# ¿Sale rentable emplear microservicios en tu empresa?

Implementar un sistema basado en microservicios directamente siempre es más costoso y complejo. Por ejemplo, establecer las interfaces o contratos con los que comunican dos servicios es más complejo. Mientras que en un sistema monolítico estas llamadas son intraprocedurales, en un sistema basado en microservicios son interprocedurales. Esto hace que su depuración sea más complicada.

En el contexto de mi empresa, hemos construido herramientas que nos permiten agilizar el desarrollo de los microservicios. En la memoria se menciona que existe código duplicado o muy similar en los diferentes microservicios. Por ejemplo, clases para el registro de dependencias, para transformar objetos de entidades a DTOs, clases para configurar un servicio. Una forma de abordar este problema es mediante la generación automática de código. Las herramientas de mi empresa se basan en el uso de modelos para la generación automática de código.

Por ejemplo, podemos modelar las entidades de dominio y a partir del modelo generar tanto las capas de dominio como de persistencia. Podemos también modelar lo que llamamos “acciones ad-hoc”, que son métodos que el microservicio expone. Para estos métodos generamos el código necesario para que sean invocadas en todas las capas, salvo la lógica del método que debe ser implementada por un programador.

# ¿Por qué es necesario monitorizar los SLA?

Un acuerdo de nivel de servicio es un acuerdo entre un proveedor de un servicio y su cliente para acordar la calidad de dicho servicio. Este acuerdo se puede expresar con muchos parámetros: el tiempo medio en que el servicio de soporte responde a la llamada de un usuario, el tiempo medio que tarda en resolverse una incidencia o la disponibilidad del servicio.

La disponibilidad se puede medir como el porcentaje de tiempo que un servicio ha estado en funcionamiento durante un año. Este porcentaje se suele expresar en los sistemas altamente disponibles como el número de nueves. Por ejemplo, 2 nueves es una disponibilidad del 99% y se corresponde con un servicio que puede estar inoperativo sobre 3,5 días a lo largo del año; 3 nueves se corresponden con un servicio que puede estar alrededor de 9 horas.

Para garantizar esta disponibilidad es necesario monitorizar los servicios desplegados en producción. De acuerdo con nuestro número de 9s necesitaremos unos mecanismos de monitorización u otros. Los más simples consisten en un dashboard donde se muestra visualmente (con colores rojo y verde) el estado de salud de los servicios. Los más complejos son aquellos que son capaces de detectar un servicio inoperativo e instanciarlo de nuevo automáticamente.

# ¿Por qué la capa de Persistencia tiene acceso a la capa de Dominio?

El motivo principal es el uso de Entity Framework Core. Los objetos de C# en los que se transforma el esquema relacional de base de datos son las clases que se encuentran en la capa de dominio. Además, cuando la capa de Aplicación pregunta por una operación sobre una entidad, lo hace en términos de una entidad del dominio. Es por eso que tanto la capa de Aplicación como la de Persistencia tienen una referencia a la capa de Dominio, porque comunican con las entidades de la capa de dominio.

Un ejemplo, cuando un usuario solicita un conjunto de valores a la capa de aplicación, está contacta con la de persistencia para realizar la operación de lectura. Empleando Entity Framework y consultas LINQ, la capa de persistencia obtiene esas entidades expresadas ya como objetos del dominio. Le devuelve a la capa de aplicación un conjunto de objetos del dominio y es la capa de aplicación quien transforma eso en una colección de objetos para la transferencia de datos o DTOs, que será lo que finalmente devuelva al usuario.

# ¿Por qué emplear una arquitectura con tantas capas?

En la memoria se hace una reflexión sobre esto. No hay una arquitectura que sea una panacea para cualquier sistema. Una arquitectura de 6 capas puede resultar excesiva para microservicios como el de informes, ya que no requieren persistir datos ni modelan, como tal, entidades del dominio. Lo mismo ocurre en el servicio de notificaciones.

Para el resto del sistema, la arquitectura ofrece algunas características y hacen una correcta división de responsabilidades. Cada capa existe por un motivo. Como he dicho, es la arquitectura que empleamos en mi empresa.

Las capas de persistencia y aplicación son obvias y suelen estar así divididas hasta en los sistemas más básicos de 3 capas. La capa de dominio es muy “fina” y por este motivo podría combinarse con la de persistencia. Sin embargo, podría incluir más lógica de la que en el caso de estudio tiene. Por ejemplo, podría incluir validaciones de los campos de una entidad, como que el valor de un campo está entre cierto rango permitido. Son validaciones que pertenecen al dominio.

En cuanto a la capa de Servicios, es el punto de entrada del sistema, el método Main. Está separada de la de aplicación para separar en dos la lógica del sistema, asociada al negocio, de la definición de los métodos HTTP que pueden ser invocados, más relacionado con la implementación del sistema, el lenguaje de programación empleado…

Por último, la capa de proxy existe para dar flexibilidad a la arquitectura. Porque nos podría servir para desplegar el componente de software como un servicio (en otro proceso) o como una librería, si se referencia desde proxy directamente a la capa de aplicación. En mi organización definimos 3 tipos de proxies: el Online, que siempre invoca a la capa de Servicios a través de llamadas HTTP, el OnlineOffline, que es capaz de detectar cuando el sistema está caído y pasar de realizar llamadas HTTP a emplear la capa de Aplicación que con el proxy se ha desplegado, y el proxy InProcess, que siempre invoca a la capa de aplicación y se despliega como si de una librería se tratara.

# ¿Por qué se han seleccionado esos requisitos?

Mi tutor, Patricio Letelier, y yo hicimos una revisión de los atributos de calidad de la ISO 25010 y seleccionamos estos porque eran los más evidentemente relacionados con los microservicios. Eran aquellos que consideramos que más fáciles eran de satisfacer por una arquitectura basada en microservicios frente a una arquitectura monolítica.

Algunas subcaracterísticas como la modularidad o reusabilidad las descartamos del análisis por ser demasiado obvias. En su lugar, cogimos la característica de Mantenibilidad y la evaluamos con un enfoque más práctico, presentando situaciones de mantenimiento y la forma en qué se abordarían en cada alternativa.

Otras subcaracterísticas como la coexistencia, la interoperabilidad o la capacidad de recuperación hubieran sido también interesantes de medir. Quizás incluirlas habría excedido ya el esfuerzo que se espera de un trabajo de final de grado.

# ¿Por qué escogiste emplear Kubernetes?

Fue una elección personal más que una decisión técnica. En la industria tanto Kubernetes como Docker Swarm son muy empleadas. Los requisitos que esperábamos del orquestador a utilizar eran muy básicos. No nos planteamos medir como los sistemas escalaban. Simplemente buscábamos tener una herramienta con la que desplegar el sistema basado en microservicios como un clúster en un entorno de producción.

Había escuchado muchas veces hablar de Kubernetes y tenía una curiosidad personal. Del trabajo realizado creo que al final hicimos una exploración más a lo ancho que en profundidad. Por ejemplo, hemos presentado varias formas de configurar el sistema con diferentes objetos de Kubernetes. Sin embargo, no hemos profundizado en evaluar como escala y el uso de estos objetos ha sido muy básicos. Como autocrítica digo que quizás no ha sido el uso el óptimo.

De la experiencia de su uso extraigo que es sencillo de emplear. Expresas tu sistema a muy alto nivel y con muy pocos comandos lo tienes desplegado.

# ¿Por qué hiciste el sistema en Kubernetes así?

Del trabajo realizado creo que al final hicimos una exploración más a lo ancho que en profundidad. Por ejemplo, hemos presentado varias formas de configurar el sistema con diferentes objetos de Kubernetes. Sin embargo, no hemos profundizado en evaluar como escala y el uso de estos objetos ha sido muy básicos. Como autocrítica digo que quizás no ha sido el uso el óptimo.

De todas las tecnologías empleadas, Kubernetes era la que más desconocía y una de las que mayor complejidad tenían. Me leí el libro de Kubernetes escrito por el Director de Servicios en la Nube de Google, que quizás se quedaba en un enfoque demasiado teórico. La configuración que apliqué la extraje de ese libro. Leí más literatura, pero fue esta configuración la que más me gustó porque me parecía la que hacía a cada componente más independiente. Además, se asemejaba a la idea que tenía en la mente del patrón API Gateway.

En definitiva, mi experiencia con la herramienta no es mucha, pero se ha desarrollado una configuración que satisfacía las necesidades que teníamos: hacer un despliegue en un entorno de producción que pudiera ser empleado por la aplicación front-end.

# ¿Por qué hiciste las modificaciones en el sistema basado en microservicios?

Inicialmente, no nos planteamos como tal evaluar lo que catalogamos como programación y persistencia políglota. También he visto que se le da el nombre a esto de heterogeneidad tecnológica. Sin embargo, cuando ya estábamos enfrascados en el desarrollo del caso de estudio lo vimos interesante.

Algunos cambios como hacer la arquitectura interna del servicio de informes más sencilla provenían más de la lógica que de las ganas de explorar esta característica. No tenían sentido algunas capas, entonces era natural no incluirlas.

Podría haber venido aquí y decir que uno de los objetivos del trabajo era evaluar esto, pero no era lo que nos planteamos inicialmente. Creo que en un trabajo como este se deben mantener los objetivos estáticos y no modificarlos a mitad del caso de estudio.

# ¿Comentarios sobre el proceso de desarrollo?

En la memoria, hay una parte centrar muy importante que habla sobre esto: los capítulos con el Plan de trabajo, la especificación, la implementación de ambas soluciones, etc.

La especificación es la única actividad antes de la elección de una arquitectura. Por eso, es la misma para un sistema y otro. Sin embargo, como hemos dicho los RNFs son los que nos conducirán hacia una arquitectura u otra. En nuestro caso de estudio, la especificación no daba ninguna restricción, nos centramos sobre todo en detalles de funcionalidad o RF. Teníamos libertad para emplear una arquitectura u otra según la especificación. En una situación real, esto no suele ser así.

Sobre la actividad de diseño, en el sistema monolítico simplemente decidimos la arquitectura a capas que íbamos a emplear y la aplicamos en la fase de implementación. En el sistema basado en microservicios, se añade un paso extra que es extraer los contextos del dominio y dar a cada microservicio una arquitectura interna. Cobra mayor relevancia esta actividad.

En la implementación no hay mayores diferencias. Simplemente se reorganizó mucho código en la refactorización de una alternativa en la otra. También se hicieron algunos cambios relacionados a la hora de consumir lo que ahora eran otros servicios, como es inyectar su proxy.

En la actividad de pruebas, las pruebas unitarias tuvieron la misma implementación gracias a que el sistema monolítico estaba bien modularizado y era fácil de probar. Las pruebas de integración también fueron fáciles de adaptar gracias al uso de fakes, como el del servicio de seguridad. Las pruebas de extremo a extremo si que resultaron ser mucho más desafiantes.

Por último, el despliegue también resulto más desafiante en el segundo sistema debido a la complejidad de las nuevas herramientas como Kubernetes y a la existencia de más piezas a desplegar.

La fase de mantenimiento solo se evaluó de forma hipotética.

# Relación con las metodologías ágiles

Las metodologías ágiles promueven, por ejemplo, los equipos autoorganizados y multifuncionales. Esto encaja perfectamente con el enfoque de los microservicios. Cada microservicio es desarrollado íntegramente por un equipo, desde la persistencia hasta la interfaz de usuario. Además, están autoorganizados porque tienen total libertad en la implementación concreta del servicio (lenguaje de programación, arquitectura interna) siempre que permitan integrar su servicio con el resto. Estos mecanismos de integración pueden estar estandarizados para todos los equipos. En este sentido, el arquitecto de software es un puente que facilita la colaboración entre los diferentes equipos.

Otra característica de las metodologías ágiles es el desarrollo iterativo e incremental. Se pueden hacer iteraciones donde cada vez se refinan más los detalles de un servicio o una iteración se puede corresponder con el desarrollo completo de un servicio. Además, el desarrollo del sistema completo es incremental, porque se va completando conforme se terminan los microservicios que lo constituyen.

# ¿You build, you run it?

Es natural que quien construye un mantenimiento sea quien lo mantenga, ya que es quien mejor lo conoce. Así lo he visto en algunas asignaturas de la carrera como Mantenimiento de Software.

De acuerdo con la ley de Conway, una organización que diseña un sistema producirá un diseño que será una copia de su estructura organizacional. Si por ejemplo dividimos en dos equipos a nuestros desarrollados, un equipo de back-end y uno de front-end, el diseño de la aplicación estará dividido de forma muy evidente en estas dos partes. Sin embargo, cualquier cambio que requiera modificar ambas partes requerirá la coordinación de ambos equipos. La coordinación suele alargar los tiempos necesarios para implementar un cambio.

Si se realiza un equipo multifuncional por cada microservicio, el cambio que antes involucraba a 2 equipos pasa a involucrar solo a uno. Todo queda en casa. Este mismo argumento se puede emplear para justificar que el mismo equipo sea quien realice el mantenimiento. Si es un equipo diferente, puede que se cree una comunicación continuada entre el equipo mantenedor y el equipo de desarrollo con el fin de conocer mejor el sistema.

No se debe confundir tampoco con la actividad de pruebas, donde se suele recomendar que ciertas pruebas (especialmente manuales o de aceptación) las implemente otro equipo con el fin de explorar casos que el desarrollador no haya contemplado.

# ¿Cómo habéis estimado el tiempo para reemplazar un microservicio?

En el cronograma hay una tarea para el desarrollo de cada módulo del sistema monolítico y otra tarea para la refactorización de este módulo en un microservicio. Si sumamos estos tiempos para un microservicio como el de incidencias obtenemos que su desarrollo ha costado 14 días.

Si lo hacemos con el resto de los servicios y hacemos una media obtenemos que el tiempo de implementar el monolítico y refactorizarlo es de 10 días. Si lo visualizamos, deberíamos quizás descartar de esta media al servicio de notificaciones, que solo ha costado 2 días en realizar este proceso y se podría catalogar como un micro-micro-servicio.

En este cálculo no se incluyen la realización de los formularios asociados a cada microservicio.

No se puede afirmar que la suma de implementar un componente en el sistema monolítico y refactorizarlo sea igual al tiempo necesario para implementarlo directamente como un microservicio. Desarrollar un sistema de microservicios directamente siempre es más costoso porque, por ejemplo, evaluar así las interfaces entre los distintos microservicios es más costoso porque se debe realizar a través de llamadas interprocedurales. Se asumía la hipótesis de que el tiempo para desarrollar un componente es el mismo que el de reemplazarlo. Sin embargo, hay algunas tareas que nos ahorramos al reemplazarlo. No tenemos que evaluar de nuevo las interfaces entre los microservicios. Su construcción ya está refinada.

# Tiempo dedicado al TFG.

No puedo responder a esa pregunta de forma exacta. La información más precisa que tengo sobre el esfuerzo realizado es el número de commits que he realizado en cada repositorio. Como comento en la memoria, he creado un repositorio de código para cada uno de los back-end, para el front-end y para la redacción de la memoria.

El hecho de haberlo hecho con la memoria es porque me servía como una herramienta para el control de versiones de la memoria, donde podía etiquetar cada cambio a través del nombre que se le da al commit y donde podía ver las modificaciones hechas en cada cambio gracias a que he usado Latex.

* Memoria: 180 commits.
* Back-end monolítico: 140 commits.
* Back-end microservicios: 45 commits.
* Front-end: 45 commits.

En los commits de la memoria se incluyen también algunos resúmenes que hice sobre el libro de Building microservices y la preparación de la presentación.

Son un total de 410 commits. Aproximadamente, puedo hacer un commit cada hora, entonces se puede corresponder con 410 horas hombres.

Si hablamos del período en el que he realizado este esfuerzo, aproximadamente empecé en mayo a leer literatura. El 19 de Junio empecé con el caso de estudio, que duró hasta el 29 de Julio. Y la redacción de la memoria se alargó hasta mediados de Agosto.

# Colaboración con el tutor.

Con Patricio me he reunido al menos 6 veces. Tuvimos una primera reunión para acordar el tema del trabajo a realizar. La segunda ya concretamos más el contenido, los objetivos y el caso de estudio a realizar. La tercera fue una revisión del trabajo realizado hasta el momento, cuando ya llevaba escrita media memoria. Las 3 últimas reuniones han sido con más compañeros que también exponen hoy su trabajo de final de grado para preparar las presentaciones.

Además, hemos estado en continuo contacto. Y siempre ha estado dispuesto a echar una mano, reunirse con nosotros en horas intempestivas, etc. Como tutor se ha portado de 10. He tenido muy buena experiencia con el TFG y estoy muy satisfecho con el trabajo realizado.

# ¿MVVM?

El patrón arquitectónico Model-View-ViewModel (MVVM) divide la interfaz de usuario en 3 capas: la vista, empleando páginas XAML, los datos tal como se obtienen en su origen, también llamados el modelo, y el modelo de la vista, que conecta ambos y se utiliza para rellenar la vista.

La diferencia entre este patrón y otros como el Model-View-Controller (MVC) es que en MVC los datos con los que se llena la vista son los que se obtienen en el origen y es el controlador quien ha de adaptarlos a cómo se visualizan en la vista. En cambio, en MVVM la vista y el modelo de la vista suelen relacionarse a través de enlaces (bindings) en el propio XAML y se notifican mutuamente cuando alguna de sus propiedades cambia.

Sin embargo, usar este patrón puede ser excesivo para IUs muy sencillas. Algunas tecnologías que emplean MVVM son WPF o Xamarin. Algunas personas coinciden es que es una adaptación del patrón “Presentation model” propuesto por Martin Fowler.

# Orquestador vs coreógrafo.

A la hora de implementar un proceso de negocio, múltiples servicios deben participar. Para su coordinación, existen dos estilos arquitectónicos que se pueden seguir: la orquestación, basado en un componente que guía y dirige al resto de servicios como un director de orquesta, y la coreografía, donde cada parte sabe el trabajo que debe realizar y se coordina con el resto como bailarines realizando una coreografía.

Por una parte, el patrón orquestador es más sencillo de implementar, pudiéndose hacer con herramientas para el modelado de procesos de negocio. Sin embargo, al usarlo se otorga demasiada autoridad a uno de los servicios del sistema y el resto pueden llegar a transformarse en simples servicios CRUD.

Por otra parte, el patrón coreógrafo se basa en la comunicación asíncrona. El proceso se inicia mediante un evento al que los servicios que deban reaccionar se suscriben. Así se puede formar una cadena donde cuando uno de los servicios acaba puede comenzar la actividad siguiente mediante el envío de un evento. Su principal ventaja es que reduce el acoplamiento entre los servicios, pero puede llegar a dificultar la visión del proceso de negocio y requiere más trabajo para ser monitorizado.

En general, las soluciones que siguen el patrón orquestador son más difíciles de cambiar y se suelen programar de forma síncrona, por lo que puede ser perjudicial para procesos largos. Además, suponen un único punto de fallo. Es por esto por lo que se recomienda más el segundo patrón, mucho más flexible, aunque se puede optar por una solución híbrida.