ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

HERRAMIENTA WEB SOBRE TWITTER PARA EL ESTUDIO Y ANÁLISIS DEL IMPACTO DE LAS EMERGENCIAS

TWITTER-BASED WEB TOOL FOR THE STUDY AND ANALYSIS OF THE IMPACT OF EMERGENCIES

Realizado por
ADRIÁN GARCÍA HUMANES
Tutorizado por
EDUARDO GUZMÁN DE LOS RISCOS
Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JULIO 2017

Fecha defensa:
El Secretario del Tribunal

Resumen: Hemos desarrollado una aplicación web capaz de obtener datos y tuits a través de la API de Twitter y posteriormente visualizar información a través de ellos de cara a evitar pérdidas o minimizar riesgos en emergencias. Para esto la aplicación se basa en tomar o bien hasta cinco hashtags o menciones de usuarios, o bien en tomar el nombre de un usuario para sacar estos datos. Una vez hecho esto, podemos ver diversos campos de ellos, como la fecha en que han sido escritos, quién lo escribió, el contenido en sí mismo del tuit, etc. Además, permitimos filtrar la lista de tuits obtenidos o añadir un elemento nuevo a la búsqueda hasta llegar a la ya mencionada cifra de cinco elementos. Junto a ello presentamos un módulo para situar cada tuit en un mapa con su correspondiente localización. Por último, se ha desarrollado un módulo utilizando MongoDB que permita almacenar estos tuits para realizar una posterior consulta de ellos. Adicionalmente a toda la lógica principal de la aplicación en sí misma hemos desarrollado otro módulo que nos permite interpretar las URLs y nombres de usuario de Twitter y establecer un hipervínculo a ellos, así como resaltar de manera visual cada hashtag o mención introducida inicialmente con un color distinto para una rápida identificación.

Palabras claves: Twitter, Spring, MongoDB

Abstract: It has been developed a web application that can retrieve data and tweets from the Twitter's API and then show information about them in order to avoid or minimize risks in emergencies. For this, the application uses up to five hashtags or a user's nickname to get this data. Once this is done, we can see different fields of them, such as the date it was written, who wrote it, the context of the tweet itself, etc. In addition, we can also filter the list of tweets obtained or add a new element to the search until you reach at the limit of five elements. Furthermore, we have a module to place each tweet on a map with its corresponding location. Finally, a module has been developed using MongoDB that allows the user to store this tweets for a future revision of them. In addition to all the main logic of the application itself, we have developed another module to interpret URLs and Twitter usernames and to set a

hyperlink to them, as well as visually highlight hashtag initially introduced with a different color for each of them for a quick identification.

Keywords: Twitter, Spring, MongoDB

ÍNDICE

1 INTRODUCCIÓN

A lo largo de este primer punto vamos a mencionar resumidamente el contenido de este documento.

Empezaremos hablando a modo de introducción, del objetivo principal del proyecto, que no es otro que desarrollar una herramienta que sea útil de cara a la obtención de manera rápida y tratamiento de la información relativa a emergencias en tiempo real.

Esta herramienta obtendría la información desde Twitter a partir de unos parámetros proporcionados por el usuario, y a partir de aquí podríamos ver varios datos de ellos como:

- Localización de emergencia.
- Hashtags relacionados con la incidencia.
- Nuevos hashtags relacionados con los anteriores al ser incluidos en tuits.
- Usuarios en el centro de la emergencia emitiendo tuits de forma constante a los que poder empezar a monitorizar.

Por último, vamos a detallar el procedimiento a la hora de rellenar este documento.

Vamos a abordar el desarrollo del proyecto y la escritura de la memoria de una forma similar a lo que podría ser un desarrollo iterativo e incremental. Se irán explicando y documentando todas y cada una de las fases por las que pasará el proyecto durante su realización.

Con esto, en cada una de las partes relacionadas con una fase del proyecto encontraremos en primer lugar como puede resultar obvio qué tratábamos de conseguir, qué se consiguió, partes de código que nos resulten interesantes de pasar por ellas, bugs o fallos encontrados al término de la fase y otros que tal vez se hayan corregido de fases anteriores, y posibles actualizaciones y/o revisiones de la fase que se hiciesen poco después de terminar esta. Adicionalmente, si en alguna fase por razones de diseño o funcionales fuese necesario una reestructuración del código o cambios drásticos se mencionarían de igual forma.

Finalmente, y aunque pueda resultar evidente, el propósito de realizar el proyecto de manera iterativa e incremental, es obtener al final de cada iteración siempre un nuevo módulo o funcionalidad completos por pequeños que puedan llegar a ser. Además, algunas iteraciones deberán dedicarse por completo bien a unir módulos realizados

de manera individual o bien a pulir el diseño. (Nótese que ya de por sí, en cada una de las iteraciones, realizamos tanto el trabajo de la programación y funcionalidad interna, como el del diseño y estructura visual necesarios para mostrar y utilizar estas funciones y utilidades).

1.1 OBJETIVOS A CONSEGUIR Y NECESIDADES A CUBRIR

Los principales objetivos y necesidades que pretendemos cubrir con el desarrollo de este proyecto son, en primer lugar, poder tener una herramienta funcional y útil para poder sesgar u obtener información específica de la red social Twitter. Junto a esto, esperamos que la plataforma permita, debido al carácter inmediato de la propia información en Twitter, poder reducir el peligro concerniente a estas emergencias y, a través de los cuerpos de seguridad, prevención y tratamiento de emergencias que pudieran llegar a usarla, evitar o reducir las pérdidas tanto materiales como humanas al dar un punto más de velocidad para la obtención de los datos.

1.2 PRIMEROS PASOS ANTES DE INICIAR EL PROYECTO

A la hora de elegir este proyecto y plantear su realización la primera pregunta que debemos hacernos es sobre qué lenguaje de programación trabajaremos y qué herramientas utilizaremos para ayudarnos y poder llevarlo a cabo.

En nuestro caso nos decantamos por Java como lenguaje de programación principal por la experiencia que hemos ido acumulando con él durante estos años.

Ahora bien, se nos abren un abanico de posibilidades aun teniendo decidido Java como lenguaje de programación. Dentro de él tenemos multitud de frameworks que nos pueden proporcionar un conjunto de herramientas suficientes para lo que buscamos hacer. No obstante, al inicio del proyecto, nuestra decisión se tomó de entre estas tres posibles opciones:

- Java junto a JSP (Java Server Pages) o Servlets
- JSF (Java Server Faces) como framework principal sobre Java
- Spring Framework como framework principal sobre Java

Finalmente optamos por la última, Spring Framework. Aunque cualquiera de las tres nos hubiera sido útil para esto, escogimos Spring por, en primer lugar, la posibilidad de aprender una nueva tecnología y aumentar aún más nuestra base de

conocimientos. En segundo lugar, tenemos las facilidades que nos proporciona y de las que hablaremos a continuación.

2 TECNOLOGÍAS Y HERRAMIENTAS UTILIZADAS

2.1 SPRING FRAMEWORK Y SUS VENTAJAS

Cuando comenzamos a desarrollar en cualquier plataforma y lenguaje, normalmente solemos optar para facilitarnos algo las cosas por utilizar algún marco de trabajo o framework, esto es, un conjunto de clases ya definidas o herramientas que nos brindan utilidades para realizar tareas de diversa índole y, en definitiva, ahorrarnos algo de tiempo y complejidad en el desarrollo.

Junto a esto, es posible que, en varias ocasiones, al trabajar, hayamos necesitado de varios frameworks en lugar de uno solo, lo que nos puede haber llevado a que cada uno de los necesarios nos den su propio conjunto de clases y objetos.

Spring Framework solventa estos problemas al poder construir los objetos necesarios a partir de anotaciones o XML, y así, resulta ser el único encargado de todos ellos y consigue una integración total sin problemas.

Algunas de las principales características que nos brinda Spring Framework son las inyecciones de dependencias, Spring MVC web application y RESTful web service framework, además de proporcionar apoyo a JDBC, JPA, etc.

Un último punto a favor, a pesar de ser algo que va más allá de las propias características, es el hecho de que es un framework que crece día a día en popularidad, con lo que podemos encontrarnos con una gran ayuda de parte de la comunidad y multitud de guías para los distintos módulos de Spring disponibles.

2.2 MÓDULOS DE SPRING FRAMEWORK

Hemos hablado ya de poder tener objetos y herramientas muy diversas controladas por Spring. ¿Cómo se lleva esto a cabo? A través de los módulos. Y este es un gran punto fuerte de usar este framework, la posibilidad de ampliar o reducir las funcionalidades y utilidades a usar a través de ellos, sin tener que obligarnos a un conjunto de las mismas cerrado o instalar otro framework con algunas que sean obligatorias y nunca vayamos a usar.

Entre los principales módulos tenemos servicios como un contenedor de inversión de control (inyección de dependencias), el acceso a datos, procesamiento por lotes, autenticación y autorización, modelo vista controlador, etc.

Y nos vamos a detener un momento en el contenedor de inversión de control, que nos facilita la gestión de los objetos y de sus ciclos de vida específicos.

Estos objetos se denominan beans y nos permiten configurar el contenedor mediante XML o a través de anotaciones específicas sobre las clases. Estos beans pueden obtenerse por la inyección o búsqueda de dependencias.

Podemos encontrar una lista con todos los módulos disponibles al acceder al enlace disponible en [1] y seleccionar la opción 'Switch to the full version'.

2.3 TWITTER Y SU API

Todos conocemos en gran medida Twitter. Es una red social basada en el microblogging que nos permite mandar mensajes de texto plano con una longitud máxima de 140 caracteres llamados tweets o tuits en español. Dentro de ella, podemos suscribirnos o en este caso seguir a otros usuarios, de igual manera que ellos pueden seguirnos a nosotros. Al seguir a un usuario automáticamente accedemos a que se nos muestren sus tuits para poder seguir su actividad (Excepciones perfiles privados). Junto a esto podemos elegir hacer un retuit sobre el tuit de una persona para difundir este mensaje a las personas que nos sigan a nosotros o indicar que nos gusta.

Es una red social muy potente, con una gran capacidad de difusión, y es la base sobre la que vamos a cimentar nuestro proyecto.

Sobre esto tenemos que comentar varios puntos en lo que a la API (Interfaz de programación de aplicaciones) de Twitter se refiere. El primero de ellos, y es uno muy de agradecer es que Twitter nos brinda una API fácil y sencilla de utilizar, sin querer ocultar o poner pegas al desarrollador en ningún momento en lo que a facilidad y posibilidades se refiere, teniendo opciones para hacer diversas funcionalidades, desde crear nuevos tuits, obtener información de un usuario, encontrar tuits relacionados con lugares, etc.

Por otro lado, esta gran versatilidad de opciones y posibilidades contrasta con el hecho de que, en cuanto a cifras, es algo limitada. Es decir, a priori tenemos la sensación de que podemos obtener un montón de información relativa a un hashtag por poner un ejemplo, pero a la hora de obtener esa información, los ratios de tuits o peticiones por

unidad de tiempo que nos da Twitter nos limitan mucho en cuanto a la información obtenida.

Para acabar, mencionar que, aunque la API de Twitter es funcional total e independientemente, en nuestro caso optaremos por apoyarnos en un módulo de Spring Framework, Spring Social Twitter.

2.4 MAVEN

Al trabajar bajo Java es algo normal y común necesitar el uso de librerías.

No obstante, esto puede acabar resultando un problema, ya que a veces podemos necesitar una librería, pero además de esto, necesitar una versión en concreto de la misma. Para culminar, a veces unas librerías pueden depender de otras, que a su vez tienen varias versiones, por lo que necesitamos mucha información por ello.

Es aquí donde entra Maven a solventar estos problemas utilizando lo que se denomina *artefacto*, esto es, una especie de estructura que contiene no solo a una librería, sino todo lo necesario para gestionarla de manera apropiada, es decir, el grupo, las dependencias que pudiera tener, versiones, etc.

Con esto, no tenemos más que definir los artefactos en un fichero pom.xml que se encargará de contener todos estos artefactos e información.

Una vez hecho esto, no tenemos más que hacer uso de Maven, que nos proporcionará un repositorio donde estarán alojadas estas librerías, versiones, etc.

Esto simplifica enormemente la gestión de dependencias y librerías y casi se ha convertido en una obligación a día de hoy en proyectos medianos y grandes, además de que en muchas ocasiones no tendremos más que copiar y pegar en nuestro pom.xml sin tener que pensar mucho más para encontrar cierta librería.

Como último apunte, la instalación de Maven para su uso es simple, pero tiene ciertos pasos muy concretos que detallaremos en el Manual de instalación.

2.5 MONGODB

Lo primero que debemos saber de MongoDB es que es una base de datos NoSQL o no relacional, esto es, no tenemos registros, tablas o incluso SQL, sino que lo que nos encontramos es una base de datos que va orientada a documentos, es decir, guardamos datos en documentos en lugar de datos en registros. Estos datos

previamente se codifican en JSON. Es una tecnología muy potente y la gran ventaja de NoSQL respecto a las bases de datos tradicionales o relacionales es que no tenemos por qué seguir un esquema, sino que los documentos que almacenamos de una misma colección pueden tener esquemas diferentes. Y de igual manera para las consultas lo que tenemos que hacer es pasar objetos JSON como parámetros.

Como apunte adicional y, al igual que pasaba con Spring, nos hemos decantado por MongoDB al ser una tecnología con la que no habíamos trabajado antes (Por partida doble si se tiene en cuenta la experiencia nula en bases de datos NoSQL).

Su instalación es sencilla y una vez hecho en Windows nos basta con seguir o ejecutar dos comandos una vez todo está listo para comenzar o levantar de nuevo el servicio tal como podemos ver en [2]. De igual manera explicaremos más detalles sobre el proceso en el Manual de instalación.

2.6 THYMELEAF

Thymeleaf es una librería Java que funciona como motor de plantillas de HTML/5/XHTML, además proporciona un módulo que no es obligatorio en principio para integración con Spring MVC.

El punto fuerte que supone utilizar Thymeleaf es el poder obviar el uso de las etiquetas de JSP, por ejemplo, así como dotar de dinamismo a nuestras páginas HTML sin tener que añadirle nuevos tags a este, solo le incluiríamos el tipo de elemento por el que queremos interpretarlo con Thymeleaf dentro de la propia etiqueta.

Como punto negativo, a lo largo del desarrollo de este proyecto hemos encontrado ciertas dificultades a la hora de hacerlo funcionar conjuntamente con algunos elementos de JavaScript o de que realizase una correcta interpretación de algunos elementos HTML.

2.7 BOOTSTRAP

No queremos pasar al siguiente punto sin dejar de comentar un último framework, Bootstrap en este caso. En esta ocasión es distinto ya que este framework de CSS (hojas de estilo en cascada) desarrollado por Twitter va dirigido a la creación de interfaces de usuario simples/limpias además de adaptables (responsive). Sus

grandes características son el soporte casi perfecto y completo con HTML5 y CSS3, un sistema GRID que permite diseñar usando un GRID de hasta 12 columnas, imágenes adaptables, etc.

Prácticamente nos permite darle un diseño agradable, simple y correcto a casi cualquier elemento y estructura en HTML, pudiendo acudir para ello al catálogo o anexo completo de elementos y posibilidades con el framework en [3].

Además, es compatible con un gran número de navegadores webs.

En nuestro caso, hemos optado por usarlo debido a que, por un lado, facilita mucho el diseño de la interfaz de usuario y agiliza tareas y, por otro lado, afianzar más aun los conceptos que tenemos de él. No obstante, no nos cerraremos únicamente a él y en algún punto del desarrollo del proyecto programaremos usando CSS puro en lugar de Bootstrap.

3. INICIANDO EL PROYECTO

Lo primero que debemos hacer es instalar varios plugins en Eclipse IDE. En este caso vamos a necesitar ‘Spring IDE’, y adicionalmente hemos instalado ‘Maven Integration for Eclipse’. Para esto no tenemos más que acceder a ‘Eclipse Marketplace’, buscarlos e instalarlos.

Una vez hagamos eso, lo siguiente será acceder al enlace ya mencionado en [1] y seleccionar los módulos de Spring Framework con los que deseamos crear un proyecto inicial. Este paso puede saltarse y podemos introducir los módulos manualmente en el pom.xml pero hemos elegido esta opción por comodidad.

El siguiente paso sería acceder a [4] para obtener una API Key de Twitter para poder acceder a usar sus funcionalidades para desarrolladores. Para ello tenemos que iniciar sesión con nuestra cuenta de Twitter, y elegir la opción de crear ‘Create New App’, tras esto rellenamos el formulario y una vez completado, entramos en ella, en la sección ‘Keys and Access Tokens’, y dentro de ella copiamos los valores de ‘Consumer Key (API Key)’ y ‘Consumer Secret (API Secret)’. Anexo Obtención de API Key – Twitter.

Terminado todo esto, es hora de abrir Eclipse IDE y crear un nuevo proyecto del tipo ‘Maven Project’ rellenar lo necesario para poder crear el proyecto, y una vez creado, copiar dentro del directorio nuevo el contenido del proyecto con los módulos elegidos anteriormente en [1] para finalmente ejecutar con Maven el comando ‘mvn clean install’.

En este punto tenemos la estructura del proyecto lista para empezar a desarrollar en él. Podemos observar al acceder a Eclipse IDE que tenemos una estructura de carpetas ya creada también, con ‘src/main/java’, ‘src/main/resources’ y ‘src/test/java’, con lo que podemos deducir que en la primera de ellas tendremos clases Java, como nuestros controladores, la segunda correspondería a las propias páginas HTML, nuestras hojas de estilo… y la tercera de ellas correspondería a pruebas. Además, podemos ver un archivo propio de Spring ‘Spring Elements’ donde Spring Framework irá almacenando las distintas anotaciones y estereotipos que usemos durante el desarrollo. Adicionalmente, por seguir un orden, debemos crear dentro de la primera carpeta otras nuevas. La primera, ‘controller’, donde crearemos y organizaremos

nuestros controladores. La segunda, ‘model’, para hacer lo propio con los objetos que nos vayan a servir como modelos.

Como último paso, dentro de la segunda carpeta mencionada anteriormente podemos encontrar un archivo ‘application.properties’, aquí debemos indicarle a Spring las Keys obtenidas antes de Twitter como propiedades de configuración inicial. Podemos ver una lista completa de estas propiedades en [5].

4. ITERACIONES DEL PROYECTO

4.1 PRIMERA ITERACIÓN: OBTENIENDO LA PRIMERA LISTA DE TUITS

En esta primera iteración lo que intentamos conseguir es aquello sobre lo que se apoyará el resto del proyecto, es decir, tuits relacionados con algo en concreto.

Para esto vamos a crear inicialmente dos controladores, uno llamado 'ControladorInicial' y otro 'ControladorTwitter' (Figura 1). El primero se encargará de servirnos de puerta de entrada (o proxy) de la propia aplicación, es decir, implementaremos el patrón de diseño controlador frontal.

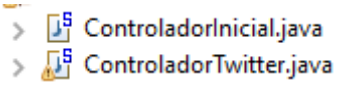

Figura 1: Lista de controladores

En este momento tenemos que comentar las anotaciones ya mencionadas anteriormente de Spring. En este caso, los controladores se anotan con '@Controller', junto a otra etiqueta 'RequestMapping("dirección")' (Figura 2) que es la ruta que nos permite acceder a ese controlador en concreto directamente a través de la barra del navegador.

```
@Controller
@RequestMapping("/twittercontrolador")
public class ControladorTwitter {
```

Figura 1. Anotación del Controlador

Además, los métodos que vamos a usar para interactuar con la vista del proyecto serán todos de tipo ModelAndView (Figura 3), esto es, devuelven un objeto de tipo ModelAndView que, como puede deducirse de su nombre, incluye tanto la vista o el HTML a dónde queremos ir, como atributos requeridos por este HTML.

```
@PostMapping(value="/buscartuits")
public ModelAndView obtenerTuits(@
```

Figura 2: Método ModelAndView

Estos métodos ModelAndView también deben incluir bien una etiqueta 'GetMapping("dirección")' o bien una 'PostMapping("dirección")', dependiendo de si vamos a llamarlos a través de una petición Get o Post y la ruta que lo identifica dentro del propio controlador con respecto a otros métodos.

En este punto del proyecto creamos dos archivos HTML: uno incluyendo una caja de texto para introducir una cadena que contuviese un hashtag para extraer información relacionada con él. Al darle al botón de buscar, pasaríamos a interactuar con el segundo controlador, que es el que (de momento) dirigirá el resto de acciones.

Con este planteamiento en mente, creamos un método en la clase ControladorInicial que nos llevará a la vista donde introduciremos el hashtag a partir del cual queremos obtener la información. Ahora bien, necesitamos pasarle de antemano a la vista los objetos sobre los que ella va a trabajar (en este caso inicial, únicamente el String sobre el que escribir). Aun así, por posibles ampliaciones en el futuro, decidimos crear una clase TextoEnPlano (Figura 4) (algo a mencionar en este momento es que las clases de modelo en String necesitan necesariamente un constructor vacío y que no reciba ningún parámetro), cuyo principal atributo fuesen los Strings necesarios en las vistas, pero contenido en una clase por modularidad y ampliaciones futuras como ya hemos comentado. Por esta razón, finalmente, la llamada al ControladorInicial nos devuelve un objeto de tipo ModelAndView que nos apunta a la vista para introducir el hashtag y, además, le proporcionamos a la vista un objeto de tipo TextoEnPlano con un nombre determinado y que debe ser igual al nombre del elemento en la vista que esperamos rellenar escribiendo. Esto se hace con el método que recibe el objeto ModelAndView addObject ('nombre', 'objeto') (Figura 5) que nos permite nombrar el objeto y pasarlo a la vista.

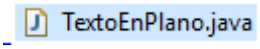

Figura 3: Clase TextoEnPlano

```
mav.addObject("textoplano", new TextoEnPlano());
```

Figura 4: Método addObject(nombre, objeto)

Este elemento sería recibido de la siguiente forma:

En primer lugar, los atributos action de los formularios (form) desaparecen y pasamos a tener un campo 'th: action=" dirección"' que apunta al método del controlador que se ejecutará al enviar el formulario. Junto a él y si enviamos algún parámetro de entrada al método, tenemos otro campo 'th: object=" objeto"' (Figura 6) que nos permite, en primer lugar, establecer qué es lo que recibimos inicialmente del método que nos llevó hasta aquí (en nuestro caso, un objeto de la clase TextoEnPlano procedente del

ControladorInicial); y en segundo, enviar este objeto una vez modificado o no, al controlador que actuará en consecuencia. Por último, dentro del form tenemos un input HTML, pero en este caso con un campo 'th: field=" campo"' (Figura 7) que es el que indica qué campo de todos los posibles del objeto recibido rellenaremos (en nuestro caso, el campo 'texto' del objeto de la clase TextoEnPlano recibido).

```
<form class="navbar-form navbar-left" action="#" th:action="@{/twittercontrolador/buscartuits}" th:object="${textoplano}" method="post">
```

Figura 5: Campos th:action y th:object

```
th:field="*{texto}"
```

Figura 6: Campo th:field

Una distinción que es necesaria hacer para llegar a entender la sintaxis y el dialecto que usamos es que hay tres formas distintas de referenciar objetos o métodos. En primer lugar, tenemos las referencias que empiezan por @ como las del campo th:action, que nos indican que se trata de la llamada a un método localizado en la URL que le sigue. También podemos ver referencias que empiezan por $ lo que significa, que en este caso se está referenciando a un objeto. Por último, tenemos las referencias con * que apuntan a campos dentro de un objeto propio referenciado previamente por la segunda referencia $. Estas tres formas siempre van seguidas de dos llaves {} que contienen bien la URL, el objeto, o el campo en cuestión.

Una vez construido el código de manera correcta, es cuando pasamos al segundo controlador para desarrollar el método que recibirá el objeto TextoEnPlano y buscará los tuits.

Para ello tendremos otro método ModelAndView (en este caso etiquetado como @PostMapping), que nos llevará al finalizar, a una vista donde mostraremos los tuits resultantes. Este método cuenta con una etiqueta nueva hasta ahora, @ModelAttribute (Figura 8), que nos permite indicarle al método el nombre que posee en la vista el objeto que queremos tomar como parámetro, pudiendo ser este tanto un string o un número entero, hasta una clase en concreto como es TextoEnPlano en nuestro caso.

```
@ModelAttribute("textoplano") TextoEnPlano texto){
```

Figura 7: Anotación @ModelAttribute

Una vez recibamos el objeto y tengamos el string que contiene el hashtag a partir del cual queremos obtener la información, es el momento de confeccionar la consulta por así decirlo que nos deberá retornar esa información.

De acuerdo a la documentación disponible en [6], para pasar un hashtag hemos de introducir antes del mismo '%23' y tras él, el hashtag.

Una vez que tenemos la consulta formateada e incluyendo el hashtag, es cuando hacemos uso por primera vez del módulo de Spring Social Twitter para apoyarnos en él y no tener que realizar la petición a la API de Twitter nosotros mismos sino a través de una clase desarrollada específicamente con este objetivo.

Para ello hemos de introducir una nueva etiqueta de Spring, '@Autowired' (Figura 9) que en este caso lo que nos da es la facilidad de saltarnos el proceso de configuración de un objeto y además lo inyecta de manera sencilla. Como lo necesitaremos en más de un sitio posiblemente en el futuro, optamos por hacerlo a nivel de clase y no inyectarlo por ejemplo a nivel de método.

```
@Autowired
private Twitter twitter;
```

Figura 8: Anotación @Autowired

En este momento, a través del objeto de la clase Twitter, accedemos a la colección de sus operaciones y métodos de búsqueda a través del método searchOperations () y una vez dentro, search (). Esto nos devuelve un objeto con una serie de metadatos y una lista de objetos tipo Tweet, con lo que finalmente ejecutamos el método getTweets () (Figura 10) para quedarnos con ellos. Este objeto Tweet sigue siendo propio de Spring y contiene mucha información la gran mayoría irrelevante para este proyecto. Es por ello por lo que creamos una nueva clase de modelo, esta vez 'ObjetoTuit' (Figura 11) en la que sí vamos a conservar los atributos que consideremos necesarios y que son los que siguen: el id del tuit, el texto que contiene, id del usuario, el nombre de usuario, su imagen de perfil, la localización si es que existe, la fecha en que se creó, y el número de retuits y "me gusta" que ha obtenido el tuit en concreto.

```
List<Tweet> listaTuits = twitter.searchOperations().search(resultadoBusqueda).getTweets();
```

Figura 9: Método searchOperations

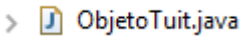

Figura 10: Clase ObjetoTuit

Para finalizar el método simplemente recorremos la lista de Tweet y vamos creando una nueva lista de ObjetoTuit (Figura 12) con los atributos que nos interesaban y la añadimos a la respuesta de igual manera a como añadimos el TextoEnPlano inicial, teniendo cuidado siempre claro, en que el nombre que se manda del objeto y el que hay en la vista que se espera recibir sean iguales.

```
List<ObjetoTuit> listaPropia;
```

Figura 11: Lista de ObjetoTuit

El último paso en esta iteración es confeccionar la segunda vista que mostrará la lista de tuits y que no requiere de dificultad adicional más allá de crear una tabla en HTML. Para desarrollar esta funcionalidad, podemos ayudarnos nuevamente de Thymeleaf y usar el campo ‘th: each=” objeto: colección”’ (Figura 13) para iterar fácilmente sobre la lista que hemos recibido y otros dos nuevos, uno ‘th:text=” objeto.atributo”’ para mostrar tributos de cada ObjetoTuit como el texto, nombre de usuario… y otro ‘th:src=” objeto.atributo”’ (Figura 14) en este caso para interpretarlo como la imagen de usuario.

```
<tbody th:each="tuit : ${listaobjetostuits}">
```

Figura 12: Campo th:each

```
<th><img th:src="${tuit.urlImagenPerfil}" class="img-thumbnail" /></th>
<th th:text="${tuit.localizacionUser}"></th>
```

Figura 13: Campos th:src y th:text

4.1.1 REVISIÓN DE ITERACIÓN

Decidimos mejorar algo la interfaz HTML utilizando unas pequeñas pinceladas del framework Bootstrap añadiéndole a la tabla cebreado y bordes, así como incluyéndola en un div clasificado como container para ajustar los márgenes.

Adicionalmente, se modificó el método que nos confecciona el string a utilizar como petición a la API de Twitter para poder admitir más de un hashtag a la vez. Esto se llevó a cabo añadiendo entre cada '%23hashtag' un 'AND' ya que, inicialmente queríamos que todos los tuits obtenidos contuviesen todos los hashtags introducidos.

4.1.2 BUGS CONOCIDOS

Al obtener la información del texto de un tuit, a menudo encontramos enlaces e hipervínculos que nos están completos. Además, en este punto no tenemos límite a la hora de introducir hashtags a pesar de que en [7] nos recomiendan un máximo de diez.

4.1.3 CASOS DE USO

(Figura 15)

ID	CU-1
Caso de uso	Base de tuits
Descripción	Obtención de tuits relacionados con hashtags desde la API de Twitter
Precondición	-
Escenario principal	1. El usuario introduce los hashtags a buscar y pulsa el correspondiente botón. 2. El usuario es redirigido a una nueva ventana donde se muestran una serie de tuits con ciertos campos como nombre de usuario, fecha, etc. relacionados con ellos.
Escenario alternativo	-

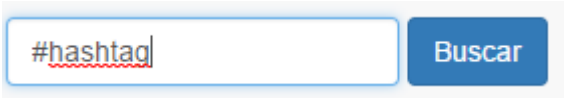

Figura 14: Cajón de búsqueda por hashtags

4.2 SEGUNDA ITERACIÓN: FILTRANDO LISTAS

El objetivo que nos planteamos en esta segunda iteración era poder llegar a filtrar los tuits ya obtenidos anteriormente que contengan una cadena de caracteres en concreto.

Para esto, primeramente, nos planteamos cómo llegar a introducirlo visualmente, y se decidió el introducir una caja de texto nueva, una vez tuviésemos los tuits ya comentados. Esta caja de texto contaría con un botón que iniciaría el proceso de filtrado (Figura 16) y que enviaría un objeto TextoEnPlano que sería el que rellenaríamos escribiendo lo que deseamos introducir como filtro.

Figura 15: Cajón de texto para filtrar

Posteriormente, se introdujo una nueva etiqueta o campo de Thymeleaf, 'th: if=" condición"' (Figura 17) [6], que nos permite elegir qué partes y elementos HTML se mostrarán a la hora de recibir determinados atributos.

```
<span th:if="${botonfiltrar}">
```

Figura 16: Campo th:if

Para poder continuar hemos de realizar una pequeña modificación en el método para buscar los tuits para incorporar una nueva funcionalidad: si tras realizar la búsqueda existe algún elemento de la clase ObjetoTuit (es decir, tenemos algún tuit), activaremos o pondremos a verdadera la condición para mostrar la nueva caja de búsqueda.

Una vez ajustada la lógica ya existente, pasamos a crear la nueva funcionalidad que no consistiría más que en un nuevo método de tipo ModelAndView cuya función sea la de recorrer la lista de elementos de tipo ObjetoTuit y ver, si en su atributo texto, contiene la cadena que hayamos pasado como argumento en la vista en esta nueva caja de búsqueda.

Para poder guardar la información anterior antes del tratamiento de filtrado, introducimos una segunda lista de objetos que será la que se devolverá ahora. Es esta lista la que se añade ahora al modelo con los tuits que cumplieron la condición impuesta. Para finalizar, volvemos a la misma vista como si de una búsqueda de hashtags se tratase.

En este punto de la iteración decidimos permitir varios filtrados de forma sucesiva, con lo que el nuevo cajón de búsqueda estará siempre disponible una vez tengamos tuits.

4.2.1 REVISIÓN DE ITERACIÓN

En esta revisión, tras realizar un filtro, añadimos un nuevo botón que nos permite eliminarlo (Figura 18), es decir, volver a la lista que teníamos en el estado inmediatamente anterior. Llevamos a cabo este proceso a través de un método auxiliar que se encarga de intercambiar los contenidos de dos listas entre sí mismas, y lo llevamos a cabo entre la antigua y la nueva lista de elementos ObjetoTuit.

Figura 17: Botón para eliminar filtro

4.2.2 BUGS CONOCIDOS

En esta iteración encontramos varios bugs, además de seguir contando con los que ya teníamos de la iteración anterior. En primer lugar, tenemos el hecho de no controlar la aparición del cajón para el filtro, sino que lo mostramos siempre, con lo que, aunque el filtro no devuelva ningún tuit que cumpla la condición, seguiremos pudiendo filtrar sobre una lista vacía. Además, al intercambiar las listas (Figura 19) tras eliminar el filtro, en los casos en que filtramos más de una vez, perdemos la lista que obteníamos originalmente. No obstante, son fallos presumiblemente fáciles de remediar con lo que esperamos corregirlos en la siguiente iteración. Por último, al realizarse el filtrado mediante el método “contains”, esto implica que la secuencia de caracteres debe encontrarse de manera idéntica dentro del texto y, por ejemplo, al buscar una cadena ‘abc’ y encontrar en el texto una ‘aBc’, esta no sería tomada en consideración. Esto se llega a agravar un poco más al comprobar que la API de Twitter no funciona así y nos devuelve como tuits relacionados con un hashtag #abc todas las posibilidades y

combinaciones de mayúsculas y minúsculas, esto es, desde #abc hasta #ABC y, en este momento, no controlamos esas situaciones. Por último, por el propio método "contains" no nos es posible filtrar varios elementos a la vez, sino uno solamente.

```
private void intercambiarListas(List<ObjetoTuit> l1, List<ObjetoTuit> l2){
    List<ObjetoTuit> listaAux = new ArrayList<ObjetoTuit>(l1);
    l1.clear();
    l1.addAll(l2);
    l2.clear();
    l2.addAll(listaAux);
}
```

Figura 18: Método para intercambiar dos listas

4.2.3 CASOS DE USO

ID	CU-2
Caso de uso	Filtro de tuits
Descripción	Obtención de tuits mediante criba sobre una lista ya existente de ellos
Precondición	Haber realizado una búsqueda en algún momento anterior (CU-1)
Escenario principal	1. El usuario introduce la palabra para usar a modo de filtro y pulsa el botón que iniciará el proceso de filtro. 2. El usuario es redirigido a la misma ventana esta vez con la lista de tuits que contengan el filtro introducido.
Escenario alternativo	-

ID	CU-3
Caso de uso	Eliminación de filtro
Descripción	Obtención de lista de tuits previa a la realización del filtro
Precondición	Haber realizado un filtro previamente (CU-2)
Escenario principal	1. El usuario pulsa el botón para eliminar el filtro existente. 2. El usuario vuelve a la misma ventana esta vez con la lista que existía antes de realizar este filtro.
Escenario alternativo	

4.3 TERCERA ITERACIÓN: BÚSQUEDA POR USUARIO/AUTOR

Llegamos a la tercera iteración con una idea simple y una necesidad. Podemos tener el escenario de estar siguiendo información relacionada con un hashtag y observar a algún usuario que está aportando, en cada momento, nuevos tuits relativos a lo que nos interesa conocer en ese momento. Sin embargo, no podemos centrarnos únicamente en lo que escriba dicho usuario, sino que actualmente solo nos podemos centrar o fijar en hashtags. Por esto, lo que buscamos en este momento es poder introducir un nombre de usuario para poder ver los tuits escritos en concreto por una persona, independientemente de los hashtags que incluya.

Lo primero que debemos hacer es comprobar si la petición a la API de Twitter sigue el mismo esquema para hashtags que para usuarios. En [6] podemos verificar que se trata de peticiones realizadas de distinta forma, ya que, en este caso, el patrón que se sigue es el de “from: usuario”.

Por esta razón, no queda otra que realizar un método similar. En primer lugar, añadimos al lado de la caja de texto y botón de búsqueda iniciales para hashtags, otros iguales, en este caso, para el nombre de usuario (Figura 20). A su vez, esta caja de búsqueda, al igual que ya lo hacía el primero y como también pasaba en el caso del filtro, recibe otro TextoEnPlano para rellenar el parámetro que pasaremos al controlador.

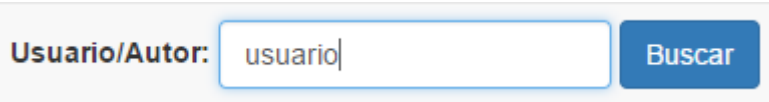

Figura 19: Cajón de texto para búsqueda por usuario

Una vez en el controlador, la secuencia es la misma al caso anterior de los tuits: el objeto Twitter se encarga de devolvernos una lista de tuits que pasaremos a la vista para mostrarla al usuario.

Además, al haber sido una funcionalidad algo simple o sencilla, ya que gran parte estaba ya hecha por ser muy similar a la búsqueda de tuits, decidimos intentar solventar algunos de los bugs ya existentes. En primer lugar, a la hora de recibir

hashtags para realizar una búsqueda se lleva a cabo un proceso de separación en tokens, por el cual permitimos un máximo de cinco de ellos. Por otro lado, al filtrar controlamos ahora que se hayan obtenido tuits que cumplan la condición o, de lo contrario, no aparecerá el botón de seguir filtrando.

4.3.1 REVISIÓN DE ITERACIÓN

Al tratarse de una búsqueda por usuario, nos centramos en una única cuenta en este caso, por lo que, para evitar problemas en caso de introducir más de una palabra o nombres de usuario, separar en tokens el objeto TextoEnPlano que nos llega y nos quedamos con el primer token como parámetro de entrada o nombre de usuario escogido para la búsqueda.

4.3.2 BUGS CONOCIDOS

Volvemos a tener los que ya existían y que no han sido corregidos, ya que esto no es más que otra búsqueda de tuits como lo fue la primera.

4.3.3 CASOS DE USO

ID	CU-4
Caso de uso	Búsqueda de usuario
Descripción	Obtención de lista de tuits emitidos por un usuario concreto desde la API de Twitter
Precondición	-
Escenario principal	1. El usuario introduce el nombre de usuario en el cajón de búsqueda correspondiente y pulsa el botón. 2. El usuario es redirigido a la ventana principal donde le aparecen la serie de tuits que el usuario en concreto escribió
Escenario alternativo	-

4.4 CUARTA ITERACIÓN: INTRODUCIENDO MAPA EN EL CÓDIGO Y REMODELANDO VISTA

Por primera vez, en esta cuarta iteración tenemos dos objetivos o ideas principales. Por un lado, queremos introducir un mapa para mostrar la ubicación en la que se mandan los tuits. Por otro lado, cambiaremos el propio diseño de la aplicación, cambiando algunos elementos e introduciendo otros nuevos.

Empezando por el primer objeto de los propuestos, el primer paso es ver qué necesitaríamos para la inclusión de este mapa en nuestro proyecto. En [8] se muestran varias posibilidades para ello, de las que escogeríamos el mapa a través de la API de JavaScript. Tras esto, finalmente se nos muestran tres tipos de mapas posibles: marcador, visualizar datos y clústeres. Escogemos la opción de marcador ya que es lo que buscamos: establecer un marcador en cada posible ubicación. Para ello es necesario, además, obtener una clave de API y activarla para poder disfrutar de las funciones deseadas (Anexo Obtención de API Key – Google)

Conseguido este paso, la aplicación está preparada para trabajar con el mapa, que ya se muestra (Figura 21) en la vista, siendo nuestra idea mostrarlo siempre, vacío en caso de no tener tuits en nuestra tabla, y relleno con las ubicaciones que existan de cada tuit, en caso de tener algunos contenidos en nuestra tabla.

Figura 20: Mapa en blanco

Para empezar el proceso, recorremos las listas de tuits tanto al realizar una búsqueda, como al filtrar la lista o eliminar el filtro. Lo que comprobamos es que haya ubicación, y, en ese caso, la añadimos a una lista de tipo String que posteriormente, si tiene algún elemento, pasaremos a la vista (Figura 22). Esta lista, como puede resultar evidente, es actualizada con las nuevas ubicaciones cada vez que se llama al método de buscar, filtrar o eliminar el filtro.

```
List<String> ubicaciones = new ArrayList<String>();
for(int i=0;i<listaPropia.size();i++){
    String ubicacionAux = listaPropia.get(i).getLocalizacionUser();
    if(!ubicacionAux.isEmpty()){
        ubicaciones.add(ubicacionAux);
    }
}
```

Figura 21: Ubicación de cada tuit

Una vez en la vista, modificamos la función JavaScript que en la API de Google Maps permite recorrer el array pasado por el controlador y, de esta forma, realizar un proceso de geocodificación con ayuda de la API de Google Maps, para obtener unas coordenadas de latitud y longitud (Figura 23). Con estas coordenadas el último paso es colocar un marcador en esa dirección y repetir el proceso por cada ubicación que tengamos.

```
geocoder.geocode({'address': element}, function(results, status){
```

Figura 22: Función geocode de la API de Google Maps JS

Algo a comentar es que, al depender de la API de Google para el proceso de geocodificación, los valores basura que pudiéramos encontrar en la ubicación, tales como direcciones falsas, mensajes sin sentido, etc. son cribados por ella.

Por desgracia, en esta ocasión y, tras intentarlo de varias maneras distintas, no conseguimos pasar a través de Thymeleaf de manera correcta los datos para obtener la información adecuada utilizando JavaScript en la vista. Es decir, hemos intentado pasarlo como parámetros como si de una lista normal se tratase, pero no obtenemos dato alguno en la vista para tomarlo por JavaScript. También hemos intentado tener una variable ya creada en la vista y asignar los datos a ella al devolver las ubicaciones y tampoco ha surtido efecto: la variable estaba vacía. Como último recurso intentamos crear en el controlador una variable de tipo String que simulase ser una variable JavaScript, simulando a su vez entre comillas las etiquetas necesarias y siguiendo el esquema de JavaScript (“var ubicaciones=…). Sin embargo, esta solución tampoco tuvo éxito, por lo que en este objetivo queda pendiente el paso de información para interpretarla de manera adecuada.

Pasamos al segundo objetivo, remodelación del diseño y la vista.

El primer paso que consideramos oportuno y correcto es integrar las cajas y botones de búsqueda tanto por hashtag como por nombre de usuario en la misma vista en la que tenemos la tabla y la posterior caja de texto para filtrar.

Estos elementos los colocamos en una barra de navegación en la parte más superior de la página, añadiendo también a la derecha de la misma un mensaje de “TFG – Adrian García Humanes”.

El último paso consiste en la adición de un botón resaltado en rojo que se encargará de limpiar las listas y con ello la tabla (Figura 24).

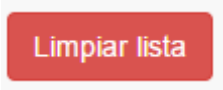

Figura 23: Botón para limpiar la lista de tuits

Para ello creamos una nueva función que nos limpie las listas de tuits y nos devuelva al escenario inicial de la aplicación. Además, este botón estará disponible siempre que se haya realizado una búsqueda con anterioridad, permitiendo la limpieza, una vez obtenemos los tuits, o tras realizar algún filtrado, etc.

4.4.1 REVISIÓN DE ITERACIÓN

En esta revisión nos centramos únicamente en bugs ya existentes, por lo que eliminamos la posibilidad de filtrados sucesivos, es decir, permitiremos realizar un filtrado, y tras ello únicamente eliminar este filtro, no volver a realizar un filtro sobre la lista ya filtrada.

4.4.2 BUGS ENCONTRADOS

Encontramos un nuevo bug añadido en este punto: la imposibilidad de pasar, de forma apropiada, las ubicaciones del controlador a la lista. Junto a ella, todos los anteriores no resueltos.

4.4.3 CASOS DE USO

ID	CU-5
Caso de uso	Limpieza general
Descripción	Volver al momento inicial antes de realizar ninguna acción
Precondición	Haber realizado una búsqueda de tuits con anterioridad
Escenario principal	1. El usuario pulsa el botón de limpiar lista. 2. El usuario es redirigido a la ventana principal donde todo y todas las variables están reseteadas y tal y como se encontraban al inicio tras entrar a la ventana sin realizar ninguna acción.
Escenario alternativo	

4.5 QUINTA ITERACIÓN: AÑADIENDO NUEVOS ELEMENTOS A LA BÚSQUEDA

En esta quinta iteración del proyecto nos planteamos un supuesto escenario en el que realizásemos una búsqueda cualquiera. Una vez hecha esta consulta, queremos tener en consideración también un nuevo hashtag o menciones a cierto usuario. En este punto de desarrollo no tendríamos más remedio que volver a introducir todos los elementos que ya teníamos y buscar de nuevo los tuits. Esto plantea varios inconvenientes o incomodidades. Uno de ellos puede ser que el usuario tal vez no llegue a recordar la totalidad de elementos para la búsqueda que introdujo inicialmente, con lo que, aunque por un lado añadamos el nuevo, perderíamos ese supuesto elemento olvidado. Otro inconveniente puede ser la recursividad de este problema en sí mismo, es decir, que al introducir un nuevo elemento volvamos a pensar en añadir otro y de nuevo tengamos que volver a escribir todos los elementos que teníamos, más el que acabábamos de añadir, más este nuevo elemento, con lo que termina resultando un proceso lento y tedioso.

Por todo esto es que surge la idea de considerar dos escenarios. El primero es realizar una búsqueda alcanzando el número máximo de elementos permitidos para ella, es decir, cinco. El segundo escenario es otra búsqueda, una en la que utilicemos un número de elementos inferior al límite. En este caso, sería interesante el añadir un

nuevo cajón de texto para la adición de este nuevo elemento a la búsqueda, eliminando así la necesidad de, por un lado, tener que recordar los elementos ya introducidos antes y, por otro lado, el tener que volver a escribir los que ya usábamos este nuevo elemento. Por último, decidimos no aplicar esta funcionalidad a la búsqueda por usuario, quedándose esta tal y como estaba, es decir, admitiendo un único nombre de usuario como parámetro de búsqueda.

En esto consiste nuestra quinta iteración. Para llevarla a cabo, establecemos una variable en nuestro controlador que cuente el número de elementos introducidos ya para realizar búsquedas, y en caso de que este número sea menor a cinco, cuando realizamos una búsqueda y regresamos a la vista ya con los tuits y la tabla con los datos de estos, se nos muestra una nueva caja de texto con su correspondiente botón para añadir el nuevo elemento, al lado del que ya teníamos para filtrar los tuits. Por cuestiones de diseño y de diferenciar algo ambas funcionalidades, al tener las dos un botón de tipo 'btn-success' proporcionado por Bootstrap, en el caso de filtrar las palabras le hemos establecido a la caja de texto un borde tipo 'solid 2px black' y al de añadir un nuevo elemento de tipo 'dotted 2px black' (Figura 25). Por señalar de manera clara el botón de eliminación de filtro, le establecemos el tipo de Bootstrap de 'btn-warning'. Ya, por último, y por darles un poco de estilo a todos los botones que tenemos, tanto al de búsqueda por hashtags o menciones como al de búsqueda por nombre de usuario, le establecemos el tipo de 'btn-primary' y al de eliminar o limpiar la lista de tipo 'btn-danger', quedándonos con un diseño tan simple como efectivo para distinguir bien unos de otros.

Figura 24: Cajón para añadir nuevo elemento como parámetro de búsqueda

Pasamos a la parte relativa al controlador. De nuevo, tenemos un método similar a los anteriores en varios aspectos. Vuelve a la misma página, accedemos a él de nuevo a través de una petición post pasándole como parámetro un nuevo TextoEnPlano, etc.

Para poder volver a usar los elementos que ya teníamos, además del número que nos indica cuántos son, necesitamos una lista de String que los contengan para volver a introducirlos en la petición a Twitter. No obstante, no nos convence mucho esta solución y se remodelará en una futura iteración.

Con todo esto, el método añadiría el nuevo elemento a la lista de los que ya hay y aumentaría en uno el número de ellos, para seguir añadiendo o no la opción de añadir uno siguiente en la vista.

A partir de aquí no difiere del procedimiento de una búsqueda normal, es decir, realizamos la petición de forma análoga y siguiendo el esquema que podíamos encontrar en [6] usando '%23' y demás, y los volvemos a añadir a la lista tal y como ya hacíamos.

4.5.1 REVISIÓN DE ITERACIÓN

Se nos ocurrió la idea o posibilidad de que el usuario no quisiese introducir un solo nuevo elemento, sino varios, con lo que volveríamos a tener el problema de tener un proceso tedioso para llevar a cabo esto, con lo que nos planteamos remodelar el método para añadir varios. La modificación viene en la parte en la que añadimos el parámetro, en lugar de eso, dividir en tokens el texto que nos viene añadiéndolos a la lista bien hasta que terminamos de hacerlo o bien hasta que alcanzamos el límite de cinco de ellos en total. Con esto ahorramos tiempo y reducimos el número de peticiones a Twitter para añadir más de un solo elemento.

4.5.2 BUGS ENCONTRADOS

Tenemos un nuevo bug que consiste en introducir un hashtag que ya tengamos como parámetro de búsqueda. En ningún momento comprobamos que esté o no repetido, por lo que podemos agotar el límite de elementos de búsqueda en el peor caso con uno que se repita cinco veces.

4.5.3 CASOS DE USO

ID	CU-6
Caso de uso	Adición de nuevo elemento o elementos
Descripción	Añadir uno o varios elementos a los ya usados para realizar una búsqueda.
Precondición	Haber realizado una búsqueda de tuits con anterioridad y no haber alcanzado el número máximo de elementos de búsqueda.
Escenario principal	1. El usuario introduce el o los elementos nuevos de búsqueda y pulsa el botón. 2. El usuario es redirigido a la ventana principal con una nueva lista de tuits obtenidos a partir de los elementos ya existentes y los nuevos introducidos ahora, con la posibilidad de añadir más elementos al no alcanzar el límite de los mismos.
Escenario alternativo	1. El usuario introduce el o los elementos nuevos de búsqueda y pulsa el botón. 2. El usuario es redirigido a la ventana principal con una nueva lista de tuits obtenidos a partir de elementos ya existentes y los nuevos introducidos ahora, sin poder añadir ninguno más al alcanzar el límite de cinco elementos.

4.6 SÉPTIMA ITERACIÓN: MEJORANDO LA ESTRUCTURA DE CLASES CON RESPECTO A LOS ELEMENTOS

En esta iteración intentamos corregir algunos problemas de diseño estructural arrastrados de alguna iteración anterior. Ya comentamos que la idea de tener los hashtags guardados en una lista de String, así como el número de ellos en el propio controlador no nos parecía una solución adecuada, con lo que vamos a pasar a intentar modularizarla en una clase distinta.

Para ello creamos una clase ‘ListaDe5Elementos’ (Figura 26), en la que tendremos cinco String distintos, uno por cada elemento posible, y un contador para saber el número de ellos (Figura 27). Adicionalmente le añadimos a esta clase un nuevo método para formar la cadena de hashtags para realizar la petición a la API (Figura 28), así como un método para obtener cada uno de los elementos por separado.

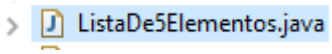

Figura 25: Clase ListaDe5Elementos

```
private String elemento1;
private String elemento2;
private String elemento3;
private String elemento4;
private String elemento5;

private int numElementos = 0;
```

Figura 26: Atributos de la clase ListaDe5Elementos

```
public String montarCadena(){
    StringBuilder sb = new StringBuilder();
    int contador = this.getNumElementos();
    while(contador>0){
        sb.append("%23");
        sb.append(this.getHashtagSwitch(contador));
        contador--;
        if(contador>0){
            sb.append(" OR ");
        }
    }
    return sb.toString();
}
```

Figura 27: Método para formar la petición a la API de Twitter

Pasando a la corrección de anteriores bugs, vamos a eliminar el que nos permitía insertar varios hashtags idénticos como parámetro de búsqueda. Para ello vamos a nuestra nueva clase y creamos un método que recibe cada uno de los elementos nuevos y comprueba que no estén ya almacenados, salvo en el caso de que haya cero elementos, que devolverá que nunca está y en el caso de que ya tengamos los cinco huecos ocupados, en cuyo caso no importará ya que nunca insertaremos nada más.

Por otra parte, y mientras probábamos a introducir elementos, descubrimos que, por el propio funcionamiento que tiene la API de Twitter y aunque no se mencione en [6], podemos introducir un nombre de usuario como uno de los cinco elementos y el funcionamiento que obtenemos es, el que ya teníamos de buscar tuits relacionados con hashtags, y ahora también el de obtener los tuits que contengan menciones a este usuario.

En un primer momento pensamos que se trataba de la misma funcionalidad que la segunda caja y botón de búsqueda que tenemos por usuario, pero la diferencia radica en que, si introducimos un @nombredeusuario en el cajón para búsquedas por usuario, obtendremos los tuits escritos por este usuario, mientras que en este nuevo caso lo que recibimos son los tuits que lo mencionen a él, que pueden estar escritos por cualquier usuario (Figura 29).

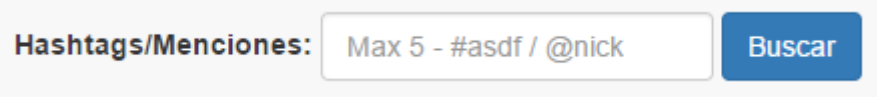

Figura 28: Cajón de búsqueda de hashtags o menciones a usuarios

4.6.1 REVISIÓN DE ITERACIÓN

En un primer momento no creíamos necesaria una revisión de esta iteración, pero tras probar varias combinaciones y formas de hacer fallar la aplicación es posible ver que, aunque a la hora de introducir nuevos elementos (ahora ya hashtags o menciones a usuarios) no es posible repetir un elemento, sí lo es a la hora de los elementos que se introducen por primera vez al realizar una búsqueda, por lo que modificamos un poco el código para que se realice una comprobación también en los primeros elementos.

4.6.2 BUGS CONOCIDOS

Todos los que existían desde iteraciones anteriores a excepción del que hemos corregido en esta.

4.6.3 CASOS DE USO

ID	CU-7(CU-1 SEGUNDA VERSION)
Caso de uso	Base de tuits segunda versión
Descripción	Obtención de tuits relacionados con hashtags y menciones a usuarios desde la API de Twitter
Precondición	-
Escenario principal	1. El usuario introduce los hashtags o nombres de usuario a buscar y pulsa el correspondiente botón. 2. El usuario es redirigido a una nueva ventana donde se muestran una serie de tuits con ciertos campos como nombre de usuario, fecha, etc. relacionados con estos hashtags (Ninguno de ellos tomado varias veces por repetición).

Escenario alternativo	-

4.7 OCTAVA ITERACIÓN: INCLUYENDO LAS UBICACIONES EN EL MAPA

El propósito de esta iteración es corregir o completar parte de lo que intentábamos en la iteración número cuatro, es decir, el tener un mapa funcional que nos muestre las ubicaciones relativas a los tuits. En aquel momento no logramos pasar de manera correcta los datos a través de Thymeleaf para su correcta interpretación en JavaScript, a pesar de probarlo de varias maneras distintas como un simple paso de parámetros normal, intentar asignar nuestros valores a una variable JavaScript o incluso simular una a través de un String,

La solución finalmente obtenida y funcional pasar por la utilización de secciones CDATA (Figura 30) (Figura 31). Estas son necesarias si necesitamos parsear algo como XML y no como HTML, es decir, lo que introduzcamos en su interior será ignorado a la hora de parsear. Por poner un ejemplo más claro sabemos que las expresiones que contienen un carácter como '<' son inválidas para usar en nuestra página y hemos de usar en este caso concreto '<'. De esta manera evitaríamos este problema en este caso concreto de ubicaciones que nos atañe.

```
/*<![CDATA[*/
```

Figura 29: Comienzo de sección CDATA

```
/*]]>*/
```

Figura 30: Fin de sección CDATA

Con el problema ya solucionado, el resto de la lógica es bien sencillo. En primer lugar, ahora tomamos el valor de la variable ubicaciones (que contiene la lista de String de las mismas) insertándola entre dos corchetes, esto es, [[${ubicaciones}]] (Figura 32). Con esto tenemos los datos ya en JavaScript de manera correcta. El siguiente paso es recorrer el array y realiza la función de geocode para obtener una posición ya válida a partir del String, siempre que este también sea válido, y una vez tengamos la posición, establecer un nuevo marker en nuestro mapa. Como es posible que tengamos ubicaciones en sitios de lo más diversos ya que la aplicación es aplicable a todo el territorio mundial, hemos optado por centrarla sobre Málaga y reducir el zoom

hasta un nivel que nos permita observar todo el mapa a primera vista, y a partir de aquí, hacer más zoom hacia donde nos interese según queramos. (Figura 33)

```
var arrayDirecciones = [[${ubicaciones}]];
```

Figura 31: Variable arrayDirecciones

```
<div id="map" style="margin-bottom:10px"></div>
<script th:inline="javascript">
  /*<![CDATA[*/
          function initMap() {
            var map = new google.maps.Map(document.getElementById('map'), {
              zoom: 2,
              center: {lat: 36.720, lng: -4.420}
            });
            var geocoder = new google.maps.Geocoder();
            geocodeAddress(geocoder, map);
          }

          function geocodeAddress(geocoder, resultsMap) {
              var arrayDirecciones = [[${ubicaciones}]];
              for(var i= arrayDirecciones.length >>> 0; i--;){
              }
              arrayDirecciones.forEach(function(element){
                  geocoder.geocode({'address': element}, function(results, status){
                      if(status=='OK'){
                         var marker = new google.maps.Marker({
                              map:resultsMap,
                              position: results[0].geometry.location
                          });
                      }
                  });
              });
          }
  /*]]>*/
</script>
```

Figura 32: Método para posicionar las ubicaciones en mapa

Tal y como comentamos en la iteración que correspondía a esta funcionalidad, esta operación de obtener las ubicaciones y marcarlas en el mapa se realizará siempre que trabajemos con nuevas listas o modificaciones de ellas, es decir, al obtener tuits, filtrarlos, añadir un nuevo elemento, etc.

Para acabar, nos gustaría mencionar que perdemos un poco de eficiencia en el caso de búsquedas por autor o usuario, ya que, en el caso de tener ubicación, todas van a ser la misma.

4.7.1 REVISIÓN DE ITERACIÓN

No se han realizado cambios tras la revisión de esta iteración

4.7.2 BUGS CONOCIDOS

Todos los ya existentes menos los que acabamos de corregir, en esta iteración no hemos añadido o descubierto nuevos.

4.7.3 CASOS DE USO

En esta ocasión no añadimos ningún caso de uso nuevo, ya que no podemos usar la funcionalidad del mapa por sí misma suelta, sino que es un dato más de cada tuit.

4.8 NOVENA ITERACIÓN: MEJORANDO LOS FILTROS, LA INTERPRETACIÓN DE LOS TUITS Y FACILITANDO IDENTIFICACIÓN VISUAL DE ELEMENTOS

Vamos a tomar esta iteración como una forma de mejorar y añadir varias funcionalidades que mejoran o dan valor a la aplicación, antes de pasar a desarrollar la funcionalidad de almacenar y recuperar tuits desde una base de datos.

Nos planteamos varios objetivos. El primer de ellos es mejorar la manera de filtrar que tenemos, para admitir pequeñas variaciones como mayúsculas o minúsculas. El segundo consistiría en interpretar si hay o no algún hipervínculo o nombre de usuario en el texto y cambiarlos por links reales hacia esas direcciones o perfiles de Twitter, así como hacer lo propio con el nombre de usuario perteneciente a cada uno de los tuits que traigamos. El tercer punto pasaría por marcar con un color distinto cada uno de los cinco posibles elementos que tengamos para una mejor y más rápida identificación visual.

Empezamos por la primera de ellas. Hasta este momento, el filtro de que disponemos funciona de acuerdo a la función contains, además de que no nos permite por el mismo motivo filtrar varias palabras. Vamos por ello a cambiar el método por completo. El primer paso que realizamos es dividir en tokens cada tuit para obtener cada palabra individualmente, así como separar en tokens lo que nos llega por el filtro. Además, como mostramos solo los tuits que cumplan o tengan todos los filtros, también nos sirve ahora para mostrar los tuits que contengan varios tuits a la vez y no solo algunos de ellos. Nos basta con escribir como filtros los hashtags que nos interesen que estén en los tuits al mismo tiempo. Tras esto, vamos comparando por cada palabra del filtro, todas las del texto, pasando tanto la palabra filtro como cada una de las del texto a minúscula, para evitar que por alguna letra que no sea igual pero sí lo sea la palabra

no se nos muestre, y las comparamos ahora con el método “equals”. Con todo ello tenemos una forma de filtrar mucho más adecuada y versátil, capaz de filtrar por varias palabras de manera estricta y admitiendo variaciones en minúsculas y mayúsculas (Figura 34).

```
if(contAux>0 && (palabraAuxiliar.toLowerCase()).equals((this.listaDe5Elementos.getElemento1().toLowerCase()))){
```

Figura 33: Nueva forma de comparar palabras para filtrar

Pasamos ahora al segundo objetivo, la interpretación de tuits para detectar hipervínculos y nombres de usuario.

Este proceso es similar al anterior. Cada vez que recibamos u obtengamos nuevos tuits (De lo que deducimos que este proceso no se lleva a cabo si venimos de realizar un filtro o eliminar el mismo) procedemos a un análisis por cada palabra. Si detectamos ‘https’ o ‘@’ y en este último caso la longitud es mayor a eso (es decir, no se trata del propio carácter ‘@’ solo), pasamos a cambiar en el texto por otras cadenas de caracteres que realicen las funciones que queramos.

En el caso de nombres de usuario, sabemos que el patrón que debemos seguir y que podemos ver en la URL al entrar a algún perfil de Twitter es ‘https://twitter.com/nombre’ siendo este el nombre de usuario sin el carácter ‘@’ por lo que, en caso de detectar un nombre de usuario, no tenemos más que cambiar ese fragmento de texto dentro del tuit, por una etiqueta <a href> que apunte a esa dirección quitando el carácter ‘@’ y mostrando en el texto el nombre de usuario en lugar de toda la URL.

En el segundo caso, el de los hipervínculos, es más sencillo ya que solo tenemos que encerrarlos en la misma etiqueta y mostrar el mismo hipervínculo en el texto ya que no podemos cambiarlo por ningún texto identificativo. Tal vez una posibilidad fuese la de sustituirlo visualmente por ‘Link externo’ pero finalmente se ha preferido dejar esta opción.

Llegando ya al tercer objetivo, nos encontramos con la idea de colorear de manera distinta cada uno de los cinco elementos posibles para buscar. En primer lugar, establecemos estos colores en nuestro HTML como clases con estilos CSS (no utilizamos ids sino clases porque presuponemos que habrá más de uno por cada elemento) (Figura 35). De nuevo nos encontramos con que debemos analizar cada

una de las palabras y ver si se corresponden con alguno de los elementos que ya tenemos en nuestra clase ListaDe5Elementos. En tal caso, comprobamos cuál de los elementos es, y, en consecuencia, sustituiríamos esa palabra por ella misma encerrada en una etiqueta etiquetada con la clase del elemento que sea, a saber, desde 'elem1' hasta 'elem5', y ya en la vista se le aplicará un color distinto según el caso (Figura 36).

```
String palabraTransformada = "<span class=\"elem1\">"+palabraAuxiliar+"</span> ";
resultado = resultado + palabraTransformada;
```

Figura 34: Modificación de palabra para poder cambiar el color en la vista

```
<style>
    .elem1 { color: 'red' };
    .elem2 { color: 'blue'};
    .elem3 { color: 'green'};
    .elem4 { color: 'yellow'};
    .elem5 { color: 'brown'};
</style>
```

Figura 35: Colores a usar en la vista para resaltar elementos

4.8.1 REVISIÓN DE ITERACIÓN

Tras probar las funcionalidades anteriores, se descubrió un problema de nuevo con Thymeleaf, consistente en no interpretar las etiquetas que se le pasaban desde el controlador. Es decir, tras la sustitución de palabras por ellas mismas con las correspondientes etiquetas bien por el color, o bien por ser un hipervínculo se nos mostraban en la vista tal cual, en lugar de interpretarse como etiquetas HTML. Para solucionar esto, recurrimos a cambiar las etiquetas de esas columnas de la tabla de th:text a th:utext (Figura 37). Esta etiqueta nos permite que lo que vaya dentro de ella sea interpretado y las etiquetas se transforme en lo que deban, en lugar de escaparlas y mostrarlas tal cual. Con esto resolvemos nuestro problema, aunque es una solución de la que no se debe abusar, ya que puede acarrear problemas de seguridad, aunque es cierto que en nuestro caso lo usamos en un texto con la única finalidad de mostrarlo, sin permitir modificaciones sobre él.

```
<th th:utext="${tuit.texto}"></th>
<th th:utext="${tuit.nombreUsuario}"></th>
```

Figura 36: Nuevo campo th:utext

Además, en caso de que algún elemento fuese una mención a un usuario y no un hashtag, no se mostraba visualmente con el color correspondiente, sino que aparecía

como un hipervínculo más. Por esto, en caso de que algún elemento sea una mención a un usuario realizamos dos transformaciones sobre él: por un lado, le adjuntamos el hipervínculo necesario y por otro, etiquetamos a su vez el texto que se mostrará con la etiqueta de clase ‘elem’ que le corresponda (Figura 38).

```
if(contAux>0 && (palabraAuxiliar.toLowerCase()).equals((this.listaDe5Elementos.getElemento1().toLowerCase()))){
    if(palabraAuxiliar.charAt(0)=='@'){
        String nickname = palabraAuxiliar.substring(1);
        String palabraTransformada = "<a href=\"https://twitter.com/"+nickname+"\"><span class=\"elem1\">"+palabraAuxiliar+"</span></a> ";
        resultado = resultado + palabraTransformada;
```

Figura 37: Transformación de mención a usuario en hipervínculo con color

Por último, detectamos un pequeño problema a la hora de establecer los hipervínculos. En algunas ocasiones, por las limitaciones de tamaño de Twitter, a pesar de mostrar 140 caracteres, es posible tener algunos más y mostrar hipervínculos que se truncan al llegar a esta cifra con puntos suspensivos, pero aun así siguen siendo completamente funcionales y podemos navegar hacia esos enlaces. En nuestro caso recibimos por parte de la API de Twitter estos puntos suspensivos que a veces están al principio de la URL tras ‘https.//t.co’, o a veces casi al final, por lo que realizamos varios procesos de filtrado y criba para intentar eliminar los posibles links y enlaces fantasma (Figura 39).

```
if(palabraAuxiliar.contains("...")){
    resultado=resultado+" ";
```

Figura 38: Eliminación de palabras incompletas

4.8.2 BUGS CONOCIDOS

En este momento no tenemos ningún bug de los mencionados en iteraciones pasadas aun sin corregir ni hemos detectado otros nuevos.

4.8.3 CASOS DE USO

En esta ocasión, tanto el añadir color como interpretar los hipervínculos y enlaces a perfiles no son nada nuevo, sino que lo consideramos más bien un valor añadido a la vista y al propio texto que ya teníamos. Pero el poder filtrar varios elementos y obviar pequeñas diferencias en cuanto a mayúsculas o no sí lo consideramos una nueva funcionalidad.

ID	CU-8 (CU-2 SEGUNDA VERSION)
Caso de uso	Filtro de tuits segunda versión
Descripción	Obtención de tuits mediante criba con varios elementos sobre una lista ya existente de ellos
Precondición	Haber realizado una búsqueda en algún momento anterior (CU-1/CU-7)
Escenario principal	1. El usuario introduce la palabra o palabras para usar a modo de filtro y pulsa el botón que iniciará el proceso de filtro. 2. El usuario es redirigido a la misma ventana esta vez con la lista de tuits que contengan todos los filtros introducidos, sin importar si hay ligeras variaciones en cuanto a mayúsculas y minúsculas.
Escenario alternativo	-

4.9 DÉCIMA ITERACIÓN: ALMACENANDO Y RECUPERANDO TUITS

Con esta novena iteración llegamos al último objetivo que nos marcamos cuando iniciamos este proyecto: el poder almacenar para su posterior consulta, los tuits que tengamos en nuestra tabla en un determinado momento.

Tal y como comentábamos en las primeras páginas de este documento, la base de datos elegida es MongoDB, de tipo no relacional.

Para empezar con esta iteración lo primero es realizar la instalación de MongoDB de acuerdo a [9] y al Manual de instalación.

Con esto ya hecho, el siguiente paso es ayudarnos de un último módulo de Spring que incorpora utilidades para esta base de datos. Para llevar eso a cabo, siguiendo las indicaciones en [10] no tenemos más que actualizar nuestro pom.xml de manera correcta y, usando Maven, traer estas librerías y funcionalidades sin mayor problema.

Tras esto, debemos crearnos una nueva clase, en este caso una interfaz, que nos servirá para realizar las consultas de una manera simple y rápida a través de ella. Con lo que, nos creamos nuestra interfaz 'ObjetoTuitRepository' que, en este caso, extiende de MongoRepository<ObjetoTuit, String> entendiendo por esto que trabajará con objetos de tipo ObjetoTuit según la sentencia y el parámetro String recibido.

Por último, no tenemos más que crear un método en la interfaz de una manera clara y especificando de manera correcta qué es lo que esperamos conseguir u obtener con él, y gracias a Spring y a este módulo no nos hará falta programar nada de él.

Como nuestra idea es almacenar los tuits al darle a un botón sin necesidad de añadir nada para identificarnos, la fecha del momento en que realizamos la operación de guardado parece una solución correcta y simple para establecer como título de toda una colección de tuits. Es por esto que nuestro método en la interfaz será de tipo List<ObjetoTuit> findByTituloColeccion (String col) (Figura 40). Ahora no tenemos más que crear un nuevo campo de tipo String en nuestra clase ObjetoTuit que sea título. Este atributo inicialmente será nulo, y en el momento de guardar los tuits será cuando estableceremos para cada uno de ellos la fecha a modo de título de la colección. La última modificación restante en esta clase es la de elegir un elemento a modo de id para el correcto funcionamiento con MongoDB. En este caso concreto al tratarse de tuits, observamos que ya de por sí tienen un id único cada uno de ellos establecido por Twitter, y es sobre el que establecemos la anotación @Id (Figura 41).

```
public interface ObjetoTuitRepository extends MongoRepository<ObjetoTuit, String>{
    List<ObjetoTuit> findByTituloColeccion(String col);
}
```

Figura 39: Interfaz ObjetoTuitRepository

```
@Id
private long idTuit;
```

Figura 40: Anotación @Id

En el controlador, al igual que pasaba con el objeto de clase Twitter, tendremos nuestra interfaz marcada con la anotación '@Autowired' (Figura 42)

```
@Autowired
private ObjetoTuitRepository repositorio;
```

Figura 41: ObjetoTuitRepository con anotación @Autowired

Una vez hecho todo esto, no tenemos más que crear un nuevo botón para guardar los tuits, que estará disponible siempre y cuando haya tuits en nuestra tabla y no hayamos realizado previamente otra operación de guardado. Será entonces cuando al llamar al controlador tomemos la fecha en tiempo real, la establezcamos como título en cada uno de los tuits que tengamos y llamemos al método insert () de nuestra interfaz 'ObjetoTuitRepository' y con ello completaríamos la funcionalidad encargada de guardar los tuits. (Figura 43) (Figura 44)

Figura 42: Botón para almacenar tuits

```
this.repositorio.insert(this.listaPropia);
```

Figura 43: Método insert

Pasamos al poder acceder a cada una de las colecciones. En este caso, creamos un nuevo botón cerca de las cajas de búsqueda que nos permita acceder en cualquier momento a la lista de colecciones. (Figura 45)

Figura 44: Botón para acceder a colecciones almacenadas

Para esta lista hemos creado un nuevo archivo HTML. Al pulsar sobre el botón que mencionábamos, llamaremos a un nuevo método del controlador con el método get que se encargará de guardar todos los títulos de colección disponibles y pasarlos a este nuevo HTML. Además, si ya teníamos una lista de tuits en pantalla la guardamos en una nueva lista de ObjetoTuit para no perder esta información. (Figura 46)

```
if(this.listaPropia!=null){
    if(!this.listaPropia.isEmpty()){
        this.listaGuardada = new ArrayList<ObjetoTuit>();
        for(int i=0; i<listaPropia.size();i++){
            this.listaGuardada.add(this.listaPropia.get(i));
        }
    }

}
```

Figura 45: Guardando los tuits en lista adicional

Al llegar a esta nueva página nos encontraremos una pequeña tabla y un botón fuera de ella. El botón nos permite volver a la página anterior, es decir, la principal, y recuperar nuestra lista de tuits que tuviésemos. Por el contrario, si nos fijamos en la lista encontraremos que cada fila tiene tres columnas, una con el nombre de la colección que tienen esos tuits, y un botón en cada una de las columnas restantes. El primero nos da la opción de recuperar esa lista y el segundo nos permite eliminarla. (Figura 47)

Figura 46: Botón para regresar a la vista principal

Empezando por el segundo de ellos, el botón de eliminar realiza una llamada a un nuevo método de tipo post de nuestro controlador pasándole como argumento el título de la colección, que se limita a eliminar de nuestra base de datos la colección o los tuits con ese parámetro. Tras esto, comprobamos si teníamos o no tuits guardados antes de redirigir el control a esta nueva página. En tal caso, los recuperamos de nuevo y volvemos a mostrar como si no hubiese sucedido nada. (Figura 48) (Figura 49)

```
listaEliminar = this.repositorio.findByTituloColeccion(texto);
this.repositorio.delete(listaEliminar);
```

Figura 47: Método para eliminar una colección

```
if(!this.listaGuardada.isEmpty()){
    this.recuperarLista(listaPropia, listaGuardada);
```

Figura 48: Método para recuperar una lista de tuits

En el caso del primer botón que nos permite recuperar una colección guardada con anterioridad, hemos de añadir varios campos en el propio HTML para su correcto funcionamiento. El problema viene porque hemos de pasar como parámetro a nuestro

método de recuperación de colección uno que acabamos de recibir como parámetro en la propia vista, lo que supone a priori un problema para Thymeleaf si tratamos de introducirlo directamente por parámetro en th:object.

Por esto, hemos de añadir en el caso de recuperar una colección dos campos al botón que realiza la llamada, uno de id=valor y name=valor, ambos con el valor del nombre del objeto que queremos pasar en el campo th:object del formulario, y además usando, de nuevo en el botón, el campo th:value para establecer como valor, ahora sí, el parámetro que acabamos de recibir y que queremos enviar al controlador al realizar la llamada. (Figura 50)

```
th:action="@{/twittercontrolador/recuperarColeccion}" th:object="${textocoleccion}" method="post" >
' id="textocoleccion" name="textocoleccion" th:value="${titulo}">RECUPERAR</button>
```

Figura 49: campos id y name

Ya una vez en nuestro controlador de vuelta, creamos un nuevo método de tipo post, y es aquí donde hacemos uso del método que creamos en la interfaz, llamándolo con el parámetro recibido. Ayudándonos del módulo de Spring relacionado con MongoDB, esta simple llamada a un método que no tenemos definido en ningún sitio nos devuelve de igual forma lo que queríamos, los tuits relacionados con este título. Con los tuits de vuelta no queda más que añadirlos de manera normal a nuestra vista o página principal. (Figura 51)

```
this.repositorio.insert(this.listaPropia);
```

Figura 50: Almacenando tuits en base de datos

4.9.1 REVISIÓN DE ITERACIÓN

Al probar la nueva funcionalidad y realizar varias combinaciones de pasos a seguir, se puede dar un caso en que al tener tuits en pantalla y recuperar alguna colección, apareciesen estos tuits que teníamos antes mezclados, por lo que al traer o recuperar una colección de tuits para evitar estos escenarios hemos optado por rellenar nuestra lista solo con estos nuevos tuits.

4.9.2 CASOS DE USO

ID	CU-9
Caso de uso	Guardado de tuits
Descripción	Almacenamiento de tuits en base de datos (MongoDB) de acuerdo a la fecha en que se realiza el guardado
Precondición	Tener tuits para guardar y no haber realizado de manera anterior e inmediata otro guardado idéntico
Escenario principal	1. El usuario pulsa el botón de guardar 2. El usuario es redirigido a la misma ventana con los mismos tuits una vez se ha almacenado en base de datos la colección de los mismos. Esta vez la opción de guardar no está disponible
Escenario alternativo	-

ID	CU-10
Caso de uso	Acceso a vista de colecciones
Descripción	Navegación hacia la página con todas las colecciones de tuits almacenadas disponibles.
Precondición	-
Escenario principal	1. El usuario pulsa el botón para acceder a la vista donde están todas las colecciones 2. El usuario es redirigido a una nueva ventana donde podrá ver la lista de las colecciones disponibles, recuperarlas o eliminarlas y volver atrás.
Escenario alternativo	-

ID	CU-11
Caso de uso	Recuperar colección
Descripción	Obtener los tuits almacenados pertenecientes a una colección
Precondición	Haber accedido a la vista de colecciones y que exista alguna de ellas (CU-10)
Escenario principal	1. El usuario pulsa el botón de la colección que quiere recuperar 2. El usuario es redirigido a la ventana principal esta vez con la tabla conteniendo los tuits pertenecientes a la colección que acaba de recuperar
Escenario alternativo	-

ID	CU-12
Caso de uso	Eliminar colección
Descripción	Borrad de tuis pertenecientes a una colección
Precondición	Haber accedido a la vista de colecciones y que exista alguna de ellas (CU-10)
Escenario principal	1. El usuario pulsa el botón de la colección que quiere eliminar. 2. El usuario es redirigido a la vista principal una vez la colección en cuestión ha sido eliminada. La tabla contiene los tuits que ya estaban en ella antes de acceder a la nueva vista o estará vacía si no existían tuits antes de acceder a ella.
Escenario alternativo	-

ID	CU-13
Caso de uso	Regreso a vista principal
Descripción	Volver a la vista principal desde la vista que contiene a las colecciones,
Precondición	Haber accedido a la vista de colecciones (CU-10)
Escenario principal	1. El usuario pulsa el botón inferior para volver a la vista principal. 2. El usuario es redirigido a la ventana principal. La tabla contendrá los tuits que hubiese antes, o estará vacía si no existían previamente.
Escenario alternativo	-

5. CONCLUSIONES

A continuación, procedemos a comentar las conclusiones que pueden extraerse una vez finalizado el proyecto.

Empezando por la rama más tecnológica y docente, este proyecto ha servido como forma o método de aprender varias tecnologías y frameworks nuevos, así como afianzar más los conocimientos que se tenían en otros lenguajes y herramientas.

Se ha trabajado con Spring Framework, lo que ha supuesto una puerta abierta a su mundo y el de sus múltiples módulos, a una nueva forma de trabajar, de realizar proyectos y de adquirir algo de soltura con él. También hemos trabajado con Maven, siendo este proyecto el primero que realmente me ha permitido ver el potencial de Maven y entender claramente su utilidad. Además, hemos tenido que trabajar con MongoDB, lo que ha supuesto un pequeño acercamiento a las bases de datos no relacionales, y a entender cuáles son sus puntos fuertes y limitaciones.

En cuanto al tema de APIs, este proyecto ha servido igualmente como primera toma de contacto con algunas de ellas como la de Twitter. Ha resultado útil para entender cómo podemos beneficiarnos de las APIs que ya existen, cómo navegar por su documentación y entender cómo y de qué manera funcionan y hemos de utilizar sus peticiones.

En cuanto a lo personal, este proyecto ha supuesto uno de los mayores logros a lo largo de todos estos años: el poder ver cómo se ha ido desarrollando y creciendo ciclo a ciclo, funcionalidad a funcionalidad, desde la obtención de una simple tabla rellena con tuits hasta poder acoplar y servirnos de varios módulos interrelacionados entre sí. En este punto llega el momento de observar cómo empezamos y cómo terminamos este proyecto y ver la gran cantidad de cosas aprendidas, no solo en lo tecnológico, sino en lo personal, en intentar darle la vuelta a alguna situación que no conseguimos resolver, algunas veces encontrando la salida de alguna otra forma, otras no, pero siempre tratando de tomar otra dirección que pueda llevarnos por el buen camino.

Y con todo, no solo es el momento de mirar hacia atrás únicamente hasta el inicio del proyecto, sino también al inicio de estos cuatro años de universitario, en fijarnos en lo que sabíamos y pensábamos que sabríamos al acabarlos, y en lo que sabemos ahora una vez que ha pasado el tiempo, en reflexionar sobre que supone el fin una etapa de constante aprendizaje, y se abre otra nueva etapa que, si bien no cesará en cuanto a

aprender cosas nuevas, nos ofrecerá la posibilidad de demostrar al mundo los conocimientos adquiridos en este centro o los valores inculcados por el cuerpo docente.

Por último y aparte de todo lo anterior, debemos mencionar que el proyecto se ha presentado dentro de la Cátedra de Seguridad, Emergencias y Catástrofes de la UMA [12]. Esta idea de proyecto parte precisamente del grupo de trabajo de esa cátedra.

6. REFERENCIAS BIBLIOGRÁFICAS

[1]: http://start.spring.io/

[2]: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/

[3]: http://getbootstrap.com/

[4]: https://apps.twitter.com/

[5]:https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

[6]: https://dev.twitter.com/rest/public/search

[7]: http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html

[8]: https://developers.google.com/maps/

[9]: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/

[10]: https://spring.io/guides/gs/accessing-data-mongodb/

[11]: https://maven.apache.org/

[12]: http://www.umaemergencias.es

ANEXOS TÉCNICOS

A lo largo de esta última sección vamos a explicar los pasos que debemos llevar a cabo en caso de querer desplegar la aplicación. Esto abarca dos procesos para la obtención y configuración de la API Key de Twitter y la de Google, y un tercer proceso que contiene el resto de configuraciones necesarias sin contar estas dos ya mencionadas.

A. OBTENCIÓN DE API KEY – TWITTER

Para obtener una API Key de Twitter y poder acceder a las operaciones y funciones que nos brindan, hemos de acceder a [4] y seleccionar la opción de 'Create New App' (Ilustración 1) (Ilustración 2)

Ilustración 1: Create New App

Tras esto, tendremos que rellenar los campos indicados

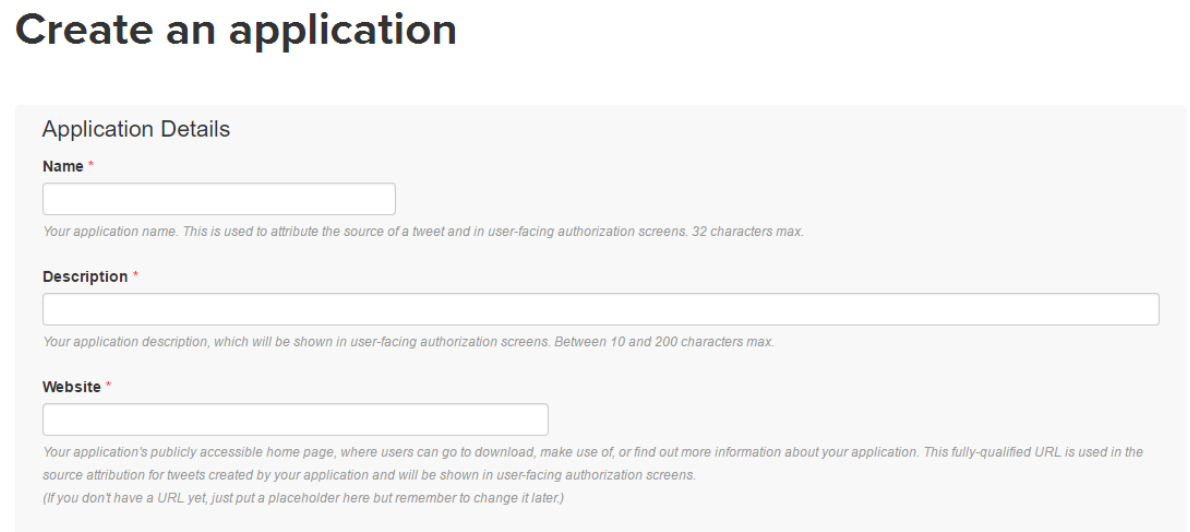

Ilustración 2: Create an application

Una vez creada correctamente, entramos en ella y navegamos hasta la sección 'Keys and Access Tokens' (Ilustración 3)

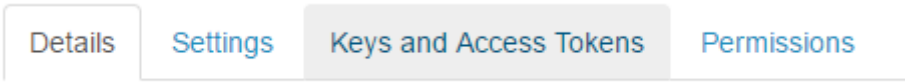

Ilustración 3: Keys and Access Tokens

En esta sección ya podremos copiar tanto el Consumer Key (API Key), como el Consumer Secret (API Secret) en nuestro application. properties dentro de nuestro proyecto como 'spring.social.twitter.app-id=' y 'spring.social.twitter.app-secret=' (Ilustración 4)

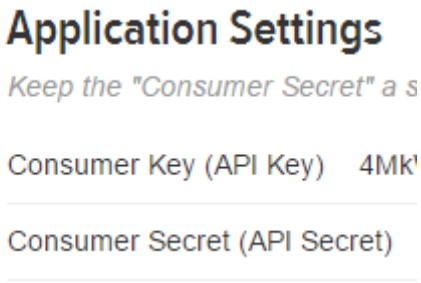

Ilustración 4:API Key y API Secret

B. OBTENCIÓN DE API KEY – GOOGLE

En este segundo apartado vamos a obtener una API Key de Google para poder utilizar las funcionalidades de Google Maps.

Para ello accedemos a [8] y, primeramente, elegimos la opción web de entre las cuatro que nos aparecerán. (Ilustración 5)

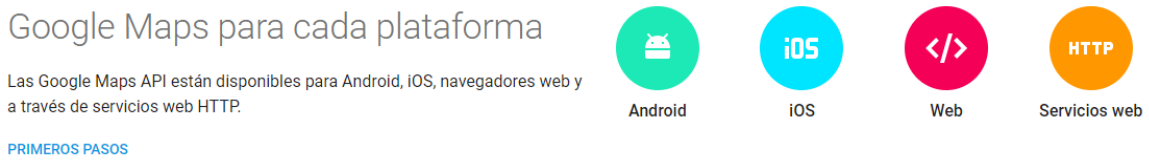

Ilustración 5: Opciones Google

Una vez dentro, tendremos que escoger el tipo que buscamos, en este caso ‘Google Maps JavaScript API’ (Ilustración 6).

Ilustración 6: Tipos API

Escogida ya la opción anterior, ahora hemos de seleccionar el mapa que queramos tener como base para trabajar, en nuestro caso fue el primer de ellos. (Ilustración 7)

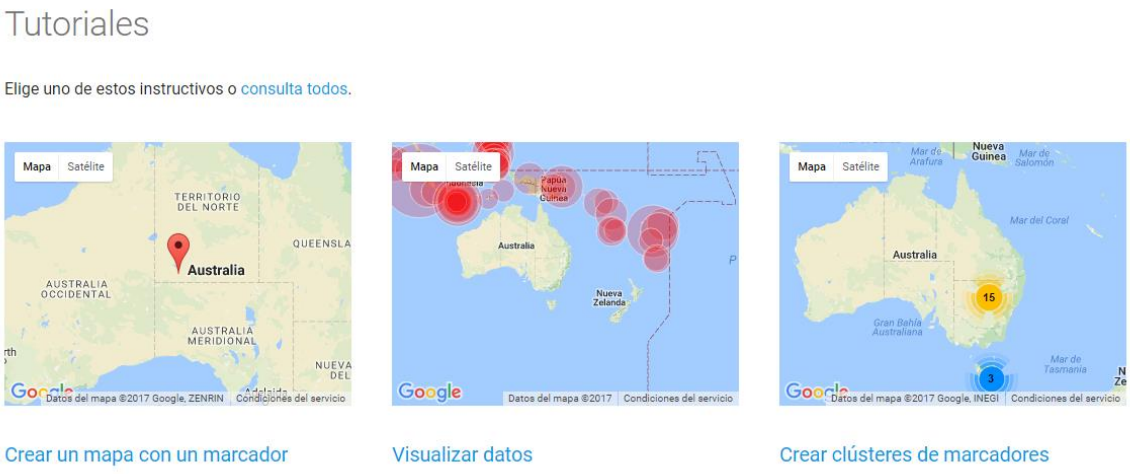

Ilustración 7: Tipos de mapas

Finalizada la elección, obtendremos el código base que debemos insertar en nuestro proyecto para empezar a mostrar y usar el mapa a falta de añadir nuestra API Key (Ilustración 8)

```
<!DOCTYPE html>
<html>
  <head>
    <style>
       #map {
        height: 400px;
        width: 100%;
       }
    </style>
  </head>
  <body>
    <h3>My Google Maps Demo</h3>
    <div id="map"></div>
    <script>
      function initMap() {
        var uluru = {lat: -25.363, lng: 131.044};
        var map = new google.maps.Map(document.getElementById('map'), {
          zoom: 4,
          center: uluru
        });
        var marker = new google.maps.Marker({
          position: uluru,
          map: map
        });
      }
    </script>
    <script async defer
    src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap">
    </script>
  </body>
</html>
```

Ilustración 8: Código base

Para obtener la API Key, seleccionamos la opción de dirigirnos a 'Google API Console' (Ilustración 9)

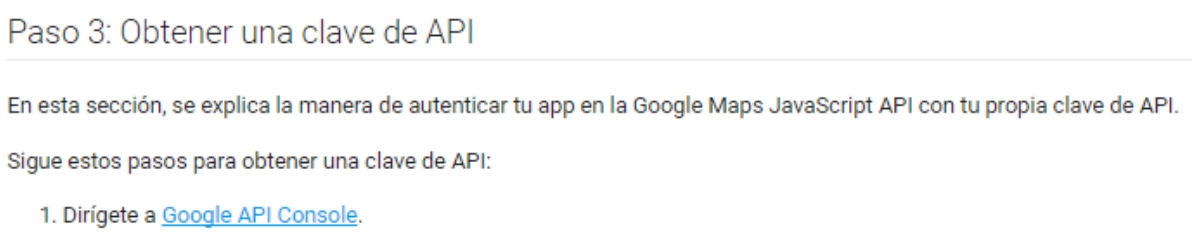

Ilustración 9: Dirigirnos a Google API Console

Dentro de esta nueva sección, elegiremos la opción de ‘Crear proyecto’ (Ilustración 10)

Ilustración 10: Crear proyecto

Una vez creado, vamos a añadir las credenciales a nuestro proyecto, seleccionamos la opción de ‘Google Maps JavaScript API’. (Ilustración 11)

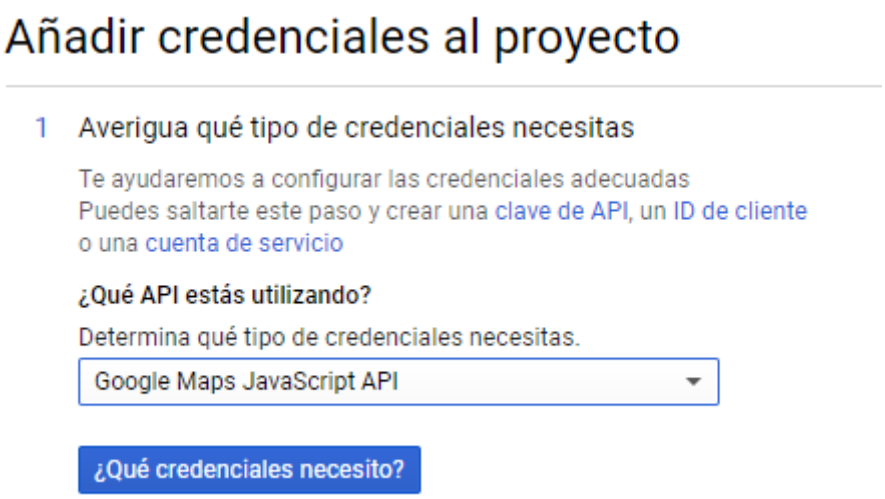

Ilustración 11: Seleccionando credenciales

Tras esto obtendremos nuestra API Key. Posteriormente, pulsamos en ‘Listo’; navegamos hacia la sección ‘Panel de Control’ y seleccionamos ‘Habilitar API’. Llegaremos a una sección con varias subsecciones. (Ilustración 12)

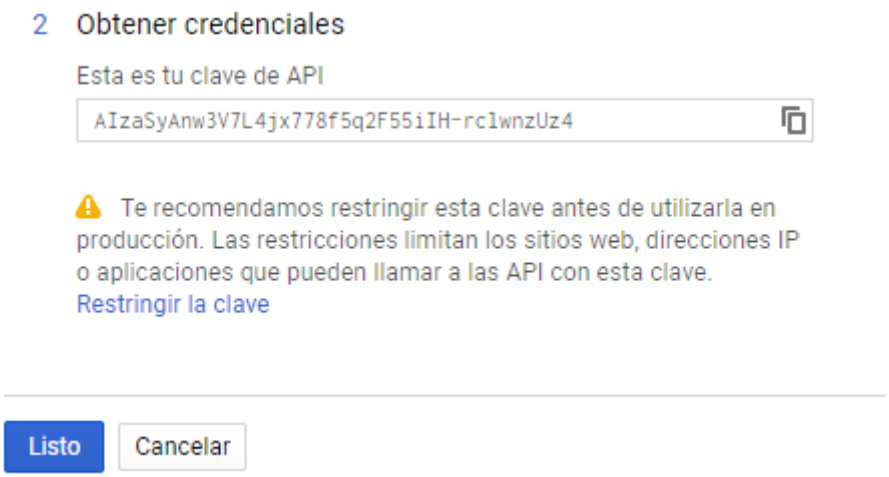

Ilustración 12: API Key obtenida

En la subsección de ‘APIs de Google Maps’ escogemos ‘Google Maps Geocoding API’ (Ilustración 13)

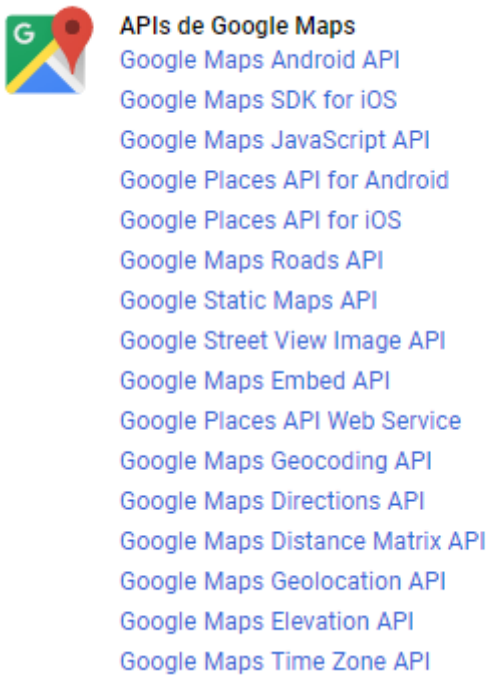

Ilustración 13: Google Maps Geocoding API

Dentro de esta nueva página, finalmente elegimos “Habilitar” y ya tendremos nuestra API Key lista para copiarla en el código base que obtuvimos en la parte que se indica para ubicar nuestra API Key. (Ilustración 14)

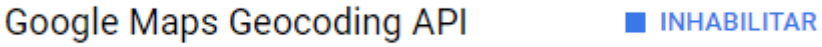

Ilustración 14: Geocoding habilitado

C. MANUAL DE INSTALACION

En esta última parte del anexo vamos a describir todos los pasos necesarios para poder desplegar el proyecto en un ordenador desde cero.

Como nota adicional, se presupone que Eclipse IDE está ya instalado, ya que lo consideramos un paso trivial.

La primera herramienta que vamos a proceder a instalar es Maven. Para empezar, accedemos a [11]

Una vez dentro, pulsamos sobre la opción de ‘Download’, y tras ello procedemos a instalarlo para lo que entramos en la opción de ‘Install’ (Ilustración 15)

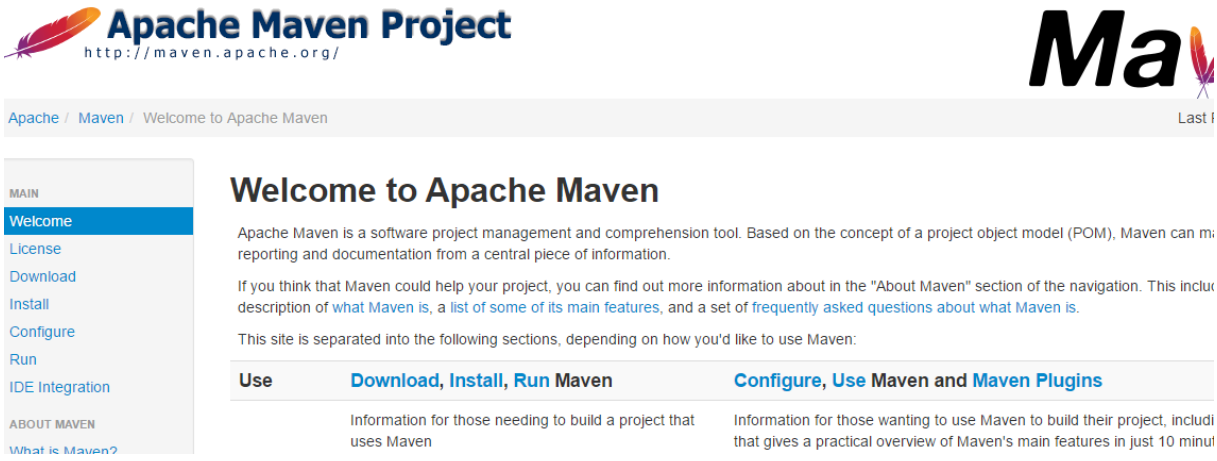

Ilustración 15: Maven principal

Una vez estemos en la sección correspondiente a la instalación de Maven el resto consiste en seguir los pasos señalados. El primer de ellos es asegurarnos de que nuestra variable de entorno 'JAVA_HOME' está establecida y que apunta a la carpeta donde tenemos instalado nuestro JDK.

Podemos acceder a nuestras variables de entorno en Windows 10, escribiendo en la barra de búsqueda de Windows variables de entorno, entrando a través de la opción resaltada, y entrando por la opción 'Variables de entorno…'

Tras ello descomprimimos lo descargado anteriormente y añadimos el directorio bin al 'PATH'. Una vez hecho esto procedemos a comprobar si la instalación ha sido satisfactoria, utilizando en un terminal el comando 'mvn -v' (Ilustración 16) (Ilustración 17) (Ilustración 18) (Ilustración 19) (Ilustración 20)

Installing Apache Maven

The installation of Apache Maven is a simple process of extracting the archive and adding the `bin` folder with the `mvn` command to the `PATH`.

Detailed steps are:

- Ensure `JAVA_HOME` environment variable is set and points to your JDK installation
- Extract distribution archive in any directory

```
1. unzip apache-maven-3.5.0-bin.zip
```

or

```
1. tar xzvf apache-maven-3.5.0-bin.tar.gz
```

Alternatively use your preferred archive extraction tool.

- Add the `bin` directory of the created directory `apache-maven-3.5.0` to the `PATH` environment variable
- Confirm with `mvn -v` in a new shell. The result should look similar to

```
1. Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T04:57:37-07:00)
2. Maven home: /opt/apache-maven-3.3.3
3. Java version: 1.8.0_45, vendor: Oracle Corporation
4. Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/jre
5. Default locale: en_US, platform encoding: UTF-8
6. OS name: "mac os x", version: "10.8.5", arch: "x86_64", family: "mac"
```

Ilustración 16: Instalando Maven

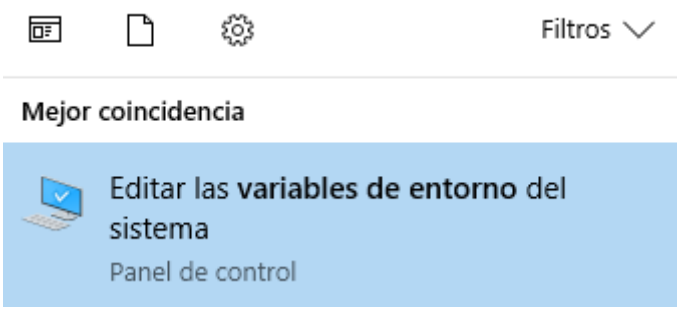

Ilustración 17: Búsqueda por Windows

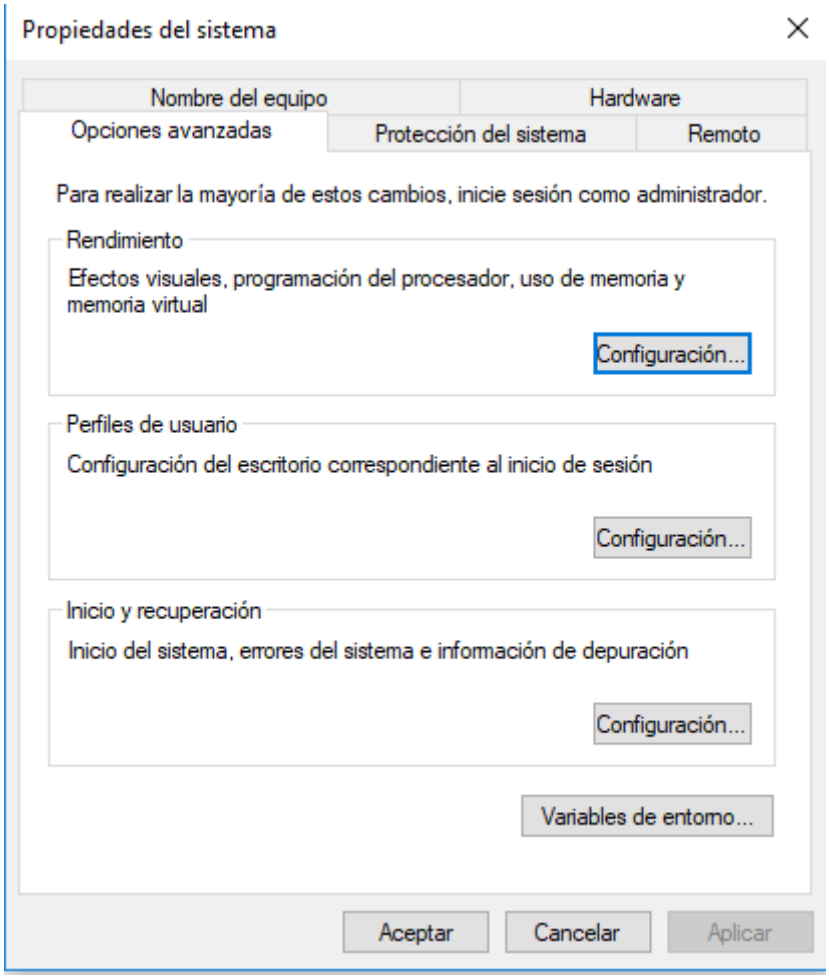

Ilustración 18: Panel de control

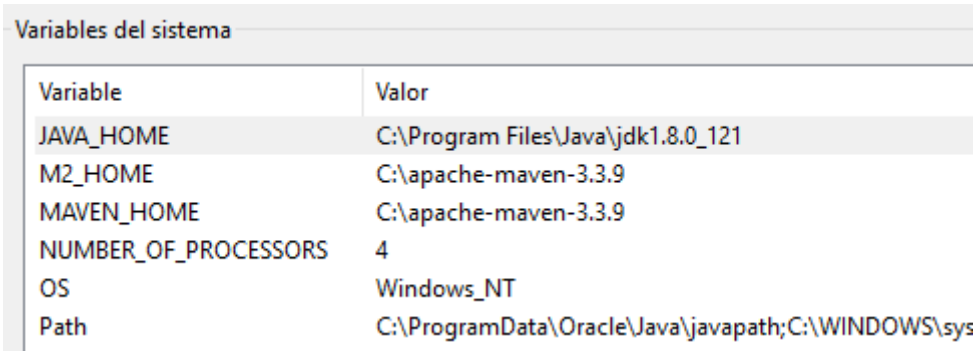

Ilustración 19: Variables del sistema

```
C:\Users\AdriGH>mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T17:41:47+01:00)
Maven home: C:\apache-maven-3.3.9
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_121\jre
Default locale: es_ES, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"

C:\Users\AdriGH>
```

Ilustración 20: mvn -v

Una vez tenemos Maven instalado, vamos a proceder a la instalación de MongoDB.

Para ello, vamos a [9] y buscamos la opción de 'MongoDB downloads page' (Ilustración 21)

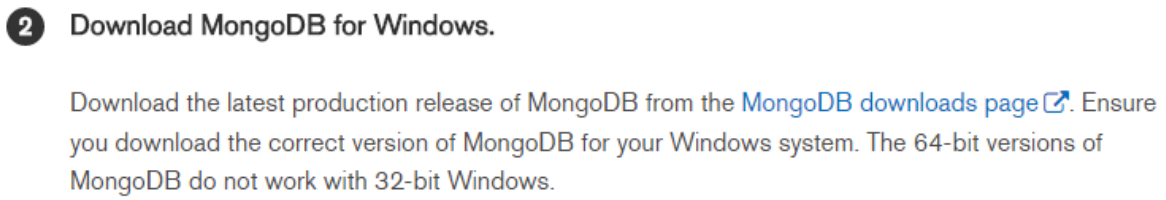

Ilustración 21: MongoDB downloads page

Dentro de ella elegimos la siguiente opción (En este caso según cada pc) **(Ilustración 22)**

Ilustración 22: Versión de MongoDB

Tras ello ejecutamos el archivo y completamos la instalación.

Seguimos asumiendo que se ha instalado en la siguiente ruta (Ilustración 23)

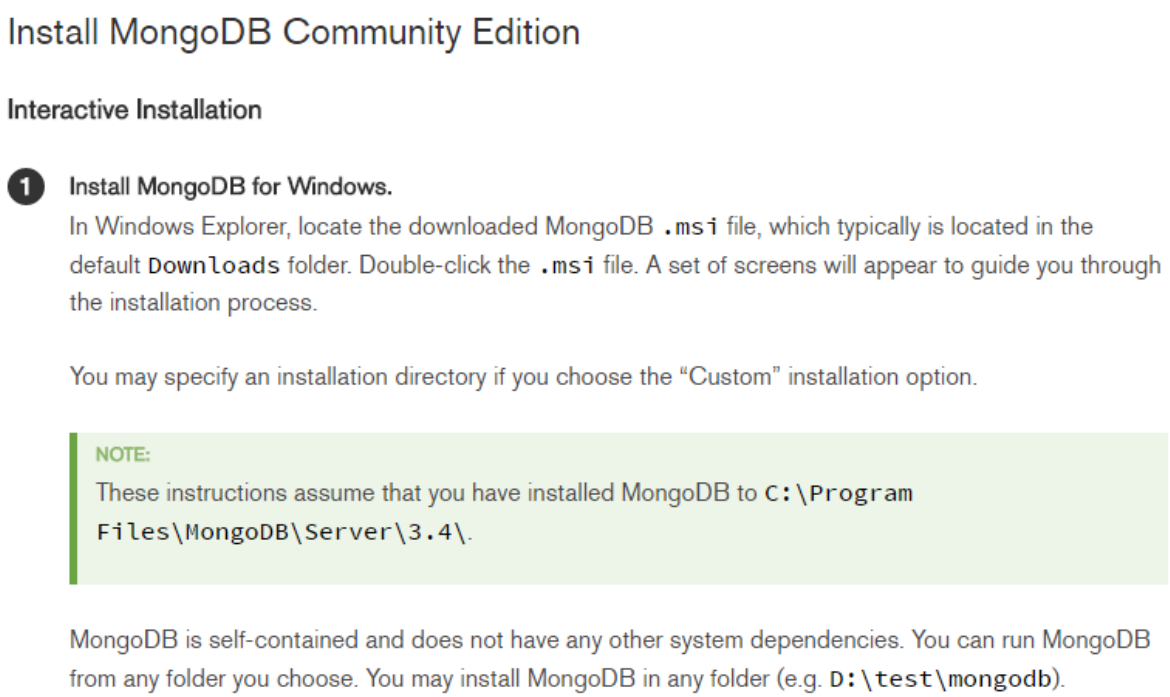

Ilustración 23: Ruta instalación

Tras ello ejecutamos el siguiente comando (Ilustración 24)

Run MongoDB Community Edition

WARNING:

Do not make `mongod.exe` visible on public networks without running in "Secure Mode" with the `auth` setting. MongoDB is designed to be run in trusted environments, and the database does not enable "Secure Mode" by default.

1 Set up the MongoDB environment.

MongoDB requires a data directory to store all data. MongoDB's default data directory path is the absolute path `\data\db` on the drive from which you start MongoDB. Create this folder by running the following command in a **Command Prompt**:

Copy

```
md \data\db
```

Ilustración 24: MongoDB primer comando

Y una vez hecho esto, a la hora de usarlo, ejecutamos los siguientes comandos. (Ilustración 25) (Ilustración 26)

2 Start MongoDB.

To start MongoDB, run `mongod.exe`. For example, from the **Command Prompt**:

Copy

```
"C:\Program Files\MongoDB\Server\3.4\bin\mongod.exe"
```

This starts the main MongoDB database process. The `waiting for connections` message in the console output indicates that the `mongod.exe` process is running successfully.

Depending on the security level of your system, Windows may pop up a **Security Alert** dialog box about blocking "some features" of `C:\Program Files\MongoDB\Server\3.4\bin\mongod.exe` from communicating on networks. All users should select `Private Networks, such as my home or work network` and click `Allow access`. For additional information on security and MongoDB, please see the Security Documentation.

Ilustración 25: MongoDB Comando 1 de uso

Ilustración 26: MongoDB Comando 2 de uso

Una vez realizados los pasos anteriores, tendremos MongoDB listo para ser usado.

Adicionalmente, dentro de Eclipse hemos de instalar un plugin de Spring Framework. (Ilustración 27) (Ilustración 28)

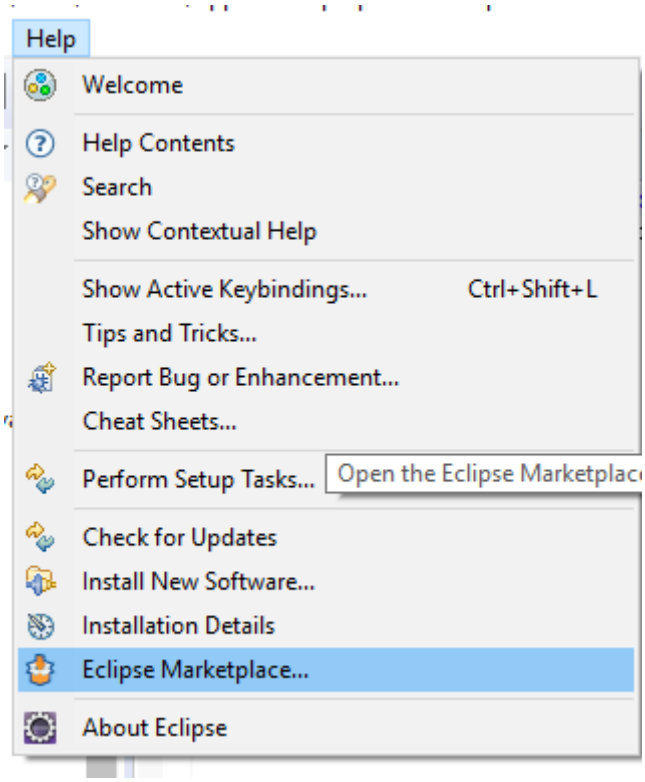

Ilustración 27: Eclipse Marketplace

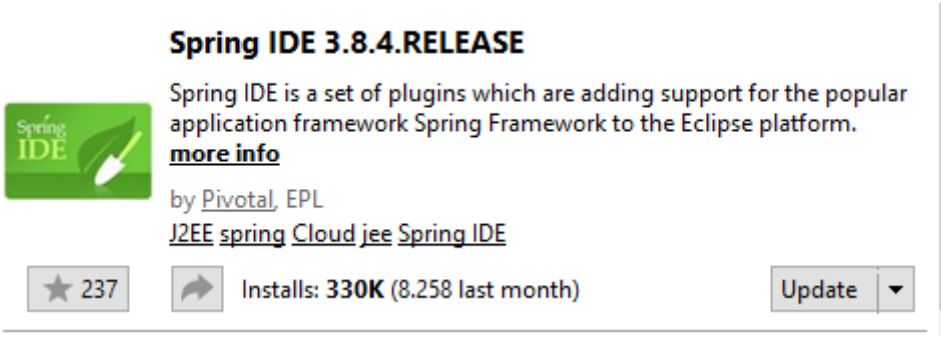

Ilustración 28: Plugin Spring

Finalmente, para la configuración inicial del proyecto configuramos los distintos módulos de Spring a través de [1], descargamos el comprimido, lo descomprimimos en nuestra carpeta de proyecto, y a través de un terminal ejecutamos 'mvn clean install' y con ello tendremos todas las dependencias y librerías preparadas. De igual manera, para tener todas las necesarias para trabajar y usar el proyecto, así como para la inclusión de nuevas librerías o módulos.