UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un API REST para transmisión de datos de sensores GPS

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Gabriele Bianchini
Tutor: Sara Blanc Clavero

Curso 2017-2018

Resum

Desenvolupament d'un API REST capaç d'emmagatzemar i servir les dades retornats per un GPS a través d'una sèrie d'antenes de camp. Actualment la placa Raspberry Pi que rep aquestes dades els torna cada un interval fix de temps en un fitxer de text, el que impedeix la seva petició a la carta i fa complex i tediós el tractament d'aquests per a l'equip de Geomàtica, que són els interessats en explotar aquesta informació. L'API farà de client / servidor, nodrint-se de la informació del GPS i emmagatzemant-la en una base de dades no relacional, per després poder mostrar-la a través d'una interfície a demanda de l'usuari i facilitar el tractament de la mateixa en el futur. Per al desenvolupament s'utilitzarà MEAN stack, a més de fer-se el desplegament amb Docker.

Paraules clau: API, REST, MEAN, GPS, Raspberry, Geomática, Docker, MongoDB, Javascript, Express, Angular

Resumen

Desarrollo de un API REST capaz de almacenar y servir los datos devueltos por un GPS a través de una serie de antenas de campo. Actualmente la placa Raspberry Pi que recibe estos datos los devuelve cada un intervalo fijo de tiempo en un fichero de texto, lo que impide su petición bajo demanda y hace complejo y tedioso el tratamiento de estos para el equipo de Geomática, que son los interesados en explotar esta información. El API hará de cliente/servidor, nutriéndose de la información del GPS y almacenándola en una base de datos no relacional, para después poder mostrarla a través de una interfaz a demanda del usuario y facilitar el tratamiento de la misma en el futuro. Para el desarrollo se utilizará MEAN stack, además de hacerse el despliegue con Docker.

Palabras clave: API, REST, MEAN, GPS, Raspberry, Geomática, Docker, MongoDB, Javascript, Express, Angular

Abstract

Development of a REST API capable of storing and serving the data retrieved by a GPS through a series of field antennas. Currently, the Raspberry Pi board that recieves this data returns it in a fixed time interval in a text file, which prevents petitioning it on demand and makes complex and tedious the treatment of said data for the Geomática team, who are those interested in using it to their advantage. This REST API will be used as a client/server, obtaining data from the GPS and storing it in a non-relational database, being able afterwards to present it in a graphic interface on demand and to ease the treatment of this data in the future. To this end, MEAN stack will be used for the development and will be deployed with Docker.

Key words: API, REST, MEAN, GPS, Raspberry, Geomatica, Docker, MongoDB, Javascript, Express, Angular

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Este proyecto se ha realizado en colaboración con la E.T.S. de Geomática de la Universita Politècnica de València. La Escuela de Geomática [1] de la UPV dispone de estaciones geodésicas portátiles que capturan información de reflectometría proporcionada por las constelaciones GPS, GALILEO y GLONAS. Durante la toma de datos en campo, la información recogida por una sola estación puede suponer un número de datos considerables, en función del número de satélites y frecuencia de las lecturas. Actualmente, estos datos se vuelcan en ficheros de texto en local, almacenados en algún dispositivo empotrado conectado físicamente a la estación, tipo Raspberry Pi. Una vez recogida la estación tras una campaña, se realiza el tratamiento de los ficheros que es tedioso ya que necesita un filtrado, un parseo específico según la aplicación y el correspondiente tratamiento para obtener las medidas de campo esperadas por cada uno de los ficheros almacenados en local. En resumen, hablamos de sensores que obtienen información compleja y de tamaño no despreciable. En la siguiente imagen, que es un extracto de un fichero de ejemplo, se puede ver la forma en la que se recibe la información actualmente:

```
$GNVTG,,T,,M,0.011,N,0.020,K,A*3F
$GNGNS,150012.00,3925.02116,N,00024.96130,W,AAAN,17,0.65,20.5,50.0,,,V*3D
$GNGGA,150012.00,3925.02116,N,00024.96130,W,1,12,0.65,20.5,M,50.0,M,,*67
$GNGSA,A,3,21,27,25,05,31,16,26,29,20,,,,1.09,0.65,0.87,1*09
$GNGSA,A,3,87,76,86,77,75,85,,,,,,,1.09,0.65,0.87,2*09
$GNGSA,A,3,02,30,,,,,,,,,,,1.09,0.65,0.87,3*06
$GPGSV,3,1,12,04,,,43,05,06,045,38,14,04,224,31,16,25,304,45,0*5F
$GPGSV,3,2,12,20,12,135,41,21,62,158,40,23,03,309,33,25,29,108,45,0*67
$GPGSV,3,3,12,26,54,315,43,27,09,253,42,29,44,048,44,31,58,218,45,0*6F
$GLGSV,2,1,06,75,36,148,39,76,87,176,39,77,29,327,35,85,22,037,40,0*7E
$GLGSV,2,2,06,86,74,350,43,87,36,235,42,0*7B
$GAGSV,1,1,03,02,14,135,36,27,,,40,30,66,136,40,0*70
$GNGLL,3925.02116,N,00024.96130,W,150012.00,A,A*60
```

Figura 1.1: Ejemplo fichero de texto generado.

Este proyecto trata de avanzar en la recogida de datos de las estaciones geodésicas portátiles en línea y su procesado automatizado mejorando la disponibilidad de la información y dando un servicio que facilitará a los usuarios de estas estaciones obtener medidas con menos esfuerzo que el que actualmente invierten. Es más, la mayoría de los datos obtenidos podrían dar servicio a diferentes aplicaciones, cuando ahora se utilizan para una única aplicación. Su recogida y almacenamiento estructurado en bases de datos tendrían potencial para su posterior uso con otras aplicaciones, para comparativas de medidas en distintas campañas, e incluso, para medidas que aún no se tienen claras

[1] http://geomaticaupv.webs.upv.es/

ya que, en gran parte, el uso de las estaciones en la Escuela de Geomática tienen un fin orientado a la investigación.

Este proyecto buscará facilitar a los usuarios la interpretación automatizada de estos datos, además de poder ofrecer una cantidad de funcionalidades interesantes para el propio equipo de Geomática de las que no disponen actualmente.

Motivación

El desarrollo de aplicaciones web y APIs [2] en conjunto han sustituido en los últimos años a las aplicaciones de escritorio. La posibilidad de crear aplicaciones que no dependan de un sistema operativo en concreto para su instalación, o de tener instalado un software que permita ejecutarlas (Java, .NET, etc..), es una liberación para la mayoría de usuarios que tienen un conocimiento bajo sobre estos temas. Lo único que requieren para funcionar correctamente desde el punto de vista del usuario es el disponer de un navegador (en cualquier dispositivo actualmente) y conexión a internet.

El interés sobre el trabajo viene dado principalmente por la motivación personal de querer ser desarrollador web full stack, y poder explotar al máximo las ventajas que nos ofrece esta tecnología web en contrapartida a las aplicaciones de escritorio de antaño. La alta disponibilidad de las webs, acceso desde cualquier lugar, flexibilidad respecto al dispositivo con el que se quiere acceder y el navegador utilizado para ello además de la capacidad de escalabilidad son todo motivos muy atractivos y que señalan el futuro del desarrollo para todo tipo de aplicaciones.

Hay varios motivos por los que se ha optado por una API REST bajo MEAN [3] stack para resolver el problema que se presenta:

- Se podrán tanto hacer peticiones de ficheros (datos) a demanda del usuario, como recibirlas por defecto a intervalos de tiempo fijos. Esto permitiría mantener el sistema de generación de ficheros que existe actualmente y su almacenamiento en local además de poder solicitar uno en caso de querer recibir la información en línea. Aunque existen formas más eficientes de leer la información de las estaciones sin necesidad de generar ficheros de texto, se mantiene esta funcionalidad por demanda del usuario y compatibilidad con sus actuales procesos. En cualquier caso, se han proporcionado consideraciones de mejora.
- Es totalmente escalable gracias a la tecnología utilizada.
- Permite separar el cliente del servidor, lo que permitiría por ejemplo migrar en el futuro solo una parte a una tecnología más reciente que fuera interesante sin afectar todo el proyecto.
- Siempre es independiente a la tecnología utilizada (MEAN en este caso) mientras las respuestas a las peticiones concuerden con el lenguaje utilizado (XML, JSON...)

Objetivos

El objetivo del proyecto es conseguir pasar de una gran cantidad de información "en masa" devuelta por los sensores de campo a poder tener esta información almacenada

[2] https://en.wikipedia.org/wiki/Application_programming_interface
[3] https://en.wikipedia.org/wiki/MEAN_(software_bundle)

en una base de datos en lugar de en disco como con los ficheros de texto, y poder ser presentada al usuario de forma clara y limpia en una interfaz gráfica.

Para poder alcanzar este objetivo final el proyecto ha sido dividido en dos objetivos secundarios:

- Implementar una API que sea capaz de procesar los ficheros de texto obteniendo la información relevante/necesaria para Geomática, y almacenándola en MongoDB [4].
- Implementar una interfaz gráfica para que los usuarios puedan consultar esta información almacenada en MongoDB.

Impacto esperado

Este proyecto supone para Geomática el poder abstraerse del proceso para el tratamiento de los ficheros de texto, ya que podrán acceder a toda la información proporcionada por los sensores a través de una interfaz gráfica. Esto evita el proceso que debían realizar hasta ahora, donde debían ejecutar sobre cada fichero de texto la transformación necesaria para poder obtener los datos de la forma que a ellos les interesaba, y aún así no disponían de una interfaz que les fuera a facilitar la consulta de los mismos. Actualmente están volcando sobre otro fichero de texto la información que estiman necesaria.

El trabajo también va a tener un impacto positivo respecto a los ODS. Uno de los principales objetivos de Geomática con estos sensores de campo es poder estudiar en tiempo real mediante la observación directa de los datos obtenidos cual es la cantidad de riego necesaria para el campo donde están localizadas las antenas. Esto permitiría un ahorro en el consumo de agua debido a la posibilidad de que los campos se estén regando de más cuando no es necesario.

Figura 1.2: Foto de un sensor de campo.

[4] https://www.mongodb.com/

Metodología

Una vez se ha establecido cual es el problema a resolver, y analizado cual es la solución que se intentará dar al problema, la metodología que se ha seguido para realizar el proyecto es, de forma enumerada, la siguiente:

1. Realizar una entrevista con un miembro del equipo de Geomática para saber cual es su punto de vista sobre el problema al que buscan solución y qué espera como resultado de este trabajo.
2. Se diseña la estructura que va a seguir el proyecto de software y se analizan los requisitos necesarios para la aplicación.
3. Hacer una primera versión del API que procese los ficheros y los almacene en MongoDB.
4. Mostrar los datos almacenados y recibir *feedback* del equipo de Geomática.
5. Una vez se procesan correctamente los ficheros (en caso negativo, repetir pasos 3 y 4), se implementa la interfaz gráfica.
6. Mostrar la forma en la que se presentan los datos al equipo de Geomática y recibir *feedback*.

Al igual que para el procesamiento de los ficheros, los dos últimos pasos se repiten hasta conseguir la interfaz que le interesa a Geomática.

Puede apreciarse que se ha optado por una metodología ágil a partir de estos pasos.

Estructura de la memoria

El actual documento ha sido dividido en nueve capítulos:

Capítulo 2: Situación actual de la tecnología

En esta sección se analizará brevemente el recorrido de la tecnología que utilizamos hasta llegar a la situación que nos encontramos.

Capítulo 3: Análisis del problema

En este capítulo se analizarán todos los posibles problemas que surjan a lo largo del desarrollo del proyecto, desde posible brechas de seguridad al supuesto presupuesto necesario para poder llevar a cabo un proyecto de este tipo.

Capítulo 4: Diseño de la solución

Aquí se hablará del diseño del proyecto. Explicaremos las partes que lo componen y como van a funcionar todas juntas.

Capítulo 5: Desarrollo de la solución propuesta

Se mostrará cual ha sido el proceso de desarrollo del proyecto, de cada parte de él, además de decisiones tomadas y particularidades del mismo.

Capítulo 6: Implantación

Se verá como se ha desplegado el proyecto para su puesta en marcha y realización de pruebas.

Capítulo 7: Pruebas

En este apartado se van a repasar brevemente las pruebas realizadas para cerciorarse que el funcionamiento de la aplicación es el esperado por los usuarios.

Capítulo 8: Conclusiones

Conclusiones finales obtenidas acerca del proyecto tras la finalización del mismo.

Capítulo 9: Trabajos futuros

Posibles trabajos futuros que se podrían llevar a cabo como continuación de este proyecto.

CAPÍTULO 2
Situación actual de la tecnología

Aplicaciones de escritorio contra aplicaciones WEB

Pese a que hoy en día la tendencia para desarrollar aplicaciones está enfocada en la web, esto no siempre ha sido así. A continuación analizaremos brevemente los factores claves en este cambio.

Eficiencia en costes

En las aplicaciones web, los usuarios acceden el sistema a través de los exploradores. Pese a que las distintas interacciones de los usuarios con el sistema deben de ser probadas a fondo para evitar incongruencias en el comportamiento de los distintos motores de cada navegador, no hay que tener en cuenta los posibles sistemas operativos en juego. Desarrollar una aplicación y asegurarse de que su funcionamiento es el esperado en cada versión de cada sistema operativo es muy costoso, lo que causa poco soporte para los sistemas menos usados y dificultad de abarcar todos los posibles errores que surjan.

Accesible desde cualquier parte

Acceder a una aplicación web es tan sencillo como disponer de un navegar web. Independientemente de estar utilizando un ordenador, móvil, tableta, etc... seremos capaces de acceder al contenido ofrecido por una aplicación web mientras dispongamos de una conexión a internet.

Fácil instalación y mantenimiento

Una vez el servidor de la aplicación web ha sido actualizado, no es necesario actualizar individualmente cada dispositivo que haga uso de los recursos, estos automáticamente dispondrán de la versión más reciente en su cliente web nada más realizar la primera conexión con el servidor actualizado.

Mayor adaptabilidad al aumento de carga de trabajo

Si una aplicación require más potencia de proceso, solamente el hardware del servidor debe de ser actualizado. Incluso se pueden ir añadiendo más servidores que vayan repartiéndose las peticiones mediante un balanceador de carga, lo que permite de forma sencilla aumentar la rapidez con la que se manejan y devuelven los recursos requeridos por el cliente.

[5] [6]

Necesidad de la tecnología utilizada

Javascript

Javascript es un lenguaje de programación enfocado al desarrollo web creado en 1995 por Brendan Eich mientras trabajaba en Netscape Communications. La idea original de Javascript era simple, lo que se pretendía era un lenguaje que permitiera interacción con el DOM de las páginas web. Con esto se buscaba lograr de forma más sencilla la creación de páginas web más dinámicas, con animaciones y diversos tipos de interacciones por parte del usuario a los que la página HTML fuera respondiendo en el momento, queriendo alejarse de las web estáticas. Gracias a los scripts que se podían introducir directamente en el código HTML esto fue posible. [4]

Hoy en día Javascript no solo es utilizado en la parte del cliente, lo que nos lleva al siguiente punto de esta sección.

NodeJS

NodeJS [1] surge debido a la necesidad de los desarrolladores de trabajar con el paradigma de I/O de eventos no bloqueantes, y además poderlo hacer con el propio Javascript utilizado para la parte del cliente. Hasta el momento la forma de poder utilizar Javascript en el servidor era, entre las más conocidas, con Netscape's LiveWire Server o Microsoft Active Server Pages (ASP). Por motivos que salen de la intención de este apartado, estas otras alterativas no tuvieron tanto éxito y se redirigió su foco a otras tecnologías (por ejemplo ASP de Microsoft a C#).

Las posibilidades de escalabilidad, sencillez de uso, y gran apoyo de la comunidad con incontables librerías han hecho de NodeJS la opción favorita para los desarrolladores web.

MongoDB

MongoDB fue creada en 2007 por Eliot Horowitz y Dwight Merriman. La idea detrás de MongoDB era poder explotar las bases de datos no relacionales. La ventaja de estas bases de datos respecto las relacionales, aparte de la mayor flexibilidad que ofrecen, es la productividad para los desarrolladores. Los desarrolladores quieren un modelo de datos que tenga sentido para las estructuras que se quieren almacenar hoy en día. Las tablas y columnas son idóneas para contabilidad, registros, etc... pero ofrecen quebraderos de cabeza para su correcta organización para modelos de objetos con estructuras más complejas o que pueden cambiar con frencuencia.

[1] https://nodejs.org/en/

CAPÍTULO 3
Análisis del problema

A continuación se va a entrar en detalle sobre los problemas con los que se encuentra el equipo de Geomática y que vamos a intentar dar solución con este trabajo:

- El principal problema a resolver es el de poder permitir un fácil acceso a los datos proporcionados por los sensores de campo. Actualmente Geomática debe de lidiar con unos ficheros de texto muy engorrosos, y se pretende conseguir mediante la interfaz gráfica poder satisfacer las necesidades de los usuarios abstrayéndolos de todo el proceso de tratamiento de datos. Gracias a esto los usuarios podrán centrarse únicamente en la interpretación de los datos obtenidos.

- La situación actual respecto a los ficheros de datos da solo dos posibilidades para su posible tratamiento por parte del equipo de Geomática. La primera es el tratamiento manual de estos ficheros para obtener la información buscada, algo practicamente inviable debido a la extensión de estos ficheros. La segunda, y la que se está aplicando ahora mismo, es el procesamiento de cada fichero de texto mediante un script escrito en Python, que a su vez genera otro fichero de texto con los datos ya transformados al interés del usuario. Esto nos lleva al problema de que para poder tratar estos datos, debe de haber alguien con conocimientos mínimos en programación para poder crear dicho script. Uno de los objetivos de este trabajo es el poder abstraer a todos sus usuarios de este proceso. Además, es obvio que para un usuario estándar es mucho más sencillo interpretar datos de una interfaz gráfica que sobre los resultados volcados en un fichero de texto.

- Actualmente la información no está disponible en línea. Es necesario recoger el equipo tras la campaña para descargar los ficheros generados. Esto impide detectar problemas en la lectura de los datos hasta finalizar la campaña. Otro objetivo del proyecto será construir el sistema de tratamiento de ficheros a modo de servicio, que permita la conectividad entre el equipo de campo y el servidor de base de datos.

En definitiva, todo surge a partir de un problema de comodidad y facilidad para tratar datos. En lugar de poder centrarse plenamente en la interpretación de los datos, Geomática debe de perder tiempo tratándolos.

Para este proyecto se va a abordar una funcionalidad particular solicitada por un miembro del equipo de Geomática, que consistirá en a partir de los datos almacenados en MongoDB, poder generar un fichero de texto en formato CSV almacenando solamente una serie de datos concretos y que posteriormente le permitirán obtener una gráfica de tendencia de la humedad.

Casos de uso

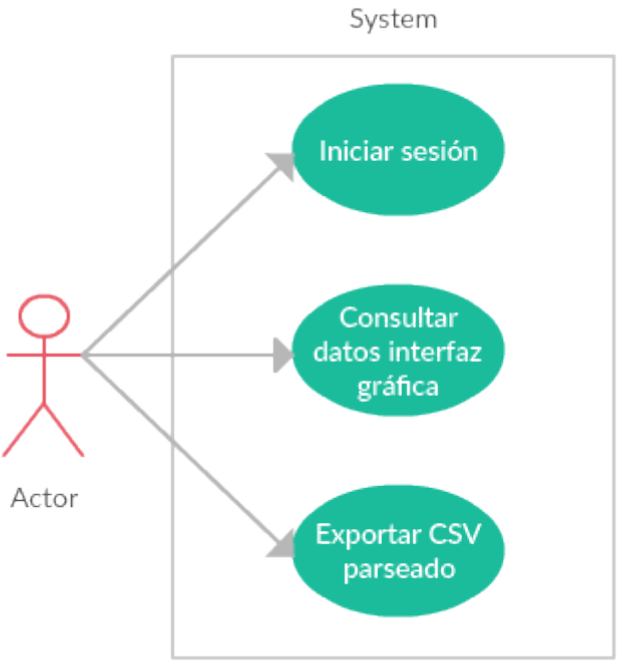

Figura 3.1: Casos de uso.

El objetivo de este proyecto es conseguir implementar tres casos de uso para el usuario. Como ya se ha mencionado anteriormente, dos de ellos serán casos generales para los usuarios de Geomática, donde deberán poder iniciar sesión y consultar todos los datos devueltos en el fichero de texto en la interfaz gráfica de usuario. El tercero será el caso de uso solicitado por el miembro de Geomática.

Requisitos funcionales

Nos encontramos con los siguientes requisitos funcionales para el sistema:

- Automatizar el sistema de transmisión de los ficheros desde su generación hasta que son alamcenados. Se deben de poder obtener en red estos ficheros, mediante la comunicación entre el sistema que recibe el fichero de texto, en este caso la Raspberry Pi, y el servidor que manejará la información, el APi desarrollada.
- Cada vez que se reciba un fichero de texto en la ruta expuesta por el API, mediante el protocolo HTTP, se deberá parsear a formato JSON y almacenar en MongoDB. Esta ruta deberá estar siempre disponible, para que tan pronto como la Raspberry decida enviar un fichero completo, pueda recibirlo el servidor.
- Se debe de poder solicitar el fichero de texto mediante el protocolo HTTP a la Raspberry en caso de que se quiera actualizar la información en un momento dado sin esperar a que se reciba el fichero completo.
- Se debe de poder parsear la información recibida en los ficheros de texto. El API desarrollada deberá tener la capacidad de entender las sentencias NMEA [1] volcadas

[1] http://www.gpsinformation.org/dale/nmea.htm

sobre el fichero por los sensores de campo, parsearlas a la estructura de datos de MongoDB (JSON), y almacenarlo en la misma.

- Mediante la interfaz gráfica, se debe de poder permitir a los usuarios el inicio de sesión, que asegure que el API no puede ser atacada con peticiones inesperadas desde dispositivos en la misma red.
- Mediante la interfaz gráfica, se debe de permitir al usuario consultar los datos parseados que se encuentren almacenados hasta el momento en la base de datos.
- Mediante la interfaz gráfica, se debe de permitir a los usuarios exportar, entre dos fechas seleccionadas, un fichero de texto CSV con la información necesaria para poder generar una gráfica con la tendencia de la humedad.

Requisitos no funcionales

- Al cargar la interfaz gráfica de usuario, que no lleve un tiempo excesivo mostrar los datos almacenados en base de datos. Para este fin se deberá paginar la tabla e ir recuperando entradas de la base de datos conforme el usuario quiera ver más datos, pero no es un requisito crítico.
- Disponibilidad de los datos gracias a la comunicación mediante el protocolo HTTP siguiendo el patrón REST. El protocolo HTTP es el estándar utilizado por los navegadores y puede ser consumido por prácticamente cualquier cliente. Si se hiciera la comunicación directamente por TCP a través de binarios por ejemplo, sería mucho más compleja la implementación de lado a lado de la comunicación. Además con el patrón REST separamos el cliente del servidor, y no necesita saber el cliente cual es el estado del servidor y viceversa. Esto permite que ambos puedan entender el mensaje del otro sin necesidad de haber establecido una comunicación previa. Esto se consigue gracias al uso de recursos en lugar de atacar desde un lado al otro mediante comandos. [7]

Lo que tarde la aplicación en procesar los ficheros de texto no debería ser un problema al estarse recibiendo periódicamente (por ejemplo cada hora). En el caso que el usuario quiera actualizar los datos desde la interfaz solicitándose un nuevo fichero, este debe ser pequeño.

También se ha estudiado el tamaño que pueden tener los ficheros de texto, pero no llegan a ser nunca lo suficientemente grandes como para requerir un tratamiento especial (por ejemplo con streams) por posible falta de memoria al procesarlos.

Análisis de la seguridad

La aplicación no maneja datos sensibles o personales del usuario que puedan requerir encriptación en su tratamiento, pero sí que al tener unas rutas expuestos será necesario un sistema de inicio de sesión para poder evitar procesar peticiones que se realicen sin autorización desde dentro de la misma red en la que se despliegue el proyecto.

Este inicio de sesión permitirá atacar las rutas con un token indentificativo que permitirá el acceso a las funcionalidades de la aplicación.

Tambien se validará que la información recibida en el fichero de texto sea válida mediante el mecanismo checksum. Los ficheros de texto en cada sentencia que contienen

devuelven al final de línea el valor de este checksum, que permite mediante un algoritmo local en el servidor validar que esta información es la que originalmente se generó y no ha sido corrompida en el camino. [3]

Análisis de riesgos

El proyecto cuanta con un riesgo técnico común a la mayoría de proyectos de software, y es que no acabe cumpliendo con sus requerimientos funcionales. Para poder mitigar este efecto, se ha desarrollado con esto en mente para que pueda ser modificado en el futuro con la mayor facilidad posible para añadir funcionalidades nuevas que si que resulten ventajosas para sus usuarios.

Al ser un proyecto innovador, no existe el riesgo de que los usuarios prefieran una herramienta anterior que usaran para consultar los datos. Sí que existe el un riesgo de usabilidad en cambio de que la herramienta no sea intuitiva y acabe no gustando a los usarios. También existe el riesgo de utilidad en el que los usarios no consideren que la herramienta es lo suificientemente útil para sus labores.

Identificación y análisis de soluciones posibles

Se han estudiado dos posibles enfoques distintos para la resolución del problema inicial planteado:

1. Se podría haber creado un simple script dentro del propio chip Raspberry Pi (con python, javascript, etc...), que parseara el fichero de texto diretamente al generarse a otro fichero en formato CSV con los datos ya ordenados de forma legible. El problema de esta solución es que no permite explotar estos datos parseados debido a la naturaleza de los ficheros de texto. No podrían ser reordenados, filtrados, acotados... ni existiría la posibilidad en el futuro de añadir nuevas funcionalidades como si que se puede hacer con una aplicación. Esta opción requeriría volver a formatear el fichero de texto cada vez que se quisiera organizar la información de forma distinta.

2. Podría haberse recurrido, de forma similar a la anterior, a volcar los datos del fichero de texto nada más lleguen sobre una hoja de cálculo. La ventaja que tendría esta opción respecto a la anterior es que en una hoja de cálculo si que se pueden añadir funcionalidades para ordenar los datos, incluso pudiéndose llegar a añadir la funcionalidad de exportar a otro CSV que interesara mediante un script combinando la opción anterior con esta. Esta solución sigue careciendo de la posibilidad de expandir en un futuro las funcionalidades extra que se quieran añadir, como por ejemplo obtención de gráficas, mapa con la posición de los sensores de campo e información sobre cada uno de ellos...

Se puede deducir de las opciones descartadas que lo que se buscaba además de las opciones de personalización que ofrece una interfaz gráfica era poder dejar abierto al futuro la posibilidad de añadir funcionalidades más complejas sobre la aplicación para poder seguir explotando los datos obteniudos como interese al equipo de Geomática.

Plan de trabajo

El proyecto se ha dividido en las siguientes partes partes:

- Análisis de los requisitos de la aplicación conjuntamente con el equipo de Geomática. Aquí es donde se descubre cuales son las necesidades de los futuros usuarios, y que funcionalidades deberán ser desarrolladas.
- Desarrollo del API, encargada del parseo de ficheros y su posterior almacenamiento en base de datos para poder ser servidos a la interfaz gráfica.
- Desarrollo del frontal. Será necesario acoplar la vista (interfaz) con el API para poder obtener los datos almacenados previamente por el API.
- Implantación. Se desplegarán tanto API como interfaz gráfica, junto con todas las librerías y paquetes necesarios. De esto se encargará Docker.
- Pruebas. Se deberán realizar pruebas en conjunto con el equipo de Geomática para asegurarse de que el funcionamiento es el esperado y cumple las expectativas de los usuarios.

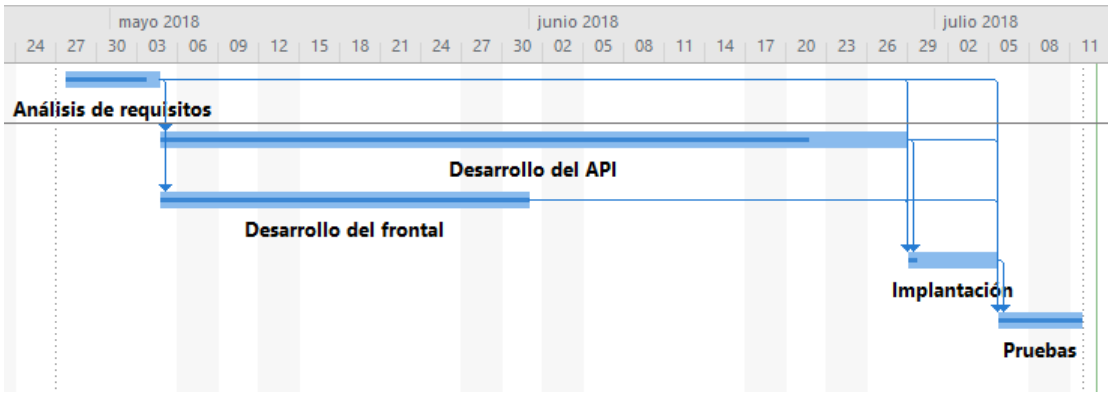

Figura 3.2: Diagrama de Gantt.

Presupuesto

El cálculo del presupuesto que sería necesario para llevar a cabo este proyecto podría desglosarse de la siguiente forma:

- Análisis de los requisitos de la aplicación. 10 horas.
- Desarrollo del API. 70 horas
- Desarrollo del frontal. 30 horas.
- Implantación. 8 horas.
- Pruebas. 5 horas.

Sabiendo que el salario medio en España para un informático junior es de 17.248€, que se quedaría en aproximadamente 8,5€/h. El proyecto require de 123h, por lo tanto necesitaríamos un presupuesto de unos 1050€ para el desarrollador.

A esto habría que sumarle 40€ del coste de hardware de la Raspberry Pi, además de un servidor en el que poder desplegar el proyecto, en este caso se ha elegido un mini PC de 134€.

Todo en conjunto subiría aproximadamente a los 1130€.

CAPÍTULO 4
Diseño de la solución

Tecnología utilizada

Lenguaje

Figura 4.1: Logo Javascript.

Javscript es un lenguaje de programación orientado a objetos [1]. Gracias a la gigantesca comunidad que lo acompaña, que ha crecido de forma masiva en los últimos años, han surgido una gran cantidad de frameworks que hacen de este lenguaje la opción idónea si se quiere desarrollar una aplicación web. En la siguiente sección veremos las poderosas herramientas que podemos utilizar en conjunto con Javascript.

Uno de los motivos más importante a la hora de decantarse por Javascript, ha sido que se puede desarrollar el proyecto de forma homogénea con el mismo lenguaje [2], tanto la parte del cliente como la del servidor.

MEAN Stack

Para el desarrollo del proyecto se ha optado por utilizar el stack MEAN. A continuación se hará una breve introducción a los componentes de este stack.

Figura 4.2: Logo NodeJS.

- NodeJS es un entorno de ejecución para Javascript basado en el motor V8 de Google. Creado en un principio para el desarrollo de aplicaciones web y APIs con un uso constante de I/O, su naturaleza basada en nodos permite la creación de aplicaciones altamente escalables.

  Ha sido elegido para este proyecto por dos principales motivos. El primero de ellos la existencia previa de conocimientos sobre esta tecnología del autor del proyecto. La segunda, que gracias al NPM [1] (Node Package Manager) y la enorme comunidad y constante soporte, se pueden encontrar paquetes de todo tipo para facilitar la labor de desarrollo y no centrarse en tareas triviales ya resueltas anteriormente por la comunidad.

Figura 4.3: Logo MongoDB.

- MongoDB es una base de datos no-relacional (NoSQL) [2], que utiliza como estructura para almacenar datos objetos JSON. Proporciona un esquema dinámico flexible a modificaciones sobre colecciones de datos ya existentes. Gracias a esto se pueden hacer cambios en la estructura de datos de forma sencilla incluso una vez las colecciones ya están populadas.

  Esta flexibilidad junto con la librería Mongoose [3] (explicada a continuación) hace MongoDB una elección ideal para almacenar la gran cantidad de campos que recuperan los sensores de campo.

Figura 4.4: Logo ExpressJS.

[1] https://www.npmjs.com/
[2] https://en.wikipedia.org/wiki/NoSQL
[3] http://mongoosejs.com/

- Express es desde hace tiempo un standard a utilizar en conjunto con NodeJS. Es un framework de NodeJS que se encarga de gestionar las peticiones y enrutarlas.

Figura 4.5: Logo AngularJS.

- AngularJS [4] es un framework de Javascript desarrollado por Google. Proporciona grandes funcionalidades dentro del propio cuerpo del código HTML, y permite la modificación de este en tiempo real sin necesidad de insertar scripts de Javascript o la utilización de frameworks como jQuery. Se basa en el MVC (Modelo Vista Controlador).

  Se ha optado por AngularJS sobre otras opciones como EmberJS o React debido a experiencia previa del autor con esta tecnología.

Otras tecnologías usadas

Figura 4.6: Logo Mongoose.

- Mongoose es una librería de NodeJS que facilita la integración de MongoDB con Javascript, proporcionando métodos para atacar la base de datos y un sistema de esquemas que valida la estructura del documento a insertar en la misma.

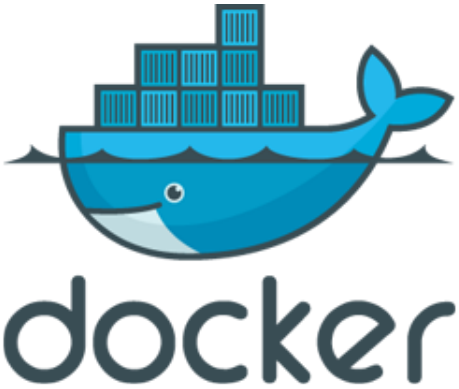

Figura 4.7: Logo Docker.

[4] https://angularjs.org/

- Docker [5] es una herramienta basada en contenedores de virtualización a nivel de sistema operativo. Permite ejecutar varios sistemas operativos nativamente y aislados entre sí. Solamente compartirían el kernel del sistema que levanta estos contenedores.

  Ha sido elegido para abstraer todo lo posible a los futuros usuarios del despliegue de la aplicación, ya que gracias a Docker se puede desplegar todo lo necesario para su correcto funcionamiento sin necesidad de que el usuario interectue de ninguna otra forma que no sea ejecutar el comando que arranca todo el proceso.

Figura 4.8: Logo PugJS.

- Pug [6] es una alternativa a HTML basado en el propio HTML. Ofrece algunas funcionalidades extra, como la capacidad de incluir código Javascript directamente en la vista, o crear componentes llamados "mixins", que pueden invocarse desde varios sitios e introducen ese componente en el cuerpo HTML, lo que evita tener que repetir código idéntico en distintas ventanas de la aplicación. También cambia la semántica del código. Pug es indentado, y no requiere cerrar llaves, por lo tanto la lectura del código es mucho más sencilla y clara en comparación a HTML puro.

Arquitectura del sistema

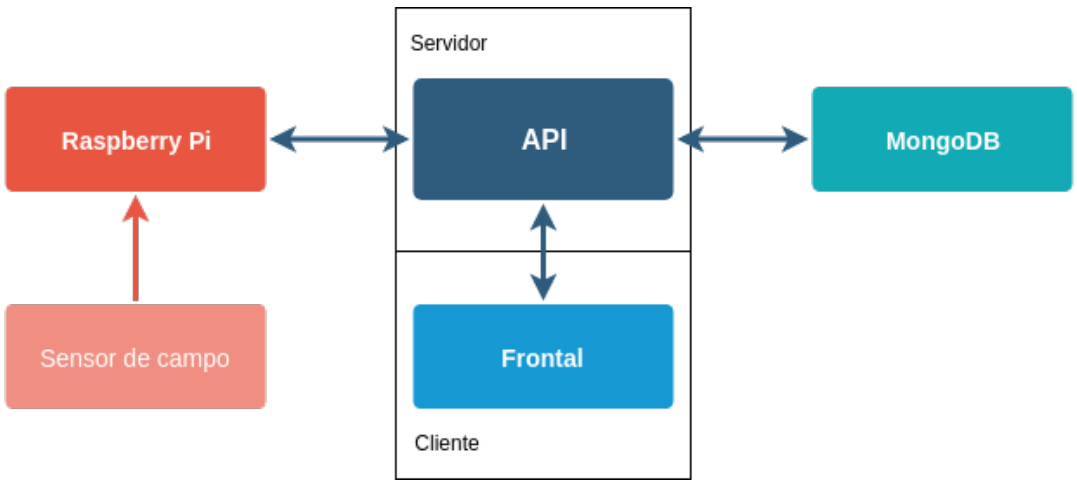

Figura 4.9: Componentes involucrados.

[5] https://www.docker.com/
[6] https://pugjs.org/

Para la arquitectura de sistema, se ha optado por desarrollar la parte servidor (API) y la parte cliente (frontal) individualmente [7], como recomienda el estilo arquitectónico en el que se basa la aplicación, REST [7]. De esta forma conseguimos poder mantener independientemente ambas partes, lo que nos permitiría incluso hacer cambios tan grandes en alguna de ellas como rehacerla con una tecnología alternativa.

La API será la encargada de gestionar las peticiones que puedan realizar tanto la Raspberry Pi (para enviar un fichero) como el cliente (para recuperar infromación de la base de datos o hacer *login* por ejemplo). Contendrá una serie de rutas y será la encargada del tratamiento de los ficheros de texto para acabar almacenando o recuperando los documentos en formato JSON en la base de datos.

El frontal, la parte cliente, a través de la API obtendrá los recursos que necesite para el correcto funcionamiento de las vetanas que lo compongan.

Además tendremos como base de datos MongoDB, encargada de guardar los documentos generados a partir de los ficheros de texto para servirlos posteriormente bajo demanda del API.

Los sensores de campo serán quienes proporcionen a la Raspberry Pi los datos leídos del GPS para que sean volcados sobre un fichero de texto.

Diseño detallado

API

El desarrollo de la API se ha hecho con la premisa de que el entorno de ejecución va a ser NodeJS. Express ha sido utilizado para manejar las peticiones a las rutas REST (procesamiento de la petición, enrutamiento de la misma, aplicación de middlewares si aplica, etc...) y mongoose para validar los modelos creados antes de insertar en la base de datos.

Las rutas REST han sido implementadas siguiendo lo que se consideran buenas prácticas dentro de la ingeniería del software [8]. La estructura de ficheros que componen el API se muestra en la imagen:

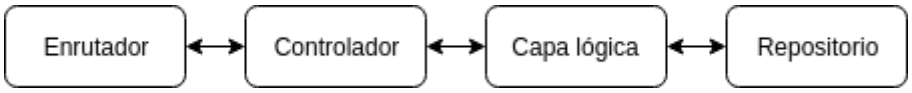

Figura 4.10: Estructura de los ficheros del API.

- Enrutador: Es el encargado de manejar por que flujo navegan las peticiones recibidas en la API, llamando a la función expuesta por el controlador correspondiente.
- Controlador: Se limita a obtener los datos que interesen que lleguen en la petición, y llamará a la capa lógica de negocio. Una vez obtenga respuesta devuelve la respuesta al cliente.
- Capa lógica o de negocio: Implementa la lógica de negocio y se encarga de llamar al repositorio para hacer peticiones a la base de datos. Aquí será por ejemplo donde se haga la tranformación del fichero de texto a JSON.
- Repositorio: Es donde se utiliza la librería mongoose. Aquí se hará la conexión con la base de datos. En el repositorio también se requerirá otro fichero nombrado DTO,

[7] https://en.wikipedia.org/wiki/Client%E2%80%93server_model

que será el que contenga la estructura que debe tener el documento que se va a insertar en la base de datos. Este DTO será un esquema de mongoose que en el repositorio generará el modelo necesario para las validaciones.

Las rutas REST que requieran de autenticación para poder acceder a ellas, contarán con un middleware (que manejará express) que validará que el usuario ha hecho login en el sistema antes de poder proseguir con la misma. En caso negativo devolverá un error de no autorizado.

Frontal

La parte del cliente se desarrollará con AngularJS y Pug en conjunto. Ambas tecnologías nos dan un gran control sobre el HTML de las ventanas del frontal. La lógica de las ventanas será controlada principalmente por AngularJS gracias a los controladores de cada ventana y los servicios proporcionados, ya sean nativos de AngularJS o creados para la ocasión.

Figura 4.11: Estructura de los ficheros del frontal.

- Vista: Creadas con Pug, contendrán las ventanas de la aplicación además de la lógica que AngularJS permite incluir en ellas para poder interactuar con el controlador.
- Controlador: Será donde se aplique la lógica de las ventanas y se traten los datos que recupere a través de los servicios inyectados para poder mostrarlos en la vista.
- Servicios: Abstraen el controlador de AngularJS de la comunicación con la API, a la que podrá realizar peticiones. Además aquí se pueden declarar métodos comunes que se quieran utilizar en distintos controladores para no repetir código. Casa servicio que interese ser utilizado deberá ser inyectado en el controlador y la vista.

CAPÍTULO 5
Desarrollo de la solución propuesta

API

Al haber sido desarrollada con NodeJS y Express, una vez esté levantado el servidor de Node, el API estará lista escuchando a cualquier petición que se realice a su URL base. Incluso si la ruta no existe, entrará por el punto de partida de la aplicación y express será quien se de cuenta que no existe un enrutador con tal fin, por lo que devolverá un error controlado.

Las rutas son las que implementan en la API la funcionalidad necesaria para el funcionamiento esperado de la aplicación.

Ruta	Requiere autenticación
/login	No
/gps	Si
/gps/filter-data	Si
/gps/export-csv	Si

Tabla 5.1: Rutas REST

A express se le indica que maneje las rutas de la siguiente forma:

router.js

```
1 const router = express.Router();
2
3 router.use('/gps', gps);
4 router.use('/login', login);
```

De esta forma express sabe que cualquier ruta comenzada por /gps debe ir al enrutador del GPS, lo mismo para /login:

gps.router.js

```
1 router.get('/', controller.getGpsData);
2 router.post('/', controller.parseTxtFile);
3 router.post('/filter-data', controller.filterData);
4 router.post('/export-csv', controller.exportCsv);
```

Una vez ha encontrado la ruta correspondiente, se llama al controlador que se encarga de obtener los datos de la petición y mandarlos a la capa lógica:

gps.controller.js

```
exports.filterData = async (req, res) => {
  const dateFrom = req.body.dateFrom;
  const dateTo = req.body.dateTo;
  const data = await bll.filterData(dateFrom, dateTo);
  res.send(data);
}
```

Una vez se encuentra en la capa lógica (bll en el código de business logic layer), se aplica la lógica de negocio requerida por la petición. Si la petición require llamada a la base de datos, de requerirá el repositorio para poder aprovechar las funciones de mongoose para hacer *queries* contra MongoDB.

Un ejemplo ilustrativo (ya que un ejempo real del GPS es largo y complejo de leer) de como funcionan las validaciones de mongoose es la siguiente:

person.dto.js

```
module.exports = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  age: {
    type: Number,
  }
});
```

Con esto estaríamos creando un esquema de mongoose para una persona, donde el nombre sería obligatorio y un string y la edad un número pero no requerida.

En el repositorio se importaría este DTO, y se generaría un modelo de mongoose a partir del esquema importado haciendo *const Person = mongoose.model('Person', schema);*. Con esto ya tendríamos acceso a todas las funciones que nos ofrece la librería, por ejemplo:

```
// Para buscar
Person.find().lean().exec();

// Para insertar
Person.save(data);
```

Con esto llegaríamos al final del flujo de la API, que devolvería la respuesta a través de express.

Es importante tener en cuenta que Javascript es un lenguaje asíncrono, por lo tanto cada vez que se haga una llamada asíncrona (como es el caso de una *query* a MongoDB) lo que será devuelto es una promesa. Una vez la llamada asíncrona haya finalizado y devuelto una respuesta, la promesa será resuelta. Gracias a las palabras reservadas de Javascript "async" y "await", podemos crear funciones asíncronas y esperar la respuesta en lugar de seguir con la ejecución antes de que la llamada haya sido resuelta.

Para el requisito del procesamiento de ficheros de texto, se debió valorar la posibilidad de que fueran muy grandes. Si este hubiera sido el caso, no se hubiera podido tratar el fichero directamente debido a la posibilidad de quedarse sin memoria. Se hubiera requerido una solución alternativa como los *streams* de NodeJS, que permiten mediante eventos ir procesando la información por bloques. Una vez se hubiera procesado un bloque (que sería el equivalente a una pequeña parte de ese fichero de texto), el *stream* nos

notificaría mediante un evento y pasaríamos al siguiente bloque. En el caso de este proyecto, no ha sido necesario ya que los ficheros de texto nunca van a alcanzar un tamaño que pueda dejar el servidor sin memoria.

En cuanto a la ruta para poder recuperar toda la información del GPS de base de datos, se ha debido en conjunto con el frontal, crear un sistema de paginado. En la API esto consiste en mediante opciones proporcionadas por MongoDB que mongoose sabe explotar, indicarle al hacer la petición que se quiere obtener solo un límite de resultados. Cuando se fuera a avanzar a la página siguiente en el frontal, en la petición al API se le indicaría el número de página para al realizar la siguiente petición saltarse los x primeros elementos que fueran ya recuperados, y obtener otra vez un número limitado de elementos pero que no coincidirían con los primeros.

Frontal

En la implementación de la parte del cliente, hemos aprovechado el *$scope* de AngularJS, que es un objeto JSON accesible tanto desde la vista HTML (PugJS en este caso) como desde el controlador de AngularJS.

Gracias a esto podemos por ejemplo, mediante un servicio que apunte a una ruta de la API que recupera los datos del GPS, guardar en el *$scope* los datos para después desde la vista en una tabla HTML acceder a este *$scope* y rellenarla de forma prácticamente trivial.

viewController.js

```
gpsService.retrieveData()
  .then((data) => {
    $scope.gpsData = data;
  })
```

view.pug

```
table
  thead
    tr
      th Fecha
      th Satelite
      th Latitud
      th Altitud
  tbody
    tr(ng-repeat="element in data") // no es necesario indicar
        $scope delante en la vista
      td {{element.date}}
      td {{element.satellite}}
      td {{element.latitude}}
      td {{element.altitude}}
```

CAPÍTULO 6
Implantación

El hecho de tener una serie de componentes que son independientes entre sí (Frontal, API, MongoDB) hace que debamos recurrir a Docker para poder hacer el despliegue de la forma más sencilla y escalable posible.

La parte del cliente debe de poder tener acceso al API, a quién le hará peticiones para aprovechar sus recursos y funcionalidades expuestas mediante rutas. También debe de poder estar accesible mediante el protocolo HTTP.

El API debe de poder tener recibir las peticiones tanto de la Raspberry PI como del cliente, lo que significa tener sus rutas expuestos y accesibles. También debe de poder conectarse a la base de datos de MongoDB para poder crear, modificar o consultar datos.

MongoDB debe de poder ser accesible desde la API porque es lo que utilizaremos para la persistencia de datos.

Docker nos va a permitir crear contenedores para estos componentes, y poder desplegarlos todos a la vez mediante un solo comando.

Docker

Es una herramienta de virtualización a nivel de sistema operativo que se basa en contenedores.

Linux da soporte a los contenedores y consigue aislar casi por completo las aplicaciones que se ejecutan en los mismos en el sistema operativo servidor. Así cada contenedor tiene sus propios procesos, su propio sistema operativo, su propia red, usuario y sistema de archivos, compartiendo solamente funciones del kernel para limitar y gestionar recursos (CPU, memoria, I/O y red).

Docker está compuesto de tres partes fundamentales:

- Dockerfile: es un fichero de texto que tiene todos los comandos que un usuario tendría que lanzar en la terminal para poder levantar el programa, es decir, instalar las dependencias y la aplicación.
- Imagen: es un archivo que tiene todos los datos necesarios para levantar un contenedor con la aplicación.
- Contenedor: un contenedor es una instancia de una imagen. En Docker cada contenedor es un proceso de individual de aplicación y, por cada imagen, podemos desplegar varios contenedores, que es idóneo para la escalabilidad.

Compose

Compose es una herramienta que permite de forma sencilla levantar en conjunto varios contenedores de docker mediante un fichero de configuración.

Gracias a esta herramienta podemos lanzar mediante un único comando todos los que deberían de ser necesarios por cada contenedor de docker para dejar desplegado por completo el proyecto.

CAPÍTULO 7
Pruebas

Se han llevado a cabo las siguientes pruebas:

- Se ha simulado la comunicación HTTP entre lo que sería la Raspberry Pi y el servidor API. En este proyecto no se ha abarcado la integración del chip. La prueba se ha realizado entre dos máquinas virtuales distintas en la misma red.
- Se ha validado el resultado del parseado de diversos ficheros de texto en conjunto con el equipo de Geomática para asegurarse que el resultado era el esperado.
- Se ha probado que el fichero CSV generado (funcionalidad del sistema) coincide con los ficheros que ya tenían en Geomática, utilizando como ficheros de prueba los que ellos habían parseado con anterioridad para dar validez a la prueba.
- Se ha probado el despliegue de los distintos componentes (servidor, cliente, base de datos) tanto individualmente como en conjunto con Docker.

CAPÍTULO 8
Conclusiones

Finalmente los objetivos inciales planteados para el proyecto han podido ser cumplidos con éxito. Se ha logrado implementar un API capaz de almacenar una gran cantidad de información, y poder acceder a esta de forma rápida y sencilla, además de visual para el usuario. El caso particular planteado por un miembro de Geomática ha podido ser llevado a cabo tambien y tiene accesible la generación del CSV a golpe de clic.

La posibilidad de que este proyecto pueda ser desplegado de forma sencilla con Docker no hace más que facilitar la labor de los posibles futuros usuarios que no tengan conocimientos informáticos. El único requerimiento es un servidor en el que poder realizar tal despliegue.

Por otra parte, se ha conseguido desarrollar un software totalmente escalable, el cual solo necesitaría de más servidores para poder desplegar más nodos. En el caso de que en un futuro se quisieran poner muchos más sensores de campo, la aplicación solo necesitaría añadir un balanceador de carga que permitiera distribuir equitativamente la carga de trabajo (parseo de ficheros de texto) sobre todos los nodos desplegados con Docker para aprovechar al máximo las posibilidades ofrecidas por NodeJS.

Gracias al proyecto hemos descubierto Docker, una gran herramienta para realizar despliegues que no conocíamos antes. Además al haber separado en comoponentes el proyecto hemos podido dar uso a la funcionalidad de Compose de Docker que ha resultado muy práctico al evitar tener que levantar individualmente cada uno de ellos.

Ha sido tambien una oportunidad para acercarse a un área en la que no habíamos trabajado anteriormente y que no está relacionada con la informática, gracias a haber trabajado de forma cercana con el equipo de Geomática.

Relación del trabajo desarrollado con los estudios cursados

El proyecto ha sido una gran oportunidad para poder exprimir los conocimientos adquiridos en la rama de ingeniería del software. En todo momento se han intentado mantener las buenas prácticas recomendadas para cada tecnología utilizada, y con las visiones de escalabilidad, usabilidad y fácil mantenimiento siempre en mente. Se han seguido guías de estilo instaladas en el editor de texto (eslint), que mejoran la legibilidad del código y utilizado alternativas como Pug para casos en los que el código inevitablemente suele acabar siendo complejo de leer (HTML).

CAPÍTULO 9
Trabajos futuros

Hubiera sido idóneo poder añadir una mayor cantidad de filtros en la interfaz de usuario. Hay una gran cantidad de datos y actualmente tras la finalización de este proyecto no se puede buscar por todos los campos o realizar filtros combinados de varios campos. Esto consideramos que sería un añadido muy interesante para el equipo de Geomática, que les abriría puertas para una mayor explotación de toda la información que tienen a su disposición.

Este proyecto ha sido realizado bajo la asumpción de que ya teníamos integrada la Raspberry Pi con el API desarrollada, usando ficheros de prueba para las cargas de datos. Un trabajo futuro que queda pendiente es esta integración, en la que la comunicación entre el chip y el API sea real.

También, y aprovechando un módulo de NodeJS, podríamos cambiar el enfoque visto en este proyecto para la solución de este problema. En lugar de basarse en la transimisión de los ficheros de texto, desarrollar un pequeño proyecto directamente en la Raspberry Pi, que haría uso del módulo de NodeJS serial-port, que permitiría escuchar en el puerto en el que la información de los sensores de campo está siendo recibida en tiempo real. Aquí, en lugar de esperar a que haya un fichero generado y enviarlo, se irían recibiendo las sentencias una a una, parseándolas al formato JSON, y transmitiéndolas mediante el protocolo HTTP a un servidor que solamente se encargue de la recepción y almacenamiento de cada objeto individualmente. Con esto, evitaríamos el parseado de los ficheros de texto completos.

Otro trabajo futuro podría ser necesario en el caso antes mencionado de que se quiera aumentar de forma considerable la cantidad de antenas, o la extensión del campo cubierto. Sería entonces necesario el desarrollo de un sencillo balanceador de carga que repartiera entre los nodos levantados con NodeJS la carga del procesamiento de ficheros.

Bibliografía

[1] Stoyan Stefanov. *Object-Oriented JavaScript*. Packt Publishing (July 24, 2008).

[2] The JavaScript Disruption. Consultat a http://www.richardrodger.com/2011/04/05/the-javascript-disruption/.

[3] Checksum Consultat a https://en.wikipedia.org/wiki/Checksum.

[4] A Brief History of JavaScript Consultat a https://auth0.com/blog/a-brief-history-of-javascript/.

[5] Desktop vs. Web Applications: A Deeper Look and Comparison Consultat a https://www.seguetech.com/desktop-vs-web-applications/.

[6] The benefits of web-based applications Consultat a https://www.magicwebsolutions.co.uk/blog/the-benefits-of-web-based-applications.htm.

[7] What is REST?. Consultat a https://www.codecademy.com/articles/what-is-rest.

[8] Matthias Biehl. *RESTful API Design: Best Practices in API Design with REST*. API-University; 1 edition (August 28, 2016).