Presentación TFG

Víctor Iranzo

8 de septiembre de 2018

1. Presentación

Con el permiso del jurado, comenzaré con la exposición de este trabajo de final de grado.

2. Introducción

2.1. Motivación

Uno de los objetivos principales de una organización debe ser desarrollar sistemas de calidad. Se debe poner el foco en emplear una arquitectura de software que se adapte a las necesidades del negocio.

Actualmente, trabajo en una organización que apuesta por el uso de arquitecturas basadas en microservicios. Como motivación personal, busco profundizar mi conocimiento en las tecnologías asociadas a estos.

Además, queremos presentar un caso de estudio con la suficiente envergadura para reflejar su realidad: sus ventajas, inconvenientes, cuándo es recomendable su uso, etc.

2.2. Objetivos

Los objetivos que persigue este trabajo son los siguientes:

- Desarrollar una misma aplicación para el comercio electrónico siguiendo dos arquitecturas diferentes: una basada en microservicios y otra monolítica.

- Comparar el proceso de desarrollo de ambos sistemas a lo largo del ciclo de vida del software.
- Evaluar cómo realizar diferentes modificaciones durante el mantenimiento de ambas aplicaciones.
- Examinar ambas arquitecturas respecto a los requisitos no funcionales de disponibilidad, tolerancia a fallos, utilización de recursos y capacidad para ser reemplazado.

2.3. Microservicios

Antes de entrar en materia, vamos a definir primero qué son los microservicios. Los microservicios son servicios pequeños y autónomos que cooperan entre ellos. Vamos a desglosar esta definición:

Un servicio es un conjunto de funcionalidades que se expone a los clientes.

En cuanto a su tamaño, es más importante respetar los principios de alta cohesión y bajo acoplamiento que preocuparse por hacer los servicios lo más pequeños posible.

Por último, la autonomía de los microservicios significa que cada uno puede evoluciona de forma independiente al resto.

2.4. Aplicación monolítica

En contraposición, una aplicación monolítica se define como aquella cuyos módulos no pueden ejecutarse de forma independiente.

Las arquitecturas monolíticas se caracterizan porque:

- Conforme aumenta su tamaño, aumenta su complejidad, haciendo más costosa la introducción de cambios.
- No se puede escalar cada módulo de acuerdo a sus necesidades. En su lugar, se debe escalar el sistema como una única pieza o monolito.
- Se limita el uso de lenguajes y herramientas, haciendo que en todos los módulos se deban emplear prácticamente los mismos.

3. Proceso de desarrollo

A continuación, analizaremos el impacto del uso de los microservicios a lo largo de las actividades del proceso de desarrollo.

3.1. Especificación de requisitos

En la especificación de requisitos, los requisitos no funcionales son restricciones impuestas sobre el sistema a desarrollar, estableciendo por ejemplo como ha de ser de fiable, escalable, etc. Por este motivo, conducen hacia la elección de una arquitectura u otra.

Los siguientes atributos de calidad de la ISO 25010 se pueden expresar como requisitos no funcionales que nos plantearían emplear microservicios.

3.2. Diseño del sistema

El diseño de una solución basada en microservicios está estrechamente ligado al enfoque de diseño guiado por el dominio.

El dominio representa lo que hace una organización, que se divide en contextos bien delimitados recogiendo las áreas que son independientes unas de otras.

Dentro de cada contexto se emplea un lenguaje ubicuo: un lenguaje común que emplean tanto los desarrolladores como los expertos del dominio.

En la figura se muestran dos contextos bien delimitados que pertenecen a un mismo dominio, la gestión de un hospital. En dos contextos diferentes pueden aparecer conceptos nombrados diferentes pero que refieren a la misma entidad del mundo real, como ocurre con Paciente y Cliente. Dentro de cada contexto, cada concepto tendrá unos atributos diferentes. Sin embargo, existen algunos atributos como el Nombre, que aparecen en ambos conceptos. Este tipo de atributos requieren que el sistema sincronice los cambios del atributo que se hagan en un contexto hacia el otro.

3.3. Implementación del sistema

Respecto a la implementación del sistema, cada microservicio puede desarrollarse empleando una tecnología diferente. Por ejemplo, en la figura se muestran diferentes microservicios realizados con distintos lenguajes

de programación y que emplean bases de datos diferentes.

En cuanto a la integración entre los microservicios, se deben emplear mecanismos de comunicación ligeros a través de la red. Entre ellos, podemos citar:

- RPC: permite ejecutar una llamada a un servicio como si de una llamada local se tratara. Su uso no está tan ampliamente soportado a como es el caso de REST.
- REST: emplea los verbos definidos en HTTP para poder acceder a un recurso de un servicio.
- Integración basada en eventos: fomenta la escalabilidad y la resiliencia y reduce el acoplamiento entre los microservicios. No obstante, requiere aprovisionar nuevas infraestructuras y añade complejidad a la hora de razonar sobre el sistema.

3.4. Pruebas

En cuanto a las pruebas, vamos a clasificarlas conforme lo hace Newman en Building Microservices.

- Las pruebas unitarias involucran a un único método, por lo que son sencillas de implementar y rápidas de ejecutar.
- Las pruebas de servicio evalúan las diferentes funcionalidades que expone un microservicio. Pueden sustituir uno de los servicios de los que depende por un fake.
- Las pruebas de extremo a extremo cubren un caso de uso que involucra a más de un servicio. Su correcta seguridad dan un mayor nivel de confianza, pero suelen ser pruebas frágiles.

Conforme se sube la pirámida que vemos en la figura, mayor número de pruebas se recomienda tener de ese tipo.

3.5. Despliegue

Para el despliegue vamos a comparar el uso de la virtualización tradicional con el uso de contenedores.

Las máquinas virtuales requieren un sistema operativo completo, mientras que los contenedores emplean el kernel de la máquina sobre la que se despliegan.

Además, para el uso de máquinas virtuales se de be incluir un hipervisor, que reparte recursos de la máquina física como la CPU o la y permite al usuario la gestión de las máquinas virtuales existentes.

Comparando un despliegue con otro, las máquinas virtuales tardan más tiempo en desplegarse y consumen más recursos que un contenedor.

3.6. Fase de mantenimiento

Por último, la fase de mantenimiento suele ser realizada por el mismo equipo que implementa el sistema para acercar a clientes y desarrolladores, tal como hacen empresas como Amazon.

Además, se deben garantizar los acuerdos de nivel de servicio monitorizando la salud de los servicios y se ha de controlar la deuda técnica fruto de un proceso de desarrollo más rápido.

4. Estado del arte

4.1. Contenedores

Un contenedor es una unidad de aislamiento que puede acceder al sistema operativo de la máquina donde reside.

Hemos estudiado dos tecnologías en la memoria.

La principal ventaja de los contenedores Linux es que son una implementación muy ligera. Sin embargo, limitan al uso de Linux como base del entorno porque están muy acoplado a su kernel.

En cuanto a los contenedores Docker, son los usados por alrededor del 25 % de las organizaciones. Su uso conlleva la construcción de imágenes que permiten la creación de un contendor de manera reproducible. Además,su facilidad hace que las tareas del despliegue puedan ser llevadas a cabo por diferentes equipos.

4.2. Orquestadores

Un orquestador es una herramienta para la gestión de clústeres y contenedores. Permiten gestionar las imágenes que originan los contenedores, los hosts, las redes de contenedores, etc.

De nuevo, se han estudiado los dos más empleados.

Kubernetes se emplea principalmente para especificar el número de replicas que se desea tener simultánemanete de un pod, que representa un conjunto de contenedores. La herramienta es buena para asegurar la disponiblidad de un servicio, pero no garantiza la escalabilidad de esta, que debería basarse en reglas.

Docker Swarm es el orquestador nativa propuesta por Docker. Gracias a esto, está completamente integrado en la línea de comandos de Docker.

5. Caso de estudio

5.1. Especificacióm del caso de estudio

El objetivo desarrollo es realizar una aplicación móvil con Xamarin donde destacan los siguientes casos de uso:

- Realizar pedido.
- Ver factura de un pedido.
- Crear una incidencia.

A partir de la descricpión que se da en la descripción del caso de estudio, se puede obtener el siguiente diagrama de dominio. Un pedido está compuesto de productos, al igual que una incidencia contiene comentarios que intercambian los clientes con los empleados de la tienda.

5.2. Proceso de desarrollo

El desarrollo de la aplicación siguiendo ambas alternativas se divide en tres grandes bloques: el desarrollo del back-end monolítico (en azul), el desarrollo del front-end (en verde) y el desarrollo del back-end basado en microservicios (en naranja).

El desarrollo del sistema comienza el 19 de junio y dura hasta el 29 de julio. En el cronograma se pueden ver las prinicpales tareas que se han llevado a cabo.

El back-end monolítico es el bloque que se alarga por más tiempo porque es donde se han evaluado diferentes detalles de implementación.

Después, explotando ese back-end se construyó la aplicación móvil, cuyo código debía ser el mismo para comunicar con el sistema de microservicios o el monolítico.

Por último, en naranja aparece las tareas de refactorización del sistema monolítico en uno basado en microservicios, de menor duración porque consiste en la reorganización del código.

5.3. Arquitectura monolítica

Como arquitectura para el sistema monolítico se va a seguir la arquitectura de 6 capas que se emplea en mi organización.

- En la capa de Contratos se definen las interfaces con las operaciones que desde el exterior se pueden solicitar a la parte servidora. También aquí se definen los objetos para la transferencia de datos.
- La capa de persistencia es la única que accede directamente a la base de datos. Para hacerlo emplea Entity Framework Core, que mapea el esquema relacional en objetos C#.
- La capa de dominio contiene las entidades de dominio que hemos analizado anteriormente.
- La capa de aplicación contiene la lógica del negocio, la implementación concreta de las interfaces definidas en la capa de contratos.
- La capa de servicios es el punto de entrada al sistema, donde se definen los métodos HTTP asociados a cada método de la interfaz de contratos. La capa de servicios delega en todo lo posible en la capa de aplicación.
- La capa de proxy, que se ejecuta en los consumidores de la parte servidora, como la UI. Se emplea para realizar llamadas HTTP al servidor a través de código C#.

5.4. Herramientas para la construcción

Para la construcción del sistema se han usado un gran número de herramientas, ya que se persigue desarrollar el sistema lo más similar a un software profesional. Entre estas herramientas destacamos:

- Entity Framework Core: para la transformación del esquema relacional en objetos en el código.
- Swagger UI: para la creación de una API interactiva para probar la parte servidora.
- CodeMaid y StyleCop: para controlar la calidad del sistema desarrollado.
- Docker, Kubernetes y Azure: para el despliegue del sistema en un entorno de producción.

5.5. Descomposición en microservicios

Pasamos ahora a la descomposición del sistema en microservicios. Siguiendo el enfoque del diseño guiado por el dominio, podemos extraer a partir del modelo de dominio los siguientes contextos bien delimitados.

Cada contexto es firme candidato a convertirse en un microservicio, tal y como veremos. Otras descomposiciones son posibles, como por ejemplo incluir los Productos en un contexto separado al de pedidos. Sin embargo, no lo vamos a hacer así porque solo los pedidos están relacionados con los productos.

5.6. Arquitectura basada en microservicios

En esta figura podemos ver un diagrama de componentes del modelo, donde se han implementado como microservicios los mismos contextos que antes señalábamos.

Cada microservicio es dueño de sus datos, por lo que tendrá su propia base de datos.

Además, código que existía compartido entre los módulos de incidencias y pedidos relacionado con operaciones CRUD o el acceso a persistencia, se extraerá a una librería para que ambos microservicios puedan seguir empleándolo sin duplicar el código.

Por último, para evaluar que se pueden emplear diferentes tecnologías en cada microservicio, se van llevar a cabo una serie de modificaciones respecto al sistema monolítico que ahora detallaremos.

5.7. Cambios respecto a la solución monolítica

La primera observación que se puede realizar es que al no almacenar todos los datos dentro de una única base de datos, la integridad referencial entre elementos de distintos contextos se pierde. Por ejemplo, ya no se puede tener una clave externa de la tabla pedidos a la de usuarios porque esas tablas ahora están en diferentes bases de datos.

Después, se ha desarrollado un microservicio en otro lenguaje de programación. Se trata del microservicio de notificaciones en Java. Se ha elegido este por ser de tamaño reducido y fácil de migrar.

También se ha desarrollado un microservicio empleando una tecnología de base de datos distinta. En este caso, se ha empleado Firebase, una base de datos clave-valor desarrollada por Google. Para emplearla, se han mantenido las interfaces existentes en la capa de persistencia y se ha modificado su implementación para acceder a la base de datos a través de llamadas HTTP en lugar de a través de Entity Framework Core.

Por último, hemos explicado que una de las mayores ventajas del uso de microservicios es que pueden evolucionar de forma independiente. Esto se plasma en el siguiente archivo de versiones donde se ve como la versión de cada servicio es diferente y evoluciona según sus necesidades.

5.8. Aplicación móvil desarrollada

Por último, me gustaría mostrarles el resultado final de este proceso, que es la aplicación móvil desarrollada.

De izquierda a derecha se observan capturas reales para la visualización de una incidencia, la visualización de una factura, la creación de un pedido y la búsqueda de productos.

Aunque se ha empleado una herramienta como Xamarin.Forms para hacer que esta aplicación sea multiplataforma, nos hemos centrado exclusivamente en el sistema operativo Android.

6. Evaluación

6.1. Balance de los microservicios

Podemos realizar el siguiente balance sobre el uso de los microservicios:

- Es una arquitectura que escala mejor porque lo hace a nivel de microservicio y de acuerdo a sus necesidades reales.
- Fomenta la alta cohesión y el bajo acoplamiento porque agrupa conjuntamente lo que en el dominio está relacionado y separa en otros contextos lo que no.
- Como hemos visto, permite que cada microservicio evolucione, se modifique y se despliegue de forma independiente al resto.

Por otro lado, podemos citar los siguientes inconvenientes asociados a su uso:

- La descomposición en microservicios es un proceso arduo que se mejora de forma iterativa. Supone un sobrecoste. Es muy difícil acertar en la descomposición a la primera, porque pueden surgir nuevos requisitos que nos replanteen el sistema completo.
- La depuración es más costosa porque el número de piezas que intervienen para ofrecer una funcionalidad es mayor. En un caso de uso, pueden participar dos o tres microservicios que se deben instanciar en el proceso de depuración.
- No se pueden establecer restricciones de integridad referencial ni operaciones atómicas porque existe más de una base datos. Esto conduce a la llamada consistencia eventual, que puede producir que en algún momento se observe en el sistema un dato incorrecto debido a que todavía no se ha propagado su actualización.

6.2. Otras consideraciones

- Cada microservicio puede estar desarrollado internamente de forma diferente, pero los microservicios cooperan entre ellos. Los mecanismos de integración que establecen (REST, colas, RPC) deben estandarizarse en todo el sistema para facilitar su consumo.

- Por último, puede existir código duplicado por usar microservicios ya que algunas clases tienen que existir en todos ellos para su ejecución. Una forma de abordar este problema es mediante la generación automática de código, que aumenta la velocidad de desarrollo ya que un cambio puede suponer simplemente actualizar un modelo origen y regenerar código.

6.3. Conclusiones

Para finaliar, vamos a repasar las conclusiones de este trabajo:

- Se ha desarrollado satisfactoriamente el caso de estudio siguiendo una arquitectura basada en microservicios y una monolítica.
- En cuanto al proceso de desarrollo, de una misma especificación hemos construído dos sistemas distintos. Actividades como la implementación, el despliegue o las pruebas en una solución basada en microservicios han resultado ser más desafiantes. Otras, como las de diseño, cobran mayor relevancia.
- En general, el mantenimiento es más simple en un sistema basado en microservicios.
- Los RNFs que hemos mencionado son más fáciles de alcanzar en una arquitectura basada en microservicios.

Gracias por su atención. Pasemos ahora al turno de preguntas.