos en una estructura cliente-servidor clásica. En estos sistemas, el repositorio al que se apunta se encuentra localizado en un servidor de posición fija, con lo que los usuarios deben estar conectados a él para tener acceso. Como explica Verma (2015), esto puede suponer una limitación ya que, si el servidor sufre algún tipo de mal funcionamiento impidiendo su acceso, el equipo que se puede encontrar en una situación en la que su trabajo estará bloqueado por incapacidad de acceder al repositorio. No obstante, este tipo de sistemas ofrece alguna que otra ventaja.



Al ser el servidor la única puerta de acceso al repositorio, permite ofrece un mayor control sobre los usuarios que tienen acceso a él, estableciendo un sistema de seguridad por defecto. La curva de aprendizaje es relativamente baja, convirtiéndolos en los sistemas por excelencia para aquellos usuarios que se adentran en el mundo de los sistemas de control de versiones por primera vez mientras que los comandos necesarios para llevar a cabo tareas como el *branching* o *merging* suelen acarrear complicaciones extra. CVS y *Subversion* representan algunos de los más conocidos.

5.1.5.1.1 Subversion

Subversion es un *software open source* de control de versiones que permite la dirección de archivos y directorios, así como la recuperación de versiones antiguas y un registro del historial de cambios. Como Collins-Sussman, Fitzpatrick y Pilato describen en su libro *Version Control with Subversion* (2006), esta herramienta perfectamente podría adoptar la definición de “máquina del tiempo”. Reúne todos los aspectos que caracterizan a los CVCS: estructura cliente-servidor, facilidad de uso, repositorios no limitados, etc.

Actualmente, y en vísperas a un cambio cada vez más cercano, el sistema utilizado por excelencia para la gestión de código compartido en Everis corresponde a *Subversion*. Ciertamente, hasta mi incorporación en Everis, únicamente había escuchado hablar de él en pocas ocasiones y su funcionamiento no quedaba del todo claro para mí. Al poco tiempo de empezar a gastarlo, descubrí que la curva de aprendizaje es bastante simple. Te conectas al servidor requerido la primera vez, seleccionas los archivos que deseas compartir, añades un comentario en referencia al trabajo que has realizado y le das a subir cambios. Cabe decir también que se cuenta con la ayuda de *Tortoise*, un cliente para *Subversion* que facilita la tarea a la hora de gestionar los cambios.

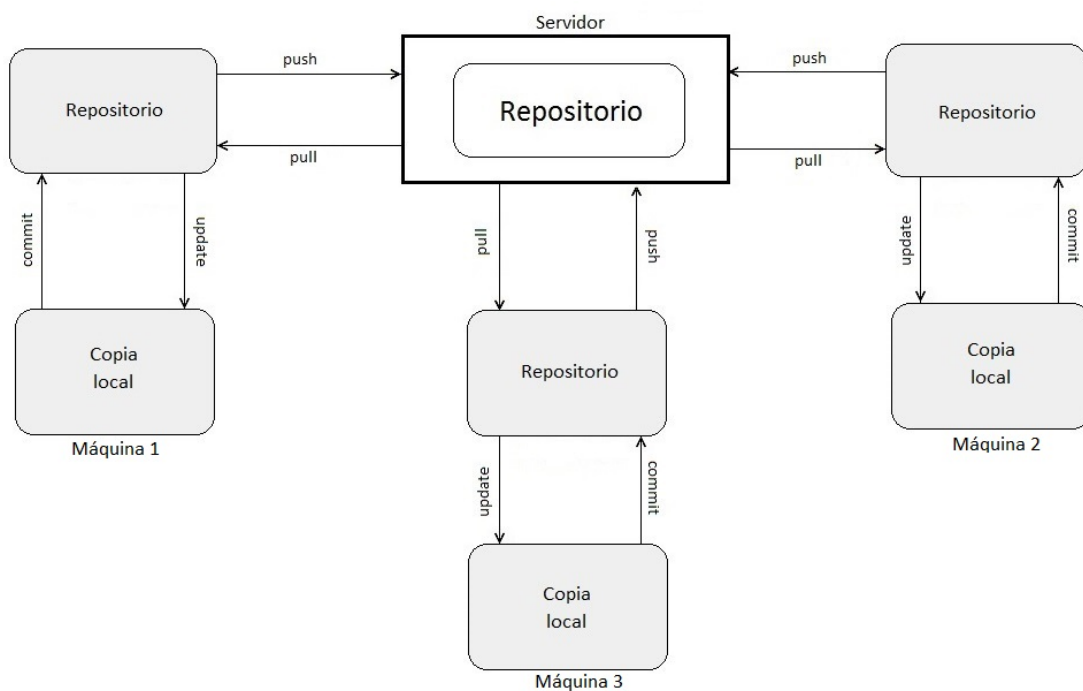
A primera vista pensé que era una forma genial y sencilla de compartir trabajo entre varios desarrolladores pero, conforme avanzó el tiempo, se daban situaciones donde no es tan cómodo utilizarlo. Como se comenta más avanzado en el documento, la idea de integración continua con el apoyo de los sistemas de control de versiones se basa, mayormente, en la implicación del equipo al seguir las pautas necesarias. Más de una vez se ha dado la situación en el que el servidor en el que se encuentra el repositorio caía, impidiendo así la integración de cambios por parte de cualquier compañero. Frente a tal problema y en casos en los que era necesario para no bloquear el trabajo de nadie, se recurría al punto más básico, copiar los archivos en un USB e ir pasándolo de persona en persona. Esto se puede realizar de vez en cuando si realmente es necesario pero no representa la forma adecuada de trabajo.

Por si fuera poco, existen proyectos en los que se necesita acceder al servidor del cliente para descargarse los últimos cambios mediante VPN. Esta práctica también contiene sus limitaciones como el acceso imposible mediante la red *Wi-Fi*, necesitando una entrada *ethernet* con IP que finalice en siete para lograr conectarse y actualizar a la última versión.



5.1.5.2 Sistemas distribuidos

Los sistemas de control de versiones distribuidos o descentralizados predicen la existencia de una copia local del repositorio de trabajo por miembro perteneciente al grupo. En este caso, no existe ningún servidor único que contenga el código, cada participante trabaja en la copia clon del repositorio localizado en su máquina y se encarga de incluir los cambios realizados sobre ella de forma regular. Al contar con una estructura descentralizada, los desarrolladores tienen la capacidad de trabajar sin conexión a la red una vez han realizado la clonación desde el repositorio principal. Únicamente las acciones para actualizar la copia individual o subir cambios a la general requieren conexión a internet. En caso de falla sobre el repositorio principal, los desarrolladores pueden generar una nueva rama que la sustituya de forma inmediata y gracias a su copia local.



28. DVCS

A diferencia de los sistemas centralizados, los DVCS presentan una curva de aprendizaje más elevada (Verma, 2015). Aquí, el usuario debe aprender a familiarizarse con el uso de comandos requeridos para llevar a cabo las acciones deseadas. La respuesta ante los comandos ejecutados es de gran rapidez, demorándose únicamente durante el primer proceso de clonación en local o con aquellas acciones que necesiten acceso a la red como a la hora de subir los cambios realizados. Acorde con Neagle (n.d.) los DVCS aventajan a los CVCS en tareas tan importantes como lo son el *branching* y *merging*. *Git* y *Mercurial* se encuentran entre los sistemas de control de versiones distribuidos más populares del momento.

5.1.5.2.1 Git

Git no solo representa un sistema de versión de controles sino que fundamenta los cimientos de servicios como *Bitbucket*, *GitHub* o *GitLab* (Kenlon, 2016). Sin ningún añadido, *Git* solo es una aplicación diseñada para ejecutarse desde un terminal en un entorno *Linux*, sin embargo, existen múltiples proyectos *open source* desarrollados para ofrecer nuevas formas de acceso a dichos comandos. Muchos de ellos ofrecen aplicaciones web que minimizan la curva de aprendizaje con respecto a la ejecución original mediante terminal.

Acorde con Bruce (2012), la característica clave de *Git* es su capacidad para tomar capturas del estado actual de un proyecto y convertirlas en versiones únicas. Con este mecanismo, es posible establecer un historial de versiones en el que retroceder sobre los pasos seguidos en caso de llegar a un punto de error, pudiendo continuar desde la última versión estable.

Para esta prueba de concepto y con la finalidad de establecer un cambio de pensamiento en lo que a sistemas de control de versiones se refiere, *Git* es el candidato elegido para realizar esta gestión. Para su uso, se emplea junto a la herramienta web para repositorios *Git* o *Mercurial* llamada *Bitbucket*.



29. Bitbucket

Para tener un mejor visión de que es *Git*, cuál es su forma de trabajar y uso correcto, se presenta a continuación su modo de empleo, paso a paso y descrito de forma impecable por Driessen (2010). Dicho modelo para el control de versiones de una misma aplicación es el elegido para comandar el actual proyecto de la aplicación web piloto (*On Air*), así como los futuros proyectos que surjan a partir de esta apuesta por un cambio filosofía y trabajo.

Git ofrece una amplia cobertura y fiabilidad a la hora de realizar acciones como el *mergin* o el *branching*, convirtiéndolas en acciones simples que en sistemas como *Subversion* infunden un mayor miedo debido a no presentarse de una forma atractiva. Con *Git*, dichas acciones representan algo rutinario que se lleva a cabo con una alta frecuencia sin que nadie sienta que puede repercutir negativamente en el desarrollo. Es por esto que *Git* se presenta como la solución por excelencia dentro del mundo de los sistemas de control de versiones.

Como se ha comentado anteriormente, *Git* está clasificado como un sistema de control de versiones descentralizado, no obstante, es una práctica común el contar con un repositorio central denominado *origin* y común a todos los desarrolladores. Cada miembro del equipo trabaja en base a *origin*, empezando con la clonación de este para obtener una copia local en la que trabajar individualmente. Teniendo la copia local, cada desarrollador puede dedicarse a realizar sus tareas asignadas y ser responsable de actualizar la rama principal de forma regular, actualizando siempre su repositorio en primer lugar. La interacción entre repositorios



no es exclusiva entre las copias locales y la rama *origin*, también es posible la comunicación entre ramas de los mismos desarrolladores, ayudándose unos a otros en caso de que la situación lo demande.

La estructura principal encontrada en la mayoría de los proyectos desarrollados con la presencia de *Git* cuenta con un repositorio central formado por dos ramas: *develop* y *master*. La primera de ellas representa el estado actual de desarrollo del proyecto y en ella se van integrando los cambios realizados por los miembros que lo conforman. Cuando se ha llegado a una revisión lo suficientemente estable como para considerarse como entrega, la versión seleccionada se incluye en la rama *master*, en la cual se encuentran todas aquellas que una vez representaron una entrega. En el caso del proyecto descrito en este documento, la estructura de este consta de dos repositorios: uno para la parte *back-end* y otra para la parte *front-end*, que al mismo tiempo tiene dos ramas cada una, la *master* y la de *develop* (denominada *on-air* o *on-air-server* dependiendo del repositorio). Para trabajar de forma correcta, se empieza haciendo una clonación local de la rama *master* en mi máquina local. Al tratarse de un proyecto real, se ha con la supervisión de mi tutor en Everis por lo que se apunta a la rama *on-air* y *on-air-server* como repositorios en los que debería trabajar. De tal manera, el tutor se podría encargar de controlar el trabajo y comprobar cuando la versión está en buen estado para integrarse en *master*.

Pero no solo existen estas dos ramas, es posible encontrar y definir otros tipos para cumplir los diferentes propósitos. Para conseguir un mayor nivel de paralelismo entre los desarrolladores de un mismo proyecto, Driessen (2010) denomina tres tipos de ramas, de usar y desechar, que pueden ayudar a un mejor control. Estas ramas se clasifican en: ramas de funcionalidades, ramas de entregas y ramas *hotfix* o solucionadoras de errores.

- **Rama de funcionalidad:** son ramas que se originan desde *develop* y cuya integración final apunta a esta misma. Este tipo de ramas nacen con la finalidad de dar soporte a una funcionalidad específica del proyecto que se quiere tener implementada para un punto en concreto de la agenda. Un ejemplo aplicado a esta página web piloto sería la codificación de la gráfica de araña para el cálculo de calidad de aire máxima de cada contaminante. Para ello, sería necesario crear una nueva rama con el nombre de la funcionalidad que colgase directamente de *on-air* (rama que representaría a *develop* dentro de este proyecto). Una vez hecho esto, se empezaría a desarrollar dicho componente, siempre incorporando los cambios a su propia rama e integrándose con *on-air* cuando su desarrollo estuviese completo.

Las ramas de funcionalidad podrían considerarse como “usar y tirar” ya que, una vez cumplido su propósito, no tendría sentido continuar manteniéndolas.

- **Ramas de entrega:** se originan desde *develop* y tienen como finalidad acabar integrándose con *master* o *develop*. Antes de la entrega de una versión del producto, Driessen (2010) puntualiza la necesidad de preparar algunos aspectos para su entrega o resolver aquellos pequeños problemas que puedan surgir. Para ello, y con la finalidad de no bloquear la rama *develop* al resto de desarrolladores, existe la posibilidad de crear un rama específica sobre la que realizar dichas tareas. De este

modo, los miembros del equipo pueden seguir trabajando en funcionalidades que necesiten ser implementadas para la siguiente revisión.

En el preciso momento en el que se crea esta rama, la versión es bautizada acorde con las reglas establecidas antes de empezar el proyecto. Cuando el resultado está pulido y listo para ser entregado, se llevan a cabo dos acciones: se actualiza la rama *master* con la última revisión del producto preparada para ser entregada y se integra la rama de entrega con *develop* para reflejar las modificaciones hechas en esta primera.

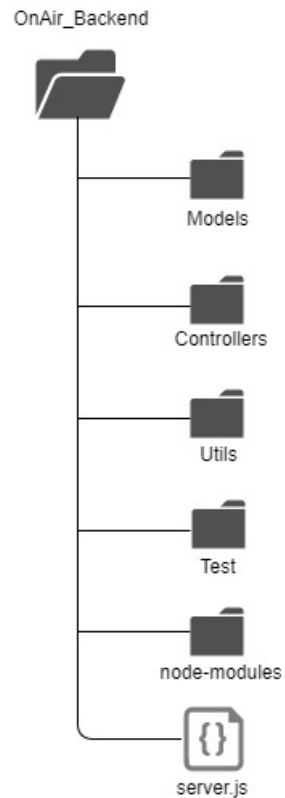
- **Ramas *hotfix*:** nacen desde la rama *master* y se integran en esta misma y *develop*. Como su nombre indica, este tipo de ramas surge junto a la aparición repentina de errores en versiones listas para la entrega y que se localizan en la rama *master*. El hecho de que se detecte un *bug* en una versión en producción desata una alarma que debe corregirse lo antes posible. Con el fin de atacar dicho problema, se puede crear una rama en concreto para el tratamiento de dichos errores, de este modo, es posible trabajar en paralelo solventando los problemas mientras el desarrollo continúa. Una vez encontrada la solución y cerrado el problema, los cambios realizados se integran con la rama *develop* y la rama *master* recibe una nueva versión con su correspondiente etiquetado.

Los tres tipos de ramas que he descrito son ideas propuestas por Diressen (2010) como ejemplo del poder de *Git* a la hora del *branching* y *merging*. Existen muchas más posibilidades para exprimir el potencial que ofrece esta herramienta, adaptándose siempre al tipo de proyecto con el que se esté trabajando.

5.2 Estructura de ficheros y directorios

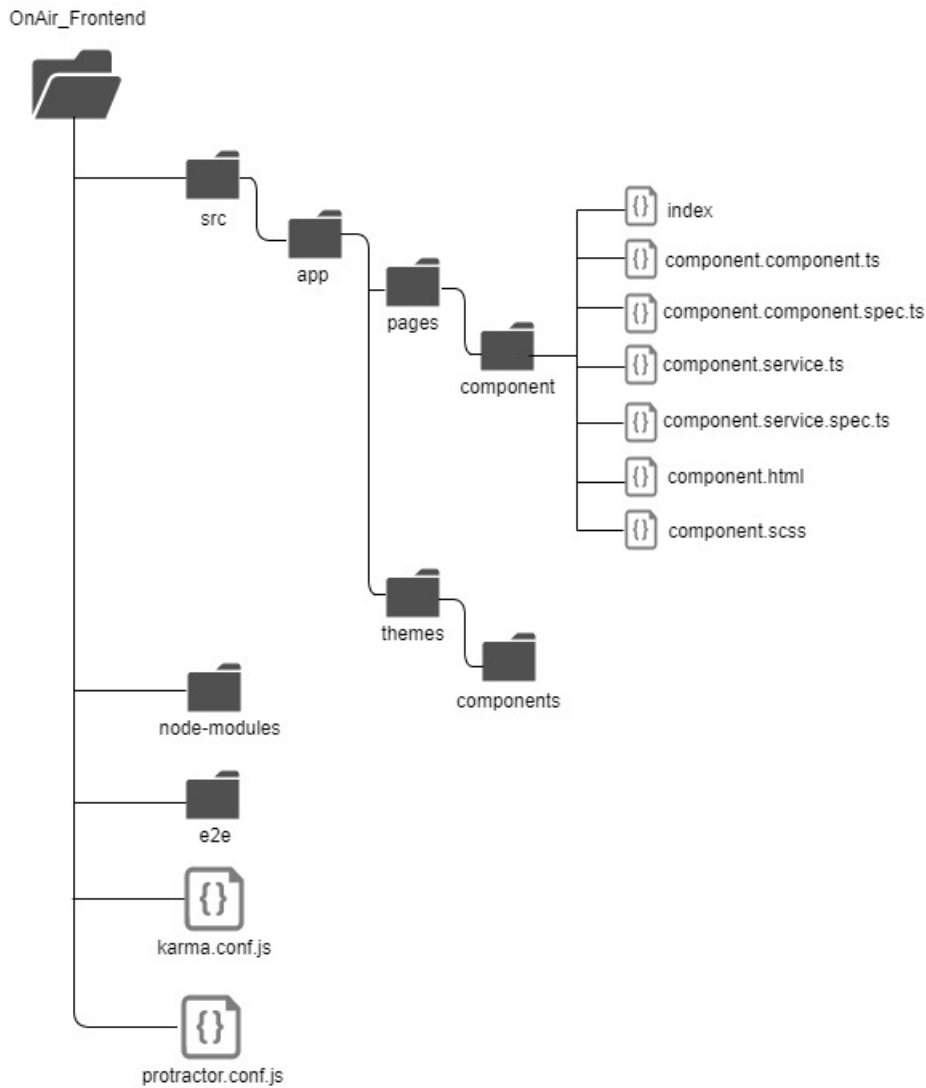
El desarrollo del proyecto se ha realizado en dos directorios diferentes: uno con el contenido de la parte *front-end* y el otro con la parte *back-end*. Debido a la longitud de la estructura de directorios, sobre todo la de la parte *front-end*, se ha decidido mostrar el panorama general de la organización de carpetas y ficheros nombrando aquellos con mayor peso dentro del proyecto.

- **Back-end:** directorios y ficheros clave de la parte *back-end*.



30. Estructura de repositorios de la parte back-end

- **Models:** contiene los modelos de datos de las colecciones representadas en el diagrama de clases.
 - **Controllers:** contiene los controladores de cada colección mediante los que se realizan operaciones sobre ellas. Ejemplo: acciones CRUD.
 - **Utils:** contiene el fichero *azure-hub-connect* encargado de la conexión con el servidor *lot Hub* para la gestión de dispositivos, usuarios, sensores, categorías y alarmas.
 - **Test:** contiene los ficheros test para cada controlador del directorio *Controllers* realizados con *Chai*.
 - **node-modules:** contiene todos los paquetes instalados mediante *npm*.
 - **server.js:** contiene las rutas con las llamadas a los métodos de cada controlador.
-
- **Front-end:** directorios y ficheros clave de la parte *front-end*.



31. Estructura de repositorios de la parte front-end

- **src/app/pages:** contiene todos los componentes creados específicos para este proyecto, encapsulados cada uno en su propia carpeta. El ejemplo de la imagen muestra una carpeta *component* con los ficheros que contiene cada una de ellas:
 - **index:** contiene la referencia al componente.
 - **component.component.ts:** contiene las llamadas a los métodos del *componenten.service.ts*, así como la referencia a los archivos *component.html* y *component.scss*.
 - **component.component.spec.ts:** contiene los test unitarios realizados con *Jasmine* para el fichero *component.component.ts*.
 - **componenten.service.ts:** contiene los métodos de cálculo y llamadas de datos requeridas por el componente.

Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

- **component.service.spec.ts:** contiene los test unitarios realizados con *Jasmine* para el fichero *component.service.ts*.
- **component.html:** contiene la estructura visual de los elementos.
- **component.scss:** contiene los estilos del componente.
- **src/app/themes:** contiene los componentes genéricos de la aplicación web, bajo la carpeta *components*, como pueden ser: barra superior, menú lateral, notificaciones, pie de página, etc.
- **node-modules:** contiene todos los paquetes instalados mediante *npm*.
- **e2e:** contiene los ficheros *e2e-spec* con los test de integración desarrollados con *Protractor*.
- **karma.conf.js:** contiene la configuración de *Karma* para los test unitarios desarrollados con *Jasmine*.
- **protractor.conf.js:** contiene la configuración de *Protractor* para los test de integración localizados en el directorio *e2e*.

6. Testeo

El sexto capítulo se encarga de cubrir uno de los claros objetivos establecidos dentro de este proyecto, el desarrollo de test con vistas al futuro de su automatización. Se ha desarrollado e implementado algunas pruebas haciendo uso de las tecnologías descritas en capítulos anteriores. Dichos test se encargan de garantizar el correcto funcionamiento de la prueba piloto presentada a lo largo de esta memoria y servirán, más adelante, para su ejecución automática a la hora de realizar cualquier despliegue.

6.1 Jasmine y Karma

Para el desarrollo de los test de *front-end*, se ha hecho énfasis en los diferentes componentes que forman la aplicación en sí. Cada uno de ellos cuenta con su plantilla *HTML*, *CSS*, componente, servicio y fichero *spec* donde se deben escribir los test haciendo uso de la terminología que ofrece *Jasmine*. Las partes importantes que se han tenido en cuenta a la hora de aplicar las pruebas unitarias son: el componente tiene que mostrar el gráfico o mapa correspondiente cuando se carga la página, la comunicación entre el componente y su servicio para la obtención de datos, la comunicación del servicio con el *back-end* para la obtención de datos, la correcta estructura de los datos que se deben recibir y el cálculo de las fórmulas relacionadas con la calidad del aire y umbrales de los contaminantes.

Para ejecutar dichos test a estas alturas del proyecto, se ha implementado un *script* que se encarga de ejecutar de forma instantánea todos los test que se encuentren en el *front-end*. Para ello, ha sido necesaria la configuración necesaria de *Karma* en su fichero *karma.conf.js*.

```
27 05 2017 19:21:00.709.INFO [launcher]: Launching browser Chrome with unlimited
concurrency
27 05 2017 19:21:00.753.INFO [launcher]: Starting browser Chrome
27 05 2017 19:21:00.388.INFO [Chrome 58.0.3029 (Windows 7 0.0.0)]: Connected on
socket ypi0UTQtwyLHum4AAAA with id 53989556
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 1 of 17 SUCCESS (0 secs / 0.004 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 2 of 17 SUCCESS (0 secs / 0.005 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 3 of 17 SUCCESS (0 secs / 0.005 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 4 of 17 SUCCESS (0 secs / 0.006 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 5 of 17 SUCCESS (0 secs / 0.006 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 6 of 17 SUCCESS (0 secs / 0.006 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 7 of 17 SUCCESS (0 secs / 0.006 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 8 of 17 SUCCESS (0 secs / 0.006 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 9 of 17 SUCCESS (0 secs / 0.006 sec
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 10 of 17 SUCCESS (0 secs / 0.006 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 11 of 17 SUCCESS (0 secs / 0.006 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 12 of 17 SUCCESS (0 secs / 0.006 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 13 of 17 SUCCESS (0 secs / 0.006 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 14 of 17 SUCCESS (0 secs / 0.007 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 15 of 17 SUCCESS (0 secs / 0.007 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 16 of 17 SUCCESS (0 secs / 0.007 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 17 of 17 SUCCESS (0 secs / 0.007 se
Chrome 58.0.3029 (Windows 7 0.0.0): Executed 17 of 17 SUCCESS (1.731 secs / 0.00
7 secs)
```

32. Visualización de los test Karma y Jasmine en consola

Los resultados de las pruebas se muestran en la consola donde se haya ejecutado el *script* o, también pueden ser visualizados en el navegador gracias a un *plugin* para *Karma* que así lo

permite. Las imágenes 32 y 33 muestran las diferentes visualizaciones en una misma ejecución de test.



33. Visualización de los test Karma y Jasmine en el navegador

6.2 Mocha y Chai

En la implementación de test para el *back-end*, la parte crucial a testear son las llamadas a base de datos que se encargan de obtener y guardar los datos de cada sensor. Para ello, se focaliza en comprobar los siguiente: la obtención de todos los sensores, el estado de la conexión, que el *array* de datos no esté vacío, el formato de los datos recibidos, los parámetros de cada objeto así como el tipo de cada uno de ellos, la correcta creación de un nuevo registro, etc.


```

SENSORS
  /GET getHistory()
    1) should list ALL sensors
      U should have a 200 HTTP status
    2) should not be empty
      U should have a JSON format
      U should be an array
  /POST postSensors()
    U should have a 200 HTTP status
    U should not be empty
    U should have a JSON format
    U should be an object
    U should have the properties: PM25, CO, NO2, O3, humidity, temperature, pressure, timestamp, latitude, longitude and deviceId
    U should have the types: number, number, number, number, number, number, number, date, number, number and string
    U should post a sensor on /sensors POST postSensor()

10 passing (464ms)
2 failing

```

34. Visualización de test Mocha y Chai en la consola

En el caso de *Mocha*, no se ha encontrado otra forma de visualizar los test que no sea a través de la consola. En la imagen de arriba es posible observar los resultados de los test escritos en *Chai* que *Mocha* se ha encargado de lanzar mediante un *script*. En este caso en concreto, es posible observar que solo diez de los doce test han pasado la prueba y, por lo tanto, aparecen en verde. Aquellos que se muestran en rojo corresponden a test que no han cumplido con lo que se les pedía y cuyas funciones deben ser revisadas.

6.3 Protractor

El diseño de test *E2E* con *Protractor* ha significado la comprensión de la interacción del usuario con la aplicación, es decir, pensar en los caminos que estos pueden seguir para llevar a cabo las diferentes funcionalidades: iniciar sesión con su usuario y desconectarse, registrarse como nuevo usuario, cambiar la visualización de datos en los gráficos, etc. Esto implica que cada test representa una función en concreto que se describen paso a paso las acciones que realizaría una persona al interactuar con la aplicación.

7. Pruebas

El séptimo capítulo expone las condiciones de validación bajo las cuales la aplicación web piloto debe funcionar de forma correcta. En la segunda parte, se realiza una demostración de su funcionamiento paso a paso y con capturas. En ella se muestra la interacción que pueden tener los usuarios con las diferentes funcionalidades.

7.1 Validación

La aplicación web piloto ofrece una amplia capacidad de ejecución en un gran número de navegadores (<https://github.com/angular/angular>).

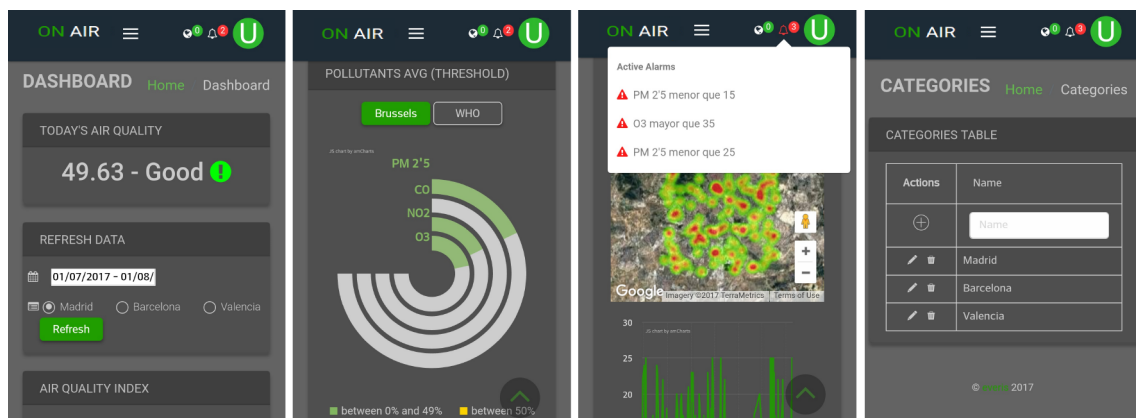
Firefox		Chrome		IE		Safari		Edge	
Versión	Compatible	Versión	Compatible	Versión	Compatible	Versión	Compatible	Versión	Compatible
50	Sí	54	Sí	9 (Windows 7)	Sí	7+	Sí	14	Si
				10 (Windows 8)	Sí				
				11 (Windows 8.1)	Sí				

24. Navegadores

Del mismo modo, su naturaleza *responsive* permite adaptar la resolución a distintos tamaños de pantalla, ya se trate de ordenadores o *smartphones*.

Tamaño	Mínimo (píxeles)	Máximo (píxeles)
XS	600	
Pequeño	601	959
Medio	960	1279
Grande	1280	1919
XL		1920

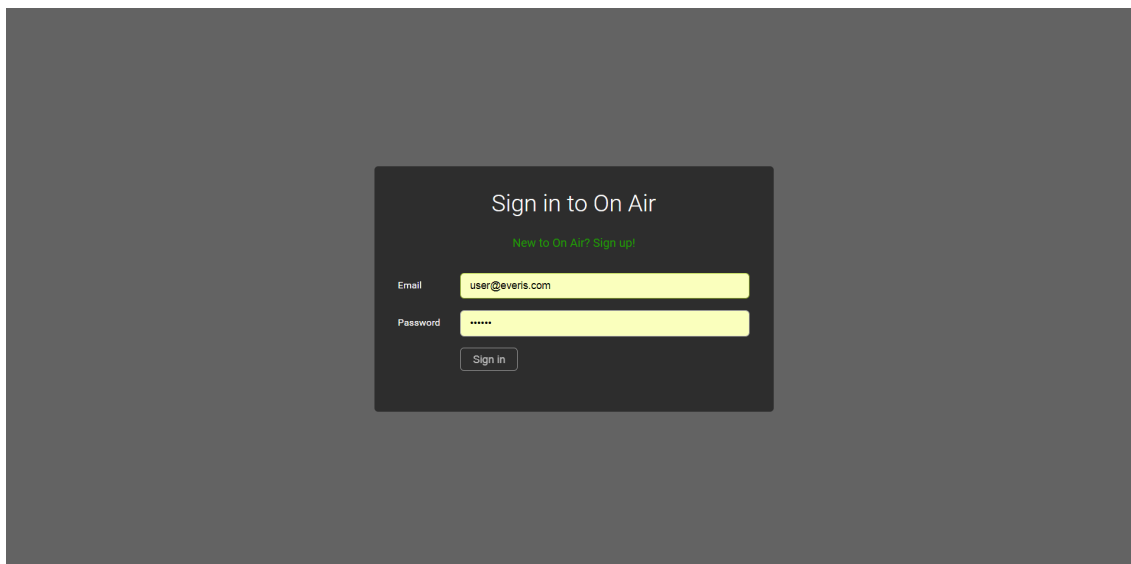
25. Resoluciones



35. Versión de smartphone

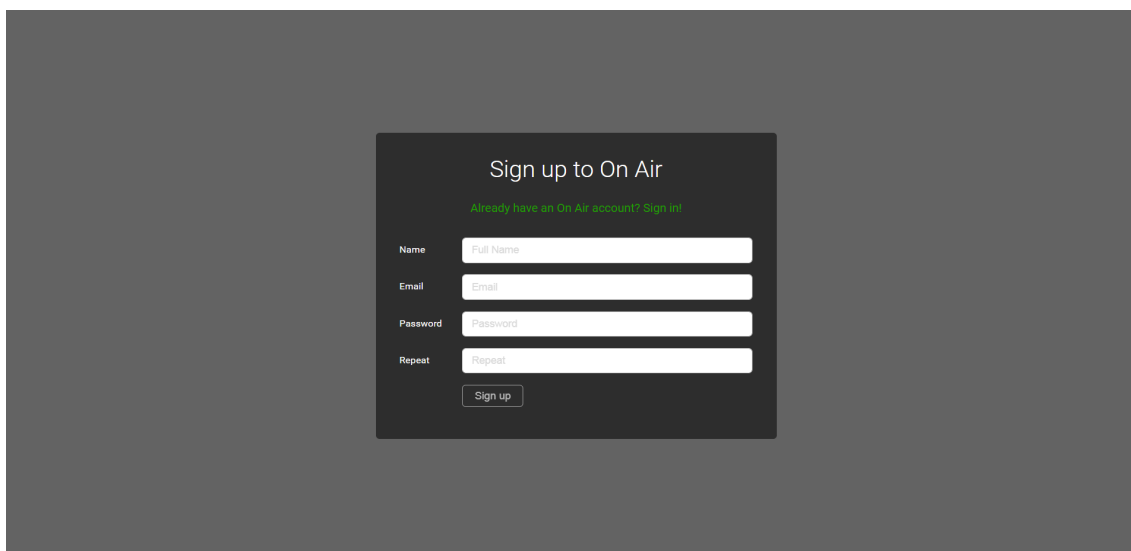
7.2 Pruebas de uso

La primera ventana que nos encontramos cuando cargamos la página web es la de *Login*. En ella se debe introducir el email y contraseña con el que se haya realizado el registro en la aplicación. En caso de no constar con una cuenta de la aplicación, se debe pulsar en el *link* “*New to On Air? Sign up!*” mediante el cual el usuario es redirigido a la ventana de registro.

A screenshot of a login form titled "Sign in to On Air". The form is centered on a dark gray background. It features a title "Sign in to On Air" in white, followed by a green link "New to On Air? Sign up!". Below this are two input fields: "Email" with the text "user@everis.com" and "Password" with six asterisks. A "Sign in" button is located at the bottom of the form.

36. Login

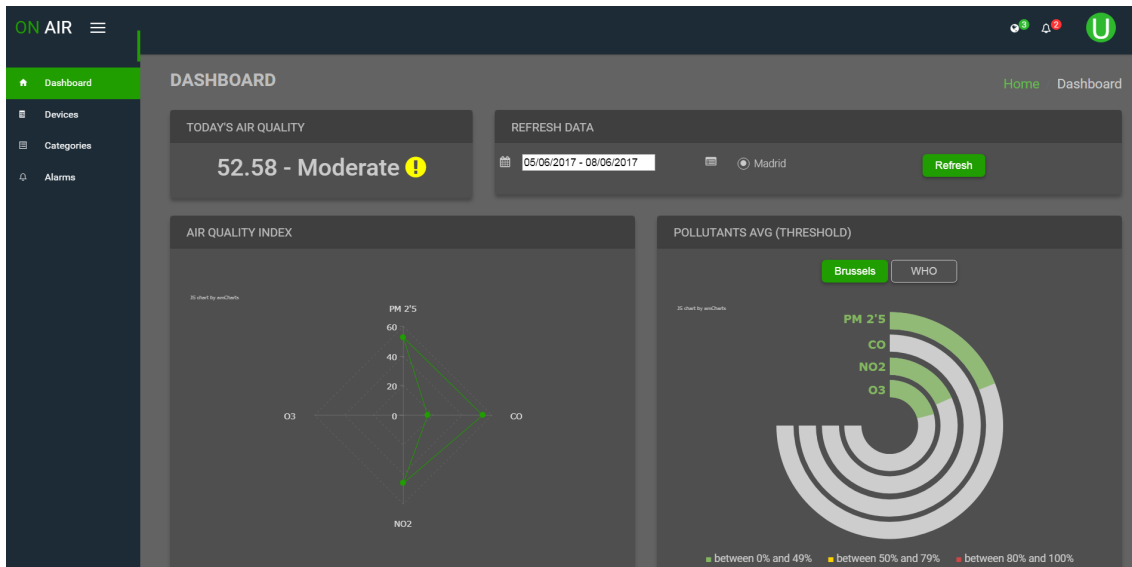
En la ventana de *Sign up* o registro, se deben introducir el nombre, email, contraseña y la confirmación de esta mediante un segundo campo con el fin de comprobar que la contraseña que se ha introducido es la deseada. Cuando se haya completado la información necesaria, se pulsa el botón Sign up para finalizar el proceso de registro y quedar reconocido por el sistema como usuario. En caso de que el usuario ya sea poseedor de una cuenta y pulse por error el *link* para registrarse, debe pulsar el *link* “*Already have an On Air account? Sign in!*” para ser devuelto a la pantalla de Login y poder introducir sus credenciales.

A screenshot of a registration form titled "Sign up to On Air". The form is centered on a dark gray background. It features a title "Sign up to On Air" in white, followed by a green link "Already have an On Air account? Sign in!". Below this are four input fields: "Name" with the placeholder "Full Name", "Email" with the placeholder "Email", "Password" with the placeholder "Password", and "Repeat" with the placeholder "Repeat". A "Sign up" button is located at the bottom of the form.

37. Sign up

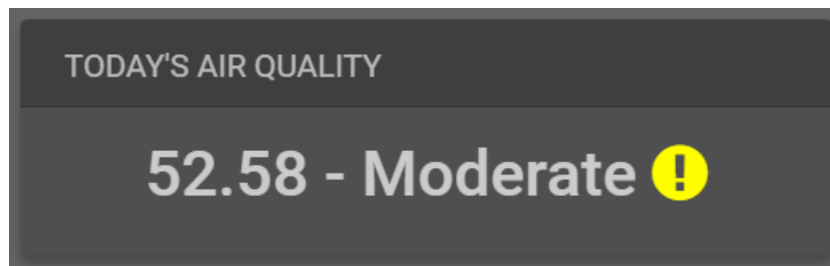
Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Iniciamos sesión y accedemos a la aplicación. La primera ventana que se observa es la del *Dashboard* encargada de interpretar y mostrar la información recibida por el sistema. Esta consta de una barra superior y otra lateral común a todas las ventanas. En la barra superior se puede encontrar un icono para colapsar el menú lateral, un icono de historial de notificaciones recibidas, otro de notificaciones para las alarmas activas y la información del usuario, a través de la cual se puede pulsar para acceder a la opción de *Sign out*.



38. Dashboard superior

El primer elemento que se puede observar es el informador sobre el índice de calidad del aire, que hace referencia al ICA más alto de la gráfica inferior. Dependiendo del valor de este, el mensaje mostrado y el color de cambian acorde con los umbrales establecidos.

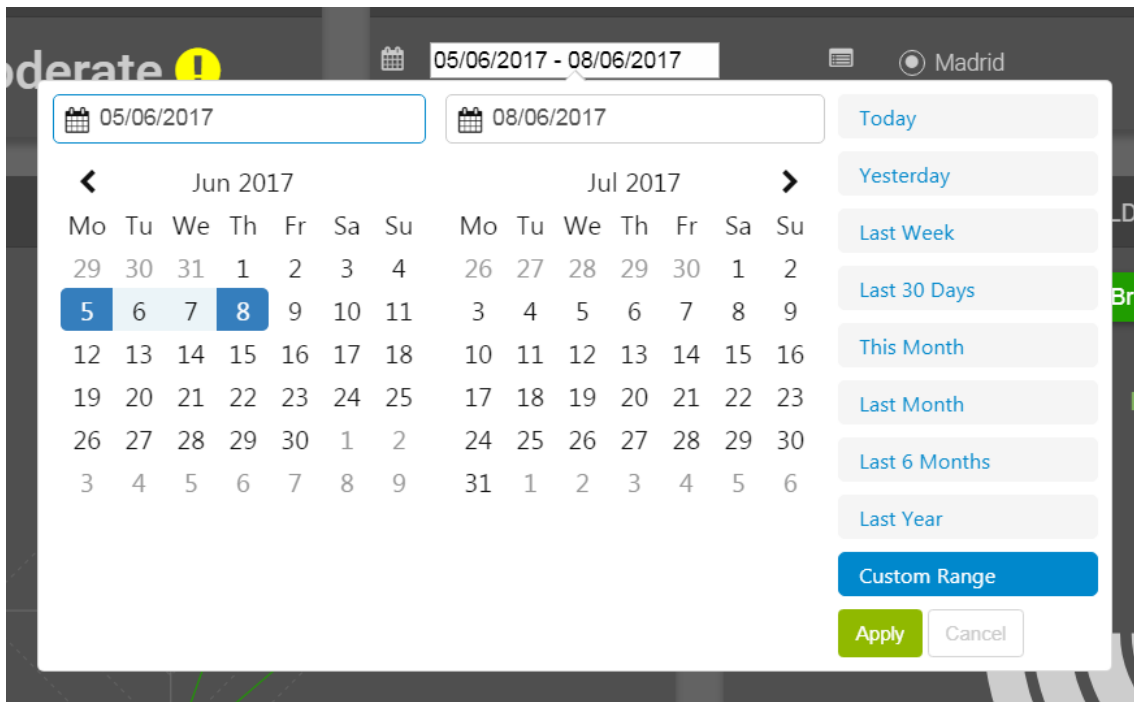


39. Calidad de aire diaria

El elemento *Refresh Data* es el encargado de gestionar el filtrado de información que desee establecer el usuario para su consulta. El primer campo a seleccionar se corresponde con el rango de fechas entre las que se quiere obtener información que será representada en los gráficos del *Dashboard*. Cuando se pulsa sobre él, se despliega un menú con dos calendarios para elegir las fechas deseadas sobre ellos o introduciéndolas de forma manual. Del mismo modo, a la derecha se ponen a disposición del usuario unos rangos de fechas predefinidos. Para hacer efectiva la selección, se debe pulsar el botón *Apply* o pulsar en cualquier punto fuera del él.

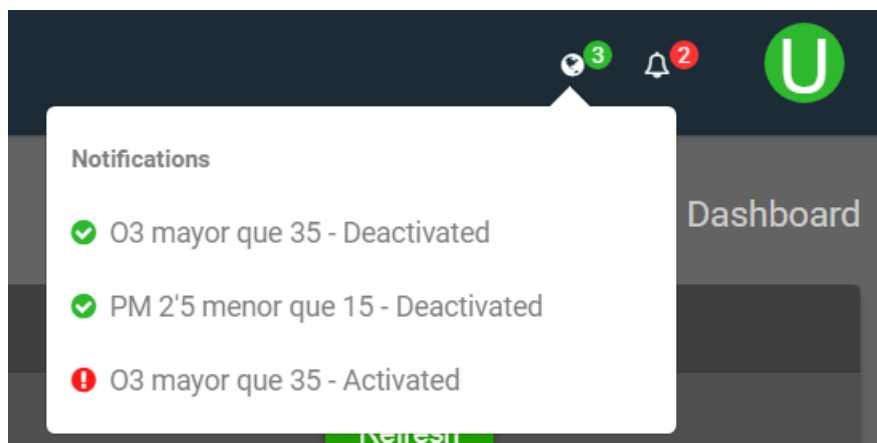
Justo al lado del calendario se encuentran las categorías asociadas al usuario. Este puede seleccionar la que quiera entre las que tenga asignadas. Una vez se esté satisfecho con los

parámetros seleccionados, se debe pulsar el botón *Refresh* para que la aplicación refresque los datos con la información que se ha introducido.



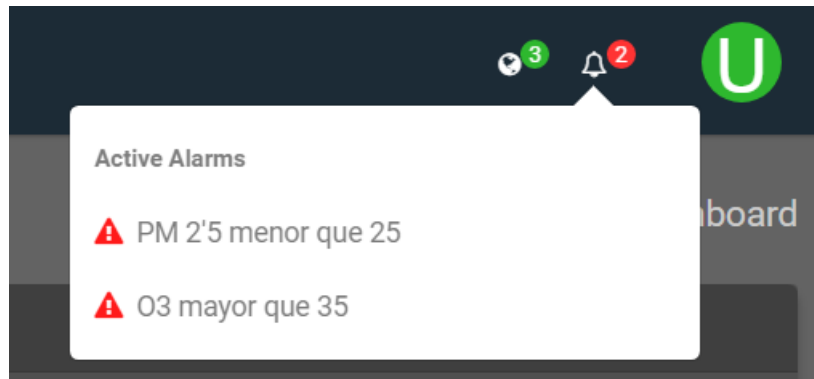
40. Calendario

El icono de historial de notificaciones recibe una notificación por cada alarma que ha sido activada y desactivada.



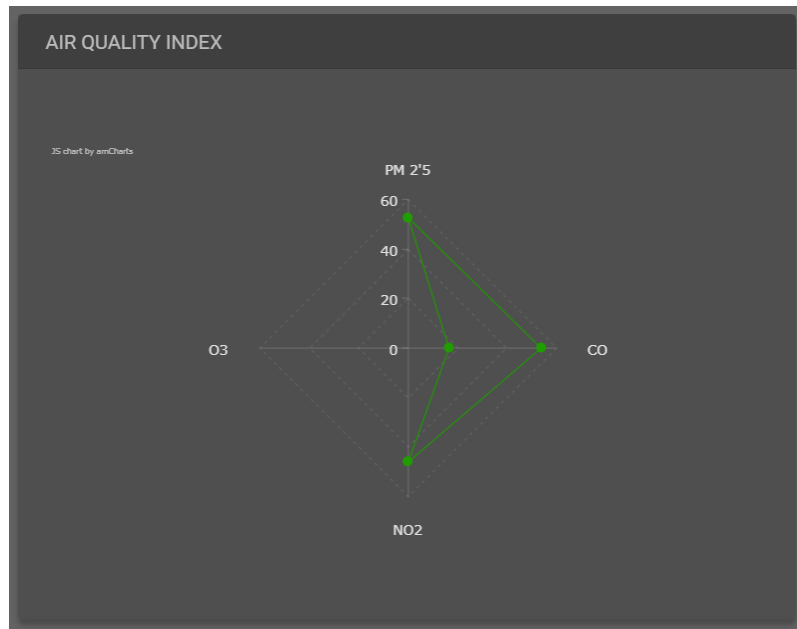
41. Notificaciones de cambio de estado de alarmas

El icono de notificaciones sobre alarmas activas recibe una notificación cada alarma cuyo estado actual se activado. En caso de que la alarma se desactive durante el transcurso de la navegación, esta es eliminada de la lista de notificaciones.



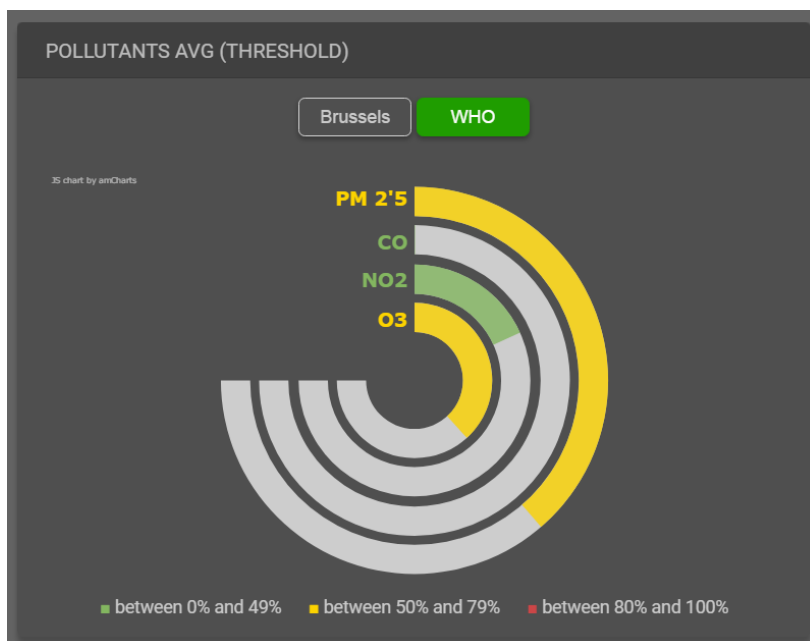
42. Notificaciones de alarmas activas

El primer gráfico que se observa es el referente a la calidad del aire. En él se realiza un cálculo en base a cada contaminante para obtener el ICA correspondiente. La información se representa sobre un gráfico de araña en el que el usuario puede pasar por encima el ratón, o pulsar en caso de ser un *smartphone*, y obtener el resultado del cálculo para cada contaminante.



43. Gráfico de calidad del aire

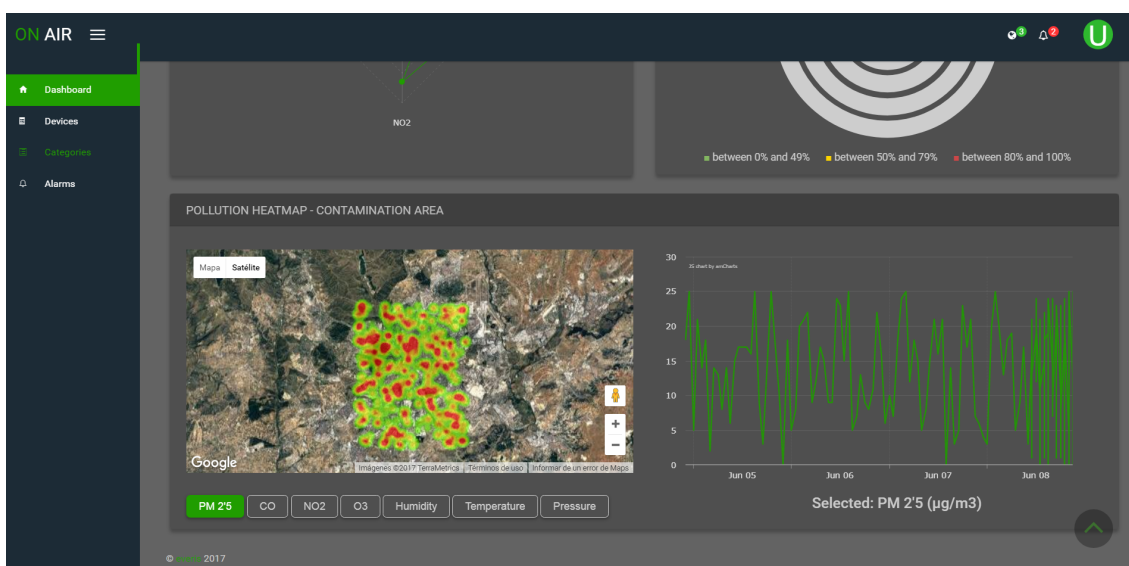
El siguiente gráfico se encarga de realizar un cálculo de la media de cada contaminante. El resultado de cada uno es comparado con el umbral máximo correspondiente a cada contaminante, en base a los cuales, cambia el color de los valores dependiendo de si se acercan o sobrepasan el umbral correspondiente. Para este cálculo, se puede elegir entre los umbrales establecidos por Bruselas o los registrados por la OMS, mediante los cuales el resultado de la gráfica varía.



44. Gráfico de la media de los contaminantes OMS

En la parte inferior del *Dashboard* se encuentra el último de los elementos mostrados en esta ventana. En ella se visualizan dos elementos clave: un mapa de calor y una gráfica lineal.

Sobre el mapa de calor se representan los sensores en la posición geográfica en la que se encuentran. Cada punto indica la intensidad del valor recibido por dicho sensor, siendo verde un valor bajo, amarillo uno mediano y rojo el más alto. La gráfica lineal muestra el valor recibido por cada sensor en el tiempo. La información representada en ambos componentes viene dada por el parámetro seleccionado mediante los botones que se encuentran en la parte inferior del mapa e indican los elementos recibidos por el sensor.



45. Dashboard inferior

En la parte inferior derecha se puede observar una flecha que al pulsarla posiciona al usuario sobre la parte superior de la ventana.

Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua



46. Mapa de calor y gráfica temporal de la temperatura

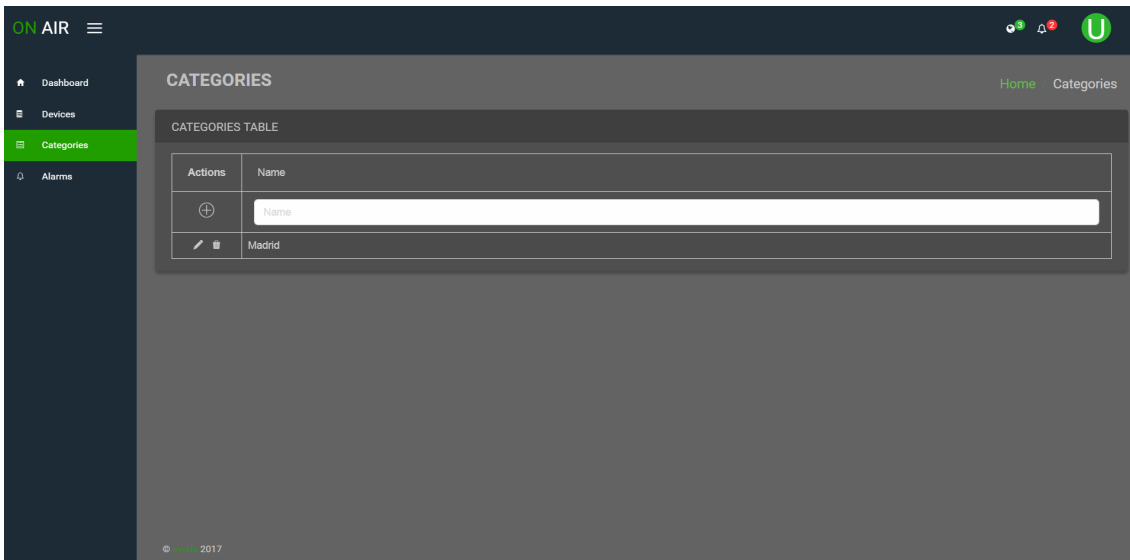
Mediante el menú lateral se puede navegar hasta la ventana de *Devices*, donde se muestra una tabla con los dispositivos asociados al usuario. Es posible realizar acciones de creación, filtrado, actualización y borrado desde la misma tabla.

The screenshot shows the 'ON AIR' interface with a sidebar menu containing 'Dashboard', 'Devices', 'Categories', and 'Alarms'. The 'DEVICES' section is active, displaying a 'DEVICES TABLE' with the following data:

Actions	Name	Device ID	Category ID	Device Key	Connection String
<input type="button" value="⊕"/>	<input type="text" value="Name"/>	<input type="text" value="Device ID"/>	<input type="button" value="Select a Category"/>	<input type="text" value="Device Key"/>	<input type="text" value="Connection String"/>
<input type="button" value="✎"/>	libellum	libellum	5936d57c734d1d2b1e933198	uHkXFa1pxdGv9DhZjh9qPlxzs8xjLT+NdrDBHK04=	HostName=jco-fothub.azure-devices.net;DeviceId=libellum;SharedAccessKey=
<input type="button" value="✎"/>	PruebaARE	pruebaare	5936d57c734d1d2b1e933198	YaH250alRyTwTPFR503MeNH80vWx0MbtfrPYe+FYA=	HostName=jco-fothub.azure-devices.net;DeviceId=pruebaare;SharedAccessKe

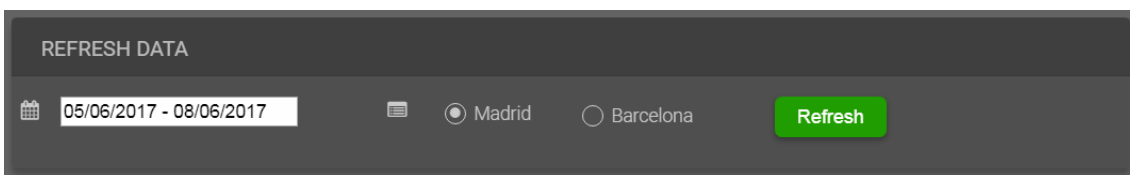
47. Tabla de dispositivos

Del mismo modo que con los dispositivos, la sección de *Categories* ofrecen la opción de crear, filtrar, actualizar y borrar categorías.



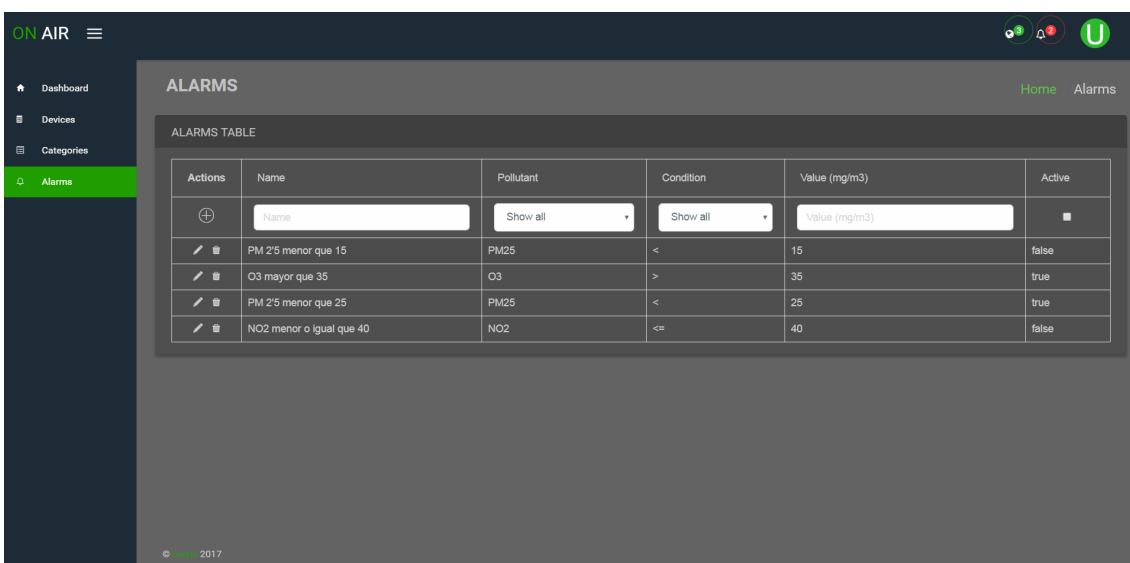
48. Tabla de categorías

Las categorías que se creen o modifiquen en esta ventana aparecen en el componente *Refresh Data* del *Dashboard*, donde el usuario puede seleccionarlas a la hora de consultar datos. En este ejemplo se ha añadido la categoría *Barcelona* que aparecerá en la ventana de *Dashboard* junto a *Madrid*.



49. Categoría añadida a las opciones del Dashboard

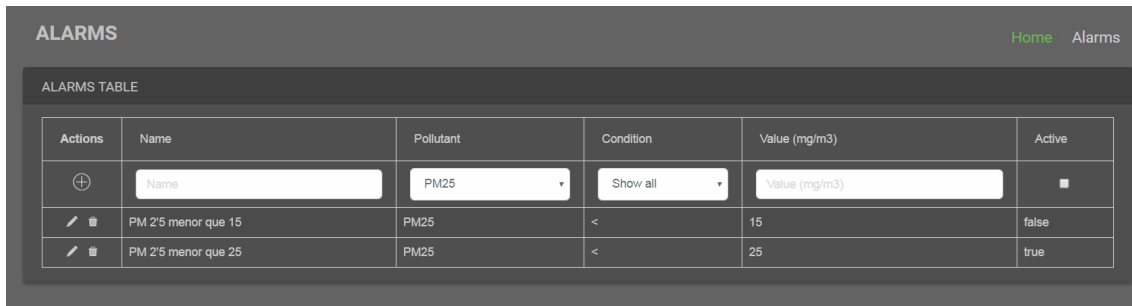
La última venta, *Alarms*, permite realizar las mismas acciones que las anteriores ventanas de *Devices* y *Categories*. Las alarmas creadas pasan de forma automática a ser comprobadas por la aplicación, encargada de registrar cada cambio de estado de estas y notificarlo, de igual modo que mostrar aquellas que se encuentren activas.



50. Tabla de alarmas

Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Mediante los campos de texto superiores es posible el filtrado de la información disponible en la tabla. En este caso se ha filtrado por PM 2'5 (partículas en suspensión de menos de 2,5 micras), uno de los contaminantes registrados por el dispositivo.

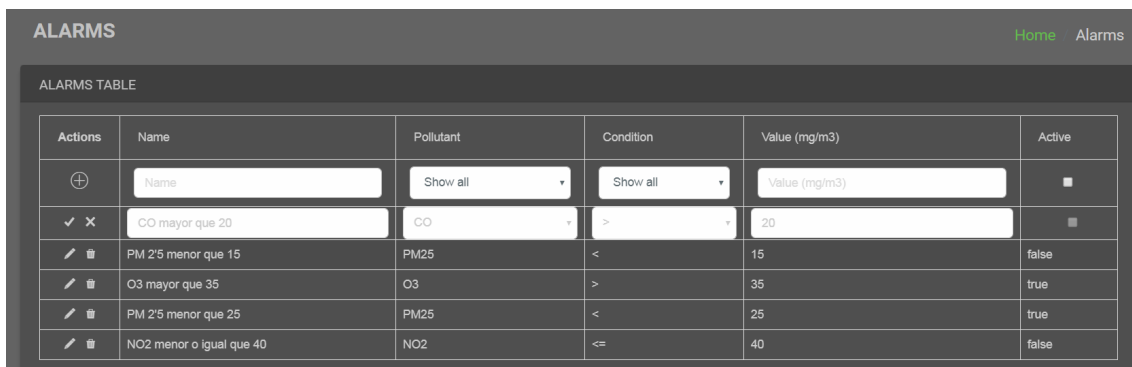


The screenshot shows the 'ALARMS' section of a web application. At the top right, there are links for 'Home' and 'Alarms'. Below the title is the 'ALARMS TABLE' header. The table has six columns: 'Actions', 'Name', 'Pollutant', 'Condition', 'Value (mg/m3)', and 'Active'. The first row is a header row with input fields for filtering: a plus icon, a 'Name' text box, a 'Pollutant' dropdown menu set to 'PM25', a 'Condition' dropdown menu set to 'Show all', a 'Value (mg/m3)' text box, and an 'Active' checkbox. Below this are two rows of data for PM2.5 alarms: one with a value of 15 and 'Active' set to 'false', and another with a value of 25 and 'Active' set to 'true'. Each data row has edit and delete icons in the 'Actions' column.

Actions	Name	Pollutant	Condition	Value (mg/m3)	Active
+	<input type="text" value="Name"/>	PM25	Show all	<input type="text" value="Value (mg/m3)"/>	<input type="checkbox"/>
	PM 2'5 menor que 15	PM25	<	15	false
	PM 2'5 menor que 25	PM25	<	25	true

51. Filtrado de alarmas con contaminante PM 2'5

De igual manera, pulsado el icono con el símbolo más y rellenando los campos es posible crear una nueva alarma que pasa a ser vigilada por la aplicación. En este caso se crea una alarma para controlar que avise cuando los niveles de CO superen los 20 mg/m³.

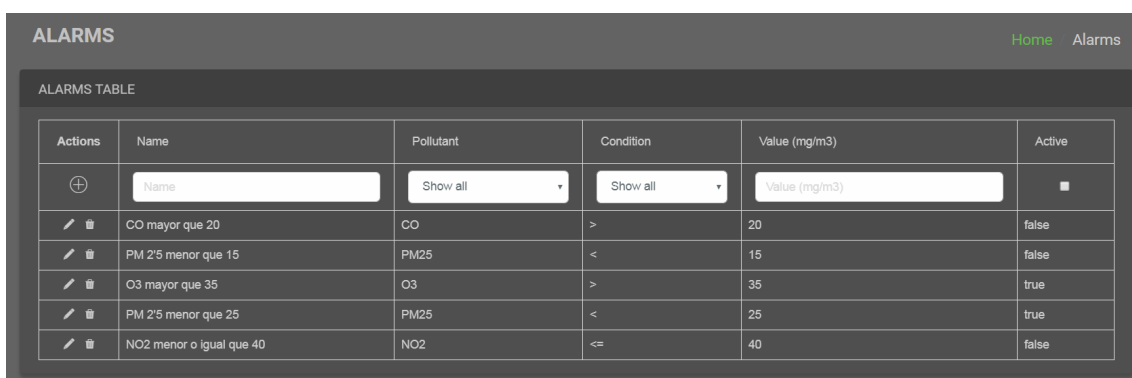


The screenshot shows the 'ALARMS' section with the 'ALARMS TABLE' header. The table has six columns: 'Actions', 'Name', 'Pollutant', 'Condition', 'Value (mg/m3)', and 'Active'. The first row is a header row with input fields for creating a new alarm: a plus icon, a 'Name' text box, a 'Pollutant' dropdown menu set to 'Show all', a 'Condition' dropdown menu set to 'Show all', a 'Value (mg/m3)' text box, and an 'Active' checkbox. Below this are five rows of data for existing alarms: CO mayor que 20 (Active: false), PM 2'5 menor que 15 (Active: false), O3 mayor que 35 (Active: true), PM 2'5 menor que 25 (Active: true), and NO2 menor o igual que 40 (Active: false). Each data row has edit and delete icons in the 'Actions' column.

Actions	Name	Pollutant	Condition	Value (mg/m3)	Active
+	<input type="text" value="Name"/>	Show all	Show all	<input type="text" value="Value (mg/m3)"/>	<input type="checkbox"/>
	CO mayor que 20	CO	>	20	false
	PM 2'5 menor que 15	PM25	<	15	false
	O3 mayor que 35	O3	>	35	true
	PM 2'5 menor que 25	PM25	<	25	true
	NO2 menor o igual que 40	NO2	<=	40	false

52. Creación de una nueva alarma

Las primera vez que se crea una alarma esta se encuentra desactivada.

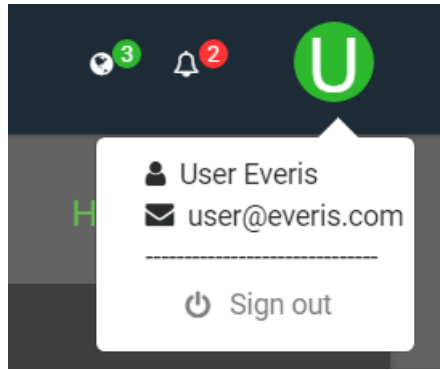


The screenshot shows the 'ALARMS' section with the 'ALARMS TABLE' header. The table has six columns: 'Actions', 'Name', 'Pollutant', 'Condition', 'Value (mg/m3)', and 'Active'. The first row is a header row with input fields for creating a new alarm: a plus icon, a 'Name' text box, a 'Pollutant' dropdown menu set to 'Show all', a 'Condition' dropdown menu set to 'Show all', a 'Value (mg/m3)' text box, and an 'Active' checkbox. Below this are five rows of data for existing alarms: CO mayor que 20 (Active: false), PM 2'5 menor que 15 (Active: false), O3 mayor que 35 (Active: true), PM 2'5 menor que 25 (Active: true), and NO2 menor o igual que 40 (Active: false). Each data row has edit and delete icons in the 'Actions' column.

Actions	Name	Pollutant	Condition	Value (mg/m3)	Active
+	<input type="text" value="Name"/>	Show all	Show all	<input type="text" value="Value (mg/m3)"/>	<input type="checkbox"/>
	CO mayor que 20	CO	>	20	false
	PM 2'5 menor que 15	PM25	<	15	false
	O3 mayor que 35	O3	>	35	true
	PM 2'5 menor que 25	PM25	<	25	true
	NO2 menor o igual que 40	NO2	<=	40	false

53. Tabla con la nueva alarma creada

Para finalizar, cuando se desee abandonar la sesión, se deberá pulsar el icono superior derecho que contiene la primera letra de nuestro nombre. Al hacerlo se despliega un menú que permite ver el nombre de usuario y correo con el que se ha realizado el inicio de sesión, así como la opción de *Sign out* con la que volver a la ventana de *Login*.



54. Sign out

8. Integración continua

El capítulo ocho realiza un estudio sobre la práctica de integración continua para proyectos de desarrollo *software*, presentando una descripción de las principales prácticas la caracterizan. Después, introducen dos de las principales herramientas de integración continua actualmente, describiendo los aspectos de cada una y estableciendo una pequeña comparación. A continuación, se argumenta la herramienta elegida además del *software* extra necesario para completar el proceso. Para finalizar, se explica paso a paso como montar el entorno mediante la conexión del repositorio de trabajo con la herramienta de integración continua.

8.1 Introducción

Cuando se habla de integración continua, se hace referencia a la práctica empleada para el desarrollo *software* donde los colaboradores de cada proyecto incorporan su trabajo de forma frecuente con el del equipo en conjunto. Dichas integraciones se realizan de manera diaria por cada uno de los miembros equipo, dando lugar a múltiples actualizaciones en el código en común del software en cuestión. Cada uno de estas piezas que se suman al total de la aplicación o sistema, es sometida a test que se despliegan de forma automática con la finalidad de detectar y evitar los posibles errores entre la parte existente y aquella que se haya incluido recientemente. Debido a esto, se consigue un proceso de depuración de *software* mucho más rápido y eficaz, permitiendo a los desarrolladores focalizarse en el desarrollo sin tener que preocuparse.

El hecho de no realizar un proceso de integración continua durante el tiempo de desarrollo para un producto *software* implica que, una vez finalizadas todas las partes, la tarea de juntarlas en una única se convierte en un proceso tedioso. La aparición de múltiples errores está garantizada, ya que nunca antes todas las piezas han estado juntas y la identificación y solución de dichos errores puede ser un quebradero de cabeza porque estos se pueden dar en implementaciones implementadas en fases muy tempranas del proyecto, obligando a los miembros del proyecto a dedicar tiempo en hacer memoria sobre dicha parte.

Este tratamiento del proceso de integración continua, no resulta muy eficaz y puede disuadir a muchos desarrolladores y empresas de considerarlo en sus proyectos. Como describe (Fowler, 2006) es necesario considerar la integración continua como un proceso *non-event*, es decir, un proceso que no tiene una fase concreta del proyecto dedicada a ello sino que se lleva a cabo con total normalidad en el día a día. Adoptando este pensamiento, es posible incluir y depurar código de forma casi diaria aligerando el proceso de despliegue.

La gran evolución entre una forma de ver la integración continua y la otra no reside en el desembolso de grandes cantidades de dinero para el uso de herramientas que propicien el cambio a mejor (Fowler, 2006). Se trata única y exclusivamente de la adopción y ejecución de las prácticas correctas por parte de todos los miembros de un mismo equipo.

Es importante que los fallos de integración no persistan por mucho tiempo una vez detectados. De este modo, siempre será posible contar con una versión funcional del *software* y que esté a disposición de todos los integrantes del proyecto para su utilización.

8.2 Prácticas

En todo proceso, bien sea *software* o no, es necesario la aplicación de unas prácticas o pautas que indiquen su correcto desarrollo e implementación. La integración continua no es una excepción, ya que requiere de un adecuado conocimiento de estas para que el resultado final sea satisfactorio.

- **Mantenimiento de un repositorio principal único:** normalmente, el desarrollo de aplicaciones *software* constituye un amplio manejo de ficheros, a lo que se le añade la complicación de tener a distintas personas trabajando a la vez sobre el mismo código. Para evitar las colisiones entre los programadores, es necesario tener definida una estructura clara que permita el manejo adecuado de todas las versiones producidas de un mismo proyecto. Para ello, existen las herramientas de nominadas como controladores de versiones (*Git*, *Subversion*, etc.) encargadas de la organización de ficheros y versionados de estos.

Es imprescindible contar con un buen controlador de versiones, al alcance de todos, si se pretende aplicar al cien por cien la integración continua. Los miembros de un mismo equipo son responsables de gestionar e incluir en el repositorio todos los cambios realizados en sus versiones.

- **Automatizar el desarrollo:** como se ha mencionado anteriormente, el proceso de integración continua puede ser realizado paso a paso de forma totalmente manual, sin embargo al tratarse de acciones repetitivas que se ejecutan de forma periódica es posible hablar de una automatización que se encargue de realizarlo, agilizando el proceso de desarrollo.

En la automatización, es necesario incluir todas las partes que conforman el proyecto en sí y permiten que sea ejecutado de manera exitosa, en otros términos, se requiere tener una aplicación lista para ser ejecutada desde cualquier máquina. No obstante, es importante tener presente que no todas las automatizaciones van a requerir las mismas pautas a la hora de ser desplegadas, siendo necesario tener en consideración el tipo de *software* sobre el que se está aplicando.

- **Automatización de test para el desarrollo:** como se especifica en el punto anterior, la automatización tiene que dar como resultado final un código capaz de ser ejecutado desde cualquier equipo. Para que al final de este proceso se obtenga un *software* capaz de funcionar en distintas máquinas, es necesario la aplicación de test que corroboren el funcionamiento correcto de dicha versión. Para ello, es requerido el



Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

despliegue de test cada vez que se quiera hacer una nueva versión del sistema entero, siendo posible su automatización.

- **Subidas de código a la rama principal de forma diaria:** es importante recordar que el éxito de una buena integración continua no viene dado únicamente por la utilización de herramientas que faciliten su implementación, sino por la colaboración y compromiso de los participantes en el proyecto. El hecho de incluir el código individual de cada programador al principal, de forma activa y diaria, permite la aparición, identificación y resolución de choques entre desarrollos de los diferentes miembros.
- **Arreglar versiones con errores:** en el caso en el que se detecte algún error al integrar nuevo código, estos deben solucionarse lo más rápido posible.
- **Testear en un entorno como el de producción:** para que se pueda saber realmente el comportamiento deseado del sistema, es indispensable que las pruebas se realicen sobre un entorno lo más semejante al de producción o una copia de este mismo en la medida de lo posible. Solo de ese modo es posible confirmar con certeza el correcto funcionamiento debido a que otros entornos simulados podrían causar resultados que, a primera vista, parecieran los deseados pero que no se diesen en el lugar final para el que está diseñado. No obstante, no siempre es permisible el contar con un marco como el de producción ya que habrá sistemas para los que resulte muy costoso o complicado. Sabiendo esto, la meta siempre es el intentar realizar pruebas en el ámbito más cercano a la realidad de la aplicación.
- **Despliegue automático:** la simple adopción de integración continua lleva implícito la realización de tareas que se repiten en el tiempo con una frecuencia considerable. Cada vez que se hace una nueva incorporación de código al conjunto general, la aplicación debe compilarse, ejecutar los test y pruebas necesarias, y ser desplegada. Valorando que en un equipo formado por diversos desarrolladores realiza múltiples subidas de código cada día, se observa claramente la necesidad de una automatización de test que permita trabajar de forma más rápida y eficaz.

Los despliegues en producción no suelen realizarse con una frecuencia tan elevada como las integraciones diarias en desarrollo, no obstante, su automatización resulta una pieza clave para la calidad del producto. Del mismo modo, Fowler (2006) aconseja automatizar un proceso que permita volver al último estado exitoso en caso de que el despliegue en producción desemboque en algún error fatal. De esta manera, la tarea de despliegue se abstrae del esfuerzo requerido para llevarla a cabo.

A primera vista puede parecer que la integración continua ofrece muchas ventajas, permitiendo ahorrar tiempo mediante la automatización de procesos y acciones que de otra forma se realizan manualmente. Pese a que hay muchos conceptos e ideas nuevas uno puede ser propenso a pensar que la mayor desventaja reside en la curva de aprendizaje de herramientas de control de versiones y todos los términos que las rodean, aunque el principal desafío a afrontar es la implicación que los miembros del equipo pongan en seguir las pautas

necesarias. Los pilares que sostienen el éxito en el empleo de integración continua son aquellos que pretendan hacer uso de ella, ya que si una de las piezas falla, el esfuerzo habrá sido en vano. Debe haber un alto nivel de comunicación entre los miembros de un equipo para que todos los engranajes funcionen porque el fallo de una de ellas puede desencadenar desenlaces no deseados. De la misma manera, no tiene sentido embarcarse en la integración continua si no hay ningún interés en desarrollar test que pongan a prueba el sistema, dando como resultado una pérdida de tiempo y esfuerzo para no conseguir ningún avance o mejora.

8.3 Herramientas

La aplicación de integración continua a cualquier desarrollo *software* requiere de un ochenta por ciento, que abarca los procesos y actitudes que debe adoptar el equipo frente a un veinte por ciento que representa las tecnologías y herramientas *software* necesarias para llevar a cabo dicho proceso (Stafford, n.d.). Sin embargo, el hecho de que el triunfo de la integración continua esté basado mayormente en el compromiso de los desarrolladores respecto a este, no implica que el uso de aplicaciones para su control y automatización no tenga la menor importancia.

Actualmente existen herramientas como *Jenkins* o *Bamboo* que permiten la configuración y mecanización de despliegues, facilitando la vida a los desarrolladores.

8.3.1 Jenkins

Jenkins está catalogado como una herramienta *open source* para la implementación continua en cualquier tipo de proyecto *software*, bien sea mediante la ejecución de comandos en un terminal o haciendo uso de su aplicación web (Kunja, 2017). Además, también es posible encontrarlo definido como servidor de desarrollo en *Java*.



55. Jenkins

La labor principal de *Jenkins* es el facilitar la construcción de versiones, así como su posterior despliegue en los distintos entornos junto a la automatización de test. Su trabajo no es más que el de un intermediario entre el directorio o repositorio en el que se encuentra el código y el servidor en el que se desea desplegar. Cada vez que se detecta un nuevo cambio, *Jenkins* se encarga de lanzar un evento en el que se ejecutan los test sobre la nueva versión y su subsiguiente despliegue en el entorno configurado, ya sea para la parte de desarrollo, preproducción o producción.

Su carácter *open source* es decir, *software* desarrollado por diversas personas y con una política de uso y compartición gratuita, hacen de *Jenkins* una opción ideal para aquellos proyectos u organizaciones que no dispongan de un alto presupuesto para destinar a la implementación de integración continua pero quieran adentrarse en esta práctica para asegurar la calidad de sus productos. Este aspecto, sumado a su fácil instalación y el hecho de estar desarrollado en *Java*, lo convierten en el candidato perfecto para adaptarse a todo tipo de proyectos y plataformas.

Gracias a características como las comentadas en el anterior párrafo, *Jenkins* ha incrementado su popularidad dentro del mundo de los servidores de integración continua, permitiendo así, el surgimiento de una comunidad enorme y comprometida con su mantenimiento. Es cierto que existen muchos servidores de integración continua, pero pocos o ninguno cuenta con una comunidad tan extensa y entregada, facilitando de este modo, la resolución de dudas o problemas que aparezcan al elegir *Jenkins* para el desarrollo de un proyecto. Esto ha permitido crear un ecosistema capaz de albergar una de las mayores bibliotecas de *plugins* en lo que a este tipo de herramientas se refiere, ofreciendo al mismo tiempo facilidad absoluta para su creación y distribución (Inman, 2011). La vasta documentación disponible en la red favorece la continuidad de aquellos que se aventuren a utilizarlo por primera vez.

Indiscutiblemente, *Jenkins* es el rey de los *plugins*, brillando gracias a su comprometida comunidad. La variedad de ellos se mueve entre tres pilares principales: dirección, organización e informes. Actualmente, *Jenkins* cuenta con más de mil quinientos *plugins* acorde con la información dispuesta en su web *Jenkins Wiki* (<https://wiki.jenkins.io/display/JENKINS>).

Como ejemplo, Grebenets (2015) propone echar un vistazo al que para él es una de las categorías de *plugin* más importantes dentro de la integración continua, la de la información. En este caso, *Jenkins* cuenta con alrededor de ciento treinta opciones disponibles para suplir dicha necesidad, sin embargo, el tener tal cantidad y variedad puede convertirse en un arma de doble filo. No todos los *plugin* que se encuentren serán igual de útiles o tendrán las mismas opciones por lo que el elegir aquel que cumpla con nuestros requisitos consistirá en un proceso de prueba y error.

Otro concepto importante a es el de como configurar aquello que se quiere que *Jenkins* ejecute de forma automática con la introducción de cada cambio realizado. Para ello, *Jenkins* hace uso de las *pipelines*, a pesar de no ofrecerlo por defecto. Estas *pipelines* no son más que un fichero de código en el que se especifica las tareas o *jobs* que se deseen automatizar. Estas *pipelines* se estructuran en pasos, denominados *stages*, en los que se puede especificar desde la ejecución de test hasta el la ejecución completa de la aplicación.

8.3.2 Bamboo

Bamboo, igual que *Jenkins*, también es un servidor de integración continua. Su objetivo no es otro que el de proveer a los desarrolladores de un entorno de compilación y aplicación de test dinámicos focalizándose en una alta subidas a producción y entregas (Curran, 2017). Como

medida principal, *Bamboo* se encarga de establecer un registro exhaustivo de todos y cada uno de los eventos que suceden desde el momento en el que comienza el desarrollo e implementación de una funcionalidad hasta que es entregada al cliente.

Forma parte de la familia de productos *Atlassian*, entre los que se encuentra la herramienta de control de incidencias y gestor de aplicaciones *software*, *JIRA*. Este aspecto es de gran interés ya que dentro de Everis el uso de *JIRA* está bastante extendido por lo que su configuración con *Bamboo*, si así se desea, resulta simple.



56. Bamboo

A diferencia de *Jenkins*, *Bamboo* cuenta con una librería de *plugins* mucho más reducida, poco más de doscientos (<https://marketplace.atlassian.com/addons/app/bamboo>). A primera vista puede dar la sensación de que *Bamboo* carece de funcionalidades respecto a *Jenkins*, pero es necesario recordar que no todos los *plugins* de este último ofrecen la misma calidad y puede existir más de uno que solventa una misma funcionalidad.

Aspecto	<i>Jenkins</i>	<i>Bamboo</i>
Modelo de negocio	<i>Open source</i>	Licencia de pago
Funcionalidad	Basado en <i>pipelines</i>	Basado en <i>stages</i> y <i>pipelines</i>
Configuración	Sencilla. Implementado en <i>Java</i>	Sencilla. Implementado en <i>Java</i>
Usabilidad	UI por defecto bastante simple. Requiere de <i>plugins</i> para actualizarla	UI personalizable por defecto
Software externo	Permite la integración de <i>software</i> de terceros mediante <i>plugins</i>	Permite la integración de <i>software</i> de terceros mediante <i>plugins</i>
Control de ramas	Requiere de <i>plugins</i> y <i>scripts</i>	Sistema de auto detección de ramas
<i>Pipelines</i>	No existe soporte por defecto. Requiere de <i>plugins</i>	Simple configuración de <i>pipelines</i> por defecto
<i>Atlassian</i>	Requiere de <i>plugins</i> . Información limitada	Integración nativa con productos <i>Atlassian</i>

26. Jenkins vs Bamboo



8.4 Tecnologías

Para la implementación final se ha decidido utilizar *Jenkins* como herramienta de integración continua junto al controlador de versiones *Git* mediante *Bitbucket*, como se especificó anteriormente en este documento. Las razones principales por las que se ha optado por elegir *Jenkins* frente a *Bamboo* son las siguientes:

- **Open source:** al tratarse de una herramienta *open source*, su uso es completamente gratuito para cualquier usuario, convirtiéndolo en un *software* perfecto para la experimentación. Al tratarse este proyecto de una prueba de concepto para el futuro cambio de rumbo de Everis, la licencia de *Bamboo* resulta demasiado cara.
- **Mejoras en *Jenkins*:** desde que este fue lanzado, ha experimentado muchas mejoras por parte de la comunidad que han ido convirtiéndolo en una herramienta cada vez más atractiva para los desarrolladores. La reciente introducción de una nueva interfaz, motiva y facilita su uso y visualización de los acontecimientos dados durante el proyecto.
- **Compatibilidad con *Git* y *Bitbucket*:** como se describe en anteriores puntos el controlador de versiones para esta prueba de concepto es *Git* junto a *Bitbucket*. Gracias a la extensa biblioteca ofrecida por *Jenkins*, existen *plugins* que ofrecen una perfecta sincronización entre ambas herramientas.

Eligiendo estas como las razones de más peso a la hora de determinar el servidor de control de versiones, ya es posible comenzar a montar el entorno para su ejecución. La prueba de concepto está realizada sobre el sistema operativo *Windows 7* así que, antes de continuar, es necesaria la instalación de dos aplicaciones que actúan como nexo de unión entre *Jenkins* y *Bitbucket*: *Docker ToolBox* y *Ngrok*.

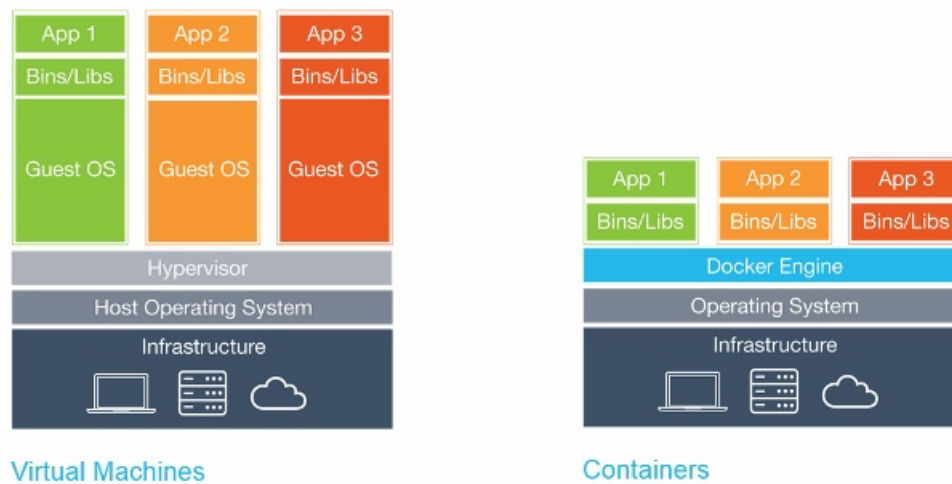
El primer paso consiste en la instalación de *Docker*. Consiste en una plataforma *open source* para el desarrollo *software* cuya funcionalidad apunta al empaquetado de aplicaciones en *containers* capaces de ser ejecutadas en cualquier sistema *Linux*. No obstante, también es posible su ejecución en sistemas *Windows* mediante máquina virtual. Acorde con Vaughan-Nichols (2017) la popularidad de *Docker* ha ido en constante aumento desde el momento de su salida.



57. Docker

Docker trabaja mediante la creación de contenedores, cuya representación no es más que la virtualización del sistema operativo dando lugar a bloques capaces de ser ejecutados con la

configuración *software* establecida. Se diferencia así de las máquinas virtuales en que estas separan el *hardware* en diferentes VMs de manera que su poder queda repartido entre los diferentes usuarios o máquinas, mientras que *Docker* separa el *software* en contenedores que son ejecutados sobre una única máquina (figura 58), ofreciendo una amplia ayuda a la hora de gestionar recursos. Además, *Docker* ha conseguido una mayor estandarización en el mundo de los contenedores, permitiendo un despliegue más eficiente, proporcionando la opción de empaquetar, mandar y ejecutar aplicaciones de forma ligera, portable y autosuficiente.

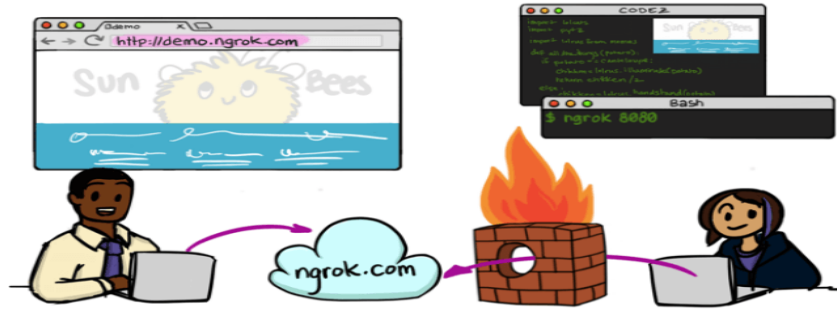


58. Máquinas virtuales vs contenedores

Entre otras ventajas se encuentra el control total del entorno de ejecución para cada aplicación, teniendo la disponibilidad para ejecutar test sobre un contenedor que replique las condiciones existentes en el entorno de producción. De este modo, los resultados de las pruebas permiten dotar al desarrollo de una mayor fiabilidad, sin embargo, existen aspectos a tener en cuenta si se opta por este *software*.

El uso de *Docker* implica la inclusión de una capa más dentro del desarrollo (Hauer, 2015) añadiendo algo más de complejidad. Es necesario prestar especial atención en el manejo de contenedores si la creación de estos crece de forma considerable ya que, al compartir un mismo *kernel*, un fallo puede causar la desconexión de todos ellos.

El segundo programa a instalar, *Ngrok*, es un *tunelador* multiplataforma *revertidor* de *software proxy* encargado de proveer canales seguros desde puntos públicos como lo puede ser *Internet*, a redes de servicio locales. Al mismo tiempo, *Ngrok* se encarga de capturar el tráfico de dicho túnel de comunicación, ofreciendo la posibilidad de observar todo lo que pasa por él (Kajmohideen, 2015).



59. Representación del funcionamiento de Ngrok

8.5 Funcionamiento

A continuación se detallan los pasos a seguir para la puesta en marcha de la integración continua:

- Obtener una imagen de *Docker* que contenga aquello que necesitemos para ejecutar nuestra aplicación. En el caso de este proyecto, se necesita *Jenkins*, para el control de versiones y *NPM*, *Angular-Cli*, *Typescript*, y *NodeJS* para la ejecución de mi aplicación web.
- Abrir la consola de *Docker* y ejecutar la imagen que contenga *Jenkins*. Una vez hecho esto es necesario obtener la *IP* de la máquina *Docker*.
- Abrir un navegador y escribir como *URL* la *IP* de la máquina *Docker* y el puerto 8080 para acceder a *Jenkins*. Ejemplo: `http://192.168.99.100:8080`
- Una vez dentro, instalar los *plugin Blu Ocean*, para la nueva interfaz y *Bitbucket Branch Source Plugin*, para la conexión con los repositorios en *Bitbucket*.
- Crear dos nuevas tareas de tipo *Bitbucket Team/Project*, una para el repositorio que contiene el código del *front-end* y otra para el código del *back-end*. Configurarlas y comprobar que se conecta exitosamente con cada repositorio.
- Ejecutar *Ngrok* y escribir el siguiente comando con la *IP* de la máquina de *Docker* y el puerto: `ngrok http 192.168.99.100:8080`. Copiar la *URL* generada y guardarla en la configuración web de *Jenkins* como su dirección web. Establecerla también como *webhook* para ambos repositorios en *Bitbucket*.
- Los repositorios ya están enlazados con *Jenkins*, tan solo queda crear un fichero *Jenkinsfile* en el *root* de cada repositorio que contenga una *pipeline* con las

funciones que se deseen llevar a cabo. En el caso descrito en este documento, tanto la *pipeline* del *front-end* como la del *back-end* se encargan de ejecutar sus respectivos test, configurados en el apartado de *Test* de este documento.

Al llegar a este punto el sistema ya está en marcha. Cada vez que se actualice alguno de los repositorios mediante *Git*, *Jenkins* se encargará de ejecutar los test de forma automática y comunicarnos su resultado.

9. Estructura de trabajo

Con el fin de afrontar todos los objetivos propuestos y conseguir unos resultados exitosos, se ha decidido plantearme un modelo de trabajo distribuido en *sprints* o iteraciones basado en el formato de desarrollo ofrecido por el *framework* para metodologías ágiles llamado SCRUM. La finalidad de esta estructuración de trabajo consiste en el desglose de los objetivos principales, referentes a este proyecto, en partes que compartan una misma temática. La meta al final de cada *sprint* es el completar los objetivos propuestos y, como resultado, obtener una parte perfectamente funcional.

La elección de del número de *sprints* corresponde a las fases bien definidas dentro del proyecto: desarrollo de la web piloto, desarrollo test e integración continua. A continuación se presenta la estructura final del desarrollo formado por cuatro *sprints*:

Sprint 1	Prueba piloto	
Descripción	Implementación de una aplicación web básica que haga uso de todas las tecnologías presentadas para la nueva arquitectura.	
Duración	Comienzo	22 de marzo de 2017
	Fin	21 de abril de 2017
Objetivos	1	Desarrollo <i>front-end</i> : <i>Dashboard</i>
	2	Desarrollo <i>back-end</i>
	3	Desarrollo base de datos
	4	Primera iteración de la memoria
Resultado	Aplicación web piloto funcional de principio a fin	

27. Sprint 1

Sprint 2	Desarrollo de test	
Descripción	Implementación de test unitarios y de integración para la aplicación web. Introducción de nuevas funcionalidades.	
Duración	Comienzo	22 de abril de 2017

	Fin	19 de mayo de 2017
Objetivos	1	Implementación de test unitarios <i>front-end</i> con <i>Karma</i> y <i>Jasmine</i> .
	2	Implementación de test unitarios <i>back-end</i> con <i>Mocha</i> y <i>Chai</i>
	3	Implementación de test end-to-end con Protractor
	4	Implementación de nuevas funcionalidades: <i>Devices</i> , <i>Categories</i> y <i>Alarms</i>
	5	Segunda iteración de la memoria
Resultados	Aplicación web con los test desarrollados y aplicados	

28. Sprint 2

Sprint 3	Integración continua	
Descripción	Configuración del entorno de integración continua para la automatización de test sobre la aplicación web	
Duración	Comienzo	20 de mayo de 2017
	Fin	16 de junio de 2017
Objetivos	1	Configuración de <i>Jenkins</i> como servidor de integración continua
	2	Configuración de <i>Git</i> con <i>Bitbucket</i>
	3	Configuración <i>Docker</i>
	4	Tercera iteración de la memoria
Resultado	Automatización de test y despliegue	

29. Sprint 3

Sprint 4	Finalización memoria y preparación de la presentación	
Descripción	Integración de todas las partes de la memoria y refinamiento. Preparación de las diapositivas para la presentación del trabajo de fin de grado	



Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Duración	Comienzo	17 de junio de 2017
	Fin	5 de julio de 2017
Objetivos	1	Finalización de la memoria
	2	Diapositivas presentación
Resultado	Memoria completa y presentación preparada	

30. Sprint 4

Esta estructura de trabajo permite tener una visualización clara de todas las partes que conforman el proyecto, al mismo tiempo que ayuda a la hora de controlar el tiempo dedicado a cada objetivo.

10. Conclusiones

En resumen, he explicado la implementación de esta nueva filosofía de desarrollo *software* para la empresa Everis, desde las tecnologías, pasando por la nueva arquitectura con sus test, hasta el proceso de automatización de estos y los despliegues con ayuda de *Jenkins* y *Git*. Cuando uno pone todo esto encima de la mesa, se da cuenta de la cantidad de conceptos, tecnologías y formas de trabajo ha aprendido durante el desarrollo de este proyecto.

A continuación detallo las aportaciones más notables que recibirá la empresa al optar por esta filosofía para el desarrollo de *software*:

- **Reutilización de código:** como se ha nombrado anteriormente, esta filosofía tiene como concepto base la creación de componentes genéricos tanto para la parte del *front-end* con *Angular* como para la del *back-end* con *NodeJS* que contengan funcionalidades específicas y bien definidas para poder ser reusados en cualquier proyecto que se vaya a desarrollar y cuente con esos requisitos. Por ejemplo, un requisito fundamental en la mayoría de aplicaciones web es la de contar con un sistema de registro de usuarios. En este caso sería posible crear un componente para el servidor de autenticación, permitiendo a los demás proyectos que necesitase de un sistema de registro la reutilización de este en vez de desarrollarlo de nuevo.

Esto se traduce en que el todo tiempo para cada proyecto que se haya invertido en la creación de componentes genéricos será tiempo de dedicación que se le restará a los siguientes proyectos, optimizando la cadena de desarrollo y permitiendo a los clientes tener su proyecto en un período más corto.

- **Homogeneización de código:** es necesaria la introducción de un sistema de gestión de repositorios para que todo el mundo se rija bajo las mismas pautas a la hora de desarrollar. Repositorios de este tipo, como *Git*, dan la oportunidad de crear una pequeña comunidad dentro de la empresa. Cada vez que una nueva pieza de código es creada, esta no se integra directamente a la rama principal de la aplicación, pasa a otra en la que al menos otro desarrollador del equipo revisará el código para ver que esté libre de errores.

Con esto se desea conseguir que todos los miembros, independientemente del proyecto que tengan asignados, sigan las mismas metodologías y patrones de trabajo a la hora de desarrollar aplicaciones.

Al contar con dichos repositorios para la compartición de código donde se almacenan aquellos componentes que son genéricos, cualquier persona puede ser capaz de navegar por los ejemplos y aprender, de forma rápida, el estilo de programación establecido en la empresa. De este modo es posible reducir la curva de aprendizaje para nuevas incorporaciones o traslados de personas entre diferentes proyectos.



- **Distribución de trabajo y colaboración:** la creación de repositorios para compartir componentes genéricos facilita la colaboración de los desarrollos indistintamente de la oficina en la que estén. Esto supone que dos personas que se encuentren en oficinas diferentes y que cada cual esté involucrado en un proyecto diferente, a fin de cuentas estarán implementando código que tendrá los mismos estándares de la empresa en ambos lados. Para tener una mejor idea de cuál sería el resultado expongo un pequeño ejemplo de lo que podría ser un desarrollo colaborativo factible.

Pongamos que existen dos equipos de trabajo, uno en Valencia y el otro en Barcelona, donde se dedican al desarrollo de componentes *front-end* con *Angular*. Actualmente hay en marcha tres proyectos que cuentan con dos programadores de dicho tipo para cada uno. El proyecto *A*, llevado a cabo en Valencia, se encarga de implementar un componente para un listado emergente y un *popup* con filtros, orden, paginación, etc. Una vez se han completado estos componentes, son subidos al repositorio genérico, que en este caso se ha denominado *Angular_Everis_Componentes* y comienzan con el desarrollo de las ventanas específicas para el uso de dichos componentes en uso propio proyecto

El proyecto *B*, desarrollado en Barcelona, tiene como uno de sus requisitos la implementación de una ventana de listado. Los programadores acuden al repositorio genérico, nombrado anteriormente, y encuentran dicho componente que ha sido producido en la oficina de Valencia. Lo descargan y lo incluyen en su proyecto, ahorrando tiempo que habría sido desperdiciado en implementar algo ya existente. Con este tiempo que han ganado integran el componente del listado con uno de mapas. Suben dicho componente al repositorio de *Angular_Everis_Componentes* para ponerlo a disposición de todos.

Empieza un nuevo proyecto en Valencia, el *C*, consulta el repositorio en busca de aquello que necesita y descubre que todos los componentes requeridos ya se encuentran implementados. Hace una descarga e implementa las ventanas que les darán uso.

El proyecto *A* recibe un evolutivo en el que se especifica la inclusión de un nuevo listado que haga uso de mapas, consultan el repositorio y encuentran dicho componente ya elaborado. Este es descargado y adaptado al dicho proyecto. El resultado de todo esto son tres proyectos y seis personas que han colaborado entre sí, ahorrando tiempo para cada desarrollo y de forma indirecta aun encontrándose en localizaciones separadas.

- **Focalización en la tareas:** al automatizar cuestiones como el entorno de programación, pruebas, despliegues en producción, testeo, etc., los programadores pueden concentrarse plenamente en el desarrollo. Las tareas se ven simplificadas y los programadores pueden aprovechar el tiempo ahorrado que supone para aumentar su experiencia.

- **Eficiencia:** al incluir una nueva incorporación a un proyecto, este tendrá un proceso de adaptación mucho menor debido a la organización y documentación ya escrita, teniendo incluso materias suficiente para empezar un nuevo proyecto. Con un primer proyecto acabado ya no será necesario empezar ninguno nuevo desde cero.
- **Disminución de costes:** el tener la opción de reutilizar trabajo creado por otros compañeros para diferentes proyectos permite disminuir los costes de producción respecto a un comienzo desde cero. Se trata de un proceso retroalimentado en el que cada implementación en el presente supone una menos en el futuro. Con esto en mente, la empresa puede tener menos reparo en embarcarse en proyectos de mayor complejidad sin tener que llevar a cabo un aumento en los costes de producción.
- **Disminución de riesgo de errores:** el hecho de la reutilización de componentes ya implementados, reduce la aparición de errores. Si a esto le sumamos los métodos de integración continua y los test, tanto los unitarios como los de integración, los costes en producción referentes al control de errores se ven reducidos drásticamente.
- **Mejora de rendimiento y seguridad:** resumidamente, esta filosofía se centra en la realización de iteraciones constantes sobre los desarrollos ya realizados de manera que, si se descubre una mejora para algún componente ya implementado, una vez introducida el resto de proyectos que hagan uso de este contarán con dicha actualización. Por ejemplo, y volviendo al ejemplo anterior, si se descubre una forma más segura de fortalecer el servidor de autenticación, tan solo hay que introducir la mejora y el resto de proyectos heredarán la actualización.
- **Automatización de test y despliegues:** al no tener la necesidad de reservar un tiempo específico del proyecto a la realización de test o despliegues, debido a que estos están automatizados, el nivel de agilidad que se adquiere permite a los desarrolladores focalizarse más en la calidad del producto y sus funcionalidades. Al mismo tiempo, el contar con la aplicación de test para cada modificación introducida, eleva el nivel de seguridad que se tiene sobre el *software* que se desarrolla.

Todas estas ventajas se presentan de forma muy suculenta, no obstante el optar por esta arquitectura de microservicios junto al sistema de control de versiones y la integración continua es solo una parte de todo el proceso. La otra es la introducción de una nueva filosofía de trabajo y desarrollo de *software* basada en la eficiencia. Es imprescindible que se comprenda como tal si se pretende aprovechar al máximo todas y cada una de las cualidades que ofrece esta arquitectura. Es necesario un pequeño cambio de mente hacia una mayor colaboración entre personas y proyectos dentro de la empresa, basada en la implantación de un conocimiento común y una compartición total de trabajo. Solo de esa forma, el cambio será realmente a mejor.

En el ámbito de lo que concierne al futuro de este proyecto, me complace decir que ya se han empezado a dar los primeros pasos. Nuevos cursos sobre las tecnologías descritas en esta



Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

memoria están empezando a impartirse, amén de la sustitución de *Subversion* por *Git* en cada vez más proyectos. La prueba piloto desarrollada ha gustado al cliente, ofreciendo interés por desarrollar cuatro productos nuevos que hagan uso de su estructura. Al mismo tiempo, nuevos proyectos que surgen hacen uso de la plantilla resultante de esta aplicación web, convirtiéndose en el inicio de la forma de trabajo del futuro en esta empresa.

11. Bibliografía

Abbott, M.L. and Fisher, M.T. (2015), *The Art of Scalability*, 2ª Edición, Disponible en: <http://theartofscalability.com/> (Fecha de acceso: 20 Abril 2017)

Adanza, F. (2016), *Implementing continuous delivery: Jenkins or Bamboo?*, Disponible en: <https://www.getzephyr.com/insights/implementing-continuous-delivery-jenkins-or-bamboo> (04 Junio 2017)

Atlassian, (n.d.), *Understanding the Bamboo CI Server*, Disponible en: <https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html> (04 Junio 2017)

Badola, V. (2015), *Microservices architecture: advantages and drawbacks*, Disponible en: <http://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/> (Fecha de acceso: 20 Abril 2017)

Braithwaite, B. (2015), *Getting started with Karma for AngularJS Testing*, Disponible en: <http://www.bradoncode.com/blog/2015/05/19/karma-angularjs-testing/> (19 Mayo 2017)

Bruce, J. (2012), *What Is Git & Why You Should Use Version Control If You're a Developer*, Disponible en: <http://www.makeuseof.com/tag/git-version-control-youre-developer/> (10 Junio 2017)

Burgess, A. (2011), *Testing Your JavaScript With Jasmine*, Disponible en: <https://code.tutsplus.com/tutorials/testing-your-javascript-with-jasmine--net-21229> (14 Mayo 2017)

cjrequena (2016), *Arquitectura Orientada a Microservicios*, Disponible en: <https://cjrequena.github.io/micro-services/2016/09/20/micro-services-architecture-es.html> (20 Abril 2017)

Clemson, T. (2014), *Testing Strategies in a Microservice Architecture*, Disponible en: <https://martinfowler.com/articles/microservice-testing/> (13 Mayo 2017)

Collins-Sussman, B. Fitzpatrick, B. y Pilato, C.M (2006), *Version Control with Subversion*, Disponible en: <http://svnbook.red-bean.com/en/1.7/svn.intro.whatis.html> (08 Junio 2017)

Curran, P. (2017), *Bamboo vs Jenkins*, Disponible en: <https://www.checkmarx.com/2017/03/12/bamboo-vs-jenkins/> (04 Junio 2017)

Dan North & Associates (2006), *Introducing BDD*, Disponible en: <https://dannorth.net/introducing-bdd/> (14 Mayo 2017)

Demiss, D. (2016), *End-to-End Testing: What, Why, and How?*, Disponible en: <https://pspdfkit.com/blog/2016/e2e-testing/> (15 Mayo 2017)



Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Driessen. V (2010), *A successful Git branching model*, Disponible en : <http://nvie.com/posts/a-successful-git-branching-model/> (10 Junio 2017)

Eriksson, E. (2016), *A step-by-step white box testing example*, Disponible en: <http://reqtest.com/testing-blog/white-box-testing-example/> (19 Mayo 2017)

Fernández, R. (2014), *MongoDB: qué es, cómo funciona y cuándo podemos usarlo (o no)*, Disponible en: <https://www.genbetadev.com/bases-de-datos/mongodb-que-es-como-funciona-y-cuando-podemos-usarlo-o-no> (22 Abril 2017)

Grebenets, M. (2015), *Bamboo vs Jenkins*, Disponible en: <http://mgrebenets.github.io/mobile%20ci/2015/01/29/bamboo-vs-jenkins> (04 Junio 2017)

Gupta, L. (2016) , *Jasmine – JavaScript Unit Testing Tutorial with Examples*, Disponible en: <http://howtodoinjava.com/scripting/javascript/jasmine-javascript-unit-testing-tutorial/> (14 Mayo 2017)

Hauer .P (2015), *Discussing Docker. Pros and Cons*, Disponible en: <https://blog.philiphauer.de/discussing-docker-pros-and-cons/> (13 Junio 2017)

Huston, T. (2017), *What is a Microservices Architecture?*, Disponible en: <https://smartbear.com/learn/api-design/what-are-microservices/> (12 Abril 2017)

Inman, H. (2011), *Five Reason Why Developers Choose Jenkins Over Hudson for Continuous Integration*, Disponible en: <https://www.cloudbees.com/blog/five-reasons-why-developers-choose-jenkins-over-hudson-continuous-integration> (04 Junio 2017)

Kajamohideen. A (2015), *Expose your localhost to web in 50 seconds using Ngrok*, Disponible en: <https://vmokshagroup.com/blog/expose-your-localhost-to-web-in-50-seconds-using-ngrok/> (13 Junio 2017)

Kenlon. S (2016), *What is Git?*, Disponible en: <https://opensource.com/resources/what-is-git> (10 Junio 2017)

Martin Fowler (2006), *Continuous Integration*, Disponible en: <https://martinfowler.com/articles/continuousIntegration.html> (03 Junio 2017)

Myllärniemi, V. (2015), *CSE-C3600-SoftwareArchitecturePart1* (12 Abril 2017)

Nadalin, A. (2015), *On monoliths, service-oriented architecture and microservices*, Disponible en: <http://odino.org/on-monoliths-service-oriented-architectures-and-microservices/> (24 Abril 2017)

Neagle, C. (n.d.) *An introduction to version control*, Disponible en: <http://guides.beanstalkapp.com/version-control/intro-to-version-control.html> (05 Junio 2017)

Patil, R. & Simmons, F. (XXXX), *11 Things You Cannot Do With Jenkins, And Never Will*, Disponible en: <https://www.flosum.com/limitations-of-jenkins-for-salesforce/> (04 Junio 2017)

Ragnar .L (n.d.), *When should I start load testing?*, Disponible en: <https://techbeacon.com/when-should-i-start-load-testing> (13 Mayo 2017)

Roslonek, L. (2016), *Microservice testing*, Disponible en: <https://testdetective.com/microservices-testing/> (13 Mayo 2017)

Rouse, M. (n.d.), *Unit Testing*, Disponible en: <http://searchsoftwarequality.techtarget.com/definition/unit-testing> (08 Mayo 2017)

Selenium easy (n.d.), *What is Jenkins and Use of it?*, Disponible en: <http://www.seleniumeasy.com/jenkins-tutorials/what-is-jenkins-and-advantages-of-jenkins-continuous-integration-tool> (04 Junio 2017)

Sneha Kunja, S. (2017), *What is Jenkins?*, Disponible en: <https://vmokshagroup.com/blog/what-is-jenkins/> (04 Junio 2017)

Software Testing Fundamentals (n.d.), *Unit Testing Fundamentals*, Disponible en: <http://softwaretestingfundamentals.com/unit-testing/> (08 Mayo 2017)

Software Testing Help, (2017), *Why End to End Testing is Necessary and How to Perform?*, [Online], Disponible en: <http://www.softwaretestinghelp.com/what-is-end-to-end-testing/>, (09 Mayo 2017)

Sommerville, I. (2011), *Software Engineering*, 9ª Edición (24 Abril 2017)

Stafford, J. (n.d.), *Why automated continuous integration is a must for microservices success*, [Online], <http://searchmicroservices.techtarget.com/feature/Why-automated-continuous-integration-is-a-must-for-microservices-success> (03 Junio 2017)

Svnvsgit (2016), *Subversion vs. Git: Myths and Facts*, Disponible en: <https://svnsgit.com/> (12 Junio 2017)

Tee, J. (2016), *Modularity thrives when microservices and SOA comes together*, Disponible en: <http://www.theserverside.com/feature/Microservices-and-SOA-together-From-monolith-to-modularity> (18 Abril 2017)

Up Guard (2017), *Github vs Bitbucket*, Disponible en: <https://www.upguard.com/articles/github-vs-bitbucket> (10 Junio 2017)

Vaughan-Nichols. S (2017), *What is Docker and why is it so darn popular?*, Disponible en: <http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/> (13 Junio 2017)

Verma, R. (2015), *Centralized vs Distributed Version Control System*, Disponible en: <http://scmquest.com/centralized-vs-distributed-version-control-systems/> (06 Junio 2017)



Estudio de sistema de control de versiones y uso de los mismos en proyectos de integración continua

Wallgren, A. (2016), *Continuous Delivery and Release Automation for Microservices*, Disponible en: <http://electric-cloud.com/blog/2016/01/continuous-delivery-and-release-automation-for-microservices/> (03 Junio 2017)