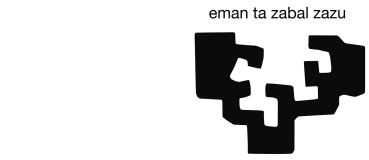


Universidad del País Vasco
Euskal Herriko Unibertsitatea

Grado en Ingeniería Informática

Ingeniería de Computadores

Proyecto de Fin de Grado

Desarrollo y pruebas en entorno real de un smart reader Wi-Fi

Autor

Urtzi Diaz Arberas

Noviembre 2016

Resumen

En el proyecto documentado en esta memoria, se ha desarrollado un sistema que permite comunicar datos entre un servidor y un lector de tarjetas, mediante un smart-reader Wi-Fi. Conjuntamente, podemos encontrar un estudio y unas pruebas de rendimiento sobre las diferentes alternativas para servir a dicho Smart reader en un entorno real.

Realizado para la titulación de Ingeniería Informática, en la facultad de Informática de la universidad UPV-EHU de Donostia/San Sebastián.

Palabras clave: Applets, Smart-cards, ESP8266, Servicios web, SOAP, REST, API, PHP, Codeigniter, Node.JS y Web server benchmarking.

Agradecimientos

En primer lugar quisiera agradecer a David Ruiz la oportunidad que me ha brindado para realizar este proyecto y aprender de él. Agradezco mucho a la empresa ATELEI por darme la oportunidad de realizar este proyecto con ellos, hacerme sentir como si fuese uno más de la plantilla desde el primer día y por todos los buenos momentos vividos durante estos más de 8 meses.

A mis padres por los valores que me han inculcado, los cuales son responsables del esfuerzo y dedicación constante que se requieren para poder superar retos como el que estoy concluyendo. Y por supuesto al resto de mi familia, mis abuelas, mis tías y tíos, primos... por vuestra confianza en mí, y sobre todo, a mi padre por aguantar todos los momentos de frustración que he pagado con él.

A todos mis profesores, por todo lo que he aprendido gracias a vosotros, y en especial a mi tutor Iñaki Alegria por guiarme durante todo el proceso, especialmente a la hora de preparar esta memoria.

A todos mis compañeros y amigos de la universidad, sobre todo a Itziar y Jon, porque sin todos vosotros, vuestros resúmenes, vuestros consejos, y vuestra ayuda seguro que no estaba escribiendo estas líneas.

A la kuadrilla de Gasteiz y a todos los demás que siempre estáis ahí. A Rafa, porque tu apoyo y paciencia durante toda la carrera han sido muy importantes para mí.

Por último, quiero agradecer a todos los trabajadores, o mejor dicho, a todo el equipo de On4u el trato recibido durante estos últimos meses. Gracias por dejarme disfrutar la filosofía On4u y aprender día a día de los mejores.

Eskerrik asko guztioi.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

Este documento es la memoria del proyecto de fin de grado de Ingeniería informática de la Universidad del País Vasco (UPV-EHU).

En él se explican y describen todos los detalles de dicho proyecto. Más adelante, en este mismo capítulo se describe la estructura de esta memoria. El proyecto se ha llevado a cabo en la empresa ATELEI Engineering SLU, con sede en el polígono Arretxe-Ugalde (Irún). Dirigido por el profesor Iñaki Alegria y bajo la supervisión de David Ruiz e Iván Piquer, además de con la colaboración de Iñigo Olaso y Andoni Sánchez.

Introducción
1. Introducción 2. Estructura del trabajo

1.1. Introducción

El 23 de Mayo de 1995 Java salió a la luz de forma pública, durante la conferencia SunWorld la compañía Sun Microsystems presentó el lenguaje en el que había estado trabajando durante más de cinco años. Un lenguaje moderno concebido para funcionar casi en cualquier dispositivo.

Para acompañar el anuncio del nuevo lenguaje, el pelotazo fue que Java se integraría con el navegador Netscape, con lo que se pudo observar el nacimiento de los Applets[1] de Java. La web empezaba a asomar su potencial e hizo virar el rumbo de los lenguajes de programación. Los Applets eran "la moda" y muchas páginas web te exigían instalarlo para ejecutar ciertas funcionalidades, hasta entonces imposibles de otro modo.

Durante años los distintos navegadores para plataformas de escritorio han convivido con los plugins de Java que Oracle suministraba. Pero, desde hace ya tiempo, se vienen sufriendo ciertos problemas con Java que están poniendo cada día más difícil considerar los applets como una solución de futuro. Desde la falta de soporte en las nuevas plataformas operativas (Apple iOS, Google Android, Microsoft Windows Phone, BlackBerry 10, etc...) hasta los grandes problemas de seguridad que plantean, pasando por las dificultades que los navegadores ponen a su ejecución, como advertencias y diálogos de confirmación. Podríamos resumir diciendo que su fin era algo anunciado, pero no se sabía cuándo llegaría ese momento, sin embargo, Oracle ha anunciado que los Applets de Java ya no estarán presentes en la JDK (Java Development Kit) que se lanzará en 2017 con Java 9.

La empresa ATELEI utiliza una aplicación del tipo de las mencionadas anteriormente dentro de su sistema de control de accesos. La funcionalidad de esta es programar o leer tarjetas smart card mediante estos applets y un dispositivo card-reader conectado al equipo.

La sustitución de los Applets por otra tecnología no es tarea fácil, ya que, por una parte, se necesita una comunicación bidireccional con el JavaScript de la página Web y, por otra, un acceso a los almacenes de claves y certificados. En este sentido, son varias las opciones posibles, por ejemplo, el desarrollo de extensiones para navegadores web, la implantación de un API JavaScript o el uso de aplicaciones nativas con invocación por protocolo.

[1] Los Applets son aplicaciones Java que se ejecutan dentro de un navegador. Pueden ser muy útiles para complementar la funcionalidad de nuestras aplicaciones Web, ya que permiten hacer cosas que con JavaScript son muy complicadas o incluso imposibles

Después de un exhaustivo estudio de la mano del compañero Andoni Sanchez y la empresa ATELEI, se decidió añadir un componente hardware al sistema, un componente creado por la empresa Espressif, el módulo ESP8266. Este componente puede intercambiar información con un servidor, facilitándonos las comunicaciones que necesitamos sin tener que hacer uso de los Applets de java.

En los siguientes capítulos encontraremos más información sobre dicha comunicación, objetivos, resultados, etc...

1.2. Estructura del trabajo

La estructura de capítulos se describe a continuación:

- **Antecedentes y objetivos del proyecto:** En este capítulo podemos encontrar los antecedentes del proyecto, las características de la empresa y su especialización, así como las necesidades, soluciones y mi motivación personal frente a este reto. A continuación, se describen los objetivos del proyecto, se presenta el alcance, la arquitectura a construir, las exclusiones del proyecto y los recursos a utilizar. Finalmente, se hará una reflexión sobre las diferentes herramientas y tecnologías disponibles a la hora de realizar el proyecto.

- **Servicios web y API REST:** En la primera parte de este capitulo podemos encontrarnos con una pequeña introducción sobre los servicios web. Más tarde, se hará una reflexión sobre los diferentes estilos, a continuación analizaré las posibles alternativas escogidas para llevar acabo el desarrollo del proyecto. En la segunda parte nos centraremos en conocer los sistemas REST. El inicio versará sobre los conceptos básicos y las reglas que definen esta arquitectura. A continuación hablaré sobre cuales son los pasos adecuados para la construcción de una API de estas características. Por último, expondré los lenguajes de programación (y sus frameworks) más frecuentes para la creación de una API REST.

- **Análisis y comparativa de las alternativas escogidas:** En este capítulo se expondrán las dos alternativas (PHP y Node.JS) para la construcción de un sistema que permita interactuar un servidor con un dispositivo, en mi caso un lector de tarjetas. Primero, se analizaran los puntos fuertes y débiles de cada una de las alternativas para más adelante comparar el flujo de instrucciones. Por ultimo veremos un ejem-

plo sencillo de benchmarking, debido a que en el capítulo 9 llevare a cabo el análisis en profundidad con los sistemas desarrollados.

- **Análisis de los requisitos:** En este capítulo se tratará detalladamente la arquitectura del proyecto. En primer lugar se dará una explicación de las características que ha de tener, a continuación se explicará la arquitectura diseñada, se presentaran los diagramas de flujo, se detallarán los casos de uso, se definirán los diferentes tipos de acciones a tratar y por ultimo definiré el intercambio mensajes y la consistencia de la base de datos.

- **Diseño y desarrollo en Node.JS:** En este apartado se detallarán los temas concernientes al diseño del sistema con Node.JS. Por un lado, se hará hincapié en la arquitectura definida más allá de la división por capas. Más adelante, veremos cuál ha sido la elección del motor de plantillas para generar la parte front-end de este proyecto. Por otro lado, se hablará del desarrollo del proyecto y las decisiones que se han ido tomando a medida que iba avanzando el proceso. Se hará una pequeña introducción sobre los módulos de Node.JS, para más adelante ir viendo que modulo conviene para los diferentes funcionamientos. Finalmente se describirán las diferentes implementaciones.

- **Diseño y desarrollo en PHP:** En este apartado se detallarán los temas concernientes al diseño del sistema con PHP, más concretamente con el framework codeigniter. Se hará hincapié en la arquitectura definida más allá de la división por capas. Por otro lado, se hablará del desarrollo del proyecto y las decisiones que se han ido tomando a medida que iba avanzando el proyecto con PHP y codeigniter. Finalmente se describirán las diferentes implementaciones.

- **Estudio de rendimiento y simulación:** En la primera parte de este capítulo abordaremos las pruebas realizadas para comprobar el correcto funcionamiento del servidor. En la segunda parte de esta sección el problema a tratar será la respuesta del servidor web ante un benchmarking, es decir, se medirá el rendimiento de la aplicació n, y compararlo entre los dos sistemas desarrollados. Primero, se presentara la herramienta con la que poder llevar acabo las pruebas, más adelante se hará un breve resumen de las diferentes pruebas y el capítulo concluirá con unos gráficos donde podemos ver comparados los resultados de las mencionadas pruebas.

- **Conclusiones:** En este capítulo atenderemos a las conclusiones obtenidas una vez terminado el proyecto, en primer lugar se hará una valoración del trabajo realizado.

En segundo lugar se hará una reflexión sobre las lecciones aprendidas y por último se darán unas posibles líneas de trabajo futuras.

- **Bibliografia**
- **Anexos**

2. CAPÍTULO

Antecedentes y objetivos del proyecto

En este capítulo podemos encontrar los antecedentes del proyecto, las características de la empresa y su especialización, así como las necesidades, soluciones y mi motivación personal frente a este reto.

A continuación, se describen los objetivos del proyecto, se presenta el alcance, la arquitectura a construir, las exclusiones del proyecto y los recursos a utilizar.

Finalmente, se hará una reflexión sobre las diferentes herramientas y tecnologías disponibles a la hora de realizar el proyecto.

Antecedentes del proyecto
1. Características de la empresa 2. Necesidades de la empresa 3. Solución 4. Motivación personal 5. Alcance 6. Exclusiones 7. Herramientas y tecnologias

2.1. Características de la empresa

ATELEI Engineering SLU, con sede en el polígono Arretxe-Ugalde de Irún, es una empresa fundada por el CEO Iván Piquer en el año 2012. Está especializada en sistemas de control de acceso y monitorización remota, definiéndose como *"diseñadores de productos, especializados en el diseño y desarrollo de productos electrónicos desde su conceptualización hasta su fabricación"*.

ATELEI, es una empresa de ingeniería multidisciplinar que pone a disposición de sus clientes su experiencia dentro de los campos de la electrónica, la mecatrónica, la mecánica y el software para crear productos electrónicos y sistemas innovadores. En los siguientes apartados podremos ver las áreas con las que más trabajan.

Figura 2.1: Logo de la empresa ATELEI Engineering SLU.

2.1.1. Diseño de producto electrónico

Son especialistas en el diseño electrónico de PCBs de una, dos o más capas. Cuentan con experiencia en diseño y desarrollo de producto para Marcado CE y superación de pruebas de compatibilidad electromagnética.

Expertos en electrónica Wireless: Presentan una amplia experiencia en el diseño electrónico de productos wireless. Han trabajado satisfactoriamente con tecnologías como: LoRa, Sigfox, Miwi, WIFI, BLE y ANT.

Identificación por radiofrecuencia: Han diseñado productos electrónicos que integran tecnologías de identificación por radiofrecuencia (RFID).

Expertos en "internet of things": Durante los últimos años han desarrollado productos electrónicos de sensoria, englobados en la denominada "Internet of Things (IoT)".

Son especialistas en la creación y despliegue de redes de sensores que se comunican entre sí de forma inalámbrica para monitorizar diversos parámetros.

2.1.2. Diseño mecatrónico

Llevan a cabo una combinación sinérgica de ingeniería electrónica, mecánica, computación y sistemas informáticos para diseñar sistemas inteligentes. Integran y desarrollan sistemas automatizados y/o autónomos que involucran tecnologías de varios campos de la ingeniería.

2.1.3. Desarrollo software

Ofrecen a los clientes la experiencia de los ingenieros informáticos, formados en las últimas tecnologías software y con amplia experiencia en: SIG, SaaS, PaaS e IaaS.

Expertos en sitemas cloud: Desarrollar un sistema SaaS, no es solamente cuestión de escribir una aplicación web. La escalabilidad, la robustez y la gestión de volúmenes de datos adquieren un papel relevante en el diseño. En ATELEI, analizan el proyecto para crear un sistema que crezca con los clientes.

Sistemas de información geográfica: Cuentan con expertos en SIG que han desarrollado plataformas usando bases de datos geoespaciales. Ellos son capaces de crear aplicaciones escalables en las que el número de nodos no afecta al rendimiento del sistema.

Sensorica / IOT: Cuentan con experiencia en el tratamiento de información proveniente de sensores implantados en diversos sectores de cara a realizar un análisis de los datos recibidos. Ayudando a crear sistemas de gestión preventiva en múltiples escenarios.

2.1.4. Servicios de prototipado

Ponen a disposición de sus clientes, las últimas tecnologías de prototipado. Desde impresoras 3D hasta máquinas pick and place para el montaje de circuitos electrónicos. El personal de la empresa, tiene amplia experiencia en modelado 3D para diversos sectores, desde el aeronáutico hasta el industrial.

2.1.5. Sistemas wireless de control y acceso

Los dispositivos fabricados por ATELEI, forman una red inalámbrica invisible, que permite que éstos se comuniquen entre sí mediante mensajes. De esta manera es posible configurar la red para que, ante ciertos eventos en un dispositivo, cualquiera que se encuentre en la misma red, esté o no alejado de él, pueda llevar a cabo acciones que vayan desde la apertura de una puerta o barrera de parking hasta el encendido de la iluminación de una estancia.

Cerraduras Electrónicas: En ATELEI prestan especial atención a los pequeños detalles. Esos pequeños detalles les diferencian del resto de competidores. Por este motivo, han creado una cerradura electrónica en la que el cliente elige el diseño. Tanto si es una empresa que desea usar su imagen corporativa, como un particular con grandes dotes como artista.

Lectores Murales: Dan al cliente la posibilidad de elegir el diseño de sus mecanismos. Gracias a los lectores fabricados por ATELEI, sus clientes pueden elegir entre una amplia gama de colores que permiten al sistema de control y acceso integrarse a la perfección dentro de la estética de los edificios. Todo ello además mejorado con los últimos avances tecnológicos de un sistema wireless innovador.

Controladores: Un sistema de control, tiene que permitir integrar productos de terceros, de forma que estos se integren en la instalación como un elemento más. Con este objetivo en mente, en ATELEI han diseñado un grupo de dispositivos que les permiten llevar a cabo esta integración. De esta manera, gestionar el encendido de la calefacción o activar el aire acondicionado se convierte en una tarea trivial que puede ser llevada a cabo desde cualquier lugar del planeta.

2.2. Necesidades de la empresa

En el capítulo anterior he mencionado como la empresa ATELEI hacia uso de los famosos applets de Java para que su aplicación de control de accesos se comunicase con el lector de tarjetas instalado en el ordenador del usuario. Estos applets hacen uso de librerías de Java por lo que se requería tener instalada la JVM (Java Virtual Machine) y asignarle una serie de permisos. Mediante el Applet se conseguía realizar acciones de lectura y escritura sobre tarjetas smart card desde un entorno web, resultando ser una aplicación multiplataforma accesible desde cualquier ordenador.

Pero ahora, ATELEI necesita un nuevo sistema ya que los applets de java están a punto de extinguirse. En ese punto es donde nos encontramos con este proyecto.

2.3. Solución

Con motivo de dar solución al problema previamente mencionado, nace la finalidad de este proyecto: *crear desde cero un sistema que pueda llevar a cabo acciones de lectura y escritura sobre tarjetas sin hacer usos de los applets de java*.

Desarrollar un sistema que tenga en cuenta las necesidades de todas las empresas es una tarea casi imposible. De esta manera, la solución acordada con la empresa ATELEI ha sido utilizar el módulo ESP8266 y gracias a él en los siguientes capítulos vamos a descubrir una nueva alternativa para poder seguir satisfaciendo las necesidades de la empresa. Por lo cual, podríamos resumir diciendo que los objetivos de este proyecto son analizar una serie de tecnologías emergentes en el ámbito del tratamiento de información en entornos distribuidos y escalables en el lado del servidor y el de desarrollar un sistema eficiente para un comunicar datos entre un lector de tarjetas y un servidor.

2.4. Motivación personal

A la hora de elegir el proyecto de fin de grado ésta fue mi primera opción, el hecho de poder trabajar dentro del departamento I+D de ATELEI me pareció una gran oportunidad para ir tomando contacto con el mundo empresarial, además de poder desarrollar sistemas para problemas reales. Además, el hecho de desarrollar un proyecto basado en tecnologías emergentes me pareció un reto muy interesante para poder así testar mis capacidades y perder el miedo a trabajar en algo desconocido.

2.5. Alcance del proyecto

Este apartado me centraré en el alcance del proyecto. A la hora de definirlo hare hincapié en primer lugar en los requerimientos del proyecto, a continuación se detallarán los entregables, y por último se hará una descomposición del trabajo a realizar en un diagrama EDT (Estructura de Descomposición de Trabajo).

2.5.1. Requerimientos

El objetivo principal de este proyecto es desarrollar un sistema que permita comunicarnos con un lector de tarjetas (encoder). Para ello, utilizando un módulo ESP8266, ATELEI ha desarrollado un lector de tarjetas propio que se conecta vía WI-FI a un servidor e intercambia información.

De esta manera, las empresas no tendrán que invertir en desarrollar un sistema propio y específico para la gestión de tarjetas electrónicas. También, mediante este proyecto veremos cuáles son las mejores alternativas para construir el sistema de comunicaciones con el servidor.

El proyecto se puede dividir en tres fases, la primera fase está dedicada al estudio de las diferentes tecnologías para la construcción de un servidor que intercambie información con el Smart-reader Wi-Fi, la segunda fase al desarrollo de las diferentes alternativas escogidas, y la última fase al estudio de rendimiento y simulación de las alternativas escogidas anteriormente. De este modo podemos separar los objetivos para cada una de las distintas fases, a pesar de la concordancia entre ellas en el aspecto del desarrollo.

El estudio deberá contar con las siguientes características:

- Introducción a los servicios web.
- Análisis y comparación entre los servicios REST y SOAP.
- Análisis de PHP.
- Análisis de Node.JS.
- Gestión clientes de las diferentes alternativas.
- Comparativa del nivel de carga de las diferentes alternativas.
- Ventajas y desventajas de las diferentes alternativas.

Los sistemas desarrollados deberán contar con las siguientes características:

- Desarrollo en Node.JS prototipo funcional.
- Desarrollo en PHP prototipo funcional.

- Comunicación mediante HTTPS.
- Un servidor que se rige por el protocolo descrito para la realización de las comunicaciones entre cliente-servidor.
- El servidor recibirá peticiones del módulo, peticiones TCP-IP mediante SSL.
- El servidor recibirá/envía acciones en formato JSON.
- El sistema ha de ofrecer facilidades para construir una arquitectura escalable.
- Facilidad de instalación e implantación del sistema.
- El sistema ofrecerá la posibilidad de visualizar información del servidor.
- Sesiones basadas en JWT.
- Diferentes pruebas para comprobar el funcionamiento del lado del servidor.

El estudio de rendimiento deberá contar con las siguientes características:

- Simulación con diferentes tipos de peticiones.
- Estudio y comparativa de los resultados.
- Valoración de los resultados.

Arquitectura del sistema

- Un módulo ESP8266 con un encoder o transceiver que interacciona con las tarjetas.
- Una estación Wi-Fi que dispone de conexión a internet.
- Un dispositivo que dispone de conexión a internet y de un navegador web.
- Un servidor en la nube que interpreta y se comunica a través del protocolo de comunicación descrito en los siguientes capítulos.

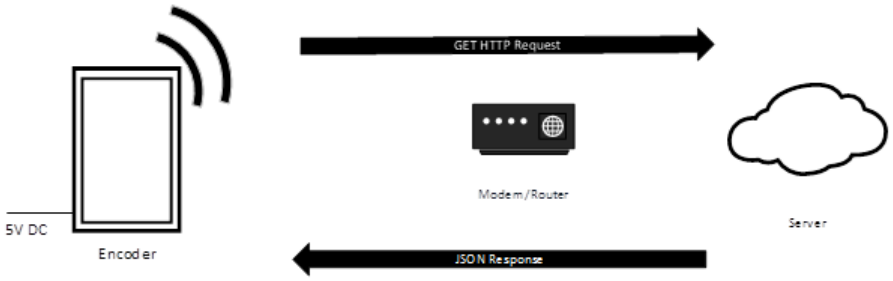


Figura 2.2: Arquitectura del sistema 01

2.5.2. Entregables

En este proyecto hay dos entregables principales, por un lado el proyecto en sí y el código, que será entregado a la empresa ATELEI que tendrá todos los derechos sobre éste y podrá usarlo según sus intereses.

Y por otro lado estará esta memoria, que seguirá los estándares y que recogerá la información sobre el trabajo realizado. En esta memoria no se enseñara el código completo por motivos de confidencialidad. La memoria está protegida por las leyes de la propiedad intelectual, por lo tanto pertenece al autor.

La memoria del proyecto será entregada antes del día 2 de noviembre en la plataforma ADDI de la Universidad del País Vasco, previamente, una copia de dicha memoria será entregada al director del proyecto para que pueda revisarla y darle su aprobación.

2.5.3. E.D.T.

En la imagen 2.3 se puede ver la descomposicón de las diferentes partes del trabajo, a continuación se explica el significado de cada una:

- **PFG** - Proyecto de fin de grado, raíz del diagrama y objetivo final.
  - **Administración del proyecto** - Desarrollo del plan general del proyecto necesario para garantizar su consecución.
    - **Definición de los requisitos** - Evaluación y definición de los requisitos necesarios para llevar a cabo el proyecto.
  - **Inicio** - Formación y aprendizaje de los conocimientos necesarios para llevar a cabo el proyecto.

- **Toma de contacto** - Primeros pasos en la empresa, lectura de difentes manuales y obtencion de los nuevos conocimientos.
- **Formación** - Formación en profundidad de los diferentes temas que se tratarán en el proyecto.

- **Desarrollo del proyecto** - Puesta en marcha de los objetivos del proyecto.
  - **Estudio de alternativas** - Estudio sobre las tecnologías a implementar en el proyecto.
  - **Desarrollo** - Formación en profundidad de los diferentes temas que se tratarán en el proyecto.
    - **Especificacion** -
    - **Diseño** - Planificación de los diferentes componentes del proyecto, su arquitectura e interfazes.
    - **Implementación** - Implementación de lo diseñado previamente.
    - **Pruebas** - Obtención de los datos resultados del desarrol y comparacion con los esperados.
  - **Simulación y rendimiento** – Estudio, comparación y valoración de los resultados obtenidos.
    - **Pruebas** - Definicion del tipo de pruebas.
    - **Estudio** - Analisis de las pruebas anteriores.

- **Universidad** - Fase final del proyecto.
  - **Documentación** - Redacción de la memoria del proyecto y demás documentos.
  - **Defensa** - Exposición pública del proyecto con una presentación.

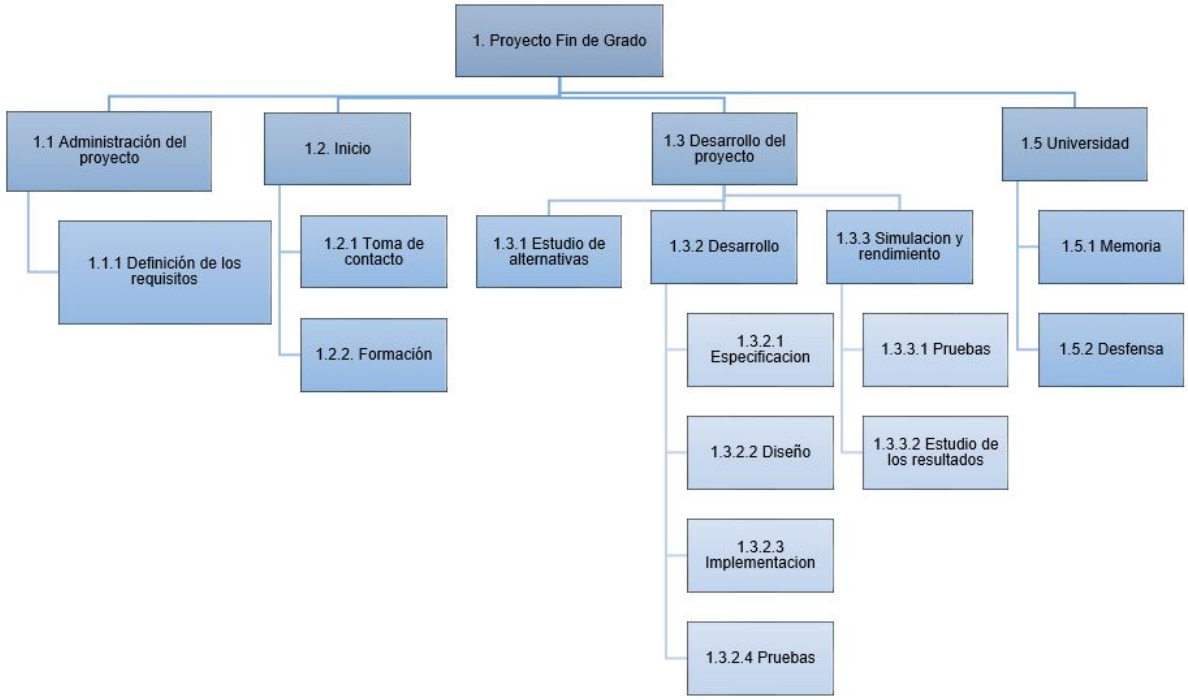


Figura 2.3: EDT - Diagrama de la estructura de descomposción de trabajo.

2.6. Exclusiones del proyecto

En esta memoria no se enseñara el código completo, ni algunos apartados por motivos de confidencialidad. A pesar de esto, el día de la defensa se podrán conocer más detalles sobre los apartados confidenciales.

- El estudio sobre las diferentes alternativas para solucionar el problema de los Applets.
- El funcionamiento interno del módulo ESP8266.
- El esquema del hardware utilizado.
- Se excluye toda documentación relativa a la configuración y gestión del lector que brinda las peticiones.
- De la misma forma, se excluye la documentación y desarrollo realizado del encoder o transceiver encargado de la lectura de tarjetas.

- Además se excluyen del alcance las pruebas unitarias de cada módulo o método de código como las pruebas de integración de la unión de cada uno de los métodos.
- Manual de instalación de los diferentes APIs.
- Manual sobre JMeter.

2.7. Herramientas y tecnologías

En este apartado se especificarán los recursos utilizados durante el proyecto. En la siguente lista podemos encontrar los principales componentes del proyecto. Pero, voy a necesitar alguna herramienta adicional para llevar aacabo el poryecto de una forma mas eficiennte. La elección de dichas herramientas ha quedado en gran parte en mis manos, por lo cual más adelante se hace una descripción más detallada de las elecciones.

- El lector de tarjetas (módulo ESP8266).
- Tarjetas electrónicas.
- Portátil personal Asus F555L.
- Servidor linux.

Como se ha explicado previamente, el objetivo principal del proyecto es desarrollar un sistema que sirva a un lector de tarjetas. Por lo cual, tendremos 3 principales actuadores; el servidor, el hardware y los usuarios.

2.7.1. Software

Como se ha mencionado previamente, gran parte de las decisiones han recaído en mis manos, por lo cual, dependiendo las necesidades he escogido la herramienta adecuada. En la siguientes tablas se pueden observar las herramientas más importantes seleccionadas para llevar adelante este proyecto.

Necesidad	Tecnología/Herramienta
Sistema Operativo	Linux (Ubuntu 14.04)
Entorno desarrollo PHP	PHPStorm
Entorno desarrollo Node.JS	WebStorm
Base de datos	MYSQL 5.0

Tabla 2.1: Tabla de las herramientas y tecnologías escogidas (Software).

2.7.2. Hardware

Además del servidor, el otro actuador principal en el proyecto es el lector de tarjetas. Este aparato nos ayuda a capturar información de una tarjeta, en ATELEI hacen uso de la tarjetas MIFARE, tarjetas inteligentes sin contacto.

MIFARE es una tecnología de tarjetas inteligentes sin contacto (TISC), de las más ampliamente instaladas en el mundo, con aproximadamente 250 millones de TISC y 1,5 millones de módulos lectores vendidos. Estas tarjetas de memoria están protegidas, divididas en sectores, bloques y mecanismos simples de seguridad para el control de acceso. Su uso está orientado a monederos electrónicos simples, control de acceso, identidad corporativa o como tarjeta de trasporte urbano.

El smart-reader o encoder es un dispositivo de detección que proporciona una respuesta. Los encoders convierten el movimiento en una señal eléctrica que puede ser leída por algún tipo de dispositivo de control en información. Además del mencionado encoder o transceiver que interacciona con las tarjetas habrá un módulo ESP8266 para la conectividad.

El módulo ESP8266 es un microcontrolador que incorpora procesador y conectividad WIFI integrada en un único y pequeño componente. Esta pequeña placa permite conectar microcontroladores a una red Wi-Fi y realizar comunicaciones mediante el protocolo TCP/IP.

Necesidad	Tecnología/Herramienta
Estación de trabajo	Portátil personal Asus F555L
Tarjetas	MIFARE
Lector de tarjetas	El smart-reader o encoder
Conectividad wifi en el lector	El módulo ESP8266

Tabla 2.2: Tabla de las herramientas y tecnologías escogidas (Hardware).

2.7.3. Integración y ayuda

Para poder llevar a cabo el proyecto se han hecho uso de otras herramientas fuera del ámbito del desarrollo.

Necesidad	Tecnología/Herramienta
Comunicación	Gmail y Slack
Gestión de versiones	GIT (bitbucket)
IDE para redactar la memoria	TeXstudio
Gráficos y tablas	Microsoft office

Tabla 2.3: Tabla de las herramientas y tecnologías escogidas (Integración y ayuda).

3. CAPÍTULO

Servicios web y API REST

En la primera parte de este capitulo podemos encontrarnos con una pequeña introducción sobre los servicios web. Más tarde, se hará una reflexión sobre los diferentes estilos, a continuación analizaré las posibles alternativas escogidas para llevar acabo el desarrollo del proyecto. En la segunda parte nos centraremos en conocer los sistemas REST. El inicio versará sobre los conceptos básicos y las reglas que definen esta arquitectura. A continuación hablaré sobre cuales son los pasos adecuados para la construcción de una API de estas características. Por último, expondré los lenguajes de programación (y sus frameworks) más frecuentes para la creación de una API REST.

Servicios web y API REST

1. Servicios web
2. REST vs SOAP
3. Reglas de la arquitectura REST
4. Metodología REST
5. Lenguajes para la construcción de un API REST
6. Decisión

3.1. Servicios web

El consorcio W3C define los Servicios Web como sistemas software diseñados para soportar una interacción interoperable máquina a máquina sobre una red. Estos servicios proporcionan mecanismos de comunicación estándares entre diferentes aplicaciones, que interactúan entre sí para presentar diferente información dinámica al usuario. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores.

La definición de Servicios Web propuesta alberga muchos tipos diferentes de sistemas, pero el caso de uso más común se refiere a clientes y servidores que se comunican mediante mensajes XML siguiendo el estándar SOAP.

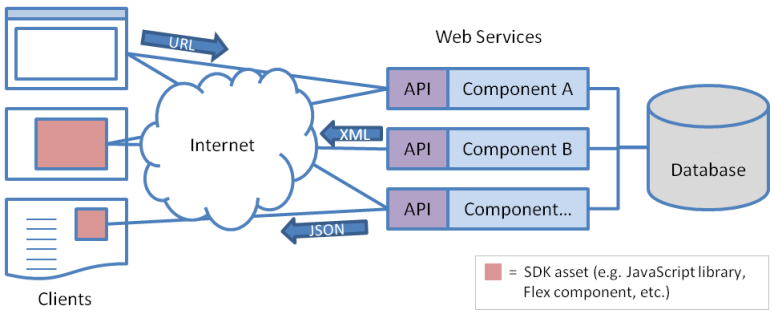


Figura 3.1: Diagrama de la estructura de un servicio web basado en SOAP.

3.1.1. Estilos de WS

Los servicios web pueden funcionar con diferentes estilos, en la siguente lista veremos los más estandarizados a nivel mundial:

SOAP - Simple Object Access Protocol

- Protocolo escrito en XML para el intercambio de información entre aplicaciones.
- Es un formato para enviar mensajes, diseñado especialmente para servir comunicación en Internet. Básicamente es un protocolo para acceder a un servicio web.
- Define qué información se envía y cómo se envía mediante XML.

WSDL - Web Services Description Language

- WSDL es un lenguaje basado en XML
- Es el formato estándar que describe servicios web y cómo acceder a ellos.
- Es una parte integral del estándar UDDI, y es el lenguaje que éste utiliza.

UDDI - Universal Description, Discovery and Integration

- Estándar XML para describir, publicar y encontrar servicios web.
- Es un directorio de interfaces de servicios web descritos en WSDL que se comunican mediante SOAP donde las compañías pueden registrar y buscar servicios web.

REST

Arquitectura que, haciendo uso del protocolo HTTP, proporciona una API que utiliza cada uno de sus métodos (GET, POST, PUT, DELETE, etc...) para poder realizar diferentes operaciones entre la aplicación que ofrece el servicio web y el cliente.

3.2. API REST vs SOAP

Recordemos que nuestro sistema va hacer uso de llamadas del tipo un usuario mediante interfaz envía a servidor una petición, el servidor le envía petición al lector de tarjetas y el lector de tarjetas lee información de la tarjeta y envía datos a servidor. Teniendo en cuenta ese diseño podemos reducir nuestras posibilidades de arquitectura a dos estilos de API, REST o SOAP. Como vimos en el apartado anterior estos dos diferentes estilos están asociados a estándares para el diseño y desarrollo de servicios web.

Antes de nada cabe mencionar que una API, es el conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. En otras palabras, una API es una especificación formal sobre cómo un módulo de un software se comunica o interactúa con otro. Una diferencia que se pude percibir entre una web y una API, es que mientras la primera es diseñada e implementada para ser consumida y usada por usuarios finales, una API es creada para ser usada por desarrolladores y consumida por aplicaciones.

Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores nos beneficiamos de las ventajas de la API haciendo uso de su funcionalidad, evitando el trabajo de programar todo desde el principio. Cabe destacar que en este proyecto me ayudare de estas APIs para construir el sistema que dará servicio al lector de tarjetas.

Como ya he mencionado anteriormente aunque REST y SOAP son arquitecturas muy similares de intercambio de información entre aplicaciones web con servidores. Para tener un conocimiento más amplio en los siguientes apartados intentare explicar qué diferencias existen entre desarrollo de servicios web basados en SOAP y REST, cómo se relacionan y los beneficios esperados de cada uno de los estilos.

3.2.1. SOAP

Es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambios de datos XML, el punto identificativo de SOAP es que las operaciones son definidas como puertos WSDL. Es por esto que será aconsejable utilizar este protocolo en entornos donde se establecerá un contrato formal y donde se describirán todas las funciones de la interfaz así como el tipo de datos utilizados tanto de entrada como de salida.

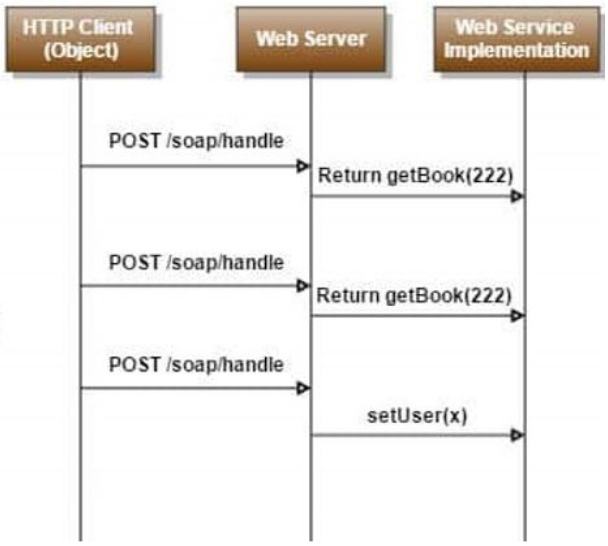


Figura 3.2: Diagrama de comunicaciones SOAP.

Ventajas

- Muy sencillo de consumir si trabajas con componentes y utilizas .NET o Java.
- El resultado (XML) contiene una definición específica del tipo de dato, lo que hace que el protocolo sea estricto.
- Seguro, la implementación suele hacerse del lado del servidor.

Desventajas

- Una vez implementado, si se desea cambiar algo en el servidor impacta de forma negativa en los clientes ya que estos tienen que hacer muchas modificaciones al código.
- Las respuestas son demasiado complejas y difíciles de interpretar si no se tienen las herramientas correctas para hacerlo.
- Los clientes necesitan saber las operaciones y su semántica antes de la comunicación.
- Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones.
- Las instancias del proceso se creadan implícitamente.

3.2.2. REST

A diferencia de SOAP, este estilo se centra en el uso de los estándares HTTP y XML para la transmisión de datos, sin la necesidad de contar con una capa adicional. Las operaciones se solicitarán mediante GET, POST, PUT y DELETE, por lo que no requiere de implementaciones especiales para consumir estos servicios. Además se podrá utilizar JSON en vez de XML como contenedor de la información, por lo que será aconsejable utilizar este protocolo cuando busquemos mejorar el rendimiento, o cuando disponemos de escasos recursos, como sería el caso de este proyecto.

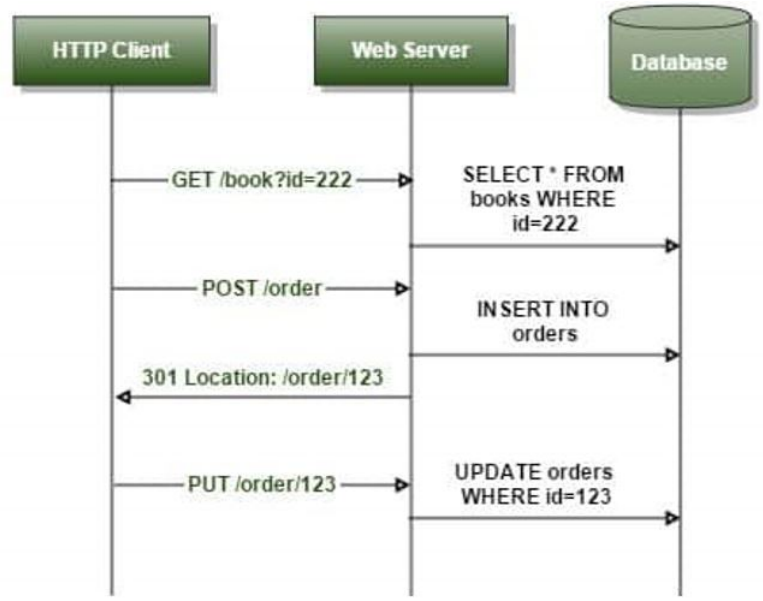


Figura 3.3: Diagrama de comunicaciones REST.

Ventajas

- Es muy ligero, sus respuestas contienen exactamente la información que necesitamos.
- Es sencillo de desarrollar y no se necesita mucho código extra.
- Los clientes pueden tener una interfaz ”listener” genérica para las notificaciones.
- Generalmente fácil de construir y adoptar.
- La API REST siempre es independiente del tipo de plataformas o lenguajes. Con una API REST se pueden tener servidores PHP, Java, Python o Node.JS.

Desventajas

- La seguridad es un problema y puede ser una tarea difícil implementarla correctamente.
- No hay un estándar en sus respuestas por lo que no se definen tipos de datos.
- Manejar el espacio de nombres (URIs) puede ser engorroso.

3.2.3. Comparación de las alternativas

En la siguente sección se analizaran los rasgos más importantes de ambos estilos:

Características

REST:

- Las operaciones se definen en los mensajes.
- Una dirección única para cada instancia del proceso.
- Cada objeto soporta las operaciones estándares definidas.
- Componentes débilmente acoplados.

SOAP:

- Las operaciones son definidas como puertos WSDL.
- Dirección única para todas las operaciones.
- Múltiple instancias del proceso comparten la misma operación.
- Componentes fuertemente acoplados

Tecnología

REST:

- Interacción dirigida por el usuario por medio de formularios.
- Mecanismo consistente de nombrado de recursos (URI).
- Pocas operaciones con muchos recursos.
- Se centra en la escalabilidad y rendimiento a gran escala para sistemas distribuidos hipermedia.

SOAP:

- Flujo de eventos orquestados.
- Falta de un mecanismo de nombrado.
- Muchas operaciones con pocos recursos.
- Se centra en el diseño de aplicaciones distribuidas.

Protocolo

REST:

- XML auto descriptivo
- Síncrono

SOAP:

- Tipado fuerte, XMl Schema
- Síncrono y Asíncrono

Descripción del servicio

REST:

- Confía en documentos orientados al usuario que define las direcciones de petición y las respuestas.
- Interactuar con el servicio supone horas de testeado y depuración de URIs.
- No es necesario el tipado fuerte, si ambos lados están de acuerdo con el contenido

SOAP:

- WSDL.
- Se pueden construir automáticamente stubs (clientes) por medio del WSDL.
- Tipado fuerte.

Descripción del servicio

REST:

- El servidor no tiene estado (stateless).
- Los recursos contienen datos y enlaces representando transiciones a estados válidos.
- Los clientes mantienen el estado siguiendo los enlaces.
- Técnicas para añadir sesiones: Cookies

SOAP:

- El servidor puede mantener el estado de la conversación.
- Los mensajes solo contienen datos.
- Los clientes mantienen el estado suponiendo el estado del servicio.
- Técnicas para añadir sesiones: Cabecera de sesión (no estándar)

Metodología de diseño

REST:

- Identificar recursos a ser expuestos como servicios.
- Definir URLs para direccionarlos.
- Distinguir los recursos de solo lectura (GET) de los modificables (POST,PUT,DELETE).
- Implementar e implantar el servidor Web.

SOAP:

- Listar las operaciones del servicio en el documento WSDL.
- Definir un modelo de datos para el contenido de los mensajes.
- Elegir un protocolo de transporte apropiado y definir las correspondientes políticas QoS, de seguridad y transaccional.
- Implementar e implantar el contenedor del servicio Web.

Seguridad

REST:

- HTTPS.
- Comunicación punto a punto y segura.
- Implementada desde hace muchos años.

SOAP:

- Comunicación origen a destino segura.
- Implementada desde hace muchos años.

3.2.4. Conclusiones

Según hemos visto a lo largo del documento, el principal beneficio de SOAP recae en ser fuertemente acoplado, lo que permite poder ser testado y depurado antes de poner en marcha la aplicación sin ningún problema. Si se está pensando en servicios web para corporaciones donde se manejan datos complejos y se necesita una precisión detallada en las respuestas se debe de utilizar SOAP.

Tanto el productor como el consumidor de nuestro servicio de lectura de tarjetas conocen el contexto y contenido que va a ser comunicado, ambas partes están de acuerdo en el modo de intercambiar de información. Sabiendo que REST tiene mucho potencial de escalabilidad, así como el escaso consumo de recursos debido al limitado número de operaciones (suficientes para todos los casos de usos de este proyecto) hacen que sea la mejor elección.

Teniendo en cuenta todo lo anteriormente mencionado y conociendo el protocolo de comunicación (más información en el capítulo 5) de nuestro servicio, podemos concluir diciendo que un modelo basado en REST va a ser más beneficioso para nuestro proyecto. Por lo cual en los siguientes capítulos podremos ver cómo construir dicha API REST.

3.3. Reglas de la arquitectura REST

Según la definición teórica, REST es un estilo de arquitectura que abstrae los elementos de dicha arquitectura dentro de un sistema distribuido. Pero esto de por sí no nos dice demasiado a no ser que nos dediquemos al estudio teórico de este tipo de cosas. Resumiendo, podría decir que REST es un conjunto de principios, o maneras de hacer las cosas, que define la interacción entre distintos componentes, es decir, las reglas que dichos componentes tienen que seguir.

Cabe aclarar que cuando hablamos de Rest y RestFull no hablamos de conceptos diferentes, sino que es lo mismo, salvo ciertos matices. Rest, como vimos antes, es un concepto de arquitectura que siguen los principios anteriormente mencionados, en cambio RestFull son los servicios web que siguen esos principios.

RESTless		RESTful	
VERB	HREF	VERB	URI
POST	/users/create	POST	/users
GET	/users/1	GET	/users/1
POST	/users/1/update	PUT	/users/1
????	/users/1/delete	DELETE	/users/1

Figura 3.4: Diferencias entre servicios RESTless y RESTful.

REST define una serie de reglas que toda aplicación que pretenda llamarse REST debe cumplir. Conviene repasar estos conceptos para tener una visión más profunda, y para más adelante ayudarme de ello a la hora de implementar la API de este proyecto.

3.3.1. Arquitectura cliente-servidor

Consiste en una separación clara y concisa entre los 2 agentes básicos en un intercambio de información: el cliente y el servidor. Estos 2 agentes deben ser independientes entre sí, lo que permite una flexibilidad muy alta en todos los sentidos.

3.3.2. Stateless

Esto significa que nuestro servidor no tiene porqué almacenar datos del cliente para mantener un estado del mismo. Esta limitación es sujeto de mucho debate en la industria,

incluso ya empiezan a usarse tecnologías relacionadas que implementan el estado dentro de la arquitectura, como WebSockets.

3.3.3. Cacheable

Esta norma implica que el servidor que sirve las peticiones del cliente debe definir algún modo de cachear dichas peticiones, para aumentar el rendimiento, escalabilidad, etc.

3.3.4. Sistema por capas

Nuestro sistema no debe forzar al cliente a saber por qué capas se tramita la información, lo que permite que el cliente conserve su independencia con respecto a dichas capas.

3.3.5. Interfaz uniforme

Esta regla simplifica el protocolo y aumenta la escalabilidad y rendimiento del sistema. No quiero que la interfaz de comunicación entre un cliente y el servidor dependa del servidor al que hago las peticiones, ni mucho menos del cliente, por lo que esta regla nos garantiza que no importa quien haga las peticiones ni quien las reciba, siempre y cuando ambos cumplan una interfaz definida de antemano.

3.4. Metodología REST

3.4.1. Servicios web RESTful

Sabiendo que un servicio web RESTful hace referencia a un servicio web que implementa la arquitectura REST, puedo ya dar una definición concisa, lo cual nos dejará claro cómo tenemos que implementarlo en los siguientes capítulos. Un servicio web RESTful contiene lo siguiente:

- **URI del recurso:** Por ejemplo: http://api.servicio.com/recursos/casas/1 (esto nos daría acceso al recurso “Casa” con el ID “1”)

- **El tipo de la representación de dicho recurso:** Por ejemplo, podemos devolver en nuestra cabecera "Content-type: application/json", por lo que el cliente sabrá que el contenido de la respuesta es una cadena en formato JSON, y podrá procesarla como prefiera. El tipo es arbitrario, siendo los más comunes JSON, XML y TXT.
- **Operaciones soportadas:** HTTP define varios tipos de operaciones (verbos) [3], que pueden ser GET, PUT, POST, DELETE, PURGE, entre otros. Es importante saber para que están pensados cada verbo, de modo que sean utilizados correctamente por los clientes.
- **Operaciones soportadas:** HTTP define varios tipos de operaciones (verbos) [3], que pueden ser GET, PUT, POST, DELETE, PURGE, entre otros. Es importante saber para que están pensados cada verbo, de modo que sean utilizados correctamente por los clientes.

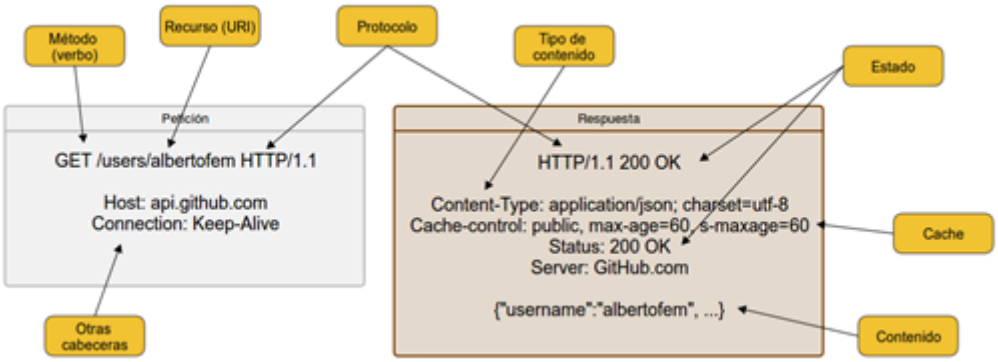


Figura 3.5: Ejemplo estructura de una petición al servicio web de GitHub

3.4.2. Pasos para la construcción

1. Identificar todas las entidades conceptuales que se desean exponer como servicio.
2. Crear una URL para cada recurso. Los recursos deberían ser nombres no verbos (acciones).
3. Categorizar los recursos de acuerdo con si los clientes pueden obtener un representación del recurso o si pueden modificarlo. Para el primero, debemos hacer los recursos accesibles utilizando un HTTP GET. Para el último, debemos hacer los recursos accesibles mediante HTTP POST, PUT y/o DELETE.

4. Todos los recursos accesibles mediante GET no deberían tener efectos secundarios. Es decir, los recursos deberían devolver la representación del recurso. Por tanto, invocar al recurso no debería ser el resultado de modificarlo.

5. Ninguna representación debería estar aislada. Es decir, es recomendable poner hipervínculos dentro de la representación de un recurso para permitir a los clientes obtener más información.

6. Especificar el formato de los datos de respuesta mediante un esquema (DTD, W3C Schema,...). Para los servicios que requieran un POST o un PUT es aconsejable también proporcionar un esquema para especificar el formato de la respuesta.

7. Describir como nuestro servicio ha de ser invocado, mediante un documento WSDL/WADL o simplemente HTML.

3.4.3. Diseño

De nuevo tomare como ejemplo a la Web. La Web evidentemente es un ejemplo clave de diseño basado en REST, ya que muchos principios son la base de REST.

HTML puede incluir JavaScript y applets, los cuales dan soporte al codeon-demand, y además tiene implícitamente soporte a los vínculos. HTTP posee un interfaz uniforme para acceso a los recursos, el cual consiste de URIs, métodos, códigos de estado, cabeceras y un contenido guiado por tipos MIME.

Los métodos HTTP más importantes son PUT, GET, POST y DELETE. Ellos suelen ser comparados con las operaciones asociadas a la tecnología de base de datos, operaciones CRUD: CREATE, READ, UPDATE, DELETE. Todas las analogías se representan en la siguiente tabla:

Acción	HTTP	SQL	Copy and Paste	Unix Shell
Create	PUT	Insert	Pegar	>
Read	GET	Select	Copiar	<
Update	POST	Update	Pegar después	»
Delete	DELETE	Delete	Cortar	Del/rm

Tabla 3.1: Tabla de analogías de los metodos REST.

Las acciones (verbos) CRUD se diseñaron para operar con datos atómicos dentro del contexto de una transacción con la base de datos. REST se diseña alrededor de transferen-

cias atómicas de un estado más complejo, tal que puede ser visto como la transferencia de un documento estructurado de una aplicación a otra.

El protocolo HTTP separa las nociones de un servidor y un navegador. Esto permite a la implementación cada uno variar uno del otro, basándose en el concepto cliente/servidor.

Cuando utilizamos REST, HTTP no tiene estado. Cada mensaje contiene toda la información necesaria para comprender la petición cuando se combina el estado en el recurso. Como resultado, ni el cliente ni el servidor necesita mantener ningún estado en la comunicación. Cualquier estado mantenido por el servidor debe ser modelado como un recurso.

3.5. Lenguajes para la construcción de un API REST

Una vez descrito la metodología para crear nuestra API. Tenemos varias opciones a la hora de elegir el lenguaje de programación. A continuación vamos a ver una pequeña introducción sobre los leguajes y frameworks más populares para la construcción de una API Rest.

3.5.1. Python: Djngo

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero efectivo a la programación orientada a objetos. La elegante sintaxis de Python y su tipado dinámico, junto con su naturaleza interpretada, hacen de éste un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas.

Djngo es un framework de desarrollo web de código abierto, escrito en Python, que respeta el paradigma conocido como Model Template View.

La meta fundamental de Django es facilitar la creación de sitios web complejos. Django pone énfasis en el re-uso, la conectividad y extensibilidad de componentes, el desarrollo rápido y el principio No te repitas (DRY, del inglés Don't Repeat Yourself). Python es usado en todas las partes del framework, incluso en configuraciones, archivos, y en los modelos de datos.

La distribución principal de Django también aglutina aplicaciones que proporcionan un sistema de comentarios, herramientas para sindicar contenido via RSS y/o Atom, "páginas planas"que permiten gestionar páginas de contenido sin necesidad de escribir controladores o vistas para esas páginas, y un sistema de redirección de URLs.

Figura 3.6: Logo de Python y Django.

3.5.2. Ruby: Rails

Ruby es un lenguaje de programación enfocado a la simplicidad y a la productividad, con una sintaxis elegante y natural, que le hace muy fácil de entender. Es un lenguaje no compilado, totalmente orientado a objetos. Mientras que otros lenguajes tienen tipos primitivos que no son objetos, aquí todo es un objeto y por tanto se le pueden asociar propiedades y métodos, así como redefinir o extender su comportamiento.

Rails es un framework creado para el desarrollo de aplicaciones web. Entre las características de Ruby on Rails podemos destacar que está basado en el patrón de diseño MVC (Modelo-Vista-Controlador) con el que se separa la lógica de la presentación, que nos permite dejar de pensar siempre en SQL para pasar a pensar en objetos, dispone de utilidades para generar rápidamente interfaces de administración, es independiente de la base de datos, podemos realizar teses continuos para garantizar que nuestra aplicación funciona como esperamos, se puede extender mediante el uso de plugins, se integra muy fácilmente con librerías de efectos y todo ello escribiendo muy pocas líneas de código, muy claras y fáciles de entender.

3.5.3. .NET: WebAPI

.NET es un entorno para construir, instalar y ejecutar servicios Web y otras aplicaciones. Se compone de tres parte principales: el Common Language Runtime, las clases

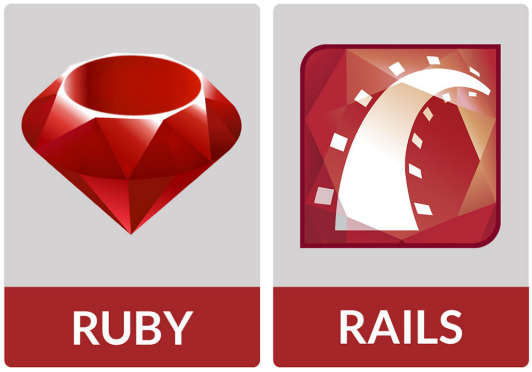


Figura 3.7: Logo de Ruby y Rails.

Framework y ASP.NET.

.NET ofrece una plataforma sencilla y potente para distribuir el software en forma de servicios que puedan ser suministrados remotamente y que puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido desarrollados.

Web API es un marco que facilita la creación de servicios HTTP disponibles para una amplia variedad de clientes, entre los que se incluyen exploradores y dispositivos móviles. ASP.NET Web API es la plataforma perfecta para crear aplicaciones RESTful en .NET Framework.

La ventaja, como todo framework, es que nos da hecho el trabajo repetitivo, y nos provee de una arquitectura MVC con unas buenas bases desde la que iniciarnos en el desarrollo. También incluye las referencias para utilizar Entity Framework 5.0, JQuery y, muy importante, construye el proyecto de testing adecuado para el desarrollo en TDD (o casi).

Figura 3.8: Logo de .NET.

3.5.4. PHP: Codeigniter

PHP es un lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Es usado principalmente en interpretación del lado del servidor (server-side scripting) pero actualmente puede ser utilizado desde una interfaz de línea de comandos o en la creación de otros tipos de programas incluyendo aplicaciones con interfaz gráfica usando las bibliotecas Qt o GTK+.

CodeIgniter es un entorno de desarrollo abierto que permite crear webs dinámicas en PHP. Su principal objetivo es ayudar a que los desarrolladores, puedan realizar proyectos mucho más rápido que creando toda la estructura desde cero. A parte del núcleo MVC, CodeIgniter viene con muchísimos módulos que nos ayudarán a no tener que reescribir los fragmentos de código más comunes. Hay bibliotecas que permiten, desde manejar la base de datos, hasta generar thumbnails de las imágenes que suben tus usuarios.

Figura 3.9: Logo de PHP y codeigniter.

3.5.5. JavaScript: Node.JS.

JavaScript también es un lenguaje de programación interpretado. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios. A pesar de su nombre, JavaScript no guarda ninguna relación directa con el lenguaje de programación Java. Legalmente, JavaScript es una marca registrada de la empresa Sun Microsystems.

A excepción de todos los demás ejemplos que hemos visto anteriormente, Node.JS es un entorno de programación de lado de servidor que utiliza un modelo asíncrono y dirigido por eventos y no un framework para JavaScript. Construida encima del entorno de ejecución de Chrome para construir rápidas y escalables aplicaciones de red. Node.JS usa un modelo de E/S no bloqueante dirigido por eventos que lo hace ligero y eficiente, perfecto para aplicaciones data-intensive en tiempo real.

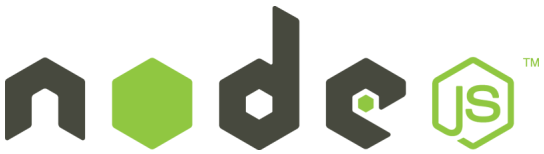


Figura 3.10: Logo de Node.JS.

3.6. Decisión

Hemos visto que tenemos diferentes tipos de posibilidades a la hora de abordar la construcción de la API REST, pero después de analizar los distintos tipos de leguajes y debido a las exigencias de la empresa ATELEI decidí desarrollar el sistema con Node.JS y con el framework codeigniter de PHP. En los siguientes capítulos podremos profundizar más ya que me centrare en analizar y comparar las dos alternativas escogidas para la construcción del sistema.

4. CAPÍTULO

Análisis y comparativa de las alternativas escogidas

En este capítulo se expondrán las dos alternativas (PHP y Node.JS) para la construcción de un sistema que permita interactuar un servidor con un dispositivo, en mi caso un lector de tarjetas. Primero, se analizaran los puntos fuertes y débiles de cada una de las alternativas para más adelante comparar el flujo de instrucciones. Por ultimo veremos un ejemplo sencillo de benchmarking, debido a que en el capítulo 9 llevare a cabo el análisis en profundidad con los sistemas desarrollados.

Objetivos del proyecto
1. PHP - Codeigniter 2. Node.JS 3. Sistema síncrono vs asíncrono 4. Introducción al benchmarking 5. Conclusión

4.1. PHP - Codeigniter

Codeigniter es un framework para PHP, por lo cual antes de nada vamos a conocer un poco más algo sobre el leguaje.

PHP es un lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Usado principalmente en interpretación del lado del servidor (server-side scripting) pero actualmente puede ser utilizado desde una interfaz de línea de comandos o en la creación de otros tipos de programas incluyendo aplicaciones con interfaz gráfica usando las bibliotecas Qt o GTK+.

Tiene capacidad de conexión con la mayoría de los motores de base de datos que se utilizan en la actualidad, cabe destacar su conectividad con MySQL y PostgreSQL. Con PHP puedes procesar la información de formularios, generar páginas con contenidos dinámicos, o enviar y recibir cookies, entre muchas más cosas. PHP lo utilizan desde pequeñas páginas web hasta grandes empresas. Muchas aplicaciones web están construidas usando PHP. Podemos citar Joomla y Drupal (gestores de contenido de páginas web), osCommerce y Prestashop (tiendas on-line para comercio electrónico), phpBB y SMF (sistemas de foros para páginas web), Moodle (plataforma educativa para educación on-line), etc.

4.1.1. Modelo de ejecución

El primer paso para una comunicación es escribir en la barra de direcciones del navegador la url de la página web que queremos ver. En segundo lugar, el navegador envía el mensaje a través de internet al ordenador remoto (servidor), de acuerdo con un protocolo estandarizado, solicitando la página (archivo) index.php.

El servidor web recibe el mensaje, comprueba que se trata de una petición válida, y al ver que la extensión es "php"solicita al intérprete de PHP que le envíe el archivo.

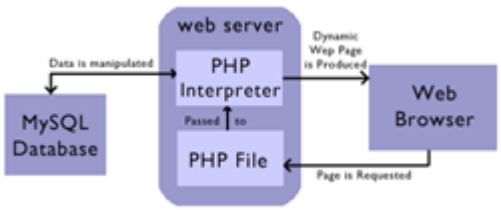


Figura 4.1: Actuadores implicados en una comunicación PHP.

No se trata de una simple extracción de un archivo desde el disco duro, sino que está actuando un agente intermediario: el intérprete PHP. El intérprete PHP lee desde el disco duro del servidor el archivo index.php y empieza a procesar las instrucciones que contenga dicho archivo. El intérprete PHP "ejecuta" los comandos contenidos en el archivo y, si es necesario, se comunica con un gestor de base de datos. La comunicación con base de datos no siempre se produce, pero es algo muy frecuente cuando trabajamos con PHP. Tenemos pues otra diferencia con las webs estáticas: interviene otro agente más, el gestor de base de datos, que es capaz de devolver la información contenida en lugares determinados de una base de datos. Y una base de datos podemos verla simplemente como un gran almacén de información organizada en tablas.

Una vez el intérprete PHP termina de ejecutar el código contenido en el archivo y ha recibido toda la información necesaria del gestor de base de datos, envía los resultados al servidor web. El servidor web envía la página al cliente que la había solicitado y el navegador muestra en pantalla la información que le envía el servidor web.

En cuanto a la gestión de hilos vemos que en PHP tenemos un sistema síncrono donde cada petición abrirá un nuevo hilo, como podemos comprobar en la imagen 4.2.

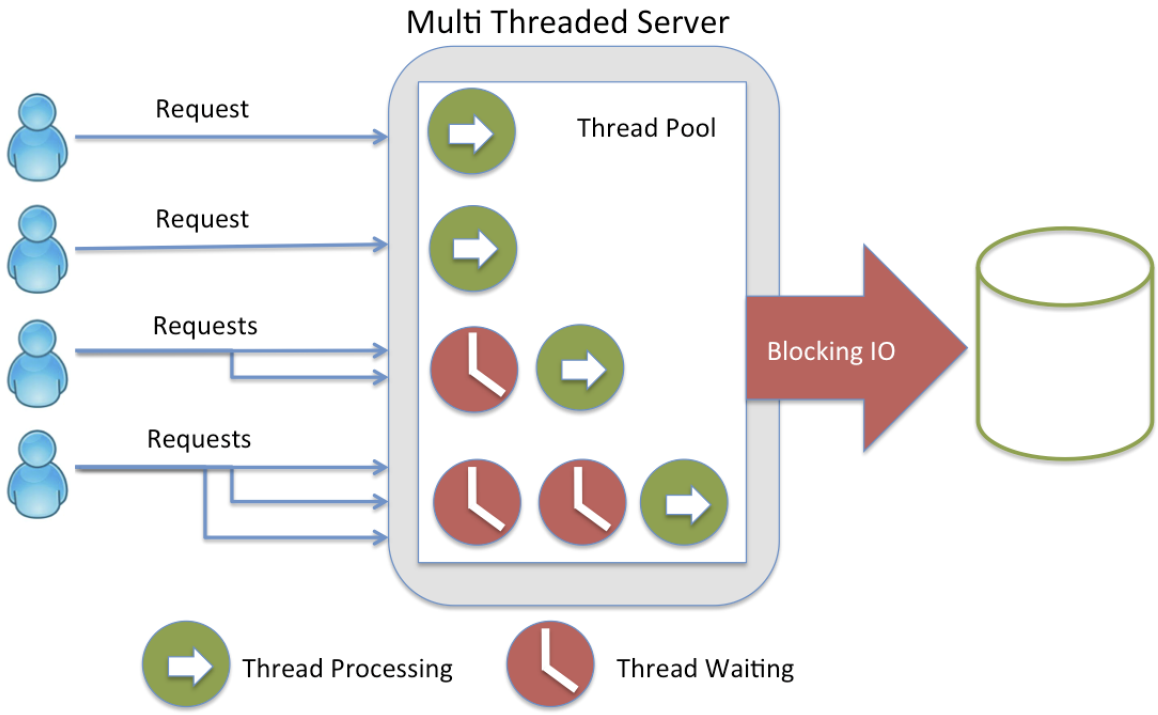


Figura 4.2: Modelo de gestion de hilos en servidores PHP.

4.1.2. Ventajas

La principal ventaja de un lenguaje interpretado como PHP es que es independiente de la máquina y del sistema operativo ya que no contiene instrucciones propias de un procesador sino que contiene llamadas a funciones que el intérprete deberá reconocer. Basta que exista un intérprete de un lenguaje para dicho sistema y todos los programas escrito en ese lenguaje funcionaran. Además un lenguaje interpretado permite modificar en tiempo de ejecución el código que se está ejecutando así como añadirle nuevo, algo que resulta idóneo cuando queremos hacer pequeñas modificaciones en una aplicación y no queremos tener que recompilarla toda cada vez.

4.1.3. Desventajas

Debido a que es un lenguaje interpretado, los scripts en PHP suele funcionar considerablemente más lento que su equivalente en un lenguaje de bajo nivel, sin embargo este inconveniente se puede minimizar con técnicas de caché tanto en archivos como en memoria. Además como se interpreta en ejecución, para ciertos usos puede resultar un inconveniente que el código fuente no pueda ser ocultado.

4.1.4. Framework: Codeigniter

PHP es un lenguaje ya asentado desde hace muchos años con lo que hay un gran numero frameworks en el mercado, en la siguiente tabla podemos ver los más populares:

Framework	PHP 4	PHP 5	MVC	Modules	ORM	EDP	Auth	Cache	Validator	Ajax	License
Zend Framework	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	New BSD
CakePHP	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	MIT
symfony	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	MIT
CodeIgniter	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	Apache/BSD
Seagull	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	BSD
Prado	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	BSD
Solar	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	New BSD
eZ Components	✗	✓	✓	✓	✗	✗	✓	✓	✓	✓	New BSD

Figura 4.3: Tabla de los frameworks mas populares en PHP.

Tomé la decision de escoger codeigniter para la construccion de mi API REST debido a las exigencias de intregracion de la empresa ATELEI. Mas adelante en este mismo capitulo podemos ver las ventajes que nos proporciona este framework.

Como cualquier otro framework, Codeigniter contiene una serie de librerías que sirven para el desarrollo de aplicaciones web. Esto es, marca una manera específica de codificar las páginas web y clasificar sus diferentes scripts, que sirve para que el código esté organizado y sea más fácil de crear y mantener. CodeIgniter implementa el proceso de desarrollo llamado Model View Controller (MVC), que es un estándar de programación de aplicaciones, utilizado tanto para hacer sitios web como programas tradicionales.

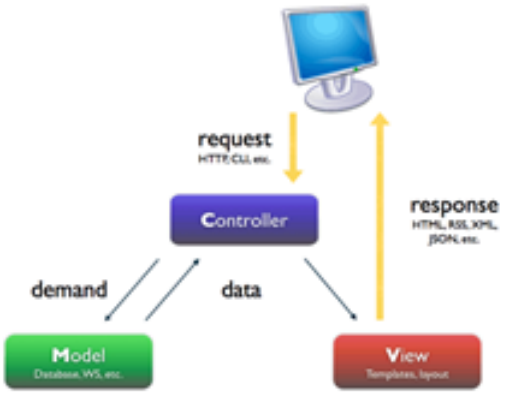


Figura 4.4: Arquitectura MVC base de Codeigniter.

Características generales de CodeIgniter

Algunos de los puntos más interesantes sobre este framework, sobre todo en comparación con otros productos similares, son los siguientes:

Versatilidad: Quizás la característica principal de CodeIgniter, en comparación con otros frameworks PHP. CodeIgniter es capaz de trabajar la mayoría de los entornos o servidores, incluso en sistemas de alojamiento compartido, donde sólo tenemos un acceso por FTP para enviar los archivos al servidor y donde no tenemos acceso a su configuración.

Compatibilidad: CodeIgniter, al menos en el momento de escribir este artículo de desarrolloweb.com, es compatible con la versión PHP 4, lo que hace que se pueda utilizar en cualquier servidor, incluso en algunos antiguos. Por supuesto, funciona correctamente también en PHP 5.

Actualizado: Desde la versión 2 de CodeIgniter ya solo es compatible con la versión 5 de PHP. Para los que todavía usen PHP 4 pueden descargar una versión antigua del framework, como CodeIgniter V 1.7.3, que todavía era compatible. Estas versiones están en la página de descargas de CodeIgniter.

Facilidad de instalación: No es necesario más que una cuenta de FTP para subir CodeIgniter al servidor y su configuración se realiza con apenas la edición de un archivo, donde debemos escribir cosas como el acceso a la base de datos. Durante la configuración no necesitaremos acceso a herramientas como la línea de comandos, que no suelen estar disponibles en todos los alojamientos.

Flexibilidad: CodeIgniter es bastante menos rígido que otros frameworks. Define una manera de trabajar específica, pero en muchos de los casos podemos seguirla o no y sus reglas de codificación muchas veces nos las podemos saltar para trabajar como más a gusto encontremos. Algunos módulos como el uso de plantillas son totalmente opcionales. Esto ayuda muchas veces también a que la curva de aprendizaje sea más sencilla al principio.

Ligereza: El núcleo de CodeIgniter es bastante ligero, lo que permite que el servidor no se sobrecargue interpretando o ejecutando grandes porciones de código. La mayoría de los módulos o clases que ofrece se pueden cargar de manera opcional, sólo cuando se van a utilizar realmente.

4.1.5. Ventajas de CodeIgniter

- Las páginas se procesan más rápido, ya que el núcleo de CodeIgniter es bastante ligero.
- Es sencillo de instalar, basta con subir los archivos al ftp y tocar un archivo de configuración para definir el acceso a la base de datos.
- Reutilización de código, desarrollo ágil.
- Existe abundante documentación en la red.
- Facilidad para crear nuevos módulos, páginas o funcionalidades.
- Acceso a librerías públicas y clases.
- Estandarización del código. Fundamental cuando hay que tocar código hecho por otra persona o cuando trabaja más de una persona en un mismo proyecto.

- URLs amigables con SEO. Hoy en día creo que nadie duda de la importancia del posicionamiento web.
- Separación de la lógica y arquitectura de la web, el MVC.
- CodeIgniter es bastante menos rígido que otros frameworks. Define una manera de trabajar, pero podemos seguirla o no(esto puede convertirse en un inconveniente también)
- Cualquier servidor que soporte PHP+MySQL sirve para CodeIgniter.
- CodeIgniter se encuentra bajo una licencia open source, es código libre.
- CodeIgniter usa una versión modificada del Patrón de Base de Datos Active Record. Este patrón permite obtener, insertar y actualizar información in tu base de datos con mínima codificación. Permite queries más seguras, ya que los valores son escapadas automáticamente por el sistema.

4.2. Node.JS

Node.JS es una librería y entorno de ejecución de Entrada/Salida dirigida por eventos y por lo tanto asíncrona que se ejecuta sobre el intérprete de JavaScript creado por Google V8. Es una idea completamente revolucionaria ya que, a diferencia de las aplicaciones web tradicionales que usan PHP en el servidor, utiliza JavaScript, un modelo completamente asíncrono.

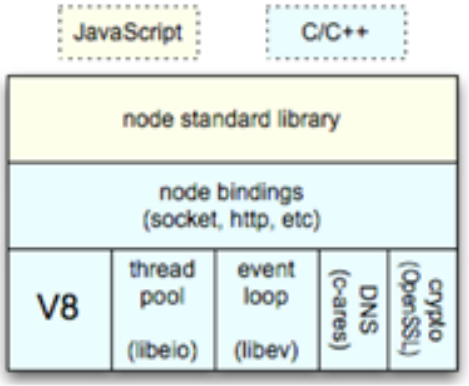


Figura 4.5: Arquitectura del Node.JS.

V8 es el motor de JavaScript que Google usa en su navegador Chrome. Un motor normal de JavaScript, interpreta el código y lo ejecuta. El V8 es ultra-rápido, está escrito en C++ y se puede descargar por separado e incorporarlo a cualquier aplicación. Así nace Node.JS, cambiando el propósito por el que se creó V8 y usándolo en el lado del servidor.

Además de la alta velocidad de ejecución de Javascript, la verdadera magia detrás de Node.JS es algo que se llama Bucle de Eventos (Event Loop). Para escalar grandes volúmenes de clientes, todas las operaciones intensivas I/O en Node.JS se llevan a cabo de forma asíncrona. El enfoque tradicional para generar código asíncrono es engorroso y crea un espacio en memoria no trivial para un gran número de clientes (cada cliente genera un hilo, y el uso de memoria de cada uno se suma). Para evitar esta ineficiencia, así como la dificultad conocida de las aplicaciones basadas en hilos, (programming threaded applications), Node.JS mantiene un event loop que gestiona todas las operaciones asíncronas.

Node.JS dispone de un gestor de módulos (librerías, plugins, frameworks) npm (Node Package Manager), que amplían la funcionalidad de este y facilitan tareas. En otro futuro post trataremos de hablar de varios frameworks típicos de Node.js, en concreto de Fusker, que es un firewall de aplicación que previene y maneja una gran cantidad de ataques en Node.JS.

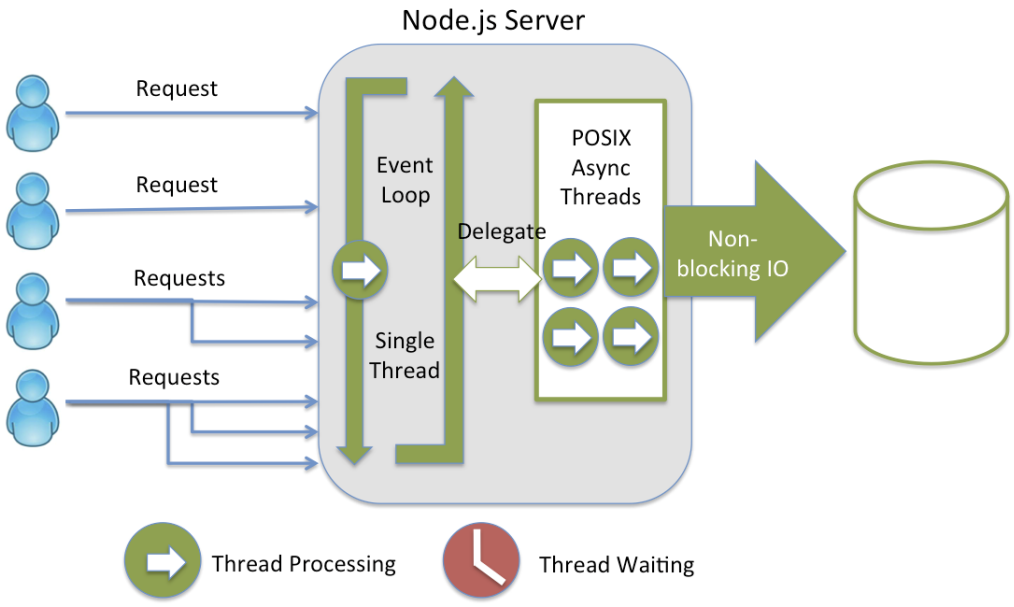


Figura 4.6: Modelo de gestion de hilos en servidores Node.JS.

4.2.1. ¿Qué es un sistema basado en eventos?

Mientras que en la programación secuencial es el programador el que determina el flujo del programa, en la programación basada en eventos el flujo del programa viene determinado por eventos.

- Los eventos: Un evento es un suceso que cambia la ejecución del programa para ser atendido. Estos sucesos pueden ser: Interrupciones del sistema, entrada del teclado, comunicación con un puerto, etc.
- Los cambios en el programa: Cuando sucede un evento, este cambia la ejecución del programa, que viene definido por el programador. Normalmente existe un bucle externo al programa, que registra estos eventos y ejecuta la función correspondiente. Node.JS ya incorpora este bucle para registrar eventos de forma transparente al programador.

Esto no significa que cuando llega un evento, se abandona la ejecución anterior, simplemente el programa no ejecuta nada hasta que un evento sea activado. En el caso en que llamen dos eventos consecutivos, uno no anula o detiene al otro.

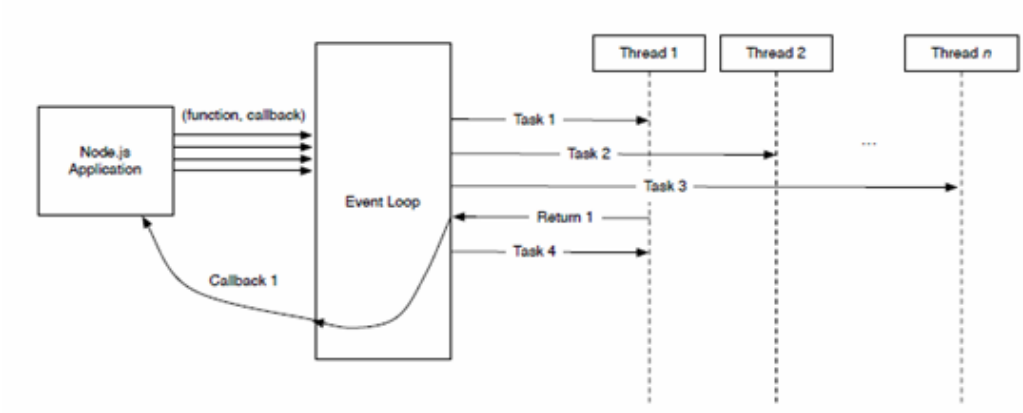


Figura 4.7: Funcionamiento interno del bucle de eventos en Node.JS.

4.2.2. Arquitectura

Node.JS está organizado en una arquitectura por capas, que garantiza su estabilidad y compatibilidad. Esta estructura consta de cinco capas, las cuales realizan la siguiente función:

- **Dependencias**: Son librerías que contienen el motor V8, JavaScript, libuv, openssl, etc. Estas dependencias son las encargadas del funcionamiento básico.

- Interfaz binaria de la aplicación: Permite la comunicación y proporciona acceso entre los módulos y aplicaciones con las dependencias.

- **Biblioteca del núcleo**: Es la interfaz principal a través de la cual la mayoría de los módulos y aplicaciones de la capa superior de Node.JS realizan operaciones de E/S, manipular datos, acceso a la red, etc. Algunos módulos y aplicaciones optan por enlazar directamente a la Interfaz binaria de la aplicación o incluso saltar directamente a las dependencias para realizar operaciones más avanzadas.

- **Capa de abstracción binaria**: Consiste en una capa opcional utilizada para amortiguar los cambios entre la Interfaz binaria de aplicaciones y las Dependencias con las aplicaciones desarrolladas en la capa superior.

- **Entorno de módulos y aplicaciones**: Son las aplicaciones desarrolladas en Node.JS por los usuarios. Esta estructura por capas garantiza la compatibilidad cuando se realizan cambios importanarquitectura-node.pngtes entre versiones.

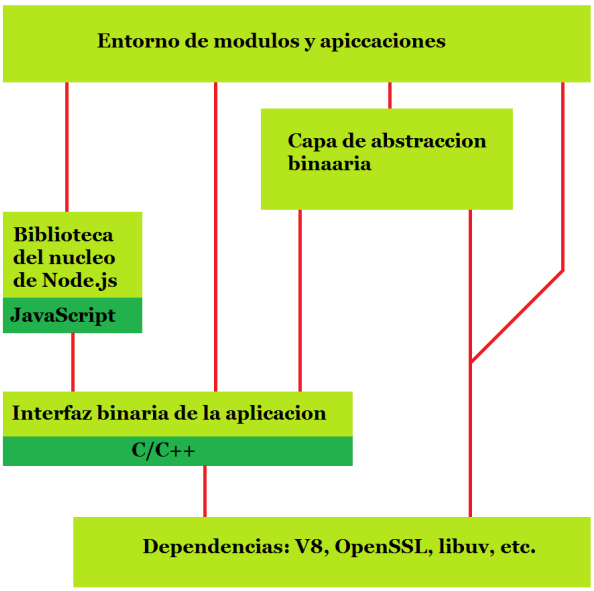


Figura 4.8: Esquema de la arquitectura de Node.JS.

4.2.3. Modelo de ejecución

El modelo de programación de Node.JS es mono hilo, asíncrono y dirigido por eventos. Node.JS trabaja con un único hilo de ejecución que es el encargado de organizar todo el flujo de trabajo que se deba realizar.

Para poder trabajar de una forma óptima Node.JS delega todo el trabajo en un pool de threads. Este pool de threads está construido con la librería libuv. Esta librería dispone de su propio entorno multithread asíncrono. Node.JS envía el trabajo que hay que realizar al pool. Libuv realizará a través de alguno de sus threads el trabajo. Una vez que el trabajo haya sido completado libuv emitirá un evento que será recibido por Node.JS.

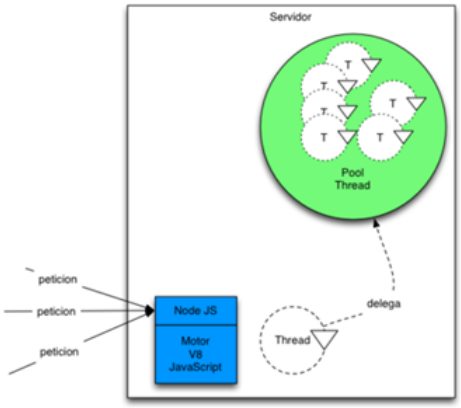


Figura 4.9: Ejemplo 01 gestión de hilos en Node.JS

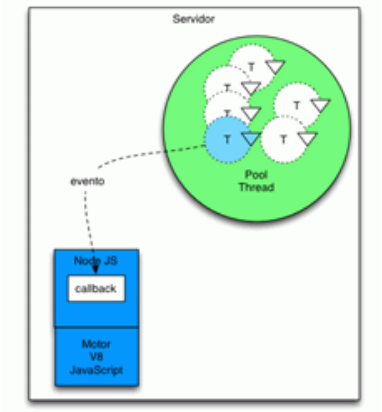


Figura 4.10: Ejemplo 02 gestión de hilos en Node.JS

Tener un sólo hilo de ejecución para todo el código tiene varias consecuencias en nuestro código de las que hay que estar alerta, estas son:

- No puede haber código bloqueante o todo el servidor quedará bloqueado y esto incluye no responder a nuevas peticiones entrantes. Cualquier tarea que pueda dar lugar a esperas activas o simplemente su tiempo de ejecución sea demasiado grande debe ser tratado de manera asíncrona. Esto incluye muy especialmente todas las tareas que impliquen algún tipo de comunicación I/O, salida a red, a base de datos o incluso al sistema de ficheros. Si tenemos alguna tarea computacionalmente intensiva deberemos intentar trocearla en varios bloques o trocearla en el tiempo (time-slice) para darle respiro al servidor y que pueda atender otros bloques de código, afortunadamente esto último no es nada habitual.

- La “asincronicidad” implica que no sabemos cuándo ni en qué orden se va a ejecutar el código, generalmente esto no es importante pero en ocasiones sí lo es y habrá que tenerlo en cuenta. En cuanto al cuándo lo más habitual será tener que tener en cuenta un posible evento de timeout, si estamos utilizando un módulo externo será tan simple como suscribirnos al evento, si es código propio utilizaremos la función setTimeout(). En cuanto al orden de ejecución, algo también poco habitual, si tenemos que controlarlo utilizaremos callbacks anidados o la función process.nextTick() para retrasar la ejecución dentro de la cola de eventos. Luego veremos ejemplos de utilización de estas funciones.

- En caso de error inesperado debemos capturarlo y controlar el posible estado en que haya podido quedar la ejecución del código. Muy especialmente en caso de haya recursos que no hayan sido liberados, haya tareas (bloques de código o callbacks) dependientes de la tarea que ha generado el error o tengamos una política de reintentos. Aunque si nuestro código es realmente asíncrono y dirigido por eventos la liberación de los recursos ocurrirá cuando salte el correspondiente evento de timeout asociado al recurso o el timeout asociado a la tarea dependiente. Recibido el evento una función de callback se encargará de terminar de procesarlo. Por eso cuando trabajamos con Node.JS prácticamente toda la programación es asíncrona.

En la siguiente figura vemos las diversas interacciones entre los componentes de la plataforma durante una instancia de Node.JS. Esto demuestra como el bucle de eventos delega el trabajo al hilo asociado a la Liberia libuv permitiendo una solución asíncrona.

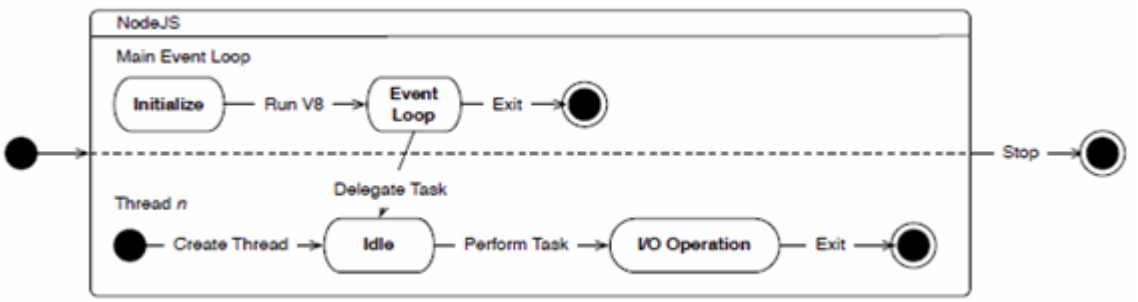


Figura 4.11: Flujo de interacciones de Node.JS.

4.2.4. Ventajas

Es especialmente útil cuando se van a realizar muchas operaciones simultáneas, sobretodo operaciones E/S. Node.JS te permite implementar al patrón de diseño MVC a través de varios archivos o módulos donde alojar el código de nuestras aplicaciones y de esta forma no tener que escribir todo el código en un solo archivo JavaScript.

- El lenguaje de programación es Javascript, con lo que se utilizaría el mismo lenguaje en el servidor que en el cliente, lo que permite la especialización en un tipo de tecnología determinado.
- Es una tecnología reciente, por lo que la especialización en un campo como este, sería una apuesta por una tecnología joven que ya ha captado la atención de una gran parte de desarrolladores.
- Existe una gran comunidad por detrás.
- Habría la posibilidad de hacer extensiones escritas en C/C++ para optimizar ciertas acciones que consuman muchos recursos.
- Es la implementación más cercana al planteamiento de los WebSockets
- Es utilizado en páginas web como http://www.craigslist.org/about/sites (una de las más importantes de EEUU), gmail o LinkedIn
- Cuando se hacen peticiones de muchos recursos a la misma vez, la filosofía de no bloquear hilos de Node.JS, permite atender a todas las peticiones de una forma más óptima.
- Si se cae otro tipo de servidor (apache, Tomcat, nginx, etc), o se pone en mantenimiento, sería completamente independiente a este servidor.
- Salvo casos excepcionales, no lanza errores, sino que excepciones, con lo que es difícil que caiga el servidor.

4.2.5. Desventajas

JavaScript es un lenguaje con un buen núcleo pero con una flaca librería estándar. Cosas que darías por hecho en otro lenguaje del lado del servidor simplemente no existen.

La falta inherente de organización de código se puede considerar una gran desventaja. Se nota su efecto claramente cuando el equipo de desarrollo no está muy familiarizado con la programación asíncrona o los patrones de diseño estándar. Simplemente hay demasiadas formas de programar y de obtener código desparejo y difícil de mantener.

- Si no se cuida el código, puede ser muy complicado de depurar a posteriori (acordémonos que es Javascript, y el tratamiento de eventos difiere un poco con respecto a la filosofía de otros lenguajes de programación).
- Los puertos necesarios para el funcionamiento del servidor, pueden ser un handicap a la hora de decantarse por esta tecnología.
- Muchas de las partes de la API, no tienen el estado "locked"que implica que no va a haber cambios en una determinada funcionalidad (parámetros, datos devueltos, etc).
- El envío de imágenes o archivos CSS muy pesados va a funcionar igual o un poco peor que otros servidores del mercado.
- Si la aplicación está detrás de un proxy, puede complicar el desarrollo.
- Si se cae en un nodo el manejador de eventos, implica que el servidor no podrá aceptar ni enviar peticiones.

4.3. Sistema síncrono vs asíncrono

Cuando un sistema síncrono ejecuta una llamada, las instrucciones posteriores a esa llamada no se ejecutan hasta que esta ha sido completada.

Node.JS es un sistema asíncrono, justo al contrario que PHP. Esto significa que las llamadas y métodos son ejecutados de forma secuencial, pero sin esperar a que la anterior llamada haya finalizado.

Un programa está sometido a esperas constantes como lectura de disco, llamadas a métodos de mucho coste, entradas de teclado, etc. Para que el programa se ejecute lo antes posible es necesario reducir esos tiempos de espera. Una solución consiste en continuar la ejecución del código sin esperar a la finalización de la llamada, a esto se le llama ejecución asíncrona.

Como podemos ver, con la programación asíncrona se reduce considerablemente el tiempo de ejecución, pero esto genera nuevos problemas, el más destacado es que las funciones no serán capaces de retornar valores. Este problema se puede resolver con distintas técnicas, pero requiere cambiar el estilo de programación para adaptarse a este nuevo sistema.

Figura 4.12: Modelos de ejecución, sistema síncrono vs. asíncrono.

4.4. Rasgos principales

En la siguiente tabla podemos ver un resumen de las diferencias de una forma más sencilla:

	PHP	Node.JS
Ejecución	Intérprete.	JIT de V8, muy bueno pero sin tipado estático que limita el rendimiento.
Fácil acceso para empezar con pocos recursos	PHP suele venir instalado por defecto y lo tenemos en muchos servidores web gratuitos.	No es lo normal.
Libraría básica / peso / complejidad	Librería en C con infinidad de funciones y extensiones. No demasiado ligera.	Librería de Javascript con un API asíncrona en C++, todo el resto programando en JavaScript. Bastante ligera.
Asistencia de IDE	El PDT basado en eclipse tiene muchas limitaciones por culpa de Zend. El Zend Studio es el ideal, pero es de pago. Mantiene bastantes más tipos y ayuda a detectar problemas en tiempo de edición.	La asistencia de IDE en javascript no es muy apropiada. Intenta suplirlo con APIs pequeñas y fáciles pero a largo plazo y con proyectos grandes eso no funciona.
Unittesting	Hay herramientas, pero, aun asi pocos proyectos opensource hacen uso de unittesting.	Tiene soporte nativo limitado, pero se suele hacer bastante. Es la única forma de hacer cosas estables con algo tan flexible.
Uso asíncrono (I)	Prácticamente nulo. La escritura de funciones anónimas es tediosa por requerir el uso de "use" para acceder a variables del ámbito superior.	Las funciones anónimas permiten el acceso al ámbito superior. Escribir funciones a parte hace que seguir el flujo acabe siendo una tarea difícil.
Uso asíncrono (II)	No pensado para ello y con poco soporte.	Soporte maduro y multiplataforma, y pensado para ello.
Paquetes	pear, pecl	npm
Posibilidades de optimización a nivel de ejecución	Buscar variables, más engorroso pero más eficiente.	Tipado completamente dinámico, múltiples ámbitos en los que encontrar las variables.

Figura 4.13: Tabla con los rasgos principales de PHP y Node.JS.

4.5. Introducción al benchmarking

Un programa de prueba o benchmark se define como un programa o conjunto de programas que evalúan las prestaciones de un sistema informático reproduciendo una carga de trabajo genérica en dicho sistema informático. En general, para evaluar las prestaciones de un sistema informático es necesario conocer y caracterizar previamente cuál es la carga de trabajo. Sin embargo, en muchos casos tal carga no se conoce de antemano, es difícil de caracterizar o es suficientemente amplia como para considerarla una carga

genérica.

Mucha gente desconoce que al instalarse un servidor Apache no sólo pueden servir páginas, sino que gracias a ApacheBench también pueden medir el rendimiento del servidor.

En la tabla que vemos a continuación podemos encontrarnos con una de las pruebas más simples en el mundo de la programación que no es otra que una aplicación que permite imprimir una cadena de caracteres. Más tarde en el capítulo 9 se hará un benchmarking más completo de nuestros sistemas desarrollados.sistemas desarrollados.

Maquina	Dual-core Intel T4200 2 GHZ, 4 GB RAM
S.O.	Ubuntu 10.04
Versión Node.JS	Node.js 0.1.103
Versión PHP	Apache 2.2.14 / PHP 5.2.10
Tester	ApacheBench 2.3
Prueba 01	100.000 peticiones con 1.000 simultaneas
Prueba 02	1.000.000 peticiones con 2.000 simultaneas

Tabla 4.1: Tabla de las herramientas y tecnologías escogidas (Hardware).

Código Node.JS

```
var sys = require('sys'),
http = require('http');
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<p>Hello World</p>');
  res.end();
}).listen(8080);
```

Código PHP

```
<?php echo '<p>Hello World</p>';
```

4.5.1. Resultados prueba 01 Node.JS

```
Concurrency Level:      1000
Time taken for tests:   21.162 seconds
Complete requests:      100000
Failed requests:        147
   (Connect: 0, Receive: 49, Length: 49, Exceptions: 49)
Write errors:           0
Total transferred:      8096031 bytes
HTML transferred:       1799118 bytes
Requests per second:    4725.43 [#/sec] (mean)
Time per request:       211.621 [ms] (mean)
Time per request:       0.212 [ms] (mean, across all concurrent requests)
Transfer rate:          373.61 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  135 821.9      0    9003
Processing:     1   40 468.5     25   21003
Waiting:        1   30  64.1     25   12505
Total:          2  175 949.1     26   21003

Percentage of the requests served within a certain time (ms)
  50%     26
  66%     33
  75%     36
  80%     39
  90%     55
  95%     94
  98%   3030
  99%   3090
 100%  21003 (longest request)
```

4.5.2. Resultados prueba 01 PHP

```
Concurrency Level:      1000
Time taken for tests:   121.451 seconds
Complete requests:      100000
Failed requests:        879
   (Connect: 0, Receive: 156, Length: 567, Exceptions: 156)
Write errors:           0
Total transferred:      29338635 bytes
HTML transferred:       1889607 bytes
Requests per second:    823.38 [#/sec] (mean)
Time per request:       1214.510 [ms] (mean)
Time per request:       1.215 [ms] (mean, across all concurrent requests)
Transfer rate:          235.91 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   38 321.8     20    9032
Processing:     0  565 5631.0     51  121380
Waiting:        0  262 2324.1     41   52056
Total:         29  603 5641.7     73  121431

Percentage of the requests served within a certain time (ms)
  50%     73
  66%     78
  75%     82
  80%     83
  90%     89
  95%    105
  98%   4251
  99%  13205
 100%  121431 (longest request)
```

4.5.3. Comparación prueba 01

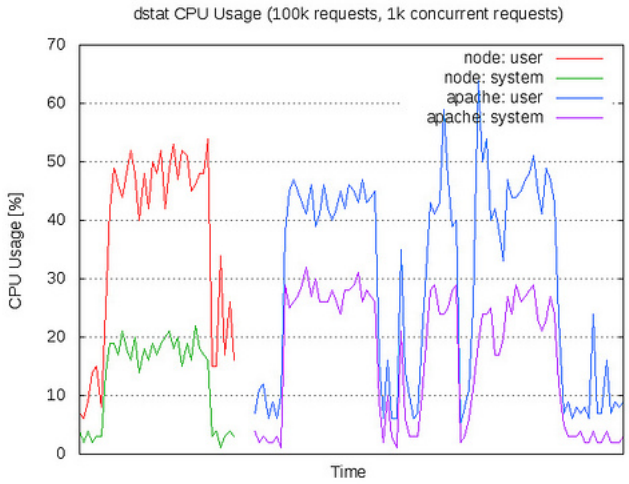


Figura 4.14: Grafica uso de la CPU prueba 01

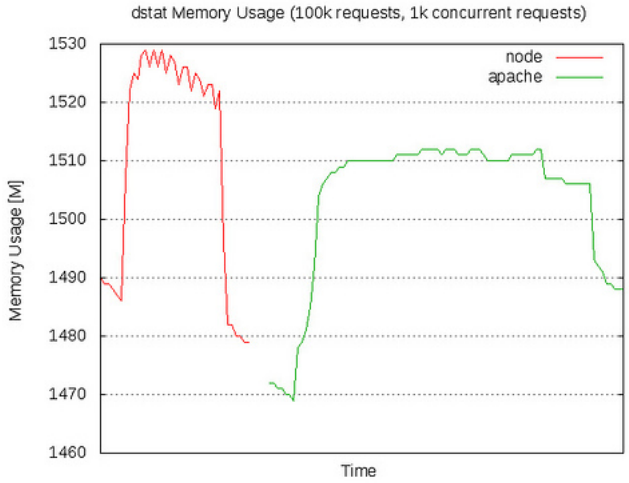


Figura 4.15: Grafica uso de memoria prueba 01

4.5.4. Resultados prueba 02 Node.JS

```
Concurrency Level:      20000
Time taken for tests:   1043.076 seconds
Complete requests:      1000000
Failed requests:        25227
   (Connect: 0, Receive: 8409, Length: 8409, Exceptions: 8409)
Write errors:           0
Total transferred:      81265680 bytes
HTML transferred:       18059040 bytes
Requests per second:    958.70 [#/sec] (mean)
Time per request:       20861.529 [ms] (mean)
Time per request:       1.043 [ms] (mean, across all concurrent requests)
Transfer rate:          76.08 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0 10201 2391.8  10840   20177
Processing:   595 10455 3239.1  10904   39809
Waiting:        0 8323 2331.0   8728   38740
Total:       1181 20656 4758.5  21795   44333

Percentage of the requests served within a certain time (ms)
  50%  21795
  66%  21929
  75%  22047
  80%  22135
  90%  22667
  95%  24252
  98%  24727
  99%  25942
 100%  44333 (longest request)
```

4.5.5. Resultados prueba 02 PHP

```
Concurrency Level:      20000
Time taken for tests:   3570.753 seconds
Complete requests:      1000000
Failed requests:        2617614
   (Connect: 0, Receive: 848121, Length: 886497, Exceptions: 882996)
Write errors:           0
Total transferred:      36832520 bytes
HTML transferred:       2372264 bytes
Requests per second:    280.05 [#/sec] (mean)
Time per request:       71415.058 [ms] (mean)
Time per request:       3.571 [ms] (mean, across all concurrent requests)
Transfer rate:          10.07 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0 4259 14734.0      0   79497
Processing:     4 64979 51442.2  65543  381910
Waiting:        0 2725 16784.2      0  249108
Total:         87 69238 56233.8  68138  426365

Percentage of the requests served within a certain time (ms)
  50%  68138
  66%  80099
  75%  84390
  80%  85475
  90%  91309
  95%  134983
  98%  303390
  99%  333308
 100%  426365 (longest request)
```

4.5.6. Comparación prueba 02

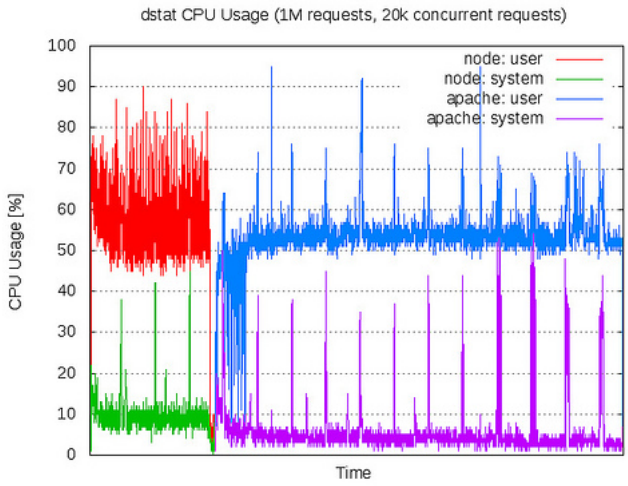


Figura 4.16: Grafica uso de la CPU prueba 02

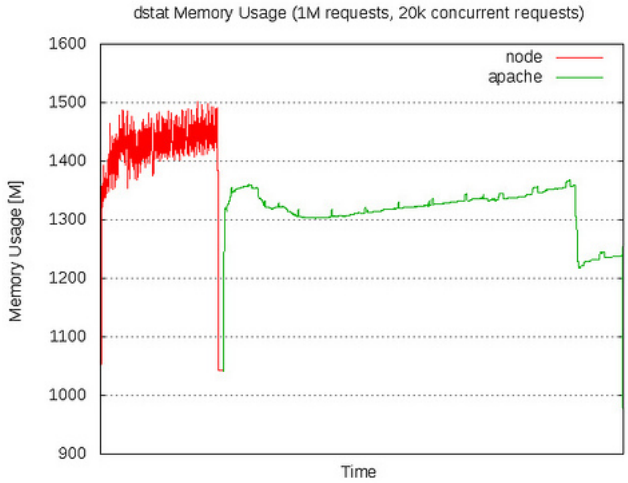


Figura 4.17: Grafica uso de memoria prueba 02

4.5.7. Valoración de los resultados

En lenguajes como PHP, cada conexión genera un nuevo hilo que potencialmente viene acompañado de 2 MB de memoria. En un sistema que tiene 8 GB de RAM, esto da un número máximo teórico de conexiones concurrentes de cerca de 4.000 usuarios. A medida que crece su base de clientes, si necesitas que la aplicación soporte más usuarios, necesitaras agregar más y más servidores. Por lo cual, el cuello de botella en toda la arquitectura de aplicación Web (incluyendo el rendimiento del tráfico, la velocidad de procesador y la velocidad de memoria) es el número máximo de conexiones concurrentes que puede manejar un servidor.

Node.JS resuelve este problema cambiando la forma en que se realiza una conexión con el servidor. En lugar de generar un nuevo hilo de OS para cada conexión (y de asignarle la memoria acompañante), cada conexión dispara una ejecución de evento dentro del proceso del motor de Node.JS.

4.6. Conclusión

Hemos visto como Apache crea un nuevo hilo por cada conexión cliente-servidor. Esto funciona bien para pocas conexiones, pero en los anteriores capítulos podemos ver que crear nuevos hilos tiene un coste, así como la realización de cambios de contexto. A partir de 400 conexiones simultáneas, el número de segundos para atender las peticiones crece considerablemente. Por lo que puedo concluir diciendo que Apache funciona bien, pero no es el mejor servidor para lograr máxima concurrencia.

Un servicio web como el que voy a construir en este proyecto que proporcione una API REST que toma algunos parámetros, los interpreta, arma una respuesta y descarga esa respuesta de vuelta al usuario es una situación ideal para Node.JS, ya que puede construirse para que maneje decenas de miles de conexiones. Tampoco requiere una gran cantidad de lógica y básicamente sólo busca valores de una base de datos y los reúne como una respuesta. Como la respuesta es una pequeña cantidad de texto y la solicitud entrante es una pequeña cantidad de texto, el volumen de tráfico no es alto, podría decir que el servidor que lo ejecute podrá soportar decenas de miles de conexiones concurrentes.

PHP es para aplicaciones más clásicas, donde hay páginas con mucho contenido o datos complejos o donde tienes que paginar muchos datos (es decir, aquellos casos donde hacer muchas pero muchas consultas asincrónicas no tiene mucho sentido).

Después de lo visto y teniendo en cuenta los casos de uso del sistema de este proyecto (más información en el capítulo 5) parece que un servidor Node.JS sería lo ideal, pero, para reforzar esta posición, en los siguientes capítulos voy a construir el sistema con las dos alternativas y finalizare el proyecto comparando los resultados de ambos modelos.

5. CAPÍTULO

Análisis de los requisitos

En este capítulo se tratará detalladamente la arquitectura del proyecto. En primer lugar se dará una explicación de las características que ha de tener, a continuación se explicará la arquitectura diseñada, se presentaran los diagramas de flujo, se detallarán los casos de uso, se definirán los diferentes tipos de acciones a tratar y por ultimo definiré el intercambio mensajes y la consistencia de la base de datos.

Análisis de los requisitos

1. Características desarrollo
2. Arquitectura del servidor
3. Capa de presentación
4. Capa de negocio
5. Capa de datos
6. Diagrama de comuniccaciones
7. Casos de uso
8. Lista de acciones
9. Intercambio de datos
10. Bases de datos

5.1. Características desarrollo

En este capítulo se abordaran las líneas generales que tiene que seguir el desarrollo del sistema.

- **Intuitivo**: los usuarios no deberán tener un perfil técnico para poder usar o integrar el sistema. Con una explicación relativamente simple y/o un manual, estarán en condiciones de usar todas sus funcionalidades.
- **Extensible**. El sistema será fácilmente extensible. Se podrán añadir funcionalidades nuevas de una manera sencilla.
- **Robusto**: el sistema será sólido y a prueba de errores. Durante el desarrollo se tendrán en cuenta todos los posibles casos de fallo para darles solución.

5.2. Arquitectura del sistema

En este apartado se dará a conocer la arquitectura que se llevara a cabo en el desarrollo del proyecto.

Voy a seguir el patrón MVC para la construcción del sistema. Este es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

En la capa de presentación, podremos simular diferentes acciones de lectura y escritura de tarjetas, también cabe comentar que las interfaces deberán ser usables, es decir, han de ser sencillas e intuitivas. Debido a los requisitos de la empresa y a la facilidad de integración con otros sistemas he decido gestionar los datos con un sistema de base de datos relacional, un SQL. ES tipo de sistemas permite combinar de forma eficiente diferentes tablas para extraer información relacionada, mientras que NoSQL no lo permite o lo permite muy limitadamente.

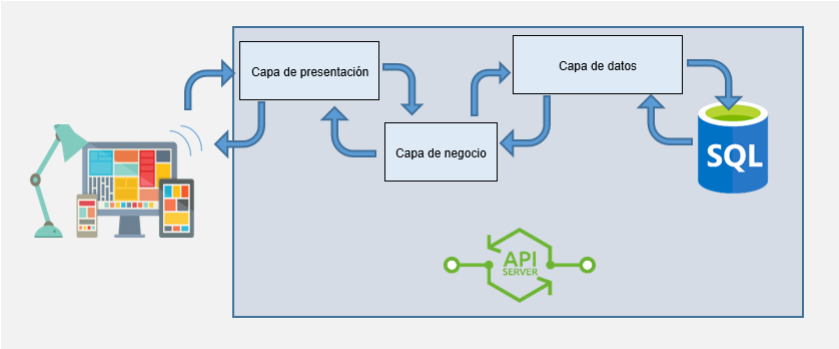


Figura 5.1: Modelo de arquitectura del servidor a desarrollar.

5.2.1. Labor del servidor

Cada x segundos el encoder se conecta con a una dirección URL para solicitar acciones. Si hay acciones pendiente, la respuesta del servidor serán los datos solicitados por el demandante, si no hay acciones pendientes simplemente responderá con un mensaje reflejando que no tiene trabajo para enviar. Una vez completada la acción (incluso si se ha completado con errores) el encoder enviara un mensaje al servidor para que este refleje la situación conveniente de cada acción.

5.2.2. Requisitos para la solución

1. El módulo ESP8266 con el firmware de ATELEI

2. Un servidor que siga el protocolo anteriormente descrito.

5.3. Diagramas de las comunicaciones

5.3.1. Login

En este primer diagrama se puede ver el flujo cuando el lector hace una petición de login contra el servidor.

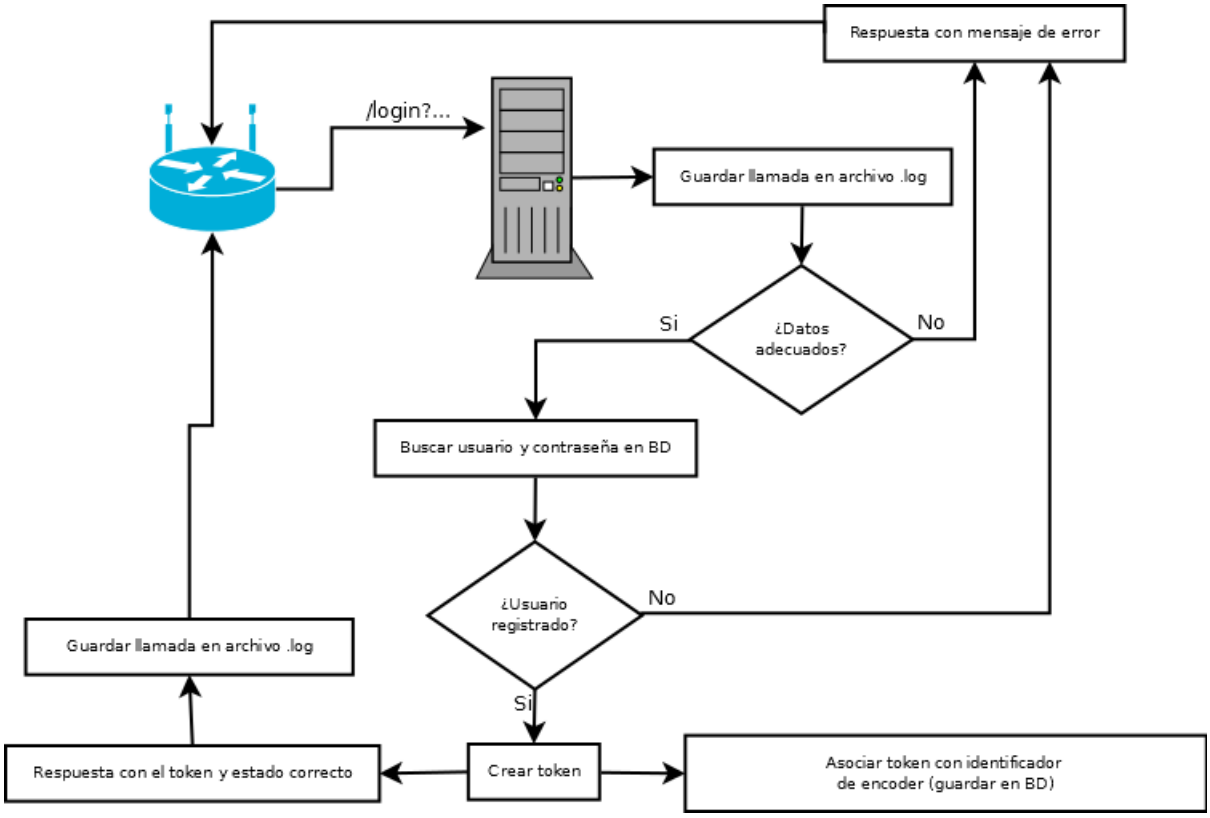


Figura 5.2: Diagrama de ejecución (interacción con login).

5.3.2. Execute

En el siguiente diagrama se puede observar el flujo habitual del lector de tarjetas, este flujo varía dependiendo el tipo de acción que recibimos desde el lector en los siguientes capítulos se profundice mas sobre los diferentes tipos de acciones.

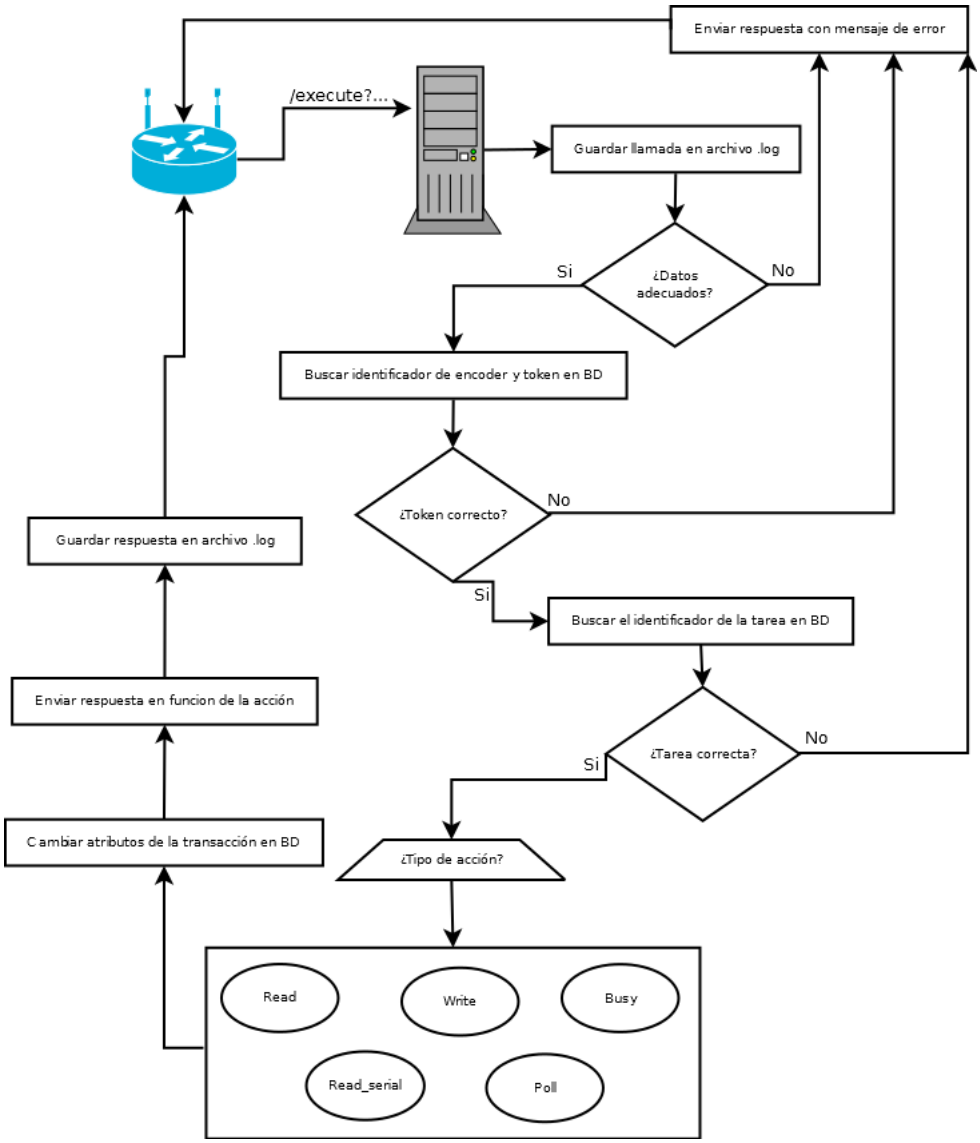


Figura 5.3: Diagrama de ejecución (interacción con execute).

5.4. Casos de uso

La imegen 5.4 nos muestra los casos de uso para nuestra aplicaciónnos muestra los casos de uso para nuestra aplicación.

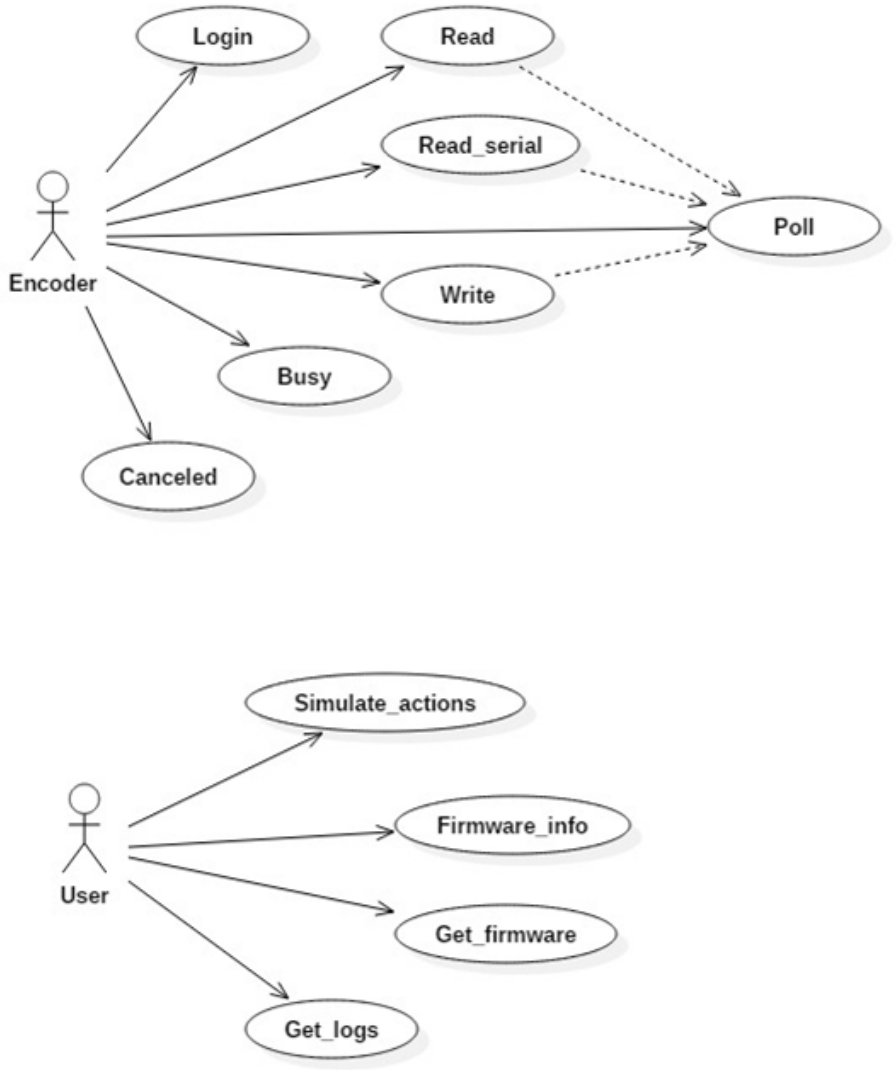


Figura 5.4: Casos de uso

5.4.1. Login

Objetivo:

Autenticarse con el servidor remoto.

Resumen:

Esta es la primera comunicación que debe hacer el encoder ya que para las demás acciones necesita un token que solo obtendrá al logearse. El encoder conoce previamente la dirección del servidor por lo cual solo necesitara un usuario y una contraseña determinada que le dará acceso al mencionado token. Cuando el encoder le manda los datos el servidor le responderá con un token si los datos son correctos o con un código de error si los datos para la conexión son incorrectos.

Flujo corriente:

1. El encoder accede a la url login del servidor con un usuario y contraseña válidos.
2. El servidor guarda en base de datos el token y el usuario.
3. El servidor le responde con el token.

Flujo alternativo:

1. El encoder accede a la url login del servidor con un usuario y contraseña inválidos.
2. El servidor le responde con un mensaje que refleja que la autenticación ha fallado.

Flujo erróneo:

1. El encoder accede a la url login del servidor con un usuario y contraseña válidos.
2. El servidor le responde con un mensaje que refleja que algo ha fallado.
3. El encoder reintentara logearse, volviendo a enviar el primer mensaje.

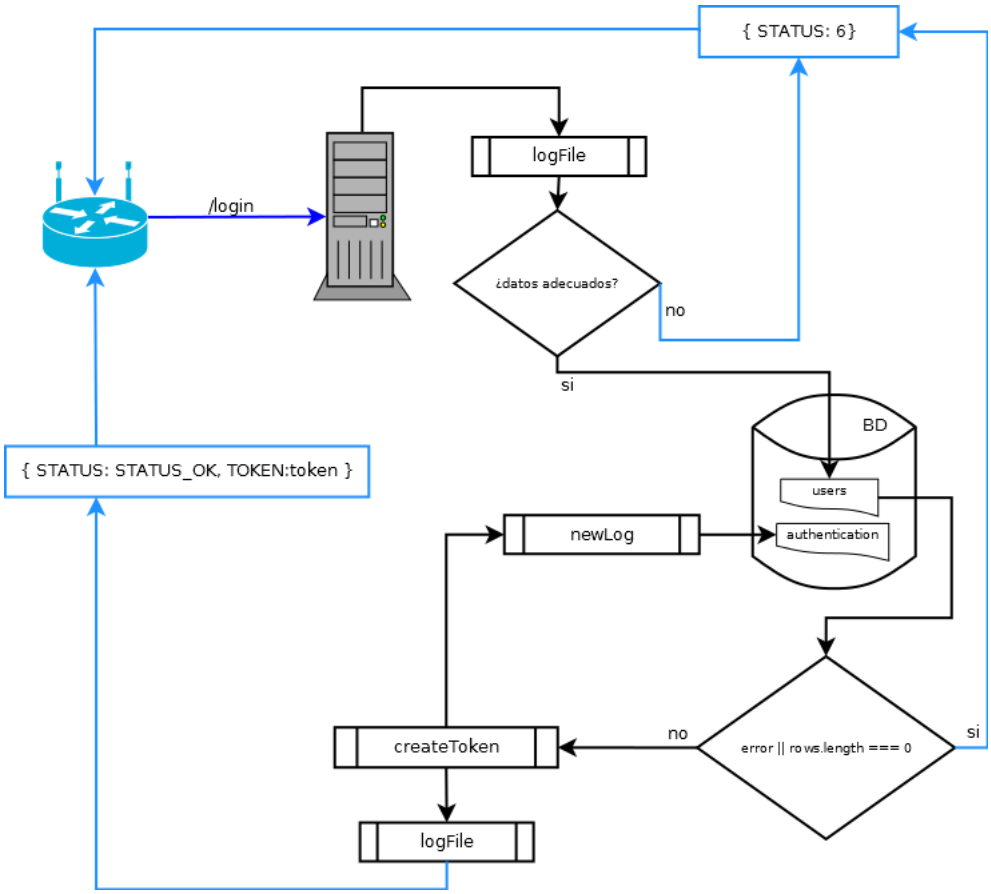


Figura 5.5: Diagrama login.

Ejemplos:

Las siguientes tablas muestran la estructura básica para la autenticación (login):

Nombre	Tipo	Descripción
USER	String	Usuario para autenticarse en el servidor.
PASSWORD	String	Usuario para autenticarse en el servidor.
IDENCODER	String	Identificador único del encoder.

Tabla 5.1: Tabla de envios desde el encoder. (login)

Nombre	Tipo	Descripción
STATUS	String	Estado del login (error / correcto)
TOKEN	String	Token generado en el servidor utilizado para guardar la sesión con el encoder.

Tabla 5.2: Tabla de respuestas del servidor. (login)

Paso	Origen	Mensaje
1	Encoder	http://server/login?USER=correctUser& PASSWORD =correct-Password& IDENCODER=correctIDEncoder
2	Server	STATUS=1; TOKEN=”validToken”
3	Encoder	*Ahora, el encoder empieza a pedir tareas.*

Tabla 5.3: Tabla de flujo de un login correcto.

Paso	Origen	Mensaje
1	Encoder	http://server/login?USER=correctUser& PASSWORD =incorrectPassword& IDENCODER=correctIDEncoder
2	Server	STATUS=6; TOKEN=""

Tabla 5.4: Tabla de flujo de un login incorrecto.

Paso	Origen	Mensaje
1	Encoder	http://server/login?USER=correctUser& PASSWORD =correctPassword& IDENCODER=correctIDEncoder
2	Server	*El servidor no responde*
3	Encoder	http://server/login?USER=correctUser& PASSWORD =iorrectPassword& IDENCODER=correctIDEncoder

Tabla 5.5: Tabla de flujo de un login timeout.

5.4.2. Read

Objetivo:

Obtener los datos de una tarjeta

Resumen:

Cuando el servidor tiene la tarea de leer una tarjeta le enviara la acción de leer al encoder, el usuario posa la tarjeta en el encoder y este le envía el dato que ha leído al servidor.

Flujo corriente:

1. 1. El encoder pregunta al servidor
2. El servidor le responde con la acción de leer tarjeta. Además guarda la acción como enviada (intentos - 1)
3. El encoder recibe la acción e intenta leer una tarjeta
4. El encoder le envía el resultado de la lectura anterior (OK o ERROR_CODE)
5. El servidor actúa en consecuencia a los parámetros de entrada, reenvía la acción anterior o pasa a la siguiente.

Flujo alternativo:

1. El encoder pregunta al servidor
2. El servidor le responde con la acción de leer tarjeta. Además guarda la acción como enviada (intentos - 1)
3. El encoder no responde.
4. La acción se queda en estado enviado a la espera que reciba alguna comunicación con el identificador de la acción enviada en un principio.

Flujo erróneo:

1. El servidor le envía un mensaje de error al encoder.

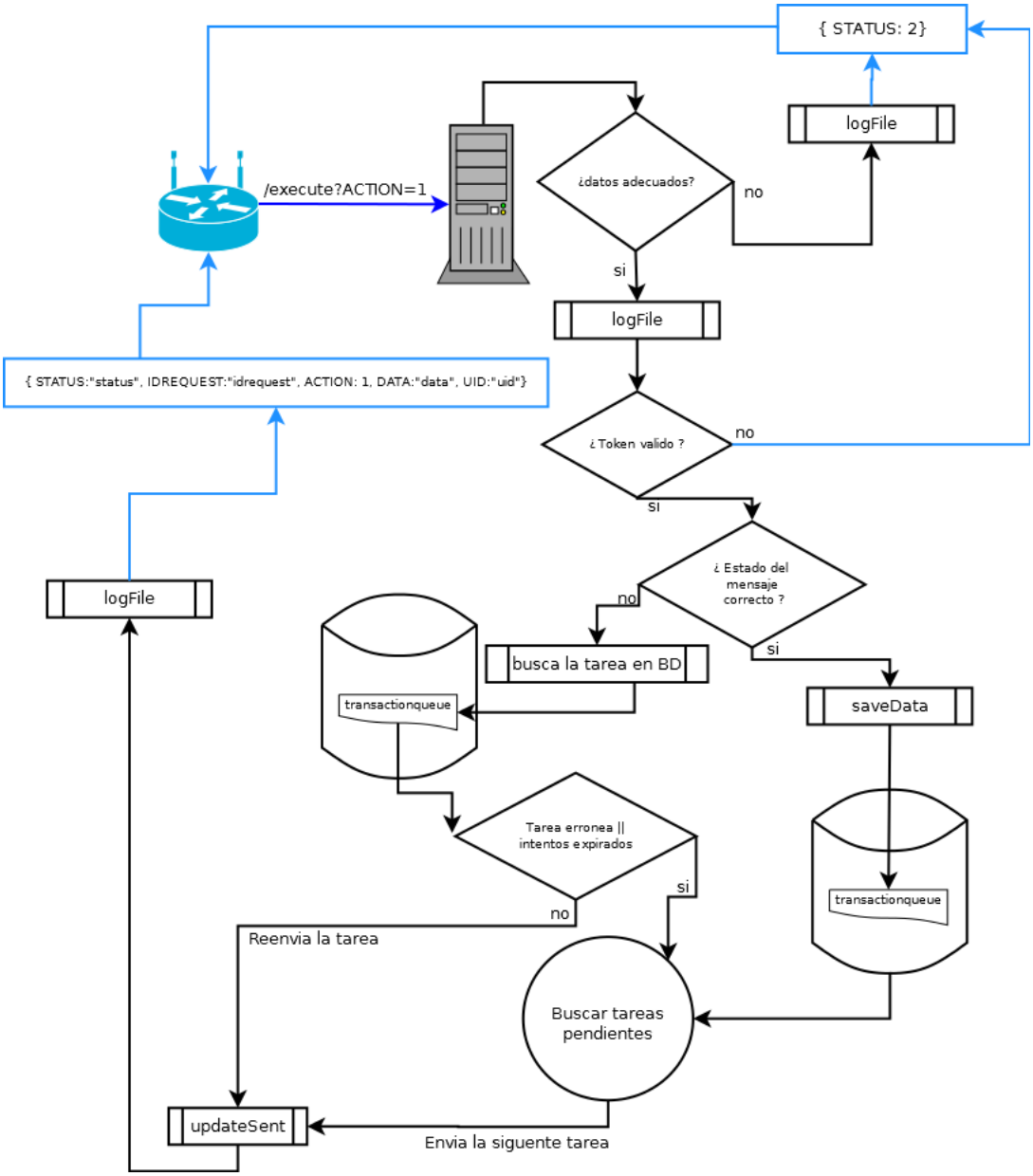


Figura 5.6: Diagrama read.

Ejemplos:

Las siguientes tablas muestran la estructura básica para la lectura de tarjetas (read):

Nombre	Tipo	Descripción
IDENCODER	String	Identificador único del encoder.
TOKEN	String	Token generado en el servidor para identificar la sesión del encoder.
ACTION	Int	Acción a ejecutar.
STATUS	Int	Estado de la tarea actual. (OK, ERROR, BUSY...).
IDREQUEST	Int	Identificador de la tarea (asignada en el servidor).
DATA	String	Dato para la lectura o escritura de tarjeta.

Tabla 5.6: Tabla de envios desde el encoder (read).

Nombre	Tipo	Descripción
STATUS	Int	Estado de la tarea actual(OK, ERROR, BUSY...).
ACTION	String	Acciona a ejecutar.
IDREQUEST	Int	Identificador único del encoder.
DATA	String	Dato para la lectura o escritura de tarjeta.

Tabla 5.7: Tabla de respuestas del servidor.

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=1; ACTION=1; DATA=””
	Encoder	*...leyendo...*
3	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=1& DATA=”FFAB113423”& STATUS=1
4	Server	STATUS=1; ACTION=4;

Tabla 5.8: Tabla de flujo de lectura satisfactoria .

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=2; ACTION=1; DATA=””
	Encoder	*. . . Intento de lectura fallido (timeout). . .*
3	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=2& DATA=””& STATUS=7
	Server	*Ahora el servidor decide cual es la tarea a enviar*

Tabla 5.9: Tabla de flujo de lectura errónea.

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”invalidToken”& ACTION=4
	Server	*Servidor detecta token erróneo (expirado o incorrecto)*
2	Server	STATUS=2;
	Encoder	*Encoder vuelve a intentar logearse*

Tabla 5.10: Tabla de flujo de lectura con token invalido.

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=1; ACTION=1; DATA=””
3	Encoder	*(Mientras este leyendo el encoder notificara una acción tipo BUSY)* https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=6& IDREQUEST=1
4	Server	STATUS=1; IDREQUEST=1; ACTION=6;
5	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=1& DATA=’FFAB113423’& STATUS=1
6	Server	STATUS=1; IDREQUEST=2; ACTION=1; DATA=””
7	Encoder	*(Mientras este leyendo el encoder notificara una acción tipo BUSY)* https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=6& IDREQUEST=2
8	Server	STATUS=1; IDREQUEST=2; ACTION=6;
9	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=2& DATA=”FFAB113423”& STATUS=1
10	Server	STATUS=1; IDREQUEST=3; ACTION=1; DATA=””
11	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=3& DATA=”FFAB113423”& STATUS=1
12	Server	STATUS=1; ACTION=4;

Tabla 5.11: Tabla de flujo de lectura con token invalido.

5.4.3. Write

Objetivo:

Escribir los datos almacenados en el servidor en una tarjeta.

Resumen:

Cuando el servidor tiene la tarea de escribir una tarjeta, este le enviara la acción de escribir y el dato que quiere escribir al encoder, el usuario posa la tarjeta en el encoder y este le envía la confirmación de que todo ha salido bien al servidor.

Flujo corriente:

1. El encoder pregunta al servidor
2. El servidor le responde con la acción de escribir tarjeta y el dato a escribir. Además guarda la acción como enviada (intentos - 1)
3. El encoder recibe la acción e intenta leer y escribir la tarjeta
4. El encoder le envía el resultado de la escritura anterior (OK o ERROR-CODE)
5. El servidor actúa en consecuencia a los parámetros de entrada, reenvía la acción anterior o pasa a la siguiente.

Flujo alternativo:

1. El encoder pregunta al servidor
2. El servidor le responde con la acción de escribir tarjeta y el dato a escribir. Además guarda la acción como enviada (intentos - 1)
3. El encoder no responde.
4. La acción se queda en estado enviado a la espera que reciba alguna comunicación con el identificador de la acción enviada en un principio.

Flujo erróneo:

1. El servidor envía un mensaje de error al encoder.

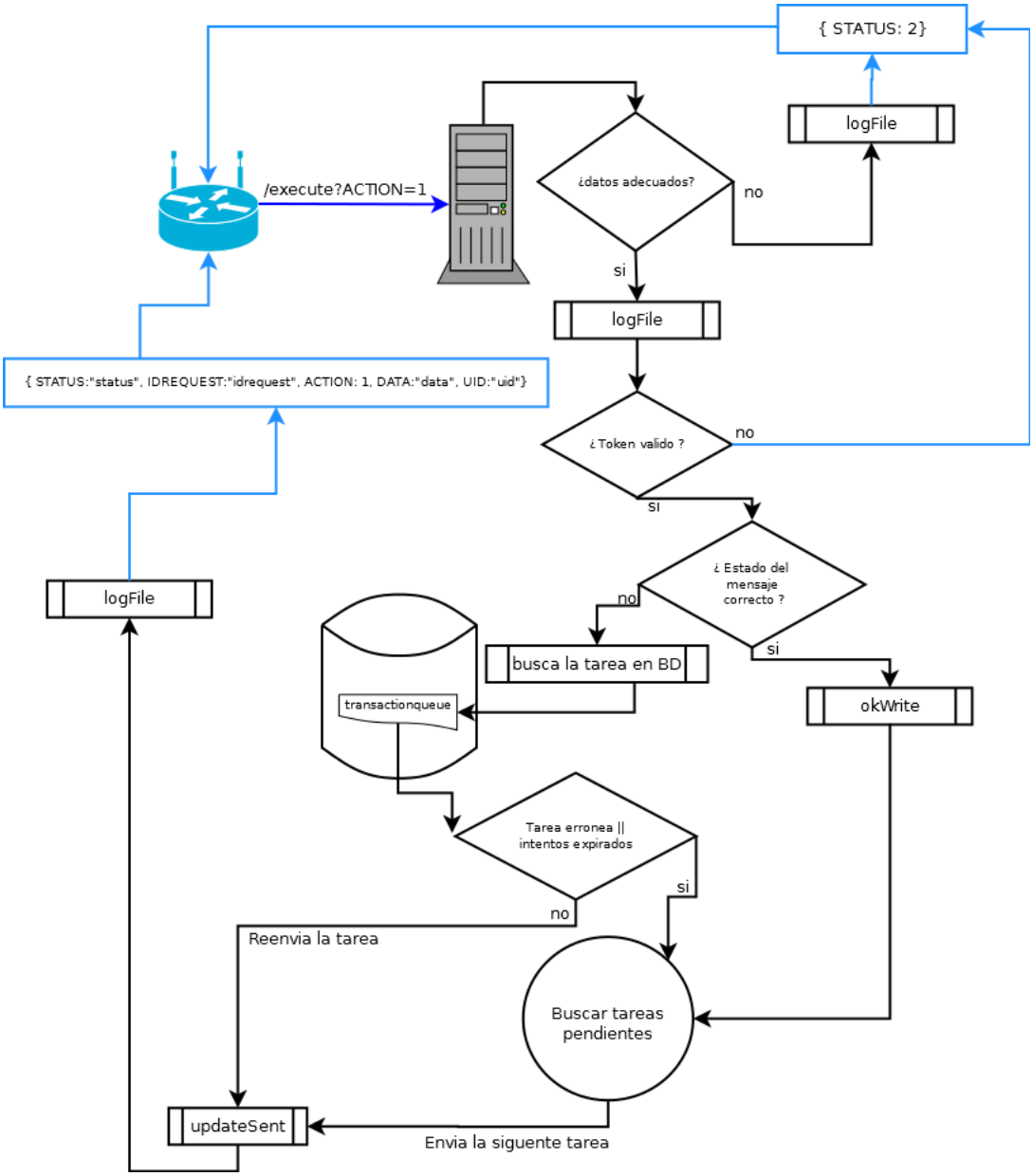


Figura 5.7: Diagrama write.

Ejemplos:

Las siguientes tablas muestran la estructura básica para la lectura de tarjetas (write):

Nombre	Tipo	Descripción
IDENCODER	String	Identificador único del encoder.
TOKEN	String	Token generado en el servidor para identificar la sesión del encoder.
ACTION	Int	Acción a ejecutar.
STATUS	Int	Estado de la tarea actual. (OK, ERROR, BUSY...).
IDREQUEST	Int	Identificador de la tarea (asignada en el servidor).
DATA	String	Dato para la lectura o escritura de tarjeta.

Tabla 5.12: Tabla de envios desde el encoder (write).

Nombre	Tipo	Descripción
STATUS	Int	Estado de la tarea actual. (OK, ERROR, BUSY...).
ACTION	String	Acciona a ejecutar.
IDREQUEST	Int	Identificador único del encoder.
DATA	String	Dato para la lectura o escritura de tarjeta.

Tabla 5.13: Tabla de respuestas del servidor(write) .

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=1; ACTION=1; DATA=” FFAB113423”
	Encoder	... *escribiendo...*
3	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=1& DATA=”FFAB113423”& STATUS=1
4	Server	STATUS=1; ACTION=4;

Tabla 5.14: Tabla de flujo de escritura satisfactoria .

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=2; ACTION=1; DATA=” FFAB113423”
	Encoder	*. . . Intento de escritura fallido (timeout). . .*
3	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=1& IDREQUEST=2& DATA=””& STATUS=7
	Server	*Ahora el servidor decide cual es la tarea a enviar*

Tabla 5.15: Tabla de flujo de escritura errónea.

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”invalidToken”& ACTION=4
2	Server	STATUS=2;
	Encoder	*Intenta logearse de nuevo*

Tabla 5.16: Tabla de flujo de escritura con token inválido.

5.4.4. Read_Serial

Objetivo:

Obtener el número de serie de una tarjeta

Resumen:

Cuando el servidor tiene la tarea de leer un número de serie le enviara la acción de leer serial al encoder, el usuario posa la tarjeta en el encoder y este le envía el dato que ha leído al servidor.

Flujo corriente:

1. El encoder pregunta al servidor
2. El servidor le responde con la acción de leer serial. Además guarda la acción como enviada (intentos - 1)
3. El encoder recibe la acción e intenta leer una tarjeta
4. El encoder le envía el resultado de la lectura anterior (OK / ERROR_CODE + número de serie)
5. El servidor actúa en consecuencia a los parámetros de entrada, reenvía la acción anterior o pasa a la siguiente.

Flujo alternativo:

1. 1. El encoder pregunta al servidor
2. El servidor le responde con la acción de leer serial. Además guarda la acción como enviada (intentos - 1)
3. El encoder no responde.
4. La acción se queda en estado enviado a la espera que reciba alguna comunicación con el identificador de la acción enviada en un principio.

Flujo erróneo:

1. El servidor envía un mensaje de error al encoder.

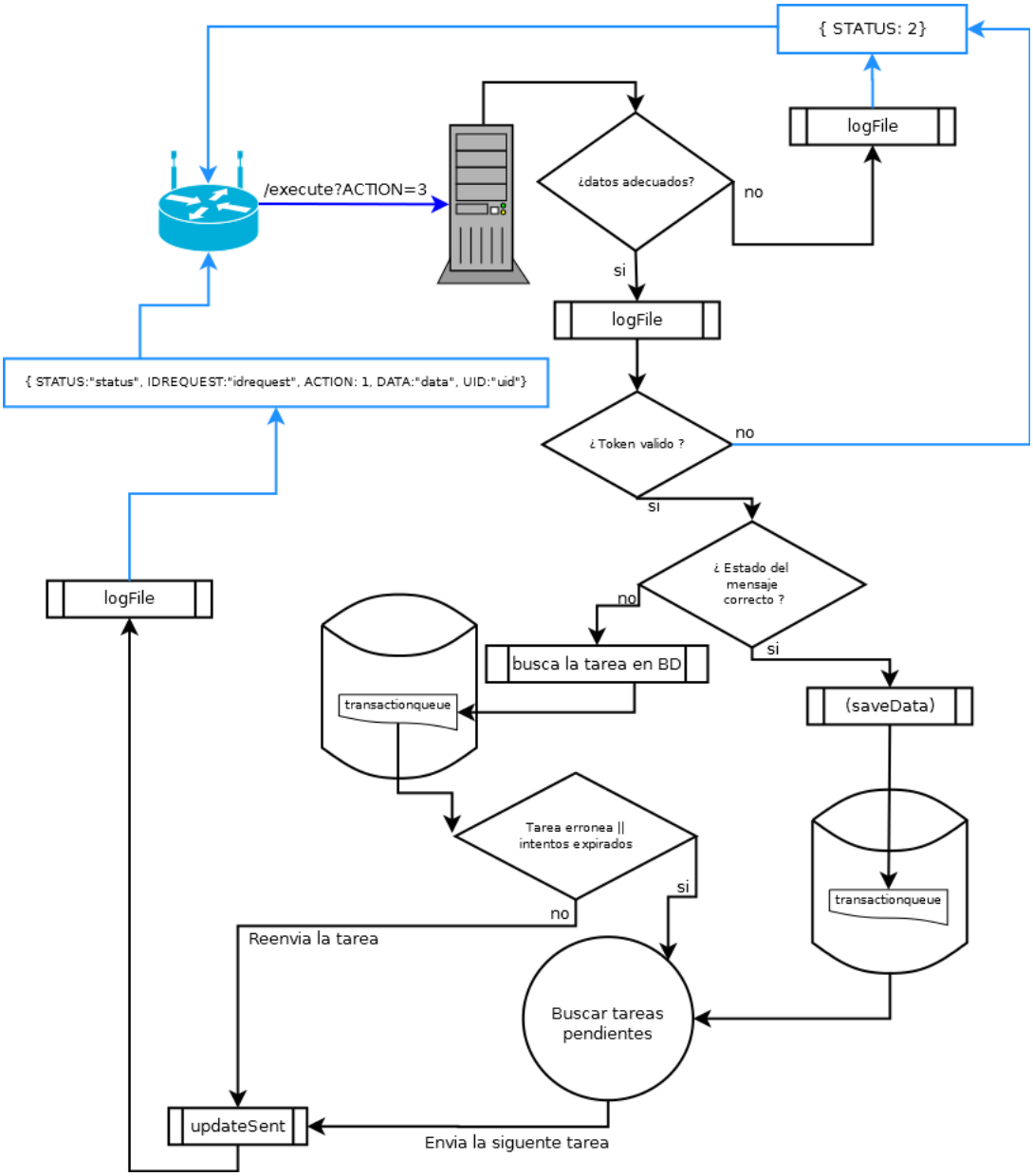


Figura 5.8: Diagrama Read_Serial.

Ejemplos:

Existen dos modos distintos de leer un número de serie de una tarjeta. Una manera es poner una tarjeta en el encoder mientras que el encoder está pidiendo tareas. La otra forma es enviar desde el servidor la acción de leer un número de serie. Estos son los dos ejemplos:

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=3& DATA=”FFAA121244”
2	Server	*(El servidor decidirá la acción a enviar cuando le llega un número de serie)* STATUS=1; ACTION=4;

Tabla 5.17: Tabla de flujo al posar tarjeta en encoder.

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=1; ACTION=3; DATA=””
3	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=3& DATA=”FFAB113423”
4	Server	STATUS=1; ACTION=4;

Tabla 5.18: Tabla de flujo petición de lectura de serial desde el servidor.

5.4.5. Poll

Objetivo:

El encoder cada x segundos hace una petición al caso de uso poll para preguntar si el servidor tiene tareas pendientes para él.

Resumen:

Este caso de uso es el más común en el flujo de comunicaciones. El encoder le pregunta al servidor a ver si tiene alguna tarea, el servidor dependiendo de la base de datos le enviara la tarea que más tiempo lleve guardada y que este en estado pendiente o le señalara que no tiene tareas pendientes.

Flujo corriente:

1. El encoder pregunta al servidor
2. El servidor le responde con la acción que más tiempo lleva en pendiente.
3. El encoder le envía el resultado de la acción enviada.
4. El servidor actúa en consecuencia a los parámetros de entrada, reenvía la acción anterior o pasa a la siguiente.

Flujo alternativo:

1. 1. El encoder pregunta al servidor
2. El servidor le responde con la acción que más tiempo lleva en pendiente.
3. El encoder no responde.
4. La tarea se queda en estado enviado a la espera que reciba alguna comunicación con el identificador de la acción enviada en un principio.

Flujo erróneo:

1. El servidor le envía un mensaje de error al encoder.

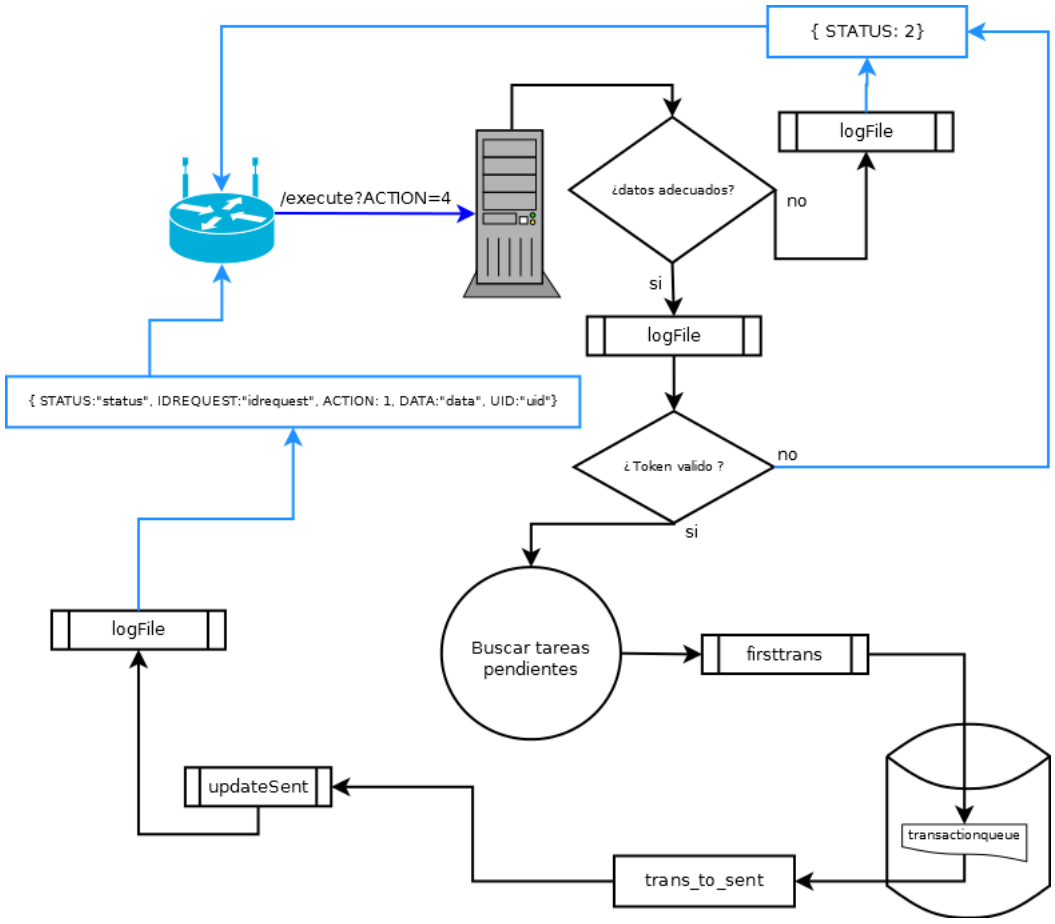


Figura 5.9: Diagrama poll.

Ejemplo:

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=4
2	Server	STATUS=1; IDREQUEST=1; ACTION=1; DATA=” FFAB113423”

Tabla 5.19: Tabla de flujo al hacer una petición de poll.

5.4.6. Busy

Objetivo:

El encoder le manda una acción tipo Busy cuando el servidor le manda alguna nueva tarea y el encoder sigue ocupado en alguna transacción anterior.

Resumen:

El encoder le envía al servidor una transacción con la acción Busy, lo que significa que está en proceso de completarla. El servidor dependiendo del estado de esa transacción le responderá con la transacción correspondiente.

Flujo corriente:

1. El encoder le envía una transacción con la acción en Busy
2. El servidor le responde con la acción que más tiempo lleva en pendiente o con la misma acción que recibió dependiendo la consist

encia en la base de datos del servidor.

Flujo erróneo:

1. El servidor le envía un mensaje de error al encoder.

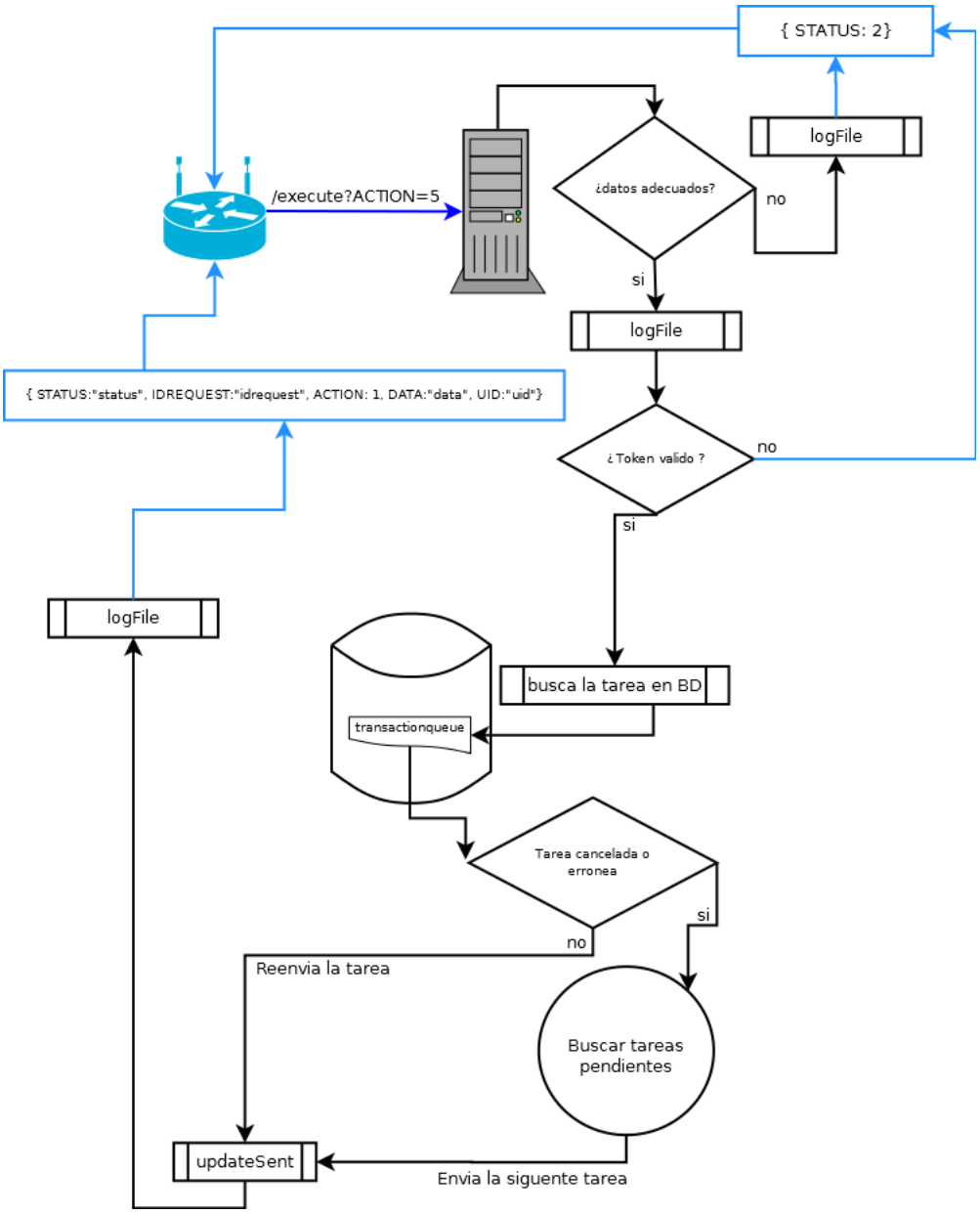


Figura 5.10: Diagrama busy.

Ejemplo:

Paso	Origen	Mensaje
1	Encoder	https://server/execute?IDENCODER=idEncoder& TOKEN=”token”& ACTION=5& DATA=”FFAB113423”
2	Server	STATUS=1; IDREQUEST=1; ACTION=1; DATA=””

Tabla 5.20: Tabla de flujo al hacer una petición busy.

5.5. Lista de acciones

El servidor soportará los siguientes seis tipos de diferentes acciones:

Id (ACTION)	Nombre	Descripción
1	RD_CARD	Instrucción para leer una tarjeta
2	WR_CARD	Instrucción para escribir una tarjeta
3	RD_SERIAL	Instrucción para leer el número de serie de una tarjeta
4	POLL	Instrucción para preguntar al servidor por tareas. También sirve para que el servidor notifique que no tiene nada pendiente.
5	BUSY	Instrucción que usa el encoder para notificar que está ocupado en alguna acción.
6	CANCELED	Instrucción que usa el servidor para cancelar la acción del encoder.

Tabla 5.21: Tabla de las diferentes acciones soportadas por el servidor.

5.6. Intercambio de datos

Después de analizar los diferentes casos de uso podemos ver que la API se base en el intercambio de mensajes. Tenemos varias formas de intercambiar mensajes, debido a las especificaciones del módulo ESP8266 se decidío que seria mediante diferentes tipos de mensajes JSON.

He diferenciado 3 tipos de flujo generales; el primero es el de login, esto sucede cuando el encoder intenta logearse en el servidor. El segundo, el de execute cuando el encoder y el servidor intercambian acciones. Y por último el de actions, que es el que no da acceso a la interfaz para simular acciones.

Recordemos que llamamos transacciones al las tareas que queremos trasladarle al encoder desde el servidor.

5.6.1. Situaciones de las transacciones (BD)

Anteriormente en el flujo de los casos de uso hemos visto que las transacciones pueden pasar por varios estados, en el sistema desarrollado se han definido los siguientes estados para cada transacción.

Id (SITUATION)	Nombre	Descripción
1	PENDING	Tarea pendiente
2	SENT	Tarea enviada al encoder
3	NO_OK	Tarea fallida
4	OKAY	Tarea finalizada satisfactoriamente
5	CANCEL	Tarea cancelada

Tabla 5.22: Tabla de los estados de las tareas (consistencia en base de datos)

Estos estados se irán guardando en la base de datos en forma de Integer y cambiando en la tabla *transactionqueue* sobre la variable SITUATION dependiendo del estado de la tarea. Ejemplo de un flujo estándar:

1. Cuando el usuario crea las diferentes peticiones se guarda SITUATION=1 para que quede reflejado que son transacciones pendientes.
2. Al enviar la transacción se cambia la SITUATION=2
3. Si todo va bien y el encoder responde de forma correcta SITUATION=4 de lo contrario SITUATION=3
4. Por ultimo podemos dejar las transacciones en SITUATION=5 cuando queremos cancelar esa tarea.

5.6.2. Estados de los mensajes (JSON)

Como ya he mencionado, las transacciones que están almacenadas en el servidor se envían mediante mensajes JSON al encoder. Estos mensajes irán acompañados de unos estados para definir mejor el flujo de comunicaciones. Los siguientes estados se definen en los mensajes entre el servidor y el cliente:

Id (SITUATION)	Nombre	Descripción
1	STATUS_OK	Mensaje correcto.
2	NOK_WRONG_TOKEN	El token recibido es incorrecto (expirado o invalido)
3	UNKNOWN_COMMAND	El comando recibido es ilegible para el servidor.
4	WRONG_ENCODER_KEY	La clave almacenada en el codificador no está funcionando en la tarjeta actual.
5	CARD_TIMEOUT	Notificación sobre que no han posado tarjetas en el encoder.
6	LOGIN_INCORRECT	Validación incorrecta de usuario o contraseña
7	ERROR_READ	La acción de lectura ha finalizado con errores.
8	ERROR_WRITE	La acción de escritura ha finalizado con errores.

Tabla 5.23: Tabla de estados de las transacciones

Los estados se almacenaran en forma de Integer en cada envío realizado en la variable STATUS del JSON a enviar.

5.7. Base de datos

Recordemos que el objetivo principal de este proyecto es crear un sistema de comunicación con un Smart-reader via Wi-Fi. Para la consistencia de ese objetivo tenderemos las siguientes tres tablas en la base de datos:

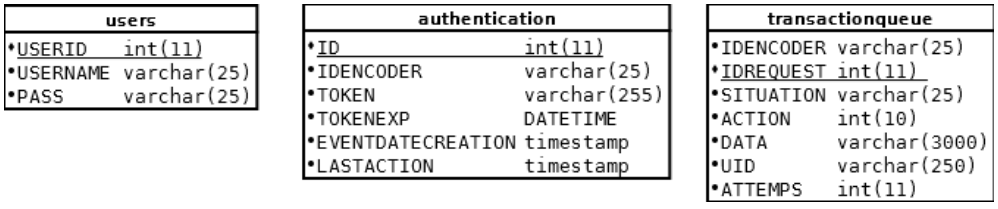


Figura 5.11: Esquema base de datos.

5.7.1. Tabla users

Esta tabla como su propio nombre indica será la que gestiona los usuarios. Cuando el encoder intenta logearse en el sistema comprobaremos la consistencia del usuario y la contraseña mediante esta tabla.

USERID	USERNAME	PASS
1	admin	admin

Figura 5.12: Tabla *users*.

5.7.2. Tabla authentication

Cuando un encoder se logea en el sistema tiene que quedar constancia de ello ya que las siguientes comunicaciones irán autenticadas con el token que le proporciona el servidor. Para eso nos ayudaremos de esta tabla.

ID	IDENCODER	TOKEN	TOKENEXP	EVENTDATECREATION Date and time of record creation	LASTACTION
1	MAC	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE0N...	2016-06-28 12:07:14	2016-06-28 12:06:14	2016-06-28 12:06:14
2	MAC	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE0N...	2017-06-28 12:10:57	2016-06-28 12:10:57	2016-06-28 12:10:57
3	MAC	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE0N...	2016-06-28 12:07:14	2016-06-28 12:06:14	2016-06-28 12:06:14

Figura 5.13: Tabla *authentication*.

5.7.3. Tabla transactionqueue

Tabla donde se guarda el listado de las tareas.

IDENCODER	IDREQUEST	SITUATION	ACTION	DATA	UID	ATTEMPS	EVENTDATECREATION Date and time of record creation
MAC	1201	1	1			10	2016-10-20 19:46:31
MAC	1200	1	1			10	2016-10-20 19:46:31
MAC	1199	1	1			10	2016-10-20 19:46:31

Figura 5.14: Tabla *transactionqueue*.

6. CAPÍTULO

Diseño y desarrollo en Node.JS

En este apartado se detallarán los temas concernientes al diseño del sistema con Node.JS. Por un lado, se hará hincapié en la arquitectura definida más allá de la división por capas. Más adelante, veremos cuál ha sido la elección del motor de plantillas para generar la parte front-end de este proyecto. Por otro lado, se hablará del desarrollo del proyecto y las decisiones que se han ido tomando a medida que iba avanzando el proceso. Se hará una pequeña introducción sobre los módulos de Node.JS, para más adelante ir viendo que modulo conviene para los diferentes funcionamientos. Finalmente se describirán las diferentes implementaciones.

Diseño y desarrollo en Node.JS

1. Arquitectura
2. Elección de motor de plantillas
3. Módulos
4. Implementación
5. Instalación

6.1. Arquitectura

Para realizar el sistema, me he basado en la estructura cliente-servidor, ya que es la que mejor se adapta a las necesidades: un servidor que recibe peticiones y se las sirve al cliente. Anteriormente hemos visto una introducción al tipo de arquitectura escogida, se implementara una división por capas, teniendo así una capa de presentación, una capa de lógica de negocio y una capa de datos. En ese capítulo hará una introducción a cada una de ellas.

6.1.1. Capa de presentación y lógica

Como se ha explicado previamente la capa de presentación es aquella que interactúa con el usuario, enseña en pantalla o envía la información que ha recibido desde la capa inferior y recibe e interpreta las ordenes que ejecuta el usuario. Debido a la simplicidad de Node.JS y a que todas las peticiones de los agentes serán rutas HTTP, en este la capa de presentación y la lógica de negocio irán mergeadas en un mismo archivo donde se recogerá la ruta y se tomara la acción a aplicar.

Figura 6.1: Esquema capa de presentación

Por un lado van a haber dos interfaces web para el usuario. Una donde podrá crear peticiones para el encoder y otra donde podrá ver las peticiones creadas. Las interfaces son páginas web estáticas construidas con la ayuda de boostrap y handelbars (actions.hbs y transactions.hbs).

Por otro lado tendremos los controladores que decidirán qué hacer con las peticiones recibidas. Se reciben las peticiones del usuario y se envían las respuestas tras el proceso, es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos almacenar o recuperar datos de él.

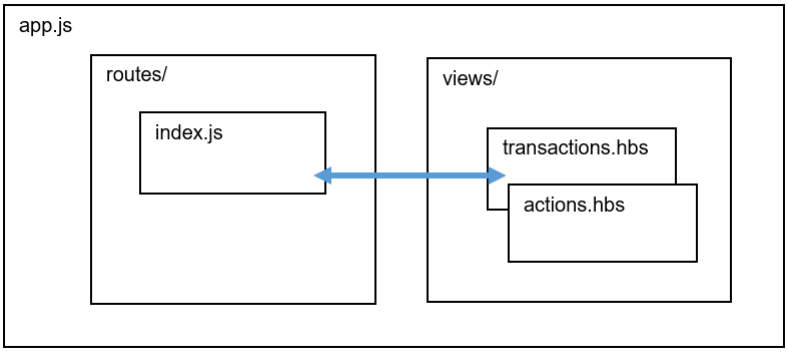


Figura 6.2: Modelo de la capa de presentación y lógica para Node.JS

6.1.2. Capa de datos

Se encarga de la persistencia, es decir, recibe la información de la capa de lógica y almacena esos datos en una base de datos. En esta capa se definirán todas las acciones tanto de entrada como de salida en la base de datos.

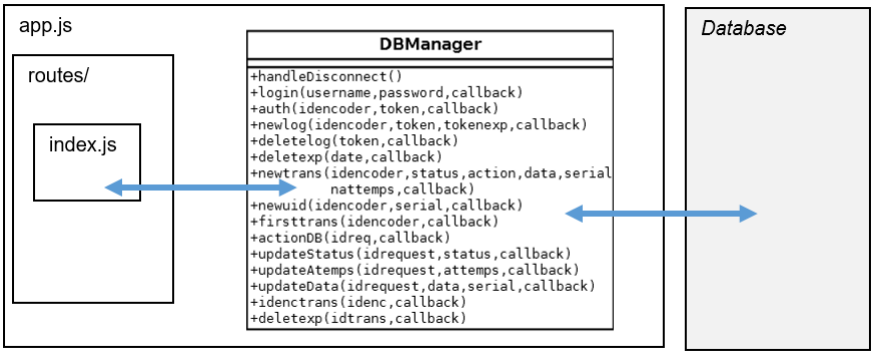


Figura 6.3: Modelo de la capa de datos para Node.JS

6.2. Elección de motor de plantillas

Toda aplicación web necesita generar páginas dinámicamente. La mayoría utiliza el propio PHP del servidor, que permite inserción de código HTML directamente.

Igual que en el resto de lenguajes de programación, Node.JS dispone de módulos para reducir la complejidad de los programas y simplificar el código principal. Este entorno incorpora varios "módulos básicosçompilados en el propio binario, como por ejemplo el módulo de red, que proporciona una capa para programación de red asíncrona y otros módulos fundamentales, como por ejemplo Path, FileSystem, Buffer, Stream, etc. Algunos módulos desarrollados por terceros los proporciona la página oficial de Node.JS. Una fuente importante de estos módulos es GitHub, una página donde la comunidad de programadores puede publicar código abierto. Los módulos pueden ser archivos ".node"precompilados, o archivos en JavaScript plano. Los módulos JavaScript se implementan siguiendo la especificación CommonJS, utilizando una variable de exportación para dar a estos scripts acceso a funciones y variables. Pueden extender Node.JS o añadir un nivel de abstracción, implementando varias utilidades middleware para utilizar en aplicaciones web. Los módulos pueden instalarse como archivos simples o utilizando el Node.JS Package Manager (npm), utilidad que facilita la instalación, actualiza y resuelve dependencias de forma automática. Estos módulos se instalan en un directorio por defecto, si un programa necesita un módulo que no esté en ese directorio, se necesitará indicar una ruta en el programa para poder usarlos.

Puesto que nosotros no vamos a utilizar PHP en el servidor, necesitamos un motor de plantillas que se adapte a nuestras necesidades, es decir, que sea dinámico y totalmente compatible con la arquitectura de nuestra aplicación.

Después de un pequeño análisis deduje que la mejor solución era dejar de lado incluso el HTML, y utilizar un motor llamado handelbars, el cual genera las plantillas desde el servidor y las envía al cliente en formato HTML.

Handelbars es un lenguaje de plantillas desarrollado por el creador de Express para simplificar la sintaxis de HTML y acelerar el proceso de desarrollo.

6.3. Módulos

Igual que en el resto de lenguajes de programación, Node.JS dispone de módulos para reducir la complejidad de los programas y simplificar el código principal. Este entorno incorpora varios "módulos básicosçompilados en el propio binario, como por ejemplo el módulo de red, que proporciona una capa para programación de red asíncrona y otros módulos fundamentales, como por ejemplo Path, FileSystem, Buffer, Stream, etc.

Algunos módulos desarrollados por terceros los proporciona la página oficial de Node.JS. Una fuente importante de estos módulos es GitHub, una página donde la comunidad de programadores puede publicar código abierto.

Los módulos pueden ser archivos ".node"precompilados, o archivos en JavaScript plano. Los módulos JavaScript se implementan siguiendo la especificación CommonJS, utilizando una variable de exportación para dar a estos scripts acceso a funciones y variables.

Pueden extender Node.JS o añadir un nivel de abstracción, implementando varias utilidades middleware para utilizar en aplicaciones web. Los módulos pueden instalarse como archivos simples o utilizando el Node.JS Package Manager (npm), utilidad que facilita la instalación, actualiza y resuelve dependencias de forma automática.

Estos módulos se instalan en un directorio por defecto $\backslash nodejs\backslash node\_modules$, si un programa necesita un módulo que no esté en ese directorio, se necesitará indicar una ruta en el programa para poder usarlos.

6.3.1. Módulos incluidos

Los módulos incluidos son los que vienen con la instalación de Node.JS y son creados por los mismos desarrolladores, por lo que se garantiza la estabilidad y la actualización. Los módulos comunitarios son creados por terceros, si resultan ser de una relevante utilidad y estabilidad pueden pasar a incluirse en el código de Node.JS en un futuro.

- **http:** Puede ser el módulo más usado de Node.JS, permite atender a las llamadas entrantes en cualquier puerto y devolverles una respuesta. Se usa principalmente para realizar funciones de servidor web pudiéndose aplicar también al resto de aplicaciones.

- **cluster:** Permite trabajar en multihilo, lo que ayuda a utilizar toda la potencia de la máquina. dns: Es un módulo especializado en resolver peticiones DNS, capaz de resolver la petición con la tabla DNS local o trasladar la petición a un servidor de mayor nivel.

- **tls (ssl):** Permite establecer conexiones seguras usando OpenSSL utilizando el sistema de clave pública-privada. El servidor necesita un certificado para poder ser usado.

- **OS:** Módulo que ofrece funciones de consulta sobre el estado del sistema operativo, como la versión del sistema, el estado de la memoria o el de la CPU.

6.4. Implementación

En este apartado se darán los detalles más significativos de la implementación, se mostrarán las diferentes interfaces de usuario previamente diseñadas y ya integradas dentro del sistema, y se hablará sobre las diferentes decisiones tomadas. Conviene recordad que el carácter del proyecto es confidencial, por lo tanto el código más específico quedará en secreto.

6.4.1. Framework: Express

Express.js es un framework extensible de manejo de servidores HTTP, el cual provee plugins de alto rendimiento conocidos como middleware. Me decidí por este framework debido a la simplicidad de su uso y al fácil manejo de rutas. También ofrece, entre otras características, un router de URL $get, post, put$, el cual vamos a usar para capturar los diferentes eventos que se puedan producir, desde cambiar de página, hasta hacer operaciones con la base de datos.

Además, Express nos ofrece la posibilidad de crear un proyecto predefinido con las funcionalidades básicas y a partir del cual empezar a desarrollar nuestra aplicación siguiendo los fundamentos del framework. El proyecto generado tiene la siguiente estructura:

- **node_modules**: Contiene las librerías instaladas mediante NPM.

- **public**: Contiene todos los recursos que estarán disponibles para la parte del cliente, tales como imágenes, ficheros JavaScript, u hojas de estilo (CSS).
- **routes**: Aquí se encontrarán los métodos que se empezarán la lógica del sistema cuando se capte un evento mediante get o post. Se puede decir, que es como el intermediario entre la petición del cliente y la ejecución de la misma.
- **views**: Contiene la vista del proyecto, las plantillas que generarán nuestra página web. Como he mencionado en el apartado anterior, en este proyecto vamos a utilizar JADE para poder generar páginas dinámicas.
- **app.js**: El fichero principal de la aplicación, el cual se ejecutará para arrancarla.
- **package.json**: Este fichero incluye las declaraciones de las dependencias de nuestra aplicación.

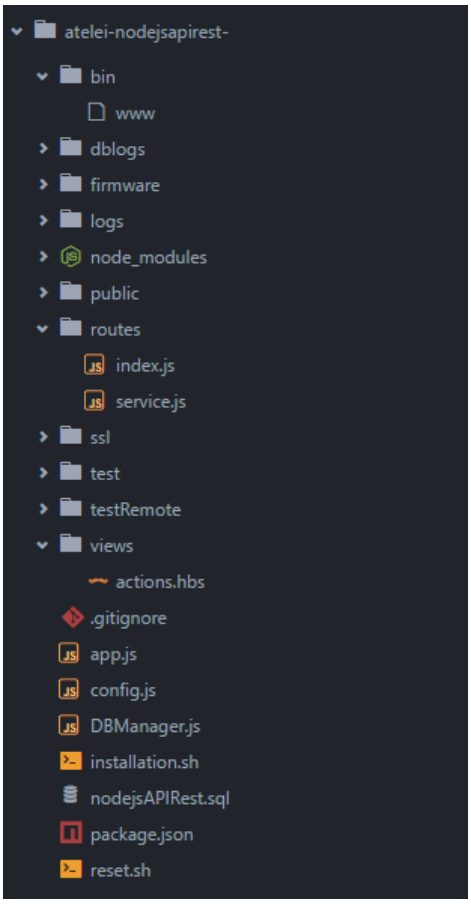


Figura 6.4: Estructura de directorios con el framework Express

6.4.2. Conexión segura

El módulo que lee tarjetas ha de conectarse a internet a través de una estación Wi-Fi con un protocolo de encriptación como mínimo de WPA2, para poder comunicarse con un servidor a través de una arquitectura REST mediante el protocolo de comunicación TCP SSL.

Para ayudarnos con este tipo de conexiones he integrado el modulo https y cambiado el puerto del servidor a el 443 para una conexión segura. Además tenemos que tener en cuenta que el servidor debe disponer los certificados necesarios (los archivos ssl.key y ssl.crt).

En el siguiente ejemplo se ven las líneas de código donde se define lo mencionado anteriormente:

```
1  //Get port from environment and store in Express.
2  var port = normalizePort(process.env.PORT || '443');
3  app.set('port', port);
4
5  //Create HTTPS server.
6  var https = require('https');
7  var fs = require('fs');
8  var server = https.createServer({
9    key: fs.readFileSync('./ssl/ssl.key'),
10   cert: fs.readFileSync('./ssl/ssl.crt')
11 }, app);
```

6.4.3. Autenticación basada en tokens

La autenticación es una de las grandes piezas de cada aplicación y la seguridad es algo que está cambiando y evolucionando. En la actualidad, una de las formas más utilizada para añadir seguridad a un API son los JSON Web Tokens.

La mayoría de las aplicaciones actuales al igual que la de este proyecto, consumen servicios REST y están alojadas en distintos dominios con lo cual no podemos trabajar con sesiones ya que se almacenan en este. Podemos decir que la mejor alternativa es llevar a cabo la autenticación haciendo uso de tokens que vayan del servidor al cliente, un usuario hace login (no necesita enviar token porque no lo tiene), una vez el servidor valide ese usuario le retorna un token cómo respuesta y el usuario debe enviar dicho token en las siguientes peticiones para poder acceder a los recursos del servicio.

En cada petición el servidor debe comprobar el token proporcionado por el usuario y si es correcto podrá acceder a los recursos solicitados, de otra forma deberá denegar la petición.

También nos añade más seguridad. Al no utilizar cookies para almacenar la información del usuario, podemos evitar ataques CSRF (Cross-Site Request Forgery) que manipulen la sesión que se envía al back-end. Por supuesto podemos hacer que el token expire después de un tiempo lo que le añade una capa extra de seguridad.

El formato de un JWT está compuesto por 3 strings separados por un punto . algo así como:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9 . eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9 . pcHcZspUvuiqIPVB_i_qmcvCJv63KLUgIAKIlXI1gY8

- Cada string significa una cosa:

- **Header**: La primera parte es la cabecera del token, que a su vez tiene otras dos partes, el tipo, en este caso un JWT y la codificación utilizada.

- **Payload**: El Payload está compuesto por los llamados JWT Claims donde irán colocados los atributos que definen nuestro token. También podemos añadirle más campos, incluso personalizados, como pueden ser el rol del usuario, etc... Exiten varios, pero, los más comunes a utilizar son:

  - sub: Identifica el sujeto del token, por ejemplo un identificador de usuario.
  - iat: Identifica la fecha de creación del token, válido para si queremos ponerle una fecha de caducidad. En formato de tiempo UNIX
  - exp: Identifica a la fecha de expiración del token. Podemos calcularla a partir del iat. También en formato de tiempo UNIX.

- **Signature**: La firma es la tercera y última parte del JSON Web Token. Está formada por los anteriores componentes (Header y Payload) cifrados en Base64 con una clave secreta (almacenada en nuestro backend). Así sirve de Hash para comprobar que todo está bien. En el proyecto documentado en esta memoria me ayude de los módulos 'moment' y 'jwt-simple' para determinar la duración y crear el String que los tokens. En el siguiente ejemplo vemos como se crean tokens con un día de caducidad:

//index.js:

```
function createTok(idencoder){
    var token = service.createToken(idencoder);

    var pay = jwt.decode(token, config.TOKEN\_SECRET);
    var tokenExp = pay.exp;
    var expDate = moment.unix(tokenExp).format("YYYY-MM-DD HH:mm:ss");
    var ret = [token, expDate];

    return ret;
}
```

// services.js:

```
var jwt = require('jwt-simple');
var moment = require('moment');
var config = require('../config');

exports.createToken = function(user) {
    var payload = {
        sub: user.\_id,
        iat: moment().unix(),
        exp: moment().add(1, "days").unix()
    };
    return jwt.encode(payload, config.TOKEN\_SECRET);
};
```

// config.js:

```
module.exports = {
    TOKEN\_SECRET: process.env.TOKEN\_SECRET || "tokenultrasecreto"
};
```

6.4.4. Mensajes JSON

La estructura para enviar mensajes JSON es la siguiente:

```
JSON.stringify\(\{ VAR: 'data'\}, null, 3\)\);
```

6.4.5. Interfaz para simular transacciones

Se necesitan crear tareas para poder simular los diferentes casos de usos con el encoder. Para eso y ayudándome de tecnologías agiles como boostrap he creado dos interfaces web para la gestión de tareas. Como podemos ver en la imagen 6.5 esta primera interfaz le posibilita al usuario crear transacciones.

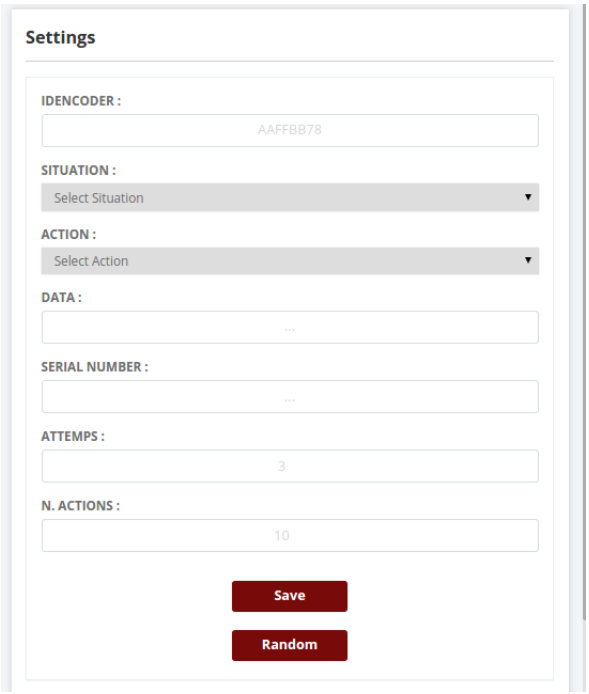


Figura 6.5: Interfaz web para la inserción de tareas.

- **Idencoder**: Identificador del encoder
- **Situation**: Situación con la que se quiere insertar la tarea en la BD. (Pending,sent, no_ok, okay, cancel)
- **Action**: Tipo de acción que se requiere para la tarea. (rd_card, wr_card, rd_serial)
- **Data**: Dato que se le asigna a la tarea de escritura.
- **Serial number**: Dato que se le asigna a la tarea de escritura
- **Attemps**: Numero de intentos máximo disponibles por el servidor para completar la tarea.

- **N. Actions**: Numero de transacciones o tareas que se quieren insertar en la fila de tareas del servidor.
- **Save**: Botón para guardar las simulación en la Base de Datos.
- **Random**: Botón para guardar las simulaciones que se han definido en el formulario de forma aleatoria en el tiempo. Este botón nos dará la posibilidad de simular un flujo real, ya que no siempre se hace todas las inserciones de tareas al mismo tiempo.

La siguiente sección de la interfaz web que he diseñado es un formulario para cancelar una transacción en cualquier momento. Este formulario hará que el identificador de tarea que pongamos pase a tener un estado cancelado en base de datos con lo que el servidor cambiara el flujo de comunicaciones sobre esa transacción.

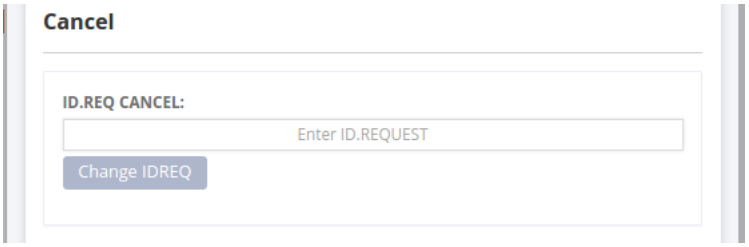


Figura 6.6: Interfaz web para cancelar una tarea determinada.

La sección que vemos en la imagen 6.7 me ha sido de gran ayuda para depurar las comunicaciones y tener controlado en todo momento lo que el servidor tenía guardado. Cuando se pulsa el botón de reset se hará un dump de la base de datos con el primer backup guardado en el servidor.

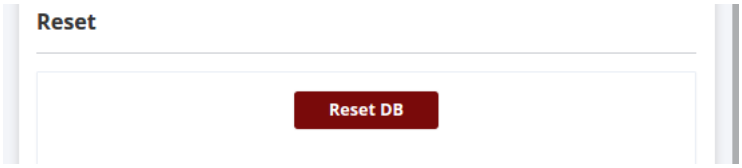


Figura 6.7: Interfaz web para resetear la base de datos.

La última sección nos da la posibilidad de ver la fila de acciones del servidor para eso solo tenemos que escribir el ID del encoder y al darle al botón podemos observar la lista de tareas que tiene el servidor para ese encoder. Además, en la interfaz de la lista podemos borrar o cancelar determinadas tareas mediante el botón de update y los checkbox de la parte derecha como se puede observar en la imagen 6.9.

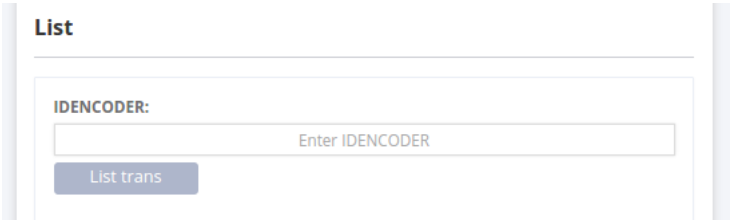


Figura 6.8: Interfaz web para ver el listado de tareas de un encoder determinado.

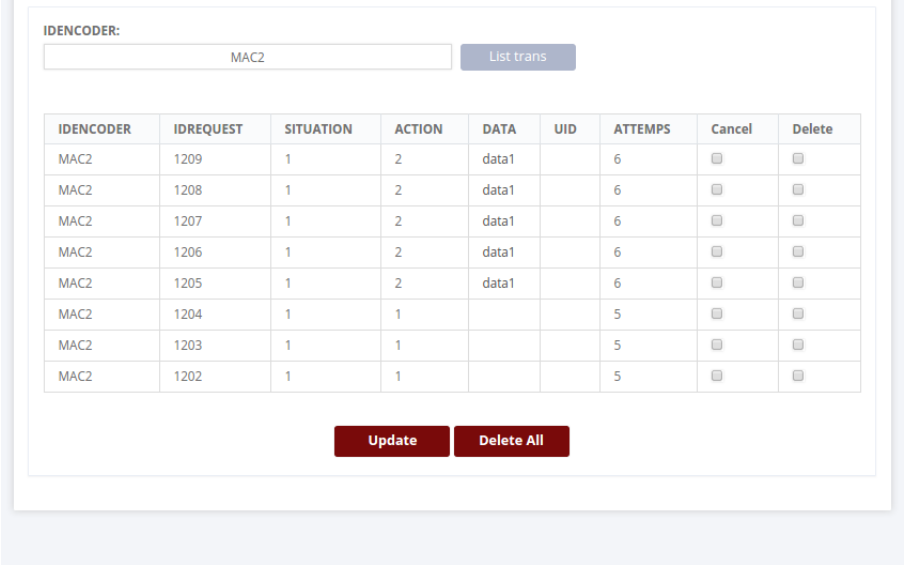


Figura 6.9: Interfaz web para ver el listado de tareas de un encoder determinado.

6.4.6. Herramienta para depurar las conexiones y comunicaciones

Para una mejor depuración de errores he creado dos sistemas logs. El primero de ellos y más sencillo, es el de logs en base de datos, en ese archivo guardare las conexiones diarias a la base de datos y el estado de dicha conexión junto la hora de acceso. Ejemplo:

```
1 > 04/27/2016 15:31:39 - Connected successfully to DB!
```

El otro archivo de logs estará compuesto de los log de las comunicaciones. Este tendrá una línea con la hora, la fecha y los datos por cada comunicación éntrate o saliente del servidor. Además este archivo lo podrá descargar el administrador de sistema mediante una url. Ejemplo:

```
1 > 07/31/2016 20:52:30 <-> GET execute {IDENCODER: MAC, TOKEN: eyJ0e..., IDREQUEST: 70, ACTION: 2,
      DATA: data123456789, UID: 123, STATUS: 1 }
2 > 07/31/2016 20:52:30 <-> SERVER SENT--> {STATUS:1, IDREQUEST:70, ACTION:1, DATA:data123456789 ,
     UID:123 }
```

En el ejemplo podemos ver como al servidor le llega una petición get a la ruta execute y como el servidor le responde inmediatamente con la transacción que tenía pendiente.

6.4.7. Módulos empleados en la construcción de la API

- **body-parser**: Devuelve el middleware que sólo analiza JSON
- **cookie-parser**: Analiza la cabecera Cookie y rellena req.cookies con un objeto introducido por los nombres de cookies.
- **debug**: Utilidad para depurar Node.JS.
- **express**: Framework.
- **hbs**: Plugin para las plantillas handelbars.
- **fs-extra**: Añade métodos del sistema de archivos que no están incluidos en el módulo de fs nativo.
- **Mysql**: Un controlador Node.JS para MySQL. Está escrito en JavaScript, no requiere compilación, y es 100

- **utils-merge**: Combina las propiedades de un objeto de origen en un objeto de destino.
- **http**: Permite atender a las llamadas entrantes en cualquier puerto y devolverles una respuesta.
- **https**: Permite atender a las llamadas entrantes en el puerto 443 y devolverles una respuesta.
- **request**: Está diseñado para ser la forma más sencilla posible de hacer llamadas HTTP. Es compatible con HTTPS y sigue las redirecciones de forma predeterminada.
- **multer**: Es un middleware de Node.JS para el manejo de multipart y form-data, se utiliza principalmente para la carga de archivos.
- **request-json**: Esta librería simplifica las llamadas HTTP para el uso de mensajes JSON.
- **jwt-simple**: Modulo para codificar y decodificar JSON Web Tokens.
- **moment**: Una biblioteca de fechas de JavaScript para analizar, validar, manipular y dar formato a las fechas.
- **express-session**: Middleware simple para las sesiones en express.
- **session-file-store**: Es una disposición para almacenar los datos de sesión en el archivo de sesión.
- **client-sessions**: Es un middleware de conexión que implementa sesiones en encrypted tamper-free cookies.
- **fs**: Ayuda para el manejo de archivos.
- **node-schedule**: Un modulo de ayuda para un pplanificador de tareas.
- **socket.io**: Framawork para el servidor en tiempo real.

6.5. Instalación

En el Anexo-B podremos encontrar una guía para la instalación de la aplicación en nuestro sistema.

7. CAPÍTULO

Diseño y desarrollo en PHP

En este apartado se detallarán los temas concernientes al diseño del sistema con PHP, más concretamente con el framework codeigniter. Se hará hincapié en la arquitectura definida más allá de la división por capas.

Por otro lado, se hablará del desarrollo del proyecto y las decisiones que se han ido tomando a medida que iba avanzando el proyecto con PHP y codeigniter. Finalmente se describirán las diferentes implementaciones.

Diseño y desarrollo en PHP
1. Arquitectura 2. Implementación 3. Instalación

7.1. Arquitectura

A continuación se muestra la arquitectura de CodeIgniter:

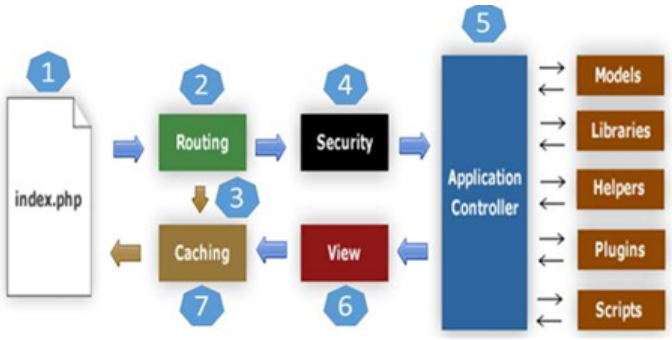


Figura 7.1: Modelo de ejecucion en codeigniter

- Como se muestra en la imagen anterior, cada vez que llega una petición de CodeIgniter, primero ira a la página index.php.
- En el segundo paso, el routing o enrutamiento decidirá si pasar esta solicitud al paso 3 para el almacenamiento en caché o pasarla a la etapa 4 para la comprobación de seguridad.
- Si la página solicitada ya está en caché, entonces enrutamiento hará llegar la petición al paso 3 y la respuesta se le devolverá al usuario.
- Si la página solicitada no existe en caché, entonces routing pasa a la página solicitada al paso 4 para las verificaciones de seguridad.
- Antes de pasar la petición al controlador de aplicaciones, se comprueba la seguridad de los datos presentados. Después de la comprobación, el controlador de aplicación carga los modelos, bibliotecas, ayudantes, plugins y scripts necesarios y lo transmiten a la vista.
- La vista renderizara la página con los datos disponibles y la pasara a cachear, debido a que antes la página no se ha cacheado en este paso se cacheara para procesar rápidamente en futuras solicitudes.

7.1.1. Estructura de directorios

La estructura de directorios se divide en 3 carpetas principales:

- Application: Se encuentran todos los archivos de nuestro proyecto y es donde trabajaremos.
- La carpeta **cache** almacena los caches generados por el módulo de caché.
- La carpeta **codeigniter** almacena los archivos necesarios para que CI funcione.
- La carpeta **database** almacena todos los drivers de base de datos y las clases que habilitan las conexiones a bases de datos.
- La carpeta **fonts** almacena todas las fuentes que pueden ser usadas por el módulo de manipulación de imágenes.
- La carpeta **helpers** contiene los atajos del núcleo de CI, pero es posible ubicar otros atajos personales para que accedan todas las aplicaciones.
- La carpeta **language** contiene los archivos de lenguaje del núcleo de CI que usan sus módulos y atajos, pero es posible ubicar carpetas de lenguaje para que accedan todas las aplicaciones.
- La carpeta **libraries** almacena las librerías del nucleo de CI, pero es posible ubicar otras librerías personales para que sean accedidas por todas las aplicaciones.
- La carpeta **logs** contiene todos los logs (registros) generados por CI.
- La carpeta **plugin** almacena todos los plugins que pueden ser usados. Los plugins son casi idénticos a los atajos (helpers). Son funciones dirigidas a ser compartidas por la comunidad.
- La carpeta **scaffolding** almacena todos los archivos que hacen que la clase scaffolding (andamiaje) funcione. Scaffolding permite una interfície CRUD (Create-Read-Update-Delete) conveniente para acceder a la información de la base de datos durante el desarrollo.
- La **user_guide** guarda la guía de usuario de CI.
- El archivo **index.php** es el elemento que gestiona toda la magia de CI y permite cambiar el nombre de las carpetas de sistema y de aplicación.

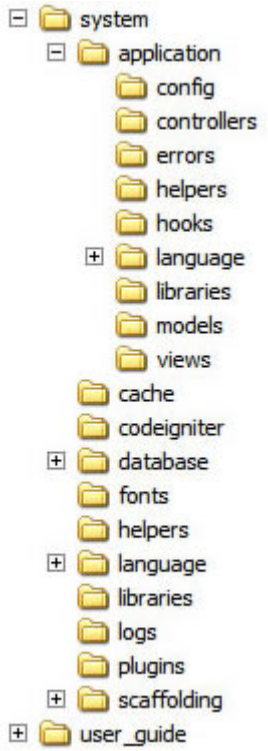


Figura 7.2: Estructura de directorios con el framework Codeigniter

Dentro de la carpeta application nos podemos encontrar con el anteriormente mencionado MVC, es es la forma de estructurar una aplicación separándola en 3 grupos:

- **Modelo**: Gestiona el acceso o manipulación de la base de datos.
- **Vista**: Es la parte visual y suele ser código HTML puro y duro pero en Codeigniter esto no es así y se usa bastante PHP cosa que no a todos les gusta.
- **Controlador**: Se encarga de hacer de intermediario a parte de procurar, como su nombre indica, controlar todo lo que pasa en la web, acceso de usuarios, errores...

Además de las tres carpetas básicas dentro de application nos encontramos con:

- **Helpers**: Se encuentran aquellas funciones que nos facilitarán la escritura de código. Codeigniter proporciona algunos, como facilitarnos la creación de formularios. Nosotros crearemos otros más adelante para comprenderlos mejor.
- **Hooks**: En esta carpeta alteraríamos el comportamiento que tiene normalmente el framework sin tocar en los archivos los contenidos de la carpeta system. Nosotros lo usaremos para cambiar la seguridad y controlar la entrada de usuarios.

- **Libreries**: La usaremos para la creación de funciones complicadas, que involucren varias tablas o que sencillamente no creamos conveniente alojarla en el modelo. La intención de esto es aislar la lógica de la aplicación con el acceso a la base de datos.
- **Third_party**: Serian las librerías que nosotros nos descargamos para ampliar funcionalidades de PHP. Como podría ser leer PDF o generarlos, envió de correos... Se alojarían todos aquellos plugins creados por personas ajenas a nuestra aplicación.
- **Config**: Se encuentras los archivos más comunes de configuración.

Capa de lógica

Después de desarrollar las interfaces de simulación para Node.JS, la empresa ATELEI y yo decidimos que no era necesario tener un gestor de transacciones en PHP ya que no iba a ver apenas diferencias con el desarrollado para Node.JS.

Capa de presentación

Después de desarrollar las interfaces de simulación para Node.JS, la empresa ATELEI y yo decidimos que no era necesario tener un gestor de transacciones en PHP ya que no iba a ver apenas diferencias con el desarrollado para Node.JS.

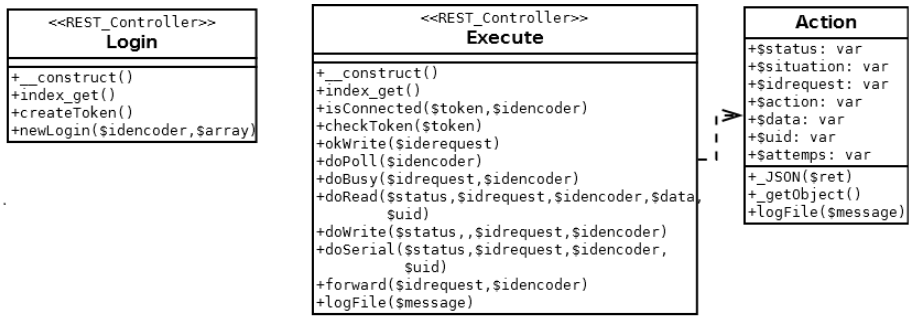


Figura 7.3: Modelo de la capa de logica para PHP.

Capa de datos

Esta es la capa que se encarga de la persistencia, es decir, recibe la información de la capa de lógica y almacena esos datos. En esta capa se definirán todas las acciones tanto de entrada como de salida en base de datos.

Para nuestro proyecto decidí crear 2 clases, una que se encargara de los datos cuando el encoder hace peticiones de login y la otra para la interacción con execute.

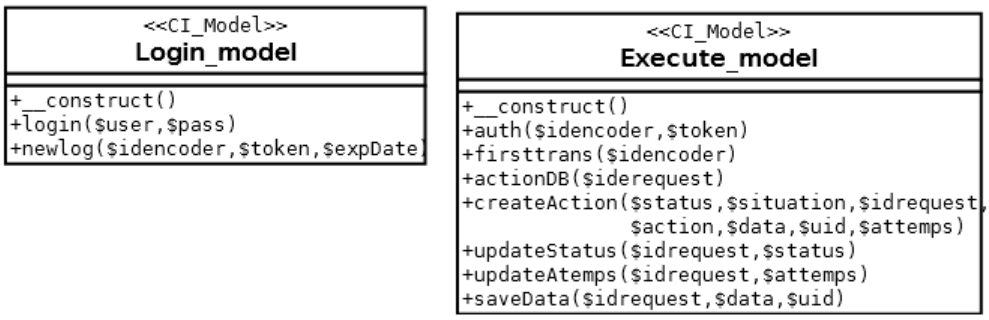


Figura 7.4: Modelo de la capa de datos para PHP

7.2. Implementación

En este apartado se darán los detalles más significativos de la implementación con PHP y codeigniter, y se hablará sobre las diferentes decisiones tomadas. Conviene recordad que el carácter del proyecto es confidencial, por lo tanto el código más específico quedará en secreto.

7.2.1. Framework: Codeigniter

La elección del framework adecuado fue la primera decisión cuando analice esta tecnología. Ya hemos visto las ventajas de la elección de codeigniter y sus características más importantes en capítulos anteriores.

7.2.2. Librerías

Emplearé una librería PHP de un tercero para poder integrar las funciones de una API REST de una forma más sencilla; específicamente la librería llamada: CodeIgniter Rest Server la cual es sencilla de implementar, emplear y hace su propósito; además cuenta con algunas características extras como el manejo de API Keys y autenticación. Para instalar la librería simplemente debemos copiar los siguientes archivos en nuestro proyecto CodeIgniter alojado en nuestro servidor Apache:

- Copiar los archivos *.../Format.php* y *.../REST_Controller.php* dentro de la carpeta library.
- Copiar el archivo *application/config/rest.php* dentro conf.
- Colocar la llamada *require APPPATH . '/libraries/REST_Controller.php'* al inicio del archivo de nuestros controladores.

7.2.3. Conexión segura

Codeigniter nos permite redireccionar toda la web a ssl de una forma muy sencilla. Una vez que tengamos instalado y funcionando el certificado lo que se va a hacer es que aunque se escriba http://www.urtzi.com, se accedan a https://www.urtzi.com obligatoriamente. Para conseguir ese propósito me voy a ayudar de los hooks. Para ello, hacemos los siguentes tres pasos:

1.- Activar hooks en Codeigniter. En el fichero '*application/config/config.php*' y cambiar la linea para activar los hooks.

2.- Se modifica el fichero hooks.php. En el fichero 'application/config/hooks.php'.

3.- Se crea el fichero con la función que ejecutará el hook 'application/hooks/ssl.php' y añadimos el siguiente contenido:

```
1   function redirect_ssl() {
2       $CI =& get_instance();
3       $class = $CI->router->fetch_class();
4       $exclude = array('client'); // add more controller name to exclude ssl.
5       if(!in_array($class,$exclude)) {
6           // redirecting to ssl.
7           $CI->config->config\[ 'base_url'\]  = str_replace('http://', 'https://', $CI->config
    ->config\[ 'base_url'\] );
8           if ($_SERVER\[ 'SERVER_PORT'\]  != 443) redirect($CI->uri->uri_string());
9       } else {
10          // redirecting with no ssl.
11          $CI->config->config\[ 'base_url'\]  = str_replace('https://', 'http://', $CI->config
    ->config\[ 'base_url'\] );
12          if ($_SERVER\[ 'SERVER_PORT'\]  == 443) redirect($CI->uri->uri_string());
13      }
14  }
15
```

De esta manera, obligamos a navegar siempre con https://

7.2.4. Autenticación basada en tokens

En el capítulo anterior ya hemos visto en porque utilizar los Json Web Tokens con lo que en las siguientes líneas simplemente voy a explicar el proceso de integración con codeigniter.

1. Usando composer podemos administrar las dependencias e instalar PHP-JWT. *composer require firebase/php-jwt*

2. La siguiente función es el ejemplo de como he usado esta librería y la e integrado en el sistema.

```
1        function createToken(){
2        $key = "example_key";
3        $issuedAt   = time();
4        $notBefore  = $issuedAt+0;                    //Adding 0 seconds
5        $expire     = $notBefore + 86400;            // Adding 86400 seconds
6        $php_expire_date = date("Y-m-d H:i:s ", $expire);
7
8        $token = array(
9            "iat"  => $issuedAt,
10           "exp"  => $expire
11       );
12
13       $token = JWT::encode($token, $key);
14       $expDate=$php_expire_date;
15
16       $array = array(
17           "token" => $token,
18           "expDate" => $expDate,
19       );
20       return $array;
21   }
22
```

En mi caso podemos ver que no solo le devuelvo el token al controlador, también le devuelvo la fecha de caducidad para que más adelante el modelo la guarde en base de datos.

7.2.5. Mensajes JSON

La estructura para enviar mensajes JSON es la siguiente:

```
header\('Content-Type: application\/json'\);
json_encode( array(VAR => "data") );
```

7.2.6. Herramienta para depurar las conexiones y comunicaciones

De la misma forma que con Node.JS para una mejor depuración de errores, he creado un sistema de logs. Este estará compuesto de los log de las comunicaciones. Tendrá una línea con la hora, la fecha y los datos por cada comunicación entrate o saliente del servidor. Ejemplo:

```
08/14/2016 13:16:19 <-> GET execute {IDENCODER:MAC, TOKEN: eyJ0e..., IDREQUEST: 70, ACTION:5,
    DATA: , UID: , STATUS: 1}
08/14/2016 13:16:19 <-> SERVER sent--> {"STATUS":1,"IDREQUEST":0,"ACTION":0,"DATA":"","UID":""}
```

7.3. Instalación

En el Anexo-C podremos encontrar una guía para la instalación de la aplicación en nuestro sistema.

8. CAPÍTULO

Estudio de rendimiento y simulación

En la primera parte de este capítulo abordaremos las pruebas realizadas para comprobar el correcto funcionamiento del servidor. En la segunda parte de esta sección el problema a tratar será la respuesta del servidor web ante un benchmarking, es decir, se medirá el rendimiento de la aplicación, y se compararan los tiempos de respuesta de los dos sistemas desarrollados. Se presentara la herramienta con la que poder llevar acabo las pruebas, más adelante se hará un breve resumen de las diferentes pruebas y el capítulo concluirá con unos gráficos donde podemos ver comparados los resultados de las mencionadas pruebas.

Validación del lado servidor

1. Validación del lado servidor
2. Benchmarking de servidores
3. Tipos de pruebas
4. Buenas practicas
5. Herramientas
6. Implementación de las pruebas
7. Resultados
8. Valoración

8.1. Validación del lado servidor

Debido al grado complejidad de las comunicaciones decidí crear diferentes pruebas para asegurarme el correcto flujo de las comunicaciones. En estas pruebas se va a simular las comunicaciones provenientes del encoder con la herramienta cURL que soporta los protocolos FTP, FTPS, HTTP, HTTPS, TFTP, SCP, SFTP, Telnet, DICT, FILE y LDAP, entre otros...

El principal propósito y uso para cURL es automatizar transferencias de archivos o secuencias de operaciones no supervisadas. En mi caso, una herramienta excelente para simular las acciones que deberían llegar desde el lector de tarjetas al servidor desarrollado.

Por otro parte, para automatizar y facilitarme las pruebas he creado varios script que serán los encargados de llevar acabo las diferentes simulaciones deoendiendo los parametros de entrada.

Ejemplo script simulaciones:

```
1  \# !/usr/bin/env bash
2  testNumber=\$1
3  testOut=\$2
4  max=\$3
5  inicio_ns=`date +\%s\%N`
6  inicio=`date +\%s`
7  for i in `seq 1 \$max`
8  do
9      echo -e "--Test n.\$i--" >> test\$testOut.txt
10     inicio_ns2=`date +\%s\%N`
11     inicio2=`date +\%s`
12     \$(./test\$testNumber.sh \$testOut)
13     fin_ns2=`date +\%s\%N`
14     fin2=`date +\%s`
15     echo -e "" >> test\$testOut.txt
16     let total_ns2=\$fin_ns2-\$inicio_ns2
17     let total2=\$fin2-\$inicio2
18     echo -e "T.n.\$i: (\$total_ns2 ns), (\$total2 s)" >> test\$testOut.txt
19     echo -e "" >> test\$testOut.txt
20 done
21 fin_ns=`date +\%s\%N`
22 fin=`date +\%s`
23 let total_ns=\$fin_ns-\$inicio_ns
24 let total=\$fin-\$inicio
25 echo "Tiempo ejecucion: (\$total_ns ns), (\$total s)"
26 echo -e "~~~~~~~~~~~~~~~    i:\$i, T: (\$total_ns ns), (\$total s)    ~~~~~~~~~~~~~~\n" >> test\
       $testOut.txt
```

Ejemplo script cURL:

```
\# !/usr/bin/env bash

testOut=test\$1.txt

token='eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE0NjcxMDg2NTcsImV4cCI6MTQ5ODY0NDY1N30.
    Yp3UbK5zXKZKlumjWTQTSfREj75pwKSa323wY_8GNt4'
idencoder='MAC'
idrequest=70
action=5
data=''
uid=''
status=1

stringToSent='TOKEN='\$token'&IDENCODER='\$idencoder'&IDREQUEST='\$idrequest'&ACTION='\$action'&
    DATA='\$data'&UID='\$uid'&STATUS='\$status
curl -k "https://localhost/execute?\$stringToSent" >> \$testOut
```

8.1.1. Registro de los resultados

Para llevar el control de los resultados se va a seguir la tabla del Anexo-D, en ella podemos ver la petición que simula cURL como si fuese el encoder, la respuesta esperada y la respuesta de nuestro servidor para cada caso de prueba. El id de cada petición también corresponde al archivo donde se guardaran los resultados de cada simulación.

Ejemplo fichero de resultados

En el fichero de resultados podemos ver el JSON con el que responde el servidor en cada peticion del script de pruebas. Ademas del mensaje JSON, se guarda el tiempo utilizado en cada iteración y al final del bucle también se escribe el tiempo total para todas las iteraciones probadas.

```
--Test n.1--
{
   "STATUS": 1,
   "IDREQUEST": 64,
   "ACTION": 1,
   "DATA": "",
   "UID": ""
}
T.n.1: (22659710 ns), (0 s)

```

```
--Test n.2--
{
   "STATUS": 1,
   "IDREQUEST": 65,
   "ACTION": 1,
   "DATA": "",
   "UID": ""
}
T.n.2: (85628805 ns), (0 s)

--Test n.3--
{
   "STATUS": 1,
   "IDREQUEST": 66,
   "ACTION": 1,
   "DATA": "",
   "UID": ""
}
T.n.3: (63957989 ns), (0 s)

--Test n.4--
{
   "STATUS": 1,
   "IDREQUEST": 67,
   "ACTION": 1,
   "DATA": "",
   "UID": ""
}
T.n.4: (66240546 ns), (0 s)

--Test n.5--
{
   "STATUS": 1,
   "IDREQUEST": 68,
   "ACTION": 1,
   "DATA": "",
   "UID": ""
}
T.n.5: (66121482 ns), (0 s)

~~~~~~~~~~~~~~    i:5, T: (597129081 ns), (0 s)    ~~~~~~~~~~~~~~
```

8.2. Introducción benchmarking de servidores

Ya hemos visto en anteriores capítulos que es el benchmarking y como el proceso de benchmarking es utilizado por las empresas para ver su competitividad respecto a otras del mismo sector. En sistemas como los que he desarrollado en este proyecto hay diversos

factores en juego, por ejemplo, el diseño front-end, la latencia/ancho de banda de red, la configuración del servidor Web, el servidor en la memoria caché, la capacidad hardware crudo, la carga del servidor de alojamiento compartido, etc... Para comparar, optimizar el rendimiento y para ver la competitividad entre los dos sistemas desarrollados he realizado una serie de pruebas que veremos a continuación.

8.3. Tipos de pruebas

Podemos agrupar las pruebas en tres grupos diferentes, cada tipo de prueba tiene un objetivo y unos resultados acordes al tipo de objetivo.

8.3.1. Performance Testing (Pruebas de actuación o conducta)

El objetivo es predecir anticipadamente problemas de rendimiento y degradación de recursos del sistema antes de su paso a producción, para así facilitar su corrección. La evaluación apunta a medir si se cumplen los requerimientos establecidos por el cliente. Suelen llevarse a cabo en conjunción con stress testing que analizare a continuación. Este tipo de pruebas se suelen hacer para tener un estimativo de cómo puede llegar a responder la aplicación en el ambiente de producción, aunque es casi imposible saber exactamente cómo va a responder, ya que nunca se tienen los datos o el equipamiento necesario para poder hacer simulaciones 100 % reales. Técnicamente, se obtiene la siguiente información mediante este tipo de pruebas:

- Evaluar la entrega:
  - ¿ Cumple con lo que espera el cliente?
  - ¿ Cómo se estima que funcione la aplicación en producción?
- Evaluar la infraestructura elegida:
  - ¿Es adecuada para la capacidad que va a soportar?;
  - ¿Se comparó con otras tecnologías?
  - ¿Se producen cuellos de botella?

8.3.2. Load Testing (Pruebas de carga)

Se centra en la “cantidad” que puede manejar la aplicación. La idea de estas pruebas es poner al límite del alcance la cantidad de usuarios simultáneos en la aplicación y acercarse lo más posible al limite de cómo va a reaccionar la aplicación en el mundo real. Se suelen dividir en dos ramas:

- Longevity testing: Evaluar la estabilidad del sistema para manejar una constante carga de trabajo durante un periodo extenso.
- Volume Testing: Poner al sistema al límite de carga de trabajo durante un periodo corto.

8.3.3. Stress Testing (Pruebas de estrés)

Recoge diversos resultados en materia de mediciones sobre diversos modelos de carga y actividades que son más .estresantes"de lo que la aplicación va a utilizar a la hora de entregarse a los verdaderos usuarios (excede las especificaciones). Nos indica cómo va a reaccionar el sistema cuando este se pasa su lí mite de tolerancia.

Tipo de prueba	Beneficios	Vulnerabilidades
Performance	Determina las características de velocidad, escalabilidad y estabilidad. Se enfoca en determinar si el usuario del sistema va a estar satisfecho con la actuación del sistema.	Si no es diseñado y validado correctamente, puede arrojar resultados muy poco significativos. A menos que las pruebas se realicen en el ambiente de producción, simulando las mismas máquinas de los usuarios finales, siempre habrá un grado de incertidumbre en los resultados.
Load	Determina el rendimiento o throughput requerido para soportar anticipadamente los picos en producción. Determina si el hardware del ambiente es adecuado y detecta problemas de concurrencia. Ayuda a determinar la cantidad de usuarios que pueden manejar la aplicación antes que la conducta se vea afectada.	No está diseñado para enfocarse en la velocidad de respuesta. Los resultados únicamente pueden ser usados para comparar con otras pruebas de carga.
Stress	Determinar la consistencia de la información cuando la aplicación se lleva más allá de sus límites. Determinar un estimativo de hasta dónde (antes de ocasionar errores y lentitud) puede llegar el sistema al llevarlo al límite. Ayuda a determinar qué clase de errores son más importantes.	Como las pruebas de stress son irreales, los usuarios finales pueden no considerar los resultados. Es difícil determinar cuánto stress es necesario aplicar en la aplicación. Puede ocasionar grande fallas en la aplicación y la red.

Tabla 8.1: Tabla comparativa de los tipos de pruebas para el benchmarking.

8.4. Buenas practicas (Metodología)

Planificación: Tiene como objetivo identificar la carga de pruebas.

- Obtención de requisitos.
- Elección de las métricas adecuadas.
- Elección del tipo de pruebas.
- Descripción del sistema a probar.
- Caracterización de la carga de trabajo.

Construcción: Tiene como objetivo la construcción de las pruebas.

- Definición de escenarios
- Distribución de las tareas

Ejecución: Tiene como objetivo ejecutar las pruebas o scripts desarrollados en la etapa anterior de construcción.

- Dependiendo del tamaño del proyecto puede ocurrir 1 o varias veces.
- Se registran las fecha y hora de ejecución

Análisis: Tiene como objetivo recabar toda la información generada por las pruebas, procesarla de manera que se pueda interpretar y sacar conclusiones.

- Recopilación de Resultados
- Foco en métricas más significativas
- Definición de gráficos más adecuados para mostrar la información

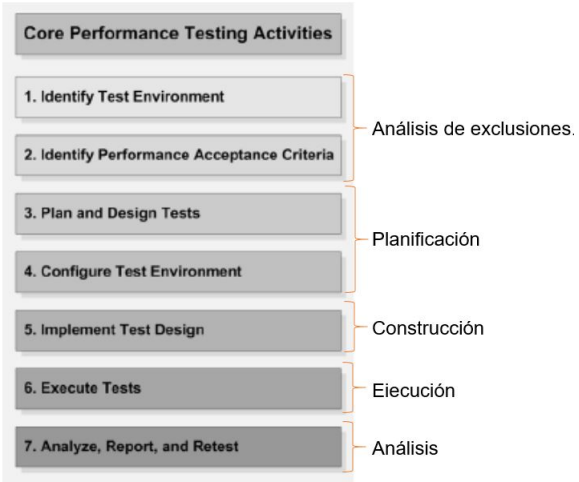


Figura 8.1: Metodologia para un buen benchmarking

8.5. Herramientas

Las herramientas más populares son; JMeter y ApacheBench. Ambas herramientas están creadas para testear el rendimiento de servidores tipo Apache, sobre todo ApacheBench, la cual está integrada dentro de la instalación del propio Apache, por lo que no es necesaria su instalación, de ahí que sólo es posible utilizarla desde la propia máquina donde está instalado el servidor Apache. JMeter a diferencia de ApacheBench, sí puede ser ejecutada desde un equipo distinto al que tiene instalado la aplicación web. Debido a la cantidad de diferentes tipos de comunicaciones para nuestro benchmarking y la facilidad de aprendizaje se ha escogido JMeter como único simulador de rendimiento. En los siguientes capítulos conoceremos un poco más sobre las pruebas realizadas y su funcionamiento.

8.5.1. JMeter

JMeter es una herramienta Java la cual fue diseñada para hacer pruebas de carga en aplicaciones Web, pero se expandió a otras funciones. Típicamente es usada para medir Performance y Pruebas de carga, es una herramienta OpenSource. No requiere una instalación como tal, ya que se trata de una aplicación diseñada en Java 100 % que puede ser ejecutada en cualquier plataforma que tenga instalada la máquina virtual java 1.5 o superior.

Una vez descargada lo único que tenemos hacer es ejecutar el archivo contenedor y se mostrará la ventana de la aplicación. A partir de ahí habrá que realizar los distintos cambios y configuraciones que creamos oportunos para realizar las pruebas pertinentes.

Como ya he mencionado antes, con JMeter podemos hacer que se graben todas las peticiones que hagamos desde un navegador web hacia el servidor web, por lo que nos ahorramos la comunicación entre el cliente web y la aplicación. Para llevar a cabo dicho trabajo tendremos que realizar una serie de configuraciones tanto en la herramienta JMeter como en el navegador donde reside dicha herramienta.

8.6. Implementación de las pruebas

Para poder evaluar los resultados utilicé el componente de informe agregado de JMeter. Este componente es similar a la herramienta de informe resumido, pero permite obte-

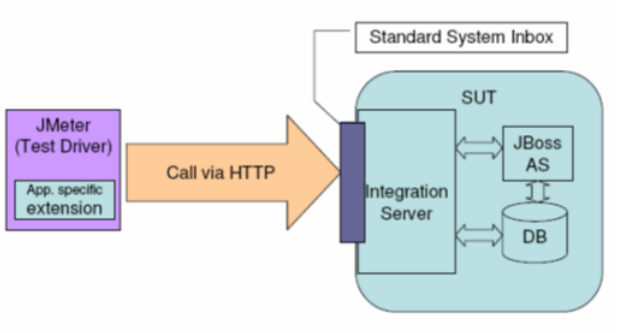


Figura 8.2: Funcionamiento JMeter

ner resultados más precisos. Utiliza más memoria, ya que calcula la mediana y la línea al 90 %, la cual requieren que todos los datos estén almacenados. Mediante esta prueba los datos obtenidos son:

- **URL** : etiqueta de la muestreo
- **Muestras**: cantidad de hilos utilizados para la URL.
- **Media**: tiempo promedio en milisegundos para un conjunto de resultados.
- **Mediana**: Simplemente es el valor medio del juego de datos cuando estos son ordenados de menor a mayor.
- **Línea de 90 %**: Es el valor por debajo del cual el 90 % de las muestras fallan.
- **Min**: tiempo mínimo que demora un hilo en acceder a una página.
- **Max**: tiempo máximo que demora un hilo en acceder a una página.
- **%Error**: porcentaje de requerimientos con errores.
- **Rendimiento (Throughput)**: Es la tasa promedio de mensajes entregados satisfactoriamente. Comúnmente se mide en bits por segundos (bps).
- **KB/sec**: rendimiento medido en Kbytes por segundo.

Cada prueba implementada va asociada a un id para poder identificar el tipo de prueba en la tabla fácilmente. La siguiente tabla muestra las diferentes pruebas que se han realizado contra los servidores. El número de peticiones por hilo tiene siempre un periodo de subida de 1 segundo.

ID de la prueba	Cantidad de hilos	Número de peticiones
1	1	1
2	5	1
3	10	1
4	10	10
5	10	100
6	10	1000
7	50	1
8	100	1
9	100	10
10	100	100
11	100	1000
12	500	1
13	1000	1
14	1000	10
15	1000	100
16	5000	1
17	10000	1

Tabla 8.2: Tabla informativa sobre las pruebas implementadas.

8.7. Resultados

En el Anexo-E podemos ver las tablas de los resultados de las simulaciones con JMeter. Mediante esos resultados podemos analizar las características del servidor Node.JS y el servidor PHP y sacar colusiones o mejoras de nuestros servicios antes de pasar los sistemas a producción.

8.7.1. Graficas login

Con los tiempos medios de cada petición he creado los siguientes gráficos para comprender mejor el comportamiento de los servidores. Las siguientes mediciones pertenecen a peticiones del tipo login. Me voy a centrar más en este tipo de peticiones ya que todas las respuestas van a ser similares aunque varié la petición, en cambio en las peticiones del tipo execute varía más el flujo en el servidor.

En las dos primeras primera graficas vemos las pruebas con id 1, 2, 3, 4, 5, 8, 12, 13 y 16, como podemos observar en la tabla 8.3 estas pruebas pertenecen a los tests de una sola petición por hilo (este es comportamiento habitual en nuestro sistema).

ID de la prueba	Cantidad de hilos	Número de peticiones
1	1	1
2	5	1
3	10	1
7	50	1
8	100	1
12	500	1
13	1000	1
16	5000	1

Tabla 8.3: Tabla para las graficas 8.3 y 8.4

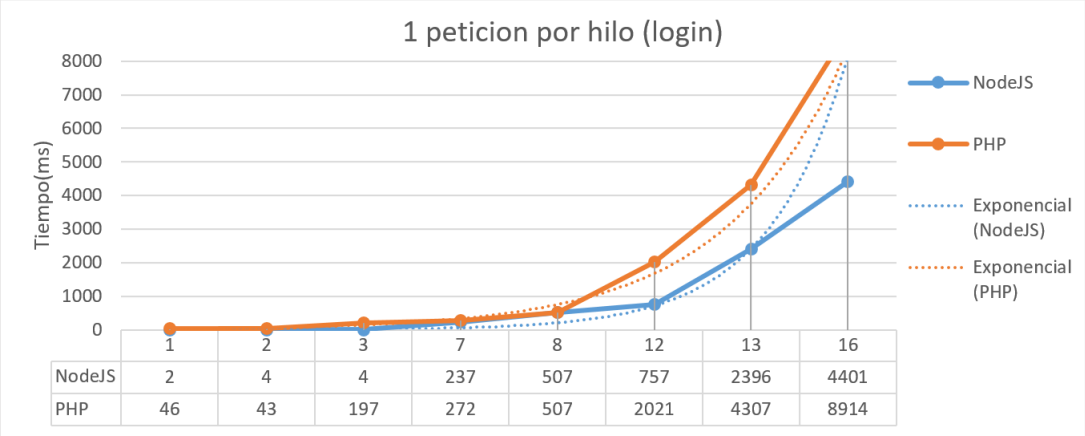


Figura 8.3: Grafica 1 petición por hilo (login)

La siguiente grafica nos muestra los mismos datos que el primer gráfico, pero, ahora en una escala exponencial, esta escala nos ayuda a apreciar mejor la diferencia de tiempos entre Node y PHP.

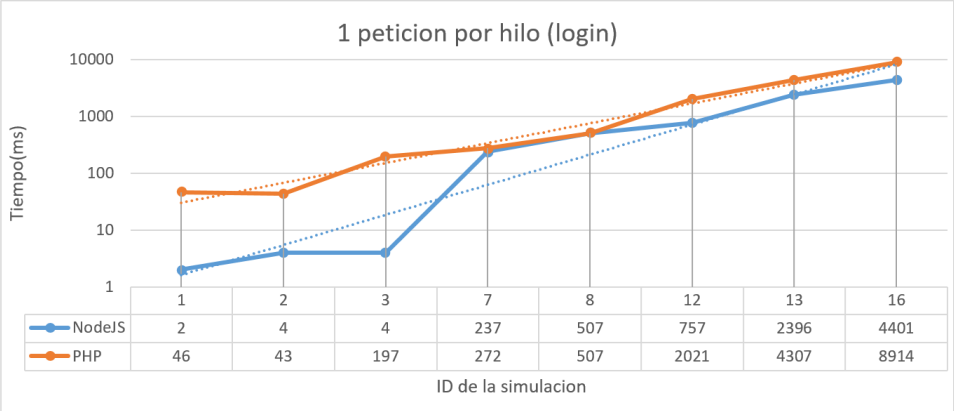


Figura 8.4: Grafica eje algorítmico 1 petición por hilo (login)

Las siguientes tres graficas muestra el rendimiento de los servidores cuando los hilos hacen más de una petición al servidor. Debido al flujo de datos de cada una de las alternativas es aquí donde vemos más diferencia a favor de la comunicación asíncrona de Node.JS.

ID de la prueba	Cantidad de hilos	Número de peticiones
4	10	10
9	100	10
14	1000	10

Tabla 8.4: Tabla para la grafica 8.5

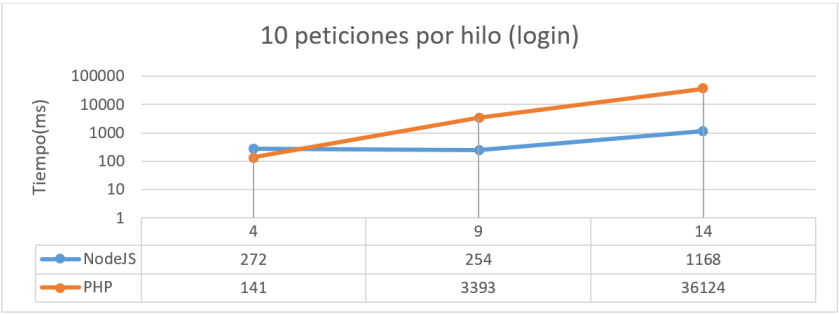


Figura 8.5: Grafica 10 peticiónes por hilo (login)

ID de la prueba	Cantidad de hilos	Número de peticiones
5	10	100
10	100	100
15	1000	100

Tabla 8.5: Tabla para la grafica 8.6

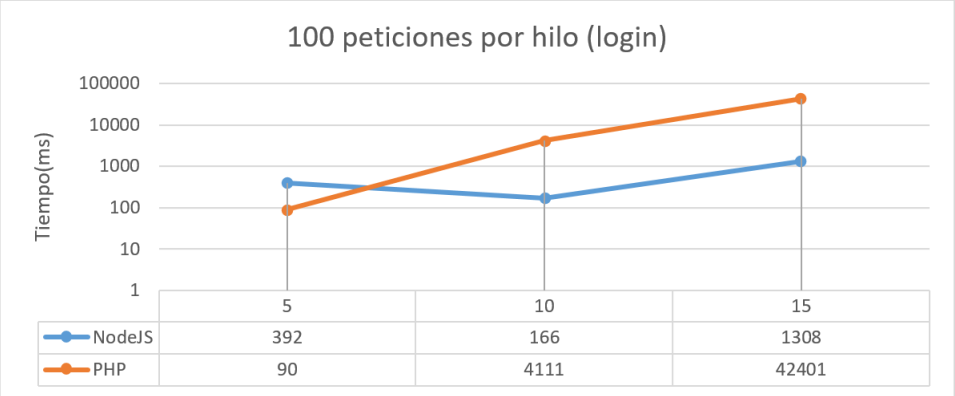


Figura 8.6: Grafica 100 peticiónes por hilo (login)

ID de la prueba	Cantidad de hilos	Número de peticiones
6	10	1000
11	100	1000

Tabla 8.6: Tabla para la grafica 8.7

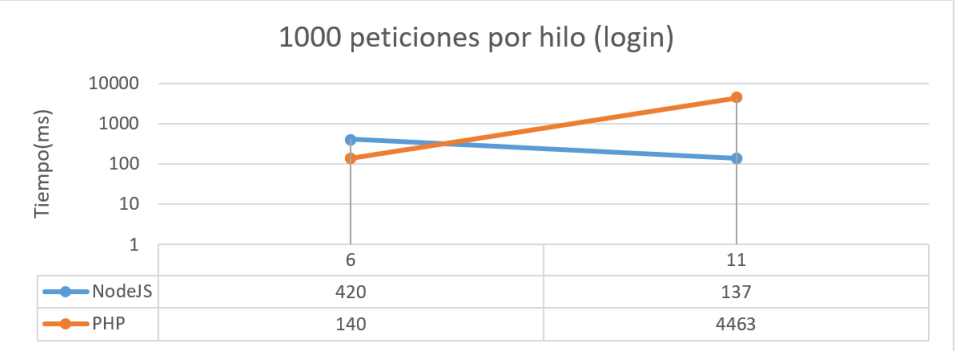
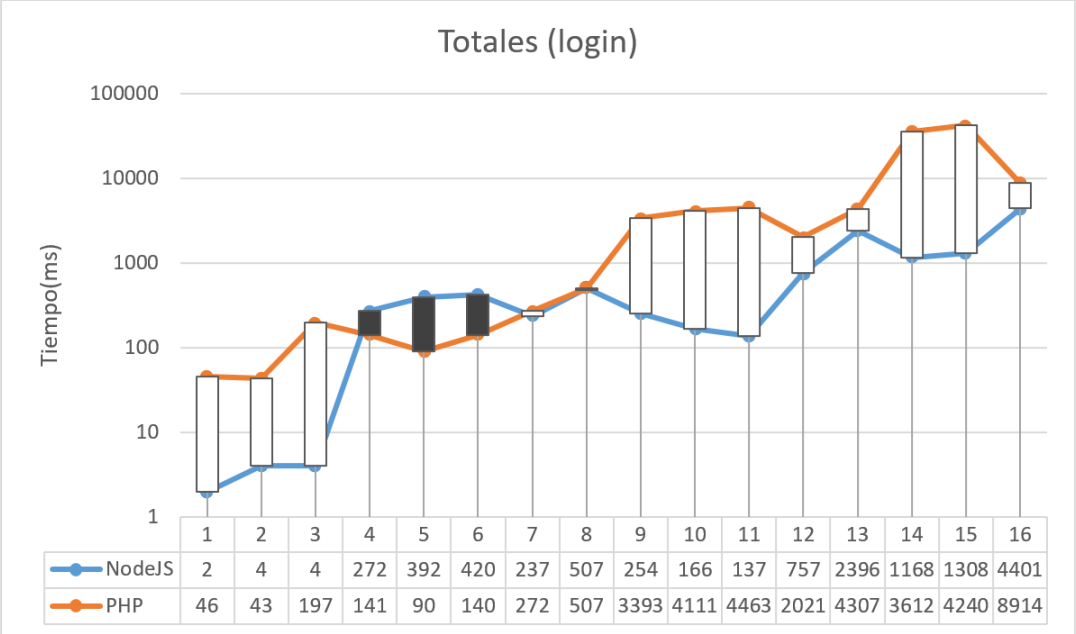


Figura 8.7: Grafica 1000 peticiónes por hilo (login)

El siguente grafico nos muestra todas las pruebas contra los servidores haciendo peticiones del tipo de login. Se ve claramente como el servidor en Node.JS es el que tiene un tiempo de respuesta optimo ya que PHP solo mejora para las pruebas 4, 5 y 6.

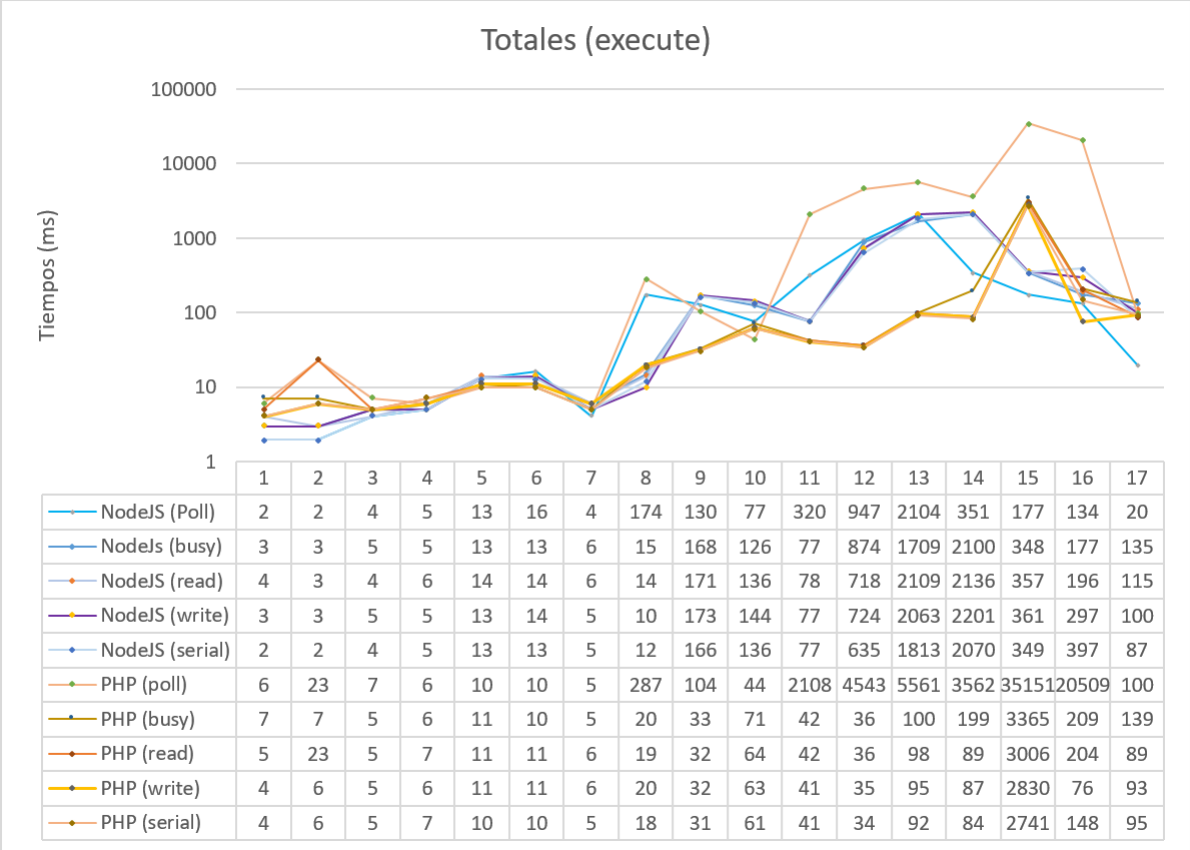


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
NodeJS	2	4	4	272	392	420	237	507	254	166	137	757	2396	1168	1308	4401
PHP	46	43	197	141	90	140	272	507	3393	4111	4463	2021	4307	3612	4240	8914

Figura 8.8: Grafica de todas las pruebas del tipo login

8.7.2. Graficas execute

Al igual que en las peticiones del tipo login, para el flujo de execute están todos los datos en el Anexo-5. El comportamiento del servidor es el mismo, con lo que no ha sido necesario obtener más gráficas en este apartado.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
NodeJS (Poll)	2	2	4	5	13	16	4	174	130	77	320	947	2104	351	177	134	20
NodeJs (busy)	3	3	5	5	13	13	6	15	168	126	77	874	1709	2100	348	177	135
NodeJS (read)	4	3	4	6	14	14	6	14	171	136	78	718	2109	2136	357	196	115
NodeJS (write)	3	3	5	5	13	14	5	10	173	144	77	724	2063	2201	361	297	100
NodeJS (serial)	2	2	4	5	13	13	5	12	166	136	77	635	1813	2070	349	397	87
PHP (poll)	6	23	7	6	10	10	5	287	104	44	2108	4543	5561	3562	35151	20509	100
PHP (busy)	7	7	5	6	11	10	5	20	33	71	42	36	100	199	3365	209	139
PHP (read)	5	23	5	7	11	11	6	19	32	64	42	36	98	89	3006	204	89
PHP (write)	4	6	5	6	11	11	6	20	32	63	41	35	95	87	2830	76	93
PHP (serial)	4	6	5	7	10	10	5	18	31	61	41	34	92	84	2741	148	95

Figura 8.9: Grafica de todas las pruebas del tipo execute

8.8. Valoración

Como ya hemos visto cuando analizábamos PHP, este lenguaje crea un nuevo hilo por cada conexión cliente-servidor, esto funciona bien para pocas conexiones, pero crear nuevos hilos tiene un coste, así como la realización de cambios de contexto y esto ha quedado reflejado en las anteriores tablas, es decir, PHP incrementa el tiempo de respuesta frente a Node.JS.

Por lo contrario, uno de los puntos fuertes de Node.JS es su capacidad de mantener muchas conexiones abiertas y esperando. Node.JS se programa sobre un solo hilo. Solo si en el sistema existe una operación bloqueante, Node.JS creará entonces otro hilo en segundo plano, pero no lo hará sistemáticamente por cada conexión como lo haría PHP. Node.JS puede mantener tantas conexiones como número máximo de archivos descriptores (sockets) son soportados por el sistema, aunque en la realidad la cifra depende de muchos factores, como la cantidad de información que se esté distribuyendo a los clientes.

9. CAPÍTULO

Conclusiones

En este capítulo atenderemos a las conclusiones obtenidas una vez terminado el proyecto. En primer lugar se hará una valoración del trabajo realizado. En segundo lugar se hará una reflexión sobre las lecciones aprendidas y por último se darán unas posibles líneas de trabajo futuras.

Conclusiones
1. Valoración de los objetivos 2. Valoración personal 3. Lecciones aprendidas 4. Posibles líneas de trabajo

9.1. Valoración de los objetivos

En primer lugar he tenido la oportunidad de realizar un análisis sobre diferentes tecnologías emergentes. Esto me ha ayudado para poder situarme en un mundo donde saber porque se toma cada decisión es muy importante en la vida de un desarrollador. Un análisis como el que hemos visto en anteriores capítulos es costoso, pero, me ha beneficiado a la hora de desarrollar el sistema, ya que conocía cuales eran las herramientas adecuadas para cada tarea.

Centrándome un poco más en la parte del desarrollo, he conseguido desarrollar dos sistemas para la comunicación con el lector de tarjetas. En un primer momento he tenido que gestionar un proceso de formación en ambas tecnologías, para así, más tarde, poder afrontar el proceso de diseño y desarrollo de la forma más óptima posible, quizás el proceso de aprendizaje ha sido más costoso de lo esperado ya que no tenía ningún tipo de experiencia en proyectos similares. En cuanto a los objetivos que se especificaron en el alcance inicial, todos se han podido completar de manera satisfactoria. Además, he perdido el posible miedo a trabajar en proyectos o tecnologías que no he estudiado durante la carrera. En cuanto a la parte más negativa, cabe indicar que los procesos del ciclo de vida del software y su cronología, no han estado tan diferenciados ni ordenados como se ha mostrado en la documentación. La fase de toma de requisitos no sólo duró más de lo esperado, sino que las modificaciones de alguno de ellos se entremezclaron con la fase de desarrollo, haciendo que me descentre del proyecto.

Para finalizar mencionar que la parte de valoración de las pruebas me ha parecido muy interesante, me ha ayudado a asentar los conceptos teóricos que se habían estudiado en capítulos anteriores, dándome una visión más realista de un proyecto que va a salir a mercado.

9.2. Valoración personal

En primer lugar volver a dar las gracias a la empresa ATELEI y a todos sus miembros por ofrecerme la oportunidad de trabajar con ellos. Valoro como muy positiva la experiencia de hacer el proyecto fin de grado en una empresa y se la recomiendo a cualquier alumno. Escribir estas últimas líneas del proyecto me provoca una enorme satisfacción tanto a nivel personal, porque sé que con dedicación y esfuerzo todo se supera, como a nivel profesional, por todo el trabajo que ha supuesto la realización del proyecto durante

todo este tiempo sin olvidarme, por supuesto, de todas y cada una de las asignaturas de la carrera, que, unas más que otras, todas han contribuido a que este proyecto sea hoy una realidad.

Teniendo en cuenta los principales objetivos del proyecto, el tiempo invertido realizándolo y los resultados obtenidos, me encuentro bastante satisfecho con el devenir del proyecto.

9.3. Lecciones aprendidas

Durante el este proyecto se han obtenido diferentes lecciones que conviene mencionar y tener en cuenta en cualquier proyecto que se vaya a afrontar, tanto en el ámbito universitario como en un futuro dentro de una empresa.

- **Un buen entorno de trabajo**. Durante estos últimos meses he aprendido a valorar lo importante que es convivir en un entorno laboral adecuado. Tener un ambiente de trabajo saludable resultaba en un extra de motivación para trabajar día a día y afrontar el proyecto con más ganas.

- **No menospreciar los riesgos**. A la hora de afrontar un riesgo en el proyecto es importante tener en cuenta todos aquellos riesgos que se puedan identificar aunque la probabilidad sea mínima e intentar establecer estrategias para minimizarlos y evitar las posibles consecuencias.

- **La formación es una parte importante**. Más allá de lo obvio, el hecho de haber tomado con calma el tema de la formación, haber previsto unos meses de formación ha sido de gran ayuda. Este proceso resultó en unos cimientos sólidos que han servido de base para desarrollar un producto con mejores prestaciones.

- **Combinar vida laboral y universidad**. Estos últimos meses me he embarcado un reto profesional de una exigencia alta. Tuve la oportunidad de empezar a trabajar en una de las empresas punteras de España en el desarrollo de software y e-commerce. Empecé a trabajar sin llegar a haber cerrado el proyecto de fin de grado. Combinar este tipo de trabajo en el que las horas pasan más rápido que nunca y un proyecto universitario no me ha resultado fácil. He tenido semanas en las que no me separaba del ordenador en todo el día y eso ha llegado a causarme estrés e incluso algún problema en el ámbito privado. Aun así, estoy contento con poder haber llegado

al final de este proceso ya que me ha ayudado a comprender como afrontar las dificultades diarias.

9.4. Posibles líneas de trabajo

Con este apartado el proyecto ha llegado a su fin, por lo que en el siguiente listado se recogen las posibles propuestas de mejora. Todo ello con fin de que en un futuro se dé continuidad al trabajo realizado.

- Se podría añadir algún tipo de interfaz web para poder configurar el servidor o el lector de tarjetas.
- Se podrían añadir nuevos tipos de acciones al servidor, por ejemplo, ACTION=7 para llamadas de configuración del lector de tarjetas.
- Se podría mejorar la base de datos. El motor de base de datos actual almacena las tareas en un archivo único, siendo más ligero y rápido, pero en principio no está preparado para un gran volumen de datos. Podría hacerse distribuida si el volumen creciese mucho. (PostgreSQL)
- Se podría exportar toda la información contenida en la base de datos en formato XML, esto sería de gran ayuda para la depuración de errores.
- Cabe mencionar que un inconveniente de Node.JS es que debido a su arquitectura sólo podrá usar una CPU. Aunque, un método para usar múltiples núcleos sería iniciar múltiples instancias de Node.JS en el servidor y poner un balanceador de carga delante de ellos.

Bibliografía

[1] Ethan Brown, 2014. *Web Devlopment with Node & Express*, O'Reilly, Leveraging the javascript stack.

[2] George Ornbo, 2013. *Node.js*, Sams, Anaya multimedia.

[3] Bayo Erinle, 2013. *Performance Testing with JMeter*, Packt publishing, Quick answers to common problem.

[4] Leslie Lamport, 1985. *LaTeX—A Document Preparation System—User's Guide and Reference Manual*, Addision-Wesley, Reading.

Páginas Web

- Buenas prácticas para el Diseño de una API RESTful. https://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-restful-pragmatica/
- REST vs Web Services http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf
- Curso Node.JS http://jonmircha.github.io/slides-nodejs/#/
- Autenticación en un servidor usando Node.js https://carlosazaustre.es/blog/autenticacion-con
- Node.js y Websockets https://manuais.iessanclemente.net/index.php/Node.js_y_Websockets#.C2.BFQu.C3.A9_diferencias_tiene_Node.js_respecto_a_Apache_u_otros_servidores_web.3F
- Programación Asíncrona en Node JS http://es.slideshare.net/jvelez77/presentacion-3526491
- Introducción a Codeigniter http://digitta.com/2009/01/empezando-con-codeigniter.html

- PHP Authorization with JWT (JSON Web Tokens) https://www.sitepoint.com/php-authorization-jwt-j
- PHP-JWT https://github.com/firebase/php-jwt/
- Node.JS vs. CodeIgniter http://vschart.com/compare/node-js/vs/codeigniter
- Benchmarking Node.js - basic performance tests against Apache + PHP http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php
- Scraping – Nodejs Vs Php http://rojan.com.np/scraping-nodejs-vs-php/
- JMeter http://jmeter.apache.org/

  http://www.javaworld.com/article/2071953/testing-debugging/jmeter-tips.html
- LaTeX Table Generator http://www.tablesgenerator.com/

Conferencias

- Node.js Live:*Gareth Ellis, Node.js Community Benchmarking Efforts*. https://www.youtube.com/watch?v=9shnujei8VU
- Demo ESP8266 and web server https://www.youtube.com/watch?v=cBnmrtdHWZg

Anexos

A. ANEXO

Acrónimos

API: (Application Programing Interface) Interfaz de programación de aplicaciones como conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizada por otro software.

El módulo ESP8266: es un microcontrolador que contiene la prestación de realizar comunicaciones inalámbricas mediante su conectividad Wi-Fi. Encoder: En este contexto se refiere al dispositivo lector de la información contenida en las tarjetas.

Transceiver: Es un dispositivo que cuenta con un transmisor y un receptor que comparten parte de la circuitería o se encuentran dentro de la misma caja. HTTP: (Hypertext Transfer Protocol) Protocolo de transferencia de hipertexto, es el protocolo de comunicación que permite las transferencias de información en la World Wide Web (WWW).

IDE: (Integrated Drive Enviroment) Entorno de desarrollo integrado, es una aplicación informática que proporciona servicios integrales para facilitar al desarrollador o programador el desarrollo de software.

JSON: (JavaScript Object Notation) Formato de texto ligero para el intercambio de datos.

REST: (Representational State Transfer) Transferencia de Estado Representacional, es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.

WS:(Web Services)

XML: (eXtensible Markup Language) Es un meta-lenguaje que permite definir lenguajes de marcas.

SOAP: (Simple Object Access Protocol)

WSDL: (Web Services Description Language)

UDDI: (Universal Description, Discovery and Integration)

URI: (Uniform Resource Identifier) Es una cadena de caracteres que identifica los recursos de una red de forma unívoca.

CRUD: (Create, Read, Update and Delete) Se usa para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.

MIME: (Multipurpose Internet Mail Extensions)

URLs: (Uniform Resource Locator)

FTP: (File Transfer Protoco) Protocolo de red para la transferencia de archivos entre sistemas conectados a una red.

IDE: (Integrated Development Environment)

EDT: (Estructura de Descomposición de Trabajo)

SQL: (Structured Query Language)

ERP: (Enterprise Resource Planning)

SDK: (Software Development Kit)

SSL: (Secure Sockets Layer) Capa de puertos seguros, es un protocolo criptográfico que proporciona comunicaciones seguras por una red.

TCP: (Transmission Control Protocol) ESs uno de los protocolos fundamentales en Internet para que puedan comunicarse de forma segura.

W3C: (World Wide Web Consortium) Es un consorcio internacional que genera recomendaciones y estándares que aseguran el crecimiento de la World Wide Web a largo plazo.

B. ANEXO

Manual de instalación Node.JS

B.1. Descargar la aplicación

Mediante el enlace que proporcionara la empresa ATELEI se podrá descargar la aplicación.

B.2. Ejecutar el script de instalación y seguir los pasos de ejeción

En este punto es importante asegurarnos que no tenemos ningún recurso consumiendo el puerto 443 del servidor, debido a que la aplicación que vamos a instalar va intentar comunicarse por el puerto 443 como hemos visto en capítulos anteriores.

Mediante este script obtendremos varios elementos:

- mysql-server, mysql-client y php5-mysql
- (Creación inicial de la base de datos)
- node
- npm

```
#!/usr/bin/env bash
echo ""
echo "Welcome to Atelei-NodeJS installation script (press intro to continue) "
read any
user=$(whoami)
if [ "$user" == "root" ]; then
    # actualizar librerias OS
    apt-get update
    echo "update OK"

    #instalar MYSQL
    apt-get install mysql-server mysql-client php5-mysql -y
    echo "install OK MYSQL"

    # bd copy
    echo "Write MYSQL user:"
    read user
    echo "Write MYSQL password:"
    read pass
    mysql --host=localhost --user=$user --password=$pass < nodejsAPIRest.sql

    #instalar node y npm
    apt-get install nodejs-legacy -y

    node=$(node -v)
    echo "install OK $node"

    apt-get install npm -y
    npm=$(npm -v)
    echo "install OK $npm"

    echo "Are you in the project directory? (yes/no) "
    read cue
    if [ "$cue" == "yes" ]; then
        # descargar librerias ( direct .json)
        npm install
    else
       echo "Execute installation in the project directory"
       exit
    fi
    echo "install OK!"
else
   echo "You are not root, execute command with root mode"
   exit
fi
```

C. ANEXO

Manual de instalación PHP

C.1. Descargar la aplicación

Mediante el enlace que proporcionara la empresa ATELEI se podrá descargar la aplicación.

C.2. Ejecutar el script de instalación y seguir los pasos de ejeción

En este punto es importante asegurarnos que no tenemos ningún recurso consumiendo el puerto 443 de nuestro servidor, ya que la aplicación va intentar comunicarse por el 443 como hemos visto en capítulos anteriores.

Mediante este script obtendremos varios elementos:

- apache2
- php5
- (Creación inicial de la base de datos)
- (Asigna permisos necesarios a la app)

```
#!/usr/bin/env bash
echo ""
echo "Welcome to Atelei-PHP installation script (press intro to continue) "
read any
user=$(whoami)
if [ "$user" == "root" ]; then
    # actualizar librerias OS
    apt-get update
    echo "update OK"

    #Instalar Apache & PHP
    #groupadd www-data
    #usermod -a -G www-data www-data
    apt-get install apache2 -y
    apt-get install php5 -y
    #Libreries
    #apt-get install libapache2-mod-php5 libapache2-mod-perl2 php5 php5-cli php5-common php5-curl
      php5-dev php5-gd php5-imap php5-ldap php5-mhash php5-mysql php5-odbc -y

    #instalar MYSQL
    apt-get install mysql-server mysql-client php5-mysql -y
    echo "install OK mysql"

    # bd copy
    echo "Write your MYSQL user:"
    read user
    echo "Write your MYSQL password:"
    read pass
    mysql --host=localhost --user=$user --password=$pass < APIRest.sql

    echo "Are you in the project directory? (yes/no) "
    read cue
    if [ "$cue" == "yes" ]; then
        # descargar librerias ( direct .json)
        mkdir /var/www/html/codeigniter
        mv * /var/www/html/codeigniter
    else
       echo "Execute installation in the project directory"
       exit
    fi
    echo "Write your system user name:"
    read username
    chown -R $username:www-data /var/www/html/codeigniter
    chmod -R 755 /var/www/html/codeigniter
    echo "install OK!"
else
   echo "You are not root, execute command with root mode"
   exit
fi
```

D. ANEXO

Pruebas validación de servidor

Como ya hemos visto en el capítulo de la validación del servidor en este apartado podemos encontrarnos la tabla que he segundo para depurar las comunicaciones del servidor. En ella quedan reflejadas la petición que simula cURL como si fuese el encoder, la respuesta esperada y la respuesta de nuestro servidor para cada caso de prueba.

ID			DESCRIPCIÓN	RESULTADO ESPERADO				RESULTADO
				valores DB		saveDB		
1	login		User or param incorrect (no DB)	{"STATUS":6}				{"STATUS":6}
2	login		Falta algun parametro	{"STATUS":6}				{"STATUS":6}
3	login		Nombre parametros diferentes	{"STATUS":6}				{"STATUS":6}
4	login		Sin parametros o parametros null	{"STATUS":6}				{"STATUS":6}
5	login		Parametros correctos (DB)	{ "STATUS": 1, "TOKEN": "..." }		user+token+date		{ "STATUS": 1, "TOKEN": "..." }
6	execute		Sin parametros	{ STATUS: 2}				{ STATUS: 2}
7	execute		Idencoder or token or status null	{ STATUS: 2}				{ STATUS: 2}
8	execute		Token caducado	{ STATUS: 2}				{ STATUS: 2}
9	execute		Token incorrect – user incorrect (DB)	{ STATUS: 2}				{ STATUS: 2}
10	execute		Token incorrect – user ok (DB)	{ STATUS: 2}				{ STATUS: 2}
11	execute		Token ok -- user incorrect (DB)	{ STATUS: 2}				{ STATUS: 2}
12	execute	poll	Token ok, user ok – status !=1	actions pending = 0			{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
				actions pending > 0		actionPoll.attemps -1	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
13	execute	poll	Token ok, user ok – status=1	actions pending = 0			{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
				actions pending > 0		actionPoll.attemps -1	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
14	execute	busy	T&u ok – idrequest=70	idrequest no DB			{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
				idrequest.SITUATION=CANCEL			{"STATUS": 1,"IDREQUEST": 70,"ACTION": 6,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 70,"ACTION": 6,"DATA": "", "UID": ""}
				else			{"STATUS": 1,"IDREQUEST": 70,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 70,"ACTION": 1,"DATA": "", "UID": ""}
15	execute	busy	T&u ok – idrequest=null	actions pending = 0			{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
				actions pending > 0			{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
16	execute	busy	T&u ok – idrequest=noDB	actions pending = 0			{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
				actions pending > 0			{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
17	execute	read	Token wrong	{ STATUS: 2}				{ STATUS: 2}
18	execute	read	T&u&staus ok -- idrequest=55	idrequest no DB	actions pending = 0	data, uid, status=4	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				idrequest.SITUATION=CANCEL	actions pending = 0	data, uid, status=4	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				else	actions pending = 0	data, uid, status=4	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
19	execute	read	T&u ok - staus wrong - idrequest=100	idrequest no DB	actions pending = 0	-	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				idrequest.SITUATION=NO_OK	actions pending = 0	-	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				idrequest.SITUATION=CANCEL			{"STATUS": 1,"IDREQUEST": 100,"ACTION": 6,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 100,"ACTION": 6,"DATA": "", "UID": ""}
				else		(if attemps =0)status= 5	{"STATUS": 1,"IDREQUEST": 100,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 100,"ACTION": 1,"DATA": "", "UID": ""}
20	execute	write	T&st ok - user wrong - idrequest=100	{ STATUS: 2}				{ STATUS: 2}
21	execute	write	T&u&staus ok -- idrequest=55	idrequest no DB	actions pending = 0	status=4	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				idrequest.SITUATION=CANCEL	actions pending = 0	status=4	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				else	actions pending = 0	status=4	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
22	execute	write	T&u ok - staus wrong - idrequest=100	idrequest no DB	actions pending = 0	-	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				idrequest.SITUATION=NO_OK	actions pending = 0	-	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 0,"ACTION": 0,"DATA": "", "UID": ""}
					actions pending > 0		{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 1,"ACTION": 1,"DATA": "", "UID": ""}
				idrequest.SITUATION=CANCEL			{"STATUS": 1,"IDREQUEST": 100,"ACTION": 6,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 100,"ACTION": 6,"DATA": "", "UID": ""}
				else		(if attemps =0)status= 5	{"STATUS": 1,"IDREQUEST": 100,"ACTION": 1,"DATA": "", "UID": ""}	{"STATUS": 1,"IDREQUEST": 100,"ACTION": 1,"DATA": "", "UID": ""}
20	-		parametros incorrectos	{"STATUS":6}				{"STATUS":6}

Login incorrecto | Token incorrecto | Login OK | Primera accion pendiente | No hay acciones pendientes | Misma accion idreq pero con action cancelada | Misma accion idreq | Misma accion idreq (intentos -1)

E. ANEXO

Benchmarking

E.1. Node.JS

En la tabla que podemos encontrar en la siguiente página vemos un registro de la validación del servidor que se llevó a cabo para comprobar el correcto funcionamiento.

Node												
N. Prueba	Numero de Hilos	N. de peticiones	Peticion		Tiempos de respuesta (ms)					% Er	Th	Kb/s
			Ruta	Muestras	Media	Mediana	Linea 90 %	Min	Max			
1	1	1	/login	1	2	2	2	2	2	0.0	500	177
2	5	1	/login	5	4	5	6	3	6	0.0	6	2
3	10	1	/login	10	4	3	4	2	19	0.0	11	4
4	10	10	/login	100	272	267	423	14	481	0.0	25	9
5	10	100	/login	1000	392	362	499	18	912	0.0	25	9
6	10	1000	/login	10000	420	399	526	7	1288	0.0	23	8
7	50	1	/login	50	237	164	584	4	666	0.0	33	12
8	100	1	/login	100	507	578	888	18	948	0.0	64	23
9	100	10	/login	1000	254	65	181	38	3207	0.0	195	73
10	100	100	/login	10000	166	106	189	31	6468	0.0	399	150
11	100	1000	/login	100000	137	123	198	32	7121	0.0	373	141
12	500	1	/login	500	757	798	1079	641	1188	0.0	232	88
13	1000	1	/login	1000	2396	2342	3470	40	7082	0.0	120	45
14	1000	10	/login	10000	1168	177	578	36	31216	0.0	291	110
15	1000	100	/login	100000	1308	144	288	30	127339	611	352	136
16	5000	1	/login	5000	4401	1568	10157	80	37293	0.0	75	28

Node												
N. Prueba	Numero de Hilos	N. de peticiones	Peticion		Tiempos de respuesta (ms)					% Er	Th	Kb/s
			Ruta	Muestras	Media	Mediana	Linea 90 %	Min	Max			
1	1	1	/execute.poll	1	2	2	2	2	2	0.0	500	138
1	1	1	/execute.busy	1	3	3	3	3	3	0.0	333	92
1	1	1	/execute.read	1	4	4	4	4	4	0.0	250	69
1	1	1	/execute.write	1	3	3	3	3	3	0.0	333	92
1	1	1	/execute.serial	1	2	2	2	2	2	0.0	500	138
2	5	1	/execute.poll	5	2	2	2	2	4	0.0	6	2
2	5	1	/execute.busy	5	3	3	4	2	6	0.0	6	2
2	5	1	/execute.read	5	3	3	4	3	6	0.0	6	2
2	5	1	/execute.write	5	3	3	3	3	4	0.0	6	2
2	5	1	/execute.serial	5	2	3	3	2	3	0.0	6	2
3	10	1	/execute.poll	10	4	3	5	3	18	0.0	11	3
3	10	1	/execute.busy	10	5	4	6	3	17	0.0	11	3
3	10	1	/execute.read	10	4	5	6	4	6	0.0	11	3
3	10	1	/execute.write	10	5	5	6	4	6	0.0	11	3
3	10	1	/execute.serial	10	4	4	5	3	6	0.0	11	3
4	10	10	/execute.poll	100	5	5	11	2	21	0.0	88	24
4	10	10	/execute.busy	100	5	5	9	2	18	0.0	89	26
4	10	10	/execute.read	100	6	5	8	2	28	0.0	89	24
4	10	10	/execute.write	100	5	5	8	2	14	0.0	89	24
4	10	10	/execute.serial	100	5	5	8	2	14	0.0	89	25
5	10	100	/execute.poll	1000	13	13	20	2	54	0.0	74	20
5	10	100	/execute.busy	1000	13	13	21	2	50	0.0	74	22
5	10	100	/execute.read	1000	14	14	21	3	53	0.0	74	20
5	10	100	/execute.write	1000	13	14	21	3	52	0.0	74	20
5	10	100	/execute.serial	1000	13	13	20	2	37	0.0	74	20
6	10	1000	/execute.poll	10000	16	14	22	2	2679	0.0	66	18
6	10	1000	/execute.busy	10000	13	14	22	2	81	0.0	67	20
6	10	1000	/execute.read	10000	14	15	23	2	86	0.0	67	19
6	10	1000	/execute.write	10000	14	14	23	2	82	0.0	67	19
6	10	1000	/execute.serial	10000	13	14	22	2	73	0.0	67	19
7	50	1	/execute.poll	50	4	3	10	2	17	0.0	45	12
7	50	1	/execute.busy	50	6	5	12	3	25	0.0	46	13
7	50	1	/execute.read	50	6	5	12	3	23	0.0	46	13
7	50	1	/execute.write	50	5	5	10	3	15	0.0	47	13
7	50	1	/execute.serial	50	5	4	11	3	23	0.0	47	13
8	100	1	/execute.poll	100	20	20	44	4	160	0.0	65	18
8	100	1	/execute.busy	100	15	12	27	4	56	0.0	65	18
8	100	1	/execute.read	100	14	10	29	4	47	0.0	65	18
8	100	1	/execute.write	100	10	8	28	4	61	0.0	65	18
8	100	1	/execute.serial	100	12	8	25	4	48	0.0	65	18
9	100	10	/execute.poll	1000	174	176	249	5	343	0.0	79	22
9	100	10	/execute.busy	1000	168	171	240	7	384	0.0	78	23
9	100	10	/execute.read	1000	171	174	247	8	391	0.0	77	21
9	100	10	/execute.write	1000	173	174	255	6	388	0.0	76	21
9	100	10	/execute.serial	1000	166	167	247	5	350	0.0	76	21

Node												
N. Prueba	Numero de Hilos	N. de peticiones	Peticion		Tiempos de respuesta (ms)					% Er	Th	Kb/s
			Ruta	Muestras	Media	Mediana	Linea 90 %	Min	Max			
10	100	100	/execute.poll	10000	130	116	210	2	809	0.0	9	25
10	100	100	/execute.busy	10000	126	115	206	2	313	0.0	9	26
10	100	100	/execute.read	10000	136	121	222	2	324	0.0	9	25
10	100	100	/execute.write	10000	144	127	236	2	361	0.0	9	25
10	100	100	/execute.serial	10000	136	123	226	2	311	0.0	9	25
11	100	1000	/execute.poll	100000	77	75	149	2	1945	0.0	26	7
11	100	1000	/execute.busy	100000	77	75	149	2	1319	0.0	26	8
11	100	1000	/execute.read	100000	78	75	149	2	1940	0.0	26	7
11	100	1000	/execute.write	100000	77	75	148	2	1940	0.0	26	7
11	100	1000	/execute.serial	100000	77	75	148	2	1946	0.0	26	7
12	500	1	/execute.poll	500	320	98	925	2	964	0.0	151	42
12	500	1	/execute.busy	500	874	874	1396	4	1492	0.0	113	31
12	500	1	/execute.read	500	718	789	901	10	971	0.0	97	27
12	500	1	/execute.write	500	724	849	895	4	970	0.0	91	25
12	500	1	/execute.serial	500	635	777	888	15	949	0.0	89	25
13	1000	1	/execute.poll	1000	947	904	1862	3	2680	16	146	45
13	1000	1	/execute.busy	1000	1709	1770	2852	6	2960	0.0	105	31
13	1000	1	/execute.read	1000	2109	2603	2783	127	2853	0.0	86	24
13	1000	1	/execute.write	1000	2063	2152	2741	681	2959	0.0	81	22
13	1000	1	/execute.serial	1000	1813	1667	2803	371	2948	0.0	82	23
14	1000	10	/execute.poll	10000	2104	2289	3090	2	3990	0.0	79	22
14	1000	10	/execute.busy	10000	2100	2391	3017	2	3983	0.0	77	22
14	1000	10	/execute.read	10000	2136	2368	3144	3	3989	0.0	76	21
14	1000	10	/execute.write	10000	2201	2459	3294	2	3985	0.0	75	21
14	1000	10	/execute.serial	10000	2070	2293	3144	3	3967	0.0	74	20
15	1000	100	/execute.poll	100000	351	111	1037	2	3395	0.0	28	8
15	1000	100	/execute.busy	100000	348	109	1062	2	3390	0.0	28	8
15	1000	100	/execute.read	100000	357	116	1077	2	3361	0.0	28	8
15	1000	100	/execute.write	100000	361	113	1063	2	3421	0.0	28	8
15	1000	100	/execute.serial	100000	349	109	1022	2	3406	0.0	28	8
16	5000	1	/execute.poll	5000	177	17	434	0	2837	7.976	210	337
16	5001	1	/execute.busy	5000	177	22	614	0	3004	7.976	235	386
16	5002	1	/execute.read	5000	196	14	744	0	2436	7.838	248	408
16	5003	1	/execute.write	5000	297	32	1006	0	2090	6.572	329	471
16	5004	1	/execute.serial	5000	397	87	1068	0	1552	503	479	574
17	10000	1	/execute.poll	10000	134	6	352	0	1774	8.835	402	707
17	10000	1	/execute.busy	10000	135	4	739	0	1755	8.493	317	543
17	10000	1	/execute.read	10000	115	4	421	0	1722	8.227	255	428
17	10000	1	/execute.write	10000	100	3	344	0	1676	7.829	212	341
17	10000	1	/execute.serial	10000	87	3	330	0	1652	7.194	186	280

E.2. PHP

En la tabla que podemos encontrar en la siguiente página vemos un registro de la validación del servidor que se llevó a cabo para comprobar el correcto funcionamiento.

PHP										
N. de peticiones	Peticion		Tiempos de respuesta (ms)					% Error	Rendimiento/sec	Kb/s
	Ruta	Muestras	Media	Mediana	Linea 90 %	Min	Max			
1	/login	1	46	46	46	46	46	0.0	22	8
1	/login	5	43	35	43	34	69	0.0	6	2
1	/login	10	197	142	210	107	218	0.0	8	3
10	/login	100	141	106	230	42	772	0.0	37	14
100	/login	1000	90	74	158	37	367	0.0	85	32
1000	/login	10000	140	121	228	39	1083	0.0	67	25
1	/login	50	272	270	290	241	351	0.0	44	17
1	/login	100	507	578	888	18	948	0.0	64	23
10	/login	1000	3393	3874	4433	28	4636	0.0	26	9
100	/login	10000	4111	4115	4544	20	5339	0.0	24	8
1000	/login	100000	4463	4422	4963	23	9582	0.0	22	8
1	/login	500	2021	938	5756	689	6425	0.0	65	23
1	/login	1000	4307	2004	10733	30	12061	0.0	66	24
10	/login	10000	36124	41818	43805	36	45116	0.0	25	9
100	/login	100000	42401	42951	44106	295	47450	0.0	23	8
1	/login	5000	8914	3323	28823	0	100876	5.626	97	121

PHP													
N. Prueba	N. hilos	N. de peticiones	Peticion		Tiempos de respuesta (ms)					% Error	Rendimiento/sec	Kb/s
			Ruta	Muestras	Media	Mediana	Linea 90 %	Min	Max			
1	1	1	e.poll	1	6	6	6	6	6	0.0	167	47
1	1	1	e.busy	1	7	7	7	7	7	0.0	143	40
1	1	1	e.read	1	5	5	5	5	5	0.0	200	56
1	1	1	e.write	1	4	4	4	4	4	0.0	250	70
1	1	1	e.serial	1	4	4	4	4	4	0.0	250	70
2	5	1	e.poll	5	23	24	24	23	25	0.0	5	1
2	5	1	e.busy	5	7	7	8	6	8	0.0	625	175
2	5	1	e.read	5	23	24	24	23	25	0.0	200	56
2	5	1	e.write	5	6	6	8	5	8	0.0	556	156
2	5	1	e.serial	5	6	6	7	6	8	0.0	500	140
3	10	1	e.poll	10	7	6	10	5	11	0.0	11	3
3	10	1	e.busy	10	5	5	6	4	6	0.0	11	3
3	10	1	e.read	10	5	5	6	4	6	0.0	11	3
3	10	1	e.write	10	5	5	6	4	8	0.0	11	3
3	10	1	e.serial	10	5	5	5	4	6	0.0	11	3
4	10	10	e.poll	100	6	6	11	3	20	0.0	83	23
4	10	10	e.busy	100	6	6	9	4	17	0.0	83	23
4	10	10	e.read	100	7	6	11	4	21	0.0	83	23
4	10	10	e.write	100	6	6	10	4	18	0.0	83	23
4	10	10	e.serial	100	7	6	11	4	22	0.0	83	23
5	10	100	e.poll	1000	10	9	18	4	163	0.0	69	19
5	10	100	e.busy	1000	11	10	19	4	137	0.0	69	19
5	10	100	e.read	1000	11	10	19	4	34	0.0	69	19
5	10	100	e.write	1000	11	10	18	4	36	0.0	69	19
5	10	100	e.serial	1000	10	9	18	4	47	0.0	69	19
6	10	1000	e.poll	10000	10	9	18	3	325	0.0	58	16
6	10	1000	e.busy	10000	10	10	18	4	86	0.0	58	16
6	10	1000	e.read	10000	11	10	18	4	87	0.0	58	16
6	10	1000	e.write	10000	11	10	18	4	89	0.0	58	16
6	10	1000	e.serial	10000	10	9	18	4	84	0.0	58	16
7	50	1	e.poll	50	5	5	7	3	8	0.0	50	14
7	50	1	e.busy	50	5	6	7	4	8	0.0	50	14
7	50	1	e.read	50	6	6	8	5	11	0.0	50	14
7	50	1	e.write	50	6	6	8	5	10	0.0	50	14
7	50	1	e.serial	50	5	5	7	4	7	0.0	51	14
8	100	1	e.poll	100	100	116	148	5	160	0.0	78	22
8	100	1	e.busy	100	20	18	30	5	56	0.0	78	22
8	100	1	e.read	100	19	17	31	7	47	0.0	77	22
8	100	1	e.write	100	20	17	34	6	61	0.0	78	22
8	100	1	e.serial	100	18	17	29	4	48	0.0	78	22
9	100	10	e.poll	1000	287	28	86	4	4998	0.0	128	36
9	100	10	e.busy	1000	33	28	63	5	146	0.0	128	36
9	100	10	e.read	1000	32	28	63	5	122	0.0	128	36
9	100	10	e.write	1000	32	27	63	4	143	0.0	128	36
9	100	10	e.serial	1000	31	26	60	3	124	0.0	129	36

PHP												
N. Prueba	N. hilos	N. de peticiones	Peticion		Tiempos de respuesta (ms)					% Error	Rendimiento/sec	Kb/s
			Ruta	Muestras	Media	Mediana	Linea 90 %	Min	Max			
10	100	100	e.poll	10000	104	47	140	4	7038	0.0	100	28
10	100	100	e.busy	10000	71	48	139	4	6536	0.0	100	28
10	100	100	e.read	10000	64	49	139	4	442	0.0	100	28
10	100	100	e.write	10000	63	48	138	4	555	0.0	100	28
10	100	100	e.serial	10000	61	46	137	4	523	0.0	100	28
11	100	1000	e.poll	100000	44	28	92	3	6790	0.0	28	78
11	100	1000	e.busy	100000	42	29	95	3	5846	0.0	28	78
11	100	1000	e.read	100000	42	29	96	3	1349	0.0	28	78
11	100	1000	e.write	100000	41	29	94	3	1348	0.0	28	78
11	100	1000	e.serial	100000	41	28	93	3	1348	0.0	28	78
12	500	1	e.poll	500	2108	1870	3807	5	4416	0.0	86	24
12	500	1	e.busy	500	36	31	74	5	176	0.0	85	24
12	500	1	e.read	500	36	30	71	5	165	0.0	84	24
12	500	1	e.write	500	35	30	72	6	174	0.0	84	23
12	500	1	e.serial	500	34	28	69	6	210	0.0	84	23
13	1000	1	e.poll	1000	4543	4278	8175	5	15038	0.0	60	17
13	1000	1	e.busy	1000	100	62	239	4	585	0.0	60	17
13	1000	1	e.read	1000	98	65	226	4	552	0.0	60	17
13	1000	1	e.write	1000	95	62	225	4	628	0.0	60	17
13	1000	1	e.serial	1000	92	55	230	4	561	0.0	60	17
14	1000	10	e.poll	10000	5561	65	342	4	127365	223	65	20
14	1000	10	e.busy	10000	199	61	221	4	13572	0.0	64	18
14	1000	10	e.read	10000	89	62	212	4	676	0.0	64	18
14	1000	10	e.write	10000	87	59	207	4	719	0.0	64	18
14	1000	10	e.serial	10000	84	57	201	3	706	0.0	65	18
15	1000	100	e.poll	100000	3562	27	134	3	407155	1.438	27	8
15	1000	100	e.busy	100000	3365	28	137	3	350861	1.269	27	8
15	1000	100	e.read	100000	3006	29	136	3	400042	1.195	27	8
15	1000	100	e.write	100000	2830	28	133	3	374311	1.081	27	8
15	1000	100	e.serial	100000	2741	27	129	3	421936	1.008	27	8
16	5000	1	e.poll	5000	35151	22561	71241	0	189432	696	26	10
16	5000	1	e.busy	5000	209	33	249	0	31555	334	26	8
16	5000	1	e.read	5000	204	35	219	0	56423	29	26	8
16	5000	1	e.write	5000	76	32	208	0	2146	284	26	8
16	5000	1	e.serial	5000	148	32	205	0	29667	23	26	8
17	10000	1	e.poll	10000	20509	125	61127	0	159521	507	61	55
17	10000	1	e.busy	10000	139	5	69	0	99038	4.968	61	55
17	10000	1	e.read	10000	89	6	68	0	80535	4.925	61	55
17	10000	1	e.write	10000	93	6	66	0	122472	4.878	61	55
17	10000	1	e.serial	10000	95	5	66	0	48407	4.795	61	55