UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

2019

Centro de Innovación para la Sociedad de la Información

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Visualización de eventos de tráfico en Londres en tiempo real utilizando Clúster Hadoop y Spark Streaming

GRADO EN INGENIERÍA INFORMÁTICA. ULPGC - CICEI

BETANCOR OLIVARES, JUAN RAMÓN
TUTOR: SÁNCHEZ MEDINA, JAVIER JESÚS

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Tabla de contenido

1. Introducción: estado actual y objetivos iniciales 4
2. Justificación de las competencias específicas cubiertas 5
3. Aportaciones 6
4. Desarrollo 7
   4.1. Entorno de trabajo. Requisitos mínimos 7
   4.2. Descargando Spark 8
   4.3. Aplicación Scala 9
      4.3.1. Desglosando y estructurando la aplicación 9
      4.3.2. Tecnologías utilizadas y entorno de desarrollo. 9
      4.3.3. Creación de la aplicación 11
      4.3.4. Trabajando con las incidencias de TLP 16
      4.3.5. Trabajando con la API de Twitter 31
      4.3.6. Repositorio público con el código fuente 41
      4.3.7. Posibles propuestas de mejora para la aplicación. 42
   4.4. Rest API Spring Boot 42
      4.4.1. Desglosando y estructurando la aplicación + Diseño previo 42
      4.4.2. Tecnologías utilizadas y entorno de desarrollo. 44
      4.4.3. Creación de la aplicación 46
      4.4.4. Configurando la seguridad de la aplicación (Spring Security). 77
      4.4.5. Despliegue a Heroku 83
      4.4.6. Repositorio público con el código fuente de la aplicación 84
      4.4.7. Posibles propuestas de mejora para la aplicación 84
   4.5. Resultado final del proyecto + Demo. Ejecución simultanea de ambas aplicaciones. 85
5. Conclusiones y trabajos futuros 86
6. Normativa y legislación 87
7. Glosario de términos 87
8. Tabla de ilustraciones 90
9. Fuentes de información 93
   9.1. Referentes a la aplicación. 93
   9.2. Referente a la memoria y puesta en marcha de la aplicación 95
   9.3. Repositorios y DEMO de la aplicación 95

1. Introducción: estado actual y objetivos iniciales

Es un hecho que cada familia dispone de al menos un vehículo con el que sale a las calles todos los días. Las aglomeraciones, las incidencias de tráfico, cortes de carretera son aspectos cotidianos e inevitables que nos vamos a encontrar, y cada vez más, cada vez que tengamos que salir a la calle.

Como decía el artículo publicado en La Vanguardia, una persona, dependiendo de la ciudad donde vive, puede estar hasta un 13% del tiempo que gasta en las carreteras en retenciones, esto es, por ejemplo en Londres, unas 74 horas al año. Cada persona debe de concienciarse de este hecho y actuar acorde a ello.

¿Y si somos capaces de ofrecer alguna herramienta que permita en tiempo real la visualización de estas retenciones de nuestra ciudad, de tal manera que podamos tomar carreteras secundarias o rutas alternativas que permitan reducir el tiempo que uno está retenido en las carreteras?

Cuando buscamos una ciudad en Google Maps y hacemos zoom a una determinada zona, ¿verdad que podemos ver diferentes establecimientos como gasolineras, tiendas, en las cuales, si clicamos, nos ofrecen una cierta información como el horario de apertura y cierre?

Pues, ¿y si creamos una aplicación similar, pero en lugar de establecimientos, mostramos incidencias de tráfico en tiempo real? Un ejemplo claro puede ser la herramienta eTraffic que proporciona la DGT para la visualización de incidencias de carreteras en ámbito nacional, pero la nuestra orientada a una determinada ciudad.

Por lo tanto, nuestro objetivo principal será representar en un mapa las incidencias de tráfico activas de una determinada ciudad, en este caso Londres, debido a que el *dataset* proviene de ahí, además de una integración con Twitter para buscar, en tiempo real, posibles Tweets de lo que la gente de la ciudad habla sobre dichas incidencias. Todo esto dividido en dos aplicaciones, una aplicación Scala utilizando Spark Streaming que se encargará de la obtención, el tratamiento y el envío de los datos ya filtrados a otra aplicación, una aplicación web, encargada de representar dichos datos.

Obtendremos los datos a través de los denominados receptores o *receivers*, en caso de las incidencias, recopilaremos tan sólo las incidencias activas y tan sólo los campos importantes de cada una de ellas, como pueden ser la localización, descripción, etc. En el caso de los tweets, nos quedaremos con los tweets que provengan de Londres y los tweets cuyo *análisis de sentimientos* nos devuelva una valoración neutra o superior. Una vez tengamos la información deseada sólo debemos de crear nuestro objeto JSON con dicha información para enviarlo a la Rest API.

Desde la aplicación web, recogeremos los datos y los haremos persistentes utilizando una base de datos, luego enviaremos los datos almacenados a la vista, la cual se encargará de mostrar la información al usuario.

Por lo tanto, se dividirá en dos bloques principales el contenido de la memoria, cada bloque hará referencia a cada aplicación. En el primer bloque hablaremos de la obtención de los datos, el

tratamiento de este y la preparación para el envío. En el segundo bloque, veremos cómo obtenemos los datos y los hacemos persistentes, como mostramos dichos datos y hablaremos sobre aspectos de seguridad de la aplicación.

También a lo largo de la memoria utilizaremos una serie de convenciones para mejorar la lectura de este:

- El código fuente se expondrá mediante capturas de la propia aplicación de tal manera que sea más fácil la lectura de esta.
- Al final de la memoria se encontrará el glosario de términos, donde se expondrán las definiciones de los términos más importantes y usados a lo largo de la memoria.
- Todas las imágenes están enumeradas con su correspondiente título. La tabla de ilustraciones se podrá consultar al final de la memoria.

Una vez está expuesta las intenciones de este proyecto, ¿por qué este y no otro?

Debido a que durante los últimos 4 años carrera se ha puesto un poco de moda o se ha dado a conocer más, por así decirlo, el Big Data y el análisis de datos, aspecto el cual no tratamos en la carrera, siempre he querido dedicarle un tiempo a este ámbito para probar y, quién sabe si seguir adelante por este camino en un futuro. Se ha presentado una oportunidad perfecta para hacer un TFG en colaboración con el CICEI de tal manera que pueda aprender de este mundo y a la par acabar mis estudios. Otra de las alternativas y, por la que inicialmente me quería decantar, era por realizar un proyecto relacionado con la ciberseguridad, pero quise iniciar en el mundo del Big Data porque era desconocido totalmente para mí y quería adentrarme en él.

Es momento de iniciar la memoria donde explicaré paso a paso como crear nuestro propio dispositivo de visualización de incidencias de tráfico y de tweets relacionados, el cual es el objetivo principal de este trabajo de fin de carrera.

2. Justificación de las competencias específicas cubiertas

TI02: capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes, dentro de los parámetros de coste y calidad adecuados.

Durante la realización de las aplicaciones me he tenido que decantar por utilizar ciertas tecnologías y descartar otras, por ejemplo, utilizar SCALA o Java, realizar un proyecto Spring Boot con Java o con Ruby on Rails para realizar la aplicación web, etc., teniendo en cuenta ciertos parámetros. El parámetro que más resalta sin duda alguna es el conocimiento previo que se tiene de cada tecnología, así como, los costes computacionales que requiere la puesta en marcha de las aplicaciones (teniendo en cuenta las características del computador que he usado durante el desarrollo del TFG). También he tenido que integrar varios lenguajes de programación para que funcionen en conjunto, Scala, Java, HTML, CSS, JavaScript, JQuery, librerías externas de Scala, Spark, etc.

TI03: capacidad para emplear metodologías centradas en el usuario y la organización para el desarrollo, evaluación y gestión de aplicaciones y sistemas basados en tecnologías de la información que aseguren la accesibilidad, ergonomía y usabilidad de los sistemas.

No hay que olvidar que estamos construyendo una herramienta que será utilizada por una gran variedad de perfiles de usuario, por lo que habrá que adaptar el estilo y la funcionalidad del aplicativo para que cualquier persona pueda utilizarlo sin problema alguno. Para ello, hay que resaltar desde un principio qué se puede hacer con la aplicación y cómo acceder dichas herramientas, ayudado de un diseño intuitivo de tal manera que el usuario desde un primer vistazo sepa como navegar en la aplicación.

Todo esto sin olvidar aspectos también importantes como la seguridad de la aplicación, usando para ello librerías de gestión de cuentas de usuario como Spring Security, de tal manera que sea necesario acceder con una determinada cuenta al contenido de la aplicación.

Hemos podido mejorar la fiabilidad de la aplicación integrando persistencia de los datos, de tal manera que gracias a una base de datos PostgreSQL, podemos almacenar toda la información y que, en caso de caídas en el servidor o problemas en ambas aplicaciones, podamos recuperar toda la información guardada de anterioridad.

TI04: capacidad para seleccionar, diseñar, desplegar, integrar y gestionar redes e infraestructuras de comunicaciones en una organización.

El trabajo realizado en el clúster MIMD, proporcionada por el CICEI para la ejecución y la obtención de pruebas del resultado de la ejecución de la aplicación. Instalación de las herramientas necesarias, como los entornos y kits de desarrollo, librerías, etc. en las diferentes máquinas utilizadas en la realización del proyecto.

TI06: capacidad de concebir sistemas, aplicaciones y servicios basados en tecnologías de red, incluyendo Internet, web, comercio electrónico, multimedia, servicios interactivos y computación móvil.

Sería el resultado de la aplicación terminada donde integraría una gran cantidad de tecnologías tales como internet, aplicación web, fuentes multimedia (imágenes), redes sociales (Twitter), etc.

3. Aportaciones

Como comenté anteriormente en la introducción, es común las incidencias de tráfico en las carreteras todos los días. Si de alguna manera, podemos ofrecer una herramienta a la sociedad donde se vea de manera gráfica y visual donde está presente cada incidencia y toda la información relacionada de esta, permite al usuario crear una trayectoria secundaria que reduzca el tiempo que gasta en las carreteras (ahorro de tiempo). Si miramos desde el punto de vista del propio ayuntamiento, por ejemplo, permite tener un mapa con todas las incidencias

activas, permitiendo una mejor coordinación entre los activos que estén solucionando cada incidencia y una mayor optimización de las tareas.

El uso de Twitter en este caso nos ofrece una idea de cómo reacciona la sociedad a dichas incidencias de tráfico (análisis de sentimientos), e incluso, podría permitir localizar nuevas incidencias en las carreteras que aún no han sido notificadas.

Respecto al ámbito económico no habría apenas gastos en la puesta en marcha del aplicativo, tan sólo el hosting de la web y un computador (no necesariamente un clúster) donde podamos tener nuestro programa en continua ejecución, aunque es importante tener herramientas de seguridad en caso de fallo. Una posible opción es utilizar un clúster virtual para este caso en concreto, de tal manera que ahorraremos en gastos de hardware, pero tendremos las ventajas computacionales de un clúster y la seguridad de ésta. Este aplicativo tampoco tendría un impacto positivo desde el punto de vista económico para el propietario, ya que su principal objetivo es ofrecer un servicio al público desde, se presupone, una administración u organización pública, como puede ser un ayuntamiento. Nunca tendrá una finalidad de generar capital, aunque de usarlo siempre podría generar algún ahorro por los beneficios que aporta el uso del aplicativo.

Y desde el punto de vista de aportaciones al ámbito tecnológico, esta es prácticamente nula ya que la herramienta no es ni revolucionaria ni novedosa (no pretende serla, sólo ofrecer un servicio), ya que existen ya varios ejemplos, como puede ser eTraffic, como ya vimos al inicio de la memoria.

4. Desarrollo

Durante este apartado, mostraremos paso a paso cómo crear nuestras dos aplicaciones de las que consta la herramienta, es decir, la aplicación Scala y la aplicación web.

4.1. Entorno de trabajo. Requisitos mínimos.

Durante la realización de las aplicaciones se han utilizado dos equipos:

- Portátil: Huawei MateBook D.
  - Intel Core i5-8250 CPU QuadCore @1.60GHz – 1.80GHz.
  - 8GB RAM.
  - Windows 10 de 64 Bits.
- PC Sobremesa:
  - Intel Core i7-6700k CPU QuadCore @4.00GHz – 4.20GHz
  - 16GB RAM.
  - Windows 10 de 64 Bits

Cuando tengamos las dos aplicaciones terminadas, podemos tener los dos entornos de desarrollos ejecutándose simultáneamente (aunque podemos apoyarnos de Heroku para disminuir la carga de la aplicación web dentro de la máquina). Por lo tanto, se requiere tener, al

menos, un procesador de 4 núcleos (al menos un i5 de Intel para que vaya todo medianamente fluido). Además de 8 GB de RAM, ya que debemos tener dos entornos de desarrollo ejecutándose simultáneamente. Con relación al sistema operativo, se podría hacer sin problema alguno sobre una máquina Linux, ya que ambos entornos de desarrollo y las tecnologías usadas son compatibles con ambos sistemas operativos.

4.2. Descargando Spark

El primer paso es descargar Spark en nuestro equipo ya que necesitaremos, al menos, los JARS para nuestra aplicación. También podremos usar el Shell de Spark para ejecutar nuestra aplicación, aunque nosotros lo haremos mediante el entorno de desarrollo para mayor comodidad.

Tan sólo tenemos que acceder a la página oficial de Spark y darle al botón de descargar. Nos aparecerá dos opciones para modificar las versiones, podemos dejarlo tal y como está, es decir, usar las últimas versiones disponibles. Clicamos finalmente en "*Download Spark*".

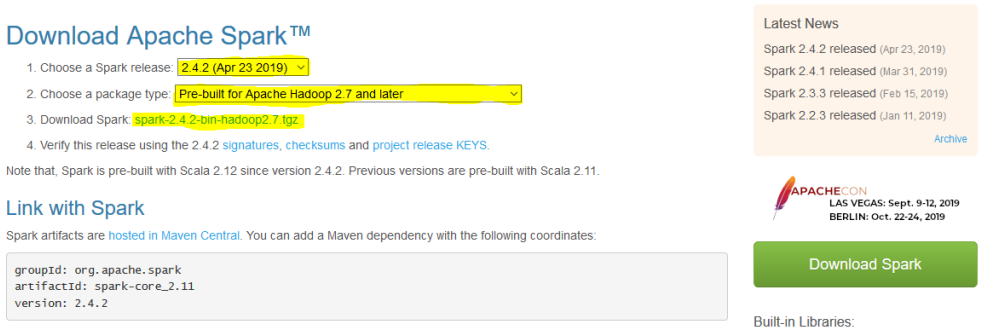


Ilustración 1. Descargando Apache Spark

Y en la siguiente ventana elegiremos los *mirrors* HTTP para proceder a la descarga (seleccionamos cualquiera de las dos opciones que se nos presenta):

Ilustración 2. Mirrors HTTP para descargar Apache Spark

Una vez descargamos el fichero, procederemos a descomprimirlo en una carpeta aparte, recomendable nombrar dicha carpeta con un nombre fácil de reconocer como Spark, ya que más adelante accederemos al contenido de la carpeta para acceder a las librerías. La carpeta importante que usaremos será la de nombre "*jars*", que contiene toda la librería Spark. Una vez

hecho esto, ya estamos listos para continuar, no es necesario hacer ninguna instalación en este paso.

4.3. Aplicación Scala

Esta aplicación, como ya hemos adelantado, será la encargada de adquirir los datos sin tratar, aplicar los diferentes filtros y posteriormente enviar la información filtrada a la aplicación web. Es la aplicación fundamental del proyecto.

4.3.1. Desglosando y estructurando la aplicación

La aplicación se dividirá en dos secciones claramente diferenciadas:

- Sección de las incidencias de tráfico de TFL.
- Sección de los tweets utilizando la API de Twitter.

Por cada sección antes mencionadas podemos subdividir el código a su vez en tres bloques:

- Adquisición de la información sin tratar:
  - Sección incidencias: utilizando un *receiver* propio
  - Sección tweets: utilizando el receiver de Twitter API.
- Tratamiento de la información y filtrado:
  - Sección incidencias: utilizaremos varias librerías de Scala, como Scala XML.
  - Sección tweets: usaremos los métodos que nos brinda la API de Twitter para acceder a la información de los Tweets.
- Preparación y envío de la información ya tratada:
  - Para ambas secciones utilizaremos la librería Scala JSON para preparar el objeto que enviaremos a la aplicación web y, con la librería HTTP POST de Java, realizaremos el propio envío.

4.3.2. Tecnologías utilizadas y entorno de desarrollo.

Scala IDE for Eclipse.

Como entorno de desarrollo utilizaremos Scala IDE for Eclipse, no es nada más que un *plugin* de Scala para el entorno de Eclipse, el cual hemos utilizado en varias ocasiones durante la carrera, ya que, entre otras cualidades, es *open source*. También existen *plugins* para IntelliJ IDEA, el IDE de Java de JetBrains del cual disponemos una licencia de estudiantes, pero para que esta memoria se pueda replicar en un futuro, o por otras personas que no dispongan de dicha licencia, utilizaremos Eclipse, aunque si fuera por preferencia, elegiría IntelliJ (ofrece una mejor interfaz, más facilidades a la hora de programar, formateadores de código, etc.).

Ilustración 3. Logo ScalaIDE for Eclipse

Scala + librerías externas.

Como lenguaje de programación principal utilizaremos Scala, el cual integra características de lenguajes funcionales y de programación orientada a objetos (corre sobre una máquina virtual de Java y es compatible con las aplicaciones de Java existentes).

Además, se requerirá de librerías Scala externas para el tratamiento y el envío de los datos, estas son:

- Scala XML: librería que nos permite manejar estructuras de datos de estilo XML, permitiéndonos convertir estructuras de datos simples como puede ser un array o una *string* a formato XML (y viceversa) para acceder a los datos que se quieren de una manera ágil, sencilla y ordenada.
- ScalaJSON: JSON library for Scala (Play framework). Esta librería nos aporta una serie de métodos y herramientas para tratar estructuras de datos en formato JSON. Ofreciéndonos métodos para crear objetos de tipo JSON, transformar estos a *String* y viceversa (conversión), validadores (validación de la estructura y contenido), etc.

Ilustración 4. Logo Scala

Spark + Spark Streaming + Twitter API (sentiment analysis).

Spark es un sistema de computación en clúster de propósito general y orientado a la velocidad. Por otro lado, Spark Streaming utiliza el núcleo de Spark para el tratamiento de grandes flujos de información en lotes y en tiempo real. Utilizaremos Spark Streaming para la obtención del *dataset* con el que vamos a trabajar y junto a la API de Twitter, para acceder a los tweets en tiempo real.

Utilizaremos la librería de The Stanford NLP Group, para el análisis de sentimientos (sentiment analysis) de cada tweet, como herramienta de filtro.

Ilustración 5. Logos de Spark Streaming, Twitter API y de Stanford University respectivamente

Bitbucket

Como sistema de control de versiones, se utilizará en esta aplicación Bitbucket, el cual es un servicio de alojamiento basado en web, como GitHub, pero que ofrece también la posibilidad de crear repositorios privados de manera gratuita.

Ilustración 6. Logo de Bitbucket

4.3.3. Creación de la aplicación

Preparando el entorno

Una vez presentadas las tecnologías a utilizar, lo primero que hay que hacer es descargar el entorno de desarrollo, en este caso, *ScalaIDE for Eclipse*:

- Accedemos a la página oficial y clicamos en "*Download IDE"*. Posteriormente elegimos el sistema operativo que utilizamos en nuestra máquina y procedemos a la descarga.
- Una vez descargado, deberemos tener descargado el "Kit de Desarrollo de Java" (versión 8, *JDK* 8), ya que, en el proceso de instalación, o al crear un proyecto, nos lo solicitará el IDE. Se puede descargar en la página oficial.
- Una vez tengamos el IDE descargado, junto el JDK 8, ya procedemos a descomprimir el fichero para poder acceder al *launcher* (o ejecutable) de la aplicación: *eclipse.exe*. Todo esto sin necesidad de instalación.
- En caso necesario, la propia web ofrece una guía de inicio.

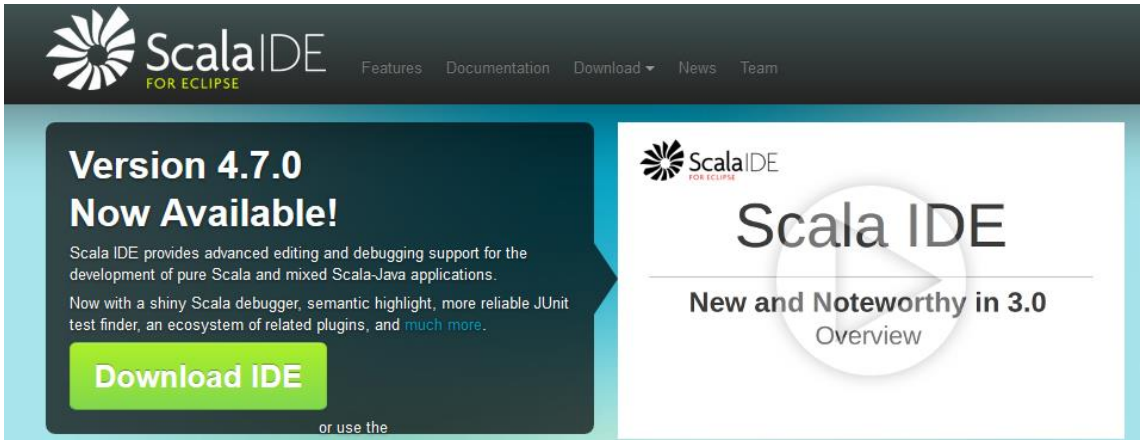


Ilustración 7. Descargando Scala IDE.

Creando el proyecto

Comenzamos creando el proyecto, para ello iniciamos el IDE y seleccionamos un espacio de trabajo (podemos dejarlo por defecto o crear una carpeta donde pondríamos todo lo relacionado con nuestro proyecto).

Nos dirigimos a: *File > New > Scala Project* en la esquina superior izquierda del IDE. Nos aparecerá la siguiente ventana.

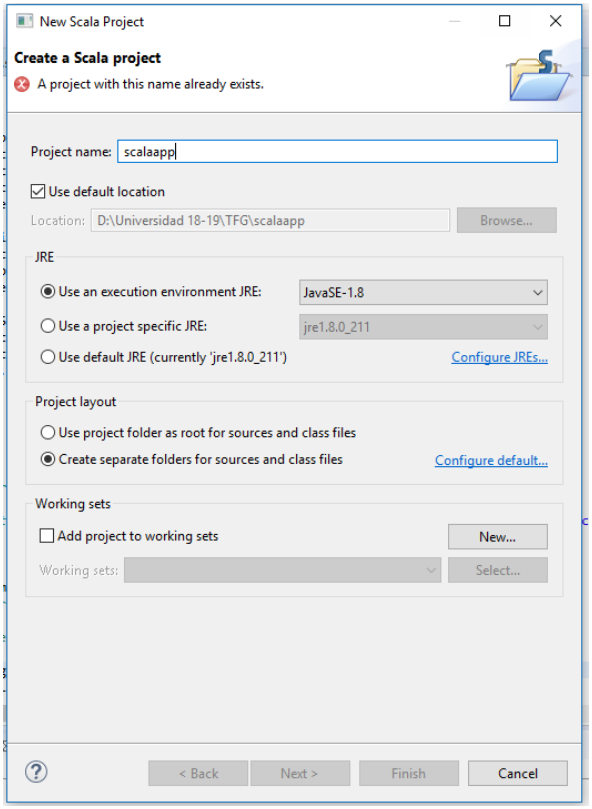


Ilustración 8. Configurando del proyecto

Tan sólo tendremos que poner un nombre a nuestra aplicación y seleccionar el kit de desarrollo de java que utilizaremos (si lo tenemos instalado en la carpeta por defecto nos aparecerá directamente), todas las demás opciones las dejaremos tal y como está.

Una vez hecho esto, ya tendremos nuestro proyecto creado y listo para empezar.

Creando nuestra main class

Cuando creamos una aplicación desde 0, nuestra composición del proyecto inicial debe de ser de esta manera o similar (librerías de Scala, librerías del JRE y carpeta *src*):

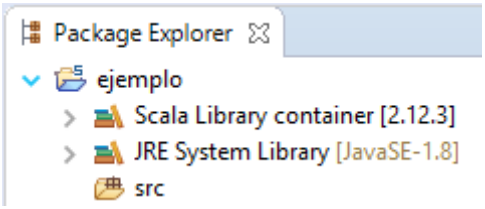


Ilustración 9. Estructura inicial del proyecto Scala

El primer paso será crear un "*Package*" donde crearemos todo el código de nuestra aplicación. Para ello hacemos clic derecho en la carpeta "*src*" mostrada en la imagen anterior y clicamos en *New > Package*. Nos aparecerá un cuadro de dialogo donde tendremos que poner el nombre deseado al paquete, siempre y cuando éste siga la convención de Java. Por ejemplo, de la siguiente manera:

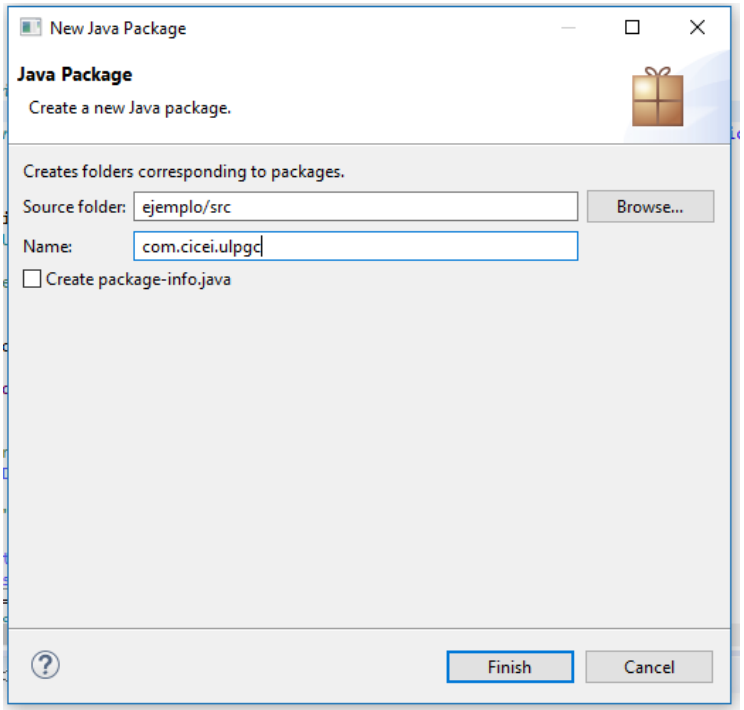


Ilustración 10. Añadiendo un package a nuestra aplicación.

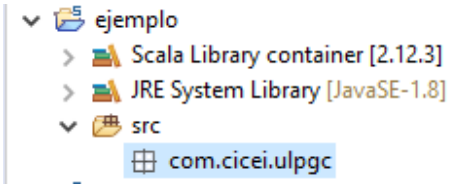


Ilustración 11. Paquete creado correctamente.

Ahora ya podemos crear nuestra *main class*, para ello, hacemos clic derecho sobre el paquete recién creado y vamos a *New > Scala Object.* Se nos abrirá un nuevo cuadro de diálogo donde nos pedirán el nombre del fichero y, por tanto, el nombre del objeto. Por ejemplo, se nos quedará de esta manera:

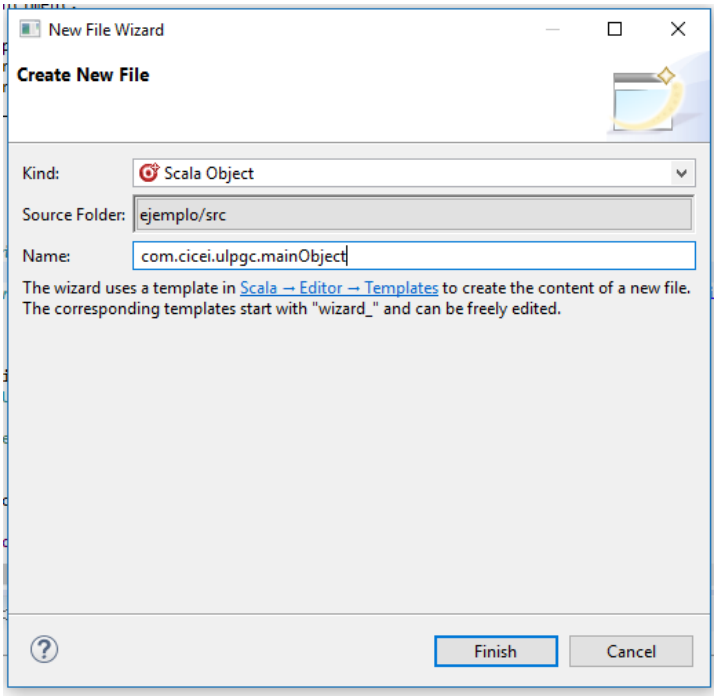


Ilustración 12. Creando la main class (Scala object)

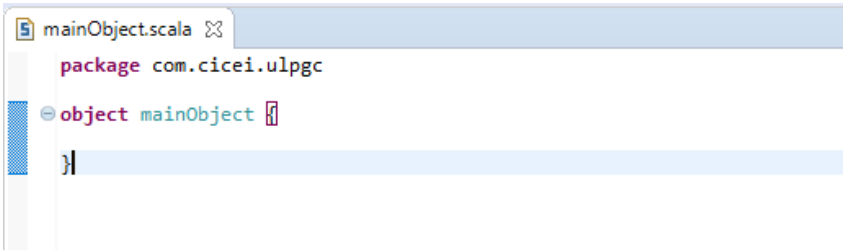


Ilustración 13. Resultado al crear el main.

Una vez tengamos el objeto de Scala creado, vamos a añadir nuestro método *main*, para ello seguimos esta notación:

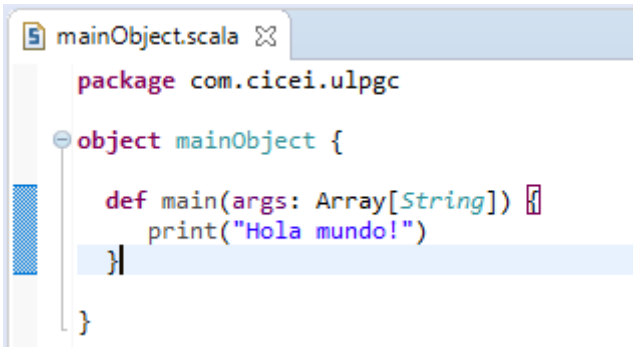


Ilustración 14. Creando el método main

Entrando un poco en detalle, vemos que la forma de escribir/declarar los métodos se asemeja bastante a Python (ambos son lenguajes especialmente funcionales). Utilizamos "*def*" seguido del nombre del método, en este caso utilizamos el nombre restringido *main* para indicar que será el método inicial desde donde se empezará a ejecutar el programa. Opcionalmente, entre paréntesis, indicaremos los argumentos o parámetros que utilice nuestro método, podemos utilizar *Array[String]* de tal manera que podemos aceptar más de un parámetro e incluso ninguno.

Como prueba, mostraremos por pantalla una frase mediante el método *print()* para cerciorarnos de que todo está en orden.

Cómo ejecutar nuestro programa

Siguiendo con el apartado anterior, para comprobar que todo está bien vamos a ejecutar nuestro programa. Para ello, clicamos en el menú desplegable del botón *run* y seleccionamos la opción "*run configurations".* Nos aparecerá una ventana de diálogo nueva. En el menú de la izquierda buscamos "*Scala Application"* y hacemos clic derecho sobre él y elegimos *new.* Nos aparecerá la siguiente ventana*:*

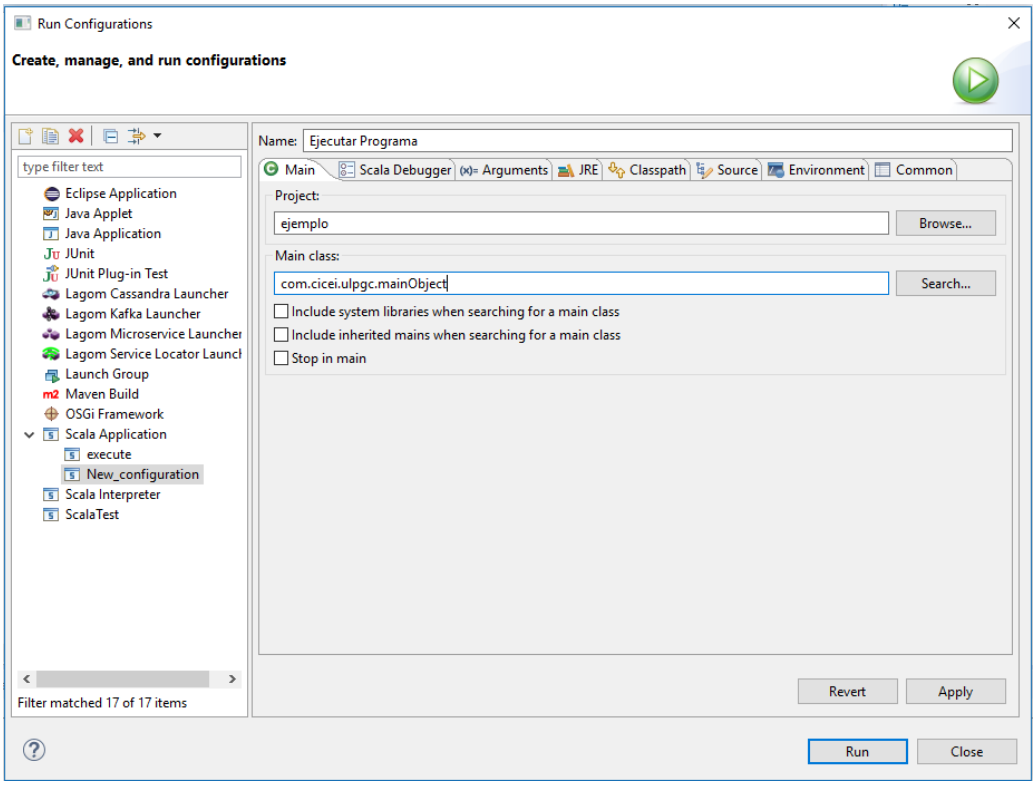


Ilustración 15. Configuración de ejecución del programa

Como vemos en la imagen anterior nos pedirá un nombre (podemos dejar el nombre por defecto) y la ubicación de la *main class*, para ello tenemos que poner el nombre de nuestro paquete y a continuación el nombre del objeto (respetando mayúsculas y separado por puntos). Una vez hecho esto, podemos ejecutarlo con el botón *Run* (para posteriores ejecuciones ya nos aparecerá esta configuración por defecto por lo que no tenemos que volver a hacerlo).

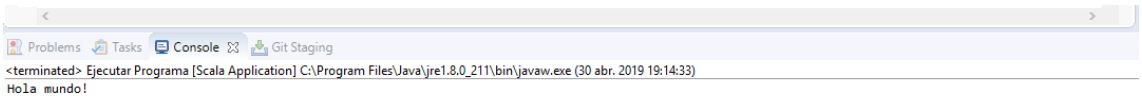


Ilustración 16. Resultado al ejecutar el método main

4.3.4. Trabajando con las incidencias de TLP

Creando el custom receiver para TLP

Una vez creada la aplicación con nuestro método *main* y hayamos comprobado que funciona todo correctamente ya podemos empezar a crear nuestra aplicación.

Lo siguiente que tenemos que hacer es registrarnos en la página oficial de TFL, en la cual podemos acceder al *dataset* de las incidencias de tráfico de Londres.

Una vez hayamos creado una cuenta ya podemos acceder a la lista de *feeds* que nos ofrece la API de la organización. A nosotros la que nos interesa es la *feed* de las incidencias de tráfico (*Live Traffic Disruptions - TIMS*). Para acceder a ella tan solo clicaremos en el enlace.

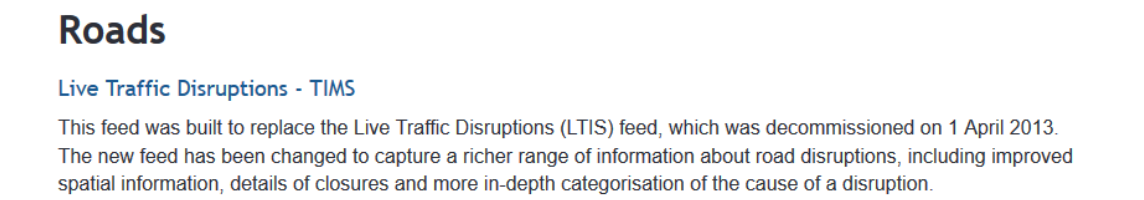


Ilustración 17. Dataset con los accidentes de tráfico (TIMS)

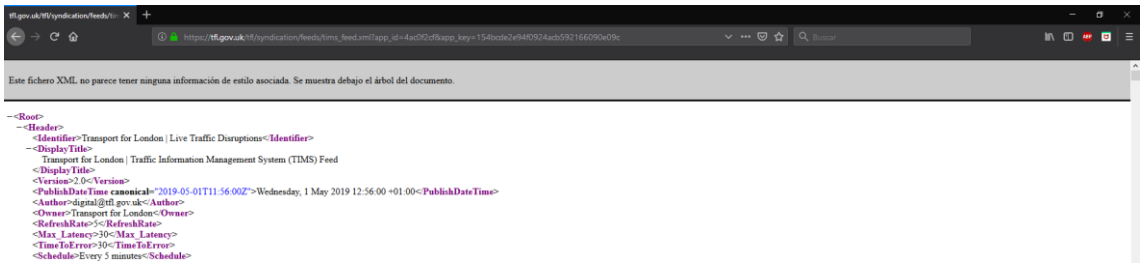


Ilustración 18. Dentro del XML de las incidencias

Vemos en la URL del XML que aparece automáticamente el ID y la KEY (*app_id* y *app_key* respectivamente) que se le asignó a nuestra aplicación en el proceso de creación de la aplicación, cuando nos pidió que introdujésemos el nombre de este. Es importante guardar esta URL ya que la usaremos posteriormente:

*https://tfl.gov.uk/tfl/syndication/feeds/tims_feed.xml?app_id=**4ac0f2cf**&app_key=**154bcde2e94f0924acb592166090e09c***

Más adelante comentaremos un poco más detenidamente como se estructura el XML, pero es importante observar el campo "*RefreshRate*" y "*Schedule*", que nos indica la frecuencia de actualización del *dataset*, en este caso, vemos que se actualiza cada 5 minutos:

```
<PublishDateTime canonical="2019-05-01T11:56:00Z">Wednesday, 1 May 2019 12:56:00 +01:00</PublishDateTime>
<Author>digital@tfl.gov.uk</Author>
<Owner>Transport for London</Owner>
<RefreshRate>5</RefreshRate>
<Max_Latency>30</Max_Latency>
<TimeToError>30</TimeToError>
<Schedule>Every 5 minutes</Schedule>
```

Ilustración 19. Tiempo de actualización del dataset.

El campo *PublishDateTime* nos indicará cuando se hizo la última actualización como vemos en la imagen anterior.

Ahora que ya tenemos acceso al *dataset*, ya podemos ponernos a elaborar el receiver para poder acceder a dichos datos. Para ello vamos a hacer uso de la página oficial de Spark Streaming, en la cual nos muestran un ejemplo de cómo debe de ser un *custom receiver*.

Antes que nada, un *receiver* no es más que un *thread* o hilo de ejecución, que en sí consiste en un bucle *while* infinito, siempre que no se pare intencionalmente, que descargará mediante una petición *get* el contenido del *dataset* (mediante la URL que hemos guardado previamente) y posteriormente se dormirá el hilo un determinado tiempo hasta que se repita el proceso. Este hilo implementará dos métodos, *onStart()* que se encargará de crear el hilo e iniciarlo y *onStop(),* que se encargará de interrumpirlo. El *thread* al utilizar la interfaz "*Runnable*", debe de implementar un método run, donde estableceremos la lógica del hilo en ejecución, es decir, todo lo que realizará el hilo mientras esté iniciado, que en este caso será descargar el contenido del *dataset* y dormir el hilo durante un tiempo determinado.

Teniendo esto claro, haremos los siguiente:

1) Creamos un nuevo Scala Object haciendo clic derecho sobre el *package* > *New* > *Scala Object*:

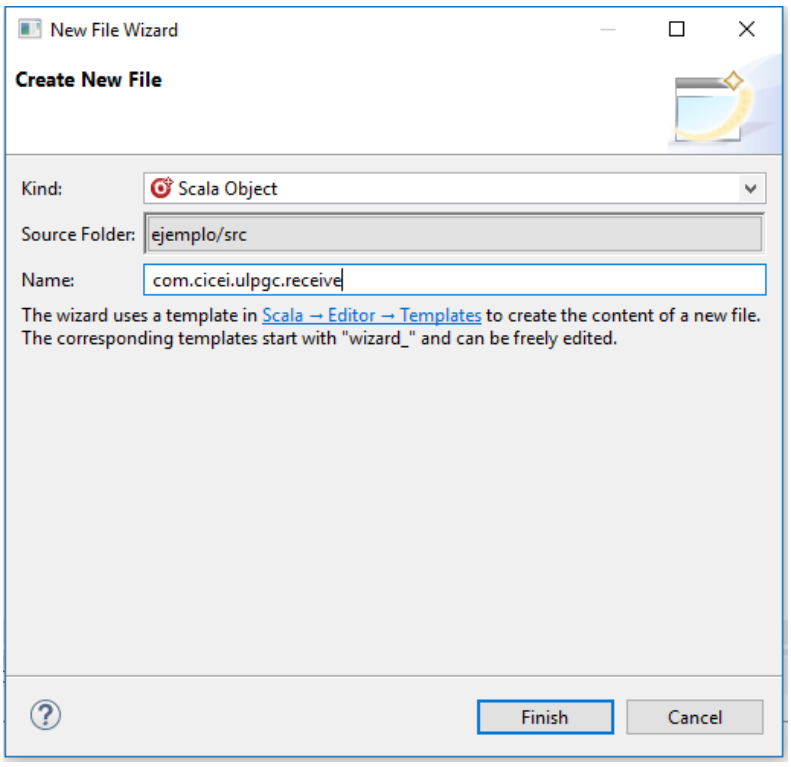


Ilustración 20. Creando el receiver de TFL.

2) Añadimos ahora todas las librerías de Spark que previamente hemos descargado en el ordenador, ya que haremos uso de muchas de ellas. Para ello, sobre el proyecto, hacemos clic derecho y nos dirigimos a *properties* > *Java Build Path* > *Add External JARS*. Se nos abrirá un cuadro de diálogo en el cual tenemos que seleccionar todos los JARS necesarios. Para ello vamos a la ruta donde hemos guardado los archivos Spark y, dentro

de la carpeta *jars*, seleccionamos todos los archivos JARS que nos aparecen tal y como se muestra en la siguiente imagen.

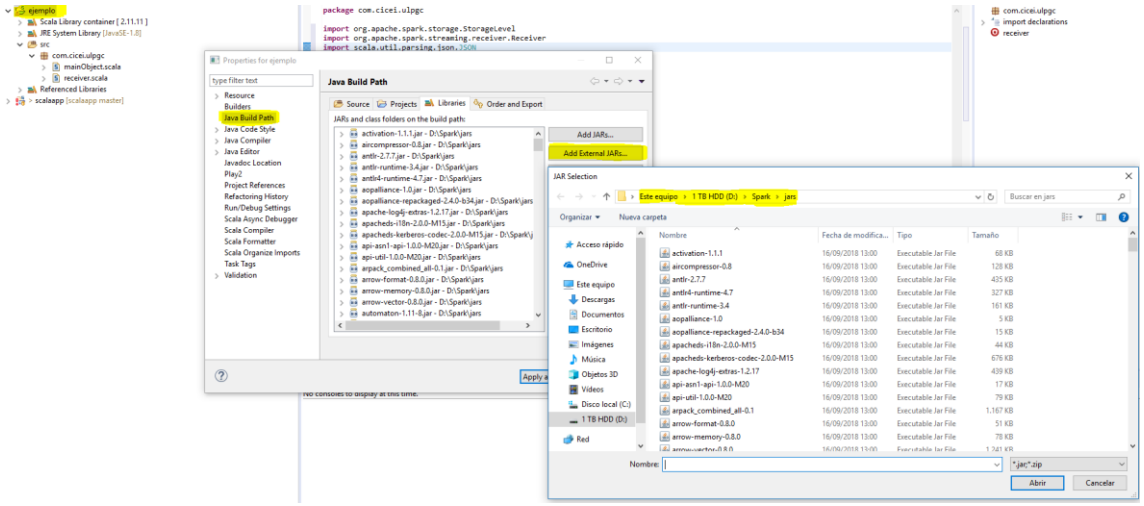


Ilustración 21. Proceso de importar los JARS

3) Ahora ya podemos introducir el código en nuestra aplicación. Quedaría de la siguiente manera:

```
package com.cicei.ulpgc

import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.receiver.Receiver
import scala.util.parsing.json.JSON
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

object receiver {

  private val tflUrl = "https://tfl.gov.uk/tfl/syndication/feeds/tims_feed.xml?app_id=09a40888&app_key=b8e2f385b58c9be640ff95c0c3c930f7"

  class TFLTimsFeed() extends Receiver[String](StorageLevel.MEMORY_ONLY) with Runnable {

    @transient
    private var thread: Thread = _       // Atributo de la clase de tipo thread

    override def onStart(): Unit = {
      thread = new Thread(this)          // Creamos el hilo
      thread.start()                     // Iniciamos el hilo
    }

    override def onStop(): Unit = {
      thread.interrupt()                 // Interrumpimos el hilo
    }

    override def run(): Unit = {
      while (true) {                     // Bucle infinito
        receive();                       // Hacemos la llamada al método donde descargaremos el contenido del dataset
        Thread.sleep(150 * 1000);        // Dormimos el hilo durante 2 minutos y medio
      }
    }
```

Ilustración 22. Creando los métodos de la interfaz Runnable

Como vemos, lo primero que tenemos que hacer es incluir las librerías necesarias para trabajar con los *thread* y las necesarias para realizar la petición *get* a la URL del *dataset* que hemos guardado anteriormente. Lo importante aquí es la variable *tflUrl* que hace referencia a la URL donde está el *dataset* y los tres métodos nombrados anteriormente (*onStart, onStop, run*). Tan

sólo falta el método *receive()* que será el encargado de descargar el contenido del *dataset*. Tiene el siguiente aspecto:

```
    private def receive(): Unit = {

      val httpClient = new DefaultHttpClient();
      val getRequest = new HttpGet(tflUrl);
      getRequest.addHeader("accept", "application/json");

      val response = httpClient.execute(getRequest);
      if (response.getStatusLine().getStatusCode() != 200) {
        throw new RuntimeException("Failed : HTTP error code : "
          + response.getStatusLine().getStatusCode());
      }

      val br = new BufferedReader(
        new InputStreamReader((response.getEntity().getContent())));

      var output = br.readLine();
      while (output != null) {
        println(output)
        output = br.readLine()
      }
    }

  }

}
```

Ilustración 23. Método receive() con la lógica del receptor

Básicamente estamos haciendo un "*get request*" a la dirección donde está ubicado el *dataset*, comprobamos que la respuesta sea un código 200 "*OK*" (en caso contrario, mostraremos un mensaje de error) y leeremos línea por línea todo el contenido de la respuesta (almacenado dentro del *Buffer*), que será el XML al completo.

4) Por último, desde nuestro método main debemos de llamar a este *receiver* para que nos muestre por consola todo el contenido del dataset. Haremos lo siguiente:

```
package com.cicei.ulpgc

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{ Seconds, StreamingContext }
import receiver._

object mainObject {

  def main(args: Array[String]) {

    val ssc = new StreamingContext("local[*]", "exampleXML", Seconds(5))
    val stream = ssc.receiverStream(new TFLTimsFeed())
    stream.print()

    ssc.start()
    ssc.awaitTermination()


  }

}
```

Ilustración 24. Invocando el receiver TFL desde el main

Es importante importar las librerías de Spark Streaming y también el *Scala Object* que creamos previamente con el *receiver (import receiver._)*.

Muy sencillo. Básicamente lo que tenemos que hacer es crear el contexto de ejecución de Spark Streaming:

```
val ssc = new StreamingContext("local[*]", "exampleXML", Seconds(5))
```

En esta línea de código estamos creando el contexto del *stream*, estamos indicando que vamos a ejecutar la aplicación localmente y no en un servidor de terceros, mediante *local[*] (*en caso de ejecutarse en un servidor de terceros o clúster indicaríamos la *IP:PORT* donde se esté escuchando*).* El siguiente parámetro es el nombre del contexto y, por último, la frecuencia de recepción de los lotes, en este caso cada 5 segundos (cada 5 segundos recibimos un lote nuevo de información). Como estamos utilizando un receptor propio, la frecuencia de descarga de la información la controlaremos desde el *receiver* con el *thread.sleep()* como vimos anteriormente*,* y no mediante este parámetro, pero si será muy importante cuando trabajemos con la API de Twitter (usaremos el mismo contexto para ambos *streams*, aunque se podría crear varios contextos con diferentes frecuencias), ya que esta sí usa esa frecuencia, por lo que cada 5 segundos nos traerá un lote nuevo de tweets.

En la siguiente línea estamos indicando qué receiver vamos a utilizar, indicaremos el que acabamos de crear.

```
val stream = ssc.receiverStream(new TFLTimsFeed())
```

Por último, como vemos en la imagen anterior, tan sólo hay que llamar a *print()* para que se muestre todo el *stream*, iniciaremos el *Stream Context* mediante la función *start*() y que este se ejecute hasta que lo finalicemos manualmente (*awaitTermination*()).

Ya podemos ejecutar la aplicación y comprobar que se descarga toda la información del *dataset*.

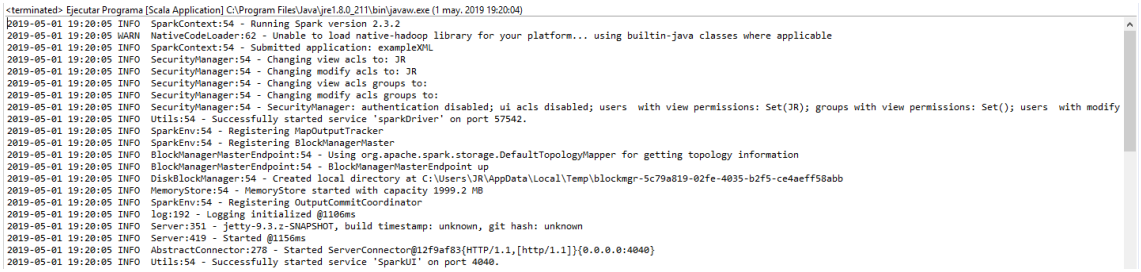


Ilustración 25. Ejecución de Spark Streaming.

```
</Street>
 <Street>
  <name>Tabernacle Street (EC2A)</name>
   <closure>Open</closure>
   <directions>All Directions</directions>
   <Link>
    <toid>4000000030769853</toid>
    <Line>
     <coordinatesEN>532962,182462,533020.34,182513.153</coordinatesEN>
     <coordinatesLL>-.084763,51.525358,-.083903,51.525804</coordinatesLL>
    </Line>
   </Link>
</Street>
 <Street>
  <name>Willow Street (EC2A)</name>
   <closure>Open</closure>
   <directions>All Directions</directions>
   <Link>
    <toid>4000000030147554</toid>
    <Line>
     <coordinatesEN>533152,182428,533019,182452</coordinatesEN>
     <coordinatesLL>-.082038,51.525008,-.083945,51.525255</coordinatesLL>
    </Line>
   </Link>
</Street>
```

Ilustración 26. XML descargado y visto desde la terminal.

Si al seguir estos pasos aparece un error relacionado con los *imports* de las librerías, puede que sea debido a que la versión del compilador de Scala y la versión de Spark no son las mismas o, al menos, estas no son compatibles. Para ello debemos cambiar la versión del compilador de Scala del IDE (para que se adapte a la versión y características del proyecto), para ello hacemos lo siguiente: accedemos mediante clic derecho sobre el proyecto > *Properties* > *Scala Compiler*. Seleccionamos la casilla de "*Use project settings*" e indicamos la versión de Scala siguiente: *Fixed Scala Installation: 2.11.11 (built-in).*

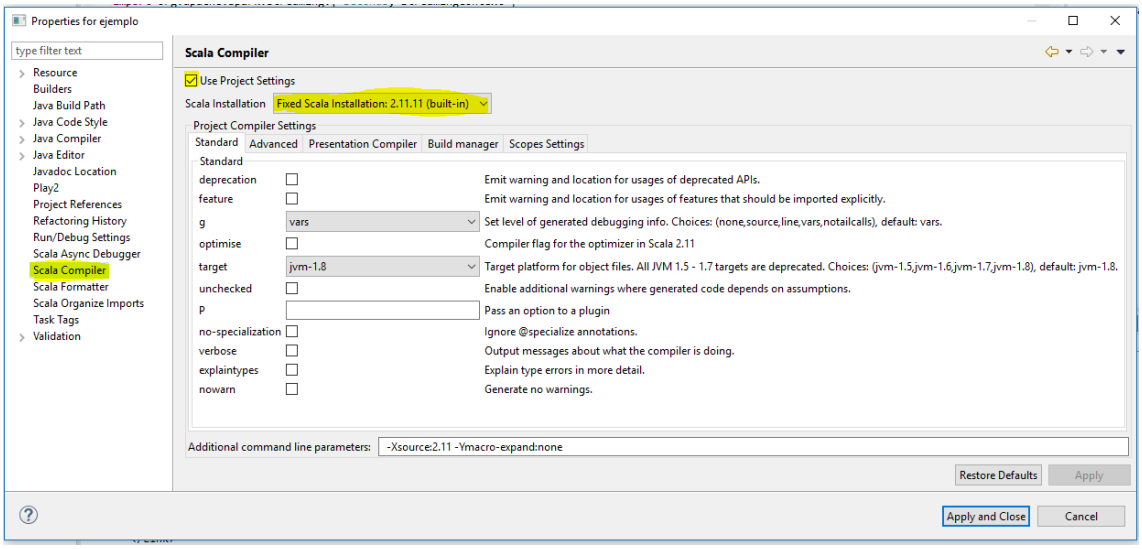


Ilustración 27. Fix. Igualando las versiones de Scala y Spark.

Filtrando los datos

Ahora ya podemos acceder al contenido del *dataset*, pero como podemos observar, al realizar la petición, nos devuelve el contenido de la página entera y con la estructura del XML. De esta manera no nos sirve la información, debemos filtrarla/tratarla para tan sólo quedarnos con los campos que necesitemos. Para ello, haremos uso de la librería Scala XML, que nos ayudará a

tratar la información. Hay una librería muy interesante y útil llamada Spark XML, pero no es compatible con Spark Streaming, tan sólo se usa trabajando con ficheros cargados localmente, por lo que nos decantamos con la otra solución, Scala XML.

Lo primero que tenemos que hacer es descargarnos el JAR de la librería y cargarlo al proyecto de la misma manera que hicimos con los JARS de Spark. Para ello vamos a la página de descarga y procedemos a bajarnos el fichero ZIP con la librería y sus dependencias. Descomprimiremos el fichero y lo guardamos en una carpeta aparte. Esto es muy importante ya que, si subimos el proyecto a GitHub, por ejemplo, debemos de subir todos los JARS que utilicemos por si queremos ejecutar la aplicación en otra máquina. Por lo que es recomendable guardar todos los JARS dentro de una carpeta bajo el propio directorio raíz del proyecto.

Una vez hecho esto, tan sólo tenemos que ir a la aplicación, clic derecho sobre el proyecto > *Properties* > *Java Build Path* > *Add External JARS* > y seleccionamos todos los JARS que acabamos de descargar.

Una vez hecho esto, ya podemos importarlos directamente desde el código para hacer uso de la librería. Lo primero que haremos pues, será importar la librería. También importaremos la herramienta de Java *SimpleDateFormat*, para dar formato a las fechas que aparezcan en las incidencias, de manera que estas sean más visibles y fáciles de reconocer. Crearemos entonces dichos formatos, el de la fecha de entrada (el que recibimos del XML) y el de la fecha de salida:

```
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

import java.text.SimpleDateFormat
import scala.xml._

object receiver {

  private val tflUrl = "https://tfl.gov.uk/tfl/syndication/feeds/tims_feed.xml?app_id=09a40888&app_key=b8e2f385b58c9be640ff95c0c3c930f7"
  val inputFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'Z'")
  val outputFormat = new SimpleDateFormat("dd/MM/yyyy - HH:mm")

  class TFLTimsFeed() extends Receiver[String](StorageLevel.MEMORY_ONLY) with Runnable {

    @transient
    private var thread: Thread = _       // Atributo de la clase de tipo thread
```

Ilustración 28. Creando los formatos de entrada y salida de las fechas.

Scala XML trabaja con un tipo de objeto llamado *Elem*, que hace referencia a una estructura XML, y estos *Elem*, a su vez, están formados por "*nodes*", que son cada elemento o nodo de dicho XML *(<node>Contenido</node>).* Por lo tanto, debemos convertir el contenido que recibimos al realizar el *get*, es decir, todo el XML del *dataset*, a un objeto de tipo Elem. ¿Cómo lo hacemos? Pues muy sencillo. La librería Scala XML ofrece una serie de métodos para convertir ciertos tipos de objetos a objetos de tipo *Elem*, por ejemplo, el más usado, de *String* a *Elem*. Por lo tanto, tan sólo debemos de almacenar en una *string* todo el XML y hacer la llamada al método que efectúa la conversión. De la siguiente manera:

```
getRequest.addHeader("accept", "application/json");

val response = httpClient.execute(getRequest);
if (response.getStatusLine().getStatusCode() != 200) {
  throw new RuntimeException("Failed : HTTP error code : "
    + response.getStatusLine().getStatusCode());
}

val br = new BufferedReader(
  new InputStreamReader((response.getEntity().getContent())));

val str = Stream.continually(br.readLine()).takeWhile(_ != null).mkString("\n")
val xml = scala.xml.XML.loadString(str)

var output = br.readLine();
while (output != null) {
  println(output)
```

Ilustración 29. Creando nuestro objeto XML desde una String

Añadimos esas dos líneas, en la primera de ellas, guardamos en una sola variable de tipo *string* lo que nos devuelve el *get request*, es decir, el contenido del *dataset*. Lo que hace es concatenar cada línea que recibimos del Buffer (siempre que no sea *null*), quitando los saltos de línea.

Y en la siguiente línea convertimos con el método *loadString(),* la variable de tipo String a un objeto de tipo *Elem*, que en este caso se llama *xml*.

Y ya por último, lo que nos falta por hacer es recorrer el XML y recoger/filtrar la información que nos interese mediante una iteración for-yield que nos muestra la propia wiki:

```
    val str = Stream.continually(br.readLine()).takeWhile(_ != null).mkString("\n")
    val xml = scala.xml.XML.loadString(str)

    /*var output = br.readLine();
    while (output != null) {
      println(output)
      output = br.readLine()
    }*/

    for {
      disruption <- xml \\ "Disruption"
      location <- disruption \ "location"
      status <- disruption \ "status"
      severity <- disruption \ "severity"
      category <- disruption \ "category"
      startTime <- disruption \ "startTime"
      endTime <- disruption \ "endTime"
      comments <- disruption \ "comments"
      currentUpdate <- disruption \ "currentUpdate"
      causearea <- disruption \ "CauseArea"
      displaypoint <- causearea \ "DisplayPoint"
      point <- displaypoint \ "Point"
      coordinates <- point \ "coordinatesLL"
    } yield {
      if (status.text == "Active" || status.text == "Active Long Term") {

        val splitcord = coordinates.text.split(",")
        val longitud = "-0" + splitcord(0).substring(1)
        val latitud = splitcord(1)
        val coordenadas = "(" + latitud + "," + longitud + ")"

        println("Incidencia ID: " + disruption \@ "id")
        println("Localización: " + location.text)
        println("Estado: " + status.text)
        println("Importancia: " + severity.text)
        println("Categoría: " + category.text)
        println("T.inicio: " + outputFormat.format(inputFormat.parse(startTime.text)))
        println("T.fin: " + outputFormat.format(inputFormat.parse(endTime.text)))
        println("Comentarios: " + comments.text)
        println("Estado actual: " + currentUpdate.text)
        println("Estado: " + status.text)
        println("Coordenadas: " + coordenadas)
        println("Latitud: " + latitud + " - Longitud: " + longitud)
        println("========================================================================")
      }
    }

  }

}
```

Ilustración 30. Filtrando las incidencias del XML utilizando Scala XML

Borramos el trozo de código comentado ya que no nos hará falta para mostrar el contenido por consola. Lo mostraremos con la sucesión de *println()* dentro de la estructura *yield*.

Mediante la iteración *for-yield*, lo que haremos es iterar por cada nodo del XML (almacenado en la variable *xml*) cuyo *tag* sea "*Disruption" (<Disruption></Disruption>),* es decir, se buscará de manera iterativa cada elemento del XML cuyo tag sea *Disruption* dentro del array de *Disruptions*. Cabe destacar el uso de la herramienta "\\", que quiere decir que la búsqueda se realiza de manera iterativa. Se diferencia de "\" en que esta última tan sólo se queda con el contenido del nodo de la primera coincidencia (cuyo *tag* coincida), no itera con las siguientes.

Pues dentro del *for*, iteraremos por cada incidencia (variable *disruption*), quedándonos con la localización, estado, coordenadas, etc. (*location, status, coordinates, …*) de esta.

En la siguiente imagen podemos ver el contenido del XML (estructura de una incidencia). Lo importante es que cada *Disruption* o incidencia se encuentra dentro de un array de incidencias cuyo *tag* o nombre es *Disruptions*, el cual es el array que recorremos:

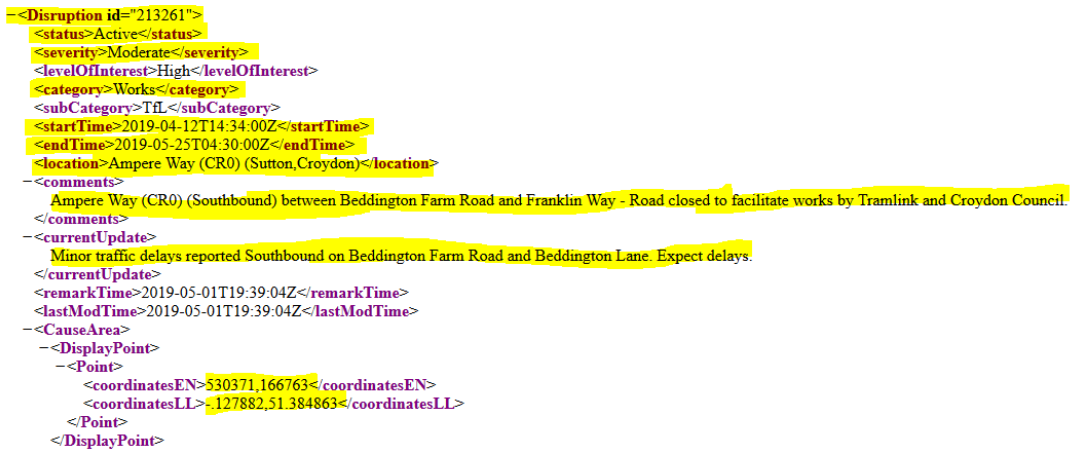


Ilustración 31. Campos importantes de la estructura de las incidencias del XML

Nos quedaremos con los campos marcados en amarillo y descartamos el resto.

Dentro de la estructura *yield*, tan sólo filtramos y nos quedamos con aquellas incidencias cuyo estado sea *Active* o *Active Long Term*, descartando las incidencias programadas (*Scheduled*). Además, le damos forma a las coordenadas para que estas sean más legibles, de la siguiente manera:

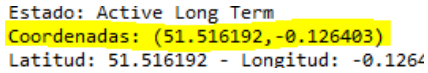


Ilustración 32. Formato legible de las coordenadas

Y, por último, mostramos toda la información de cada incidencia por la consola. Importante fijarse en el uso que se le da al método *format(),* que formatea las fechas siguiendo los patrones establecidos inicialmente en este apartado, haciendo uso de la librería de Java *SimpleDateFormat.* El resultado final al ejecutar el programa será el siguiente:

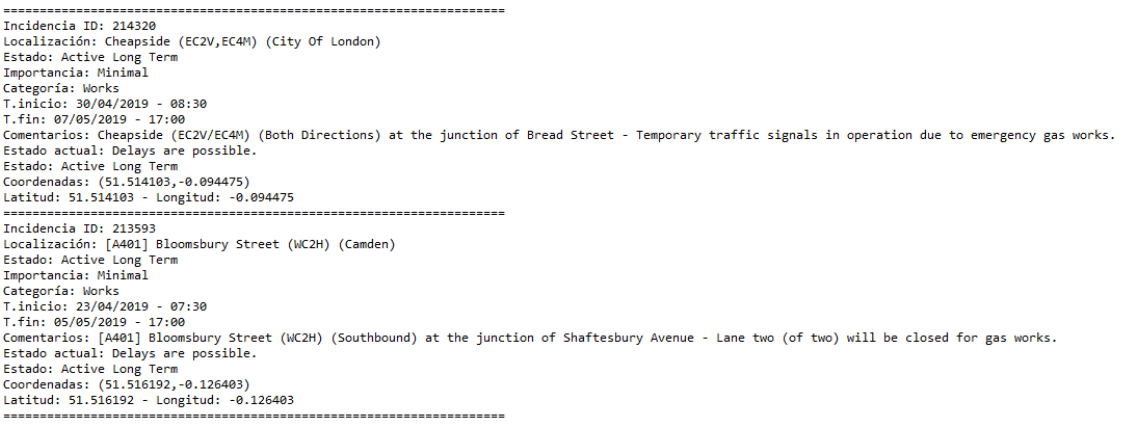


Ilustración 33. Incidencias filtradas

Tan sólo nos queda preparar el objeto JSON para mandarlo posteriormente a la aplicación web para terminar con la parte de TFL. Esto lo veremos en el siguiente apartado.

Preparando los datos para el envío (JSON)

Para crear objetos JSON, utilizaremos la librería Scala JSON perteneciente a Play Framework. Lo primero que tenemos que hacer es descargar los JARS siguiendo los mismos pasos que el apartado anterior.

Una vez descargado, descomprimido y guardado todos los JARS en la carpeta destinada a guardar todo los JARS dentro del proyecto, lo añadimos a este de la misma manera que en las otras ocasiones. Clic derecho sobre el proyecto *> Properties > Java Build Path > Add External JARS* y seleccionamos el JAR.

Lo siguiente es importarlo desde el código:

```
import java.text.SimpleDateFormat
import scala.xml._

import play.api.libs.json._

object receiver {

  private val tflUrl = "https://tfl.gov.
  val inputFormat = new SimpleDateFormat
```

Ilustración 34. Importando Scala JSON.

La idea es muy simple, es crear un *array JSON* de objetos, donde cada objeto JSON dentro del *array* es una incidencia. De esta manera:


```
[ {
  "location" : "A12 (RM1,RM5,RM7) (Havering)",
  "status" : "Active",
  "severity" : "Serious",
  "category" : "Works",
  "startingtime" : "2019-03-20T16:13:00Z",
  "endtime" : "2019-03-20T19:00:00Z",
  "comments" : "A12 (RM7/RM5/RM1) (Eastbound) at the junction of North Street - Lane one (of two) closed during emergency sewer cover works.",
  "currentupdate" : "Traffic remains slow on approach, with tailbacks beyond Whalebone Lane. Traffic is also slow on diversion via Whalebone Lane North, northbound through Collier Row and southbound through Chadwell Heath.
      North Street and Mawney Road are also congested, with tailbacks onto the Romford Ring Road. Expect delays.",
  "lat" : "51.586044",
  "lon" : "-0.172331"
}, {
  "location" : "[B504] Judd Street (NW1,WC1H) (Camden)",
  "status" : "Active",
  "severity" : "Moderate",
  "category" : "Works",
  "startingtime" : "2019-01-28T08:00:00Z",
  "endtime" : "2019-03-31T17:00:00Z",
  "comments" : "[B504] Judd Street (WC1) (Both Directions) at the junction of [A501] Euston Road - The road is closed for junction improvement work and the installation of a cycle route. Additionally, [A501] Euston Road
      (Westbound), lane one (bus lane) closed, [A5202] Midland Road (Southbound), lane three closed. From approximately 07:00 Wednesday 20 March, [A501] Euston Road, lane restrictions in both directions. ",
  "currentupdate" : "Euston Road westbound is slow moving through the restrictions. Expect delays.",
  "lat" : "51.528833",
  "lon" : "-0.12596"
}, {
  "location" : "[A308] Hampton Court Road (TW12) (Richmond Upon Thames)",
  "status" : "Active",
  "severity" : "Moderate",
  "category" : "Works",
  "startingtime" : "2019-03-20T08:59:00Z",
  "endtime" : "2019-03-24T16:00:00Z",
  "comments" : "[A308] Hampton Court Road (TW12) (Both Directions) at the junction of Hogarth Way - Two-way temporary traffic signals are in operation due to roadworks.",
  "currentupdate" : "Traffic is reportedly slow westbound on approach, with tailbacks to Hampton Court Way and eastbound on approach, with tailbacks to Percy Road. Expect delays.",
  "lat" : "51.412289",
  "lon" : "-0.358249"
}, {
  "location" : "[A3] Kingston By-Pass (KT9,KT6,KT5) (Kingston Upon Thames)",
```


Ilustración 35. Cuerpo/Contenido del objeto JSON. Imagen sacada de Postman en las pruebas.

Para ello tan sólo tenemos que hacer lo siguiente. Lo primero es crear la estructura del array, siendo este un array vació inicialmente. Como cuando lo creábamos en Java (*String [] array*). Esto en Scala se hace de la siguiente manera (*Json.arr*()):

```
val str = Stream.continually(br.readLine()).takeWhile(_ != null).mkString("\n")
val xml = scala.xml.XML.loadString(str)

var arrjson: JsArray = Json.arr()

for {
  disruption <- xml \\ "Disruption"
  location <- disruption \ "location"
  status <- disruption \ "status"
  severity <- disruption \ "severity"
  category <- disruption \ "category"
  startTime <- disruption \ "startTime"
  endTime <- disruption \ "endTime"
  comments <- disruption \ "comments"
  currentUpdate <- disruption \ "currentUpdate"
  causearea <- disruption \ "CauseArea"
```

Ilustración 36. Creando el array JSON.

Antes de entrar en la estructura *for-yield* creamos el array JSON inicialmente vacío. La idea es que cada vez que una incidencia cumpla nuestro criterio de búsqueda creamos un objeto JSON con la información de la incidencia para añadirlo al array. Procedemos de la siguiente manera:

```
} yield {
  if (status.text == "Active" || status.text == "Active Long Term") {

    val splitcord = coordinates.text.split(",")
    val longitud = "-0" + splitcord(0).substring(1)
    val latitud = splitcord(1)
    val coordenadas = "(" + latitud + "," + longitud + ")"


    val objeto: JsValue = Json.obj("location" -> location.text, "status" -> status.text, "severity" -> severity.text,
      "category" -> category.text, "startingtime" -> startTime.text,
      "endtime" -> endTime.text, "comments" -> comments.text,
      "currentupdate" -> currentUpdate.text, "lat" -> latitud, "lon" -> longitud)

    arrjson = arrjson :+ (objeto)


    println("Incidencia ID: " + disruption \@ "id")
    println("Localización: " + location.text)
    println("Estado: " + status.text)
    println("Importancia: " + severity.text)
    println("Categoría: " + category.text)
```

Ilustración 37. Creando el objeto JSON y añadiéndolo al array JSON

Dentro del bloque *yield*, si se cumple nuestra condición de que la incidencia esté activa, creamos un objeto JSON tal y como se muestra en la imagen anterior, utilizando el método *Json.obj()* estableciendo entre comillas la *key* o tag junto con el contenido de este.

Para añadirlo al array tan sólo debemos de usar la herramienta ":+", funciona exactamente igual al concatenado de Java.

Una vez se termine de recorrer todas las incidencias, el array se habrá llenado con todas las incidencias que cumplan la condición de filtro. Ahora tan sólo queda preparar el *post request* fuera del bloque *for-yield,* poniendo en el *body* precisamente el array que acabamos de llenar, para mandarlo en un futuro a la aplicación web.

Primero, para poder realizar una petición post debemos de importar dos nuevas librerías, estas son las siguientes:

```
import java.text.SimpleDateFormat
import scala.xml._

import play.api.libs.json._

import org.apache.http.client.methods.HttpPost
import org.apache.http.entity.StringEntity

object receiver {

  private val tflUrl = "https://tfl.gov.uk/tfl/syndication/fee
  val inputFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:s
  val outputFormat = new SimpleDateFormat("dd/MM/yyyy - HH:mm"

  class TFLTimsFeed() extends Receiver[String](StorageLevel.MEM
```

Ilustración 38. Importando las librerías HTTP POST de Java

Una vez importadas (no es necesario descargar en este caso nuevos JARS), tan sólo tenemos que poner el siguiente código al acabarse el bloque *for-yield*:

```
      println("T.fin: " + outputFormat.format(inputFormat.parse(endTime.text)))
      println("Comentarios: " + comments.text)
      println("Estado actual: " + currentUpdate.text)
      println("Estado: " + status.text)
      println("Coordenadas: " + coordenadas)
      println("Latitud: " + latitud + " - Longitud: " + longitud)
      println("=================================================================")
    }
  }

  val readableString: String = Json.prettyPrint(arrjson)
  println(readableString)
  val post = new HttpPost("http://localhost:8080/") //Rest API SpringBoot
  post.setHeader("Content-type", "application/json")
  post.setEntity(new StringEntity(readableString))
  val responses = (new DefaultHttpClient).execute(post)
  println("--- HEADERS ---")
  response.getAllHeaders.foreach(arg => println(arg))

}
```

Ilustración 39. Estructura para las peticiones POST

En este bloque realizaremos la petición *post* al servidor REST cuando lo tengamos hecho más adelante, para ello debemos de poner el array en el cuerpo de la petición. Como actualmente el array es de tipo *JsArray*, un tipo de objeto de la librería Scala JSON, no nos vale para realizar la petición, debemos pasarlo a *String*. Eso lo hacemos mediante el método *prettyPrint()* de la librería Scala JSON. Por último, mediante el método *setEntity(),* añadimos el array con las incidencias al *body* de la petición y ejecutamos la petición con *execute().* Es recomendable mostrar las cabeceras de la respuesta, para comprobar el código que nos devuelve la petición, por si hubo algún problema en el envío y así poder detectarlo (código diferente al 200).

Si actualmente ejecutamos la aplicación, funcionaría todo a excepción de la petición *post*, ya que nos devolvería un error de que actualmente no hay una aplicación escuchando peticiones en la URL propuesta. Cuando hagamos la aplicación *Rest* en la segunda parte de la memoria ya se podrá enviar la petición sin problema alguno, hasta entonces podemos dejarlo así o comentarlo para que no salte errores cuando nos pongamos con la API de Twitter.

Un ejemplo gráfico con Postman de una petición *post* sería el siguiente:

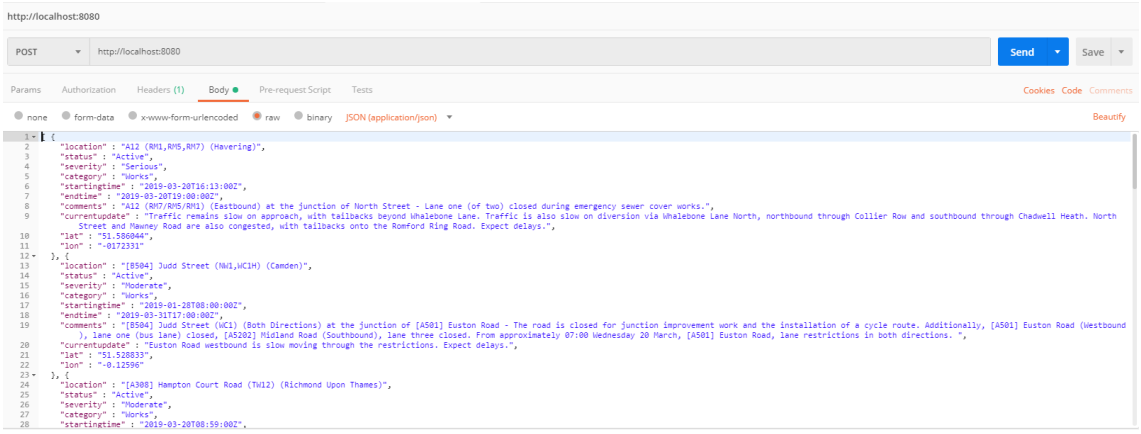


Ilustración 40. Petición POST utilizando Postman

Como añadido podemos hacer una última mejora, podemos permitir guardar el stream de incidencias de una manera sencilla en un fichero de texto por si se requiere. Tan sólo tenemos que ir a nuestro *main* donde llamamos a nuestro receiver y añadir el siguiente código.

```
    val stream = ssc.receiverStream(new TFLTimsFeed())
    stream.print()

    if (args.length > 2) {
        stream.saveAsTextFiles(args(2))
    }

    ssc.start()
    ssc.awaitTermination()

}
```

Ilustración 41. Guardando el stream en un fichero de texto

Si se ejecuta la aplicación y se pasa por parámetro el nombre de un fichero, se guardará el *stream* completo dentro de un fichero con ese nombre, de tal manera que se podrá consultar en caso de que se requiera de manera local (sirve como posible copia de seguridad).

4.3.5. Trabajando con la API de Twitter

Invocando el stream de Tweets

Una vez tengamos la parte de la aplicación relacionada con las incidencias hecha, ya podemos ponernos manos a la obra con la API de Twitter. La idea de esta sección es usar Twitter para buscar posibles opiniones o simplemente tweets relacionados con incidencias de tráfico, ya sea las que aparezcan en el *dataset* o nuevas incidencias que todavía no se han reportado.

Lo primero que tenemos que hacer para acceder a la API de Twitter es hacernos una cuenta de desarrollador en "*Twitter Developer Platform*", la plataforma de desarrolladores de Twitter. Para ello, accedemos a la página y realizamos el formulario de registro, en la que nos pedirá información sobre la cuenta de Twitter que vamos a crear. Podemos saltarnos los pasos que no sean necesarios.

Una vez creada la cuenta e iniciado sesión, podemos ir a la pestaña de Apps y apretar en "*Create an app*" para acceder a los distintos formularios. Esta tarea es un poco engorrosa, ya que nos pedirá, entre otras cosas, explicar en determinadas palabras en qué consistirá nuestra aplicación, a quién va dirigida y con qué propósitos, etc.

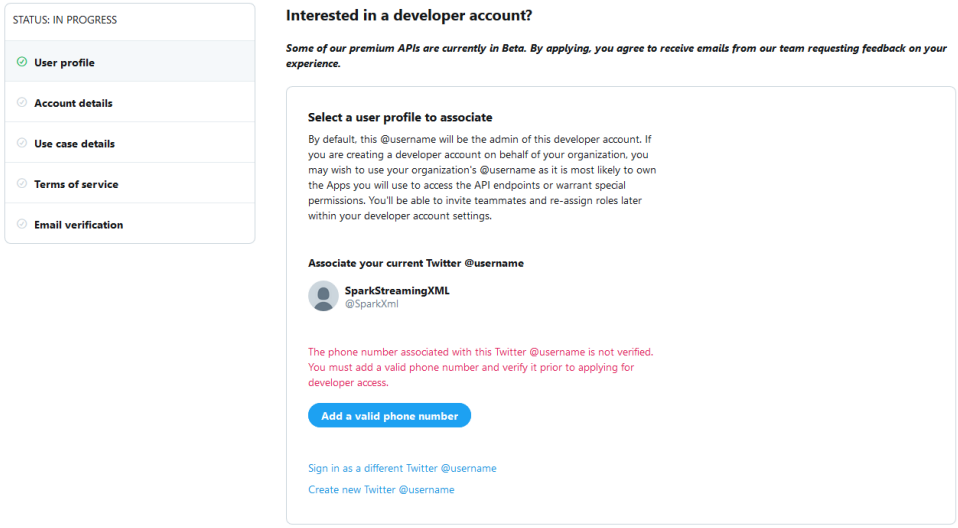


Ilustración 42. Formulario de registro en Twitter Dev.

Una vez hayamos terminado de rellenar todos los campos necesarios y ya tengamos nuestra aplicación creada, podemos ir al panel de Apps, en donde aparecerá la *app* recién creada:

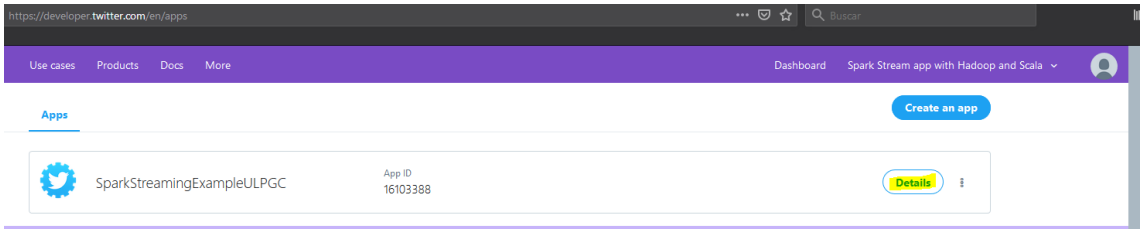


Ilustración 43. Aplicación creada en Twitter Dev.

Ahora viene la parte interesante. Para poder iniciar sesión e identificarnos desde nuestra aplicación Scala, necesitaremos una serie de *Keys* y de *Tokens* de acceso que identifican a cada aplicación. Para acceder a ellas, tan sólo tenemos que apretar en el botón *Details* como se muestra en la imagen anterior. Una vez dentro, accedemos a la pestaña *Keys and Tokens* y podemos ver los cuatro códigos que necesitaremos luego. Dejaremos esta pestaña abierta para tenerlo a mano para el siguiente paso.

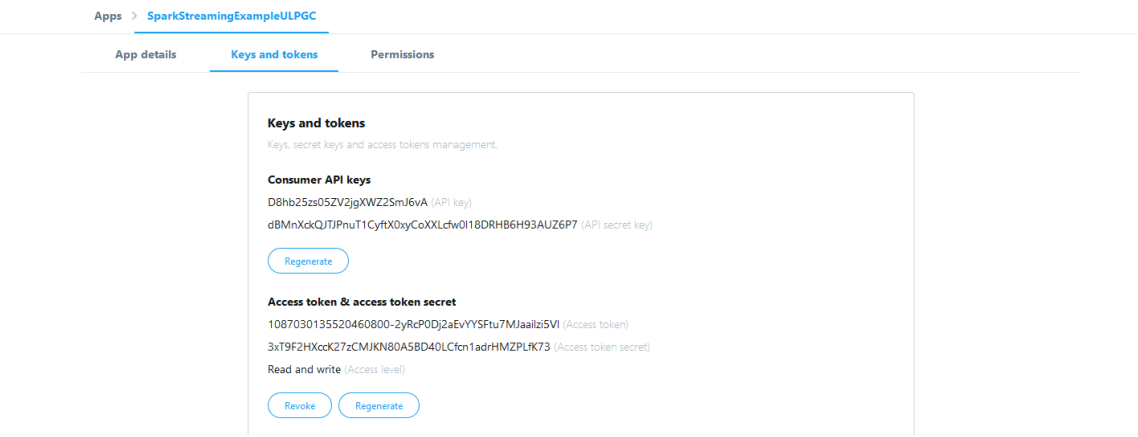


Ilustración 44. Credenciales de nuestra aplicación

Ahora debemos de ir a la raíz de nuestra aplicación Scala (o desde el propio IDE). Procedemos a crear un fichero TXT (lo llamaremos *twitter.txt*) con el siguiente contenido:

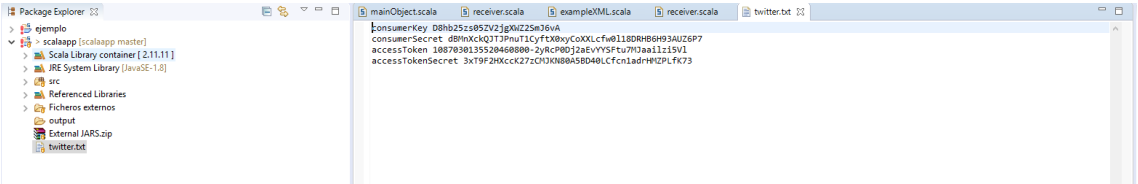


Ilustración 45. Fichero TXT con las credenciales de nuestra aplicación para autenticarnos en Twitter

Como pueden ver, tan sólo tenemos que poner los tokens de acceso y las *keys* que buscamos en el paso anterior dentro del fichero, al lado de su correspondiente nombre. Este fichero se utilizará para iniciar sesión dentro del propio código más adelante.

Lo siguiente que tenemos que hacer para estar listo y empezar a escribir código es descargarnos las librerías de Twitter para Spark y Scala, como hemos hecho en pasos anteriores. Para ello, accederemos a la página de descarga para proceder a bajarnos el fichero. Posteriormente lo descomprimiremos dentro de la carpeta destinada a almacenar los *jars* dentro del proyecto. Luego, dentro del IDE, hacemos lo de siempre, clic derecho sobre el proyecto *> Java Build Path > Add External Jars* y seleccionamos los JARS recién descargados. Una vez hecho esto y guardados los cambios podemos pasar directamente a escribir el código.

Lo primero que tenemos que hacer es importar las librerías que usaremos. Como ya se ha comentado anteriormente, vamos a usar el mismo *StreamingContext* que usamos con el otro

receiver (aunque se podría crear varios simultáneamente), ya que como hemos comentado, la frecuencia de los lotes con el receiver para TFL no se controla desde el *StreamingContext*, si no desde el código del propio receiver con la operación *thread.sleep(),* es decir, no se aprovecha realmente el parámetro de la frecuencia dentro de la llamada al método. Entonces, para darle un uso al parámetro de la frecuencia, usaremos el mismo contexto para ambos receivers (escribiremos la lógica relacionada con Twitter API dentro del *main*).

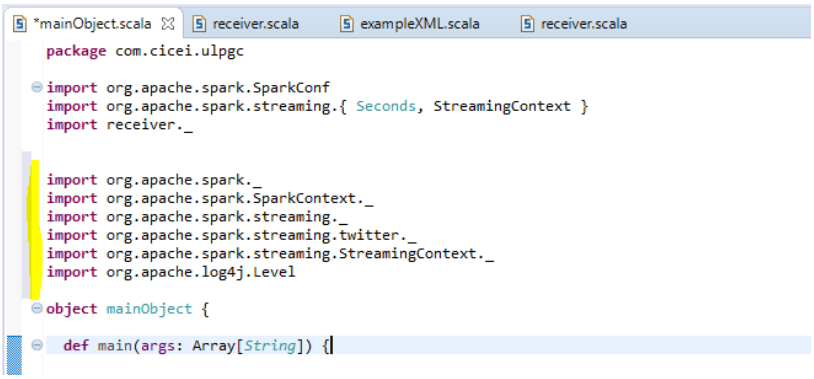


Ilustración 46. Importando las librerías para Twitter API

Una vez tengamos las librerías importadas, el siguiente paso antes de empezar a recibir los tweets sería iniciar sesión en Twitter utilizando las credenciales que hemos preparado dentro del fichero *twitter.txt* situado en la raíz de nuestro proyecto. Este código de inicio de sesión se puede poner dentro del mismo *main,* aunque también podríamos crear unos métodos auxiliares dentro del propio fichero para luego hacer la llamada y así mejorar la comprensión del código. Yo me decantaré por esta última opción, pero se puede mejorar aún más, crearemos un nuevo *Scala Object* llamado *Utilities*, donde pondremos todos los métodos auxiliares que necesitemos y que podremos importarlo desde cualquier otro fichero, pudiendo reutilizar código. Para ello, clic derecho dentro del paquete de nuestro proyecto, donde tenemos todos los ficheros, *New > Scala Object*:

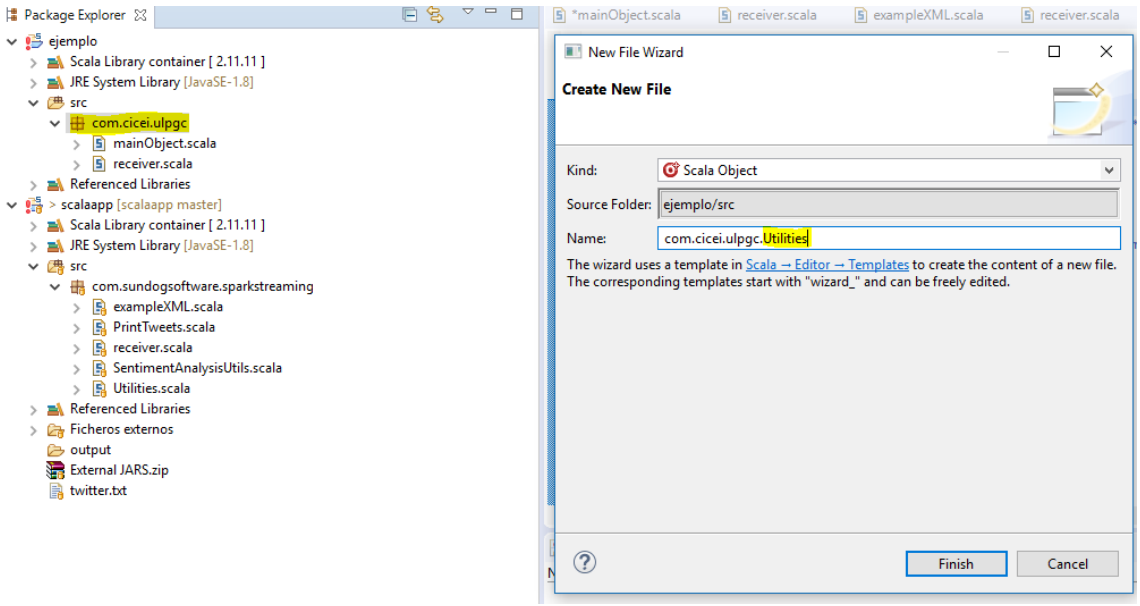


Ilustración 47. Creando la clase Utilities para los métodos auxiliares

Dentro del Scala Object recién creado, debemos de importar dos librerías que usaremos en el código, la primera de ella asociada a los Logs de la aplicación y la segunda para poder leer ficheros localmente (*twitter.txt* previamente creado con las credenciales):

```
package com.cicei.ulpgc

import org.apache.log4j.{Level, Logger}
import scala.io.Source

object Utilities {
```

Ilustración 48. Importando las librerías necesarias en Utilities

Ahora ya podemos crear los dos métodos para identificarnos dentro de Twitter a través de código:

```
package com.cicei.ulpgc

import org.apache.log4j.{Level, Logger}
import scala.io.Source

object Utilities {

  def setupLogging() = {
    val rootLogger = Logger.getRootLogger()
    rootLogger.setLevel(Level.ERROR)
  }


  def setupTwitter() = {
    for (line <- Source.fromFile("twitter.txt").getLines) {
      val fields = line.split(" ")
      if (fields.length == 2) {
        System.setProperty("twitter4j.oauth." + fields(0), fields(1))
      }
    }
  }

}
```

Ilustración 49. Métodos setupLogging() y setupTwitter() de Utilities

El primero de los métodos creados es prescindible, pero se recomienda como buena práctica crearlo, ya que, nos avisará de posibles errores relacionados con Twitter, y, si establecemos que tan sólo se guarde mensajes de ERROR (*Level.ERROR*) evitaremos el spam de tal manera que sólo se almacene y muestre información importante, por ejemplo, fallos de inicio de sesión.

El segundo método es el realmente importante, gracias a la librería que importamos en el inicio (*scala.io.Souce*), somos capaces de cargar la información almacenada dentro de un fichero local (*twitter.txt*). Dentro de este fichero establecimos las credenciales, lo que haremos es dividirlos por columnas (separados por espacio), y con el método *System.setProperty*, indicando como primer parámetro la propiedad a la que se hace referencia (twitter4j.oauth, la autenticación de Twitter, concatenado con el nombre de cada *key*) y como segundo parámetro el valor de la propiedad en sí, que son la primera y segunda columna respectivamente.

Una vez creado el fichero lo podemos guardar e importarlo desde nuestro *main (import Utilities._)*:

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.twitter._
import org.apache.spark.streaming.StreamingContext._
import org.apache.log4j.Level

import Utilities._


object mainObject {

  def main(args: Array[String]) {
```

Ilustración 50. Importando la clase Utilities desde el main

Ahora si podemos iniciar sesión y empezar a traernos los lotes de tweets y filtrarlos. Como ya hemos importado el fichero que contiene ambos métodos, tan sólo debemos de hacer la llamada:

```
object mainObject {

  def main(args: Array[String]) {

    setupTwitter()
    setupLogging()

    val ssc = new StreamingContext("local[*]", "exampleXML", Seconds(5))

    val stream = ssc.receiverStream(new TFLTimsFeed())
    stream.print()

    ssc.start()
    ssc.awaitTermination()


  }
```

Ilustración 51. Llamada a los métodos setupTwitter() y setupLogging() desde el main

Como estamos guiando/influenciando la búsqueda de tweets a mensajes relacionados con incidencias/accidentes de tráfico, necesitamos establecer un array con diferentes palabras claves o *keywords* para que el motor de búsqueda tan sólo nos devuelva tweets que contengan, al menos, una palabra de ese array. Para ello, lo hacemos de la siguiente manera (es importante poner las palabras en inglés ya que nos dirigimos a los ciudadanos de Londres):

```
setupTwitter()
setupLogging()

val keywords: Seq[String] = Seq("tfl", "TfLTrafficNews","@TfLTrafficNews","disruption","traffic", "road is closed",
    "accident", "incident","collision","roadworks","breakdown","update","cleared","traffic light","emergency repairs")


val ssc = new StreamingContext("local[*]", "exampleXML", Seconds(5))
```

Ilustración 52. Palabras claves para orientar la búsqueda

En ese array de palabras (*Seq[String]*) he puesto algunos términos que pueden guiar la búsqueda de determinados tweets. Entre más palabras pongamos, el tamaño de cada lote será mayor, pero hay que tener en cuenta que la Api de Twitter establece un tamaño máximo de palabras dentro del array/secuencia.

El siguiente paso es realmente importante, consistente en hacer la llamada al receiver encargado de recoger los Tweets. A diferencia del receiver que creamos nosotros mismos para leer las incidencias del dataset, la Api de Twitter ya nos ofrece un método/*receiver* que se encarga de crear el stream de tweets. Tan sólo debemos invocarlo de la siguiente manera:

```
setupTwitter()
setupLogging()

val keywords: Seq[String] = Seq("tfl", "TfLTrafficNews","@TfLTrafficNews","disruption","traffic", "road is closed",
    "accident", "incident","collision","roadworks","breakdown","update","cleared","traffic light","emergency repairs")


val ssc = new StreamingContext("local[*]", "exampleXML", Seconds(5))

val tweets = TwitterUtils.createStream(ssc, None, keywords)

val stream = ssc.receiverStream(new TFLTimsFeed())
stream.print()
```

Ilustración 53. Llamada al receiver de Twitter para crear el stream

Mediante la llamada a *TwitterUtils.createStream*, creamos el stream de tweets (los lotes de tweets), al cual le pasamos tres parámetros: el primero de ellos es el contexto de Spark Streaming que utilizamos, como ya he dicho anteriormente, usaremos el mismo para los dos receivers, aunque se podría crear dos independientes. El segundo parámetro acepta un objeto de tipo *twitter4j.auth.Authorization,* el cual sirve para autenticarse en Twitter. Nosotros, al iniciar sesión previamente haciendo uso de las propiedades del sistema, podemos dejar vacío este campo con *None*. Por último, podemos poner la lista de palabras claves para guiar e influir la búsqueda de los lotes de tweets, en nuestro caso, ponemos la creada anteriormente.

Filtrando los tweets (sentiment analysis + geolocation) + envío (JSON)

Una vez hecho esto, ya podemos ver por consola diferentes lotes de tweets, pero nuestro único filtro de búsqueda hasta ahora es el array con *keywords* que usamos para guiar la búsqueda. Debemos de poner una serie de filtros extras para intentar pulir más la búsqueda, para ellos crearemos los siguientes filtros:

- Sentiment Analysis.
- Geolocalización de los Tweets.

Sentiment Analysis o análisis de sentimientos de tweets es una librería de The Stanford NLP Group creada para hacer un análisis de sentimientos de tweets dependiendo del contexto en el que estos se encuentran (se suele usar, por ejemplo, para que las empresas vean cómo reacciona el público a sus productos/servicios e incluso, para estudios de intención de voto de cada partido político en época electoral).

El otro filtro que podemos usar es la geolocalización de los Tweets, para tan sólo quedarnos con los tweets mandados desde Londres (incluso podríamos representarlos en un mapa). Esto es posible ya que la Api de Twitter nos ofrece una serie de métodos que podríamos usar para conseguir dicha información:

- *tweet.getGeoLocation.getLatitude()*
- *tweet.getGeoLocation.getLongitude()*
- *tweet.getPlace()*

Pero hay un problema con el uso de estos métodos. Estos métodos utilizan la información del GPS de cada usuario que manda el Tweet. Normalmente los usuarios casuales de Twitter, o bien, no tienen el GPS activo cuando twittean, o si estos están activos, el usuario no ha dado permiso para que el GPS acceda a su ubicación (puede ser también por fallos en la Api, según se comenta en algunos foros), de tal manera que, si accedemos a la información almacenada con estos métodos, hay un 95% de posibilidades de que esta información sea nula (aunque las grandes empresas como Twitter, Facebook o Google saben seguramente en todo momento esta información, pero no es accesible por nosotros), por lo que no se puede hacer un correcto filtrado de la información.

Buscando por diferentes foros sobre esta problemática llegamos a la conclusión de que una posible solución sería buscar, en lugar de la geolocalización de cada tweet, en la localización del perfil del usuario, es decir, cuando un usuario se crea una cuenta, puede elegir una localización de donde vive, para que el público sepa de donde es cada persona. Esta información si es más común de encontrar, aunque la ubicación sea mucho menos precisa que la idea inicial. En esta aplicación implementaremos esta solución.

Pues una vez indicados los diferentes filtros a utilizar, primero debemos de hacer unos pasos previos a ponernos de lleno con el código. Primero debemos preparar el análisis de sentimientos de los tweets. Lo primero que tenemos que hacer es descargarnos los JARS necesarios para poder utilizar la librería, para ello vamos a la página oficial de la universidad y procedemos a la descarga del fichero. Luego lo descomprimimos dentro de la carpeta donde guardamos todos los *jars* bajo la raíz de nuestro proyecto para tenerlos ubicados.

Una vez tengamos descomprimidos los ficheros, hacemos lo de siempre, clic derecho sobre el proyecto *> Properties > Java Build Path > Add External JARs* y seleccionamos todos los *jars* recién descargados.

Si tenemos los *jars* importados en nuestro proyecto, podemos seguir dos caminos: crearnos nuestro propio fichero donde verteremos nuestra propia lógica de análisis de sentimientos para cada tweet utilizando las librerías recién importadas, donde comprobaremos el cuerpo del tweet y devolveremos una valoración siguiendo nuestros propios criterios, o, en caso contrario, utilizar un modelo con una lógica y criterios ya definidos que nos aporta la propia organización. Por facilidad de uso haremos uso de un modelo ya definido, el cual siempre podemos modificar el código si queremos añadir más criterios. Para ello accedemos al GitHub donde se encuentra el Scala Object con la lógica, lo descargamos y lo añadimos a nuestro proyecto de la siguiente manera: clic derecho sobre nuestro paquete *> Import… > File System >* y seleccionamos la ubicación del fichero "*.scala*". Terminamos pulsando *Finish*.

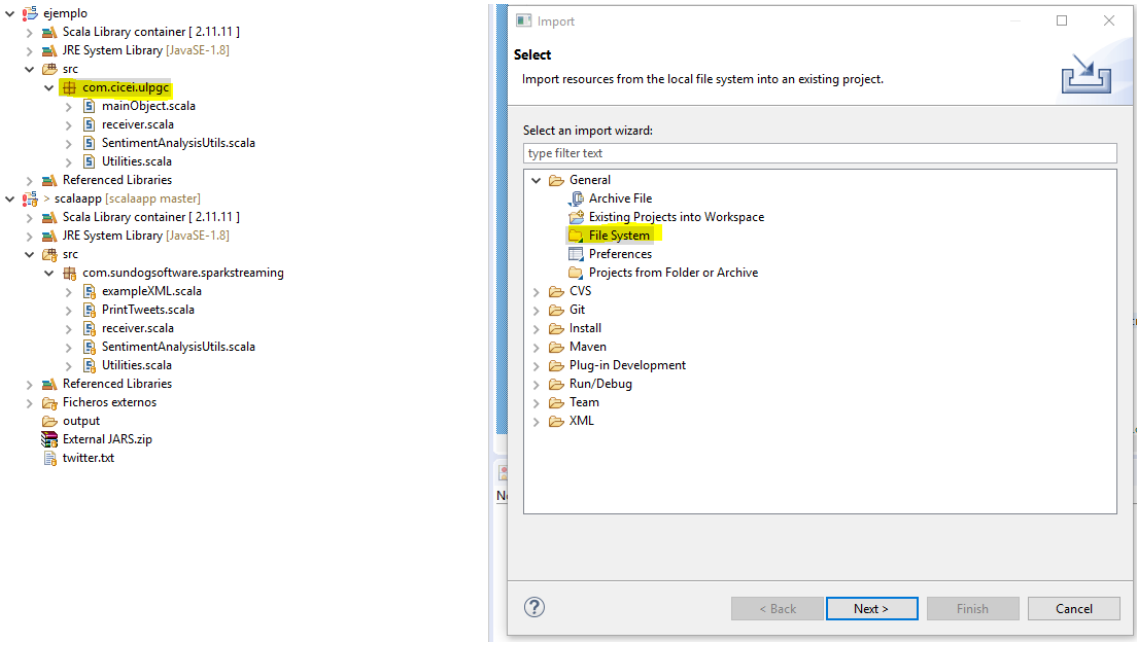


Ilustración 54. Importando el fichero para el análisis de sentimientos

Una vez ya tengamos importado el fichero en nuestro proyecto ya somos capaces de acceder al método que realiza el análisis de sentimientos, no necesitamos importar este fichero dentro del fichero del *main* ya que haremos la llamada al método mediante la ruta completa: *SentimentAnalysisUtils.detectSentiment()* (importante estar bajo el mismo paquete, tanto el *main*, como el fichero con la lógica del análisis de sentimientos)

Ya tenemos listo todo lo relacionado al análisis de sentimientos, antes de pasar al código debemos de importar dos librerías más que ya hemos usado (no hay que descargar más JARs), las cuales son las de Scala JSON y las de las peticiones HTTP POST:

```
import org.apache.spark.streaming.StreamingContext._
import org.apache.log4j.Level

import Utilities._

import play.api.libs.json._

import org.apache.http.client.methods.HttpPost
import org.apache.http.entity.StringEntity
import org.apache.http.impl.client.DefaultHttpClient;

object mainObject {

  def main(args: Array[String]) {

    setupTwitter()
```

Ilustración 55. Importando Scala JSON y HTTP POST

Una vez importadas las librerías, podemos adentrarnos al código, añadimos el bloque siguiente:

```
val tweets = TwitterUtils.createStream(ssc, None, keywords)

var totalTweets: Long = 0
var contadortweets: Long = 0

tweets.foreachRDD((rdd, time) => {
  // Obviamos lotes vacios
  if (rdd.count() > 0) {
    // Combinamos cada partición en un único RDD
    val repartitionedRDD = rdd.repartition(1).cache()
    val tweets = repartitionedRDD.collect()
    for (tweet <- tweets) {
      contadortweets += 1
      val sentiment = SentimentAnalysisUtils.detectSentiment(tweet.getText).toString
      val location = tweet.getUser.getLocation
      val userName = tweet.getUser.getName
      val userAcc = tweet.getUser.getScreenName
      val imgurl = tweet.getUser.getProfileImageURL
      val timestamp = tweet.getCreatedAt.toString()
      //val coordinates = "("+tweet.getGeoLocation.getLatitude+","+tweet.getGeoLocation.getLongitude+")"
      if (location != null) {
        if (location.contains("London") && (sentiment == "NEUTRAL" || sentiment == "POSITIVE" || sentiment == "VERY_POSITIVE")) {
          println("Sentimiento: " + sentiment + " Tweet #" + contadortweets + ": " + tweet.getText + " Localización: " + location)

          val objeto: JsValue = Json.obj("number" -> contadortweets, "text" -> tweet.getText, "userName" -> userName,
            "userAcc" -> userAcc, "imgUrl" -> imgurl,
            "location" -> location, "timestamp" -> timestamp)

          val readableString: String = Json.prettyPrint(objeto)
          println(readableString)

          val post = new HttpPost("http://localhost:8080/tweets")
          post.setHeader("Content-type", "application/json")
          post.setEntity(new StringEntity(readableString))
          val responses = (new DefaultHttpClient).execute(post)
          println("--- HEADERS TWEET POST ---")
          responses.getAllHeaders.foreach(arg => println(arg))
        }
      }
    }
    totalTweets += repartitionedRDD.count()
    println("Tweet count: " + totalTweets)
  }
})

val stream = ssc.receiverStream(new TFLTimsFeed())
stream.print()

ssc.start()
ssc.awaitTermination()
```

Ilustración 56. Filtrando los tweets para su posterior envío

Este bloque de código a priori parece muy grande y complicado, pero no es nada de eso, simplemente hay que tener en cuenta unas pocas ideas, el *stream* de tweets, el filtrado de estos y el envío de los tweets filtrados a la REST API como hicimos en el *custom receiver*:

1. Como habíamos puesto anteriormente, invocamos el *stream* de tweets con las utilidades de la API de Twitter, utilizando para ello la llamada al método *createStream()*.
2. Los *streams* se dividen por lotes o también llamados RDD (*Resilient Distributed DataSet*). Lo que hacemos a continuación es recorrer el *stream* con una estructura *foreachRDD(rdd, time)*, donde *rdd* será el lote actual del *stream* que estemos tratando y *time* es el *timestamp* correspondiente a dicho lote (no utilizaremos esta información, tan sólo la mostraremos por consola para diferenciar los lotes). Un lote/RDD puede estar dividido en diferentes particiones, donde cada partición tendrá una cantidad determinada de tweets, es decir, si en un lote hay 20 tweets, esos 20 tweets estarán almacenados en diferentes particiones, por ejemplo, 4 particiones de 5 tweets cada una.
3. Ahora, por cada RDD/lote de tweets del *stream*, siempre que este tenga algún tweet (*rdd.count() > 0*), uniremos todas las particiones en una única partición con el método *repartition()* y con el método *collect()* pasaremos el objeto de tipo RDD a una lista de tweets (objeto *list* de Java/Scala, donde cada elemento es de tipo Tweet, proveniente de la Api de Twitter) para tratar la información de manera sencilla.
4. Una vez tengamos esto, los siguientes pasos son muy sencillos. Recorreremos la lista con los tweets del lote y le aplicamos a cada tweet el análisis de sentimientos. Además, almacenaremos en diferentes variables la información de cada Tweet que se necesite, por ejemplo, el contenido o cuerpo del tweet, el usuario, la localización, etc. Es importante fijarse que la localización del tweet proviene de la ubicación del perfil del usuario que escribió el tweet y no de la geolocalización propiamente dicha (línea comentada). Esto es debido a lo que se explicó anteriormente, hay un porcentaje alto de que los campos de latitud y longitud de cada tweet estén vacíos. Una vez tengamos el análisis y los campos requeridos, comprobaremos que la localización contenga la palabra “London” y que el análisis de sentimientos nos devuelva una valoración neutra, positiva o muy positiva (*NEUTRAL , POSITIVE, VERY POSITIVE*).
5. Si un tweet pasa el filtro, además de imprimirlo por consola, crearemos un objeto JSON exactamente igual que como lo hicimos con las incidencias en el *custom receiver*, con los campos que hemos guardado anteriormente y los que nos aporta información importante de cada tweet, como el usuario, la fecha de creación del tweet, el contenido del tweet, etc. Dichos campos son los que mostraremos en la aplicación web (se puede añadir toda la información que queramos y que por supuesto nos sea útil).
6. Por último, enviaremos dicho objeto (vía POST HTTP, como hicimos en el *custom receiver*), como un objeto *String* (método *prettyPrint()*), hacia el Rest Api, que se encargará de mostrarlo por pantalla, pero esto lo haremos en la siguiente sección de la memoria.

También se ha llevado un contador de los tweets que recibimos, pero tan sólo de manera auxiliar y para ver cómo va fluyendo los tweets y contar la cantidad de tweets que maneja la aplicación, cuáles cumplen nuestros criterios de filtro y cuáles no. Ahora ya podemos ejecutar nuestra aplicación y ver los tweets filtrados que nos devuelve la aplicación (nos saldrá un error al hacer la petición POST ya que todavía no hemos creado la aplicación, podemos comentar ese trozo de código por ahora).

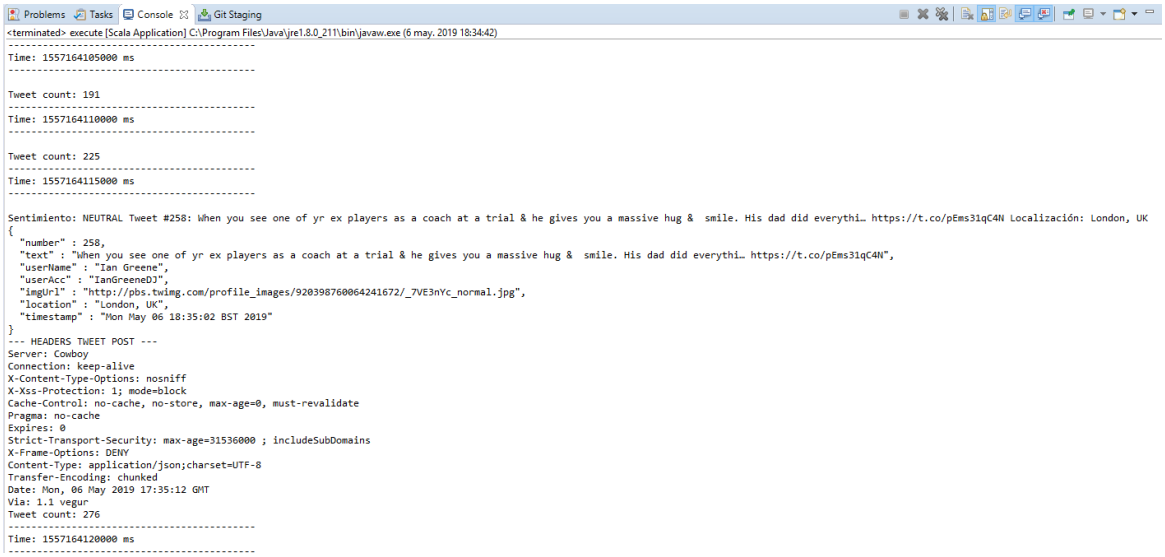


Ilustración 57. Resultado de la ejecución final de la aplicación Scala

Como se puede ver en la imagen anterior, hubo tres lotes de tweets (cada 5 segundos, identificado por su timestamp) donde no se mostró ningún tweet por consola. Esto es debido a que ningún tweet de dichos lotes cumplió con nuestros filtros (215 tweets no cumplieron los filtros), pero en el último lote si se cumplió las condiciones en un tweet (el análisis de sentimientos devolvió NEUTRAL y la localización es de Londres). Este tweet si se enviará a la Rest Api y se mostrará por pantalla. El flujo de los tweets en esta ejecución fue de aproximadamente 30-40 tweets por lote, es decir, cada 5 segundos.

Con esto concluimos con la aplicación Scala – Spark Streaming, encargada de hacer el tratamiento de los datos. Ahora pasaremos a la aplicación web, encargada de representar dicha información.

4.3.6. Repositorio público con el código fuente

Se ha habilitado un repositorio donde se puede consultar todo el código fuente y librerías usadas durante el desarrollo de la aplicación:

Repositorio Bitbucket con el código y librerías usadas en la aplicación Scala: aquí.

Juan Ramón Betancor Olivares. Última actualización 01-05-2019

Si se quiere ejecutar la aplicación directamente desde el código del repositorio tan sólo debemos de seguir los siguientes pasos:

1. Clonar el repositorio donde se encuentra el código fuente.
2. Abrir el proyecto en el IDE que se quiera utilizar.
3. Importar los JARS/librerías que se encuentra dentro del proyecto (*External JARS.zip*)

4. Ya se podría ejecutar la aplicación sin problema alguno. Recordar si aparece el error de que las versiones de Scala y Spark son diferentes o incompatibles debemos de igualarlas como se propuso en la solución de la página 22.

4.3.7. Posibles propuestas de mejora para la aplicación.

- Tratamiento de las librerías usando Maven o SBT, aportando una mayor flexibilidad y facilidad de manejo de las librerías sin necesidad de descargar los JARS.
- En caso de que se pudiese dar otra solución a la problemática de la geolocalización de los tweets, que aportase una mayor precisión a la hora de localizar los tweets sería conveniente implementarla.
- Implementar nuevos métodos de filtrado a los tweets o mejorar los ya hechos de tal manera de que se podría pulir aún más las búsquedas.
- En lugar de descargarnos cada vez el dataset al completo, sería interesante buscar alguna manera de descargar tan sólo las incidencias cuyo estado sean "*Active*" o "*Active Long Term*" ya que estas son las utilizaremos, mejorando así los tiempos de descarga.

4.4. Rest API Spring Boot

Con esta aplicación buscamos ofrecer, de manera visual y ordenada, la información tratada en la aplicación Scala al público, mediante una página web sencilla. Para ello representaremos las incidencias en un mapa de la ciudad y los tweets de los usuarios en una tabla ordenada.

4.4.1. Desglosando y estructurando la aplicación + Diseño previo.

Como framework de desarrollo utilizaremos Spring Boot el cual es un framework basado en MVC. Por lo que la aplicación la dividiremos en tres partes principales:

- **Modelo** o entidades: representará los objetos que vamos a tratar en la aplicación. Crearemos dos entidades, una que referenciará a los tweets y otra a las incidencias.
- **Controlador**: encargado de recibir la información de la aplicación de Scala, convertir ese JSON entrante a la entidad correspondiente y almacenarla posteriormente en la base de datos. También se encargará de leer desde la base de datos la información para mandárselo a las vistas.
- **Vistas**: encargado de mostrar la información e interactuar con el usuario. Utilizando la web Marvel, la cual nos permite crear diseños y prototipos de nuestros proyectos, he creado un posible diseño de cómo se verá en un futuro el aspecto de la página principal de nuestra aplicación:

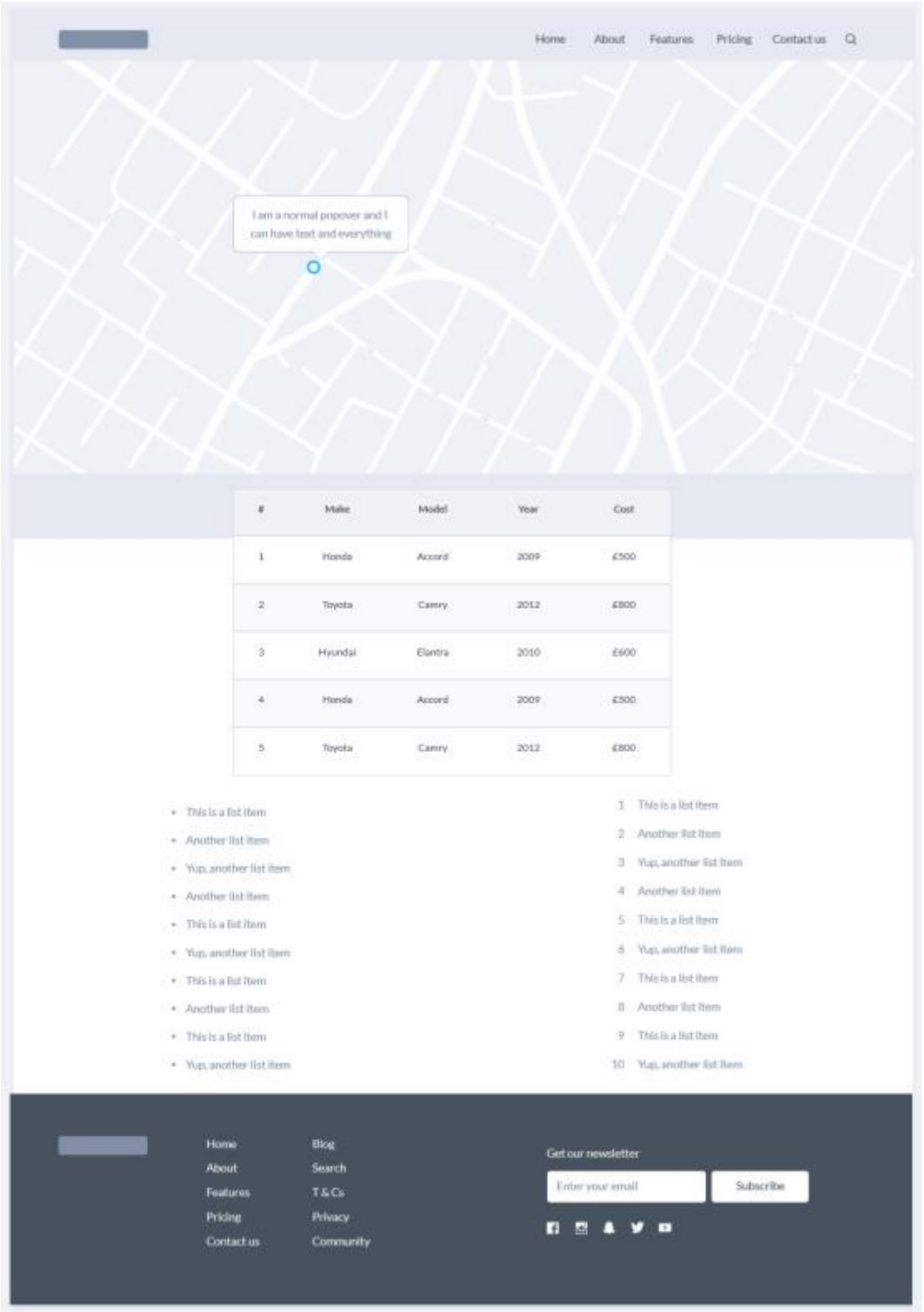


Ilustración 58. Diseño previo de la página donde mostraremos las incidencias y los tweets

La cual se divide principalmente en la barra de navegación, el mapa con las incidencias, la tabla con todas las incidencias, las listas con los tweets y por último el footer que delimita el final de la página.

4.4.2. Tecnologías utilizadas y entorno de desarrollo.

IntelliJ IDEA JetBrains

Como entorno de desarrollo utilizaremos en este caso IntelliJ IDEA, el entorno de desarrollo orientado a Java por parte de JetBrains. Gracias a que disponemos la licencia de estudiante ofrecida por la universidad me he decantado por esta solución para crear la aplicación. Además, durante la carrera la he usado en más de una ocasión para desarrollar proyectos, por lo que ya estoy acostumbrado a usarlo y no necesitaré demasiado tiempo para adaptarme. También existe otras soluciones de código abierto, como puede ser Eclipse, la más común.

Ilustración 59. IntelliJ IDEA Logo

Spring Boot (+ librerías) + Java

Como framework de desarrollo utilizaremos Spring Boot con varias de sus librerías externas como pueden ser Spring Security. Spring Boot es un framework para el desarrollo de aplicaciones web, para la plataforma Java (lenguaje de programación orientado a objetos). Me he decantado por esta solución en lugar de otra, como puede ser Ruby on Rails, porque ya he hecho un proyecto previo a este con este Framework y me pareció muy cómodo e intuitivo de usar, en gran parte porque es de Java, el principal lenguaje que se da en la carrera. Además, hay un foro y una gran cantidad de tutoriales propuestos por la propia organización que facilitan bastante el desarrollo de aplicaciones.

Ilustración 60. Spring Boot logo

Thymeleaf

Thymeleaf es una librería Java que implementa un motor de plantillas de HTML5 (también extensible a otros formatos) que puede ser utilizado tanto en modo web como en otros entornos no web. Se acopla muy bien para trabajar en la capa vista del MVC de aplicaciones web. El objetivo principal de Thymeleaf es permitir la creación de plantillas de una manera elegante y un código bien formateado.

Ilustración 61. Thymeleaf logo

HTML/JavaScript/CSS3/JQuery/Bootstrap/Flexbox Grid

Para desarrollar todo lo relacionado con el *frontend* de la aplicación web utilizaremos una serie de lenguajes como HTML, para formar el cuerpo de la página, JavaScript para manejar eventos y CSS3 para mejorar la vista de la web, junto a Bootstrap. Utilizaremos Flexbox Grid para estructurar el contenido.

Ilustración 62. Lenguajes de programación web

OpenStreetMap + Leaflet

Para mostrar el mapa de las incidencias utilizaremos en conjunto Leaflet y OpenStreetMap (*OSM*). Leaflet no es más que una librería JavaScript utilizada para trabajar con mapas en cualquier aplicación web. Dicho mapa lo crearemos mediante OpenStreetMap, una solución open source a Google Maps que nos permite crear mapas en nuestras páginas web de manera gratuita.

Ilustración 63. Leaflet y OpenStreetMap logo

PostgreSQL

Para establecer la persistencia de los datos, utilizaremos el servicio de bases de datos que nos brinda Heroku, que, en este caso, se trata de una base de datos basado en PostgreSQL.

PostgreSQL no es más que un sistema de gestión de bases de datos relacional orientado a objetos y de código abierto.

Ilustración 64. PostgreSQL logo

Heroku/GitHub

Por último, para el *deploy* o despliegue de nuestra aplicación web utilizaremos la opción de Heroku. Heroku es una plataforma que, entre otras funcionalidades, nos permite alojar nuestras aplicaciones web de manera gratuita. Y como sistema de control de versiones esta vez utilizaremos GitHub, que, al igual que Bitbucket, nos permite crear y gestionar nuestro repositorio donde almacenaremos el código fuente de la aplicación.

Ilustración 65. Heroku y GitHub Logo

4.4.3. Creación de la aplicación

Preparando el entorno de desarrollo (IDE)

Como ya hemos comentado, el entorno de desarrollo que vamos a utilizar en el desarrollo de esta aplicación será IntelliJ IDEA, el cual la universidad nos brinda una licencia de estudiante.

Para poder descargar el IDE tenemos que registrarnos en la página oficial, utilizando para ello la cuenta de correo institucional de la universidad. Este paso es muy importante ya que nuestra licencia está asociada a dicha cuenta de correo.

Una vez hayamos creado nuestra cuenta e iniciado sesión, podemos entrar en nuestro perfil y, en el apartado licencias, aparecerá la licencia de estudiante que disponemos. Tan sólo

accedemos al apartado donde se encuentra el IntelliJ IDEA y una vez dentro procedemos a descargar la aplicación.

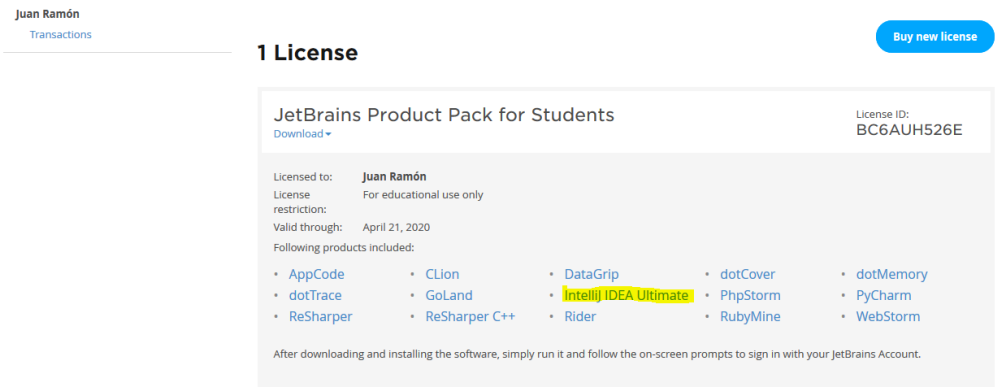


Ilustración 66. Licencia de estudiante JetBrains

Una vez tengamos el ejecutable descargado, lo ejecutaremos como administrador para proceder a la instalación. Durante el proceso de instalación nos pedirán nuevamente poner las credenciales de la cuenta para que la licencia se active en el IDE. Podemos dejar todos los campos que nos aparecen durante el proceso de instalación por defecto (nos pedirá el JDK de Java que instalamos al principio de la memoria. Este aparece por defecto). Una vez instalado nos aparecerá una ventana como la siguiente (sin el proyecto creado):

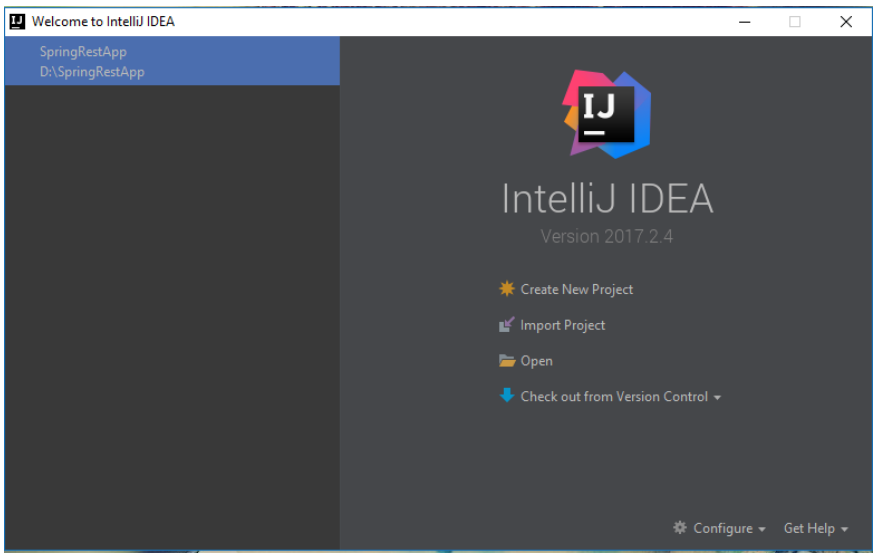


Ilustración 67. Interfaz inicial IntelliJ IDEA

Creando el proyecto Spring Boot

Como comenté anteriormente, una de las ventajas de usar Spring Boot como framework de desarrollo es que la propia organización nos ofrece una serie de tutoriales o guías de cómo crear aplicaciones básicas, que luego se pueden escalar fácilmente. Para el desarrollo de esta aplicación seguí el tutorial básico de cómo crear un servicio web REST.

Lo primero que tenemos que hacer será crear la aplicación, para ello, hacemos clic en "*Create New Project"* como se puede ver en la imagen anterior. Posteriormente seleccionaremos Maven con herramienta para la gestión de librerías, además, automáticamente aparecerá el SDK que usaremos:

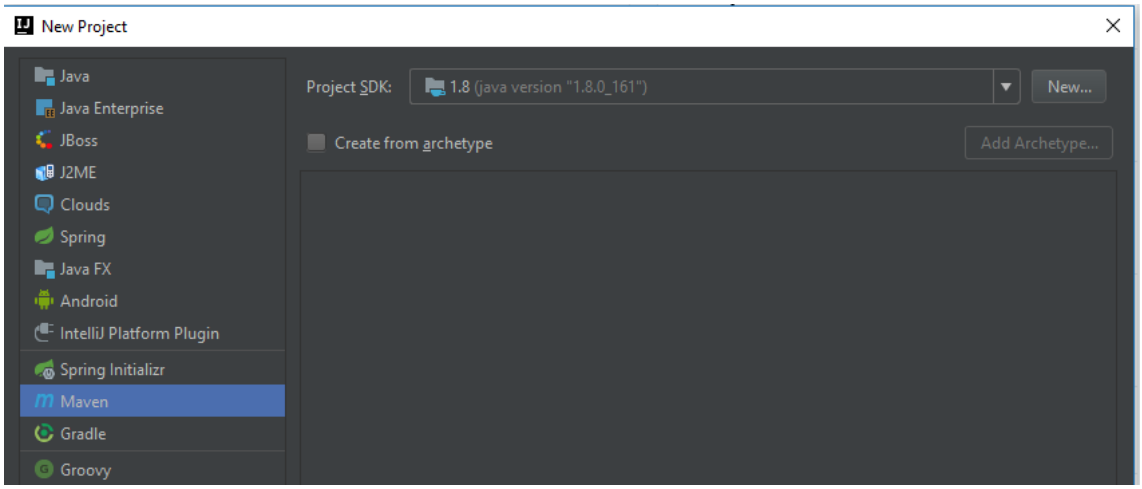


Ilustración 68. Estableciendo Maven en nuestro proyecto

En el próximo paso nos pedirá un "*GroupID"* y un "*ArtifactID"*, básicamente este primer parámetro permitirá identificar nuestro proyecto frente a otros proyectos, sigue la misma convención que el nombre de los paquetes de Java como vimos en la aplicación Scala. Por ejemplo: *es.cicei.ulpgc*, tal y como usamos en la aplicación Scala. El "*artifactID"* en este caso no cumple ninguna función especialmente importante, así que podemos poner cualquier nombre, yo pondré el nombre de la aplicación:

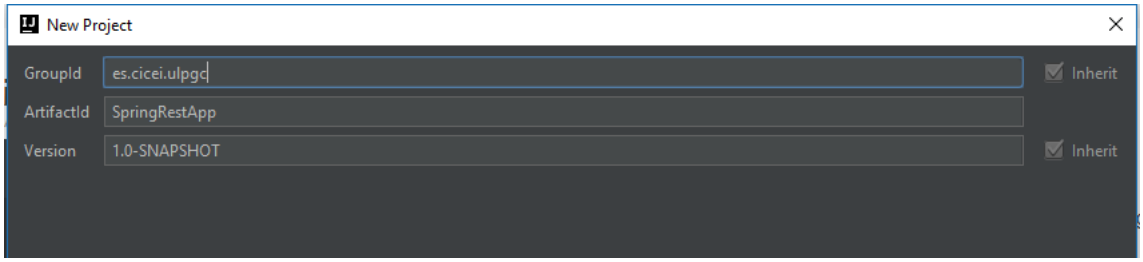


Ilustración 69. GroupID y ArtifactID

Posteriormente, en la siguiente ventana nos pedirá el nombre del Proyecto y la ubicación de este. Una vez hecho esto, ya podemos finalizar. Se nos quedará un proyecto con la siguiente estructura:

Ilustración 70. Estructura del proyecto inicial

Los directorios/archivos importantes con los que trabajaremos son:

- *pom.xml*: para la gestión de dependencias.
- *src > main > java*: pondremos todos los archivos pertenecientes al *backend*.
- *src > main > resources*: pondremos todos los archivos pertenecientes al frontend.

Antes de empezar a crear código, primero haremos dos cosas previas. La primera de ella es crear un paquete como lo hicimos en la aplicación Scala, donde crearemos todo el código. Para ello, desde el IDE, sobre el directorio *src > main > java*, hacemos clic derecho, *New > Package* y ponemos el nombre deseado (en este caso, *es.cicei.ulpgc* nuevamente).

Por otra parte, vamos a importar de una todas las librerías que vamos a utilizar en el proyecto. Este paso es extremadamente importante, ya que la mayoría de los posibles errores que puedan ocurrir durante el desarrollo del proyecto estarán relacionados con las librerías o dependencias. Estas librerías son las pertenecientes a:

- Spring Boot
- Thymeleaf
- JPA + PostgreSQL (base de datos)
- Spring Security

Aquí entra en juego el archivo *pom.xml*, un fichero muy importante dentro del proyecto, perteneciente a Maven y, mediante este, gestionaremos todas las librerías y dependencias. Este fichero se encuentra en la raíz del proyecto, lo abrimos y debajo de *"<version>1.0-SNAPSHOT<version>"* añadimos lo siguiente:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>
```

Ilustración 71. Bloque parent dentro de pom.xml

Donde referenciamos el bloque *parent* y establecemos la versión de Spring Boot. Posteriormente, añadimos lo verdaderamente importante, las dependencias, justo debajo del bloque que acabamos de añadir:

```
<dependencies>
    <!-- Spring Boot -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Thymeleaf -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring3</artifactId>
        <version>3.0.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <!-- JPA + POSTGRE (Base de datos) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
```

Ilustración 72. Dependencias de nuestro proyecto (1)

```
        <!-- SPRING SECURITY -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity4</artifactId>
        </dependency>
    </dependencies>

    <properties>
        <java.version>1.8</java.version>
        <spring-version>5.0.6.RELEASE</spring-version>
    </properties>


    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

Ilustración 73. Dependencias de nuestro proyecto (2)

Este es posiblemente el paso más crítico de la aplicación debido a la cantidad de dependencias que tenemos que añadir y las diferentes versiones de cada una que existen. Durante el desarrollo del proyecto, el 75% de los errores que me han surgido han sido relacionado con las dependencias, ya sea por fallos en las versiones o que me faltaban dependencias por poner.

En las dos imágenes anteriores se pueden ver las diferentes dependencias asociadas a Thymeleaf, la seguridad, Spring Boot y la base de datos (separadas por los comentarios en el código). Una vez añadido esto, en caso de que el Maven no se actualice automáticamente (si hay cambios en el pom.xml se debería de actualizar automáticamente), hacemos clic derecho sobre el proyecto dentro del IDE, vamos a Maven (en la parte inferior del listado) y clicamos en "*Generate sources and Update folders"* para actualizar el proyecto de manera manual.

Preparación de la base de datos (conexión, repositorios y entidades)

Es hora de empezar a programar el backend. Inicialmente no tenía pensado utilizar ninguna base de datos para almacenar los tweets y las incidencias, directamente los almacenaba en una variable de clase de tipo *ArrayList* dentro del controlador, pero hay una problemática con esta forma de actuar y no es nada menos que los datos no son persistentes. Cada vez que ejecutaba la aplicación los datos anteriores se borraban o cuando durante un tiempo no se accedía a la aplicación en Heroku, al hibernar el servidor los datos también se perdían. Para solucionar esto, me decanté por utilizar una base de datos, en este caso, la que nos brinda Heroku. Para ello debemos crearnos una cuenta en Heroku e iniciar sesión en ella. Una vez creada la cuenta, podemos crear una aplicación dentro del *Dashboard* (tan sólo nos pedirá un nombre para la aplicación)*:*

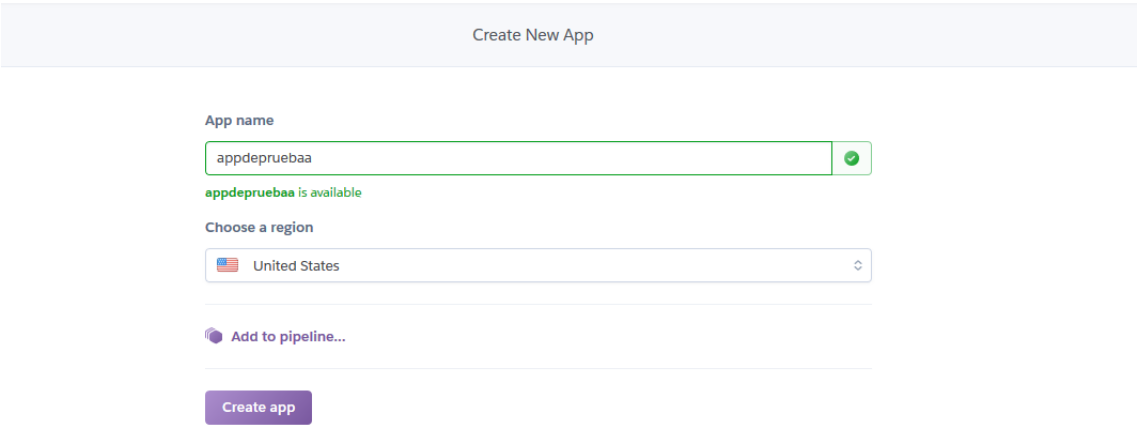


Ilustración 74. Creando aplicación desde Heroku

Una vez creada la aplicación dentro de Heroku, accedemos a la pestaña *Resources* de nuestra aplicación y en la barra de búsqueda de *add-ons* buscamos la herramienta Heroku PostgreSQL y la añadimos a la aplicación:

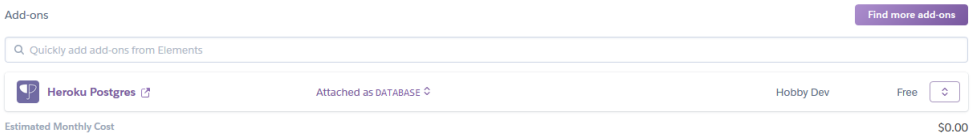


Ilustración 75. Aplicación Heroku creada

Una vez añadida, accedemos a ella clicando sobre esta. Una vez dentro, accedemos a la pestaña *settings* y clicamos sobre *View credentials*:

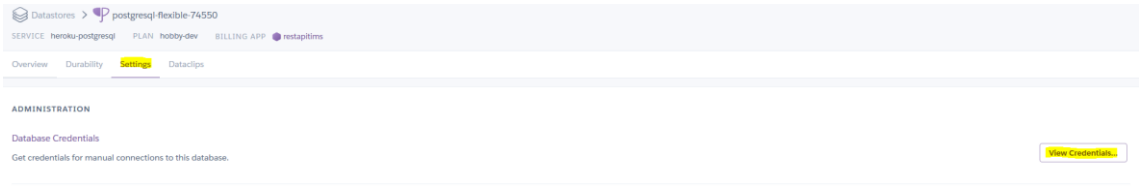


Ilustración 76. Pestaña credentials de la aplicación

Ahora ya nos aparecerá la información importante, la URL, el puerto, el usuario de acceso con la contraseña, etc. de tal manera que podemos establecer una conexión desde el código para trabajar con la base de datos:

Ilustración 77. Credenciales de la aplicación

Dejaremos esta pestaña abierta durante un momento y nos vamos al entorno de desarrollo. Una vez ahí, dentro del directorio *src > main > resources* crearemos un fichero llamado *application.properties* (clic derecho sobre el directorio *resources > new > file*) en el cual pondremos la conexión a la base de datos de la siguiente manera:

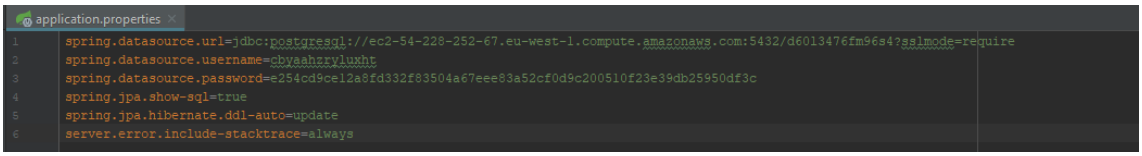


Ilustración 78. Conexión a la base de datos desde Spring

Al lado del nombre de cada propiedad pondremos las diferentes credenciales que buscamos en el paso anterior tal y como se ve en la imagen. Una vez hecho esto, procedemos a guardar el

archivo. Ahora que ya tenemos la conexión hecha, el siguiente paso es crear los repositorios JPA (Java Persistente API).

Los repositorios JPA permiten interactuar con la base de datos como si fueran objetos (siguiendo la principal finalidad de Java), suprimiendo las consultas (*queries*) a la base de datos como normalmente se suelen hacer (Select * from TABLE tableName). Estos repositorios ya ofrecen una serie de métodos que nos facilitan las consultas a la base de datos, haciendo la aplicación más fácil de comprender. Para crear estos repositorios tan sólo tenemos que hacer lo siguiente: haremos clic derecho sobre el paquete que inicialmente creamos *> new > Java Class* e insertamos los nombres. En este caso, como vamos a guardar dos tipos de objetos o entidades, vamos a crear dos repositorios, uno para las incidencias, lo llamaremos *DisruptionRepository.java* y otro para los tweets, *TweetsRepository.java,* respectivamente.

Una vez creado añadimos lo siguiente dentro de ellos:

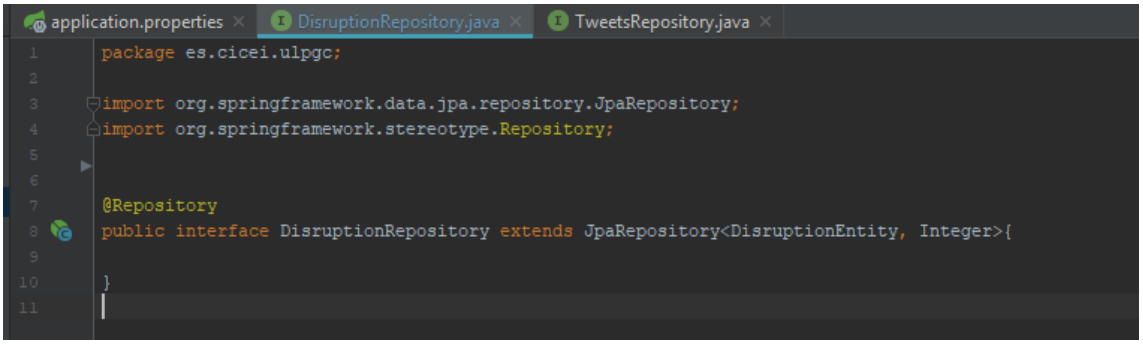


Ilustración 79. DisruptionRepository

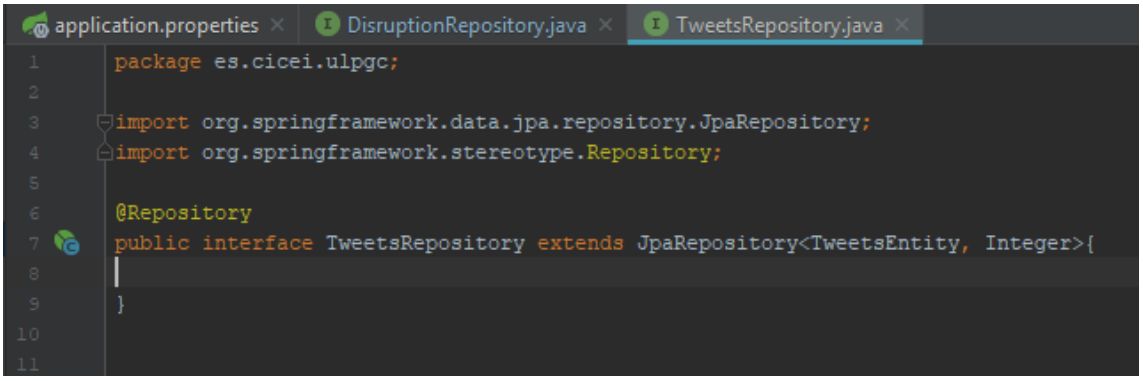


Ilustración 80. TweetsRepository

Como se puede observar, el código de ambos es identico. Tan sólo debemos declarar que la interfaz es un reporitorio mediante la anotación *@Repository* y que cada repositorio extiende de *JpaRepository*. Dentro de estas interfaces se puede declarar nuevos métodos para acceder/consultar a la base de datos, como por ejemplo *findAllByNameSortBySurname()*. En nuestro caso, no necesitamos realizar ninguna consulta diferente a las que *JpaRepository* ya nos ofrece por defecto así que lo dejaremos en blanco, pero es importante saber que si necesitamos hacer alguna *query* específica, dentro de estas interfaces es donde debemos colocarlas.

Una vez creado los repositorios, lo siguiente que tenemos que hacer es crear las dos entidades, la entidad que hace referencia a un tweet y la entidad que referencia a una incidencia.

Para crear estas entidades hay que echar la vista atrás y recordar que estructura tenían los objetos dentro del JSON que enviabamos de la aplicación de Scala.

La estructura del JSON que representaba a la incidencia era la siguiente:

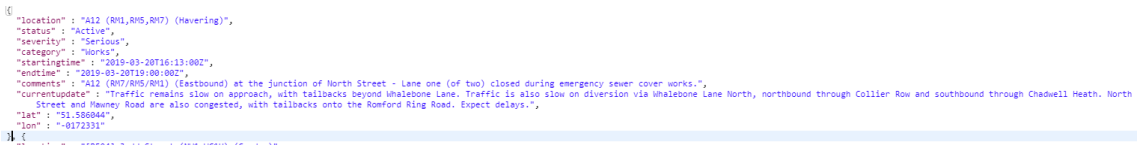


Ilustración 81. Objeto JSON que representa a la incidencia de tráfico

Donde se aprecia una localización, un estado, una magnitud etc. Hay que tener en cuenta estos campos. Por otra parte un tweet tenía la siguiente estructura:

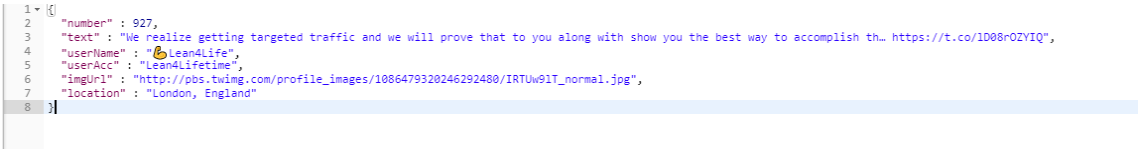


Ilustración 82. Objeto JSON que representa un tweet

En la que se aprecia un número de tweet, el texto, usuario, etc.

La idea es crear una clase en la que se tenga cómo atributos de clase estos campos, acompañados de sus correspondientes *getters*() y *setters*(). Para ello repetimos el proceso, clic derecho sobre el paquete *> new > Java Class* y le pondremos el nombre a la clase, en este caso crearemos dos entidades, *DisruptionEntity.java* y *TweetsEntity.java* respectivamente.

El contenido de la entidad que referencia a una incidencia es la siguiente:

application.properties | DisruptionRepository.java | TweetsRepository.java | DisruptionEntity.java

```java
package es.cicei.ulpgc;

import javax.persistence.*;

@Entity
@Table(name = "disruptions")
public class DisruptionEntity {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    @Column(length = 20048)
    private String location;
    @Column(length = 20048)
    private String status;
    @Column(length = 20048)
    private String severity;
    @Column(length = 20048)
    private String category;
    @Column(length = 20048)
    private String startingtime;
    @Column(length = 20048)
    private String endtime;
    @Column(length = 20048)
    private String comments;
    @Column(length = 20048)
    private String currentupdate;
    @Column(length = 20048)
    private String lat;
    @Column(length = 20048)
    private String lon;

    public long getId() { return id; }

    public void setId(long id) { this.id = id; }

    public String getLocation() { return location; }

    public void setLocation(String location) { this.location = location; }

    public String getStatus() { return status; }

    public void setStatus(String status) { this.status = status; }

    public String getSeverity() { return severity; }

    public void setSeverity(String severity) { this.severity = severity; }

    public String getCategory() { return category; }
```

Ilustración 83. Entidad DisruptionEntity que referencia a una incidencia

Para crear los getters y setters en el IDE tan sólo basta con ir a *Code > Generate* y dentro de la ventana, *Getter and Setter* (seleccionamos los atributos a los cuales queremos crearle los getters y setters, en este caso a todos los atributos de la clase. Estos atributos debemos crearlos previamente). Es obligatorio olvidarnos de crear un constructor de clase ya que nos devolverá un error si lo creamos, tan sólo debemos indicar los atributos y los getters y setters.

Hay que señalar varias cosas importantes en el código:

- Hay que importar la clase *javax.persistence.** para acceder a los anotadores (@)
- Vemos que la entidad debe estar anotada bajo la sentencia @Entity, lo que estamos diciendo aquí es que estamos frente a una entidad, como su nombre indica.
- Con el anotador *@Table*, estamos diciendo que en la base de datos, la tabla que contendrá todas las incidencias llevará por nombre “disruptions”.
- Todo fila dentro de la tabla, es decir, todo objeto (incidencia) dentro de la tabla deberá de estar identificado con un ID (*@Id*). Este campo no aparece en la estructura del JSON,

ya que este campo es autogenerado (*@GeneratedValue*) por la aplicación y este debe de ser único.

- Todos los campos/columnas/atributos de la entidad debe de estar bajo el anotador *@Column*. En la tabla habrá una columna por cada *@Column* en el código.
- Y posteriormente deberá de estar presente todos los getters y setters.

En la siguiente imagen vemos el contenido de la entidad que referencia a los tweets:

application.properties | DisruptionRepository.java | TweetsRepository.java | DisruptionEntity.java | TweetsEntity.java

```
package es.cicei.ulpgc;

import javax.persistence.*;


@Entity
@Table(name = "tweets")
public class TweetsEntity {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    public long getId() { return id; }

    public void setId(long id) { this.id = id; }

    private int number;
    @Column(length = 20048)
    private String text;
    @Column(length = 20048)
    private String userName;
    @Column(length = 20048)
    private String userAcc;
    @Column(length = 20048)
    private String imgUrl;
    @Column(length = 20048)
    private String location;
    @Column(length = 20048)
    private String timestamp;

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) { this.timestamp = timestamp; }

    public int getNumber() { return number; }

    public void setNumber(int number) { this.number = number; }

    public String getText() { return text; }

    public void setText(String text) { this.text = text; }

    public String getUserName() { return userName; }

    public void setUserName(String userName) { this.userName = userName; }
```

Ilustración 84. Entidad TweetsEntity que representa un tweet

Como se puede observar, el contenido es prácticamente el mismo con la única diferencia de los atributos de la clase. La explicación anterior sirve perfectamente para esta clase, con la diferencia de que la tabla, en este caso, se llamará *tweets* dentro de la base de datos.

Una vez tenemos ya las entidades y repositorios hechos ya podemos proceder a crear el controlador principal de la aplicación, encargado del *routing*, recibir los JSON y tratarlos y el envío de información entre la base de datos y las vistas.

Creando el controlador

Volvemos a hacer clic derecho sobre el paquete *> new > Java Class*, en este caso lo llamaremos *MainApplicationController.java*. Una vez creado, vamos añadiendo lo siguiente:

```java
package es.cicei.ulpgc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.ModelAndView;

import java.util.ArrayList;
import java.util.List;

@RestController
public class MainApplicationController {

    @Autowired
    private DisruptionRepository disruptionRepository;
    @Autowired
    private TweetsRepository tweetsRepository;

    private List<DisruptionEntity> disruptions = new ArrayList<DisruptionEntity>(){};
    private List<TweetsEntity> tweets = new ArrayList<>();
```

Ilustración 85. Importando utilidades y creando los atributos dentro del controlador

Lo primero es importar las librerías que utilizaremos, en este caso, las relacionadas con el paquete HTTP, las anotaciones, el *ModelAndView* (para devolver las vistas cuando hacemos peticiones *get*) y las utilidades *List* y *ArrayList* que usaremos como auxiliares durante la creación de los diferentes *routings*.

Es importante establecer la anotación *@RestController* sobre la clase ya que con esto estamos indicando que nos encontramos con un controlador *Rest,* que estará en todo momento escuchando peticiones entrantes a la aplicación para tratarlas según nuestros criterios.

Una vez dentro de la clase debemos de crear los atributos de clase que usaremos. Como comenté anteriormente, inicialmente las incidencias y los tweets se guardaban en ArrayList (como se pueden ver en la imagen anterior: *disruptions* y *tweets*), pero estos datos no eran persistentes, por lo que se decidió usar una base de datos. Aun así, usaremos un ArrayList como objeto auxiliar, por lo que crearemos al menos uno (*tweets* en este caso).

Por otra parte, tenemos que crear un objeto de tipo *DisruptionRepository* y otro objeto de tipo *TweetsRepository*. Estos referencian a los repositorios JPA que creamos anteriormente y, mediante ellos, haremos las consultas de manera sencilla a la base de datos (leer, borrar y guardar datos). Cada repositorio referenciará a su correspondiente tabla, es decir, *DisruptionRepository* referencia a la tabla *disruptions* y por su parte, *TweetsRepository* referencia a la tabla *Tweets* de la base de datos.

El uso de *@Autowired* nos permite obviar los getters y los setters para esos atributos. Ahora que tenemos todo inicializado, podemos empezar a crear las rutas:

```
    private List<DisruptionEntity> disruptions = new ArrayList<DisruptionEntity>(){};
    private List<TweetsEntity> tweets = new ArrayList<~>();

    @RequestMapping("/")
    public ModelAndView index() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("homepage");
        return mav;
    }

    @GetMapping("/login")
    public ModelAndView login() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("login");
        return mav;
    }
```

Ilustración 86. Routing para la raíz y /login de la aplicación

Primero es el turno de la ruta raíz de la aplicación web (por ejemplo, http://localhost:8080/), que nos devolverá la *homepage* cuando hagamos una petición *get* a la ruta raíz de la aplicación desde el navegador. Lo importante de este primer método es la creación de un objeto de tipo *ModelAndView* y, que mediante la llamada al método *setViewName(),* indicaremos el nombre del fichero HTML que se mostrará cuando se acceda al recurso (no es necesario poner el nombre completo: homepage.html, tan sólo el nombre). Cuando creamos un objeto *ModelAndView*, la aplicación buscará los templates dentro del directorio *src > main > resources > templates*, por lo que tendríamos que poner todas las plantillas dentro de esa carpeta, o bien, si usamos otra localización, debemos de indicarla correctamente. La creación de las vistas lo veremos en otro apartado más adelante.

Haremos lo mismo con la ruta */login*, para más adelante, asegurar nuestra aplicación de manera que tan sólo los usuarios registrados puedan acceder al contenido de la aplicación. En este caso devolveremos la plantilla con nombre *login* (*login.html* bajo el directorio *resources > templates*) y lo dejamos tal cual aparece en la imagen, más adelante cuando se explique el apartado de Spring Security retomaremos la ruta.

Ahora vamos por la parte importante, donde crearemos las rutas donde debemos realizar las peticiones POST con los JSON desde la aplicación Scala, para posteriormente transformarlas a la entidad correspondiente y añadirlas a la base de datos. La primera ruta será la de las incidencias:

```
    @RequestMapping(value = "/", method = RequestMethod.POST)
    public ResponseEntity<List<DisruptionEntity>> getJson(@RequestBody List<DisruptionEntity> disruptions) {
        //this.disruptions = disruptions;
        disruptionRepository.deleteAll();
        for (int i = 0; i < disruptions.size(); i++) {
            disruptionRepository.save(disruptions.get(i));
        }
        return new ResponseEntity<List<DisruptionEntity>>(disruptions, HttpStatus.OK);
    }
```

Ilustración 87. Ruta POST para las incidencias de tráfico

Hay que tener en cuenta varias cosas en la imagen anterior:

- La anotación *@RequestMapping* establece la ruta y el tipo de la petición para acceder al recurso/funcionalidad. En este caso, estamos estableciendo que el método a continuación se ejecutará, siempre y cuando se realice una petición POST a la raíz (“/”) de la aplicación.

- En la declaración del método *getJson(),* que devuelve un objeto de tipo *ResponseEntity*, es importante el uso de la anotación *@RequestBody*, ya que, gracias a esto, se transformará el cuerpo de la petición JSON, es decir, el JSON con todas las incidencias que enviamos desde la aplicación Scala a un objeto de tipo *List<DisruptionEntity>,* siendo *DisruptionEntity* la entidad que creamos nosotros. Esta anotación prácticamente realizará todo el trabajo, ya que ahora tan sólo debemos de recorrer la lista y añadir cada elemento a la base de datos.
- En la línea que aparece comentada era la solución anterior a la implementación de la base de datos, en la cual, tan sólo sobrescribíamos el contenido del atributo de la clase a la lista que recibimos desde la petición. Pero como se ha comentado en varias ocasiones, esta solución no ofrece la persistencia de los datos.
- Lo primero que tenemos que hacer es borrar la tabla *disruptions*, la cual es la tabla donde se almacenará todas las incidencias. Para ello haremos uso de los métodos que nos ofrece la interfaz JPA, que, de manera sencilla, nos permite hacer las consultas con una interfaz, como si las conexiones a la base de datos fuesen objetos de Java. Con la llamada al método *deleteAll()* borraremos todos los registros de la tabla.
- Una vez borrada la tabla, procedemos a llenarla nuevamente con las nuevas incidencias. Para ello, con un *for* simple que recorra la lista recibida, haremos la operación *save()* sobre el repositorio correspondiente por cada incidencia de la lista. El método *save()* guardará la incidencia en un registro de la tabla de datos (se efectúa el INSERT de SQL).
- Una vez guardado todas las incidencias en la tabla, devolvemos un objeto ResponseEntity con la misma lista recibida y el código de respuesta 200 (*HttpStatus.OK*)
- Esta operación de borrado completo de la tabla y posterior llenado se realizará en cada petición POST que hagamos a la ruta, que será cada 2 minutos y medio, como establecimos en el receiver de la aplicación Scala. Esta puede que no sea la manera correcta de actuar ya que se borrarán incidencias de la base de datos que se colocarán nuevamente, por lo que se puede ahorrar consultas y, por lo tanto, recursos y tiempo. Una posible solución puede ser añadir un campo más al JSON y a la entidad, siendo este el ID de la incidencia que podemos sacar directamente desde el dataset. Una vez añadimos las incidencias a la base de datos y vengan una nueva petición de incidencias podemos comparar el ID de las incidencias que llegan para saber si ya está en la base de datos y, de esta manera, no borrarlas si se encuentran ya (añadimos solo las nuevas).

<Disruption id="214847">

Una vez tengamos la ruta de las incidencias, procedemos a la ruta de los tweets:

```
47
48      @RequestMapping(value = "/tweets", method = RequestMethod.POST)
49      public ResponseEntity<TweetsEntity> getJsonTweet(@RequestBody TweetsEntity tweet) {
50          this.tweets = tweetsRepository.findAll();
51          this.tweets.add( index: 0, tweet);
52          if(this.tweets.size() > 15){
53              this.tweets = this.tweets.subList(0,15);
54          }
55          tweetsRepository.deleteAll();
56          for (int i = 0; i < this.tweets.size(); i++){
57              tweetsRepository.save(this.tweets.get(i));
58          }
59          return new ResponseEntity<TweetsEntity>(tweet,HttpStatus.OK);
60      }
```

Ilustración 88. Ruta POST para los tweets

Este bloque de código es casi igual que el anterior con algunas diferencias:

- En este caso el recurso se encuentra en la ruta */tweets*, la cual accederemos mediante una petición POST.
- Lo que recibiremos de la petición JSON no será una lista de tweets, sino que serán tweets independientes (objeto de tipo TweetsEntity, la entidad que creamos nosotros).
- Para no mostrar una lista muy grande de tweets en la web, lo que hacemos es establecer un umbral máximo de tweets para guardar en la base de datos. Este umbral lo he establecido en 15 tweets, pero se podría poner cualquier valor deseado. Lo que haremos es leer los tweets que hay actualmente en la base de datos y lo guardamos en una variable (para ello usamos la lista auxiliar que creamos previamente como atributo de clase). Luego añadimos el tweet que acabamos de recibir por la petición JSON en la primera posición de la lista (método *add(index:0, tweet)*), para ordenarlo de más reciente a menos. Luego comprobamos si el tamaño de la lista supera el umbral, y si es así, haremos una lista (método *subList()*) sin el elemento más antiguo de la lista, que en este caso será el de la posición 15, el primero que se añadió. Por último, borramos el contenido de la tabla y añadimos nuevamente la lista de tweets con el tweet que recién acaba de llegar a la base de datos.
- Por último, devolvernos el objeto *ResponseEntity* con el tweet recibido y la respuesta 200 "OK".
- También se podría mejorar las consultas a la base de datos preguntando directamente a la base de datos la cantidad de tweets que hay actualmente en la tabla. Si hay más de 15, se borrará el tweet de la posición 15 de la tabla y se añadirá en la primera posición el tweet recién recibido (sin necesidad de leer todo y borrar todo). Para realizar esto, si es necesario realizar algún método personalizado dentro del repositorio (*TweetsRepository.java*) ya que seguramente la interfaz JPA no disponga de los métodos necesario, como un método que directamente te borre el tweet de la posición 15 o un método que te añada en la primera posición de la tabla el tweet y que despliegue a la derecha una posición los demás tweets.

Para acabar con el controlador falta una última ruta, la ruta *GET* más importante por la que gira toda la aplicación. En esta última ruta mostraremos todos las incidencias en el mapa y los tweets que hemos guardado en la base de datos. Para ello añadimos lo siguiente:

```
62      @RequestMapping("/disruptions")
63      public ModelAndView renderIndex() {
64          List<DisruptionEntity> disruptions = disruptionRepository.findAll();
65          List<TweetsEntity> tweets = tweetsRepository.findAll();
66          ModelAndView mav = new ModelAndView();
67          mav.addObject( attributeName: "disruptions", disruptions);
68          mav.addObject( attributeName: "tweets", tweets);
69          mav.setViewName("index");
70          return mav;
71      }
72
```

Ilustración 89. Ruta /disruptions donde mostraremos el mapa y los tweets

Bajo la ruta */disruptions*, mediante una petición GET (por defecto, cuando usamos la anotación *@RequestMapping,* nos estamos refiriendo a una petición GET) accederemos a este recurso.

Posteriormente crearemos dos variables auxiliares, *List* en este caso, en la cual verteremos el contenido de las tablas de la base de datos mediante la llamada al método *findAll*(). Posteriormente creamos un objeto *ModelAndView* con el que mediante el método *addObject*(), le pasamos como parámetro el nombre del objeto o atributo que queremos pasarle a la vista y el contenido de este, que, en este caso, serán las listas con los tweets y las incidencias. Por último, indicamos el nombre de la vista encargada de mostrar la información (*index*, que hace referencia al fichero index.html bajo el directorio *resources*) y lo retornamos.

Ahora que tenemos terminado el controlador, ya disponemos de dos de las tres partes del MVC. Disponemos de los modelos o entidades en este caso, al igual que disponemos del controlador. Ahora nos falta por crear las diferentes vistas que hemos ido viendo durante el desarrollo del controlador.

Creando las vistas de la aplicación

Como ya he comentado en varias ocasiones anteriormente, el directorio ideal para crear todas las plantillas es bajo la dirección *src > main > resources > templates* ya que Thymeleaf y, en general Spring Boot busca los *templates* bajo ese directorio, es decir, la carpeta *templates* es la raíz por donde se empieza a buscar. Esta carpeta *templates* inicialmente no está creada, así que procedemos a crearla: clic derecho sobre el directorio *resources > new > Directory* e indicamos como nombre: *templates*.

Dentro de esta podremos crear un directorio para los estilos CSS y otra para los ficheros JS en caso de que se requieran (recomendable).

En esta sección tenemos que crear al menos 3 vistas:

- El homepage, con nombre *homepage.html*
- El formulario de login, con nombre *login.html*
- El índice o página principal, con nombre *index.html*

Para ello haremos uso de Thymeleaf como motor para crear las plantillas, Bootstrap para unos estilos simples y también haremos uso de un GRID para intentar colocar mejor todos los elementos dentro de la plantilla y que estos a su vez sean *responsive (*clases "*col"* y "*row")*.

Para usar Thymeleaf ya añadimos las dependencias al principio de la sección y Bootstrap lo añadiremos mediante los CDN. Por otra parte, debemos descargarnos el GRID que usaremos. Podemos usar el de Bootstrap, pero me decanté por usar *Flexbox Grid* ya que es el que solía usar durante los proyectos de carrera y ya estoy habituado a usarlo en los proyectos. Para ello accedemos a la página de descarga y procedemos a bajarnos el fichero.

Una vez descargado el fichero y descomprimido, accedemos a él y, bajo el directorio */css*, copiamos/cortamos el fichero cuyo nombre es *flexboxgrid.css* y lo pegamos en el directorio destinado a guardar todos los *css* dentro de nuestro proyecto. Si hemos creado la carpeta */css* dentro de *src > main > resources > templates* lo pondremos ahí, aunque también podemos dejarlo en *templates*.

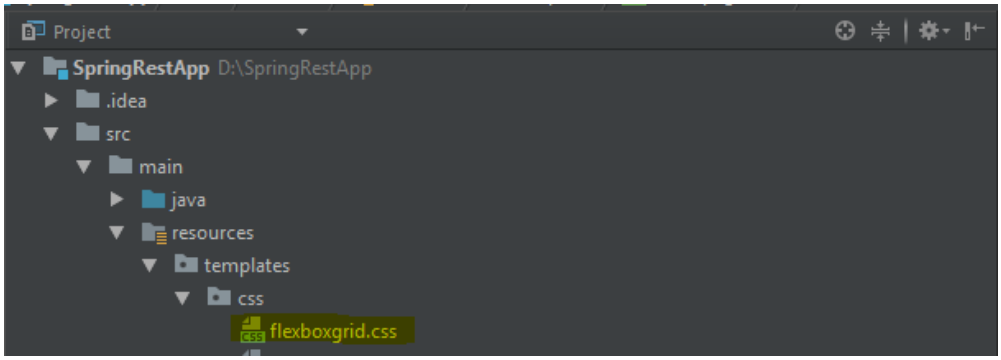


Ilustración 90. Ubicación de flexboxgrid.css dentro del proyecto

Una vez tengamos estos preparativos terminados podemos empezar a crear nuestras plantillas. Empezaremos creando la plantilla más sencilla de las tres, en este caso se trata de la página principal o homepage. La idea de esta página es simplemente mostrar, con una imagen de fondo, el título del aplicativo, de tal manera que el usuario al clicar sobre éste acceda al verdadero contenido de la aplicación. Para crear el fichero HTML hacemos clic derecho sobre *src > main > resources > templates* y luego vamos a *New > HTML file*. Posteriormente ponemos como nombre de fichero *homepage.html* (es importante respetar los nombres de los ficheros que pusimos en el *routing* cuando creamos el controlador). Inicialmente al crearlo, deberá de tener una estructura similar a la siguiente:

Ilustración 91. Estructura inicial de archivo HTML recién creado

Haremos unos pocos añadidos. Dentro del bloque <head></head> le cambiaremos el título y le pondremos el nombre del aplicativo y, por otra parte, añadiremos un bloque CSS para modificar el estilo del título y para poner una imagen a tamaño completo como *background*:

```
<title>London Transport Disruptions LTD</title>
<style>
    body, html {
        height: 100%;
        margin: 0;
        font: 400 15px/1.8 "Lato", sans-serif;
        color: #777;
    }

    .bgimg-1 {
        position: relative;
        opacity: 0.65;
        background-position: center;
        background-repeat: no-repeat;
        background-size: cover;

    }
    .bgimg-1 {
        background-image: url("https://blog.printsome.com/wp-content/uploads/london-infographic-1.jpg");
        height: 100%;
    }

    .caption {
        position: absolute;
        left: 0;
        top: 50%;
        width: 100%;
        text-align: center;
        color: #000;
    }

    .caption span.border {
        background-color: #111;
        color: #fff;
        padding: 18px;
        font-size: 25px;
        letter-spacing: 10px;
    }

    h3 {
        letter-spacing: 5px;
        text-transform: uppercase;
        font: 20px "Lato", sans-serif;
        color: #111;
    }

</style>
```

Ilustración 92. Bloque head con el CSS de la página

Debido al tamaño que pueden tener los códigos CSS que hagamos en las diferentes plantillas, a partir de ahora se pondrá tan sólo un fragmento de este (el código completo se podrá acceder desde el repositorio) para así no llenar la memoria con estilos CSS. El estilo CSS en sí es bastante simple, utilizaremos una imagen de fondo de tamaño *cover* que ocupe todo el ancho y alto de la página. Posteriormente centraremos vertical y horizontalmente un *div* en el que pondremos un título sobre un pequeño recuadro gris para que se diferencie de la imagen.

Dentro del bloque <body></body> añadiremos el siguiente fragmento de código:

```
<body>

<div id="bgimg" class="bgimg-1">
    <div onclick="location.href='https://restapitims.herokuapp.com/disruptions';" class="caption">
        <span class="border">LONDON ROADS DISRUPTIONS</span>
    </div>
</div>
</body>
```

Ilustración 93. Bloque body del homepage

Básicamente consiste en dos *div*, uno dentro de otro. El *div* exterior para la imagen de fondo y en el interior mostraremos el título del proyecto. Como aspecto interesante aquí es la función *onclick(),* el cual disparará una redirección cuando hagamos clic dentro del *div* a */disruptions*, que, como pusimos en el routing, referencia a *index.html*, es decir la página principal donde mostraremos el mapa y los tweets.

Si queremos probar la aplicación para comprobar cómo se vería el homepage no basta con ejecutar la aplicación desde el IDE, debemos crear el main que se encargará de iniciar la aplicación, ya que este, todavía no lo hemos creado. Es muy sencillo, para ello vamos a *src > main > java > es.cicei.ulpgc (package)* y hacemos clic derecho sobre él desde el IDE *> New > Java Class* y pondremos como nombre *MainApplicationClass.java*. Una vez creado dentro del paquete añadimos este pequeño trozo de código:

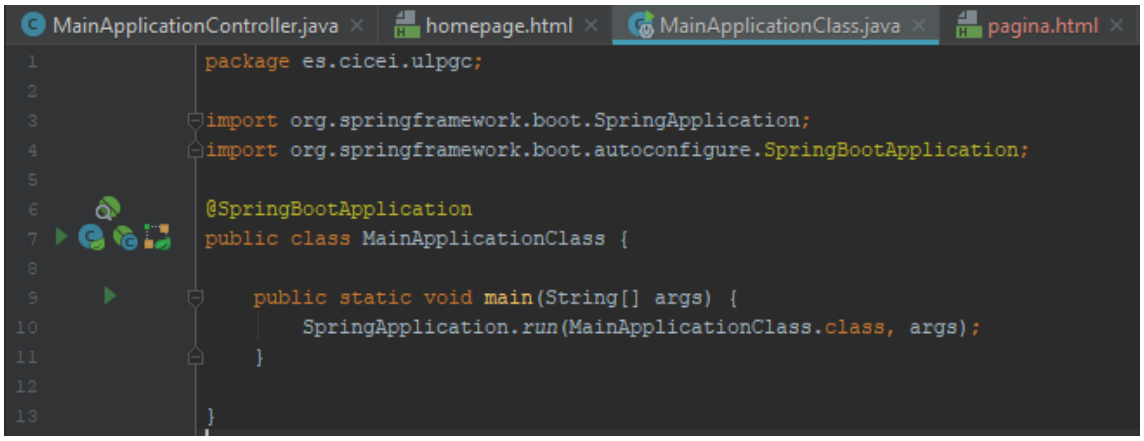


Ilustración 94. Método main de la aplicación web

En ese fragmento de código tan sólo importamos dos librerías relacionadas con spring Boot para indicar que estamos frente a un proyecto Spring mediante la anotación *@SpringBootApplication*. Posteriormente creamos el main de manera similar a Scala, pero esta vez usando la llamada *SpringApplication.run().*

Una vez creado el main y hayamos seguido todos los pasos anteriores de esta sección de manera correcta, ya seremos capaces de ejecutar la aplicación para hacer las primeras comprobaciones de que todo está en orden. Deberemos acceder al navegador y acceder a la raíz de la aplicación y, gracias al controlador del aplicativo, nos devolverá la página *homepage.html* tal y como definimos.

Spring Boot iniciará el servidor local (Tomcat) y escuchará en la siguiente dirección por defecto: http://localhost:8080/. Es probable que tarde unos pocos segundos en iniciar y esto es debido a que la aplicación debe de realizar la conexión a la base de datos. Procedemos de la siguiente manera:

1. Ejecutamos la aplicación desde el botón del play en la parte superior del IDE:

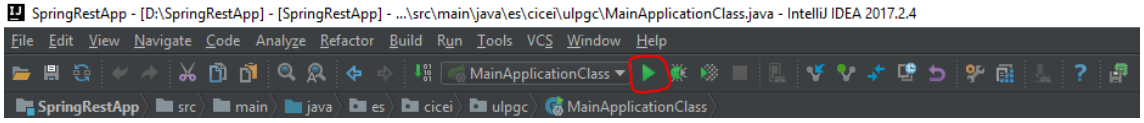


Ilustración 95. Botón run para ejecutar el programa

2. Posteriormente se ejecutará el programa. Esperaremos a que Spring Boot nos devuelva la URL donde está escuchando (tardará unos pocos segundos por el motivo antes mencionado):

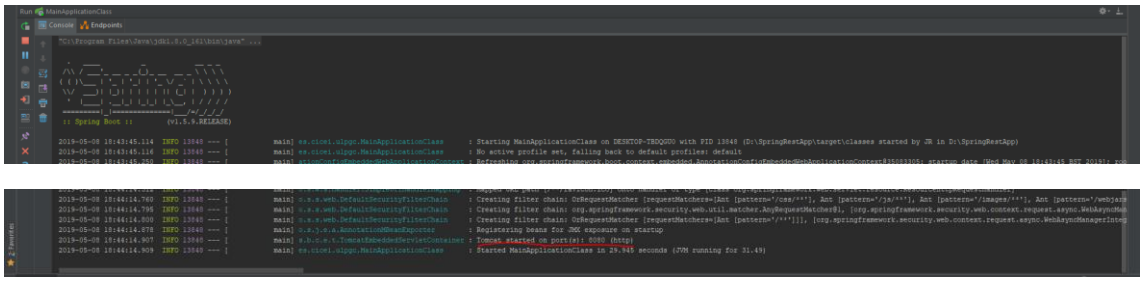


Ilustración 96. Ejecución de Spring Boot

En la imagen anterior se puede observar que el servidor Tomcat está escuchando en el puerto 8080.

3. Con cualquier navegador accedemos a la dirección recibida (http://localhost:8080/). Si todo va bien, nos devolverá la página tal y como se espera:

Ilustración 97. Resultado del homepage de la aplicación

Ya con esto tendríamos creada la primera página de nuestra aplicación web. Ahora podemos proceder con la siguiente, esta será *login.html*. Procedemos a crearla de la misma manera que con el homepage. Clic derecho sobre *src > main > resources > templates* y luego vamos a *New > HTML file.* Pondremos como nombre *login.html*. Abrimos el fichero y añadimos el CSS de la página. Podríamos crear un fichero CSS aparte con todas las clases y posteriormente importarla directamente al HTML, aunque en este caso el CSS lo pondré directamente dentro del HTML.

Ilustración 98. Bloque head del login

Como comenté antes, para no llenar la memoria con mucho CSS, tan sólo pondré una previa de cómo está presente. El código completo se puede ver directamente siguiendo este enlace: código de login.html.

Hay que señalar varias cosas importantes de la imagen anterior:

Realmente en el *homepage.html* no se usó nada de Bootstrap ni se aplicó Thymeleaf, por lo que no fue necesario importarlos. En esta página si se hará uso de ellos por lo que es necesario añadirlos en el código. De ahí la importancia de la sentencia *<link href="">* la cual usaremos para importar, mediante CDN, la hoja de estilo de Bootstrap. Por otra parte, para importar Thymeleaf a la plantilla usaremos: *xmlns:th="http://www.w3.org/1999/xhtml"* y *xmlns:sec=http://www.w3.org/1999/xhtml* dentro del bloque *<html>*. Con esto estamos importando las herramientas "*th*" y "*sec*" de Thymeleaf, que son las que usaremos en esta plantilla.

Dentro del body añadiremos el cuerpo del formulario de login:

```
<body>
<div>
    <div class="site-wrapper center-div">
        <div class="site-wrapper-inner">
            <div class="cover-container">
                <div class="inner cover" sec:authorize="isAnonymous()">
                    <h1 class="cover-heading" style="...">Login</h1>
                    <form th:action="@{/login}" method="post">
                        <div th:if="${param.error}">
                            <div class="alert alert-danger">
                                Invalid username or password.
                            </div>
                        </div>
                        <div th:if="${param.logout}">
                            <div class="alert alert-info">
                                You have been logged out.
                            </div>
                        </div>
                        <div class="form-group">
                            <label  class="col-sm-2 col-form-label" for="username">Username</label>
                            <input type="text"
                                   id="username"
                                   name="username"
                                   class="form-control form-control-lg"
                                   autofocus="autofocus"
                                   placeholder="Username"/>
                        </div>
                        <div class="form-group">
                            <label class="col-sm-2 col-form-label" for="password">Password</label>
                            <input type="password"
                                   id="password"
                                   name="password"
                                   class="form-control form-control-lg"
                                   placeholder="Password"/>
                        </div>
                        <div class="form-group" style="...">
                            <div class="row" style="...">
                                <div class="col-sm-6 col-sm-offset-3" style="...">
                                    <input type="submit"
                                           name="login-submit"
                                           id="login-submit"
                                           class="form-control btn btn-info btn btn-lg btn-primary btn-block"
                                           value="Log In"/>
                                </div>
                            </div>
                        </div>
                    </form>
                </div>
```

Ilustración 99. Bloque body del formulario login (1)

```
                <div sec:authorize="isAuthenticated()">
                    <h1 class="cover-heading">You are already Logged In!</h1>
                    <p class="lead">
                        <a th:href="@{/logout}" class="btn btn-lg btn-secondary">Logout</a>
                    </p>
                </div>

                <p><a href="/" th:href="@{/}">Back to home page</a></p>

                </div>
        </div>
    </div>
</div>
</body>
```

Ilustración 100. Bloque body del formulario login (2)

Las clases que aparecen en los dos fragmentos de código anterior provienen tanto de Bootstrap como del bloque de estilos que creamos en la misma página.

Es importante señalar varios aspectos relacionados con Thymeleaf en el código:

- Cuando un usuario rellena el formulario y lo manda, usando *th:action="@{/login}"* de Thymeleaf (podemos trabajar con rutas parciales), estos datos irán directamente al

controlador encargado de los POST *request* a /login, que, en este caso, no es el controlador que creamos nosotros (tan sólo hicimos el GET request en /login, no el POST). Este recurso lo gestiona Spring Security independientemente, el cual veremos más adelante.

- Con la función *th:if* podemos hacer consultas a determinadas variables como de un simple *if* de Java se tratase. Por ejemplo, en este caso, si param.logout (el cual es un booleano que está a true en caso de que el usuario haya cerrado sesión y false en caso de que no) devuelve true, mostrará un mensaje de advertencia. En caso de que esto no se cumpla, el bloque de código no se ejecutará.
- Con *th:ref* trabajamos exactamente igual que con el href de HTML, pero con la ventaja de que con Thymeleaf podemos trabajar con rutas parciales de tal manera que no tenemos que modificar las rutas a la hora de subir los cambios al servidor una vez hayamos terminado de trabajar en local.
- Por último, *sec:authorize="isAuthenticated()"* es una herramienta muy valiosa que nos sirve como un th:if pero que en este caso se consulta si hay un usuario *logueado* en memoria. En caso afirmativo, se ejecuta la acción del *logout*(), en caso contrario, el bloque de código no se mostrará al usuario, ya que, si no hay un usuario *logueado*, no es necesario mostrar el botón de cierre de sesión.

Esta página será accesible desde http://localhost:8080/login, pero la idea de esta página es que se muestre automáticamente una vez un usuario intente acceder a http://localhost:8080/disruptions, siempre y cuando este no esté ya logueado. Esta lógica la veremos en el apartado de Spring Security más adelante. La página se vería de la siguiente manera (simplemente un campo para el usuario y otro para la contraseña):

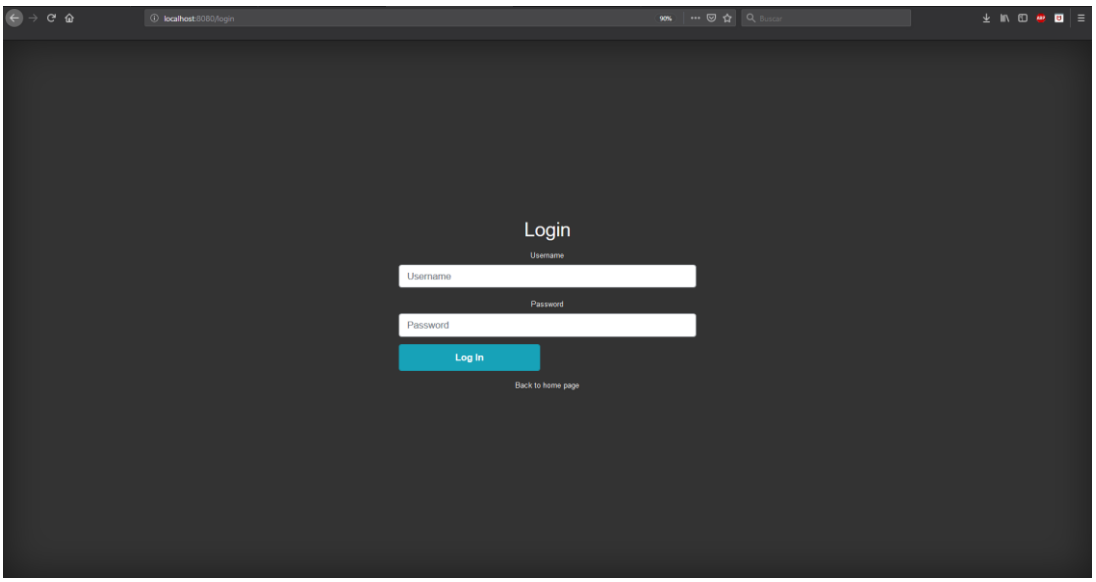


Ilustración 101. Resultado de la página de inicio de sesión

Ahora podemos realizar la última de las plantillas, posiblemente esta sea la más compleja de las 3 debido al tamaño del código, ya que la dificultad no aumenta respecto a lo que ya hemos visto.

Creamos la última plantilla de la misma manera de las anteriores: clic derecho sobre *templates > New > HTML file*. Esta la llamaremos *index.html*.

Lo primero que haremos es importar las librerías necesarias y el CSS que utilizaremos, ya sea de manera externa o mediante el bloque *<style></style>:*

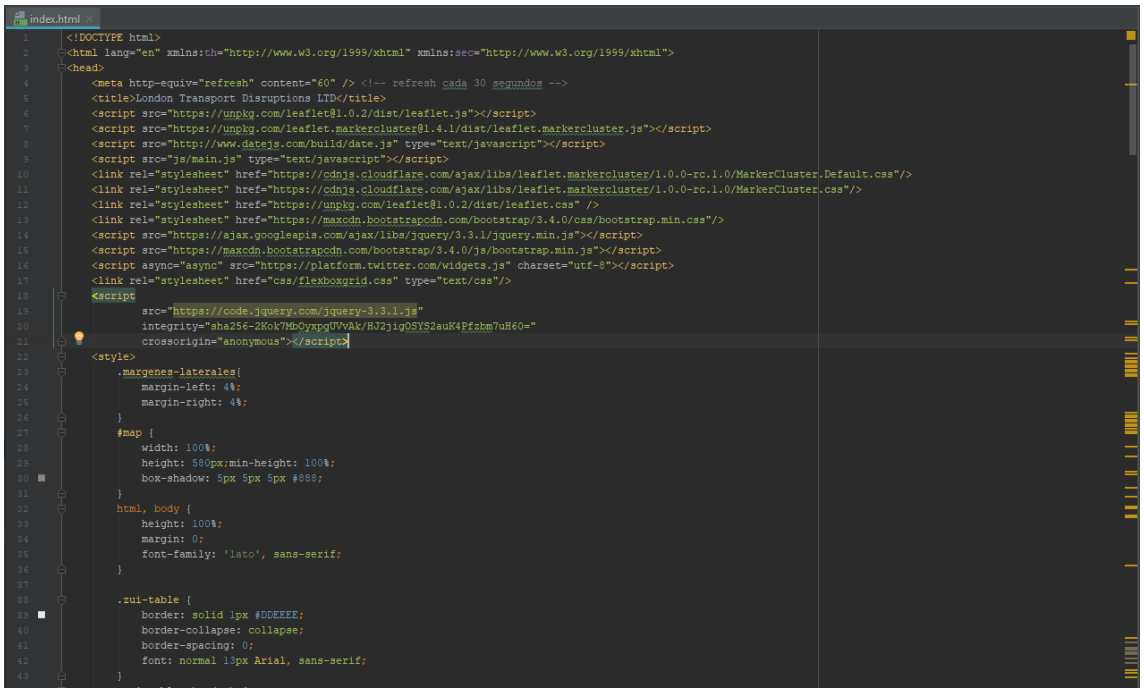


Ilustración 102. Bloque head de index.html

El código completo se puede consultar directamente desde el repositorio siguiendo el siguiente enlace: [index.html].

Las librerías/JS/CSS que hemos insertado son:

- Thymeleaf
- Leaflet
- Bootstrap
- JQuery
- Widget de Twitter
- Flexbox Grid (previamente descargado)

Hay un aspecto importante en el *head* que no aparece en las otras plantillas y es el contenido del bloque *<meta>.* Con *http-equiv="refresh" content="60"* se establece que la página se refresque o recargue cada 60 segundos en este caso, para que las nuevas incidencias y los nuevos tweets aparezcan automáticamente sin que el usuario tenga que recargar la página.

El código HTML pese a ser un poco extenso se puede dividir en varias partes muy diferenciadas:

- Navbar (barra de navegación)
- Mapa de incidencias

- Tabla de incidencias
- Zona de Twitter que incluye los Tweets de la base de datos y el widget de Twitter con el perfil oficial de TFL.
- Footer

Procedemos a explicar cada uno de estos bloques:

1. **Navbar o barra de navegación** la usaremos para movernos alrededor de las diferentes páginas y secciones que componen la aplicación. Tan sólo debemos de añadir dentro del bloque <body> el siguiente código:

```
<body>

<ul style="background-color: #333;">
    <li><a class="active" th:href="@{/disruptions}">TIMS</a></li>
    <li><a th:href="@{/}">Homepage</a></li>
    <li><a href="#map">Disruption map</a></li>
    <li><a href="#disruption-table">Disruption table</a></li>
    <li><a href="#tweets-feed">Tweets Feed</a></li>
    <li><a href="https://github.com/JuanRaBetancor/RestAPI-TIMS">Github</a></li>
    <li><a href="https://bitbucket.org/JuanraBO/sparkstreamingapp/">Bitbucket</a></li>
    <li><a sec:authorize="isAuthenticated()" th:href="@{/logout}">Logout</a></li>
    <li><a sec:authorize="isAnonymous()" th:href="@{/login}">Login</a></li>
</ul>
```

Ilustración 103. Navbar de index.html

En ella podemos ver que volvemos a utilizar las herramientas de Thymeleaf para los enlaces mediante *th:href* y utilizando rutas relativas. Además, también usamos *sec:authorize* para comprobar si el usuario ha iniciado sesión. En caso afirmativo, mostramos el enlace para cerrar sesión. En caso contrario, mostramos el enlace para iniciar sesión. Esto último carecerá de sentido en un futuro cuando configuremos Spring Security, ya que un usuario que no haya iniciado sesión en el sistema nunca podrá acceder a dicha página.

2. EL **mapa de incidencias** en su totalidad está gestionado mediante JavaScript. Leaflet tan sólo necesita un div cuyo "*id*" sea *map* como código HTML. Todo el trabajo será realizado mediante JavaScript, en el cual se enlazará el mapa de OpenStreetMap y añadiremos todas las incidencias. Para ello creamos el div de la siguiente manera:

```
    <li><a sec:authorize="isAnonymous()" th:href="@{/login}">Login</a></li>
</ul>

<div id="map"></div>
```

Ilustración 104. DIV donde se incluirá el mapa

Y luego mediante JavaScript, lo crearemos todo. Añadiremos antes de acabar el bloque body, un bloque <script></script> en el que añadiremos lo siguiente (Aquí se puede acceder al código fuente completo):

```
<script type="text/javascript" th:inline="javascript">

    var LeafIcon = L.Icon.extend({
        options: {
            iconSize:     [38, 40],
            iconAnchor:   [22, 94],
            popupAnchor:  [-3, -76]
        }
    });

    var disruptionsMarkerActive = new LeafIcon({
        iconUrl: 'https://www.freeiconspng.com/uploads/warning-icon-md-png-4.png'
    });

    var disruptionsMarkerLTA = new LeafIcon({
        iconUrl: 'https://cdn1.iconfinder.com/data/icons/ui-navigation-1/152/alert-512.png'
    });


    var map = L.map('map').
    setView([51.5000,  -0.1333], 12);

    L.tileLayer('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        attribution: 'Map data &copy; <a href="http://openstreetmap.org">OpenStreetMap</a> contributors, <a href="http://creativecommons.org/licenses/by-sa/2.0/">
        maxZoom: 18
    }).addTo(map);

    L.control.scale().addTo(map);
    L.marker([41.66, -4.71], {draggable: true}).addTo(map);
```

Ilustración 105. Código JavaScript para configurar el mapa (1)

Hay que comentar varias cosas para que se entienda bien el código:

- Importante indicar *th:inline="javascript"* para que Thymeleaf reconozca el script.
- Lo primero que haremos será crear un objeto llamado *LeafIcon* que extiende del objeto *Icon* (objeto que referencia los *markers* de Thymeleaf). Tan sólo establecemos el largo, el ancho y el tamaño del cuadro de diálogo cuando apretemos el icono. Estos iconos tan sólo son las imágenes que representarán cada incidencia.
- Posteriormente creamos dos iconos de tipo LeafIcon, el objeto que acabamos de crear. La diferencia entre los dos iconos tan sólo es la imagen, Uno de ellos representará las incidencias de tipo *Active* y, el otro, las de tipo *Active Long Term*.

Ilustración 106. Iconos para las incidencias "Long Term Active" y "Active" respectivamente

- Luego sigue uno de los pasos más importante que es la creación en sí del mapa haciendo la llamada a OpenStreetMap. Con *L.map('map').setView()* creamos el mapa en el div cuyo "*id*" es el indicado (el que creamos anteriormente) e indicamos las coordenadas por defecto y el nivel de zoom inicial. En este caso hemos indicado las coordenadas de la ciudad de Londres. Con *titleLayer()* creamos el "pie de página" del mapa

Ilustración 107. Layer del mapa

- *L.control.scale()* no es más que un control que aparece en la esquina superior izquierda del mapa que nos permitirá alejar o acercar la vista del mapa (zoom).

Ahora que tenemos el mapa creado, podemos añadir todas las incidencias que hemos pasado como parámetro desde el controlador (variable *disruptions*) utilizando los iconos creados:

```
var data = [[${disruptions}]];
if(data!=null) {
    data.forEach(function (element) {
        var lon = element.lon;
        var lat = element.lat;
        var stdate = new Date(element.startingtime);
        var endate = new Date(element.endtime);
        var stdateformat = stdate.toString("dd/MM/yyyy - HH:mm");
        var enddateformat = endate.toString("dd/MM/yyyy - HH:mm");
        var popupText = "<ul><li><strong>Location: </strong>" + element.location + "</li><li><strong>Status: </strong>"+element.status+"</li>" +
            "<li><strong>Severity: </strong>"+element.severity+"</li><li><strong>Category: </strong>"+element.category+"</li>" +
            "<li><strong>Prevision: </strong>"+stdateformat+" - "+ enddateformat+"</li><li><strong>Comments: </strong>"+element.comments+"</li>" +
            "<li><strong>Current Update: </strong>"+element.currentupdate+"</li></ul>";

        var markerLocation = new L.LatLng(lat, lon);
        if(element.status == "Active"){
            var marker = new L.Marker(markerLocation, {icon: disruptionsMarkerActive});
        }else{
            var marker = new L.Marker(markerLocation, {icon: disruptionsMarkerLTA});
        }

        map.addLayer(marker);
        marker.bindPopup(popupText);
    });
}
```

Ilustración 108. Código JavaScript para configurar el mapa (2)

La explicación es la siguiente:

- Creamos una variable *data* que incluirá la lista de incidencias que pasamos del controlador a través del objeto *disruptions*.
- Si la lista no está vacía la recorremos. Por cada elemento de la lista, es decir, por cada incidencia, almacenaremos los atributos de esta en diferentes variables para una mejor comprensión, formateamos las fechas y crearemos un texto, donde mostraremos toda la información de la correspondiente incidencia (*popupText*).
- Creamos un marcador y establecemos la localización de este. La localización de la latitud y la longitud la obtendremos de los atributos de la incidencia y mediante la llamada al método *LatLng()* establecemos las coordenadas.
- Comprobamos de qué tipo de incidencia es la incidencia actual, dependiendo de si es *Active* o *Long Term Active*, le asignaremos al marcador un icono de un tipo o de otro, de los que creamos anteriormente.
- Por último, añadiremos el marcador al mapa y el mensaje de dialogo o *PopUp* al marcador y así sucesivamente con cada incidencia.

3. La **tabla de incidencias**, como su nombre indica es una simple tabla donde mostraremos todas las incidencias de manera que se puedan consultar estas sin necesidad de acceder al mapa. Utilizando Thymeleaf, no hay complicación ninguna:

```
<div id="disruption-table" class="margenes-laterales row">
    <table th:if="${not #lists.isEmpty(disruptions)}" style="margin-top: 50px" class="zui-table">
        <thead class="table100-head">
            <th class="column1" th:text="${'Nº Disruptions: ' + #lists.size(disruptions)}"></th>
            <th class="column1">Search</th>
            <th class="column1"><input id="searchInput" class="myInput" value="Type To Filter"/></th>
            <th class="column1"></th>
            <th class="column1"></th>
            <th class="column1"></th>
            <th class="column1"></th>
            <th class="column1"></th>
            <th class="column1"></th>
        </thead>
        <thead class="table100-head">
        <tr>
            <th class="column1"> Location </th>
            <th class="column1"> Status </th>
            <th class="column1"> Severity </th>
            <th class="column1"> Category </th>
            <th class="column1"> Starting Time </th>
            <th class="column1"> End Time </th>
            <th class="column1"> Comments </th>
            <th class="column1"> Current Update </th>
            <th class="column1"> Coordinates </th>
        </tr>
        </thead>
        <tbody id="fbody">
        <tr th:each="disruption : ${disruptions}">
            <td class="column1"><span th:text="${disruption.location}"> Location </span></td>
            <td class="column1"><span th:text="${disruption.status}"></span></td>
            <td class="column1"><span th:text="${disruption.severity}"></span></td>
            <td class="column1"><span th:text="${disruption.category}"></span></td>
            <td class="column1"><span th:text="${disruption.startingtime}"></span></td>
            <td class="column1"><span th:text="${disruption.endtime}"></span></td>
            <td class="column1"><span th:text="${disruption.comments}"></span></td>
            <td class="column1"><span th:text="${disruption.currentupdate}"></span></td>
            <td class="column1"><span th:text="${disruption.lat + ' ' + disruption.lon}"></span></td>
        </tr>
        </tbody>
    </table>
</div>

<div style="margin-top: 25px" th:if="${#lists.isEmpty(disruptions)}" class="alert alert-warning margenes-laterales">
    <strong>Warning!</strong> No disruptions available!
</div>

<div style="margin-top: 25px" th:if="${#lists.isEmpty(tweets)}" class="alert alert-warning margenes-laterales">
    <strong>Warning!</strong> No tweets available!
</div>
```

Ilustración 109. Código de la tabla de incidencias

Hay varias cosas para tener en cuenta:

- Tan sólo se mostrará la tabla en caso de que la lista recibida desde el controlador no esté vacía: *th:if="${not #lists.isEmpty(disruptions)}"*
- Crearemos dos cabeceras para la tabla, en el primero indicaremos el número de incidencias que aparecen en la tabla (*th:text="${'Nº Disruptions: ' + #lists.size(disruptions)}"*) junto con un filtro para buscar incidencias por palabras clave, utilizando para ello JQuery. En la segunda cabecera tan sólo mostraremos el nombre de cada columna.
- Con el uso de *th:each="disruption : ${disruptions}"* es muy sencillo recorrer la lista de incidencia para mostrarla. Mostraremos por columna cada atributo de cada incidencia.
- Por último, en caso de advertencia se han añadido dos *warnings* de Bootstrap en caso de que ambas listas estén vacías (al usar persistencia de datos con la base de datos, estos mensajes no deberían de ser visibles a excepción de alguna pérdida de datos que haya ocurrido).

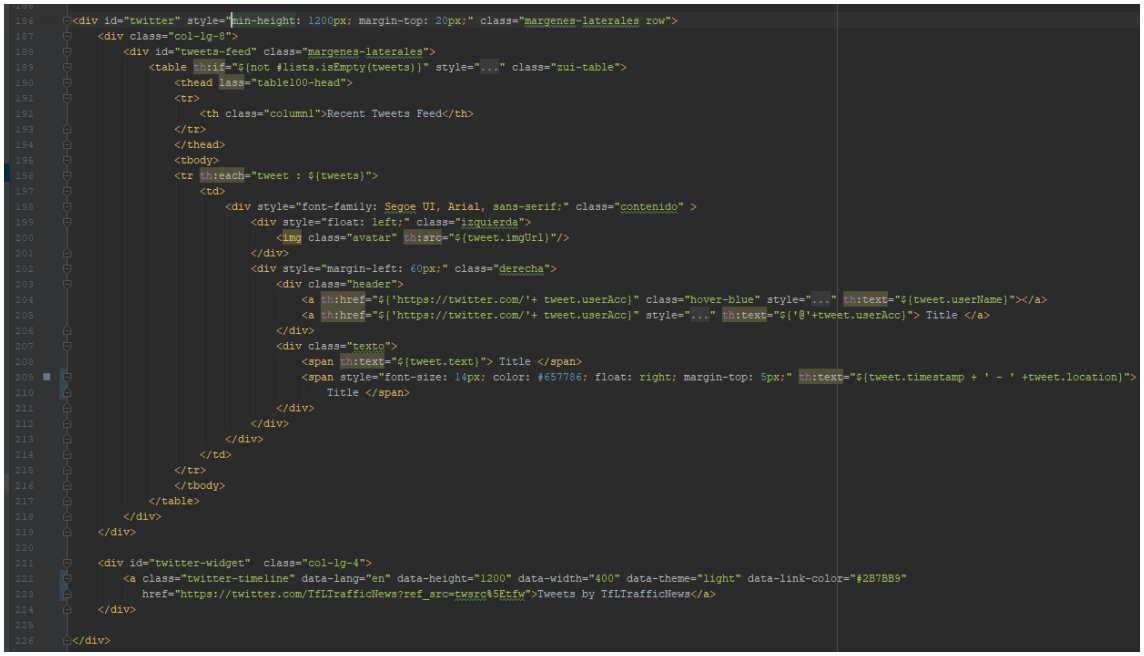
El código JQuery que gestiona el filtro es el siguiente, lo pondremos seguido del anterior script (el JavaScript que controla el mapa):

```
    $("#searchInput").keyup(function () {
        //split the current value of searchInput
        var data = this.value.split(" ");
        //create a jquery object of the rows
        var jo = $("#fbody").find("tr");
        if (this.value == "") {
            jo.show();
            return;
        }
        //hide all the rows
        jo.hide();

        //Recusively filter the jquery object to get results.
        jo.filter(function (i, v) {
            var $t = $(this);
            for (var d = 0; d != data.length; ++d) {
                if ($t.is(":contains('" + data[d] + "')")) {
                    return true;
                }
            }
            return false;
        })
        //show the rows that match.
            .show();
    }).focus(function () {
        this.value = "";
        $(this).css({
            "color": "black"
        });
        $(this).unbind('focus');
    }).css({
        "color": "#C0C0C0"
    });
</script>
```

Ilustración 110. Filtro de búsqueda de la tabla

Se separa por espacios todas las palabras que pongamos en el campo y buscamos por cada fila de la tabla (y cada columna) si hay alguna coincidencia. En caso de que haya coincidencia, la fila se sigue mostrando, en caso de que no, las fila se oculta.

4. En la denominada **Zona de Twitter** mostraremos en una tabla similar a la anterior (de tan sólo una columna) todos los tweets guardados en la base de datos. Como mejora, se ha añadido de la API de Twitter un widget con el perfil y los tweets de la cuenta oficial de TFL:

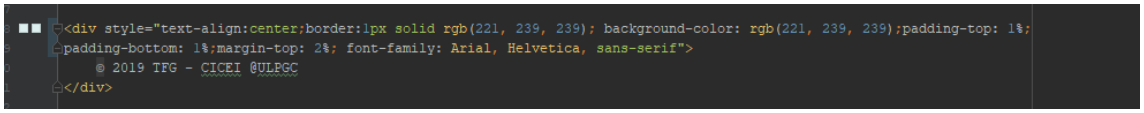


```
<div id="twitter" style="min-height: 1200px; margin-top: 20px;" class="margenes-laterales row">
    <div class="col-lg-8">
        <div id="tweets-feed" class="margenes-laterales">
            <table th:if="${not #lists.isEmpty(tweets)}" style="..." class="zui-table">
                <thead lass="table100-head">
                <tr>
                    <th class="column1">Recent Tweets Feed</th>
                </tr>
                </thead>
                <tbody>
                <tr th:each="tweet : ${tweets}">
                    <td>
                        <div style="font-family: Segoe UI, Arial, sans-serif;" class="contenido" >
                            <div style="float: left;" class="izquierda">
                                <img class="avatar" th:src="${tweet.imgUrl}"/>
                            </div>
                            <div style="margin-left: 60px;" class="derecha">
                                <div class="header">
                                    <a th:href="${'https://twitter.com/'+ tweet.userAcc}" class="hover-blue" style="..." th:text="${tweet.userName}"></a>
                                    <a th:href="${'https://twitter.com/'+ tweet.userAcc}" style="..." th:text="${'@'+tweet.userAcc}"> Title </a>
                                </div>
                                <div class="texto">
                                    <span th:text="${tweet.text}"> Title </span>
                                    <span style="font-size: 14px; color: #657786; float: right; margin-top: 5px;" th:text="${tweet.timestamp + ' - ' +tweet.location}">
                                        Title </span>
                                </div>
                            </div>
                        </div>
                    </td>
                </tr>
                </tbody>
            </table>
        </div>
    </div>

    <div id="twitter-widget"  class="col-lg-4">
        <a class="twitter-timeline" data-lang="en" data-height="1200" data-width="400" data-theme="light" data-link-color="#2B7BB9"
           href="https://twitter.com/TfLTrafficNews?ref_src=twsrc%5Etfw">Tweets by TfLTrafficNews</a>
    </div>

</div>
```

Ilustración 111. Bloque destinado a mostrar los Tweets

Consiste en un div que a su vez incluye dos separados verticalmente. En el primero de ellos siempre y cuando la lista no esté vacía, mostraremos toda la información de cada tweet (*th:each="tweet : ${tweets}"),* junto al usuario, imagen de perfil, fecha, localización, etc. Se enlazará el nombre de usuario a su cuenta real de Twitter por si se quiere consultar su perfil completo. En el segundo tan sólo haremos la llamada al widget de Twitter indicando que cuenta de Twitter queremos mostrar, en este caso, la de TFL.

5. Por último, tenemos el footer, donde mostraremos información relacionada con el proyecto, para delimitar el final de la página.



```
<div style="text-align:center;border:1px solid rgb(221, 239, 239); background-color: rgb(221, 239, 239);padding-top: 1%;
padding-bottom: 1%;margin-top: 2%; font-family: Arial, Helvetica, sans-serif">
    © 2019 TFG - CICEI @ULPGC
</div>
```

Ilustración 112. Footer de index.html

El resultado cuando accedemos a la página es el siguiente (con incidencias y tweets almacenados previamente en la base de datos como prueba, para no mostrar el mapa y las tablas vacías):

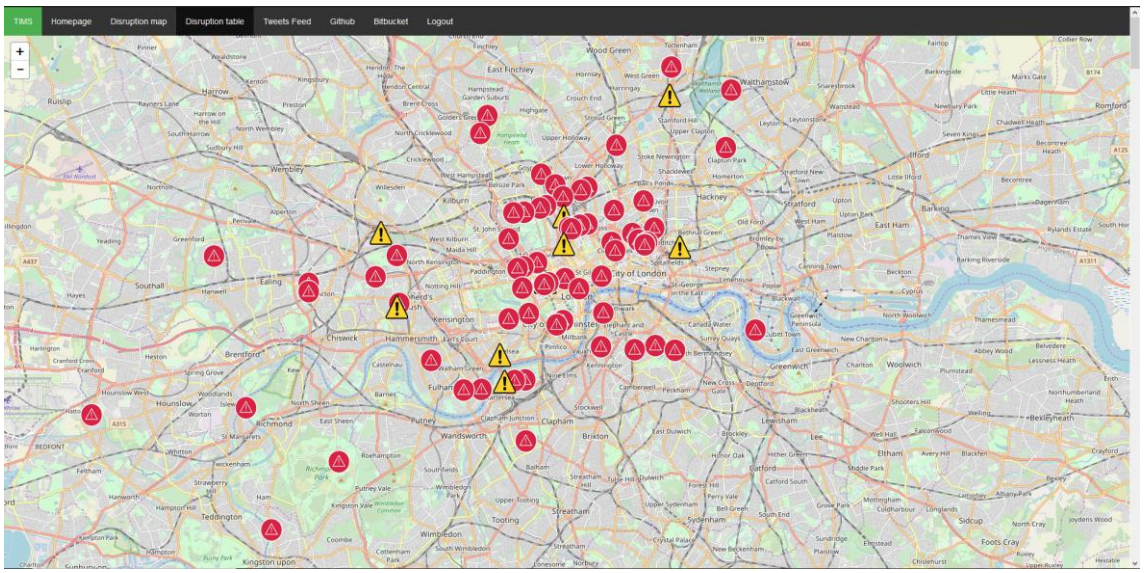


Ilustración 113. Mapa completo con incidencias

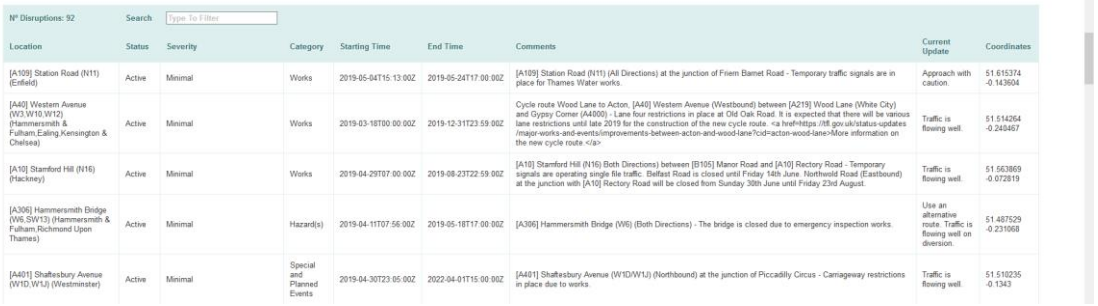


Ilustración 114. Tabla de incidencias completa

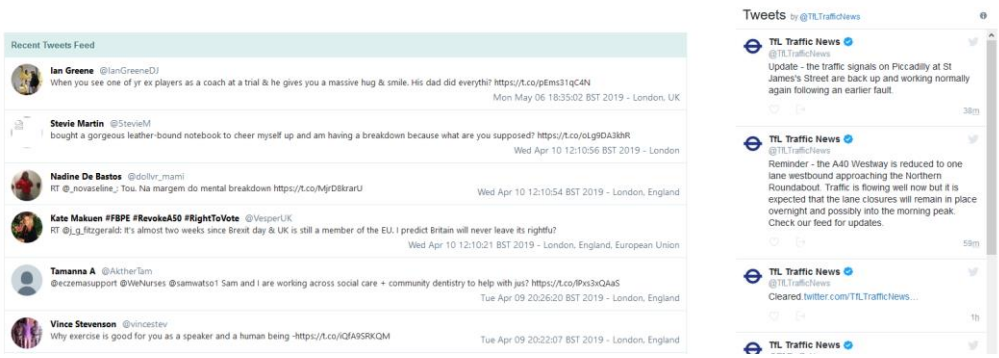


Ilustración 115. Lista de Tweets finalizada

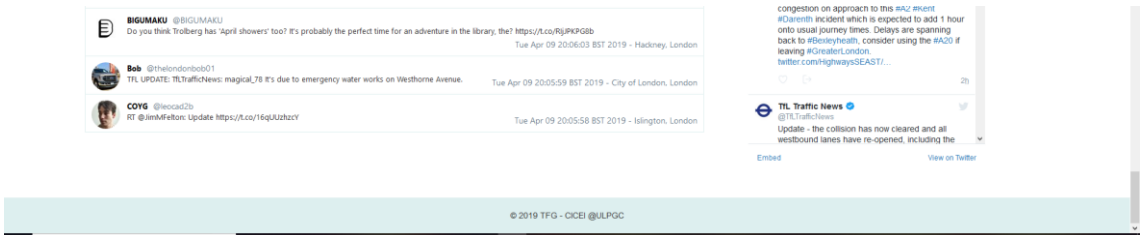


Ilustración 116. Footer finalizado

4.4.4. Configurando la seguridad de la aplicación (Spring Security).

Durante el transcurso de la creación de la aplicación web he nombrado en varias ocasiones la librería de Spring Security o en su defecto, me he referido a la seguridad de la aplicación en general. Es momento de configurar la seguridad de la aplicación para que cada usuario, dependiendo de su rol, pueda acceder a determinados recursos de la web. Con este bloque podemos dar por terminado la aplicación web a falta, por supuesto, del despliegue a Heroku.

Spring Security no es más que una herramienta que nos provee mecanismos de autorización y autenticación. Ya hemos hecho un primer contacto con Spring Security, ya que hemos creado la ruta /login, donde nos mostrará el formulario de inicio de sesión que también hemos hecho.

Lo primero que tendríamos que hacer es indicar en el *pom.xml* las dependencias, pero, como lo hicimos al principio, no hace falta hacerlo de nuevo. Spring Security necesitará un fichero de configuración donde estableceremos todos los criterios de seguridad de nuestra aplicación. Dentro del package de nuestra aplicación, haremos clic derecho *> New > Package*, pondremos como nombre *config*. Dentro de este directorio/package crearemos el fichero de configuración: clic derecho sobre dicho directorio *> New > Java Class*, lo nombraremos como *SpringSecurityConfig.java*. Abrimos el fichero recién creado y añadimos lo siguiente:

SpringSecurityConfig.java | MyAccessDeniedHandler.java

Match Case | Regex | Words

```
package es.cicei.ulpgc.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.web.access.AccessDeniedHandler;

@Configuration
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AccessDeniedHandler accessDeniedHandler;

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.csrf().disable()
                .authorizeRequests()
                .antMatchers( ...antPatterns: "/").permitAll()
                .antMatchers( ...antPatterns: "/disruptions").hasAnyRole( ...roles: "USER")
                .antMatchers(HttpMethod.POST, ...antPatterns: "/").permitAll()
                .antMatchers(HttpMethod.POST, ...antPatterns: "/tweets").permitAll()
                .anyRequest().authenticated()
                .and()
                .formLogin()
                .loginPage("/login")
                .defaultSuccessUrl( defaultSuccessUrl: "/disruptions", alwaysUse: true)
                .permitAll()
                .and()
                .logout()
                .permitAll().and()
                .exceptionHandling().accessDeniedHandler(accessDeniedHandler);
    }


    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

        auth.inMemoryAuthentication()
                .withUser( username: "user").password("user").roles("USER");
    }


}
```

Ilustración 117. Configuración Spring Security

En el trozo de código anterior tenemos lo siguiente:

- Lo primero es importar los diferentes métodos y anotadores que vamos a utilizar. La creación de la clase debe de estar bajo el anotador *@Configuration*, ya que estamos frente a una clase de configuración de Spring y no un controlador, y este tiene que extender de la interfaz *WebSecurityConfigureAdapter*, el cual establece que métodos debemos desarrollar (*método configure()*).
- Crearemos un atributo de clase que hace referencia a un gestor de accesos denegados que crearemos nosotros mismos para manejar los accesos a los diferentes recursos, de tal manera que si un usuario con su rol intenta acceder a un recurso en el cual no tiene permisos, la aplicación le mostrará la plantilla con el error 403. Como ya hemos visto, el uso del anotador *@Autowired* nos permite obviar los getters y setters del atributo. Este gestor lo crearemos posteriormente.
- Ahora toca poner toda la lógica de la seguridad bajo el método *configure*(). La explicación es la siguiente:
    - Con *http.csrf().disable()*, prohibimos las peticiones de tipo CSRF (*Cross-site request forgery*), de tal manera que evitamos los posibles ataques basados en el robo de sesiones.
    - Con *authorizeRequests*() permitimos que se puedan hacer peticiones a la aplicación pero, con los métodos *andMatches*(), establecemos las reglas de validación de la ruta indicada (en caso de que se haga una petición a una ruta la cual no exista, es decir, no está establecida en ningún controlador, automáticamente Spring muestra un error 404 y busca una plantilla con nombre *404.html* bajo el directorio */resources/templates/error* y, si esta existe, la muestra), por lo tanto tenemos lo siguiente: permitimos que cualquier usuario, autenticado o no, acceda a la raíz de la aplicación, en este caso *homepage*. Permitimos que tan sólo los usuarios cuyo rol sea USER y estén autenticados (*anyRequest().authenticated()*) puedan acceder a */disruptions* (donde está el mapa y los tweets). También permitimos que cualquier usuario pueda hacer peticiones POST a la raíz de la aplicación y a */tweets*, ya que en uno recibiremos las incidencias y en el otro los tweets desde la aplicación de Scala.
    - Con *formLogin().loginPage("/login")* indicamos que la plantilla encargada del formulario de inicio de sesión es *login.html*, estando esta como siempre bajo el directorio *resources > templates*.
    - Con *defaultSuccessUrl("/disruptions", true)* estamos indicando que siempre que el inicio de sesión sea un éxito, la aplicación redirija directamente a */disruptions*, es decir, se muestre el mapa automáticamente.
    - Por último, con *exceptionHandling().accessDeniedHandler(accessDeniedHandler*) indicamos que usaremos un gestor de accesos personalizado que crearemos posteriormente para gestionar los accesos denegados.
- Por último, aparece un método *configureGlobal()* en el cual establecemos los usuarios que existen "creados" en el sistema. Realmente tan sólo existen creados en memoria, no hay ninguna tabla en la base de datos con los usuarios registrados en la aplicación, sino que se crea artificialmente uno para realizar las pruebas. Dicho usuario tiene como nombre *user* y es de tipo (rol) USER. Sería una buena idea en caso de que se quiera escalar la aplicación, crear una tabla en la base de datos con los usuarios registrados y, a su vez, crear un formulario de registro.

Una vez tengamos el fichero de configuración terminado, falta por hacer dos cosas: el gestor de accesos y las plantillas de los errores 403 y 404. Procedemos primero a crear rápidamente el gestor, para ello crearemos otro directorio bajo el *package*, como ya hicimos con el fichero de configuración: clic derecho sobre el package *> New > Package* y pondremos como nombre *error*. Una vez creados el directorio, clic derecho sobre el directorio recién creado *> New > Java Class*, lo llamaremos *MyAccessDeniedHandler.java*. Lo abrimos y copiamos lo siguiente:

SpringSecurityConfig.java | MyAccessDeniedHandler.java

```
package es.cicei.ulpgc.error;



import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.access.AccessDeniedHandler;
import org.springframework.stereotype.Component;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;


@Component
public class MyAccessDeniedHandler implements AccessDeniedHandler {

    private static Logger logger = LoggerFactory.getLogger(MyAccessDeniedHandler.class);

    @Override
    public void handle(HttpServletRequest httpServletRequest,
                       HttpServletResponse httpServletResponse,
                       AccessDeniedException e) throws IOException, ServletException {

        Authentication auth
                = SecurityContextHolder.getContext().getAuthentication();

        if (auth != null) {
            logger.info("User '" + auth.getName()
                    + "' attempted to access the protected URL: "
                    + httpServletRequest.getRequestURI());
        }

        httpServletResponse.sendRedirect( s: httpServletRequest.getContextPath() + "/403");

    }
}
```

Ilustración 118. Gestor de accesos denegados

El código es muy sencillo:

- Importamos las utilidades que necesitamos e indicaremos que la clase es de tipo *@Component* e implementa la clase *AccessDeniedHandler*.
- Crearemos un objeto *Logger* de la clase y creamos un método *handle*() en el cual introduciremos la lógica en caso de que un usuario acceda a un recurso del cual no disponga de permisos:
    - Con el método *getAuthentication*() accedemos al usuario de la sesión que está actualmente abierta.
    - Preguntamos por dicho usuario, si ha iniciado sesión (es diferente de null), accedemos al nombre y formamos el mensaje de Log que se mostrará y

posteriormente redireccionaremos a la página *403.html* (bajo el directorio */resources/templates/error*) donde se mostrará el mensaje de error.

Una vez ya tenemos hecho el gestor de accesos denegados podemos crear las dos plantillas de error, el error 403 y 404. Como ya hemos señalado, Spring Security busca las plantillas dentro del directorio *resources > templates > error*, por lo que tenemos que crearla: clic derecho sobre *resources > templates* y vamos a *New > Directory*, lo llamaremos *error*. Una vez creado el directorio crearemos primero, por ejemplo, la plantilla del error 403: clic derecho sobre el directorio recién creado *> New > HTML file*, lo llamaremos *403.html*. Abrimos y añadimos lo siguiente:

Ilustración 119. Bloque head de 403.html

Tan sólo en el bloque *head* añadimos los CSS que usaremos en la plantilla. Se puede consultar el código completo con todos los estilos aquí. Ahora añadimos el bloque del *body*:

```
<body>
<div class="site-wrapper">

    <div class="site-wrapper-inner">

        <div class="cover-container">

            <div class="inner cover" style="...">
                <h1 class="cover-heading" style="...">403 - Access is denied</h1>
                <div th:inline="text" style="...">Hello '[[${#httpServletRequest.remoteUser}]]',
                    you do not have permission to access this page.</div>
            </div>

            <div id="bgimg" class="bgimg-1">
                <div onclick="location.href='https://restapitims.herokuapp.com/';" class="caption">
                    <span class="border">GO HOMEPAGE</span>
                </div>
            </div>

        </div>
    </div>
</div>
```

Ilustración 120. Bloque Body de 403.html

Es muy sencillo. Tan sólo mostraremos un mensaje de error 403 de acceso denegado y añadiremos un botón para regresar al *home*. Con la función *httpServletRequest.remoteUser* podemos acceder al nombre del usuario de la sesión actual. En este caso, tan sólo tenemos un usuario cuyo rol es USER y el único recurso protegido es */disruptions* (los usuarios cuyo rol sea USER pueden acceder a dicho recurso), por lo que, si iniciamos sesión, podremos acceder a todas las páginas y este mensaje no se mostrará en ningún caso. En el caso de que tengamos otro rol dentro de la aplicación, esta plantilla cobrará mucha importancia.

Para realizar la plantilla del error 404 realizamos exactamente el mismo proceso que con el anterior con el siguiente contenido:

Ilustración 121. Bloque head de 404.html

Pondremos en el *header* exactamente el mismo CSS que con la plantilla anterior. Se puede consultar el código fuente completo aquí. El cuerpo *html* es prácticamente igual, tan sólo se cambia el mensaje mostrado, que ahora será un error 404:

```
<body>
<div class="site-wrapper">

    <div class="site-wrapper-inner">

        <div class="cover-container">

            <div class="inner cover" style="...">
                <h1 class="cover-heading" style="...">404 - Page not found</h1>
                <div th:inline="text" style="...">It seems like you're exploring an unknown place!</div>
            </div>

            <div id="bgimg" class="bgimg-1">
                <div onclick="location.href='https://restapitims.herokuapp.com/';" class="caption">
                    <span class="border">GO HOMEPAGE</span>
                </div>
            </div>

    </div>
</div>
</div>


</body>
```

Ilustración 122. Bloque body de 404.html

Dando como resultado lo siguiente:

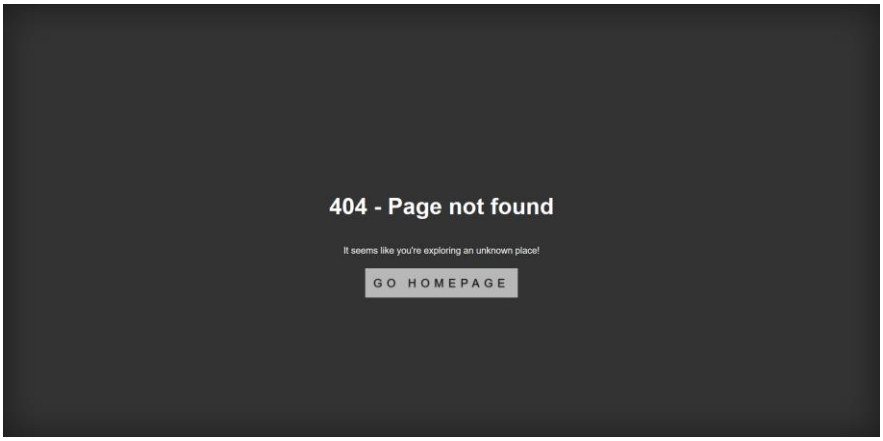


Ilustración 123. Página 404 terminada

Ya tendríamos completo el proyecto web, ahora podemos realizar el despliegue en Heroku. La estructura final del proyecto deberá quedar de la siguiente manera:

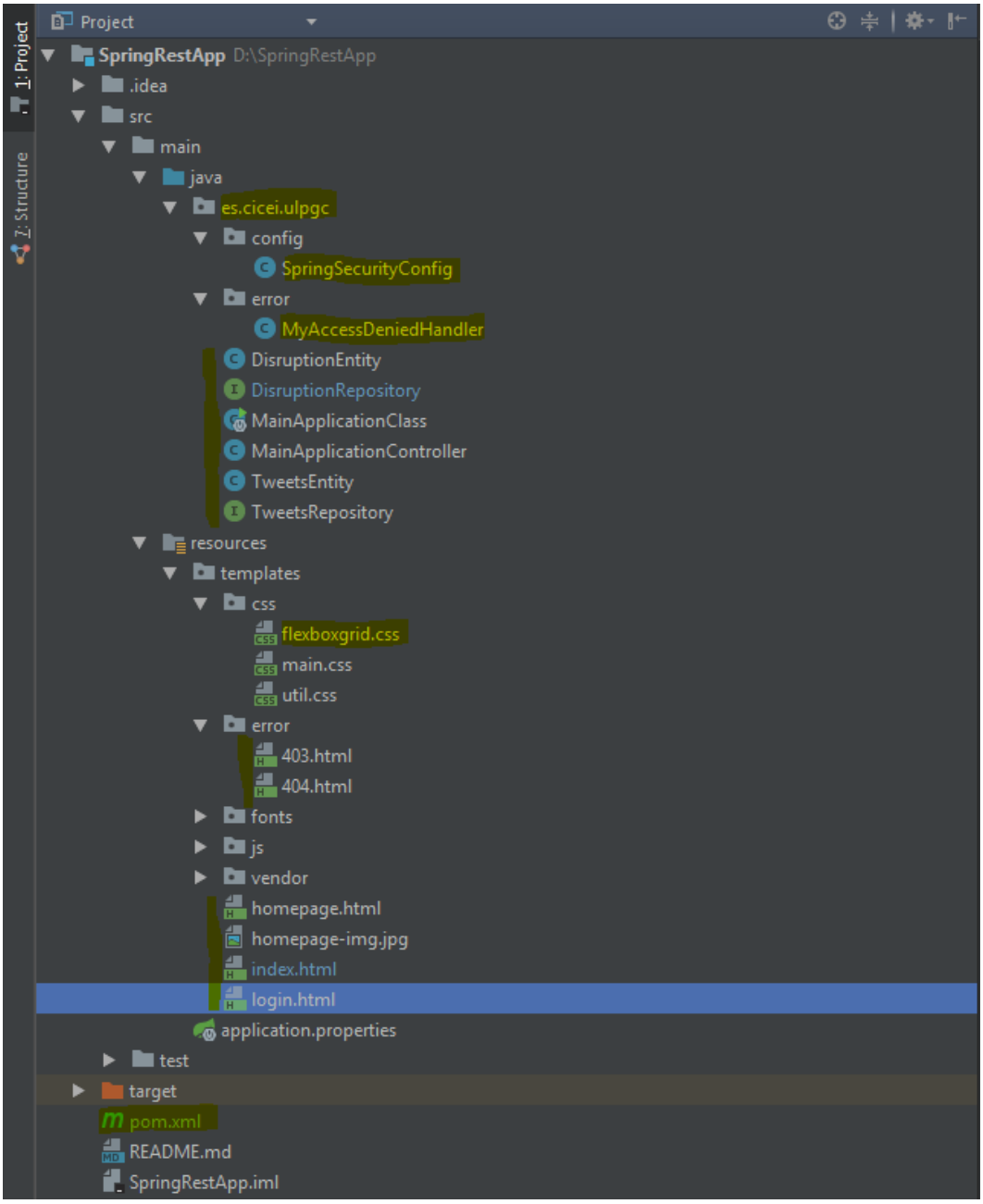


Ilustración 124. Estructura final del proyecto

Se han resaltado todos los ficheros que hemos creado en el proceso o los que de alguna manera hemos modificado en algún momento.

4.4.5. Despliegue a Heroku

Para desplegar la aplicación terminada a Heroku debemos de acceder nuevamente a nuestra cuenta de Heroku y entrar dentro de la aplicación. Posteriormente, accedemos a la pestaña "*Deploy*", ahí podemos ver los diferentes métodos de despliegue que soporta Heroku y los pasos a seguir:

- Heroku CLI
- GitHub
- Container Registry

Yo usaré el Heroku CLI para subir el proyecto. Si seguimos los pasos que aparecen, lo primero que tenemos que hacer es descargar el Heroku CLI siguiendo el enlace que aparece. Heroku CLI no es más que una herramienta que se integra dentro de la consola del sistema o CMD que nos ofrece una serie de comandos específicos para subir los ficheros.

Una vez instalado, abrimos una ventana de CMD con permisos de administrador. Accedemos al directorio donde se encuentra la aplicación mediante la utilidad *cd /directorio.*

Posteriormente seguimos lo pasos que nos aparecen en la página:

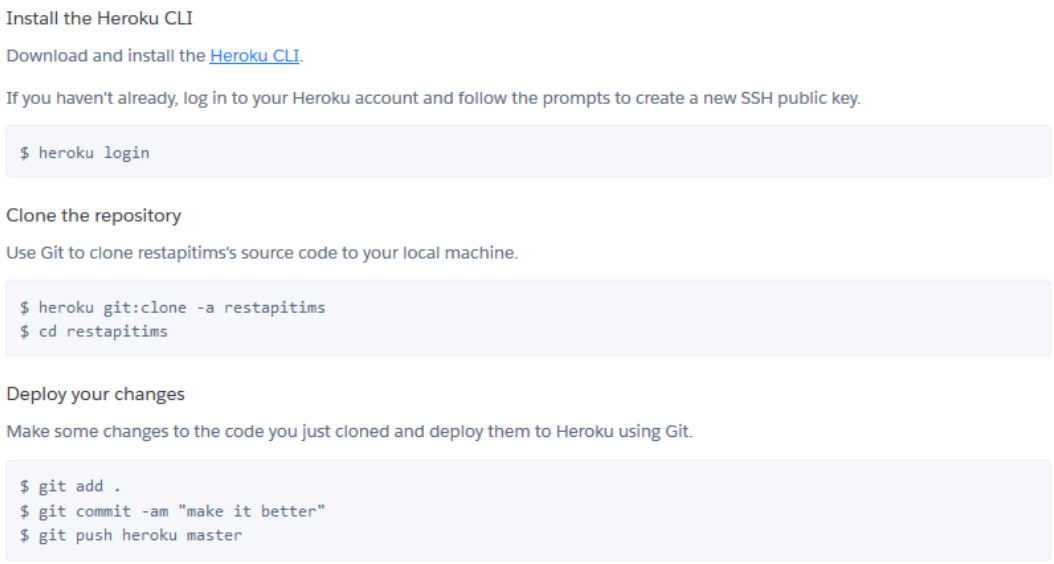


Ilustración 125. Pasos para hacer el despliegue en Heroku

1. Iniciamos sesión con H*eroku login* desde el CMD. Nos pedirá que apretemos una tecla distinta de la "Q". Una vez lo hagamos se nos abrirá una pestaña en el navegador con una página de Heroku con un formulario de inicio de sesión donde indicaremos nuestras credenciales. Una vez hecho esto, si volvemos al CMD ya nos aparecerá que hemos iniciado sesión.
2. Si ya estamos dentro del directorio donde se encuentra el proyecto (la raíz), haremos uso de "*git add .*" para añadir todos los ficheros bajo el directorio donde estamos.

3. Posteriormente haremos un *commit* con la utilidad *git commit -am "Mensaje de commit"*, de tal manera que confirmaremos los fichero o cambios realizados.
4. Por último, para que se efectúe los cambios y se suban a Heroku, finalizaremos con un *git push heroku master*. Posteriormente, tras esperar unos segundos a que finalice el proceso de subida ya podremos ir nuevamente a Heroku para ver nuestra página web desplegada, clicando en el botón "*Open App*":

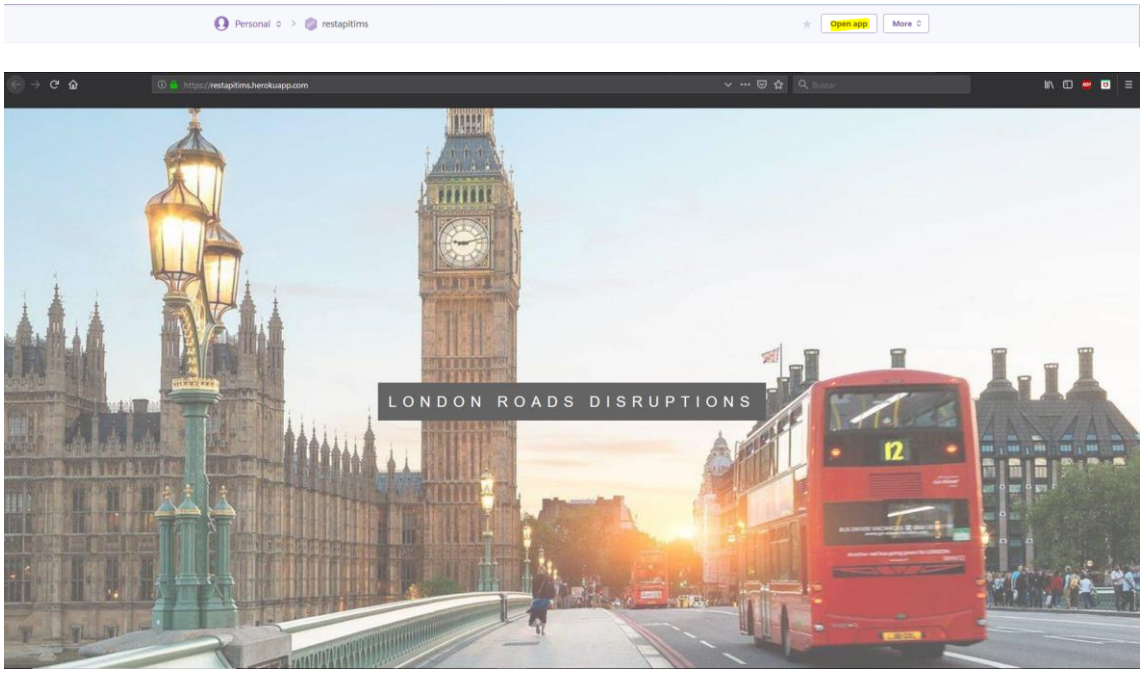


Ilustración 126. Aplicación desplegada finalmente en Heroku

4.4.6. Repositorio público con el código fuente de la aplicación

A diferencia del repositorio de la aplicación de Scala en la que usamos Bitbucket, para esta usaremos GitHub. En el siguiente enlace se ha abierto un repositorio con el código fuente completo de la aplicación:

- Repositorio *GitHub. Juan Ramón Betancor Olivares. Última actualización: 09–04-2019.*

4.4.7. Posibles propuestas de mejora para la aplicación

- Añadir registros de usuarios y utilizar una tabla en la base de datos para gestionar las cuentas de la aplicación, en lugar de utilizar la memoria para almacenar las cuentas.
- Como se comentó durante la creación del controlador, podemos mejorar las consultas a la base de datos, es decir, disminuir las inserciones y los borrados para mejorar el rendimiento de la aplicación.
- Mejorar la seguridad de la aplicación añadiendo nuevos criterios de seguridad.

4.5. Resultado final del proyecto + Demo. Ejecución simultanea de ambas aplicaciones.

Aprovechando que ya tenemos la aplicación desplegada en Heroku, podemos abrir Eclipse y ejecutar la aplicación de Scala para ver las incidencias y los tweets en tiempo real.

Antes debemos asegurarnos de que las direcciones donde se hacen las dos peticiones POST son las correctas. Abriremos la App alojada en Heroku y copiamos la URL. Debe ser similar a la siguiente: https://restapitims.herokuapp.com/ (tan sólo cambiaría el nombre del proyecto). Luego comprobaremos las dos peticiones POST y pondremos dicha URL, para las incidencias y https://restapitims.herokuapp.com/tweets para los tweets (ya que ahí es donde el controlador REST está escuchando):

```
    val readableString: String = Json.prettyPrint(arrjson)
    println(readableString)
    val post = new HttpPost("https://restapitims.herokuapp.com/")
    post.setHeader("Content-type", "application/json")
    post.setEntity(new StringEntity(readableString))
    val responses = (new DefaultHttpClient).execute(post)
    println("--- HEADERS ---")
    response.getAllHeaders.foreach(arg => println(arg))
```

```
val readableString: String = Json.prettyPrint(objeto)
println(readableString)

  val post = new HttpPost("https://restapitims.herokuapp.com/tweets")
  post.setHeader("Content-type", "application/json")
  post.setEntity(new StringEntity(readableString))
  val responses = (new DefaultHttpClient).execute(post)
  println("--- HEADERS TWEET POST ---")
  responses.getAllHeaders.foreach(arg => println(arg))
```

Ilustración 127. Peticiones POST actualizadas para el envío a Heroku

Una vez comprobado ya podemos ejecutar la aplicación Scala. Procedemos a acceder a la página web para ver en tiempo real como van llegando diferentes Tweets cada vez que la web recarga, así como las incidencias:

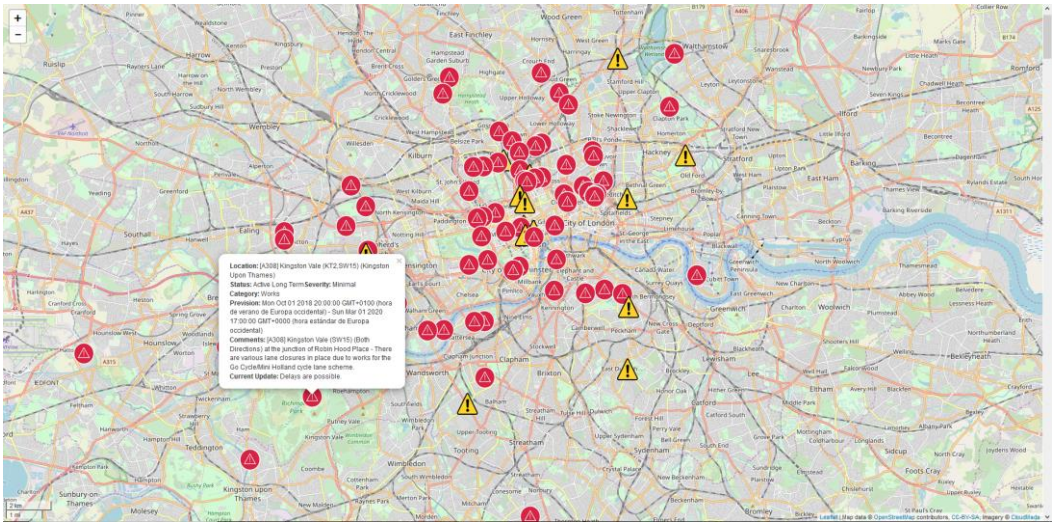


Ilustración 128. Mapa con diferentes incidencias desde Heroku

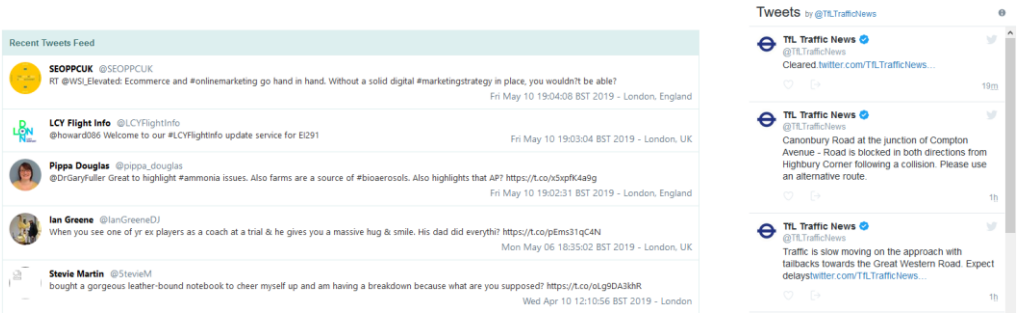


Ilustración 129. Muestras de Tweets desde Heroku

Se puede consultar la demo siguiendo el siguiente enlace: https://restapitims.herokuapp.com/

Tan sólo necesitamos hacer clic en el nombre de la página (*London Roads Disruption*) para acceder al mapa y a los tweets. Aunque antes, si no hemos iniciado sesión, debemos autenticarnos. Se ha creado un usuario público para que se pueda acceder:

- Username: user
- Password: user

5. Conclusiones y trabajos futuros

Durante el desarrollo de proyecto hemos completado todos los objetivos propuestos en el TFT01 inicialmente. Además, podemos ver el resultado de ambas aplicaciones a través de la demo habilitada en Heroku, donde se podrá ver la última ejecución del programa de Scala hasta el momento (al no estar la aplicación continuamente ejecutándose aparece las incidencias y los tweets de la última ejecución, ya que estos son los almacenados en la base de datos). En esta demo podemos ver el trabajo conseguido (los dos principales objetivos):

- Disponemos de una aplicación web donde podemos ver las incidencias representadas en un mapa.
- Además, tenemos un listado con diferentes tweets que pueden estar relacionados con las incidencias.

Al finalizar cada aplicación he propuesto diferentes mejoras que se podría realizar si se quisiera escalar. De forma general podemos establecer algunas extensiones:

- En lugar de basarnos en el dataset de una única ciudad, podemos hacer algo similar a la herramienta de la DGT y extender las incidencias a un país entero o, al menos, diferentes ciudades.
- Se ha de mejorar la aplicación web aportando una mejor interfaz al usuario, añadir diseño responsive a la aplicación entera, añadir registros de usuario, dividir las incidencias y los tweets en diferentes páginas, mejora de la seguridad, etc.

- Podemos afinar más los filtros de los tweets para intentar quedarnos con los tweets que realmente nos aporte una información relevante.
- Mejorar las consultas a la base de datos para mejorar el rendimiento de la aplicación.
- En caso de que se expanda a un país, añadir filtros de búsqueda para las incidencias: buscar por provincias, carreteras, etc.
- Añadir en otro mapa, las incidencias programadas (cuyo estado en el dataset sea *Scheduled*), de tal manera que el usuario pueda hacer una planificación futura (mejor previsión).
- Podríamos establecer guardado de las incidencias en ficheros físicos cada cierto tiempo como si fuesen copias de seguridad
- Se podría además de representar las incidencias, ofrecer directamente desde la aplicación rutas alternativas o instrucciones de actuación para evitar dichas incidencias.

6. Normativa y legislación

Al estar utilizando un dataset público, no estamos tratando con información confidencial en ese aspecto, sin embargo, al estar recopilando tweets de usuarios sin su respectivo consentimiento explícito si puede que exista algún aspecto de la ley de protección de datos que debamos tener en cuenta (se guardan en una base de datos). Al utilizar una base de datos, este debe tener un cierto control y seguridad reflejada en la Ley de Protección de Datos, por lo que tanto la LOPD como la RGPD tienen que estar presente.

7. Glosario de términos

Para la obtención de todas las definiciones se ha utilizado como fuente de información Wikipedia.

B

Backend
El backend es la parte del desarrollo web que se encarga de que toda la lógica de una página web funcione. Se trata del conjunto de acciones que pasan en una web pero que no vemos como, por ejemplo, la comunicación con el servidor. 49, 51

C

CDN
Content Delivery Network. es un sistema de servidores distribuidos (red) que entrega páginas y otro contenido web a un usuario, según las ubicaciones geográficas del usuario y el origen de la página web 62, 66

CLI
La interfaz de línea de comandos o interfaz de línea de órdenes (en inglés, command-line interface, CLI) es un método que permite a los usuarios dar instrucciones a algún programa informático por medio de una línea de texto simple. 83

Clúster MIMD
Multiple Instruction, Multiple Data. Es un sistema con flujo de múltiples

instrucciones que operan sobre múltiples datos. 6

Clúster virtual
Estructura clúster basado en máquinas virtuales en lugar de hardware físico..... 7

CMD
El símbolo del sistema (en inglés, Command prompt, también conocido como cmd.exe o simplemente cmd) es el intérprete de comandos en OS/2 y sistemas basados en Windows 83

CSS
CSS (siglas en inglés de Cascading Style Sheets), en español "Hojas de estilo en cascada", es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. 62, 63, 93

D

Dataset
es una colección de datos habitualmente tabulada... 4, 10, 17, 18, 19, 20, 21, 22, 23, 24, 31, 36, 59, 86, 87

F

Footer
Un footer o pie de página es una línea o bloque de texto que aparece en la parte inferior de una página o documento separado del texto principal. Los footers incluyen notas en el título de la página, o navegación dentro del documento.. 43, 75

Framework
Es una estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software. 10, 42, 44, 47

Frontend
El desarrollo web Front-end consiste en la conversión de datos en una interfaz gráfica para que el usuario pueda ver e interactuar con la información de forma digital usando HTML, CSS y JavaScript.. 45, 49

G

Get
El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.. 18, 19, 20, 23, 24, 57, 58

H

HTML
HyperText Markup Language (lenguaje de marcas de hipertexto), hace referencia al lenguaje de marcado para la elaboración de páginas web.... 5, 45, 62, 66, 68, 69, 70, 80, 94

I

IDE
Un entorno de desarrollo integrado12 o entorno de desarrollo interactivo, en inglés Integrated Development Environment (IDE), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software. .9, 11, 12, 22, 32, 46, 47, 49, 51, 55, 64, 65

J

JARS
Java Archive. Formatos de fichero usado principalmente para archive clases de Java a nuestros proyectos. 8, 18, 19, 23, 27, 32, 37, 42

JPA
Java Persistence API, más conocida por sus siglas JPA, es la API de persistencia desarrollada para la plataforma Java EE49, 53, 57, 59, 94

JSON..*4, 9, 10, 27, 28, 29, 37, 38, 39, 40, 42, 54, 56, 58, 59, 60, 93*

M

Maven
Maven es una herramienta de software para la gestión y construcción de proyectos Java. Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. 42, 48, 49, 51, 94

MVC
Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que

separa los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador 42, 44, 61, 93

N

Navbar
Navigation Bar. Es una sección de una interfaz gráfica de usuario diseñada para ayudar a los visitantes a acceder a la información..... 69, 70

P

POST
El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.... 9, 29, 30, 38, 39, 40, 58, 59, 60, 68, 78, 85, 93

R

RDD
Resilient Distributed DataSet que es un multiset de solo lectura de ítems de datos distribuidos a lo largo de un clúster. 40

Receiver
Mecanismo encargado de obtener la información de la fuente. 4, 33, 36, 93

Rest API
Abreviatura de Representational State Transfer, o Transferencia de Estado Representacional es un estilo de arquitectura para diseñar aplicaciones en red basado en peticiones HTTP. 4, 42

Ruby on Rails
Es un framework de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby, siguiendo el paradigma del patrón Modelo Vista Controlador (MVC). 5, 44

S

SBT
Herramienta software para la construcción de aplicaciones Scala y Java. Similar a Maven, permite la gestión de las dependencias. 42

SQL
SQL (por sus siglas en inglés Structured Query Language 46, 59

Stream
Es una secuencia de datos disponibles a lo largo del tiempo. 21, 30, 31, 36, 39, 40

T

TFL
Transport for London (TfL) es el organismo del gobierno local responsable de la mayoría de los aspectos del sistema de transportes en Londres, Inglaterra. Su rol es implementar las estrategias de transporte y administrar los servicios de transporte de Londres..... 9, 17, 18, 20, 27, 33, 70, 74, 75

Timestamp
Sello de tiempo o timestamp, es una secuencia de caracteres que denotan la hora y fecha (o alguna de ellas) en la/s que ocurrió determinado evento.....40, 41

X

XML
eXtensible Markup Language, es un meta-lenguaje que permite definir lenguajes de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible.....9, 10, 17, 20, 22, 23, 24, 25, 26, 93

8. Tabla de ilustraciones

Ilustración 1. Descargando Apache Spark 8
Ilustración 2. Mirrors HTTP para descargar Apache Spark 8
Ilustración 3. Logo ScalaIDE for Eclipse 10
Ilustración 4. Logo Scala 10
Ilustración 5. Logos de Spark Streaming, Twitter API y de Stanford University respectivamente 11
Ilustración 6. Logo de Bitbucket 11
Ilustración 7. Descargando Scala IDE. 12
Ilustración 8. Configurando del proyecto 12
Ilustración 9. Estructura inicial del proyecto Scala 13
Ilustración 10. Añadiendo un package a nuestra aplicación. 13
Ilustración 11. Paquete creado correctamente. 14
Ilustración 12. Creando la main class (Scala object) 14
Ilustración 13. Resultado al crear el main. 14
Ilustración 14. Creando el método main 15
Ilustración 15. Configuración de ejecución del programa 16
Ilustración 16. Resultado al ejecutar el método main 16
Ilustración 17. Dataset con los accidentes de tráfico (TIMS) 17
Ilustración 18. Dentro del XML de las incidencias 17
Ilustración 19. Tiempo de actualización del dataset. 17
Ilustración 20. Creando el receiver de TFL. 18
Ilustración 21. Proceso de importar los JARS 19
Ilustración 22. Creando los métodos de la interfaz Runnable 19
Ilustración 23. Método receive() con la lógica del receptor 20
Ilustración 24. Invocando el receiver TFL desde el main 20
Ilustración 25. Ejecución de Spark Streaming. 21
Ilustración 26. XML descargado y visto desde la terminal. 22
Ilustración 27. Fix. Igualando las versiones de Scala y Spark. 22
Ilustración 28. Creando los formatos de entrada y salida de las fechas. 23
Ilustración 29. Creando nuestro objeto XML desde una String 24
Ilustración 30. Filtrando las incidencias del XML utilizando Scala XML 25
Ilustración 31. Campos importantes de la estructura de las incidencias del XML 26
Ilustración 32. Formato legible de las coordenadas 26
Ilustración 33. Incidencias filtradas 26
Ilustración 34. Importando Scala JSON. 27
Ilustración 35. Cuerpo/Contenido del objeto JSON. Imagen sacada de Postman en las pruebas. 27
Ilustración 36. Creando el array JSON. 28
Ilustración 37. Creando el objeto JSON y añadiéndolo al array JSON 28
Ilustración 38. Importando las librerías HTTP POST de Java 29
Ilustración 39. Estructura para las peticiones POST 29
Ilustración 40. Petición POST utilizando Postman 30
Ilustración 41. Guardando el stream en un fichero de texto 30
Ilustración 42. Formulario de registro en Twitter Dev. 31
Ilustración 43. Aplicación creada en Twitter Dev. 31
Ilustración 44. Credenciales de nuestra aplicación 32
Ilustración 45. Fichero TXT con las credenciales de nuestra aplicación para autenticarnos en Twitter 32
Ilustración 46. Importando las librerías para Twitter API 33
Ilustración 47. Creando la clase Utilities para los métodos auxiliares 34

Ilustración 48. Importando las librerías necesarias en Utilities 34
Ilustración 49. Métodos setupLogging() y setupTwitter() de Utilities 34
Ilustración 50. Importando la clase Utilities desde el main 35
Ilustración 51. Llamada a los métodos setupTwitter() y setupLogging() desde el main 35
Ilustración 52. Palabras claves para orientar la búsqueda 36
Ilustración 53. Llamada al receiver de Twitter para crear el stream 36
Ilustración 54. Importando el fichero para el análisis de sentimientos 38
Ilustración 55. Importando Scala JSON y HTTP POST 39
Ilustración 56. Filtrando los tweets para su posterior envío 39
Ilustración 57. Resultado de la ejecución final de la aplicación Scala 41
Ilustración 58. Diseño previo de la página donde mostraremos las incidencias y los tweets 43
Ilustración 59. IntelliJ IDEA Logo 44
Ilustración 60. Spring Boot logo 44
Ilustración 61. Thymeleaf logo 45
Ilustración 62. Lenguajes de programación web 45
Ilustración 63. Leaflet y OpenStreetMap logo 45
Ilustración 64. PostgreSQL logo 46
Ilustración 65. Heroku y GitHub Logo 46
Ilustración 66. Licencia de estudiante JetBrains 47
Ilustración 67. Interfaz inicial IntelliJ IDEA 47
Ilustración 68. Estableciendo Maven en nuestro proyecto 48
Ilustración 69. GroupID y ArtifactID 48
Ilustración 70. Estructura del proyecto inicial 48
Ilustración 71. Bloque parent dentro de pom.xml 49
Ilustración 72. Dependencias de nuestro proyecto (1) 50
Ilustración 73. Dependencias de nuestro proyecto (2) 50
Ilustración 74. Creando aplicación desde Heroku 51
Ilustración 75. Aplicación Heroku creada 52
Ilustración 76. Pestaña credentials de la aplicación 52
Ilustración 77. Credenciales de la aplicación 52
Ilustración 78. Conexión a la base de datos desde Spring 52
Ilustración 79. DisruptionRepository 53
Ilustración 80. TweetsRepository 53
Ilustración 81. Objeto JSON que representa a la incidencia de tráfico 54
Ilustración 82. Objeto JSON que representa un tweet 54
Ilustración 83. Entidad DisruptionEntity que referencia a una incidencia 55
Ilustración 84. Entidad TweetsEntity que representa un tweet 56
Ilustración 85. Importando utilidades y creando los atributos dentro del controlador 57
Ilustración 86. Routing para la raíz y /login de la aplicación 58
Ilustración 87. Ruta POST para las incidencias de tráfico 58
Ilustración 88. Ruta POST para los tweets 59
Ilustración 89. Ruta /disruptions donde mostraremos el mapa y los tweets 60
Ilustración 90. Ubicación de flexboxgrid.css dentro del proyecto 62
Ilustración 91. Estructura inicial de archivo HTML recién creado 62
Ilustración 92. Bloque head con el CSS de la página 63
Ilustración 93. Bloque body del homepage 63
Ilustración 94. Método main de la aplicación web 64
Ilustración 95. Botón run para ejecutar el programa 64
Ilustración 96. Ejecución de Spring Boot 65
Ilustración 97. Resultado del homepage de la aplicación 65

Ilustración 98. Bloque head del login 66
Ilustración 99. Bloque body del formulario login (1) 67
Ilustración 100. Bloque body del formulario login (2) 67
Ilustración 101. Resultado de la página de inicio de sesión 68
Ilustración 102. Bloque head de index.html 69
Ilustración 103. Navbar de index.html 70
Ilustración 104. DIV donde se incluirá el mapa 70
Ilustración 105. Código JavaScript para configurar el mapa (1) 71
Ilustración 106. Iconos para las incidencias "Long Term Active" y "Active" respectivamente 71
Ilustración 107. Layer del mapa 71
Ilustración 108. Código JavaScript para configurar el mapa (2) 72
Ilustración 109. Código de la tabla de incidencias 73
Ilustración 110. Filtro de búsqueda de la tabla 74
Ilustración 111. Bloque destinado a mostrar los Tweets 75
Ilustración 112. Footer de index.html 75
Ilustración 113. Mapa completo con incidencias 76
Ilustración 114. Tabla de incidencias completa 76
Ilustración 115. Lista de Tweets finalizada 76
Ilustración 116. Footer finalizado 76
Ilustración 117. Configuración Spring Security 77
Ilustración 118. Gestor de accesos denegados 79
Ilustración 119. Bloque head de 403.html 80
Ilustración 120. Bloque Body de 403.html 80
Ilustración 121. Bloque head de 404.html 81
Ilustración 122. Bloque body de 404.html 81
Ilustración 123. Página 404 terminada 81
Ilustración 124. Estructura final del proyecto 82
Ilustración 125. Pasos para hacer el despliegue en Heroku 83
Ilustración 126. Aplicación desplegada finalmente en Heroku 84
Ilustración 127. Peticiones POST actualizadas para el envío a Heroku 85
Ilustración 128. Mapa con diferentes incidencias desde Heroku 85
Ilustración 129. Muestras de Tweets desde Heroku 86

9. Fuentes de información

9.1. Referentes a la aplicación.

[1] Transport for London, TLP. (2019). *Transport for London | Traffic Information Management System (TIMS) Feed.* Retrieved from: https://tfl.gov.uk/tfl/syndication/feeds/tims_feed.xml [Último acceso: 16 Abril 2019]

[2] Daniel Rodríguez, *makigas* (producer/director). (2016-2017). *Tutorial SCALA [video].* Retrieved from: *https://www.makigas.es/series/scala*

[3] Udemy, Inc & Sundog Education by Frank Kane. (Fecha de la última actualización: 4/2019). *Taming Big Data with Spark Streaming and Scala - Hands On! [course].* Retrieved from: https://www.udemy.com/taming-big-data-with-spark-streaming-hands-on/?src=sac&kw=taming%20big

[4] The Apache Software Foundation, Apache Spark. (2019). *Spark Streaming Custom Receivers.* Retrieved from: https://spark.apache.org/docs/2.2.0/streaming-custom-receivers.html

[5] Aurobindo Sarkar, Aurobindo's Blogs. (October 30, 2017). *Spark Streaming: Writing a receiver for a custom data source [blog post].* Retrieved from: http://www.twesdai.com/category/spark-streaming/

[6] Community-maintained library for Scala. Lead maintainer is Aaron S. Hawley. (2019). *The standard Scala XML library.* Retrieved from: https://github.com/scala/scala-xml

[7] Felipe Almeida, queirozf.com. (03 Dec 2014, last update: 23 Mar 2019). *Scala Date Examples: Formatting, Converting and other Useful Operations on Dates*. Retrieved from: http://queirozf.com/entries/scala-date-examples-formatting-converting-and-other-useful-operations-on-dates

[8] Justin Musgrove, leveluplunch.com. (22 August 2014). *Post JSON to spring REST webservice.* Retrieved from: https://www.leveluplunch.com/java/tutorials/014-post-json-to-spring-rest-webservice/

[9] Alvin Alexander, alvinalexander.com. (June 3 2016). *Adding JSON to the body of an HTTP POST request in Scala or Java*. Retrieved from: https://alvinalexander.com/scala/add-json-string-body-http-post-request-scala-java

*[10]*Aurelio Morales, mappinggis.com. (28 agosto, 2018). *Cómo crear un mapa con Leaflet.* Retrieved from: https://mappinggis.com/2013/06/como-crear-un-mapa-con-leaflet/

*[11]*Andrew Shcherbakov, baeldung.com. (Last modified: February 18, 2019). *Accessing Spring MVC Model Objects in JavaScript.* Retrieved from: https://www.baeldung.com/spring-mvc-model-objects-js

*[12]*Solution posted by Adam Harte, stackoverflow.com. *Make a always full screen [forum post].* Retrieved from: https://stackoverflow.com/questions/1719452/how-to-make-a-div-always-full-screen

*[13]*W3School, w3schools.com. (2019). *How TO - Full Page Image.* Retrieved from: https://www.w3schools.com/howto/howto_css_full_page.asp

*[14]*Aigars, colorlib.com. (April 21, 2018). *Responsive Table V1.* Retrieved from: https://colorlib.com/wp/template/responsive-table-v1/

*[15]*Dimitar Ivanov, jsfiddle.net. (2019). *CSS Tables: Simple table.* Retrieved from: https://jsfiddle.net/zinoui/dB93J/

[16] Solution posted by Jainil, stackoverflow.com. (Jan 11 '17). C*hange leaflet marker icon [forum post].* Retrieved from: https://stackoverflow.com/questions/41590102/change-leaflet-marker-icon

[17] Sandeep Dayananda, edureka.co. (Feb 18,2019). *Spark Streaming Tutorial – Sentiment Analysis Using Apache Spark.* Retrieved from: https://www.edureka.co/blog/spark-streaming/

[18] Nicolas A. Perez, medium.com. (Apr 5, 2016). *Spark Streaming and Twitter Sentiment Analysis.* Retrieved from: https://medium.com/@anicolaspp/spark-streaming-and-twitter-sentiment-analysis-c860938d484

[19] Chaitanya Singh, beginnersbook.com. (2019). *Java ArrayList add() Method Example.* Retrieved from: https://beginnersbook.com/2013/12/java-arraylist-add-method-example/

[20] Oracle, Inc, docs.oracle.com. (2015). Class ArrayList<E>. Java Platform SE 6. Retrieved from: https://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html#add(E)

[21] RohitPrasad3, geeksforgeeks.org. (2019). *ArrayList subList() method in Java with Examples.* Retrieved from: https://www.geeksforgeeks.org/arraylist-sublist-method-in-java-with-examples/

[22] Vincent Spiewak, github.com. (2 Mar 2015). *SentimentAnalysisUtils.scala [scala file].* Retrieved from: https://github.com/vspiewak/twitter-sentiment-analysis/blob/master/src/main/scala/com/github/vspiewak/util/SentimentAnalysisUtils.scala

[23] Rajeev Singh, callicoder.com. (Apr 30, 2018). *Spring Boot, PostgreSQL, JPA, Hibernate RESTful CRUD API Example.* Retrieved from: https://www.callicoder.com/spring-boot-jpa-hibernate-postgresql-restful-crud-api-example/

[24] Pivotal Software, Inc, spring.io. (2019). *Accessing Data with JPA.* Retrieved from: https://spring.io/guides/gs/accessing-data-jpa/

[25] JSFiddle, jsfiddle.net. (2019). *Filter HTML table.* Retrieved from: http://jsfiddle.net/ukW2C/3/

[26] Eugen Paraschiv, baeldung.com. (Last modified: April 7, 2019). Spring Security with Maven. Retrieved from: https://www.baeldung.com/spring-security-with-maven

[27] MemoryNotFound, memorynotfound.com. (ublished November 2, 2017 · Updated November 2, 2017). *Spring Boot + Spring Security + Thymeleaf Form Login Example.* Retrieved from: https://memorynotfound.com/spring-boot-spring-security-thymeleaf-form-login-example/

[28] Luis Miguel Gracia, unpocodejava.com. (15 diciembre 2017). *Procesando y representando datos en un mapa Leaflet con Apache Zeppelin.* Retrieved from: https://unpocodejava.com/2017/12/15/procesando-y-representando-datos-en-un-mapa-leaflet-con-apache-zeppelin/

[29] Wikipedia (28 abr 2019). *Spring Framework*. Retrieved from: https://es.wikipedia.org/wiki/Spring_Framework

[30] Wikipedia (18 may 2018). *Thymeleaf*. Retrieved from: https://es.wikipedia.org/wiki/Thymeleaf

[31] Wikipedia (4 abr 2019). *PostgreSQL*. Retrieved from: https://es.wikipedia.org/wiki/PostgreSQL

[32] Pivotal Software, Inc. (2019). *Building a RESTful web service*. Retrieved from: https://spring.io/guides/gs/rest-service/

9.2. Referente a la memoria y puesta en marcha de la aplicación

[1] Milagros Escalera Venancio, Biblioteca Amaury Veray. (agosto 2011). *PREPARACIÓN DE BIBLIOGRAFÍAS SEGÚN MANUAL DE ESTILO DE LA AMERICAN PSYCHOLOGICAL ASSOCIATION (APA) 6ª ed* [pdf file]. Retrieved from: https://cmpr.edu/docs/bib/bibliografia-apa-CMPR.pdf

[2] Solution posted by Corona Luo, stackoverflow.com. (Nov 12 '13). *Bitbucket - Push project files to an empty Git repository [forum post].* Retrieved from: https://stackoverflow.com/questions/17921404/bitbucket-push-project-files-to-an-empty-git-repository

[3] Solution posted by Kevan Ahlquist, stackoverflow.com. (Jun 30 '14). *How to stop an app on Heroku? [fórum post].* Retrieved from: https://stackoverflow.com/questions/2811453/how-to-stop-an-app-on-heroku

[4] GitHub, Inc., help.github.com. (2019). *Adding a file to a repository using the command line.* Retrieved from: https://help.github.com/en/articles/adding-a-file-to-a-repository-using-the-command-line

[5] Heroku, Inc., devcenter.heroku.com. (Last updated 20 February 2019). *Deploying Java Applications to Heroku from Eclipse or IntelliJ IDEA.* Retrieved from: https://devcenter.heroku.com/articles/deploying-java-applications-to-heroku-from-eclipse-or-intellij-idea

[6] Universidad Politécnica de Valencia. (2019). *Cómo escribir un trabajo de fin de grado [pdf].* Retrieved from: http://personales.upv.es/fjabad/pfc/comoEscribir.pdf

[7] Universidad de Las Palmas de Gran Canaria, EII. (26/06/2018). *Manual operativo de trabajos de fin de grado.* Retrieved from: https://www.eii.ulpgc.es/tb_university_ex/sites/default/files/files/trabajos%20fin%20de%20grado/Formularios/Manual%20Operativo%20TFT%20EII%20(Rev_%2010).pdf

9.3. Repositorios y DEMO de la aplicación

[1] Juan Ramón Betancor Olivares, github.com. (Ultima actualización: 9 de abril del 2019). *Repositorio GitHub con el contenido de la aplicación REST con Springboot (RestAPI-TIMS).* Available: *https://github.com/JuanRaBetancor/RestAPI-TIMS*

[2] Juan Ramón Betancor Olivares, heroku.com. (Ultima actualización: 9 de abril del 2019). Aplicación desplegada en servidor Heroku. Available: https://restapitims.herokuapp.com/. Utilizar *user* como usuario y contraseña.

[3] Juan Ramón Betancor Olivares, bitbucket.org. (Ultima actualización: 8 de abril del 2019). *Repositorio con el contenido de la aplicación Scala encargada del análisis de los datos.* Available: https://bitbucket.org/JuanraBO/sparkstreamingapp/src/master/