[Propuesta de Arquitectura end-to-end 2](#_Toc18671131)

[Arquitectura de desarrollo 2](#_Toc18671132)

[Cómo se van a realizar las aplicaciones nuevas 2](#_Toc18671133)

[Estructura de la aplicación 2](#_Toc18671134)

[API Rest 3](#_Toc18671135)

[Separation of concerns 4](#_Toc18671136)

[Controller 4](#_Toc18671137)

[Tipos de objetos 5](#_Toc18671138)

[Con qué herramientas o tecnologías 5](#_Toc18671139)

[Spring Boot 5](#_Toc18671140)

[Spring Boot Actuator 6](#_Toc18671141)

[Eureka 6](#_Toc18671142)

[Cloud-config 7](#_Toc18671143)

[Zuul 7](#_Toc18671144)

[Feign 8](#_Toc18671145)

[Hystrix 8](#_Toc18671146)

[Spring Data 8](#_Toc18671147)

[Docker 9](#_Toc18671148)

[Arquitectura de ejecución 9](#_Toc18671149)

[DevOps 9](#_Toc18671150)

[Metodología 10](#_Toc18671151)

[Metodologías Agile 10](#_Toc18671152)

[Roadmap 12](#_Toc18671153)

[CI/CD 12](#_Toc18671154)

[Continuous Integration (CI) 12](#_Toc18671155)

[GitLab CI 12](#_Toc18671156)

[Continuous Delivery and Deployment (CD) 13](#_Toc18671157)

[GitLab CD 13](#_Toc18671158)

[Entornos 13](#_Toc18671159)

[Git-Flow 14](#_Toc18671160)

[Monitorización y control de errores de la aplicación 14](#_Toc18671161)

[Prometheus 15](#_Toc18671162)

[Grafana 15](#_Toc18671163)

[Monitorización y control de errores del servidor 15](#_Toc18671164)

[Nagios 16](#_Toc18671165)

[Seguridad 16](#_Toc18671166)

[Implementación 16](#_Toc18671167)

# Propuesta de Arquitectura end-to-end

Teniendo en cuenta lo comentado por Negocio y por IT, como propuesta para mejorar las capacidades de la compañía a nivel de arquitectura end-to-end (desarrollo y ejecución) será necesario establecer la arquitectura que se define a continuación en el documento.

## Arquitectura de **desarrollo**

### **Cómo se van a realizar las aplicaciones nuevas**

Con el fin de estandarizar y simplificar los nuevos desarrollos, se usarán las mismas tecnologías base, lo cual prescinde de la necesidad de un aprendizaje nuevo de cada proyecto, independientemente de la lógica de negocio de los proyectos.

Las nuevas aplicaciones se desarrollarán bajo una **Arquitectura de Microservicios**, como una serie de pequeños servicios. Cada microservicio será independiente y su código debe poder ser desplegado sin afectar a los demás, ejecutándose de forma autónoma y comunicándose entre sí.

Por lo tanto, se tendrá una aplicación modular a base de “pequeñas piezas” escalables, que se pueden ir ampliando o reduciendo a medida que se requiera.

### Estructura de la aplicación

Dependiendo de los requisitos y de la estructura de la aplicación, se puede organizar los microservicios de diferentes formas.

**Proyectos separados**: Si los microservicios no son dependientes entre ellos se debe de optar por crear proyectos separados, ya que de esta forma se consigue una mayor flexibilidad y pueden diferir en cuanto a configuración sin que afecten a otros microservicios.

**Proyecto multimódulo**: Si los microservicios tienen dependencias entre ellos a nivel de lógica de aplicación, se agruparían en un proyecto multimódulo, de esta forma nos permite identificar si la modificación de un microservicio puede afectar al resto de los microservicios de los que depende.

**Proyecto sin módulos**: Cuando los microservicios tienen dependencia a nivel técnico y de lógica de aplicación, ya que podrían a compartir las mismas librerías o incluso la misma base de datos, sería conveniente tener varios servicios dentro del mismo proyecto, pudiendo iniciarlos por separado.

Para independizar bien el diseño, el microservicio sólo debería poder comunicarse con el exterior (incluyendo comunicación con otros microservicios) a través de su interfaz **API** pública (REST con JSON).

Este modelo posibilita que la aplicación pueda ser consumido por externos.

##### API Rest

API es un conjunto de reglas y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas. Garantizando el intercambio de información en formato estándar.

Las API Rest aportan sustanciales ventajas, cabe destacar:

**Separación entre cliente y servidor**, de esta manera, aumenta la escalabilidad de los proyectos y permite que los desarrollos evolucionen de forma independiente.

**Interfaz uniforme:** Se consigue sistematizar el proceso con la información gracias a las acciones que contiene REST.

Siempre es **independiente** del tipo de plataformas o lenguajes, por tanto, se adapta al tipo de sintaxis o plataformas con las que se estén trabajando. No obstante, se debe tener en cuenta que las respuestas a las peticiones se hagan siempre en el **lenguaje de intercambio** de información usando **JSON** o **XML**.

**Simplifica los desarrollos**, puesto que no se necesita picar código desde cero, sino hacer solo modificaciones dependiendo el uso que se le va a dar y el tipo de sitio dónde se quiera implementar.

##### Separation of concerns

Se usará la Separación de intereses como buena práctica de diseño de software, la cual consiste en asegurarse que cada clase, componente, etc. tenga una responsabilidad bien definida, de manera que sus funcionalidades o métodos no hagan más de un tipo de actividad.

La separación de responsabilidades dependerá de la función del microservicio, pudiendo ser aplicada de diferentes formas y niveles. Por ejemplo, en una aplicación sencilla podríamos separar tres responsabilidades bien definidas:

* **Capa presentación**: Interactuar con el cliente. Aquí agrupamos clases como las del controlador, donde expondremos los servicios de la aplicación.
* **Capa de servicio/negocio**: Ejecutar procesamientos y aplicar reglas de negocio. Estas son clases que contienen la lógica de lo que hace el servicio. Aquí agrupamos los servicios.
* **Capa acceso a datos**. Almacenar y extraer la información de una base de datos. Aquí agrupamos Repository, DAO, etc.

##### Controller

Un controlador, aunque es más conocido con el término “controller” será el punto de entrada de nuestra aplicación, es el encargado de orquestar a las peticiones recibidas en nuestro servidor, que usualmente son realizadas por el usuario, aunque también pueden ser peticiones más automatizadas como APIS, etc.

* + - * Para este caso, usaremos Java, se podría usar Jackson o Gson para parsear el texto de las peticiones REST provenientes del cliente en formato JSON y convertirlo en una serie de DTOs que contienen esa información, que posteriormente serían consumidos por las capas inferiores. El mecanismo de emisión de la respuesta es el contrario: el core de la aplicación genera unos DTOs de respuesta que serán devueltos mediante una cadena JSON entendible para el cliente.

##### Tipos de objetos

Como se ha mencionado anteriormente, tendremos un tipo de objeto que transporta la información entre estas capas de nuestra aplicación, para reducir el acoplamiento. Son los llamados **DTO** (Data Transfer Object), simples contenedores de datos, sin métodos o comportamiento.

Un tipo de objeto muy parecido a los DTO son las **Entity** que representan tablas o registros en la base de datos. Estas entidades tienen la particularidad de ser clases que están mapeadas contra una tabla de la base de datos, lo cual nos permiten manipular la base de datos a través de objetos.

La mejor práctica es extraer un Entity de la base de datos y declarar algunos de sus valores en un DTO equivalente para devolverlo a la capa superior en lugar de devolver el Entity. De esta manera se evita que el cliente reciba información de campos indeseados, reduciendo así el acoplamiento y mejorando la abstracción. De la misma manera, para aumentar la abstracción, podemos tener un Request DTO y un Response DTO, personalizando los campos que irán en la petición y en la respuesta.

* + - * Cada microservicio se considera aislado de los demás, por ello, no debe compartir los DTOs de la interfaz del API.

### **Con qué herramientas o tecnologías**

En este apartado se detalla el stack tecnológico que contendrán los nuevos microservicios al crear una aplicación.

##### Spring Boot

Para crear las aplicaciones usaremos Spring Boot, el cual posibilita que se consuma un menor número de recursos. Permitiendo al desarrollador, con el fin de agilizar el desarrollo, que solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar. A continuación, se detallan las principales ventajas que ofrece:

* **Configuración**: Cuenta con un complejo módulo que auto configura todos los aspectos de nuestra aplicación para directamente ejecutarla.
* **Resolución de dependencias**: Se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
* **Despliegue**: Permite desplegar las aplicaciones mediante un servidor web integrado.
* **Métricas**: Cuenta con servicios que permiten consultar el estado de la aplicación, memoria utilizada y disponible, entre otras funcionalidades.

##### Spring Boot Actuator

##### Eureka

Eureka, como “***Service Discovery***”, es un **servidor para el** **registro y localización** de microservicios, balanceo de carga y tolerancia a fallos. Registra las diferentes instancias de microservicios existentes, su localización, estado, metadatos, etc.

Cada microservicio, durante su arranque, y cada 30 segundos una vez arrancado, se comunicará con el servidor Eureka para notificar que está disponible, dónde está situado, etc. (lo que denominan “*heartbeats*”). De esta forma Eureka mantendrá en su registro la información de todos los microservicios del ecosistema

Cada cliente de Eureka podrá también recuperar el registro para localizar otros microservicios con los que necesite comunicarse. Dicha información de registro se cachea en el cliente.

Esto le aporta a Eureka una serie de ventajas importantes:

* **Abstracción** de la localización física de los microservicios: cualquier microservicio que sea un cliente Eureka solo necesita conocer el identificador del microservicio al que desea invocar y Eureka resolverá su localización, puerto, etc.
* **Conocer el estado** de nuestro ecosistema de microservicios: Eureka proporciona un “*Dashboard*” que permite ver los microservicios existentes actualmente en el registro.
* Se puede **configurar como cluster** incrementando notablemente su tolerancia a fallos.
* **Soporte a multiregión**, pudiendo definir diferentes agrupaciones de microservicios.

##### Cloud-config

Es un **servidor de configuración** pensado para sistemas distribuidos. Su función es **almacenar** las **propiedades de configuración** de los microservicios del ecosistema.  
  
Durante el arranque de un microservicio éste consultará al servidor de Cloud-config las propiedades asociadas, quién se las descargará del repositorio (por defecto del git).   
  
Las ventajas que aporta son múltiples, cabe destacar:

* **Se integra con Eureka** de forma que, durante el arranque de un microservicio, puede consultar al servidor Eureka donde se encuentra el servidor de configuración y le solicitará su configuración.
* Al almacenar la configuración en un repositorio git esto proporciona un **repositorio centralizado** de configuración e **histórico del mismo**.
* Permite **gestionar configuración** de diferentes entornos, consultar por etiquetas, etc.
* En integración con Spring, es posible reiniciar el contexto entero de Spring en caliente. Esto permite que los microservicios se adapten a **cambios en ejecución** en las properties.

##### Zuul

Zuul, como “***Edge Service****”*, actúa como un proxy inverso, proporcionando un punto de entrada a nuestro ecosistema de microservicios, proporcionando capacidades de enrutamiento dinámico, seguridad y monitorización de las llamadas que se realicen.

Nos permite implementar el patrón **API Gateway**, un servicio en el lado servidor responsable de la agregación de datos y enrutamiento de peticiones a los servicios que correspondan.

Zuul **utiliza Ribbon** para localizar otros servicios y conseguir enrutar una llamada externa a una instancia concreta de un microservicio.

Netflix Ribbon (Dynamic Routing y Load Balancer), se comunicará con Eureka para obtener información de localización de los servicios. Como **balanceador de carga**, si Ribbon encontrase más de una instancia disponible, distribuirá las peticiones de la manera más óptima.

##### Feign

Feign facilita la **integración entre microservicios** mediante la creación de clientes HTTP de forma declarativa. De manera que simplifica tanto la creación de clientes, que el desarrollador simplemente necesita anotar una interfaz para tenerlo.

Feign fue desarrollado por Netflix y por ello es ideal para su integración con [Eureka](#_Eureka), Ribbon e [Hystrix](#_Hystrix).

##### Hystrix

Se trata de una librería ofrecida por Netflix diseñada para aislar puntos de acceso a sistemas remotos, servicios y librerías de terceros, deteniendo fallos en cascada, proporciona opciones de respaldo permitiendo mejorar la resiliencia en sistemas complejos distribuidos donde la probabilidad de fallo es inevitable.

*Por ejemplo, cuando se está llamando a aplicaciones de terceros, se necesita más tiempo para enviar la respuesta. En ese momento, el control pasa al método alternativo y devuelve una respuesta personalizada.*

##### Spring Data

Spring Data permite unificar la configuración, simplificar la implementación de la capa de acceso a datos y crear una jerarquía de excepciones común, tanto para bases de datos relacionales como a las del tipo NoSQL.

Integra las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas. Además, proporciona interfaces genéricas como CrudRepository, PagingAndSortingRepository, etc. e implementaciones específicas para cada tipo de tecnología de persistencia.

##### Docker

Docker permite **crear contenedores ligeros y portables** para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando al mismo tiempo los despliegues.  
  
Las principales ventajas se pueden resumir de la siguiente manera:

* Solo se tiene que **programar la aplicación una sola vez**. Dado que una aplicación en Docker se ejecuta dentro de un contenedor, y el contenedor se puede ejecutar en cualquier sistema operativo que tenga Docker instalado.
* Se obtiene una **mayor consistencia** entre los entornos de prueba y los entornos de producción. Tanto las pruebas de la aplicación, como los despliegues se realizan dentro de contenedores. Lo cual conlleva a que el entorno de pruebas es idéntico al entorno en el que se va a ejecutar el software.
* Se obtiene **mayor modularidad**. El desarrollo con contenedores es ideal para un enfoque basado en microservicios para el diseño de aplicaciones. Hace que la aplicación sea modular, reduciendo la complejidad de tener que mantener y actualizar la aplicación, dado que un error o un cambio relacionado con una parte de la aplicación no requiere que se revise la aplicación completa.

## Arquitectura de ejecución

Una definición sólida de la Arquitectura de ejecución a seguir es fundamental para poder llevar a cabo proyectos que brinden de cierta calidad.

##### DevOps

Es necesario adoptar una cultura o filosofía que permita utilizar de forma correcta todas las herramientas y una comunicación constante entre todos los equipos implicados. Este cambio de cultura se conoce como DevOps y fomenta la colaboración entre áreas como desarrollo, operaciones y QA, permitiendo desarrollar el software de forma más rápida, eficiente y por consiguiente con mayor calidad junto a un coste inferior.

Está compuesto por diferentes fases como la integración continua, entrega continua, despliegue continuo y desarrollo bajo [metodologías agile](#_Metodologías_Agile) que buscan siempre el incremento de valor y la mejora

Algunas de las características principales que serán desarrolladas posteriormente son las siguientes:

* Habilita los mecanismos necesarios para dar la posibilidad a cualquier usuario de aportar ideas y que estas puedan ser transformadas en requerimientos u objetivos.
* Ayuda a la construcción de software propiciando mecanismos para la **creación de entornos** de desarrollo repetibles, utilizando tecnologías como [Docker](#_Docker).
* Permite **detectar problemas** de software de forma temprana, mediante [Continuous Integration](#_Continuous_Integration_(CI)_1).
* Permite **entregas continuas y despliegues automáticos** gracias al [Continuous Delivery and Deployment](#_Continuous_Delivery_and).

### Metodología

Además del DevOps, será necesario aplicar nuevos métodos de trabajo más dinámicos que permitan adaptar los proyectos a los nuevos escenarios que necesiten soluciones con mayor rapidez y flexibilidad, siendo capaces de actuar de forma proactiva ante situaciones imprevistas.

#### Metodologías Agile

Las metodologías ágiles permiten el cambio a estrategias emergentes, al tiempo que liberan al personal para que puedan remodelar su negocio en tiempo real, minimiza la confusión y la complejidad.

El sistema tiene como fin satisfacer la satisfacción del cliente, lo cual es la base de todo. Para ello será necesario cumplir los siguientes aspectos:

* Entrega de productos de valor que cubran una necesidad.
* Adaptación a circunstancias cambiantes.
* División del trabajo en fases productivas.
* Medición del progreso.
* Desarrollo sostenible: La forma de ejecutar los proyectos debe garantizar su continuidad.
* Trabajo cercano: Los líderes de los proyectos trabajan junto a los equipos.
* Comunicaciones eficaces: Mensajes eficaces desde los gestores.
* Reuniones periódicas tanto con el cliente como con sus colaboradores.
* Motivación y confianza: Los procesos sólo tendrán éxito si quienes los llevan a cabo son personas motivadas y que interactúan en climas de confianza y empatía.
* Excelencia técnica y buen diseño: Cuidando las formas y la calidad del trabajo.
* Simplicidad: Las tareas han de ser lo más sencillas posible, dividiendo en iteraciones para reducir su nivel de complejidad, cuando sea posible.
* Autogestión de los equipos: Si bien debe existir una figura que monitorice los equipos de trabajo, éstos deben ser capaces de organizarse por sí mismos. El exceso de jerarquías crea dependencia entre los colaboradores.

Hay que tener en cuenta que para implementar la metodología Agile, será necesario partir de unos principios que serán la base.

**Liderazgo**: Los líderes deben ser el motor que impulse la introducción de sistemas de trabajo Agile. Deben desarrollar una mentalidad que valore la flexibilidad que caracteriza a esta metodología, pensando y actuando de manera diferente.

**Cultura alineada**: La cultura organizacional debe estar alineada con esta nueva forma de funcionamiento. Asentando las bases para que haya una comunicación fluida y transparente entre todo el capital humano.

**Estrategia Lean:** Evaluación continua con motivo de aumentar la velocidad de respuesta de los equipos, permitiendo que se reduzcan procesos innecesarios, ahorro de costes y se acorten los tiempos de los desarrollos.

**Herramientas Agiles**: La implementación de métodos de trabajo Agile requiere de la incorporación del software adecuado que posibilite el desarrollo de este enfoque, con herramientas basadas en modelos como *Scrum* o *Kanban*.

#### Roadmap

Roadmap: ¿cómo sería el roadmap de mejoras/cambios/etc propuestos para la organización de forma que no se pierda capacidad de Negocio?

### CI/CD

Cabe destacar los beneficios que trae aplicar CI/CD.

* Mejora de productividad de desarrollo al evitar tareas manuales, fomentando el ambiente *Agile*.
* Detecta y corrige errores rápidamente.
* Tanto las nuevas características como la resolución de incidencias llegarán mayor rapidez y frecuencia a los clientes. Por ende, es posible recibir feedback más rápido.

#### Continuous Integration (CI)

La integración continua es una técnica que tiene como objetivo detectar los posibles problemas que pueda tener el software, para así solucionarlos preventivamente. Para esta ello es imprescindible realizar de forma periódica compilación del código, ejecución de tests unitarios y revisión de la calidad del código. Para visualizar los resultados de forma ágil, se hará uso de [GitLab CI](#_GitLab_CI).

##### GitLab CI

Forma parte del conocido sistema de control de versiones GitLab. Además de integración continua, GitLab ofrece despliegue y entrega continua. Está basado en un modelo de fichero de configuración escrito en YAML y almacenado en la raíz de cada repositorio. Además, disfruta de conexión directa con otros productos de Atlassian, los cuales son muy útiles en la metodología Agile.

Las principales ventajas que posee:

* Ofrece un control de versiones (commits subidos al repositorio).
* La configuración está versionada, lo cual fomenta que cada rama tenga su propia configuración.
* Integración con Docker sin necesidad de configurar nada, incluyendo un Docker registry privado por proyecto.
* Incluye explorador de artefactos para acceder a los resultados de cada etapa.
* Estrecha vinculación con la automatización de tests, construcción y pruebas.
* Despliegues en ambientes aislados las versiones de la aplicación (el despliegue en producción se realiza manualmente).

#### Continuous Delivery and **Deployment** (CD)

El despliegue continuo consiste en automatizar el proceso de despliegue de la aplicación en producción lo más rápido posible, sin necesidad de intervención humana.

Para lograr este propósito, el pipeline de producción tiene una serie de pasos que deben ejecutarse en orden. De la misma manera se asegura de que se cumplan la reglas y se lleve a cabo el despliegue.

##### GitLab CD

Al igual que para la integración continua, es posible usar GitLab para el despliegue continuo.

* Aseguramiento de que todos los componentes son compatibles unos con los otros.
* Garantiza la integridad mientras el software avanza entre los diferentes entornos: desarrollo, testing, preproducción y producción.
* Permite realizar un rollback al último estado estable.
* Programación de despliegues de diferentes historias de usuario en versiones, lo que permite además disponer de una trazabilidad completa de qué y cuándo se despliega en cada entrega.

### Entornos

Parte del dimensionamiento de un proyecto se basa en la elección adecuada de los necesarios entornos de desarrollo. En cuanto a la elección de los entornos de desarrollo, irá en función de las características y volumen del proyecto.

Respecto a los entornos, podríamos considerar:

* **Entorno local**: Suelen estar ubicados directamente en las estaciones de trabajo de cada desarrollador. Trabajando de forma aislada con el resto de las capas, los desarrolladores pueden probar cambios radicales en el código sin afectar de forma adversa al resto del equipo de desarrollo.
* **Entorno de integración o desarrollo**: En entorno común donde todos los desarrolladores hacen *"commits"* de los cambios en el código. Con el objetivo de combinar y validar el trabajo del equipo completo del proyecto antes de ser promovido al entorno de testing.
* **Entorno de testing**: Es el entorno en el cual los desarrolladores o QA prueban la aplicación con baterías de test.
* **Entorno preproducción**: Es un entorno lo más idéntico posible al entorno de producción. El propósito principal del entorno de testing es simular al entorno de producción asegurando que no se corrompe la aplicación existente en los servidores en producción. Además de minimizar las caídas del sistema en producción.
* **Entorno de** **producción**: Es el entorno donde trabajan los usuarios finales y con datos de negocio reales.

Es posible que los entornos compartan el mismo servidor físico, pero el entorno de producción debe estar aislado y no debe compartirse con cualquiera de los otros entornos.

##### Git-Flow

Es una herramienta que pone en práctica y automatiza convenciones simplificando el trabajo.

### Monitorización y control de errores de la aplicación

Una vez la versión del software esté operativa y en producción, es fundamental realizar un seguimiento de la misma. Hay que prestar especial atención en los siguientes puntos.

**Monitorización del rendimiento de la aplicación y servidor**. Realizar automatización para avisar en caso de fallo de la aplicación o del servidor salten alertas automáticamente.

**Seguimiento de problemas, incidentes y cambios**. Es fundamental una comunicación fluida entre las diferentes personas encargadas del proyecto para poder identificar y resolver los posibles problemas que se vayan encontrando y comunicarlo inmediatamente.

Cualquier proyecto que se tenga en producción deberá tener una monitorización para conocer el estado de nuestra aplicación, los recursos que consume, etc. [Prometheus](#_Prometheus) y [Grafana](#_Grafana) proporcionan una amplia cantidad de funcionalidades de cara a gestionar y monitorizar nuestras aplicaciones Spring Boot.

##### Prometheus

Prometheus es una base de series de tiempo y un sistema de monitoreo y alertas. Permite agregar instrumentos a nuestros servicios. Las métricas reportadas por los instrumentos (a través de un *“endpoint”* HTTP) son coleccionadas por el servidor de Prometheus de forma periódica.

Las series de tiempo almacenan datos ordenados cronológicamente, midiendo variables a lo largo del tiempo y las bases de datos enfocadas a series de tiempo almacenan y permiten consultar estos datos.

Algunos de los principales indicadores en los que se debería prestar atención, y que proporciona Prometheus, además de entre otros son:

* **Latencia**: El tiempo que le toma a una solicitud ser servida.
* **Tráfico**: Mide la demanda de un servicio.
* **Errores**: Cantidad de solicitudes fallidas.
* **Saturación**: Capacidad del sistema en función del uso, memoria RAM necesaria para satisfacer la demanda, etc.

Además, **permite configurar alertas** en función de la configuración y la serie de datos recibidos.

##### Grafana

Grafana es una plataforma de análisis para métricas. Permite consultar, visualizar y alertar. Además, permite crear, explorar y compartir tableros con el resto del equipo. Uno de sus puntos más fuertes respecto a la competencia es la visualización y personalización del tablero de instrumentos.

### Monitorización y control de errores del servidor

Para asegurar la continuidad del negocio es imprescindible darle importancia a la monitorización de la infraestructura para poder realizar acciones preventivas antes de recurrir a soluciones reactivas.

##### Nagios

Nagios es un **sistema de monitorización** que permite vigilar equipos y servicios, generando alertas sobre el comportamiento. Entre sus características principales figuran:

* Monitorización de servicios de red (SMTP, POP3, HTTP, SNMP...).
* Monitorización de los recursos de sistemas hardware (carga del procesador, uso de los discos, memoria, estado de los puertos...).
* Posibilidad de monitorización remota mediante túneles SSL cifrados o SSH.
* Independencia de sistemas operativos,

### Seguridad

## Implementación

Como buen Arquitecto de Soluciones será necesario que lideres el proceso de transformación tecnológica mediante labores de desarrollo y de establecimiento de buenas prácticas. Por ello te pedimos que codifiques una sencilla PoC end-to-end, en la que aplicar algunos de los principios de la arquitectura.

El MVP debe contener cómo mínimo:

* + Una aplicación frontal en la que exista una vista que muestre un listado de pólizas. Al hacer click en una de ellas se mostrará su detalle en una vista nueva. El detalle contendrá datos relativos a la propia póliza (producto, prima,...,etc), datos del tomador (nombre, apellidos, teléfono y email) y datos del vehículo asegurado. En esa vista de detalle se permitirá únicamente la edición de los datos del tomador, persistiendo los cambios en base de datos.
  + Servicios que den soporte a las vistas y las operaciones necesarias

Extras opcionales:

* + Autenticación y seguridad
  + CI/CD

Al igual que en el primer apartado, no es necesario que la aplicación Font tenga unos estilos muy trabajados. No dediques mucho esfuerzo a ello.