7. Sistemas Centralizados y Distribuidos

7.1. Introducción y objetivos

A la hora de desarrollar una nueva aplicación o sistema web es muy importante pensar en el tipo de sistema que se quiere construir, ya que esa decisión influirá fuertemente en los patrones de desarrollo a utilizar para su construcción, en el mantenimiento de esta, en su capacidad de escalado y, por supuesto, en el coste económico total de la solución.

En este sentido encontramos dos tipos de sistemas:

* Centralizados: todo el procesamiento de los datos y la lógica de negocio se ubica en un único lugar.
* Distribuidos: el procesamiento de datos y la lógica de negocio se ubica en varios lugares.

Como veremos, cada una de estas opciones tiene sus ventajas y desventajas y es adecuada para ciertos tipos de situaciones. A lo largo de este tema estudiaremos dos patrones muy conocidos:

* MVC (modelo-vista-controlador) en el caso de los sistemas centralizados.
* Microservicios en el caso de los sistemas distribuidos.

7.2. Sistemas centralizados

Un sistema centralizado (o monolítico) es un sistema donde las capas de la interfaz de usuario, de lógica de negocio y de acceso a datos se encuentran combinadas dentro de un único programa o plataforma, que podríamos denominar núcleo.

Como podrás imaginar, el hecho de que este tipo de sistemas tenga dentro de él todas las herramientas necesarias para funcionar persigue el objetivo de crear aplicaciones totalmente autónomas e independientes de otras. En general, dentro de un sistema centralizado o monolítico no se suelen encontrar rastros de modularidad, aunque esto ha ido cambiando a lo largo de los años con la aparición de patrones como MVC.

Los sistemas centralizados y monolíticos presentan las siguientes desventajas principalmente:

* Por su naturaleza son sistemas muy rígidos y el coste de adaptarlos con nuevas versiones de *software*, nuevas plataformas o incluso añadir una nueva funcionalidad es elevado.
* Su capacidad de escalado es vertical principalmente. Esto quiere decir que, si queremos que el sistema disponga de más capacidad de cómputo, es necesario sustituir el servidor físico por uno con más potencia.
* Punto único de fallo.

No obstante, estos sistemas también presentan algunas ventajas:

* Son sistemas generalmente estables y seguros, ya que es más fácil aportar seguridad a un único componente que a un sistema con cientos de ellos.
* La capacidad de administración es mayor.

Algo importante para tener en cuenta es que, en su mayoría, los sistemas centralizados son proveídos por empresas de *software* propietario por lo que, en estos casos, es muy común que haya detrás un equipo de soporte con un conocimiento altamente especializado en su producto. Sin embargo, al usar una tecnología propietaria, la dependencia que se crea con el proveedor puede suponer un problema a largo plazo.

En los comienzos de la ingeniería de *software*, las aplicaciones y sistemas que se desarrollaban seguían este estilo monolítico y acababan convirtiéndose en grandes soluciones que impedían su uso de forma modular. Este fue uno de los motivos que llevó a la famosa crisis del *software.*

Se puede profundizar sobre este tema en el recurso *Crisis del software,* el cual está disponible en A fondo.

Patrón MVC

El patrón MVC es un patrón muy conocido y utilizado en aplicaciones web. Este permite aportar cierta claridad (gracias a la modularización) en las aplicaciones monolíticas.

El patrón MVC (*model view controller* o modelo-vista-controlador) gestiona la idea de combinar tres piezas diferentes que trabajan en consonancia para construir una aplicación compleja.

La idea, por tanto, es sencilla y, además, extremadamente flexible: separamos la lógica de nuestra aplicación en tres capas. Estas capas son las siguientes:

A screenshot of a computer

Description automatically generated

Tabla 1. Capas del patrón MVC. Fuente: elaboración propia.

Gráficamente, las relaciones que aparecen entre los tres componentes del patrón se plasmarían de la siguiente manera:

A diagram of a software development

Description automatically generated

Figura 1. Patrón MVC. Fuente: elaboración propia.

Evidentemente, el hecho de implementar un patrón de este tipo requiere de un esfuerzo inicial adicional a la hora de la implementación y el desarrollo. No obstante, los beneficios que se obtienen de su uso compensan dicho esfuerzo. Algunos de estos son:

* El código es más flexible, más reutilizable y escalable.
* Permite dividir el trabajo entre diferentes programadores de forma más sencilla.
* Hace que la aplicación sea más ligera.

Para ir ilustrando la funcionalidad de cada una de las partes, trabajaremos con el siguiente ejemplo:

Una empresa nos solicita crear una aplicación web para gestionar la información de empleados de esta. A simple vista, lo que necesitamos es implementar funcionalidad para mostrar los datos de los empleados, agregarlos, eliminarlos y actualizar su información asociada.

Modelo

Este componente se encarga de implementar las características y restricciones del problema particular que estamos resolviendo. Por tanto, es el elemento más cercano a la empresa o negocio para el que estamos desarrollando nuestra aplicación. Lo que tendremos que hacer será implementar aquí el acceso a la base de datos o, en general, el acceso a la capa de persistencia de los datos que se usen en la aplicación.

Generalmente, el modelo realizará una conexión a una base de datos y efectuará diferentes consultas sobre ella para obtener, almacenar o eliminar datos. Sin embargo, esta es la versión más «común» y básica que podemos encontrarnos.

En aplicaciones complejas podemos encontrarnos con que el modelo accede a diferentes orígenes de datos (bases de datos Oracle, mySql, no relacionales, etc.) y es necesario que sea capaz de interactuar correctamente con dichos orígenes.

En el modelo MVC, la lógica de negocio queda del lado del modelo. Por tanto, el modelo debe ser capaz de entender y llevar a cabo todas las reglas del negocio en cuestión, además del tratamiento de los orígenes de datos.

A diagram of a software development process

Description automatically generated

Figura 2. Modelo en MVC. Fuente: elaboración propia.

En el ejemplo anterior, el modelo sería el encargado de ofrecer al controlador todas aquellas operaciones y datos que demande en relación con los empleados de la empresa.

Vista

La vista es la encargada de presentar la información al usuario, por lo que solo tiene que coger los datos que alguien ha calculado y mostrarlos de forma que el usuario pueda entender.

La vista solo conocerá las diferentes pantallas o plantillas que puede mostrar y, en función de la información que reciba del controlador, seleccionará una u otra. La vista o la plantilla será modificada en consonancia a los datos que el modelo ha calculado y que ha entregado al controlador.

Es importante tener en cuenta que una vista no es solo una página más dentro de la aplicación web, sino que puede tener diferentes formatos. Lo más habitual es que nos encontremos vistas escritas en código HTML haciendo uso de CSS o PHP, pero también podríamos esperar otro tipo de vistas, como archivos PDF, Word, Excel, etc. Al fin y al cabo, la vista es la forma de mostrar la información solicitada al usuario y puede tomar numerosas formas.

A diagram of a computer network

Description automatically generated with medium confidence

Figura 3. Vista en MVC. Fuente: elaboración propia.

En el ejemplo propuesto podríamos identificar las siguientes vistas (entre muchas otras):

* Vista web de todos los empleados.
* Vista web con formularios para añadir, eliminar o modificar empleados.
* Un documento PDF con la información de los empleados de un departamento.

Controlador

El controlador maneja las peticiones de entrada que recibe el sistema. Esta capa sirve de enlace entre el modelo y la vista, pero no se encarga de manipular los datos ni mostrar la salida, cuestiones que quedan a los otros elementos del patrón.

En términos un poco más técnicos, se encarga de implementar la lógica de control de la aplicación, es decir, controlar el orden en que se tienen que ejecutar las cosas y/o responder a los eventos que ocurren en la aplicación.

Un controlador sencillo se encargaría de solicitar los datos al modelo y pasarlos a la vista para que los mostrara al usuario.

En el ejemplo, el controlador debería encargarse de los siguientes aspectos (entre otros):

* Quién puede acceder a los datos.
* Quién puede actualizarlos.
* Qué restricciones existen cuando se actualiza un campo.

Patrón MVC y arquitecturas monolíticas

Ahora que conocemos que el patrón MVC nos permite separar y organizar el código de la aplicación de una forma elegante y que, además, favorece la flexibilidad y el trabajo independiente por parte de los desarrolladores, ¿deberíamos seguir diciendo que una aplicación que utilice este patrón es monolítica?

Por desgracia, la respuesta es sí. No hay ninguna duda en cuanto a que este patrón es una muy buena práctica a llevar a cabo dentro del mundo del desarrollo web, pero no debemos olvidar que las vistas, el controlador y el modelo se encuentran y funcionan dentro de un mismo lugar (la plataforma, el servidor, etc.) y, aunque el mantenimiento de la aplicación sea más sencillo, seguiremos encontrando las mismas desventajas que explicábamos al inicio de la sección (y sus ventajas también).

### **7.3. Sistemas distribuidos**

Un sistema distribuido es un sistema software donde los diversos componentes (capa frontal, servicios que componen el back-end, capas de persistencia) se encuentran separados físicamente y conectados a través de la red (privada o pública) y que se comunican entre ellos mediante algún mecanismo de paso de mensajes.

Las **principales características** de un sistema distribuido son las siguientes:

A blue and white text on a blue background

Description automatically generated

Tabla 2. Principales características de un sistema distribuido. Fuente: elaboración propia.

Para comunicar los diferentes nodos que pueden componer un sistema distribuido, es necesario algún tipo de ***middleware*,** es decir, un programa que proporcione un **enlace entre sistemas totalmente independientes.** Los más habituales son los orientados a mensajes (MOM).

Debido al aumento de la popularidad de la computación en la nube (cloud computing) la inmensa mayoría de los sistemas que se desarrollan actualmente son distribuidos, y no es casualidad, ya que la computación distribuida permite realizar los mismos trabajos que la centralizada, pero con mayor velocidad y eficiencia y mejorando notablemente la tolerancia a fallos.

No obstante, no todo son ventajas en el mundo de los sistemas distribuidos:

* Es, por su naturaleza, **mucho más complejo de diseñar, implementar y configurar.** Es necesario tener en cuenta muchos factores, ya que cada uno de los nodos del sistema contará con una infraestructura y seguridad propias.
* **La inversión en seguridad debe ser mayor** de forma prácticamente obligatoria. El hecho de que tengamos múltiples componentes o nodos que traten la información y los datos del negocio hace necesario establecer conexiones seguras entre todos los componentes y asegurar que dicha información no quede comprometida. Además, dado que la lógica estará repartida en diferentes componentes, es necesario monitorizar constantemente cada una de ellas para prevenir y mitigar posibles ataques.
* **El esfuerzo de mantenimiento es mayor,** ya que en un sistema distribuido podemos encontrar máquinas físicas totalmente diferentes (diferentes sistemas operativos, diferentes versiones de estos, diferentes configuraciones, etc.). Poner a funcionar de forma unísona sistemas a priori diferentes es mucho más complejo que gestionar un único servidor, como ocurre en los sistemas centralizados.

Existen varios tipos de arquitecturas basadas en sistemas distribuidos. Algunos de ellos son:

* **Cliente-servidor.** Se trata del modelo clásico de sistema distribuido. Disponemos de dos nodos. El cliente, que es con el cual el usuario interactúa, y el servidor, que se encarga de realizar la lógica de negocio y de tratar todos los datos. Ambos componentes se comunican normalmente mediante una API REST o a través de un servicio web.

A diagram of a computer network

Description automatically generated

Figura 4. Arquitectura distribuida cliente-servidor. Fuente: elaboración propia.

* **Editor-suscriptor.** En este caso disponemos de N máquinas que se comportan como editores (publican mensajes) y de N máquinas que se comportan como suscriptores (reciben mensajes). Los suscriptores se suscriben a un topic o tema concreto y, en el momento en el que un editor envía un mensaje de dicho topic, los suscriptores lo reciben. Además, este sistema contempla la posibilidad de que los actores se comporten de forma dinámica, es decir, que un editor deje de serlo y que un suscriptor pueda suscribirse a más de un topic o que deje de seguir alguno de ellos. Es una excelente opción si se busca procesamiento asíncrono (en segundo plano).

A diagram of a diagram

Description automatically generated

Figura 5. Arquitectura distribuida editor-suscriptor. Fuente: elaboración propia.

* **Colas de mensajería.** Muy similar al editor-suscriptor solo que, en lugar de utilizar topics, se utilizan colas. La diferencia entre un topic y una cola es que un **topic** puede tener varios componentes escuchando de él (suscritos) y cuando se recibe un mensaje en el topic, todos esos componentes son notificados. Las colas, por el contrario, solo permiten tener un único componente a la escucha.Al igual que en el caso anterior, **este modelo es muy efectivo si se busca implementar procesamiento asíncrono** de la información.
* **Microservicios.** Las arquitecturas orientadas a microservicios explotan todo el potencial de un sistema distribuido, ya que pueden combinar en una sola los diferentes modelos expuestos anteriormente.

### **Arquitecturas orientadas a microservicios**

Para entender qué es un microservicio, es muy útil explorar las causas que llevaron a su aparición. Principalmente, surgieron debido a una **necesidad.**

Las arquitecturas centralizadas se encontraban con serios problemas en cuanto a la escalabilidad horizontal y las arquitecturas distribuidas clásicas no satisfacían algunas necesidades, como la separación total de la lógica de negocio de los sistemas.

Los microservicios deben encapsular una pequeña, pero bien definida, parte de la lógica de negocio y exponer esa lógica a través de una API REST delimitada.

Un microservicio es un servicio especializado en resolver un problema concreto y que se desarrolla, despliega y mantiene de forma totalmente independiente del resto de componentes del sistema distribuido.

Sus características principales son:

A blue and white text on a blue background

Description automatically generated

Tabla 3. Características de los microservicios. Fuente: elaboración propia.

A la hora de trabajar con microservicios es conveniente contar con una estrategia de integración y despliegue continua. Esto es debido a que, si tenemos, por ejemplo, diez microservicios para un sistema medianamente complejo, desplegar manualmente cada uno de estos microservicios puede conllevar un trabajo adicional considerable.

### **Ventajas de los microservicios**

Elegir una arquitectura orientada a microservicios puede aportar numerosas **ventajas.**

**Agilidad**

El hecho de que los microservicios se especialicen en un área concreta agiliza enormemente el ciclo de desarrollo, ya que es fácil identificar dónde es necesario actuar.

**Escalabilidad**

Los microservicios escalan muy bien horizontalmente. Esto facilita enormemente la gestión de la demanda por parte de los clientes y permite asegurar la disponibilidad del servicio.

Si dentro de un sistema distribuido que ofrece un servicio a una tienda de ropa tenemos un microservicio encargado de procesar las compras de los clientes y sabemos que durante un período de tiempo concreto (debido a rebajas, por ejemplo) habrá una mayor afluencia de peticiones, es posible desplegar más copias de ese microservicio para que sean capaces de atender toda la demanda de peticiones que se espera.

**Libertad tecnológica**

Este es sin duda un aspecto muy importante. Gracias a esto podemos disponer de un sistema donde las diferentes piezas pueden estar desarrolladas con tecnologías totalmente diferentes.

Los microservicios se comunican entre sí únicamente a través de API REST bien definidas, por lo que la tecnología subyacente es irrelevante en el proceso de comunicación. Es posible tener componentes escritos en JavaScript, Python, PHP, Java, etc., y esto no será un problema, ya que la comunicación se hará a través de una API y de mensajes (normalmente, JSON).

**Tolerancia a errores**

En un sistema centralizado o monolítico, nos encontramos con un punto único de fallo. Si uno de los servicios del núcleo falla, es muy probable que toda la aplicación falle.

Sin embargo, con un enfoque de microservicios esto no debería pasar, ya que cada componente es independiente de los demás a nivel físico y de infraestructura. Esto es debido a que, en este tipo de arquitectura, se busca que el acoplamiento entre las partes sea lo más bajo posible.

Componentes principales de una arquitectura de microservicios

Cuando trabajamos con microservicios es muy posible que su número se vaya incrementando a lo largo del tiempo y esto, sin duda, crea una dificultad en la gestión del sistema. Para lidiar con esta complejidad, existen **varios componentes que suelen estar presentes** en cualquier arquitectura de este tipo.

**Servicio de registro**

Proporciona independencia a los microservicios, ocultando la ubicación física o lógica de estos. Este servicio, que también se suele llamar **de descubrimiento** (service discovery), es un componente que mantiene constantemente una lista o registro de todos los microservicios activos en el sistema. Al desplegarse y arrancar, todos los microservicios se registran obligatoriamente en este servicio y así se guarda la información de cada uno de ellos.

Es muy habitual en estas arquitecturas que las direcciones IP de los diferentes microservicios cambien constantemente. Gracias a este servicio de registro, eso no es un problema, ya que en **todo momento se sabrá en qué dirección se encuentra** un microservicio, sin importar las veces que cambie de dirección IP (sería necesario un redespliegue para ello y, por consiguiente, un nuevo registro en el servicio).

**Balanceadores**

Dado que los microservicios pueden escalar de forma horizontal, es necesario que exista algún mecanismo para saber a qué microservicio enviar las peticiones entrantes. Si tenemos tres copias de un mismo microservicio y dos de ellas se encuentran atendiendo peticiones y una no, el balanceador se encargará de redirigir a la tercera copia las peticiones entrantes.

Como podrás imaginar, los balanceadores están **estrechamente ligados al servicio de registro,** ya que necesitan conocer en todo momento las direcciones IP y puertos de los microservicios desplegados para redirigir correctamente las peticiones.

**Servidor de configuración**

Aunque puede tomar muchas formas, este servidor se encarga de alojar configuraciones comunes a todos los microservicios del sistema, de forma que siempre esté disponible para ellos.

**Servidor perimetral**

Es el punto de acceso de cualquier llamada externa. Funciona como un proxy inverso. Es un punto de entrada que recibe todas las peticiones y las redirige al sitio correcto, al hacer uso de los balanceadores.

**Gestión de** ***logs***

En un modelo donde tenemos múltiples componentes es imprescindible algún tipo de sistema que nos permita ver todas las trazas de ejecución de los microservicios. En dos niveles:

* **A nivel de microservicio**. Necesitamos ver todos los logs que un microservicio va generando.
* **A nivel de petición.** Una petición del usuario probablemente se dividirá en llamadas a múltiples microservicios. Es muy útil disponer de un mecanismo que nos permita ver todos los lugares por los que pasó una petición concreta antes de completarse.

Uno de los mejores ejemplos en cuanto a tecnologías que implementan todo lo anterior es Netflix. Esta conocida empresa implementó, desde hace ya varios años, una arquitectura orientada a microservicios y liberó (bajo licencia open source) algunos de los componentes de su arquitectura. Entre ellos:

* Eureka (service discovery).
* Ribbon (balanceador).
* Zuul (servidor perimetral).
* Hystrix y Turbine (monitorización).

Estos son ampliamente utilizados con uno de los frameworks de Java para el back-end, Spring. Spring dispone de un servidor de configuración (config-server).

Se puede indagar más sobre este tema en Spring Cloud Netflix, recurso que se encuentra disponible en A fondo.

### **Orquestación y coreografía**

Otro punto muy importante para tener en cuenta cuando trabajamos con microservicios es **definir la secuencia de llamadas o de operaciones a realizar** para satisfacer una petición concreta del usuario. Por ejemplo, en una tienda online, la compra de un artículo puede implicar el trabajo de varios microservicios. ¿Cómo se decide en qué orden se ejecuta cada cosa? A esto lo llamamos **orquestación.**

La orquestación consiste en disponer de un componente que recibirá las peticiones de más alto nivel e irá realizando llamadas a todos los microservicios involucrados en el orden necesario.

Siguiendo el ejemplo de una compra de una aplicación web, podríamos encontrarnos el siguiente flujo:

* El cliente confirma la compra.
* Se efectúa una llamada a una pasarela de pago para hacer efectiva la compra.
* Se envía una notificación al almacén correspondiente para preparar el pedido.
* Se emite una factura que se guardará en la sección de documentación del cliente.
* Se envía un correo electrónico al cliente indicando que la compra se ha completado correctamente.

Como podrás intuir, en este proceso podríamos **identificar diferentes microservicios:**

* Un microservicio que se encargue de realizar peticiones a una pasarela de pago para efectuar los cobros.
* Un microservicio encargado de notificar a diferentes partes de la empresa.
* Un microservicio encargado de generar documentación y almacenarla para su posterior revisión.
* Un microservicio encargado de enviar correos electrónicos.

Para que todas las acciones anteriores se realicen en el orden correcto, es necesario disponer de un orquestador que vaya encadenando las llamadas a los microservicios correspondientes y evalúe sus respuestas para continuar o detener el proceso. Las **ventajas de la orquestación** son las siguientes:

* Es una forma de mantener el control de las ejecuciones sencilla y fácil de sostener.
* Gran capacidad de control en flujos de trabajo síncronos.

Sin embargo, esta forma de organizar la ejecución de los microservicios también presenta algunas **desventajas:**

* Se crean **dependencias estrictas entre microservicios.** Si hay una gran afluencia de peticiones en los microservicios que contactan con la pasarela de pago, el resto del flujo sufrirá una espera.
* Si hay un único orquestador, tenemos un **punto único de fallo,** con los problemas que eso conlleva.
* Al tratarse de un sistema síncrono, el **tiempo de respuesta total de la operación** será la suma de los tiempos de operación de cada uno de los microservicios subyacentes.
* Va **en contra del principio de los microservicios** donde, idealmente, todos los componentes son independientes y el acoplamiento es bajo.

Aunque la orquestación es hoy en día uno de los mecanismos más usados, presenta las desventajas indicadas y es importante ser consciente de ellas. Habrá flujos de trabajo donde sea necesaria una aproximación síncrona. En el ejemplo está claro que no podemos proceder con un pedido si el pago no se completa.

Sin embargo, una vez se ha completado el pago, las acciones siguientes (3, 4 y 5) podrían realizarse de forma totalmente paralela, ya que no hay ningún tipo de dependencia entre ellas. En este caso, se puede utilizar una alternativa a la orquestación, llamada **coreografía**, la cual está basada en eventos.

La **coreografía** recibe ese nombre porque se basa en la idea de la danza, donde cada bailarín ejecuta sus movimientos de forma independiente, a la vez que el resto de los bailarines. Lo que se busca en este caso es maximizar la independencia de los microservicios y reducir lo más posible el acoplamiento que veíamos antes. En este caso, los microservicios deberán estar suscritos a un topic o a una categoría de eventos concretos. Cuando se reciba uno de esos eventos, el microservicio comenzará a trabajar.

Las **ventajas de la coreografía** frente a la orquestación son las siguientes:

* Se consigue un mayor desacople, lo que permite que el flujo de trabajo no se detenga si un componente se cae o queda inactivo.
* El flujo de trabajo ahora es asíncrono y distribuido, por lo que no existe un punto único de fallo.

Si se quiere profundizar sobre este tema, se puede acceder a Orquestación y coreografíaen la sección A fondo.

Observemos ahora un diagrama de los componentes que podrían formar parte de una arquitectura orientada a microservicios.

A diagram of a service

Description automatically generated

Figura 6. Arquitectura basada en microservicios. Fuente: Hinds, 2015.

Podemos destacar los siguientes aspectos:

* Existe un consumidor del sistema distribuido (color naranja).
* El consumidor siempre efectúa llamadas contra el servidor perimetral (edge server), que en este caso utiliza la tecnología Zuul.
* Dentro del servidor perimetral, se hace uso de un balanceador de carga con Ribbon para poder acceder al microservicio product-composite. Este microservicio es el encargado de la orquestación.
* El orquestador hace uso de Eureka (el encargado de ofrecer Service Discovery) y de un balanceador para localizar la dirección del microservicio con el que debe contactar.
* Además del orquestador, existen tres microservicios, uno para productos, otro para recomendaciones y otro para reseñas.