UNIVERSIDAD REY JUAN CARLOS

PRÁCTICA I

DISEÑO Y ARQUITECTURA DEL SOFTWARE

Grupo 7

Grado en Ingeniería del Software

Alejandro Fernández San Román
Gabriel Fuentes Villasevil
Aurora María Fernández Basanta
David Ovidio Rubio Caballero
Juan Álvarez Loeches
Álvaro Sepúlveda Crespo

1. Tabla de roles.

Los distintos integrantes del grupo nos dividimos los roles de la siguiente forma:

1. Seniors: Alejandro Fernández San Román y Gabriel Fuentes Villasevil.
2. Cognitivos: Aurora María Fernández Basanta y David Ovidio Rubio Caballero.
3. Juniors: Juan Álvarez Loeches y Álvaro Sepúlveda Crespo.
4. Portavoz del equipo: Aurora María Fernández Basanta, con correo am.fernandez.2019@alumnos.urjc.es

2. Toma de requisitos

En *Figura 6: Enunciado de la práctica subrayado* del anexo podemos ver una captura del enunciado subrayado con los distintos colores para simplificar así el dilucidar qué requisitos había y cómo tomarlos, puesto así también para que se sepa cómo se han tomado los requisitos y de donde salen éstos.

A su vez, en *Figura 7: Tabla de requisitos final con el subrayado pertinente* del anexo, podemos también ver una captura de la tabla de requisitos resultantes con los colores haciendo referencia a qué parte del título han sido capturadas, también teniendo en el read.me del GitHub los requisitos junto con la decisión en la que han sido usados.

Por último, hay que mencionar que, en los requisitos se usó un color rosáceo apagado para indicar aquellas partes del requisito que están puestas para facilitar su diseño, a su vez también hay una parte en el enunciado, dedicada al componente de presentación que se ignoró pues, aunque tenían estructura de requisito no iban a ser usadas para la migración de la aplicación.

3. Desarrollo de las iteraciones

3.1 Iteración 1

3.1.1 Decisiones.

Las decisiones de la primera iteración fueron orientadas a tener ya decidido el estilo del programa a diseñar, que, a través del RF1 y RF2, llegamos a la conclusión que se ha de mantener el estilo por capas dividiendo la capa de negocio en microservicios.

Esta decisión fue única en la iteración pues sin tenerla clara no se podían tomar muchas más decisiones que se viesen reflejadas en el diseño, a parte nos dio pie al resto de decisiones tomadas en las siguientes iteraciones.

3.1.2 Discusión con los cognitivos.

Entre las opciones consideradas se llegó al acuerdo de escoger la segunda (Dividir todas las funcionalidades del previo estilo por capas en distintos microservicios) ya que no encontramos otras alternativas viables, y en el enunciado nos hablaban de utilizar microservicios en más partes del sistema y no solo en la capa de negocios.

3.1.3 Diseño tras las decisiones tomadas.

Para cumplir con la **Decisión 1**, hemos seguido un estilo por capas. Estas son: la **CapaBaseDeDatos** que contendrá las bases de datos necesarias, **CapaPresentación** y **CapaNegocios** que quedará divida en microservicios siguiendo las decisiones.

Observando el anunciado de la práctica adelantamos un poco la estructura añadiendo un paquete **Microservicios** en la capa de Negocios en el cuál almacenaremos los microservicios que se nos definirán en futuras decisiones, y en la capa de bases de datos definimos componentes externos que nos conectarán a estas (en la última iteración arreglamos este pequeño error, pues debemos indicar qué driver nos permite conectarnos con cada base de datos.)

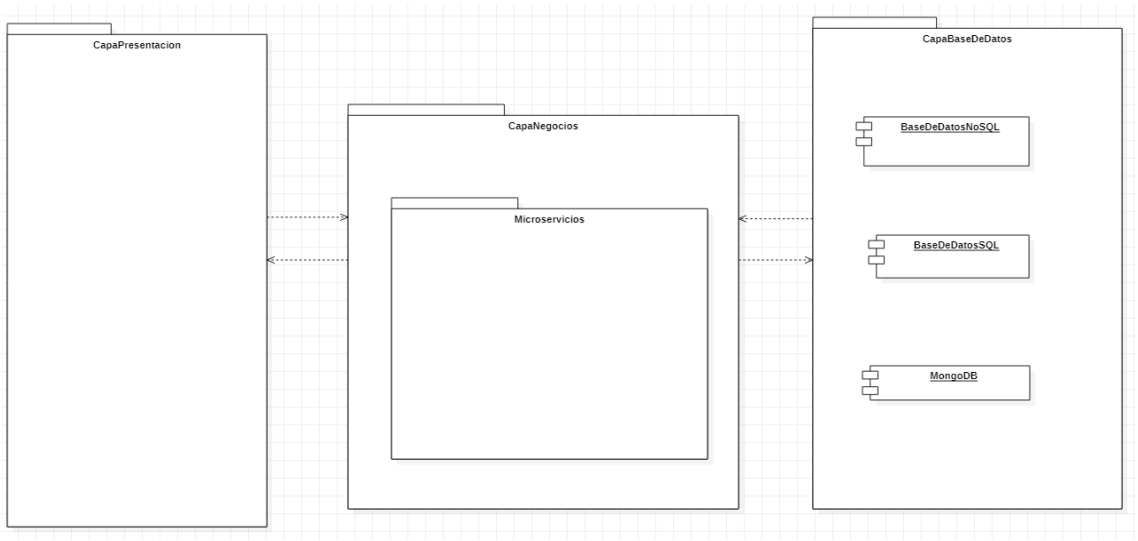

Figura 1: Iteración 1-UML

3.2 Iteración 2

3.2.1 Decisiones.

En esta iteración los seniors nos centramos en definir cuáles iban a ser los microservicios, cuales sus comunicaciones (RF1.2-RF1-5) así como cuales iban a ser los módulos que cumplir y cómo integrarlos (RF2.2-2.4). Esto nos permitía en las próximas iteraciones tener prácticamente todos los requisitos cubiertos y poder centrarnos en cómo distribuir el diseño y en especificar cosas más concretas del mismo.

Dentro de estas se tomó la decisión de implementar el diseño de la seguridad del microservicio de compras pues tenía una aplicación directa dentro del diseño y con una funcionalidad clara dentro del microservicio. Aunque pudiese tomarse dicha decisión como una decisión de calidad, la intención no es dar calidad a la aplicación sino añadirle funcionalidades claves que tienen que ser utilizadas por los requisitos que se propusieron.

3.2.2 Discusión con los cognitivos.

Entre las primeras opciones que se nos dio a considerar (la decisión 2) escogimos la segunda ya que así podríamos elegir las comunicaciones entre microservicios que viéramos necesaria y no gastar recursos en hacer conexiones innecesarias y mantenerlas. A continuación, en las decisiones 3_1 y 3_2 se escogieron las primeras opciones debido a que eran las opciones que nos daban mayor independencia. Por último, en la decisión

3_3, como la primera y la segunda opción se podían unificar, decidimos escoger ambas opciones, ya que así la seguridad aumentaba mucho más que si solo escogiéramos una opción u otra.

3.2.3 Diseño tras las decisiones tomadas.

Era necesario que un microservicio almacenara las preferencias de los usuarios, almacenando los artículos que compra, para ello hemos creado el "**Microservicio Preferencias**". Los datos recogidos se almacenarán en una base de datos NoSQL ya existente. (La cual especificamos en la última iteración, que será la CouchDB)

Se ha conseguido que el usuario pueda realizar una devolución de una compra a través de un microservicio, éste es "**Microservicio Devoluciones**" que se comunica con el "**Microservicio de Pedidos-Compras**" para verificar que la devolución pertenece a las compras realizadas.

El microservicio dedicado a las compras sirve para que el usuario haga pedidos y compras, éstos se gestionan gracias al id del producto y del cliente que compra, éstas se almacenan en una BBDD SQL (la cual especificamos en la última iteración, que será la AzureDB), este módulo cuenta con un sistema de hash mediante el uso del módulo **EncriptadorHash,** para la autenticidad del sistema, así como un sistema de logs para mantener la trazabilidad dentro del módulo de compras.

Hemos soportado el módulo de mensajería como una capa horizontal debajo de la capa de negocio tal y como indica la **Decisión 3_2**

Además, hemos incluido un componente **Gateway** entre la capa de Presentación y la capa de Negocio que será el encargado de llevar las peticiones entre ambas capas mediante una API REST que aplica el mismo componente.

Por último, hemos representado una comunicación directa con un banco central para que gestione y valide la compra a través del módulo **ComunicaciónBancoCentral,** que establece y autoriza el pago mediante unos métodos específicos.

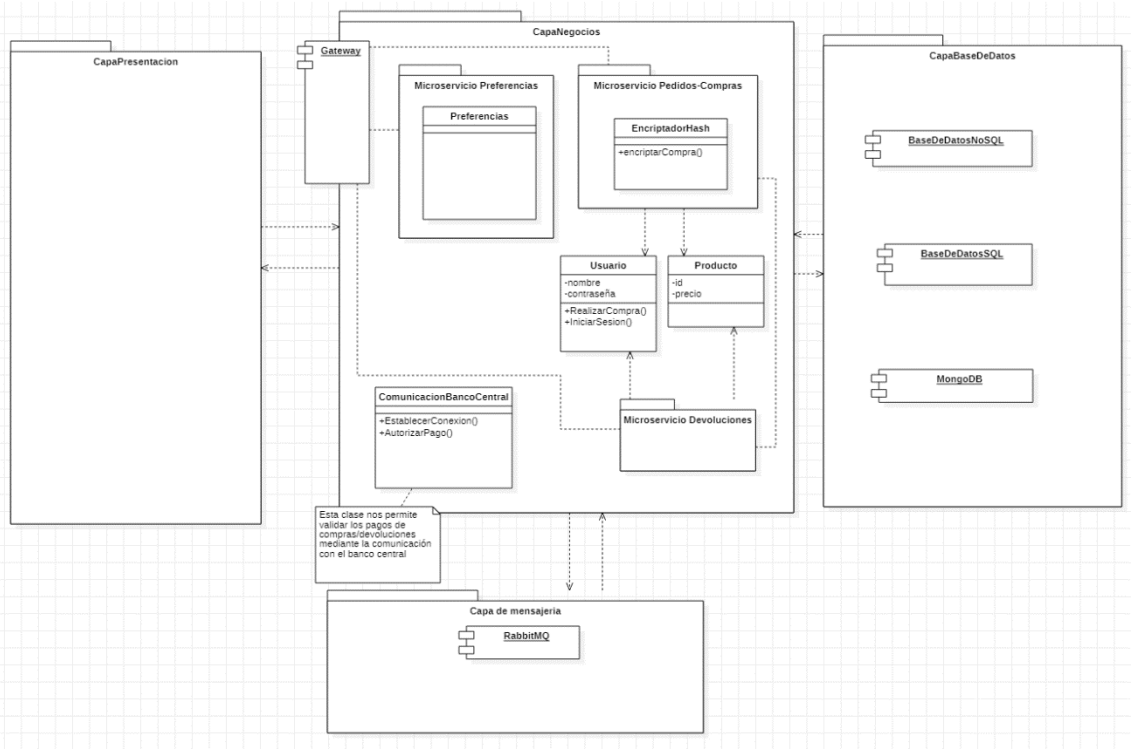

Figura 2: Iteración 2 - UML

3.3 Iteración 3

3.3.1 Decisiones.

En esta iteración solo se ha tomado una decisión para dejar cubiertos todos los requisitos y dejar para la última iteración las decisiones en relación con patrones de diseño que mejoren el sistema que tenemos que hacer.

3.3.2 Discusión con los cognitivos.

En esta iteración teníamos tres opciones diferentes a considerar respecto a la coherencia entre las distintas bases de datos en la decisión 4. Se escogió la opción de utilizar un sistema de mensajería como RabbitMQ (la segunda opción) ya que podíamos reutilizar el módulo de la capa de mensajería y además nos permitía una mayor escalabilidad.

3.3.3 Diseño tras las decisiones tomadas.

En esta tercera iteración, los júnior no realizamos ninguna modificación significativa ya que la decisión a representar (**Decisión 4**) no indicaba ningún cambio de representación, debido a que la comunicación entre la capa de Bases de Datos y la capa de Mensajería ya estaba cubierta en nuestro diseño, ya que ambas capas se conectan a través de la capa de Negocio, por lo tanto, se trata de una decisión no relevante para representar, sino más bien de una para el desarrollador que deba programar esta decisión.

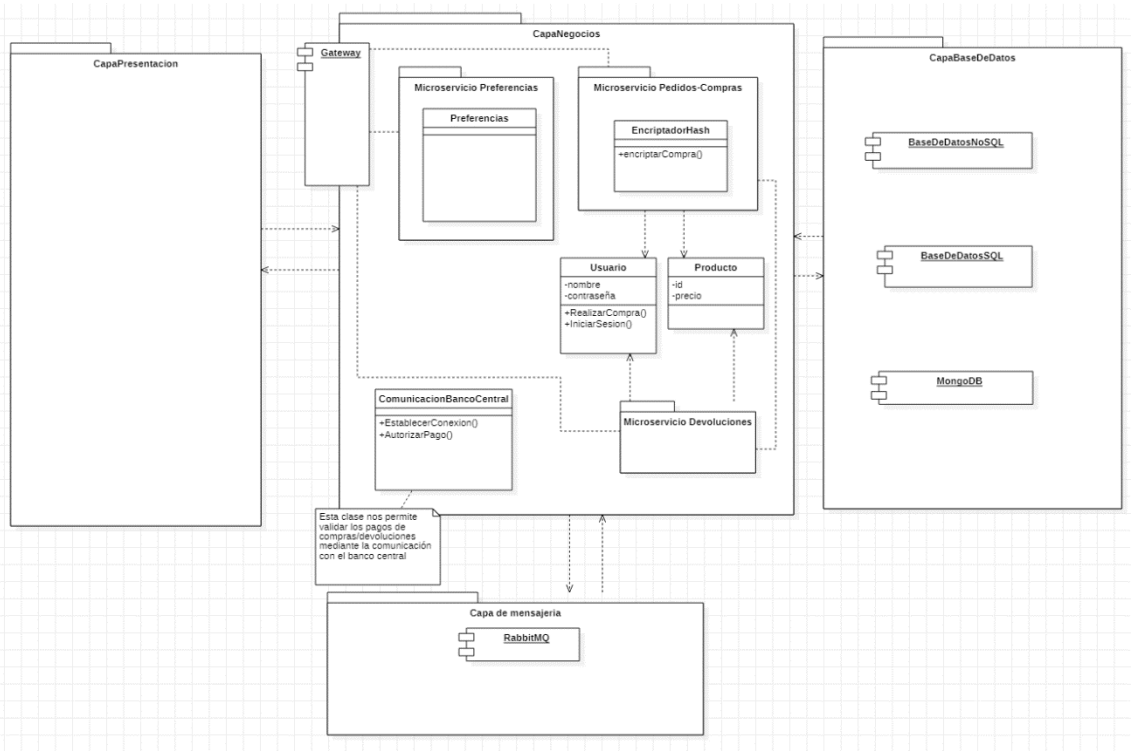

Figura 3: Iteración 3 - UML

3.4 Iteración 4

3.4.1 Decisiones.

En esta decisión nos encargamos de añadir la funcionalidad de vetar más de 5 peticiones de compras, pues no lo habíamos decidido hasta ahora, así como también añadir consecuencias neutras a algunas clases que han podido quedar algo ambiguas.

3.4.2 Discusión con los cognitivos.

En esta última iteración teníamos 2 opciones a considerar (en la decisión 5) con respecto al límite de intentos de compra. Aunque al principio la opción no estaba disponible, al final discutiendo con los senior se llegó a la conclusión de utilizar un patrón Circuit Breaker debido a que reduciría la probabilidad de fallo.

3.4.3 Diseño tras las decisiones tomadas.

En esta última iteración se han realizado diversos cambios respecto a las anteriores debido a retoques en las decisiones, una tutoría con el profesor Rafael Capilla Sevilla, y una reunión con los miembros del equipo para concretar detalles de los módulos, a continuación, enumeramos todos los cambios apoyándonos en la *Figura 4: Iteración 4-Final – UML:*

Algunos de los cambios más notables son los microservicios(todos los módulos con "**MS_**" en el nombre son microservicios); el microservicio de **Preferencias**, que pasa a ser una clase simple que recoge los datos que hemos considerado importantes para las preferencias de un producto (Marcas que se compran, rango de edad, precios, género de los compradores, etc.), el microservicio de **Devoluciones** también pasa a convertirse en una clase única que recoge los identificadores del pedido y del usuario que la realizó, y por último el microservicio de **Compras-Pedidos**; el cuál pasa a contener clases que

contemplan las decisiones 2, la decisión 3_1, y la decisión 3_3, con algunos elementos nuevos que reinterpretamos en la reunión con los sénior, como por ejemplo el módulo **ComunicacionBancoCentral** que ahora tiene los atributos que consideramos necesarios para llevar a cabo una transacción; el número de identificación de la transferencia/operación, el identificador del usuario, el número de cuenta del usuario y el valor del pago a realizar.

En la capa de **Bases de Datos** también ha habido modificaciones, pues ya no tenemos componentes con los nombres de las bases de datos a las que nos conectaremos, sino los drivers que nos permiten conectarnos de manera externa a estas, en el diagrama de despliegue se contempla mejor esta comunicación, pero lo resumimos rápidamente:

- Para la base de datos **MongoDB** hemos buscado un driver para utilizar esta base de datos con lenguaje Java, ya que es un lenguaje muy usado, y buscando en la página oficial de MongoDB encontramos que el driver más actualizado era el siguiente: '*Reactive Streams Driver versión 4.3*' (mongoDB, 2021).
- Para la base de datos SQL hemos elegido utilizar una base de datos **AzureSQL**, en específico un driver que nos permite usar Java para comunicarnos con ella; estamos hablando del '*Microsoft JDBC Driver'* (Microsoft, 2022).
- Para la base de datos NoSQL hemos elegido utilizar la base de datos **CouchDB** (grapheverywhere, 2022), ya que es una base de datos NoSQL documental, que nos permite guardar todos los datos de las preferencias en un solo documento, y el driver que encontramos para conectarnos de forma externa a esa base de datos sería el siguiente: '*CouchDB_ODBC_Driver_21.0.2137*' (CDATA, 2022)

Siguiendo la **decisión 5** hemos implementado el patrón **Circuit Breaker** (Budny, 2008) entre el módulo de usuarios y el microservicio de Pedidos-Compras siguiendo la documentación correspondiente al patrón en el anexo, con un ligero cambio en los estados ya que no contemplamos un estado de estar 'Semi-Abierto/Semi-Cerrado'.

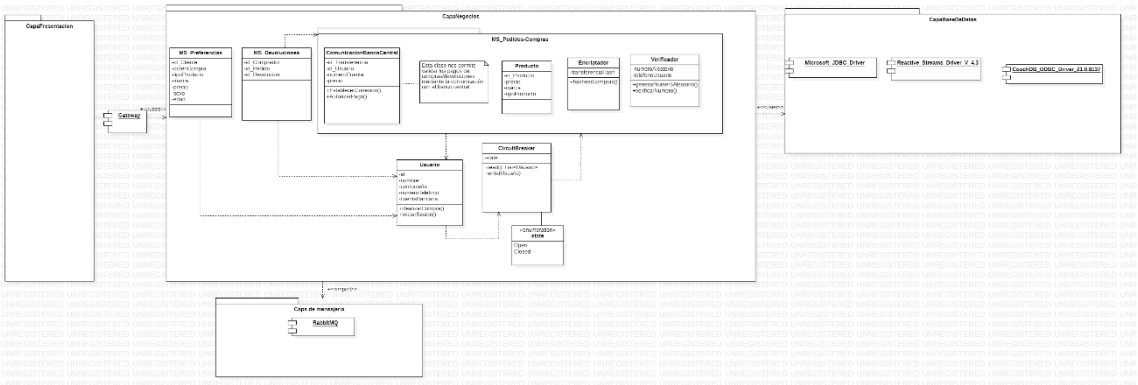

Figura 4: Iteración 4-Final – UML.

Como un añadido que no presentamos en las anteriores iteraciones, hemos complementado el diagrama UML de la última iteración con un diagrama de despliegue (*Figura 5: Iteración 4-Final – Diagrama de Despliegue*) que muestra de forma sencilla cómo se comunica nuestro sistema con las diferentes bases de datos y con el sistema del banco:

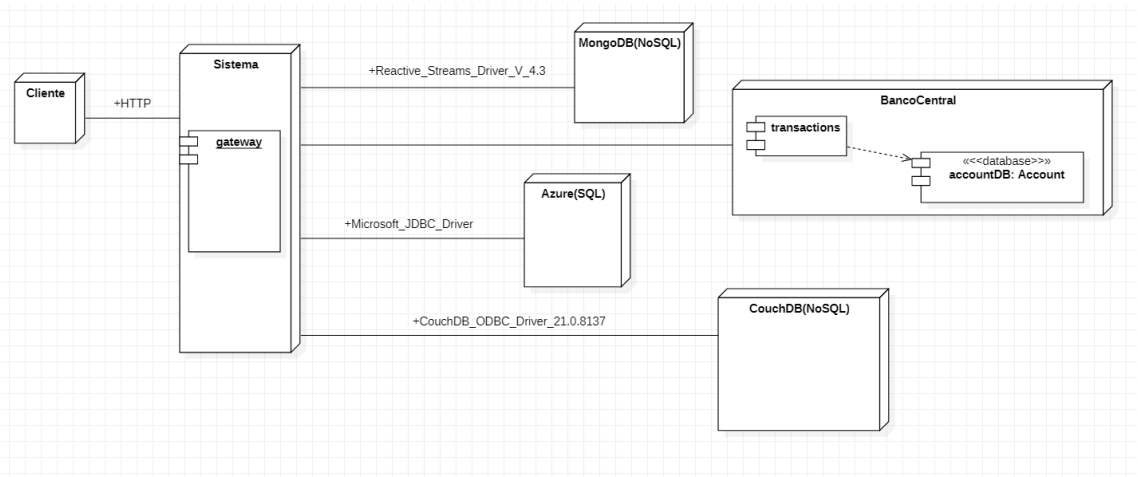

Figura 5: Iteración 4-Final - Diagrama de Despliegue.

3.5 Conclusiones tras las iteraciones.

En esta segunda revisión a esta práctica, hemos llegado a la conclusión que, en la anterior no pulimos tanto tantas partes, arrastramos errores antes de consultarlos, así como arrastramos un mal análisis de requisitos que nos acarreó muchos problemas. En esta segunda vez que realizamos esta práctica hemos aprendido cosas nuevas, como a mejorar nuestro análisis de requisitos o a clarificar más nuestras decisiones en relación con la primera vez que nos enfrentamos a ella.

Hacer esta práctica dos veces, si bien en un inicio ha sido un problema abrumador y nos enfrentamos a él con una actitud algo derrotista, una vez ya acabado hemos visto varios errores que cometimos anteriormente y hemos aprendido de ellos.

Hay que mencionar también que para la facilitación de lectura y evaluación de la práctica se ha prestado especial cuidado a los ".md" del GitHub para que sean agradables y fáciles de leer, la mayor cantidad de información del avance de la práctica está recogido en el ReadMe, así como también se han creado etiquetas para el final de cada una de las iteraciones.

Por último, al final del anexo, en *Figura 8: Tabla de tiempos de cada iteración* se encuentra la tabla de tiempos, que también se sitúa en nuestro ReadMe.

4. Bibliografía

Budny, M. (04 de 11 de 2008). *Circuit Breaker pattern AOP style*. Obtenido de marcinbudny.com: https://blog.marcinbudny.com/2008/11/circuit-breaker-pattern-aop-style.html

CDATA. (27 de 04 de 2022). *Real-Time Data Connectos*. Obtenido de cdata.com: https://cdata.com/drivers/

grapheverywhere. (27 de 04 de 2022). *Bases de Datos NoSQL |Qué son, marcas, tipos y ventajas*. Obtenido de www.grapheverywhere.com: https://www.grapheverywhere.com/bases-de-datos-nosql-marcas-tipos-ventajas/

Microsoft. (26 de 3 de 2022). *Use Java and JDBC with Azure SQL Database*. Obtenido de Microsoft Build: https://docs.microsoft.com/es-es/azure/azure-sql/database/connect-query-java?view=azuresql

mongoDB. (24 de 10 de 2021). *MongoDB Java Reactive Streams*. Obtenido de mongodb.com: https://www.mongodb.com/docs/drivers/reactive-streams/

5. Anexo

Un sistema basado en una arquitectura Web de tres capas (tiers) se desea migrar a una arquitectura de microservicios más flexible y escalable. El sistema actual gestiona las solicitudes mediante una lógica de negocio con dos bases de datos (una SQL y otra NoSQL) que almacenan datos de compras por Internet y de preferencias de los clientes respectivamente. Actualmente, el uso de móviles y tabletas para las compras demanda una mayor escalabilidad del sistema actual. La aplicación existente consta de los siguientes módulos.

- Componente de presentación: Son los responsables del control de la interfaz de usuario y el consumo de servicios remotos.
- Componentes de lógica de negocio: Módulo de pedidos y compras de clientes, módulos de detección de preferencias, módulo de conexión a sistemas de pago, módulo de mensajería a dispositivos móviles, módulo de seguridad en compras y módulo de devoluciones.
- Lógica de acceso a bases de datos. Son los componentes de acceso a datos responsables de obtener acceso a las bases de datos (SQL o NoSQL).
- El sistema de mensajería se soporta por un middleware independiente.

La nueva aplicación deberá ser capaz de integrar microservicios de forma asíncrona y altamente escalable para soportar un mayor número de compras vía móvil y detección de preferencias de usuarios de una forma más eficiente. Los microservicios que soporten la reingeniería y migración de está funcionalidad deberán usar el protocolo REST mientras que la mensajería usará AMQP. Los arquitectos software decidirán cuantos microservicios y en que lenguaje de programación se van a desarrollar así como el middleware necesario para albergar dichos microservicios.

Entre los problemas adicionales de diseño que se plantean es decidir cuántos microservicios son necesarios dado que se deberá contar con una nueva base de datos en MongoDB para almacenar la localización de microservicios de la empresa y otros similares. Además, es necesario que exista una coherencia entre las bases de datos de los diferentes microservicios que se gestione mediante a través de un bus de eventos lógicos (e.g. Command and Query Responsibility Segregation) que utilicen tecnología de mensajería como RabbitMQ o un "bus" dedicado como pueden ser Azure Service Bus, NServiceBus, MassTransit o Brighter. La selección de la tecnología se basará en aspectos de escalabilidad y prestaciones así como de interoperabilidad.

Asimismo, se deberá limitar el número de intentos de compra a 5 de manera que no se sobrecargue el número de peticiones a los microservicios correspondientes. Los arquitectos software deberán determinar cuantos contenedores de microservicios son necesarios.

Todas las comunicaciones son vía HTTP. Los clientes se comunican con los microservicios mediante un componente Gateway que contiene un módulo para monitorizar el estado de cada microservicio.

El alumno deberá suponer que datos requiere cada microservicio o el cliente para gestionar una tienda virtual (e.g. número de pedido, identificación de cliente, producto, cantidad, precio, total, forma de pago, etc.).

Figura 6: Enunciado de la práctica subrayado.

Nº DE REQUISITO	NOMBRE CORTO	DESCRIPCIÓN
RF1	Migración de la arquitectura	Se desea migrar la antigua capa de negocio a una basada en microservicios más escalables y flexibles, éstos se deben poder integrar en la aplicación de forma asíncrona y se debe de almacenar la dirección del microservicio en una BBDD MongoDB para poder acceder a ellos en todo momento. El sistema contendrá los siguientes microservicios:
RF1.2	Microservicio preferencias	Es necesario que un microservicio almacene las preferencias de los usuarios, almacenando los artículos que compra, el nombre del usuario, y el precio que ha pagado por el artículo en cuestión, éstos datos se almacenarán en una base de datos NoSQL ya existente.
RF1.3	Microservicio pedidos y compras	Es necesario un microservicio dedicado a que el usuario haga pedidos y compras, éstos se gestionan gracias al id del producto y del cliente que compra, éstas se deben de almacenar en una BBDD SQL a través de un bus de eventos lógicos, y se debe impedir al usuario intentar comprar más de 5 veces.
RF1.4	Comunicación de microservicios	Los microservicios han de comunicarse entre ellos usando el protocolo REST, así como comunicarse con los elementos fuera del sistema a través de un gateway que debe soportar comunicaciones HTTP.
RF1.5	Microservicio de devoluciones	Se desea que el usuario pueda realizar una devolución de una compra a través de un microservicio, éste se debe de comunicar con el microservicio de compras para verificar que la devolución pertenece a las compras realizadas.
RF2	Módulos de negocio	La aplicación que migramos se divide en distintos módulos que se han de mantener, los cuales son:
RF2.2	Módulo de pagos	El sistema debe de gestionar los pagos realizados por nuestros clientes desde el microservicio de compras, que lo valide para que ésta se realice con éxito.
RF2.3	Módulo de mensajería	El sistema debe tener un módulo de mensajería, soportado por un middleware independiente, que usará el protocolo de AMQP.
RF2.4	Módulo de seguridad	Es necesario que exista un módulo de seguridad en compras.
RF3	Coherencia entre las bases de datos.	La coherencia entre las distintas bases de datos se certificará mediante a través de un bus de eventos lógicos. El cliente sugiere que utilicen tecnología de mensajería como RabbitMQ o un "bus" dedicado como Azure Service Bus.

Hemos encontrado un apartado indicado a la capa de presentación que no afectará a la migración del sistema, ergo no se ha añadido a los requisitos.

Figura 7:Tabla de requisitos final con el subrayado pertinente.

Semana	Iteración	Tiempo en ADD	Tiempo de reflexión	Tiempo de refinado	Tiempo de diseño
1	1	30	15	0	0
1	2	30	20	5	180
2	3	40	20	20	130
3	4	120	25	35	150
3	5	200	35	15	810

Figura 8:Tabla de tiempos de cada iteración.