TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

IPManager

Infraestructura bajo demanda

Autor
Ángel Gómez Martín

Director
Juan Julián Merelo Guervós

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Julio de 2020

IPManager

Infraestructura bajo demanda

Autor
Ángel Gómez Martín

Director
Juan Julián Merelo Guervós

Granada, Julio de 2020

IPManager: Infraestructura bajo demanda

Ángel Gómez Martín

Palabras clave: servicios, infraestructura, aprovisionamiento, backend, frontend, *API REST*

Resumen

El aprovisionamiento, despliegue de servicios y almacenamiento de configuraciones de máquinas a gran escala son tareas que tradicionalmente se han desarorollado de forma independiente. Esto supone a corto y largo plazo una gran inversión de tiempo para los administradores de sistemas.

En este proyecto se ha desarrollado una solución que busca unificar todos estos procesos en un puesto centralizado compuesto por un backend y un frontend que permite agilizar el desarrollo de estas tareas. También ofrece modularidad en dos grandes aspectos: permitiendo que se puedan agregar nuevas funcionalidades de forma sencilla en el backend y proporcionando una *API REST* que permite la comunicación con otro tipo de aplicaciones.

IPManager: Infrastructure on demand

Ángel Gómez Martín

Keywords: services, infrastructure, provisioning, backend, frontend, *REST API*

Abstract

Provisioning, deployment of services and storage of large-scale machine configurations are tasks that have traditionally been carried out independently. In the short and long term, this is a major time investment for system administrators.

In this project, a solution has been developed to seek unify all these processes in a centralized position composed of a backend and a frontend that allows to speed up the course of these tasks. It also offers modularity in two major aspects: allowing new functionalities to be added easily in the backend and providing a *REST API* that allows communication with other type of applications.

Yo, **Ángel Gómez Martín**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75570479T, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Ángel Gómez Martín

D. **Juan Julián Merelo Guervós**, Profesor del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***IPManager, Infraestructura bajo demanda***, ha sido realizado bajo su supervisión por **Ángel Gómez Martín**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 7 de Julio de 2020.

El director:

Fdo: Juan Julián Merelo Guervós

Agradecimientos

A mi tutor, por la ayuda, conocimientos e ideas prestadas para la realización de este proyecto.

A mis padres y mi hermana, por todo el cariño y apoyo que me han dado durante toda esta etapa.

Índice general

Capítulo 1

Introducción

Cada día el número de equipos que se utilizan de forma profesional en empresas y organizaciones crece de forma exponencial. Todos ellos requieren una configuración inicial para empezar a trabajar, la cual puede comprender tanto la instalación de aplicaciones de forma local como el despliegue de servicios para poder trabajar con ellos, entre otros.

Esta configuración inicial se suele realizar en varias etapas, comenzando por la instalación física de los equipos, configuración de red, instalación de software y finalmente la configuración de este. Cada uno de estos pasos comprende una serie de tareas secundarias que generalmente se suelen realizar de forma independiente. La independencia de las tareas hace que la puesta a punto de todas las máquinas tome un tiempo muy valioso. Un ejemplo sería la instalación de un mismo software en cientos de máquinas, en la que hacerlo de manera secuencial tomaría mucho tiempo. Para solucionar esto han surgido herramientas que permiten realizar estas instalaciones en un solo paso, algo que soluciona en parte el problema.

Por otro lado el despliegue de los servicios que pueden necesitarse también suele tomar un tiempo considerable, algo que aunque está ligado al paso anterior, no se realiza al mismo tiempo.

Además todas las configuraciones de las máquinas que se manejan no suelen estar centralizadas. Cierto es que los routers o switches actuales tienen capacidades para almacenar todos esos datos, pero implica que se tiene que acceder a ellos para consultarlos.

En este proyecto se desarrolla una solución al problema expuesto, **IP-Manager**. Esta se compone de un backend y un frontend y busca unificar todos los procesos descritos anteriormente en un puesto centralizado que permite a un administrador de sistemas agilizar el desarrollo de estas tareas. También ofrece modularidad en dos grandes aspectos: permitiendo que se puedan agregar nuevas funcionalidades de forma sencilla en el backend y

proporcionando una *API REST*.

Con la reciente crisis provocada por el COVID-19 muchas empresas han tenido que dejar de lado sus oficinas para operar mediante teletrabajo. Herramientas como **IPManager** permiten que el administrador del sistema pueda administrarlo todo de forma remota sin tener que desplazarse de su hogar. Todas las facilidades que se puedan brindar al trabajo remoto son necesarias, por lo que ofrecer soluciones que permitan esto tiene una especial importancia.

1.1. Objetivos

El desarrollo de este proyecto tiene los siguientes objetivos:

- Ofrecer una solución que unifique todos estos procesos, para simplificarlos y para reducir el tiempo empleado en ellos. Estos son el aprovisionado de sistemas, el despliegue de servicios y el almacenado de configuraciones de máquinas.
- Ofrecer una solución liviana y no intrusiva en el ecosistema donde se instale.
- Ofrecer una solución que provea una de *API REST* para permitir el desarrollo de otros frontend y la integración de estos.
- Ofrecer una solución totalmente compuesta por software libre.

1.2. Estructura del documento

En este documento se explica todo el desarrollo del proyecto, así como las decisiones tomadas y los diferentes aspectos que se han tenido en cuenta.

Se comienza con una descripción más concreta del problema y con el análisis de requisitos en el *capítulo 2*. Tras esto, en el *capítulo 3*, se hace un repaso al estado del arte en este ámbito.

El capítulo 4 se ha dedicado a la exposición de las herramientas y tecnologías que se han empleado, y en el *capítulo 5* y *capítulo 6* se desarrolla en profundidad la solución que se propone y la implementación de la misma.

Finalmente, en el *capítulo 7*, se exponen algunas conclusiones personales tras desarrollar este proyecto y también se enumeran los posibles trabajos futuros que se podrían incluir en este proyecto.

De forma anexa se ha incluido también la especificación de los endpoints del backend y de las variables de entorno, una guía de instalación del proyecto y una introducción a los primeros pasos a dar tras instalar el software.

Capítulo 2

Análisis

En este capítulo se hace un análisis del sistema, describiendo los diferentes actores y los requisitos de este. Finalmente se expone el modelo de negocio y el presupuesto del primer año de vida del proyecto.

2.1. Descripción de los actores

En este sistema hay dos actores:

Usuario regular. Este actor puede realizar todas las acciones disponibles en el frontend y en el backend. Aun teniendo conocimientos en la administración de sistemas, no se le permite realizar acciones de este tipo sobre los clientes y usuarios del sistema.

Administrador. Este actor puede realizar todas las acciones que se le permiten a un usuario regular y además puede realizar tareas de administración en los clientes y usuarios del sistema.

2.2. Requisitos del sistema

2.2.1. Requisitos funcionales

- **R.F. 1** Se distinguirán los clientes por medio de un subdominio en la *URL*.
- **R.F. 2** El sistema tendrá un sistema de autenticación.
- **R.F. 3** La visualización de datos en el frontend deberá ser en forma de listados.

- **R.F. 4** El backend proveerá una *API* capaz de funcionar sin la necesidad de un frontend.
- **R.F. 5** El sistema permitirá aprovisionar máquinas.
- **R.F. 6** En el sistema podrá haber distintos clientes.
- **R.F. 7** El sistema permitirá la administración de clientes.
- **R.F. 8** En cada cliente podrá haber diferentes usuarios.
- **R.F. 9** El sistema permitirá la administración de usuarios.
- **R.F. 10** Los usuarios podrán ser de tipo administrador o usuario común.
- **R.F. 11** Un usuario administrador podrá crear usuarios comunes.

2.2.2. Requisitos no funcionales

- **R.N.F. 1** La autenticación del usuario será mediante *JWT*.
- **R.N.F. 2** El puesto centralizado estará compuesto por un backend y un frontend.
- **R.N.F. 3** El despliegue de servicios se hará mediante contenedores Docker.
- **R.N.F. 4** El frontend tendrá una interfaz sencilla.
- **R.N.F. 5** El sistema funcionará para sistemas basados en *GNU Linux*.
- **R.N.F. 6** El sistema deberá ser escalable.
- **R.N.F. 7** La interfaz de usuario del sistema será mediante una aplicación web.

2.2.3. Requisitos de información

- **R.I. 1** Se almacenarán las diferentes configuraciones de aprovisionamiento.
- **R.I. 2** El sistema almacenará los detalles de los sistemas que aprovisiona.
- **R.I. 3** El sistema almacenará para cada cliente un identificador único, un dominio y el nombre de la base de datos correspondiente a ese cliente.

- **R.I. 4** El sistema almacenará para cualquier tipo de elemento si ha sido eliminado o no. En el caso de borrar el elemento lo marcará como borrado pero no borrará sus datos.
- **R.I. 5** El sistema almacenará para todo usuario registrado un identificador único, el tipo de usuario (administrador o regular), el nombre y apellido del usuario, un email, un nombre de usuario y una contraseña almacenada en un formato seguro.
- **R.I. 6** El sistema almacenará para cada máquina un identificador único, un nombre de la máquina, una descripción de la máquina, su dirección IPv4 y un conjunto de identificadores de scripts asociados a esa máquina.
- **R.I. 7** El sistema almacenará para cada script de aprovisionamiento un identificador único, un nombre y el script en sí.
- **R.I. 8** El sistema almacenará para cada grupo de hosts un identificador único, un nombre y el conjunto de direcciones IP asociadas.
- **R.I. 9** El sistema almacenará para cada máquina un identificador único, un nombre, una descripción, un tipo de máquina, dirección IPv4 e IPv6, dirección MAC, máscara de red, dirección broadcast y dirección de red.

2.3. Modelo de negocio y presupuesto

Aunque la solución propuesta se caracteriza por ser software libre el coste de desarrollo y de implantación nunca es cero.

En el momento de comenzar el desarrollo los fondos son escasos, por lo que el modelo de negocio inicial estaría centrado en obtener los ingresos mínimos que permitan continuar con el proyecto. Una vez superado este primer obstáculo y el software se encuentre más asentado se cambiaría el modelo de financiación para poder obtener mayores beneficios y valor de mercado.

Por tanto se crearía una *Sociedad Limitada de Nueva Empresa*, la cual permite crear una pequeña empresa con pocos recursos iniciales y ofrece ciertas ventajas en este tipo de proyectos:

- Rápida constitución.
- No es necesario un registro de socios.
- Se pueden aplazar deudas del impuesto de sociedades y no existe obligación de realizar pagos fraccionados de este.

- El cambio de denominación social es gratuito temporalmente.

- Se permite el aplazamiento y fraccionado de retenciones del *IRPF*.

El salario medio (datos de 2020) en el sector de la Información y Telecomunicaciones se sitúa en torno a los 34000 euros brutos anuales, lo que supondría aproximadamente unos 2800 euros brutos mensuales, una cifra que en el momento de creación de la empresa es inviable. Una consulta al Instituto Nacional de Estadística nos revela que el gasto medio anual por persona (datos de 2018) es de 12000 euros, lo que se traduce en unos 1000 euros mensuales. A fin de reducir los gastos al máximo y haciendo un promedio a la baja de las dos cifras manejadas anteriormente supondremos un sueldo bruto por trabajador de 1500 euros mensuales (18000 euros anuales).

Otros gastos importantes a tener en cuenta son:

- El capital inicial a aportar en el momento de creación de la empresa (el mínimo son 3000 euros).

- La cuota de autónomos, que en 2020 es de 286,15 euros y que debe pagarse mensualmente, lo que asciende a 3433,80 euros al año.

- Los gastos burocráticos, que ascienden aproximadamente a 250 euros.

- Local en el que desarrollar la actividad laboral y gastos derivados, aproximadamente 600 euros.

En cuanto a gastos relacionados con el propio desarrollo del software se encuentran algunos bienes y servicios como podrían ser repositorios para el almacenamiento del código, integración continua, *Cloud Computing*, adquisición de equipos informáticos y de telecomunicaciones, etc. Muchos de estos servicios tienen versiones gratuitas, lo que ayudaría en el desarrollo inicial del negocio, pero en el momento que se integren nuevos componentes en el equipo puede ser necesario adquirir planes que ofrezcan mejores características.

En el caso de este presupuesto se ha hecho una estimación de los diferentes servicios que serían necesarios para un equipo de tamaño reducido. Además se han tenido en cuenta los diferentes gastos en material informático y otros consumibles necesarios para realizar la actividad laboral.

Las siguientes tablas resumen los gastos iniciales y anuales a afrontar:

Concepto	**Euros/Ud.**	**Cantidad**	**Total (en Euros)**
Capital inicial SLNE	3000	1	3000
Trámites	250	1	250
Salario	1500	12	18000
Cuota autónomos	286,15	12	3433,80
Local y derivados	600	12	7200
		Total	**31883,8**

Concepto	**Euros/Ud**	**Cantidad**	**Total (Euros)**
GitHub Team	3,5	12	42
Atlassian Jira	9	12	108
DockerHub	8	12	96
Equipos informáticos	3500	1	3500
		Total	**3746**

Como se puede observar, el primer año de vida de la empresa costaría aproximadamente 36000 euros, esto teniendo en cuenta que solo tendría un empleado inicial, ya que en el momento de ampliar el equipo cada nuevo integrante implicaría unos 25000 euros al año más.

Aunque se quiere ofrecer una solución de software libre, para sufragar estos gastos se implementaría un sistema de suscripciones mensuales o anuales, en las que se ofrezca soporte personalizado y funcionalidades adicionales o personalizadas.

Capítulo 3

Estado del arte

En este capítulo repasan de las diferentes soluciones que existen actualmente al problema expuesto. Cada una de ellas destaca sobre las demás en un aspecto u otro, pero ninguna llega a aunar todas las características deseadas. Tras eso se hace una crítica a las soluciones vistas y se comenta por qué no son ideales.

3.1. Soluciones actuales

En el ámbito de los despliegues y automatización hay una gran cantidad de software que en mayor o menor medida permiten realizar estas tareas. Con características similares a las de este proyecto destacan tres y son las siguientes:

- **GECOS**. *Guadalinex Escritorio Corporativo Estándar* es un proyecto de la Junta de Andalucía que ofrece una distribución de *GNU Linux* centrada en la administración de sistemas. Este ecosistema, que está basado totalmente en software libre, actúa como centro de control, que permite el despliegue de equipos, su soporte y administración. Además cuenta con un repositorio de software para proveer a las máquinas que se aprovisionan. Utiliza herramientas y tecnologías como *MongoDB* para la gestión de los datos, *Chef* para el aprovisionamiento o *Celery* para el manejo de las colas de tareas.

  Aunque no se trata de un sistema altamente intrusivo para las máquinas que gestiona, este sí requiere que se instale una imagen en la máquina que se vaya a utilizar como maestra. De este modo sólo un equipo es el que tiene acceso a la administración del resto. La gestión de los sistemas se hace desde una interfaz web y permite organizar los equipos según los criterios que se desee e identifica a cada uno de ellos

mediante certificados digitales únicos.

- **Ansible Tower**. Es una solución de *Red Hat* para la administración de ecosistemas de ámbito empresarial ya que se centra en multitud de sistemas operativos, servidores, infraestructuras virtuales y redes. Esta gestión se realiza desde una interfaz web o se puede integrar con cualquier otro sistema mediante una *API REST*.

  La principal característica es que todos los procesos se pueden realizar de forma muy visual y con poca configuración por parte del administrador. Se centra principalmente en el aprovisionado de sistemas mediante *Ansible*, aunque también permite monitorizar estas máquinas y gestionar permisos a los diferentes usuarios.

- **Portainer**. Se trata de una herramienta de gestión y administración de contenedores que amplía las funcionalidades que ofrece *Kinematic*, la *GUI* oficial a la que *Docker* da soporte. Al igual que las anteriores soluciones *Portainer* ofrece sus capacidades a través de una interfaz web y es la más liviana de todas ya que se puede ejecutar fácilmente en un contenedor.

  Permite administrar todos los aspectos de *Docker* como son las imágenes, contenedores, redes y volúmenes entre otros. Además la interacción con todos estos aspectos es muy sencilla. Se trata también de un proyecto *open source*, por lo que es gratuíto.

3.2. Crítica

Si bien todas estas herramientas mencionadas permiten realizar las funcionalidades que se desean ninguna de ellas las aúna todas. En primer lugar *GECOS* ofrece una herramienta de administración de sistemas y de aprovisionado, pero no ofrece ningún tipo de despliegue de servicios. *Red Hat* hace uso de su herramienta estrella en el aprovisionado de sistemas y la lleva al siguiente nivel con *Ansible Tower*, pero al igual que con *GECOS* no ofrece nada relevante en cuando a servicios en contenedores. Por último *Portainer* provee de una excelente solución para este problema, pero no para el aprovisionado y gestión de configuraciones.

Otro aspecto importante es el económico. *GECOS* y *Portainer* son sistemas completamente gratuítos, por lo que serían una opción ideal. En el caso del segundo, *Portainer* ofrece planes superiores con mayor soporte y funcionalidades avanzadas, pero la versión principal es gratuíta. En cambio *Ansible Tower* es de pago, teniendo la suscripción más básica un coste aproximado anual de 5000 dólares anuales y la más avanzada de unos 14000 dólares. Si bien estos precios pueden ser asequibles para una gran organización, para una pequeña o mediana empresa pueden ser inasumibles.

Capítulo 4

Planificación

En este capítulo se expone la metodología de desarrollo que se ha empleado en este proyecto, la cual es una mezcla entre SCRUM y Kanban y las razones por las que se ha elegido este sistema frente a las metodologías clásicas. También se indican todas las tareas que se han realizado en cada *sprint*, los cuales corresponden a cada uno de los meses en los que ha transcurrido el desarrollo.

4.1. Metodología de desarrollo

Podría definirse una metodología de desarrollo como el proceso disciplinado de desarrollo de software con el fin de hacerlo más eficiente. En la actualidad existen muchas de estas metodologías que han surgido a lo largo del tiempo mejorándose unas a otras teniendo en cuenta factores como los costes, planificación, calidad y las dificultades asociadas al desarrollo de un software.

Aunque las bases esenciales no difieren entre metodologías sí hay diferencias en los ámbitos en los que se centran principalmente. Por este motivo voy a utilizar una mezcla de dos metodologías ágiles: *SCRUM* y *Kanban*.

Lo interesante de Scrum es la forma de dividir el proceso de desarrollo del software. Para ello se usan *sprints*, los cuales son un periodo de tiempo (variables) en el que se planifican tareas, se desarrollan y luego se entregan de forma funcional. Esto permite entregas paulatinas, un *feedback* continuo y un desarrollo más dinámico por parte de todos los implicados.

Agrupa todas las buenas prácticas de las metodologías ágiles, y si bien en mi caso estoy algo más limitado al ser *Product Owner*, *Scrum Master* y desarrollador al mismo tiempo, la organización es muy atractiva en este tipo de proyectos. La división por *sprints* hace que cada mes (tiempo que estimo

suficiente para un *sprint* completo) se tenga un conjunto de funcionalidades nuevas. Por otro lado, las reuniones diarias son conmigo mismo (y en ocasiones con el tutor, las cuales pueden solucionar dudas o problemas que surgen), lo que permite que la organización sea mejor y sepa qué hacer cada día.

Kanban por su parte es una metodología que utiliza tarjetas para simbolizar las tareas que se tienen que realizar en el desarrollo de un software. Estas tarjetas se utilizan en un tablero dividido por columnas, las cuales simbolizan tareas que no se han empezado aún, tareas que estan en progreso, ya terminadas, etc.

Pienso que la organización visual de las tareas es bastante provechosa, pues dicha visualización permite saber las tareas que son más urgentes o el estado en el que se encuentran. *GitHub* cuenta con una herramienta llamada *Proyectos* que ofrece un tablero *Kanban* para la organización de tareas y además integración con las *issues* y *pull requests* que se van creando en el proyecto, algo que pienso que es beneficioso para el proyecto y que se ha usado durante el desarrollo del mismo.

Las metodologías tradicionales surgieron cuando aparecieron los primeros sistemas software y están caracterizadas por tener una estructura lineal, en la que al principio se acuerdan las características que debe tener el software y no se modifican durante el desarrollo del sistema, y finalmente se entrega el producto sin dar lugar a cambios. Por lo general se centran en la documentación exhaustiva del proyecto y suelen ser llevadas a cabo por equipos compuestos por multitud de desarrolladores. Además, durante el desarrollo no se tiene una especial comunicación con el cliente.

Aunque tienen algunas ventajas, como tener los objetivos claros desde el primer momento o tener un seguimiento continuo, las desventajas hacen que no tenga cabida este tipo de metodología en el proyecto. Estas son los costes elevados, que no se permitan cambios durante el desarrollo y que la entrega del producto se haga al final del desarrollo, entre otros.

4.2. Temporización

Como se indicaba en la sección anterior los *sprints* planeados tienen una duración aproximada de un mes. Comencé el desarrollo del proyecto a comienzos del mes de febrero y lo he terminado a finales de junio, por lo que se han realizado cinco *sprints*. En total se han creado y finalizado 61 *issues* que se han ido repartiendo por los hitos y desarrollando a lo largo de todo este tiempo.

Al comienzo del desarrollo se definieron tres hitos que corresponden con las tres partes principales de este proyecto. Son: desarrollo del backend,

desarrollo del frontend y documentación.

El trabajo realizado dividido por *sprints* ha sido el siguiente:

4.2.1. Febrero

Comienzo de la investigación de herramientas y tecnologías, además de algunas tareas de documentación. Creación del repositorio en *GitHub*, del proyecto del backend y pipelines en *GitHub Actions*. Clase *Item*, *Client* y sus servicios. Creación del *login* y del despliegue de servicios. Clase *DockerEngine*.

Issues: DEV-1, DEV-3, DEV-4, DEV-6, DEV-11, DEV-12, DEV-13, DEV-17, DEV-25, DEV-27 y DEV-29.

4.2.2. Marzo

Aprovisionado de máquinas. *Multiclient* en el backend. Arreglo de algunos *bugs*. *Dockerizado* del backend y creación de *pipelines*.

Issues: DEV-22, DEV-32, DEV-37, DEV-40, DEV-44 y DEV-45.

4.2.3. Abril

Tests adicionales. Creación del *CLI*. Servicios para el estado del servidor y *heartbeat*. Clase *Machine* y servicio para el manejo de máquinas. Test en profundidad del backend y arreglo de *bugs*. Documentación.

Issues: DEV-19, DEV-24, DEV-26, DEV-35, DEV-37, DEV-40, DEV-43, DEV-49 y DEV-59.

4.2.4. Mayo

Sprint dedicado completamente al frontend. Creación del proyecto y *pipelines*. Servicios para comunicación frontend-backend. *Routing*. Diseño. *Login* y *multiclient*. Componentes.

Issues: DEV-33, DEV-41, DEV-42, DEV-68, DEV-69, DEV-70, DEV-73, DEV-75, DEV-77, DEV-78, DEV-81, DEV-82, DEV-83, DEV-88, DEV-91 y DEV-94.

4.2.5. Junio

Test en profundidad del frontend y arreglo de *bugs*. Documentación y maquetado.

Issues: DEV-20, DEV-23, DEV-92, DEV-100, DEV-102, DEV-103, DEV-104, DEV-105, DEV-106, DEV-107 y DEV-109.

Capítulo 5

Herramientas y tecnologías utilizadas

Este capítulo desarrolla en profundidad todas las decisiones tomadas respecto a las herramientas y tecnologías que se han utilizado en **IPManager**. En cada una de las secciones se hace un breve repaso de las diferentes soluciones que existen actualmente y se exponen los motivos por los que se sigue un rumbo u otro.

Se comienza por el aspecto más básico, la arquitectura de microservicios seleccionada, y se continúa con la integración continua, *GitHub Actions*, y el despliegue en contenedores, *Docker*. Tras esto se elige la herramienta a utilizar en al aprovisionado de sistemas, *Ansible*, y el sistema de gestión de base de datos, *MongoDB*. Finalmente se desarrollan las decisiones tomadas en cuanto a *frameworks* y lenguajes de programación usados tanto en el backend, *Python* y *Flask*, como en el frontend, *Angular*.

5.1. Arquitectura

En este aspecto tan esencial la elección está bastante clara y opto por la arquitectura de microservicios. En el proyecto que se desarrolla no tiene sentido un sistema monolítico en el que frontend y backend se encuentren juntos. En este caso las funcionalidades a desarrollar se centran en aspectos muy concretos y los beneficios de este tipo de arquitectura superan a los inconvenientes.

Entre estos beneficios descato en primer lugar la modularidad, ya que el backend sería totalmente independiente del frontend, lo que permite que se puedan desarrollar infinitas interfaces de usuario, cada una adecuada al uso que se vaya a dar del sistema. Por otro lado destaco la escalabilidad, ya que

en el caso de necesitar más instancias se pueden desplegar de forma rápida y sencilla; y finalmente la seguridad y aislamiento, ya que cada instancia del sistema no interviene con las demás, aislando en este caso los datos de cada caso de uso.

Tradicionalmente el software ha tenido una arquitectura monolítica en la que todos los servicios y funcionalidades están integrados y programados en un mismo sistema, pero como se indica antes, no tiene sentido en este tipo de proyecto.

Las principales ventajas de los microservicios sobre la arquitectura monolítica son:

- **Agilidad**. No es necesario desarrollar todas las funcionalidades completas, por lo que se pueden reutilizar otros microservicios ya desarrollados para suplir las necesidades actuales.
- **Modularidad**. Cada microservicio es independiente del resto, lo que facilita el desarrollo y el despliegue de estos.
- **Escalabilidad**. Debido a la modularidad de estos la escalabilidad horizontal es asequible y muy beneficiosa.
- **Seguridad y aislamiento**. Cada uno de estos servicios encapsula toda su funcionalidad, quedando aislados del resto. Cualquier tipo de vulnerabilidad de la seguridad queda reducido a una parte del sistema general, evitando así pérdidas de información y posibles fallos a otros microservicios.

En cambio este sistema también tiene desventajas. Tener multitud de servicios en ejecución conlleva una configuración y un coste de implantación más alto de lo habitual, además de que no hay una uniformidad a la hora de desempeñar un despliegue. Esto conlleva una complejidad añadida ya que aunque los servicios son mas ligeros y sencillos el sistema que conforman es mucho más complejo. La administración también es más compleja ya que se requieren conocimientos específicos de cada microservicio para este cometido.

5.2. Integración continua

La integración continua es una práctica que consiste en el control de versiones del código que se desarrolla y en la ejecución de pruebas automáticas del mismo de forma periódica con el fin de detectar errores o un mal funcionamiento de una forma rápida. Actualmente es un requisito necesario e indispensable en cualquier tipo de software y debe abordar todos los aspectos del mismo.

GitHub Actions es el sistema elegido para la integración continua. Esto se debe a dos motivos, el primero es que el proyecto se encuentra alojado en *GitHub* y por otro lado *GitHub Actions* permite la ejecución de contenedores *Docker*. Este último aspecto es muy importante en el software que se desarrolla debido a que el backend trabaja con este tipo de tecnología y los tests unitarios deben probar todas estas funcionalidades. Además para las pruebas del resto de funcionalidades el poder ejecutar un contenedor con una base de datos facilita mucho el proceso de integración continua. Así mismo son sistemas que se encuentran perfectamente integrados el uno con el otro.

Existen multitud de servicios para este tipo de pruebas:

- Jenkins
- Travis CI
- Bamboo
- GitHub Actions
- GitLab CI
- Circle CI

Todos ellos comparten características y son realmente similares. Algunos como *Jenkins* permiten la integración de multitud de plugins y otros permiten además despliegues continuos. En el caso de este software la característica que más nos interesa es que sea gratuito y la mayoria de ellos lo son para proyectos de software libre. Esta es otra de las características que hacen que estos sistemas gratuitos tengan tanta popularidad.

5.3. Despliegue del software en contenedores

En este ámbito se va a utilizar *Docker* para desplegar tanto frontend como backend. Este sistema provee una capa adicional de abstracción y de virtualización de las aplicaciones, lo que nos permite ejecutar un software de manera aislada sin tener que depender de complejas configuraciones de máquinas virtuales o hipervisores. Por otro lado los recursos pueden ser también aislados.

Para la creación de estos contenedores se utilizan los denominados *Dockerfile*, que son archivos de texto plano con las diferentes instrucciones que crearán a voluntad el entorno de ejecución de nuestro software. En el caso de este proyecto se van a crear dos de estos archivos, uno el frontend y otro

para el backend, y la creación y despliegue de las imágenes generadas se realizará mediante *DockerHub*.

Como se ha indicado anteriormente este proceso también se puede automatizar, ya sea con scripts creados a mano o con el uso los hooks de *DockerHub*, que construyen las imágenes específicas cada vez que se haga un cambio en el código o cada vez que se produzca un evento concreto.

Docker Compose es otra herramienta que facilita aún más este proceso, ya que a partir de un archivo *YAML* permite crear estos contenedores, configurarlos y conectarlos de una forma muy sencilla. En este proyecto también se incluye uno de estos archivos.

Finalmente existen otras herramientas muy interesantes, como son *Kubernetes*, *OpenShift* o *Mesos*, que nos permiten orquestar y escalar los contenedores según los criterios que se configuren. En este proyecto no se hace uso de ellas pero en el caso de que se quisiera dar un paso más en el despliegue del software podrían ser muy interesantes.

5.4. Aprovisionamiento

Existen muchas herramientas que permiten el aprovisionamiento de sistemas, algunas más centradas en el ámbito *Cloud* y otras de uso local.

En el caso de este proyecto la opción que mejor encaja es *Ansible*. Debido a su sencillez de uso y de no requerir un servidor central lo hace ideal para el uso en el backend. Además, ya que se encuentra desarrollado en *Python* y a que existe un *SDK*, la integración es inmediata, solo teniendo que desarrollar las funcionalidades que se quieran.

Por otro lado la sencillez de los *Playbooks* lo hace más atractivo aún, ya que la sintaxis de los archivos *YAML* es muy sencilla. En cuanto a la conexión mediante *SSH* solo se requiere que el backend tenga conexión a internet, por lo que no es necesario ningún protocolo o configuración adicional para que funcione.

Algunas otras alternativas son:

- Chef. La configuración de las máquinas se hace de forma procedural y se depende de un servidor central que almacene las configuraciones o *recetas*. Además ofrece análisis e informes de las máquinas aprovisionadas.
- Puppet. Es un conjunto de herramientas que permiten orquestar y administrar grandes conjuntos de máquinas. Al igual que *Chef* depende de un servidor central y permite ampliar su funcionalidad a través de módulos.

5.5. Base de datos

Elegir un sistema de gestión de base de datos es una de las decisiones más importantes a la hora de diseñar y desarrollar un software. Existe una gran variedad de tipos de bases de datos y hay que tener en cuenta una serie de cuestiones que serán determinantes a la hora de elegir un tipo u otro. En este proyecto se va a usar una base de datos *NoSQL*, concretamente *MongoDB*.

Las bases de datos de tipo *SQL* se basan en las relaciones entre los datos. Estos se introducen en registros y luego se organizan por tablas, columnas y tuplas, permitiendo relacionarlos de manera sencilla. El principal lenguaje de consultas es el *Standard Query Language* (*SQL*), el cual esta compuesto por una serie de comandos de diferentes tipos, que se usan para unos cometidos u otros. Sus principales características son el esquema rígido que se define previo al uso que garantiza el esquema *ACID*.

Por las características del proyecto este modelo queda excluido, ya que el tipo de datos que se va a manejar no requiere de grandes relaciones entre ellos y además el esquema puede ser cambiante. Podría ocurrir que ciertos valores no se encontraran almacenados, bien porque no son necesarios o bien porque el usuario decide no insertarlos, por lo que sería mantener una estructura que no se está cumpliendo.

Por otro lado el tipo de consultas que se van a realizar no son extremadamente complejas. Los datos manejados no tienen relaciones entre sí y las consultas serían realmente básicas. Otro punto a favor en este aspecto para las bases de datos *NoSQL* es la velocidad a la hora de realizar las consultas.

En el caso de la integración con el software en las bases de datos *SQL* se utilizan los llamados *ORM* (*Object Relation Mapper*). Estos permiten realizar consultas a estas bases de datos de una forma más amigable en el lenguaje que se esté usando, lo que implica que se tenga que volver a redefinir el esquema para poder manejar estos datos. En cambio con *NoSQL* esta integración suele ser mas sencilla, al utilizarse directamente objetos como diccionarios.

Dentro de las bases de datos *NoSQL* existen diferentes tipos. En el caso de este proyecto el modelo que mas encaja es el documental, en la que una semiestructura flexible almacenada en forma de documentos es ideal. *MongoDB* es una gran elección, ya que los datos se almacenan en *BSON* (*Binary JSON*), lo que ofrece aún más flexibilidad a la hora de almacenar objetos. También permite crear índices en cualquier clave y el balanceo de carga en el caso de realizar grandes cantidades de consultas simultáneas.

Actualmente también están destacando las bases de datos en la nube o *DBaaS* (*DataBase as a Service*), las cuales estan optimizadas para operacio-

nes en entornos virtualizados. La principal característica de estos servicios es que se suele pagar por el uso de almacenamiento y además conceptos como la escalabilidad o la alta disponibilidad estan asegurados. En el caso de *MongoDB* existe *Atlas*, que incluso ofrece planes gratuitos. Otro ejemplo de *DBaaS* sería *mLab*, muy similar al anterior pero con una configuración más sencilla. Para el desarrollo de este software este aspecto es muy interesante, ya que al tratarse de un microservicio el no estar atado a una base de datos local permite que se pueda desplegar tambien en la nube.

5.6. Backend

5.6.1. Lenguaje de programación y framework

Actualmente existen multitud de lenguajes de programación que podrían usarse sin problema alguno para desarrollar una API de las características que se requieren. Las características deseadas que deberían ofrecernos estos en el caso de este software son sencillas aunque a la par difíciles de encontrar en ocasiones. Algunos de estos lenguajes que se suelen utilizar en el desarrollo de *APIs* son *Java*, *JavaScript*, *PHP*, *Python*, *Ruby*, *C#* y *Go*, entre otros.

- Para el manejo de datos se requiere que se pueda conectar a *MongoDB* y esto lo cumplen los lenguajes mencionados, por lo que todos son buenos candidatos en este caso.
- En cuanto al aprovisionamiento se requiere algún tipo de *SDK* de *Ansible*. Los lenguajes que soportan esto son *Go*, *Python*, *Ruby*, *PHP* y *JavaScript*, mientras que el resto tienen un soporte limitado.
- Se debe poder administrar *Docker* y en este caso, al igual que con *MongoDB*, todos los lenguajes tienen *SDKs* disponibles.

Por otro lado, debido a la arquitectura del proyecto, no es necesario que el *framework* elegido tenga la arquitectura *MVC* ya que de la vista se encarga el frontend. Por este motivo todos aquellos que siguen este modelo quedan descartados. Podrían utilizarse sin problema alguno, pero no tendría mucho sentido ya que no le estaríamos sacando todo el partido posible a los mismos.

Anteriormente se mencionaba como aspecto importante los recursos que tienen estos lenguajes y cierto es que algunos pueden ofrecer más que otros, bien sea porque son más antiguos o bien porque son más usados y la comunidad es mayor. En esto destaca *Python*, también motivado por opinión personal, ya que existen una infinidad de librerías y recursos para este lenguaje y se puede desarrollar cualquier aplicación de forma sencilla e intuitiva. Además, en el caso del aprovisionamiento, *Ansible* está programado en

Python, por lo que la integración con su *SDK* sería directa, sin problema alguno.

La ausencia de tipado en *Python* da una mayor flexibilidad y libertad a la hora de desarrollar, pero puede inducir a errores, por lo que es necesario tener un especial cuidado. Otro aspecto es que se trata de un lenguaje interpretado, algo que agiliza el desarrollo ya que no hay que emplear tiempo extra en el compilado. También tiene una sintaxis muy sencilla que facilita la comprensión del código.

En cuanto al *framework*, como se indicaba antes no es necesario que disponga de arquitectura *MVC*, por lo que *Django* queda descartado. En su defecto se usará *Flask* junto a otros módulos como *Flask-RESTPlus* o *Marshmallow* (usado para la definición de esquemas).

La elección de un lenguaje u otro también depende de los recursos que nos ofrezca, esto es librerías, *frameworks* y todas aquellas características que hagan destacar un lenguaje sobre otro. También influye la experiencia que se tenga, ya que afrontar un gran proyecto con un lenguaje que nunca has usado puede ser un gran reto.

Otros *frameworks* que se han tenido en cuenta a la hora de esta elección han sido:

- **Java**. Destacan *frameworks* como *Spring* y *Struts*.
- **JavaScript**. *Express* es el más utilizado y se ejecuta sobre *Node.js*. Otros ejemplos son *Sails* o *Meteor*.
- **PHP**. Los más popular son *Slim* y *Lumen*. Ambos son *microframeworks* bastante sencillos con muchas funcionalidades incorporadas.
- **Ruby**. *Roda* y *Sinatra* son los más utilizados.
- **C#**. El más popular es *.NET Core*, de *Microsoft*.
- **Go**. Tanto *Revel* como *Gin* son los más usados.

5.7. Frontend

En el caso de este proyecto la primera elección que se toma es la de abandonar el *stack HTML/CSS/JS* ya que realmente no ofrece nada novedoso sobre las demás soluciones. Tras esto, elijo *Angular*.

Angular es un *framework* que aborda todos los aspectos del desarrollo frontend, desde la parte visual hasta las comunicaciones. Su arquitectura se basa en componentes que se pueden crear y personalizar a voluntad y el

lenguaje usado para su desarrollo es *TypeScript*, lo cual permite un mayor control de los datos que se manejan. Integra además multitud de librerías, como *RxJS*, para aprovechar sus virtudes. Entre sus principales características destacan el enlace de datos bidireccional (*2-way data-binding*) entre el modelo y la vista, la inyección de servicios y dependencias, que facilita el desarrollo y la comprensión del código, y la validación de datos y mecanismos de seguridad integrados. Por contra la curva de aprendizaje es mas grande, ya que integra multitud de conceptos diferentes que no se contemplan en las demás soluciones.

Por otro lado otras opciones que se han barajado han sido:

React. Tiene un enfoque reactivo y trabaja con un *DOM* virtual en varias capas, lo que permite que sólo se actualicen aquellas partes de la página que deban actualizarse. También permite la creación y reutilización de componentes personalizados, lo que dota a esta librería de mucha flexibilidad a la hora del desarrollo. Por contra sólo se trata de una librería centrada en la parte visual del frontend, por lo que el manejo de datos entre componentes o cualquier tipo de comunicación externa, como *HTTP*, quedan a cargo del desarrollador.

Vue. En este caso Vue comparte conceptos de *React* y de *Angular*, por lo que sería el punto intermedio entre ambos. Destaca por ser mas liviano que estos y por su simplicidad a la hora del desarrollo, lo que hace que la curva de aprendizaje sea bastante menor. Por el contrario, al igual que con *React*, las comunicaciones corren a cargo del programador, lo que es un punto en su contra.

En cuanto a estas alternativas, aunque es muy interesante el enfoque que tienen se quedan cortas a la hora de la comunicación con el backend. El desarrollador es el que debe proveer de los métodos de comunicación, lo que requiere más tiempo. En cambio con *Angular* esos aspectos ya se encuentran integrados, lo que simplifica mucho el proceso. Por otro lado aspectos como el *2-way data-binding* y la validación de datos son también interesantes, ya que aportan flexibilidad y agilizan el desarrollo.

Por estos motivos *Angular* es la mejor solución en el caso de este software y en caso de quedarse corta en algunos aspectos, permite la integración de otras librerías. Además *Angular* incluye librerías y mecanismos para tests unitarios e integración continua, algo muy necesario en el desarrollo de un software.

Finalmente para los test unitarios y *end-to-end* (*e2e*) se van a utilizar herramientas que se integran perfectamente con *Angular*. Para los primeros se utiliza *Karma*, que viene incluido por defecto en el *framework* y permite comprobar el funcionamiento unitario de cada uno de los componentes.

Para los tests *e2e* se utilizará *Cypress*, una herramienta gratuita muy potente que permite realizar tests al frontend como si de un usuario se tratara, haciendo uso de datos sobre las diferentes funcionalidades del sistema y además permite ejecutar estos tests en casi todos los navegadores actuales. En este aspecto se han tenido en cuenta opciones como *Puppeteer* y *Nightwatch.js*, pero se descartaron pues la sintaxis era mas compleja.

Capítulo 6

Solución propuesta

En este capítulo se expone la solución que se propone al problema. Aunque en las siguientes secciones se explica con mas detenimiento el funcionamiento propuesto de cada módulo, a grandes rasgos se propone la creación de un backend y un frontend con las siguientes características:

El **backend** estaría compuesto por una serie de módulos con características únicas, buscando la simplicidad. El módulo más básico sería el de almacenamiento (sección 6.1), el cual actuaría como conector con *MongoDB* y permitiría realizar operaciones con datos. A partir de este módulo se crearían los correspondientes al manejo de todos los tipos de datos, estos son *Clientes* (seccion 6.2), *Usuarios* (seccion 6.3), *Machines* (seccion 6.6), *Hosts* y *Playbooks*. Estos dos últimos se manejan a través del módulo de aprovisionado, que permite también la ejecución de *Playbooks* de *Ansible* (seccion 6.5).

Para el despliegue de servicios se ha propuesto un módulo que permite realizar operaciones con las imágenes y contenedores de *Docker* de manera sencilla (sección 6.4).

En cuanto a la *API* que provee el backend cada módulo implementa una serie de *endpoints*. Para garantizar que no se producen accesos no deseados a estos se ha propuesto un módulo de autenticación a partir del módulo de almacenamiento y de *tokens JWT* (sección 6.7). También se explica la estructura de los servicios usados para conocer el estado del backend (sección 6.8).

Finalmente se desarrolla la propuesta de las diferentes variables de entorno a utilizar y el *CLI* para realizar algunas operaciones básicas desde una línea de comandos (secciones 6.9 y 6.10).

El **frontend** se compondría de diferentes páginas, cada una dedicada a uno de los principales módulos del backend: aprovisionado, despliegue de servicios y administración de máquinas y usuarios (sección 6.11).

6.1. Almacenamiento

La principal idea en cuanto al almacenamiento es tener algún tipo de clase que sirva de conector de cualquier tipo de dato que se quiera almacenar y que a su vez permita administrar estos elementos de forma sencilla.

Desarrollar una estructura de herencia es el camino a seguir en este caso, el siguiente sería intentar lograr la mayor versatilidad posible.

Para facilitar el desarrollo de los diferentes módulos se propone desarrollar una base común que sirva como puente para realizar operaciones en las diferentes colecciones de la base de datos.

Esta podría llamarse *Item* e implementaría las cuatro operaciones básicas necesarias para manejar cualquier tipo de dato: crear, modificar, obtener y eliminar. Una vez implementados esos métodos básicos el resto de módulos que se desarrollen sólo tienen que sobrescribir las operaciones necesarias para adecuarlas a cada uso concreto.

Por otro lado, para manejar el cliente de *MongoDB* se propone crear una clase, llamada *MongoEngine*, que permita realizar diferentes operaciones en las colecciones y bases de datos. Además, podría obtenerse también algún tipo de estadísticos de este servicio. Ambos módulos funcionarían conjuntamente para ofrecer un conector a la base de datos sencillo y capaz de adaptarse a cualquier tipo de uso.

Para la configuración de cada clase que pueda heredar de *Item* se debería definir un nombre de la colección a usar por esa clase y además el esquema de la colección. Este esquema sería el conjunto de datos que se pueden almacenar en la colección.

En conclusión, se propone:

- Clase *Item*
- Clase *MongoEngine*

6.2. Clientes

Los clientes son aquella unidad que permite diferenciar un conjunto de datos de otro ya que cada uno de ellos cuenta con su propia base de datos.

Debido a la *API REST* que proporciona el backend se propone que para acceder a un cliente u otro se utlice el subdominio de la *URL* a la que se le hacen las peticiones.

Por tanto, este módulo sería el encargado de diferenciar y manejar los diferentes clientes que podrían acceder al backend. La clase *Customer* es la

propuesta en este caso. Heredaría las funcionalidades de *Item* y las complementaría con la gestión de estos clientes.

Para almacenar la información de estos clientes se propone también tener un "cliente base" el cual se encargaría de tener un registro de cada uno de los clientes disponibles en el sistema y de la base de datos que utiliza cada uno.

6.3. Usuarios

Este módulo es el encargado de la gestión de los usuarios asociados a un cliente y sus funcionalidades serían las siguientes:

- Crear usuarios
- Modificar usuarios
- Obtener información de los usuarios
- Eliminar usuarios

Para satisfacer los requisitos del software se propone lo siguiente:

- Los datos a almacenar por usuario son: Identificador único, Tipo de usuario, Nombre, Apellidos, Email, Nombre de usuario, Contraseña.
- La contraseña se encriptaría con encriptado simétrico *Fernet*.
- Los tipos de usuario aceptados serían 'admin' y 'regular'.
- El email será único.

Debido a que partimos del módulo *Item* para desarrollar el de usuarios solo es necesario heredar de este y hacer algunas modificaciones. En cuanto al borrado y obtención de usuarios no sería necesario hacer ningún tipo de modificación. Por otro lado, en las operaciones de inserción y modificación sólo habría que añadir el código necesario para el encriptado de la contraseña y para la comprobación del tipo de usuario y del email único.

Este módulo contaría con los siguientes *endpoints*:

- *GET /user/:user* - Obtener la información de un usuario.
- *POST /user* - Crear un usuario.

- *PUT /user* - Modificar un usuario.
- *DELETE /user* - Eliminar un usuario.
- *POST /user/query* - Listar usuarios.

6.4. Despliegues

Este módulo sería el encargado de realizar los despliegues de los servicios mediante contenedores *Docker*. Para llevar a cabo esto el módulo se conectaría a un servidor de *Docker* y contendría los métodos necesarios para ejecutar contenedores, imágenes y operaciones en ambos.

En el caso de los despliegues no es necesario el almacenamiento de datos de ningún tipo por lo que tampoco sería necesario crear clases que hereden de *Item*. En cambio, se propone la creación de una clase, llamada *DockerEngine*, que permita conectarse al cliente de *Docker* e implemente los métodos necesarios para hacer las operaciones deseadas.

Estas serían:

- Ejecutar operaciones en todos los contenedores.
- Ejecutar operaciones en un contenedor en concreto.
- Ejecutar operaciones en todas las imágenes.
- Ejecutar operaciones en una imagen en concreto.

Las anteriores operaciones corresponderían con los siguientes *endpoints*:

- *POST /deploy/container*
- *POST /deploy/container/single*
- *POST /deploy/image*
- *POST /deploy/image/single*

Además, del mismo modo que se propone en el módulo de almacenamiento, podrían obtenerse una serie de datos estadísticos de este servicio.

6.5. Aprovisionamiento

Sería el encargado de aprovisionar sistemas mediante el uso de Ansible y se centraría exclusivamente en la ejecución de *Playbooks*. Por el funcionamiento de *Ansible* debería establecer una conexión *SSH* con los hosts indicados y ejecutaría las órdenes que se encuentran en el *Playbook*.

En el caso de este módulo son necesarias dos clases extra, una para almacenar los *Playbooks* y otra para almacenar los grupos de hosts donde se van a ejecutar esos *Playbooks*. Las clases serían:

- *Hosts*
- *Playbooks*

Los *endpoints* que se proponen para manejar ambas clases son:

- *GET /provision/hosts/:name* - Obtener la información de un grupo de *hosts*.
- *POST /provision/hosts* - Crear un grupo de *hosts*.
- *PUT /provision/hosts* - Modificar un grupo de *hosts*.
- *DELETE /provision/hosts* - Eliminar un grupo de *hosts*.
- *POST /provision/hosts/query* - Listar grupos de *hosts*.
- *GET /provision/playbook/:name* - Obtener la información de un *Playbook*.
- *POST /provision/playbook* - Crear un *Playbook*.
- *PUT /provision/playbook* - Modificar un *Playbook*.
- *DELETE /provision/playbook* - Eliminar un *Playbook*.
- *POST /provision/playbook/query* - Listar *Playbooks*.

Siguiendo los requisitos del software, las restricciones son:

- Se almacenará para cada *Playbook* un identificador único, un nombre y el *Playbook* en sí.
- Se almacenará para cada grupo de *Playbooks* un identificador único, un nombre y el conjunto de direcciones IP asociadas.

Por otro lado, para ejecutar los *Playbooks* se propone la creación de una clase *AnsibleEngine*, que sería la encargada de implementar aquellos métodos necesarios para ejecutarlos. También se propone el siguiente *endpoint*:

- *POST /provision*

6.6. Máquinas

Módulo encargado del almacenamiento y gestión de máquinas y dispositivos. Tendría estructura similar a las clases *Host* o *Playbook*, ya que heredaría las funcionalidades que ofrece la clase base *Item*.

Los *endpoints* propuestos para este módulo son:

- *GET /machine/:user* - Obtener la información de una máquina.
- *POST /machine* - Crear una máquina.
- *PUT /machine* - Modificar una máquina.
- *DELETE /machine* - Eliminar una máquina.
- *POST /machine/query* - Listar máquinas.

Requisitos del software:

- El sistema almacenará para cada máquina un identificador único, un nombre, una descripción, un tipo de máquina, dirección IPv4 e IPv6, dirección MAC, máscara de red, dirección broadcast y dirección de red.

6.7. Autenticación

El objetivo de este módulo es agregar una capa de seguridad al backend evitando que puedan acceder a él usuarios que no se encuentran registrados. Esta capa se aplicaría a todos los *endpoints* que se quieran proteger de accesos indeseados.

Se propone por tanto un módulo que permita la autenticación de usuarios en el sistema. Este haría uso de *JWT* (*JSON Web Token*) para encriptar la información. Debido a que el backend es una *API REST* el método de enviar este *token* en cada una de las peticiones será la inclusión de este en

cabeceras de las peticiones que se realicen. La cabecera a usar podría ser: *x-access-token*.

En cada petición este *token* deberá ser decodificado, se comprobará al usuario al que pertenece y finalmente se permitirá el acceso o no. Todos los *endpoints* del backend estarían protegidos por esta autenticación, salvo:

- *GET /login*
- *GET /api/heartbeat*

El servicio de autenticación a implementar debería implementar:

- *GET /login*
- *GET /logout*

6.8. Estado del backend y *heartbeat*

Para comprobar el estado del backend se propone la creación de un servicio que devuelva información asociada al modulo de despliegues y al de almacenamiento. Para ello se propone agregar métodos a las clases *MongoEngine* y *DockerEngine* que devuelvan esta información asociada.

El *endpoint* sería el siguiente:

- *GET /status*

Debido a que este *endpoint* devolvería información relevante, este debería estar también protegido por la autenticación comentada en secciones anteriores.

Anexo a este estado se propone el siguiente *endpoint*:

- *GET /api/heartbeat*

En este caso solo devolvería si los diferentes módulos del backend se encuentran funcionando correctamente o no, y no sería necesario que estuviera autenticado. Este *endpoint* podría ser usado por *Docker* en el caso de que el backend se ejecute en un contenedor de este tipo.

6.9. Variables de entorno

Para el funcionamiento del backend y el frontend sería necesaria la definición de variables de entorno que permitan configurar ciertos aspectos de estos. Serían:

- *Hostname* y puerto de *MongoDB*.
- Nombre de la base de datos a utilizar.
- Claves de encriptado para las contraseñas y los token de autenticación.
- *Hostname* de *Docker*.
- URL y puerto del backend.

En el apéndice B se encuentra un listado con todas las variables de entorno que se utilizan en el backend y en el frontend, junto a sus valores por defecto.

6.10. *CLI*

Módulo propuesto para poder realizar ciertas operaciones desde la terminal, sin necesidad de ejecutar el backend. Podría ser usado en la primera instalación de este y/o para crear unos primeros usuarios o clientes.

Funcionalidad propuesta:

- Crear clientes.
- Activar o desactivar clientes.
- Agregar usuarios a un cliente.

6.11. Frontend

El desarrollo de un backend que ofrezca una *API REST* permite que se pueda desarrollar cualquier tipo de frontend, ya sea web, una aplicación móvil o incluso acceso mediante línea de comandos. En este caso, para satisfacer los requisitos del software, se propone crear un frontend que permita realizar todas las operaciones anteriormente mencionadas.

Este podría tener las siguientes páginas:

- /: Donde mostrar el estado general del sistema.
- */admin*: Administración de usuarios.
- */deploy*: Administración de los despliegues, contenedores e imágenes.
- */provision*: Administración del aprovisionamiento, grupos de *hosts* y *Playbooks*.
- */machines*: Administración de las máquinas.

Por otro lado, atendiendo a los requisitos del software, se propone la creación de un componente para generar tablas de forma dinámica que encapsule todas las funcionalidades requeridas y que además permita tener una sincronía en la forma de mostrar los datos al usuario. Además debe incluir autenticación de los usuarios.

Para la comunicación con la *API* se propone la creación de diferentes servicios centrados en cada uno de los módulos del backend. De esta manera los servicios pueden inyectarse en los componentes y la comunicación es directa. Estos serían:

- Autenticación
- Clientes
- Usuarios
- Hosts
- Playbooks
- Máquinas
- Aprovisionamiento
- Despliegues
- Estado

Capítulo 7

Implementación

Este capítulo desarrolla de manera profunda la implementación de cada uno de los módulos que se han propuesto en el *capítulo 6*. Esto también comprende las características de los módulos, estructuras creadas, decisiones tomadas y el flujo de trabajo en cada uno de ellos. Además se incluyen algunos ejemplos de uso.

Las secciones 7.1 y 7.2 se centran en las partes más importantes en el backend. Estas son el almacenamiento de datos, desarrollando las clases *MongoEngine* e *Item*; y los servicios, que cuentan con una estructura común en algunos casos.

Tras esto se desarrollan el resto de módulos, con especial atención al despliegue de sistemas, donde se explica el concepto de "operación" que se ha tomado.

Finalmente se expone la implementación del frontend, comenzando por los servicios que permiten las comunicaciones con el backend. Se continúa con la capa de seguridad que aporta el *Guard* de *Angular* y se finaliza con las interfaces y componentes usados para representar y mostrar los datos (sección 7.11).

7.1. Almacenamiento

Como se explicaba en el capítulo anterior este módulo es el que permite el almacenamiento de los datos en *MongoDB*. Para ello se han desarrollado las clases que se proponían. La primera, *MongoEngine*, es el conector con el gestor de bases de datos; y la segunda, *Item*, que abstrae todas las operaciones que se pueden realizar con los datos.

7.1.1. MongoEngine

Esta clase es la que nos permite conectarnos al servidor de *MongoDB* y realizar todas aquellas operaciones que se deseen. Se ha desarrollado como un *singleton* para que solo haya una instancia activa al mismo tiempo.

Cada instancia de *MongoClient* que se crea tiene un *pool* de conexiones, que abre y cierra sockets bajo demanda para manejar todas las operaciones que se realicen de forma simultánea. Por este motivo es contraproducente crear una instancia de *MongoClient* cada vez que se quiera realizar una operación. Por defecto el tamaño de este *pool* es de 100, pero podría incrementarse si fuera necesario.

Los métodos implementados son los siguientes:

- Creación del cliente.
- Borrado de bases de datos y de colecciones. Usados principalmente en los tests unitarios.
- Asignación de base de datos y colección actual. Usados para seleccionar la base de datos necesaria para cada cliente, y la colección donde realizar operaciones.
- Datos estadísticos. Se extraen del cliente de *MongoDB* y se devuelve en un diccionario con la siguiente forma:

```
{
        "is_up": bool,
        "data_usage": list,
        "info": dict || str
}
```

- *is_up*: Indica el estado del servicio, *true* si se encuentra funcionando correctamente, *false* en caso contrario.
- *data*: Información de uso de datos de las bases de datos almacenadas.
- *info*: Información adicional del cliente.

7.1.2. Item

Clase base de la que heredan todas las clases que necesitan algún tipo de almacenamiento de datos. Sus datos miembros son:

- *table_name*: Nombre de la tabla (o colección) donde se van a almacenar los datos.
- *table_schema*: Esquema de la tabla equivalente a la proyección de *MongoDB*. Es un diccionario compuesto por claves (nombres de los datos que va a almacenar la tabla) y por un valor 1 ó 0. Todas las claves que aparezcan en este diccionario serán claves válidas para almacenar en la tabla. Los valores indican lo siguiente:
  - *1*: El dato se devuelve al hacer una consulta.
  - *0*: El dato no se devuelve al hacer una consulta.

Un ejemplo de uso sería:

```
table_schema = {
        'domain': 1,
        'db_name': 1
}
```

Al realizar consultas se puede sobrescribir este esquema, para obtener únicamente los datos deseados.

- *data*: Diccionario donde se almacenan los datos de los objetos que se creen de este tipo o que se obtengan al hacer una consulta.

Para el diseño y desarrollo de esta clase se ha intentado abstraer y simplificar al máximo las funcionalidades de esta, para que se pueda adaptar a cualquier tipo de uso. No se han hecho uso de todas las funciones que ofrece *PyMongo* en cuanto a manejo de datos en colecciones. Aún así los métodos son los mínimos para que se pueda cualquier tipo de operación básica. Los métodos implementados en la clase *Item* son los siguientes:

- *cursor*: Hace uso del nombre de la tabla para obtener el cursor a ella y poder realizar todas las operaciones necesarias.
- *find(criteria, projection)*: Devuelve todos los elementos (sólo los parámetros indicados en la proyección) que cumplan os criterios de búsqueda.
- *insert(data)*: Inserta un elemento o lista de elementos en la colección. También agrega dos claves adicionales a este que son:
  - *enabled*: Por defecto a *true*. Indica si el elemento está activo o no.
  - *deleted*: Por defecto a *false*. Indica si el elemento ha sido borrado o no.

- *update(criteria, data)*: Actualiza todos los elementos que cumplan con el criterio. *Data* es un diccionario con las claves y valores a actualizar.
- *remove(criteria, force)*: Elimina los elementos que cumplan con el criterio. Por defecto el parámetro *force* tiene valor *true*, lo que elimina completamente los elementos. En el caso de asignarle un valor false en lugar de eliminar los elementos los modifica, cambiando sus propiedades *enabled* a *false* y *deleted* a *true*.

Flujo de trabajo

Con esta implementación cualquier clase que se quiera que tenga la capacidad de almacenar datos solo tendrá que heredar de esta clase. Podrá sobrescribir aquellos métodos a los que quiera agregar más funcionalidad y podrá también implementar nuevos métodos, ya que tiene acceso al cursor de *MongoClient* para realizar cualquier tipo de operación permitida.

Un ejemplo de uso podría ser:

```
# Definicion de la clase Feature y sobreescritura el metodo insert
class Feature(Item):
        table_name = 'features'
        table_schema = {
                'name': 1,
                'description': 1,
                'data': 1
        }

        def insert(self, data=None):
                if data is not None:
                        if data['name'] != '':
                                return super(Machine, self).insert(
                                    data)

                return False


# Uso de esta clase y sus metodos
# Se crea un objeto
Feature().insert({
        'name': 'feature_1',
        'description': 'Cool feature 1',
        'data': {}
})

# Se obtiene la descripcion del objeto con nombre 'feature_1'
Feature().find({name': 'feature_1'}, {'description': 1})

# Se modifica la descripcion del objeto con nombre 'feature_1'
Feature().update({'name': 'feature_1'}, {
        'description': 'Updated feature 1'
})

# Elimina completamente el objeto
Feature().remove({name': 'feature_1'}, force=true)
```

Como se puede observar con muy poco código se pueden crear nuevos elementos almacenables en base de datos. Esto permite que la creación de nuevos módulos sea muy sencilla. En el caso ideal sólo sería necesario crear una clase y sobrescribir sus datos miembros, ya que toda la funcionalidad la heredaría de *Item*.

7.2. Servicios

Cada uno de los servicios implementa los *endpoints* que estarán disponibles para los usuarios de la *API* y permiten el primer procesado de los datos previo envío a los módulos que se encargan de cada tipo de dato.

Los servicios usados para manejar datos, que provienen de clases que heredan de *Item*, tienen la estructura que se muestra a continuación. En cambio, los que no se utilizan para el manejo de datos tienen una estructura libre acorde a las necesidades de cada uno.

En los siguientes *endpoints* suponemos que el servicio *service* permite hacer operaciones con objetos de tipo *Service*.

- *GET /service/:name*: Devuelve los datos asociados a un objeto de tipo *Service*.
- *POST /service*: Crea un objeto de tipo *Service*.
- *PUT /service*: Modifica un objeto de tipo *Service*.
- *DELETE /service*: Elimina un objeto de tipo *Service*.
- *POST /service/query*: Permite hacer consultas más elaboradas haciendo uso del criterio de búsqueda y de la proyección de *MongoDB*. Los objetos obtenidos se devuelven en forma de diccionario (en el caso de un sólo resultado) o de lista de diccionarios (más de un resultado).

Para simplificar el código en los servicios he creado algunas funciones auxiliares:

- *response_by_success*: Devuelve un mensaje predeterminado y un código en función del resultado de la operación que se haya procesado.
- *response_with_message*: Devuelve un mensaje y código personalizados.
- *validate_or_abort*: Valida los datos de entrada del *endpoint* en función del esquema que se quiera validar.

- *parse_data*: Devuelve los datos que se le pasan con la forma del esquema que se quiera. Tiene en cuenta si es un solo dato o un conjunto.

Por otro lado, se han creado diferentes esquemas con *Marshmallow* para validar todos los datos que se manejan en los servicios. De este modo se tiene control absoluto del tipo de dato que se esté manejando en cada momento.

Flujo de trabajo

En el momento de recibir una petición los datos se cotejan con el esquema que se use en el *endpoint* y se asegura que los datos son los esperados, en caso contrario se rechaza la petición, para ello se hace uso de la función anterior *validate_or_abort*. En cuanto a la salida de datos se usa *parse_data* para asegurar que los datos devueltos tenga la estructura esperada.

Además, en cada petición se comprueba cuál es el cliente al que pertenece la misma. Por tanto cada una de ellas procesa los datos acorde al cliente seleccionado.

Un ejemplo de uso de los conceptos desarrollados podría ser:

```
1 api = Namespace(name='feature', description='Features management')
2
3 @api.route('')
4 class FeatureService(Resource):
5         @staticmethod
6         @token_required
7         def post():
8                 data = validate_or_abort(FeatureSchema, request.
                      get_json())
9                 return response_by_success(Feature().insert(data))
```

El ejemplo anterior hace uso del decorador *@token_required*, el cual se explica con más detalle en la siguiente sección.

7.3. Clientes

Clase que se encarga del manejo de los *customers* o clientes del backend. Todas las operaciones relacionadas con clientes así como los datos asociados a ellos se almacenan en la base de datos que se define en la variable de entorno *BASE_DATABASE*. Este último aspecto es el que en el *capítulo 6* se denominaba "cliente base".

Los datos miembros de esta clase son:

- *table_name*: customers

- *table_schema*: (Por defecto todos los valores a 1).
  - *domain*: Subdominio al que hace referencia este cliente.
  - *db_name*: Nombre de la base de datos donde se almacenarán todos sus datos.

Los métodos que implementa esta clase son:

- *is_customer(customer)*: Comprueba si el customer existe. En caso de existir se devuelve si está activo o no.
- *set_customer(customer)*: Asigna el customer al que se le van a hacer consultas de base de datos. Esto es: se consulta el cliente en *BASE_DATABASE* y se obtiene su *db_name*, a continuación se asigna este nombre de colección como la base de datos a utilizar, haciendo uso del método *set_collection_name* de *MongoEngine*.
- *insert*: Se ha sobreescrito este método de *Item* para realizar comprobaciones previa inserción de nuevos clientes.
- *find*: Se ha sobreescrito para asegurar que las operaciones se hacen sobre *BASE_DATABASE*.
- *update*: Se ha sobreescrito para asegurar que las operaciones se hacen sobre *BASE_DATABASE*.
- *remove*: Se ha sobreescrito para asegurar que las operaciones se hacen sobre *BASE_DATABASE*.

Los *endpoints* desarrollados tienen la forma que se indica en el punto 7.2, pero no se han implementado todos ellos, sólo los siguientes:

- *POST /customer*
- *PUT /customer*
- *DELETE /customer*
- *POST /customer/query*

Flujo de trabajo

Como se indicaba en la sección 7.2, para controlar el cliente que se debe utilizar en cada petición se utiliza el subdominio del host de la petición. Se ha creado una función que comprueba si este subdominio si se trata de un cliente válido; en caso afirmativo se asigna como cliente para esa petición y en caso negativo se aborta la petición con un código 404.

7.4. Usuarios

Esta es la clase encargada del manejo de los usuarios y hereda de *Item*. Trabaja junto con la clase *Login* para permitir el acceso a la *API*. Los datos miembros de esta clase son:

- *table_name*: users
- *table_schema*: (Por defecto todos los valores a 1).
  - *type*: Tipo del usuario, puede ser *admin* o *regular*.
  - *first_name*: Nombre del usuario.
  - *last_name*: Apellido/s del usuario.
  - *username*: Nickname del usuario.
  - *email*: Email del usuario.
  - *password*: Contraseña del usuario.
  - *public_id*: *UUID* del usuario.

Se han sobreescrito los métodos de inserción y actualización de datos para tener en cuenta las restricciones de tipo de usuario, para la generación del *UUID* y para el cifrado de la contraseña. Este cifrado se hace con *Fernet*, el cual es de tipo simétrico.

En cuanto al servicio, este está estructurado de la misma forma que se especifica en el punto 7.2, siendo los *endpoints*:

- *GET /user/:username*
- *POST /user*
- *PUT /user*
- *DELETE /user*
- *POST /user/query*

Flujo de trabajo

Una vez disponemos tanto de la clase como del servicio sólo tenemos que crear tantos usuarios queramos o procesar y redirigir los datos que se obtengan en el servicio a esta clase.

7.5. Despliegues

Este módulo es el encargado de realizar los despliegues de los servicios mediante contenedores *Docker*. Para llevar a cabo esto se conecta a un servidor de *Docker* y contiene los métodos necesarios para ejecutar contenedores, imágenes y operaciones en ambos.

Actualmente no permite realizar algunas funciones, como son el manejo de redes, nodos, o volúmenes. Una futura mejora o ampliación del módulo podría incluir estas u otras nuevas funcionalidades. Por el momento no eran necesarias y se han priorizado los contenedores y las imágenes. Por otro lado también permite obtener información del estado del cliente, del uso de almacenamiento e información general.

Contextualización:

- En este módulo se entiende por **cliente** al conector que se crea en el sistema para comunicarnos con el *daemon* de *Docker*.
- Un **objeto** puede ser una imagen o un contenedor.
- Por **operación** se entiende toda aquella tarea que se puede ejecutar en un contenedor o en una imagen.

El principal propósito de estas operaciones es unificar la forma con la que se trabaja, en este caso, con Docker. Si en lugar de abstraer este aspecto se crearan métodos para cada una de las funcionalidades disponibles la lista de estos tendría un tamaño considerable.

Las operaciones constan de nombre, datos y, en ocasiones, de un objeto concreto:

- *operation*: Nombre de la operación.
- *data*: Datos necesarios para ejecutar la operación.
- *object*: Objeto al que se dirige la operación.

Para manejar esto se ha creado la clase *DockerEngine*, la cual se conecta al *daemon* de *Docker* e implementa los métodos necesarios para realizar las funciones anterior mencionadas. Concretamente estas son:

- *run_container_operation(operation, data)*: Ejecuta una operación en todos los contenedores.

- *run_image_operation(operation, data)*: Ejecuta una operación en todas las imágenes.
- *run_operation_in_object(object, operation, data)*: Ejecuta una operación en un objeto, contenedor o imagen.
- *get_container_by_id(container_id)*: Devuelve el contenedor denotado por *container_id*.
- *get_image_by_name(name)*: Devuelve la imagen denotada por *name*.

Los *endpoints* desarrollados son los siguientes:

- *POST /deploy/container*: Operaciones en todos los contenedores.
- *POST /deploy/container/single*: Operaciones en un único contenedor.
- *POST /deploy/image*: Operaciones en todas las imágenes.
- *POST /deploy/image/single*: Operaciones en una única imagen.

La información sobre el estado de este módulo se devuelve con la misma estructura que se utiliza en *MongoEngine*:

```
{
        "is_up": boolean,
        "data_usage": list,
        "info": dict || string
}
```

Puede ocurrir que el backend se esté ejecutando en un contenedor *Docker*, por lo que intentar conectarse al *daemon* no sería posible en ese caso. Cuando se ejecuta el backend este comprueba internamente en qué entorno se esta ejecutando y la variable que almacena la localización del *daemon* o servidor. Estas comprobaciones determinarán si este módulo se encontrará activo o no.

Flujo de trabajo

El flujo de trabajo de este módulo es algo mas complejo que los anteriores. En primer lugar se tienen que tener claro el concepto de operaciónes, tras eso solo hay que ejecutarlas.

Ejecutar una operación consiste en tomar un objeto, que puede ser una imagen, un contenedor o el conjunto de estos independientemente; elegir

lo que se quiere hacer con ese objeto (ejecutar, borrar, listar,...); agregar los parámetros necesarios (un nombre, un valor numérico,...); y, finalmente ejecutar la operación. La salida de estas operaciones se devuelve en cada método.

Un ejemplo de uso de estas operaciones podría ser:

```
# Utilizando un metodo de la clase DockerEngine
DockerEngine().run_container_operation(
        operation='run',
        data = {
                'image': 'hello-world'
        }
)

# Haciendo una peticion al servicio
post(self.URL + '/image',
        headers = self.headers,
        data = json.dumps({
                'operation': 'run',
                'data': {
                        'image': 'hello-world'
                }
        })
)
```

7.6. Aprovisionamiento

Es el encargado de aprovisionar sistemas mediante el uso de Ansible y se centra exclusivamente en la ejecución de *Playbooks* y en la gestión y almacenamiento de grupos de *hosts* y *Playbooks*.

- Un **grupo de *hosts*** es aquella máquina o grupo de máquinas que se quiere aprovisionar.
- Un ***Playbook*** es el conjunto de ordenes, comandos y tareas que se quiere que se ejecuten en un grupo de hosts.

Tanto los *hosts* como los *Playbooks* se pueden almacenar en base de datos si se desea y por tanto se han desarrollado las clases *Hosts* y *Playbook*.

7.6.1. Hosts

Clase que hereda de *Item* cuyos datos miembros son:

- *table_name*: hosts
- *table_schema*: (Por defecto todos los valores a 1).

  - *name*: Nombre del grupo de *hosts*.
  - *ips*: Lista de direcciones IP que componen el grupo de *hosts*.

Comparte los mismos *endpoints* que se indican en el punto 7.2, siendo estos:

- *GET /provision/hosts/:name*
- *POST /provision/hosts*
- *PUT /provision/hosts*
- *DELETE /provision/hosts*
- *POST /provision/hosts/query*

7.6.2. Playbook

Clase que hereda de *Item* cuyos datos miembros son:

- *table_name*: *playbooks*
- *table_schema*: (Por defecto todos los valores a 1).
  - *name*: Nombre del *Playbook*.
  - *playbook*: Contenido del *Playbook* codificado como *JSON*.

Comparte los mismos *endpoints* que se indican en el punto 7.2, siendo estos:

- *GET /provision/playbooks/:name*
- *POST /provision/playbooks*
- *PUT /provision/playbooks*
- *DELETE /provision/playbooks*
- *POST /provision/playbooks/query*

Para la ejecución de los *Playbooks* se ha desarrollado la clase *AnsibleEngine*, la cual implementa un método para este cometido, es:

- *run_playbook(hosts, playbook, passwords)*

Este método toma como entrada el grupo de *hosts* a los que se le va a ejecutar el *Playbook*, el *Playbook* en cuestión y un diccionario con las contraseñas para acceder a las máquinas. La librería de *Ansible* usada requiere que los *hosts* se pasen como un archivo de texto plano, por lo que se ha desarrollado una función auxiliar que crea un fichero cuando se ejecuta un *Playbook*. El directorio donde se guardan estos archivos puede configurarse mediante una variable de entorno.

El *endpoint* creado es el siguiente:

- *POST /provision*

Flujo de trabajo

El aprovisionamiento es un proceso que consta de tres pasos bien diferenciados:

El primero consiste en la creación de uno o varios grupos de *hosts* del modo que se ha explicado en secciones anteriores. Una vez disponemos de estos grupos de *hosts* se procede a la creación de un *Playbook*, también de igual manera a la explicada anteriormente. En los *Playbooks* se debe indicar en qué *hosts* se quieren ejecutar las órdenes de este, estos *hosts* deben estar ya creados previamente.

El tercer y último paso consiste en la ejecución del *Playbook*. Previa ejecución de este se comprueba que el *Playbook* seleccionado y que el grupo de *hosts* existen y estan almacenados en el sistema, tras eso se ejecuta.

7.7. Máquinas

Para el almacenamiento y gestión de máquinas se ha creado la clase *Machine*, la cual también hereda de *Item*. Sus datos miembros son:

- *table_name*: machines
- *table_schema*: (Por defecto todos los valores a 1).
  - *name*: Nombre de la máquina.
  - *description*: Descripción breve de la máquina.
  - *type*: Tipo de máquina, puede ser *local* o *remote*.
  - *ipv4*: Dirección IPv4 de la máquina.
  - *ipv6*: Dirección IPv6 de la máquina.
  - *mac*: MAC del adaptador de red que conecta la máquina a la red.

  - *broadcast*: Dirección broadcast de la red a la que está conectada la máquina.
  - *netmask*: Máscara de red.
  - *network*: Red a la que está conectada la máquina.

En el caso de las máquinas todas las direcciones IP que se manejan deben ser validadas, por lo que se ha creado un método auxiliar para realizar esta función. Además se han sobrescrito los métodos de inserción y actualización para realizar esta validación.

El servicio tiene la misma estructura que la explicada en el punto 7.2 y sus *endpoints* son:

- *GET /machine/:name*
- *POST /machine*
- *PUT /machine*
- *DELETE /machine*
- *POST /machine/query*

Flujo de trabajo

Este flujo de trabajo es equivalente a los que explicados en las clases y servicios comparten estructura a partir de *Item*.

7.8. Autenticación

Utilizado para permitir el uso de la *API* a los usuarios registrados. Esta autenticación se hace mediante *JWT* (*JSON Web Token*).

Para la creación de esta clase se ha partido de la clase *Item*, definiendo una nueva clase *Login* con los siguientes datos miembros:

- *table_name*: login
- *table_schema*: (Por defecto todos los valores a 1).
  - *token*: *JWT* del usuario que tenga acceso actualmente al backend.
  - *username*: Nombre de usuario.
  - *exp*: Fecha y hora a la que expira el acceso.

- *login_time*: Fecha y hora a la que el usuario realizó el acceso.
- *public_id*: *UUID* que identifica al usuario.

Flujo de trabajo

Para controlar los accesos que puedan quedar obsoletos o en los que no se haya realizado un logout correcto, cada vez que se instancia la clase se comprueba si hay *tokens* con estas características y se eliminan, evitando así una acumulación innecesaria de *tokens* sin usar.

Los métodos que se han desarrollado han sido los siguientes:

- *login(auth)*: En este método se realiza todo el proceso del *login*. Se parte de los datos de autenticación, compuestos por un usuario y una contraseña y se comprueba si tal usuario existe. Se verifica que la contraseña sea la correcta y en caso afirmativo se procede a la creación del *token*. Tanto si el usuario no estaba *logueado* previamente como si ya lo estaba, se generan todos los datos asociados nuevamente, y se insertan o actualizan. En el *token* se codifica el *public_id* y la fecha y hora de expiración, *exp*. Si el proceso ha sido correcto se devolverá el *token* creado.

- *logout(username)*: Desloguea al usuario denotado por *username*. Verifica que existe el usuario y borra cualquier tipo de información asociada de este en la colección actual.

- *token_access(token)*: Decodifica el *token* y devuelve al usuario logueado que tenga tal *public_id*.

- *get_username(token)*: Devuelve el nombre del usuario asociado al *token*.

Una vez desarrollada la clase que permite accesos de usuarios se ha desarrollado el servicio. Todas las rutas del backend, salvo *GET /login* y *GET /api/heartbeat* están protegidas con esta autenticación. Para ello se ha creado un decorador que comprueba el *token* de acceso cada vez que se quiere acceder a un *endpoint*. Es el siguiente:

- *token_required*: Obtiene el *token* de la cabecera *x-access-token*, comprueba si es válido y permite el acceso o no al *endpoint*. En caso de no ser válido se devuelve un mensaje de error y un código de error 401.

El servicio de login cuenta con dos *endpoints*, los cuales son:

- *GET /login*: Loguea al usuario, para ello toma los datos de acceso de la cabecera *Basic auth* y devuelve o no el *token* asociado al usuario.
- *GET /logout*: Desloguea al usuario que previamente debe estar *logueado*.

7.9. Estado del backend y *heartbeat*

Para consultar el estado del backend se han creado dos *endpoints*. El primero ofrece una información más detallada de los dos servicios más importantes y el segundo ofrece sólo el estado general del backend. Son:

- *GET /status*: *Endpoint* autenticado que devuelve un diccionario con los estados de *MongoDB* y *Docker*, con la forma indicada anteriormente. Los usuarios administradores obtienen más información en el caso de *MongoDB*. La estructura devuelta es:

```
{
        'mongo': {
                'is_up': bool,
                'data_usage': list,
                'info': dict || str
        },
        'docker': {
                'is_up': bool,
                'data_usage': list,
                'info': dict || str
        }
}
```

  En el caso que *Docker* no se encuentre funcionando correctamente o se encuentre desactivado el estado devuelto es:

```
{
        'status': bool,
        'disabled': bool,
        'msg': str
}
```

- *GET /api/heartbeat*: *Endpoint* no autenticado que devuelve el estado simplificado. Puede ser usado para comprobar que el backend se encuentra activo de forma manual o por ejemplo por la funcionalidad *Heartbeat* que incorpora Docker para conocer el estado de salud de los contenedores.

```
{
        'ok': bool
}
```

Flujo de trabajo

En estos casos el flujo de trabajo es muy sencillo, se toman los datos de los clientes de *MongoDB* y *Docker* y se devuelven en forma de diccionario.

7.10. *CLI* e inicialización de la base de datos

Se ha desarrollado un *CLI* interactivo que permite realizar operaciones básicas con los clientes y usuarios mediante una terminal de comandos. Para ello hace uso de las clases y métodos anterior explicados. Estas operaciones son:

- Crear, activar y desactivar clientes.
- Agregar usuarios a un cliente.

Además se ha creado un *script* que permite inicializar la base de datos, insertando un usuario administrador por defecto.

7.11. Frontend

El frontend implementado es una de las infinitas propuestas que satisfacen los requisitos del software deseados. En las siguientes secciones se detallan los módulos que se han desarrollado.

7.11.1. Servicios *HTTP*

Para la comunicación frontend-backend se han creado diferentes servicios, que de forma asíncrona realizan las peticiones *HTTP* a la *API*. Todos ellos hacen uso de *HttpClient*, que es el múdulo de *Angular* para realizar este tipo de peticiones. La estructura en general es muy similar, habiéndose creado métodos concretos para cada tipo de operación o *endpoint*.

Cada uno de estos métodos tiene la forma:

```
metodo(params: any): Observable<any> {
        return this.httpClient.put(url, {
                data
        }, {
                headers: access_token
        }).pipe(
        map(data => {
                return {
                        ok: true,
                        data
                };
        }),
        catchError(error => {
                return of({
                        ok: false,
                        error
                });
        })
        );
}
```

Se puede observar que:

- Se hace una petición *HTTP* (en el caso del ejemplo se usa *PUT*) a través del módulo *HttpClient* con diferentes argumentos. El primero es la *URL* a la que se hace esa petición, el segundo los datos que se quieren pasar en el *body* de esta y el tercero son las cabeceras, que en el caso de la *API* desarrollada es necesaria la cabecera *x-access-token* con el *token* de acceso.
- El *observable* que se devuelve es asíncrono, lo que quiere decir que se lanza la petición y que de forma asíncrona se completará la misma y se procesarán los datos.
- Se hace un *pipe*. Esto permite hacer un primer procesado de los datos y en este caso se agrega una clave que indica que se han recibido los datos.
- Se capturan los posibles errores. En caso de que la petición no se complete o surja algún tipo de error se obtiene el error y se devuelve.

Los servicios desarrollados son los siguientes:

URL

Este servicio no es un servicio como tal, ya que no realiza peticiones *HTTP*, pero sí se encarga del procesado de la *URL* a la que se hacen las

peticiones. Es usado por el resto de servicios para obtener la *URL* a la que tienen que hacer dichas peticiones. El procesado consiste en la creación de la *URL* a partir del protocolo de acceso a la *API* (*HTTP* o *HTTPS*), del cliente y de la *URL* donde se encuentra la *API*.

De este modo el resto de servicios solo tienen que hacer una llamada a este servicio para obtener la *URL* actual.

Autenticación

La autenticación es el módulo más importante del frontend, ya que se encarga de toda la gestión del *token* y del acceso de los usuarios a las diferentes rutas de la aplicación. Este servicio se compone de dos métodos principales, *login* y *logout*, y de algunos secundarios. El funcionamiento de estos es:

- *login(auth)*: A partir de los datos de autenticación del usuario realiza el *login* y se almacena en las *cookies* del navegador el *token* de acceso. Se comprueba si existe el *token* en las *cookies*: en caso afirmativo se comprueba si es válido y autoriza o no el acceso; en caso contrario se hace una petición al *endpoint GET /login* con los datos de acceso para obtener un nuevo *token*. Finalmente se almacena el *token* en las *cookies*. Debido a la asincronía de las peticiones este método también devuelve un observable.
- *logout()*: Hace una petición de logout a la *API* y cuando obtiene la respuesta borra el *token* de las *cookies* del navegador.

Resto de servicios

Los demás servicios tienen la estructura ya explicada al principio de esta sección, con métodos para hacer peticiones a los diferentes *endpoints* de la *API* y manejo de los posibles errores. Cuando ha sido preciso se han desarrollado también métodos auxiliares. Estos servicios son:

- Clientes
- Usuarios
- Hosts
- Playbooks
- Máquinas
- Aprovisionamiento

- Despliegues
- Estado del backend

7.11.2. Guard

Para proteger las rutas de usuarios no identificados se ha creado un *Guard*. Este realiza una serie de comprobaciones acordes a nuestras necesidades antes de permitir o no el acceso a una página. En el caso de este frontend la única ruta que se quiere accesible por cualquier usuario es el *login*, por lo que el resto se han protegido.

Se ha hecho uso del método *CanActivate* que proporciona *Angular* para este cometido. Además, se han tenido en cuenta diferentes aspectos a la hora de diseñar y desarrollar esta funcionalidad, ya que por ejemplo no se debería poder acceder a los despliegues con *Docker* si este esta desactivado en el backend.

Hace uso del *router* que también proporciona *Angular* del servicio de autenticación y su funcionamiento es el siguiente:

- Si se quiere ir al *login* se permite el acceso.
- En caso contrario se comprueba si el usuario está *logeado*.
    - Si lo está:
        - Si se quiere acceder a la página de administración de usuarios se comprueba el usuario que está intentando acceder y se le permite o no el acceso.
        - Si se quiere acceder a la página de los despliegues se comprueba si *Docker* está activo en el backend.
    - Si no lo está:
        - Se le redirige al *login*.

Una vez implementado esto solo ha sido necesario indicar en el *router* las rutas que se quieren proteger.

7.11.3. Variables de entorno

Para que la comunicación frontend-backend pueda llevarse a cabo es necesario definir una serie de variables de entorno. Estas indican la *URL* en la que se encuentra la *API*, entre otros. Son:

- *production*: Utilizada por *Angular* a la hora de construir la aplicación.

- *backendUrl*: *URL* sin protocolo de la *API*.
- *httpsEnabled*: Indica si el protocolo de la *API* es *HTTP* o *HTTPS*.

Actualmente el proyecto cuenta con cuatro entornos distintos. El primero es el de desarrollo, utilizado durante el desarrollo del proyecto. El segundo es el de producción, el cual es el que se debe usar a la hora de utilizar el proyecto en producción. El tercero, llamado *on-premise* es el usado a la hora de construir la imagen de *Docker* y está configurado por defecto para funcionar *out-of-the-box* con el *docker-compose*. El último, que se encuentra en el archivo *env.js* en la raíz del directorio *src* del proyecto, permite redefinir las variables de entorno sin tener que reconstruir el backend. Por defecto se encuentra desactivado.

7.11.4. Interfaces

Se han creado además una serie de interfaces para controlar más aún los datos que se manejan en el frontend. Estas contemplan cada tipo de dato y son:

- *AccessToken*: *Token* de acceso.
- *BasicAuth*: Credenciales de usuario.
- *Container*: Contenedor, usado en el módulo de despliegues.
- *SingleContainerOperation*: Operación que se ejecuta en un único contenedor.
- *ContainerOperation*: Operación que se ejecuta en todos los contenedores.
- *Image*: Imagen, usada en el módulo de despliegues.
- *SingleImageOperation*: Operación que se ejecuta en una única imagen.
- *ImageOperation*: Operación que se ejecuta en todas las imágenes.
- *Customer*: Cliente.
- *DockerHubImage*: Imagen de *DockerHub*, usada al utilizar la operación de búsqueda de imágenes.
- *Host*: Grupo de *hosts*.
- *Machine*: Máquina.

- *Playbook*: *Playbook*.
- *Query*: Criterio de búsqueda, usado en todos los *endpoints* para hacer consultas complejas.
- *StatusResponse*: Estado del backend.
- *User*: Usuario.

7.11.5. Componentes

En cuanto a componentes se ha intentado abstraer y reutilizar lo máximo posible.

Tabla

Usado para crear tablas dinámicas sin necesidad de configuraciones exhaustivas. Sus funcionalidades extra son una barra de búsqueda de elementos, un selector de las columnas que se muestran, una barra inferior para paginación y la posibilidad de agregar un botón de acción personalizado para cada *item* de la tabla.

Parámetros de entrada:

- *title*: Título de la tabla.
- *displayedColumns*: Columnas que se quieren mostrar.
- *deselectedColumns*: Columas que por defecto no aparecerán mostradas.
- *data*: *Array* con los datos a mostrar.
- *actions*: *Array* de strings para configurar las acciones disponibles para cada item.
- *customActionData*: Diccionario con la configuración de la acción personalizada adicional.

Por defecto cada item tiene cuatro acciones asociadas: *play* (P), *detail* (D), *edit* (E) y *remove* (R). Estas se activan mediante el array *actions*, agregando los idenficativos de las acciones que se deseen a este. En cambio, para configurar la acción adicional se debe incluir en *customActionData* el icono y el tooltip a mostrar y agregar el identificativo 'C' a *actions*.

Estas acciones son botones que se agregan en una columna de la tabla y cuando estos se pulsan emiten un evento con los datos del item al que pertenezcan. De este modo solo queda implementar la lógica en el componente que esté usando la tabla para que tome esos datos y los procese a voluntad.

Un ejemplo de uso de esta tabla sería el siguiente:

```
<app-ipmtable
        *ngIf="data"
        [displayedColumns]="displayedColumns"
        [deselectedColumns]="['id']"
        [actions]="['P','C','R']"
        [data]="data"
        (playCallback)="runImage($event)"
        (removeCallback)="removeImage($event)"
        (customActionCallback)="manageImage($event)"
        [customActionData]="manageContainerCustomActionData"
>
```

Parámetros de salida. Emiten un evento con el *item* correspondiente a la acción que se ha presionado.

- *playCallback*
- *detailCallback*
- *editCallback*
- *removeCallback*
- *customActionCallback*

Navegador superior

Se trata de una barra de navegación superior que cuenta con diferentes botones para navegar por las diferentes páginas. Cuenta con lógica interna para desactivar o no el enlace a la página de despliegues en caso de que *Docker* no se encuentre activado y para mostrar o no el enlace a la administración de usuarios.

Los enlaces con los que cuenta son:

- Home
- Despliegues
- Aprovisionamiento
- Máquinas

- Administración de usuarios
- Logout

Home (/)

Este componente muestra tarjetas con información sobre el estado del backend. Esta información depende del usuario que se encuentre visitándola, ya que un administrador recibirá más información que un usuario regular. Por otro lado también se adapta a la disponibilidad de *Docker* en el backend.

Login (/login)

Componente bastante sencillo que cuenta con un formulario con tres campos. El primero es opcional y sirve para indicar el cliente al que se quiere conectar el usuario. El segundo y tercer campo son su usuario y contraseña.

Admin (/admin)

Administración de usuarios. Este componente solo es accesible por usuarios administradores. Hace uso de la tabla anterior para mostrar los usuarios que se encuentran registrados en el cliente actual y permite operar con ellos. Se ha creado un diálogo anexo a este componente para crear y editar estos usuarios y se puede acceder a él mediante las acciones de la tabla.

Despliegues (/deploy)

Este componente se compone de dos pestañas, *Containers* e *Images*.

- Containers: En esta primera pestaña se muestran los contenedores que se encuentran en el backend (en cualquier estado) y se pueden administrar. Esta administración consiste en un diálogo en el que se pueden ejecutar diferentes operaciones sobre cada contenedor, son:
- Pausar y despausar
- Recargar
- Reiniciar
- Parar
- Forzar parada
- Cambiar nombre

- Obtener logs

Además cuenta con dos botones específicos para este componente en la esquina superior derecha. Estos son para ejecutar imágenes, lo que nos lleva a la segunda pestaña, y para eliminar los contenedores obsoletos (mediante una operación).

- *Images*: Aquí aparecen las imágenes que se encuentran también en el backend. Cada una de estas imágenes se puede ejecutar, administrar (recargar y obtener historial) y borrar del backend. Los botones de la esquina superior derecha son para hacer búsquedas de imágenes en *DockerHub* y para eliminar aquellas imágenes obsoletas del sistema.

Aprovisionamiento (/provision)

En el caso del aprovisionamiento este componente cuenta con tres pestañas, las cuales son:

- *Playbooks*: Gestión de los *Playbooks*. Se pueden ejecutar, modificar y eliminar. Además cuenta con un botón para crear nuevos *Playbooks*.

- *Editor*. Editor de texto configurado con la sintaxis *YAML* para crear y modificar los diferentes *Playbooks*. Esta pestaña se encuentra enlazada con la primera ya que cuando se crea un *Playbooks* nuevo o se quiere modificar uno existente se redirige al usuario aquí.

- *Host groups*: Gestión de los grupos de hosts de manera similar a los clientes o usuarios. Se encuentra enlazado con el servicio de máquinas para obtener las direcciones IP de las máquinas que se encuentran almacenadas en el sistema.

Del mismo modo que en otros componentes, se hace uso de diálogos para la gestión de los datos.

Máquinas (/machines)

Este componente es muy similar a *Admin*. Cuenta con una tabla para mostrar los datos de todas las máquinas actuales y además se ha creado un diálogo (también accesible por las acciones) para crear y modificar los datos de las máquinas.

AreYouSureDialog

En ocasiones es necesaria es necesaria la confirmación del usuario a la hora de realizar una acción. Se ha creado este diálogo para tener un componente común a todas estas confirmaciones. Al cerrarse emite un valor (*true* o *false*) que indica la decisión del usuario.

Flujo de trabajo

En el caso del frontend se pueden diferenciar cuatro flujos distintos:

- **Administración de usuarios**. En esta página (*/admin*), al igual que en todas las que provienen de *Item*, se ofrece una tabla en la que se permite realizar una administración de los datos que se muestran. El usuario puede consultar, crear, eliminar y modificar usuarios.
- **Administración de máquinas**. Flujo equivalente al punto anterior. En este caso la página donde se encuentra es */machines*.
- **Despliegue de servicios**. El flujo es algo mas complejo que el anterior. El ideal sería que el usuario se dirigiera a esta página y una vez dentro de ella navegara por las dos pestañas que se le ofrecen. En la primera puede consultar los contenedores que se están ejecutando y en la segunda las imágenes disponibles. El usuario podría ahora realizar las diferentes opciones que se le ofrecen, como buscar, descargar y ejecutar imágenes o administrar los contenedores en ejecución.
- **Ejecución de *Playbooks***. Página */playbooks* El flujo ideal comienza creando máquinas y guardando sus configuraciones. Una dirección IPv4 válida es necesaria para crear un grupo de *hosts*, por lo que es mandatorio que al menos exista una máquina almacenada en el sistema. Tras esto se procedería a crear un grupo de *hosts* con tantas direcciones IPv4 como se desee. El siguiente paso sería crear un *Playbook* y para ello se utiliza el editor de texto existente. Una vez se finaliza y se guarda el *Playbook* se le asigna un grupo de *hosts* previamente creado. Finalmente, una vez almacenado este *Playbook* se puede ejecutar con la acción que aparece junto a los datos de este en la tabla que se muestra.

Capítulo 8

Conclusiones y trabajos futuros

Tras el desarrollo de este proyecto se puede comprobar que los objetivos que se marcaron al inicio se han alcanzado exitosamente.

IPManager es una solución que responde al problema que se plantea ya que unifica los procesos de aprovisionado de sistemas, despliegue de servicios y almacenamiento de configuraciones de una manera sencilla y modular. Además, basada en una arquitectura de microservicios, provee de un backend y un frontend que se pueden desplegar de forma independiente el uno del otro.

Es también una solución liviana y no intrusiva ya que se puede instalar en sistemas con pocos recursos de manera local o a través de contenedores *Docker*, o incluso se puede desplegar en sistemas *Cloud*.

Ofrece tambien una *API REST* que posibilita las comunicaciones con otros sistemas. Un ejemplo de este uso es **IPMDroid**, una aplicación móvil para Android que se ha desarrollado de forma paralela a este proyecto para la asignatura "Programación de Dispositivos Móviles". Esta ofrece una interfaz sencilla para la administración de configuraciones y despliegue de servicios y se puede encontrar en este repositorio [35] en *Github*.

Además, todo el proyecto está compuesto por software libre bajo la licencia GNU GPLv3 y se encuentra en este repositorio [33].

Por último, y no por ello menos importante, se ha aprendido a desarrollar un proyecto de grandes dimensiones desde las etapas más tempranas, como son las primeras investigaciones e ideas, hasta las finales, dejando la puerta abierta a posibles mejoras y actualizaciones.

En cuanto a trabajos futuros hay ciertos aspectos que se podrían mejorar:

- Actualmente el módulo para despliegues de servicios solo permite realizar operaciones con imágenes y contenedores, por lo que una mejora sería integrar el manejo de redes, volúmenes u otras configuraciones relacionadas con *Docker*.
- Se podrían ampliar los datos que se almacenan en el módulo de configuraciones. Las actuales se centran en configuraciones de red, pero podrían ampliarse para almacenar cualquier otro aspecto.
- El módulo de aprovisionamiento se centra exclusivamente en la ejecución de *Playbooks*. Una mejora en este aspecto podría ser un mejor manejo de los parámetros de ejecución de estos.
- El frontend desarrollado intenta ser sencillo a la par que funcional, pero hay características que se podrían mejorar. Una mejora podría ser las vistas individuales de las configuraciones de las máquinas que se almacenan.

Apéndice A

Especificación de *endpoints*

Leyenda:

- **E/S**: Parámetro de **E**ntrada o **S**alida.
- **Opcional**
  - **No**. Parámetro necesario por el endpoint.
  - **Sí**. Parámetro opcional.

A.1. Status

A.1.1. *GET /status*

Devuelve el estado actual de los dos servicios principales del backend, *MongoDB* y *Docker*.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Respuesta:

Parámetro	Key	Tipo	Descripción
mongo		dict	
	is_up	bool	Si el servicio se encuentra activo o no.
	data_usage	list[dict]	Información de uso de datos de cada base de datos.
	info	string	Información adicional del cliente de MongoDB.
docker		dict	
	is_up	bool	Si el servicio se encuentra activo o no.
	data_usage	dict	Información de uso de las imágenes y contenedores.
	info	string	Información adicional del cliente de Docker.

En caso de que el servicio de *Docker* no se encuentre activado o presenta errores los *endpoints* correspondientes al servicio de despliegue no estarán disponibles y el estado de este en la respuesta anterior tendrá la siguiente forma:

Parámetro	Key	Tipo	Descripción
docker		dict	
	status	bool	False. El servicio no funcina correctamente.
	disabled	bool	Si el servicio se encuentra desactivado o no.
	msg	string	Información adicional.

A.2. Heartbeat

A.2.1. *GET /api/heartbeat*

Devuelve el estado actual de los dos servicios principales del backend de forma simplificada.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si los servicios del sistema funcionan correctamente o no.

A.3. Autenticación

A.3.1. *GET /login*

Loguea al usuario en el cliente.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Respuesta:

Parámetro	Tipo	Descripción
token	string	Token JWT utilizado para identificar al usuario.

A.3.2. *POST /logout*

Desloguea al usuario del cliente.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.4. Clientes

A.4.1. Cliente

Parámetro	Tipo	Opcional	E/S	Descripción
domain	string	No	E/S	Subdominio del cliente.
db_name	string	No	E/S	Base de datos del cliente.

A.4.2. *POST /customer/query*

Devuelve los clientes que cumplen los criterios de búsqueda.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
query	dict	No	Criterio de búsqueda.
filter	dict	Sí	Parámetros que se quieren en la respuesta.

Respuesta (un solo usuario):

Parámetro	Tipo	Descripción
data	Cliente	Cliente que cumple el criterio de búsqueda.

Respuesta (más de un usuario):

Parámetro	Tipo	Descripción
total	int	Número de clientes que cumplen el criterio de búsqueda.
items	list[Cliente]	Clientes que cumplen el criterio de búsqueda.

A.4.3. *POST /customer*

Crea un nuevo cliente.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*): Cliente

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.4.4. *PUT /customer*

Modifica los datos de un cliente.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
domain	string	No	Subominio del cliente que se quiere modificar.
data	Cliente	No	Nuevos datos del cliente.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.4.5. *DELETE /customer*

Elimina un cliente.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
domain	string	No	Subdominio del cliente que se quiere eliminar.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.5. Usuarios

A.5.1. Usuario

Parámetro	Tipo	Opcional	E/S	Descripción
type	string	No	E/S	Tipo de usuario.
public_id	string	-	S	UUID del usuario.
first_name	string	No	E/S	Nombre del usuario.
last_name	string	No	E/S	Apellido del usuario.
username	string	No	E/S	Nickname del usuario.
email	string	No	E/S	Email del usuario.
password	string	No	E	Contraseña del usuario.

A.5.2. *GET /user/:username*

Devuelve toda la información asociada a un usuario.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Parámetros de la URL:

Nombre	Opcional	Descripción
username	No	Nombre del usuario a consultar.

Respuesta:

Parámetro	Tipo	Descripción
data	Usuario	Diccionario con toda la información del usuario consultado.

A.5.3. *POST /user/query*

Devuelve los usuarios que cumplen los criterios de búsqueda.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
query	dict	No	Criterio de búsqueda.
filter	dict	Sí	Parámetros que se quieren en la respuesta.

Respuesta (un solo usuario):

Parámetro	Tipo	Descripción
data	Usuario	Usuario que cumple el criterio de búsqueda.

Respuesta (más de un usuario):

Parámetro	Tipo	Descripción
total	int	Número de usuarios que cumplen el criterio de búsqueda.
items	list[Usuario]	Usuarios que cumplen el criterio de búsqueda.

A.5.4. *POST /user*

Crea un nuevo usuario.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*): Usuario

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.5.5. *PUT /user*

Modifica los datos de un usuario.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
email	string	No	Email del usuario que se quiere modificar.
data	Usuario	No	Nuevos datos del usuario.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.5.6. *DELETE /user*

Elimina un usuario.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
email	string	No	Email del usuario que se quiere eliminar.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.6. Máquinas

A.6.1. Machine

Parámetro	Tipo	Opcional	E/S	Descripción
name	string	No	E/S	Tipo de usuario.
description	string	Sí	E/S	Descripción de la máquina.
type	string	No	E/S	Tipo de la máquina.
ipv4	string	Sí	E/S	Dirección IPv4 de la máquina.
ipv6	string	Sí	E/S	Dirección IPv6 de la máquina.
mac	string	Sí	E/S	Dirección MAC de la máquina.
broadcast	string	Sí	E/S	Broadcast de la red a la que se conecta la máquina.
gateway	string	Sí	E/S	Gateway de la red a la que se conecta la máquina.
netmask	string	Sí	E/S	Netmask de la red a la que se conecta la máquina.
network	string	Sí	E/S	Network de la red a la que se conecta la máquina.

A.6.2. *GET /machine/:name*

Devuelve toda la información asociada a una máquina.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Parámetros de la URL:

Nombre	Opcional	Descripción
name	No	Nombre de la máquina a consultar.

Respuesta:

Parámetro	Tipo	Descripción
data	Machine	Diccionario con toda la información de la máquina consultada.

A.6.3. *POST/machine/query*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
query	dict	No	Criterio de búsqueda.
filter	dict	Sí	Parámetros que se quieren en la respuesta.

Respuesta (una sola máquina):

Parámetro	Tipo	Descripción
data	Machine	Máquina que cumple el criterio de búsqueda.

Respuesta (más de una máquina):

Parámetro	Tipo	Descripción
total	int	Número de máquinas que cumplen el criterio de búsqueda.
items	list[Machine]	Máquinas que cumplen el criterio de búsqueda.

A.6.4. *POST /machine*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*): Machine

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.6.5. *PUT /machine*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre de la máquina que se quiere modificar.
data	Machine	No	Nuevos datos de la máquina.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.6.6. *DELETE /machine*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre de la máquina que se quiere eliminar.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.7. Grupos de hosts

A.7.1. *Hosts*

Parámetro	Tipo	Opcional	E/S	Descripción
name	string	No	E/S	Nombre del grupo de hosts.
ips	list[string]	No	E/S	Direcciones IPv4.

A.7.2. *GET /provision/hosts/:name*

Devuelve toda la información asociada a un grupo de *hosts*.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Parámetros de la URL:

Nombre	Opcional	Descripción
name	No	Nombre del grupo de hosts a consultar.

Respuesta:

Parámetro	Tipo	Descripción
data	Hosts	Diccionario con toda la información del grupo de hosts consultado.

A.7.3. *POST/provision/hosts/query*

Devuelve los grupos de *hosts* que cumplen los criterios de búsqueda.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
query	dict	No	Criterio de búsqueda.
filter	dict	Sí	Parámetros que se quieren en la respuesta.

Respuesta (una sola máquina):

Parámetro	Tipo	Descripción
data	Hosts	Grupo de hosts que cumple el criterio de búsqueda.

Respuesta (más de una máquina):

Parámetro	Tipo	Descripción
total	int	Número de grupos de hosts que cumplen el criterio de búsqueda.
items	list[Hosts]	Grupos de hosts que cumplen el criterio de búsqueda.

A.7.4. *POST /provision/hosts*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*): Hosts

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.7.5. *PUT /provision/hosts*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre del grupo de hosts que se quiere modificar.
data	Hosts	No	Nuevos datos del grupo de hosts.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.7.6. *DELETE /provision/hosts*

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre del grupo de hosts que se quiere eliminar.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.8. *Playbooks*

A.8.1. Playbook

Parámetro	Tipo	Opcional	E/S	Descripción
name	string	No	E/S	Nombre del Playbook.
playbook	dict	No	E/S	Contenido del Playbook codificado como *JSON*.

A.8.2. *GET /provision/playbook/:name*

Devuelve toda la información asociada a un Playbook.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Parámetros de la URL:

Nombre	Opcional	Descripción
name	No	Nombre del *Playbook* a consultar.

Respuesta:

Parámetro	Tipo	Descripción
data	Hosts	Diccionario con toda la información del *Playbook* consultado.

A.8.3. *POST/provision/playbook/query*

Devuelve los *Playbooks* que cumplen los criterios de búsqueda.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
query	dict	No	Criterio de búsqueda.
filter	dict	Sí	Parámetros que se quieren en la respuesta.

Respuesta (un solo *Playbook*):

Parámetro	Tipo	Descripción
data	Playbook	*Playbook* que cumple el criterio de búsqueda.

Respuesta (más de un *Playbook*):

Parámetro	Tipo	Descripción
total	int	Número de Playbooks que cumplen el criterio de búsqueda.
items	list[Playbook]	*Playbooks* que cumplen el criterio de búsqueda.

A.8.4. *POST /provision/playbook*

Crea un nuevo *Playbook*.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*): Playbook

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.8.5. *PUT /provision/playbook*

Modifica los datos de un *Playbook*.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre del grupo de hosts que se quiere modificar.
data	Playbook	No	Nuevos datos del *Playbook*.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.8.6. *DELETE /provision/playbook*

Elimina un *Playbook*.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre del *Playbook* que se quiere eliminar.

Respuesta:

Parámetro	Tipo	Descripción
ok	bool	Si la operación se ha ejecutado correctamente o no.
message	string	Mensaje complementario al estado de la operación.

A.9. Aprovisionamiento

A.9.1. *POST /provision*

Ejecuta un *Playbook*.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Key	Tipo	Opcional	Descripción
hosts		list[string]	No	Lista de grupo de hosts donde se quiere ejecutar el *Playbook*.
playbook		string	No	Nombre del *Playbook* a ejecutar.
passwords		dict	No	Contraseñas necesarias para la conexión a los *hosts*.
	conn_pass	string	Sí	Contraseña de acceso.
	become_pass	string	Sí	Contraseña para acceder al root.

Respuesta:

Parámetro	Tipo	Descripción
result	string	Respuesta de la ejecución del *Playbook*.

A.10. Despliegue

A.10.1. Contenedor

Parámetro	Tipo	E/S	Descripción
id	string	S	Identificador del contenedor
short_id	string	S	Identificador del contenedor truncado a 10 carácteres.
name	string	S	Nombre del contenedor.
labels	dict	S	Etiquetas del contenedor.
status	string	S	Estado del contenedor.
image	Image	S	Imagen que se esta ejecutando en el contenedor.

A.10.2. Imagen

Parámetro	Tipo	E/S	Descripción
id	string	S	Identificador de la imagen.
labels	dict	S	Etiquetas de la imagen.
short_id	string	S	Identificador de la imagen truncado a 10 carácteres.
tags	list[string]	S	Tags de la imagen.

A.10.3. Filtro de contenedores

Parámetro	Key	Tipo	Opcional	Descripción
filters		dict	No	
	exited	boolean	Sí	Si el contenedor ha finalizado su ejecución o no.
	status	string	Sí	Estado del contenedor.
	id	string	Sí	Identificador del contenedor.
	name	string	Sí	Nombre del contenedor.

A.10.4. Filtro de imágenes

Parámetro	Key	Tipo	Opcional	Descripción
filters		dict	No	
	dangling	boolean	Sí	Si la imagen se encuentra colgada o no.
	label	string	Sí	Etiqueta de la imagen.

A.10.5. *POST /deploy/container*

Permite ejecutar operaciones básicas en todos los contenedores.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
operation	string	No	Nombre de la operación que se quiere ejecutar. Valores posibles: *run*, *get*, *list*, *prune*.
data	dict	No	Argumentos de la operación.

run

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	image	string	No	Imagen a ejecutar.
	command	list[string]	Sí	Comando a ejecutar en el contenedor.
	auto_remove	bool	Sí	Eliminar o no el contenedor al terminar la ejecución.
	detach	bool	Sí	Ejecutar o no el contenedor en segundo plano. Por defecto *true*.
	entrypoint	list[string]	Sí	Entrypoint del contenedor.
	environment	dict	Sí	Variables de entorno.
	hostname	string	Sí	Hostname del contenedor.
	mounts	list[string]	Sí	Lista de volumenes que se montan en el contenedor.
	name	string	Sí	Nombre del contenedor.
	network	string	Sí	Nombre de la red a la que se conecta.
	ports	dict	Sí	Puertos a enlazar.
	user	string	Sí	Usuario para ejecutar los posibles comandos dentro del contenedor.
	volumes	dict	Sí	Volumenes a montar.
	working_dir	string	Sí	Directorio de trabajo.
	remove	bool	Sí	Eliminar el contenedor al terminar la ejecución.

get

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	container_id	string	No	Identificador del contenedor

Respuesta: Contenedor

list

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	all	bool	No	Mostrar o no todos los contenedores (en ejecución y detenidos o finalizados).
	since	string	No	Mostrar contenedores creados desde este identificador.
	before	string	No	Mostrar contenedores creados previos a este identificador.
	filters	Filtro	No	Filtros para afinar la búsqueda.

Respuesta (un solo contenedor):

Parámetro	Tipo	Descripción
data	Contenedor	Contenedor que cumple el criterio de búsqueda.

Respuesta (más de un contenedor):

Parámetro	Tipo	Descripción
total	int	Número de contenedores que cumplen el criterio de búsqueda.
items	list[Contenedor]	Contenedores que cumplen el criterio de búsqueda.

prune

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	filters	Filtro	Sí	Filtros.

A.10.6. *POST /deploy/container/single*

Permite ejecutar operaciones básicas en un contenedor en concreto.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
container_id	string	No	Identificador del contenedor.
operation	string	No	Nombre de la operación que se quiere ejecutar. Valores posibles: *kill*, *logs*, *pause*, *reload*, *rename*, *restart*, *stop*, *unpause*.
data	dict	No	Argumentos de la operación.

rename

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	name	string	No	Nuevo nombre del contenedor.

logs

Respuesta:

Parámetro	Tipo	Descripción
data	string	Logs del contenedor.

A.10.7. *POST /deploy/image*

Permite ejecutar operaciones básicas en todos las imágenes.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
operation	string	No	Nombre de la operación que se quiere ejecutar. Valores posibles: *list*, *get*, *prune*, *pull*, *remove*, *search*.
data	dict	No	Argumentos de la operación.

list

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	name	string	Sí	Mostrar sólo las imágenes pertenecientes a este repositorio.
	all	bool	Sí	Mostrar todas las imágenes o no (incluídas las imágenes de capas intermedias).
	filters	Filtro	Sí	Filtros adicionales.

Respuesta (una sola imagen):

Parámetro	Tipo	Descripción
data	Imagen	Diccionario con los datos de la imagen.

Respuesta (más de una imagen):

Parámetro	Tipo	Descripción
total	int	Número de imágenes listadas.
items	list[Imagen]	Imágenes listadas.

get

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	name	string	No	Nombre de la imagen.

Respuesta: Imagen

prune

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	filters	Filtro	Sí	Filtros adicionales.

pull

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	repository	string	Sí	Repositorio e imagen a descargar.
	tag	string	Sí	Tag de la imagen.
	auth_config	dict	Sí	Sobreescribir las credenciales.
	platform	string	Sí	Plataforma en formato: os[/arch[/variant]]

Respuesta: Imagen descargada.

remove

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	image	string	No	Imagen a eliminar.
	force	bool	Sí	Forzar borrado.
	noprune	bool	Sí	Borrar o no imágenes padre sin tag.

search

Parámetro	Key	Tipo	Opcional	Descripción
data		dict	No	Argumentos de la operación.
	term	string	No	Término de búsqueda.

Respuesta:

Parámetro	Key	Tipo	Descripción
total		int	Número de imágenes encontradas.
items		list[dict]	Imágenes encontradas.
	star_count	int	Número de estrellas en DockerHub.
	is_official	bool	Si la imagen es oficial o no.
	name	string	Nombre de la imagen
	is_automated	bool	Imagen automatizada.
	description	string	Descripción.

A.10.8. *POST /deploy/image/single*

Permite ejecutar operaciones básicas en una imagen en concreto.

Cabeceras necesarias:

Nombre	Opcional	Descripción
x-access-token	No	Token de acceso.

Cuerpo de la petición (*JSON*):

Parámetro	Tipo	Opcional	Descripción
name	string	No	Nombre de la imagen.
operation	string	No	Nombre de la operación que se quiere ejecutar. Valores posibles: *history*, *reload*.
data	dict	No	Argumentos de la operación.

history

Respuesta:

Parámetro	Tipo	Descripción
data	string	Historia de la imagen.

reload

Recarga la imagen y aplica los posibles cambios que tenga.

Apéndice B

Variables de entorno

IPManager necesita algunas variables de entorno para funcionar correctamente. Todas tienen un valor por defecto, aunque se recomienda que se revisen y se modifiquen antes de comenzar a usar tanto backend como frontend.

B.1. Backend

Estas variables se pueden configurar como variables de entorno o se pueden configurar directamente en config/server_environment.py.

- MONGO_HOSTNAME. Hostname donde se encuentra el host de *MongoDB*. Por defecto 127.0.0.1.
- MONGO_PORT. Puerto del host de *MongoDB*. Por defecto 27017.
- TESTING_DATABASE. Nombre de la colección usada para ejecutar los tests unitarios. Por defecto ipm_root_testing.
- BASE_DATABASE. Nombre de la colección base de **IPManager**. En esta colección se almacenan datos de los clientes del backend. Por defecto toma el valor que tenga la variable TESTING_COLLECTION.
- ENC_KEY. Clave usada para encriptar las contraseñas. Puedes generar una clave ejecutando el archivo generate_key.py que se encuentra en utils. Por defecto se usa una aleatoria.
- JWT_ENC_KEY. Clave usada para encriptar los *JWT* usados en el login. Puedes generar una clave ejecutando el archivo generate_key.py que se encuentra en el directorio *src/utils*. Por defecto se usa una aleatoria.

- DOCKER_BASE_URL. *URL* o *path* donde se encuentra el socket de *Docker*. Por defecto unix://var/run/docker.sock.
- DOCKER_ENABLED. Activa o desactiva los endpoints para gestión de servicios. Por defecto comprueba si el backend se está ejecutando en un *Docker* para desactivarlos en caso afirmativo.
- ANSIBLE_PATH. *Path* relativo o absoluto donde se van a almacenar los diferentes archivos generados por el backend necesarios para el aprovisionamiento. Por defecto ./.

B.2. Frontend

El frontend cuenta con dos variables de entorno que se tienen que configurar previamente, son:

- backendUrl. *URL* del backend sin protocolo.
- httpsEnabled. *True* si el backend cuenta con *HTTPS*, *false* en caso contrario.

La configuración de estas variables se puede hacer de dos maneras:

- En los archivos environment.*.ts que se encuentran en fronend/src/environments. Estos archivos una vez configurados son inmutables una vez se ha construido el frontend.
  - environment.on-premise.ts. Entorno utilizado para construir la imagen de *Docker*. Por defecto se utilizan los valores usados en el *docker-compose*.
  - environment.prod.ts. Entorno utilizado para construir una versión de producción.
- En el archivo env.js. Este archivo se encuentra en fronend/src y se puede modificar sin tener que reconstruir el frontend. En caso de no necesitar esta característica se recomienda utilizar el environment anterior. El archivo tiene la siguiente forma y se deben descomentar las líneas 2 a 5 para su uso.

```
1 (function (window) {
2 // window.__env = {
3 //   backendUrl: '172.20.0.3:5000',
4 //   httpsEnabled: false
5 // };
6 }(this));
```

Apéndice C

Instalación del sistema

Una vez clonado el repositorio se puede instalar y ejecutar tanto backend como frontend siguiendo los pasos que se describen a continuación.

C.1. Backend

El backend de **IPManager** tiene algunas dependencias que se tienen que instalar para que funcione correctamente, son las siguientes:

```
apt install sshpass
apt instal openssl
apt install libffi6

pip3 install -r requirements.txt
```

La versión instalada del paquete *werkzeug* debe ser 0.16.1.

Se puede ejecutar con *Flask* o *Gunicorn*.

- Flask:

```
export FLASK_APP=wsgi.py

flask run
```

- Gunicorn:

```
gunicorn -b 0.0.0.0:5000 wsgi:app
```

C.1.1. *Docker*

El backend también está disponible en *Docker*, la imagen puede descargarse de la siguiente manera:

```
docker pull harvestcore/ipm-backend:<tag>
```

Se recomienda siempre utilizar la última versión disponible de la imagen, la cual puede consultarse en la sección *Releases* del repositorio en *GitHub* [34] o en la sección *tags* del repositorio en *DockerHub* [31]. Deben comprobarse también las variables de entorno necesarias para ejecutar el backend.

Ejemplo de ejecución:

```
docker run -e MONGO_HOSTNAME=172.20.0.2 harvestcore/ipm-backend:<tag>
```

En el caso de querer construir la imagen se debe ejecutar:

```
cd backend

docker build . -t ipm-backend:<tag>
```

C.2. Frontend

Para instalar el frontend se deben revisar y configurar las variables de entorno, tras eso se debe ejecutar lo siguiente:

```
cd frontend

npm build --prod
```

Para ejecutarlo se recomienda utilizar *Nginx* u otro tipo de servidor web. En la raíz del frontend se adjunta el archivo de configuración (*nginx.conf*) usado para construir la imagen de *Docker*, y tambien puede ser usado en este caso.

C.2.1. *Docker*

Se puede ejecutar el frontend con *Docker*, para ello se puede descargar la imagen del repositorio disponible o se puede construir de forma local.

```
docker pull harvestcore/ipm-frontend:<tag>
```

El tag o versión se puede consultar en la sección *Releases* del repositorio en *GitHub* [34] o en la sección *tags* del repositorio en *DockerHub* [32]. Se recomienda usar siempre la última versión estable.

```
cd frontend

// Construir imagen
docker build . -t ipm-frontend:<tag>

docker run ipm-frontend:<tag>
```

C.3. Docker-compose

En el caso de utilizar el *docker-compose* que se encuentra en la raíz del repositorio solo es necesario ejecutar lo siguiente:

```
docker-compose build

docker-compose up
```

Por supuesto se pueden agregar variables de entorno para configurar el backend. Un ejemplo sería:

```
docker-compose up -e BASE_DATABASE=ipm_root
```

El docker-compose tiene configurada una red *bridge* con la siguiente *subnet*:

- 172.20.0.0/16

Por otro lado las máquinas cuentan con las siguientes direcciones IP estáticas asignadas:

- mongo: 172.20.0.2
- ipmanager-backend: 172.20.0.3
- ipmanager-frontend: 172.20.0.4

También se fija la variable de entorno *BASE_DATABASE* con valor *ipm_root*.

Apéndice D

Primeros pasos y *CLI*

D.1. Inicialización de la base de datos

Tras instalar el backend se debe ejecutar un script llamado init_database.py (se encuentra en la raíz del proyecto), el cual creará un usuario en el cliente base. Este usuario es administrador y sus credenciales deben ser cambiadas una vez haya sido creado.

El cliente base viene denominado por el nombre de la base de datos principal, el cual se toma de la variable de entorno BASE_DATABASE (toma valor ipm_root en caso de no encontrarse configurada).

Tras crear este primer usuario administrador, este puede comenzar a crear otros usuarios o clientes usando la *API* o el *CLI*.

D.2. *CLI*

El *CLI* que se incluye en el directorio raíz del backend permite realizar algunas operaciones con clientes (o *customers*) y usuarios. Se puede ejecutar de la siguiente manera:

```
1 cd backend
2
3 python3 cli.py
```

Es interactivo y permite:

- Crear un cliente.
- Activar un cliente.
- Desactivar un cliente.

- Agregar un usuario a un cliente.

Internamente utiliza la configuración de las variables de entorno que se encuentre establecida en el momento de utilizar el *CLI*.

Bibliografía

[1] Atlassian. Scrum. https://www.atlassian.com/agile/scrum.

[2] Multiple authors. Fernet (symmetric encryption). https://cryptography.io/en/latest/fernet/.

[3] Cypress.io. Cypress. https://www.cypress.io/.

[4] Ministerio de Industria Comercio y Turismo. Sociedad limitada nueva empresa. http://www.ipyme.org/es-ES/creaciondelaempresa/ProcesoConstitucion/Paginas/SLNE.aspx?cod=SLNE&nombre=Sociedad+Limitada+Nueva+Empresa&idioma=es-ES.

[5] Digité. Kanban. https://www.digite.com/kanban/what-is-kanban/.

[6] Enterat. Salario medio en españa 2020. https://www.enterat.com/actualidad/salario-medio-espana.php.

[7] Inc. Free Software Foundation. Gnu general public license version 3. https://www.gnu.org/licenses/gpl-3.0.html.

[8] Python Software Fundation. Python. https://www.python.org/.

[9] The Linux Fundation. Kubernetes. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[10] GECOS. Gecos. https://github.com/gecos-team.

[11] Google. Angular. https://angular.io/docs.

[12] Axel Haustant. Flask-restplus. https://flask-restplus.readthedocs.io/en/stable/quickstart.html.

[13] IBM. Database-as-a-service. https://www.ibm.com/cloud/learn/dbaas.

[14] Chef Software Inc. Chef. https://www.chef.io/products/chef-infra/.

[15] Docker Inc. Docker compose. https://docs.docker.com/compose/.

[16] Docker Inc. Docker sdk for python. https://docker-py.readthedocs.io/en/stable/index.html.

[17] Facebook Inc. React. https://es.reactjs.org/.

[18] Github Inc. Github actions. https://github.com/features/actions.

[19] Karma Software Inc. Karma. https://karma-runner.github.io/latest/index.html.

[20] MongoDB Inc. Pymongo. https://pymongo.readthedocs.io/en/stable/.

[21] Red Hat Inc. Ansible. https://www.ansible.com/.

[22] Red Hat Inc. Ansible tower. https://www.ansible.com/products/tower.

[23] Mahdi Javanmard and Maryam Alian. Comparison between agile and traditional software development methodologies. https://dergipark.org.tr/en/download/article-file/713866.

[24] Jenkins. Jenkins. https://jenkins.io.

[25] Steven Loria. Marshmallow. https://marshmallow.readthedocs.io/en/stable/.

[26] Microsoft. Typescript. https://www.typescriptlang.org/.

[27] José Padilla. Pyjwt. https://pyjwt.readthedocs.io/en/latest/.

[28] Portainer. Portainer. https://www.portainer.io/.

[29] RapidAPI. The rapidapi blog. https://rapidapi.com/blog/.

[30] Evan You. Vue.js. https://vuejs.org/.

[31] Ángel Gómez Martín. Dockerhub tags backend. https://hub.docker.com/r/harvestcore/ipm-backend/tags.

[32] Ángel Gómez Martín. Dockerhub tags frontend. https://hub.docker.com/r/harvestcore/ipm-frontend/tags.

[33] Ángel Gómez Martín. Ipmanager. https://github.com/harvestcore/tfg.

[34] Ángel Gómez Martín. Ipmanager releases. https://github.com/harvestcore/tfg/releases.

[35] Ángel Gómez Martín. Ipmdroid. https://github.com/harvestcore/ipmdroid.