



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

TRABALLO FIN DE GRAO

GRAO EN ENXEÑARÍA INFORMÁTICA

Mención en Enxeñaría do Software

Análisis, diseño e implementación de un sistema para divulgar las causas y efectos del Cambio Climático a través de datos

Autor: Diego Hermida Carrera

Director: Diego Andrade Canosa

A Coruña, junio de 2018

A mi padre, por hacer todo esto posible.

Agradecimientos

A mi director, Diego Andrade Canosa, por su buen juicio y experiencia, de gran ayuda a lo largo del proyecto.

A otros profesores de la Facultad de Informática: Laura Castro, Pedro Cabalar o Javier Parapar, que arrojaron luz sobre aspectos como pruebas, metodologías, etc.

A mi familia: mi padre, madre, hermano y padrinos; también los que están allá arriba. En especial, a mi abuelo, el del bigote.

A mi novia Alejandra, por haberme aguantado durante toda la carrera, y ayudarme a ser cada día la mejor versión de mí mismo.

A mis amigos, tanto a los de toda la vida, como a los que surgen durante esta etapa universitaria.

Resumen

Fuertes evidencias científicas demuestran que la mayor causa del calentamiento global que sufre nuestro planeta desde mitad del siglo XX son las emisiones contaminantes producidas por el ser humano. Pese a los esfuerzos internacionales, plasmados en los protocolos de Montreal (1987), Kyoto (1997), y el reciente acuerdo de París (2016), las emisiones antropogénicas –principal causa del cambio climático– no solo no se han reducido, sino que se incrementan a un ritmo cada vez mayor. Con todo, el nivel de compromiso de la población con la causa medioambiental es generalmente bajo.

En este contexto, surge la idea de crear un sistema cuya finalidad sea concienciar a las personas sobre la situación climática actual.

De cara al usuario final, el sistema consistirá en una aplicación Web que muestre de forma clara y concisa información que demuestre que la acción del hombre contamina cada vez más nuestro planeta, y que esto tiene un efecto evidente sobre el clima. Con el objetivo de poder llegar al mayor número de usuarios, la interfaz se adaptará a cualquier dispositivo, y contará con un diseño «Mobile First». Para mejorar el impacto de la información sobre el usuario, los datos proporcionados estarán personalizados en base a su localización geográfica. Internamente, la aplicación recopilará todos estos datos periódicamente de fuentes heterogéneas; los filtrará, ordenará y procesará, para poder ser utilizados por la aplicación Web. Debido a la alta probabilidad de que las fuentes de los datos cambien a lo largo del tiempo, se hará especial énfasis en que el sistema sea modular y fácilmente extensible. Del mismo modo, al tratarse de un sistema con una demanda de usuarios y de recursos difícil de predecir, se tratará de construir un sistema escalable.

Palabras clave

- ✓ API
- ✓ Cambio Climático
- ✓ Django
- ✓ Docker
- ✓ Jenkins
- ✓ Python
- ✓ Scrum
- ✓ SonarQube
- ✓ Telegram

Índice General

1. INTRODUCCIÓN	1
1.1. OBJETIVOS.....	1
1.2. ANÁLISIS DEL ESTADO DEL ARTE.....	2
1.2.1. <i>Electricity Map</i>	2
1.2.2. <i>World Air Quality Index</i>	3
1.2.3. <i>NASA: Vital Signs of the Planet</i>	3
1.3. ESTRUCTURA DEL DOCUMENTO	3
2. CONTEXTUALIZACIÓN	5
2.1. HISTORIA DE LA TIERRA	5
2.2. EL SER HUMANO Y EL CLIMA	7
2.3. CAMBIO CLIMÁTICO. ¿REALIDAD O SUPERSTICIÓN?	9
2.4. PREDICCIONES CLIMÁTICAS	11
2.5. JUSTIFICACIÓN DEL PROYECTO.....	15
3. METODOLOGÍA.....	17
3.1. DESCRIPCIÓN DE LA METODOLOGÍA	17
3.1.1. <i>Aspectos metodológicos de Scrum</i>	17
3.1.2. <i>Roles</i>	17
3.1.3. <i>Artefactos</i>	18
3.1.4. <i>Reuniones</i>	18
3.2. JUSTIFICACIÓN.....	18
3.3. ADAPTACIÓN DE SCRUM AL PROYECTO	19
3.4. OTROS ASPECTOS METODOLÓGICOS.....	20
3.5. ANÁLISIS Y FASES PREVIAS.....	20
4. ANÁLISIS. TAREAS PREVIAS	21
4.1. ADQUISICIÓN DE CONOCIMIENTO Y TRABAJO TÉCNICO	21
4.2. ELABORACIÓN DEL PROTOTIPO.....	21
4.3. RECOPILACIÓN DE FUENTES DE INFORMACIÓN.....	22
4.4. ESPECIFICACIÓN DE REQUISITOS.....	23
4.4.1. <i>Requisitos no funcionales</i>	23
4.4.2. <i>Requisitos funcionales</i>	25
4.5. DEFINICIÓN DE LA ARQUITECTURA DEL SISTEMA	26
4.5.1. <i>Descripción y justificación de la arquitectura</i>	26
4.5.2. <i>Diseño del Subsistema de Recolección de Datos</i>	28
5. TECNOLOGÍA Y HERRAMIENTAS.....	30
5.1. PYPY	30
5.2. PYCHARM.....	30
5.3. MONGODB.....	31
5.4. PYMONGO	32
5.5. GIT	32
5.6. DOCKER.....	32

5.7.	POSTGRESQL	33
5.8.	DJANGO	33
5.9.	JENKINS Y SONARQUBE	33
5.10.	HTML5, CSS3, JQUERY, JAVASCRIPT Y AJAX.....	34
5.11.	BOOTSTRAP 4	34
5.12.	TRELLO.....	34
5.13.	OTRAS TECNOLOGÍAS.....	35
6.	PRIMER SPRINT.....	37
6.1.	COMPONENTES RECOLECTOR.....	37
6.1.1.	<i>La clase base DataCollector.....</i>	37
6.1.2.	<i>Máquina de estados finitos</i>	38
6.1.3.	<i>Métodos de un DataCollector</i>	41
6.1.4.	<i>Estructura modular de los DataCollectors</i>	44
6.1.5.	<i>Configuración y estado</i>	45
6.1.5.1.	<i>Configuración.....</i>	45
6.1.5.2.	<i>Estado.....</i>	46
6.1.6.	<i>MongoDB</i>	49
6.2.	COMPONENTE LOGGER.....	50
6.3.	COMPONENTE IMPORTADOR DE MÓDULOS	51
6.4.	COMPONENTE PRINCIPAL.....	51
6.4.1.	<i>Concurrencia</i>	52
7.	SEGUNDO SPRINT	55
7.1.	COMPONENTE SUPERVISOR.....	55
7.1.1.	<i>Paso de mensajes</i>	55
7.1.2.	<i>Comprobación de ejecuciones</i>	56
7.1.3.	<i>Reporte de ejecución.....</i>	57
7.2.	CONFIGURACIÓN.....	59
7.3.	EL MECANISMO DE RETRASO EXPONENCIAL	60
7.4.	EJECUCIONES FINITAS.....	62
7.5.	UTILIDADES	63
7.6.	PRUEBAS.....	63
7.6.1.	<i>Pruebas de unidad</i>	63
7.6.2.	<i>Pruebas de integración.....</i>	64
7.6.3.	<i>Resultados.....</i>	66
7.7.	DESPLIEGUE.....	66
7.7.1.	<i>Volúmenes</i>	68
7.7.2.	<i>Puertos</i>	68
7.7.3.	<i>Logs y ficheros de estado</i>	69
8.	TERCER SPRINT	70
8.1.	REFACTORIZACIÓN	71
8.1.1.	<i>Paquetes</i>	71
8.1.2.	<i>Clases.....</i>	71
8.2.	COMPONENTE API.....	72
8.2.1.	<i>Características</i>	72
8.2.2.	<i>Tokens y ámbitos.....</i>	72
8.2.3.	<i>Operaciones</i>	73
8.2.4.	<i>Implementación</i>	75

8.3.	SUBSISTEMA DE CONVERSIÓN DE DATOS	76
8.3.1.	<i>Componentes conversor</i>	76
8.3.2.	<i>El ORM de Django</i>	78
8.3.2.1.	Claves primarias	79
8.3.2.2.	El script manage.py.....	80
8.4.	<i>BUGS</i>	80
8.4.1.	[BUG-032]	81
8.4.2.	[BUG-040]	84
8.4.2.1.	Docker y la seguridad	84
8.5.	ESCALABILIDAD	85
8.5.1.	<i>Consecuencias</i>	85
8.6.	INTEGRACIÓN E INSPECCIÓN CONTINUA	86
8.7.	PRUEBAS.....	86
8.8.	DESPLIEGUE.....	87
9.	CUARTO Y QUINTO SPRINT	88
9.1.	<i>SCRIPTS DE INSTALACIÓN</i>	88
9.2.	<i>HEALTHCHECK</i>	90
9.3.	SUBSISTEMA DE LA APLICACIÓN WEB.....	91
9.3.1.	<i>Mobile-First</i>	91
9.3.2.	<i>Geolocalización</i>	92
9.3.3.	<i>Google Maps</i>	93
9.3.4.	<i>Uso de gráficos y diagramas</i>	93
9.3.5.	<i>Validaciones</i>	94
9.3.6.	<i>AJAX</i>	95
9.3.7.	<i>Caché</i>	95
9.3.8.	<i>Internacionalización (i18n)</i>	97
9.3.9.	<i>Administración</i>	97
9.3.10.	<i>Contador de likes</i>	98
9.3.11.	<i>Reconocimiento de fuentes de información</i>	99
9.3.12.	<i>Tratamiento de errores</i>	99
9.4.	PRUEBAS.....	100
9.4.1.	<i>Pruebas contra base de datos</i>	101
9.5.	DESPLIEGUE.....	102
9.6.	SEGURIDAD	103
9.6.1.	<i>Cross Site Request Forgery (CSRF)</i>	103
9.6.2.	<i>HTTPS</i>	103
10.	ANÁLISIS DEL PROYECTO	105
10.1.	PLANIFICACIÓN INICIAL	105
10.2.	PLANIFICACIÓN POR <i>SPRINT</i>	106
10.3.	RECURSOS.....	107
10.4.	CALENDARIZACIÓN	107
10.5.	COSTE.....	107
10.6.	MÉTRICAS DEL PROYECTO.....	108
11.	CONCLUSIONES	109
11.1.	TRABAJO FUTURO.....	109
ANEXO A.	SELECCIÓN DE LOCALIZACIONES A MONITORIZAR	111

A.1.	ESPECIFICACIÓN DE CRITERIOS.....	111
A.2.	CLASIFICACIÓN	112
A.2.1.	<i>Regiones climáticas</i>	112
A.2.2.	<i>Índice de Desarrollo Humano</i>	114
A.2.3.	<i>Popularidad en Twitter</i>	115
A.3.	RESULTADOS.....	116
ANEXO B. INTEGRACIÓN E INSPECCIÓN CONTINUA		123
B.1.	INTEGRACIÓN CONTINUA.....	123
B.1.1.	<i>Soluciones</i>	123
B.1.2.	<i>Configuración de Jenkins</i>	124
B.1.2.1.	Problemas iniciales.....	124
B.1.2.2.	Pasos.....	125
B.1.2.3.	Resultados	127
B.1.3.	<i>Despliegue</i>	128
B.2.	INSPECCIÓN CONTINUA.....	128
B.2.1.	<i>Soluciones</i>	129
B.2.2.	<i>Configuración de SonarQube</i>	129
B.2.3.	<i>Integración de SonarQube con Jenkins</i>	130
B.2.4.	<i>Integración de SonarQube con PyCharm</i>	130
B.2.5.	<i>Despliegue</i>	131
ANEXO C. CONFIGURACIÓN DE TELEGRAM.....		132
C.1.	<i>BOTS</i>	132
C.2.	COMPONENTE TELEGRAM.....	132
C.3.	PASOS	133
C.3.1.	<i>Errores de configuración</i>	136
ANEXO D. CONVENCIONES DE ETIQUETADO.....		138
D.1.	ETIQUETADO DE <i>RELEASES</i>	138
D.2.	ETIQUETADO DE <i>BUGS</i>	138
ANEXO E. CONTENIDO DEL CD.....		141
ANEXO F. DOCUMENTACIÓN ADICIONAL		142
BIBLIOGRAFÍA		143

Índice de Código

Código 1: Definición de la clase <code>TransitionState</code> . Por simplicidad sólo se muestra el método constructor.....	39
Código 2: Extracto de la clase <code>DataCollector</code> . En él, se muestran los métodos principales que pueden ser sobreescritos o requieren algún comentario para ser comprendidos.....	41
Código 3: Ejemplo de fichero de configuración de un módulo del subsistema.....	45
Código 4: Campos requeridos para el <code>STATE_STRUCT</code> de cualquier <code>DataCollector</code>	47
Código 5: Ejemplo de sobreescritura de las operaciones de un <code>DataCollector</code> cuando existe un campo en la <code>STATE_STRUCT</code> que no es JSON serializable.....	48
Código 6: Ejemplo de uso del Patrón Proxy. Este método delega en la implementación de la librería <code>pymongo</code> ; aunque la conexión es inicializada antes de la llamada al método, si fuera necesario (lazy loading).....	49
Código 7: Firmas de los métodos de la clase base <code>DataCollector</code> empleados por el componente supervisor.....	57
Código 8: Ejemplo de reporte de ejecución del subsistema.....	58
Código 9: Ejemplo de carga de configuración del sistema.....	59
Código 10: Acceso a una variable de configuración del sistema desde código Python.....	60
Código 11: Fragmento de un registro de log indicando que el mecanismo de retraso exponencial ha frenado la recopilación de datos.....	60
Código 12: Caso de uso de la función <code>time_limit</code>	62
Código 13: Registro de log que indica una ejecución abortada, tras alcanzar ésta el tiempo máximo permitido.....	62
Código 14: Ejemplo de uso de mocks en pruebas de unidad.....	64
Código 15: Ejemplo de prueba de integración del subsistema.....	65
Código 16: Configuración de un servicio usando <code>docker-compose</code>	68
Código 17: Estructura del fichero que almacena los tokens de acceso al API.....	73
Código 18: Uso de la Programación Orientada a Aspectos para simplificar la implementación del API.....	76
Código 19: Implementación del método <code>_check_dependencies_satisfied</code>	77
Código 20: Definición de una entidad empleando Django.....	79
Código 21: Creación de un objeto de tipo <code>ObjectId</code>	81
Código 22: Extracto del fichero de generación de índices de MongoDB.....	83
Código 23: Opciones del script de instalación del Subsistema de Conversión de Datos.....	89
Código 24: Implementación del healthcheck para el componente API.....	90
Código 25: Implementación de un método del servicio que hace uso de la caché.....	96
Código 26: Obtención de una instancia del servicio mediante una clase factoría.....	97
Código 27: Implementación de un escenario de prueba para una operación de un servicio. ...	101
Código 28: Invocación del script de instalación del subsistema.....	103

Índice de Gráficos

Gráfico 1: Resultados de una encuesta a nivel mundial, reflejando las mayores preocupaciones de la población. Los votos obtenidos proceden de distintos países, grupos de edad, niveles de educación y desarrollo económico.....	15
Gráfico 2: Evolución del uso promedio de Internet en la Unión Europea durante la última década. Fuente: Eurostat	16
Gráfico 3: Ejemplo de gráfico Sprint Burndown generado con la herramienta Trello y el plugin “Scrum by Vince”. El gráfico pertenece al sprint 3 del proyecto.....	35

Índice de Ilustraciones

Ilustración 1: Proyecciones de las temperaturas hacia el año 2100 en los escenarios RPC 2.6 (izquierda) y RPC 8.5 (derecha). Fuente: IPCC.....	13
Ilustración 2: Porcentaje de individuos con acceso a Internet en algunos países europeos (2016). Fuente: Eurostat.....	16
Ilustración 3: Comparativa entre el prototipo y la aplicación web implementada. Se puede apreciar que la estructura es prácticamente idéntica.....	22
Ilustración 4: Contenido inicial del Product Backlog para el proyecto. Los requisitos o user stories están ordenados según su prioridad.....	25
Ilustración 5: Vista física de la arquitectura del sistema.....	26
Ilustración 6: Diagrama de componentes en el que se muestra el mecanismo de comunicación entre el Subsistema de Recolección de Datos y el Subsistema de Conversión de Datos a través de un API, que exporta una interfaz bien definida.....	27
Ilustración 7: Diagrama de secuencia que muestra la interacción entre los componentes del Subsistema de Recolección de Datos.	29
Ilustración 8: Primer user story del Product Backlog, “Recopilación de datos de fuentes de terceros”.....	37
Ilustración 9: API ofrecida por la clase base DataCollector	38
Ilustración 10: Estados de transición de un DataCollector	40
Ilustración 11: Estructura de un módulo del subsistema.	44
Ilustración 12: Elementos de un DataCollector	44
Ilustración 13: Ejemplo de mensaje de error CRITICAL enviado automáticamente por uno de los subsistemas vía Telegram.....	50
Ilustración 14: Descripción del proceso recursivo de importación de módulos.....	51
Ilustración 15: Diagrama de concurrencia para una ejecución del subsistema.....	53
Ilustración 16: User stories a implementar durante el sprint 2.	55
Ilustración 17: Implementación de la clase Message . Los objetos de esta clase actúan como forma de comunicación entre los DataCollectors y el supervisor.	56
Ilustración 18: Directorio de configuración global del sistema.....	59
Ilustración 19: Escenario donde se muestra el uso del mecanismo de retraso exponencial.	61
Ilustración 20: Comparación entre contenedores y máquinas virtuales. Fuente: Docker.....	67
Ilustración 21: User stories a implementar durante el sprint 3.	70
Ilustración 22: Refactorización de la estructura de paquetes del proyecto.	71

<i>Ilustración 23: Diagrama entidad-relación del sistema.</i>	78
<i>Ilustración 24: Estructura de directorios del sistema tras haber escalado un subsistema.</i>	86
<i>Ilustración 25: User stories a implementar durante el sprint 4.</i>	88
<i>Ilustración 26: Visualización de una página de la aplicación web en un iPhone 5S vs iPad.</i> ...	92
<i>Ilustración 27: Botón que permite activar la geolocalización.</i>	92
<i>Ilustración 28: Marcadores de localizaciones monitorizadas en Google Maps.</i>	93
<i>Ilustración 29: Gráficos en un iPhone 6/6s/7/8 vs dispositivo de escritorio.</i>	94
<i>Ilustración 30: Feedback negativo al detectar datos inválidos.</i>	95
<i>Ilustración 31: Feedback positivo tras enviar datos válidos.</i>	95
<i>Ilustración 32: Interfaz de la página de gestión de mensajes de contacto.</i>	98
<i>Ilustración 33: Botón “me gusta”.</i>	99
<i>Ilustración 34: Ejemplos de reconocimiento de fuentes de información.</i>	99
<i>Ilustración 35: Página de error ante un recurso no encontrado.</i>	100
<i>Ilustración 36: Vista de despliegue de la aplicación web.</i>	102
<i>Ilustración 37: Diagrama de Gantt con la planificación grosso modo del proyecto.</i>	105
<i>Ilustración 38: Conjunto de regiones climáticas de la Tierra (entre 1901-2010), según la clasificación Köppen.</i>	111
<i>Ilustración 39: Footer de la aplicación web, con el enlace que permite listar todas las localizaciones monitorizadas.</i>	122
<i>Ilustración 40: Pipeline de construcción de la aplicación.</i>	127
<i>Ilustración 41: Vulnerabilidad detectada por SonarLint, al introducir una dirección IP “hard-coded” en el fichero.</i>	131
<i>Ilustración 42: Etiquetas para caracterizar bugs del sistema.</i>	139

Índice de Tablas

<i>Tabla 1: Características más representativas de los cuatro escenarios empleados por los modelos para predecir el clima durante el siglo XXI.</i>	12
<i>Tabla 2: Especificación de requisitos no funcionales del sistema.</i>	24
<i>Tabla 3: Comparativa entre el esfuerzo estimado y real por sprint.</i>	106
<i>Tabla 4: Estimación de coste para los recursos del proyecto.</i>	107
<i>Tabla 5: Desglose del coste del proyecto.</i>	108
<i>Tabla 6: Regiones climáticas según la clasificación Köppen. Las columnas hacen referencia a la primera, segunda y tercera letra del código de las regiones climáticas, respectivamente.</i>	113
<i>Tabla 7: Resultados de la clasificación por regiones climáticas. Cada localización obtendrá la puntuación de su región asociada (en azul).</i>	114
<i>Tabla 8: Estadísticas según los niveles de HDI. La puntuación asignada a cada nivel aparece a la derecha (en azul).</i>	115
<i>Tabla 9: Clasificación de las localizaciones según los criterios combinados. Las puntuaciones finales pueden verse a la derecha (en azul).</i>	122

1. INTRODUCCIÓN

Fuertes evidencias científicas demuestran que la mayor causa del calentamiento global que sufre nuestro planeta desde mitad del siglo XX son las emisiones contaminantes producidas por el ser humano. Pese a los esfuerzos internacionales, plasmados en los protocolos de Montreal (1987), Kyoto (1997), y el reciente acuerdo de París (2016), las emisiones antropogénicas –principal causa del cambio climático– no solo no se han reducido, sino que se incrementan a un ritmo cada vez mayor. De hecho, existe un número máximo de toneladas de dióxido de carbono –CO₂– que *podemos* emitir. De sobrepasarse, se producirán cambios irreversibles en el clima de la Tierra; provocando, entre otros, la desaparición del Ártico. En 2011, ya se habían emitido más de la mitad.

Sin embargo, no es necesario sobreponerse dicho límite para darse cuenta de que la el ser humano ya ha comenzado a alterar el clima. Numerosos organismos internacionales miden el impacto de la acción humana en el medio ambiente, destacando hechos alarmantes a múltiples niveles: aumento de la temperatura media del planeta, deshielo en los polos, pérdida de biodiversidad, aumento exponencial de las emisiones contaminantes, etcétera.

Pese a la gravedad de la situación, el nivel de compromiso de la población con la causa medioambiental es generalmente bajo, según datos recogidos por encuestas internacionales y nacionales. Este hecho resulta especialmente alarmante, dado que el desconocimiento por parte de la población impide una implicación más directa para frenar el Cambio Climático.

De lograrse una mayor difusión sobre este fenómeno, podría ejercerse una mayor presión sobre los gobiernos; de forma que se establezcan políticas más restrictivas y mayores sanciones a las organizaciones que las incumplan.

En este contexto, surge la idea de crear un sistema cuya finalidad sea concienciar a las personas sobre la situación climática actual.

1.1. Objetivos

Se plantea construir un sistema para divulgar las causas y consecuencias del Cambio Climático a través de datos. Entre los objetivos concretos del proyecto, se sitúan:

- Mostrar, de forma clara y concisa, información que demuestre que la acción del hombre contamina cada vez más nuestro planeta, y que esto tiene un efecto evidente –y negativo– sobre el clima.

- Llegar al mayor número de usuarios que sea posible. Para ello, se empleará un diseño que permita visualizar la información en cualquier dispositivo. Además, dado que la mayoría de la población no posee conocimiento científico, se priorizará el uso de elementos multimedia: gráficos, diagramas, etc. para mostrar el contenido.
- Mejorar el impacto de la información sobre el usuario, mostrando los datos en base a su localización geográfica.
- La construcción de un mecanismo capaz de recopilar datos de fuentes heterogéneas, de forma periódica; filtrarlos, ordenarlos y procesarlos, para poder ser visualizados por la aplicación que empleará el usuario final.
- Debido a la alta probabilidad de que las fuentes de los datos cambien a lo largo del tiempo, se hará especial énfasis en que el sistema sea modular y fácilmente extensible.
- Del mismo modo, al tratarse de un sistema con una demanda de usuarios y de recursos difícil de predecir, se tratará de construir un sistema escalable.

La enumeración anterior recoge los objetivos del proyecto, a alto nivel. En las primeras etapas del desarrollo, estos objetivos se refinrarán, conformando los requisitos del sistema. Esto se detalla en la sección [4.4. Especificación de requisitos](#).

1.2. Análisis del estado del arte

Tras realizar una breve investigación, se encuentran varias aplicaciones con características comunes a las del sistema que se pretende implementar.

1.2.1. Electricity Map

De la mano de un desarrollador franco-danés nace [Tomorrow](#), una compañía enfocada en la concienciación sobre el Cambio Climático.

De su oferta de productos, cabe destacar [Electricity Map](#); una aplicación web que, en tiempo real, muestra datos sobre las fuentes de energía de los países: renovables, carbón, nuclear, petróleo, gas... junto con el porcentaje que éstas representan respecto al total. También se muestran las importaciones y exportaciones energéticas entre países, y el coste del kW/h; todo ello, en tiempo real.

El objetivo principal de este sistema, además de concienciar a la población, es que los dispositivos inteligentes accedan a la red energética cuando más económico resulte el acceso, y menor sea su huella de CO₂.

Esta aplicación expone un API de pago, mediante el que se pueden acceder a predicciones y datos históricos de las fuentes de energía.

1.2.2. World Air Quality Index

Este proyecto supone un esfuerzo de cientos de organizaciones gubernamentales, entre otros, con el fin de monitorizar la calidad del aire a nivel mundial, en tiempo real. Cuenta con más de 10000 estaciones en más de 80 países, y expone un API público y gratuito que permite acceder a los últimos registros de calidad del aire.

Entre sus objetivos, se encuentra el alertar a la población de la calidad del aire en las ciudades, más aún cuando se alcanzan límites insalubres.

De cara al usuario final, se muestran los datos mediante una aplicación web, empleando mapas. Se puede acceder a un reporte detallado por estación, con información meteorológica y predicciones de la calidad del aire. La aplicación es accesible desde [este](#) enlace.

1.2.3. NASA: Vital Signs of the Planet

La NASA ha creado un portal informativo con el fin de divulgar las causas, efectos y posibles soluciones al Cambio Climático. Además, muestra datos de las “constantes vitales” de la Tierra: el nivel de CO₂ en la atmósfera, el aumento de la temperatura media global del planeta, la disminución de las masas de hielo ártico y antártico, etc.

El sitio web está disponible en el [siguiente](#) enlace.

1.3. Estructura del documento

Esta memoria se estructura de la siguiente manera:

- a) En el presente capítulo, se detallan los objetivos del proyecto y analiza el estado del arte.
- b) El siguiente capítulo ofrece una perspectiva histórica y analítica del Cambio Climático, a modo de contextualización. Al final, se realiza la justificación del proyecto.
- c) El capítulo tercero trata los aspectos metodológicos, y su adaptación al proyecto.
- d) El cuarto capítulo detalla aspectos importantes del proyecto, que deben quedar fuera de las iteraciones; como la captura de requisitos, definición de la arquitectura, etc.
- e) El capítulo quinto aborda la elección y justificación de las tecnologías que se emplean a lo largo del proyecto.
- f) Los cuatro siguientes capítulos tratan los aspectos desarrollados durante la acometida de los *sprints*. Al principio de cada uno, se ofrece un resumen de la planificación de la iteración en curso.

- g) Prosigue el análisis del proyecto, en términos de esfuerzo, tiempo y coste. A mayores, se incluye una sección de estadísticas.
- h) El último capítulo describe las conclusiones extraídas de la realización del proyecto, y las líneas de trabajo futuro.

2. CONTEXTUALIZACIÓN

El objetivo de este capítulo es el de dotar al lector del contexto en el que se sitúa el proyecto. Concretamente, en este apartado se ofrece:

- Una breve perspectiva de la Historia de la Tierra.
- Detalles sobre momentos clave en la relación entre el ser humano y el clima.
- Evidencias de la existencia del Cambio Climático.
- Proyecciones climáticas de aquí a cien años.

Teniendo en cuenta estos aspectos, junto a lo visto en el capítulo anterior, se aborda al final de este capítulo la **justificación** del proyecto.

2.1. Historia de la Tierra

La Tierra tiene aproximadamente 4.600 Ma (millones de años) de edad. Durante la Historia han tenido lugar siete grandes eras glaciales de las cuales seguimos inmersos en la última.

Durante la formación del planeta, éste era una bola semi-incandescente con una frenética actividad volcánica, bombardeada por grandes meteoritos y cometas, sufriendo enormes cataclismos. Durante esos 500 Ma –duración estimada de la formación de la Tierra– el gas principal dominante en la atmósfera era el dióxido de carbono (CO₂), siendo nuestra atmósfera de composición similar a las de Venus y Marte.

Alrededor de hace 3.800 Ma se data el origen del agua líquida, a raíz del vapor de agua que escapaba del interior de la Tierra; formando un manto nuboso cada vez más denso que, con la caída de grandes precipitaciones, formó los océanos primitivos. Hace 3.600 Ma surgieron las primeras formas de vida, las cianobacterias. Éstas empezaron a consumir grandes cantidades de CO₂ y enriquecer la atmósfera de oxígeno. Durante este período, la Tierra era bastante más cálida que en la actualidad. Fue alrededor de hace 2.300 Ma cuando la superficie terrestre se cubrió de hielo¹, produciéndose el fenómeno conocido como «Tierra Blanca». Unos 300 Ma más tarde, el planeta volvió a calentarse y se fue poblando por organismos cada vez más complejos, hasta que tuvo lugar la segunda «Tierra Blanca» –hace

¹ los científicos discrepan en las causas del enfriamiento de la Tierra, siendo las teorías más aceptadas, actualmente: el impacto de un meteorito, que generó una nube de polvo tan densa que provocó un enfriamiento global; un aumento extremo de la actividad volcánica; y una nube interestelar de polvo cósmico que redujo significativamente la cantidad de radiación solar incidente en el planeta.

unos 1.200 Ma–, sobreviviendo solo los seres vivos mejor adaptados, en el fondo de los océanos y en la zona ecuatorial.

Tras esta segunda era glacial siguió una nueva etapa cálida, que abarcó hasta hace 700 Ma. Entonces, sucedió el tercer episodio de «Tierra Blanca», el más importante y frío de todos. Se cree que el hielo se extendió por todo el planeta, llegando a cubrir casi la totalidad del área ecuatorial. Los científicos consideran que el fenómeno responsable de fundir la densa capa de hielo fue la actividad volcánica, al generar un potente efecto invernadero.

Con esto, se inició la Era Paleozoica, produciéndose el alza de las temperaturas, hasta el final del período Ordovícico –hace unos 430 Ma–, al producirse la quinta era glacial en la Tierra. Tras el fin de ésta, durante el período Silúrico, y sobre todo en los períodos Devónico y Carbonífero, nuevamente sucede una etapa cálida, dando lugar a la mayor explosión de vida del planeta: grandes y frondosos bosques; multiplicación de las poblaciones de insectos; salto evolutivo de los anfibios a tierra firme (apareciendo los primeros reptiles), etcétera. Al final del Carbonífero se produjo el cuarto episodio de «Tierra Blanca», junto con un cambio radical en el clima y paisaje: la formación del supercontinente Pangea (que se fractura en siguientes épocas geológicas hasta conseguir una distribución similar a la actual).

Al terminar el Paleozoico, se inició el Mesozoico –que duró unos 180 Ma–, en cuyos períodos Triásico, Jurásico y Cretácico se produjo otra explosión de vida; siendo la Tierra dominada por los dinosaurios. Esto terminó bruscamente hace 65 Ma, momento en el que se produjo la extinción de la mitad de la flora y fauna del planeta (incluyendo a los dinosaurios), siendo la teoría más aceptada como causa de dicho fenómeno el impacto de un asteroide –de unos 10 km de diámetro– en la Península de Yucatán, México [1].

Este hito marca el fin del Mesozoico, entrando en la última Era, el Cenozoico. Ésta consta de dos períodos, Terciario y Cuaternario –iniciado hace 1,8 Ma, y el actual–; caracterizado por una alternancia más regular de ciclos fríos y cálidos (interglaciales). Dentro del período Cuaternario, nos encontramos en la última época interglacial: el Holoceno².

² hay autores que sugieren que nos encontramos ya en el Antropoceno, refiriéndose a la época en la que los seres humanos hemos comenzado a influir en el clima.

2.2. El ser humano y el clima

La aparición de los primeros homínidos –hace entre 4,5 y 5 Ma– fue posible, entre otros aspectos, porque comenzaron a darse unas condiciones climáticas adecuadas. Pese a que desde hace unos 13.000 años que gozamos de un clima *benigno* y *uniforme*, a través de la Historia puede observarse que el clima es uno de los factores clave que provocan la decadencia de una civilización.

Los científicos sitúan al *Australopithecus afarensis* (primer humanoide) alrededor de hace 3 Ma en la sabana africana. A consecuencia de la glaciación en la que estaba sumida la Tierra, los monos más inteligentes decidieron bajar de los árboles y caminar en busca de alimento, motivados por las duras condiciones climáticas. El período cálido posterior a esa glaciación permitió el desarrollo de las comunidades de humanoides (género *Homo*), y su posterior evolución y expansión de África a otros continentes.

Hace unos 100.000 años apareció el hombre de Neandertal (*Homo sapiens*) y, posteriormente, el hombre de Cromagnon (*Homo sapiens sapiens*). Ambos convivieron durante varios miles de años, hasta que este último se impuso al final del último ciclo glacial (glaciación *Würm*), debido a que los neandertales, pese a ser más robustos, se adaptaron peor a las condiciones climáticas y poco a poco fueron desapareciendo.

Un período cálido –entre los años 13.000 y 11.000 a.C.– fue el que permitió el movimiento migratorio desde Asia hacia América a través del estrecho de Bering. Otro de estos períodos –entre los años 5.000 y 3.000 a.C.– favoreció la fundación de la cuna de la civilización occidental, en Mesopotamia.

El clima también constituyó un factor clave en el declive de la civilización egipcia, con la aridificación del Sahara central y oriental; y de la expansión del Imperio Romano, beneficiada por las abundantes cosechas junto con veranos más secos e inviernos más suaves de lo normal en toda la región mediterránea (aunque también en buena parte de Europa). Esta época es conocida como «*Período Cálido Romano*», y tocó techo hacia el año 400 d.C. En los años sucesivos, los inviernos se fueron volviendo cada vez más fríos, desplazando a los pueblos bárbaros hacia el sur de Europa. Éstos, cada vez más numerosos, fueron haciendo mella en el Imperio; provocando su caída en el 476 d.C. Este hecho dio paso a la época más oscura de la Historia: la Edad Media.

La población europea sobrevivió a duras penas durante varios siglos³ debido a las inclemencias del tiempo, hasta la llegada de otro período cálido acuñado como «*Pequeño Óptimo Medieval*» –entre los años 1100 y 1300– Este período vino acompañado de generosas precipitaciones, incrementando la producción agrícola y ganadera⁴.

A mediados del siglo XIV, el «*Pequeño Óptimo Medieval*» dio paso a lo que se conoce como «*Pequeña Edad de Hielo*» (PEH), que se prolongaría hasta mediados del siglo XIX. Los climatólogos están seguros de que la humedad y el frío de este período fueron una de las causas de la gran magnitud de la Peste Negra, responsable de la muerte de un tercio de la población europea entre los años 1347 y 1352. La PEH se resume como la consecución –casi ininterrumpida– de 150 años de inviernos largos y fríos, junto con veranos cortos y frescos. De todos estos años, se considera que los peores fueron los situados entre 1650 y 1700. No obstante, existen indicadores que apuntan a que la PEH no produjo grandes efectos en el hemisferio sur.

Los últimos años de la PEH coincidieron con el establecimiento de una red mundial de observatorios meteorológicos, hacia 1850. El período iniciado desde entonces y hasta nuestros días, podemos considerarlo como cálido y benigno; lo que, sin duda, ha contribuido al crecimiento económico y demográfico más importante de la historia de la humanidad.

No obstante, desde mediados del siglo XX, es constatable científicamente que ha aumentado la variabilidad climática, conduciendo a fenómenos meteorológicos cada vez más extremos⁵. El físico José Miguel Viñas Rubio considera que “*hemos entrado en un nuevo ciclo climático, nunca antes conocido por los seres humanos, aunque sí por la Tierra, al que debemos de adaptarnos lo mejor posible para evitar una catástrofe humana de enormes dimensiones*” [2].

³ la duración de este período frío no fue igual en toda Europa. En Escandinavia perduró hasta el año 700, aproximadamente. En Centroeuropa se postergó hasta mediados del siglo VIII a principios del siglo IX, mientras que en la Península Ibérica no fue hasta principios del siglo XI cuando se suavizaron las temperaturas.

⁴ en España, el buen clima favoreció la aparición de la Mesta: privilegios reales para la producción de lana y su exportación a Europa. Ésta fue fundada en 1273, durante el reinado de Alfonso X El Sabio

⁵ durante los últimos años se han batido récords de calor en ciertos lugares; aunque, en menor medida, también se han registrado temperaturas negativas nunca antes alcanzadas [4] [5].

2.3. Cambio climático. ¿Realidad o superstición?

Habiendo realizado un breve recorrido por el panorama histórico de la Tierra, al lector podrán resultarle significativos los siguientes hechos:

- La Tierra, de forma natural, intercala ciclos fríos y cálidos; de duración variable –desde varias décadas hasta cientos de millones de años–.
- La humanidad es sumamente vulnerable a cambios en el clima, llegando éstos a ser un factor clave en la desaparición de civilizaciones enteras.
- Actualmente nos encontramos en un ciclo cálido, tras el fin del período conocido como «*Pequeña Edad de Hielo*» (PEH).
- El clima se ha vuelto más extremo desde mediados del siglo XX.

Los escépticos acerca del cambio climático podrán plantearse la siguiente cuestión:

“Si tras finalizar la PEH hemos entrado en un ciclo cálido, ¿es realmente la humanidad la causante del aumento de las temperaturas (y otros fenómenos), o es la propia Tierra –como ya sucedió en otras ocasiones– la principal responsable de estos sucesos?”. La respuesta viene de mano de la comunidad científica [3]:

“Las emisiones antropogénicas de gases invernadero han aumentado desde la era pre-industrial, conducidas en su mayor parte por el crecimiento de la población y la economía, y son mayores que nunca. Esto lleva a concentraciones de dióxido de carbono (CO_2), metano (CH_4) y óxido nitroso (N_2O) sin precedentes en los últimos 800.000 años. Sus efectos, junto con otros fenómenos causados por el hombre, han sido detectados en todo el sistema climático, y es extremadamente probable que hayan sido la causa dominante del calentamiento observado desde mediados del siglo XX.”

Más en detalle, las concentraciones de CO_2 en la atmósfera se incrementaron un 40% desde 1750; las de CH_4 un 150% y las de N_2O un 20%. El aumento es más considerable desde 1970; de hecho, la mitad de las emisiones de CO_2 desde 1750 se han producido en los últimos 40 años. De estas emisiones, el 30% son absorbidas por los océanos, causando su acidificación.

Los combustibles fósiles y los procesos industriales conforman la principal causa de emisiones de CO_2 (un 78% del total). Desde el año 2000, éstas han crecido más rápido que anual en todos los sectores –excepto en agricultura y explotación maderera-. Los más contaminantes son el sector energético (35%), industrial

(21%) y transportes (14%); no obstante, un 25% de las emisiones son indirectas (producción de energía y calor para uso final, etcétera).

Los científicos consideran que más de la mitad del incremento de las temperaturas –desde 1950– ha sido causado por la actividad humana. Además, se tiene una certeza entre *media* y *muy alta* de que los siguientes fenómenos han sido causados por dicha actividad:

- Se ha visto afectado el ciclo global del agua –desde 1960–, cambiando los patrones de precipitaciones y modificando la salinidad de los océanos (en la superficie y también en las profundidades).
- En algunas regiones, el cambio de la frecuencia de precipitaciones, junto con el deshielo y la contaminación, han provocado alteraciones en los sistemas hidrológicos, afectando a la disponibilidad del agua en términos de cantidad y calidad [6].
- Ha aumentado el nivel del mar –desde los años setenta–, debido al calentamiento de las aguas y a la pérdida de masa glaciar.
- Han aumentado las temperaturas –desde mediados del siglo XX– en todos los continentes. Esto ha contribuido enormemente a la retirada de los glaciares por todo el globo desde los años 60, y al incremento de la superficie de hielo que se derrite anualmente en Groenlandia desde 1993, y en el hemisferio norte en general.
- Se ha reducido la masa de hielo en el Ártico desde 1979. Según la NASA, ésta disminuye un 13,3% cada década [7]. Desde el primer registro en 1979, la extensión de hielo ártico ha menguado un 39,8%.
- Se han alterado los patrones migratorios y estadias temporales de especies terrestres y marinas; se ha incrementado la tasa de mortalidad de especies vegetales; se han dañado o destruido arrecifes de coral, debido a la acidificación y aumento de temperatura de ciertas regiones oceánicas; y han aumentado las zonas pobres en oxígeno (O_2) en los océanos Pacífico, Atlántico y Índico, entre otros.
- La actividad humana ha causado un impacto más negativo que positivo en las cosechas, sobre todo entre las regiones productoras de trigo y maíz; a pesar de que en zonas en las que se cultiva arroz o soja, los cambios son menos sustanciales.
- Se ha incrementado –de forma no significativa– el número de muertes durante olas de calor, y disminuido los casos de muerte por frío. Además, los cambios en temperaturas y precipitaciones han favorecido la transmisión de enfermedades por el agua y vectores.

- Ha aumentado el número de eventos extremos observados desde la segunda mitad del siglo XX, de los cuales, ligados a la actividad humana podemos distinguir:
 - La disminución del número de días fríos y, por consiguiente, el aumento del número de días cálidos al año; sobre todo en Europa, Asia y Australia. En algunas regiones, se ha duplicado la probabilidad de ocurrencia de olas de calor.
 - El aumento del número de regiones que sufren precipitaciones torrenciales, sobre todo en Norteamérica y Europa, desde 1970.
 - El aumento de la frecuencia y magnitud de inundaciones a nivel mundial, aumentando el coste por daños para estos eventos.
 - La aparición de sequía en zonas geográficamente inconsistentes.
 - El aumento de la actividad ciclónica en el Atlántico norte.
 - El aumento del nivel del mar y eventos extremos en los océanos.

Anteriormente se mencionó que la humanidad es vulnerable a cambios en el clima. Pero, ¿en qué medida? En primer lugar, la exposición y vulnerabilidad están ligados a un amplio rango de factores sociales, económicos y culturales: distribución de la riqueza, demografía, migraciones, acceso a la tecnología e información, empleo, valores sociales, acciones por parte de los gobiernos e instituciones, etcétera. La población marginada en uno o más de estos factores es especialmente vulnerable al cambio climático: destrucción de zonas residenciales, reducciones en las cosechas, aumento de precios en bienes de primera necesidad... Además, la existencia de conflictos armados dificulta la adaptación ante cambios en el clima.

Ante la situación actual, algunos gobiernos están comenzando a desarrollar planes de adaptación y políticas, que consideran el cambio climático como un problema grave, a tratar a medio y largo plazo [8] [9].

2.4. Predicciones climáticas

Las predicciones del clima se basan en modelos climáticos, que estimulan aspectos como la temperatura de la atmósfera y los océanos, precipitaciones, vientos, formaciones nubosas, corrientes oceánicas y masas de hielo. Además, se tienen en cuenta factores de la actividad humana tales como el crecimiento económico y demográfico, estilos de vida, uso energético, tecnología y políticas climáticas.

Para obtener las proyecciones de los parámetros a evaluar –concentración de gases invernadero, temperatura media del planeta, etc.– se emplean los modelos climáticos usando información presente en escenarios, conocidos como *Representative Concentration Pathways* o RPCs. Éstos describen trayectorias alternativas para los parámetros de interés, y cubren hasta el año 2100. Se describen cuatro escenarios, del más esperanzador al más pesimista, en la siguiente tabla [10]:

RPC	Emisiones de CO₂	Población	Energía
RPC 2.6	Alcanzan un máximo en 2020 (0 hacia 2080), con una concentración máxima de 440 ppm.	Alrededor de 9.000 millones en 2050. El desarrollo económico es alto.	Abandono del petróleo como combustible, sustituido por otros combustibles fósiles y energías renovables.
RPC 4.5	Niveles un 50% más altos que los del 2000 hacia 2050. Hacia 2080 se estabilizan a los niveles del 2000. Concentración de 520 ppm en 2070, incrementándose levemente.	Crecimiento económico y demográfico ligeramente inferior al de RPC 2.6.	Consumo de petróleo prácticamente constante hasta 2100. Energías nuclear y renovable como secundarias.
RPC 6.0	Emisiones un 100% superiores en 2060, con concentraciones siempre crecientes –reduciendo el ritmo a final de siglo–, alcanzando 620 ppm en 2100.	Crecimiento hasta los 10.000 millones de habitantes, con bajos PIB.	Pico energético hacia 2060, llegando al nivel de RPC 2.6 en 2100. Consumos similares a RPC 4.5 con menor importancia las energías secundarias.
RPC 8.5	Emisiones un 375% superiores a las actuales en 2100. Concentración de 950 ppm en 2100 y subiendo durante, mínimo, 100 años más.	Crecimiento alto de población, hasta los 12.000 en 2100. Desigualdad social.	Consumo creciente durante todo el siglo, multiplicándose por tres respecto al consumo del año 2000. Uso del petróleo hasta 2070 y carbón hasta 2100.

Tabla 1: Características más representativas de los cuatro escenarios empleados por los modelos para predecir el clima durante el siglo XXI.

Si se logra cumplir con los objetivos de RPC 2.6, las predicciones indican que la temperatura media del planeta se mantendrá a final de siglo por debajo de 2°C sobre niveles pre-industriales. Sin embargo, si se llega a RPC 8.5, antes de 2100: las temperaturas habrán superado en 4°C los niveles pre-industriales, el nivel del mar habrá subido en torno a 0,7 metros, la masa de hielo del Ártico habrá desaparecido y el pH de los océanos será un 105% superior al actual. No obstante, para cualquier RPC (aunque los efectos serán más acusados para RPC más altos):

- El Ártico comenzará a calentarse más rápido que el planeta en general, y éste, que los océanos. No obstante, la tendencia al calentamiento es común.
- Si se da el escenario RPC 8.5, se extremarán las sequías en las zonas secas (latitudes medias y regiones subtropicales) y las precipitaciones en las zonas húmedas. Sin embargo, se harán más frecuentes e intensos los deslizamientos de tierra e inundaciones en zonas tropicales húmedas; y se intensificará el efecto de El Niño⁶ y el período del monzón en las zonas afectadas por estos fenómenos, respectivamente.

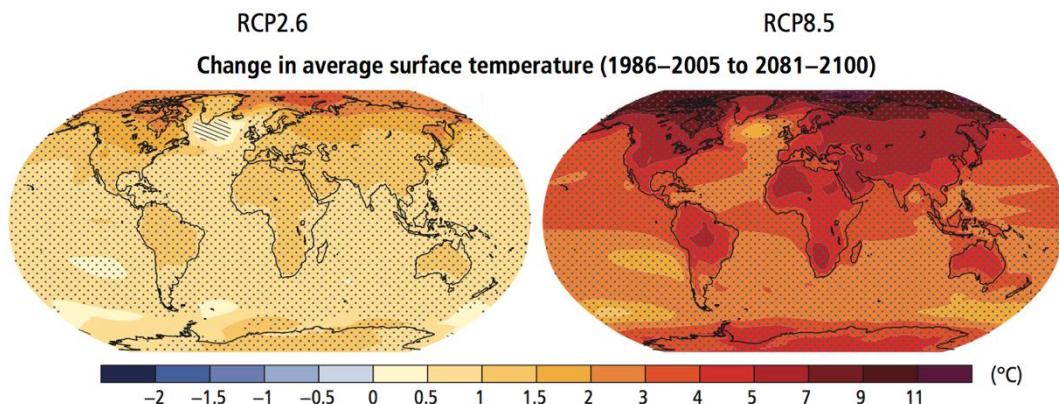


Ilustración 1: Proyecciones de las temperaturas hacia el año 2100 en los escenarios RPC 2.6 (izquierda) y RPC 8.5 (derecha). Fuente: IPCC.

- Se producirán olas de calor con mayor frecuencia y duración. Podrían producirse inviernos extremadamente fríos ocasionalmente.
- Se empobrecerán los océanos –sobre todo en latitudes medias–, al disminuir la cantidad de O₂ disuelta en ellos.

⁶ El Niño es un fenómeno meteorológico que afecta al Océano Pacífico. De frecuencia irregular y duración variable –entre 9 meses y varios años–, produce un calentamiento de las aguas superficiales en las regiones este y central del Pacífico ecuatorial. Suele producirse durante el invierno, causando alza de temperaturas en el oeste de Canadá y el noroeste de EE.UU., y temperaturas más bajas en el sur de EE.UU. El fenómeno opuesto a El Niño se conoce como La Niña, y produce efectos contrarios a éste: alza de temperaturas en el sur de EE.UU. y temperaturas más bajas en el oeste de Canadá y el noroeste de EE.UU [11].

Estas dos fases opuestas también son conocidas como *ENSO cycle (El Niño-Southern Oscillation)*.

Los expertos aseguran que el calentamiento global causado por emisiones de CO₂ es irreversible hasta pasados varios siglos, por lo que deben tomarse medidas para eliminar el CO₂ de la atmósfera. Para el cumplimiento del objetivo de mantener la temperatura global por debajo de 2°C sobre niveles pre-industriales, es necesario limitar las emisiones a un tope de 3650 GtCO₂ (giga-toneladas de CO₂), la mitad de las cuales ya fueron emitidas hacia 2011 [12]. De no cumplirse estos objetivos, las consecuencias serán negativas a múltiples niveles:

- Un porcentaje considerable de especies terrestres y marinas será incapaz de adaptarse y aumentará su riesgo de extinción –sobre todo en los escenarios RPC 4.5, RPC 6.0 y RPC 8.5–, causado por la combinación de diversos factores climáticos (aumento de temperaturas, pérdida de masa de hielo, acidificación de los océanos...) junto con la actividad humana directa (sobreexplotación de hábitats, contaminación, introducción de especies invasivas...).
- Las especies marinas en zonas sensibles al cambio climático migrarán a otras zonas, condicionando la actividad pesquera a nivel mundial. Se estima que las especies afectadas migrarán de las regiones de latitudes tropicales hacia latitudes medias y altas.
- Se verán afectados los ecosistemas marinos, especialmente los arrecifes de coral y ecosistemas polares; debido a la acidificación de los océanos, junto con la actividad humana.
- La mortalidad de especies vegetales, junto con la deforestación, incrementará el riesgo de aumento de temperaturas, sequías, tormentas, vendavales, fuegos y expansión de enfermedades. Se verá amenazada la biodiversidad, la producción maderera, la calidad del agua, la absorción de CO₂ por parte de las especies vegetales, y la actividad económica, entre otros.
- Debido al aumento del nivel del mar, se incrementará el riesgo de inundación y erosión de las zonas costeras. Teniendo en cuenta que la mayoría de la población mundial se concentra en estas zonas [13], la exposición al riesgo será sustancialmente mayor en las próximas décadas.
- A lo largo de las próximas décadas se reducirá la cantidad de agua potable (superficial y subterránea) disponible en la mayoría de regiones subtropicales. No obstante, se cree que en latitudes altas aumentarán las reservas de agua. Sin embargo, debido a la contaminación, a la sedimentación y a las inundaciones, la calidad –en general– del agua potable será menor, aumentando el riesgo de propagación y contagio de enfermedades.
- Se reducirán (según ciertos modelos) hasta en un 25% las cosechas, afectando al acceso y estabilidad del precio de los alimentos. Las zonas que

mayor impacto sufrirán son regiones rurales, viendo aumentado el riesgo de caer en la pobreza.

- El cambio climático incrementará la propagación de enfermedades por vectores, agua y alimentos; causando lesiones y llegando incluso a ser fatales para la población afectada –sobre todo, en países en vías de desarrollo–. También aumentará el riesgo de desnutrición en regiones pobres. Para el escenario RPC 8.5, en 2100 las altas temperaturas y humedad en ciertas áreas habrán comprometido la actividad humana, limitando la producción de alimentos y el trabajo en zonas exteriores.

2.5. Justificación del proyecto

En definitiva, la mayor parte de la comunidad científica coincide en que el **cambio climático** es real [14] y que, de seguir con la tendencia actual de emisión de gases invernadero y otras actividades dañinas, se alcanzarán resultados catastróficos para la Tierra y la vida que reside en ella. Pese a todo, el nivel de compromiso de la población con el problema del cambio climático es muy bajo [15]:

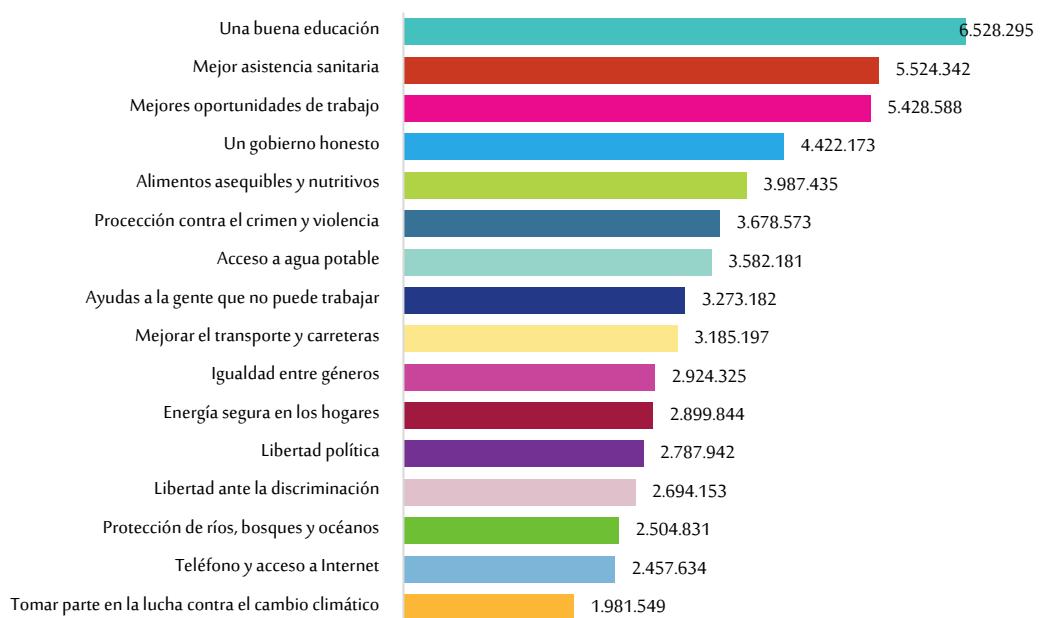


Gráfico 1: Resultados de una encuesta a nivel mundial, reflejando las mayores preocupaciones de la población. Los votos obtenidos proceden de distintos países, grupos de edad, niveles de educación y desarrollo económico.

A nivel nacional, los resultados son similares. Según un estudio del CIS (Centro de Investigaciones Sociológicas), tan sólo un 0,3% de los españoles considera que los problemas medioambientales se encuentran entre las tres mayores preocupaciones de la población [16].

Teniendo en cuenta la aparente falta de interés de la población por la cuestión, junto con el aumento exponencial del uso de las TIC (Tecnologías de la Información y Comunicación) por todos los grupos de edad –en especial Internet– [17],

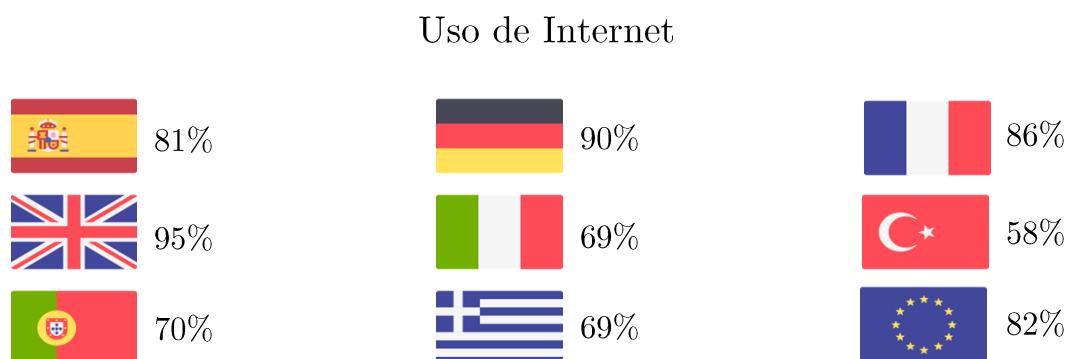


Ilustración 2: Porcentaje de individuos con acceso a Internet en algunos países europeos (2016). Fuente: Eurostat.

© iconos: <http://www.flaticon.com/packs/international-flags>, Freepik.

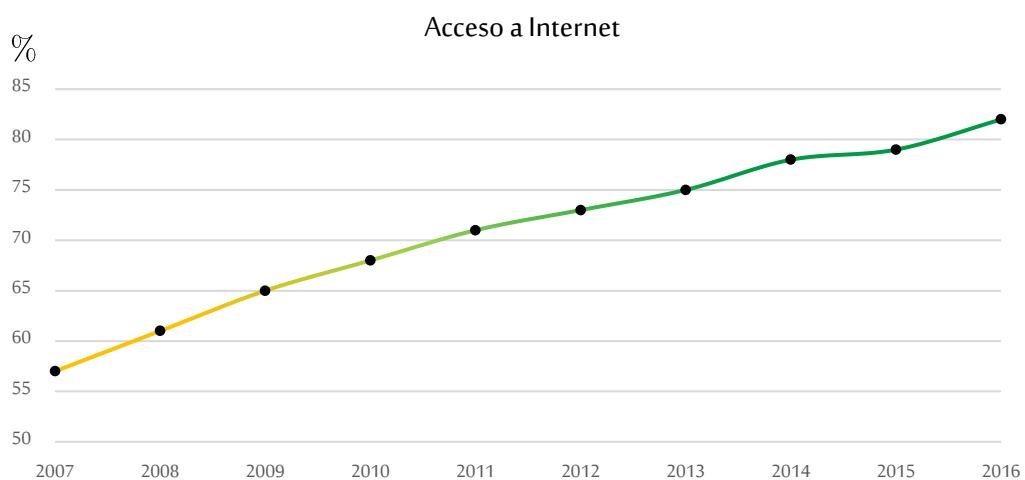


Gráfico 2: Evolución del uso promedio de Internet en la Unión Europea durante la última década. Fuente: Eurostat

parece viable la creación de una aplicación cuya finalidad sea **concienciar** a las personas sobre la situación actual del clima, mostrando información relevante sobre distintos aspectos, de forma **clara y sencilla**.

3. METODOLOGÍA

Este capítulo se centra en la descripción del proceso y metodología seguidos durante el desarrollo del proyecto. En concreto, se detallarán:

- La descripción de la metodología empleada en el proyecto: aspectos metodológicos, características y particularidades.
- La justificación de la metodología elegida.
- La adaptación de la metodología al proyecto.
- Otros aspectos metodológicos.

Además, al final del capítulo se realizan unas consideraciones acerca de la fase de análisis del proyecto.

3.1. Descripción de la metodología

Se ha decidido emplear una metodología basada en **Scrum** para acometer el proyecto, si bien adaptada a las necesidades del proyecto. En esta sección, se recogen los aspectos metodológicos de Scrum –según lo visto en la asignatura *Metodologías de Desarrollo*–, y la adaptación de la metodología al proyecto.

3.1.1. Aspectos metodológicos de Scrum

Scrum se encuadra dentro de lo que se conocen como metodologías ágiles⁷. Ha sido usada con éxito en todo tipo de proyectos (desde equipos de 6 personas hasta múltiples equipos con cientos de personas).

En Scrum, las iteraciones se denominan *sprints*, y su duración aproximada es de entre dos semanas y un mes. Las entregas se denominan *releases*.

Scrum no prescribe prácticas de ingeniería, y proporciona libertad absoluta a la hora de elaborar documentación.

Entre *sprints* se permiten cambios de tareas, pero una vez comienza uno, éstas no pueden sufrir modificaciones. Dentro del *sprint*, los equipos superponen la toma de requisitos, diseño, implementación y pruebas.

3.1.2. Roles

- *Product Owner*: Define las funcionalidades del producto y las prioriza, decide las fechas y contenido de las *releases*, y acepta o rechaza el trabajo del equipo.

⁷ **metodologías ágiles**: Surgen con el fracaso de las metodologías clásicas como *Waterfall* (Cascada), y constituyen un cambio de filosofía a la hora de desarrollar software. De entre sus características, se puede destacar la importancia del cliente, la facilidad frente al cambio, la simplicidad, la auto-organización de los equipos, la búsqueda de la excelencia técnica y el diseño, etc.

- *Scrum Master*: Es un rol gestor/desarrollador. Ayuda a la resolución de conflictos, asegurando la productividad y cooperación del equipo.
- *Stakeholders*: Constituye el personal interesado en el proyecto.
- *Equipo*: Sus miembros trabajan a tiempo completo en el proyecto (excepto algún especialista, como un administrador de bases de datos, que puede estar presente a tiempo parcial). El entorno de trabajo ideal ha de ser abierto, con conversación continua.

3.1.3. Artefactos

- *Product Backlog*: Contiene los requisitos del proyecto. Éstos se reordenan antes de cada *sprint* (ya que cada *sprint* debe implementar los más importantes).
- *Sprint Backlog*: Contiene las tareas a realizar en el *sprint*, junto a su planificación. Éstas se deciden por todo el equipo, y cada día debe actualizarse el trabajo estimado restante.
- *Sprint Burndown y Velocity*: El primero es un gráfico que relaciona el trabajo restante y realizado con la duración del *sprint*. El segundo es una medida del esfuerzo completado por *sprint*, que permite ver la productividad del equipo.

3.1.4. Reuniones

- *Sprint planning*: Su objetivo es analizar el *Product Backlog* y crear el *Sprint Backlog* a partir de él.
- *Daily meeting*: El objetivo de esta reunión es “ponerse al día” con el trabajo realizado.
- *Revisión del sprint*: Se realiza una *demo* de lo desarrollado durante el *sprint*. Es informal, y todo el mundo relacionado con el proyecto puede asistir.
- *Retrospectiva del sprint*: Es la única reunión en la que se decide si es necesario refactorizar⁸ algún aspecto del producto desarrollado. Además, se contemplan otras mejoras y prácticas que deben realizarse o dejar de seguir.

3.2. Justificación

La elección de esta metodología se soporta en las siguientes razones:

1. Las metodologías ágiles aportan **seguridad** cuando los requisitos no están claros al principio. Este es el caso del proyecto, ya que, al trabajar con fuentes de datos externas, pueden surgir problemas e imprevistos que dificulten la planificación.

⁸ **refactorizar**: realizar cambios en la implementación que, aunque no afectan al comportamiento funcional del producto, sí pueden afectar al rendimiento. El objetivo de estos cambios es incorporar mejores principios de ingeniería, seguir patrones de diseño o de arquitectura, etc.

2. Este tipo de metodologías son adecuadas cuando desconocimiento de cara a la implementación: Las tecnologías y herramientas escogidas para el desarrollo del proyecto son desconocidas para el *Equipo*. Las metodologías ágiles integran la tolerancia al cambio, por lo que será posible re-planificar de forma más sencilla si surgen complicaciones.
3. Se decide emplear Scrum en lugar de otras metodologías ágiles, como *Extreme Programming* –XP– o *Kanban*; debido a que ni es restrictiva en exceso⁹ ni por defecto¹⁰.

Con todo, debe tenerse en cuenta que Scrum se basa en el trabajo colectivo, por lo que no será posible usarlo “en todo su esplendor”.

3.3. Adaptación de Scrum al proyecto

Habiendo comentado los roles, artefactos, reuniones y otros aspectos de Scrum; a continuación, se detalla cómo se adapta la metodología al desarrollo del proyecto:

- Los roles de *Product Owner* y *Scrum Master* son ejercidos por el director del proyecto, Diego Andrade Canosa.
- El rol *Equipo* está desempeñado por el alumno, Diego Hermida Carrera.

En cuanto a la duración de cada *sprint*, ésta es variable en términos de tiempo, ya que el alumno debe que compaginar el desarrollo del proyecto con otras obligaciones académicas y representación estudiantil. No obstante, los *sprints* son similares en términos de esfuerzo.

Cada *sprint* proporciona un incremento sustancial en el proyecto, operativo y listo para ser puesto en producción; si bien dada la complejidad inicial del proyecto, el primer *sprint* no cumple estas condiciones.

El *Product Backlog* se genera una vez se ha definido la arquitectura del sistema y se tienen claros los componentes a diseñar e implementar. Éste recoge los requisitos de alto nivel del producto – en forma de tarjetas– que se priorizan y ordenan.

Antes de cada *sprint*, el *Equipo* se reúne con el *Scrum Master* y el *Product Owner* (director y alumno), y se deciden qué requisitos del *Product Backlog* se implementarán, junto a su planificación en tareas, conformando así el *Sprint Backlog*.

⁹ XP define aspectos metodológicos que, por la naturaleza del equipo y el proyecto, no se adecúan bien. Ejemplos serían el *pair programming* (programación por parejas), TDD (desarrollo dirigido por pruebas), o el uso ciclos cortos de 2 semanas.

¹⁰ Kanban es la metodología ágil menos restrictiva, de las estudiadas en la asignatura *Metodoloxías de Desenvolvemento*. Al definir pocas prácticas y/o restricciones, su implementación requiere de una gran madurez organizacional.

Al final de cada *sprint*, nuevamente se mantiene una reunión: la *Revisión del Sprint*. En ella, se realiza la *demo* del producto, y se discuten los aspectos a mejorar de cara al siguiente *sprint*. Cada dos *sprints* se realiza también la *Retrospectiva del Sprint*, donde se decide la refactorización de componentes.

3.4. Otros aspectos metodológicos

Esta sección recoge los aspectos metodológicos no relacionados con Scrum. Dado que esta metodología es flexible en términos de documentación, se decide emplear una serie de convenciones de nombrado para etiquetar las *releases* del sistema, y los errores que surgen del desarrollo de éste.

Estos aspectos se recogen en el [ANEXO D. CONVENCIONES DE ETIQUETADO](#).

3.5. Análisis y fases previas

Como se ha explicado en la sección anterior, el desarrollo del proyecto se realiza a través de *sprints*. No obstante, existen una serie de tareas que, por su importancia, quedan fuera del alcance de éstos. Son las siguientes:

- La adquisición de **conocimiento** y la realización del **trabajo técnico** necesario para acotar el dominio del sistema.
- El prototipado de la aplicación web con la que interaccionará el usuario final, empleando **mockups**. En este prototipo, se detalla la estructura visual de la aplicación, las características de la información que se mostrará, y las interacciones con el usuario.
- Un trabajo de investigación, en el que se recopilan las **fuentes de información** necesarias para obtener los datos requeridos por la aplicación.
- El establecimiento de las **tecnologías** y herramientas con las que se acometerá el proyecto.
- La definición de los **requisitos** principales y la **arquitectura** del sistema. Esta fase es de vital importancia, puesto que una arquitectura bien definida proporciona una base sólida sobre la cual realizar el diseño e implementación del sistema.

En esta sección se han avanzado las tareas previas a la acometida de los *sprints*, dado que se realizan en consonancia a las buenas prácticas de las metodologías ágiles. No obstante, estos aspectos se profundizan en el siguiente capítulo, [4. ANÁLISIS. TAREAS PREVIAS](#).

4. ANÁLISIS. TAREAS PREVIAS

En este capítulo se tratan los pasos previos a la acometida de los *sprints*. Estas tareas son fundamentales para determinar el *Product Backlog*, las tecnologías y herramientas, y la arquitectura del sistema.

El lector puede encontrar un avance más detallado de los contenidos de este capítulo en la página anterior, en la sección [3.5. Análisis y fases previas](#).

4.1. Adquisición de conocimiento y trabajo técnico

En esta primera fase se realiza una labor de búsqueda de información sobre la historia del clima, el cambio climático y fuentes de datos acerca estos temas.

Se analizan herramientas y aplicaciones que proporcionen funcionalidades similares o relacionadas, y se identifican elementos potenciales a incluir en la aplicación.

Además, el alumno consulta bibliografía para aprender buenas prácticas de programación en el lenguaje Python. Esto último se realiza una vez se han elegido las tecnologías con las que se va a implementar el sistema, si bien también se considera conocimiento técnico, por lo que se sitúa en esta sección.

4.2. Elaboración del prototipo

Esta etapa constituye la primera labor de ingeniería. En ella, se diseña un prototipo no funcional empleando *mockups*. Con él, se puede acotar el dominio del sistema, y tener una idea clara de los datos que debe mostrar la aplicación web.

Además, los *mockups* sirven como base de cara a la implementación del sistema, por lo que en etapas más tardías del desarrollo se reduce la pérdida de tiempo en labores de diseño web.

Asimismo, los *mockups* actúan como **pruebas de aceptación**, ya que el *Product Owner* los ha validado anteriormente, y la aplicación implementada debe cumplir las funcionalidades y diseño descritos por el prototipo.

Los *mockups* de la aplicación web se incluyen en el CD que se adjunta con la versión escrita de esta memoria. Para mayor información, el lector puede acudir al [ANEXO E. CONTENIDO DEL CD](#).

La siguiente ilustración permite visualizar la similitud entre el diseño original y la implementación final de la aplicación web:

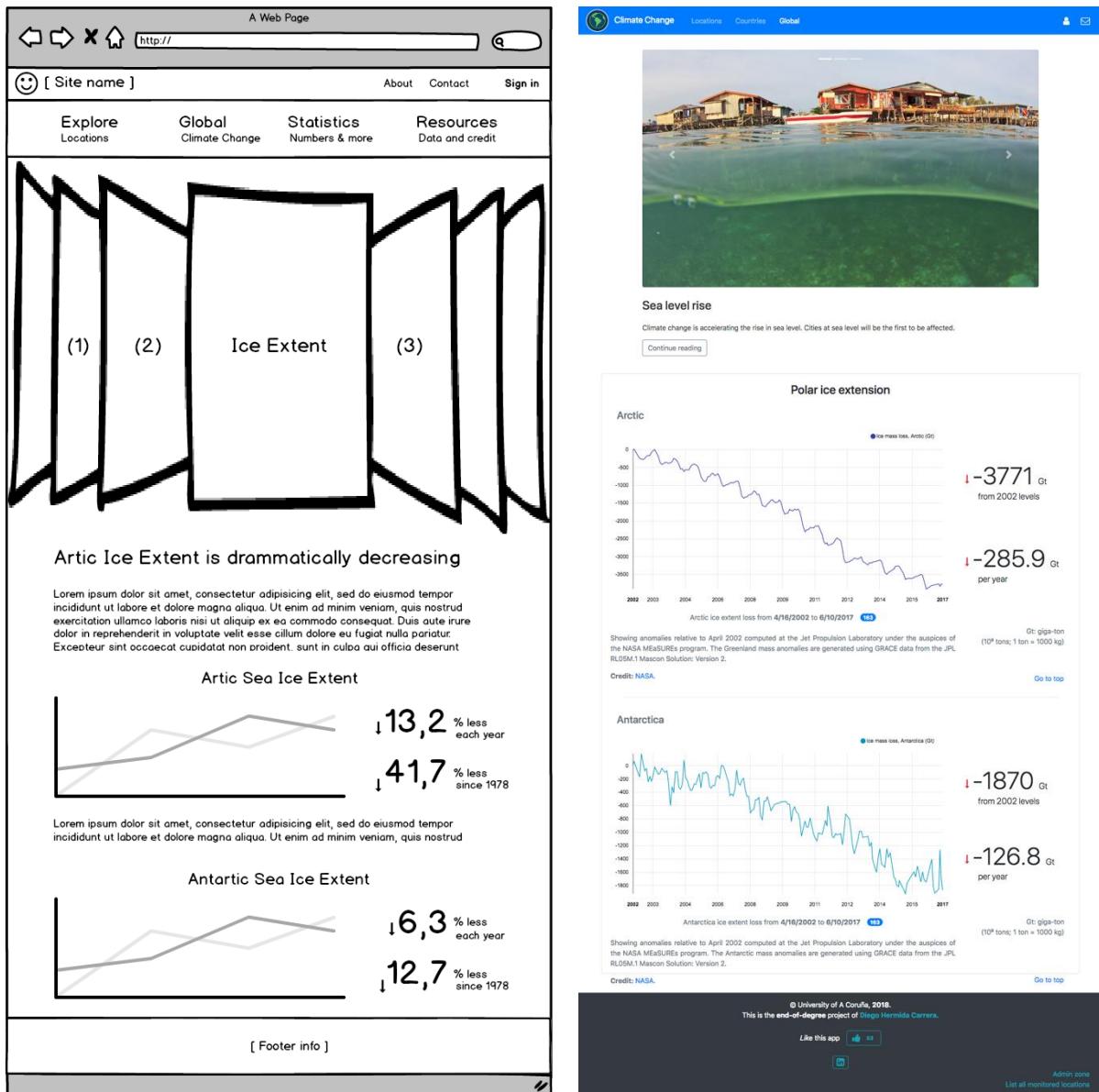


Ilustración 3: Comparativa entre el prototipo y la aplicación web implementada. Se puede apreciar que la estructura es prácticamente idéntica.

4.3. Recopilación de fuentes de información

Esta labor se realiza con la intención de identificar fuentes de datos de terceros.

Se analizan los protocolos que emplean para proporcionar la información, las licencias de uso que éstos poseen, la frecuencia de actualización de los datos, y otras restricciones, como el número máximo de peticiones por minuto admitidas.

Con todo, se obtiene un conjunto de fuentes de datos, e meta-information sobre cada una de ellas. Esto permite al desarrollador conocer las particularidades de cada fuente, de cara a la implementación del sistema.

El documento completo donde se recogen dichos detalles se incluye en el CD que se adjunta con la versión escrita de esta memoria. Para mayor información, el lector puede acudir al [ANEXO E. CONTENIDO DEL CD](#).

4.4. Especificación de requisitos

A la hora de especificar los requisitos de un sistema debe tenerse clara la diferencia entre *requerimiento* y *requisito*, como se vio en la asignatura *Enxeñaría de Requisitos*¹¹. Teniendo en cuenta que el rol de *Product Owner* es ejercido por el director del proyecto, las condiciones que ha de cumplir el sistema se obtienen directamente en forma de requisitos (no existen necesidades especificadas que no sean factibles).

Es necesario, además, conocer la diferencia entre requisitos **no funcionales** y **funcionales**¹². Dado que para acometer el proyecto se sigue una metodología ágil, en este momento solamente se obtienen los requisitos más importantes del sistema. El resto irán surgiendo a lo largo de los *sprints*, de acuerdo con las buenas prácticas de este tipo de metodologías de desarrollo.

4.4.1. Requisitos no funcionales

Los requisitos no funcionales constituyen restricciones de entorno y negocio que afectan a todo el sistema: de no verificarse, el sistema podría resultar inoperativo.

Existen diversos tipos de requisitos no funcionales: de producto, de la organización, externos, etc.

A continuación, se especificarán y justificarán los requisitos no funcionales del sistema, en una tabla:

¹¹ **requerimiento vs requisito:** Los requerimientos son necesidades especificadas por el cliente. Un requerimiento pasa a ser requisito si éste puede satisfacerse. No serían requisitos aquellos requerimientos que constituyen pretensiones de un actor concreto, o aquellos que no sea factible implementar.

La especificación de requisitos software (ERS) supone la base del contrato con el cliente.

¹² **requisito no funcional vs requisito funcional:** Los requisitos *funcionales* definen las características de funcionamiento del sistema (i.e. lo que debe hacer), mientras que los requisitos *no funcionales* constituyen restricciones y condiciones que el sistema en conjunto debe cumplir (i.e. cómo lo debe hacer).

Clave	Requisito	Justificación
RNF-01	Disponibilidad	El sistema debe estar operativo de forma permanente, haciendo hincapié en la recopilación de datos sin interrupciones.
RNF-02	Modularidad	Dado que las fuentes de datos son heterogéneas y susceptibles al cambio, debe minimizarse el acoplamiento del sistema.
RNF-03	Simplicidad	De cara al usuario final, la información visualizada ha de ser sencilla y fácil de comprender. Deberá favorecerse el uso de gráficos e imágenes sobre texto plano.
RNF-04	Usabilidad	La interfaz ha de contar con un diseño <i>Mobile-First</i> , y estar disponible en inglés y castellano.
RNF-05	Rendimiento	Debe minimizarse la complejidad del sistema, y favorecer el rendimiento. Debe garantizarse la ejecución de los módulos que recopilen información con la periodicidad adecuada.
RNF-06	Escalabilidad	El sistema ha de ser escalable, tanto a nivel de aplicación web, como de recopilación de información (sobre todo, de cara a la aparición de nuevas fuentes de datos).
RNF-07	Fiabilidad	Debe minimizarse tanto la probabilidad de fallo como el número de fallos, de cara a ofrecer las funcionalidades del sistema de la manera necesaria.
RNF-08	Seguridad	La información deberá viajar segura por la red, con especial énfasis si se trata de información como los datos personales o las coordenadas geográficas.
RNF-09	Legalidad	La información mostrada por la aplicación web deberá atribuir el origen del que procede la misma en la manera en la que lo exijan las licencias de uso de dichas fuentes de datos.

Tabla 2: Especificación de requisitos no funcionales del sistema.

4.4.2. Requisitos funcionales

Dado que se utiliza la metodología Scrum, los requisitos funcionales –*user stories* (historias de usuario)– se capturan en una lista, el *Product Backlog*. Éste posee la siguiente apariencia:

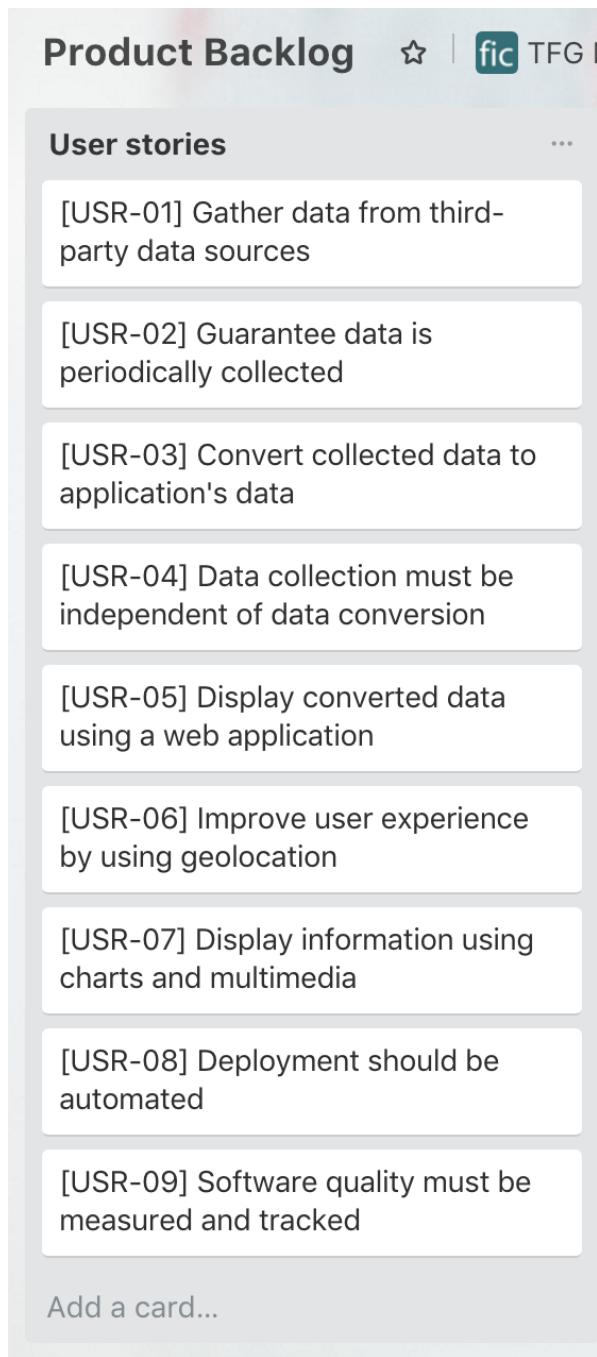


Ilustración 4: Contenido inicial del Product Backlog para el proyecto. Los requisitos o user stories están ordenados según su prioridad.

4.5. Definición de la arquitectura del sistema

La arquitectura de un sistema es un aspecto clave del mismo. Las buenas prácticas de arquitectura, según lo visto en la asignatura *Arquitectura del Software*, indican que “*en los ciclos de vida iterativo-incrementales, la elección de la arquitectura debe quedar fuera de las iteraciones; debiendo decidirse de forma sensata al principio, aunque pudiendo no estar del todo detallada*”.

La representación de la arquitectura se hace a través de la combinación de **vistas**. De cara a ofrecer una perspectiva general de la arquitectura del sistema, que sirva como referencia a lo largo del proyecto, se realiza una vista **física**, donde se pueden apreciar:

- La división del sistema en subsistemas.
- Los flujos de información e interacciones entre componentes.
- Los componentes físicos necesarios para el despliegue del sistema. Con todo, un mismo nodo podrá albergar más de un componente del sistema.

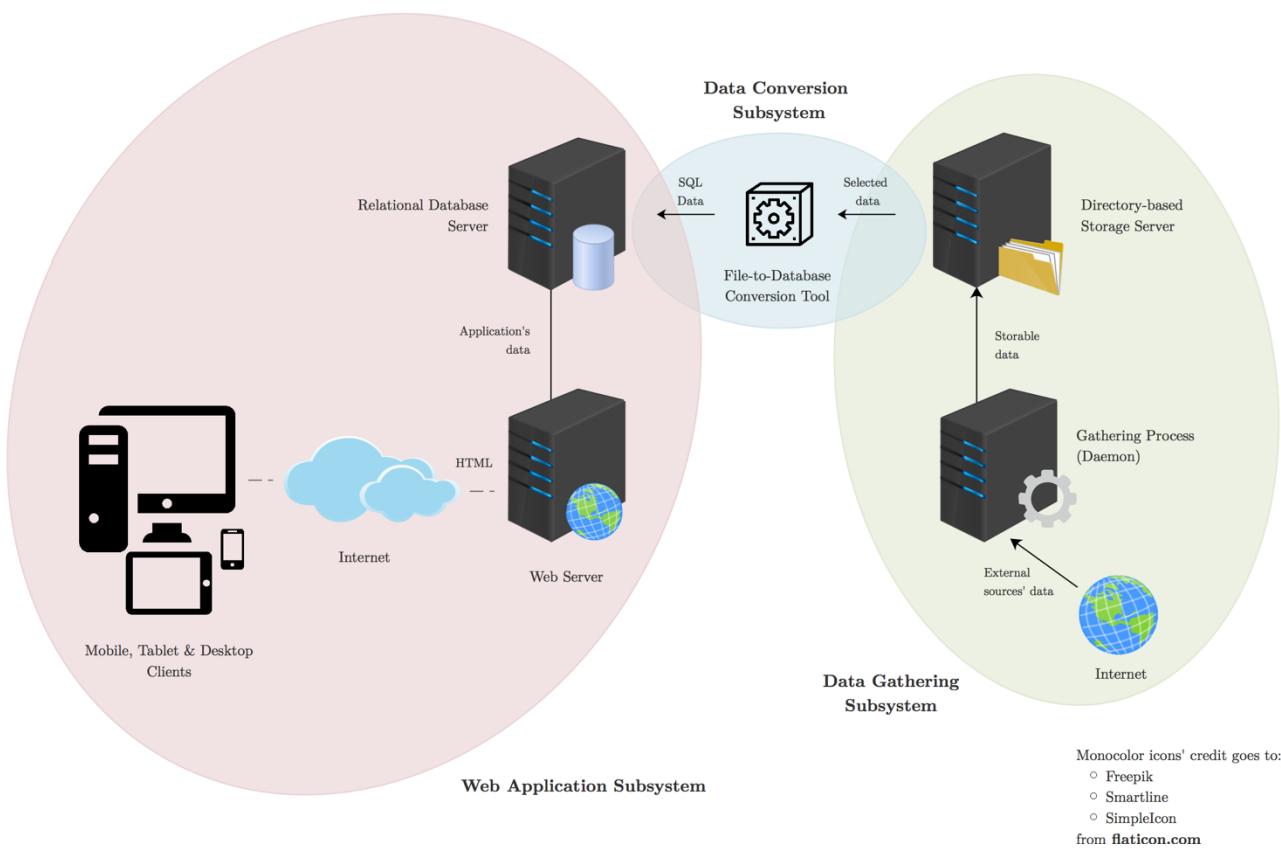


Ilustración 5: Vista física de la arquitectura del sistema.

4.5.1. Descripción y justificación de la arquitectura

El hecho más representativo es la separación del sistema en tres **subsistemas**:

- *Subsistema de Recolección de Datos*: Es el encargado de obtener datos de las fuentes de terceros y almacenarla.
- *Subsistema de Conversión de Datos*: Su función es transformar los datos recopilados por el subsistema anterior en información compatible con la aplicación web.
- *Subsistema de la Aplicación Web*: Se encarga de mostrar la información al usuario final.

Esta decisión se justifica teniendo en cuenta los requisitos no funcionales y funcionales; concretamente:

- El requisito no funcional RNF-01 *Disponibilidad*. Se requiere que la colección de datos se realice sin interrupciones.
- El requisito funcional USR-04, que obliga a separar la recolección de datos de la conversión de los mismos.

La división en subsistemas, entre otros:

- Garantiza que los fallos en un componente no afectarán al resto, lo que contribuye a la disponibilidad y fiabilidad del sistema.
- Permite desplegar el sistema de forma distribuida.
- Cumple con el *Principio de Responsabilidad Única*¹³, aplicado aquí a componentes.
- Maximiza la cohesión de los módulos y reduce el acoplamiento: cada subsistema es un componente estanco, que se comunica con el resto a través de puertos e interfaces nítidas. Esto puede apreciarse a continuación:

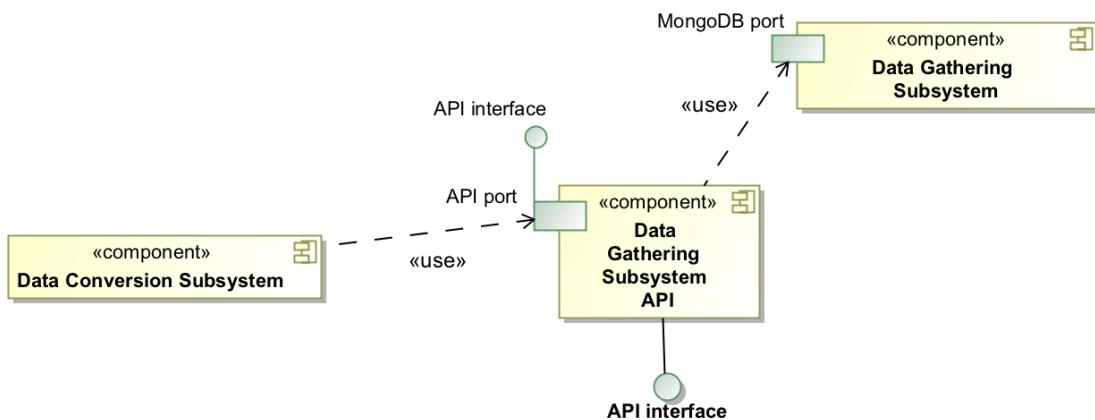


Ilustración 6: Diagrama de componentes en el que se muestra el mecanismo de comunicación entre el Subsistema de Recolección de Datos y el Subsistema de Conversión de Datos a través de un API, que exporta una interfaz bien definida.

¹³ **Principio de Responsabilidad Única**: Este principio indica que cada clase o módulo debe tener un único motivo para cambiar. En otras palabras, una unidad debe servir a un único fin.

4.5.2. Diseño del Subsistema de Recolección de Datos

En la [Ilustración 4](#), se muestra el *Product Backlog* del proyecto, con los requisitos ordenados según su prioridad.

El primer requisito funcional –*USR-01*– consiste en la *Recopilación de datos de fuentes de terceros*. Esto implica que el primer esfuerzo del *Equipo* debe centrarse en este requisito, de acuerdo con las buenas prácticas de las metodologías ágiles.

La justificación de la prioridad del requisito citado anteriormente recae en la necesidad de disponer de un nutrido conjunto de datos (varios *gigabytes*); y tras el estudio de las fuentes de datos que proporcionarán la información, el equipo se da cuenta de que son necesarios varios meses para conseguir el volumen de datos requeridos. Por lo tanto, es fundamental tener este subsistema operativo lo antes posible.

De cara a agilizar el desarrollo, la definición de la arquitectura del *Subsistema de Recolección de Datos* queda también fuera de la acometida de los *sprints*.

Teniendo en cuenta los requisitos:

- RNF-02 *Modularidad*, y
- USR-02 *Garantizar que la recopilación de datos se ejecuta de forma periódica*,

se decide basar el subsistema en la colaboración de:

- Un componente de *repetición periódica*, que será el encargado del disparo de eventos que inicien el subsistema.
- Un componente *principal* que invoque los módulos del sistema en el orden apropiado.
- Un componente *supervisor*, activo durante toda la ejecución del subsistema, que monitorice y determine si una ejecución es satisfactoria, y tome medidas en caso contrario.
- Un conjunto de componentes *recolector*. Cada uno de ellos recopilará información de una fuente de datos de terceros, de forma aislada y concurrente. Su comportamiento será similar, por lo que se decide que partirán de una estructura común con un comportamiento análogo a una **máquina de estados finitos**¹⁴. Esto permite garantizar que, pese a los errores, los módulos alcanzarán un estado consistente.
- Dos componentes auxiliares:
 - Uno encargado de importar y cargar todos los *recolectores*, y

¹⁴ **máquina de estados finitos**: sistema que cuenta con un conjunto de estados *finito*, un conjunto de reglas de transición (que definen la navegabilidad entre estados), y una serie de acciones (que disparan los cambios de estado).

- Otro que registre todos los eventos de cada ejecución; esto es, un *logger*.

El siguiente diagrama de secuencia muestra la arquitectura del subsistema enfatizando la **interacción** entre sus componentes:

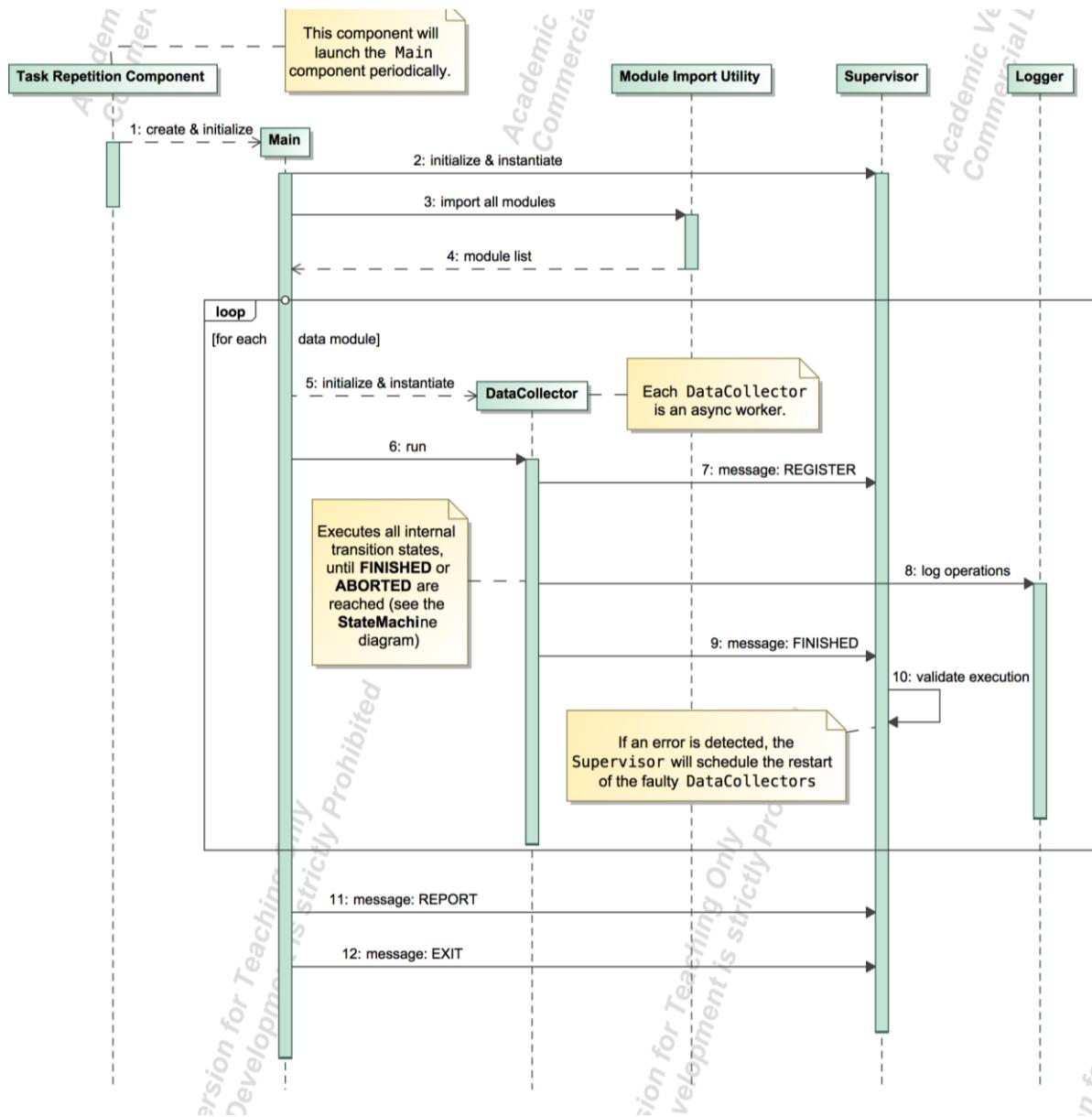


Ilustración 7: Diagrama de secuencia que muestra la interacción entre los componentes del Subsistema de Recolección de Datos.

El funcionamiento concreto del subsistema se explicará en los capítulos [6. PRIMER SPRINT](#), y [7. SEGUNDO SPRINT](#), dado que éste se centra exclusivamente en el la fase de análisis, previa al desarrollo de los *sprints*.

5. TECNOLOGÍA Y HERRAMIENTAS

En este capítulo se describirán y se justificarán las tecnologías y herramientas empleadas para el desarrollo del proyecto, cada una en su propia sección.

Esta elección se realiza una vez avanzada la fase de análisis del proyecto, detallada en el capítulo anterior, [4. ANÁLISIS. TAREAS PREVIAS](#).

5.1. PyPy

PyPy es una implementación de Python. Éste se encuentra disponible en dos versiones: 2.x y 3.x; si bien la versión 2.x dejará de tener soporte en un futuro no muy lejano, y se aconseja a los desarrolladores que empleen la versión 3.x en futuros proyectos. Haciendo uso de esta recomendación, se utiliza la versión 3.5 de PyPy, implementada sobre Python 3.5.3.

Teniendo en cuenta que Python es un lenguaje interpretado, y dada su gran flexibilidad y alto nivel de abstracción, pueden darse casos en los que el rendimiento se vea comprometido. Ahí es donde entra en juego PyPy.

PyPy emplea compilación Just-In-Time (JIT), por lo que los programas a menudo corren más rápido que en Python. Es eficiente en términos de manejo de memoria, y compatible con la mayoría de *frameworks* y librerías del Python estándar.

Además, el hecho de que el alumno solamente conocía Python a un nivel básico, también constituye una oportunidad para dominar un nuevo lenguaje de programación.

5.2. PyCharm

PyCharm es un IDE para trabajar en proyectos programados en Python. Está desarrollado por JetBrains (los creadores de IntelliJ IDEA).

Anteriormente, para trabajar en Python, el alumno ha empleado el editor de textos Atom. No obstante, dadas las características del proyecto, y después de haber estudiado las ventajas de los IDEs en la asignatura *Ferramentas de Desenvolvimento*, se ha decidido emplear una herramienta que cuente con todas esas características deseadas.

Después de leer varias comparativas, se decidió usar PyCharm porque es un IDE específico para Python; con un diseño atractivo, un *debugger*¹⁵ fácil de manejar y

¹⁵ **debugger**: herramienta que permite analizar el código de forma dinámica. Una práctica común para hacer *debug* (depuración) sobre el código es el establecer *breakpoints* (puntos donde para la

muy potente. Como otros IDEs (e.g. Eclipse), es orientado a *plugins* (y, por ende, extensible). Además, proporciona autocompletado, formateo de código, integración con sistemas de control de versiones (e.g. Git, Subversion), integración con Docker y sistemas de Integración Continua, gestión de tareas, refactorización, etc.

Asimismo, JetBrains proporciona acceso a la versión Professional de forma gratuita a estudiantes universitarios, por lo que el uso del IDE no supone un coste adicional al proyecto.

5.3. MongoDB

Al diseñar la arquitectura del sistema, el equipo se da cuenta de que, para almacenar datos de fuentes heterogéneas, un sistema gestor de bases de datos (SGBD) relacional no conformaría la solución óptima. La justificación de la elección de un SGBD noSQL¹⁶, y la elección de MongoDB se hace en base a que:

- Todas las APIs proporcionan datos en formato JSON. Esto posee las siguientes implicaciones:
 - a) Se precisa un sistema *schema-less* (sin esquema), ya que las fuentes de datos son susceptibles a cambios en el futuro, y esto conduciría a problemas en un SGBD relacional.
 - b) MongoDB almacena datos en forma de documentos BSON, que no es más que la serialización en formato binario de documentos JSON.
 - c) Python proporciona soporte nativo para trabajar con documentos JSON mediante la librería `json`.
- No se requieren las características ACID¹⁷ para almacenar los datos, y un SGBD relacional supondría una penalización de rendimiento innecesaria.

ejecución) y así poder analizar los valores de las variables en dicho punto. Pueden establecerse *breakpoints* condicionales, donde solo se parará la ejecución si se verifica una condición, etc.

¹⁶ **noSQL**: Los SGBD noSQL son bases de datos no relacionales optimizadas para modelos de datos *schema-less* y escalables. Emplean diversos modelos de datos: almacenamiento mediante pares claves/valor, documentos, columnas, etc. Son ideales para el desarrollo de aplicaciones web, *big data*, etc. Son más eficientes que los SGBD relacionales en un número limitado de situaciones. Por lo tanto, de cara a escoger un SGBD debe hacerse un estudio previo, ya que tanto los SGBD relacionales como los noSQL poseen fortalezas y debilidades.

¹⁷ **ACID**: Siglas de *Atomicity*, *Consistency*, *Isolation* y *Durability* (Atomicidad, Consistencia, Aislamiento y Durabilidad). Son características que todo SGBD relacional ha de cumplir para garantizar transaccionabilidad.

5.4. PyMongo

PyMongo es un API para trabajar con MongoDB en Python.

Una de las características más interesantes de PyMongo es que proporciona traducción automática del formato BSON a objetos Python. Además, da soporte a prácticamente todas las operaciones ofrecidas por MongoDB: *aggregation framework*, *bulk-write*, autenticación, SSL, etc.

5.5. Git

Se opta por emplear Git como sistema de control de versiones (VCS) debido, nuevamente, a la asignatura de *Ferramentas de Desenvolvimento*.

Git permite hacer *commit* en local, por lo que casa bien con la filosofía “cada *commit* debe representar una única tarea” (en SVN es necesario disponer de conexión para realizar un *commit*). Además, posee otras características como el uso de *branches* (ramas) para trabajar en paralelo, etc. No obstante, dado que el proyecto es desarrollado por un *Equipo* de un miembro, no se le sacará partido a esta característica. Con todo, supone una mejora respecto a SVN.

Se emplea GitHub para almacenar el código del proyecto.

5.6. Docker

Docker permite separar las aplicaciones de la infraestructura sobre la que se ejecutan. Esto es, proporciona una capa de abstracción que hace posible que una aplicación funcione en Windows, Linux y macOS sin problemas.

Se basa en el uso de **imágenes** (producto software construido) para generar **contenedores** (instancias de una imagen). Estos últimos son ejecutables.

Para generar una imagen, se parte de otra imagen existente y se declara un fichero especial **Dockerfile**, en el que se incluyen los pasos necesarios para su construcción. Estos ficheros permiten instalar dependencias de librerías de terceros al generar imágenes, por lo que no es necesaria su instalación en la máquina.

Docker proporciona, entre otros:

- *Seguridad*: cada contenedor se ejecuta de forma independiente, si bien los contenedores pueden comunicarse entre sí a través de puertos. No obstante, pueden aislarse de Internet, o ser visibles sólo por ciertas direcciones IP o contenedores.

- *Escalabilidad*: Se pueden replicar servicios indicándole a Docker cuántas instancias se requieren. Además, estas pueden colaborar en modo *Swarm* (enjambre).
- *SaaS (Software como Servicio)*: Existe una comunidad que proporciona y mantiene imágenes de su software (e.g. MongoDB, MySQL, Jenkins...). Esto permite desplegar servicios en un tiempo muy bajo, permitiendo al desarrollador enfocarse en lo importante: desarrollar su sistema.

El uso de Docker es ideal para este proyecto, ya que la plataforma de desarrollo es macOS, mientras que el sistema se despliega sobre un Ubuntu Server.

5.7. PostgreSQL

Si bien se usa MongoDB para almacenar los datos procedentes de fuentes de datos de terceros, se emplea un SGBD relacional para guardar la información una vez ha sido procesada y está lista para ser visualizada por el usuario.

Las decisiones arquitectónicas no se discutirán en este capítulo. No obstante, la elección de PostgreSQL sobre cualquier otro SGBD relacional se justifica porque Django, el *framework* web empleado en la aplicación, está optimizado para PostgreSQL, si bien es compatible con otros SGBD (e.g. MySQL).

5.8. Django

Django es un *framework* muy amplio. Proporciona utilidades que van desde el envío de e-mails, hasta funcionalidades de ORM¹⁸, pasando por todo lo necesario para implementar aplicaciones web.

Se le da uso de forma extensiva a lo largo del desarrollo de la aplicación web, y del Subsistema de Conversión de Datos.

5.9. Jenkins y SonarQube

Estas herramientas permiten mejorar la calidad del producto software, mediante la adopción de las disciplinas de Integración Continua e Inspección Continua, estudiadas en la asignatura *Ferramentas de Desenvolvimento*.

¹⁸ **ORM:** Siglas de *Object-Relational Mapping* (mapeo Objeto-Relacional). Los ORM permiten transformar objetos en filas de tablas dentro de un SGBD relacional. Además, facilitan la implementación de operaciones básicas sobre entidades, sin tener que hacer uso directo de SQL.

Dado que estas herramientas requieren de un proceso de configuración no trivial, se explica todo lo relacionado con las disciplinas mencionadas anteriormente junto con estas herramientas en el [ANEXO B. INTEGRACIÓN E INSPECCIÓN CONTINUA](#).

5.10. HTML5, CSS3, jQuery, JavaScript y AJAX

Se hará uso de estas tecnologías a la hora de implementar la aplicación web.

HTML5 conforma la última versión de HTML, y se siguen sus buenas prácticas en todo momento. CSS3 se emplea para proporcionar modificaciones sobre estilos. jQuery, AJAX y JavaScript se emplean para interactuar con el usuario desde el lado cliente, permitiendo mostrar animaciones, gráficos, etc. que de otra forma requerirían procesamiento del lado servidor. Asimismo, dotan a la aplicación de mayor dinamismo y usabilidad.

5.11. Bootstrap 4

Se hace uso de Bootstrap como *framework* para implementar páginas web ligeras, *Mobile-First*¹⁹ y visualmente atractivas para el usuario.

Bootstrap 4 incorpora numerosas clases CSS que permiten dotar a los elementos HTML de estilos sin tener que añadir el atributo **style**, haciéndolos más mantenibles.

Además, incluye gran cantidad de componentes (e.g. **alert**, **card**, **scrollspy...**) que ejecutan código JavaScript de forma automática, haciendo que los sitios web implementados con este *framework* sean más dinámicos y visuales, sin suponer un esfuerzo para el desarrollador.

5.12. Trello

Como soporte a la planificación y gestión del proyecto se emplea la herramienta Trello. Ésta se integra perfectamente con las metodologías ágiles, ya que hace uso de sus artefactos:

- El progreso de los *sprints* se almacena en **tableros**. Cada tablero representa un *sprint*, aunque también se emplearán para:

¹⁹ **Mobile-First**: Es la filosofía opuesta al diseño web *Responsive*. En el primero, a menor resolución, menor tamaño del contenido y supone la reorganización de los componentes.

Un diseño *Mobile-First* optimiza el contenido para móviles, y lo adapta para el resto de resoluciones y tamaños de pantalla. Esta tendencia está en auge, debido a que cada vez se accede más a los sitios web desde los dispositivos móviles.

- La imputación de horas dedicadas a la adquisición de conocimiento y trabajo técnico anteriores a los *sprints*.
- Albergar la evolución del *Product Backlog*.
- Registrar los *bugs* (fallos) del sistema, y planificar su solución.
- En los tableros se pueden crear **listas** que contienen en cada momento las tareas pendientes, en curso y finalizadas: *To do*, *Doing*, *Done*.
- Las tareas del *sprint* tienen forma de **tarjeta**. Adicionalmente, Trello soporta el uso de *plugins* que añaden funcionalidades. Se empleará el *plugin* “Scrum by Vince”, que permite realizar estimaciones de tareas, imputar tiempos y visualizar el progreso del *sprint*, empleando para ello el gráfico *Sprint Burndown*:

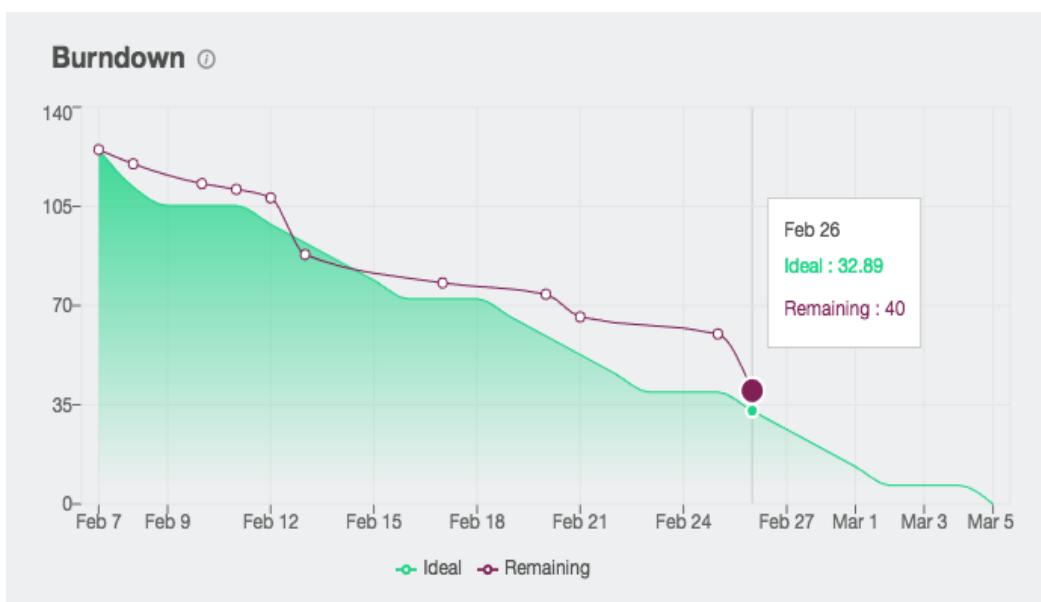


Gráfico 3: Ejemplo de gráfico Sprint Burndown generado con la herramienta Trello y el plugin “Scrum by Vince”. El gráfico pertenece al sprint 3 del proyecto.

5.13. Otras tecnologías

Se emplean otras tecnologías, como:

- **Balsamiq Mockups 3**, para el desarrollo del prototipo de la aplicación web.
- **Eve** y **Flask**, para desarrollar un API. Esto se detallará en la sección [8.2. Componente API](#).
- **amCharts** y **nvd3**, como librerías JavaScript para visualización de gráficos y mapas.

- **Google Maps**, para mostrar las localizaciones monitorizadas en la aplicación web.
- **Bash**, como lenguaje de los *scripts* de instalación del sistema.
- **CRON**, permite dar soporte a la ejecución de tareas periódicas.
- **YAML**, como formato de serialización de propiedades y configuración del sistema.
- **Magic Draw**, para elaborar diagramas UML.

6. PRIMER *SPRINT*

Este capítulo se centra en las tareas desarrolladas durante el primer *sprint*, el cual comienza el 01/10/2017 y finaliza el 26/12/2017, con un esfuerzo estimado de 108 horas*hombre.

Los esfuerzos del *Equipo* en este *sprint* se centran en satisfacer el siguiente requisito funcional o *user story*:

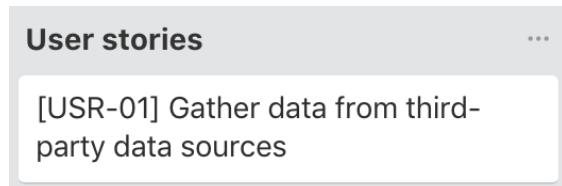


Ilustración 8: Primer user story del Product Backlog, “Recopilación de datos de fuentes de terceros”.

El *Sprint Backlog* (lista de tareas del *sprint*) contiene un total de 15 tareas.

A continuación, se detallarán los aspectos de diseño e implementación desarrollados durante la acometida de este *sprint*, cada uno en una sección.

6.1. Componentes *recolector*

Este conjunto de componentes conforma el núcleo del subsistema, ya que son los encargados de conseguir y almacenar la información de fuentes de datos de terceros.

Como ya se adelantó en el capítulo anterior, estos componentes –de aquí en adelante, **DataCollectors**– poseen un comportamiento similar y partirán de una estructura común, la cual se describe a continuación.

6.1.1. La clase base **DataCollector**

Esta clase conforma el punto de partida de cualquier componente *recolector*. Está diseñada para **facilitar** enormemente el desarrollo de nuevos componentes, dado que esta clase se encarga, de forma transparente:

- Del manejo automático de errores.
- Del *logging* de operaciones.
- De realizar las transiciones entre estados.
- De la persistencia de variables de estado.

Esto permite que el desarrollador se encargue exclusivamente de implementar los métodos que **recopilan y guardan** la información. El resto de aspectos son automáticamente manejados por la clase base **DataCollector**.

Mediante el mecanismo de la **herencia**, cualquier clase implementadora que parta de la clase base **DataCollector** tendrá acceso a sus operaciones y estructuras. El siguiente diagrama de clases permite visualizar el API que exporta la clase base **DataCollector**:

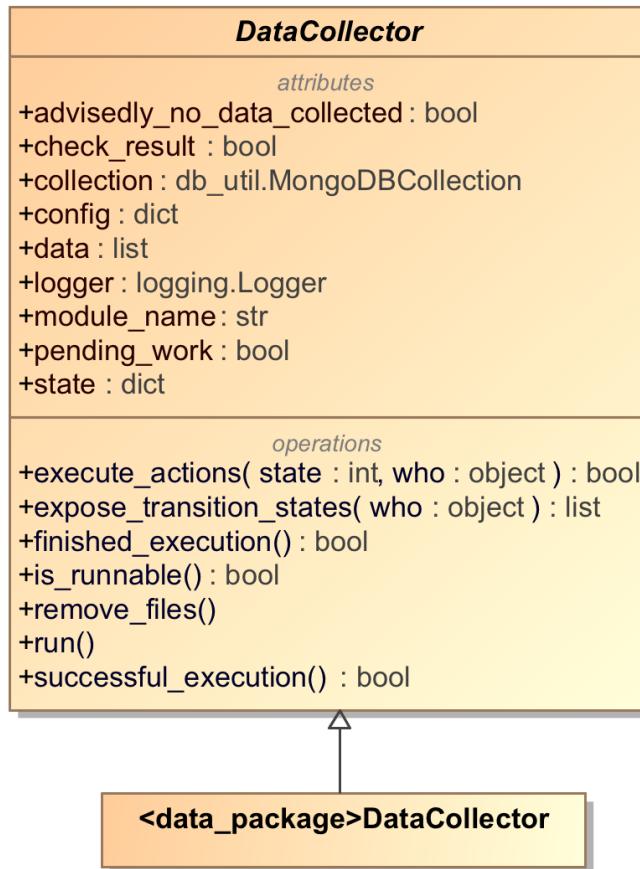


Ilustración 9: API ofrecida por la clase base DataCollector.

De todos los métodos que proporciona la clase base, el más importante y el que permite la ejecución de los **DataCollectors**, es el método **run**. La implementación interna de la clase base, de forma transparente, ejecuta los métodos necesarios para terminar la ejecución del componente en el orden adecuado.

6.1.2. Máquina de estados finitos

Una forma sencilla y elegante de garantizar la consistencia de los **DataCollectors** es hacer que la clase base se comporte como una máquina de estados finitos.

Para implementar dicho comportamiento, se emplea la clase `TransitionState`, que representa un estado de transición:

```
class TransitionState:  
    """  
        Represents a valid state within the set of StateMachine's states.  
        Supports equality, comparisons and String representation.  
    """  
  
    def __init__(self, name, code, next_state, actions: callable):  
        """  
            Initializes an state.  
            :param name: State's name (to make it more readable).  
            :param code: An unique, numerical identifier (to make faster  
                        comparisons between states).  
            :param next_state: If actions succeeded, this determines the  
                            subsequent state. Can be empty (None).  
            :param actions: Callable object (i.e. function).  
        """  
        self.name = name  
        self.code = code  
        self.next_state = next_state  
        self.actions = actions
```

Código 1: Definición de la clase `TransitionState`. Por simplicidad sólo se muestra el método constructor.

Como aclaración, el método `__init__` en Python es similar a un constructor en otros lenguajes de programación orientada a objetos.

Un estado de transición contiene:

- Un *nombre*: Una cadena de texto que ofrece una representación semántica del estado de transición.
- Un *código*: Consiste en un identificador numérico que permite realizar comparaciones menos costosas entre los estados de transición.
- Unas *acciones* a ejecutar: Una referencia a un método o función²⁰.
- Un *estado siguiente*: Una referencia a otro objeto `TransitionState`, que representa el estado siguiente. Puede ser `None` (nulo), si el estado se considera terminal.

Nótese que la implementación de la clase no permite múltiples estados de transición como estado siguiente al actual. Dado que no se le daría uso a esa característica se omite su implementación, de acuerdo a los principios YAGNI y KISS²¹.

²⁰ en Python, métodos y funciones son objetos. De hecho, toda clase desciende de `object` (incluso los *tipos básicos* de otros lenguajes de programación, como `int`, `str`, etc.)

²¹ **YAGNI** y **KISS**: son principios de diseño software, que abogan por añadir sólo funcionalidades necesarias y mantener el código lo más simple posible, respectivamente.

Los estados de transición se albergan en una **tabla de transición**, representada internamente como simples atributos de clase.

El siguiente diagrama de máquina de estados muestra el conjunto de estados de transición de un **DataCollector**, así como las acciones que desencadenan las transiciones entre éstos:

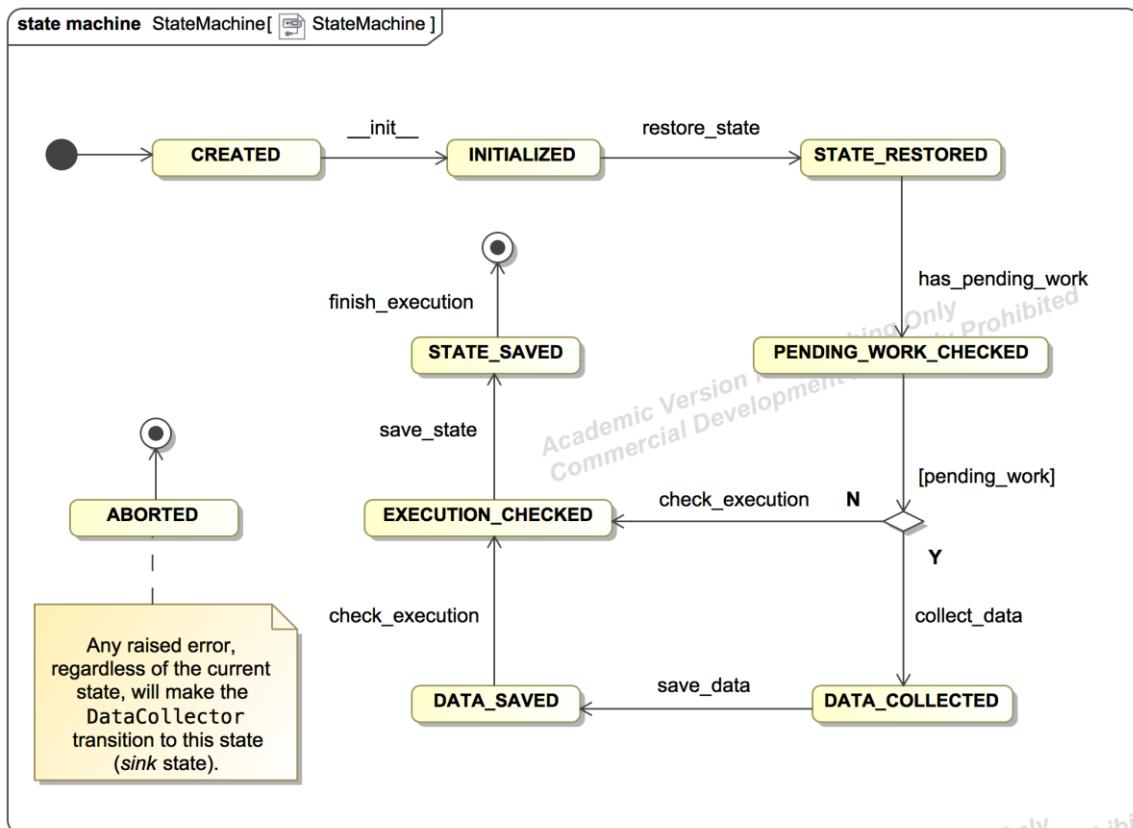


Ilustración 10: Estados de transición de un **DataCollector**.

Como se puede apreciar en el diagrama, existen dos estados finales:

- **ABORTED** representa el estado al que llega cualquier **DataCollector** que ha lanzado una excepción. Ésta se gestiona desde la clase base, evitando un error descontrolado y marcando la ejecución como errónea.
- **FINISHED** representa una ejecución satisfactoria. Este estado solamente es alcanzable mediante la ejecución del método *interno*²² `finish_execution`.

²² en Python no existen los conceptos de **permisos** y **visibilidad**, presentes en otros lenguajes de programación (e.g. `private`, `protected`, etc.). Los métodos o atributos internos de clase se marcan con un *underscore* (e.g. `_my_method`), y sólo existe la recomendación general de no usarlos.

Python también permite marcar métodos y atributos con un *double-underscore* (e.g. `__my_method`). Esto responde a un problema de colisión de nombres entre clases, y no está relacionado con permisos ni visibilidad –como mucha gente cree, de forma errónea–. Python traduce estos valores automáticamente para evitar la colisión de nombres (e.g. `_my_method` → `_MyClass__my_method`).

6.1.3. Métodos de un DataCollector

Como se puede apreciar en el diagrama de máquina de estados finitos de la subsección anterior, existen una serie de acciones que dispara las transiciones entre estados.

La clase base posee una implementación por defecto para la mayoría de los métodos, dado que el comportamiento de éstos es común a todos los **DataCollectors**.

El siguiente fragmento de código muestra un extracto de la clase base:

```
class DataCollector(ABC, Runnable):

    def run(self):
        "[...]

    def _restore_state(self):
        "[...]

    def _has_pending_work(self):
        "[..."]

    def _decide_on_pending_work(self):
        "[..."]

    @Before(action=_decide_on_pending_work)
    @abstractmethod
    def _collect_data(self):
        pass

    @abstractmethod
    def _save_data(self):
        "[..."]

    def _check_execution(self):
        "[..."]

    def _save_state(self):
        "[..."]

    def _finish_execution(self):
        pass
```

Código 2: Extracto de la clase DataCollector. En él, se muestran los métodos principales que pueden ser sobrescritos o requieren algún comentario para ser comprendidos.

A continuación, se ofrece una explicación detallada de todos los aspectos necesarios para entender el comportamiento de la clase base:

- Lo primero a destacar es el hecho de que Python, al contrario que otros lenguajes de programación, sí contempla la **herencia múltiple**. Además, el concepto de interfaz no existe.
- En Python, `self` es el equivalente a `this`: una referencia al propio objeto. Al declarar métodos de instancia o de clase, la firma debe incluir `self` o `cls` como primer parámetro, respectivamente (en otros lenguajes de programación, la referencia a `this` se incluye de manera implícita).
- La clase base hereda de `ABC` (permite crear clases abstractas), y de `Runnable`: una clase abstracta con un único método `run` (similar a la interfaz `Runnable` de Java).
- Si el método `__init__` se ejecuta correctamente, el `DataCollector` pasará al estado `INITIALIZED`.
- El siguiente método que se ejecuta es `_restore_state`, que permite al `DataCollector` recuperar el estado de la ejecución anterior (este aspecto se verá más en detalle en la sub-sección 5.1.5 Configuración y Estado). Tras ello, se alcanza el estado `STATE_RESTORED`.
- Una vez se ha deserializado el estado, el `DataCollector` puede comprobar, en base a la información de control presente en éste, si debe recuperar datos o, por el contrario, finalizar su ejecución. El resultado de la comprobación se almacena en el atributo de instancia `pending_work`. El DataCollector transiciona al estado `PENDING_WORK_CHECKED`.
- El siguiente método que se ejecutaría es `_collect_data`. Nótese que el método está decorado con `@Before(action=_decide_on_pending_work)`. Se emplea aquí la *Programación Orientada a Aspectos* (AOP)²³; concretamente, la **intercepción** de métodos. Cuando el método `_collect_data` es invocado, se intercepta la llamada y se comprueba si existe trabajo pendiente. En caso afirmativo, se ejecutará `_collect_data` y, en caso negativo, se pasará directamente a `_check_execution`.
- Si finalmente se ejecuta `_collect_data`, se recopilarán datos. Este método está decorado además con `@abstractmethod`, indicando a las clases que hereden de `DataCollector` que este método no posee una implementación por defecto, sino que debe sobreescibirse (de no hacerlo, se lanzará una excepción en tiempo de ejecución –recordemos que Python es un lenguaje interpretado, por lo que no pueden realizarse comprobaciones en tiempo de

²³ la Programación Orientada a Aspectos (AOP) permite evitar los llamados *cross-cutting-concerns*, o aspectos colaterales al desarrollo que deben gestionarse repetidamente (e.g. transaccionabilidad, el uso de una caché, o la autenticación).

AOP permite pasar de programación *procedimental* a programación *declarativa* (e.g. usar `@Transactional` sobre un método en lugar de manejar la transacción dentro, de forma explícita).

compilación–). Los **DataCollectors** deberán almacenar los datos recopilados en el atributo de instancia **data**. Se alcanza el estado **DATA_COLLECTED**.

- Tras recopilar información, es necesario almacenarla en MongoDB. Esto se realiza en el método **_save_data**, que también está decorado con **@abstractmethod**. No obstante, en este caso sí existe una implementación por defecto, aunque sus detalles se comentarán en la sub-sección [6.1.6. MongoDB](#). El método sobrescrito debe invocar **super()._save_data()** en la primera línea para heredar la implementación por defecto. Con el fin de liberar memoria, una vez se ha almacenado la información en el SGBD, debería *desreferenciarse* el atributo **data**. Tras la ejecución de este método, se alcanza el estado **DATA_SAVED**.
- El siguiente método es **_check_execution**. En cada ejecución se determina si ésta fue satisfactoria o no, y se almacena dicho resultado en el atributo de instancia **check_result**. Se realizan una serie de comprobaciones por defecto (e.g. que si había trabajo pendiente, se hayan recopilado datos y almacenada información; que no hayan ocurrido errores, etc.), aunque podrían realizarse más verificaciones si se sobrescribe este método. También se decide aquí si es necesario reiniciar el **DataCollector** (i.e. si han existido errores, reintentar la ejecución). Tras la comprobación de la ejecución, se alcanza el estado **EXECUTION_CHECKED**.
- El penúltimo paso es la serialización del estado, susceptible al cambio desde la última ejecución. Esto se realiza en el método **_save_state**. Al igual que con el método **_restore_state**, la serialización del estado se verá más en detalle en la sub-sección [6.1.5. Configuración y estado](#). Se alcanza el estado **STATE_SAVED**.
- Por último, se ejecuta el método **_finish_execution**, una *no-op* que permite alcanzar el estado **FINISHED_EXECUTION**.

Nótese que, con la implementación de la clase base, no es necesaria la sobreescritura de ningún método salvo **_collect_data** y **_save_data** (generalmente); lo cual permite al desarrollador centrarse únicamente en estos aspectos, y desarrollar nuevos **DataCollectors** de forma ágil.

Además, la clase base está verificada mediante pruebas de unidad, por lo que es más confiable partir de ésta que reinventar la rueda para cada **DataCollector**. No obstante, toda pieza de software está sujeta a fallos, por lo que nunca se puede garantizar una fiabilidad del 100%.

6.1.4. Estructura modular de los DataCollectors

Un **DataCollector** es más que una clase implementadora de la clase base. Cada clase implementadora se codifica dentro un módulo (fichero con extensión **.py**), que posee la siguiente estructura:

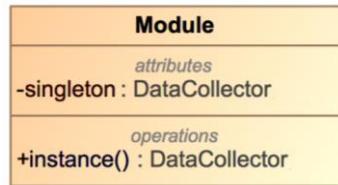


Ilustración 11: Estructura de un módulo del subsistema.

Como puede apreciarse, cada módulo implementa el *Patrón Singleton*²⁴. El método **instance** proporciona un punto de acceso global a una única instancia del **DataCollector** por ejecución, –mediante el atributo **singleton**–.

Así, la implementación de la clase puede permanecer *privada* (“protegida” con un *underscore*), dado que donde se necesite instanciar un **DataCollector** podrá obtener un objeto mediante el método **instance**.

Además del módulo, un **DataCollector** posee los siguientes elementos:

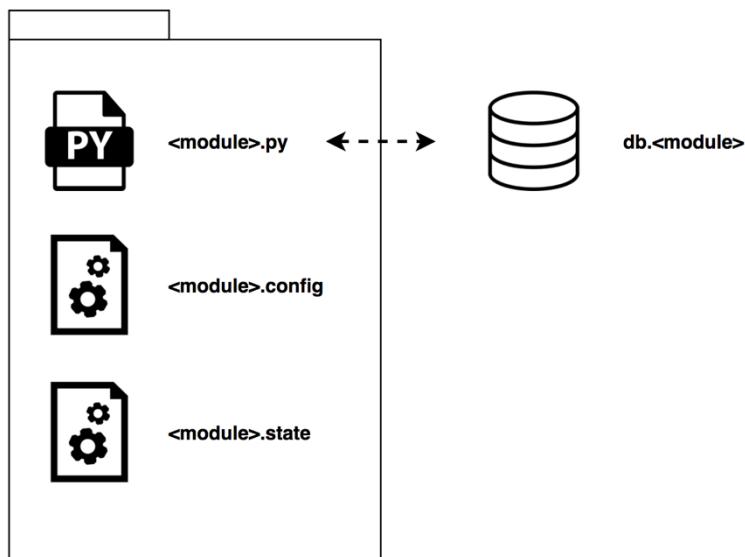


Ilustración 12: Elementos de un DataCollector.

- Un *fichero de configuración* con extensión **.config**, con el mismo nombre que el módulo.

²⁴ **Patrón Singleton:** La implementación de este patrón de diseño software permite restringir la creación de instancias de clase a una única instancia, accesible a través de un punto global (generalmente, un método **get**).

- Un *fichero* con extensión `.state`, con el mismo nombre que el módulo.
- Una *collection* en MongoDB, con el mismo nombre que el módulo.

En la siguientes sub-secciones se describirán estos tres últimos aspectos.

6.1.5. Configuración y estado

La configuración y estado de los `DataCollectors` constituyen aspectos necesarios para la ejecución de los mismos. Esta sub-sección los explicará en detalle.

6.1.5.1. Configuración

La configuración evita almacenar propiedades y constantes *hard-coded* (codificadas en propio código fuente), siendo albergada en ficheros de configuración.

Estos ficheros emplean la sintaxis del estándar de serialización YAML. Junto con la librería PyYAML, la adopción de este formato permite declarar propiedades empleando una sintaxis rica, cercana al lenguaje natural; y convertir éstas a objetos Python en tiempo de ejecución.

```
URL: https://api.foo.com/token={TOKEN}

# Tokens used for accessing the API.
TOKENS:
  - 1234567890abcdefg
  - 0987654321hijklmn

MAX_REQUESTS_PER_TOKEN_AND_MINUTE: 50
```

Código 3: Ejemplo de fichero de configuración de un módulo del subsistema.

Cada `DataCollector` posee su propio fichero de configuración, que debe estar situado en el mismo directorio que el módulo, y poseer el mismo nombre que éste. Al instanciar la clase, la configuración se recupera de forma automática durante la ejecución del método `__init__`.

Desde que el `DataCollector` ha sido instanciado, su configuración es accesible mediante el atributo de instancia `config`.

Es importante tener en cuenta que los ficheros de configuración solamente deben contener propiedades estáticas y constantes, nunca sujetas a cambios. Para esto último, se emplean los ficheros `.state`, cuyas particularidades se detallan a continuación.

6.1.5.2. Estado

Existe información de control que los **DataCollectors** emplean para ejecutar ciertas acciones. Esta debe ser persistente entre ejecuciones, y la forma más sencilla de lograrlo es mediante el uso de ficheros de texto plano que la almacenen.

Un ejemplo de información de control es el *timestamp* de la última ejecución. Cada **DataCollector** posee una frecuencia de actualización que determina si debe recolectar datos o no, aunque se ejecute cada vez que se lance el subsistema.

El estado se persiste en ficheros **.state**, que almacenan variables y *flags* de control en formato JSON.

En tiempo de ejecución, el contenido del fichero es accesible desde el atributo de instancia **state**, una vez el **DataCollector** ha alcanzado el estado de transición **STATE_RESTORED**.

El contenido de estos ficheros puede llegar a corromperse dado que se sobrescriben periódicamente. Se ha incorporado un mecanismo que permite recuperar un estado consistente en caso de que sea imposible deserializar el fichero:

- a) Se define en el fichero de configuración (**.config**) un valor especial **STATE_STRUCT**, en el que se define la estructura del fichero **.state**. Éste debe contener unos campos mínimos, requeridos por la clase base, además de otros específicos del **DataCollector** añadidos por el desarrollador.

En tiempo de ejecución, se comprueba durante el método **__init__** que el valor de **STATE_STRUCT** contiene todos los campos requeridos por la clase base. En caso negativo, se registra en un mensaje de *log* y se pasa directamente al estado **ABORTED**.

- b) En caso de que ocurra un error de deserialización, se sustituirá el contenido del fichero **.state** por **STATE_STRUCT**, que posee un formato válido, aunque se perderán los valores del estado del **DataCollector**.

Como se indicó en el punto a) de la numeración anterior, existen una serie de campos requeridos para el valor de **STATE_STRUCT**, comunes a todos los **DataCollectors**. Se indican a continuación:

- a) Campos que **deben** ser gestionados por el desarrollador:
 - i. **last_request**: *Timestamp* de la última ejecución del **DataCollector**.
 - ii. **update_frequency**: Un **dict** con la estructura **{"value": x, "units": y}**. Almacena la frecuencia de actualización actual del **DataCollector**. Esto ayuda a determinar si existe trabajo pendiente (teniendo en cuenta el *timestamp* de la última ejecución, la frecuencia de actualización y el momento en el que se realiza la comprobación).

- iii. **data_elements**: Un `int`, que almacena el número de elementos recolectados por la operación `_collect_data` en la ejecución actual.
 - iv. **inserted_elements**: Un `int`, que representa el número de elementos guardados en base de datos durante la ejecución actual.
- b) Campos que se gestionan de forma automática, pero **pueden** ser modificados por el desarrollador:
- i. **restart_required**: Un `bool`, que almacena la necesidad de reiniciar el `DataCollector` en la siguiente ejecución (e.g. si ocurre un error).
 - ii. **advisedly_no_data_collected**: Un `bool` que indica que, aunque es necesario recolectar datos, por cuestiones relativas a la lógica de negocio, el módulo no ha sido capaz de obtener información, y no se desea que esta situación se tome como un error.
- c) Campos que son gestionados de forma automática, y **no** deben ser modificados:
- i. **last_error**: Un `dict`, que almacena la clase de la última excepción lanzada por el `DataCollector`, y el mensaje de error asociado a la misma.
 - ii. **errors**: Un `dict`, que almacena la clase de todas las excepciones lanzadas por el `DataCollector`, junto con el número de veces que ha aparecido cada uno.
 - iii. **backoff_time**: Un `dict`, que almacena el siguiente retraso temporal (`backoff`) necesario para retomar una ejecución, si aparece un error repetidamente. El porqué de este atributo se explicará en la sección [7.3. El mecanismo de retraso exponencial](#).

En resumen, el valor de `STATE_STRUCT` debe contener, al menos, los campos:

```
STATE_STRUCT:
    update_frequency:
        value: null
        units: null
    last_request: null
    data_elements: null
    inserted_elements: null
    restart_required: false
    last_error: null
    error: null
    errors: {}
    backoff_time:
        value: 1
        units: s
```

Código 4: Campos requeridos para el `STATE_STRUCT` de cualquier `DataCollector`.

Además de los campos requeridos anteriormente indicados, existe la posibilidad de añadir nuevos atributos a elección del desarrollador.

No obstante, debe tenerse en cuenta que todos los campos de `STATE_STRUCT` deben ser **JSON serializables**. En caso contrario, al intentar serializar el estado se lanzará una excepción, y el `DataCollector` terminará su ejecución alcanzando el estado **ABORTED**.

Generalmente, si un campo no es serializable, deberán sobrescribirse los métodos `_restore_state` y `_save_state`, ya que es aquí donde se realiza la serialización y deserialización del estado. En siguiente ejemplo se muestra una forma correcta de implementar este escenario:

```
# 1st step
class MyObject:

    def __init__(self, foo, baz):
        self.foo = foo
        self.baz = baz

    def serialize(self) -> dict:
        return {'foo': self.foo,
                'baz': self.baz}

    @staticmethod
    def deserialize(value: dict):
        return MyObject(value['foo'],
                        value['baz'])

# 2nd step
STATE_STRUCT:
    # remaining fields
    my_value: null

class MyCollector(DataCollector):

    # 3rd step
    def _restore_state(self):
        super().__restore_state()
        self.state['my_value'] =
            MyObject.deserialize(
                self.state['my_value'])

    # 4th step
    def _save_state(self):
        self.state['my_value'] =
            self.state['my_value'].serialize()
        super().__save_state()
```

Tenemos una clase `MyObject`, que queremos incluir dentro del estado.

1. Debe crearse una forma de serializar y deserializar objetos de la clase `MyObject`. La forma más sencilla es contar con métodos dentro de la propia clase.
2. Debe añadirse al valor de `STATE_STRUCT` el nuevo campo (`my_value`) en el fichero de configuración del módulo (extensión `.config`).
3. Debe sobrescribirse la función `_restore_state`, e invocarse en la primera línea el método de la clase padre, para obtener el estado deserializado.
4. Debe sobrescribirse la función `_save_state`, e invocarse el método de la clase padre **después** de haber serializado el objeto (si se invocara antes, lanzaría una excepción, ya que el valor `my_value` aún no es serializable).

Código 5: Ejemplo de sobreescritura de las operaciones de un `DataCollector` cuando existe un campo en la `STATE_STRUCT` que no es JSON serializable.

6.1.6. MongoDB

Como se indicó en la sección [6.1.4. Estructura modular de los DataCollectors](#), cada uno se conecta a una *collection* de MongoDB, con el mismo nombre que el módulo. Con el fin de simplificar el uso de MongoDB desde los **DataCollectors**, se crea una clase **MongoDBCollection** que permite:

- Hacer uso de un *pool* de conexiones, en lugar de crear una propia para cada **DataCollector** (crear conexiones es caro en cualquier SGBD, aunque el coste en MongoDB es relativamente menor que en SGBD relacionales).
- Conseguir una conexión directa con una *collection*, en lugar de conectarse a la base de datos y luego a la *collection*. Esto reduce la duplicidad en el código.
- Actuar como *proxy* de la conexión, simplificando funcionalidades usadas por múltiples **DataCollectors**.
- Actuar como capa de abstracción entre la librería **pymongo** y el subsistema. Esta clase proporciona consistencia ante cambios en el API de la librería, ya que los cambios en ésta solo afectarán a la clase **MongoDBCollection**.

Para implementar esta clase, se emplea el *Patrón Proxy*²⁵:

- a) Al instanciar **MongoDBCollection** por primera vez se crea un objeto de tipo **pymongo.MongoClient**. De ahí en adelante, se reusará el **MongoClient** ya creado.
- b) Se definen una serie de operaciones simplificadas sobre el API de **pymongo**. El resto de operaciones se delegan directamente al atributo de instancia **collection**, que referencia al objeto **MongoClient** de la librería anterior.

Además, se emplea nuevamente la *Programación Orientada a Aspectos* (AOP), que permite, mediante la intercepción de métodos, asegurar que las llamadas a operaciones de la clase **MongoDBCollection** creen conexiones con MongoDB de forma **transparente**; dado que, al instanciar la clase, éstas no se inicializan (i.e. *lazy loading* o carga perezosa):

```
@proxy_collection
def bulk_write(self, *args, **kwargs):
    return self._collection.bulk_write(*args, **kwargs)
```

Código 6: Ejemplo de uso del Patrón Proxy. Este método delega en la implementación de la librería pymongo; aunque la conexión es inicializada antes de la llamada al método, si fuera necesario (lazy loading).

²⁵ **Patrón Proxy:** Proporciona una forma de controlar invocaciones a peticiones con un alto coste computacional (*carga perezosa*), o simplificar la interacción con elementos remotos (e.g. SOAP).

6.2. Componente *logger*

Habilitar el registro de operaciones y errores es una tarea que ayuda enormemente a la hora de depurar y administrar un sistema.

En Python, esto es sencillo, dado que existe la librería nativa `logging` que proporciona esta funcionalidad. No obstante, se decide extender esta librería, con el fin de dotar al sistema de mayores capacidades. Concretamente:

- Se aprovecha el soporte para crear ficheros rotatorios²⁶ ofrecido por la librería estándar, pero se añade la capacidad de enviar el fichero de *log* por correo electrónico antes de eliminarlo por completo.
- Se implementa una funcionalidad para enviar registros de *log* con prioridad `CRITICAL` y/o `ERROR` vía Telegram. Esto conforma una buena solución de cara a agilizar el manejo de errores graves del sistema (e.g. la caída de un componente), y así mejorar su disponibilidad –RNF-01–, y fiabilidad –RNF-07–. Se muestra un ejemplo, a continuación:

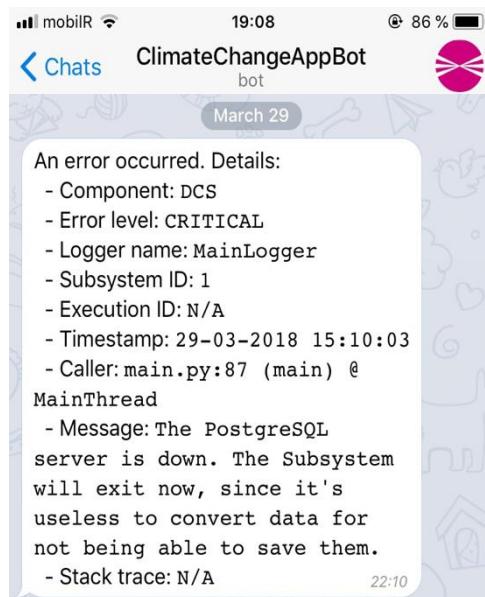


Ilustración 13: Ejemplo de mensaje de error `CRITICAL` enviado automáticamente por uno de los subsistemas vía Telegram.

Los detalles de configuración del sistema para generar alertas por medio de Telegram se especifican en el [ANEXO C. CONFIGURACIÓN DE TELEGRAM](#).

²⁶ **ficheros de *log* rotatorios:** Son ficheros de *log* configurados para que, una vez el fichero de *log* llega al máximo tamaño permitido –*S*–, se crea un nuevo fichero en lugar de sobrescribir el actual. Esto se realiza hasta que se alcanza el número máximo de ficheros de *log* permitidos –*n*–. En ese momento, se borra el más antiguo, garantizando un uso máximo de memoria de $n \times S$ bytes.

6.3. Componente *importador de módulos*

Este componente es fundamental para lograr el desacoplamiento entre los módulos de los **DataCollectors** y el resto del subsistema.

Mediante este mecanismo, es posible importar y cargar módulos –de forma dinámica y recursiva– desde un directorio base.

El comportamiento de este componente se puede apreciar en el siguiente diagrama de secuencia:

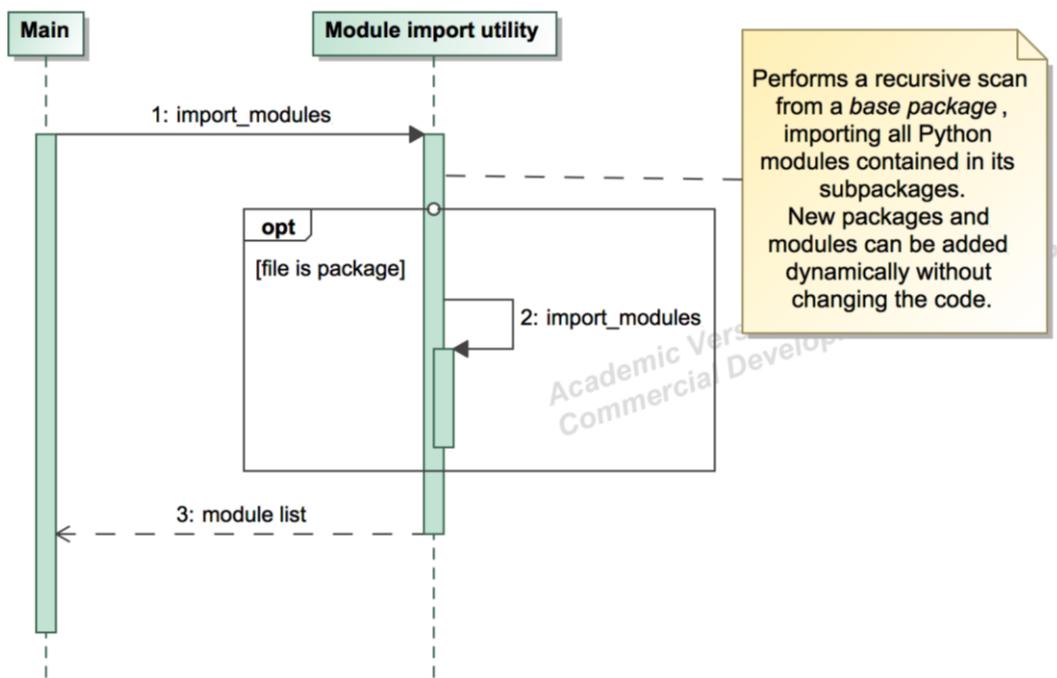


Ilustración 14: Descripción del proceso recursivo de importación de módulos.

Dada la naturaleza de Python, es posible asignar el resultado de la importación de un módulo a una variable; dado que, en este lenguaje, todo elemento desciende de la clase **object**.

Una vez obtenidos los módulos, se obtienen referencias a los **DataCollectors** mediante el método **instance**, que proporciona el punto de acceso global a éstos (esto se explica anteriormente en la sub-sección [6.1.4. Estructura modular de los DataCollectors](#)).

6.4. Componente *principal*

El componente principal no se implementa de forma completa en este *sprint*. No obstante, se define su comportamiento, en líneas generales.

Si se recuerda el flujo de ejecución planteado en el diagrama de secuencia del subsistema ([Ilustración 7](#)), las tareas de este componente son:

- a) Comprobar que el servicio MongoDB está activo. En caso contrario, no tiene sentido ejecutar el subsistema; ya que, aunque se recolecte información, esta no podrá almacenarse.
- b) Lanzar el componente *supervisor*.
- c) Instanciar todos los módulos, y obtener una referencia a sus **DataCollectors**.
- d) Ejecutar los **DataCollectors**, y esperar a que terminen todas las ejecuciones.
- e) Solicitar al supervisor que registre un informe de la ejecución y finalice.
- f) Terminar la ejecución del subsistema.

Por lo tanto, se puede apreciar que este componente es el encargado de coordinar e invocar al resto de integrantes del subsistema.

6.4.1. Conurrencia

Los **DataCollectors** son trabajadores independientes cuya ejecución puede realizarse de forma simultánea.

Como se vio en la asignatura *Concurrencia y Paralelismo*, existen dos maneras de lograr simultaneidad entre trozos de código: concurrencia, mediante el uso de hilos o *threads*; y paralelismo, empleando múltiples procesos.

Tras haberse documentado sobre cómo lograr concurrencia y/o paralelismo en Python [18], se han llegado a las siguientes conclusiones:

- a) En materia de **concurrencia**:
 - i. Python emplea un mecanismo de concurrencia conocido como **GIL** (Global Interpreter Lock), un *mutex* global.
 - ii. Para ganar en términos de eficiencia, gran parte de las librerías nativas de Python ejecutan por debajo código C pre-compilado, mucho más rápido que el *bytecode* de Python. No obstante, cada pieza de código C debe adquirir el GIL antes de ser ejecutada.
 - iii. En entornos *multi-thread*, esto se traduce literalmente a “un *thread* ejecuta código Python, mientras que el resto están *dormidos* o esperan entrada/salida”.
 - iv. En Python 3, el *thread* que ejecuta código lo hace durante 15 ms. En ese momento, se cambia de contexto, y otro *thread* adquiere el GIL y pasa a ejecución.
 - v. Emplear concurrencia en Python está sujeto al GIL y sus limitaciones.

- b) Por otro lado, respecto al uso de múltiples procesos (**parallelismo**):
 - i. Esto consigue evitar el uso del GIL, dado que se crean múltiples procesos corriendo Python, cada uno con su propio GIL.
 - ii. Necesita mayores recursos: computacionalmente, es más costoso crear procesos que *threads*.
- c) Existen casos en los que la computación paralela es más eficiente que el uso de *threads*, y viceversa (e.g. calcular la multiplicación de una matriz de orden $m \times n$, siendo m y n números del orden de 10^6 ; vs atención de peticiones en un servidor web²⁷).

Dada la naturaleza del subsistema, se opta por emplear **concurrencia** en lugar de paralelismo. Esto se justifica debido a que no está entre las funcionalidades del subsistema el ejecutar cálculos complejos que requieran múltiples procesos, sino que se parece mucho al escenario anteriormente mencionado del servidor web: peticiones a múltiples fuentes de datos a través de la red (i.e. predominancia de entrada/salida).

Es conveniente destacar el hecho de que los **DataCollectors** solo acceden a sus propios ficheros de configuración y estado. Dado que estos últimos son mutables, en caso de pretender hacer escritura desde múltiples *threads*, debería implementarse un sistema para manejar la concurrencia (e.g. uso de *locks*, *mutex* o semáforos). No obstante, este escenario no es posible, dado que existe una relación 1:1 entre los ficheros **.state** y **DataCollectors**.

Como última tarea del *sprint*, se implementa una primera aproximación de la ejecución concurrente del subsistema. El resultado obtenido es el siguiente:

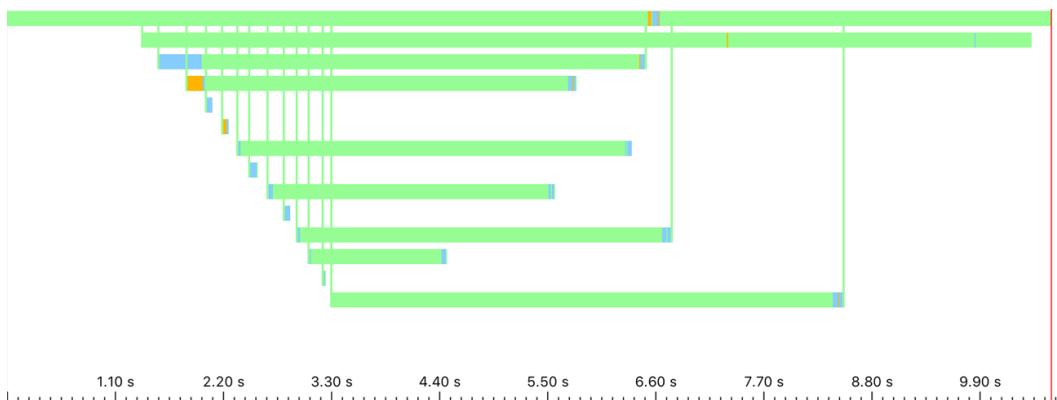


Ilustración 15: Diagrama de concurrencia para una ejecución del subsistema.

²⁷ pese a que los servidores web atienden generalmente cada petición en un *thread*, a menudo se emplean también múltiples procesos para maximizar el rendimiento. No obstante, casi siempre se dará que $T \gg p$, siendo T el número de *threads* y p el número de procesos activos, respectivamente.

Cabe destacar ciertos aspectos:

- Cada barra horizontal se corresponde con un *thread*. Además:
 - Cuando el color de la barra es **verde**, el *thread* está en ejecución.
 - Cuando es **naranja**, el *thread* está esperando por un *lock*.
 - Cuando es **azul**, el *thread* está en ejecución, con un *lock* adquirido.
 - Cuando es **roja**, ocurre un *deadlock* (interbloqueo). Esta situación no llega a darse.
- La primera barra corresponde al componente principal (**MainThread**).
- La segunda corresponde al componente supervisor (**SupervisorThread**).
- El resto de barras corresponden a 12 **DataCollectors** con una implementación singular: son clases *mock*²⁸, que extienden a la clase base **DataCollector** y cuya lógica de negocio consiste en simples esperas aleatorias.
- Las líneas verticales representan la comunicación entre *threads*.

Una vez se han explicado los elementos del diagrama, resulta sencillo apreciar que el comportamiento es el esperado, teniendo en cuenta la [Ilustración 7](#), donde se muestra el diagrama de secuencia enfocado a la interacción entre componentes:

- El componente principal realiza el *ping* a MongoDB, invoca al componente *importador de módulos*, inicializa al componente supervisor e instancia a los **DataCollectors**. (primer segundo de ejecución).
- Cada **DataCollector** se ejecuta en su propio *thread*. Antes de iniciar su ejecución –con el método *run*–, envían un mensaje al supervisor, indicando que van a proceder.
- En el momento en el que un **DataCollector** finaliza su ejecución –tanto si es de forma satisfactoria o errónea–, éste envía otro mensaje al supervisor, indicando que ha finalizado.
- El componente principal espera a que todos los **DataCollectors** han finalizado, e indica al componente supervisor que realice un informe de ejecución y finalice.
- El componente supervisor verifica la ejecución de los módulos, genera el reporte de ejecución, y se conecta a MongoDB para almacenarlo. Acto seguido, finaliza su ejecución (último segundo de ejecución).
- El componente principal, a la espera de que el componente supervisor finalizase, termina la ejecución del subsistema.

Pese a que se ha mencionado aquí, se tratarán en detalle los aspectos relacionados con el **componente supervisor** en la sección [7.1. Componente supervisor](#).

²⁸ **componente mock**: el uso de *mocks* es habitual a la hora de probar componentes cuyas operaciones son costosas (en términos de tiempo o esfuerzo computacional). También son útiles para simular el comportamiento de componentes, clases o métodos.

7. SEGUNDO *SPRINT*

Como resultado del primer *sprint*, se obtiene:

- ✓ La implementación casi completa, de todos los **DataCollectors**.
- ✓ Una primera aproximación de las implementaciones de los componentes principal y supervisor.
- ✓ Diagramas y documentación.

El *sprint* 2 se encuadra entre el 26/12/2017 y el 06/02/2018, con un esfuerzo estimado de 106 horas*hombre.

Los esfuerzos del *Equipo* en este *sprint* se centran en satisfacer los siguientes requisitos funcionales o *user stories*:

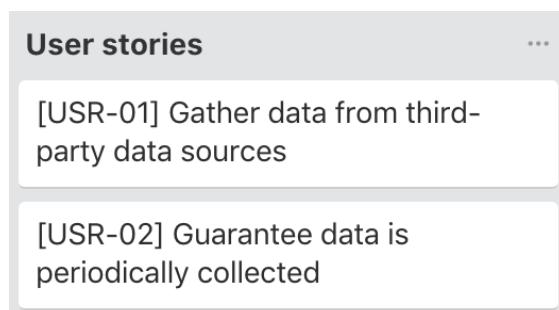


Ilustración 16: User stories a implementar durante el sprint 2.

Nótese que todavía no se ha implementado por completo la *user story* del anterior *sprint*, por lo que también se incluye en éste. El *Sprint Backlog* contiene un total de 16 tareas en esta iteración.

A continuación, se detallarán los aspectos de diseño e implementación desarrollados durante la acometida de este *sprint*, cada uno en una sección.

7.1. Componente *supervisor*

Este componente surge con la idea de detectar fallos en **DataCollectors** y responder ante ellos. Entre sus funciones también está la de elaborar los informes de ejecución del subsistema.

7.1.1. Paso de mensajes

Como se adelantó en la última sub-sección del capítulo anterior, los **DataCollectors** envían mensajes al supervisor para notificar el comienzo y fin de sus ejecuciones. Existen otras connotaciones que cabe destacar sobre este asunto:

- El paso de mensajes se realiza a través de una **cola sincronizada**, implementando el escenario *Productores-Consumidores* visto en la asignatura *Concurrencia y Paralelismo*.
- Como **productores** se sitúan los **DataCollectors**, mientras que el único **consumidor** es el componente supervisor.
- Para enviar un mensaje, el productor debe adquirir el *lock* que protege la cola. En el momento en el que es enviado, se realiza una notificación al consumidor (llamada a **notify**), que lo despierta.

El siguiente fragmento de código detalla el comportamiento de la clase que actúa como mensaje:

```
class Message:  
    """  
    Provides a unified interface to send messages through a shared channel.  
    """  
  
    def __init__(self, message_type: MessageType, content=None):  
        self.type = message_type  
        self.content = content  
  
    def send(self, channel: Queue, condition: Condition):  
        """  
        Sends the message itself through the channel.  
        :param channel: A synchronized queue.Queue object.  
        :param condition: A threading.Condition object, which notifies  
                          the receiver that a message has arrived.  
        """  
        condition.acquire()  
        channel.put_nowait(self)  
        condition.notify()  
        condition.release()
```

Ilustración 17: Implementación de la clase Message. Los objetos de esta clase actúan como forma de comunicación entre los DataCollectors y el supervisor.

7.1.2. Comprobación de ejecuciones

El objetivo del paso de mensajes entre los **DataCollectors** y el componente supervisor es el de llevar un registro de qué módulos se ejecutan, y programar el reinicio de aquellos en los que surjan errores.

Para ello, cada vez que un **DataCollector** ha finalizado, el supervisor ejecuta su método **verify_module_execution** sobre dicho **DataCollector**, donde se realizan las siguientes acciones:

- a) Si el **DataCollector** ha finalizado con éxito, se registra este hecho.

- b) En caso contrario, además de registrar el error, se persiste en el fichero `.state` del `DataCollector` la información del fallo, y se programa el reinicio de éste.

Para minimizar el acoplamiento entre los `DataCollectors` y este componente, se implementan en la clase base `DataCollector` los métodos `expose_transition_states` y `execute_actions`:

```
def expose_transition_states(self, who) -> list:  
    [...]  
  
def execute_actions(self, state: int, who) -> bool:  
    [...]
```

Código 7: Firmas de los métodos de la clase base DataCollector empleados por el componente supervisor.

La particularidad de estos métodos es que implementan el *Patrón Amigo*, por lo que solamente se ofrecerán las operaciones anteriores si quien se las pide es una instancia de `DataCollectorSupervisor` (la clase implementadora del componente supervisor).

Además, con estos métodos, el componente supervisor no necesita acceder a la implementación interna de la clase base: es el propio `DataCollector` el que ejecuta las acciones, aunque es el supervisor quien se lo indica.

7.1.3. Reporte de ejecución

El componente supervisor genera y guarda en MongoDB un reporte de cada ejecución del subsistema (en formato JSON). Con esta utilidad, se mejora la capacidad de detectar errores y obtener estadísticas, como:

- Duración máxima, mínima y media de las ejecuciones.
- Número total de elementos guardados.
- Tiempo total de ejecución.
- Relación de ejecuciones exitosas y/o fallidas respecto al total.
- Estadísticas detalladas por `DataCollector`: número de ejecuciones con trabajo pendiente, cantidad de elementos recopilados, todos los errores...

Con estos datos pueden establecerse políticas como aumentar la frecuencia de actualización de `DataCollectors`, solucionar errores de forma ágil, descartar fuentes de datos al no proporcionar valores y/o ser inestables, etc.

En la página siguiente se muestra un ejemplo real de un reporte generado y recopilado por el componente supervisor. Con el fin de no truncar el informe, sólo se muestran datos agregados de uno de los `DataCollectors`:

```
"last_execution": {
    "subsystem_version": "v2.5 (2018.3)",
    "timestamp": 1521834658407,
    "duration": 54.74983233306557,
    "collected_elements": 66,
    "inserted_elements": 66,
    "execution_succeeded": true,
    "modules_executed": 11,
    "modules_with_pending_work": {
        "historical_weather": {
            "collected_elements": 66,
            "saved_elements": 66
        }
    },
    "modules_succeeded": 11,
    "modules_failed": {
        "amount": 0,
        "modules": null
    },
    "subsystem_id": 1,
    "execution_id": 14840
},
"aggregated": {
    "collected_elements": 1100667,
    "execution_time": 341368.6509444997,
    "executions": 13675,
    "failed_executions": 2435,
    "inserted_elements": 1100667,
    "max_duration": 255.95203407399822,
    "mean_duration": 24.962972646764154,
    "min_duration": 0.4288074900396168,
    "per_module": {
        "air_pollution": {
            "total_executions": 13675,
            "executions_with_pending_work": 1023,
            "succeeded_executions": 13654,
            "failed_executions": 21,
            "failure_details": {
                "ConnectionError": [
                    1379,
                    1575,
                    1912,
                    3369,
                    3672,
                    3901,
                    4281
                ],
                "KeyError": [
                    1,
                    2
                ]
            }
        },
        "[...]"
    },
    "succeeded_executions": 11240,
    "subsystem_id": 1
}
```

Código 8: Ejemplo de reporte de ejecución del subsistema.

7.2. Configuración

Al igual que los `DataCollectors` poseen su propia configuración (ficheros con extensión `.config`), también se permite su definición a nivel de componente, subsistema y sistema.

El mecanismo es idéntico: se emplean ficheros `.config` en formato YAML. Además, para facilitar el desarrollo y la puesta en producción, se incluyen ficheros de configuración especiales para cada escenario. Dependiendo del valor de la variable de entorno `DOCKER_MODE`, se cargarán unos u otros en tiempo de ejecución. A continuación, se muestra la estructura de un directorio de configuración:

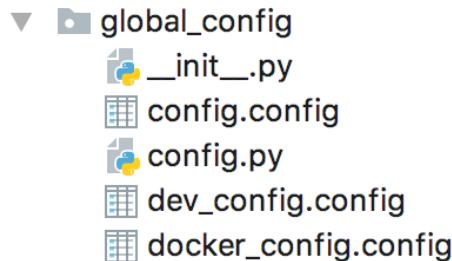


Ilustración 18: Directorio de configuración global del sistema.

En este caso, se muestra el directorio de configuración global del sistema. En él se almacenan propiedades y aspectos comunes a los distintos subsistemas: configuración de bases de datos, versión del sistema... Se detalla el contenido del mismo a continuación:

- El fichero `__init__.py` es un fichero especial, necesario para que Python detecte que un directorio conforma un paquete. Puede contener definiciones, importaciones a nivel de paquete, etc.
- El fichero `config.config` almacena información independiente del entorno de desarrollo.
- El fichero `dev_config.config` contiene propiedades a emplear sólo en el entorno de desarrollo (e.g. directorio raíz del sistema, ubicación de ficheros de *log*, etc.).
- El fichero `docker_config.config` es análogo al anterior, salvo que las propiedades se cargan sólo en el entorno de producción.
- El fichero `config.py` es el encargado de mantener la configuración en forma de variable, accesible desde cualquier componente, mediante una simple sentencia `import`:

```
from global_config.config import GLOBAL_CONFIG
```

Código 9: Ejemplo de carga de configuración del sistema.

Desde ese momento, la configuración está disponible como un **dict**, y las propiedades definidas en el fichero **.config** son accesibles como cualquier clave en un diccionario en Python:

```
app_version = GLOBAL_CONFIG['APP_VERSION'])
```

Código 10: Acceso a una variable de configuración del sistema desde código Python.

Debe tenerse en cuenta que:

- Los ficheros **dev_config.config** y **docker_config.config** deben poseer las **mismas** claves, aunque distintos valores. Si solo se va a emplear uno, es mejor dejar estos ficheros vacíos, y situar la configuración en el fichero genérico **config.config**.
- Todos los directorios de configuración del sistema siguen el **mismo** estándar a la hora de definir configuración: se usan los mismos nombres de ficheros y extensiones de archivo.

7.3. *El mecanismo de retraso exponencial*

Existen ocasiones en las que el subsistema fallará debido a causas externas: caídas de servidores, cambios de dirección IP, etc.

Con el fin de minimizar el uso de recursos y evitar denegaciones de servicio, se diseña e implementa un mecanismo que retrase la ejecución de un **DataCollector** en caso de que éste falle debido al mismo error de forma repetida.

Inspirado en el *exponential backoff* del protocolo TCP –visto en la asignatura *Redes*–, este mecanismo permite que los **DataCollectors** esperen un tiempo aleatorio –creciente, de forma exponencial– antes de volver a recopilar información.

Este *backoff* (retraso) se serializa en el fichero **.state** del **DataCollector**. En tiempo de ejecución, además de realizarse la comprobación habitual de si existe trabajo pendiente, el **DataCollector** comprueba si el *backoff* le permite continuar. En caso contrario, el evento se registrará mediante un mensaje de *log*:

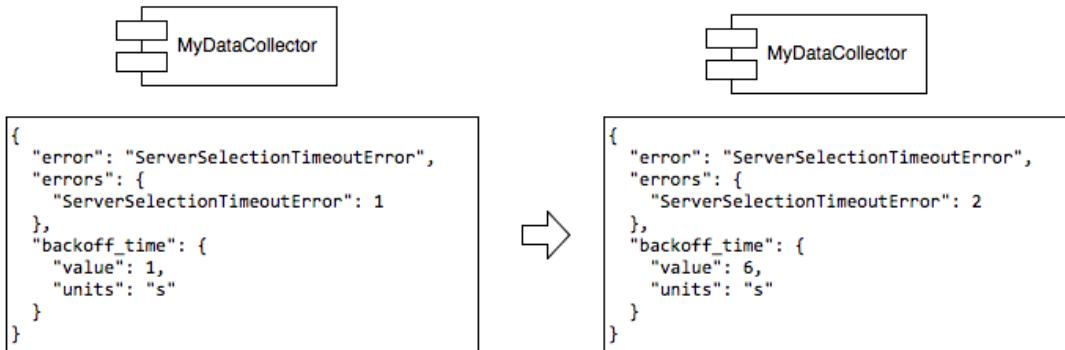
```
[INFO] [DGS] [ID:1] [EXEC:18673] 06-04-2018 14:25:03.117 {data_collector.py:229 (_has_pending_work) @ _EnergySourcesDataCollectorThread}: Exponential backoff prevented data collection. Current backoff is: 21600 s.
```

Código 11: Fragmento de un registro de log indicando que el mecanismo de retraso exponencial ha frenado la recopilación de datos.

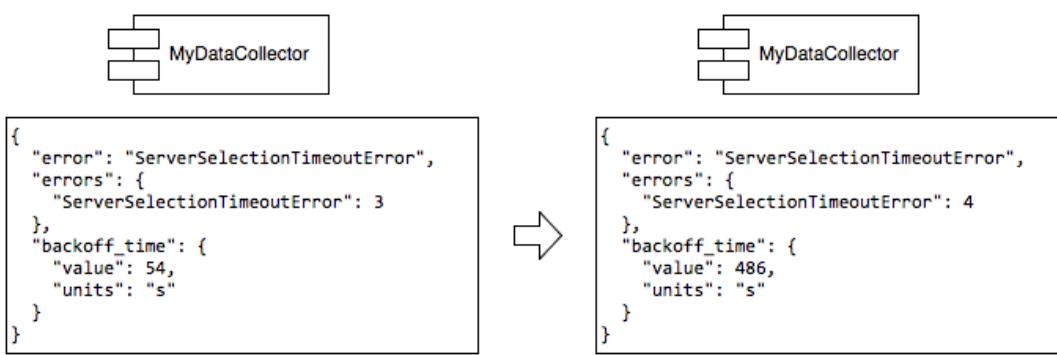
Con todo, el valor del *backoff* no crece indefinidamente. Se fija un valor máximo: 86400 s (un día). Esto implica que, a más tardar, un **DataCollector** solo permanecerá inactivo –por causas externas– un día.

A continuación, se muestra un escenario detallando el uso de este mecanismo:

1. A **ServerSelectionTimeoutError** is raised.
2. Another **ServerSelectionTimeoutError** is raised.



3. Another **ServerSelectionTimeoutError** is raised.
4. Another **ServerSelectionTimeoutError** is raised.



5. A new **HTTPError** is raised.
4. Another **HTTPError** is raised.

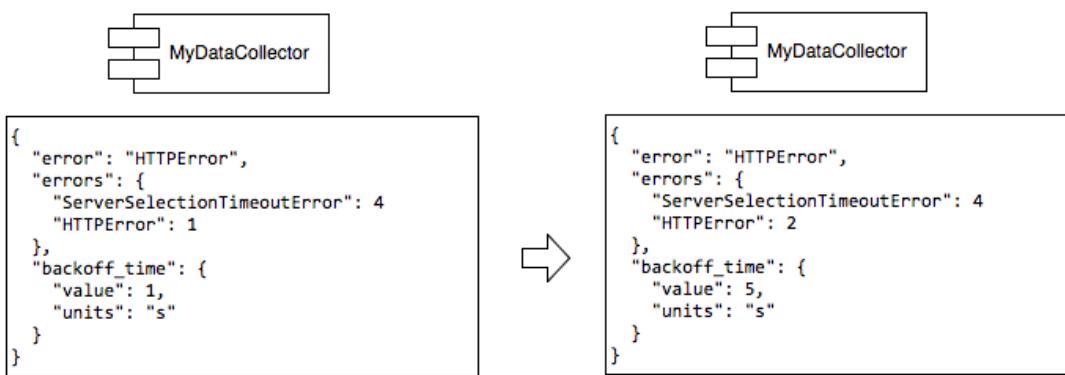


Ilustración 19: Escenario donde se muestra el uso del mecanismo de retraso exponencial.

Del escenario anterior, deben destacarse varios aspectos:

- a) El valor del siguiente *backoff* es el resultado de multiplicar el valor previo de éste por un número aleatorio comprendido entre 2 y 10.
- b) Si el error que estaba causando el retraso era de clase **X**, y se da un nuevo error de clase **Y**, el mecanismo se resetea.
- c) Cuando el *backoff* alcanza el valor límite (86400 s), en la siguiente ejecución, el mecanismo se resetea.

7.4. Ejecuciones finitas

Dada la naturaleza concurrente del sistema, y al hecho de que no se han realizado pruebas exhaustivas²⁹ que garanticen la ausencia de posibles *deadlocks* (interbloqueos), se opta por implementar un mecanismo que impida que una ejecución se prolongue en el tiempo más de lo necesario.

Para ello, se hace uso de la librería nativa **signal** y de **contextmanagers**³⁰ para implementar la función **time_limit**. La lógica del algoritmo es simple:

```
try:  
    with time_limit(seconds):  
        my_long_method()  
except TimeoutError:  
    # Define actions here
```

Código 12: Caso de uso de la función **time_limit**.

Al llegar al número máximo de segundos de espera, se lanzará un **TimeoutError** que abortará la ejecución del método en curso. Esto es efectivo si se produce un *deadlock*, ya que la ejecución finalizará a más tardar, tras el tiempo de espera:

```
[CRITICAL] [DGS] [ID:1] [EXEC:24874] 28-04-2018 11:04:48.083 {main.py:123  
<module>} @ MainThread}: The Subsystem execution has been timed out.
```

Código 13: Registro de log que indica una ejecución abortada, tras alcanzar ésta el tiempo máximo permitido.

Actualmente, el subsistema lanzará este error tras 4 minutos y 45 segundos de ejecución.

²⁹ para realizar este tipo de pruebas podría emplearse *model checking*, como la técnica SPIN vista en la asignatura *Verificación e Validación do Software*.

³⁰ **contextmanager**: Son similares a un bloque **try-with-resources** en Java. Casos de uso reales son la apertura de ficheros, adquisición de *locks*... El desarrollador también puede definir sus propios **contextmanagers** con la ayuda de la librería nativa **contextlib**.

7.5. Utilidades

Existen funcionalidades que son usadas a lo largo de todo el sistema. Desde el momento inicial, éstas se ubican en el paquete **utilities**, que cuelga del directorio raíz del proyecto.

Se incluyen en este paquete:

- Utilidades de *logging*.
- Utilidades relacionadas con MongoDB.
- Utilidades para trabajar con PostgresSQL.
- Utilidades como la carga de ficheros de configuración y estado, *parseo* de valores, operaciones con fechas, etc.

Nota: Las utilidades se implementan en distintos *sprints*, bajo demanda. En el *sprint* 2 no existen todas las funcionalidades listadas en la enumeración anterior.

7.6. Pruebas

Esta sección reúne todos los aspectos relacionados con las pruebas del Subsistema de Recolección de Datos. Éstas se realizan a varios niveles, detallados en las siguientes sub-secciones.

Un aspecto común a todas las pruebas del sistema es que se realizan con la librería nativa **unittest**. En futuros *sprints* se incorporará, además, el análisis de cobertura de código y la generación de reportes XML de resultados de *test* y cobertura.

7.6.1. Pruebas de unidad

Según lo visto en la asignatura *Verificación e Validación do Software*, en este nivel deben probarse todas las operaciones, y efectuarse la lectura/escritura de todos los atributos. Además, si los componentes poseen estado, deberá forzarse el paso por todos los estados. Teniendo en cuenta estas obligaciones, se realizan:

- Pruebas de todas las funciones utilidad del paquete **utilities**.
- Pruebas de la clase base **DataCollector**, haciendo énfasis en el paso por todos los estados (ya que este componente se comporta como una *máquina de estados finitos*).
- Pruebas sobre las implementaciones de todos los **DataCollectors**.
- Pruebas de la clase **MongoDBCollection**.

Durante la realización de las pruebas, se emplean algunas técnicas, como:

- Uso de métodos **setUp** y **tearDown**: En pruebas donde se generen datos (e.g. las de la clase **MongoDBCollection**), se emplean estos métodos para limpiar

la base de datos antes y después de cada *test*, garantizando métodos de prueba intercambiables e idempotencia.

- Uso de **mocks**: Python da soporte de forma nativa al uso de *mocks*, empleando la librería `unittest`. Mediante el uso de estos componentes, se finge el comportamiento de dependencias, agilizando la duración y coste computacional de las pruebas:

```
@mock.patch('smtplib.SMTP')
def test_email_sent(self, mock_smtp):
    logger = utilities.log_util.get_logger(file, 'TestLogger')
    for i in range(10000) * CONFIG['MAX_BACKUP_FILES']):
        logger.info('A log record.')

    # Assertions
    self.assertTrue(mock_smtp.return_value.login.called)
    self.assertTrue(mock_smtp.return value.sendmail.called)
```

Código 14: Ejemplo de uso de mocks en pruebas de unidad.

En el escenario anterior, se intenta probar la utilidad de envío de ficheros de *log* por e-mail, una vez se ha llegado al número máximo de ficheros de *log* permitidos, y el último debe ser eliminado (el lector puede obtener más información sobre este componente en la sección 6.2. *Componente logger*).

Se parte de la base de que la librería SMTP ya ha sido verificada, y no es objeto de estas pruebas comprobar su validez. De hecho, sólo es necesario comprobar que se realiza contra el servidor SMTP (llamada a `smtplib.SMTP.login`), y el envío del mensaje (`smtplib.SMTP.sendmail`).

Por lo tanto, en este método no existe conexión real con el servidor de correo ni envío de información a través de la red: sólo se comprueba que se ha realizado la llamada a dichas funciones (i.e. se prueba la lógica del método utilidad, no la de la librería `smtplib`).

Aunque solo se explica en detalle este escenario, se hace uso de *mocks* en pruebas de unidad a lo largo de todo el sistema, de forma análoga.

7.6.2. Pruebas de integración

Este tipo de pruebas son necesarias cuando se pretende comprobar las interacciones entre componentes. Se incluyen aquí:

- Pruebas del componente supervisor.
- Pruebas del componente principal.
- Pruebas del *script* de acciones de despliegue del subsistema (este aspecto se tratará en la siguiente sección).

A continuación, se muestra un ejemplo de prueba de integración. En ella, se prueba el método `supervise` del componente supervisor:

```
@mock.patch('dgs.data_collector.data_collector.get_config', Mock(CONFIG))
def test_supervise(self):
    channel = Queue(maxsize=5)
    condition = Condition()

    # Creating supervisor and DataCollectors
    s = supervisor.DataCollectorSupervisor(channel, condition)
    thread = SupervisorThreadRunner(s)
    d1 = SimpleDataCollector(data_collected=1, data_inserted=1)
    d2 = SimpleDataCollector(fail_on='_save_data')

    # Starting supervisor
    thread.start()

    # Registering DataCollectors
    Message(MessageType.register, content=d1).send(channel, condition)
    Message(MessageType.register, content=d2).send(channel, condition)

    # Simulating run
    d1.run()
    d2.run()

    # Unregistering DataCollectors
    Message(MessageType.finished, content=d1).send(channel, condition)
    Message(MessageType.finished, content=d2).send(channel, condition)

    # Make Supervisor exit, and waiting until DataCollectors have finished
    Message(MessageType.exit).send(channel, condition)
    thread.join()

    # Assertions
    self.assertEqual(2, thread.supervisor.registered)
    self.assertEqual(2, thread.supervisor.unregistered)
    self.assertListEqual([d1, d2],
                         thread.supervisor.registered_data_collectors)
    self.assertListEqual([str(d1)],
                         thread.supervisor.successful_executions)
    self.assertListEqual([str(d2)],
                         thread.supervisor.unsuccessful_executions)

    # Checking that failed modules have serialized errors and a restart
    # has been scheduled
    self.assertTrue(d2.state['restart_required'])
    self.assertIsNotNone(d2.state['error'])
```

Código 15: Ejemplo de prueba de integración del subsistema.

El código anterior simula una ejecución del subsistema en la que existen dos `DataCollectors`. Se comentan, a continuación, una serie de consideraciones:

- La clase **SimpleDataCollector** es una implementación especial de la clase base, utilizada solamente con el fin de realizar pruebas. Ésta permite establecer –de forma programática– el estado en que la ejecución fallará, el número de elementos recopilados y guardados en base de datos, etc.
- El objetivo de la prueba es verificar que el componente supervisor ha identificado correctamente el número de **DataCollectors** ejecutados, fallidos y exitosos; además de haber serializado los errores y programado el reinicio de aquellos **DataCollectors** fallidos.
- La integración de componentes se realiza, en este caso, mediante el **paso de mensajes**.

7.6.3. Resultados

Se han empleado más de **40 horas** para realizar las pruebas del subsistema, desarrollando **153 casos de prueba**. Pese a que todavía no se dispone de reportes de cobertura, se emplea uno incluido en el IDE, y se contemplan niveles de **cobertura de decisión**³¹ superiores al **90%**.

Teniendo en cuenta estos resultados, se cuenta con la **confianza** necesaria para ubicar al subsistema en el servidor de producción.

7.7. Despliegue

Para efectuar el despliegue del subsistema, se hace uso de **Docker**.

Esta plataforma permite que los componentes del sistema se ejecuten sin problemas en cualquier máquina, sin importar el sistema operativo o librerías instaladas. Esto supone ahorros importantes de tiempo para lograr la puesta en marcha del sistema.

A continuación, se especificarán los elementos que entran en juego a la hora de desplegar un componente del sistema:

- a) **Dependencias** con otros componentes: En caso de que el componente posea dependencias, deberá comprobarse antes que éstas se encuentren operativas y son alcanzables desde el componente a desplegar.
- b) **Configuración**: Bien sea a través de variables de entorno u otros medios, deben implementarse los mecanismos necesarios para efectuar un despliegue en cualquier tipo de sistema, sin necesidad de tocar el código fuente.

³¹ existen varios niveles de cobertura. El primero es el nivel de **código**, en el que solo indica las instrucciones por las que ha pasado la ejecución. El siguiente es el nivel de **decisión**, que evalúa, además de lo anterior, las alternativas en decisiones condicionales (**if ... else**) y bucles. Por último, el nivel de **condición** evalúa todo lo anterior y valores frontera, etc.

- c) **Acciones** de despliegue: Estas acciones, generalmente, se ejecutarán una única vez durante la construcción del sistema (e.g. creación de tablas, usuarios o permisos, verificación de módulos, ejecución de *tests*, limpieza de ficheros de *log* y estado...).
- d) Fichero `docker-compose.yml`: En este fichero se definen todos los servicios necesarios para la ejecución del sistema: bases de datos, subsistemas, etc. Sigue la sintaxis YAML.
- e) Ficheros `Dockerfile`: Especifican las acciones necesarias para crear una imagen. Parten de otra ya construida, a la que se le añaden pasos adicionales para obtener más funcionalidades.
- f) **Imágenes**: Éstas se obtienen directamente (de forma remota, desde un repositorio externo), o son generadas a partir de un fichero `Dockerfile`. Conforman una versión inmutable y no ejecutable de un contenedor.
- g) **Contenedores**: Son componentes ejecutables, con un único propósito. Están aislados unos de los otros, aunque existen mecanismos para hacerlos cooperar. En concepto, son similares a máquinas virtuales, aunque los contenedores emplean menos espacio y memoria RAM, y comparten el *kernel* con el *host*:

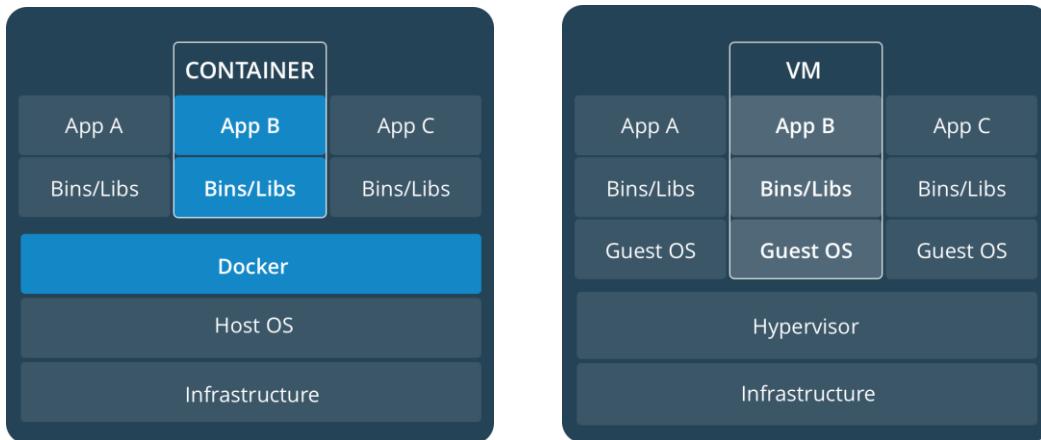


Ilustración 20: Comparación entre contenedores y máquinas virtuales.

Fuente: Docker.

- h) **Scripts de instalación**: Pese a que Docker simplifica enormemente el despliegue del sistema, se hace uso de *scripts bash* para personalizar y simplificar el proceso de instalación.
- i) **CRON**: En los sistemas Unix, esta herramienta permite la ejecución regular de procesos. Se emplea para lanzar periódicamente el subsistema (concretamente, cada **5 minutos**).

7.7.1. Volúmenes

Los contenedores son componentes efímeros. Una vez finaliza su ejecución, se destruyen y se pierde toda su información. Esto posee la ventaja de que se evitan *memory leaks* (pérdidas de memoria), ya que el subsistema se ejecuta cada vez sobre una “máquina virtual” nueva. No obstante, también implica que no se guardarán datos de forma persistente de cara a la siguiente ejecución, lo cual no resulta útil, teniendo en cuenta la naturaleza del subsistema: se emplean ficheros que guardan el estado entre ejecuciones (ficheros `.state`).

La forma de solucionar este problema es mediante el uso de **volúmenes**. Éstos permiten usar áreas del sistema de ficheros del *host* para persistir datos más allá del ciclo de vida de un contenedor.

Para declarar un volumen, se emplea la sintaxis: `<host_dir>:<container_dir>`

7.7.2. Puertos

Así como existen servicios que no es necesario que estén conectados a Internet, o ser alcanzables desde el exterior, hay otros que sí. Por ejemplo, si se despliega un contenedor con MongoDB, es necesario que éste sea accesible desde el puerto **27017** como si de una instalación local se tratase.

Docker permite exponer puertos, y hacer *port forwarding* (redirección de puertos) entre el contenedor y el *host*, lo cual resulta muy cómodo. Para declarar el *mapping* de puertos, se usa la sintaxis: `[<bind_ip>:]<host_port>:<container_port>`.

A continuación, se muestra un extracto del fichero `docker-compose.yml`, que contiene la configuración del servicio MongoDB:

```
mongodb:
  image: mongo:latest
  container_name: mongodb
  environment:
    - MONGO_DATA_DIR=/data/db
    - MONGO_INITDB_ROOT_USERNAME=root
    - MONGO_INITDB_ROOT_PASSWORD=root
  volumes:
    - ~/DataGatheringSubsystem/data/db:/data/db
  ports:
    - 27017:27017
  command: mongod --smallfiles
```

Código 16: Configuración de un servicio usando `docker-compose`.

Como se puede apreciar en el ejemplo anterior, también se permite la declaración de *variables de entorno*. Esto permite configurar un servidor MongoDB en **5 min.**

7.7.3. *Logs* y ficheros de estado

Como resultado del despliegue, en el sistema de ficheros de la máquina (en concreto, en `~/ClimateChangeApp`), aparecen los directorios:

- `/data_gathering_subsystem/log`: En este directorio se almacenan los ficheros de *log* generados por el subsistema. Cada ejecución añade los registros de *log* producidos durante la ejecución al final de los ficheros existentes.
- `/data_gathering_subsystem/state`: En este directorio se almacenan los ficheros `.state` del subsistema, que son leídos y modificados por los **DataCollectors**.
- `/data`: En este directorio se almacenan todos los datos recopilados por el subsistema. Conforma un volumen para el contenedor que ofrece el servicio MongoDB.

8. TERCER *SPRINT*

Como resultado del segundo *sprint*, se obtiene:

- ✓ La implementación completa del Subsistema de Recolección de Datos.
- ✓ El despliegue del subsistema anterior en el servidor de producción, en su versión 1.1.
- ✓ Pruebas sobre los componentes desarrollados.
- ✓ Nuevos diagramas y documentación.

El *sprint* 3 comienza el 07/02/2018 y finaliza el 05/03/2018, con un esfuerzo estimado de 144 horas*hombre, y un total de 17 tareas.

Tras una reunión de *Revisión del Sprint*, se ha decidido re-planificar el *Product Backlog*, dando como resultado que los últimos *stories* aumentasen su prioridad y, por consiguiente, se acometan durante este *sprint*:

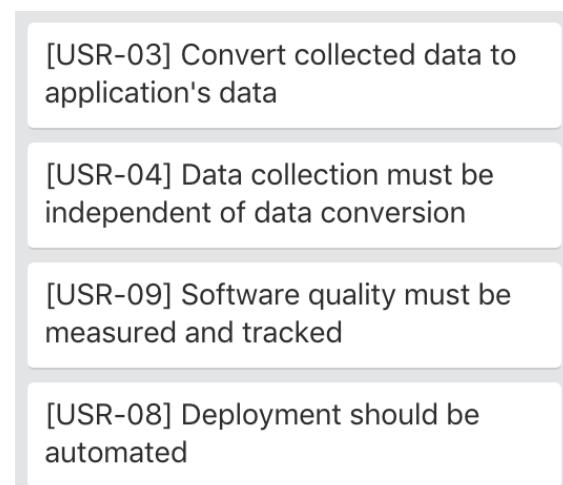


Ilustración 21: User stories a implementar durante el sprint 3.

Además, se ha tenido una reunión de *Retrospectiva del Sprint*, donde se ha decidido **refactorizar** algunos componentes.

Por último, dado que ya existe una parte del sistema en producción, comienzan a aparecer **errores**. Éstos son tratados de forma urgente en el momento en el que se encuentran: se aplaza la tarea en curso, se resuelve el problema, se verifica que la solución es buena, y se re-despliega el sistema en su nueva versión. Se siguen las convenciones detalladas en el [ANEXO D. CONVENCIONES DE ETIQUETADO](#).

A continuación, se detallarán los aspectos de diseño e implementación desarrollados durante la acometida de este *sprint*, cada uno en una sección.

8.1. Refactorización

A lo largo de este *sprint*, se aplica refactorización tanto a nivel de paquetes, como de clases; siguiendo las decisiones tomadas en la reunión *Retrospectiva del Sprint*.

8.1.1. Paquetes

En un primer momento, se pensaba mantener cada subsistema como un proyecto separado. No obstante, dado que todos los subsistemas hacen uso de utilidades y configuración común, se decide cambiar la estructura de paquetes para albergar todo el proyecto:

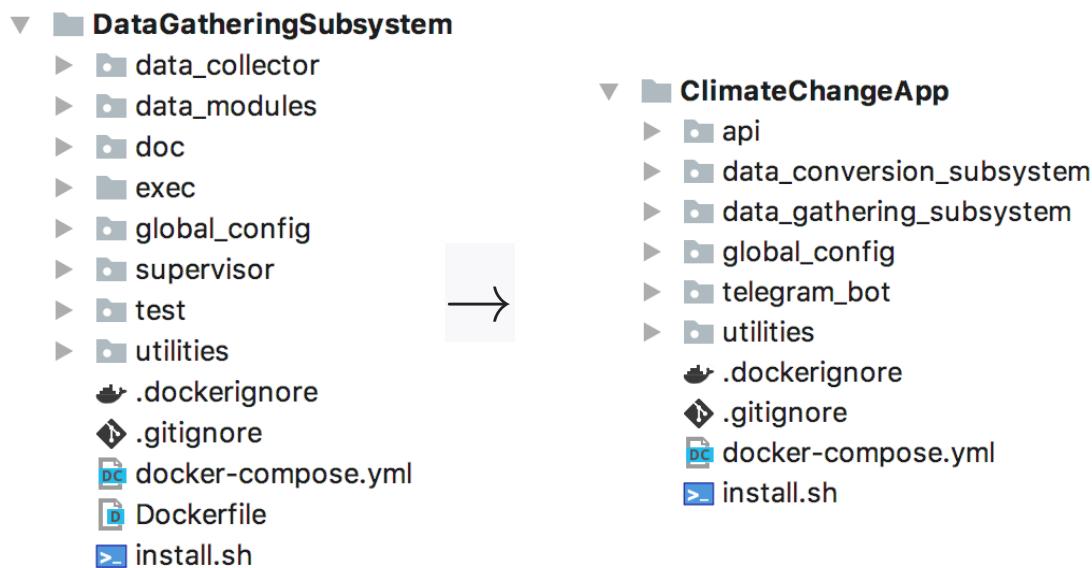


Ilustración 22: Refactorización de la estructura de paquetes del proyecto.

Tras la refactorización, cada paquete de primer nivel contiene todo lo relacionado con un componente del sistema (excluyendo el paquete **global_config**, que solo alberga configuración).

8.1.2. Clases

Para justificar la refactorización de clases, debe antes aclararse que la arquitectura del Subsistema de Conversión de Datos (a desarrollar en este *sprint*) es **análoga** a la del Subsistema de Recolección de Datos existente.

Asimismo, la clase base del nuevo subsistema —**DataConverter**—, es muy similar a la clase base **Collector**. De forma similar ocurre con el uso de *threads*, el componente *importador de módulos* y el componente *supervisor*.

Por lo tanto, se decide cambiar la estructura jerárquica de clases a otra que minimice la duplicidad de código. Las nuevas clases se ubican dentro del paquete **utilities**, en el módulo **execution_util**.

En algunos casos, basta simplemente con generalizar la nomenclatura de una clase: `DataCollectorThread` → `RunnableComponentThread`.

8.2. Componente *API*

El requisito funcional USR-04 indica que “*la recopilación de datos debe ser independiente de la conversión de los mismos*”. Por lo tanto, no puede existir ningún tipo de acoplamiento entre los Subsistemas de Recolección y Conversión de Datos.

Una solución elegante a este problema es el desarrollo de un **API REST** que exponga los datos recopilados al Subsistema de Conversión de Datos.

Se empleará el formato **JSON** para enviar datos por la red, ya que es ligero y no requiere de conversiones, dado que el Subsistema de Recolección de Datos emplea de forma nativa este formato.

8.2.1. Características

Este componente posee las siguientes características:

- a) **Escalabilidad**: Dado que el API es *stateless*, ésta puede replicarse bajo demanda.
- b) **Sencilla y ligera**: Los *endpoints* son claros, los códigos de respuesta HTTP intentan seguir el estándar RFC³², y se emplea el formato ligero JSON para enviar respuestas.
- c) **Segura**: Las operaciones definidas en el API requieren de autorización mediante el uso de *Bearer tokens* en la cabecera **HTTP Authorization**. No obstante, no se dispone de un certificado SSL, por lo que el contenido **no** va cifrado mediante TLS/SSL.
- d) Basada en **ámbitos**: Los *tokens* tienen dos funcionalidades: 1) asegurar que las llamadas al API están autorizadas, y 2) verificar que cada componente sólo tiene acceso los datos de su ámbito (i.e un **identificador numérico**). Esto se explicará con más detalle, a continuación.

8.2.2. *Tokens* y ámbitos

Los *tokens* son concedidos a las instancias del Subsistema de **Conversión** de Datos. Cada instancia posee **un** solo *token* (i.e. los *tokens* tienen una correspondencia **1:1** con instancias del Subsistema de Recolección de Datos).

³² el estándar RFC define los valores y significados de los códigos de respuesta HTTP. Éstos se dividen en cinco grandes grupos: respuestas informativas, respuestas satisfactorias, redirecciones, errores de cliente y errores de servidor [19].

Son añadidos a MongoDB durante el despliegue del componente API, y se cargan desde el fichero de configuración situado en `/api/config/authorized_users.config`, siendo `/` el directorio raíz del sistema. Este fichero posee la siguiente estructura:

```
authorized_users:  
  data_conversion_subsystem:  
    token: yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK_ZXw  
    scope: 1
```

Código 17: Estructura del fichero que almacena los tokens de acceso al API.

Para añadir un nuevo *token* debe crearse una nueva clave en el elemento `authorized_users`, del que cuelgan los campos `token` (un `str` alfanumérico) y `scope` (un `int`), que representa el ámbito del *token* y debe ser el ID de una instancia existente del Subsistema de Recolección de Datos.

Como resultado, una instancia del Subsistema de Conversión de Datos sólo tiene acceso a los datos recopilados por una única instancia del Subsistema de Recolección de Datos (aquella cuyo identificador numérico corresponda al ámbito de su *token*). Esta aproximación:

- Divide el trabajo de conversión en unidades más pequeñas.
- Es afín a la arquitectura del sistema: existe una instancia de Subsistema Conversor de Datos por cada una del Subsistema de Recolección de Datos.
- Hace que las ejecuciones del Subsistema de Conversión de Datos sean más cortas, lo que se traduce en transacciones abiertas durante menos tiempo, y menor probabilidad de error.

8.2.3. Operaciones

A continuación, se detallan los *endpoints* que proporciona el API. Éstos, pueden ser:

- **Públicos:**
 - `/alive`: Esta es la única operación expuesta de forma pública, con el propósito de determinar si la API y MongoDB están activos. Esto se explica en la sección [9.2. Healthcheck](#).
 - Método HTTP: `GET`.
 - Parámetros: Ninguno.
 - Códigos de respuesta HTTP:
 - `200` (API y base de datos activos).
 - `503` (servicio no disponible).
- **Privados** (accesibles solo mediante *tokens*):

- **/modules:** Este *endpoint* recupera el nombre de los módulos de los **DataCollectors** que han recopilado datos.
 - Método HTTP: **GET**.
 - Parámetros: Ninguno.
 - Códigos de respuesta HTTP:
 - 200 (información devuelta satisfactoriamente).
 - 401 (operación no autorizada).
 - 403 (falta el ámbito del *token* de autorización).
 - 503 (servicio no disponible).
- **/executionStats:** Recupera estadísticas de una ejecución.
 - Método HTTP: **GET**.
 - Parámetros:
 - **executionId (int):** Opcional. Si está presente, se recuperan estadísticas de la ejecución con dicho ID. En caso contrario, se recuperan las de la última ejecución del subsistema.
 - Códigos de respuesta HTTP:
 - 200 (información devuelta satisfactoriamente).
 - 400 (parámetro **executionId** presente pero inválido).
 - 401 (operación no autorizada).
 - 404 (estadísticas no encontradas).
 - 403 (falta el ámbito del *token* de autorización).
 - 503 (servicio no disponible).
- **/pendingWork/{moduleName}/:** Comprueba si existe trabajo pendiente para un módulo y una ejecución dados. En caso afirmativo, devuelve además el número de elementos recopilados durante la ejecución.
 - Método HTTP: **GET**.
 - Parámetros:
 - **moduleName (string):** Obligatorio. Nombre del módulo del que se quiere conocer si había trabajo pendiente.
 - **executionId (int):** Opcional. Si está presente, se obtienen datos de si existe trabajo pendiente para la ejecución con dicho ID. En caso contrario, se utiliza la última ejecución del subsistema.
 - Códigos de respuesta HTTP:
 - 200 (información devuelta satisfactoriamente).
 - 400 (parámetro **executionId** presente pero inválido).

- 401 (operación no autorizada).
 - 403 (falta el ámbito del *token* de autorización).
 - 404 (no existen datos).
 - 503 (servicio no disponible).
- /data/{moduleName}/: Obtiene datos recopilados por un módulo.
 - Método HTTP: GET.
 - Parámetros:
 - **moduleName (string)**: Obligatorio. Nombre del módulo del que se quiere conocer si había trabajo pendiente.
 - **executionId (int)**: Opcional. Si está presente, se obtienen los datos recopilados durante la ejecución con tal ID.
 - **startIndex (int)**: Opcional. Si está presente, especifica el *offset* del primer valor a ser devuelto.
 - **limit (int)**: Opcional. Si está presente, reduce los resultados a **limit** valores.
 - Códigos de respuesta HTTP:
 - 200 (información devuelta satisfactoriamente).
 - 400 (parámetro **executionId** y/o **startIndex** y/o **limit** presentes pero inválidos).
 - 401 (operación no autorizada).
 - 403 (falta el ámbito del *token* de autorización).
 - 404 (no existen tal módulo).
 - 503 (servicio no disponible).

Existe una documentación más detallada de este componente, realizada empleando **Swagger**. Se encuentra disponible en el siguiente enlace:

https://app.swaggerhub.com/apis/TFG_DiegoHermida/API/1.0.0-oas3

8.2.4. Implementación

No se entrará en detalle sobre los aspectos de implementación, salvo una serie de consideraciones:

- Se emplean los *frameworks* **Eve** y **Flask**, que permiten simplificar enormemente el desarrollo de APIs REST.
- El API es accesible mediante métodos HTTP estándar: **GET**, **POST**, etc.; desde el puerto **5000** de la máquina en la que esté despegado.
- Se realizan validaciones de todos los parámetros que reciben las peticiones. En caso de ser erróneos, se devolverá un código de error **400 Bad Request**.

- Se requiere autorización para todas las operaciones, excepto la llamada a `/alive`. En caso de ser una petición no autorizada, se devuelve un código de error **401 Unauthorized**. Si el *token* no posee un ámbito adecuado, se responde con un código de error **403 Forbidden**.
- Si el API no puede proporcionar información debido a que MongoDB está caído, se devolverá un código de error **503 Service Unavailable**.
- Para efectuar las comprobaciones anteriores; se emplea, una vez más, la *Programación Orientada a Aspectos* (AOP). De esta forma, se interceptan las llamadas al API para comprobar que las peticiones son autorizadas y válidas de una forma **declarativa**, simplificando el código de cada método:

```
@app.route('/data/<module_name>')
@require_auth
@require_scope
@require_validation
def data(module_name: str):
    # Implementation
```

Código 18: Uso de la Programación Orientada a Aspectos para simplificar la implementación del API.

8.3. Subsistema de Conversión de Datos

Como se ha adelantado en la sección [8.1. Refactorización](#), la arquitectura de este subsistema es análoga a la del Subsistema de Recopilación de Datos.

Esto permite aprovechar el trabajo de ingeniería y reutilizar prácticamente el total de los componentes definidos hasta el momento. Gracias a ello, lo que en el *sprint* 1 se había planificado en más de 60 horas, aquí se puede realizar en un **tercio** del tiempo, sin sacrificar documentación o pruebas.

En lugar de una explicación detallada sobre todos los aspectos del subsistema, se mencionarán aquellas cuestiones en las que éste difiera del Subsistema de Recopilación de datos.

8.3.1. Componentes *conversor*

Así como en el Subsistema de Recopilación de Datos los componentes principales son los **DataCollectors**, aquí lo son los **DataConverters**. Existe, también, una clase base **DataConverter** que proporciona las mismas características que la clase **DataCollector**: manejo automático de errores, transiciones entre estados, persistencia y carga de configuración y estado, etc.

De una clase a la otra difieren solo tres aspectos:

- a) El método `_collect_data` pasa a llamarse `_convert_data`, y el estado de transición `DATA_COLLECTED` pasa a ser `DATA_CONVERTED`.
- b) El método anterior sí proporciona una implementación por defecto, que no debe modificarse. En ella, se llama de forma **transparente** al componente **API**, y se recuperan los siguientes valores a ser convertidos por el subsistema.
- c) Dado que los valores convertidos se almacenan en un SGBD relacional, se comprueban una serie de propiedades, entre las que se encuentra la **integridad referencial**³³. Si, por ejemplo, se intentasen guardar mediciones de la calidad del aire en Madrid sin tener datos de esta localización en base de datos, se violaría la restricción de integridad referencial y, por consiguiente, no se almacenarían dichas mediciones. Por este motivo, se implementa a mayores una operación –opcional– `_check_dependencies_satisfied`, que permite comprobar, como medida cautelar, si la información a la que deberán hacer referencia los valores a convertir está presente en el SGBD.

El resultado de esta operación debe almacenarse en el atributo de instancia `dependencies_satisfied`. Por defecto, toma el valor `True`, lo que implica que sólo en caso de que existan dependencias con otros datos deberá sobrescribirse esta operación.

A continuación, se muestra un ejemplo de uso de esta operación, para el escenario anteriormente mencionado de los datos de la calidad del aire y su dependencia con datos de localizaciones:

```
class _AirPollutionDataConverter(DataConverter):  
  
    def _check_dependencies_satisfied(self):  
        self.dependencies_satisfied = Location.objects.exists()
```

Código 19: Implementación del método `_check_dependencies_satisfied`.

En el ejemplo anterior, la clase `Location` hace referencia a una entidad persistente mapeada a base de datos empleando el ORM de **Django**. En la siguiente subsección, se comentarán en detalle los aspectos de Django usados por el subsistema.

³³ **integridad referencial**: alude a una propiedad de los SGBD relacionales, que establece que “una tupla en una relación que haga referencia a otra relación deberá referirse a una tupla existente en esa relación”, según lo visto en la asignatura *Bases de Datos*. En otras palabras, que una clave foránea siempre debe referenciar a una clave primaria existente, o bien ser nula.

8.3.2. El ORM de Django

El uso del ORM de Django simplifica enormemente el arte de implementar una base de datos, ya que permite definir tablas e índices empleando objetos Python.

Una particularidad del uso de **PostgreSQL** es que Django implementa características especiales para este SGBD: campos (e.g. `ArrayField` o `JSONField`), índices (e.g. `BrinIndex`, `GinIndex`), etc.

Para Django, cada entidad persistente es un **modelo**, y se define en una clase. Además, se emplean *meta-clases* para declarar índices, restricciones...; aunque también se permite el uso de *scripts* SQL para la creación de tablas, procedimientos almacenados, etc. en caso de que la lógica a implementar sea demasiado compleja.

No obstante, este no es el caso del sistema. El modelo **entidad-relación** resultante es sencillo, como se puede apreciar en el siguiente diagrama:

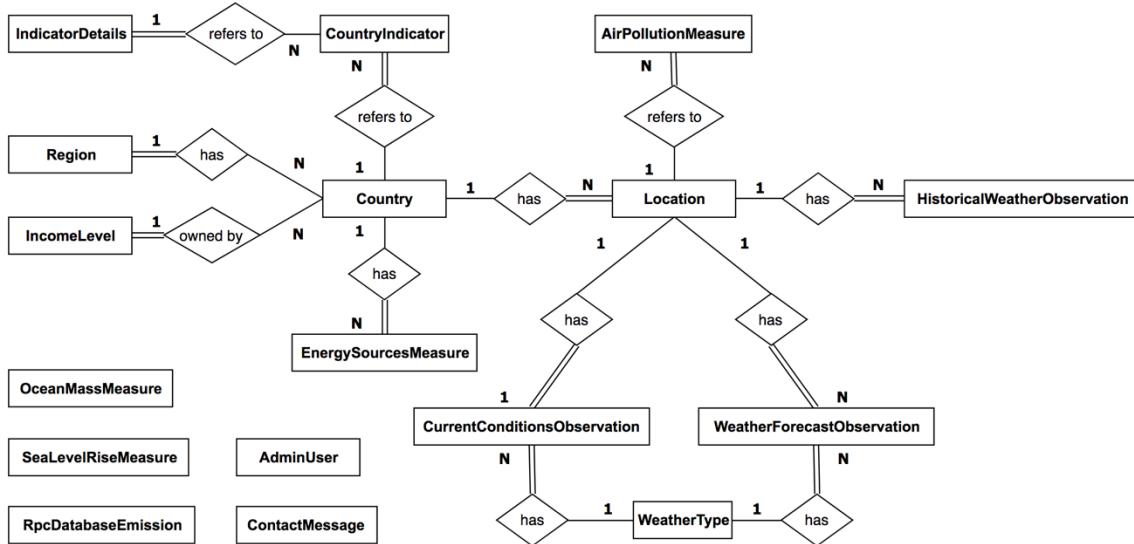


Ilustración 23: Diagrama entidad-relación del sistema.

Debe recordarse que la finalidad del sistema es la de divulgar información. Salvo por el caso de **ContactMessage**, que representa un mensaje que el usuario puede crear para contactar al *staff* de la aplicación, las acciones del usuario no generarán, modificarán o eliminarán datos. Esto solo podrá hacerlo el Subsistema de Conversión de Datos.

La elección de qué atributos poseerá cada entidad responde, tanto a detalles de implementación, como a la información que deberá ser mostrada por la aplicación web, detallada en la sección [4.2. Elaboración del prototipo](#).

A continuación, se muestra la definición de una entidad empleando Django:

```
class WeatherForecastObservation(models.Model):
    location = models.ForeignKey(Location, on_delete=models.CASCADE)
    date = models.DateField(db_index=True)
    time = models.TimeField(db_index=True)
    temperature = models.SmallIntegerField(null=True)
    pressure = models.SmallIntegerField(null=True)
    humidity = models.PositiveSmallIntegerField(null=True)
    wind_speed = models.PositiveSmallIntegerField(null=True)
    wind_degrees = models.PositiveSmallIntegerField(null=True)
    sunrise = models.DateTimeField(null=True)
    sunset = models.DateTimeField(null=True)
    weather = models.ForeignKey(WeatherType, on_delete=models.SET_NULL,
                                null=True)
    class Meta:
        unique_together = ('location', 'date', 'time')
```

Código 20: Definición de una entidad empleando Django.

Respecto al fragmento de código anterior, cabe destacar una serie de consideraciones:

- Cada elemento dentro de la clase `WeatherForecastObservation` se corresponde con un atributo de entidad, y se mapeará a una columna en base de datos.
- Se definen los tipos de datos y restricciones asignando a cada atributo un objeto que especifica dichas características.
- En el caso de campos que conforman claves foráneas, se permite especificar la **acción referencial**³⁴ que tendrá lugar al borrar el valor al que la clave foránea apunta. Django soporta el uso de `CASCADE`, `PROTECT`, `SET_NULL`, `SET_DEFAULT` y `DO NOTHING`.
- La clase `Meta` es una *meta-clase* que permite definir restricciones de unicidad, índices para múltiples campos, el nombre de la tabla, el orden en el que se ordenarán los valores por defecto, permisos especiales, entre otros.

8.3.2.1. Claves primarias

Algo que merece la pena destacar es que, por defecto, Django añade un atributo `id` como clave primaria numérica auto-incrementada, en caso de que el desarrollador no proporcione explícitamente un campo anotado con `primary_key=True`.

³⁴ **acción referencial:** Cuando se modifican o eliminan tuplas de una relación referenciada por otra mediante una clave foránea, existen una serie de acciones que el SGBD puede realizar para cada tupla afectada. Ejemplos de acciones referenciales son el borrado en cascada o establecer los valores de clave foránea como nulos. El desarrollador deberá elegir la acción referencial más adecuada a cada caso.

Esta situación es aceptable cuando la clave primaria solo es una forma de garantizar **unicidad**. En estos casos, emplear claves numéricas aporta mayor eficiencia (de cara a realizar `JOIN` de tablas, comparaciones, etc.) respecto a otros tipos de datos. Este es el caso, por ejemplo, de la entidad `AirPollutionMeasure`.

No obstante, hay casos en los que la clave primaria sí es **representativa**, y no se hacen uso de claves primarias auto-generadas, sino de una clave candidata. Un ejemplo es el caso de la entidad `Country`, cuya clave primaria es el código ISO de dos letras del país, el cual es único.

8.3.2.2. *El script manage.py*

Django proporciona muchas de sus funcionalidades a través de un *script* ejecutable tanto desde la línea de comandos como desde código Python.

Ejemplos de acciones que se pueden realizar con este *script* son:

- El chequeo de si existen cambios en entidades y el traslado de dichos cambios a base de datos.
- La generación de una plantilla con todos los mensajes a traducir en un idioma.
- La ejecución de un servidor web en el entorno de desarrollo.
- La creación de un arquetipo para un proyecto.
- La ejecución de pruebas.

El Subsistema de Conversión de Datos emplea este *script* para generar automáticamente las tablas en base de datos.

Esta sub-sección cierra los aspectos en los que difieren los subsistemas detallados hasta ahora. Como ya se comentó con anterioridad, el resto de características son similares, y no merece la pena comentarlas en este documento.

8.4. *Bugs*

Desde la puesta en producción del Subsistema de Recolección de Datos (04/01/2018), han ido apareciendo *bugs* (errores).

Éstos se solventan nada más aparecen, dado que un nuevo error se convierte en una tarea de prioridad máxima. Una vez se ha probado que se ha solucionado el *bug*, se genera una nueva *release*, que se pone en producción. Puede obtenerse más información acerca de las *releases* en la sección [D.1. Etiquetado de releases](#).

En esta sección, se comentarán aspectos sobre errores que poseen la etiqueta **Critical**, dado que son de suma importancia, y condicionan la re-implementación de aspectos clave del sistema.

8.4.1. [BUG-032]

Este error supone más de 10 horas de trabajo, e implica la modificación de más de 1,5 GB de datos. Responde al título “*el campo ‘_id’ no es adecuado para realizar consultas paginadas*”.

El problema se debe a un fallo en la implementación de los **DataCollectors** relacionado con MongoDB.

En este SGBD, todo documento debe poseer un campo especial `_id`, que actúa de forma similar a una clave primaria en SGBD relacionales: es único, identifica al documento, se indexa de forma automática y no puede ser `null`. Además, es immutable. En caso de que un documento omita este valor, MongoDB lo añadirá automáticamente.

Este campo es de tipo arbitrario. Cualquier tipo de datos válido en BSON conformará un candidato válido. Por defecto, MongoDB emplea el formato `ObjectId`, ya que estos objetos son pequeños, generalmente únicos, fáciles de generar, y ordenados. Con un tamaño de 12 bytes, y compuesto por:

- 4 bytes que representan los segundos desde *Unix time*³⁵.
- 3 bytes, que representan el identificador de la máquina.
- 2 bytes, que representan el ID del proceso.
- 3 bytes, que representan un contador, que comienza en un valor aleatorio.

Trabajar con valores de este tipo es sencillo en Python:

```
>>> import bson  
>>> x = bson.ObjectId()  
>>> x  
ObjectId('5b116851454cd9ffd5cfe3dc')
```

Código 21: Creación de un objeto de tipo `ObjectId`.

En un primer momento, se usó como `_id` un `dict` compuesto por campos que, juntos, hacen único un valor. Por ejemplo, para una medición de la calidad del aire, un valor podría ser: `{"time_utc": 151957800000, "location_id": 2}`, dado que no puede existir más de una medición para la misma localización e instante temporal.

No obstante, dado que el componente API devuelve los datos de un **DataCollector** de forma **paginada**, ordenando por el campo `_id`; al poner en producción este

³⁵ **Unix time**: representa el número de segundos pasados entre el 1 de enero de 1970 a las 00:00, UTC. Esa instantanea corresponde al valor 0, y en el momento en el que se están escribiendo estas líneas, el valor 1527867173. Este valor también es conocido con el nombre de **epoch**.

componente junto con el Subsistema de Conversión de Datos, empiezan a surgir errores de tipo `django.db.IntegrityError`.

Resulta que valores recopilados con posterioridad pueden poseer un campo `_id` menor –a la hora de establecer comparaciones– que otros almacenados anteriormente, causando que se intente convertir varias veces el mismo valor, o que éste ni siquiera llegue a convertirse.

Se opta, entonces, por rediseñar los valores del campo `_id` a objetos de tipo `ObjectId`, debido a que éstos mantienen un **orden** de forma natural. Esto posee dos implicaciones:

1. Que todos los valores ya recopilados deberán cambiar también el valor del campo `_id` a un objeto de dicho tipo.
2. Que la lógica de negocio, que hacía uso del campo `_id` para realizar inserciones en MongoDB, verá reducido su rendimiento, dado que los valores que conformaban este campo dejan de estar indexados, y los tiempos de consulta se hacen mayores.

Para solucionar el primer problema, se implementa una opción especial en el *script* de despliegue del componente API, llamada `--adapt-legacy-data`. El algoritmo implementado realiza las siguientes operaciones:

1. Obtener el nombre de todas las *collections* con datos en MongoDB.
2. Obtener todos los valores con el campo `_id` de tipo distinto a `ObjectId`.
3. Para cada valor, crear un duplicado sin el campo `_id`. Este campo es immutable, por lo que la única forma de cambiarlo es creando un valor con los mismos campos, pero distinto `_id`. Se delega en MongoDB para establecer el valor del campo `_id` con un objeto de tipo `ObjectId`.

Un aspecto importante a detallar aquí es que se traen varios GB de datos a memoria, que se duplican al crear los nuevos valores. En la primera versión de esta utilidad, el uso de memoria era tan elevado que la máquina empezaba a paginar a disco, elevando el tiempo de operación a horas.

Esto hace que se idee una estrategia en la que, como mucho, se tengan x valores en memoria de forma simultánea, reduciendo el consumo de RAM y efectuando la operación en minutos.

4. Se eliminan todos los valores con el campo `_id` de tipo distinto a `ObjectId`.
5. Se emite un reporte indicando el número de valores adaptados de forma exitosa y errónea.

Tras la ejecución de esta operación, en MongoDB solo quedan valores con el campo `_id` de tipo `ObjectId`, con lo que se asegura el orden de los elementos

recopilados. La opción `--adapt-legacy-data` es **eliminada** antes de la siguiente *release*.

El segundo problema se resuelve añadiendo en el *script* de despliegue del Subsistema de Recolección de Datos una opción `--create-indexes`, que indexe los campos requeridos por la lógica de negocio.

Para ello, se recurre a un fichero de configuración `deploy.config`, que emplea la sintaxis YAML usada a lo largo de todo el proyecto. En él, se define la estructura de los índices de forma sencilla:

```
MONGODB_INDEXES:  
  air_pollution:  
    keys:  
      - time_utc: -1  
      - location_id: 1  
    unique: true  
  # More indexes go here...
```

Código 22: Extracto del fichero de generación de índices de MongoDB.

Pese a la poca información necesaria para generar los índices, éstos permiten **incrementar el rendimiento** de las operaciones en MongoDB de forma significativa.

La utilidad que genera los índices emplea la información de configuración de la siguiente manera:

- El campo `keys` define los atributos que forman parte del índice. Cada atributo es un par clave-valor en el que se indica el nombre del atributo y se define el sentido de ordenación (1: ascendente, -1: descendente).
- El valor de `unique` sirve para establecer una restricción de unicidad sobre los valores del índice, de modo que no existan duplicados. Si un valor duplicado intenta ser insertado, se lanzará un error.

Además, se otorgan nombres significativos a los índices. Dado que MongoDB no restringe la longitud de los nombres, se emplean todos los campos para generarlo, de la forma: `<collection name>_index_on_<key_1>_<key_2>...<key_n>`. En el caso del índice definido en el fragmento del fichero de configuración mostrado arriba, en esta misma página, el índice poseerá el nombre: `air_pollution_index_on_time_utc_location_id`.

8.4.2. [BUG-040]

Este *bug* supone una vulnerabilidad de seguridad importante, y surge tras recibir un correo electrónico de parte de REDIRIS, indicando que el servicio MongoDB –desplegado en el servidor de producción– era accesible desde Internet.

Para solventar este problema, se crea un procedimiento que otorga mayor seguridad en caso de que todos los componentes se desplieguen en la misma máquina. Éste consiste en que el sistema solo expondrá la aplicación web, dado que el resto de componentes son internos y no deberían ser alcanzables. El procedimiento se puede invocar mediante la opción `--hide-containers` del *script* de instalación de cualquier componente del sistema.

Nótese que, si se realiza un despliegue distribuido, componentes como las bases de datos deberán ser accesibles a través de la red, por lo que no deberá usarse esta opción.

Actualmente, se sigue la primera aproximación. Por lo tanto, cualquier componente salvo la aplicación web debe estar *oculto*.

8.4.2.1. Docker y la seguridad

Docker permite definir **políticas de acceso** a nivel de contenedor, de forma declarativa: el desarrollador indica lo que necesita, y Docker realiza las operaciones necesarias para ello. Además, permite definir **redes**, y situar los contenedores en ellas.

Docker posee dos formas de exponer un puerto. Puede realizarse usando:

- **expose**: El puerto sólo se expone a los contenedores de la misma red.
- **ports**: El puerto será accesible por cualquier contenedor o servicio externo dependiendo de la IP que se emplee para definir la exposición de puertos. Por ejemplo, `ports: "0.0.0.0:27018:27017"` hará que el servicio MongoDB sea alcanzable desde cualquier máquina, en el puerto 27018.

Cada contenedor posee una dirección IP única dentro de la red. Esto hace que, independientemente de si el contenedor es accesible o no, el resto de contenedores en la misma red puedan comunicarse con éste a través de la dirección IP interna.

A la hora de efectuar el despliegue del sistema, todos los contenedores pertenecen a la misma red, `climatechangeapp_network`. Los contenedores que puedan ser alcanzables desde el exterior siempre expondrán sus puertos mediante `ports`. Si se usa `--hide-containers`, el contenedor solo será alcanzable desde `127.0.0.1 (localhost)`. En caso contrario (comportamiento por defecto), éste será alcanzable desde cualquier punto de la red, dado que se emplea la máscara `0.0.0.0`.

Tras realizar una serie de pruebas con la herramienta `nmap` –empleando las tácticas vistas en la asignatura *Legislación e Seguridad Informática*–, se comprueba que los puertos anteriormente alcanzables se encuentran ahora filtrados, resolviendo el *bug* y eliminando la vulnerabilidad.

8.5. Escalabilidad

Los dos subsistemas comentados hasta el momento son escalables. La escalabilidad es una necesidad cuando el número de `DataCollectors` o `DataConverters` es suficientemente grande que hace que la duración de las ejecuciones aumente tanto como para que un número significativo sea abortado automáticamente.

Por lo tanto, será necesario dividir el número de `DataCollectors`/`DataConverters` en múltiples instancias, y desplegarlas todas de forma conjunta, en la misma máquina o de forma distribuida.

No obstante, para escalar un subsistema (e.g. el Subsistema de Recolección de Datos) deben cumplirse una serie de requisitos:

- Todos los módulos deben poseer un **nombre único**. Cada `DataCollector` se implementa en un módulo, y se vincula con una *collection* en MongoDB con el mismo nombre. Por lo tanto, si varios `DataCollectors` se llamasen igual, almacenarían datos en la misma *collection*, produciendo colisiones y errores.
- Cada instancia del subsistema debe poseer un **identificador único**: Esto puede conseguirse mediante la propiedad `SUBSYSTEM_INSTANCE_ID` del fichero de configuración del subsistema, situado en `/<subsystem>/config/config.config`, siendo `/` el directorio raíz del sistema.

8.5.1. Consecuencias

Existen varias consecuencias producidas como resultado de escalar cualquier subsistema. Son:

- Los *scripts* CRON de ejecución periódica del subsistema deben ser modificados (esto solo es necesario si se escala el subsistema de forma no distribuida). Esto responde a que Docker no permite nombrar múltiples contenedores con el mismo nombre, aunque esto puede solucionarse de forma sencilla añadiendo al apelativo del contenedor el ID de su subsistema (e.g. `data_gathering_subsystem_2`).
- Si se escala el Subsistema de Recopilación de Datos, forzosamente ha de escalarse el Subsistema de Conversión de Datos.

- Si se recuerda la estructura de directorios, detallada en la sección [7.7.3. Logs y ficheros de estado](#), debe comentarse que, al escalar el subsistema ésta sufre modificaciones: los ficheros de *log* y estado de cada subsistema se almacenan en su propio subdirectorio, de la forma:

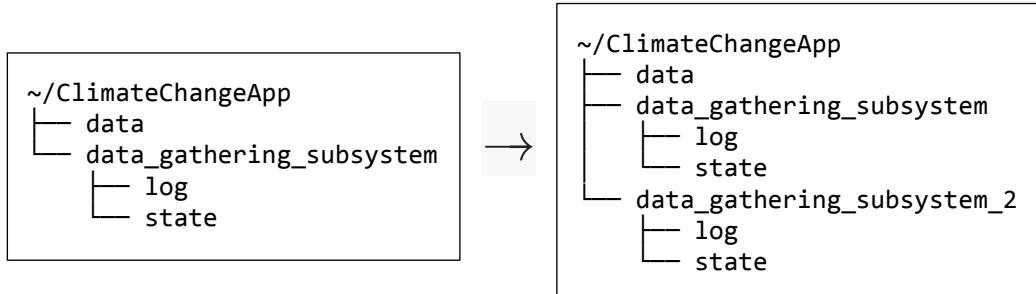


Ilustración 24: Estructura de directorios del sistema tras haber escalado un subsistema.

Esto favorece la organización de los ficheros de *log* y estado.

8.6. Integración e Inspección Continua

El requisito funcional USR-09 requiere que “*la calidad del software sea medida y seguida*”. La mejor forma de conseguir este objetivo es mediante la adopción de las prácticas de Integración Continua e Inspección Continua.

Dado que su configuración e instalación no resulta trivial, se trasladan todos los aspectos relacionados con este tema al [ANEXO B. INTEGRACIÓN E INSPECCIÓN CONTINUA](#).

8.7. Pruebas

La forma de implementar escenarios de prueba para los componente API y Subsistema de Conversión de Datos sigue fielmente lo explicado con anterioridad, en la sección [7.6. Pruebas](#).

La similitud con los componentes anteriores, y el mayor grado de familiarización con las tecnologías y técnicas de prueba permiten agilizar su implementación. Con todo, en este *sprint* se destinan cerca de **20 horas** al diseño de casos de prueba.

Como resultado, se obtienen **51** escenarios para el componente API, logrando niveles de cobertura de decisión del **89,9%**. En el caso del Subsistema de Conversión de Datos, sus **129** escenarios de prueba otorgan niveles de cobertura del **89,4%**.

8.8. Despliegue

De cara al despliegue de los nuevos componentes del sistema, se generan nuevos *scripts* de instalación. Cada componente posee su propio *script*, además de que existe un *script* general que permite la instalación del sistema en conjunto.

Se añaden nuevas opciones que permiten personalizar la instalación. Las dos más importantes son:

- `--hide-containers`: Esta opción ha sido comentada previamente en la subsección [8.4.2. \[BUG-040\]](#).
- `--external-mongodb-server` y `--external-postgres-server`: Permiten indicar que los componentes MongoDB y/o PostgreSQL se encuentran instalados en otra máquina, evitando crear nuevos contenedores locales. Estas opciones tienen sentido cuando se han escalado los subsistemas de forma distribuida. Para mayor información, el lector puede acudir a la secciones [8.2.2. Tokens y ámbitos](#), y [8.5. Escalabilidad](#).

Además de la puesta en producción del componente API y del Subsistema de Conversión de Datos, en este *sprint* se despliegan los servicios de **Jenkins** y **SonarQube**, relacionados con Integración Continua e Inspección Continua, comentados en la sección [8.6. Integración e Inspección Continua](#).

9. CUARTO Y QUINTO *SPRINT*

Como resultado del tercer *sprint*, se obtiene:

- ✓ La implementación del componente API y del Subsistema de Conversión de Datos.
- ✓ El despliegue de los componentes anteriores, en su versión 2.0.
- ✓ Pruebas sobre los componentes desarrollados.
- ✓ Nuevos diagramas y documentación.

El *sprint* 4 comienza el 18/03/2018 y finaliza el 12/05/2018, con un esfuerzo estimado de 154 horas*hombre, y un total de 5 tareas; mientras que el *sprint* 5 comienza el 13/05/2018 y tiene como fecha límite la de depósito, el 21/06/2018; y cuenta con 14 tareas, estimadas en 116 horas* hombre.

En la reunión *Sprint Planning* previa a la acometida del *sprint* 4, se contempla abordar el total de los requisitos funcionales restantes:

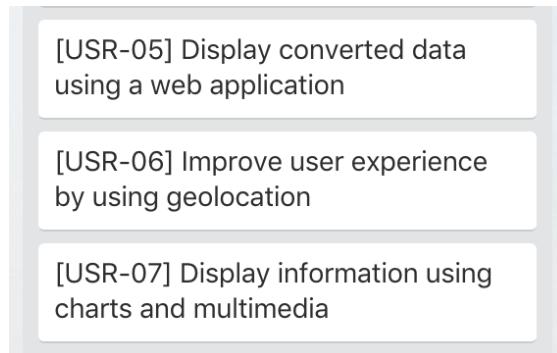


Ilustración 25: User stories a implementar durante el sprint 4.

Además, se ha tenido una reunión de *Retrospectiva del Sprint*, donde se ha decidido **refactorizar** la implementación de algunos componentes.

En el *sprint* 4 se implementa el grueso de las funcionalidades (sobre todo, la aplicación web), mientras que el *sprint* 5 se realizan pruebas, el despliegue final del sistema, y se mejoran aspectos ya implementados.

A continuación, se detallarán los aspectos relacionados con la acometida de estos *sprints*.

9.1. *Scripts* de instalación

El requisito funcional USR-08 requiere que el despliegue del sistema debería ser automático. Esto se consigue mediante el uso de *scripts* de instalación.

Además de la automatización, el uso de *scripts* también aporta mayor grado de personalización y seguridad, ya que:

- Posibilita el despliegue del sistema de forma local, distribuida y/o por componentes.
- Permite pasar parámetros de configuración por línea de comandos para dar soporte a algunas funcionalidades, en lugar de usar ficheros.

En las primeras versiones de los *scripts*, el paso de parámetros se realizaba de la forma `./install.sh PARAM=VALUE`. En el *sprint 4*, se realiza una **refactorización** para que los *scripts* sigan el estándar GNU para programas en línea de comandos [20]. Básicamente, todos los *scripts* deben poseer las opciones `--help` y `--version`. Además, el paso de parámetros debe ser de la forma `./install.sh --param value`.

Con todo, las opciones que permiten los *scripts* son mucho mayores:

```
$ ./data_conversion_subsystem/install.sh --help
Installs the Data Conversion Subsystem component.

> usage: install.sh [-h] [--help] [--version] [--api-ip xxx.xxx.xxx.xxx]
                     [--api-port xxxx] [--deploy-args "<args>"] [--external-postgres-
                     server] [--hide-containers] [--macos] [--postgres-ip xxx.xxx.xxx.xxx]
                     [--postgres-port xxxx] [--perform-deploy-actions] [--root-dir <path>]
                     [--show-ip] [--uid <val>]

• -h, --help: shows this message.
• --version: displays app's version.
• --api-ip xxx.xxx.xxx.xxx: specifies the IP address of the API server.
  Defaults to the machine's IP address. Invoke "./install.sh --show-ip"
  to display the resolved IP address.
• --api-port xxxx: sets the exposed API port. Defaults to 5000.
• --deploy-args "<args>": enables "Expert Mode", allowing to pass custom
  args to the deploy script. Defaults to "--all --with-tests". Must be
  used in conjunction with --perform-deploy-actions.
• --external-postgres-server: indicates that the PostgreSQL server is ex-
  ternally provided, and does not create a Docker container.
• --hide-containers: makes containers not reachable from the Internet.
• --macos: sets "docker.for.mac.host.internal" as the local IP address.
• --postgres-ip xxx.xxx.xxx.xxx: sets the IP address of the PostgreSQL
  server. Defaults to the machine's IP address.
• --postgres-port xxxx: sets the exposed PostgreSQL port. Defaults to
  5432.
• --perform-deploy-actions: installs the application performing all de-
  ploy steps. By default, deploy steps are skipped.
• --root-dir <path>: installs the Application under a custom directory.
  Defaults to "~/ClimateChangeApp".
• --show-ip: displays the IP address of the machine. If multiple IP's,
  displays them all.
• --uid <val>: sets the UID of the user executing the Subsystem. Using
  "0" or "root" is not recommended. Defaults to the current user's UID.
```

Código 23: Opciones del script de instalación del Subsistema de Conversión de Datos.

Cada componente posee su propio *script*, aunque se proporciona otro adicional que permite instalar el sistema en su totalidad, en la misma máquina.

También se proporciona un *script* especial, `install-ci.sh`, que se emplea para realizar el despliegue completo del sistema en el entorno de **Integración Continua**. Para mayor información, el lector puede acudir a la sección [8.6. Integración e Inspección Continua](#).

9.2. *Healthcheck*

El requisito no funcional RNF-01 *Disponibilidad* sostiene que “*el sistema debe estar operativo de forma permanente, haciendo hincapié en la recopilación de datos sin interrupciones*”.

Esto se puede lograr de forma *sencilla* mediante el uso de Docker y su mecanismo de **healthcheck**.

Este mecanismo permite conocer, de forma periódica y programática, el estado de los contenedores; permitiendo a componentes externos monitorizar dicho estado, y realizar acciones como el reinicio del servicio.

Por ejemplo, en el fichero **Dockerfile** que genera la imagen del componente API, se añaden las siguientes líneas:

```
# Adding HEALTHCHECK
RUN chmod +x /ClimateChangeApp/code/api/docker-healthcheck
HEALTHCHECK CMD ["/ClimateChangeApp/code/api/docker-healthcheck"]
```

donde `docker-healthcheck` es un *script bash* que contiene:

```
#!/bin/bash

curl -s http://127.0.0.1:${API_PORT:-5000}/alive | grep true
exit $?
```

Código 24: Implementación del healthcheck para el componente API.

El *script* comprueba que el componente API esté activo mediante la llamada al endpoint `/alive`, comentado con anterioridad en la sección [8.2.3. Operaciones](#).

Con estas escasas líneas de código, Docker monitorizará el contenedor cada 30 segundos, y determinará que su estado es `healthy` –si el código de salida es `0`–, o `unhealthy` –si el código de salida es `1` o mayor, y se ha dado 3 veces seguidas la misma situación–.

Esto permite que el componente `auto_heal` monitorice y reinicie los servicios con estado `unhealthy`, aumentando la disponibilidad del sistema.

9.3. Subsistema de la Aplicación Web

Sin duda, los esfuerzos más notables de los *sprints* 4 y 5 se realizan de cara a la implementación de la aplicación web.

Pese a que ya se cuenta con un diseño inicial, detallado en la sección [4.2. Elaboración del prototipo](#), el *Equipo* se enfrenta a los siguientes retos:

- El uso de una tecnología totalmente desconocida.
- La poca experiencia en el desarrollo de *front-end*.
- Los requisitos no funcionales RNF-03 *Simplicidad*, y RNF-04 *Usabilidad*, que exigen el uso de gráficos, diagramas y un diseño *Mobile-First*.

En las subsecciones siguientes, se comentarán las *features* más destacables de este subsistema.

9.3.1. Mobile-First

Un sitio web *Mobile-First* es aquel que está optimizado para ser visualizado por dispositivos móviles, y que se expande o reorganiza sus componentes para `viewports`³⁶ mayores.

El *framework* web **Bootstrap**, en su versión 4, es *Mobile-First*. Los nombres de las clases CSS aplican, por defecto, a los dispositivos móviles, empleando los sufijos `-xs`, `-md`, `-lg` y `-xl` para los de mayor tamaño (e.g. `col` y `col-md`).

Todas las páginas web de la aplicación siguen de forma **estricta** un diseño *Mobile-First*, llegando hasta el punto de descartar componentes o librerías que no cumplan con este requisito.

El uso de este diseño atraerá mayor número de visitas a la aplicación, dada la proliferación de dispositivos móviles de la última década, como se vio en la asignatura *Interfaces Persona-Máquina*, y en [17].

Además, mejora en gran medida la usabilidad, ya que resultaría prácticamente imposible interactuar con la aplicación si esta tuviera un diseño estático: los gráficos no se apreciarían, el texto resultaría ilegible, etc.

³⁶ **viewport**: Hace referencia al área visible de un sitio web, que varía en función del tamaño de la pantalla del dispositivo. Los sitios web actuales escalan la página para que ésta quepa en cualquier `viewport`.

En HTML5, se puede establecer esta propiedad mediante el *tag* `<meta name="viewport" content="width=device-width, initial-scale=1">`.

A continuación, se muestra un ejemplo que evidencia el carácter *Mobile-First* de la aplicación:



Ilustración 26: Visualización de una página de la aplicación web en un iPhone 5S vs iPad.

9.3.2. Geolocalización

El requisito funcional USR-06 establece la necesidad de “*mejorar la experiencia del usuario empleando para ello la geolocalización*”.

Se hace uso del API de HTML5 de geolocalización para obtener las coordenadas del usuario, y así mostrar los datos de la localización más cercana a la ubicación actual del mismo.



Ilustración 27: Botón que permite activar la geolocalización.

Al hacer *click* en el botón anterior, el usuario enviará en una petición HTTP POST sus coordenadas. En el lado servidor, se obtendrán los datos de la localización

más cercana a éste, y se guardará el ID de la misma en la **sesión web**, a través de *cookies*. De esta forma, en las peticiones siguientes, obtener la localización más cercana al usuario se reduce a una simple consulta por clave primaria.

Importante: Por motivos de seguridad, los navegadores solo permiten obtener las coordenadas geográficas del usuario a través del API de geolocalización si se emplea el protocolo seguro **HTTPS**, que mantiene la información cifrada.

9.3.3. Google Maps

A la hora de mostrar localizaciones, además de una búsqueda habitual por palabras clave, se permite al usuario seleccionar una localización monitorizada mediante el uso de Google Maps, haciendo *click* sobre uno de los marcadores  :



Ilustración 28: Marcadores de localizaciones monitorizadas en Google Maps.

Nótese que, en el caso anterior, la geolocalización está activada; y debajo del marcador relativo a la ciudad de A Coruña aparece el símbolo , que indica las coordenadas actuales del usuario.

9.3.4. Uso de gráficos y diagramas

Para facilitar la interacción con el usuario, y mejorar su experiencia, se emplean elementos multimedia a favor del uso de texto plano. Esto, además de satisfacer el requisito funcional USR-07, dota a la aplicación de una mejor apariencia.

Todos los gráficos y diagramas empleados en la aplicación web son *responsive*, por lo que se adaptarán al tamaño del dispositivo; verificando, además, que éstos responden a interacciones táctiles.

Se ha elegido emplear la librería **nvd3** en lugar de otras como **Google Charts** precisamente porque ésta última: ni es *responsive*, ni responde a eventos táctiles.

Una de las ventajas de las librerías **JavaScript** es que la labor computacional necesaria para representar los datos se realiza en el lado cliente, aliviando la carga del servidor y aumentando la interacción con el usuario.

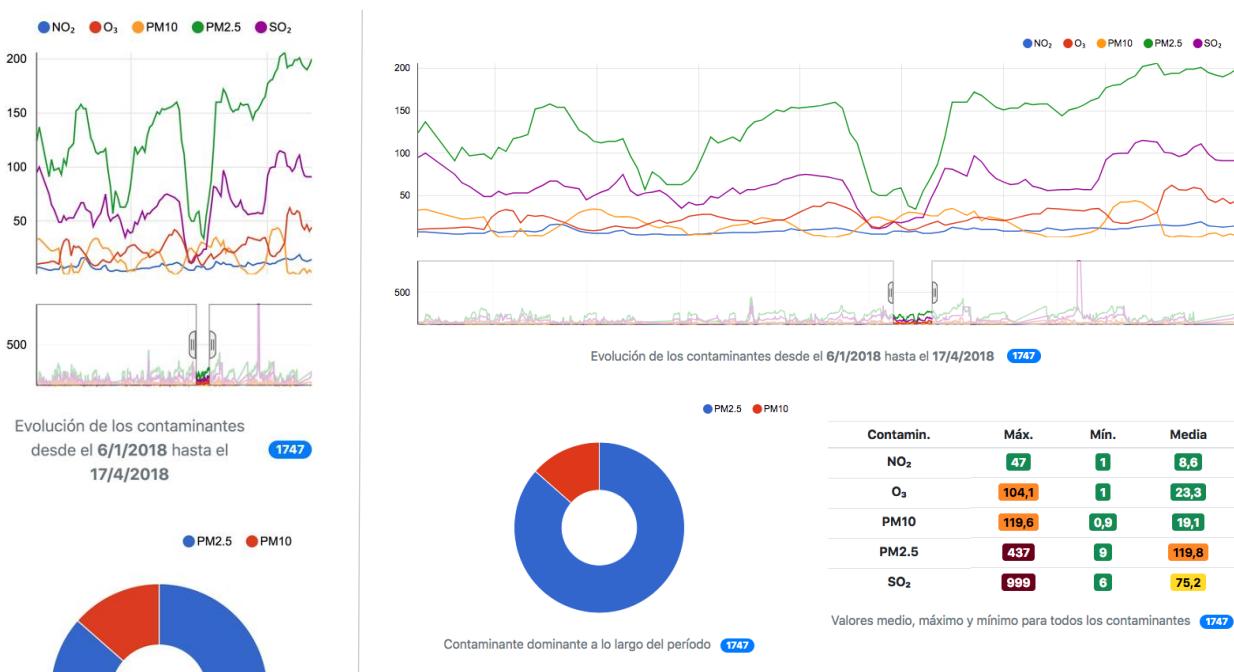


Ilustración 29: Gráficos en un iPhone 6/6s/7/8 vs dispositivo de escritorio.

9.3.5. Validaciones

De cara a minimizar el número de errores, se realizan dos tipos de validaciones:

- En el lado **cliente**, se verifican: campos requeridos, longitud de texto, rangos de fechas, etc.
- En el lado **servidor**, se vuelven a verificar todos los campos, además de realizar otras validaciones relativas a la lógica de la aplicación.

Para este último caso, se implementa un módulo **validators.py**, junto con una clase **ValidationError**, que representa cualquier error de validación detectado por las funciones de este módulo.

La forma de actuar ante cualquier petición por parte del cliente es la siguiente:

1. Si los datos son inválidos en el lado cliente, no se realiza la petición.

2. Si llega una petición al servidor, se validan nuevamente los datos. Si éstos son incorrectos, se proporciona *feedback* al cliente, de la forma:

E-mail:

nombre@ejemplo.com

La dirección de e-mail debe ser válida, y su longitud debe ser de 5-60 caracteres, y debe incluir un "@".

Le responderemos a dicha dirección tan pronto como sea posible.

Ilustración 30: Feedback negativo al detectar datos inválidos.

3. En caso de que los datos sean válidos, se ejecuta la llamada al servicio que recupera los datos a mostrar, y se devuelve una respuesta al cliente. Además, también se proporciona *feedback* al usuario:

Temperatura (mínima)	▼	2011	Máx.
	Bien!		Bien!

Ilustración 31: Feedback positivo tras enviar datos válidos.

9.3.6. AJAX

AJAX responde a las siglas de *Asynchronous JavaScript And XML*, y se emplea para aumentar la interacción con el usuario.

Con AJAX, las peticiones del cliente se envían de forma **asíncrona** (en segundo plano), son procesadas por el servidor, y los datos de la respuesta son manejados mediante código JavaScript. Esta técnica permite recargar zonas de la página sin tener que cargar la página completa, leer datos del servidor una vez se ha cargado la página, o enviar datos al servidor.

En la aplicación, se empleará mayoritariamente para enviar los datos que se representarán mediante gráficos y/o diagramas al cliente, en formato JSON; aunque también se le dará uso en formularios, o para cargar la lista de localizaciones.

Esto permite redibujar los elementos multimedia tras haber aplicado filtros sin recargar la página, disminuyendo el consumo de datos en más de un 90% para peticiones subsecuentes.

9.3.7. Caché

El uso de un mecanismo de **caché** que permita almacenar en memoria los datos más consultados disminuye de forma significativa la carga sobre el SGBD.

Django proporciona un API que permite acceder de forma sencilla a utilidades de caché de terceros; permitiendo realizar lecturas, borrados y escrituras, además de configurar el tiempo en el que cada valor permanecerá almacenado en la caché.

La estrategia que se sigue a la hora de integrar el mecanismo de caché con la implementación de los servicios ya existentes, es la de minimizar su **acoplamiento**:

- Se crean **clases abstractas** para cada servicio susceptible de ser cacheado (debe recordarse que en Python no existe el concepto de interfaz).
- Los servicios originales implementan dichas clases abstractas.
- Los servicios que hacen uso de la caché también implementan las clases abstractas, incorporando la gestión de la caché, pero **delegando** la funcionalidad en los servicios originales.
- Se utilizan clases **factoría** para instanciar la implementación concreta del servicio, implementando el Patrón de *Inversión del Control* (IoC)³⁷. De esta forma, los módulos que hacen uso de servicios no necesitan instanciarlos de forma directa, sino a través de la factoría.
- Se hace uso, nuevamente, de la *Programación Orientada a Aspectos* (AOP), para interceptar las llamadas a los métodos del servicio que hacen uso de la caché, y acceder a ésta de forma transparente y declarativa.

Todos estos aspectos pueden apreciarse en el siguiente ejemplo:

```
class CacheLocationService(AbstractLocationService):  
  
    @staticmethod  
    @fetch_from_cache(key='locations', timeout=60 * 15)  
    def get_all_locations(fields=None, order_by: tuple = ('name',)) -> list:  
        return LocationService.get_all_locations(fields, order_by)
```

Código 25: Implementación de un método del servicio que hace uso de la caché.

Donde:

- `CacheLocationService`, `LocationService` son implementaciones del servicio.
- `AbstractLocationService` es la clase abstracta de la que heredan las clases anteriores.
- `@fetch_from_cache` intercepta la llamada al método `get_all_locations`, busca la clave `locations` en la caché y la devuelve, de existir. En caso contrario, ejecuta la llamada al servicio `LocationService`, escribe el valor en la caché –válido durante `timeout` segundos–, y devuelve el resultado.

El uso de un servicio u otro se determina mediante el parámetro de configuración `USE_CACHE` del fichero de configuración del subsistema, ubicado en el directorio de configuración del subsistema: `/web/climate/config/config.config`.

³⁷ este Patrón de Diseño permite eliminar dependencias del código, favoreciendo el uso de interfaces y clases abstractas sobre implementaciones concretas.

Por ejemplo, las instancias del servicio `AbstractLocationService` pueden obtenerse mediante el código siguiente:

```
from .services.factories import LocationServiceFactory  
  
location_service = LocationServiceFactory.get_instance()
```

Código 26: Obtención de una instancia del servicio mediante una clase factoría.

9.3.8. Internacionalización (i18n)

Django proporciona un excelente soporte de cara a la localización (i10n) e internacionalización (i18n) de aplicaciones web.

Al contrario que otros *frameworks*, con Django se pueden obtener todos los mensajes a internacionalizar –de todas las páginas– mediante un solo comando. Se genera un fichero `.po` con las claves de los mensajes –texto del mensaje en inglés–, y se dejan espacios en blanco para llenar los valores. Además, se pueden usar **contextos**, que permiten indicar, para una misma clave, distintas traducciones según el valor de éste.

Para internacionalizar un mensaje, Django ofrece dos soluciones:

- `{% trans "<message>" %}`: Permite traducir mensajes simples, sin variables ni HTML. Es más eficiente que el siguiente.
- `{% blocktrans trimmed with a=b %}{{ a }}{% endblocktrans %}"`: Permite usar variables dentro del mensaje a traducir, así como *tags* HTML. Mediante el uso de `trimmed`, se eliminan todos los espacios en blanco del mensaje, muy útil para textos de más de una línea.

Por debajo, Django emplea la librería `gettext`.

Además, el *framework* formatea automáticamente fechas y números, de acuerdo al `locale` o información cultural del usuario (localización, i10n).

No obstante, esto solo es así en el lado servidor. En el lado **cliente**, al usar AJAX, para lograr este comportamiento deben implementarse métodos con **JavaScript**.

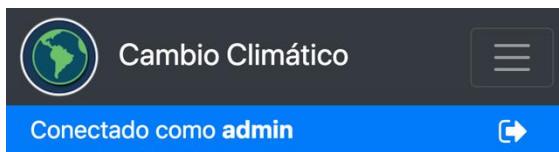
Se hace especial énfasis a lo largo de toda la aplicación web en la **localización** de fechas y números, y la **internacionalización** de mensajes.

9.3.9. Administración

La aplicación cuenta, además, con una zona de administración.

En el prototipo inicial se definieron varios aspectos que esta zona debía recoger. No obstante, por falta de tiempo, la mayoría no se han implementado.

Sí que cuenta con soporte para la autenticación y cierre de sesión, delegando en el API proporcionado por Django; además de una página que permite gestionar mensajes enviados por los usuarios de la aplicación:



Administrar mensajes

Ilustración 32: Interfaz de la página de gestión de mensajes de contacto.

Esta página permite:

- Visualizar, de forma intuitiva, el número de mensajes entrantes.
- Filtrar mensajes por contestados, entrantes o rechazados.
- Marcar un mensaje como contestado, descartado el mensaje, o contestarlo mediante correo electrónico.
- Eliminar un mensaje de forma permanente, si éste ha sido contestado o descartado.

9.3.10. Contador de *likes*

En la parte inferior de cada página, en el *footer*, se muestra información relativa al usuario, entre otros.

Esta sección también contiene un botón que permite a los usuarios de la aplicación indicar con un “*me gusta*” su satisfacción con la misma:



Ilustración 33: Botón “me gusta”.

Con el fin de evitar que cualquier usuario pueda mostrar demasiada satisfacción con la aplicación, desde el momento en el que éste da “*me gusta*” se guarda un valor en la **sesión** que inhabilita el botón hasta que la sesión caduca.

Por supuesto, la petición y la respuesta a este evento se gestionan de forma **asíncrona** con AJAX.

9.3.11. Reconocimiento de fuentes de información

De cara a satisfacer el requisito no funcional RNF-09 *Legalidad*, debe reconocerse la auditoría de todo dato mostrado que no haya sido generado por el sistema.

Esto se realiza conforme a las licencias de uso asociadas a los datos. Hay algunas que requieren citar la fuente, otras incluir un logotipo, etc.:

Los datos de la Calidad del Aire para **Beijing** están disponibles por cortesía de [The World Air Quality Index Project](#) y:

- [U.S Embassy Beijing Air Quality Monitor](#) (美国驻北京大使馆空气质量监测).
- [Beijing Environmental Protection Monitoring Center](#) (北京市环境保护监测中心).



Ilustración 34: Ejemplos de reconocimiento de fuentes de información.

9.3.12. Tratamiento de errores

Se hace un especial énfasis en el tratamiento de errores, independientemente de su tipo.

Como ya se comentó anteriormente, en la sección 9.3.5. *Validaciones*, se realizan **validaciones** en el lado cliente y el lado servidor. Además, se muestran **alertas**, si:

- Una búsqueda devuelve múltiples resultados o ninguno.
- La geolocalización no se activa correctamente.

- Una localización o país no posee algún tipo de datos.
- Surgen errores al cargar o enviar datos de forma asíncrona.
- El inicio de sesión resulta erróneo, en la zona de administración.
- No existen mensajes de acuerdo al filtro aplicado.
- Etcétera.

También se implementan **páginas de error** personalizadas, en caso de que una página no exista (HTTP **404**), se produzca una petición malformada (HTTP **400**), el usuario no posea privilegios para visualizar el contenido (HTTP **403**), o surja un error inesperado en el lado servidor (HTTP **500**).

Por ejemplo, si se intenta acceder a `/foo`, se mostrará la siguiente página:



[Take me back to home](#)

Ilustración 35: Página de error ante un recurso no encontrado.

9.4. Pruebas

Las pruebas de la aplicación web se realizan en el *sprint 5*, y su objetivo principal es verificar y encontrar errores en la implementación de los servicios y validadores.

Por falta de tiempo, no se han realizado pruebas con el nivel de detalle deseado, ya que se omiten las pruebas de aceptación. Éstas, podrían diseñarse empleando *frameworks* como **Selenium**, soportado por Django. Tampoco se han validado todos los módulos Python, aunque sí los más importantes.

Con todo, se dedican a este aspecto **18 horas**, obteniendo **204** escenarios de prueba, y logrando niveles de cobertura de decisión de más del **80%**.

El resultado obtenido es satisfactorio, ya que se encuentran **14** errores.

9.4.1. Pruebas contra base de datos

Por primera vez en todas las *suites* de prueba generadas hasta el momento, se realizan *test* que escriben en base de datos. Esto es así en las pruebas que verifican la implementación de los **servicios**.

Django proporciona un excelente soporte en estos casos: Existe una clase de pruebas ya implementada –`django.test.TestCase`–, que se encarga de:

- Crear la base de datos de pruebas: Por defecto, el nombre de la base de datos de prueba es el resultado de añadir `_test` al nombre original (e.g. `climatechange_test`). Además, se generan todas las tablas de forma automática. No obstante, puede sobrescribirse el comportamiento por defecto, mediante la propiedad `TEST` en el fichero de configuración de Django.
- Ejecutar todas las *suites* de pruebas, de forma ordenada. No obstante, Django también proporciona la clase `django.test.SimpleTestCase`, útil para escenarios de prueba en las que no se necesite acceder a base de datos. El *test runner*³⁸ ejecutará estos últimos escenarios al final.
- Destruir la base de datos al final de las pruebas: Independientemente del resultado de las pruebas, la base de datos de pruebas es eliminada. No obstante, puede mantenerse si así se desea, con la opción `--keep-db`.

Cada caso de prueba es independiente de los demás; pudiendo ejecutarse de forma aleatoria, dado que todos los cambios en base de datos son anulados después de cada *test* mediante un `ROLLBACK`.

Con todo, la implementación de un escenario de prueba se reduce a lo siguiente:

```
class LocationServiceTestCase(django.test.TestCase):  
  
    def test_get_historical_weather_stats_no_data(self):  
        HistoricalWeatherObservation.objects.all().delete()  
        stats = LocationService.get_historical_weather_stats(location_id=13)  
        self.assertEqual([], stats)
```

Código 27: Implementación de un escenario de prueba para una operación de un servicio.

³⁸ Un *test runner* es una clase especial que se encarga de ejecutar las *suites* de prueba. En el caso de la librería de pruebas `unittest` –usada por Django–, se emplea la clase `DiscovererRunner` que, dado un directorio raíz, escanea todos los ficheros de pruebas y ejecuta los *test*.

9.5. Despliegue

Dado que se pretende poner el Subsistema de la Aplicación Web en producción, debe tenerse un especial cuidado a la hora de efectuar su despliegue.

En el entorno de desarrollo, Django proporciona un servidor de aplicaciones que facilita la implementación, ya que se encarga de servir ficheros estáticos, mostrar el *stack trace* de los errores, reiniciarse ante cambios en el código...

No obstante, en producción, ésta no es, ni de lejos, la configuración recomendada. En un escenario real, una forma simple de desplegar una aplicación web, sería:

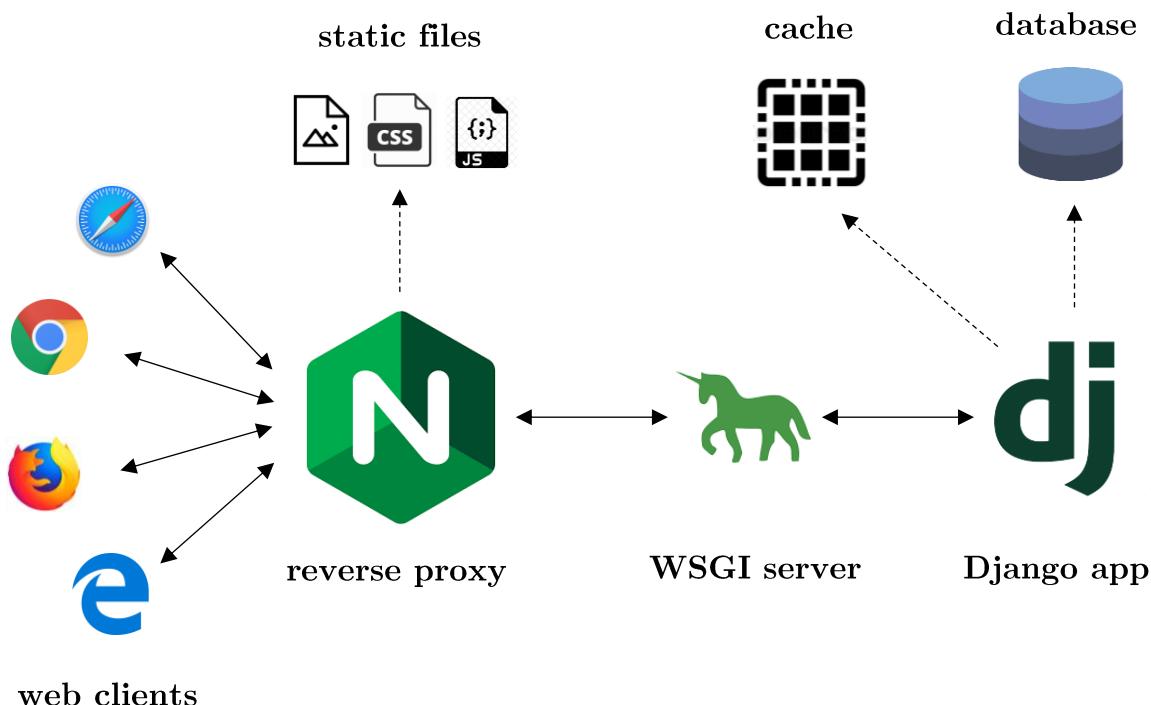


Ilustración 36: Vista de despliegue de la aplicación web.

En la vista anterior:

- **NginX** actúa como *reverse proxy*, recuperando los recursos solicitados por los clientes en nombre Gunicorn. Es, por tanto, la interfaz entre los clientes y el resto de componentes. Además, se encarga de servir directamente los **ficheros estáticos**.
- **Gunicorn** es un servidor de aplicaciones WSGI para *apps* escritas en Python. Su objetivo es redirigir las peticiones a la aplicación web. Es escalable, permitiendo definir múltiples trabajadores asíncronos, en distintos procesos. Se recomienda emplear la fórmula $N_{CPU} * 2 + 1$ para establecer el número de trabajadores.

- **Memcached** actúa como servidor de caché. En ella se escribirán los datos consultados con mayor frecuencia, para minimizar la latencia de dichas consultas y aliviar la carga del SGBD.

Esta configuración contribuye a satisfacer los requisitos no funcionales RNF-06 *Escalabilidad*, y RNF-08 *Seguridad*; dado que NginX y Gunicorn permiten establecer políticas de seguridad y configuración de múltiples instancias para satisfacer un mayor número de peticiones de clientes.

Como ocurre con el resto de componentes del sistema, también se emplea **Docker** para efectuar el despliegue de la aplicación web. Para ello, se emplean las imágenes oficiales de **NginX**, **Gunicorn** y **Memcached**.

Pese a que la arquitectura no es trivial, la puesta en marcha de la aplicación web se reduce a una simple llamada al *script* de instalación:

```
$ ./web/install.sh --perform-deploy-actions
```

Código 28: Invocación del script de instalación del subsistema.

Como resultado de la llamada anterior, se construirá una imagen del subsistema y desplegarán el resto de componentes. La aplicación web será accesible desde el puerto **80** de la máquina, y dependiendo de si la dirección IP de la misma es pública, también será accesible desde cualquier cliente. El resto de servicios quedarán **ocultos**, siendo **NginX** el único componente accesible del sistema³⁹.

9.6. Seguridad

Esta sección reúne los aspectos relacionados con la seguridad de la aplicación web.

9.6.1. Cross Site Request Forgery (CSRF)

Todas las peticiones HTTP **POST** requieren enviar un CSRF *token* para ser servidas. Esto contribuye a evitar ataques CSRF, en los que código malicioso intenta realizar acciones empleando las credenciales de un usuario autenticado.

9.6.2. HTTPS

HTTPS constituye la versión segura de HTTP, dado que las comunicaciones van cifradas mediante TLS/SSL.

³⁹ esto se verifica para una instalación local de todos los componentes sistema. En un entorno distribuido, otros componentes podrían ser accesibles desde Internet.

Lo que sí que se cumple, al menos con la configuración actual, es que el único punto de acceso a la aplicación web es el *reverse proxy* NginX.

Pese a que este aspecto satisfaría el requisito no funcional RNF-08 *Seguridad*, por falta de tiempo y recursos económicos, resulta imposible dotar de esta característica importante.

Una primera aproximación para la solución de este problema, sería la de crear un certificado HTTPS auto-firmado, como se ha visto en la asignatura *Legislación e Seguridad Informática*. No obstante, el navegador del usuario final no consideraría creíble este certificado, dado que se ha firmado por una Entidad Certificadora (CA) creada por el propio alumno.

Otra opción es la de conseguir un nombre de dominio, y obtener un certificado de algún proveedor que los emita. Se han estudiado iniciativas, como [Let's Encrypt](#), que proporcionan certificados de forma gratuita; en conjunción con servicios como [xip.io](#), que permiten obtener un nombre de dominio –rudimentario– de forma gratuita.

Este aspecto se incluye en la sección 11.1. *Trabajo futuro*, como un aspecto a incluir en futuras versiones del sistema.

10. ANÁLISIS DEL PROYECTO

Este capítulo tiene por finalidad analizar el proyecto en términos de esfuerzo, tiempo y coste; a la par que ofrecer datos cuantitativos de algunas métricas de calidad del producto obtenido.

Este apartado se sitúa a continuación del desarrollo del proyecto, con el fin de establecer comparaciones entre las estimaciones y el resultado final.

10.1. Planificación inicial

En materia de esfuerzo, la planificación se realiza, de forma ágil, siguiendo la metodología Scrum. Se definen y estiman las tareas de cada *sprint* en la reunión *Scrum Planning*, previa a la acometida de las iteraciones.

Con todo, se realiza una planificación inicial *grosso modo*, donde se definen una serie de hitos y etapas importantes, durante la etapa de análisis del proyecto:

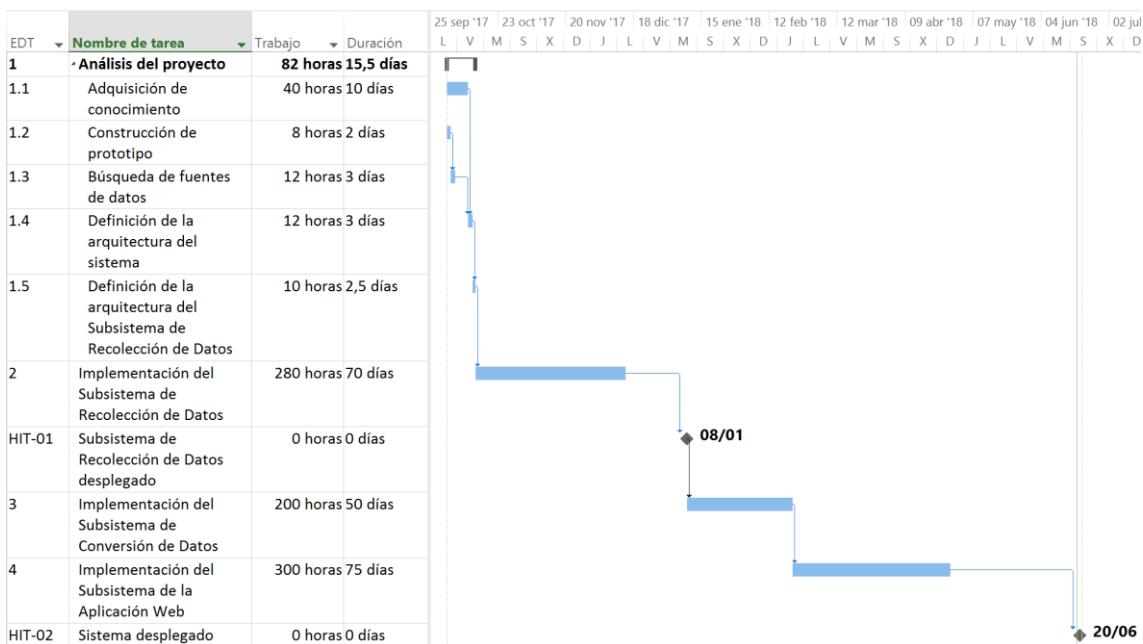


Ilustración 37: Diagrama de Gantt con la planificación grosso modo del proyecto.

Nótese que, en la fase de análisis del proyecto, la adquisición de conocimiento y la construcción del prototipo pueden realizarse en paralelo. El resto de actividades se consideran esencialmente secuenciales.

Otro aspecto a destacar es que, debido al desconocimiento inicial de las tareas concretas a realizar, las más importantes, en términos de esfuerzo, se reducen a tres: la implementación de los subsistemas.

Como se puede apreciar en el diagrama anterior, la planificación inicial contempla una holgura de más de un mes sobre la fecha de depósito del proyecto.

No obstante, debido a la dificultad inicial a la hora de estimar, se adelanta que esta planificación inicial **no** se cumple.

10.2. Planificación por *sprint*

A continuación, se desglosan los detalles del esfuerzo (planificado y real), a nivel de *sprint*:

Fase	nº tareas	Esfuerzo est. (h*h)	Esfuerzo real (h*h)	Desviación (%)
Análisis	14	82,0	78,0	-4,9
<i>sprint 1</i>	16	108,5	114,0	5,1
<i>sprint 2</i>	16	106,0	116,8	10,1
<i>sprint 3</i>	17	144,0	128,5	-10,8
<i>sprint 4</i>	6	154,0	185,5	20,4
<i>sprint 5</i>	18	116,5	78,2 ⁴⁰	-32,9
Total	87	711,0	701,0	-1,4

Tabla 3: Comparativa entre el esfuerzo estimado y real por sprint.

La planificación inicial, comentada en la sección anterior, estima el esfuerzo del proyecto en **862 h*h**.

Sin embargo, al planificar los *sprints*, el esfuerzo estimado pasa a **711 h*h**.

De todas formas, este no constituye el esfuerzo total del proyecto. Al esfuerzo real de la fase de análisis y los *sprints*, debe sumarse el *overhead* de las reuniones y el tratamiento de errores. Estos aspectos añaden 29,9 y 8,9 h*h, respectivamente; para situar el esfuerzo real total del proyecto en **739,8 h*h⁴¹**.

⁴⁰ este valor se debe a que no ha llegado el tiempo para realizar todas las tareas. El valor final del esfuerzo del *sprint* se interpola mediante una regla de tres.

⁴¹ siendo rigurosos, también debería sumarse al total el tiempo dedicado a la gestión del proyecto: imputación de tiempos, creación de tareas, la creación de este documento, etc.

10.3. Recursos

Como se ha comentado con anterioridad, en la sección [3.3. Adaptación de Scrum al proyecto](#), existen dos recursos dedicados al desarrollo del sistema:

- a) El alumno, Diego Hermida Carrera.
- b) El director, Diego Andrade Canosa.

Se contemplan, para ellos, los siguientes costes, extraídos de [21]:

Rol	Coste (€/h)
Alumno	16
Director	35

Tabla 4: Estimación de coste para los recursos del proyecto.

10.4. Calendarización

El inicio del proyecto se sitúa el **25/09/2017**⁴². Durante la acometida de éste, el tiempo que el alumno le dedica es variable, en función de su compromiso con otras actividades educativas y sus labores de representación estudiantil. Esto explica que, aunque el esfuerzo real del proyecto sea menor al estimado inicialmente, la fecha de fin de éste sea posterior a la estimada.

Con todo, la dedicación media al proyecto oscila entre **4-5** horas/día.

10.5. Coste

Con los datos de las subsecciones anteriores, se puede estimar el coste total del proyecto. Además, deben tenerse en consideración los siguientes aspectos:

- Se estima que el director dedica 1 hora semanal al proyecto, independientemente de las reuniones celebradas.
- Como coste de material, se incluye el ordenador portátil del alumno, y el monitor que emplea para trabajar.
- El coste directo del proyecto hace referencia a las horas de trabajo por parte del equipo de desarrollo.
- Como coste indirecto, se incluye el gasto en electricidad, agua y amortización de bienes, estimado en un 10% del total del coste directo, según [22].

⁴² se realiza algún esfuerzo previo a esta fecha, como un par de reuniones iniciales en las que se discute el dominio del proyecto. No obstante, su impacto real en el proyecto es despreciable.

Con todo, se muestra el resultado en la siguiente tabla:

Fecha de inicio	25/09/2017
Fecha de fin	15/06/2018
Trabajo (h*h)	739,8
Coste directo (€)	13145,60
Coste indirecto (€)	1314,56
Coste material (€)	2899,00
Coste total (€)	17359,16

Tabla 5: Desglose del coste del proyecto.

10.6. Métricas del proyecto

En esta sección, se enumeran una serie de métricas del proyecto, que permiten comprender su alcance en mayor profundidad:

- Número de líneas de código: 18346.
- Número de errores y vulnerabilidades detectados por herramientas de análisis estático de código: 172.
- Número de *bugs* detectados y corregidos: 56.
- Cobertura (decisión): 80,6%.
- Duplicaciones de código: 4,9%.
- Casos de prueba implementados: 613.
- Tamaño actual de las bases de datos del sistema: 5,6 GB.
- Subsistema de Recolección de Datos:
 - Número de ejecuciones: 36633 (90,07% satisfactorias).
 - Tiempo ininterrumpido de ejecución: 736171.09 s (8.52 días).
 - Número total de elementos recopilados: 1695614.
 - Número de **DataCollectors**: 12.

11. CONCLUSIONES

En este apartado, se resumen los aspectos que se pueden extraer de la realización del proyecto. A continuación, se citan las líneas de trabajo futuras.

Desde el punto de vista de los objetivos marcados al inicio del proyecto, se puede destacar que:

- Se han **estudiado** aspectos relacionados con el clima de la Tierra a lo largo de la Historia, y las últimas evidencias científicas relativas al Cambio Climático, sus causas y consecuencias.
- Se ha desarrollado un **sistema complejo**, en el que interaccionan múltiples subsistemas y componentes, para lograr un objetivo común: mostrar datos sobre el Cambio Climático de forma sencilla, con el fin de llegar a las personas que no posean conocimientos previos en la materia.
- Se han abordado todas las **etapas de desarrollo** de un proyecto software: desde la especificación de requisitos hasta el despliegue y puesta en producción del sistema.
- Se ha obtenido un producto software cuya **calidad** es **medible** de forma cuantitativa, gracias a las disciplinas de Integración e Inspección Continua.
- Se ha aprendido a trabajar de cara al **cliente**, el *Product Owner*, y a incorporar funcionalidades y mejoras de acuerdo a su criterio.
- Se han puesto en práctica las técnicas y principios de ingeniería vistas a lo largo de la carrera: diseño de interfaces de usuario, captura de requisitos, concurrencia, arquitectura, Patrones de Diseño, pruebas, seguridad, etc.

Además, desde una perspectiva personal, el alumno destaca que el desarrollo del proyecto ha tenido un carácter muy positivo de cara a su **formación**, dado que la mayoría de tecnologías con las que éste ha trabajado, resultaban en un principio desconocidas.

Con este trabajo en su *portfolio*, el alumno ha conseguido un puesto en el Equipo de Arquitectura de una compañía norteamericana. Sin duda, las tecnologías y metodologías empleadas en este proyecto han resultado **decisivas** para superar el proceso de entrevista, y destacar sobre el resto de candidatos.

11.1. Trabajo futuro

Tras completar el desarrollo del proyecto, quedan en el tintero ciertos aspectos:

- La promoción de la aplicación en **redes sociales**, con el fin de lograr su difusión. Asimismo, sería interesante poder *twittear* o compartir en estas

plataformas mensajes relacionados con la información presentada por la aplicación.

- La incorporación de mecanismos que permitan extraer información de los datos almacenados por el sistema, que permita obtener **conocimiento**. Para ello, podrían utilizarse tácticas de *data mining*.
- Así como los Subsistemas de Recolección y Conversión de Datos exponen los ficheros de estado de los **DataCollectors** y **DataConverters**, sería interesante también hacer lo mismo con la **configuración**. Esto permitiría que los subsistemas adoptasen cambios de configuración desde la siguiente ejecución, sin tener que volver a construir la imagen de los componentes.
- La incorporación de **pruebas** automatizadas de **aceptación**, haciendo uso de *frameworks* como Selenium.
- La **securización** de la aplicación web y del componente API, mediante el uso de protocolos seguros como HTTPS. Además, la aplicación web debería poseer un nombre de **dominio** registrado.
- El desarrollo de pruebas de **carga** y **estrés** para la aplicación web, con el fin de aventurar las necesidades computacionales a la hora de escalar este subsistema.
- El componente API permite acceder a toda la información recopilada por el Subsistema de Recolección de Datos. Sería interesante **ofrecer** los datos a otros desarrolladores, mediante alguna licencia de código abierto.

ANEXO A. SELECCIÓN DE LOCALIZACIONES A MONITORIZAR

En este anexo describe los criterios empleados para seleccionar las localizaciones a monitorizar por el sistema.

A.1. Especificación de criterios

Debido a las limitaciones en espacio, políticas de APIs y otros recursos a la hora de realizar el despliegue del sistema, resulta inviable monitorizar un número no acotado de localizaciones. Por consiguiente, el conjunto de posibles ubicaciones a monitorizar se reducirá, atendiendo a criterios que permitan establecer filtros:

- **Regiones climáticas:** Se elegirán, como mínimo, tres localizaciones *relevantes* (por importancia y número de habitantes) para cada una de las 31 regiones climáticas de la Tierra [23].

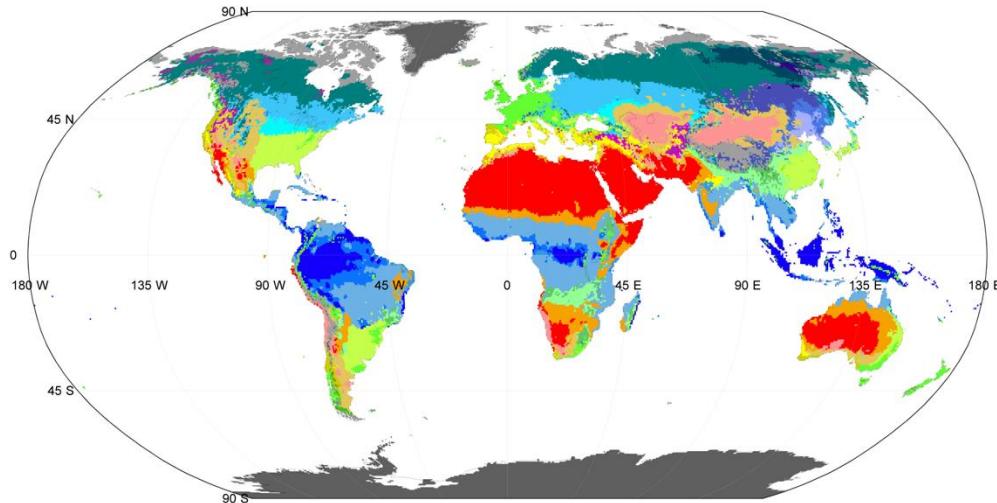


Ilustración 38: Conjunto de regiones climáticas de la Tierra (entre 1901-2010), según la clasificación Köppen.

- **Índice de Desarrollo Humano** (HDI, en inglés): Se seleccionarán localizaciones sin importar el nivel de HDI de sus países. Esto resulta interesante para visualizar si los efectos del Cambio Climático son mayores en países desarrollados o sub-desarrollados.
- **Popularidad** de las localizaciones en Twitter: Se analizará la popularidad y tendencia de hashtags de la forma **#{nombre de localización}**. Ya que uno de los objetivos de la aplicación es llegar al mayor número de personas, se priorizarán las localizaciones más populares.

A.2. Clasificación

En este apartado, se diseña un sistema que permite pasar de un conjunto no discreto de localizaciones a una lista reducida. Esta lista se ordenará, en base a la asignación de **puntuaciones** a las localizaciones, que oscilarán entre **0** y **3** puntos para cada criterio.

A.2.1. Regiones climáticas

A continuación, se muestra una tabla que reúne las características las zonas climáticas de la Tierra:

Código	Atributo 1	Atributo 2	Atributo 3
Af	Tropical	Húmedo	-
Am	Tropical	Monzón	-
As	Tropical	Verano seco	-
Aw	Tropical	Invierno seco	-
BWh	Seco	Desierto	Árido (cálido)
BWk	Seco	Desierto	Árido (frío)
BSh	Seco	Estepa	Árido (cálido)
BSk	Seco	Estepa	Árido (frío)
Csa	Templado	Verano seco	Verano cálido
Csb	Templado	Verano seco	Verano templado
Csc	Templado	Verano seco	Verano fresco
Cwa	Templado	Invierno seco	Verano cálido
Cwb	Templado	Invierno seco	Verano templado
dCwc	Templado	Invierno seco	Verano fresco
Cfa	Templado	Húmedo	Verano cálido
Cfb	Templado	Húmedo	Verano templado
Cfc	Templado	Húmedo	Verano fresco
Dsa	Frío	Verano seco	Verano cálido
Dsb	Frío	Verano seco	Verano templado

Dsc	Frío	Verano seco	Verano fresco
Dsd	Frío	Verano seco	Verano frío
Dwa	Frío	Invierno seco	Verano cálido
Dwb	Frío	Invierno seco	Verano templado
Dwc	Frío	Invierno seco	Verano fresco
Dwd	Frío	Invierno seco	Verano frío
Dfa	Frío	Húmedo	Verano cálido
Dfb	Frío	Húmedo	Verano templado
Dfc	Frío	Húmedo	Verano fresco
Dfd	Frío	Húmedo	Verano frío
ET	Polar	Tundra	-
EF	Polar	Permafrost	-

Tabla 6: *Regiones climáticas según la clasificación Köppen. Las columnas hacen referencia a la primera, segunda y tercera letra del código de las regiones climáticas, respectivamente.*

Aunque se pretendía identificar, como mínimo, tres localizaciones para cada región climática, se han dado casos en los que esto no ha sido posible. A pesar de ello, se obtienen un total de **133** localizaciones candidatas.

Una vez elegidas, se contabiliza el número de candidatos que acumula cada región climática. El objetivo es que todas las regiones posean representación, de forma que se restará importancia a las localizaciones de las regiones con mayor concentración. Por lo tanto, se conceden:

- **1,50** puntos a las regiones que tienen más de 9 representantes.
- **1,75** puntos a las regiones que tienen de 6 a 9 representantes.
- **2,00** puntos a las regiones que tienen de 4 a 5 representantes.
- **2,50** puntos a las regiones que tienen de 2 a 3 representantes.
- **3,00** puntos a las regiones que tienen menos de 2 representantes.

Clima	#	%	Puntos	Clima	#	%	Puntos
Cfa	15	11,28%	1,50	BWk	3	2,26%	2,50
Cfb	15	11,28%	1,50	Csc	3	2,26%	2,50
Dfb	10	7,52%	1,50	Dwa	3	2,26%	2,50
Csa	9	6,77%	1,75	EF	3	2,26%	2,50
BSk	6	4,51%	1,75	ET	3	2,26%	2,50
Csb	6	4,51%	1,75	Dsa	2	1,50%	2,50
Cwb	6	4,51%	1,75	Dsb	2	1,50%	2,50
BWh	5	3,76%	2,00	Dsd	2	1,50%	2,50
Cfc	5	3,76%	2,00	Cwc	1	0,75%	3,00
Dfa	5	3,76%	2,00	Dfc	1	0,75%	3,00
Am	4	3,01%	2,00	Dfd	1	0,75%	3,00
Aw	4	3,01%	2,00	Dsc	1	0,75%	3,00
BSh	4	3,01%	2,00	Dwb	1	0,75%	3,00
Cwa	4	3,01%	2,00	Dwc	1	0,75%	3,00
Af	3	2,26%	2,50				
As	3	2,26%	2,50				

Tabla 7: Resultados de la clasificación por regiones climáticas. Cada localización obtendrá la puntuación de su región asociada (en **azul**).

A.2.2. Índice de Desarrollo Humano

Según las Naciones Unidas, el HDI es una métrica que evalúa el desarrollo social y económico de los países. A la hora de clasificar los estados, se examinan cuatro áreas principales: tiempo promedio de escolaridad, tiempo esperado de escolaridad, esperanza de vida al nacer y renta *per capita* [24].

De acuerdo con la puntuación alcanzada, se dice que un país tiene un HDI:

- **Muy alto**, si su puntuación es mayor o igual a 0,800.
- **Alto**, si su puntuación es menor que 0,800 y mayor o igual a 0,700.
- **Medio**, si su puntuación es menor que 0,700 y mayor o igual a 0,550.
- **Bajo**, si su puntuación es inferior a 0,550.

Las Naciones Unidas recoge la clasificación de los países atendiendo a su HDI. De estos datos, junto con las localizaciones candidatas obtenidas en el apartado anterior, se extraen las siguientes estadísticas:

Descripción	#	%	Puntos
Localizaciones con HDI muy alto	79	59,40%	2,00
Localizaciones con HDI alto	26	19,55%	2,50
Localizaciones con HDI medio	18	13,53%	2,75
Localizaciones con HDI bajo	6	4,51%	3,00
Localizaciones sin datos sobre el HDI	4	3,01%	3,00
Total	133	100%	

Tabla 8: Estadísticas según los niveles de HDI. La puntuación asignada a cada nivel aparece a la derecha (en azul).

Dado que se pretende contar con localizaciones de cualquier tipo de nivel HDI, se otorgan más puntos a aquellos niveles que posean menor número de representantes.

A.2.3. Popularidad en Twitter

Medir la popularidad de un suceso o fenómeno es un proceso complicado, ya que entran en juego gran cantidad de variables: históricas, sociales, económicas, etc.

La aproximación elegida para medir la popularidad de las localizaciones responde al exponencial aumento del uso de redes sociales durante los últimos años. Un buen ejemplo de ello es Twitter, con más de 325 millones de usuarios activos [25].

Cuando se publica un *tweet* sobre un tema, éste suele venir acompañado de uno o más *hashtags*. Aunque existen distintos tipos de *hashtags* (corporativos, tendencias, eventos...), todos ellos promocionan el resto del contenido y lo hacen más localizable [26]. Twitter fue pionero en introducir los *hashtags*, hecho que condujo a esta red social a alcanzar un gran éxito en poco tiempo.

Cuanto más se usa un *hashtag*, más popular es; llegando incluso a convertirse en *trending topic*⁴³. Por ejemplo, durante los atentados de París ocurridos en 2015, el *hashtag* **#PrayForParis** fue *trending topic*.

⁴³ del inglés, que marca tendencia.



Apple apuesta por el **#clima** con un bono **#verde** de \$1000M, a pesar de salida de los **#EEUU** de los acuerdos de **#París**
buff.ly/2rvolAc

Figura 1: Recorte de un tweet, donde se pueden apreciar diversos hashtags (entre ellos, un país y una ciudad).

Existen varias herramientas que permiten analizar la popularidad y tendencia de los *hashtags*. Se ha optado por una gratuita, que permite obtener, además de los parámetros mencionados, otros datos como *hashtags* correlacionados [27].

Se ha registrado la popularidad (de 0,00 a 100,00) y tendencia de las 133 localizaciones candidatas, y de los 69 países en las que se encuentran. Se puntúa por separado a localizaciones y países, otorgando:

- **3,00** puntos a los *hashtags* con popularidad mayor o igual a 70,0.
- **2,50** puntos a los *hashtags* con popularidad entre 60,0 y 69,9.
- **2,00** puntos a los *hashtags* con popularidad entre 50,0 y 59,9.
- **1,50** puntos a los *hashtags* con popularidad entre 30,0 y 49,9.
- **1,00** puntos a los *hashtags* con popularidad entre 10,0 y 29,9.
- **0,00** puntos a los *hashtags* con popularidad menor a 10,0.

La puntuación final se obtiene haciendo la media ponderada entre la puntuación de las localizaciones (70%), y la puntuación de los países (30%).

A.3. Resultados

Además de los 9 puntos que puede obtener una localización sumando las calificaciones de los criterios anteriores, se decide conceder 1 punto extra a las **capitales** de los países, dado que se trata de las ubicaciones más importantes dentro de los mismos, en consonancia con el objetivo de llegar a la mayor cantidad de gente posible.

A continuación, se presentan los resultados desglosados, recordando los criterios de clasificación:

- Regiones climáticas (3 puntos).
- HDI (3 puntos).
- Popularidad en Twitter (3 puntos).
- Capitales, marcadas con ✓ (1 punto).
- Puntuación total (10 puntos).

Selección de localizaciones a monitorizar – *Resultados*

#	Localización	País	1	2	3	4	5
1	El Cairo	Egipto	2,00	2,75	2,65	✓	8,40
2	Nueva Delhi	India	2,00	2,75	2,65	✓	8,40
3	Guatemala	Guatemala	2,00	2,75	2,50	✓	8,25
4	Ciudad de México	México	1,75	2,50	3,00	✓	8,25
5	Mogadiscio	Somalia	2,00	3,00	2,15	✓	8,15
6	Bangkok	Tailandia	2,00	2,50	2,65	✓	8,15
7	Singapur	Singapur	2,50	2,00	2,50	✓	8,00
8	Kabul	Afganistán	1,75	3,00	2,15	✓	7,90
9	Katmandú	Nepal	2,00	2,75	2,15	✓	7,90
10	Ankara	Turquía	1,75	2,50	2,65	✓	7,90
11	Brasilia	Brasil	2,00	2,50	2,30	✓	7,80
12	Madrid	España	1,75	2,00	3,00	✓	7,75
13	Beijing	China	2,50	2,50	2,65		7,65
14	Astaná	Kazajistán	2,50	2,50	1,65	✓	7,65
15	Nairobi	Kenya	1,75	2,75	2,15	✓	7,65
16	Ulán Bator	Mongolia	2,50	2,50	1,65	✓	7,65
17	Ciudad de Panamá	Panamá	2,00	2,50	2,15	✓	7,65
18	Port Moresby	Papúa Nueva Guinea	2,00	3,00	1,65	✓	7,65
19	Roma	Italia	1,75	2,00	2,85	✓	7,60
20	El Alto	Bolivia	3,00	2,75	1,80		7,55
21	Addis Ababa	Etiopía	1,75	3,00	1,80	✓	7,55
22	Hanoi	Vietnam	2,00	2,75	1,80	✓	7,55
23	Berlín	Alemania	1,50	2,00	3,00	✓	7,50
24	París	Francia	1,50	2,00	3,00	✓	7,50

Selección de localizaciones a monitorizar – *Resultados*

25	Londres	Reino Unido	1,50	2,00	3,00	✓	7,50
26	Pekín	China	2,00	2,50	1,95	✓	7,45
27	Monte Everest	Nepal	2,50	2,75	2,15		7,40
28	Santiago	Chile	1,75	2,00	2,65	✓	7,40
29	Biskek	Kirguistán	2,50	2,75	1,15	✓	7,40
30	Antananarivo	Madagascar	1,75	3,00	1,65	✓	7,40
31	Túnez	Túnez	1,75	2,50	2,15	✓	7,40
32	Fortaleza	Brasil	2,50	2,50	2,30		7,30
33	Anchorage, Alaska	Estados Unidos	3,00	2,00	2,30		7,30
34	El Paso	México	2,50	2,50	2,30		7,30
35	Lagos	Nigeria	2,00	3,00	2,30		7,30
36	Vladivostok	Rusia	3,00	2,00	2,30		7,30
37	Bogotá	Colombia	1,50	2,50	2,30	✓	7,30
38	Seúl	Corea del Sur	2,50	2,00	1,80	✓	7,30
39	Pretoria	Sudáfrica	1,75	2,75	1,80	✓	7,30
40	Kiev	Ucrania	1,50	2,50	2,30	✓	7,30
41	Pyonyang	Corea del Norte	2,50	3,00	0,75	✓	7,25
42	Lukla	Nepal	3,00	2,75	1,45		7,20
43	Indore	India	2,50	2,75	1,95		7,20
44	Teherán	Irán	1,75	2,50	1,95	✓	7,20
45	Seatte, Washington	Estados Unidos	2,50	2,00	2,65		7,15
46	Buenos Aires	Argentina	1,50	2,00	2,65	✓	7,15
47	Washington DC	Estados Unidos	1,50	2,00	2,65	✓	7,15
48	Tokyo	Japón	1,50	2,00	2,65	✓	7,15
49	Bucarest	Rumanía	2,00	2,00	2,15	✓	7,15
50	Moscú	Rusia	1,50	2,00	2,65	✓	7,15

Selección de localizaciones a monitorizar – *Resultados*

51	Montevideo	Uruguay	1,50	2,50	2,15	✓	7,15
52	Praia	Cabo Verde	2,50	2,75	1,85		7,10
53	Beirut	Líbano	1,75	2,50	1,85	✓	7,10
54	Atenas	Grecia	1,75	2,00	2,30	✓	7,05
55	Estación Vostok	Antártida	2,50	3,00	1,50		7,00
56	Dubai	Emiratos Árabes Unidos	2,00	2,00	3,00		7,00
57	Chicago, Illinois	Estados Unidos	2,00	2,00	3,00		7,00
58	Miami, Florida	Estados Unidos	2,00	2,00	3,00		7,00
59	Ciudad del Cabo	Sudáfrica	1,75	2,75	2,50		7,00
60	Bruselas	Bélgica	1,50	2,00	2,50	✓	7,00
61	Dublín	Irlanda	1,50	2,00	2,50	✓	7,00
62	Ámsterdam	Países Bajos	1,50	2,00	2,50	✓	7,00
63	Estocolmo	Suecia	1,50	2,00	2,50	✓	7,00
64	Homer, Alaska	Estados Unidos	3,00	2,00	1,95		6,95
65	Yakutsk	Rusia	3,00	2,00	1,95		6,95
66	Muş	Turquía	2,50	2,50	1,95		6,95
67	Marrakech	Marruecos	2,00	2,75	2,15		6,90
68	Dras	India	2,50	2,75	1,60		6,85
69	Oral	Kazajistán	2,00	2,50	2,35		6,85
70	San Carlos de Bariloche	Argentina	2,50	2,00	2,30		6,80
71	Río de Janeiro	Brasil	2,00	2,50	2,30		6,80
72	Eureka	Canadá	2,50	2,00	2,30		6,80
73	Islas Galápagos	Ecuador	2,00	2,50	2,30		6,80
74	West Palm Beach, Florida	Estados Unidos	2,50	2,00	2,30		6,80
75	Monte Fuji	Japón	2,50	2,00	2,30		6,80

Selección de localizaciones a monitorizar – *Resultados*

76	Mbandaka	República Democrática del Congo	2,50	3,00	1,30	6,80
77	Barcelona	España	1,75	2,00	3,00	6,75
78	Los Ángeles, California	Estados Unidos	1,75	2,00	3,00	6,75
79	Minneapolis, Minnesota	Estados Unidos	2,00	2,00	2,65	6,65
80	Phoenix, Arizona	Estados Unidos	2,00	2,00	2,65	6,65
81	Johannesburgo	Sudáfrica	1,75	2,75	2,15	6,65
82	Viena	Austria	1,50	2,00	2,15	✓ 6,65
83	Bratislava	Bielorrusia	1,50	2,50	1,65	✓ 6,65
84	Copenhague	Dinamarca	1,50	2,00	2,15	✓ 6,65
85	Helsinki	Finlandia	1,50	2,00	2,15	✓ 6,65
86	Oslo	Noruega	1,50	2,00	2,15	✓ 6,65
87	Wellington	Nueva Zelanda	1,50	2,00	2,15	✓ 6,65
88	Varsovia	Polonia	1,50	2,00	2,15	✓ 6,65
89	Taipei	Taiwán	1,50	2,00	2,15	✓ 6,65
90	Vancouver	Canadá	1,50	2,00	3,00	6,50
91	Atlanta, Georgia	Estados Unidos	1,50	2,00	3,00	6,50
92	Houston, Texas	Estados Unidos	1,50	2,00	3,00	6,50
93	Nueva York	Estados Unidos	1,50	2,00	3,00	6,50
94	Budapest	Hungría	1,50	2,00	2,00	✓ 6,50
95	Attawapiskat	Canadá	2,50	2,00	1,95	6,45
96	Haleakalā, Hawaii	Estados Unidos	2,50	2,00	1,95	6,45
97	Perth	Australia	1,75	2,00	2,65	6,40
98	Sevilla	España	1,75	2,00	2,65	6,40
99	Denver, Colorado	Estados Unidos	1,75	2,00	2,65	6,40
100	San Francisco, California	Estados Unidos	1,75	2,00	2,65	6,40

Selección de localizaciones a monitorizar – *Resultados*

101	Islas Malvinas	Argentina	2,00	2,00	2,30	6,30
102	São Paulo	Brasil	1,50	2,50	2,30	6,30
103	Omaha, Nebraska	Estados Unidos	2,00	2,00	2,30	6,30
104	Cuzco	Perú	2,00	2,50	1,80	6,30
105	Bloemfrontein	Sudáfrica	1,75	2,75	1,80	6,30
106	Estambul	Turquía	1,75	2,50	1,95	6,20
107	Córdoba	Argentina	1,50	2,00	2,65	6,15
108	Melbourne	Australia	1,50	2,00	2,65	6,15
109	Sydney	Australia	1,50	2,00	2,65	6,15
110	Quebec	Canadá	1,50	2,00	2,65	6,15
111	Indianápolis, Indiana	Estados Unidos	1,50	2,00	2,65	6,15
112	Nueva Orleans, Mississippi	Estados Unidos	1,50	2,00	2,65	6,15
113	Osaka	Japón	1,50	2,00	2,65	6,15
114	Balmaceda	Chile	2,50	2,00	1,60	6,10
115	Oymyakon	Rusia	2,50	2,00	1,60	6,10
116	Mendoza	Argentina	1,75	2,00	2,30	6,05
117	A Coruña	España	1,75	2,00	2,30	6,05
118	Milán	Italia	1,50	2,00	2,50	6,00
119	Base Esperanza	Antártida	2,50	3,00	0,45	5,95
120	Alice Springs	Australia	2,00	2,00	1,95	5,95
121	Punta Arenas	Chile	2,00	2,00	1,95	5,95
122	Shangai	China	1,50	2,50	1,95	5,95
123	Valle de la Muerte, California	Estados Unidos	2,00	2,00	1,95	5,95
124	Inverness, Escocia	Reino Unido	2,00	2,00	1,95	5,95
125	Oporto	Portugal	1,75	2,00	2,15	5,90

Selección de localizaciones a monitorizar – *Resultados*

126	San Petersburgo	Rusia	1,50	2,00	2,30	5,80
127	Reikjavik	Islandia	2,00	2,00	0,75 ✓	5,75
128	Zúrich	Suiza	1,50	2,00	2,15	5,65
129	Bergen	Noruega	2,00	2,00	1,45	5,45
130	Santiago de Compostela	España	1,50	2,00	1,95	5,45
131	Chersky	Rusia	2,50	2,00	0,90	5,40
132	Seymchan	Rusia	2,50	2,00	0,90	5,40
133	Estación Summit, Groenlandia	Dinamarca	2,50	2,00	0,75	5,25

Tabla 9: Clasificación de las localizaciones según los criterios combinados. Las puntuaciones finales pueden verse a la derecha (en azul).

Una vez ordenadas las localizaciones por puntuación, se decide contar solo con aquellos candidatos cuya puntuación sea mayor o igual que **6,75** puntos. De esta forma, se reduce el número de localizaciones de 133 a **78** (un 41,35% menos), manteniendo el 100% de niveles de HDI, y el 90,32% de las regiones climáticas.

Si se pretendiese alcanzar el 100% de regiones climáticas, sería necesario bajar la puntuación mínima a 6,1 puntos; con el hándicap de aumentar el número de localizaciones a 115 (reduciendo sólo un 13,53% de las 133 iniciales). Por lo tanto, se optó por la solución anterior: seleccionar solamente aquellas ubicaciones que posean una puntuación mayor o igual a 6,75 puntos.

No obstante, la lista final de localizaciones a monitorizar sufre modificaciones, dado que no siempre es posible obtener datos de todas éstas. Esta información se puede consultar en la aplicación web, en la URL `/locations/list-all`, o mediante el enlace situado en el *footer* de la misma:



Ilustración 39: Footer de la aplicación web, con el enlace que permite listar todas las localizaciones monitorizadas.

ANEXO B. INTEGRACIÓN E INSPECCIÓN CONTINUA

Los conceptos de Integración e Inspección Continua hacen referencia a prácticas del desarrollo software que tienen como objetivos **minimizar** el esfuerzo de construir el producto software a partir del código fuente, y **mejorar** su calidad.

B.1. Integración Continua

Entre los beneficios de la Integración Continua, se pueden destacar:

- ✓ Construcciones automáticas, repetibles y medibles.
- ✓ Detección temprana de errores durante la construcción del producto.
- ✓ Mejora de la visibilidad del estado del proyecto.
- ✓ Aumento de la confianza en el software.

Para sacarle el máximo provecho a esta disciplina, existen una serie de prácticas recomendadas, según lo visto en la asignatura *Ferramentas de Desenvolvimento*. Entre ellas, se incluyen:

- Usar un sistema de control de versiones, para administrar el código fuente, scripts de instalación y ficheros de configuración.
- Habilitar construcciones usando la línea de comandos.
- Publicar cambios en el repositorio con frecuencia.
- Integrar los cambios en el código con frecuencia.
- Permitir construcciones rápidas –diez minutos de duración, a lo sumo–.
- Probar el software en un entorno lo más similar posible al de producción.
- Facilitar la obtención de resultados de pruebas y de la construcción.
- Automatizar el despliegue.

B.1.1. Soluciones

Entre las soluciones software con mayor presencia en el mercado actual, podemos citar [28]:

- AWS CodePipeline.
- Bamboo.
- CircleCI.
- Codeship.
- Jenkins.
- Puppet Pipelines.
- Semaphore.
- Team City.
- TFS.
- Travis CI.

La elección de un candidato se realiza teniendo en cuenta:

1. El coste.

2. La flexibilidad y configuración de la herramienta.
3. La integración con el resto de tecnologías empleadas para el desarrollo del proyecto.
4. El conocimiento previo de soluciones de Integración Continua.

Se decide usar **Jenkins**, ya que:

- Es gratuito y de código abierto.
- Es independiente de la plataforma y sistema operativo (está desarrollado empleando tecnologías Java).
- Es orientado a *plugins*, por lo que es posible agregar nuevas características y subsanar carencias.
- Es muy flexible a la hora de establecer el *pipeline* de construcción del producto software.
- Se ha usado en la asignatura *Ferramentas de Desenvolvimento*, y el alumno está familiarizado con sus funciones.

B.1.2. Configuración de Jenkins

Jenkins se puede ejecutar como un programa Java *standalone*, o desplegarse en un contenedor de aplicaciones Java EE.

No obstante, se emplea **Docker** para desplegar una instancia de Jenkins de forma rápida y sencilla. Se ha partido de la imagen `jenkins/jenkins:latest`, y usado `docker-compose` para definir los volúmenes y puertos que expondrá el contenedor.

B.1.2.1. Problemas iniciales

- ✖ Jenkins **necesita** poder usar Docker para construir y lanzar los componentes de la aplicación. En un primer momento, se pensó en usar lo que se conoce como *Docker-in-Docker*⁴⁴; pero, tras leer [29], se decide proporcionar el *socket* que emplea el *daemon* de Docker del contenedor Jenkins a través del de la propia máquina (`/var/run/docker.sock`). Esta solución conduce al siguiente problema.
- ✖ Si se despliega el servidor Jenkins en la misma máquina en la que está corriendo la aplicación, al usar el mismo *socket*, las imágenes y contenedores creados por Jenkins sobrescribirán los de la aplicación en producción. Como solución, se han redefinido los servicios contenidos en el fichero `docker-compose.yml`, añadiendo `_CI` a los nombres de los existentes. Los nuevos servicios

⁴⁴ **Docker-in-Docker**: Estrategia que permite usar todas las características de Docker dentro de un contenedor Docker. Nació para ayudar a simplificar el desarrollo de Docker en sí, y no se recomienda su uso (salvo en casos excepcionales).

ejecutan exactamente el mismo código, pero no interfieren con los de producción, ya que no exportan ni volúmenes ni exponen puertos al exterior.

- ✖ Dado que no se usa un sistema de construcción automática (e.g. Maven), deben emplearse *scripts* que permitan generar la aplicación. Para ello, se emplea el *script* base `install.sh`, al que se le añaden nuevas opciones.
- ✖ Los reportes de resultados y cobertura de *test* están dentro de imágenes Docker, y no son (directamente) accesibles desde Jenkins. Además, no se podrá visualizar la cobertura sobre los propios ficheros, ya que difieren los *paths* del fichero XML que genera Cobertura (dentro de la imagen Docker) y el del fichero dentro del *workspace* de Jenkins. Como solución, se crea un *script* `generate_reports.sh`, que: 1) recupera los reportes de *test* y cobertura desde las imágenes Docker al *workspace*, y 2) reemplaza las ocurrencias del *path* dentro de la imagen por el del *workspace*.

B.1.2.2. Pasos

1. Arrancar el servidor Jenkins. Suponiendo que el directorio actual es el directorio raíz de la aplicación:

```
$ chmod +x /ci/install.sh && ./ci/install.sh
```

2. Establecer el *Administrator password* con el valor del fichero `/var/jenkins_home/secrets/initialAdminPassword`.
3. Instalar los *plugins*:
 - a. **AnsiColor**: Muestra la salida del terminal con códigos de escape ANSI (i.e. usando los colores del terminal).
 - b. **Build Timeout**: Permite establecer estrategias que aborten la construcción bajo ciertas circunstancias.
 - c. **Cobertura**: Muestra los resultados del reporte de cobertura de *test* de forma gráfica.
 - d. **Embeddable Build Status**: Permite añadir un *badge* del estado de la construcción. Ejemplos son:

build failure

build passing

- e. **Green Balls**: Por defecto, Jenkins usa el color azul para indicar una construcción satisfactoria. Este *plugin* cambia ese color a verde, más intuitivo.
- f. **JUnit Realtime Test Reporter**: Muestra de forma gráfica el resultado de la ejecución de los *test*.

- g. **Locale:** Permite sobrescribir el idioma de Jenkins, ignorando la cabecera `Accept-Language` de las peticiones del navegador.
 - h. **PostBuildScript:** Permite ejecutar *scripts* como *post-build action*⁴⁵.
 - i. **SonarQube Scanner for Jenkins:** Permite realizar un análisis de SonarQube durante el proceso de construcción. Se hablará de ello más adelante.
 - j. **Timestamper:** Añade el *timestamp* a cada salida de consola.
 - k. **Workspace Cleanup:** Permite eliminar los contenidos del *workspace* tras la construcción.
4. En GitHub, añadir “Jenkins (GitHub plugin)” en Settings > Integration & Services. Debe incluirse la URL de Jenkins (con el puerto).
 5. Crear un *job* de Jenkins para el proyecto.
 6. Configurar el *job*, de la siguiente forma:
 - a. Establecer el nombre del proyecto.
 - b. Incluir en la descripción del *job* la URL del repositorio del proyecto (en GitHub).
 - c. Establecer **git** como Source Code Management (SCM).
 - i. URL (SSH) del repositorio.
 - ii. Credenciales “SSH username with private key”. Puede generarse claves SSH con el comando `ssh-keygen`. La clave pública deberá añadirse al repositorio en GitHub, en Settings > Deploy keys.
 - iii. Especificar las ramas de las cuales construir código.
 - d. Habilitar el *trigger* que permite realizar construcciones automáticas al recibir un *push* en GitHub.
 - e. Establecer un *timeout* al proceso de construcción.
 - f. Añadir *timestamps* a la salida de consola.
 - g. Usar colores ANSI en la salida de consola.
 - h. Establecer los pasos de construcción:
 - i. Ejecución del *script* base: `install.sh`.
 - ii. Ejecución del *script* de generación de reportes de resultados y cobertura de *test*: `ci/generate_reports.sh`.
 - iii. Ejecutar un análisis de Sonar Scanner. En lugar de establecer propiedades de análisis, éstas se leerán del fichero de configuración `sonar-project.properties`.
 - i. Establecer las *post-build actions*:

⁴⁵ **post-build action:** Tareas que se realizan después de la construcción. Ejemplos típicos son la publicación de los resultados de los *test*, o la limpieza del *workspace*.

- i. En caso de que la construcción sea fallida, ejecutar el *script* de generación de reportes de resultados y cobertura de *test*: `ci/generate_reports.sh`.
- ii. Eliminar todos los contenedores e imágenes Docker generados por Jenkins durante la construcción.
- iii. Publicar los resultados de cobertura de los *test*.
- iv. Publicar los resultados de ejecución de los *test*.
- v. Marcar con ✓ o ✗ el *commit* (en GitHub) que disparó la construcción, según el resultado.

A continuación, se muestra un diagrama que esquematiza el *pipeline* de construcción de la aplicación:

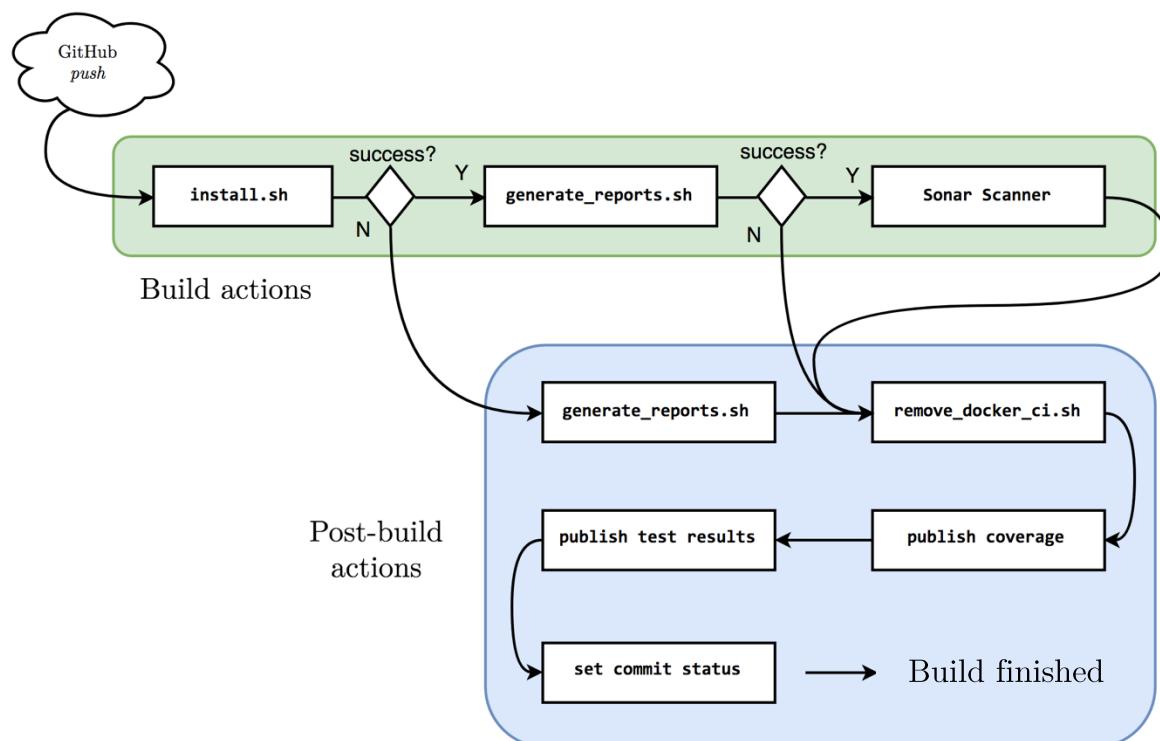


Ilustración 40: Pipeline de construcción de la aplicación.

B.1.2.3. Resultados

Una vez terminada la configuración, se realizarán construcciones automáticas cada vez que se haga *push* al repositorio.

En el momento en que finalice una ejecución, se actualizarán las estadísticas del proyecto:

- Cobertura de los *test* a lo largo de las construcciones.
- Resultados de los *test* a lo largo de las construcciones.

- Resultado del *Quality Gate* de SonarQube (se explicará más adelante).
- *Permalinks* a la última construcción estable, satisfactoria, fallida, etc.
- El *badge* que muestra el estado actual del proyecto.
- Además, se podrán visualizar los resultados de la ejecución:
- Cobertura de los *test* (sobre cada fichero).
- Resultados de la ejecución de los *test* (detalle).
- Salida de consola durante la construcción.
- Estadísticas, como el tiempo de construcción.
- Cambios que incluye el *commit* que disparó la construcción.

B.1.3. Despliegue

El servidor Jenkins desplegado para este proyecto es accesible desde la URL:

<http://193.144.50.92:8090/job/ClimateChangeApp/>

Se ha permitido el acceso con permisos de lectura a usuarios anónimos, **de forma deliberada**. No obstante, si el lector desea iniciar sesión con mayores privilegios, se ha creado un usuario a tales efectos:

Usuario: examiner
Contraseña: examiner

B.2. Inspección Continua

La Inspección Continua se enfoca en la **calidad** del producto, tanto a nivel funcional, como a nivel de código. Permite, de forma rápida, obtener gran cantidad de métricas de calidad del software, y hacerlas visibles para que todos los miembros de la organización conozcan el estado del producto.

Como ocurrió con Integración Continua, esta disciplina también posee una serie de buenas prácticas, de acuerdo a lo estudiado en la asignatura *Ferramentas de Desenvolvimento*. Entre ellas, se encuentran:

- La calidad del software debe preocupar a toda la organización, pero es responsabilidad última del equipo de desarrollo.
- La calidad del software debe ser parte del proceso de desarrollo, como requisito no funcional a satisfacer.
- Los requisitos de calidad deben ser objetivos y, en la medida de lo posible, comunes a todos los productos software, independientemente de su especificación.
- Debe medirse en la última versión disponible del código, y repetirse de forma continuada conforme aparecen cambios y nuevas funcionalidades.

- Los problemas existentes deben asignarse en materia de tiempo y recursos para su resolución.

B.2.1. Soluciones

Existen múltiples herramientas que permiten llevar a cabo Inspección Continua: CheckStyle, FindBugs, PMD, SonarQube, etc.

Se ha optado por emplear **SonarQube**, debido a las siguientes razones:

- Es gratuita (en su versión *Community Edition*).
- Evalúa gran cantidad de métricas del proyecto: cobertura, duplicidad de bloques de código, vulnerabilidades, complejidad ciclomática⁴⁶, etc.
- Resume la evaluación empleando notación fácilmente comprensible (utiliza las letras A, B, C, D, E para indicar el grado de cumplimiento de los parámetros de calidad).
- Posee integración con Jenkins a través de un *plugin*.
- Permite configurar qué aspectos analizar o excluir del análisis, y establecer restricciones de calidad a nivel de producto.
- Se ha usado en la asignatura *Ferramentas de Desenvolvimento*, y el alumno está familiarizado con sus funciones.

B.2.2. Configuración de SonarQube

SonarQube consta de dos componentes: *sonar-scanner*, que realiza el análisis de proyectos; y Sonar, una *aplicación web* que permite acceder a los resultados del análisis, guardarlos en una base de datos, y mucho más.

Se ha usado **Docker** para desplegar tanto la aplicación web como la base de datos que ésta requiere (una instancia de PostgreSQL). Una vez la aplicación web está activa:

- Se crea el usuario inicial **admin**, junto con un *token* que permite añadir a la instancia de Sonar los análisis generados con *sonar-scanner*.
- Se realiza un primer análisis desde el entorno de desarrollo, en línea de comandos, para configurar el proyecto.
- Se crea un fichero **sonar-project.properties** en el directorio raíz de la aplicación, que incluye aspectos clave de configuración para el análisis del proyecto: lenguaje de programación, ficheros excluidos, ubicación de los reportes de cobertura y resultados de *test*, etc.

⁴⁶ **complejidad ciclomática:** Hace referencia a la complicación del flujo de ejecución de un sistema. Mide el mínimo número de casos de prueba necesarios para probar todas las ramas (i.e. decisiones) presentes en el código.

- Se añade el *plugin SVG Badges*, que permite mostrar *badges* de métricas de Sonar; por ejemplo, en el fichero `README.md` del repositorio en GitHub.
- Se define un **Quality Gate**⁴⁷ más estricto que el que incluye SonarQube por defecto. Éste, impone las siguientes restricciones sobre el código del proyecto:
 - Número de *bugs* menor que 10.
 - Número de *code smells* menor que 10.
 - Porcentaje de comentarios en código mayor del 15%.
 - Porcentaje de cobertura de código mayor del 80%.
 - Inexistencia de *issues* críticos.
 - Duplicidad de código menor del 10%.
 - Menos de 20 *issues* abiertos.
 - Inexistencia de vulnerabilidades.

B.2.3. Integración de SonarQube con Jenkins

Para lograr este propósito, se instala el *plugin SonarQube Scanner for Jenkins*. Para configurarlo, se debe:

- Añadir un “SonarQube server”, cuya URL sea accesible desde Jenkins; y un *token* que permita publicar los resultados de *sonar-scanner* en el servidor. Esto se realiza en: Manage Jenkins > Configure System.
- Instalar un ejecutable del “SonarQube Scanner”. Para ello, se selecciona la opción “Install automatically”, que permite que se descargue el binario bajo demanda desde el repositorio Maven Central. Esto se realiza en Manage Jenkins > Global Tool Configuration.
- Añadir el paso “*Execute SonarQube Scanner*” como *build action* en el *pipeline* de construcción del *job* de Jenkins, comentado en la sub-sección [B.1.2.2. Pasos](#).

Una vez configurado, se ejecutará *sonar-scanner* cada vez que se dispare una construcción en Jenkins, y se publicará el resultado del análisis en Sonar. En Jenkins se mostrará si el proyecto satisface el Quality Gate impuesto o no.

B.2.4. Integración de SonarQube con PyCharm

A través del *plugin SonarLint* para el IDE PyCharm, se puede reducir el número de *issues*, *bugs* y vulnerabilidades en el código que se suba al repositorio.

⁴⁷ **Quality Gate**: Hace referencia al conjunto de condiciones que el producto software debe satisfacer antes de que pueda ser desplegado en producción. SonarQube permite establecerlo y personalizarlo a nivel de proyecto.

Este *plugin* realiza un análisis automático del fichero en el que se está trabajando, y permite detectar los mismos problemas que *sonar-scanner*.

Para configurar el *plugin*, deben realizarse los siguientes pasos:

- Habilitar el *binding* con un servidor SonarQube. Esto permite que los problemas marcados como “falso positivo” o “no resolver” no sean señalados por SonarLint.
- Configurar un servidor SonarQube existente. Para ello, se debe:
 - Introducir la URL del servidor SonarQube. Ésta debe ser accesible.
 - Generar un *token* que permita a SonarLint conectarse al servidor. Esto se puede realizar desde My Account > Security > Generate Tokens, en la aplicación web de SonarQube.

Al finalizar la configuración, se mostrarán los problemas detectados por SonarLint en el *toolbar* de la parte inferior de la pantalla:



Ilustración 41: Vulnerabilidad detectada por SonarLint, al introducir una dirección IP “hard-coded” en el fichero.

B.2.5. Despliegue

El servidor SonarQube desplegado para este proyecto es accesible desde la URL:

<http://193.144.50.92:9000/dashboard?id=ClimateChangeApp>

Se ha permitido el acceso con permisos de lectura a usuarios anónimos, **de forma deliberada**. No obstante, si el lector desea iniciar sesión con mayores privilegios, se ha creado un usuario a tales efectos:

Usuario: examiner
Contraseña: examiner

ANEXO C. CONFIGURACIÓN DE TELEGRAM

En este anexo, se incluyen los detalles de configuración de Telegram para el envío de mensajes de *log* con nivel **CRITICAL** de forma automática y directa.

C.1. *Bots*

Telegram proporciona un API para trabajar con *bots*. Éstos no son más que aplicaciones creadas por terceros que funcionan dentro de Telegram. Pueden emplearse, entre otros, para:

- Obtener notificaciones y novedades.
- Integrarlos con otros servicios: Gmail, Wiki, GitHub...
- Aceptar pagos de usuarios de Telegram.
- Crear herramientas para dar soporte a votación, recordatorios de tareas, etc.
- Construir juegos.

Para crear un *bot* hay que interactuar con... otro *bot*, llamado *BotFather*, y seguir unas instrucciones. Una vez terminado, el usuario tendrá acceso al *bot* mediante el API HTTPS que exporta Telegram. En el caso del sistema, el *bot* se llama **@ClimateChangeAppBot**.

Una consideración importante es que un *bot* nunca puede iniciar una conversación con un usuario, debiendo ser este último quien dé el primer paso.

C.2. Componente Telegram

El sistema implementa un componente capaz de interactuar con el *bot*, de forma que el usuario puede iniciar una conversación con éste y, a partir de entonces, el *bot* podrá notificar vía Telegram los errores **CRITICAL** del sistema.

Para enviar mensajes, el *bot* necesita el ID del **chat** con el usuario. Éste se puede obtener mediante la invocación del *script* de instalación de este componente.

De cara a la implementación, se emplea la librería **python-telegram-bot** para abstractar las llamadas al API de Telegram.

C.3. Pasos

Esta sección reúne los pasos necesarios para configurar correctamente el componente Telegram. Son:

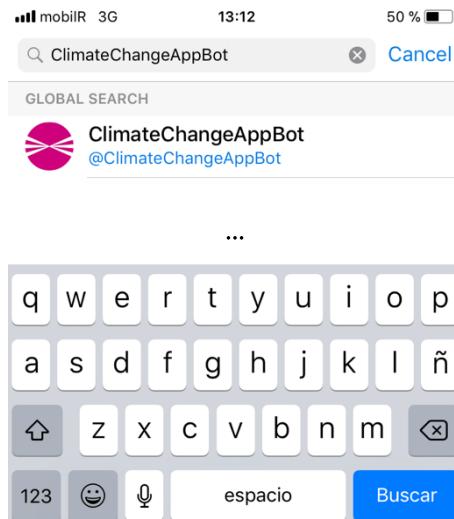
1. Desde la línea de comandos, ejecutar el *script* de instalación del componente. Suponiendo que el directorio actual es el directorio raíz del proyecto:

```
$ ./telegram_bot/install.sh
```

Aparecerá el siguiente mensaje:

```
[INFO] Running the Telegram Configurator component.  
➤ Using Climate Change App Telegram Bot configurator v1.0.  
Start a chat with the Telegram bot @ClimateChangeAppBot.  
Hit "Enter" when done:
```

2. Debe iniciarse un chat con el *bot* @ClimateChangeAppBot. Para ello, se busca al *bot* en Telegram. Éste posee el icono de la Universidad de A Coruña, por lo que resulta fácil de encontrar:



Haciendo *click* sobre el nombre del *bot*, se abrirá un chat con éste. Al fondo, aparecerá un botón, que es necesario pulsar:



Configuración de Telegram – *Pasos*

En caso de que éste no aparezca, se deberá a que ya se ha iniciado anteriormente una conversación. En ese caso, deberá escribirse `/start`:



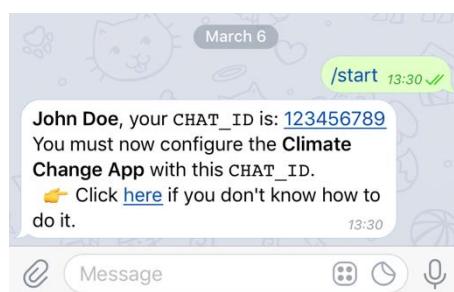
3. De vuelta a la línea de comandos, deberá pulsarse «**Enter**». Aparecerá el siguiente mensaje:

```
...
Hit "Enter" when done:
> You should have received a message from the bot.

Now, you need to configure the Climate Change App with the CHAT_ID you have
received.
Hint: If you don't know how to do it, check it out the docs, available here:
https://github.com/diego-hermida/ClimateChangeApp/wiki
IMPORTANT: If you delete the chat with the bot, don't use the Delete and Stop
option, but Delete (otherwise, you'll block the incoming messages from the
bot!).

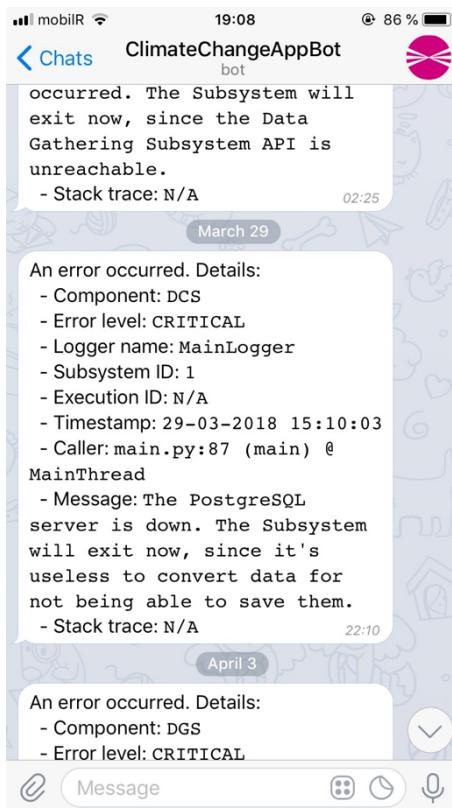
[SUCCESS] The Telegram Configurator was successful.
```

Además, en el dispositivo, habrá llegado un mensaje del *bot*:



4. Como se indica en el mensaje, debe configurarse el sistema con el `CHAT_ID` presente en el mensaje. Para ello, debe registrarse en el fichero de configuración global del sistema (`/global_config/config.config`), en la propiedad `TELEGRAM_CHAT_ID`. Por defecto, ésta es `null`, por lo que no se enviarán mensajes.

Una vez hecho esto, se habrá configurado satisfactoriamente el *bot*, y se empezarán a recibir mensajes de *log* si algún componente falla:



Como se puede apreciar en los mensajes, se muestra información detallada de los errores:

- El tipo de componente: **DCS**, **DGS**, **API**, **WEB...**
- El nivel del error: **ERROR** o **CRITICAL**. El sistema permite configurar el mínimo nivel de error para que se envíen mensajes vía Telegram. Por defecto, se establece a **CRITICAL**.
- El nombre del *logger* que registró el mensaje.
- El ID del subsistema: Esto es interesante si se ha escalado el subsistema, ya que permite conocer qué instancia concreta ha fallado. Para profundizar sobre este aspecto, puede consultarse la sección [8.5. Escalabilidad](#).
- El ID de la ejecución: Si está disponible, permite conocer qué ejecución ha fallado. Esto es útil para explorar los ficheros de *log* con mayor precisión.
- El momento en el que se ha producido el error.
- Detalles de la llamada que produjo el error: fichero, línea, método y *thread*.
- El mensaje derivado del error.
- El *stack trace* del error, si está disponible.

C.3.1. Errores de configuración

Deben tenerse en cuenta tres aspectos que pueden llevar a la aparición de errores. Son:

- Que, durante la configuración del componente, el *bot* no haya recibido el mensaje para iniciar la conversación con éste, o que hayan pasado más de 5 minutos entre la llamada a `/start` y el *click* del botón «Enter»⁴⁸:

```
[INFO] Running the Telegram Configurator component.  
> Using Climate Change App Telegram Bot configurator v1.0.  
  
Start a chat with the Telegram bot @ClimateChangeAppBot.  
Hit "Enter" when done:  
  
The bot hasn't received recent messages. Did you start the chat with the  
bot?  
Hint: If you already have an open chat with the bot, you can either send it  
/start, or remove the chat and start a new one.  
  
[ERROR] The Telegram Configurator did not exit normally. You should rerun  
this installer.
```

- Que se haya **bloqueado** al *bot*. Lo más probable es que el usuario haya eliminado el chat con éste, mediante la opción «Delete and Stop», en lugar de simplemente «Delete». Para solventarlo, debe volverse a iniciar una conversación con el *bot*, según lo explicado en la sección anterior:

```
[INFO] Running the Telegram Configurator component.  
> Using Climate Change App Telegram Bot configurator v1.0.  
  
Start a chat with the Telegram bot @ClimateChangeAppBot.  
Hit "Enter" when done:  
Warning: User Diego Hermida Carrera has blocked the bot, so he/she cannot  
receive messages.  
The user has blocked the bot, so he/she cannot receive messages from it!  
IMPORTANT: If you delete the chat with the bot, don't use the Delete and  
Stop option, but Delete (otherwise, you'll block the incoming messages from  
the bot!).  
  
[ERROR] The Telegram Configurator did not exit normally. You should rerun  
this installer.
```

⁴⁸ por defecto, el componente Telegram solo recupera mensajes `/start` recibidos durante los últimos 5 minutos. En caso de no establecer un tiempo máximo, se enviarían mensajes a personas que, probablemente, ya hubieran configurado correctamente el *bot*.

No obstante, este parámetro es modificable, mediante el fichero de configuración del componente: `/telegram_bot/config/config.config`, en la propiedad `MAX_VALID_TIME`.

- Que se no se haya establecido la propiedad TELEGRAM_CHAT_ID en el fichero de configuración global del sistema. En este caso, ante un error CRITICAL, el componente no podrá enviar mensajes. No obstante, el evento se registrará, así como la imposibilidad de enviar el mensaje vía Telegram:

```
[INFO] [DCS] [ID:1] [EXEC:2] 06-03-2018 03:42:32.654 {main.py:88
(main) @ MainThread}: Determining if the Data Gathering Subsystem
API is up.

[CRITICAL] [DCS] [ID:1] [EXEC:2] 06-03-2018 03:42:37.674 {main.py:93
(main) @ MainThread}: An HTTP error occurred. The Subsystem will exit
now, since the Data Gathering Subsystem API is unreachable.

[WARNING] [DCS] [ID:1] [EXEC:N/A] 06-03-2018 03:42:37.674
{log_util.py:55 (emit) @ MainThread}: Error message could not be sent
via Telegram. Cause: TELEGRAM_CHAT_ID is None.
```

ANEXO D. CONVENCIONES DE ETIQUETADO

En este anexo se describen las convenciones de etiquetado de algunos artefactos de la aplicación: las *releases* del sistema, y los *bugs* que surgen durante el desarrollo, cada uno en su propia sección:

D.1. Etiquetado de *releases*

Las *releases* conforman versiones del producto susceptibles de ser puestas en producción. En relación el etiquetado de éstas, se emplean convenciones de nombrado. Se anotarán, con:

- **x.0** las versiones que suponen un incremento funcional significativo sobre la versión anterior, como la implementación de un nuevo subsistema.
- **x.y** las versiones que añadan características a componentes ya existentes.
- **x.y.z** las versiones que solamente resuelvan algún *bug*.

Además, se podrá añadir al nombre de la versión los *tags*:

- **-alpha**: Si las funcionalidades añadidas por la *release* no han sido probadas.
- **-beta**: Si las funcionalidades añadidas han sido probadas, pero no en el entorno de producción.

Esta etiqueta aparece en varios lugares a lo largo de todo el proyecto. En concreto:

- En el fichero de configuración global de la aplicación, en la propiedad **APP_VERSION**.
- En los *scripts* de instalación de los componentes del sistema, al invocar éstos con la opción **--version**.
- En el repositorio donde se almacena el código del proyecto, en GitHub.
- En todos los reportes almacenados por los componentes supervisor de los Subsistemas de Recolección y Conversión de Datos.

Toda la información relativa a las *releases* del sistema están documentadas en el repositorio del proyecto, accesible desde la siguiente URL:

<https://github.com/diego-hermida/ClimateChangeApp/releases>

D.2. Etiquetado de *bugs*

Los *bugs* (errores) se caracterizan empleando las series **[BUG-xxx]** para errores del sistema, y **[EXT-xxx]** para errores derivados de fuentes externas.

Cuando aparece un *bug*, se proporciona una descripción y, si es posible, los pasos para su reproducción. Éstos se mueven entre las listas:

- *Not fixed*: El error ha sido detectado, pero no se han establecido acciones de ningún tipo para su solución.
- *Fixing*: El error está siendo solucionado.
- *Provisionally fixed*: El error ha sido solucionado, pero aún no se han realizado pruebas en el entorno de producción.
- *Fixed*: El error ha sido solucionado, y la *release* que solventa el error puesta en producción.

Además, a los *bugs* se les puede añadir las siguientes etiquetas:



Ilustración 42: Etiquetas para caracterizar bugs del sistema.

Pese a que se considera que las etiquetas son claras; se comentan, a continuación, una serie de consideraciones:

- La etiqueta **Heisenbug** hace referencia a errores **difíciles de detectar**, dado a que se dan solo bajo ciertas circunstancias [30].
- La etiqueta **Pending** solo es usada por errores de tipo externo para indicar que se está a la espera de acciones de terceras partes.

Como último apunte, un *bug* puede poseer múltiples etiquetas.

Toda la información relativa a los *bugs* del sistema está documentada empleando la herramienta Trello, de la que se hablará en la sección 5.12 *Trello*. El lector puede acceder a la documentación en el siguiente enlace:

<https://trello.com/b/QytZwcat>

Nota: Para acceder a la documentación, es necesario iniciar sesión en Trello. Se ha creado una cuenta con permisos, a tales efectos:

Usuario: `tfg_dhc_examiner@vpslists.com`

Contraseña: `examiner`

ANEXO E. CONTENIDO DEL CD

El CD que se adjunta con el presente documento contiene los archivos:

- **HermidaCarreraDiego_2018.pdf**, que se corresponde con la versión electrónica de esta memoria⁴⁹.
- **HermidaCarreraDiego_2018_code.zip**, que contiene el código fuente del sistema.
- **HermidaCarreraDiego_2018_mockups.zip**, que alberga los *mockups*, o el prototipo no funcional de la aplicación web.
- **HermidaCarreraDiego_2018_screenshots.zip**, que contiene las capturas de pantalla de la aplicación web funcional. A su vez, este fichero contiene los subdirectorios **iPhone**, **iPad** y **Desktop**; donde se almacenan las capturas correspondientes a la versión móvil, de una *tablet* y de un dispositivo de escritorio, respectivamente.
- **HermidaCarreraDiego_2018_locations.xlsx**, que posee los cálculos del proceso de selección de las localizaciones, comentado con anterioridad en el [ANEXO A. SELECCIÓN DE LOCALIZACIONES A MONITORIZAR](#).
- **HermidaCarreraDiego_2018_data_sources.xlsx**, que contiene detalles de las fuentes de datos de terceros, recopiladas durante la fase de análisis del proyecto, y comentadas en la sección [4.3. Recopilación de fuentes de información](#).
- **HermidaCarreraDiego_2018_diagrams.zip**, que recopila la versión a tamaño completo de todos los diagramas incluidos en la memoria.

⁴⁹ Para los lectores cuyo dispositivo posea el sistema operativo **macOS**, se recomienda emplear un visualizador de PDF como Adobe Acrobat Reader DC, ya que la aplicación nativa *Preview* (*Vista Previa*) no renderiza bien ciertos aspectos del documento.

ANEXO F. DOCUMENTACIÓN ADICIONAL

Como ya se comentó a lo largo de toda la memoria, existe documentación del proyecto accesible desde Internet.

El objetivo de esta sección es recopilar los orígenes de dicha documentación, de forma que el lector pueda consultarla sin tener que navegar por todo el fichero:

- a) Repositorio de código del proyecto:

<https://github.com/diego-hermida/ClimateChangeApp>

- b) Documentación relacionada con múltiples aspectos del sistema: diseño, implementación, diagramas, instalación, configuración, escalabilidad, etc.: Esta documentación –en inglés– es accesible desde la Wiki del repositorio:

<https://github.com/diego-hermida/ClimateChangeApp/wiki>

- c) Documentación del API del Subsistema de Recolección de Datos:

https://app.swaggerhub.com/apis/TFG_DiegoHermida/API/1.0.0-oas3

- d) Informes y estado de la construcción del sistema, mediante la disciplina de Integración Continua:

<http://193.144.50.92:8090/job/ClimateChangeApp/>

Nota: Para acceder con mayores privilegios, debe iniciarse sesión en Jenkins con el usuario **examiner**, y contraseña **examiner**.

- e) Parámetros de calidad del proyecto, mediante la disciplina de Inspección Continua:

<http://193.144.50.92:9000/dashboard?id=ClimateChangeApp>

Nota: Para acceder con mayores privilegios, debe iniciarse sesión en SonarQube con el usuario **examiner**, y contraseña **examiner**.

- f) Lista de *bugs* del sistema:

<https://trello.com/b/QytZwcat>

Nota: Para acceder con mayores privilegios, debe iniciarse sesión en Trello con el correo **tfgh_dhc_examiner@vpslists.com**, y contraseña **examiner**.

Haciendo *click* en el nombre del equipo ( TFG Diego Hermida Carrera Free), se puede acceder al resto de tableros, que contienen la planificación y desarrollo de los *sprints*, detalles sobre la documentación y trabajo técnico, etc.

Bibliografía

- [1] L. Alvarez, W. Alvarez, F. Asaro y H. V. Michel, «Extraterrestrial Cause for the Cretaceous-Tertiary Extinction» Science, Volume 208, Issue 4448, pp. 1095-1108, 1980.
- [2] J. M. Viñas Rubio, «El Clima de la Tierra a lo largo de la Historia” 7-9 mayo 2012.
<http://www.divulgameteo.es/uploads/Clima-Tierra-historia-JMV.pdf>
- [3] Intergovernmental Panel on Climate Change (IPCC), «Climate Change 2014: Synthesis Report” 2014.
https://www.ipcc.ch/pdf/assessment-report/ar5/syr/SYR_AR5_FINAL_full_wcover.pdf
- [4] T. McCarthy, «Antarctica records unprecedeted high temperatures in two new readings” 31 marzo 2015.
<https://www.theguardian.com/environment/2015/mar/31/potential-record-high-temperature-in-antarctica-alarms-scientists>
- [5] B. A. Kader, «Temperature dips to record low in the country” 23 enero 2008.
<http://gulfnews.com/news/uae/environment/temperature-dips-to-record-low-in-the-country-1.79161>
- [6] J. Gould-Bourn, «Popsicles Made From 100 Different Polluted Water Sources Grab World's Attention” 8 junio 2017.
<https://www.boredpanda.com/polluted-water-popsicles-taiwan/>
- [7] NASA, «Artic Sea Ice Minimum” 21 junio 2017.
<https://climate.nasa.gov/vital-signs/arctic-sea-ice/>
- [8] Sustainable Energy Authority of Ireland, «What are Ireland's renewable energy targets?”
http://www.seai.ie/Energy-Data-Portal/Frequently-Asked-Questions/Energy_Targets_FAQ/
- [9] United Nations, «The Paris Agreement” 2017.
http://unfccc.int-paris_agreement/items/9485.php
- [10] D. Furphy, «What on earth is an RPC?” 28 septiembre 2013.

- <https://medium.com/@davidfurphy/what-on-earth-is-an-rcp-bbb206ddee26>
- [11] National Ocean Service (U.S. Department of Commerce), «What are El Niño and La Niña?» 28 junio 2016.
<http://oceanservice.noaa.gov/facts/ninonina.html>
- [12] NOAA ESRL, «Climate Change: Atmospheric Carbon Dioxide» 12 junio 2017.
<https://www.climate.gov/news-features/understanding-climate/climate-change-atmospheric-carbon-dioxide>
- [13] D.A. Smith, CASA, UCL, EC JRC, CIESIN, «World Population Density» 2015.
<http://luminocity3d.org/WorldPopDen/#3/28.54/50.71>
- [14] The Sceptic Science Team, «The 97% consensus on global warming» 2016.
<https://www.skepticalscience.com/global-warming-scientific-consensus-advanced.htm>
- [15] United Nations, Overseas Development Institute, Ipsos Mori, «MYWorld2015 Analytics» 2015.
<http://data.myworld2015.org/>
- [16] Centro de Investigaciones Sociológicas (CIS), «Barómetro de febrero 2017» 1-9 febrero 2017.
<http://estaticos.elmundo.es/documentos/2017/03/06/CIS.pdf>
- [17] Eurostat, «Gender Statistics Database - Internet Use» 2016.
http://eige.europa.eu/gender-statistics/dgs/indicator/ta_scitech_ict_int_isoc_ci_ifp_iu/bar
- [18] A. Jesse, «Grok the GIL: How to write fast and thread-safe Python» 18 04 2017.
<https://opensource.com/article/17/4/grok-gil>
- [19] The Internet Society, «RFC 2616 (section 10)»
<https://tools.ietf.org/html/rfc2616#section-10>
- [20] GNU, «Standards for Command Line Interfaces»
https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html
- [21] M. G. Pascual, «Los mejores salarios seguirán siendo para los ingenieros» 17 11 2018.

- https://cincodias.elpais.com/cincodias/2016/05/10/sentidos/1462903664_151432.html
- [22] Fondo Europeo de Desarrollo Regional (FEDER), «Guia del análisis costes-beneficios de los proyectos de inversión”
http://ec.europa.eu/regional_policy/sources/docgener/guides/cost/guide02_es.pdf
- [23] H. Chen y D. Chen, «Köppen climate classification” 10 enero 2017.
<http://hanschen.org/koppen/>
- [24] United Nations, «Human Development Index - HDI” 2017.
<http://www.investopedia.com/terms/h/human-development-index-hdi.asp>
- [25] The Statistic Portal, «Number of monthly active Twitter users worldwide from 1st quarter 2010 to 1st quarter 2017 (in millions)” 2017.
<https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/>
- [26] C. G. Terol, «#Hashtag: ¿Qué es, para qué sirve, y cómo usarlo?”
<https://carlosguerraterol.com/hashtag-que-es-para-que-sirve-como-usar/>
- [27] Hashtagify, «All-time Top 10 Hashtags related to {hashtag}” 2017.
<http://hashtagify.me/>
- [28] G2Crowd, «Best Continuous Integration Software” 2018.
<https://www.g2crowd.com/categories/continuous-integration?segment=all>
- [29] J. Petazzoni, «Using Docker-in-Docker for your CI or testing environment? Think twice.”
<https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/>
- [30] StackOverflow, «Clarification of what a heisenbug is” 27 03 2015.
<https://stackoverflow.com/questions/29297896/clarification-of-what-a-heisenbug-is>
- [31] M. Lutz, Learning Python, 5th Edition.