Microservicios
Arquitectura de Referencia

Gerencia Información y Tecnología
Dirección de Arquitectura y Operación Tecnológica
Julio 30, 2018

Control de Cambios

Versión	Fecha	Responsable	Descripción
1.0	2018-07-30	John Bairo Gómez	Versión inicial del documento
1.0	2018-08-01	John Bairo Gómez	Versión preliminar para proceso de verificación y ajustes.
1.0	2018-08-02	John Bairo Gómez	Se incluye restricción técnica para mecanismo de extracción de datos para los reportes. Se incluye Antipatrón Database Push Model que aplica para reportes.
1.0	2018-08-10	John Bairo Gómez	Correcciones menores de semántica.
1.0	2018-08-13	John Bairo Gómez	Se incluyen las siguientes secciones: • Arquitectura Interna de los Microservicios. • Topología de Microservicios Mensajería Centralizada Se actualiza diagrama de tecnologías habilitadas para desarrollo de aplicativos.
1.0	2018-09-06	John Bairo Gómez	Se actualiza documento para que represente la arquitectura de referencia de Microservicios. Se adicionan las secciones: • Vista de Aplicaciones • Vista de Despliegue.

Contenido

Tabla de Ilustraciones

1 Introducción

Con la importancia que surge de abordar y dar solución a las necesidades, de aumentar y afinar las funcionalidades que hacen y harán parte del nuevo sistema -aun cuando este sea solo una porción funcional del todo-, emergen muy significativamente, desde la Arquitectura de Software, los mecanismos que orientan y definen de manera muy cuidadosa la división funcional de esas necesidades, ya que éstas, serán las primeras en permitir asimilar nuevos paradigmas, representados en nuevas tecnologías que se adoptan para dar paso a una nueva era digital, la cual se reflejará en los atributos de calidad -seguridad, modificabilidad, escalabilidad, disponibilidad, desempeño, etcétera- que permitan impactar positivamente la infraestructura tecnológica de XM, aportando de manera natural valor para el negocio.

1.1 Propósito

Presentar las mecanismos, estructuras, definiciones y restricciones fundamentales requeridas para satisfacer las necesidades presentadas, mediante requisitos funcionales y no funcionales. Se pretende, además, identificar los elementos críticos que impactan o pueden impactar la arquitectura de la solución propuesta. El presente documento, refleja los cimientos sobre los cuales será construida la solución, aun cuando las interacciones entre muchos de sus componentes puedan estar en un sistema legado, que hace presencia como un actor de gran relevancia para el nuevo sistema, ya que es este, -en caso de ser un sistema legado- quien “sufrirá” las adopciones y adaptaciones del nuevo paradigma tecnológico, y sus nuevas características girarán en torno a este gran legado; por el contrario, cuando el nuevo sistema surge como algo nuevo en todo su esplendor –sin tener un legado-, la adopción tecnológica será más natural, ya que el nuevo paradigma tecnológico, no tendrá “barreras” tecnológicas más allá de las propias, las que impone el nuevo paradigma y las complejidades para definir el *Bounded Context*, disminuirán en su nivel de sustancialmente.

La arquitectura propuesta, se presenta como el principal artefacto que refleja el análisis temprano, cuyo objetivo encamina a obtener un sistema aceptable en términos de calidad tanto técnica como funcional.

1.2 Alcance

Los elementos de arquitectura planteados a lo largo de este documento, como definición base, se fundamentan en las definiciones arquitectónicas de referencia que se han especificado al interior de XM, las cuales definen tecnologías a utilizar en las capas de frontend y backend, además de las directrices que determinan el uso de elementos y/o artefactos corporativos.

El presente documento, no va más allá de ser la referencia guía, para dar cumplimiento a las definiciones técnicas a través de la implementación de Arquitectura Candidata de todo proyecto que adopte la Arquitectura de Microservicios, la cual se reflejará en la materialización técnica y funcional que permitirá satisfacer las necesidades planteadas, se realizará en *componentes independientes de propósito único*, exponiendo -si es necesario- mediante API´s la información que sea de interés para quienes estén por fuera de los límites del dominio que se implementa.

Como complemento a esta sección de Alcance, cada proyecto deberá definir un documento técnico que especifique el detalle de la solución candidata, el (los) Microservicio (s), metas y restricciones, ya que cada proyecto viene con sus propios retos y definiciones. Cabe aclarar, que la base técnica de la arquitectura que se especifica en el presente documento, permanece inmutable, puesto que éste representa la base y referencia guía para abordar los proyectos con Arquitecturas orientadas a Microservicios.

1.3 Acrónimos, Abreviaturas y Términos

Término	Descripción
Docker	Wikipedia lo define como: “Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos”
Kubernetes	Según Wikipedia: “Kubernetes define un conjunto de bloques de construcción (primitivas) que conjuntamente proveen los mecanismos para el despliegue, mantenimiento y escalado de aplicaciones. Los componentes que forman Kubernetes están diseñados para estar débilmente acoplados, pero a la vez ser extensibles para que puedan soportar una gran variedad de flujos de trabajo. La extensibilidad es provista en gran parte por la API de Kubernetes, que es utilizada por componentes internos, así como extensiones y contenedores ejecutados sobre Kubernete”

1.4 Referencias

Documento / Referencia	Descripción
20171122 - Informe Arquitecturas de Referencia	Descripción detallada de las Arquitecturas de Referencia 2017, autorizadas a ser implementadas en XM.
XM Lineamientos y Estándares TI V4.1	Lineamientos generales de TI para la implementación de aplicaciones.
Lineamientos de Seguridad V 1.2	Describe las directrices que se deben contemplar en la seguridad de las aplicaciones
Swagger	Sitio oficial de Swagger: https://swagger.io/ Diseño, construcción, documentación y pruebas de las API´s.
Angular	Sitio oficial de Angular: https://cli.angular.io/
Reporting Services	Conceptos básicos de Reporting Services https://docs.microsoft.com/en-us/sql/reporting-services/reporting-services-concepts-ssrs?view=sql-server-2017
TypeScript	Sitio oficial de TypeScript: https://www.typescriptlang.org/
JavaScript	Sitio oficial de JavaScript: https://www.javascript.com/
Software Architecture Patterns, Mark Richards	O’Reilly Media
Microservices Architecture, Mike Amundsen	O’Really Media
Microservices on Azure	https://docs.microsoft.com/en-us/azure/architecture/microservices/
Building Microservices, Sam Newman	O’Really Media
Modelo de Datos Anémico	https://www.martinfowler.com/bliki/AnemicDomainModel.html

1.5 Audiencia Objetivo

La documentación de arquitectura debe servir para varios propósitos. Debe ser lo suficientemente abstracto para que los "nuevos" participantes puedan comprenderlo rápidamente. Debe ser lo suficientemente concreto como para servir como modelo para la construcción. Debe tener suficiente información para servir como base para el análisis. Por lo tanto; es de interés abordar el presente documento a Arquitectos Empresariales, Arquitectos de Software, Arquitectos de Aplicaciones, Arquitectos de Soluciones, Líderes Técnicos, Analistas de Requisitos, Desarrolladores, Analistas de Calidad y toda persona involucrada en el proyecto. Sin embargo, cabe anotar que es de naturaleza "*obligada*", abordar el presente documento, a todas las personas involucradas en la implementación directa de la solución, como Líderes Técnicos, Desarrolladores y Analistas de Requisitos y Calidad.

2 Objetivos

El objetivo principal del documento consiste en presentar el conjunto de decisiones más importantes mediante los elementos estructurales y las relaciones de las cuales se compone el sistema.

2.1 Objetivos específicos:

- Presentar las decisiones arquitectónicas que permitan satisfacer los atributos de calidad del sistema.
- Presentar las decisiones arquitectónicas que permitan satisfacer los requisitos de comportamiento del sistema.

3 Contexto Actual

Los anexos técnicos de cada proyecto, deberán especificar el Contexto Actual, ya que no se debe perder de vista, el problema que pretende ser solucionado mediante la Arquitectura de Microservicios, por lo tanto, el nivel de detalle esperado en esta sección, es la especificación de cada una de las necesidades que deben ser abordadas por el proyecto. Se espera además, para los casos en los cuales se interviene un sistema legado, las vistas arquitectónicas que ayuden a comprender la situación actual y la orientación de la solución.

4 Supuestos

- ✓ El *Componente Corporativo de Autorización* de Usuarios se encuentra disponible al momento de realizar las implementaciones que requieren mecanismos de autenticación.
- ✓ El *Componente Corporativo de Autorizaciones* de funcionalidades de aplicación, se encuentra disponible al momento de realizar las implementaciones que requieren hacer uso de dichas funcionalidades.
- ✓ El *Componente Corporativo Navegabilidad,* se encuentra disponible al momento de requerir implementar la funcionalidad que permita hacer visible la navegabilidad del sistema.
- ✓ El Componente de Acceso a Datos, se encuentra disponible al momento de requerir implementar operaciones sobre alguno de los repositorios definidos para el sistema.

5 Arquitectura Candidata

5.1 Estilo

El estilo arquitectónico orienta la visión de la solución, la cual se plasma y materializa en los componentes que harán parte de la definición estructural, relaciones y dependencias internas; además se garantiza estar alineada con las definiciones corporativas que determinan el uso de tecnologías de vanguardia. Para dar cumplimiento a directrices definidas desde la Gerencia de TI, el estilo arquitectónico en adopción es la **Arquitectura Orientada a Microservicios**.

El estilo de Arquitectura de Microservicios evolucionó de forma natural a partir de dos fuentes principales: aplicaciones monolíticas desarrolladas utilizando el patrón de arquitectura en capas y aplicaciones distribuidas desarrolladas a través del patrón de arquitectura orientada a servicios SOA.

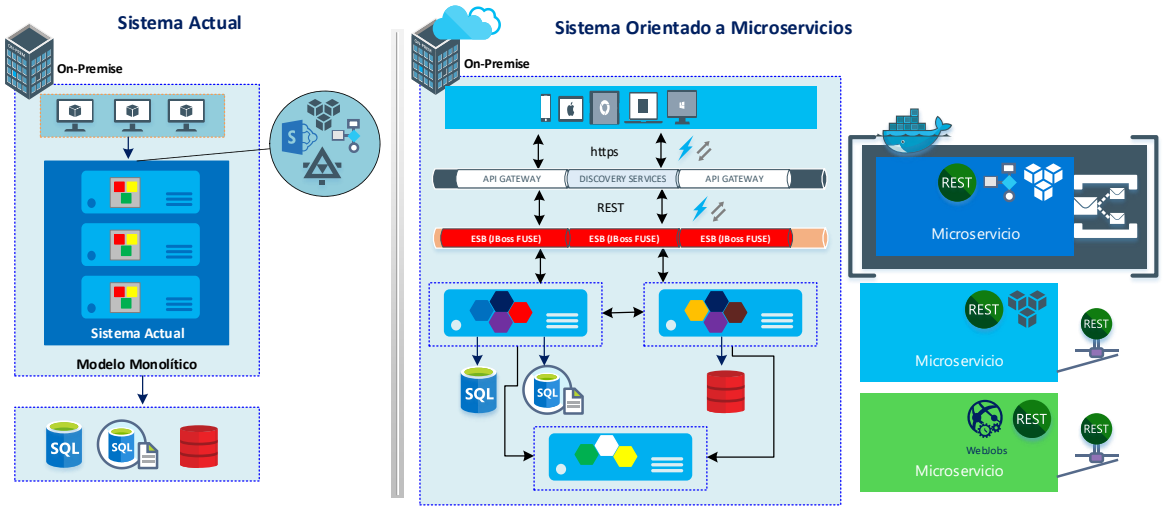

Ilustración 1. Sistema Monolítico VS Microservicios

La gráfica anterior, no va más allá de ilustrar la transición hacia el nuevo paradigma de los Microservicios; representando en una vista de muy alto nivel, la nueva visión que se tiene de las nuevas capacidades a implementar, las cuales se nombran e identifican como Microservicios, estos deberán convivir con un modelo tradicional (Cliente/Servidor o SOA), consumiendo en algunos casos información del legado, consumiendo los mecanismos existentes que este expone para compartir información. Observemos con detalle el lado derecho de la gráfica –Sistema Orientado a Microservicios-. La estrategia pretende implementar las funcionalidades nuevas, adoptando la Arquitectura de Microservicios, podemos optar por utilizar Contenedores para los Microservicios o realizar los despliegues sin hacer uso de ellos, se observa además que es posible exponer -cuando es necesario- servicios sincrónicos con REST o utilizar mensajería asíncrona con mecanismos de *Publicación/Suscripción*, de tal manera que sea posible garantizar el desempeño en los términos que se especifican para el negocio.

Se aclara que, aunque la gráfica ilustra un despliegue haciendo uso de una nube híbrida, también es posible continuar con el esquema tradicional *On-Premises*.

5.1.1 Unidades Independientes

Se enfatiza que en la noción de unidades desplegadas por separado, cada componente de la Arquitectura de Microservicios del sistema, se debe implementar como una unidad independiente, lo que permite una implementación más sencilla a través de una canalización de entrega efectiva y simplificada, mayor escalabilidad y un alto grado de aplicación y desacoplamiento de componentes dentro de su aplicación.

5.1.2 Topología de Microservicios Mensajería Centralizada

Se hace uso del patrón Topología de Mensajería Centralizada. Esta topología es similar a la topología basada en REST, excepto que en lugar de usar REST para acceso remoto, esta topología utiliza un agente de mensajería centralizado liviano (ActiveMQ), el cual estará orquestado y/o coreografiado por JBoss FUSE.

Los beneficios de esta topología sobre la topología basada en REST simple, son los mecanismos avanzados para gestión de colas, la mensajería asincrónica, la supervisión de mensajes, el manejo de errores y un mejor balanceo y escalabilidad de la carga en general.

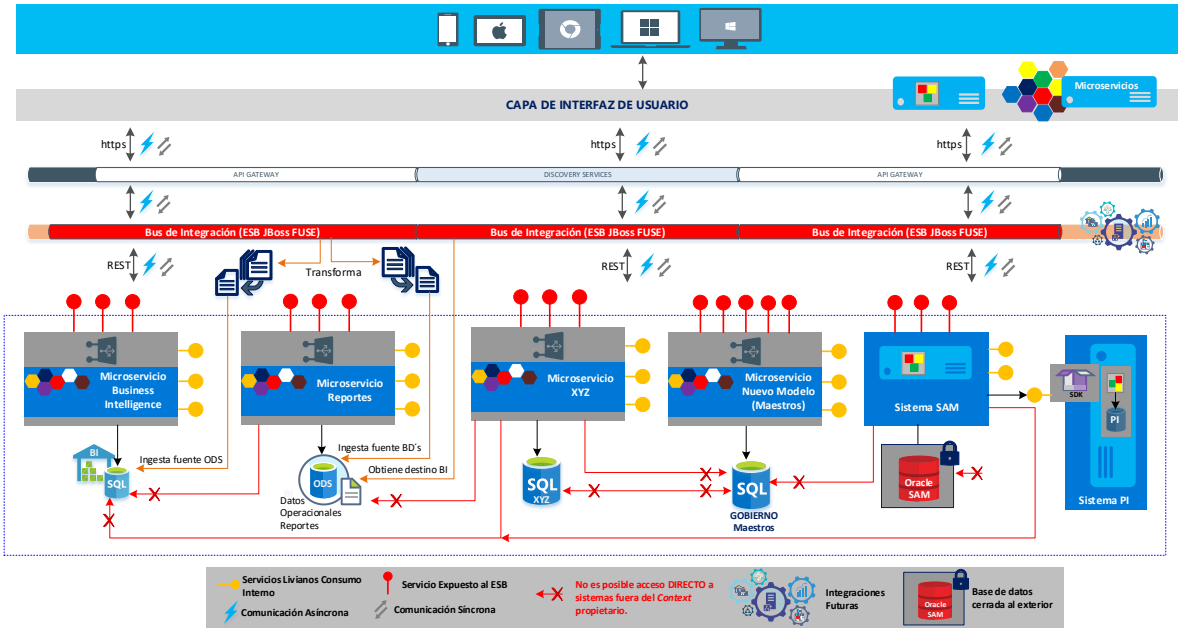

Ilustración 2. Topología Mensajería Centralizada

Se hace evidente que el punto único de falla y los problemas arquitectónicos de cuello de botella generalmente se asocian con el intermediario centralizado; la manera de abordar dicho problema es a través del clúster de intermediarios y la federación de intermediarios (dividiendo una sola instancia de intermediario en varias instancias de intermediarios, para dividir la carga de rendimiento del mensaje en función de las áreas funcionales del sistema). JBoss FUSE como intermediario, ofrecerá capacidades de instanciación múltiple, ya que de forma natural permite la creación de múltiples instancias por ruta de mensajes, evitando que se presenten los cuellos de botella que se han mencionado; de esta manera se mitiga el riesgo del punto de falla que se presenta con esta topología.

Es importante recalcar la importancia del gráfico anterior, más allá de ilustrar el flujo de comunicación, se ilustra además la restricción para acceder a los repositorios de datos. Aun

cuando se no ilustran todas las relaciones -líneas rojas- entre bases de datos y Microservicios, no será posible acceder de forma directa a los repositorios de datos haciendo uso de: Microservicios, Servicios, Componentes o cualquier otro mecanismo que se encuentre por fuera del Bounded Context al que pertenece dicho repositorio; excepto, si este se realiza por medio del bus de integración, lo que implica que toda necesidad de integración ha de realizarse con la intermediación del bus JBoss FUSE.

5.1.3 Componente de Servicio

En lugar de pensar en servicios dentro de la arquitectura de Microservicios, un mejor enfoque consiste en pensar en los componentes del servicio, que pueden variar en granularidad de un solo módulo a una gran parte de la aplicación. Los componentes de servicio contienen uno o más módulos (por ejemplo, clases de C#) que representan una función de propósito único (por ejemplo, importar archivos en formato plano) o una parte independiente del nuevo sistema (por ejemplo, Módulo de Transacciones).

5.1.4 Ambiente Distribuido

El modelo de Arquitectura de Microservicios se define como una arquitectura distribuida, lo que significa que todos los componentes dentro de la arquitectura están totalmente desacoplados y accedidos a través de algún tipo de protocolo de acceso remoto (por ejemplo, REST, SOAP). La naturaleza distribuida de este patrón de arquitectura es cómo logra algunas de sus características superiores de escalabilidad e implementación.

Nota: para los escenarios en los cuales se ha de convivir con un sistema legado y la parte evolutiva se aborda con el estilo arquitectónico de los Microservicios, el nuevo paradigma orienta la solución a realizar implementaciones y despliegues en ambientes distribuidos; sin embargo, no se puede perder de vista, que la solución arquitectónica surge como una alternativa de implementación de mínimo impacto en el sistema legado –si existe-, pero sin dejar de lado la adopción del nuevo paradigma tecnológico; por lo tanto, es impensable, bajo este esquema, que la solución no contemple un comportamiento Evolutivo, el cual se refleja en los mecanismos de comunicación entre el nuevo sistema y el legado, debido a la persistencia de los datos que conforman el nuevo modelo de Microservicios; mientras este se ampara en las definiciones de modelo independiente, va en contravía con los esquemas del sistema legado, el cual por su naturaleza Centralizada, no permite ceder terreno en términos de dominio de datos, muy propio del modelo canónico.

5.1.5 Administración de Datos Descentralizada

Unos de las características de los Microservicios en que sus datos son autónomos e independientes. Se enfatiza aplicar la noción de diseño de contexto delimitado –*Bounded Context*- por el dominio. DDD divide un dominio complejo en múltiples contextos delimitados y traza las relaciones entre ellos.

5.2 Convivencia Evolutiva: Legado VS Microservicios

Es importante tener claridad que en los procesos evolutivos de las aplicaciones en los cuales se hace presente un sistema legado, este continua "*Vivo y Operativo*"; sin embargo, la evolución de este, representado en nuevas funcionalidades y/o características, serán abordadas en un nuevo sistema de Microservicios, el cual adopta nuevos paradigmas tecnológicos; a nivel arquitectónico podemos mencionar los Microservicios y todo lo que ello conlleva, además de lenguajes y frameworks, los cuales se detallan más adelante en el documento.

Cuando se requiere reescribir el sistema legado, se hará uso del "*StranglerApplication Pattern*" de Martin Fowler. El *Patrón Estrangulador de Aplicación* permite abordar el nuevo sistema, acotando

la materialización de los riesgos, evitando hacer una transición de un "*solo golpe*". El patrón, ofrece valor de forma constante, con los despliegues frecuentes es posible monitorear el progreso más cuidadosamente. Además, los ciclos de liberación de producto permiten visualizar funciones innecesarias del legado que no deben ser reescritas en el nuevo sistema.

La convivencia entre ambos sistemas -Legado y Microservicios- se realizará bajo la premisa de "Consumir Servicios Existentes" y "Evitar" hasta donde sea posible, intervenir el sistema legado. Las necesidades de información por parte de los Microservicios, se consumirá por medio de servicios WEB existentes en el legado -si satisface las necesidades-, cuando ***no existe*** un servicio que satisface dicha necesidad, se deberá implementar la funcionalidad bajo el nuevo paradigma de los Microservicios.

5.3 Arquitectura Interna de Microservicios

5.3.1 Domain-Driven Design Microservices

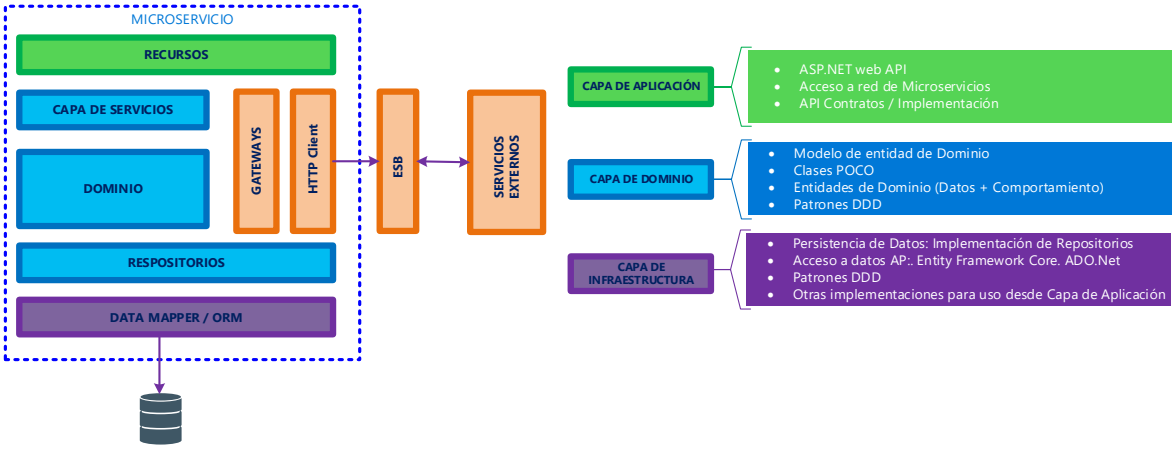

Ilustración 3. Diseño de Microservicios Orientados al Dominio (DDD)

Recursos: actúan como *Mapeadores* entre el protocolo de la aplicación expuesto por el servicio y los mensajes a los objetos que representan el dominio. Por lo general, son delgados, con la responsabilidad de verificar la cordura de la solicitud y proporcionar una respuesta específica de protocolo de acuerdo con el resultado de la transacción solicitada.

Dominio: Por lo general toda la lógica del servicio reside en un modelo de dominio que representa el dominio del negocio. De estos objetos, los *Servicios* se coordinan a través de múltiples actividades de *Dominio*, mientras que los *Repositorios* actúan en colecciones de entidades de dominio y a menudo tienen respaldo de persistencia.

Gateways: Los Gateways -puerta de enlace- encapsulan el paso de mensajes con un servicio remoto, clasificando solicitudes y respuestas desde y hacia objetos de dominio. Es probable que use un cliente que entienda el protocolo subyacente para manejar el ciclo de solicitud-respuesta.

Repositorios: Excepto en los casos más triviales o cuando un servicio actúa como un agregador en todos los recursos propiedad de otros servicios, un Microservicio deberá ser capaz de conservar los objetos del dominio entre las solicitudes. Es posible hacer uso de mapeadores, dependiendo de la complejidad de los requisitos de persistencia.

La implementación de Repositorios -*Repository Pattern*-, permitirá simplificar la administración de los datos, incrementar el desempeño y permitir la implementación de pruebas unitarias mediante el uso de Mocks. Se descarta adoptar clases DAL, debido que dicho enfoque realiza las operaciones

de forma directa sobre la base de datos, en lugar de dar un tratamiento a los datos en memoria, tal como lo hace EF Core.

Capa de Aplicación: Permite definir el trabajo y dirige los objetos del dominio para resolver los trabajos a realizar. La capa de aplicación es la encargada de "coordinar" el trabajo entre los objetos hacia las siguientes capas. No contiene reglas de negocio ni conocimiento alguno del negocio, debe permanecer delgada y liviana. No maneja estado del negocio, pero es posible que maneje el estado de las tareas que se encuentran en ejecución.

Para la capa de aplicación se implementará un proyecto ASP.NET Core Web API, el cual será encargado de interactuar con los Microservicios o el bus de integración que se ha definido -JBoss FUSE-, los cuales se utilizan generalmente desde la capa de presentación.

Capa de Dominio: No debe tomar dependencias con ninguna capa. Solo es posible tener dependencias con las bibliotecas del .NET Core y paquetes de NuGet.

Las Entidades de Dominio (Datos + Comportamiento). Las Entidades de Dominio deberán ser totalmente desconectadas, serán las encargadas de transportar los datos entre las diferentes capas, pero adicionalmente, deberán contener la lógica del dominio relativo a cada entidad; de lo contrario se estaría incurriendo en el *Antipatrón de Modelo de Objetos Anémico -Martín Fowler-.*

Las clases entidad deberán ser independientes de las tecnologías concretas de acceso a datos y deberán desconocer el interior de los repositorios -clases POCO-. Los detalles técnicos son delegados a la capa de infraestructura. Lo anterior se fundamenta en los principios: *Ignorancia de Persistencia e Ignorancia de Infraestructura*.

Capa De Infraestructura: permite la implementación de elementos que son útiles a la capa de aplicación; determina la manera como se mantiene los datos en memoria -entidades de dominio- y la manera como se persisten los datos en los repositorios.

5.3.2 Dependencias Entre Capas del Modelo DDD

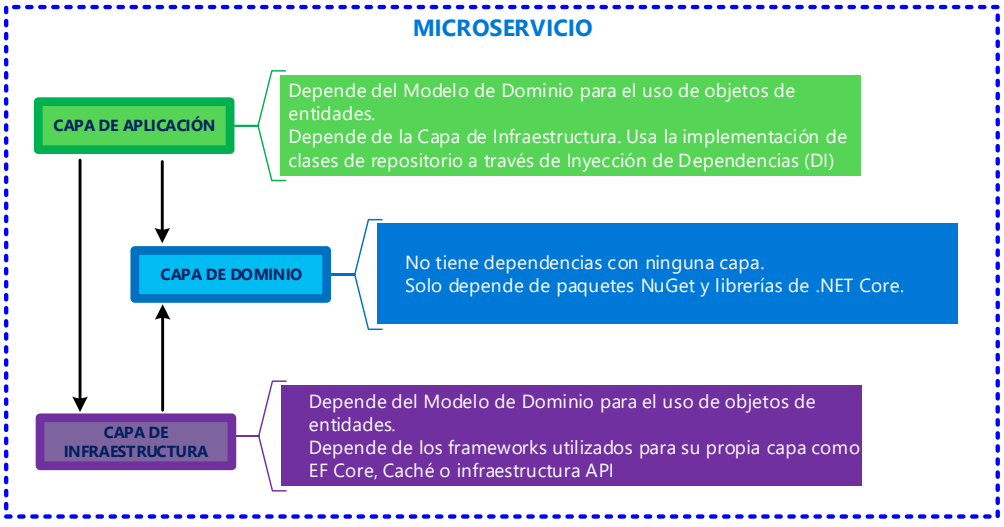

Ilustración 4. Dependencias Entre Capas Modelo DDD

5.4 Microservicios

La arquitectura propuesta deberá dotar el sistema de *Nuevas Capacidades*, las cuales se materializan al implementar Microservicios para satisfacer las necesidades que se plantean;

además, un análisis de mayor detalle podrá reflejar la necesidad de construir nuevos servicios para extraer información del legado –si este existe-.

Con el uso de Contenedores, se abre la posibilidad de operar bajo el concepto de *Multi-Plataforma*, para obtener y aprovechar las mejoras ventajas de la tecnología, las máquinas virtuales pueden "correr" con sistema operativo Windows o Linux. Se ha determinado que la primera opción a nivel de sistema operativo sea Linux.

El documento técnico de cada proyecto, deberá contemplar una sección de especificación para cada Microservicio que se propone implementar en el proyecto.

El clúster de contenedores será administrado por *Kubernetes*.

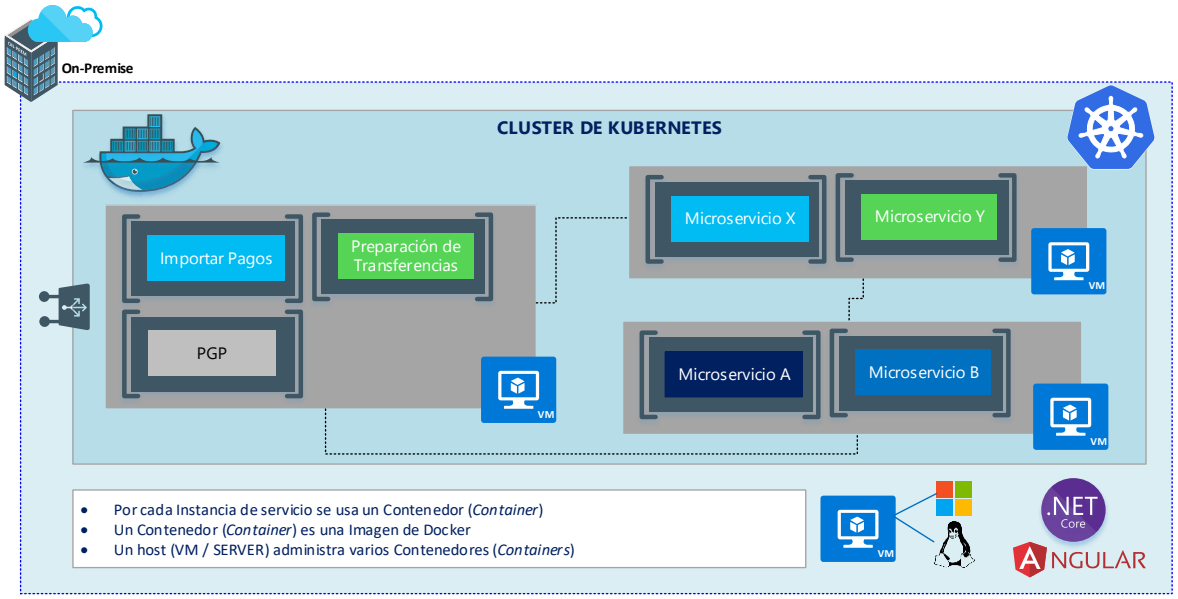

Ilustración 5. Microservicios – Docker y Kubernetes

5.4.1 Especificaciones

El sistema que se implemente deberá cumplir con las características de Alta Escalabilidad, de tal forma que permita que sus subsistemas verticales escalen horizontalmente de forma autónoma; lo anterior se fundamenta en la premisa funcional, cuyo análisis indica que algunos componentes requieren mayor escalabilidad que otros.

El sistema deber contar con las características que le permitan ser flexible en los entornos de infraestructura, ya que esté podrá tener giros técnicos que obligue a realizar despliegues en entonos híbridos -*On-Premises y nubes públicas*-, además la base del sistema operativo, tampoco deberá representar obstáculos al momento de contener las aplicaciones, ya sea en ambientes Windows o Linux, lo que permitirá cumplir con las características de multiplataforma, muy propio del nuevo paradigma tecnológico que se pretende adoptar.

Las características de *Mantenibilidad* no se pueden hacer esperar, ya que el objetivo pretende realizar despliegues de producto con mayor frecuencia, con impactos mínimos sobre el ambiente productivo, lo que implica que los cambios y/o evoluciones que se realicen, no deben presentar obstáculos de dependencia entre componentes.

El sistema deberá contemplar desde su inicio la división de subsistemas autónomos, con una orientación a Microservicios, los cuales estarán en contenedores, en los cuales cada Microservicio corresponde a un Contenedor.

La comunicación se realizará mediante protocolos HTTP -REST- al interior del Microservicios, inclusive se considera el mismo protocolo hacía afuera, siempre y cuando no se aumente la latencia entre los Microservicios. Cuando sea necesario propagar información con eventos de integración se deberá hacer uso de mecanismos asincrónicos, sin perder de vista que la orquestación de dichas integraciones se realizará sobre el ESB *JBoss FUSE* de Redhat.

El uso de Microservicios, define casi de forma implícita para su comunicación entre Microservicios, adoptar coreografías en lugar de orquestaciones. La presencia del ESB no implica que sea estrictamente necesario usar orquestaciones, ya que este, -el ESB- , está dotado y preparado para orquestar o coreografiar según sea el caso, el cual deberá ser analizado en profundidad, a fin de obtener el mejor y mayor provecho de las tecnologías que hacen parte de la arquitectura.

5.5 Consideraciones Generales

- ✓ La *Autenticación* de usuarios hará uso del C*omponente Corporativo de Autenticación* a nivel de aplicación, el cual implementa definiciones y características que hacen uso de directorio Activo a través del *Protocolo Ligero Simplificado Para Acceso a Directorios* LDAP (por sus siglas: ***L****ightweight* ***D****irectory* ***A****ccess* ***P****rotocol*).
- ✓ La *Autorización* a las funcionalidades de la aplicación para el usuario autenticado, hará uso del *Componente Corporativo de Autorización.*
- ✓ Las implementaciones concernientes a la Navegabilidad del sistema harán uso del *Componente Corporativo Navegabilidad*.
- ✓ Todas las necesidades enmarcadas en el entorno de operaciones que se realicen sobre los datos sean estos de Lectura, Escrita, y/o Lectura/Escritura; estarán cubiertas con el uso del *Componente Corporativo de Acceso a Datos*.
- ✓ Todo el tratamiento de evidencias, etiquetadas como errores y/o eventos, los cuales hacen parte de la información que aporta capacidad de análisis de traza en el comportamiento del sistema, hará uso del *Componente Corporativo Logger* para el registro de dicha información.
- ✓ Las terminaciones abruptas que tengan lugar en alguno de los flujos del sistema, independiente de su nivel de complejidad, será susceptible de ser registrado como una traza verificable; para ello, hará uso del *Componente Corporativo Registro de Excepciones*.
- ✓ El cifrado y descifrado de datos estará a cargo del *Componente Corporativo Criptografía*.
- ✓ La orquestación de los procesos para necesidades de integración hará uso del *Bus Corporativo ESB*.
- ✓ El frontend deberá contar con características *Responsive Web Design*, para que la visualización se ajuste de forma correcta, conservando los lineamientos gráficos que se definen para el "*look*" del sistema, siendo posible la visualización y operación del sistema en los diversos dispositivos que se encuentran disponibles (desktop, tables, teléfonos inteligentes). Es posible que algunas características funcionales, por su definición o por lineamientos de seguridad, que solo se habiliten para ser desplegadas sobre dispositivos desktop; sin embargo, el sistema, a fin de cumplir con el atributo de Mantenibilidad, este debe adaptarse sin problema alguno, cuando tenga lugar la visualización o se habilite para otros dispositivos alguna de las funcionalidades del sistema.

- ✓ No se contemplan implementaciones de reglas de negocio en la capa de presentación, por definición, dichas reglas deberán hacer parte del backend.
- ✓ La interfaz gráfica estará orientada por los lineamientos corporativos y/o manual de identidad que define los aspectos como tipografías, imágenes corporativas y todos los elementos que figuran como directrices para manejo de la marca XM.
- ✓ Se deben considerar mecanismos que permitan verificar la Disponibilidad y Capacidad del Servicio, debido que ante una deficiencia en uno de estos atributos, más allá de lo que su nombre indica, puede generar una "*caída en cascada*", dejando por fuera gran parte o la totalidad del sistema, por degradación en capacidad o por disponibilidad de los servicios.
- ✓ Los mensajes informativos, de advertencia o error deben ser manejados y presentados por la solución; estos deben ser claros y precisos, evitando involucrar tecnicismos o presentando información que no es útil al usuario de la aplicación.

5.6 Restricciones

5.6.1 Restricciones de negocio

- ✓ A fin de mantener la custodia y seguridad de los datos, se restringe el despliegue de las aplicaciones en la nube pública, para todos los elementos que conforman la capa de datos.

5.6.2 Restricciones tecnológicas

- ✓ Se consideran como herramientas de desarrollo, las que conforman la suite propuesta por Microsoft .net.
- ✓ El frontend se realizará haciendo uso de la tecnología *Angular 5*.
- ✓ El contenido de las páginas contempla el uso de HTML5.
- ✓ Los estilos y apariencia se realizan mediante el uso de CSS3.
- ✓ TypeScript / JavaScript como lenguaje para implementar scripts del lado del cliente.
- ✓ El backend se implementará mediante el uso de tecnología .NET CORE 2.1
- ✓ C# como lenguaje para la implementación de backend.
- ✓ Swagger como herramienta para diseñar, construir, documentar y probar API´s.
- ✓ Se contempla como tecnología de contenedores el uso de Dockers.
- ✓ Como motor para repositorio de datos, se hace uso de SQL Server 2017.
- ✓ Reporting Services en modo SharePoint ofrecerá los servicios de hosting de los reportes, además de las necesidades de publicación y subscripción de estos.
- ✓ JBoss FUSE como plataforma de integración, orquestación y /o coreografía entre servicios.

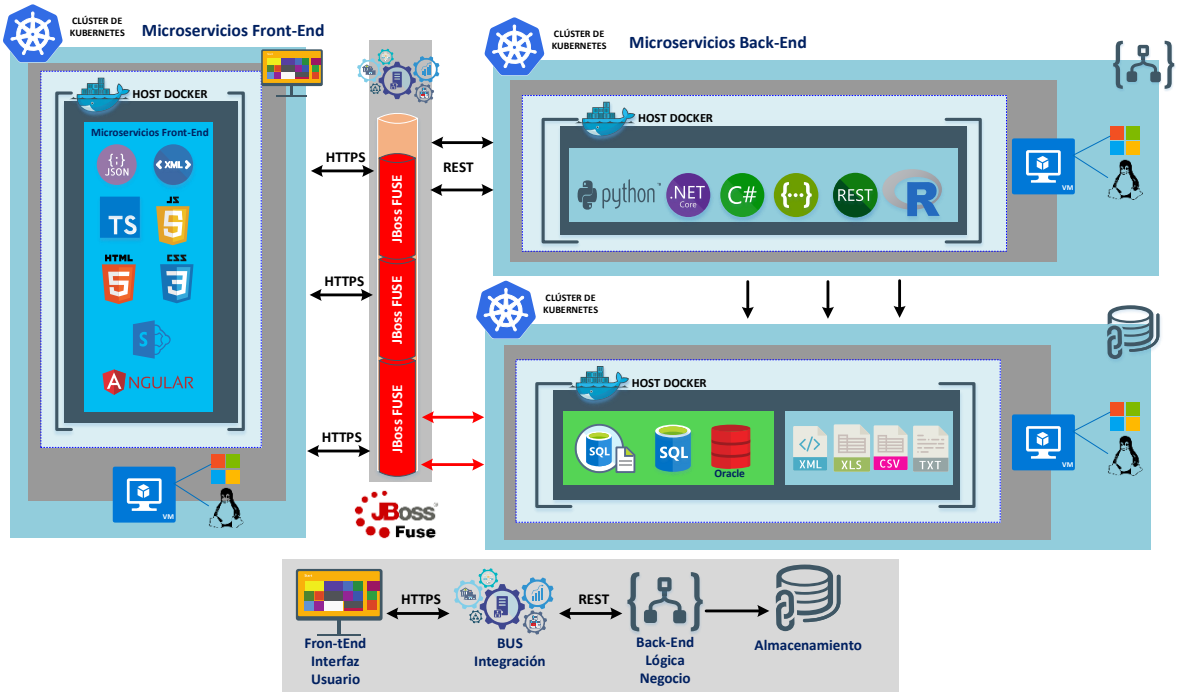

Ilustración 6. Tecnologías Habilitadas Para el Desarrollo de Aplicativos

5.6.3 Restricciones Técnicas

- ✓ Se evitará el uso de recursividad en los procedimientos almacenados, debido que puede provocar fallos inesperados en la base de datos al alcanzar el límite de invocaciones recursivas soportadas por SQL Server.
- ✓ Para SQL Server, se restringe totalmente el uso de cursores en los procedimientos almacenados; debido que esta característica funcionalidad impacta el desempeño de la base de datos, aumenta el tiempo de bloqueos de los objetos y disminuye la escalabilidad general del sistema.
- ✓ Se recalca la importancia, de tener claridad, que la estrategia -en primera instancia- no migrará datos a los Microservicios, estos deberán permanecer en el repositorio legado, hasta ver el comportamiento de las nuevas capacidades que se extienden del sistema legado a Microservicios, una vez se determine que el comportamiento del Microservicio obedece a las expectativas esperadas, se realizará la migración de los datos para dar completitud al Microservicio.
- ✓ Para las capacidades de Reportes -Reporting Services-, la manera de acceder a los datos no se realizará utilizando el mecanismo *Database Pull Model*, aun cuando es un mecanismo de acceso rápido y fácil, se generan altas dependencias entre el servicio de datos y el servicio de informes. En lugar de ello, se utilizará el mecanismo *HTTP Pull Model o Batch Pull Model.* Mientras que el primero permitirá realizar una llamada HTTP a cada servicio del cual necesite los datos, el segundo orienta a realizar una extracción en por lotes cuyo destino es un repositorio de informes con datos agregados y reducidos.

5.6.4 Restricciones generales

- ✓ No se consideran evaluaciones de costo de las plataformas.
- ✓ No se contemplan migraciones de servicios de sistema legado a la nueva arquitectura de Microservicios.
- ✓ En términos de frontend, para separar el contenido de la forma, los estilos no deberán estar inmersos en el contenido HTML, en su lugar se hará uso de las Hojas de Estilo en Cascada CSS (*Cascading Style Sheet*).

5.7 Metas Arquitectónicas

5.7.1 Escalabilidad

El sistema deberá estar en capacidad de ofrecer un crecimiento escalable, tanto vertical como horizontal. Los componentes deben adoptar la capacidad de crear instancias múltiples, sin generar conflictos de procesamientos concurrentes, aun cuando estos se ejecuten por instancias múltiples o cuando dicho componente haga parte de otro ambiente de ejecución; es decir, un componente cuya instancia no sea el mismo padre.

El despliegue de los Microservicios deberá hacerse dentro de *Contenedores,* para obtener los beneficios de empaquetado de una mayor cantidad de Microservicios en un solo host, lo que resultará en una utilización más eficiente de los recursos.

Para precisar, la Escalabilidad además significa que la latencia que percibe un usuario al realizar solicitudes al sistema -por ejemplo, una consulta retorna los datos en 2 segundos-, el sistema se comporta de forma uniforme con la carga de N usuarios -por ejemplo 10-; si en lugar de ello, la latencia se incrementa a medida que hay concurrencia de usuarios, significa que el sistema no está siendo escalable. Es importante determinar el umbral de escalabilidad del sistema, para garantizar la escalabilidad real y poder determinar el momento preciso en el cual el sistema deberá escalar de forma horizontal o vertical.

5.7.2 Desempeño

El desempeño está sujeto a la concurrencia que se determina para cada sistema, y es directamente proporcional al escalamiento que se espera de la aplicación; por lo tanto, se debe definir el desempeño esperado de forma constante para la concurrencia que se define; de lo contrario, el Microservicio no cumpliría con las definiciones de Desempeño y mucho menos de escalamiento del Microservicio.

5.7.3 Concurrencia

Cada proyecto deberá especificar la concurrencia esperada en el sistema, inclusive la concurrencia debe ser especificada en detalle para cada Microservicio, debido que cada uno de ellos, al tener dominios independientes, las implicaciones en cuento a concurrencia pueden variar de uno a otro. De esta manera es posible determinar el escalamiento automático de los Microservicios, ya que cada uno de ellos, debe tener la capacidad de escalar sin depender de los demás Microservicios con los cuales conforman el ecosistema.

5.7.4 Resilencia

Es importante dejar claridad acerca de los que se espera. No existe un software infalible, libre de errores, pero si es posible tener un sistema que reaccione de forma “positiva” ante las eventualidades que ocasionan caídas del sistema, ya sean parciales o totales; por lo tanto, el sistema deberá estar dotado de los mecanismos que permitan hacer un monitoreo constante sobre

la salud del sistema, de manera que las acciones a tomar permitan restablecer en el menor tiempo posible la operación del sistema.

La implementación del patrón Circuit-Breaker, aumenta la Resilencia, evitando que una aplicación intente repetidamente una operación que probablemente falle. La sección patrones, menciona en detalle el uso de este tipo de patrón.

5.7.5 Caché

Se deben tener en cuenta las capacidades de caché suministradas por la plataforma .net, con el fin de almacenar fragmentos de las páginas que no presenten variación durante determinados lapsos de tiempo, cuyo objetivo es mejorar el desempeño, evitando que se hagan repetidos llamados a la base de datos de la solución de la misma información, cuya consecuencia es la optimización de los recursos. De igual forma utilizar el caché para preservar información que sea recuperada desde la base de datos y cuya variabilidad no sea frecuente en el sistema, con el fin de evitar que se realicen accesos constantes a recursos de datos que no son cambiantes en el tiempo.

Las páginas de presentación deben minimizar tanto como sea posible el almacenamiento de información a nivel de objetos sesión, pues impacta el rendimiento de la aplicación por las operaciones de serialización implícitas en el proceso de almacenamiento y recuperación de grandes cantidades de datos.

Se debe definir para cada caso en el que se utilice el caché, la duración y la dependencia del mismo.

5.7.6 Interfaz Gráfica

Buscar facilidad de uso (Usabilidad) y simplicidad de la interfaz de usuario antes que elaborar características muy sofisticadas que dificulten o demoren significativamente las labores de diligenciamiento del usuario.

Utilizar tanto como sea posible, representaciones gráficas consolidadas de la información, de modo que se garantice tener retroalimentación rápida con el nivel adecuado de detalle; es decir, se debe buscar que las pantallas presenten la información relacionada de la manera más compacta posible, para que el usuario no tenga que incurrir en desplazamientos verticales u horizontales sobre el contenido de la página desplegada.

5.7.7 Dominio

Se deberá establecer un análisis detallado del dominio de datos de cada Microservicio, dicha actividad orienta el Bounded Context de cada uno de ellos.

5.7.8 Datos

Conexiones: La conexión a los repositorios de datos propios del Microservicio, se realizan haciendo uso del Componente Corporativo de Acceso a Datos, el cual los métodos necesarios para realizar las conexiones a los repositorios de datos, además de ofrecer los mecanismos que permitan realizar operaciones sobre estructuras de datos.

Recuperación de Datos: Para la recuperación de datos que no impliquen lógica, ni complejas funciones de agregado, se hará uso de *Linq-To-Entities*, de manera que sea posible conservar el asilamiento que ofrecen las entidades, siendo “agnósticos” sin preocuparse por el tipo de motor donde se hospedan los datos. La recuperación de datos que implique acceso a repositorios que conservan un alto volumen de datos o se requiere establecer para su recuperación funciones complejas de agregados o establecer relaciones con múltiples tablas; se hará uso de

procedimientos almacenados, los cuales serán ejecutados haciendo uso del *Componente Corporativo de Acceso a Datos*.

Operaciones de Persistencia y Eliminación: Las operaciones que requieren hacer Inserciones y/o, Modificaciones sobre los datos, se realizan haciendo uso de *Linq-To-Entities,*

Nivel de Aislamiento (Isolation Level): Se deberá establecer el nivel de aislamiento adecuado a fin de controlar las operaciones que modifican las estructuras de los datos. El nivel de aislamiento deberá garantizar que no se realizarán bloqueos ni a nivel de tabla, ni a nivel de registro, excepto casos explícitos en los cuales se deben tener un control sobre las transacciones, obligando a realizar bloqueos a nivel de registro. Se deben evitar los bloqueos a nivel de tabla. Una buena técnica para evitar interbloqueos cuando se realizan consultas sobre los datos; consiste en permite realizar "*Lecturas Sucias*" (*Dirty Read*), de esta manera se incrementan los tiempos de respuesta de las consultas y los interbloqueos o "abrazos mortales" se estarán evitando en gran medida.

5.7.9 Consistencia e Integridad de Datos

Todo Microservicio deberá ser soberano de sus propios datos, lo que significa que todos los Microservicios que se implementen, todo dato que haga parte de su dominio deberá proveer los mecanismos para administrarlos, de tal manera que los servicios estén *Altamente Desacoplados*. Sin embargo, es posible que se presenten escenarios de *Redundancia de Datos*, abriendo la posibilidad de presentar problemas de *Integridad de Datos*.

Adoptar el uso de transacciones atómicas permitirá garantizar la integridad de los datos en una sola transacción, de manera que cuando se presente una falla en alguno de los agregados, toda la transacción será anulada mediante los mecanismos *Rollback.* Este enfoque difiere de la *Consistencia Eventual* en el hecho de que al fallar alguna de las persistencias de los eventos secundarios, se deben realizar transacciones de compensación; en términos de código, representa una mayor complejidad, además la latencia aumenta por la cantidad de transacciones que se generan, las cuales se asocian a la cantidad de agregados que disparan los eventos secundarios de la transacción principal y la cantidad de bloqueos en la base de datos será sustancial.

En resumen, es más simple y efectivo implementar código para transacciones únicas -atómicas- en lugar confiar en la coherencia eventual entre los agregados, al tener grano fino transaccional mediante la *Consistencia Eventual*.

Es posible que las transacciones atómicas no sean necesarias para todos los escenarios, en cuyo caso, se deberá evidenciar por el experto de dominio las reglas que determinen y orienten el mejor camino a seguir para garantizar la consisten cae integridad de los datos.

5.7.10 Comunicación Entre Microservicios

La regla consiste en eliminar las dependencias entre Microservicios, de lo contrario, es posible que la definición arquitectónica no sea resistente (*resilient*); por lo tanto, el diseño debe estar orientado a que los Microservicios se preparen para un degradamiento o caída de una API de servicio que se consume. Evocando uno de los principios que se profesan sobre los Microservicios, al afirmar que estos son soberanos de su dominio de datos y su lógica de negocio, se determina que: ante la necesidad de exponer información -*solo cuando sea necesario*- se hará a través de un API de servicio, sin importar quien ha de consumirlo. Para hacer alarde de la simplicidad y flexibilidad se sugiere que se adopte REST para el dote coreográfico orientado por eventos, en lugar de utilizar el protocolo WS-* que se orienta a procesos centralizados por medio un orquestador de procesos. La comunicación podrá hacer uso de los protocolos HTTP request/response (API´s) y mensajería asíncrona ligera, cuando se requiere hacer actualizaciones de múltiples Microservicios.

Para realizar comunicación Asíncrona, se debe hacer uso del protocolo de mensajería como **AMQP (*Advanced Message Queuing Protocol*),** el cual utiliza un bróker para encolar los mensajes.

Se debe considerar el tipo de receptos, ya que, si este es *Múltiple*, la comunicación debe ser *Asíncrona,* y se deberá adoptar e implementar el patrón *Publicar/Subscribir* (*Publish/Subscribe*).

Se hará uso de HTTPS de forma ***síncrona*** y protocolos de mensajería ligera para uso de colas como soporte para la comunicación ***asíncrona*** entre Microservcios.

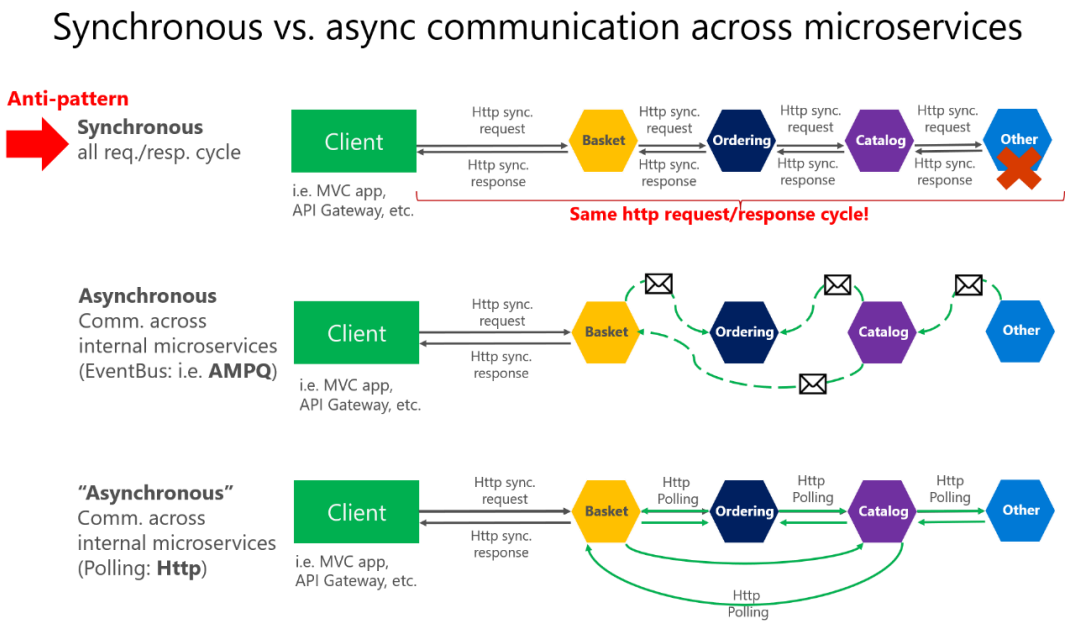

Ilustración 7. Comunicación en Arquitectura de Microservicios

En los escenarios en los cuales es imperativo tener una alta integración entre Microservicios; se deberá establecer comunicación asíncrona para realizar la integración entre los Microservicios, sin embargo, antes que todo, es necesario revisar en detalle, y encontrar el motivo de la necesidad de la integración en alta escala, ya que, si esto ocurre, es posible que se esté incurriendo en un mal diseño de los *Bounded Context* de los Microservicios y se debe realizar un rediseño de estos.

El formato de los datos es una variable que puede afectar los tiempos de respuesta de los mensajes que fluyen entre los Microservicios, o los que el mismo Microservicio espera auto-proveerse. Cabe resaltar que aun cuando el formato binario ofrece mejores tiempos de respuesta, este no está constituido como un estándar, mientras que los formatos de datos como XML y JSON, si lo son. El uso de formatos binarios se usa generalmente para uso interno del Microservicio, para uso externo se utiliza el formato XLM o JSON.

5.7.11 Topología API REST

La topología basada en REST de API es útil para los Microservicios pequeños y muy detallados, que contienen uno o dos módulos que realizan funciones específicas e independientes del resto de los servicios. Para acceder a estos componentes de servicio se utilizará una interfaz basada en REST, implementada a través de una capa de API basada en la web implementada por separado.

5.7.12 Topología de Mensajería Centralizada

En lugar de usar REST para acceso remoto, esta topología utiliza un agente de mensajería centralizado liviano (por ejemplo, **ActiveMQ**, HornetQ, etc.). Cuando se mira esta topología, es de vital importancia no confundirla con el patrón de arquitectura orientada a servicios o considerarla como "SOA-Lite". El intermediario de mensajes liviano que se encuentra en esta topología no realiza ninguna orquestación, transformación o enrutamiento complejo; es solo un transporte liviano para acceder a los componentes del servicio remoto.

5.8 Variaciones Arquitectura Candidata

Las variaciones de la arquitectura propuesta se centran en la infraestructura. El despliegue de la solución puede realizarse de forma tradicional al hacer uso de la infraestructura On-Premises, dejando de lado -tan solo de momento-, la posibilidad de que el despliegue se realice en una infraestructura híbrida, tanto On-Premises como la nube Azure de Microsoft.

El contemplar un despliegue hibrido, posibilita la capacidad de ser más agiles en el aprovisionamiento de los recursos y el despliegue continuo, el cual se considera como uno de los grandes pilares en la adopción del nuevo paradigma tecnológico. La capacidad de monitorear los recursos, a fin de poder conocer el comportamiento del nuevo ambiente, para colocar en una balanza y poder aseverar con los debidos elementos los beneficios de ese nuevo mundo -que ya no lo es- comparado con el tradicional.

Hablar de Contenerización de aplicaciones, puede significar tanto infraestructura como tecnología blanda. Los contenedores, no son “obligantes”, la solución propuesta, no se “aferra” a “tener que” contenerizar todos los Microservicios. Un buen ejercicio consiste en realizar pruebas de concepto, cuyo objetivo sea el de obtener el conocimiento necesario para determinar la viabilidad del uso de contenedores.

5.9 Vista de Aplicación

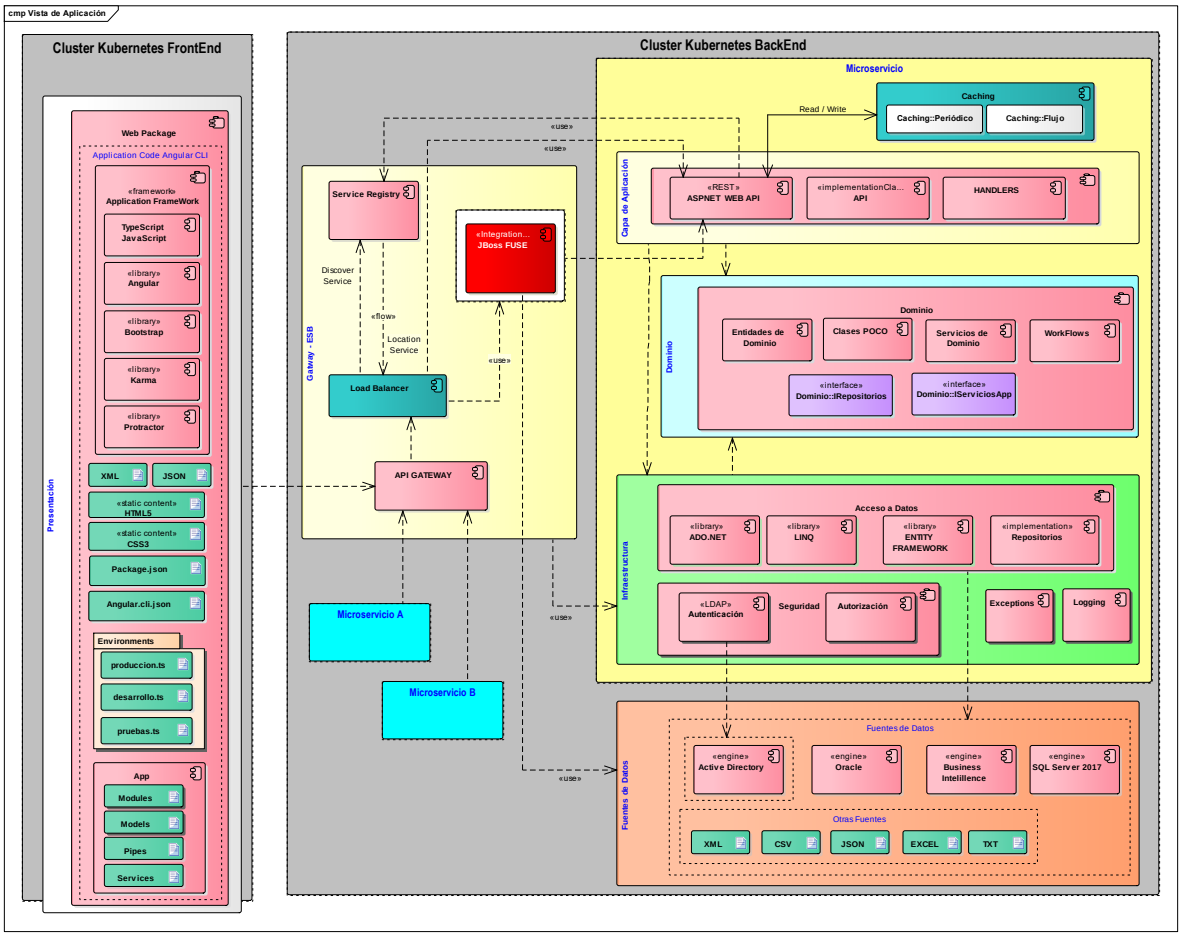

Ilustración 8. Vista de Aplicación

5.10 Vista de Despliegue

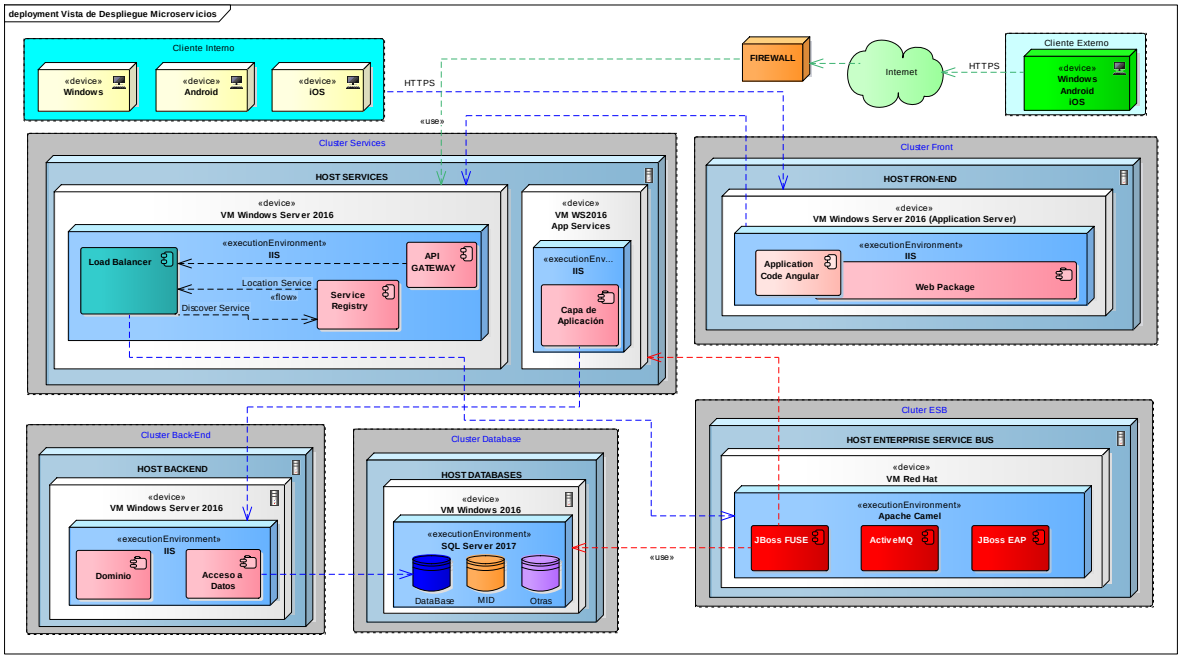

Ilustración 9. Vista de Despliegue

6 Patrones / Antipatrones

6.1 Antipatrón Timeout

La Disponibilidad del servicio es la capacidad que se ofrece a un consumidor para conectarse con el servicio y poder enviar su solicitud. La Capacidad de Respuesta del servicio es el tiempo que toma el servicio en dar respuesta a esa solicitud del consumidor.

Cuando hay disponibilidad del servicio, pero este no responde, el consumidor puede optar por esperar indefinidamente a que llegue la respuesta solicitada o esperar algún tiempo que se ha determinado mientras este responde. Definir un tiempo de espera para la capacidad de respuesta del servicio parece ser una buena idea, pero está decisión puede llevarnos por el camino del Antipatrón TimeOut (*Tiempo de Espera*).

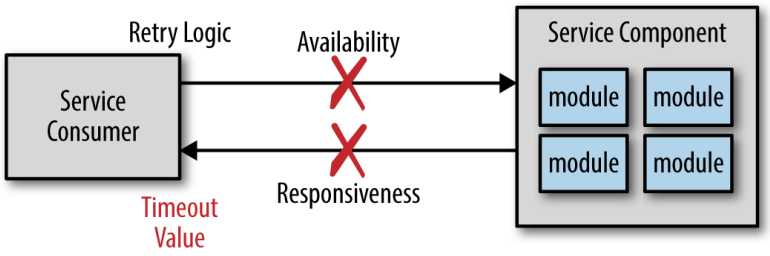

Ilustración 10. Disponibilidad del Servicio vs Capacidad de Respuesta

En lugar de confiar en los valores de tiempo de espera para las llamadas a los servicios remotos, se tiene la posibilidad de tener un mejor enfoque, haciendo uso del patrón *Cirtcuit-Breaker*

(Interruptor de Circuito). El *Circuit-Breaker* deberá permanecer monitoreando el servicio remoto hasta que esté se encuentre receptivo nuevamente.

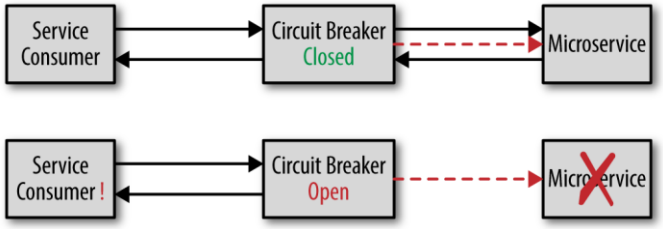

Ilustración 11. Patrón Circuit Breaker (Interruptor de Circuito)

El Circuit Breaker monitoreará el servicio remoto asegurándose que esté "vivo" y receptivo, lo que comúnmente se conoce como el "*Hearbeat*" (látido del corazón). Este mecanismo permitirá conocer si el servicio remoto se encuentra Disponible.

Para conocer la Capacidad de Respuesta del servicio, se pueden utilizar Transacciones Sintéticas, las cuales son una técnica de supervisión presente en el mismo patrón *Circuit-Breaker*. La técnica consiste en enviar transacciones falsas de forma periódica (por ejemplo, cada 10 segundos). Las transacciones falsas (sintéticas), ejecutan toda la funcionalidad requerida dentro del servicio, permitiendo al Circuit-Breaker obtener una medida de la Capacidad de Respuesta del servicio. Cuando se opta por este tipo de supervisiones, las implementaciones en el servicio deben contemplar los mecanismos que permitan diferenciar entre una transacción falsa (Sintética) y una que viene directamente desde el negocio como transacción válida.

Exista otra alternativa de monitoreo, llamada *Real-Time User Monitoring* (Monitoreo de usuarios en tiempo real). La técnica consiste en monitorear las transacciones de producción reales, llevando un registro de la cantidad de transacciones realizadas, el registro permite conocer el "Umbral Transaccional" que se ha definido para el control. Una vez que se alcanza el Umbral, el Circuit-Breaker pasa a un estado semi-abierto, donde solo se permite cierto número de transacciones, hasta que la Capacidad de Respuesta del servicio llegue a la normalidad, el Circuit-Breaker se cierra permitiendo el paso de las transacciones que llegan.

Se deberán realizar pruebas de concepto para determinar cuál es el mejor mecanismo que se adapta a las necesidades funcionales que se han expuesto.

6.2 Antipatrón Database Pull Model (Reporting Services)

El bastión de la Arquitectura de Microservicios enfatiza que tanto servicios como los datos correspondientes, deben estar dentro del mismo *Bounded Context*; sin embargo, con respecto a los informes, esté bastión se convierte en la daga que puede hacer grandes estragos en los Microservicios. Ir de forma directa al repositorio para extraer los datos requeridos de un informe, quizás parece ser la manera más eficaz y simple de hacerlo, bajo el contexto de Microservicios, este mecanismo se convierte en un Antipatrón conocido como *Database Pull Model*, el cual proporciona dependencia entre los datos y el servicio; es decir, se incrementa el acoplamiento.

Uno de los mecanismos que permite evitar el Antipatrón *Database Pull Model*, es utilizar un mecanismo de extracción utilizando una llamada HTTP al servicio para solicitar los datos, mecanismo conocido como el patrón *HTTP Pull Model*. Aunque el modelo preserva el principio del Bounded Context, bajo algunos escenarios, es posible experimentar problemas de lentitud, debido a la latencia.

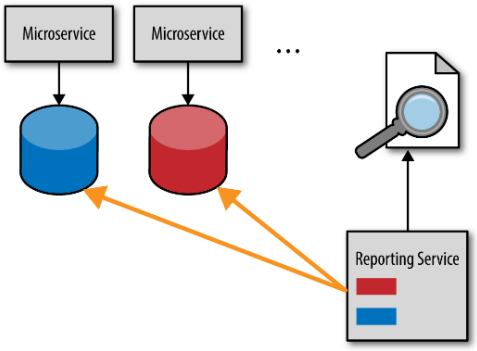

Ilustración 12. Antipatrón Database Pull Model

El patrón *Batch Pull Model* es una alternativa de solución que requiere mayor esfuerzo, la cual consiste es realizar una extracción de datos por lotes, dirigidas a un repositorio distinto, el cual representa información agregada y resumida para los informes.

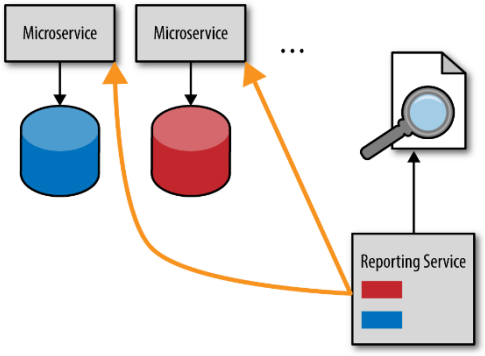

Ilustración 13. Patrón HTTP Pull Model

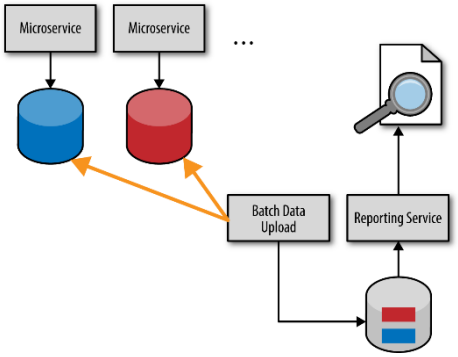

Ilustración 14. Batch Pull Model

Otra alternativa de solución para evitar el antipatrón es utilizar el modelo *Event-Base Push Model.* Sam Newman, en su libro *Building Microservices*, se refiere a esta técnica como una "*Bomba de datos"*. El modelo se basa en el procesamiento de eventos asíncronos para garantizar que la base de datos de informes tenga la información correcta lo antes posible.

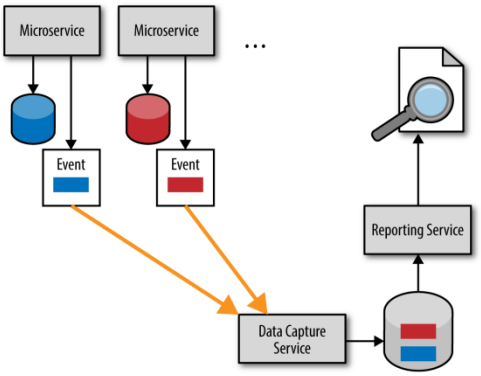

Ilustración 15. Event-Base Push Model

7 Otras Alternativas de Solución

7.1 Evolucionar el Legado

La posibilidad de implementar las nuevas capacidades y/o ajustes al sistema legado, sin adoptar un nuevo paradigma tecnológico que implique cambios de nuevas tecnologías, sigue siendo una realidad, ya que desde el punto de vista técnico, es totalmente viable evolucionar el producto con dichas funcionalidades; sin embargo, el cierre de brechas técnicas sigue impactando la “espera” para hacer cumplir las especificaciones y lineamientos que se definen desde la Gerencia de TI, los cuales se enmarcan en los documentos de Arquitecturas de Referencia.

7.2 Implementación Nuevas Capacidades Sin Microservicios

Las nuevas características de un nuevo sistema, bien pueden ser implementadas en una nueva arquitectura, totalmente ajena al uso de Microservicios, dicha arquitectura deberá ser la misma que se ha venido trabajando a nivel corporativo. La arquitectura SOA, podrá ser la guía para implementar las nuevas capacidades del sistema; sin embargo, se deben conservar la adopción de las nuevas tecnologías de desarrollo -Angular, HTML5, Framework 4.7/ .NET Core, entre otros-. En resumen, en este escenario sigue gobernando la Arquitectura SOA, pero con nuevos gregarios para el desarrollo e implementación de la arquitectura.