# Semana 1

|  |  |  |
| --- | --- | --- |
| Requisito ID | Nombre | Descripción |
| RF-01 | Gestionar Solicitudes | Gestionar las solicitudes mediante una lógica de negocio y base de datos. |
| RF-1.1 | Peticiones de Solicitudes | Las peticiones se devuelven en HTML y JSON. |
| RF-02 | Ajuste de Pantallas | La aplicación se deberá ajustar automáticamente en los diferentes dispositivos. |
| RF-03 | Componentes de la aplicación | La aplicación necesita tener cuatro componentes básicos para su correcto funcionamiento. |
| RF-3.1 | Canal de Mensajería | La aplicación debe incluir un canal de mensajería basado en agentes de mensajes |
| RF-3.2 | Componentes de Presentación | Permiten el control de la interfaz de usuario y el consumo de servicios remotos. |
| RF-3.3 | Lógica de dominio | Se trata de la lógica de dominio de la aplicación. |
| RF-3.4 | Acceso a Base De Datos | Permite el acceso a distintos tipos de BBDD, como SQL y NoSQL. |
| RF-04 | Enlazar Aplicaciones Externas | Se enlazarán aplicaciones externas de forma asíncrona para reforzar la fiabilidad de los microservicios. |
| RF-05 | Funcionalidades en microservicios independientes | Se desarrolla e implementa cada microservicio de forma independiente sin que afecte a otros subsistemas. Se soportarán en bases de datos SQL distintas, a excepción de la cesta que utiliza una base de datos NoSQL. La coherencia entre las bases de datos de los diferentes microservicios se logra mediante eventos de integración de nivel de aplicación. |
| RF-5.1 | Catálogo de Microservicios | Es una lista de los diferentes microservicios disponibles que se pueden usar. |
| RF-5.2 | Procesado de Pedidos | La aplicación debe poder procesar los pedidos que realice el usuario en la tienda. |
| RF-5.3 | Identificación de Usuarios | La aplicación es capaz de identificar el usuario que ha iniciado sesión con su cuenta de la tienda. |
| RF-5.4 | Cesta de Compra | Se utilizará una caché para almacenar la cesta de compra con los productos que desea comprar y la localización de microservicios de terceros, en una base de datos MongoDB. |
| RF-06 | Comunicación entre los clientes y los microservicios | El cliente desea comunicarse con el microservicio y lo puede hacer a través de la puerta de enlace de una API o también con el microservicio MVC, que se comunica con otros microservicios mediante la puerta de enlace de API. Se podrá usar contenedores de microservicios dentro de un host o mediante un cluster. Se utilizarán dos tipos de comunicación. |
| RF-6.1 | Consultar Actualizaciones | Comunicación de cliente a microservicio de HTTP a través de un Gateway que contiene diversas APIs  Se debe poder consultar si es necesario que la aplicación se actualice. |
| RF-6.2 | Bus de eventos | Comunicación asincrónica basada en eventos.  Propaga las actualizaciones entre microservicios y se puede implementar agentes de mensajería como RabbitMQ, Service Bus (Azure Service Bus, NServiceBus, MassTransit...) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SEMANA | ITERACION | TIEMPO ASS | REFLEXION ASS-ASC | REFELXION ASS | TIEMPO ASJ |
| 1 | 1 | 50 | 20 | 15 | 35 |
| 1 | 2 | 35 | 15 | 15 | 30 |

# Semana 2.

## Decisiones de estilo

Tras analizar el problema en profundidad los arquitectos senior (ASS) han tomado la decisión de basarse en un estilo de Cliente-Servidor. Ya que el cliente indica que quiere que los usuarios que utilicen la aplicación tengan una interfaz propia, una lógica de negocio y lógica de acceso de base de datos. Ya que la arquitectura Cliente-Servidor se divide en 3 capas. Siendo estos los puntos 3.2, 3.3 y 3.4 propios de la tabla de requisitos realizada la semana 1.

## Documento de riesgo

Tras una reunión entre los ASS y los ASC, se ha discutido si sería mejor implementar una arquitectura por Capas. Pero finalmente hemos decidido la arquitectura Cliente-Servidor ya que aporta la división por capas y

Por otro lado, hemos descartado las siguientes arquitecturas:

El modelo vista controlador ya que es un estilo iterativo y no se va a adecuar lo suficiente a nuestro problema, aunque en un primer momento hemos dudado sobre si sería el adecuado.

El modelo de tuberías y filtros ya que este realiza una función completamente distinta a lo que nuestro cliente necesita por eso lo hemos descartado instantáneamente.

El de eventos ya que nuestro problema no valora en ningún momento el uso de eventos ni la interacción con ellos.

Decisiones de Diseño:

En la reunión del día 18 de noviembre, los arquitectos senior (ASS) tomaron la decisión de implementar los siguientes patrones de diseño:

* Facade provee de una interfaz única, simple para acceder a una interfaz o grupo de interfaces de un sistema. Este patrón de diseño hace referencia al requisito RF-3.2 y al RF-2 ya que será necesaria una interfaz de usuarios y el consumo de servicios en remoto.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SEMANA | ITERACION | TIEMPO ASS | REFLEXION ASS-ASC | REFELXION ASS | TIEMPO ASJ |
| 1 | 1 | 50 | 20 | 15 | 35 |
| 1 | 2 | 35 | 15 | 15 | 30 |
| 2 | 1 | 45 | 20 | 10 | 35 |
| 2 | 2 | 40 | 15 | 15 | 25 |

# Semana 3

## Decisiones de diseño

En la reunión del día 24 de noviembre, los arquitectos senior (ASS) tomaron la decisión de implementar los siguientes patrones de diseño:

* Proxy ya que proporciona un intermediario de un objeto para controlar su uso. En este caso, este patrón hace referencia al requisito RF-3.4 porque se necesita acceder a una base de datos y las páginas más frecuentemente visitadas.
* Singleton garantiza la existencia de una única instancia para una clase. Este patrón se puede aplicar al requisito RF-5.3 y RF-5.4 ya que se necesitará una cesta de la compra y un solo sistema de identificación de usuarios.
* Parallel split es un punto de flujo donde un thread individual se divide en varios threads que pueden ejecutarse en paralelo permitiendo la ejecución simultanea de varias actividades. En nuestro caso, para comprobar si hay stock de un producto a la hora de realizar un pago. Este patrón de diseño hace referencia al requisito RF-5.2 donde se procesan los diferentes pedidos para no comprar un producto agotado.

Documento De Riesgo

En la reunión del día 25 de noviembre los ASC reflexionaron con los ASS sobre los problemas que estas decisiones podían ocasionar, en un principio los ASS al ver los patrones de diseño aprendidos pensaron en un patrón builder pero rápidamente los ASC rechazaron esta propuesto debido a que el builder no tenía sentido ahí y se propuso utilizar el singleton. Las demás decisiones fueran aprobadas sin ningún problema por los ASC

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SEMANA | ITERACION | TIEMPO ASS | REFLEXION ASS-ASC | REFELXION ASS | TIEMPO ASJ |
| 1 | 1 | 50 | 20 | 15 | 35 |
| 1 | 2 | 35 | 15 | 15 | 30 |
| 2 | 1 | 45 | 20 | 10 | 35 |
| 2 | 2 | 40 | 15 | 15 | 25 |
| 3 | 1 | 50 | 20 | 35 | 40 |
| 3 | 2 | 20 | 10 | 15 | 30 |

# Semana 4

## Decisiones de diseño

En la reunión del día 1 de diciembre, los arquitectos senior (ASS) tomaron la decisión de implementar los siguientes patrones de diseño:

* Access Token: Permite el control de acceso a la aplicación, detecta a los usuarios que intentan acceder a los recursos de nuestra aplicación, este patrón hace referencia al requisito 5.3
* API Gateway: Nos permite enrutar las peticiones desde el exterior hacia cada microservicio, este patrón hace referencia al requisito 6 y 6.1.
* Observer: Hace referencia al requisito 6.2, ya que una actualización produce un cambio de estado y este ha de notificarse.

Documento De Riesgo

Tras tomar las decisiones los ASS los ASC se reunieron para discutir sobre las decisiones tomadas y se dieron cuenta que faltaba algo que hiciese referencia al requisito 6, más en concreto al requisito 6.2, ya que necesitamos un bus de eventos y para ello necesitamos un estilo por eventos, que se incluye dentro del estilo principal de nuestra arquitectura, que es cliente servidor. Después de plantear estos problemas a los ASS, estos volvieron a reunirse y tomaron la decisión de que debía de estar implementado el estilo por eventos. Por otra parte, los patrones anteriores no generaron problema y fueron rápidamente aceptados. De esta forma añadimos una nueva decisión de diseño:

* Estilo por eventos: Los sistemas EDA monitorizan los datos del entorno y analizan los eventos para proporcionar la respuesta más adecuada. Una arquitectura EDA tiene emisores de eventos, consumidores y canales para transmitir los eventos. Todo esto hace referencia al requisito 6.2 para tener un bus de eventos y el 3.1 ya que el estilo por eventos necesita que los eventos se transmitan entre sistema poco acoplados mediante mensajes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SEMANA | ITERACION | TIEMPO ASS | REFLEXION ASS-ASC | REFELXION ASS | TIEMPO ASJ |
| 1 | 1 | 50 | 20 | 15 | 35 |
| 1 | 2 | 35 | 15 | 15 | 30 |
| 2 | 1 | 45 | 20 | 10 | 35 |
| 2 | 2 | 40 | 15 | 15 | 25 |
| 3 | 1 | 50 | 20 | 35 | 40 |
| 3 | 2 | 20 | 10 | 15 | 30 |
| 4 | 1 | 40 | 20 | 25 | 20 |
| 4 | 2 | 30 | 15 | 20 | 20 |