MANUEL ALEJANDRO ROMERO MIGUEL

ANÁLISIS, DESARROLLO E INTEGRACIÓN DE SISTEMAS DE MONITORIZACIÓN DE DISPOSITIVOS EN SISTEMAS SMARTROADS

TRABAJO FIN DE CARRERA
INGENIERÍA EN INFORMÁTICA DE SISTEMAS

DIRECTOR

JORGE CASAS CAÑADA

PONENTE

IGNACIO MARTÍNEZ RUIZ

Universidad Zaragoza

MES, AÑO

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Agradecimientos

Cómo no tener un recuerdo especial para todas estas personas y personitas que te acompañan es esta aventura de la vida, y hacen que sea tan especial. Alicia, mi compañera, mi amiga, mi ilusión de la vida que con tu paciencia y esfuerzo, ha hecho posible que este sueño mío al final se haya hecho realidad " *Ali, al final lo hemos conseguido*,".

Paula, Jorge, Sara y Alicia, mis tesoros, que ilusión compartir este trabajo con vuestra presencia, algún día de parque hemos perdido, pero lo recuperaremos.

Mi padre, que bueno sería que estuviera aquí, cómo se iba a alegrar de este momento, A mi madre que siempre se ha estado para todo lo que hemos necesitado.

Mis hermanos Cristiana, Toñin, Nacho, Diego y Anaví. Gracias Cristina por tus visitas de fin de semana queme daba más tiempo para este proyecto.

A Nacho Martínez, seguramente si no nos hubiéramos visto en la cafetería no hubiera hecho nada, y sin esas palabras que me marcaron "Ahora ya no es tiempo de dudar, termínalo".

A Jorge casas, persona humana de gran calidad y técnico de alta cualificación que me ha aguantado tanto Muchas gracias

A todos gracias por existir y estar ahí, Ali, te quiero.

Cómo no tener un

TABLA DE CONTENIDO

TABLA DE ILUSTRACIONES

RESUMEN DEL PROYECTO FIN DE CARRERA

"ANÁLISIS, DESARROLLO E INTEGRACIÓN DE SISTEMAS DE MONITORIZACIÓN DE DISPOSITIVOS EN SISTEMAS SMARTROADS"

Realizado por: Manuel Alejandro Romero Miguel.
Dirigido por: Jorge Casas Cañada.
Ponente: Ignacio Martínez Ruiz.

El presente proyecto se desarrolla en el marco de la empresa Iternova, S.L., cuyo principal desarrollo se centra en un software integral de gestión de carreteras, desarrollado de forma totalmente modular: Sistema Web de Gestión de Carreteras (SmartRoads) en la empresa Iternova, S.L. Concretamente se basa en desarrollar un módulo que se integre en este software, el cual permitirá conectarse mediante API JSON/Rest de forma periódica a servicios web ofrecidos por plataformas nagios, para obtener datos de los servidores y dispositivos monitorizados por estos sistemas de monitorización. Dichos datos, una vez recibidos por el sistema (módulo de monitorización de SmartRoads), se almacenan dentro del sistema, permitiendo realizar la visualización de los mismos, obtención de informes y generación de notificaciones.

El alcance de este proyecto permite integrar dentro de la aplicación SmartRoads los datos recogidos de los servidores nagios, para poder generar gráficas e informes con dicha información. El módulo principal desarrollado (monitorización) permite configurar cuáles son los servidores (hosts) que están siendo monitorizados por diferentes sistemas nagios de los que se quiere recoger información. También permite especificar qué dispositivos y servicios se quieren monitorizar, configurando en cada uno de ellos parámetros como el servidor nagios que lo monitoriza y los servicios que se quieren monitorizar (ofreciendo una lista de posibles servicios).

Este módulo se ha desarrollado completamente ya que, una vez configurado, a través del servicio cron se ejecuta la llamada a la API JSON/Rest para recoger datos de los diferentes servidores nagios (de los hosts y servicios configurados en dichos sistemas remotos). Estos datos se almacenan en colecciones en un sistema gestor de datos basados en documentos (base de datos NoSQL), concretamente gestionados con MongoDB. En la gestión de datos se maneja una gran cantidad de información, ya que se obtienen datos mediante las API cada minuto. Esto implica una recogida de 24*60 datos cada día de cada servicio de cada host configurado.

Para realizar el proyecto se ha creado un entorno de desarrollo con todas las librerías asociadas que permite integrarlo en la plataforma completa SmartRoads garantizando que el sistema completo funcione, respetando las versiones y dependencias establecidas. A la vez, el sistema requiere una red donde alojar servidor/es nagios, diferentes host que serían los sistemas a monitorizar, el entorno de desarrollo y el gestor de base de datos (como se explica en la memoria, tras evaluar diferentes alternativas se ha decidido optar por un sistema dockerizado, estableciendo contenedores para cada dispositivos: servidores nagios, hosts a monitorizar, entorno de desarrollo).

Las tareas generales a realizar dentro del proyecto son: análisis y conocimiento de funcionamiento e instalación de sistema **SmartRoads**, puesta en marcha de sistema de desarrollo utilizando contenedores ***Docker***, desarrollo y programación de módulo **SmartRoads** de monitorización, usando tecnologías como **PHP, Javascript, colas Gearman**, bases de datos NoSQL (**MongoDB**), etc; desarrollo de **API JSON/Rest** para obtención de datos de sistemas *nagios,* desarrollo de interfaz de usuario del módulo y generación de informes usando **API Google Charts**, y generación de tests y documentación.

Como resultado, sea obtenido un módulo completo que se integra perfectamente en la plataforma SmartRoads y que permite aumentar sus funcionalidades y valor añadido.

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Introducción

La obtención de datos de dispositivos y/o servicios es una tarea muy utilizada especialmente para controlar el correcto funcionamiento de esos sistemas y servicios. Para esta tarea existe un software libre llamado **nagios** . Es este, un sistema ampliamente utilizado para monitorizar el funcionamiento de servicios y equipos previamente configurados, de manera que podemos conocer su estado. El hecho de monitorizar valores, permite verificar su estado, así como tener la capacidad de prever posibles estados no deseados como un disco lleno o una CPU que exceda su carga, por ejemplo, y poder tomar medidas correctivas antes de que el sistema se caiga.

Dentro del sistema de software, dónde se realiza el proyecto, la gestión integral de carreteras mediante el software de Iternova, es deseable poder tener de manera automatizada gráficas y notificaciones que nos dé información de los diferentes dispositivos y servicios utilizados en la gestión de infraestructuras y tener la certeza de que están funcionando correctamente o trabajan en un intervalo correcto de rendimiento. Son muchos los dispositivos que forman parte del control de gestión de la Conservación y Explotación de carreteras y que serán monitorizados por este módulo (cámaras, estaciones metereológicas, estaciones de toma de datos ETD de aforos, ...)

Utilizar **nagios** como un sistema independiente para conocer esta información, no es una solución integral, ni cómoda para cubrir este objetivo, ya que necesitamos disponer de esta información de forma integral para poder ser utilizadas en otros módulos del sistema SmartRoads. Además no seguiría la filosofía de desarrollo integral y modular que es la tónica constante en los desarrollos de Iternova y en gran medida uno de los pilares, que le permites seguir creciendo con un gran éxito en su desarrollo.

El sistema **SmartRoads** tiene muchos módulos integrados, como ya hemos comentado. Con este proyecto se va a utilizar la información que ofrece *nagios*, para monitorizar dichos valores referentes al funcionamiento de servicios y sistemas, integrándolo como un módulo más dentro de **SmartRoads**. Este proyecto permitirá especificar varios servidores nagios (1 o más) y también configurar diferentes *host* o dispositivos (1 o más) para que sean monitorizados por los servidores *nagios*. Cuando configuremos qué *host* queremos monitorizar, detallaremos también los servicios concretos de ese *host* de los que cuales deseamos obtener valores para posteriormente almacenarlos en un sistema de bases de datos y poderlos utilizar en el sistema **SmartRoads**.

Para poder obtener datos de los servidores *nagios* para luego almacenarlos en el módulo de monitorización de forma periódica, lo haré mediante el demonio ***cron***. El sistema **SmartRoads**, integra la posibilidad de llamar a este demonio y especificar el método o métodos que queremos que se ejecuten. Esta ejecución se realizará en segundo plano usando un sistemas de colas llamado ***gearman***. De esta manera no se bloquea el sistema al obtener los datos, ya que *gearman* se ejecuta en función de los recursos disponibles en el sistema (la carga de CPU, memoria, etcétera) y en un segundo plano, de forma transparente al usuario.

Lo que se implementa en este método, que se ejecutará de forma periódica, será la llamada mediante un servicio JSON/Rest al los servidores *nagios*, de los cuales recogeremos datos en formato JSON con la información que nos dé de los servicios y *host* que monitoricen.

De la información que nos devuelvan todos los servidores *nagios* configurados en nuestro sistema (una API JSON/Rest por servidor *nagios*) , cogeremos solo aquella de los *host* que estén especificados previamente en nuestro módulo.

1.2 Antecedentes

El PFC se ha realizado íntegramente en la empresa **ITERNOVA S.L**., cuyo negocio se basa en proveer soluciones tecnológicas innovadoras para empresas, ciudadanos y administradores, con sedes en Zaragoza, Teruel y México DF. Actualmente su principal campo de trabajo es el desarrollo del Sistema Web de Gestión de Carreteras (*SmartRoads*), el cuál se trata del primer sistema web SCADA (*Supervisory Control And Data Acquisition*) desarrollado para la gestión de la Conservación y Explotación de Carreteras y otras infraestructuras. En este campo se enmarca el presente trabajo fin de grado.

ITERNOVA S.L., es una empresa con una continua apuesta por la innovación, que les ha permitido desarrollar productos de gran éxito e implantación a nivel nacional, como el sistema **SmartRoads**, por el cual la empresa recibió el VIII Premio nacional ACEX 2012 a la Seguridad en Conservación de carreteras, además de otros galardones como el Premio Sociedad de la Información Aragón 2012 Empresa Junior del año y Premio Empresa Teruel Innovación 2012.

El sistema SmartRoads desarrollado por **Iternova** plantea un importante salto cualitativo en la gestión de la explotación de carreteras, permitiendo acceder a toda la información de las mismas de una forma integrada.

Se trata de un Sistema Web multidispositivo en código libre (no requiere de licencias ni mantenimientos), modular (se adapta a las necesidades de cualquier demarcación) y multiusuario (con diferentes roles de acceso a la información)

El sistema está basado en el concepto de SaaS (*Software as a Service*), pudiendo ser adaptado a las necesidades de cada cliente.

El innovador sistema desarrollado, integra de una forma optimizada la gestión de todos los elementos de interés de las carreteras y está formado por los siguientes módulos:

1. **Módulo de información** y gestión general de carreteras, con sus elementos más significativos inventariados y geolocalizados.
2. **Módulo de la vialidad en tiempo real ITS** (*Intelligent Transportation Systems*): sistema de gestión de flotas (GPS), de cámaras de explotación, aforos y ETD, paneles de **Módulo de gestión** información PMV, estaciones meteorológicas, etcétera.
3. **Módulos de comunicación**, módulos de documentación (gestor de expedientes), y funcionalidades avanzadas de gestión Web.
4. **Módulo de agenda avanzada** (para la gestión de las actuaciones programables y no programables de vialidad), y sistema de control y seguimiento de trabajos.
5. **Módulo de gestión económica**: seguimiento a programas de trabajo, gestión financiera y gestión física.
6. **Módulo para gestionar las actividades de seguridad vial**: gestión de accidentes para poder generar informes de accidentalidad de los distintos tramos de carretera para poder tomar las medidas oportunas, que consigan reducirlos.
7. **Aplicaciones móviles**: permite la toma de datos por parte de los operarios de campo, en tiempo real, para los diferentes módulos.

Cualquier usuario, haciendo uso de un dispositivo con conexión a la red podrá consultar toda la información de la carretera accediendo con su nombre de usuario y contraseña o con su certificado digital.

El sistema de gestión es modular en el que las diferentes secciones (videocámaras, agencia de vialidad, gestión de flotas GPS, estaciones meteorológicas, paneles informativos, fotografías aéreas,……) se integran en función de las necesidades concretas del cliente. Se puede comenzar con unas herramientas básicas e ir ampliando según aumenten las necesidades. Es una plataforma que se adapta y personaliza tanto a pequeños sectores como a grandes demarcaciones de carreteras.

La información de la carretera está disponible en todo momento lo que nos ayuda a tomar decisiones de forma rápida y eficaz ante cualquier incidente (consultar las instalaciones y elementos de la carretera, disponer de fotografías aéreas en alta resolución y cartografía de Google Maps, identificar todos los puntos con restricciones de galibo, anchura, estructuras ; listar puntos de interés para gestionar la vialidad en caso de cortes u otros problemas en la carretera (cruces de mediana, intersecciones, desvíos…).

Incorpora una biblioteca técnica con los protocolos de actuación, normativas internas, gestión de expedientes o cualquier información que se precise. Se puede consultar la información en tiempo real por los dispositivos instalados en la carretera e información histórica para muy Control de flotas mediante GPS en tiempo real, consulta de recorridos, Lo numerosos informes permiten obtener datos importantes de los recorridos realizados, localizar operativos cercanos a un punto crítico, etcétera.

Los informes de aforos de las carreteras y las básculas permiten saber qué tipo de tráfico soporta la carretera en cada momento y obtener información estadística (vehículos con sobrepeso, con exceso de velocidad, tipologías de vehículos, etcétera.).

Se dispone al instante de imágenes tomadas por las cámaras de explotación, tanto las fijas instaladas en la carretera como las móviles que llevan los vehículos de vigilancia o de viabilidad invernal.

Se puede consultar los mensajes que muestran los diferentes videopaneles.

La agenda, permite registrar todas las tareas de vialidad que se llevan a cabo en la carretera y las muestra en función de prioridades establecidas por la carta de registros al ciudadano. Cada registro es acompañado de sus protocolos de actuación.

Se dispone de informes de frecuencias de incidencias de cada tipo. Se obtienen datos sobre los tiempos de respuesta para cada tipo de incidente. También, dispone de un sistema de avisos de registros pendiente y de las zonas más afectadas por cada incidente.

El sistema **SmartRoads** se ha convertido en el sistema más avanzado de gestión de la Explotación y Conservación de carreteras, cuyo éxito está siendo avalado por su implantación paulatina en las principales provincias y comunidades autónomas españolas, tales como Aragón, Murcia, Soria, Valencia, Tarragona y Castilla La Mancha entre otras, así como en las principales carreteras y autopistas de México y en empresas concesionarias dedicadas a la conservación y la explotación de carreteras.

Ilustración 1 Módulos dle SmartRoads

Con la terminación de este proyecto, el sistema contará con un módulo nuevo llamado **Módulo de monitorización** que permitirá, por tanto, gestionar el estado de los diferentes dispositivos tecnológicos utilizados en el sistema (desde cámaras a GPS, estaciones meteorológicas, estaciones de aforo ETD y básculas, otros sistemas, etcétera)

1.3 Motivación

El análisis, desarrollo e implementación del módulo de monitorización se enmarca dentro de los proyectos pendientes de mejora de **ITERNOVA S.L.**

Mediante este desarrollo se pretende tener control del correcto funcionamiento de los dispositivos y poder analizar valores que nos permitan anticiparnos a posibles caídas de sistemas y/o servicios.

En el sistema de gestión de carreteras existe un gran número de dispositivos. La posibilidad de tener monitorizados estos dispositivos dentro de un sistema de gestión integral donde se puedan obtener gráficas e informes, permitirá optimizar el funcionamiento general

La idea de realizar este proyecto, vino de un encuentro causal, con el profesor Ignacio Martínez. Me habló de esta empresa para realizar el proyecto.

Visité la empresa y desde el principio la buena comunicación, el control técnico y la gran disposición, me cautivó.

La idea del proyecto, que en un principio me despistó un poco, creó una inquietud de introducirme en el desarrollo donde se realizará una mejora de un producto ya existente. A la vez, me permitiría profundizar en los contenidos que el proyecto exigía como la comunicación entre diferentes servicios y procesos dentro del desarrollo, tecnologías relativamente nuevas como MongoDB (gestor de bases de datos de tipo no relacional, en concreto documental), gestión de colas con *gearman*, fueron el motivo por el que no dude en aceptar este proyecto. Un tema que también me llamó mucho la atención fue que este proyecto implicaba combinar diferentes elementos software dentro del sistema (sistemas remotos nagios, cron y la gestión de todo ello dentro de la aplicación).

La voluntad de realizar el PFC dentro del marco de la empresa **ITERNOVA S.L** me permitía conocer de primera mano el mundo de la empresa, y poder usar esta experiencia en mi trabajo laboral, como profesor de secundaria en ciclos formativos.

Conocer el funcionamiento de las APIs (JSON/Rest) es interesante porque en la actualidad casi todos los sistemas funcionan mediante APIs. De esta manera se produce una descentralización de los servicios pudiendo incluir servicios en cualquier servidor y directamente conectándonos a él, se pueden realizar funcionalidades de todo tipo y casuística.

1.4 Objetivos (general y específicos)

Se establecen los siguientes objetivos:

El **objetivo principal** de este PFC consiste en **el análisis, desarrollo e integración de un módulo nuevo** dentro de **SmartRoads** llamado modulo de **monitorización** para conocer el estado de dispositivos remotos utilizados en dicho sistema en tiempo real".

Como punto final se realizará la integración y puesta en producción dentro del sistema de gestión de la Demarcación de Carreteras del Estado en Aragón y también en las principales carreteras y autopistas de México. Para ello se han definido una serie de objetivos específicos a alcanzar a lo largo del desarrollo del PFC:

- Conocer el funcionamiento del mundo de la empresa desde una perspectiva diferente a la que se están realizando los estudios universitarios.
- Aprender la configuración de entornos de desarrollo como son PHP, MySQL, MongoDB, JavaScript, AJAX, Google Maps, colas de procesado Gearman, etcétera
- Establecer un sistema de desarrollo que me permita movilidad mediante herramientas de virtualización y control de versiones *git*
- Aprender el funcionamiento y características de las API JSON/Rest (explicadas brevemente en el anexo).
- Crear un nuevo módulo dentro del SmartRoads, desarrollándolo con la filosofía MVC de los desarrollos de Iternova
- Configurar servidores *nagios* y *host* dentro de dicho módulo
- Encapsular los diferentes procesados de datos obtenidos de las APIs en sus clases correspondientes para facilitar el mantenimiento de éstas puesto que los datos devueltos no tienen por qué coincidir al 100% unos con otros (por ejemplo, CPU, Memoria, Espacio en Disco, PING, etcétera) y por lo tanto el tratamiento de la información devuelta por cada tipo de servicio puede variar ligeramente, permitiendo la escalabilidad del sistema mediante la inclusión de nuevos métodos de procesado en caso de ser necesarios.
- Establecer las conexiones con los servidores *nagios* mediante llamadas a las API cada minuto, siendo ejecutado en segundo plano utilizando un sistema de colas *gearman*. Esto implica recepción de datos de cada servicio de cada *host*, procesado de los atributos obtenidos de los servidores *nagios* (*parseo y mapeo)* y almacenamiento de los datos en MongoDB.
- En el almacenamiento de datos en la base de datos no relacional, se creará un documento por día y dispositivo o *host* a monitorizar, actualizándose cada uno de estos documentos cada minuto con los datos recibidos de los servidores *nagios*. Se utilizarán técnicas de *time-series*, siguiendo buenas prácticas indicadas en webinarios y documentación de MongoDB.
- Realizar informes en forma de gráficas y tablas utilizando la API de Google para ese cometido. Estableceremos diferentes intervalos para dichas gráficas (gráficas detalladas de últimas 3 horas y gráficas a medio y largo plazo seleccionado el rango de fechas)
- Realización de pruebas de test automatizados de las distintas mejoras realizadas para que se garantice el correcto funcionamiento.
- Como producto final desarrollado se integrará el módulo de monitorización en la plataforma SmartRoads y posteriormente en sistemas en producción como puede ser el de la Demarcación de Carreteras del Estado en Aragón (carreterasaragon.com), así como en proyectos que actualmente tiene Iternova en ejecución en autopistas de México.

1.5 Estructura de la memoria

La memoria de este PFC está dividida en dos partes claramente diferenciadas. En la primera parte se desarrolla el grueso del trabajo, donde se explica el análisis y el desarrollo de los objetivos descritos anteriormente. La segunda parte contiene los anexos necesarios para detallar y ampliar los conceptos técnicos utilizados en la primera parte de la memoria. En los anexos no se comentan detalles fundamentales para el seguimiento del desarrollo pero sirven para tener una visión más completa del PFC en sus aspectos más técnicos.

La primera parte de la memoria está dividida en los siguientes capítulos:

- **Introducción y objetivos:**
  En esta primera parte se explica la problemática, el entorno en el que se ha desarrollado el PFC, los antecedentes de algunas tecnologías utilizadas, la motivación que ha llevado a la elaboración de este PFC y los objetivos que pretende abordar**.**
- **Estado del arte**
  Breve descripción del punto de partida del proyecto que consiste en desarrollar un módulo nuevo para la mejora del software; Visión general de la problemática que surge en la actualidad y por qué realizar esta mejora.
- **Análisis y diseño**
  En este apartado se realiza un análisis de los requisitos a cumplir y que posteriormente se irán desarrollando. Es una parte especial, donde usando diagramas explicativos, se van a ir analizando y diseñando cada uno de los requisitos de forma secuencial.
- **Desarrollo e implementación**
  En este apartado agruparé los requisitos con lógica funcinal y a partir de su análisis expondré su desarrollo, en varios casos mostraremos código que resulte interesante para ver la manera en la que se ha implementado el proyecto
- **Resultados**
  Exposición de los resultados obtenidos mediante la realización de este PFC. Mostraremos patallas y gráficas que visualicen el resultado de los diferentes requisitos implementados en el proyecto..
- **Conclusiones y líneas futuras**
  Se reflexiona sobre las ideas aprendidas durante la elaboración de este PFC. También se enumeran una serie de líneas futuras que mejorarían la funcionalidad de este sistema.

La segunda parte de la memoria comprende los anexos para facilitar la comprensión de la memoria. Durante el aprendizaje del leguaje utilizado, he escrito una wiki () donde se muestran las características del lenguaje que posteriormente se ha utilizado en el trabajo personal del autor. Directamente del wiki se han generado los correspondientes PDF para un incluir como anexos al presente proyecto. Los documentos anexados se han clasificado en dos grupos:

- **Relacionados con php**
  - Php: conceptos general
  - Php: formularios
  - Orientación a objetos con php
  - Autentificación desde php
  - Sesiones
  - Cookies
  - Bases de datos con php
- **Utilidades usadas**
  - Instalación de cron
  - Instalación de phpunit
  - Instalación y duso de nagos
  - Uso de git
  - Librerías a instalar en el sistema
  - Uso de vagrant y puppet en la primera fase del proyecto
  - Uso de docker en el proyecto, comando utilizados

2 ANÁLISIS Y DISEÑO

Para cubrir los objetivos marcados establecemos una serie de requisitos concretos, los cuales se especificaran con diagramas de caso de uso y de clases según correspondan, con el objetivo de dejar bien fijados los cimientos del desarrollo y con ello seguir creciendo en la implementación de este proyecto.

En primer lugar, se lista a continuación la **lista de requisitos** (funcionales y no funcionales):

Requisito	**Descripción**
RNF1	Establecer el entorno de desarrollo, un ecosistema portable
RF2	Crear un nuevo módulo **monitorización** integrado en SmartRoads
RF3	Crear plantillas para insertar servidores *nagios* en el módulo
RF4	Crear plantillas para insertar *host* a monitorizar
RF5	Guardar en *MongoDB* los servidores *nagios*
RF6	Guardar en *MongoDB* los *host* a monitorizar
RF7	Poder buscar, ver, modificar y borrar servidores *nagios*
RF7	Poder buscar, ver, modificar y borrar *host* a monitorizar
RF8	Establecer control para las acciones anteriores (zona administración o zona pública)
RF9	Crear comunicación desde la aplicación mediante un servicio JSON/Rest a los servidores *nagios*
RF10	Recoger la información de los servidores
RF11	Establecer un sistema *cron* para que el método de comunicación con los servidores *nagios* se ejecute cada minuto
RF12	Parsear la información obtenida de los servidores (un documento JSON) y recoger sólo la de los *host* configurados
RF13	Crear o modificar documentos en *MongoDB*. Tendremos un documento por día y por *host*, en el cual recogeremos información de cada servicio (*time-series* en *MongoDB*)
RF14	Establecer qué información queremos almacenar. Tanto los servicios , como los índices de los mismos, ya que *nagios* devuelve unos 60 índices de información por servicio.
RF15	Configurar la plantilla para establecer criterios a la hora de obtener informes de cada *host*
RF16	Obtener informes (gráfica y tabla con datos) entre dos fechas obteniendo valores promedio cada 30 minutos.
RF17	Obtener informes (gráfica y tabla con datos) de las tres últimas horas con resolución de 1 minuto.
RF18	Obtener informes (gráfica y tabla con datos) entre dos fechas obteniendo la media de cada aritmética de cada día.
RF19	Realizar pruebas de testeo para la verificación del correcto funcionamiento de los métodos creados.
RF20	Generar documentación del código desarrollado y manuales de usuario

La siguiente parte de esta sección procederá a analizar cada uno de estos requisitos, así como establecer su análisis y diseño.

2.1 Establecer entorno de desarrollo (RNF 1)

Este requisito parece más una decisión de diseño que de análisis, pero en mi caso la quiero tratar y dejar establecida antes de empezar a realizar ninguna tarea.

Este proyecto utiliza las herramientas de desarrollo que la empresa usa para desarrollar su software integral, pero aquí lo que se pretende es determinar qué entorno/equipo voy a utilizar para el desarrollo, por lo que se trataría de un *requisito no funcional.*

Dada la necesidad de movilidad que tengo necesito un entorno que pueda desplegar en diferentes sitios. El código fuente está solucionado usando el gestor de versiones con **git.**

Mi desarrollo va a necesitar un sistema de red (diversos equipos *nagios*, diversos *hosts* a monitorizar y un servidor web con los servicios necesarios para correr la aplicación). Por ello voy a usar virtualización, ya que así puedo tener todo el conjunto en un solo equipo.

Tras una serie de consultas e investigar se presentan las siguientes opciones a considerar:

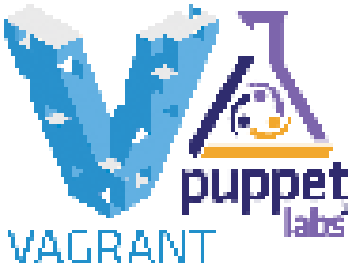
1. Máquinas virtuales Virtual Box
2. Usar Vagrant configurado mediante Puppet bajo Virtual Box
3. Usar un sistema dockerizado basado en contenedores.

Opción 1: Virtual Box[1]

Se plantea esta opción por ser VirtualBox un software gratuito frente a otros como VMware que no lo son. Otros tipo quemu, también gratuitos, no tienen o están mas limitados para instalar la parte visual, o xwindow que sí quiero tener disponible en mi entorno de desarrollo. Esta opción la descarto por el tiempo que tarda en arrancar la máquina virtual (al menos voy a necesitar 3 máquinas virtuales). Por otro lado si quiero cambiar de equipo de desarrollo (dadas mis circunstancias personales esto es muy previsible), necesitaría exportar la máquina e importarla, usando por ejemplo una memoria USB. Esto no es viable, por el tiempo que implica. Exportarla es relativamente rápido (de 1 a 3 minutos). Copiar a USB ya puede tardar más, en total de 6 a 9 minutos. Volver a copiarla del USB a un equipo y luego importarla puede rondar de nuevo los 10 minutos; Existen otras opciones como copiar ficheros de configuración que limitarían este tiempo, pero en cualquier caso es un tiempo considerable. Esto claramente me impediría poder trabajar con flexibilidad, por lo que esta opción la descarté tras valorarla.

1 Web de referencia **https://www.virtualbox.org/**

Opción 2: Vagrant[2] y puppet[3]



Esta opción se plantea y empieza a utilizar. Ambas herramientas que se complementan son de código abierto, y permiten levantar máquinas virtuales (Viritual box) de manera rápida, así como poder replicar la misma máquina con un ficher de configuración.

Para configurar la máquina virtual a nivel de software usaremos un fichero de configuración: **Vagrantfile.** Este fichero de configuración que podemos crear a mano siguiendo unas reglas, pero también se puede crear de forma gráfica usando la herramienta **puppet**.

Con ella tengo dos ventajas. Por un lado tengo un fichero llamado ***vagrantfile*** que me permite especificar a nivel de software las herramientas que tengo instaladas. Esto me permite tener de una forma rápida y segura los sistemas o distintos equipos donde yo desarrolle sincronizados. Es una forma útil para tener a varios desarrolladores en diferentes máquinas y todos sincronizados y actualizados a nivel de software (servicios, librerías, versiones, ...). Esto en el desarrollo, por mis circunstancias personales (laborales y familiares), va a ser muy frecuente poder cambiar de sitio para desarrollar; también de forma muy habitual hay que instalar nuevas librerías o aplicaciones durante el desarrollo. Así, de forma rápida, puedo garantizar la misma instalación de sistemas en diferentes equipos en poco tiempo.

Otra gran ventaja que tengo es poder tener un volumen creado o mapeo del directorio de la máquina anfitriona donde esté la máquina virtual con el contenido del directorio *home* de la máquina virtual, lo que facilita poder pasar o sincronizar ficheros de una máquina a otra, aunque habitualmente se haga con comando ***scp*** usando ***ssh*** .

Ver anexo 1 donde se muestra los comandos utilizados y el fichero puppet creado para levantar los sistemas y mantenerlos actualizado de forma más automatizada. En el anexo 1 también se muestra la forma en la que usé *vagrant* sobre *VirtualBox.*

Opción 3: Docker

Cuando empiezo a necesitar *nagios*, me presentan *docker*. Realizo un pequeño estudio para ver cómo se adaptaría a mis necesidades y me quedo maravillado de sus prestaciones. Cambio todo a este sistema de virtualización basado en contenedores. Sus ventajas son muchas. Rápido de arrancar, rápido de poder crearme una imagen a partir de un contenedor y desplegar el contenedor de dicha imagen en otro equipo en cuestión de segundos. Ligero, flexible y portable. Ideal para mi desarrollo. De hecho durante el desarrollo tuve que cambiar el disco duro y me costó muy poco realizar el nuevo depliegue.

---

2 Web de referencia https://www.vagrantup.com
3 Web de referencia https://puphpet.com

En la instalación inicial creé un *dockerfile* para crear una imagen con los servicios necesarios a partir de una imagen limpia del sistema operativo *Ubuntu*

Para el desarrollo se establece la siguiente esquema de red :

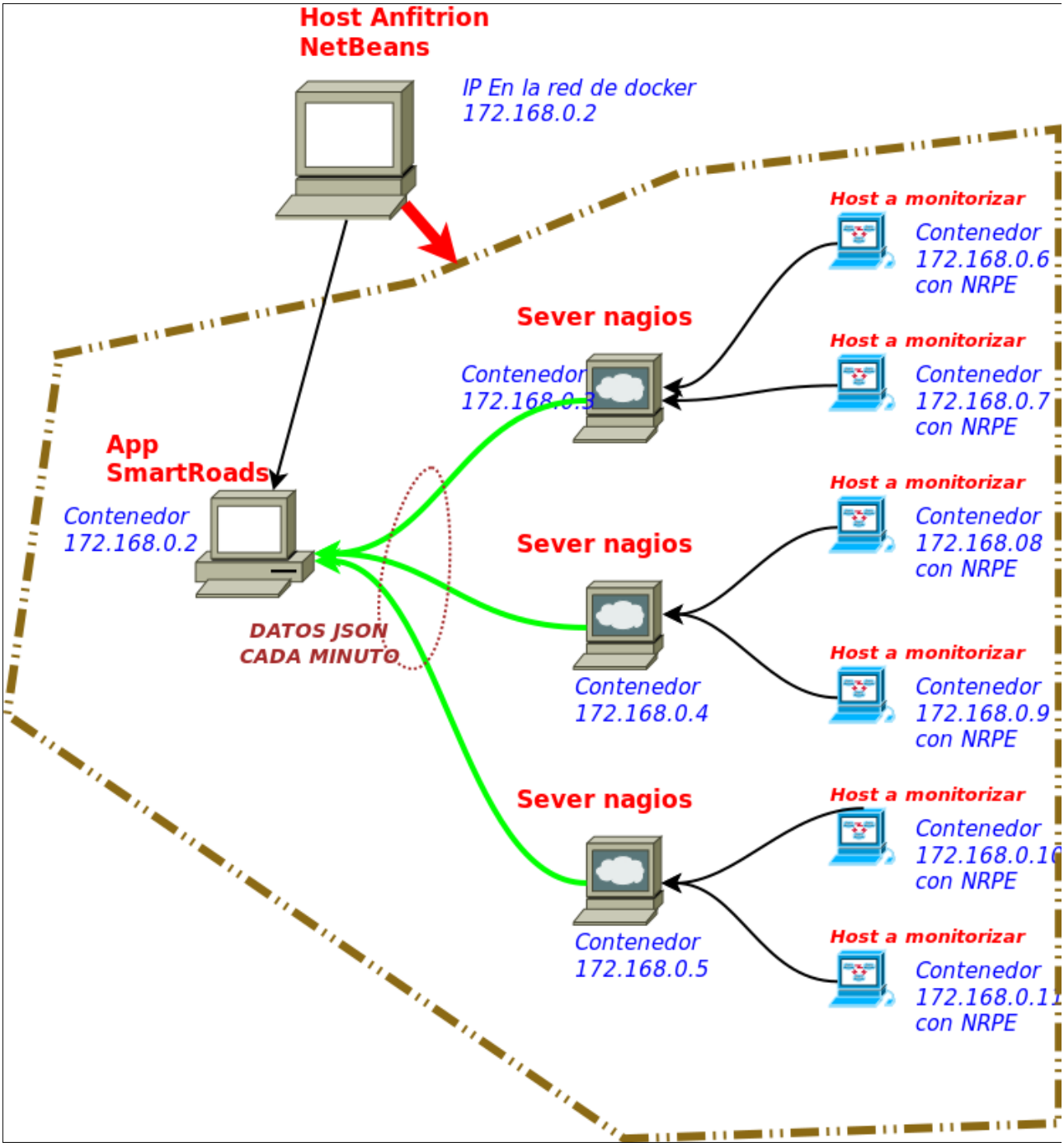


Ilustración 2 Esquema de red del entorno de desarrollo

En la imagen vemos el esquema de red con los diferentes contenedores y su interrelación. Los siguientes párrafos explican su cometido

El *host* anfitrión

Es el equipo que tiene el despliegue de los contenedores. El anfitrión tiene el IDE **Netbeans** y un volumen creado en en */var/www/iternova* de este *host,* sincronizado con el directorio */var/www* del contendedor 1 donde está el código a ejecutarse en el *virtualhost* por defecto del servidor web *Apache.* En dicho directorio se encuentra el código de la aplicación *SmartRoads*, gestionando las diferentes versiones que vaya generando mediante *git*, lo cual me permitirá crear ramas de desarrollo (*branches*) y disponer de copias de seguridad remotas. Ejecutaré también aquí otras herramientas como *MongoChef* como gestor de bases de datos de *MongoDB* para poder manipular y consultar las colecciones y documentos. Es la IP **172.168.0.1** dentro de la red de *docker*

App Smartroads

En este contenedor tendremos instaladas el servidor web *Apache*, el servidor bases de datos relacional *MySQL* y la base de datos no relacional *MongoDB*, y todas las librerías necesarias para el correcto funcionamiento del desarrollo web (Ver anexo 3 proceso de instalación). Como futuras mejoras se tiene planteado crear diferentes contenedores para cada base de datos, siguiendo la filosofía de servicios distribuidos que permite *docker.* En este contenedor configuraremos **cron** para que cada minuto ejecute un método de nuestra aplicación que mediante un servicio JSON/REST solicitará datos a las API de los servidores *nagios.* La IP 172.168.0.2. Este contenedor se llamará **iternova**

Server *Nagios*

Estos contenedores contendrán instalado servidores *nagios* y los servicios necesarios para que funcionen correctamente, como *Apache* y *MySQL*. También instalaremos el plugin de *nagios* que nos permitirá crear el servicio web JSON/Rest al que se conectará nuestra aplicación para obtener datos.En el dichero de configuración correspondiente configuraremos los dispositivos remotos y *hosts* a ser monitorizados. Las IP de estos contenedores son las que se especifican en el diagrama pudiendo crear tantos servicios *nagios* como sean necesarios.

Hosts a monitorizar

Estos contenedores simulan los *host* a monitorizar. Se corresponderían con dispositivos y equipos remotos que deben ser monitorizados. Monitorizaremos servicios de dichos equipos (como estado de servidor web, SSH, MySQL…) o directamente estado del hardware como estado de discos duros, memoria disponible o carga de CPU. En ellos debemos instalar *NRPE,* que es un demonio a ejecutar en los *host* que *nagios* quiere monitorizar para procesar las peticiones de ejecución de comandos y ofrecerle al sistema *nagios* datos de forma sencilla. Recibe la petición del equipo autorizado configurado (servidor con sistema *nagios*), devolviéndole información de estado de un servicio o hardware de dicho sistema remoto. Este escenario quedaría representado por la comunicación host a monitorizar – server nagios

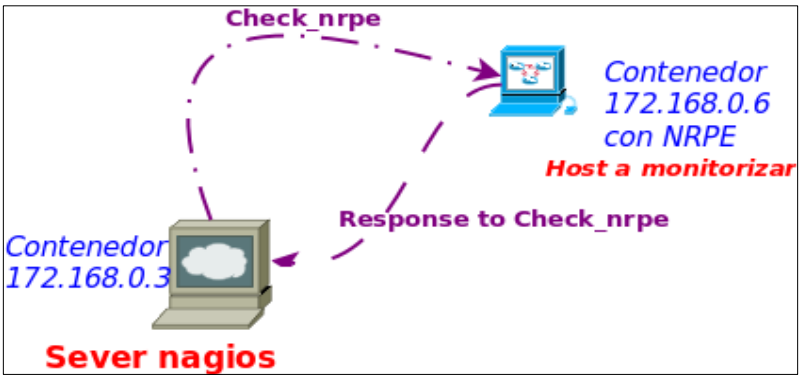


Ilustración 4 cominicación nagios - host

2.2 Crear un nuevo módulo de monitorización integrado en SmartRoads

Para crear un nuevo módulo nos basamos en módulos ya establecidos previamente en la aplicación, pudiendo replicar su estructura y analizando código ya creado.

Realizamos una observación en la estructura de carpetas de la aplicación *SmartRoads* que podemos ver en la imagen siguiente:

Realzamos una observación en la estructura de carpetas de la aplicación SmartRoads que

podemos ver en la imagen siguientes

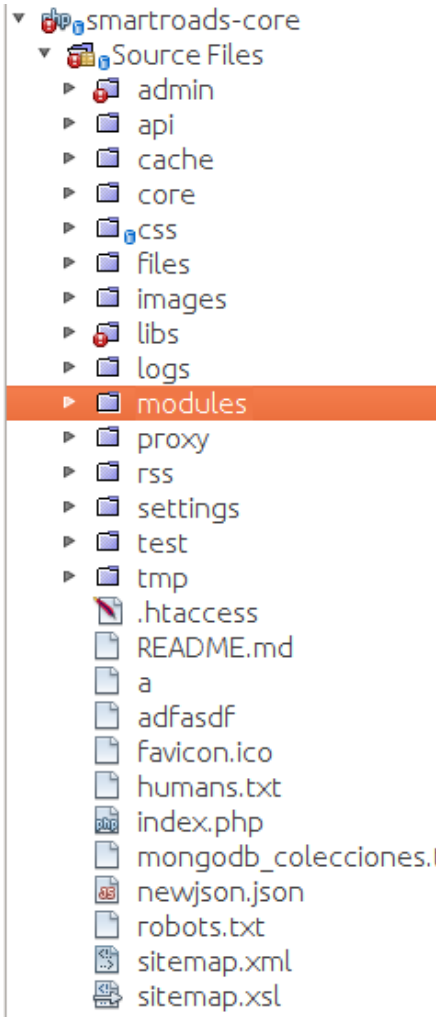


Ilustración 5 estructura de carpetas de

En ella vamos a la carpeta **modules** y analizamos su estructura. El desarrollo está basado en el modelo vista controlador **MVC** que comentaremos en la parte de implementación, que básicamente consiste en separar los métodos de la aplicación en tres niveles o carpetas

Controlador: Este método recibe la solcitud *HTTP* con todos los parámetros y valores que lleve asociado (variables *POST, GET*, …. y valores de sesión y *cookies*). Estos métodos ejecutan la lógica de negocio. Realizan peticiones al modelo lógico de datos de la base de datos y entregan a la vista valores para que incorpore en el código HTML que se visualizará en el navegador web o clientes.

Modelo: Clases con métodos que van a interactuar con la base de datos y se encargarán de recoger datos, almacenarlos y realizar las operaciones típicas *CRUD* con el gestor de bases de datos. Estos métodos interactuarán con los métodos del controlador cuando les solicite información

Vista: Clases y métodos que crearán la vista o el producto final para entregar al agente que solicitó una URI. Se comunicará con el controlador para recibir los datos a incorporar en la página.

La siguiente imagen muestra gráficamente el proceso.

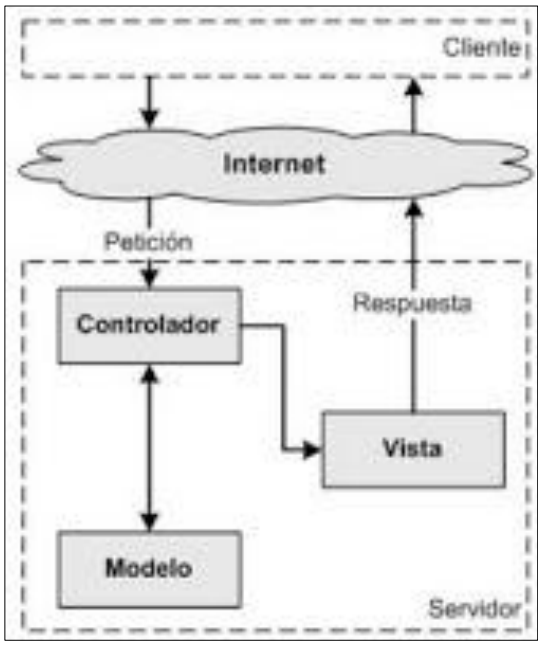


Ilustración 7 modelo vista controlador

Analizamos los módulos y creamos el módulo de monitorización con la misma estructura de directorios y ficheros, los cuales en la parte de implementación completaré y explicaré.

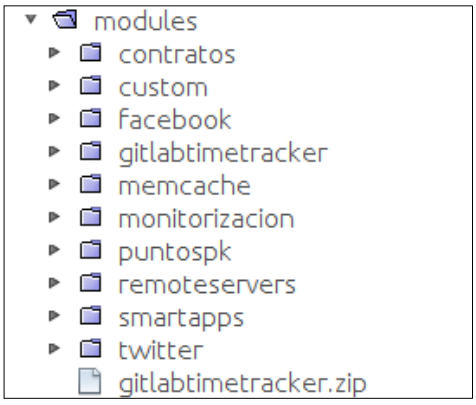


Ilustración 9moduls de SmartRoads

Primero una imagen con todos los módulos, incluido ya el de monitorización

A continuación la estructura del módulo de monitorización

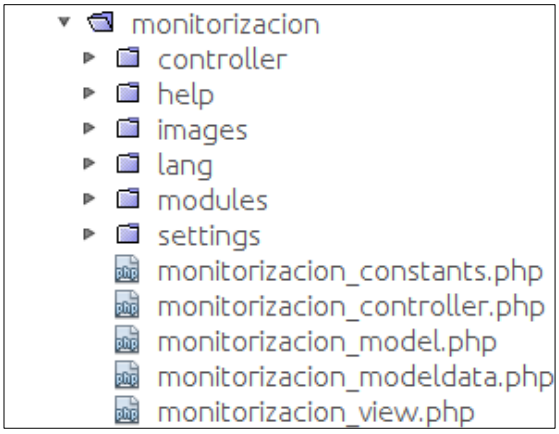


Ilustración 11 módulo de monitorización

2.3 Crear plantillas para insertar diferentes datos

Estaríamos hablando aquí de los requisitos de la parte de interfaz gráficamente, para los cuales se requiere disponer de parte controladora, modelos lógicos de datos y la generación de dichos interfaces de usuario. Primero vemos las diferentes opciones que serían 4, como podemos ver en la imagen siguiente

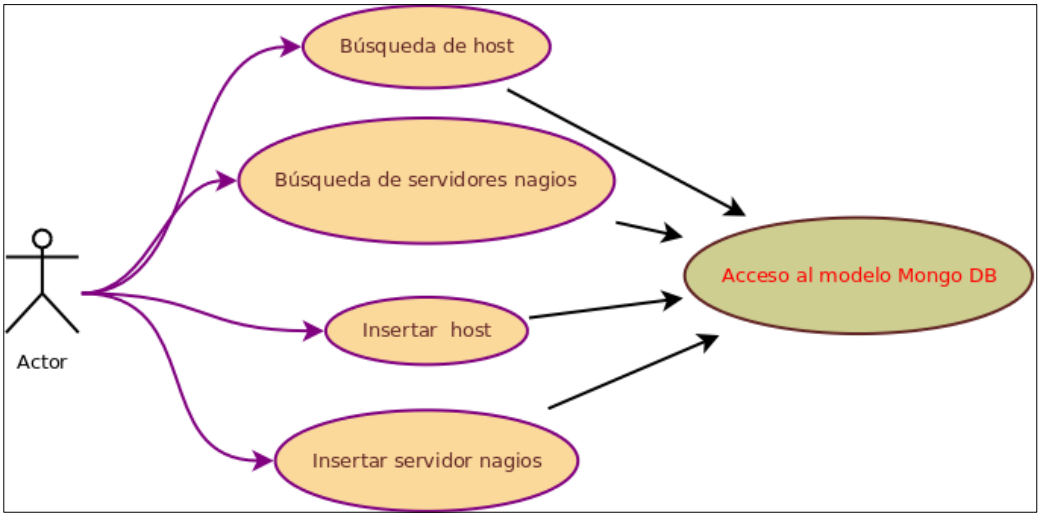


Ilustración 12: caso de uso pantallas monitorización

Esto implica que necesitamos tres botones en la pantalla de la aplicación para cada una de las acciones.

2.4 Diseño de pantallas

Las siguientes imágenes muestran los *mockups* de las pantallas que posteriormente implementaremos con los módulos de vista del desarrollo

Ilustración 14: mockup pantalla principal monitorización

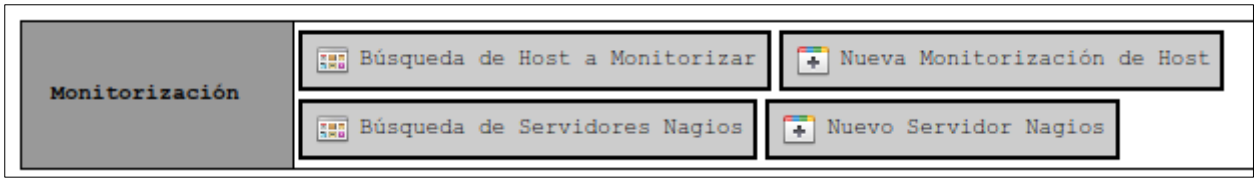


Ilustración 13: pantalla principal monitorizacion adpatada a pantalla

El sistema ha de ser *responsive* y adaptable a dispositivos móviles, por lo que al reducir el tamaño de la pantalla o en una prueba del navegador se debe de adaptar de forma automática la aplicación:

A la hora de insertar datos primero debemos de decidir qué datos vamos a necesitar tanto de los *host* como de los *nagios*. Las clase de **modelo lógico para cada *Host***

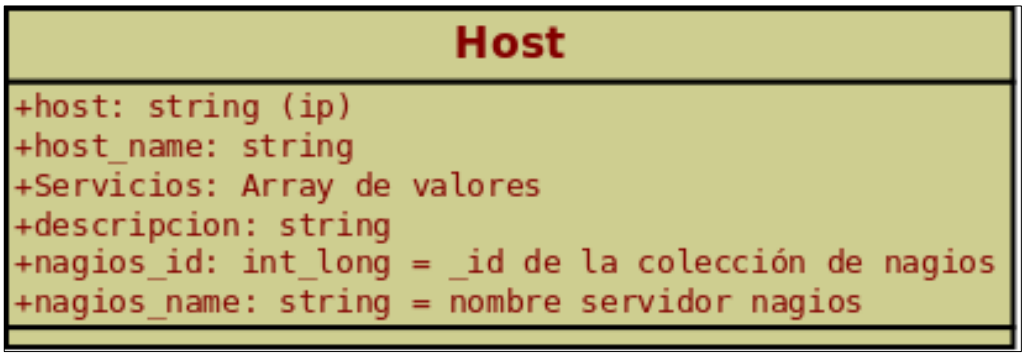


Ilustración 15: clase host

Mockup de pantalla de insertar un nuevo host:

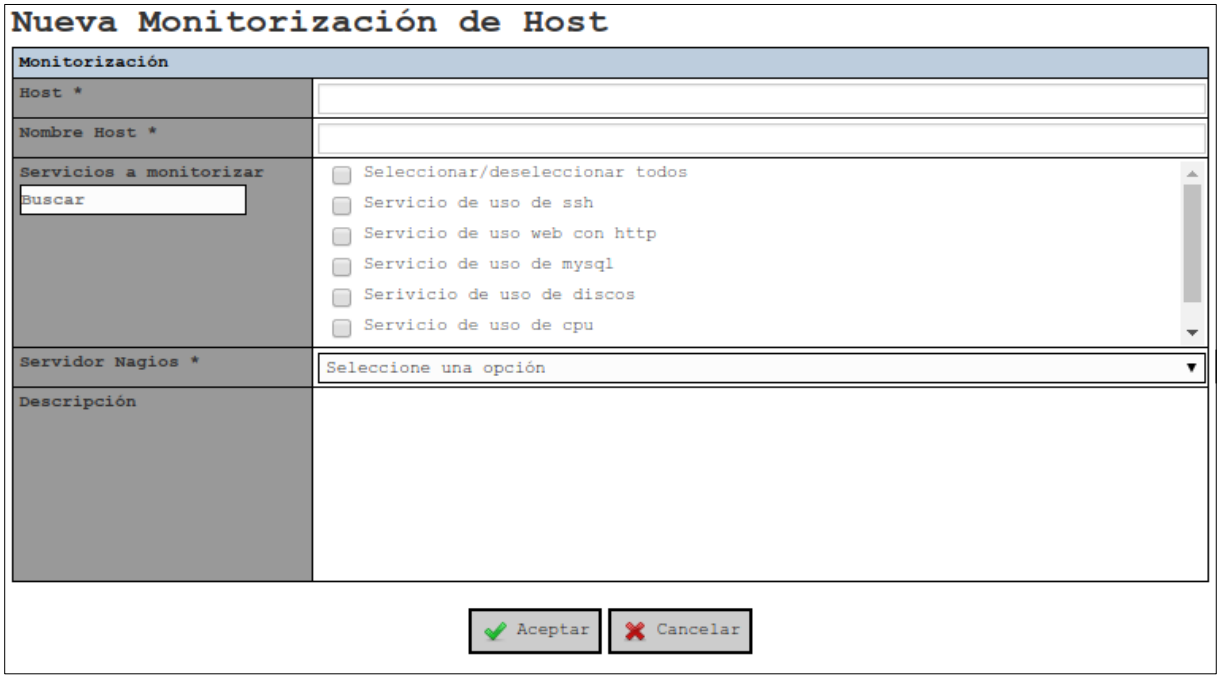


Ilustración 16: mockup pantalla nuevo host

La clase del modelo lógico para gestionar los servidores *nagios*:

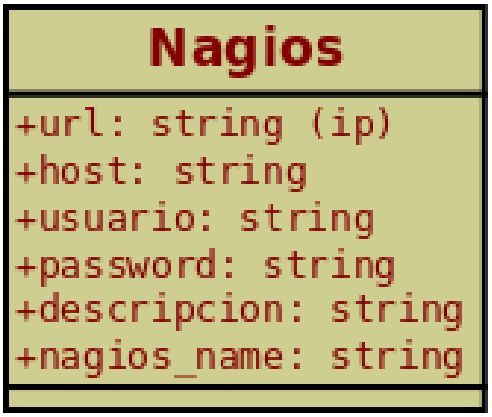


Ilustración 17: clase servidor nagios

El *mockup* de pantalla de insertar un nuevo servidor *nagios:*

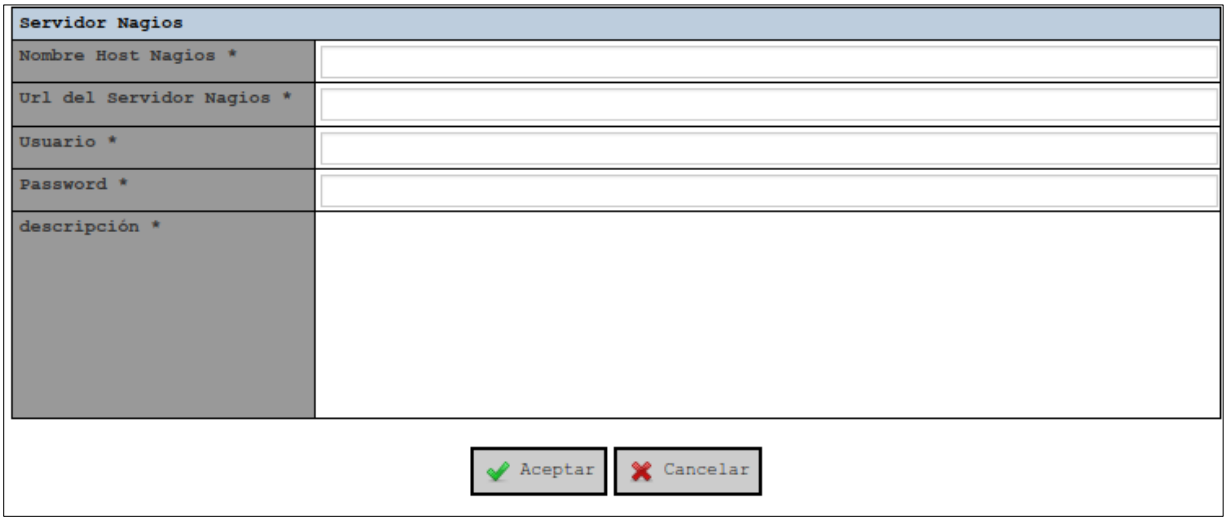


Ilustración 18: mockup pantalla nuevo servidor nagios

Para realizar búsquedas de *hosts* y *servicios* mostraremos un menú con la opción de filtrar resultados por *host* y servicios, mientras que si no se rellena el campo buscaremos todos los elementos

Ilustración 19: mockup búsqueda host

Mockup de buscador de servidores *nagios*:

Ilustración 20: mockup búsqueda servidores nagios

Para ello realizamos las acciones definidas en el siguiente diagrama de casos de uso. Expongo a continuación el proceso para almacenar datos de cada *host*, ya que el proceso de almacenar cada servidor *nagios* será el mismo proceso pero cambiando el modelo lógico que lo gestiona.

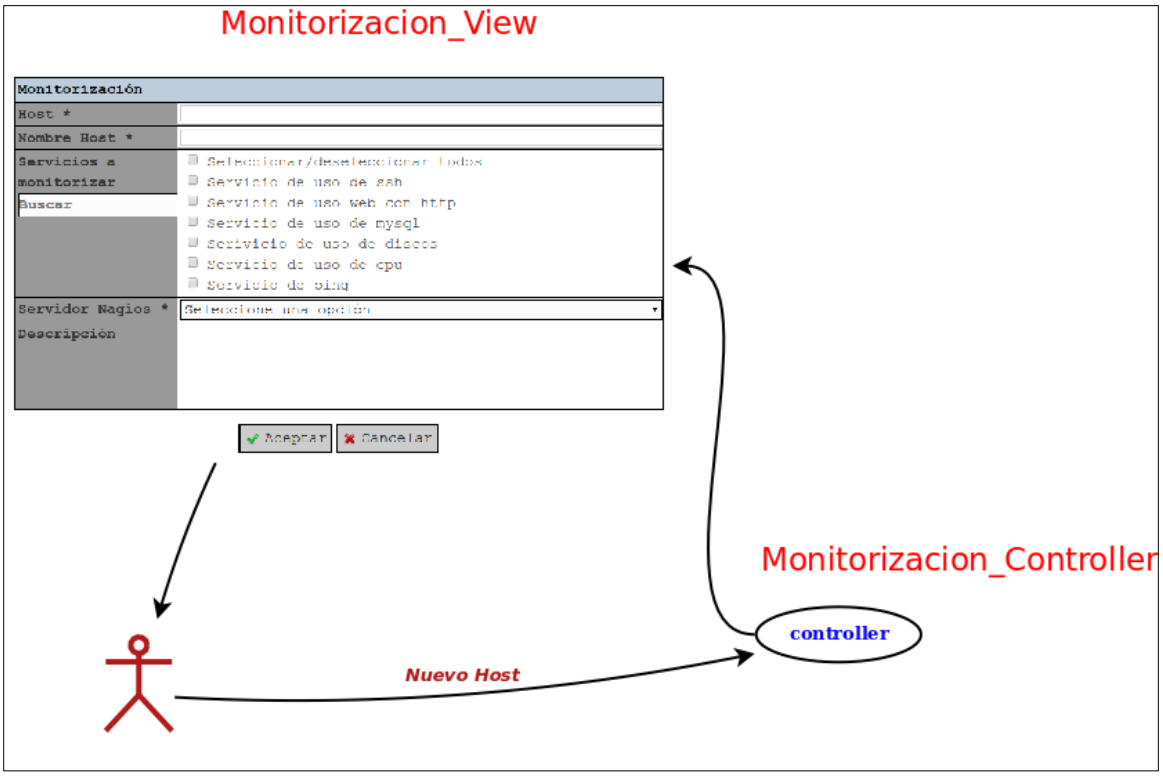


Ilustración 21: caso de uso dar de alta host a monitorizar

En este caso el agente solicita un nuevo *host* a monitorizar y recibirá la pantalla para poder registrarlo:

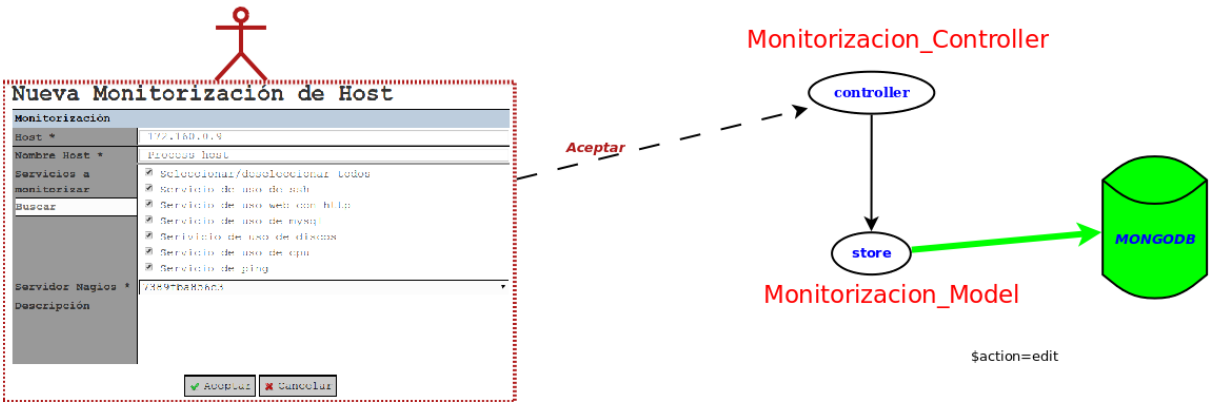


Ilustración 22: caso uso guardar host a monitorizar

Una vez completados los datos la idea es que al pulsar el botón Aceptar, el controlador invoque al modelo lógico para realice el almacenamiento (*store*) en el gestor de bases de datos y almacene la información en *MongoDB* en la colección de *host*, creando un nuevo documento (no relacional).

2.5 CRUD sobre nagios y host

Poder buscar, ver, modificar y borrar tanto servidores *nagios* como *host* a monitorizar

Estos a la vez serán requisitos tanto de *host*, como de *nagios*. Son las acciones básicas de cualquier gestor de bases de datos. Las realizaremos a través del controlador que activará en el modelo el método correspondiente pasándole valores a la instancia de la clase del modelo.

En los diagramas de caso de uso anteriores se puede intuir perfectamente este mecanismo, por lo que no se van a incluir, ya que es el mismo proceso pero utilizando el modelo lógico correspondiente asociado a otra colección de la base de datos MongoDB.

2.6 Establecer control de acceso

Para las acciones anteriores se establecerán dos zonas de acceso: (zona administración y zona pública) Cuando se ejecute el controlador deberemos de controlar con una variable que vendrá por *GET* en la solicitud si estamos en zona administración o pública, que discerniremos usando la variable **type.**

La idea es seguir con la misma metodología del *framework SmartRoads***,** que podemos ver en las siguientes imágenes.

Si estamos en opción *public* (zona pública para usuarios registrados) veremos solo la posibilidad de visualizar los datos del *host*. Esta opción también tendrá la posibilidad de ver los informes y gráficas como veremos más adelante, que es el objetivo principal del desarrollo.

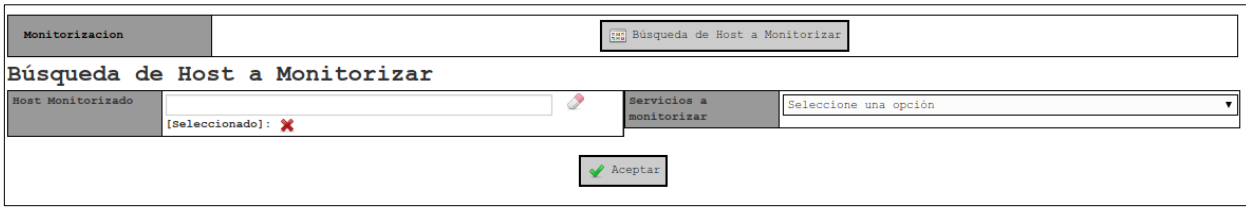


Ilustración 23: opciones monitorización en zona pública

El *mockup* preparado para la visualización de datos de los *hosts* monitorizados en zona pública será el siguiente:

En caso de que el usuario sea administrador, podrá acceder a la zona de administración. En este caso veremos la pantalla con todas las opciones (las 4 antes expuestas, para la gestión completa de servidores *nagios* y los *hosts* monitorizados por cada uno de ellos).

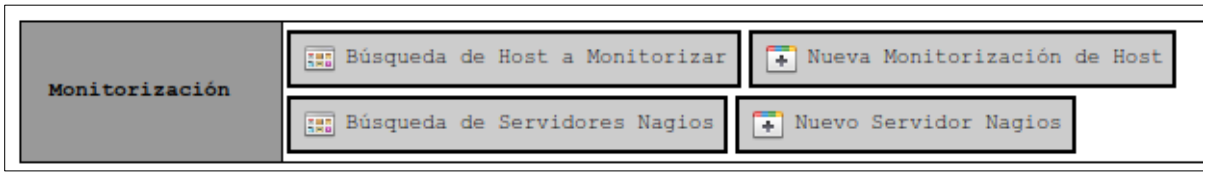


Ilustración 24: opciones zona admin

En este caso veremos la pantalla con todas las opciones (las 4 antes expuestas).

2.7 Comunicación app – Servidor nagios

Crear comunicación desde la aplicación mediante un servicio JSON/Rest a los servidores *nagios*.

Esta es una parte compleja en el desarrollo. Tendremos que instalar el *plugin* correspondiente en los servidores *nagios* para ofrecer una API JSON/Rest a la que conectarnos para obtener datos de los *host* monitorizados.

En nuestro módulo deberemos implementar el cliente JSON/Rest que consuma los datos de las *API* remotas de los servidores nagios. Utilizaremos el cliente de *API* que ya se encuentra disponible en el sistema SmartRoads, que está basado en *curl*. Básicamente se trata de un cliente *REST* genérico que permite enviar diferentes verbos en una solicitud *HTTP*.

Su invocación se va a realizar desde un método controlador. Este método se encargará de realizar peticiones a las API de los servidores *nagios* y recoger las respuestas *JSON* con los datos de los *host* monitorizados que luego procesaremos, como se puede ver en la imagen siguiente.

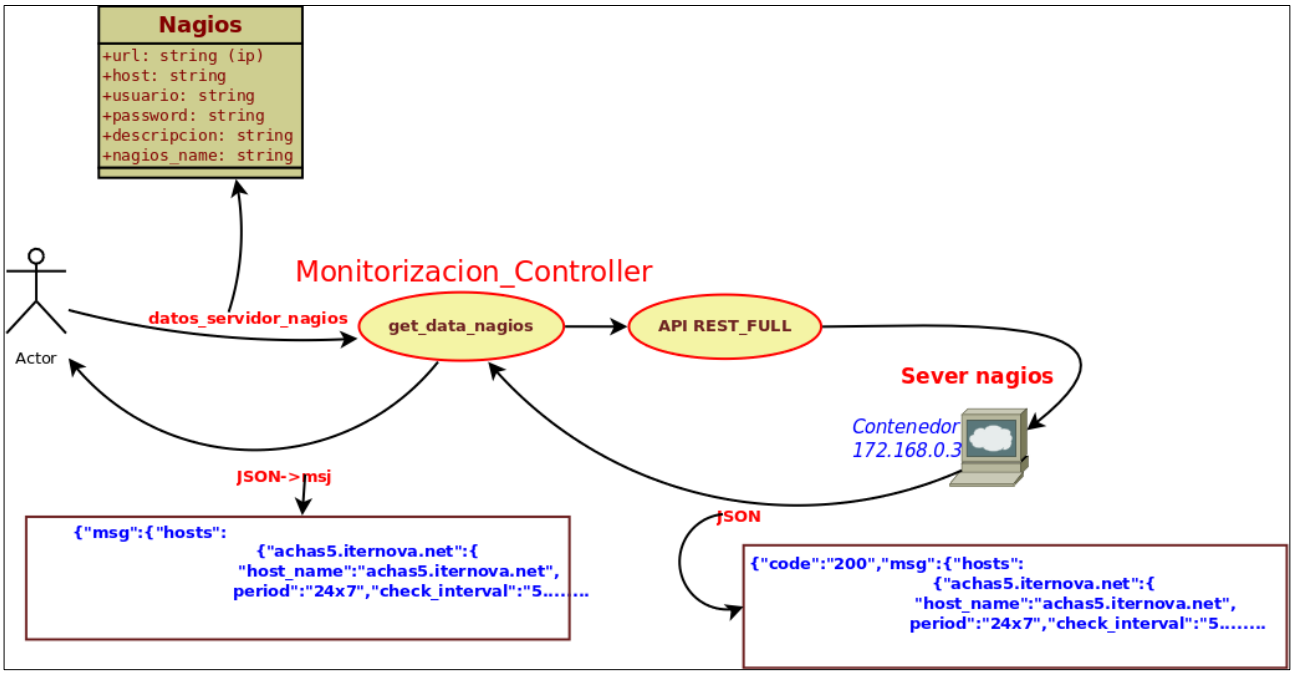


Ilustración 25: caso uso obtener datos servidor nagios

En cada petición recibiremos los datos en formato JSON con una estructura de datos definida que podemos ver en la imagen. De ella nos interesa el valor de la pareja **msg**, que será lo que devuelva este método, y que luego deberemos parsear y gestionar con el fin de almacenar la información requerida el el documento correspondiente de *MongoDB***.**

2.8 Almacenar datos de los servidores (RF 10, RF 11 y RF 12)

Recoger la información de los servidores y parsearla.

Al igual que el caso anterior es una parte fundamental en el desarrollo de esta aplicación. Para ello primero debemos establecer el escenario. Lo primero identifico tres elementos relacionados, pero independientes que tengo que hacer que interactúen entre ellos.

Por un lado tengo una colección de documentos con los servidores *nagios* (cofigurados en un modelo lógico), y por otro colección con los documentos de los *host* a monitorizar.

Ambos están relacionados en cuanto que cada *host* lo tenemos configurado para que sea monitorizado por un servidor *nagios* concreto de nuestra colección de *hosts*. La imagen ilustra esta situación.

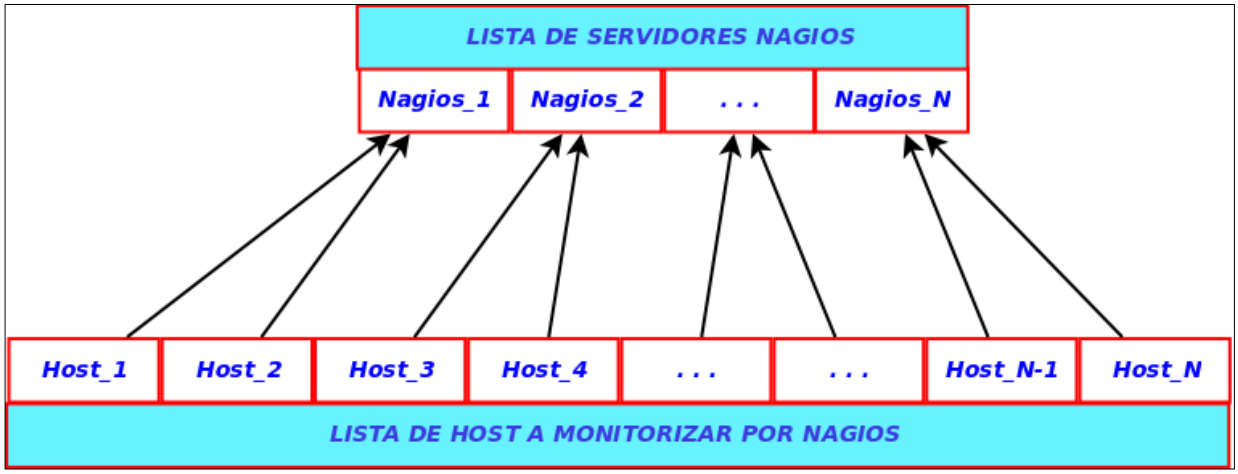


Ilustración 26: información en SmartRoads: host a monitorizar y servidores nagios

Su invocación se va a realizar desde un método controlador ejecutado de forma periódica (cada minuto). Este método se encargará de realizar peticiones a las API de los servidores *nagios* y recoger las respuestas *JSON* con los datos de los *host* monitorizados que luego procesaremos, como se puede ver en la imagen siguiente.

Ahora por otro lado tenemos la información estructurada en un *JSON* que nos ha sido devuelto por cada servidor *nagios* una vez realizada la petición a las API de dichos los servidores. Está solicitud la realizo en el método **get_data_nagios.**

En esta estructura *JSON* obtenida de un servidor *nagios*, tendremos uno o varios *host*; puede darse la circunstancia de que alguno de ellos esté o no esté en nuestra lista de *host* a monitorizar (lista definida en nuestro módulo *SmartRoads*). Este es un hecho muy importante que tengo que considerar a la hora de implementar el método

Vemos que tenemos un índice *host* que tiene todos los datos de los *hosts* disponibles en este servidor *nagios* y otro índice *service* que tiene todos los servicios que pueden ser monitorizados en cada *host*.

Ilustración 27: json con datos devueltos por el servidor nagios

A continuación establecemos un análisis mediante diagrama de uso de cómo voy a realizar el mapeo de los datos para construir/actualizar un documento y posteriormente almacenarlo en **MongoDB**

En la página siguiente se muestra un diagrama de uso del método muy importante en el desarrollo de la aplicación.

El método **get_data_server_nagios**, recibirá un JSON con el contenido del campo *msg* con los datos. Por otro lado obtiene información de todos los *nagios* y *host* que tiene configurado en el sistema.

Para cada servidor *nagios* recoge la información de cada índice que nos interese de cada servicio de cada *host* y lo incorpora en el objeto de modelo lógico correspondiente de la colección *data.* Al final guarda estos datos en un documento de la colección en la base de datos *MongoDB*.

Posteriormente en el diseño tomaremos decisiones de cómo realizar esta idea o pseudoalgoritmo.

En él veremos la manera para recoger sólo la información de los *host* configurados en el proyecto y no todos los que nos vengan del servidor *nagios*, que es parte del requisito 12.

Se puede observar a la vez, las diferentes bucles que tenemos, en el diagrama aparece el símbolo ∀representando un bucle de tipo **foreach** con el que recorreremos un *array*

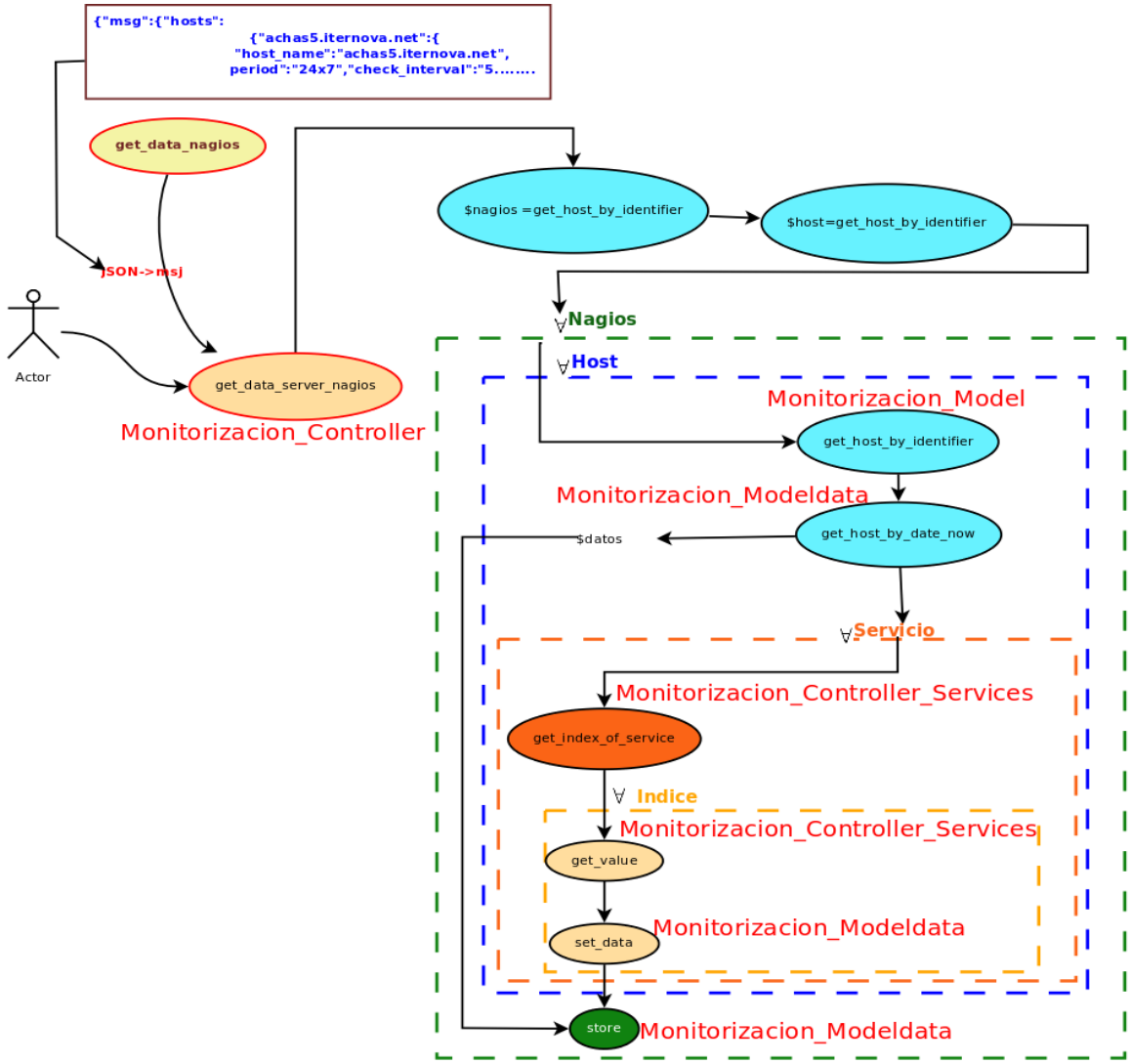


Ilustración 28: caso de uso get_data_server_nagios

El proceso es que representa es:

Para cada servidor nagios(en cada iteración Para cada host(....)).

Es muy importante comentar que en otro requisito que se va a a generar un documento por día y por *host,* el cual se actualizará cada minuto. Esto implica que el primer minuto de cada día ese documento no existirá y se tendrá que crear, el resto de los 1439 minutos del día ese documento ya existe y por lo tanto solo se tendrá que actualizar de forma eficiente.

Esto queda recogido con el método **get_host_by_identifier** el cual, antes de empezar a iterar para actualizar el documento (se actualiza al iterar para cada servicio en cada índice de ese servicio cada minuto), buscará el documento en la base de datos, y si existe lo devolverá para ser actualizado, y si no existe retornará *null* para indicar que se debe generar e inicializar un nuevo documento para ese *host* y día.

2.9 Configurar el sistema para realizar TimeSeries por minuto

Para este cometido se realizan estudios de los **Webinar MongoDB** donde se comentan *Series Data Web*, y hablan del proceso a seguir a la hora de realizar recogida de datos en intervalos relativamente cortos de tiempo. Es interesante el primer Webinar https://www.mongodb.com/presentations/mongodb-time-series-data, para familiarizarse con la importancia de estas técnicas (*time-series*) qué básicamente son acciones para establecer intervalos de tiempo para realizar algún tipo de medida. Estas son las técnicas que se utilizan en sistemas *IoT* (*Internet of Things*) para almacenar una gran cantidad de datos que evolucionan en el tiempo, como pueden ser datos meteorológicos, de control de flotas GPS o del estado en cada momento del motor de un vehículo. En caso de este proyecto, para conseguir ese requisito vamos a configurar el demonio *cron* para que cada minuto ejecute una llamada al método expuesto en el requisito anterior.Para ello configuraremos *crontab* del usuario *Apache* que se ejecute el método que tiene la aplicación *SmartRoads* llamado *crontab.php* cada minuto. Para ello escribimos en el fichero de configuración con el comando **crontab -e** (para editar)

```
* * * * * php  /var/www/smartroads-core/core/crondaemon/crontab.php
"/tmp/crontab.zpmYa8/crontab" 23L, 956C                    1,1           All
```

Ilustración 29: contenido crontab

Los asteriscos primeros marcan la temporalidad, el segundo parámetro es el comando a ejecutar, y el tercero son los parámetros del comando, en este caso el fichero php que queremos ejecutar. En nuestro caso el significado de cada asterisco

- Primer asterisco (*) => cada minuto
- Segundo => cada hora
- tercero => cada día del mes
- cuarto => cada mes
- quinto asterisco => cada día de la semana

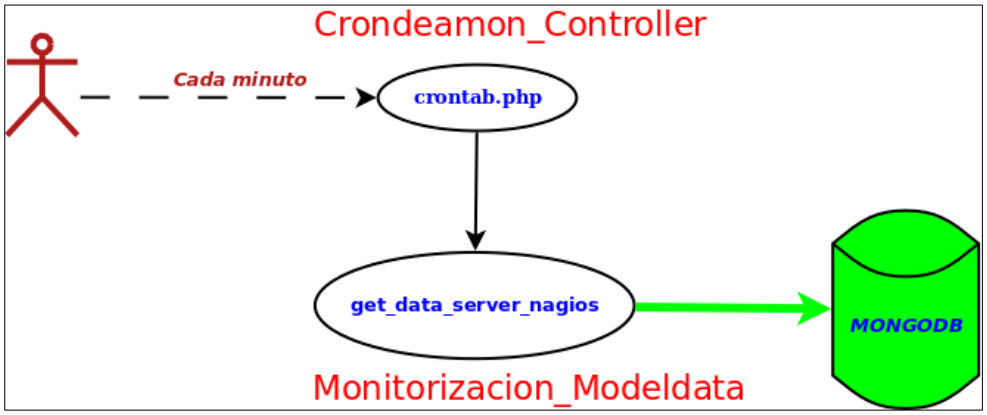


Ilustración 30: caso de uso demonio cron

El siguiente esquema muestra la idea del funcionamiento en nuestra aplicación

Crontab ejecutará en segundo plano este método y el de otros módulos configurados en otros módulos, realizando las llamadas a los métodos que necesitemos mediante el uso de colas ***gearman***. *Gearman* permite ejecutar las tareas en segundo plano sin bloquear el sistema, por lo que es transparente para el usuario, gestionando los recursos disponibles en el servidor en función de la disponibilidad. De esta forma, cada minuto se llamará al método de nuestro módulo ***get_data_server_nagios***, que será el encargado de realizar las peticiones a la API de los servidores *nagios* correspondientes.

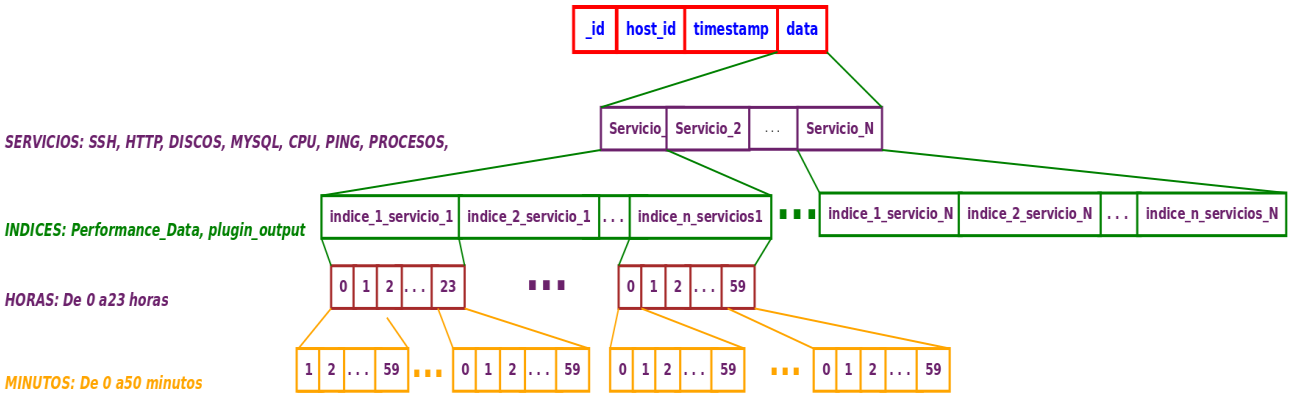


Ilustración 31: contenido model_data

2.10 Información a almacenar (RF14)

Vamos a establecer la estructura de los documentos donde vamos a almacenar la información (formato *time-series*). En el modelo lógico, estableceremos la clase **data** cuya estructura se muestra a continuación

Lo complejo de esta clase es establecer el *array* **data**. Guardaremos un documento por

Ilustración 32: clase data

host y por día **(host_id, timestamp)**. La estructura del *array* **data** sería la que muestra la imagen.

Los servicios en función de los configurados en el *host* a monitorizar serán los especificados o un subconjunto de ellos. Los servicios a monitorizar en el alcance de este proyecto se muestra a continuación, pudiendo añadir nuevos servicios de forma fácil y rápida:

SERVICIOS A MONITORIZAR
SSH
HTTP
DISCOS
MYSQL
CPU
PING
PROCESOS

Cada índice de los minutos serán las que contengan los valores. Los indices de cada servicio se establecen estos por ser los que interesan a la empresa. En la siguiente tabla tenemos un listado de los índices comunes a todos los servicios configurados que devuelve el servidor *nagios*.

INDICES DE LOS SERVICIOS DEVUELTOS POR UN SERVIDOR NAGIOS DE UN SERVICIO			
host_name	current_state	last_time_critical	passive_checks_enabled
service_description	last_hard_state	plugin_output	event_handler_enabled
modified_attributes	last_event_id	long_plugin_output	problem_has_been_acknowledged
check_command	current_event_id	performance_data	acknowledgement_type
check_period	current_problem_id	last_check	flap_detection_enabled
notification_period	last_problem_id	next_check	failure_prediction_enabled
check_interval	current_attempt	check_options	process_performance_data
retry_interval	max_attempts	current_notification_number	obsess_over_service
event_handler	state_type	current_notification_id	last_update
has_been_checked	last_state_change	last_notification	is_flapping
should_be_scheduled	last_hard_state_change	next_notification	percent_state_change
check_execution_time	last_time_ok	no_more_notifications	scheduled_downtime_depth
check_latency	last_time_warning	notifications_enabled	
check_type	last_time_unknown	active_checks_enabled	

De todos ellos en la empresa me informan que sólo interesa **plugin_output** y **performance_data.** No obstante crearé un método que me devuelva los índices de cada servicio, con lo que si se quisiera posteriormente modificar este requisito se podría hacer fácilmente.

2.11 Crear documentos en MongoDB de data (RF 13)

Para ello ya hemos visto la clase **data,** expuesta en puntos anteriores para almacenar la información. Crearemos una clase de modelo lógico **Monitorizacion_ModelData**, que será el modelo de datos a almacenar.

La forma de crear/guardar los documentos queda perfectamente visible en la imagen *casos de uso de get_data_server_nagios*

Cabe resaltar como se puede ver en la imagen que lo primero que hago es recoger el documento con fecha de hoy si existe en la base de datos, para actualizarlo.

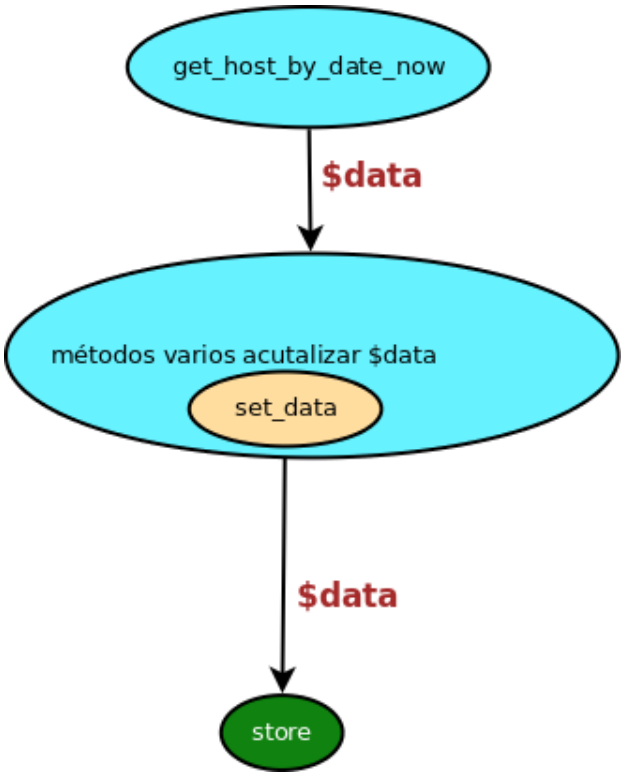


Ilustración 33: guardar datos en coleccion data

La idea de los *time-series* es la siguiente:

Lo primero buscaré en la colección de MongoDB si existe un documento de un *host* concreto para el día actual (estaremos en la iteración de para cada *host* del día actual).

Posteriormente actualizaré ese documento con los valores de los índices de los diferentes servicios que nos da *nagios* de cada *host* para el minuto actual y se actualizará en la colección *MongoDB*.

En el diseño deberemos de tener en cuenta que si no hubiera ningún documento de ese host y día previamente almacenado en la base de datos (*$data será null*), habría que inicializarlo con la estructura correspondiente.

Al final lo almacenaremos usando el método *store* del modelo lógico **ModelData (Monitorizacion_ModelData).**

2.12 Obtener informes (RF 15, RF 16 RF17 y RF18)

Los informes y gráficas serán de cada *host*, por lo que está opción estará disponible cuando se acceda a la ficha de visualización del *host*.

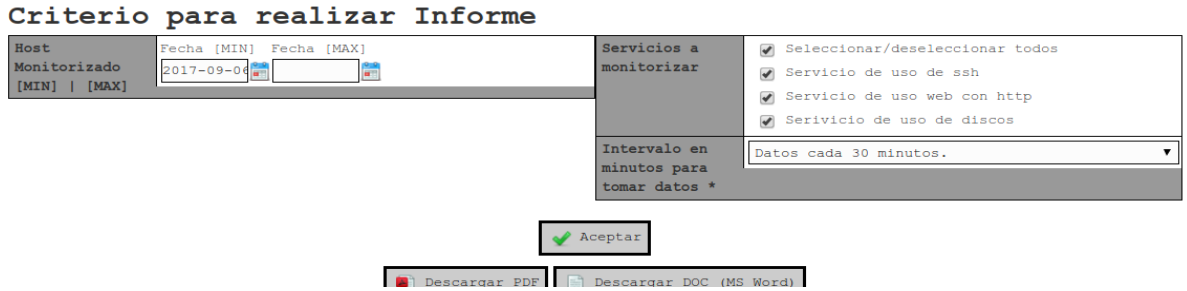
Antes de mostrar un informe que se podría hacer directamente decidiendo la aplicación directamente los intervalos (por ejemplo, datos del último día). Consultando con el tutor y técnicos, me comentan que es interesante que el usuario de la aplicación tenga diferentes opciones para solicitar informes, si bien no ven como requisito imprescindible que pueda el usuario personalizar totalmente el informe (quedaría pendiente para futuras mejoras, fuera del alcance de este PFC).

Consideraremos un informe mostrar información bien en forma de gráfica y en formato tabulado de los diferentes valores de cada índice de un determinado servicio del *host* monitorizado a un intervalo en minutos establecido (promedio cada 30 minutos o cada día por ejemplo, si los rangos temporales a visualizar son extensos); De esta forma se establecerá la media de los valores en cada intervalo. Por ejemplo si es cada 30 minutos, no mostraremos el valor de los minutos 0 y 30 de cada hora, sino la media aritmética de los valores de 1 a 30 en y de 31 a 0. Esto se establecerá entre dos intervalos de fechas del día inicio al día fin

Se implementan tres tipos de informes cómo se muestra en la tabla:

TIPOS DE INFORMES	
Datos 3 últimas horas	Cada minuto de las tres últimas horas, con resolución de minutos.
Datos cada 30 minutos	Se establecerá un fecha inicio y una fin, y se mostrará los datos cada media hora haciendo media aritmética.
Datos media de cada día	Se establecerá un fecha inicio y una fin, y se mostrará la media aritmética de cada día, mostrando un dato por día.

Por lo tanto en la pantalla necesitaré dejar al usuario que establezca una fecha de inicio, una de fin y un tipo de informe. También se deja la opción de visualizar sólo un determinado servicio, u no todos los almacenados (puede ser que en un momento determinado solo interese ver la carga de CPU, o si hay conexión con PING, o la carga de disco, etcétera).



La siguiente imagen muestra un *mockup* de la pantalla para poder llevar a cabo estos requisitos.

Podemos ver las opciones de informe

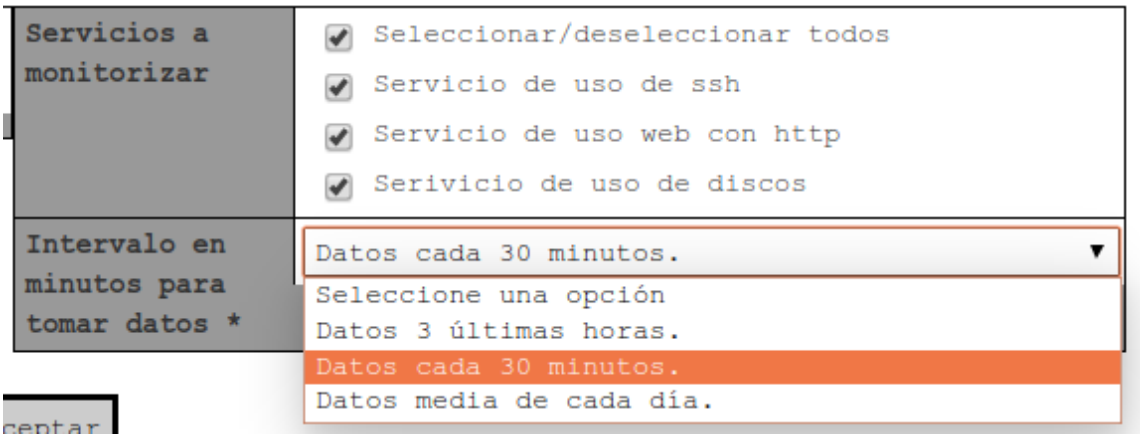


Ilustración 35: listado de opciones de informes

A continuación vamos a ver los requisitos de cómo obtener los informes en los intervalos establecidos

2.13 Obtener los gráficos

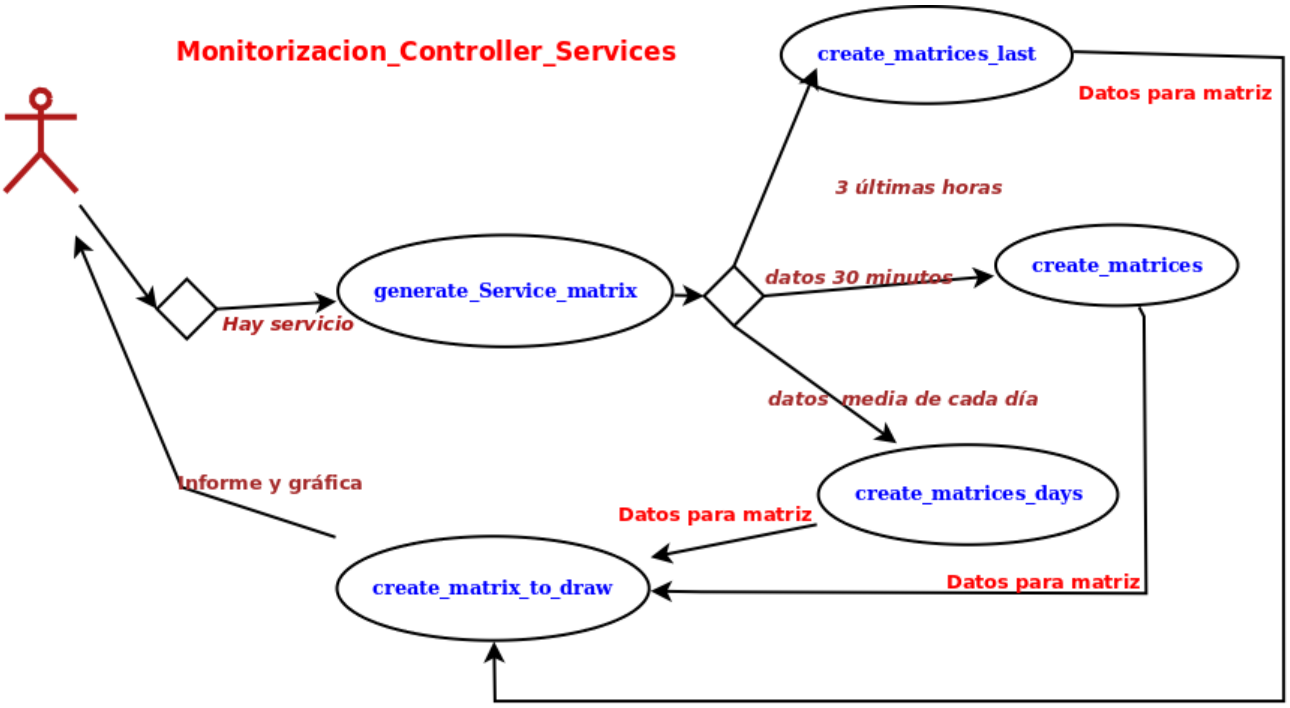


Ilustración 36: Casos de uso para generar matrices

Para obtener los informes gráficos voy a utilizar una API de Google: **Charts_Google();** Para ello debemos generar una matriz con dos columnas, una de fechas, y otras de valores del servicio del *host* para dichas fechas. Una vez que el usuario haya seleccionado su opciones de informes para un determinado *host*, planteamos el siguiente caso de uso para este cometido: La manera en la que se generan los datos para las matrices en cada caso se analiza en el diseño

2.14 Realizar pruebas de testeo

Requisito número 29, y uno de los más importantes. Para las pruebas de testeo se realizarán test unitarios utilizando *phpunit*, testeando los métodos que vayamos realizando.

Siguiendo la estructura marcada en Iternova dentro de su desarrollo **SmartRoads**, creamos un módulo en su carpeta de test para especificar el módulo de monitorización.

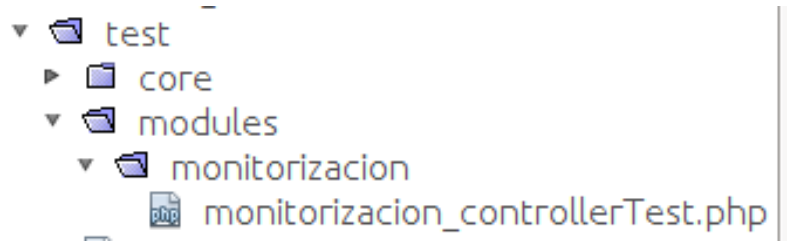


Ilustración 37: test para el módulo de monitorización

Dentro del método de la clase ***Monitorización_Controller_Test***, es dónde escribiremos un método por cada método que queramos monitorizar.

Para el testeo tenemos que instalar **phpunit** y para invocarlo simplemente es llamar al programa *phpunit* y pasarle como argumento la ubicación de la clase *Monitorización_Controller_Test*.

Realizar estas pruebas de testeo asegurarán que funcionen todos los métodos y funciones generados como esperamos, y que si en el futuro se actualiza algún método se pueda comprobar que sigue funcionando todo como se espera.

2.15 Generar documentación del código y manuales de usuarios (RF 30)

Durante la realización del proyecto siguiendo la política de desarrollo de la empresa, el tutor me informa que todos los métodos han de estar correctamente comentados, siguiente el sistema de directivas de **phpdoc**. El hacerlo correctamente es una gran ventaja ya que tenemos una ayuda online sobre lo que devuelve, lo que retorna y lo que hace. Por otro lado es algo fundamental para que otros desarrolladores en un momento dado pudieran tener acceso al código de forma legible, e incluso para nosotros mismos en el futuro. La documentación es parte del software aunque no genere instrucciones al compilarse o interpretarse.

```
/**
 * Métdo para generar una matriz de datos que posteriormente será dibujada gráficamente

 * @param string $data_ini fecha inicio
 * @param string $data_fin fecha fin
 * @param string $host_id host para el que se quiere mostrar la matriz
 * @param string $services  host a monitorizar (SSH, HTTP, ...)
 *
 * @return array con tablas para cada servicio con valores
 *                 fecha - Valor del servicio XXX
 */
public static function generate_service_matrix( $date_ini, $date_fin, $host_id, $interval, $services ) {

    $matrices_data = [];
```

Ilustración 38: método comentado

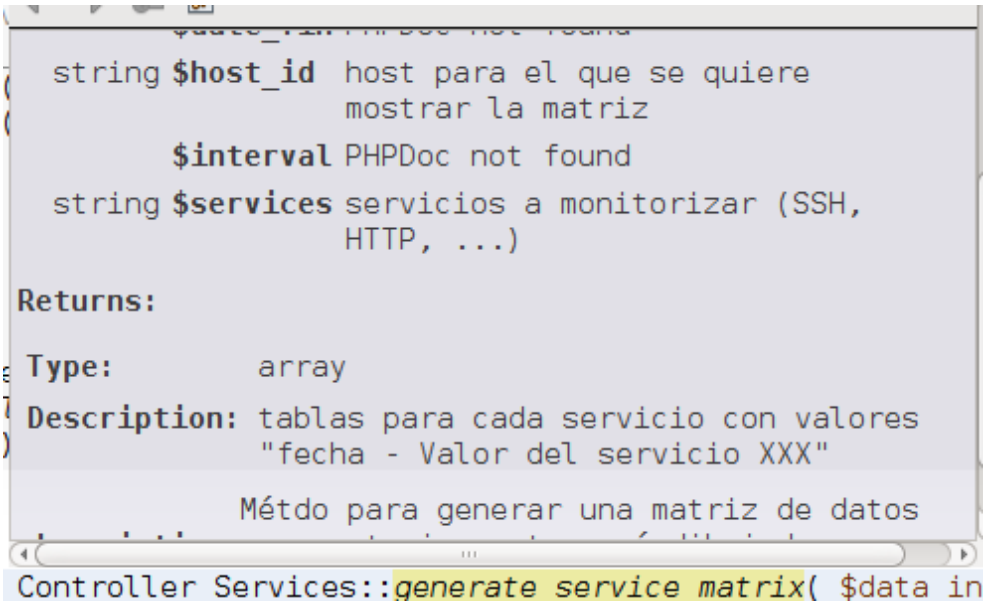


Ilustración 39: ayuda en línea de Netbeans sobre un método comentado

Ahora al usar el método veo cómo el IDE de desarrollo *Netbeans* me da ayuda en línea Asimismo se generará documentación de usuario (manuales) para explicar de forma sencilla y rápida cómo configurar los diferentes *hosts* y servidores *nagios* en el sistema *SmartRoads*.

3 DESARROLLO E IMPLEMENTACIÓN

Después de plantear y analizar de manera breve los requisitos en el apartado 3 de la memoria, así como las herramientas a usar, se va a explicar el desarrollo llevado a cabo en la realización de este proyecto. La metodología de trabajo se describe en la siguiente imagen.

La realización de un diseño previo (prototipado) de lo que se va a desarrollar es fundamental en cualquier tipo de proyecto, puesto que ayuda a organizar el trabajo a realizar y sirve de hoja de ruta en el posterior desarrollo. Además, es muy útil tanto para el cliente, como para la empresa. Por un lado el cliente se hace una idea visual del proyecto, y por otro lado sirve a la empresa como parte del pliego de condiciones, delimitando el desarrollo y evitando posibles discusiones sobre la realización del mismo debido a peticiones del cliente no contempladas. Únicamente se desarrolla lo que se valida.

Para realizar el diseño, se empezó usando *Balsamiq*, pero al final se ha utilizado un desarrollo realizado por Iternova que permite realizar los diseños de pantallas como se han mostrado en la parte del análisis los cuales se toman como los prototipos.

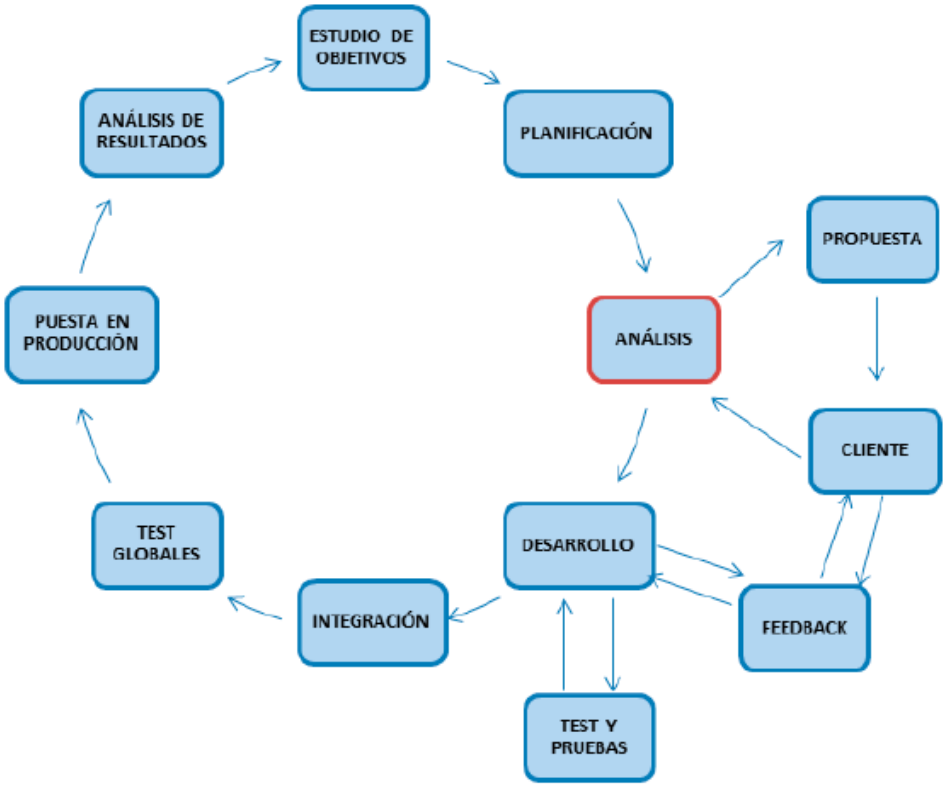


Ilustración 40: metodología de trabajo

A la vista del esquema de la metodología de trabajo, vemos que se parte de un estudio de los objetivos lo cual nos lleva a la planificación de las diversas tareas. Una vez conocidas las tareas se realiza el análisis de requisitos teniendo en cuenta las necesidades del cliente que harán que se produzca un cierto feedback o retroalimentación. En este caso la retroalimentación ha sido llevada a cabo por el cuerpo técnico de Iternova y su cliente. De esta manera se comienza con el desarrollo de las distintas funcionalidades realizando continuamente diversos test y pruebas para garantizar que el desarrollo es el adecuado en cada momento. La parte de integración, test globales y puesta en producción no han sido del alcance de mi proyecto a la hora del desarrollo sino que será mi director el que las realice para finalmente llegar al análisis de los resultados mediante el cual se ponga de manifiesto todo el trabajo realizado.

3.1 Entorno de Desarrollo (RNF1)

El proyecto se ha desarrollado haciendo uso del entorno de desarrollo de *Netbeans.* Se trata de un IDE libre, de código abierto, con soporte para múltiples lenguajes de programación y con una *comunidad mundial de usuarios y desarrolladores.*

El principal lenguaje de programación utilizado en la empresa, y también en este proyecto, es PHP. Es un lenguaje muy conocido, con un repertorio amplio de funciones y extensa documentación en Internet sobre su uso. HTML es el lenguaje que se emplea para el desarrollo de páginas de internet, dándoles estructura y contenido. Está compuesto por una serie de etiquetas que el navegador interpreta y da forma en la pantalla, utilizando hojas de estilo en cascada CSS (generadas mediante SASS y ficheros .scss) y el uso del framework bootstrap para construir los interfaces gráficos de usuario. Asimismo, el código HTML generado con diversa información puede utilizarse para generar más adelante documentos y hojas de cálculo compatibles con Word / Excel / PDF mediante las librerías propias de la empresa. Igualmente se utiliza JavaScript + AJAX (mediante el framework *jQuery* principalmente) para la carga de datos asíncronos y actualización de elementos DOM (Document Object Model) de los interfaces de las diferentes pantallas de la aplicación.

Durante el desarrollo no utilizo directamente *MySQL*, aunque es fundamental para el funcionamiento del sistema. El almacenamiento de datos devueltos por las API *nagios* se ha realizado utilizando *MongoDB*,

MySQL se trata de una base de datos relacional, es decir, un conjunto de una o más tablas estructuradas en registros (líneas) y campos (columnas), que se vinculan entre sí por un campo en común. (clave principal con la clave foránea estableciendo una restricción de integridad referencial). MongoDB va a ser el gestor de bases documental (no SQL), que vamos a utilizar para almacenar la información. *MongoDB* es un sistema ideal para trabajar con series de datos grandes en formato *time-serie* y actualizarlo. Es prácticamente inmediato guardar el documento, ya que se trata de documentos formados por *array* asociativos, una estructura muy común en PHP**.**

Para poder trabajar fuera de la aplicación con MongoDB y ver colecciones se usa **Mongochef.** Es una herramienta muy útil y sencilla de manejar.

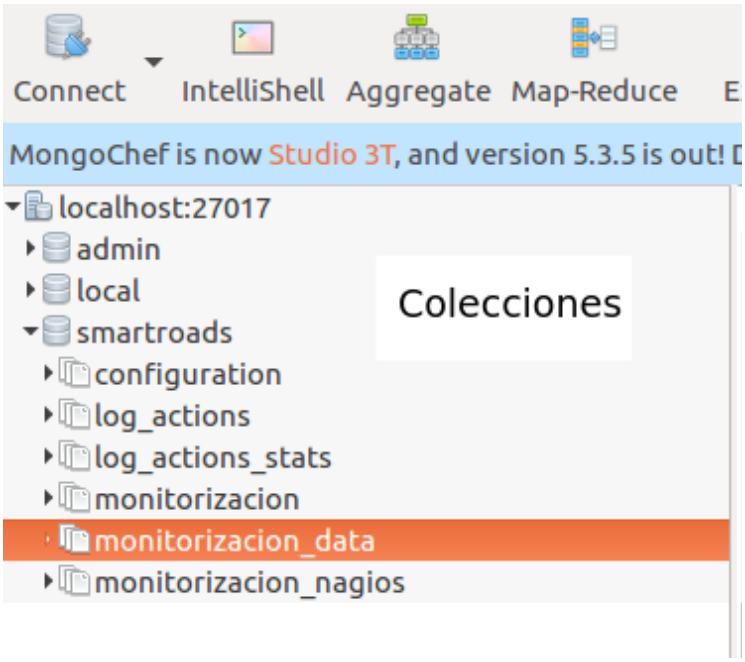


Ilustración 41: Mongochef muestra las colecciones

MongoChef también nos muestra los documentos de cada colección y nos permite realizar consultas y actualizar los datos. En la siguiente imagen tenemos seleccionada la colección **monitorizacion_data**

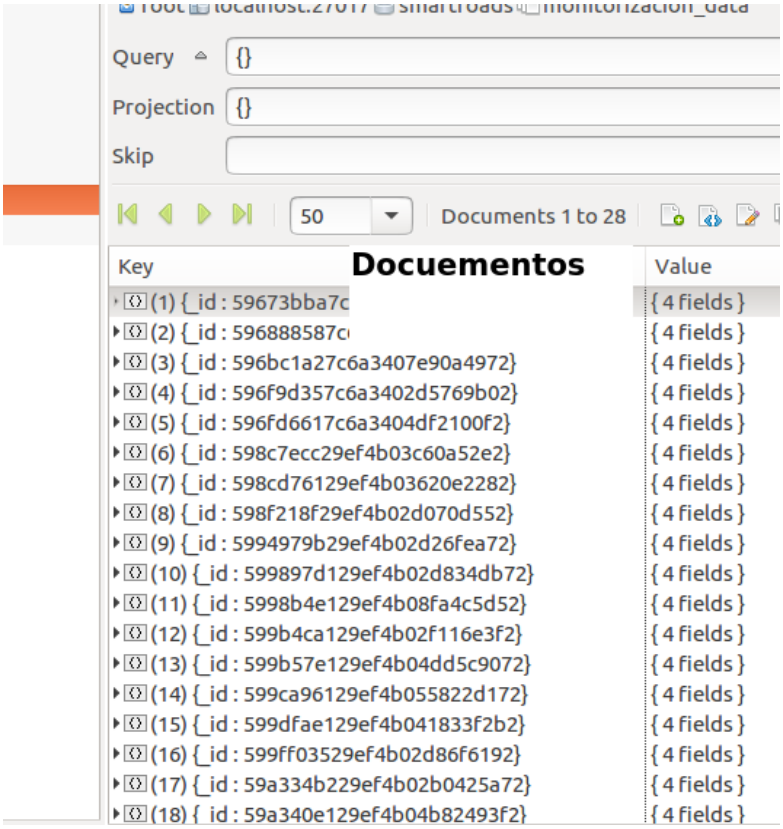


Ilustración 42: Documentos de una colección

Para realizar el arranque de todo el sistema de red dockerizado se ha realizado un script sencillo cuyo código es

```
;inicio los contenedores docker
;hay que hacerlo por orden para asignar la ip de forma consecutiva
;empezará asignado 172.17.0.2 hasta la 172.17.0.10
docker start itenova
docker start nagios1
docker start nagios2
docker start host1
docker start host2
docker start host3
docker start host4
docker start host5
docker start host6

;ejecutamos en cada contenedor un scrip que está cargado para ejecutar los servicios (apache, nagios, myslq, ssh, según corresponda.
docker exec iternova  bash ./servicios
docker exec nagios1 bash ./servicios
docker exec nagios2 bash ./servicios
docker exec  host1 bash ./services
docker exec  host2 bash ./services
docker exec  host3 bash ./services
docker exec  host4 bash ./services
docker exec  host5 bash ./services
docker exec  host6 bash ./services


;Arrancamos este contenedor en un terminal  -ti es para que se quede interactivo, ejecutamos el bash
gnome-terminal -e "docker exec -ti iternova bash"

;estos contenedores en pricipio no tengo por qué tener terminal de ellos,
;dejo opcional si paso un argumento al scritp que los arranque

if [ -z "$1" ]
then
      gnome-terminal -e "docker exec -ti nagios1 bash"
      gnome-terminal -e "docker exec -ti nagios1 bash"
      gnome-terminal -e "docker exec -ti host1 bash"
      gnome-terminal -e "docker exec -ti host2 bash"
      gnome-terminal -e "docker exec -ti host3 bash"
      gnome-terminal -e "docker exec -ti host4 bash"
      gnome-terminal -e "docker exec -ti host5 bash"
      gnome-terminal -e "docker exec -ti host6 bash"
fi
```

Se ha realizado la implementación de todos los requisitos analizados previamente. Exponemos y mostramos los requisitos más directos para documentar su implementación agrupándolos funcionalmente:

3.2 Creación del módulo y los modelos lógicos (RF2, RF5, RF6 y RF7).

La creación del módulo (RF 2) y los modelos lógicos (RF 5, RF 6 y RF 7), es la primera tarea a realizar. Una vez realizado ya es fácil crear las vistas y se puede emepzar a ejecutar la aplicación, y rápidamente realizar las acciones CRUD sobre el gestor **MongoDB.**

Para la creación del módulo, una vez creada la estructura de carpetas, debemos establecer el código correspondiente. Tenemos una filosofía de trabajo **MVC** ya explicada en el análisis, por lo tanto debemos crear el modelo lógico, el controlador y la vista. Al principio el controlador sólo entregará la pantalla o vista, y según avancemos en el código irá realizando más acciones.

Creamos el modelo que es la base de todo. Lo hacemos a partir del diagrama de clase de *host* (Imagen 10) , ya que necesitaremos los campos para la pantalla cuyo prototipo ya está establecido. En los siguientes ejemplos se han eliminado comentarios para no exceder, para verlo completo ver el anexo correspondiente al código.

```
class Monitorizacion_Model {
        /**
          * @var string $_db_collection Nombre identificador de la coleccion en la base de datos
          */
        private static $_db_collection = 'monitorizacion';
        /**
          * @var string $_id Identificador de registro dentro de la coleccion monitorizacion
          */
        public $_id = null;
        public $host = '';
        public $host_name = '';
        public $servicios = [];  //
        public $nagios_id = '';
        public $nagios_name = '';
        public $descripcion = '';
```

En la imagen vemos los métodos implementados, no habiendo ninguna acción especial, corresponderían a acciones básicas crud para recuperar, almacenar documentos de *host* de la colección correspondiente en **MongoDB**

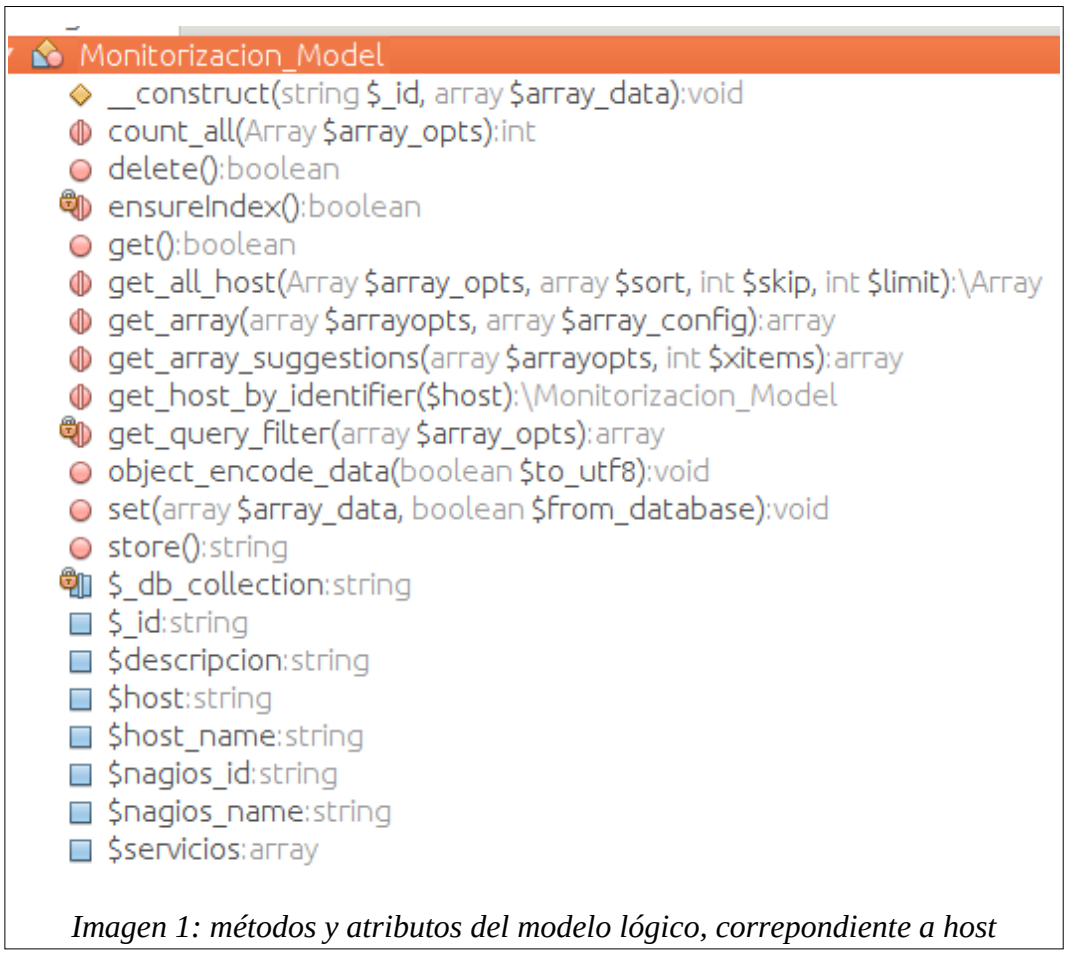


Imagen 1: métodos y atributos del modelo lógico, correpondiente a host

En total se crean 3 modelos lógicos de la misma manera

Modelos lógicos del módulo de monitorizacion	
Monitorizacion_Model	Modelo lógico para los host
Monitorizacion__Modeldata	Modelo lógico para la información que almacenamos de la monitorización de los host por parte de los servidores nagios.
Monitorizacion_Modules_nagios_model	Modelo lógico para los servidores nagios

El nombre del modelo lógico de nagios, se debe a que en la estructura de carpetas del módulo, quise crear una subcarpeta para nagios como se puede ver en la imagen. Para que funciona el mecanismo de autoload al invocar cada clase el nombre del fichero PHP debe ser el de la clase, indicando el nombre o mediante el namespace PHP la ruta en la que está ubicado.

3.3 Vistas: creación de la plantillas (RF3, RF4, RF15)

La creación de plantillas ser realiza con el *framework* de Iternova, por lo que en este aspecto, lo que he tenido que hacer es aprender a utilizarlo, sin mas, y posteriormente implementar para que los prototipos de pantallas o *mockups* diseñados (ver imágenes 8,9,11,13,14 y 15 de la sección de análisis), puedan crear la pantalla esperada en la aplicación.

De los tres modelos lógicos (Ver apartado anterior), crearemos 2 vistas, ya que el modelo **ModelData** que gestionará los documentos de los datos a monitorizar se almacenarán directamente del mapeo de los datos JSON que vengan de los servicios API de los servidores *nagios* y no necesitan pantalla.

Vistas del módulo de monitorización	
Monitorizacion_View	Vista los *host* del modelo *Monitorizacion_Model*
Monitorizacion_Modules_nagios_view	Vista de los *nagios* para el modelo *Monitorizacion_Modules_nagios_model*

Un pequeño ejemplo de cómo se implementan las vistas y exposición de la flexibilidad que tiene la aplicación en general **SmartRoads** y por lo tanto también el módulo de *monitorización* para adaptarse a idiomas. Para ello y poder hacer las traducciones correspondientes, en lugar de poner los textos directamente en el código, se usa un fichero dónde se indica, los textos y se asignan a partir de variables, de forma que realizar una traducción, simplemente consistirá en indicar el idioma y rellenar los textos traducidos en otro ficheros.

La imagen muestra la estructura de la carpeta

Parte de los contenidos de ese fichero donde vemos como ponemos etiquetas a los textos, esa misma etiqueta puede tener otro texto en otro idioma en otro fichero.

```
adjunto=Archivo adjunto
archivo=Archivo
aplicacion=Monitorización
version=Versión
monitorizacion=Monitorización
monitorizacion=Host Monitorizado
monitorizacion.search_form=Búsqueda de Host a Monitorizar
monitorizacion.edit=Editar Host Monitorizado
monitorizacion.add=Nueva Monitorización de Host


monitorizacion.intervalo.ultima_hora=Datos 3 últimas horas.
monitorizacion.intervalo.medio=Datos cada 30 minutos.
```

Y su uso en el código por ejemplo en la vista para crear el formulario , así muestro también el código de como se crea un formulario siguiendo un *framework* propio de Iternova para el desarrollo. Este código corresponde a parte del método **edit** de la clase correspondiente a la vista del host **Monitorizacion_View** Podemos ver cómo cuando asignamos un valor al *title* de un campo lo hacemos a través del contenido del fichero que hemos especificado, usando la clase **Idiomas_Controller.** También se puede establecer diferentes tipos de campos de formularios según las necesidades, mediante el uso de arrays de configración

```
 //**
  * Clase controladora de vista para administrar monitorizaciones  de SmartRoads*/
class Monitorizacion_View {

//...más código: atributos y métodos..//

 /**
        * Visualiza formulario para crear o editar un caso de test y sus condiciones
        * @paramMonitorizacion_Model $obj Objeto de tipo Controller_Modules_Configuration_Model con datos de cada fichero INI
        * @return boolean Devuelve true si los datos del formulario son correctos y est?n disponibles en la variable $_POST
        */
       public static function edit($obj) {
//.. instrucciones de inicialización de variables de entorno y otras acciones..//
//....
   //Segunda columna 'host'
                    echo '<div class="col-xs-12 col-md-4">';
                    $error[] = Controller_Forms::field(array('field_type' => 'text',
                                    'field_id' => 'host',
                                    'field_title' => Idiomas_Controller::translate('monitorizacion.host', $idioma, $langfile),
                                    'field_value' => $obj->host,
                                    'obligatory' => true,
                                    'first' => $first));

                    //Tercera columna 'host_name'
                    echo '</div><div class="col-xs-12 col-md-4">';

//Recogemos un array error posibles errores a la hora de crear los items html
                    $error[] = Controller_Forms::field(array('field_type' => 'text',
                                    'field_id' => 'host_name',
                                    'field_title' => Idiomas_Controller::translate('monitorizacion.host_name', $idioma, $langfile),
                                    'field_value' => $obj->host_name,
                                    'obligatory' => true,
                                    'first' => $first));


//.... Mas instrucciones finales ...//
}
```

En función de los prototipos podemos crear campos de formularios y diferentes *layouts* o bloques de contenidos. En los bloques establezco los valores del atributo **class** de los elementos **div** que encierra los diferentes campos de formularios para definir su tamaño y posición, mientras que los métodos controladores de generación de campos de formulario generarán el código HTML necesario y controlarán los datos introducidos y su validez. Esta información la interpreta el *framework* de CSS *Boostrap* usado en la aplicación **SmartRoads.**

3.4 Los controladores o lógica de negocio (RF10, RF11, RF12, RF13 y RF14).

Esta es la parte donde recae el código más elaborado del proyecto. Se implementarán el las clases controladores y crearemos clases adicionales para separar más el código siguiente una lógica funcional.

Podríamos considerar las siguientes partes que por el esfuerzo de desarrollarlas considero más interesantes en la lógica del proyecto.

Los métodos implementados para recoger y mapear la información de las API JSON/Rest de los servidores *nagios*, están en las siguientes clases: **Monitorizacion_controller y Monitorizacion_controller_Services.**

Para recoger la información de los servidores *nagios*, he utilizado el cliente de la API JSON/Rest, que es el que usa por defecto el sistema. Para poderlo usar se ha analizado, ya que si bien no hay que implementarlo (lógicamente no tiene sentido reinventar la rueda, siendo que además está testeado y preparado de forma genérica para solicitudes particulares), se ha estudiado su funcionamiento. También se ha estudiado el uso de **composer** (un manejador de dependencias y paquetes que nos permite utilizar librerías en las versiones necesarias e cada proyecto), ya que también es posible utilizar otras librerías existentes como guzzle para crear clientes API JSON/Rest.

El método que recoge la información de la API JSON/Rest se ejecuta de forma periódica cada minuto utilizando *cron*, como se ha expuesto en el apartado de análisis/diseño.

Para permitir que el sistema *SmartRoads* realice la llamada de forma periódica (cada minuto) a los servidores *nagios* remotos se ha realizado la configuración indicada en el análisis y se ha desarrollado el método correspondiente en el módulo *monitorización*. Cada minuto se introduce el método en la cola de tareas de ***gearman*** para que se ejecute en segundo plano, sin bloquear al resto del sistema. Cuando hay recursos suficientes en el servidor y le llega el turno a nuestra tarea (en las colas puede haber otras tareas de otros módulos), se realiza la llamada al método correspondiente de nuestro controlador que realiza las peticiones a las API *nagios* y obtiene los datos de cada *host* y servicio configurados, para luego ser parseados y almacenados como corresponda.

Este método realiza las siguientes acciones: Primero obtenemos todos los documentos de *host* y todos los documentos de servidores *nagios:*

```
/**
      Método para obtener datos del servidor nagios
     * Para ello invoco al método del cliente usando API JSON/Rest

     * Método kernel de la parte de monitorización
     * Para cada servidor nagios configurado en la app
     * Recogo datos de los host que monitoriza cada server nagios
     * Para cada host configurado si ese servidor nagios me da datos
     * Los almaceno en la colección correspondiente en mongo o creo una nueva
     *
     */

 public static function get_data_server_nagios() {
            //Obtenemos todos los los servidores nagios de mi modelo y
            //los host a monitorizar
            $servers_nagios = Monitorizacion_Modules_Nagios_Model::get_all();
            $hosts_monitorizar = Monitorizacion_Model::get_all_host();

            //Lo convierto a array para trabajar siempre de la misma manera
            $servers_nagios = Controller_Utils::cast_object_to_array( $servers_nagios );
            $hosts_monitorizar = Controller_Utils::cast_object_to_array( $hosts_monitorizar );

     }
```

Posteriormente vamos a establecer una serie de iteraciones para cada servidor *nagios*, para cada *host*, para cada servicio y para cada índice, de manera que podamos ir obteniendo los datos:

```
/* Para cada servidor Nagios registrado en la app */
    foreach ( $servers_nagios as $nagios ) {
        /* Para cada host a monitorizar
        foreach ( $hosts_monitorizar as $host ) {
        //Objeto Monitorizacion::ModelData a almacenar en la bd para este host

        $datos = null;
        //Solo si ese host está configurado para ser monitorizado por este nagios

        if ( $host[ 'nagios_name' ] == $nagios ) {

            //Recogemos los servicios configurados para monitorizar de éste host
                        $host_services = $host[ 'servicios' ];

                        //Localizamos el host de la colección
                        $host_mongo = Monitorizacion_Model::get_host_by_identifier( $host_ip );

                        $host_id = $host_mongo->_id;
        //Guardo un documento por host y por día
        //Si ya existe docuemnto de ese día y ese host lo recupero para actualizarlo
        //Si no existe $datos valdrá null y lo inicializaré posteriormente
        $datos = Monitorizacion_Modeldata::get_host_by_date_now( $host_id );


                    //Para cada servicio a monitorizar
                      foreach ( $host_services as $service ) {

            //Obtenemos la lista de índices de ese servicio

    $indexes_service =Monitorizacion_Controller_Services::get_index_of_service( $service );
                    //Para cada índice que quiero monitorizar
                      foreach ( $indexes_service as $index_service ) {
Obtengo el valor que me devuelve nagios y se lo asigno al docuemnto $data que al final almacenré
 //Aquí vamos a extraer la parte del valor que nos interesa
                                        $service_value = Monitorizacion_Controller_Services::get_value( $index_service, $service_value );

                                        $datos = Monitorizacion_Modeldata::set_data( $datos, $host_id, $service_string, $index_service,
$service_value );


                    } //End si host es monitorizado por ese
                }//End foreach host

//Almaceno la información
        if ( $datos != null ) {
                                $datos->store();
                    }
```

Es interesante ver cómo podríamos fácilmente incrementar los índices que nos interesan. Recordemos como en el análisis vimos que son muchos los índices que nos devuelve *nagios* de cada servicio que monitoriza, si bien sólo se requiere actualmente un conjunto reducido de ellos. Esto está implementado en el método *Monitorizacion_Controller_Services::get_index_of_service()*, que usamos en el método anterior.

```
 /**
        *
        * @param int $service un identificador de servicio (0, 1, 2, .. representado a SSH, HTTP, ...)
        * @return array Una lista de los índices que corresponden a los valores de lo que quiero leer del nagios
        * Consiste en crear una lista de índices (performance_data, ....)
        * Esos índices serán los datos de lo que quiero obtener de todo lo que nagios me da
        */
      public static function get_index_of_service( $service ) {
            $index = [];
            switch ( $service ) {
                  case Monitorizacion_Constants::SERVICIO_SSH || "SSH":
                        $index = self::getIndexSsh();
```

```
                    break;
                case Monitorizacion_Constants::SERVICIO_HTTP || "HTTP" :
                    $index = self::getIndexHttp();
                    break;
                case Monitorizacion_Constants::SERVICIO_MYSQL || "MYSQL":
                    $index = self::getIndexMysql();
                    break;
                case Monitorizacion_Constants::SERVICIO_DISCOS || "DISK":
                    $index = self::getIndexDisk();
                    break;
                case Monitorizacion_Constants::SERVICIO_CPU || "CPU":
                    $index = self::getIndexCpu();
                    break;
                case Monitorizacion_Constants::SERVICIO_PING || "PING":
                    $index = self::getIndexPing();
                    break;
                case Monitorizacion_Constants::SERVICIO_PROCESOS || "PROCESS":
                    $index = self::getIndexProcesos();
                    break;
                default:
                    return null;
            }
            return $index;

    }
```

Y los servicios por ejemplo de ssh

```
/** SSH - PLUGIN_OUTPUT Y PERFORMANCE DATA */

    /**
     * * @return array el listado de índice
     * A continuación El listado del array que retorna el nagios
     * Se trata de seleccionar qué indices nos interesan
       ["SSH"] =>    object(stdClass)#98 (54) {
       ["host_name"] =>      string(12) "5c43117b78e8"
       ["service_description"] =>      string(3) "SSH"

       …Listado de todos los índices …
Ver tabla de análisis
```

INDICES DE LOS SERVICIOS DEVUELTOS POR UN SERVIDOR NAGIOS DE UN SERVICIO

```
… .
. . .
       ["is_flapping"] =>      string(1) "0"
       ["percent_state_change"] =>      string(4) "0.00"
       ["scheduled_downtime_depth"] =>      string(1) "0"
       }
       }
     */
    public static function getIndexSsh() {
            $index = [
                "plugin_output",
                "performance_data"
            ];

            return $index;
    }
```

Simplemente habría que añadir servicios al método y todo funcionaría correctamente, ya que los datos los tenemos del servidor *nagios*, y los índices se incorporarían al documento **data** que se construye con los valores que devuelve este método.

Es importante que el nombre del índice sea el valor devuelto por el servidor *nagios*, ya que es así, como luego accederemos a ese valor en el mapeo de los datos de JSON.

Así, solo nos queda obtener el valor del parámetro que deseamos: De todo el valor que retorne el servidor *nagios* (trama de dato), puede ser que nos interese solo una parte, por ello hemos implementado un método para *parsear* cada valor:

```
public static function get_value( $index_service, $service_value ) {
        switch ( $index_service ) {
                case "plugin_output":
                        $value = self::get_value_of_plugin_output( $service_value );
                        break;
                case "performance_data":
                        $value = self::get_value_of_performance_data( $service_value );
                        break;
        }
        return $value;
    }
```

Esto es para cada índice cuyo valor queremos recoger.

En el caso de performance_data:

```
    /**
     *
     * @param string $value el valor de la propiedad performance_data de un servicio
     * @return float  0 si no se encuentra el valor  el tiempo que tarda en ejecutarsi sí si que lo encuentra
     * trata de ser un booleano
     */
    public static function get_value_of_performance_data( $value ) {
    // Formato de la trama que recibimos de nagios:
    //"time=0.013585s;;;0.000000;10.000000"; si ok nos interesa el valor 0.13585
            //""   si no ok.

            $value = trim( $value );
            $value = str_replace( [ "time=", "s" ], "", $value );
            $value = explode( ";", $value );

            $value = ($value[ 0 ]);
            if ( is_numeric( $value ) ) {
                    return (float) $value;
            } else return null;

    }
```

3.5 Obtención de informes (RF 16, RF17, RF18)

La implementación de creación de informes es una parte muy interesante igualmente del código. Ya se han expuesto en el análisis los métodos concretos. Vemos por encima el de crear gráficas y datos tabulados de las 3 últimas horas, siendo el resto de informes similares.

Este método tiene su dificultad en tener cuidado con las horas seleccionadas, según la hora en la que nos encontremos:

Hora	Datos a recoger para generar matriz
Hora 0	Datos del día anterior en horas 21 – 22 22 – 23 23 - 24
Hora 1	Datos del día anterior 22 – 23 23 – 0 y del día actual 0 - 1
Hora 2	Datos del día anterior 23 – 0 y del día actual 0 - 1 1 – 2
Resto de horas	Documento de datos del día actual

El método recoge datos del día actual y el anterior y en función de la hora coge los que necesitamos

```
/**
* $datas array con los datos del día actual y anterio de la colección data
* $list_services array con los servicios de los que quiero generar la matriz
* $hora actual  un valor entre 0 y 23.
*/
public static function create_matrices_last( $datas, $list_services, $hour ) {
            //Inicializo el primer nivel del array
            $matrix[ 'host' ] = array_values( $datas )[ 0 ]->host_id;
            $matrix[ 'date' ] = []; //Será un array indexado de las diferentes fechas
            $matrix[ 'services' ] = []; //Array asociativo con cada servicio cada uno tendrá
                  un arrau con los índices
                                          //Ahora recorremos para cada documento  (host - día)
                                           //primero inicializamos las fechas
            $min = "00";

            //Nos quedamos con los dos array si los ubiera si no data2 tendrá un false
            reset( $datas );
            $data1 = current( $datas ); //Datos del día anterior
            $data2 = next( $datas );    //Datos del día actual

      //Inicializamos la matriz data
            $fecha = substr( $data1->timestamp, 0, 10 ) . " $hour:00:00";
            $dateMAX = date( 'Y-m-d H:i:00', strtotime( $fecha . " -3 hour" ) );
            $fecha = date( 'Y-m-d H:i:00', strtotime( $fecha . " -2 minute" ) );
            $matrix[ 'date' ] = Charts_Controller::get_array_datetime_range( $dateMAX, $fecha, 60, 'H:i:00' );

            foreach ( $list_services as $service ) {
        //Inicializo este índice para la matriz
                  $matrix[ 'services' ][ $service ] = [];
```

```
        //Obtenemos los índices
                $indexes_service = array_keys( $data1->data[ $service ] );

                //Para cada índice de ese servicio
                $datas_of_mongo = [];
                foreach ( $indexes_service as $index_service ) {
                        //Si hora =1 o 2 necesito datos de los dos días para generar la matriz
                        //Si no lo cogeré de cada día

            switch ( $hour ) { //Ahora en función de la hora inicializo recojo los datos que me interesan de uno u otro array de datos (día actual o anterior) y de la hora indicada.

                                case 0:
                                        $h = 21;
                                        //Primer array de documentos
                                        $datas_of_mongo = $data1->data[ $service ][ $index_service ][ $h ];
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data1->data[ $service ][ $index_service ][ ++$h ] );
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data1->data[ $service ][ $index_service ][ ++$h ] );
                                        break;
                                case 1:
                                        $h = 22;
                                        //Primer array de documentos
                                        $datas_of_mongo = $data1->data[ $service ][ $index_service ][ $h ];
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data1->data[ $service ][ $index_service ][ $h + 1 ] );
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data2->data[ $service ][ $index_service ][ 0 ] );
                                        break;

                                case 2:
                                        $h = 23;
                                        //Primer array de documentos
                                        $datas_of_mongo = $data1->data[ $service ][ $index_service ][ $h ];
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data2->data[ $service ][ $index_service ][ 0 ] );
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data2->data[ $service ][ $index_service ][ 1 ] );
                                        break;
                                default:
                                        //Primer array de documentos
                                        $h = $hour - 2;
                                        $datas_of_mongo = $data1->data[ $service ][ $index_service ][ $h ];
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data1->data[ $service ][ $index_service ][ ++$h ] );
                                        $datas_of_mongo = array_merge( $datas_of_mongo, $data1->data[ $service ][ $index_service ][ ++$h ] );
                        }
                        $matrix[ 'services' ][ $service ][ $index_service ] = $datas_of_mongo;
                }//End while index (indices de los servicios
                //              echo "FECHA       -       VALOR SERVICIO<br/>";
        }//End foreach $services

        return ($matrix);

    }
```

En otros casos, obtenemos datos promedio cada media hora o cada hora entre una fecha inicial y una final, para mostrar datos de forma más resumida:

```
    //En este caso queremos sacar un valor por día que sea la media de de ese día
public static function create_matrices_days( $data, $interval, $list_services ) {
        $matrix = [];

        //Inicializo el primer nivel del array
        $matrix[ 'host' ] = array_values( $data )[ 0 ]->host_id;
        $matrix[ 'date' ] = []; //Será un array indexado de las diferentes fechas
        $matrix[ 'services' ] = []; //Array asociativo con cada servicio cada uno tendrá un array con los índices

        foreach ( $data as $data_host_day ) { //cada día registrado a un host concreto

                $s = substr( $data_host_day->timestamp, 0, 10 );
                $f_d = date( 'd-m-Y', strtotime( $s ) );
                $matrix[ 'date' ][] = $f_d;


                foreach ( $list_services as $service ) {
                    $indexes_service = array_keys( $data_host_day->data[ $service ] );
                      foreach ( $indexes_service as $index_service ) {
//              //primer inicializo la matriz
```

```
//                //me interesan solo los múltiplos de 30
//                 //Calculamos la media de cada hora
                                for ( $h = 0; $h <= 23; $h++ ) {
                                        $data_min_avg[] = Monitorizacion_Controller_Services::array_avg( $data_host_day->data[ $service ][ $index_service ][
$h ] );
                                }//end while $hour
                                        $matrix[ 'services' ][ $service ][ $index_service ][] = Monitorizacion_Controller_Services::array_avg( $data_min_avg );

                        }//End foreach index (indices de los servicios
                    }//End foreach $services
                }//end foreach $data
                return $matrix;
        }
```

3.6 Pruebas de test (RF 19)

La realización de test en entornos controlados es indispensable tras la incorporación o modificación de nuevas funcionalidades. De esta manera se han realizado test de pruebas de los métodos desarrollados. Así, se puede comprobar que las API devuelven datos y que además están funcionando como deben. En nuestro caso, se han realizado múltiples métodos de test para comprobar el correcto funcionamiento de las diferentes funciones desarrolladas. Por ejemplo, pruebas de test de que los método retornan los servicios existentes y los índices de cada servicio de forma correcta según lo esperado. Además verificamos que la matriz generada para crear el gráfico corresponde con los valores que tenemos en una colección. Para ello hemos creado un documento con valores concretos para verificar que lee concretamente esos valores.

Es necesario correr este tipo de test mientras se está desarrollando las distintas funcionalidades del PFC ya que es la mejor manera de comprobar que las diversas funcionalidades desarrolladas, lo hacen de la manera esperada. Estos test unitarios se pueden ejecutar cada vez que se modifica un fichero, y sobre todo, cada vez que se vaya a realizar un despliegue. Para la realización de dichos test unitarios, se ha hecho uso de la herramienta ***PHPUnit*** como ya hemos comentado en el análisis, que se trata de un entorno para realizar pruebas unitarias en el lenguaje de programación PHP. La idea no es otra que la detección de errores en el código para de esta manera poder solventarlos. *PHPUnit* utiliza assertions[4] para verificar que el comportamiento de una unidad de código es el esperado. En el anexo del código podemos ver su contenido.

Ejemplo de un test unitarios

```
    /**
        * @covers Monitorizacion_Controller::service_host_to_string
        * @group production
        * @runInSeparateProcess
        */
       public function testService_host_to_string() {
// Comparacion de fechas
                $array = array(
                        Monitorizacion_Constants::SERVICIO_SSH => 'SSH',
                        Monitorizacion_Constants::SERVICIO_HTTP => 'HTTP',
                        Monitorizacion_Constants::SERVICIO_MYSQL => 'MYSQL',
                        Monitorizacion_Constants::SERVICIO_DISCOS => 'DISK',
                        Monitorizacion_Constants::SERVICIO_CPU => 'CPU',
                        Monitorizacion_Constants::SERVICIO_PING => 'PING',
                        -1 => null
                );
                foreach ( $array as $servicio => $value_expected ) {
                        $this->assertEquals( $value_expected, Monitorizacion_Controller::service_host_to_string( $servicio ), "ERROR:
$value_expected != '' . Monitorizacion_Controller::service_host_to_string( $servicio ) );
                }
        }
  }
```

4 Assertions: sentencias usadas en entornos XUnit para comprobar el correcto funcionamiento de funciones implementadas.

3.7 Documentación y Manual de usuario (RF 20)

Todo el código está comentado como se puede ver en los método que hemos incluido en este apartado de *Desarrollo e Implementacion*, siguiendo las directivas marcadas por los técnicos de la empresa.

Además de comentar cada método como ya he explicado en el apartado de análisis y diseño, se comentan determinadas acciones dentro de los métodos con el fin de aclarar la finalidad de una variable en concreto o de una sección o bloque de código (se explica el porqué).

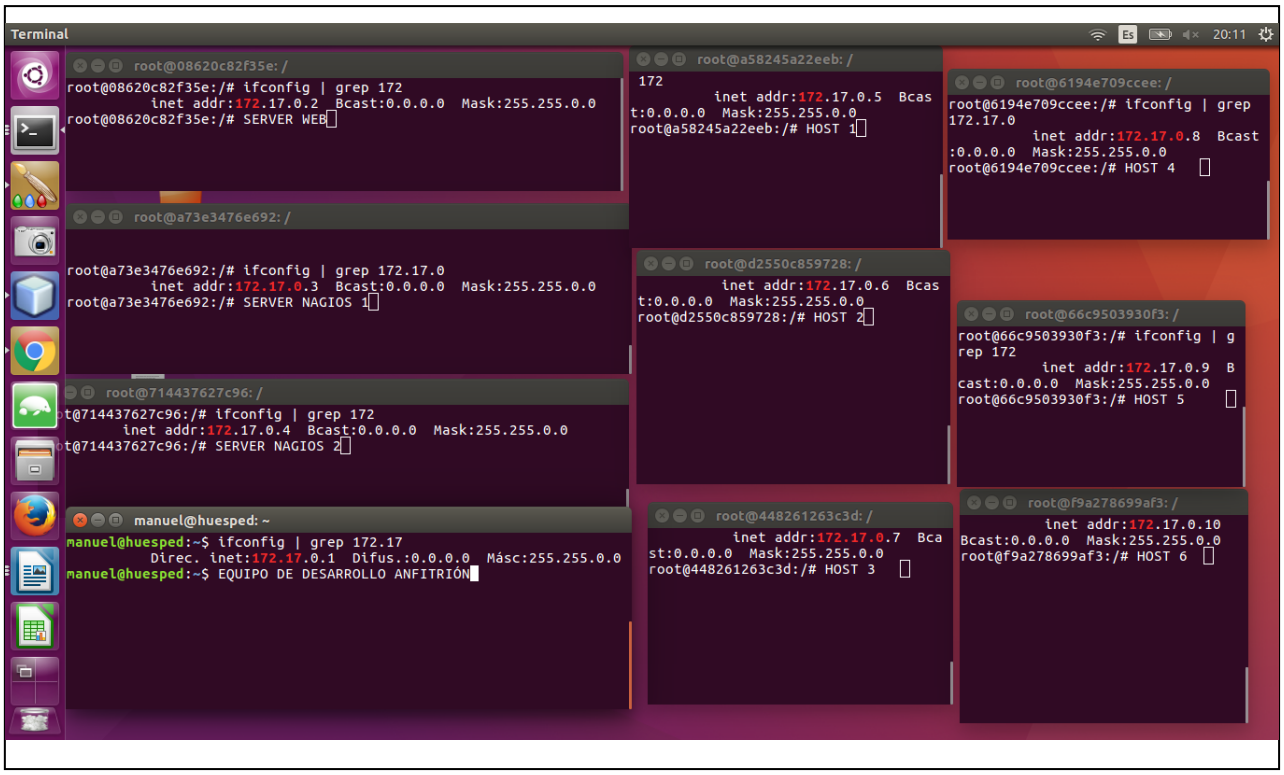
El propio documento de análisis constituye un manual de usuario que permite conocer en profundidad la aplicación desarrollada, para su uso, y que será incluido en la documentación / wiki de la empresa. Asimismo, parte de esta documentación se incluirá en la ayuda interactiva del módulo, disponible cuando los usuarios administradores accedan a las diversas pantallas del módulo.

4 RESULTADOS

Como resultado de la implementación de los requisitos analizados, podemos afirmar que se han conseguido todos los requisitos, obteniendo un software de calidad. A continuación vamos a verificar su consecución viendo la salida que nos genera la aplicación.

4.1 Entorno de trabajo (RNF 1)

Para establecer el entorno de trabajo se desarrolló un sistema *dockerizado* (crear los contenedores, configurarlo y crear un *script* para cargar dichos contenedores en el sistema). Podemos ver el resultado de ejecutar el *script*, cuando abro terminales interactivo para todos los contenedores (ver *script* en la parte de implementación).



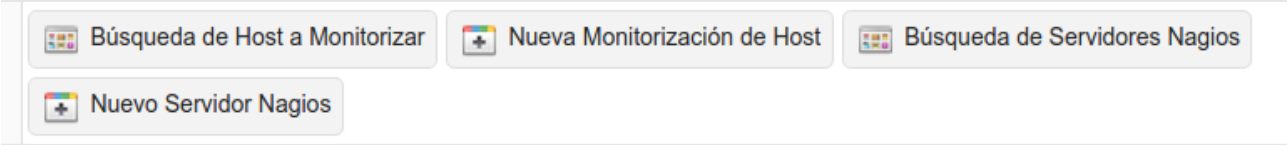
Cada *shell* es un *bash* de un contenedor, he mostrado las IP y con un texto qué contenendor representa.

4.2 Módulo, modelos lógicos y vistas (RF 2, RF 5, RF 6 y RF 7) y (RF3, RF4, RF15)

Estos apartados los vemos todos juntos ya que a nivel de ejecución están totalmente relacionados La pantalla (vista) es necesaria para visualizar información (modelo) dentro del módulo.

Ilustración 44: modulo monitorización

Al arrancar el sistema aparece el nuevo módulo. Si seleccionamos el módulo aparecen las 4 opciones disponibles (botones en formato *responsive*):



Damos de alta un nuevo host

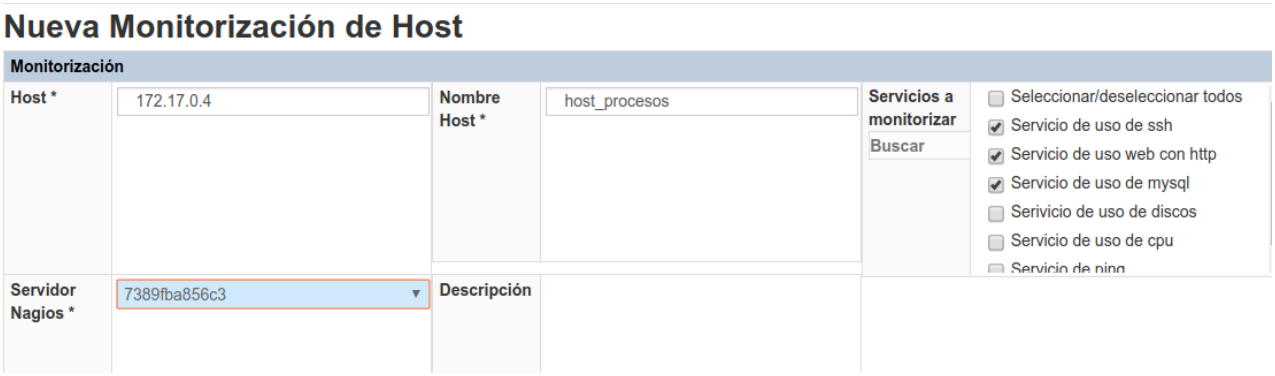


Ilustración 45: crear un nuevo host

Y aparecerá en la lista de host, todos ellos con las opciones de visualizar, editar y eliminar:

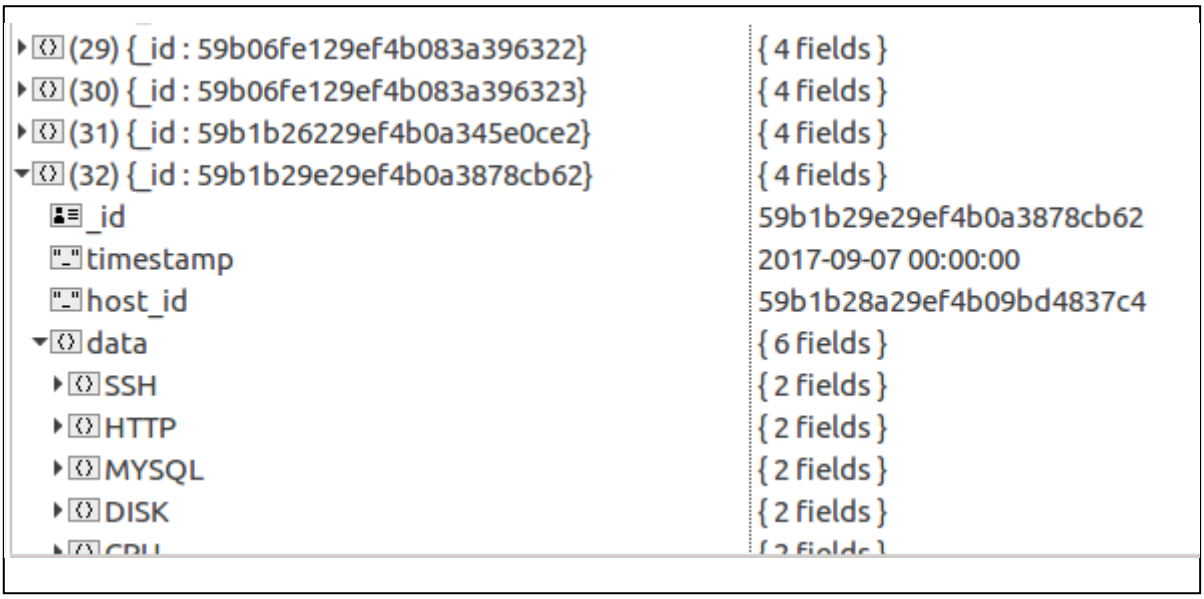
Host	Nombre Host	Nombre del Servidor Nagios	Ver	Editar	Eliminar
172.17.0.1	Central	7389fba856c3			
172.17.0.4	host_procesos	7389fba856c3			
172.17.0.6	Host base_datos	7389fba856c3			

La parte de servidores *nagios* funciona exactamente igual

4.3 Controladores

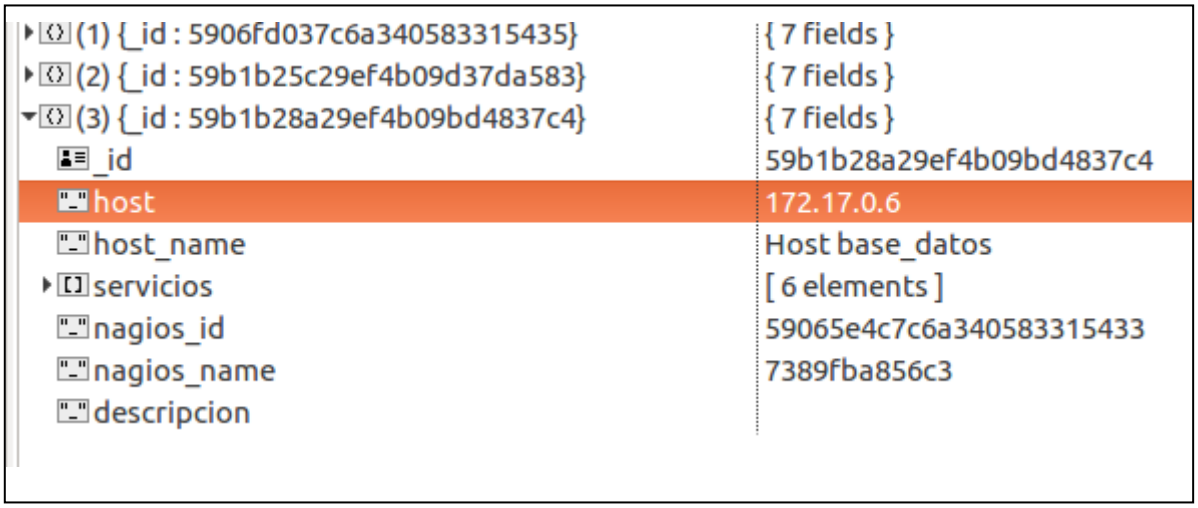
4.3.1 Recogida de datos (RF 10, RF 11, RF12, RF 13 y RF 14)

Para ver el funcionamiento de los controladores simplemente dejamos nuestra aplicación funcionando y podemos ver cómo se actualiza la colección data cada minuto. La visualizaremos con **Mongochef** comprobando que se almacenan los datos de forma correcta con la estructura deseada (formato de *time-series*)

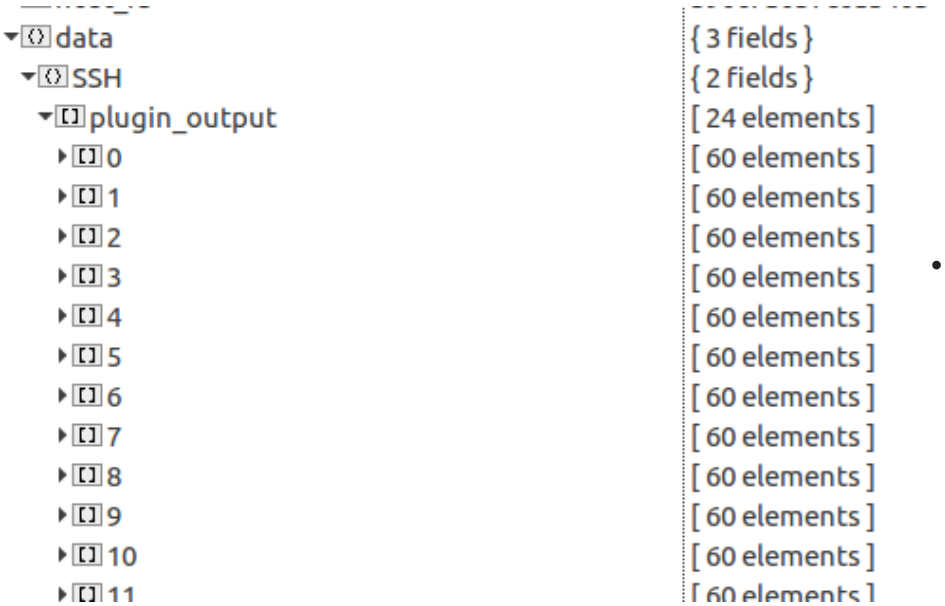


Vemos las últimas colecciones y desplegamos justo la última

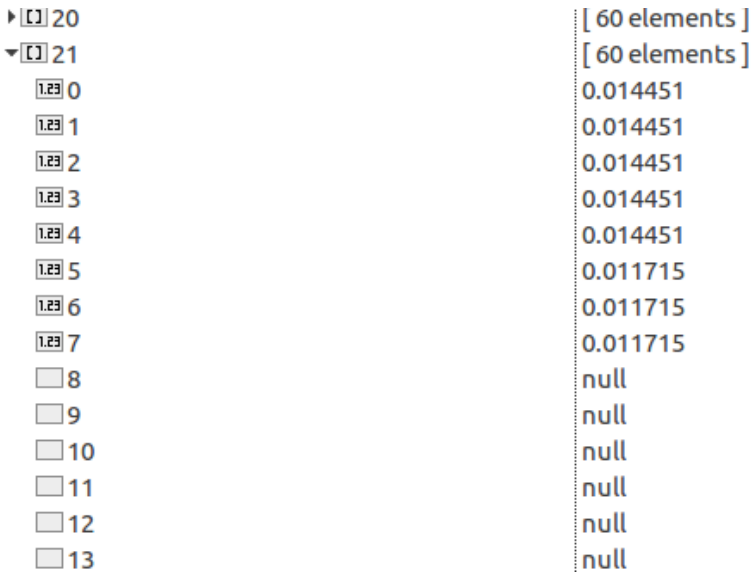
Podemos ver que es la colección del día 7/9/2017 del *host* 59b128… que si vamos a la colección del modelo de los *host* vemos que corresponde al *host* de IP 172.17.0.6:



Si desplegamos el *array* **data** vemos los servicios y cada servicio los índices asociados y cada índice las horas:



Dentro de cada hora vemos los minutos, desplegamos la hora 21 y vemos qué minutos tienen valor, hay valor hasta el minuto 7:



Esperamos 5 minutos y volvemos a cargar los valores y ahora vemos que ya hay valores hasta el minuto 12, justo 5 minutos más, vemos como cada minuto se actualiza cada colección de cada *host* del día actual

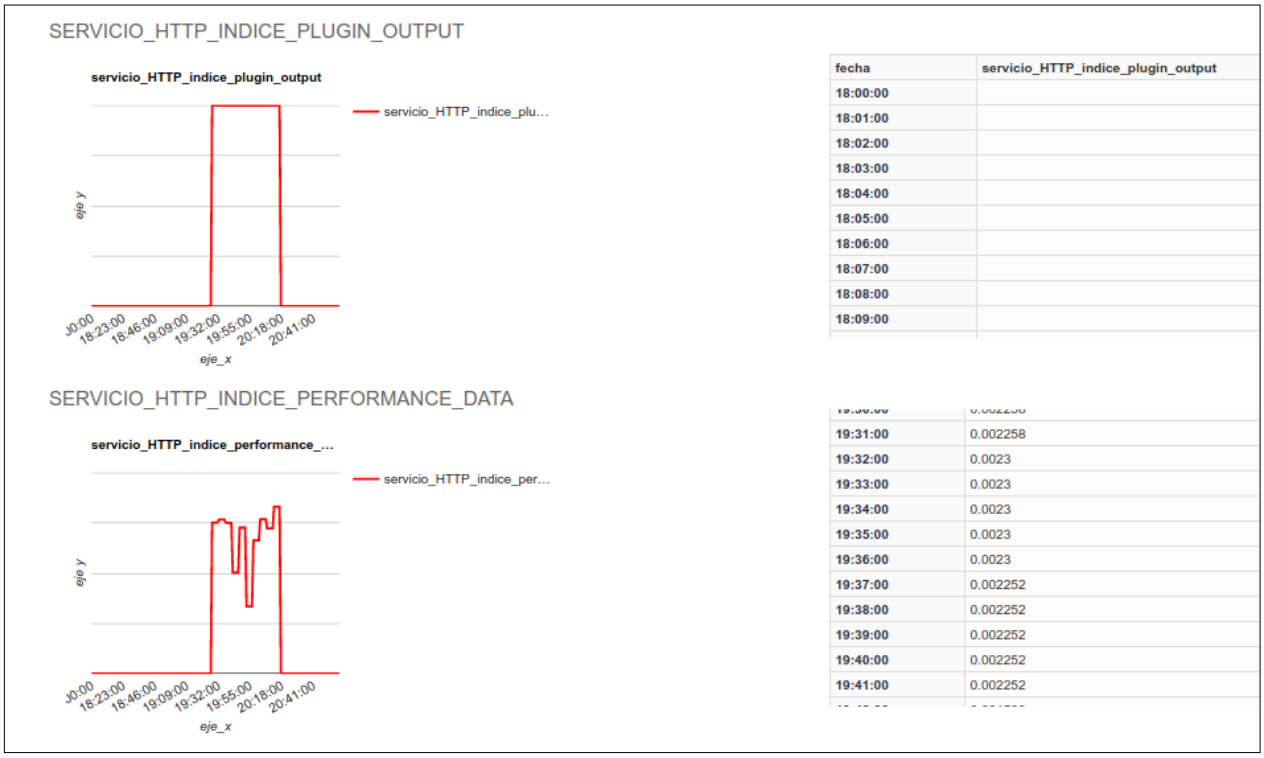
▸ [] 20	[60 elements]
▾ [] 21	[60 elements]
1.23 0	0.014451
1.23 1	0.014451
1.23 2	0.014451
1.23 3	0.014451
1.23 4	0.014451
1.23 5	0.011715
1.23 6	0.011715
1.23 7	0.011715
1.23 8	0.011715
1.23 9	0.011715
1.23 10	0.013504
1.23 11	0.013504
1.23 12	0.013504
13	null
14	null

4.3.2 **Generación de informes** (RF 16, RF17, RF18)

Respecto a las gráficas o informes, al visualizar un *host*, como estaba en los requisitos, se mostrará un menú para confeccionar los informes y gráficas. Se dejan tres opciones:

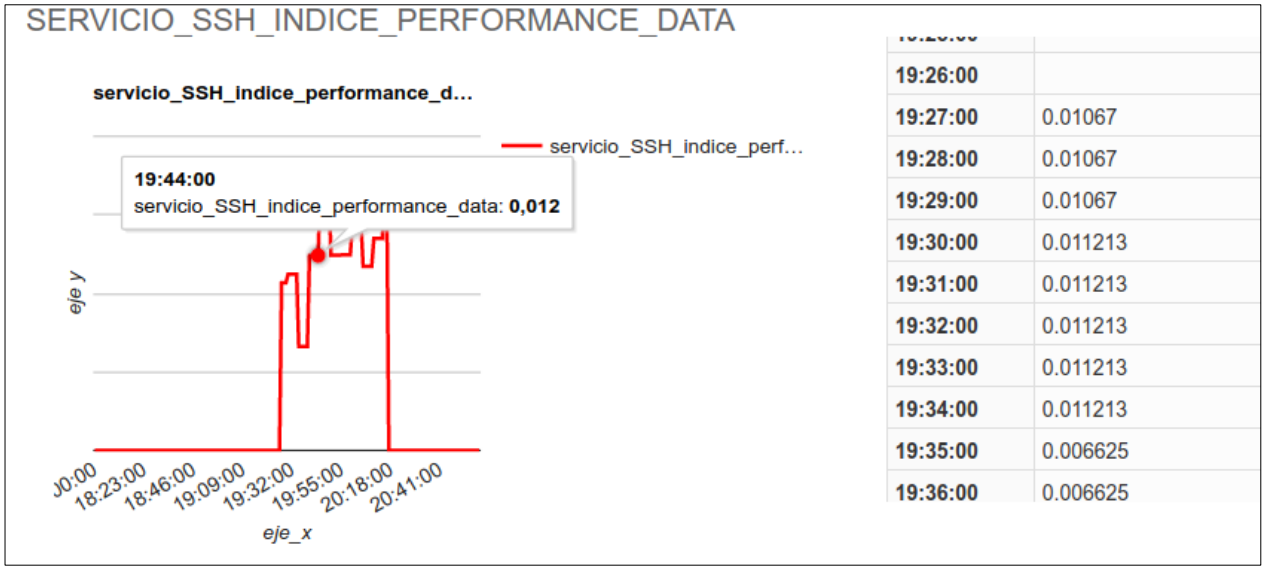
Opción 1: Datos 3 últmas horas. En este caso la fecha de días no se tienen encuenta aunque se seleccionen.

Y vemos los informes generados en formato tabulado y gráfico, mostrando en la tabla de datos la hora de cada minuto.

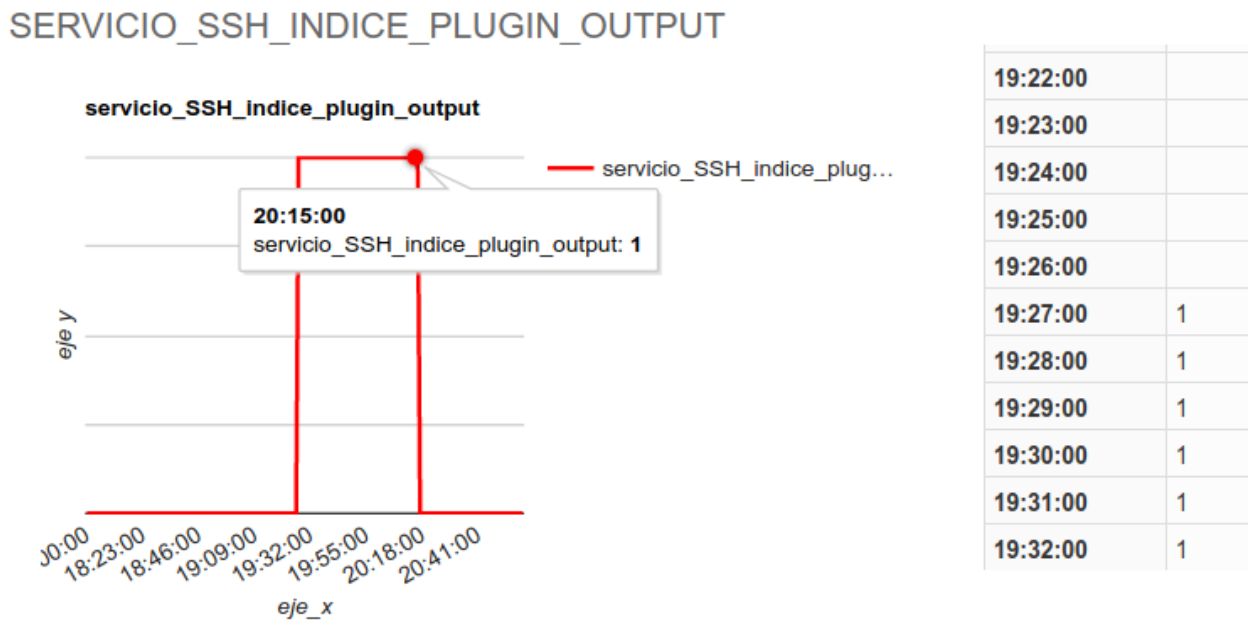


fecha	servicio_HTTP_indice_plugin_output
18:00:00	
18:01:00	
18:02:00	
18:03:00	
18:04:00	
18:05:00	
18:06:00	
18:07:00	
18:08:00	
18:09:00	

19:31:00	0.002258
19:32:00	0.0023
19:33:00	0.0023
19:34:00	0.0023
19:35:00	0.0023
19:36:00	0.0023
19:37:00	0.002252
19:38:00	0.002252
19:39:00	0.002252
19:40:00	0.002252
19:41:00	0.002252



19:26:00	
19:27:00	0.01067
19:28:00	0.01067
19:29:00	0.01067
19:30:00	0.011213
19:31:00	0.011213
19:32:00	0.011213
19:33:00	0.011213
19:34:00	0.011213
19:35:00	0.006625
19:36:00	0.006625



19:22:00	
19:23:00	
19:24:00	
19:25:00	
19:26:00	
19:27:00	1
19:28:00	1
19:29:00	1
19:30:00	1
19:31:00	1
19:32:00	1

Vemos que los informes son para cada índice de cada servicios.

Opción 2 informes de medias de días entre dos fechas

En este caso sí que es importante las fechas de inicio y fin. Por ejemplo si cogemos los informes de la última semana:

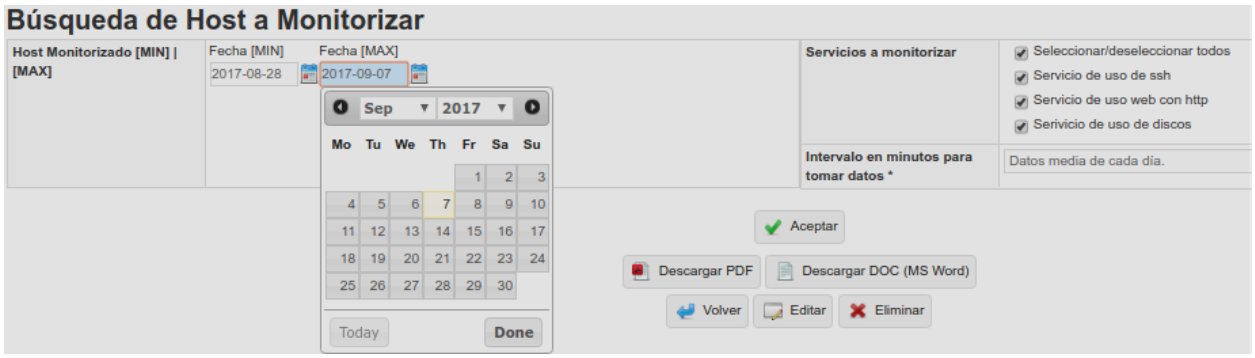
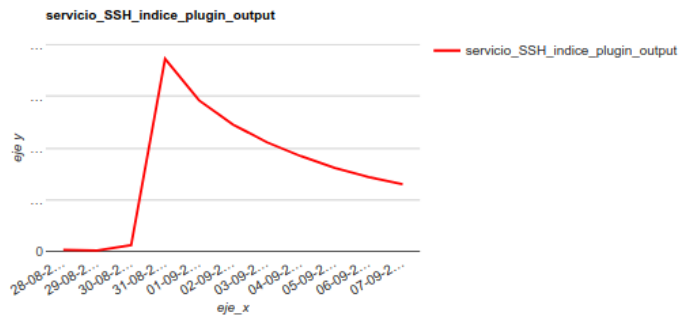


Ilustración 49: valores para informe entre dos fechas

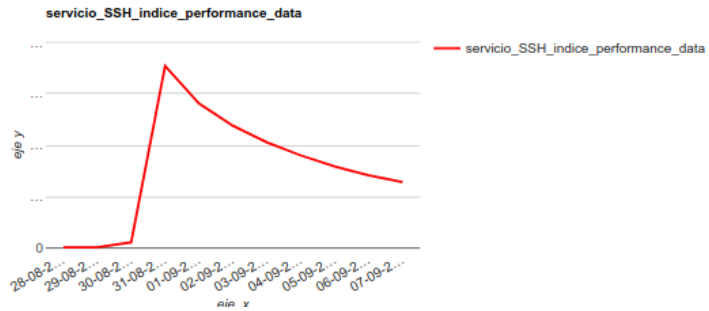
En este caso el resultado será similar, mostrando valores promedio:

SERVICIO_SSH_INDICE_PLUGIN_OUTPUT



fecha	servicio_SSH_indice_plugin_output
28-08-2017	0.010416666666667
29-08-2017	0.0067665703373016
30-08-2017	0.036791433279915
31-08-2017	1.1161827940058
01-09-2017	0.87601951447222
02-09-2017	0.73578770486111
03-09-2017	0.63336371614114
04-09-2017	0.55269012228682
05-09-2017	0.48501378078231
06-09-2017	0.43210318651515

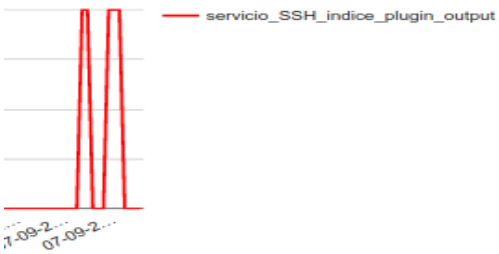
SERVICIO_SSH_INDICE_PERFORMANCE_DATA



fecha	servicio_SSH_indice_performance_data
28-08-2017	0.0052692743055556
29-08-2017	0.0059596546006944
30-08-2017	0.034504580555556
31-08-2017	1.0605020943403
01-09-2017	0.84247488955662
02-09-2017	0.71290565067274
03-09-2017	0.61680353519737
04-09-2017	0.54012898314394
05-09-2017	0.47531350516667
06-09-2017	0.42438705818452

En la gráfica de fechas muestra sólo la fecha de día. Si miramos la colección de data, los documentos de los días 28 y 29, estarán seguramente vacíos por que no se tomarían valores. El valor que muestra es la media de los valores de un día (los 1440 valores cada índice).

La gráfica de cada 30 minutos es exactamente igual que las anteriores, vemos el resultado, observando cómo muestra en la columna fecha también los minutos y horas



06-09-2017 19:00:00	
06-09-2017 19:30:00	
06-09-2017 20:00:00	0
06-09-2017 20:30:00	0
06-09-2017 21:00:00	0
06-09-2017 21:30:00	0
06-09-2017 22:00:00	0
06-09-2017 22:30:00	0
06-09-2017 23:00:00	0
06-09-2017 23:30:00	
07-09-2017 00:00:00	0

:MANCE_DATA

07-09-2017 18:00:00	0.01512
07-09-2017 18:30:00	0.006524
07-09-2017 19:00:00	
07-09-2017 19:30:00	
07-09-2017 20:00:00	
07-09-2017 20:30:00	0.011213
07-09-2017 21:00:00	0.014451
07-09-2017 21:30:00	0.007967
07-09-2017 22:00:00	
07-09-2017 22:30:00	
07-09-2017 23:00:00	
07-09-2017 23:30:00	

4.4 Test y pruebas

De los diferentes test vamos a mostrar dos de ellos.

El código de los test sería similar a los siguientes mostrados (test para uno de los métodos):

```
 /**
        * @covers Monitorizacion_Controller_Servoces::get_value_of_performance_data
        * @group production
        * @runInSeparateProcess
        */
       public function testGet_value_of_performance_data() {

            $array = array(
                array( 'expected' => 0.013585, 'input' => "time=0.013585s;;;0.000000;10.000000", 'type' => 'float' ),
                array( 'expected' => null, 'input' => "time=;;;0.000000;10.000000", 'type' => 'null' ),
            );
            foreach ( $array as $index => $data ) {
//          echo PHP_EOL;
                $value_returned = Monitorizacion_Controller_Services::get_value_of_performance_data( $data[ 'input' ] );

                $this->assertEquals( $data[ 'expected' ],
                   $value_returned,
                   "ERROR: " . $data[ 'expected' ] . " != " . $value_returned, 0.001 );
                $this->assertInternalType( $data[ 'type' ], $value_returned );
            }

       }
       /**
        * @covers Monitorizacion_Controller::service_host_to_string
        * @group production
        * @runInSeparateProcess
        */
       public function testService_host_to_string() {
// Comparacion de fechas
            $array = array(
                Monitorizacion_Constants::SERVICIO_SSH => 'SSH',
                Monitorizacion_Constants::SERVICIO_HTTP => 'HTTP',
                Monitorizacion_Constants::SERVICIO_MYSQL => 'MYSQL',
                Monitorizacion_Constants::SERVICIO_DISCOS => 'DISK',
                Monitorizacion_Constants::SERVICIO_CPU => 'CPU',
                Monitorizacion_Constants::SERVICIO_PING => 'PING',
                -1 => null
            );
            foreach ( $array as $servicio => $value_expected ) {
                $this->assertEquals( $value_expected,
                   Monitorizacion_Controller::service_host_to_string( $servicio ),
                   "ERROR: $value_expected != " . Monitorizacion_Controller::service_host_to_string( $servicio ) );
            }

       }
```

Al pasar estos dos test vemos como los pasan correctamente.

```
root@08620c82f35e:/var/www/smartroads-core/test# phpunit modules/monitorizacion/
monitorizacion_controllerTest.php
PHPUnit 4.8.9 by Sebastian Bergmann and contributors.

..

Time: 395 ms, Memory: 12.25Mb

OK (2 tests, 11 assertions)
root@08620c82f35e:/var/www/smartroads-core/test#
```

4.5 Integración

El proyecto se ha ido guardando en el control de versiones *git* en un *branch* creado de forma específica para no tener ningún tipo de efecto lateral con cualquier otro desarrollo que se estuviera realizando en la empresa, ni que yo pudiera modificar ningún código de otros desarrolladores.

Aunque no he tenido que modificar ninguna parte fuera de mi módulo, es cierto que durante el desarrollo sí que he necesitado muchas veces visualizar valores de variables que estaban fuera de mi módulo, por lo que esporádicamente he tocado analizado clases y métodos del núcleo y otros módulos del *framework* de Iternova. Siempre lo he realizado siguiendo un mecanismo que me permitía dejarlo todo como estaba.

El *merge* del módulo con la aplicación de Iternova se realizará bajo la supervisión de los técnicos de la empresa, previa ejecución de todos los test codificados.

Una vez añadido el módulo en el *branch master*, los técnicos de Iternova se encargarán de realizar la integración y revisión del módulo (creando test de integración fuera del alcance de este PFC), ya que se tiene en mente poder utilizar este módulo de monitorización en el sistema de producción que tiene Iternova implantado en la Demarcación de Carreteras del Estado en Aragón y también en un proyecto piloto que llevan en marcha en México

5 CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Conclusiones generales

Los objetivos y requisitos marcados desde el principio se han conseguido en el producto final. Como se puede ver en la memoria, han sido los que han enfocado su desarrollo (el de la memoria); Los técnicos de la empresa tenían muy claros los objetivos que se pretendía con este proyecto desde el principio. Ese hecho ha sido un punto de partida que ha permitido realizar una inmersión en el desarrollo del mismo desde el principio permitiendo mucha proactividad.

En la realización del proyecto el apoyo técnico de estas personas ha sido fundamental, pues evitaban puntos muertos en el desarrollo o situaciones de paradas para decidir por dónde seguir con el proyecto. Estas situaciones no se han producido en ningún momento.

Trabajar en un entorno real (como he comentado trabajaba sobre un *branch* creado del proyecto de la empresa), hace que en muchas ocasiones no haya que reinventar la rueda sino que muchas necesidades software ya están planteadas.

Ver cómo cada requisito planteado, tiene un estudio de análisis y una solución para su implementación, con la misma codificada, es una gran garantía de que el trabajo está bien hecho y terminado. Este es un importante punto en la calidad del producto software, que por su naturaleza lógica en muchas ocasiones es difícil de establecer ¿Cuánto de calidad tiene un software? Una buena medida para medir la calidad de un producto de naturaleza lógica, que más que una característica lo debemos de ver como una necesidad, es ver el cumplimiento funcional de cada requisito. Desarrollar software no es un arte, es un industria dónde el producto ha de adecuarse a las necesidades, respondiendo de forma correcta a cada uno de los requisitos. Esto es lo que se ha pretendido (y bajo mi punto de vista se ha conseguido) con el desarrollo de este proyecto: obtener un producto de calidad que cumple cada uno de los requisitos marcados.

5.2 Conclusiones personales

La verdad que no podría estar más contento en la terminación de este proyecto. La manera en la que he sido atendido en todo momento, con total flexibilidad (mis circunstancias personales y de familia, no me permitían tener una disponibilidad clara ni total para este proyecto), y el acercamiento han sido una tónica constante en el tiempo en el que estado con el desarrollo de este proyecto. Cuando se trabaja con personas que tienen una solución técnica a cualquier problema que pueda surgir y que a nivel humano tienen una sensibilidad que te hacen sentir a gusto, genera un ecosistema ideal para poder trabajar.

A nivel técnico he aprendido a usar *git*, he aprendido como gestionar entorno de desarrollo con *docker*. He mejorado notablemente con el uso del IDE *Netbeans* que venía usando en mi trabajo antes de empezar el proyecto. La forma que ahora tengo de trabajar con PHP, un lenguaje que ya conocía, ha mejorado notablemente. El uso de *MongoDB*, y las *times-series* para la captura y gestión de datos en intervalos corto de tiempo, son conceptos que si bien conocía, nunca había utilizado en los desarrollos; tras este proyecto puedo asegurar que aunque tenga mucho que aprender, sé trabajar con ello.

Asimismo he conocido una gran cantidad de librerías disponibles para realizar tareas en el desarrollo de aplicaciones web, como *html_purifier* para gestionar la seguridad y filtrado de los datos de entrada en un sitio web o el gestor de colas *gearman* que evita los típicos problemas que a veces plantea el servidor apache cuando está saturado por la demanda de solicitudes.

Lo curioso del *framework* que tienen en Iternova que al principio parece extraño y al final te habitúas a él viéndolo como casi como si fuera un entorno de desarrollo totalmente lógico. El uso de test con *PHPUnit* ha sido algo totalmente nuevo y muy interesante que he aprendido aquí.

Trabajar con tanta cantidad de información a sido uno de los mayores retos en este desarrollo, por que me ha costado conceptualizarlo para poderlo manejar con soltura en el desarrollo.

Es cierto que al final todo son cosas buenas que me llevo a nivel técnico, pero no por ello querría dejar de mencionar el aspecto humano que, como ya he comentado, ha sido inmejorable.

Al principio cuando empecé el proyecto en esta empresa, pensaba que iba a hacer un típico programa web de gestión de datos. No parecía nada novedoso. Pero enseguida fui consciente de mi error. Son tantas las cosas que se puede aprender que cada día hay descubrimientos que hacen que nunca se pierda la chispa en el desarrollo. La ingeniería es una ciencia, que si la estudias es por que tienes inquietud de aprender, enfrentarte a retos que resolver y solucionar. Puedo asegurar de he tenido que aplicar principios de ingeniería en el desarrollo de este proyecto, en el que si bien los objetivos, como he comentado, eran claros, la forma de resolverlos si que me ha supuesto en muchas ocasiones retos muy interesantes. En varios de ellos, agradecí mucho la ayuda de los técnicos de **Iternova.**

5.3 Líneas futuras y mejoras

El Sistema de Gestión Web de Carreteras (*SmartRoads*) está en continua evolución, ofreciendo mejoras que hacen cada vez más eficaz el tratamiento de la información para la gestión de las infraestructuras viarias. Muchas de estas mejoras se determinarán tras un uso prolongado del sistema según las necesidades que surjan por parte de los propios clientes.

Como se ha indicado también, junto con los técnicos de Iternova se concretaron los requisitos y objetivos al comienzo del proyecto, cumpliéndose todos ellos. Aun así, los sistemas siempre se pueden ir mejorando de forma iterativa.

Durante el desarrollo han sido muchas las propuestas que he realizado para el desarrollo de la aplicación. La respuesta de los técnicos, que por supuesto siempre me escuchaba y debatíamos los temas de forma muy proactiva, era que los requisitos estaban muy claros desde el principio y lo que se pretendía era lo establecido, si bien esa idea podría se considerada para futuras versiones o mejoras del módulo, ya que quedan fuera del alcance del proyecto. Algunas de estas mejoras, que hemos ido anotando como tareas de desarrollo para el futuro:

- Notificaciones vía email cuando ocurran alguna situación especial recogidas en las gráficas
- Personalización mayor a la hora de generar los informes (por ejemplo, comparando datos de múltiples hosts)
- Poder configurar tanto los servidores *nagios* como los *host* desde la aplicación vía servicios SSH o JSON/Rest
- Visualización geolocalizada sobre mapa (*Google Maps)* de estado de los *hosts* (siguiendo código de colores de estado en los marcadores) para detectar visualmente el estado de la red de dispositivos.
- Disparos (*triggers*) de eventos, utilizando el sistema de suscripción a eventos disponible en el *framework SmartRoads* para que otros módulos del sistema puedan ejecutar tareas cuando suceda algún estado no deseado en alguno de los dispositivos. Por ejemplo, marcar alerta naranja en una videocámara que tenga mala conectividad (ahora mismo el módulo únicamente muestra color verde si la cámara envía imágenes en los últimos minutos o rojo si no envió)
- Mejora en informes gráficos de los documentos *time-series* almacenados en *MongoDB*, ejecutando técnicas de *map-reduce*, para crear una nueva colección de documentos en donde se reduzcan los datos en períodos de tiempo más extensos (por ejemplo, para cada host un documento por semana, por mes y por año). De esta forma, se pueden generar en segundo plano (usando las colas *gearman* en segundo plano) de forma periódica (usando *cron*) cada semana y cada mes estos documentos resumen, utilizándose en otras gráficas genéricas (por ejemplo, gráfica de estado mensual o anual, sin necesidad de seleccionar fechas en concreto) de forma directa y muy rápida, sin tener que calcular los promedios al vuelo como se hace ahora.

6 REFERENCIAS

PHP

- **phparchitect - Zend PHP5 Certification Study Guide** => Una vez mirado este (que es la "biblia" de la programación en PHP... además de la página http://www.php.net), se puede profundizar algo más con el de patterns más comunes: **PHP Architects Guide to PHP Design Patterns**

  http://phpdevenezuela.github.io/php-the-right-way/
- Se le puede echar un ojo al sitio http://www.tuxradar.com/practicalphp/ , que tiene muchos ejemplos prácticos. Hay más ejemplos en el libro **PHP Cookbook**
- **Otras web de interés**
- https://secure.php.net/manual/es/index.php

- http://www.php.ne
- http://www.jquery.com
-
  - http://www.tuxradar.com/practicalphp/

MongoDB

- En el dropbox hay varios libros, el de MongoDB & PHP es bastante completo. Sin embargo, conviene leerse la documentación de http://www.mongodb.org primero, que está bastante bien, y aprender a hacer sentencias con JSON (porque luego en nuestro sistema tenemos nuestro propio controlador para crear las sentencias, etc... e interesa saber más qué estamos haciendo que cómo hacemos luego las llamadas al controlador de bajo nivel...)
- Para hacer pruebas, recomendado instalarse MongoDB y rockmongo (que es como un phpMyAdmin pero de MongoDB): http://code.google.com/p/rock-php/wiki/rock_mongo

SVN / GIT

- Como control de versiones usamos GIT y SVN (subversion). Interesa saber un poco la teoría y a ser posible su uso correcto, porque lo usamos internamente siempre.
- **Manuales de GIT**: https://www.atlassian.com/git/
- Solemos utilizar herramientas para gestión mediante GUI (p.e. en Windows tortoise, en Linux pagavcs / smartssvn / el interno de netbeans, etc...), pero conviene saber qué hacen por debajo (o si usamos mediante consola).
- Enlaces de interés: http://www.thegeekstuff.com/2011/04/svn-command-examples/ y http://wiki.greenstone.org/wiki/index.php/Useful_SVN_Commands

Más enlaces que pueden ser de interés

- Libros electrónicos variados (en PDF) http://it-ebooks.info/

Referencias protegidas con pass, pero muy consultadas

- En cuanto al **repositorio GIT**, la URL del proyecto es https://gitlab.iternova.net/smartroads/smartroads-core
- **WIKI** en la que hay mucha información sobre el proyecto
  - ✓ URL: https://wiki.iternova.net
  - ✓ Slack de iternova https://iternova.slack.com

jQuery

- El siguiente post es bastante explicativo (para saber por dónde empezar con javascript / jQuery): http://webdesignerwall.com/tutorials/jquery-tutorials-for-designers => Se completa con toda la documentación de jQuery en http://www.jquery.com
- Se pueden buscar libros de jQuery en http://it-ebooks.info/ (buscar jQuery por título)

MySQL

- Toda la documentación de MySQL está en http://www.mysql.com
- Hay muchos ejemplos de sentencias MySQL en http://www.artfulsoftware.com/infotree/queries.php
- Nosotros solemos utilizar como herramienta de gestión SQLYog (la community sirve) http://code.google.com/p/sqlyog/downloads/list => En Linux la usamos mediante wine, porque no nos convencen otras herramientas similares...
- Se pueden buscar libros de MySQL en http://it-ebooks.info/ (buscar MySQL por título)
- **Aprendizaje interactivo de SQL**: http://sqlzoo.net/

mongodb

- http://www.mongodb.com/presentations/webinar-time-series-data-mongodb
- http://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework
- http://www.mongodb.com/prese
- https://github.com/lozzd/nagdash
- ntations/webinar-mongodb-time-series-data-setting-stage-sensor-managemen