UNIVERSITATJAUME I DE CASTELLÓN

MASTER UNIVERSITARIO EN SISTEMAS INTELIGENTES

CREACIÓN DE UN PROVEEDOR DE SERVICIOS WEB RESTFUL MEDIANTE EL USO DE NODE.JS. UN CASO DE ESTUDIO APLICADO A LAS TECNOLOGÍAS GEOESPACIALES.

MEMORIA DE PRÁCTICAS

Presentado por:

IGNACIO MIRALLES TENA

Dirigido por:

JOAQUÍN HUERTA GUIJARRO

CARLOS GRANELL CANUT

Castellón, Septiembre, 2015

ÍNDICE

RESUMEN 5

PALABRAS CLAVE 5

1 INTRODUCCIÓN 6

1.1 Objetivos 7

1.2 Metodología 7

2 REVISIÓN TEÓRICA 9

2.1 Node.js. 10

2.2 Módulos. 12

2.3 MongoDB. 16

3 CASO DE ESTUDIO 18

3.1 Definición y objetivos. 18

3.2 Implementación 19

3.3 Modelo de datos. 25

4 REFLEXIONES 29

5 CONCLUSIONES 32

BIBLIOGRAFÍA 34

Índice de Figuras.

Figura 1: Ejemplo de creación de un servidor básico con Node.js 11
Figura 2. Ejemplo de fichero de configuración de paquetes Node.js. 13
Figura 3. Ejemplo de aplicación Node.js utilizando Express. 14
Figura 4. Ejemplo de uso de la librería Mongoose. 15
Figura 5. Arquitectura del Caso de Estudio implementado. 20
Figura 6. Configuración de las rutas d ela interfaz RESTful en el Caso de Estudio. 21
Figura 7. Código del servidor para añadir funcionalidad "*Cross-Domain*" a las respuestas. 25
Figura 8. Resumen del modelo del Caso de Estudio. 26
Figura 9. Esquema del modelo del usuario. 26
Figura 10. Esquema del modelo de la parcela. 27
Figura 11. Esquema del modelo de la información de los cultivos (CropInfo). 27
Figura 12. Esquema del modelo de la enfermedad (Disease). 27

ACRÓNIMOS

- **API**: Application Programming Interface.
- **CSS**: Cascade Style Sheets.
- **EHR**: Electronic Health Record system.
- **ERMES**: an Earth obseRvation Model based ricE information Service.
- **GEOTEC**: Geospatial Technologies Research Group.
- **HTML**: HyperText MArkup Language.
- **INIT**: Institute of New Imaging Technologies.
- **JSON**: JavaScript Object Notation.
- **ORM**: Object Relational Mapping.
- **REST**: Representational State Transfer.
- **RFID**: Radio Frequency IDentification.
- **SQL**: Standar Query Language.
- **TII**: Trabajo de Introducción a la Investigación.

RESUMEN

En el proceso de desarrollo de aplicaciones independientes que comparten información se plantea comúnmente el problema de la integración de la misma. Entre las diferentes soluciones que se encuentran disponibles se ubica la de generar una completa interfaz REST a la que las aplicaciones clientes realicen las peticiones pertinentes, siendo la interfaz del servicio que ofrece el servidor la responsable de unificar las diferentes peticiones en una única base de datos y adaptando los formatos y tipos a un esquema común.

En este trabajo se ha realizado un estudio en el estado del arte de las posibles tecnologías de servidor disponibles, analizando con detalle los trabajos realizados sobre una tecnología actual y en auge como es Node.js. Se han valorado los resultados obtenidos con esta tecnología evaluando principalmente su rendimiento. Así mismo se han valorado las diferentes soluciones existentes para mantener la persistencia de los datos, analizando trabajos que comparan bases de datos relacionales con las tecnologías NoSQL, como MongoDB. Seguidamente se ha desarrollado un Caso de Estudio donde se ponen en práctica las tecnologías revisadas.

El Caso de Estudio desarrollado se integra dentro de un proyecto europeo centrado en mejorar el rendimiento del sector agrícola europeo, sirviendo como solución a la integración de datos de las diferentes aplicaciones clientes que se están desarrollando para el mismo.

El trabajo concluye realizando unas reflexiones que comparan la información extraída de la revisión teórica llevada a cabo con la experiencia adquirida mediante el desarrollo del Caso de Estudio.

PALABRAS CLAVE

Integración de información, tecnología servidor, Node.js, ERMES, agricultura inteligente, MongoDB, tecnologías geoespaciales.

1 INTRODUCCIÓN

Este trabajo de introducción a la investigación ha sido realizado en el marco del INIT (Institute of New Imaging Technologies) [1], concretamente en el grupo GEOTEC (Geospatial Technologies Research Group) [2]. Dicho grupo forma parte del consorcio del proyecto europeo ERMES (an Earth obseRvation Model based ricE information Service) [3]. Este proyecto está financiado por la Unión Europea, concretamente el Séptimo Programa Marco de la Unión Europea para acciones de investigación, desarrollo tecnológico y demostración (FP7/2007-2013).

El sector agrícola europeo, especialmente el que se refiere a los diferentes cultivos de arroz, tiene como objetivo mejorar su competitividad y minimizar el impacto ambiental de algunas de sus prácticas. El proyecto ERMES contribuirá a ese objetivo mediante la creación de métodos capaces de monitorizar el estado de los campos durante el proceso de cultivo, permitiendo reaccionar rápidamente a los posibles problemas.

En el marco de este proyecto se han desarrollado algunas herramientas de soporte para que los usuarios sean capaces de explorar y monitorizar en todo momento datos relevantes pertenecientes a los campos de cultivo de arroz. Estas aplicaciones comparten usuarios e información relativa a, por ejemplo, las parcelas pertenecientes a los agricultores propietarios. Para integrar estas aplicaciones se requiere un servidor común capaz de comunicar la información compartida, mantenerla íntegra y permitir múltiples conexiones simultaneas. El desarrollo de este servidor así como el estudio desde el punto de vista científico de la tecnología utilizada constituyen el cuerpo de este trabajo.

Parte de la problemática reside en que las diferentes aplicaciones que se conectan pueden ser heterogéneas, de forma que la integración de los datos es una característica fundamental para el servidor. Las diferentes semánticas utilizadas por los clientes no tienen por qué coincidir, dado que los desarrollos pueden provenir de diferentes miembros del consorcio.

El objetivo de este trabajo es analizar y validar las posibilidades que ofrece la tecnología Node.js [4] para proporcionar servicios Web y realizar esta integración de datos de forma transparente a los clientes. En el **segundo punto** de este trabajo se revisa la bibliografía existente relacionada con esta tecnología, así como con las librerías de las que se nutre. También se ha estudiado su integración con la base de datos NoSQL MongoDB [5].

Tras el barrido bibliográfico inicial y el análisis de las posibilidades de la tecnología, el **tercer apartado** presenta el desarrollo de un servidor que da soporte a la integración de todas las herramientas desarrolladas y a las posibles nuevas aplicaciones.

Es importante considerar que la aplicación de la tecnología se realiza sobre un entorno que debe suministrar información “a medida”, teniendo en cuenta, según la aplicación cliente, el perfil del usuario, su ubicación, el momento de la consulta, etc., en definitiva el contexto del usuario debe ser tenido en cuenta tanto por las aplicaciones cliente como por el servidor desarrollado a la hora de mostrar y devolver los datos.

Finalmente, en el **apartado cuatro**, se realiza una discusión donde se plasman los resultados obtenidos y la las reflexiones adquiridas a cerca de esta tecnología, comparando la información

extraída en la parte de análisis de los trabajos anteriores con la experiencia propia del caso de estudio desarrollado.

A continuación se presentan los objetivos del proyecto y del alumno, la metodología que se espera llevar a cabo y la planificación inicial.

1.1 Objetivos

El trabajo tiene como objetivo general la introducción a la investigación en el análisis de una tecnología en crecimiento como es Node.JS. Sus objetivos, desglosados, se enumeran a continuación:

- Adquirir experiencia en la realización de barridos bibliográficos en diferentes bases de datos científicas. Seleccionando para estudio aquellos artículos o trabajos relevantes para el trabajo.
- Analizar y comprender los artículos seleccionados, extrayendo la información más relevante.
- Ser capaz de sintetizar y plasmar sobre el trabajo toda la información obtenida, analizando las características de la tecnología estudiada y transmitiéndola de forma accesible. Además ser capaz de referenciar de forma coherente los trabajos analizados.
- Desarrollar un caso de estudio que, basándose en el análisis previo, sea capaz de dar soporte a las necesidades de un proyecto real y activo que requiere de una solución de integración.
- Integrar en el caso de estudio la solución a los problemas de integración de datos, semánticas heterogéneas y su aplicación a los sistemas contextuales.
- Adquirir capacidad crítica y reflexiva sobre los trabajos estudiados y compararlos con la experiencia real adquirida en el desarrollo del caso de estudio, además ser capaz de redactar y transmitir estas reflexiones de forma clara y accesible.

1.2 Metodología

La metodología llevada a cabo consiste en una serie de iteraciones junto a los tutores del trabajo en las que se vayan cerrando las siguientes fases:

1. Barrido bibliográfico y selección de artículos considerados relevantes.
2. Lectura y análisis de los artículos seleccionados, descartando aquellos que no encajen y volviendo a seleccionar nuevos en caso de considerarse necesario por parte del estudiante o los tutores.
3. Redacción de un informe detallado de la tecnología analizada que resuma todo el trabajo anterior. Este informe se incluye en la memoria siendo parte relevante de la misma.
4. Análisis de las necesidades del caso de estudio a llevar a cabo, en este caso se aplica una metodología basada en Sprints de Código que cumplen pequeños objetivos. Los tutores han analizado las soluciones que han ido implementándose y se ha ido iterando hasta dar soporte a todas las funcionalidades requeridas.

5. Comparación de la experiencia adquirida durante el desarrollo con el estado del arte previo, redacción de un informe capaz de representar las lecciones aprendidas. Nuevamente este informe se incluye en la memoria.
6. Redacción de la memoria integrando los informes redactados anteriormente.

2 REVISIÓN TEÓRICA

El primer paso llevado a cabo cuando se plantea la necesidad de desarrollar un servidor Web capaz de dar soporte a las diferentes aplicaciones implicadas en el proyecto, y las que podían desarrollarse en un futuro, ha sido acudir a una base de datos científica y comprobar los trabajos anteriores sobre el rendimiento de las diferentes posibilidades que se presentaban. La base de datos consultada ha sido Scopus [6] y gracias a la vinculación de la Universitat Jaume I con diferentes revistas científicas ha sido posible acceder a los artículos en su versión digital.

Existen múltiples estudios comparando diferentes tecnologías para servidor Web y analizando sus rendimientos. Titchkosky, L., Arlitt, M., y Williamson, C. (2003) [7] evaluaron el impacto de Perl, PHP y Java, demostrando que Java ofrecía mejor rendimiento que las otras dos en la generación dinámica de contenido. Trent, S., Tatsubori, M., Suzumura, T., Tozawa, A., y Onodera, T. (2008) [8] enfrentaron Apache (PHP) con Lighttpd (JSP) utilizando el benchmark SPECWeb2005 y concluyeron que JSP presentaba mejor rendimiento. Ranjan, A., Kumar, R., & Dhar, J. (2012) [9] compararon ASP.NET, JSP y PHP utilizando diferentes herramientas. Sumedh, J. H., Huy, J. C., Mungee, S., y Schmidt, D. C. (1998) [10] evaluaron el rendimiento en redes de área local. Ramana, U. V., y Prabhakar, T. V. (2005) [11] analizaron las diferencias entre PHP y C, concluyendo que el rendimiento de C era superior. Warner, S., y Worley, J. (2008) [12] hicieron notar la importancia de utilizar PHP frente a JSP para aplicaciones valoradas en entornos de ejecución reales. Más recientemente Neves, P., Paiva, N., y Durães, J. (2013) [13] compararon, añadiéndole la perspectiva de la seguridad, PHP con Java concluyendo que PHP presenta una mejor solución en términos de seguridad y escalabilidad.

Analizando todos los trabajos anteriores posiblemente se podría concluir que PHP presenta una solución óptima no solo desde el punto de vista del rendimiento, sino además con respecto a la seguridad. Sin embargo, un aspecto que también se consideraba relevante para la selección de la tecnología era el lenguaje utilizado, dado que el tiempo de desarrollo del que se disponía en el proyecto era limitado, encontrar la tecnología adecuada con cuyo lenguaje el estudiante ya estuviera familiarizado era un aspecto relevante. En este contexto, y continuando con la búsqueda, se encontró un trabajo reciente que compara PHP con Python y Node.js (Lei, K., Ma, Y., & Tan, Z. (2014) [14]).

En este trabajo se concluye que la tecnología Node.js ofrece mucho mejor rendimiento que PHP cuando existe una elevada concurrencia, aunque presenta como contrapunto que, al tratarse de una tecnología emergente, supone un problema para aquellos desarrolladores que no están familiarizados con la forma de programar asíncrona.

En otro trabajo reciente Chaniotis, I. K., Kyriakou, K. I. D., y Tselikas, N. D. (2014) [15] evalúan el rendimiento y las posibilidades que ofrece Node.js para desarrollar aplicaciones Web modernas. Aunque el trabajo presenta principalmente el escenario de las redes sociales, se puede extraer de él la conclusión de que, en lo referente al rendimiento, cuando se trata de generar información dinámica Node.js es superior a Apache, precisamente por su característica de ser asíncrono.

En el siguiente punto se describe con detalle la tecnología Node.js, sin embargo conviene notar en este momento que el lenguaje en el que se programa esta tecnología es JavaScript. Dado que algunas de las aplicaciones cliente han sido desarrolladas en este lenguaje, supone una ventaja mantener la misma forma de trabajar para el resto de módulos del proyecto.

2.1 Node.js.

Node.js es, según sus propios creadores [4], una plataforma construida sobre el *runtime* de Chrome para JavaScript para construir fácilmente aplicaciones de red escalables. Node.js utiliza el sistema de entrada salida (I/O) sin bloqueos basado en eventos que lo hace ligero y eficiente, ideal para aplicaciones en tiempo real que ofrecen datos a través de dispositivos distribuidos.

El proyecto fue puesto en marcha en 2009 y como apuntan Ojamaa, A., & Duuna, K. (2012) [16] ha ido creciendo muy rápidamente desde entonces, el repositorio de Github tiene más visitas que el de Ruby, por poner un ejemplo. Además, grandes compañías como Yahoo [17] o Microsoft [18] se han visto implicadas en el desarrollo o lo han integrado en la nube.

Tilkov, S., & Vinoski, S. (2010) [16] describen con detalle en su trabajo las características de Node.js y explican aquello que lo convierte en una solución adecuada para el desarrollo de aplicaciones Web modernas.

La plataforma esta implementada utilizando C y C++ haciendo énfasis en el rendimiento el consumo de memoria. Una de las principales características que posee es que no se basa en *multithreading* sino en un modelo asíncrono de I/O basado en eventos. El lenguaje que se utiliza para construir las aplicaciones es JavaScript lo cual presenta dos ventajas importantes:

- Soporta *callbacks* orientados a eventos de forma muy sencilla de programar.
- Dado que la web moderna se basa en HTML5 (incluyendo CSS3 y JavaScript), el desarrollo de la parte del servidor y del cliente puede realizarse utilizando el mismo lenguaje.

La ausencia de *multithreading* puede llamar la atención en un principio. Aunque muchos desarrolladores trabajan de esta forma, la gran mayoría coincide en que posee un alto grado de complejidad y el control de las aplicaciones queda muchas veces delegado a las decisiones y la

forma de trabajar del sistema operativo. Esto genera muchos problemas y bloqueos que son difíciles de controlar o solucionar. La orientación a eventos permite al programador tener mucho más control sobre lo que está ocurriendo. Desde este punto de vista la I/O asíncrona es especialmente relevante dado que permite evitar bloqueos. Las aplicaciones se suscriben a los eventos, de este modo, cuando las funciones de *callback* son llamadas, estas reciben la respuesta. Al contrario que en otras plataformas, en Node.js la asincronía es la forma de trabajar por defecto.

La Figura 1 presenta el trozo de código mínimo para construir un servidor en Node.js. En este caso únicamente responde con "Hola" a todas las peticiones entrantes, pero sirve para comprender de un vistazo la forma de trabajar que tiene la plataforma.

Figura 1: Ejemplo de creación de un servidor básico con Node.js

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8080, '192.168.1.1');
console.log('Servidor arrancado en http://192.168.1.1:8080/');
```

Esta figura también nos sirve para presentar la forma que tiene Node.js de importar los módulos o librerías que necesita. En este caso, por ejemplo, únicamente necesita la librería *http*, capaz de gestionar peticiones. En el apartado siguiente se profundiza más sobre los módulos existentes.

En el momento de realizar este trabajo la versión de la plataforma es la v0.12.7 y se sigue desarrollando para evitar algunas vulnerabilidades como por ejemplos las detectadas por Ojamaa, A., & Duuna, K. (2012) [16].

Otra de las características que le hace interesante es la facilidad que tiene para trabajar en bases de datos No SQL, como por ejemplo MongoDB, aunque esto se tratará más en detalle posteriormente, cabe decir aquí que el uso de JavaScript como lenguaje facilita mucho esta interacción.

2.2 Módulos.

Una de las grandes ventajas que presenta Node.js son los módulos creados por la enorme comunidad que le rodea. Muchas de las funcionalidades comúnmente requeridas (Transferencia de imágenes, gestión de peticiones, identificación de usuarios local o mediante redes sociales…) están programadas y encapsuladas en librerías o módulos que facilitan enormemente la labor de incluirlas en el servidor.

Para gestionar las librerías utilizados Node.js utiliza un gestor de paquetes para JavaScript llamado NPM [20]. Esta herramienta funciona a través de la línea de comandos y ofrece grandes ventajas a la hora de mantener actualizada una aplicación con dependencias. Por ejemplo, cuando se trabaja con Node.js existe un fichero de configuración llamado *package.json* en el cual se almacena información como la versión, dominio, scripts iniciales a ejecutar y las dependencias entre módulos. Un ejemplo de un fichero de este tipo se muestra en la Figura 2. En los valores de la clave "*dependencies*" se pueden ver los módulos requeridos para esa aplicación y la versión de cada uno de ellos. Si sobre esa aplicación se ejecuta el comando "npm install", el gestor de paquetes se encargaría automáticamente de instalar la versión correcta de cada una de las dependencias, facilitando muchísimo la labor de mover el servidor de sitio o generar un nuevo entorno de pruebas, etc.

Figura 2. Ejemplo de fichero de configuración de paquetes Node.js.

```
{
  "name": "Server-Name",
  "version": "0.0.1",
  "description": "This is a description.",
  "dependencies": {
    "bcrypt-nodejs": "0.0.3",
    "body-parser": "^1.5.1",
    "connect-flash": "^0.1.1",
    "connect-form": "^0.2.1",
    "consolidate": "^0.12.1",
    "express": "^4.7.1",
    "express-session": "^1.11.1",
    "jade": "^1.9.2",
    "method-override": "^2.1.2",
    "mongoose": "~3.6.11",
    "passport": "^0.2.1",
    "request": "^2.57.0"
  },
  "repository": {
    "type": "git",
    "url": ""
  },
  "scripts": {
    "start": "node server.js"
  },
  "private": true,
  "main": "server.js"
}
```

En el momento de la redacción de esta memoria existen aproximadamente 175.000 paquetes desarrollados y disponibles para su uso. En este trabajo únicamente se presentan algunos módulos muy populares y, especialmente, los que se han utilizado en el caso de estudio que se presenta en el próximo apartado.

- **Express** [21]: Express es posiblemente la librería más utilizada de Node.js y es difícil encontrar alguna aplicación que no la utilice. Según sus propios creadores consiste en un Framework Web flexible para Node.js capaz de proporcionar un conjunto de características para aplicaciones Web y móviles.

  Con Express la creación del servidor se abstrae todavía más, mediante la utilización de muy pocas líneas de código la herramienta es capaz de gestionar peticiones y encapsular la mayoría de las necesidades de bajo nivel del servidor.

  En la Figura 3 se muestra un sencillo ejemplo de todo el código necesario para un servidor utilizando este Framework.

Figura 3. Ejemplo de aplicación Node.js utilizando Express.

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Servidor funcionando.');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Ejemplo de aplicación en http://%s:%s', host, port);
});
```

- **Passport** [22]: Passport consiste en el software intermediario para la gestión de la identificación en el servidor. Se trata de una herramienta que permite gestionar aspectos como los formularios de identificación en bases de datos locales o relacionar nuestras aplicaciones Web con las principales cuentas de Internet, como Gmail, Facebook o Twitter por ejemplo.

  Passport utiliza el concepto de estrategias para identificar las peticiones. Estas estrategias pueden variar, desde el uso de usuario y contraseña convencional hasta mecanismos de autenticación más elaborados como OAuth [23] o OpenID [24]. En el momento de redacción de esta memoria Passport cuenta con más de 300 estrategias diferentes.

- **Mongoose** [25]: Esta librería ofrece herramientas para modelar los datos de la aplicación. Permite crear objetos capaces de contener información diversa y que puede comunicarse directamente con la base de datos NoSQL MongoDB.

  La herramienta permite realizar consultas y reaccionar a los resultados de las búsquedas de forma adecuada. En la Figura 4 se muestra un ejemplo de lo sencillo que resulta crear un objeto y almacenarlo en una base de datos.

Figura 4. Ejemplo de uso de la librería Mongoose.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/MiBBDD);

var Parcela = mongoose.model('Parcela', { id: Number, nombre: String });

var par = new Parcela({ id: 123, nombre: 'Zona Norte' });
par.save(function (err) {
  if (err) console.log('La parcela no puede almacenarse');
  else console.log('La información ha sido almacenada.');
});
```

- **Multer** [26]: Esta librería consiste en un software intermediario centrado principalmente en facilitar la gestión de formularios y la gestión de ficheros en el servidor.
  La principal característica es que añade en la petición un objeto "*file*" capaz de gestionar ficheros, además de todos los valores de los campos del formulario. Aunque resulta una funcionalidad simple, esta librería es capaz de facilitar muchísimo las características de manejo de ficheros imprescindibles en muchos servidores.

- **Bcrypt** [27]: Se trata de una librería nativa de JavaScript adaptada a Node.js. Su funcionalidad consiste en permitir encriptar valores (las contraseñas, principalmente) para mejorar la seguridad de la información.

- **Consolidate** [28]: Consolidate es la librería que sirve para manejar las plantillas o *Templates* que se utilizan en el sitio Web. En el momento de la redacción de esta memoria es capaz de soportar más de 30 motores de plantillas (como por ejemplo ejs o jade).
  Mediante el uso de esta librería es muy sencillo trabajar con las respuestas del servidor utilizando el motor de plantillas que se prefiera. También es importante considerar que, aunque esta librería permite trabajar con los motores de forma sencilla, las librerías propias de cada motor siguen siendo necesarias, por lo tanto hay que importarlas. Nuevamente, con NPM es sencillo localizar el motor deseado y utilizarlo en combinación con Consolidate, o cualquiera de las librerías presentadas anteriormente.

Estas son únicamente algunas de las librerías utilizadas, pero sirven para hacerse una idea del tipo de funcionalidades que pueden utilizarse y aparecer en los servidores Node.js y quizá una de las principales características que le convierten en un buen candidato para prácticamente cualquier tipo de aplicación Web, dado que la gran mayoría de tareas están programadas, documentadas y puestas a disposición del desarrollador.

2.3 MongoDB.

Elmasri, R. y Navathe, S. (2011) [29] afirman que la gran mayoría de implementaciones de bases de datos hoy en día están basadas en el modelo relacional, utilizando principalmente SQL como lenguaje de consulta. A pesar de ello las diferentes implementaciones de NoSQL (Not Only SQL) son cada vez más habituales en el mercado. Generalmente este tipo de base de datos carece de una estructura sólida que no encaja correctamente con el modelo relacional.

Parker, Z., Poe, S., y Vrbsky, S. V. (2013) [30] enumeran en su trabajo algunos ejemplos de su uso como la ubicación de *smartphones,* información de cámaras de vigilancia, y metadatos en páginas Web. En ese mismo artículo comparan un ejemplo de implementación de MongoDB con uno que utiliza el estándar de SQL: Microsoft SQL Server. Los resultados demuestran que para aquellos casos en los que el esquema varía de forma habitual el rendimiento de MongoDB es notablemente superior. En el caso de estudio que se desarrolla en el marco del proyecto ERMES los requisitos son muy cambiantes y las estructuras que hoy son válidas pueden dejar de serlo cuando las necesidades de alguno de los miembros del consorcio o de las diferentes aplicaciones cliente necesitan añadir o modificar información, por ese motivo la decisión fue buscar una base de datos NoSQL.

Dos de las más populares bases de datos de este tipo son MongoDB [5] y Cassandra [31]. Abramova, V., & Bernardino, J. (2013) [32] comparan en su trabajo estas dos tecnologías realizando pruebas en diferentes escenarios (lecturas/escrituras/actualizaciones…). Los resultados concluyen que, en general, Cassandra ofrece resultados ligeramente superiores. Sin embargo en el momento de buscar documentación en la Web se encuentra que la cantidad de material de ayuda para trabajar con Cassandra es bastante inferior que el que hay relacionado con MongoDb. Por ejemplo, realizando una búsqueda en *StackOverflow* se obtienen la siguiente cantidad de respuestas:

- Cassandra: 18.971 Resultados.
- MongoDB 48.986 Resultados.

Por este motivo finalmente el TII se centró en el uso de MongoDB.

MongoDB guarda las estructuras de datos en documentos con formato JSON [33]y es utilizada por empresas como Craiglist, MTV o Foursquare entre otras. Sus principales características se enumeran en la propia Web de los creadores:

- Se trata de un **modelo de datos flexible**, lo que facilita la adaptación dinámica a los cambios en las estructuras.

- Es **fácilmente escalable**, por lo que es capaz de soportar un aumento repentino en la cantidad de información almacenada.
- Ofrece un **paquete de herramientas robusto**, permitiendo modificar la BBDD, realizar copias de seguridad o monitorizar de forma sencilla.
- El **lenguaje de consulta** es **expresivo** y sencillo de dominar.
- La **comunidad aumenta rápidamente**, con las ventajas que ello proporciona.

Adicionalmente a todo lo mencionado anteriormente, MongoDB ya ha sido utilizado previamente tanto en herramientas de integración como en el desarrollo de soluciones de Sistemas de Información Geográfica (SIG). Por ejemplo Ameri, P., Grabowski, U., Meyer, J., & Streit, A. (2014) [34] lo han utilizado para trabajar con información climática periódica proveniente de dos satélites diferentes, demostrando además que los resultados con respecto al rendimiento se habían mejorado en un factor de 46. Xinwei, J., & Guicheng, S. (2014) [35] lo han utilizado para almacenar la enorme cantidad de información que se obtiene al trabajar con RFID. Con respecto a la complejidad en la integración Ojamaa, A., & Duuna, K. (2012) [36] han utilizado con éxito MongoDB para solucionar los problemas de complejidad que genera el *Electronic Health Record system (EHR)*, demostrando que el rendimiento de su propuesta es considerablemente mejor que las aproximaciones basadas en SQL.

3 CASO DE ESTUDIO

Tras analizar el estado del arte de la tecnología Node.js, así como de la BBDD MongoDB y los algunos de los diferentes módulos de ayuda que existe se ha desarrollado un caso de estudio que cubre los siguientes objetivos:

- Ofrecer una **solución de integración** para las diferentes aplicaciones existentes dentro del proyecto ERMES.
- **Experimentar** de forma práctica parte de la información adquirida durante el análisis inicial de la bibliografía.

En este apartado se da una descripción o introducción general al servidor. A continuación se presenta la implementación llevada a cabo y finalmente se describe el modelo de datos utilizado mediante el uso de la tecnología MongoDB.

3.1 Definición y objetivos.

El **objetivo** del servidor es ofrecer una interfaz RESTful para que las diferentes aplicaciones sean capaces de integrar la información de usuarios y parcelas. De este modo un usuario debe ser capaz de darse de alta utilizando cualquiera de las aplicaciones que se han ido desarrollando (y van a continuar desarrollándose). Ese mismo usuario estará disponible para el resto de aplicaciones, lo mismo ocurrirá con la manipulación de parcelas. El usuario puede acceder a sus parcelas, añadir, modificar o eliminar información de ellas y estas actualizaciones se encuentran disponibles inmediatamente para el resto de aplicaciones.

Todo el **código** del servidor Web se encuentra disponible públicamente en un repositorio GIT. Es importante tener en cuenta que en el momento de redacción de este trabajo, aunque la aplicación ya se encuentra desplegada y en funcionamiento, todavía se están solicitando algunas pequeñas actualizaciones o modificaciones en las aplicaciones que, en ocasiones, requieren de pequeños cambios en el servidor. Por ese motivo es posible que algunas de las cosas mencionadas aquí tengan diferencias con el repositorio. Esto no ocurre así con el código entregado junto a la memoria, el cual no ha sido modificado desde la redacción de la misma:

https://github.com/ermes-fp7space/dataAPI-prototype.git

El servidor se encuentra desplegado en una máquina con Windows 2008 Server y escucha en el puerto 6585 para no solaparse con otras aplicaciones ofrecidas por el mismo servidor. La URL inicial del servidor es: http://ermes.dlsi.uji.es:6585/. Accediendo a ella se ha programado una respuesta que simplemente ofrece *feedback* sobre si el servidor se encuentra disponible o no.

A continuación se enumeran, de forma descriptiva, los **servicios** que debe ofrecer el servidor:

- Crear un nuevo usuario.
- Identificarse con un usuario.
- Insertar parcelas en un usuario.
- Localizar toda la información de un usuario por nombre.
- Localizar la información resumida de un usuario dado su nombre.
- Actualizar la última posición del usuario.
- Insertar diferentes tipos de productos en una parcela de un usuario, por ejemplo: Estado de la parcela, información sobre el riego, agroquímicos utilizados, enfermedades detectadas…

Todas estas necesidades forman parte de los requisitos de las diferentes aplicaciones del proyecto, los cuales quedan fuera del alcance de este trabajo.

3.2 Implementación

La implementación del servidor trata de seguir un patrón similar al de Modelo, Vista Controlador. En este caso no existe vista, dado que de la parte accesible por el usuario se encargan las propias aplicaciones. La parte pública, o visible, son los servicios Web disponibles, sobre ellos se pueden realizar peticiones GET o POST que devuelven la información requerida y actualizan la base de datos en caso de que sea necesario.

La Figura 5 presenta la arquitectura completa del servidor creado. A continuación se explican las funciones y detalles de las diferentes áreas del gráfico.

En primer lugar se presenta la parte accesible desde fuera, en la figura se representa como una caja verde que contiene tres ficheros:

- **server.js**: es el fichero principal del servidor, el que arranca todo lo demás. El fichero comienza importando todos los módulos que van a ser necesarios en la aplicación. Entre ellos se encuentran los mencionados en el apartado anterior: Passport, Express, Mongoose, etc. En esta parte también se importan los ficheros que definen el modelo de datos, aunque estos quedarán explicados más adelante.

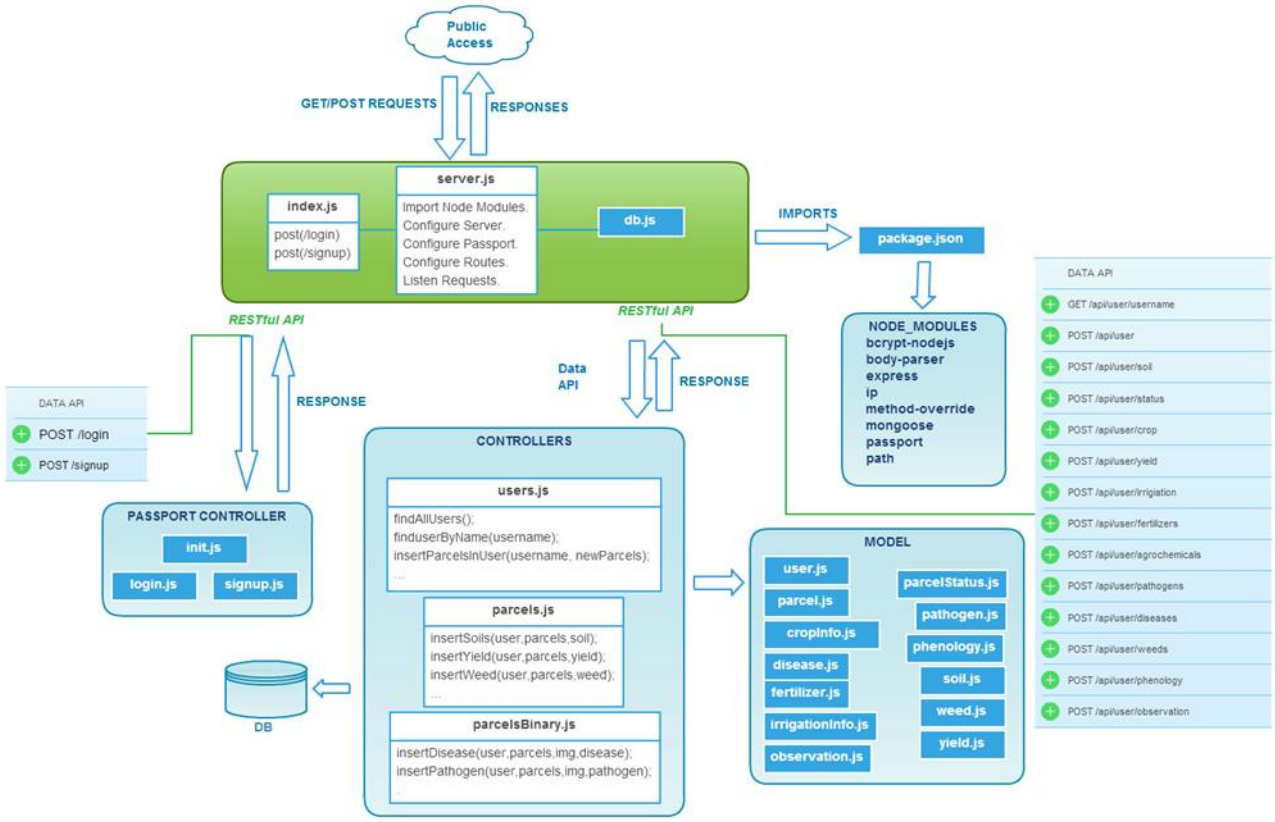

Figura 5. Arquitectura del Caso de Estudio implementado.

Una vez se han importado los módulos o ficheros necesarios el servidor configura algunos aspectos como el tipo de indentación en las respuestas, la forma de *parsear* los ficheros JSON, etc. A continuación configura las rutas del módulo PASSPORT, que será el encargado de gestionar las altas e identificaciones de usuarios.

El siguiente paso es configurar todas las rutas que van a ofrecerse de forma pública, la Figura 6 muestra un ejemplo que sirve para clarificar y mostrar el funcionamiento que tiene Node.js en este aspecto.

```
//Users API Routes
var usersApiRoute = "/users";
var usersApi = express.Router();
usersApi.route(usersApiRoute)
    .post(UsersController.insertParcelsInUser);
usersApi.route(usersApiRoute + '/:username')
    .get(UsersController.findUserByName);
usersApi.route(usersApiRoute + '/sort/:username')
    .get(UsersController.findShortUserByName);
usersApi.route(usersApiRoute + '/short/:username')
    .get(UsersController.findShortUserByName);
usersApi.route(usersApiRoute + '/position')
    .post(UsersController.insertLastPosition);
app.use('/api', usersApi);
```

Figura 6. Configuración de las rutas d ela interfaz RESTful en el Caso de Estudio.

En primer lugar se define la variable que almacena la ruta donde escuchará el servidor. En el ejemplo, "/users", después se obtiene, del módulo EXPRESS, la función de enrutado, que es la que se encarga de gestionar las rutas del servidor. Una vez tenemos estas dos variables simplemente se añaden nuevas rutas (colgando siempre de la primera) que escuchan en modo GET o POST, según el método utilizado, y redireccionan esas peticiones a un nuevo controlador, en este caso "UsersController", y sus diferentes funciones, una para cada petición. Una vez se han relacionado todas las rutas y métodos se le indica al servidor que utilice la configuración de rutas definida. Es posible añadir diferentes enrutados al mismo servidor, por lo que podemos crear, como se ha hecho aquí, uno para los usuarios, otro para las parcelas,… manteniendo de esta forma mucho más claro y modularizado el código.

Finalmente se le indica al servidor que escuche en el puerto indicado.

- **index.js**: Es el fichero inicial de PASSPORT, es el que configura que controladores se utilizarán. En este caso se inicializa con init.js, se envían los nuevos registros a login.js y las identificaciones a signup.js.

- **db.js**: Este fichero únicamente se encarga de configurar la ruta a la BBDD mongo.

Tanto index.js como bd.js son realmente controladores o ficheros de configuración que utiliza server.js para su configuración, sin embargo se consideran parte de esta primera fase donde se configura e inicializa el resto de la aplicación.

Todos los módulos externos, como EXPRESS, que importa server.js deben estar instalados en el servidor. Para ello existe el fichero **package.json**, en el cual se enumeran las dependencias e incluso

las versiones de cada una de ellas. Esta parte es responsabilidad de Node.js, por lo que simplemente indicando que módulos se necesitan y ejecutando el comando necesario, todas las dependencias son descargadas e instaladas automáticamente. Para ello se crea la carpeta **NODE_MODULES**, donde se van introduciendo y actualizando los ficheros necesarios de forma automática.

De este modo los controladores se dividen en dos secciones diferentes: Aquellos destinados a la gestión de usuarios y contraseñas (controlador del módulo PASSPORT) y aquellos que se encargan de la información de las parcelas.

Dentro de los controladores de usuarios y contraseñas se han programado tres ficheros:

- **init.js**: Se encarga de importar los otros dos controladores y de serializar y deserializar los usuarios que conectan y desconectan, es decir, los prepara para poder ser transferidos a través de la red en caso de que sea necesario.
- **login.js**: Es el responsable de comprobar que los datos introducidos sean correctos y reaccionar en consecuencia, ya sea devolviendo el error correspondiente o la información del usuario identificado. Para ello utiliza una de las estrategias de PASSPORT que se comentaban en el apartado anterior.

  Tanto este fichero como el siguiente (signup.js) aprovechan un módulo llamado "bcrypt-nodejs" para almacenar, encriptar y comparar las contraseñas introducidas.
- **signup.js**: Este controlador se encarga, en primer lugar, de comprobar si el usuario creado ya existe. De no ser así lo crea y lo añade a la base de datos. Finalmente responde en consecuencia.

Estos ficheros tienen acceso al modelo, del que se hablará posteriormente, para consultar o actualizar los datos de los usuarios.

Con el controlador anterior y el fichero inicial, explicado al comienzo del apartado, se tiene cubierta la funcionalidad para gestionar los usuarios, es decir, ofrecen los servicios REST para *login* y *signup:*

- POST: http://ermes.dlsi.uji.es:6585/login

- POST: http://ermes.dlsi.uji.es:6585/signup

A continuación se describen los controladores creados para la gestión de parcelas. Todos ellos, acceden al modelo completo y se han dividido en tres ficheros. El primero de ellos se responsabiliza de las consultas y accesos de los usuarios (añadir parcelas, localizar los datos de un usuario…). El segundo se encarga de los datos de las parcelas que no requieren de trasferencia de datos binarios, es decir, la información en formato texto plano. El tercero se responsabiliza de aquellos datos de parcelas que deben incluir información binaria (básicamente imágenes). Esta división se ha hecho así por dos motivos: en primer lugar para mantener separados los diferentes tipos de datos y en segundo lugar porque para el tratamiento de información binaria es necesario utilizar algunos módulos que modifican el código de las funciones, por lo que se ha preferido dividir en dos ficheros.

- **users.js**: Este controlador es responsable de dar funcionalidad a los servicios de consulta de usuarios e inserción de parcelas de los usuarios. Las funciones sobre las que da soporte son las siguientes:
    - Localizar todos los usuarios. Devuelve toda la información de la base de datos. Se creó para facilitar el desarrollo y en el momento de redacción de este trabajo no se encuentra disponible, es decir, no hay ningún servicio que acceda a ella.
    - Insertar parcelas en usuario. Lo que hace es actualizar las parcelas de un usuario, eliminar, añadir o modificar las que tiene en relación a las que se le pasan. Se accede mediante un POST a http://ermes.dlsi.uji.es:6585/api/users. Entre los parámetros que recibe debe incluirse usuario y contraseña para comprobar que el usuario que hace la petición es el correcto.
    - Localizar usuario por nombre. Dado un nombre de usuario devuelve toda su información. En este caso se trata de una petición GET sobre http://ermes.dlsi.uji.es:6585/api/users/NOMBRE_DE_USUARIO. En un principio este servicio estaba protegido para que únicamente usuarios identificados fueran capaces de obtener esta información, sin embargo desde los desarrollos de las diferentes aplicaciones, dado el carácter investigador del proyecto, se solicitó que estas peticiones fueran libres, es decir, cualquiera que conozca el nombre de usuario es capaz de obtener toda la información del mismo, a excepción de la contraseña.
    - Localizar información resumida por nombre de usuario. Dado que algunos usuarios poseen demasiada información se requería un servicio que ofreciera información breve sobre el usuario (correo, última posición, listado de parcelas poseídas…). Para

ello se creó un servicio que devuelve únicamente esta información. Como en el caso anterior se trata de una petición GET sobre http://ermes.dlsi.uji.es:6585/api/users/short/NOMBRE_DE_USUARIO. Nuevamente el servicio es de libre acceso, sin requerir contraseña ni identificación.

  - Actualizar la última posición del usuario. El usuario almacena su última posición para recordarlo en las diferentes aplicaciones, para ello es necesario un servicio que la actualice, en este caso se trata de un servicio POST a http://ermes.dlsi.uji.es:6585/api/users/position. Para este servicio sí que es necesario pasar, además de la posición y el usuario, le contraseña para comprobar que la inserción es segura.

- **parcels.js**: Este controlador es el responsable de gestionar las peticiones de actualización de información de parcelas. Para cada parcela existen una serie de productos que pueden ser introducidos, un producto consta de una serie de campos como puede ser fecha, tipo de fertilizante utilizado, horas de riego aplicadas, etc… Para este controlador se generó debate sobre cuál era la mejor forma de proceder, se plantearon dos posibles soluciones:
  - Generar un único servicio Web en cuya petición se incluyera el producto a actualizar y posteriormente los campos.
  - Generar, para cada producto, un servicio Web diferente.

  Tras debatir con los tutores y otros miembros del consorcio se decidió que era más sencillo para las aplicaciones posteriores ofrecer un servicio Web para cada producto. De modo que, de este controlador, se generan varios servicios POST, ocho en concreto:
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/soil
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/parcelStatus
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/cropInfo
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/yield
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/irrigationInfo
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/fertilizer
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/agrochemical
    - POST: http://ermes.dlsi.uji.es:6585/api/parcels/phenology

  El controlador en primer lugar comprueba que el usuario que ha hecho la petición está correctamente identificado (mediante el usuario y la contraseña), seguidamente comprueba que la parcela o parcelas sobre las que quiere realizar la inserción se encuentra entre las que el posee. Si se cumplen esos requisitos se conecta al modelo, crea una nueva instancia del producto en concreto y la inserta en la BBDD.

- **parcelsBinary.js**: Este controlador hace prácticamente lo mismo que el anterior, con la diferencia de que contiene funcionalidad para trabajar con datos binarios para poder gestionar las imágenes.

  Para trabajar con este tipo de datos hace uso de una librería llamada MULTER [26] que facilita la posibilidad de almacenarlas, de abrir un buffer de conexión, etc. Además se requiere la librería de acceso al sistema FS (*FileSystem* [38]) para poder acceder directamente al sistema de ficheros del servidor.

  El controlador realiza las mismas comprobaciones que el anterior: Por un lado usuario y contraseña, por el otro que las parcelas indicadas sean propiedad del usuario. Finalmente, haciendo uso de las librerías mencionadas, almacena tanto los datos de texto como las imágenes enviadas en la petición. Los servicios que gestiona son los siguientes:

  - POST: http://ermes.dlsi.uji.es:6585/api/parcels/pathogen
  - POST: http://ermes.dlsi.uji.es:6585/api/parcels/disease
  - POST: http://ermes.dlsi.uji.es:6585/api/parcels/weed
  - POST: http://ermes.dlsi.uji.es:6585/api/parcels/observation

Un detalle relevante de la implementación es que, en todas las respuestas, el servidor Node.js añade a la cabecera los siguientes campos:

- Access-Control-Allow-Origin -> *
- Access-Control-Allow-Headers -> `Origin, X-Requested-With, Content-Type, Accept`

Dado que no es posible controlar donde se alojarán las aplicaciones cliente, estos dos parámetros evitan potenciales problemas derivados de la política de dominios distintos (*Same-origin policy* [37]). El código necesario para añadir esto a la cabecera se muestra en la Figura 7.

```
res.header("Access-Control-Allow-Origin", "*");
res.header("Access-Control-Allow-Headers",
                                "Origin, X-Requested-With, Content-Type, Accept");
```

Figura 7. Código del servidor para añadir funcionalidad "*Cross-Domain*" a las respuestas.

3.3 Modelo de datos.

Como se indicó anteriormente, la base de datos utilizada en el Caso de Estudio es MongoDB. En esta BBDD la información no se almacena en tablas, sino en ficheros JSON con una estructura libre y adaptable.

Dados los requisitos de las aplicaciones el componente principal de los datos es el usuario, este usuario posee cierta información, incluida la de las parcelas. Y cada una de las parcelas puede

contener diferente cantidad de productos, que a su vez poseen su propia información. La Figura 8 muestra un diagrama que resume las relaciones entre los distintos objetos del modelo de datos.

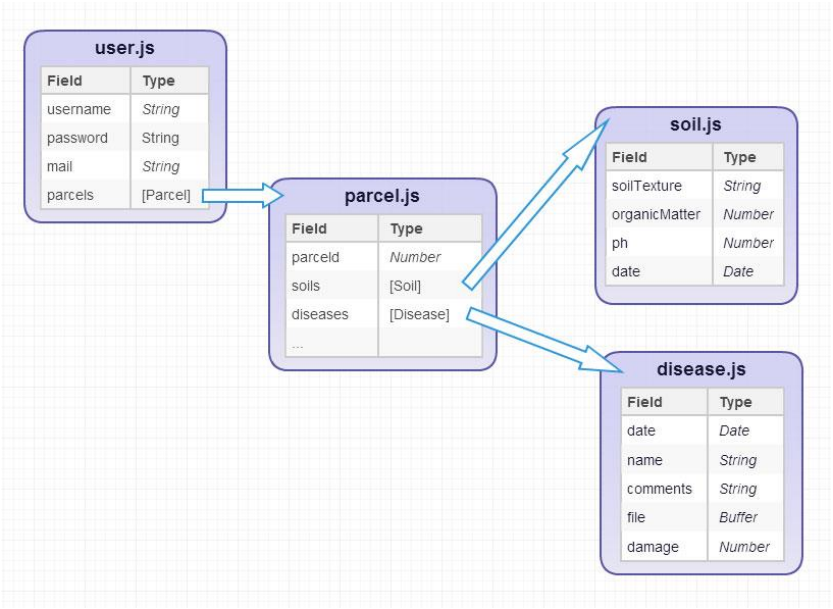

Figura 8. Resumen del modelo del Caso de Estudio.

De este modo en el servidor, mediante las funcionalidades de la librería MONGOOSE, es posible crear modelos y esquemas que almacenen la información según la figura anterior. La Figura 9 muestra cómo se genera el esquema del usuario.

```
var User = new mongoose.Schema({
    username: String,
    password: String,
    email: String,
    region: String,
    lastPosition:{
        lastX: Number,
        lastY: Number,
        zoom: Number,
        spatialReference: String
    },
    parcels: [ParcelSchema]
    });
```

Figura 9. Esquema del modelo del usuario.

Puede observarse como, para los campos sencillos se utiliza simplemente el tipo básico y para aquellos complejos, puede utilizarse como tipo otro esquema creado. En el ejemplo se ve como para el campo "*parcels*" se almacena un *array* de ParcelSchema. La Figura 10 muestra este esquema.

```
var Parcel = new mongoose.Schema({
    parcelId: String,
    soils:[SoilSchema],
    parcelStatus:[ParcelStatusSchema],
    cropInfos:[CropInfoSchema],
    yields: [YieldSchema],
    irrigationInfos: [IrrigationInfoSchema],
    fertilizers: [FertilizerSchema],
    agrochemicals: [AgrochemicalSchema],
    phenologies: [PhenologySchema],
    phatogens: [PathogenSchema],
    diseases: [DiseaseSchema],
    weeds: [WeedSchema],
    observations: [ObservationSchema]
});
```

Figura 10. Esquema del modelo de la parcela.

Nuevamente el esquema posee un elemento, "*parcelId*", que es un tipo básico, y después varios elementos complejos que poseen sus propios esquemas. Los esquemas y la forma de crearse, importarse y exportarse se pueden visualizar fácilmente en el código proporcionado. Aquí se van a mostrar dos de ellos, uno con información básica y otro con información binaria. En el caso de *cropInfo,* por ejemplo, se almacena un *array* de *CropInfoSchema.* Este esquema se muestra en la Figura 11. Para el caso de *disease* se almacena un *array* de *DiseaseSchema.* Este esquema se muestra en la Figura 12.

```
var CropInfo = new mongoose.Schema({
    cropType: String,
    riceVariety: String,
    pudding: String,
    showingPractice: String,
    date: Date
});
```

Figura 11. Esquema del modelo de la información de los cultivos (CropInfo).

```
var Disease = new mongoose.Schema({
    date: Date,
    name: String,
    comments: String,
    file: Buffer,
    damage: Number
});
```

Figura 12. Esquema del modelo de la enfermedad (Disease).

Con esta forma de estructurar la información en MongoDB y mediante la ayuda de MONGOOSE las búsquedas, modificaciones e inserciones se realizan de forma cómoda, siendo muy sencillo actualizar en la base de datos o mandar una respuesta en formato JSON, fácilmente manejable por las aplicaciones cliente (generalmente basadas en JavaScript en el proyecto ERMES).

4 REFLEXIONES

Esta sección pretende poner de manifiesto las experiencias adquiridas durante el desarrollo de la aplicación en cuenta al uso de Node.js, en el sentido de reflexiones sobre las ventajas y desventajas de la integración de datos con Node.js.

Uno de los principales beneficios del uso de Node.js es que está especialmente diseñado para el diseño, creación y despliegue de servicios web RESTful. Evidentemente, otras tecnologías y *frameworks* en el lado del servidor también soportan la creación de este tipo de servicios. Sin embargo, Node.js es minimalista, es decir, con poco líneas de código es posible armar la estructura de puntos de accesos (URL) para cada uno de los recursos de información que se quiere ofrecer en el lado del servidor (por ejemplo, usuarios, parcelas, etc.). Esta característica hace de Node.js una tecnología extremadamente eficaz para el desarrollo rápido o prototipado de aplicaciones web. Así mismo, la gran cantidad de módulos disponibles para Node.js, hace que muchas funcionalidades comunes en el desarrollo de aplicaciones web ya estén disponibles y listas para ser integradas en la aplicación final. No cabe duda que la comunidad de desarrolladores de Node.js es un factor importante para el existo de Node.js como plataforma de desarrollo en el servidor. Por ejemplo, la gestión de dependencias entre módulos, o el enrutamiento de peticiones al servicio correspondiente son tareas sencillísima cuando se importan los módulos correspondientes.

Otra característica que pone de manifiesto la accesibilidad de esta tecnología es la orientación a eventos que presenta. El control que le da al desarrollador ejecutar las cosas en el momento indicado representa un factor fundamental para garantizar la integridad y la estabilidad en la información. Con el uso de *threads* este control, aunque posible, se presenta mucho más complejo y es necesario añadir más actores como bloqueadores, "*mutex*", etc.

En los trabajos analizados se mencionaba la comodidad que presenta utilizar como gestor de dependencias el fichero *package.json*, esto ha sido comprobado en el caso de estudio dado que cada vez que se ha necesitado cambiar el equipo de desarrollo ha sido necesario reinstalar todos los módulos y para ello es tan sencillo como escribir "*npm install*" y todo se queda exactamente como estaba en el equipo original. Además, mantener el fichero es tan sencillo como añadir un *"—save*" cada vez que se añade alguna dependencia. Con esto es fácil asegurarse de que los paquetes se encuentren actualizados o en la misma versión deseada, etc. ahorrando problemas e incompatibilidades al ejecutarse en equipos diferentes.

Una de las dependencias que se planteaba como muy popular en el estado del arte era la librería *express*, en la experiencia adquirida se ha demostrado el porqué de su extenso uso. Si Node.js

presenta facilidades para realizar ciertas tareas, *express* lo simplifica todo muchísimo más pero sin dar la sensación de quitarle el control al desarrollador. Con sus funciones crear y desplegar aplicaciones web se convierte en algo tremendamente trivial incluso para programadores con poca experiencia en este tipo de tareas.

Algo que sí que se ha detectado en el marco de las competencias del master es lo poco desarrollada que está la tecnología para trabajar con bases de datos relacionales. En ese aspecto se ve que los módulos y librerías que facilitan su interacción todavía no están a la altura de otras tecnologías como por ejemplo *Ruby on Rails* [39].

También se experimentó, realizando algunas pruebas en el desarrollo, que cuando el servidor lleva una gran carga de computación (no ocurre así en el caso de estudio desarrollado) la tecnología orientada a eventos no presenta tan buen rendimiento como lo hace en otras tecnologías orientadas a *threads*.

A pesar de ser una tecnología minimalista, con multitud de módulos asociados, y especialmente indicada para el prototipado rápido de aplicaciones, Node.js no está pensado para cualquier tipo de aplicación web. Desde la experiencia del trabajo realizado se considera que las aplicaciones web que se basan en modelos de datos en JSON son idóneas para ser implementadas mediante Node.js. Por lo tanto, la combinación de un modelo de datos en JSON, junto con una base de datos basada en documentos JSON (MongoDB), hace que JavaScript sea la elección perfecta tanto en el lado cliente como en el servidor. Y aquí Node.js entra en escena para simplificar el desarrollo, gestión e integración de conjuntos de datos en JSON en el lado servidor.

Uno de los únicos aspectos en los que se ha echado de menos utilizar otro gestor de bases de datos para el caso de estudio ha sido a la hora de realizar ciertas búsquedas. Para los desarrolladores familiarizados con el lenguaje de consulta SQL, obtener fragmentos de información organizados y agrupados de la forma deseada es tan sencillo como encontrar la consulta adecuada. En el caso de MongoDB, incluso con la ayuda de la librería *Mongoose* esto se deja prácticamente todo a la parte de programación, lo que conlleva un poco de demora en el rendimiento y algo de complejidad en el código. No obstante todas las consultas necesarias se pueden llevar a cabo en unas pocas líneas de código por lo que el usuario mínimamente entrenado no tiene problemas con este aspecto.

Finalmente y como reflexión final cabe decir que, dada la "juventud" de la tecnología Node.js y su estado actual, es una opción principal en el desarrollo de la gran mayoría de aplicaciones Web. Cuando sus desarrollos se centren en aspectos relacionados con la integración de la información y se vayan puliendo ciertos aspectos, sin lugar a duda se convertirá en una herramienta que podrá

integrarse perfectamente con aplicaciones más complejas, con más costo computacional e incluso interactuando con diferentes gestores de bases de datos.

5 CONCLUSIONES

La integración de la información es un problema en constante evolución. Las técnicas utilizadas para encontrar soluciones concretas son tan variadas como los diferentes casos de uso donde se plantea la problemática. En este trabajo se ha planteado la solución desde la perspectiva de una interfaz RESTful ofrecida por un servidor responsable de unificar las diferentes aplicaciones existentes y futuras. Para ello se ha realizado en primer lugar una revisión de las diferentes tecnologías de servidor disponibles, comparando sus características principalmente en lo referente a rendimiento. También se ha revisado la información sobre las soluciones existentes en lo relativo a la persistencia de los datos. En ambas revisiones se ha hecho énfasis en las soluciones Node.js y MongoDB, tecnologías actuales y de uso creciente. Se ha desarrollado un caso de estudio y se han analizado los estudios previos con la experiencia adquirida en el desarrollo del mismo.

El alumno ha fortalecido las principales **competencias relativas al TII**. Con respecto a la **revisión** se ha trabajado con diferentes bases de datos, analizando, seleccionando y descartando los artículos que aportaban información nueva al tema de estudio. A continuación ha sido necesario realizar la **lectura** de los trabajos seleccionados, comprendiendo de este modo la forma de trabajar de sus autores y familiarizándose con las estructura de sus trabajos. Con respecto a la **difusión** se ha trabajado en la redacción de la presente memoria, siempre bajo la revisión y asesoramiento de los tutores, quienes han aportado su experiencia y conocimientos para obtener un mejor resultado. Finalmente se espera realizar una **presentación** audiovisual para mostrar el trabajo realizado frente al tribunal designado.

En relación al **caso de estudio** se ha trabajado con diferentes grupos de investigación para relacionar las diferentes aplicaciones clientes y ha sido necesario realizar un análisis de los diferentes modelos de datos unificados antes de tomar las decisiones referentes al modelo de datos global aplicado. Del mismo modo ha sido necesario experimentar con la tecnología Node.js para dar solución a todos los requisitos planteados. Los resultados obtenidos durante el tiempo que la aplicación ha estado en uso han sido satisfactorios, tanto en estabilidad y tiempos de respuesta como en los aspectos relacionados con la persistencia de los datos, la flexibilidad que ofrece MongoDB ha sido de gran ayuda a la hora de realizar modificaciones sobre el modelo de datos cuando las aplicaciones cliente lo han requerido.

El **proyecto** ERMES se ha visto beneficiado por la herramienta desarrollada dado que ahora mismo sus diferentes aplicaciones clientes poseen la información integrada y son capaces de aprovechar esta característica para usar diferentes herramientas integradas en el proyecto. Así mismo el

desarrollo de nuevas aplicaciones es capaz de integrarse directamente con los nuevos datos sin necesidad de realizar volcados previos o copias sobre bases de datos diferentes.

A pesar de los resultados satisfactorios que se han obtenido con el desarrollo del servidor, se espera mejorar su funcionalidad en un **futuro**. Con respecto a la seguridad, por ejemplo, la revisión realizada no ha sido demasiado exhaustiva, por lo que es algo que se pretende llevar a cabo próximamente para aplicar la información extraída en el desarrollo actual. Ocurre algo muy similar con la forma de trabajar con los datos multimedia, su implementación funciona pero no se ha realizado una revisión de cuál es la mejor forma de llevar a cabo su programación por lo que sería interesante profundizar en ello en un futuro.

Finalmente el estudiante desea **agradecer** a los tutores el trabajo realizado durante todo el proceso que ha constituido este trabajo, mostrando en todo momento facilidad de acceso y disponibilidad, conocimientos y permitido al alumno desarrollarse con libertad y cumpliendo los objetivos del trabajo.

BIBLIOGRAFÍA

[1]. Página Web del INIT: http://www.init.uji.es/

[2]. Página Web GEOTEC: http://www.geotec.uji.es/

[3]. Página Web del proyecto ERMES: http://www.ermes-fp7space.eu/

[4]. Página Web de Node.JS: https://nodejs.org/

[5]. Página Web de MongoDB: https://www.mongodb.org/

[6]. Base de datos Scopus: http://www.scopus.com/

[7]. Titchkosky, L., Arlitt, M., y Williamson, C. (2003). A performance comparison of dynamic Web technologies. ACM SIGMETRICS Performance Evaluation Review, 31(3), 2-11.

[8]. Trent, S., Tatsubori, M., Suzumura, T., Tozawa, A., y Onodera, T. (2008). Performance comparison of PHP and JSP as server-side scripting languages. En Middleware 2008 (pp. 164-182). Springer Berlin Heidelberg.

[9]. Ranjan, A., Kumar, R., y Dhar, J. (2012). A comparative study between dynamic web scripting languages. En Data Engineering and Management (pp. 288-295). Springer Berlin Heidelberg.

[10]. Sumedh, J. H., Huy, J. C., Mungee, S., y Schmidt, D. C. (1998). Techniques for Developing and Measuring High-Performance Web Servers over ATM Networks.

[11]. Ramana, U. V., y Prabhakar, T. V. (2005). Some experiments with the performance of LAMP architecture. En Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on (pp. 916-920). IEEE.

[12]. Warner, S., & Worley, J. (2008). SPECweb2005 in the Real World: Using Internet Information Server (IIS) and PHP. En 2008 SPEC Benchmark Workshop.

[13]. Neves, P., Paiva, N., & Durães, J. (2013). A comparison between JAVA and PHP. En Proceedings of the International C* Conference on Computer Science and Software Engineering (pp. 130-131). ACM.

[14]. Lei, K., Ma, Y., & Tan, Z. (2014, December). Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node. js. In Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on (pp. 661-668). IEEE.

[15]. Chaniotis, I. K., Kyriakou, K. I. D., & Tselikas, N. D. (2014). Is Node.js a viable option for building modern web applications? A performance evaluation study. Computing, 1-22.

[16]. Ojamaa, A., & Duuna, K. (2012). Assessing the security of Node.js platform. In Internet Technology And Secured Transactions, 2012 International Conference for (pp. 348-355). IEEE.

[17]. Yahoo Developer Network. (2010) Multi-Core HTTP Server with NodeJS. [Online]. Disponible en: http://developer.yahoo.com/blogs/ydn/posts/2010/07/multicore_http_server_with_nodejs/

[18]. Microsoft. Windows Azure Node.js Developer Center. [Online]. Available: http://www.windowsazure.com/en-us/develop/nodejs/

[19]. Tilkov, S., & Vinoski, S. (2010). Node. js: Using JavaScript to build high-performance network programs. IEEE Internet Computing, (6), 80-83.

[20]. NPM Package Manager. [Online]. Extraído de: https://www.npmjs.com/

[21]. Express Framework. [Online]. Extraído de http://expressjs.com/es/

[22]. Passport Middleware. [Online]. Extraido de http://passportjs.org/

[23]. OAuth authorization Framework. [Online]. Extraido de http://oauth.net/

[24]. OpenID foundation. [Online]. Extraido de http://openid.net/

[25]. Mongoose Middleware. [Online]. Extraido de http://mongoosejs.com/

[26]. Multer Middleware. [Online]. Extraido de https://github.com/expressjs/multer

[27]. Bcrypt Library. [Online]. Extraído de https://www.npmjs.com/package/bcrypt-nodejs

[28]. Consolidate Template Engine. [Online]. Extraído de https://www.npmjs.com/package/consolidate

[29]. Elmasri, R. and Navathe, S. (2011). Fundamentals of Database Systems, Addison-Wesley.

[30]. Parker, Z., Poe, S., y Vrbsky, S. V. (2013). Comparing nosql mongodb to an sql db. In Proceedings of the 51st ACM Southeast Conference (p. 5). ACM.

[31]. Apache Cassandra database. [Online]. Extraído de http://cassandra.apache.org/

[32]. Abramova, V., & Bernardino, J. (2013). NoSQL databases: MongoDB vs cassandra. In Proceedings of the International C* Conference on Computer Science and Software Engineering (pp. 14-22). ACM.

[33]. JavaScript Object Notation. [Online]. Extraído de http://json.org/

[34]. Ameri, P., Grabowski, U., Meyer, J., & Streit, A. (2014). On the Application and Performance of MongoDB for Climate Satellite Data. In Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on (pp. 652-659). IEEE.

[35]. Xinwei, J., & Guicheng, S. (2014). Managing RFID data in MongoDB. In Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on (pp. 206-209). IEEE.

[36]. Xu, W., Zhou, Z., Zhou, H., Zhang, W., & Xie, J. (2014). MongoDB Improves Big Data Analysis Performance on Electric Health Record System. In Life System Modeling and Simulation (pp. 350-357). Springer Berlin Heidelberg.

[37]. Same-Origin Policy. [Online]. Extraído de https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[38]. Librería FS de Node.js. [Online]. Extraído de https://nodejs.org/api/fs.html

[39]. Framework Ruby on Rails. [Online]. Extraído de http://rubyonrails.org/