UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Diseño de aplicaciones SaaS sobre plataformas de Cloud Computing
Autores: Emiliano Nieto
Director: Mg. Patricia Bazán
Carrera: Licenciatura en Sistemas

Resumen

Cloud computing es un modelo de computación que permite demandar a través de internet recursos compartidos como procesamiento en servidores, almacenamiento, aplicaciones y servicios, los cuales pueden ser rápidamente aprovisionados y liberados sin ningún esfuerzo de administración por parte del proveedor de éstos servicios.
Los tres servicios esenciales de Cloud Computing son Infraestructura como Servicio (IaaS), Plataforma como Servicio (PaaS) y Software como Servicio (SaaS).
El presente trabajo tiene como objetivo investigar el diseño de aplicaciones SaaS sobre Plataformas como Servicio, analizando las características propias de una aplicación SaaS en combinación con los servicios que ofrecen los diferentes proveedores de PaaS, de forma tal de evidenciar la problemática técnica que involucra esta combinación, con el fin de identificar conceptos de diseño y de arquitectura que permitan construir soluciones óptimas.

Palabras Claves

Cloud Computing, Computación en la nube, IaaS, Infraestructura como Servicio, PaaS, Plataforma como Servicio, SaaS, Software como Servicio, Google App Engine, Windows Azure, Multi-tenant, Escalabilidad, Configurabilidad, Aprovisionamiento, Monetización.

Trabajos Realizados

Se analizó un caso de estudio concreto, relacionado con un proveedor de software que desea construir un sistema de alertas de noticias como un SaaS.
El caso de estudio muestra alternativas para implementar la característica de multi-tenant, configurabilidad y escalabilidad propias de una solución SaaS, en las Plataformas como Servicio Google App Engine y Windows Azure. Además muestra alternativas de implementación de ciertas características específicas de un sistema de alertas sobre una PaaS.

Conclusiones

La construcción de una aplicación web para ofrecerla como un SaaS implica entender correctamente los conceptos de multi-tenant, configurabilidad y escalabilidad, ya que estas características requieren conocimientos avanzados de diseño y programación, que no se presentan en aplicaciones web tradicionales.
Además, es necesario conocer a fondo la Plataforma como Servicio donde se alojará la aplicación, ya que estas plataformas tienen particularidades que las diferencian de los servidores web tradicionales.

Trabajos Futuros

El concepto de multi-tenant está en constante debate actualmente ya que atraviesa la arquitectura completa de una aplicación. Los trabajos futuros están relacionados en cómo impacta el concepto de multi-tenant en una arquitectura SOA, en el uso de identidad federada para la seguridad de la aplicación y en el uso de motores de bases de datos con soporte nativo de multi-tenant.

Fecha de la presentación: Junio de 2013

Agradecimientos

Este trabajo final, representa para mí, no solo un hito académico, sino también un hito personal y emocional. Este trabajo es una bisagra, de una etapa de mi vida, que comenzó hace muchos años, y que después de tantas idas y vuelvas, llega a su fin. A menudo los trabajos intelectuales, desafían la reserva emocional de quien los lleva a cabo, y no haber claudicado y haber sido perseverante, son las mejores lecciones que aprendí.

Quiero dedicar este trabajo a Roxana, mi mujer, que con su amor, me enseñó a confiar y a creer, devolviéndome la seguridad que necesitaba, y a Chiche y Fabe, nuestros perros, que siempre estuvieron al lado mío, día y noche, haciéndome una compañía invaluable. A ellos, que son mi familia, por siempre, ¡GRACIAS!

Por último, un agradecimiento profundo a la vida, por haberme puesto en el camino de esta profesión, sin la cual no hubiera podido ser lo que hoy soy.

INDICE PRINCIPAL

Indice de Tablas

Indice de Figuras

Introducción

Cloud computing según el National Institute of Standards and Technology (NIST), es un modelo de computación que permite demandar a través de internet recursos compartidos como procesamiento en servidores, almacenamiento, aplicaciones y servicios, los cuales pueden ser rápidamente aprovisionados y liberados sin ningún esfuerzo de administración por parte del proveedor de éstos servicios.
Estos servicios cloud están divididos en tres grandes categorías, las cuales se agrupan en una pila, ya que cada capa superior brinda un mayor nivel de abstracción para el usuario del servicio:

Software como servicio (SaaS): las aplicaciones son diseñadas para el usuario final, y son accedidas a través de la web.
Plataforma como servicio (PaaS): es un conjunto de herramientas y servicios diseñados para codificar y desplegar aplicaciones de forma fácil y eficiente.
Infraestructura como servicio (IaaS): es el conjunto de servicios de más bajo nivel, como hardware, almacenamiento, sistemas operativos.

Desarrollar aplicaciones sobre una plataforma como servicio implica un nuevo modelo de programación, ya que las aplicaciones no interactúan con los servicios de un SO tradicional, sino que hay una capa más de abstracción.
Actualmente existen muchos proveedores de PaaS, como Google App Engine, Windows Azure, Force.com, Amazon EC2 y muchos más, lo que implica que a la hora de desarrollar una aplicación sobre una plataforma de cloud, es necesario analizarla cuidadosamente para determinar sus ventajas y desventajas, tanto en lo técnico como en lo económico.
Además, el diseño de una aplicación SaaS tiene ciertos requerimientos técnicos que una plataforma de cloud debe soportar para poder alojar correctamente una aplicación de este tipo, por ejemplo:

- Escalabilidad y Escalabilidad Automática
- Múltiples Clientes o Multi Tenants
- Múltiples Usuarios por Cliente
- Transparencia y aislamiento entre clientes
- Redundancia y balanceo de carga
- Monitorización y facturación

Por lo esbozado anteriormente, se pone en evidencia que desarrollar una aplicación SaaS sobre una plataforma de cloud computing es un gran desafío desde el punto de vista del diseño y la programación de la aplicación, y no habrá una única solución a este problema, sino que será necesario tener en cuenta muchas consideraciones sobre la plataforma de cloud y sobre la naturaleza misma de una aplicación SaaS, por lo que esta tesis se propone echar un claro de luz sobre estas cuestiones.

El presente trabajo tiene como objetivo investigar el diseño de aplicaciones SaaS sobre Plataformas como Servicio, analizando las características propias de una aplicación SaaS en combinación con los servicios que ofrecen los diferentes proveedores de PaaS, de forma tal de evidenciar la problemática técnica que involucra esta combinación, con el fin de identificar conceptos de diseño y de arquitectura que permitan construir soluciones óptimas.

El trabajo se encuentra organizado en ocho capítulos en donde se abordan los siguientes temas:

- Capítulo 1: Introducción al concepto de Cloud Computing y a las características esenciales que lo componen, mención de los modelos de servicio y de despliegue.
- Capítulo 2: Desarrollo del concepto de Software como Servicio, su implicancia desde el punto de vista técnico y del negocio, se realiza comparaciones con otras formas de entregar software al usuario final.
- Capítulo 3: Se avanza con los requisitos técnicos de las aplicaciones SaaS que condicionan el diseño y la arquitectura de una aplicación.
- Capítulo 4: Se detallan técnicas de diseño y programación que son útiles para poder implementar el requisito técnico de configurabilidad de las aplicaciones SaaS.
- Capitulo 5: Se mencionan implementaciones concretas para llevar a cabo el requisito técnico de configurabilidad de las aplicaciones SaaS.
- Capítulo 6: Se analizan dos plataformas como servicio, Google App Engine y Windows Azure, comparando las herramientas y servicios esenciales.
- Capítulo 7: Se plantea un caso de estudio de una aplicación SaaS para verter los conceptos teóricos sobre una plataforma como servicio concreta.
- Capítulo 8: Se realiza una conclusión final del trabajo y se mencionan posibles líneas futuras de investigación.

CAPÍTULO 1 CLOUD COMPUTING

En este capítulo se introducirán las bases del concepto de *Cloud Computing*, comenzando con los antecedentes tecnológicos que culminaron en el desarrollo de este nuevo modelo de computación. Luego se mencionan sus características esenciales, los diferentes modelos de servicio y de despliegue, finalizando con las ventajas y desventajas de Cloud Computing.

CLOUD COMPUTING

Cloud computing según el *National Institute of Standards and Technology [18],* es un *modelo de computación* que permite demandar a través de internet recursos compartidos como procesamiento en servidores, almacenamiento, aplicaciones y servicios, los cuales pueden ser rápidamente aprovisionados y liberados sin ningún esfuerzo de administración por parte del proveedor de éstos servicios.

Para entender el *modelo de computación* que ofrece *Cloud Computing*, es necesario dejar en claro a qué llamamos modelo de computación.
Un modelo de computación define dónde está la capacidad de cómputo y de qué forma se accede a ella[1].
En la siguiente sección se enumeran los diferentes modelos de computación que surgieron a lo largo de la historia.

ANTECEDENTES TECNOLÓGICOS

Modelo de computación Cliente/Servidor

Todo el poder de cómputo, todo el software de aplicación, toda la información y todo el control residía en súper computadoras llamadas *mainframe* o comúnmente llamados servers. Los clientes eran terminales bobas, ya que no tenían poder de cómputo ni memoria, y solo se remitían enviar órdenes al servidor.

Modelo de computación Par a Par
El modelo cliente/servidor tenía sus falencias, la principal que se convertía en un cuello de botellas, puesto que toda la comunicación debía realizarse con el server.
El modelo P2P define una arquitectura de red en donde la capacidad de cómputo está repartida y cada computador tiene capacidad y responsabilidad equivalente. P2P habilita directamente el intercambio de recursos y servicios, no hay necesidad de un servidor central.

[1] A menudo se confunde el modelo de computación con el estilo arquitectónico de la aplicación. Según [17] un estilo de arquitectura, también llamado un patrón de arquitectura, es un conjunto de principios, que dan un marco de trabajo para diseñar una aplicación o sistema. Un estilo arquitectónico se monta sobre un modelo de computación.

Modelo de computación distribuida en grid

La computación distribuida en grid es una tecnología que permite utilizar de forma coordinada todo tipo de recursos (entre ellos cómputo, almacenamiento y aplicaciones específicas) que no están sujetos a un control centralizado. En este sentido es una nueva forma de computación distribuida, en la cual los recursos pueden ser heterogéneos (diferentes arquitecturas, supercomputadores, clusters...) y se encuentran conectados mediante redes de área extensa (por ejemplo Internet).

Modelo de computación Cloud Computing

La capacidad de cómputo reside en una red masiva de servidores interconectados entre sí en forma de grilla, corriendo en paralelo y combinando los recursos para ofrecerlos de una forma uniforme. Esta colección de servidores es accesible a través de internet. El hardware típicamente es propietario y puede estar alojado en uno o más datacenters.

FUNDAMENTOS DE CLOUD COMPUTING

La diferente bibliografía consultada sobre el tema, coinciden en describir a Cloud Computing o Computación en la Nube como un conjunto de características esenciales, que se apoya en ciertos modelos de despliegue y que ofrece diferentes modelos de servicio. Las siguientes tres secciones explayan estos conceptos.

CARACTERÍSTICAS ESENCIALES

Las características esenciales son aquellas que las distinguen de los modelos de computación que la precedieron, según [5]:

> *Autoservicio a demanda*: Un consumidor puede unilateralmente aprovisionarse de recursos computacionales, como tiempo de servidor y almacenamiento, de forma automática y sin la necesidad de interacción humana.
>
> *Amplio acceso a la red*: Los servicios deben ser accesibles mediante internet y utilizando mecanismos de comunicación estándar que permitan el acceso heterogéneo de dispositivos como teléfonos móviles, laptops, PDAs, etc.
>
> *Poolling de recursos*: Los recursos de cómputo del proveedor, como almacenamiento, procesamiento, memoria, ancho de banda y máquinas virtuales deben ofrecerse como un pool capaz de servir a múltiples consumidores, estos

recursos físicos pueden virtualizarse [2], y ser asignados o reasignados dinámicamente según la demanda. La ubicación física de los recursos asignados (datacenter, estado, país, etc) debe hacerse de un modo transparente.

Elasticidad: Los recursos y servicios deben ser aprovisionados rápida y elásticamente, en algunos casos de forma automática, para permitir a las aplicaciones aumentar rápidamente su capacidad de cómputo o disminuirla, según variables que pueden ser desde determinados horarios de ciertos días o cuándo se detecta que la capacidad de cómputo actual está al máximo y se necesita más.

Medición: El uso de los recursos debe ser monitoreado, medido, controlado y reportado de una manera conveniente, de forma tal de proveer la transparencia tanto para el proveedor como para el consumidor.

MODELOS DE SERVICIO

El modelo de servicio de Cloud Computing está relacionado directamente con el modelo de negocio de Cloud Computing, en donde la característica esencial es la forma de pago: "El consumidor solo paga por lo que usa"

Debido a esta característica, el consumidor no compra un producto, sino que renta un servicio por un tiempo determinado y solo paga por el uso que le dé al mismo dentro del tiempo estipulado. En concordancia con esta definición, en la diversa literatura sobre Cloud Computing emergió el término "X as a Service", donde X representa el tipo de servicio que se ofrece.
La mayoría de los autores presentan el modelo de servicio como una pila de servicios, como en la Figura 1-1, orientados para diferente público.

[2]Virtualización es la capacidad de emular uno o más servidores dentro de un único computador físico. Esto permite a una sola computadora, tomar el rol de múltiples computadoras, aprovechando recursos que de otra manera no podrían ser utilizados.

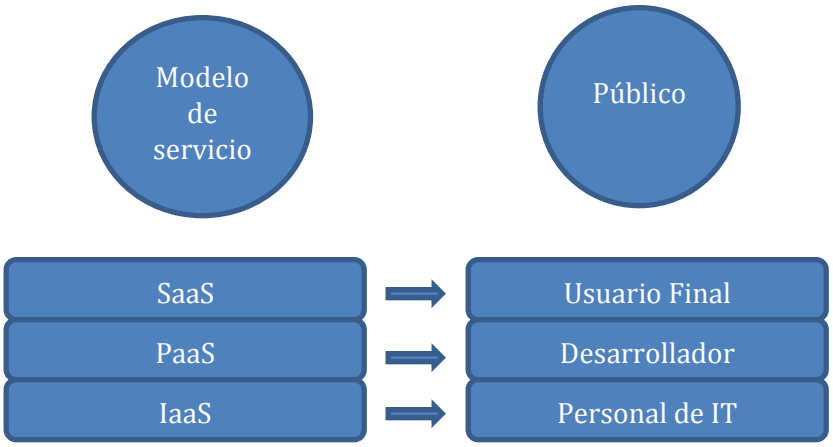

Figura 1-1 – Modelos de servicio según el público

IaaS - Infraestructura como servicio

Provee servicios de infraestructura IT como máquinas virtuales, almacenamiento, recursos de red, procesamiento, sistemas operativos, orientados al personal de IT.
Es el servicio más básico del modelo de servicio de cloud computing. Ofrece recursos de cómputo como CPU, memoria, almacenamiento mediante máquinas virtuales, las cuales son creadas utilizando un administrador de máquinas virtuales. Este administrador es conocido por el término *hypervisor*, y es un software especial dedicado para emular máquinas. Ejemplos de hypervisor son Xen y KVM.
Un proveedor de IaaS tiene en su datacenter un conjunto de hypervisors que trabajan unidos, que permiten ofrecerle al usuario la capacidad de obtener recursos de forma casi ilimitada.
Los proveedores más conocidos de IaaS son Amazon Web Services, con el servicio E2C, Windows Azure Virtual Machines, DynDNS, Google Compute Engine, HP Cloud, Rackspace Cloud.

PaaS – Plataforma como servicio

Este servicio ofrece herramientas de software orientado a desarrolladores de aplicaciones. Permite abstraer al desarrollador de las particularidades de la infraestructura proveyendo una capa de más alto nivel para administrarla.
Desde el punto de vista del desarrollador de aplicaciones, la plataforma funciona como un *middleware* entre su aplicación y la infraestructura cloud del proveedor. Este middleware es un conjunto de aplicaciones de software, portales web, APIs, etc., cuyo objetivo es permitir al desarrollador u organización la gestión integral de la plataforma.

SaaS – Software como servicio
Es el servicio de más alto nivel, ofrece aplicaciones para el usuario final. Según la definición del NIST, *SaaS es el servicio que permite al usuario utilizar aplicaciones que están corriendo en una infraestructura cloud. Estas aplicaciones son accesibles desde diferentes dispositivos a través de un browser. El consumidor de este servicio no administra ni tiene control sobre la infraestructura cloud subyacente, como ser la red, los servidores, sistemas operativos, almacenamiento, ni siquiera sobre las características técnicas de la aplicación.*

MODELOS DE DESPLIEGUE

El modelo de despliegue de *Cloud Computing* está relacionado con quién es el propietario de la nube y quién tiene permiso para accederla.

El modelo de despliegue según [5] se divide en 4 tipos de proveedores de servicios de computación en la nube.

Nube privada: La infraestructura de la nube es operada únicamente por una organización. Puede estar alojada en las mismas instalaciones de la organización o fuera de la misma.

Nube pública: La infraestructura de la nube está disponible para el público en general y es propiedad de una organización que vende los servicios de cloud.

Nube comunitaria: La infraestructura de la nube esta compartida entre varias organizaciones y da soporte a una comunidad específica que comparte ciertos cometidos como misión, requerimientos de seguridad, políticas, etc. Puede ser administrada por las mismas organizaciones o por un tercero, y puede estar alojada en las mismas instalaciones de las organizaciones o fuera de las mismas.

Nube híbrida: La infraestructura de la nube está compuesta por dos o más nubes (privadas, comunitarias o públicas) que permanecen como entidades únicas, pero que las une algún mecanismo de estandarización para facilitar la portabilidad de aplicaciones.

VENTAJAS Y DESVENTAJAS DE CLOUD COMPUTING

El modelo de computación que provee Cloud Computing se aplica muy bien para aplicaciones basadas en Internet.
Según Wikipedia y [16], a continuación se muestra un compendio de ventajas y desventajas de Cloud Computing que ayudarán a discernir si el modelo de computación se ajusta a las necesidades de la aplicación a desarrollar.

Ventajas

- Integración probada de servicios Red. Por su naturaleza, la tecnología de cloud computing se puede integrar con mucha mayor facilidad y rapidez con el resto de las aplicaciones empresariales (tanto software tradicional como Cloud Computing basado en infraestructuras), ya sean desarrolladas de manera interna o externa.

- Prestación de servicios a nivel mundial. Las infraestructuras de cloud computing proporcionan mayor capacidad de adaptación, recuperación completa de pérdida de datos (con copias de seguridad) y reducción al mínimo de los tiempos de inactividad.

- Una infraestructura 100% de cloud computing permite al proveedor de contenidos o servicios en la nube prescindir de instalar cualquier tipo de hardware, ya que éste es provisto por el proveedor de la infraestructura o la plataforma en la nube. Un gran beneficio del cloud computing es la simplicidad y el hecho de que requiera mucha menor inversión para empezar a trabajar.

- Implementación más rápida y con menos riesgos, ya que se comienza a trabajar más rápido y no es necesaria una gran inversión. Las aplicaciones del cloud computing suelen estar disponibles en cuestión de días u horas en lugar de semanas o meses, incluso con un nivel considerable de personalización o integración.

- Actualizaciones automáticas que no afectan negativamente a los recursos de TI. Al actualizar a la última versión de las aplicaciones, el usuario se ve obligado a dedicar tiempo y recursos para volver a personalizar e integrar la aplicación. Con el cloud computing no hay que decidir entre actualizar y conservar el trabajo, dado que esas personalizaciones e integraciones se conservan automáticamente durante la actualización.

- Contribuye al uso eficiente de la energía. En este caso, a la energía requerida para el funcionamiento de la infraestructura. En los datacenters tradicionales, los servidores consumen mucha más energía de la requerida realmente. En cambio, en las nubes, la energía consumida es sólo la necesaria, reduciendo notablemente el desperdicio.

Desventajas

- La centralización de las aplicaciones y el almacenamiento de los datos origina una dependencia de los proveedores de servicios.
- La disponibilidad de las aplicaciones está ligada a la disponibilidad de acceso a Internet.

- Los datos "sensibles" del negocio no residen en las instalaciones de las empresas, lo que podría generar un contexto de alta vulnerabilidad para la sustracción o robo de información.
- La confiabilidad de los servicios depende de la "salud" tecnológica y financiera de los proveedores de servicios en nube. Empresas emergentes o alianzas entre empresas podrían crear un ambiente propicio para el monopolio y el crecimiento exagerado en los servicios.
- La disponibilidad de servicios altamente especializados podría tardar meses o incluso años para que sean factibles de ser desplegados en la red.
- La madurez funcional de las aplicaciones hace que continuamente estén modificando sus interfaces, por lo cual la curva de aprendizaje en empresas de orientación no tecnológica tenga unas pendientes significativas, así como su consumo automático por aplicaciones.
- Seguridad. La información de la empresa debe recorrer diferentes nodos para llegar a su destino, cada uno de ellos (y sus canales) son un foco de inseguridad. Si se utilizan protocolos seguros, HTTPS por ejemplo, la velocidad total disminuye debido a la sobrecarga que éstos requieren.
- Escalabilidad a largo plazo. A medida que más usuarios empiecen a compartir la infraestructura de la nube, la sobrecarga en los servidores de los proveedores aumentará, si la empresa no posee un esquema de crecimiento óptimo puede llevar a degradaciones en el servicio.

CONCLUSIÓN

Cloud Computing es un modelo de computación basado fuertemente en el modelo de servicio. El modelo de servicio divide a las organizaciones en proveedores y clientes. Los proveedores desarrollan los servicios que las organizaciones clientes consumirán. Para las organizaciones proveedoras, las características esenciales de *Cloud Computing* que lo diferencian de los modelos de computación anteriores, suponen una clara ventaja para desarrollar aplicaciones basadas en internet que sean escalables, elásticas y de bajo costo.

Para las organizaciones clientes, el concepto de “solo se paga lo que se usa” les permite administrar mejor el presupuesto IT. El presente trabajo estará orientado a los desafíos técnicos y de diseño que enfrentará un proveedor de servicios para construir una solución de software con el fin de satisfacer a organizaciones clientes consumidoras de servicios. Para que un proveedor de servicios pueda construir una solución de software ofrecida como servicio, será necesario tener muy en mente qué implica esto. Por tanto, el siguiente capítulo hará un análisis de las características principales de una solución ofrecida como “Software como Servicio”.

CAPÍTULO 2 SOFTWARE COMO SERVICIO

Para desarrollar una solución de software como servicio, se debe tener en claro qué implica ofrecer un software como servicio, y abarca desde el modelo de negocio hasta las técnicas esenciales. En este capítulo se hará hincapié en las características de una solución SaaS. Se mostrarán las ventajas y desventajas comparativas de un software ofrecido como producto y un software ofrecido como servicio. Se enumerarán los diferentes modelos de maduración de las soluciones SaaS, mostrando las complejidades técnicas que acarrean.

CARACTERÍSTICAS DE UNA SOLUCIÓN SAAS

Las principales características del modelo según IDC[3] son:

- El software es accesible, manejado y comercializado vía red.
- El mantenimiento y actividades relacionadas con el software se realizan desde un lugar centralizado en lugar de hacerlo en cada cliente, permitiendo a estos acceder a las aplicaciones vía la red.
- La aplicación es distribuida típicamente bajo el modelo de uno-a-muchos, incluyendo su arquitectura, management, precio y partnering.
- Generalmente se basa en un modelo de comercialización en el cual no hay un costo inicial, sino un pago por suscripción o por utilización, en el cual no se diferencia la licencia del software del alojamiento del mismo.

Estas cuatro características son fundamentales para diferenciarlas de otros modelos que ofrecen software vía red, como el modelo *Software más Servicios (S+S)* y el modelo Application Service Providers (ASP).

Según [6] el modelo de *Software más Servicios*, utiliza servicios alojados en un servidor remoto, pero la aplicación principal se instala en el ambiente local del usuario. Este modelo no aprovecha completamente la capacidad de cómputo del servidor remoto y obliga al cliente a administrar y reservar los recursos necesarios para instalar la aplicación cliente en sus instalaciones.

Según [6], el modelo ASP comparte con SaaS la característica principal de tercerizar el alojamiento del software en un servidor externo y el acceso utilizando internet, aunque hay algunas diferencias. La principal diferencia está en la escalabilidad y en el

[3] IDC's Worldwide Software as a Service Taxonomy

aprovechamiento de los recursos. El modelo ASP fue diseñado para alojar aplicaciones que utilicen recursos dedicados, por lo que, al aumentar la cantidad de clientes y sus requerimientos de hardware, el proveedor estaba obligado a reservar los recursos físicos, aunque el cliente no los use todos al mismo tiempo ni utilice toda su capacidad. Esto muchas veces se hacía inviable para el cliente, ya que debía contratar recursos por demás sin saber si los iba a utilizar, con el consiguiente encarecimiento, y para el proveedor, que necesitaba poseer una gran cantidad de hardware dedicado. Otro problema para el cliente es que se debe encargar de la administración de los recursos físicos y de software alojados en el proveedor ASP, lo que incrementa los costos, por lo que muchos terminan optando por instalar el software en sus propios servidores. En cambio, como vimos el modelo SaaS fue diseñado específicamente para aprovechar los recursos al máximo, y además, el proveedor se encarga de la administración de los recursos físicos y de software.

En la Tabla 2-1, se muestra un resumen de las diferencias entre los diferentes modelos

Tabla 2-1 - Comparativa entre saas, s+s y asp

	SaaS	**S+S**	**ASP**
Software accesible vía Red	Si	Si	Si
Administración centralizada	Si	No, ya que hay Soft en el Cliente.	Si
Modelo uno a muchos	Si	No	No
Costo Inicial	Muy Bajo	Alto. Cada cliente alojar el software cliente.	Alto. Cada cliente paga su infraestructura.

Dado que un *Software como Servicio* es utilizado y comercializado vía red, sin la necesidad de que el cliente tenga que instalarse nada, este modelo difiere completamente del modelo tradicional de software ofrecido como un producto instalable en el cliente. En el modelo de *Software como Producto (SaaP)*, los proveedores desarrollan el software y lo empaquetan para ser distribuido bajo un nombre y una versión determinada, en un soporte físico digital como un CD o DVD, o para ser descargado desde Internet. El producto está preparado para ser alojado en las instalaciones informáticas del cliente, y viene asociado con un contrato de licencia de uso que define bajo qué condiciones se puede instalar y utilizar.

Para el proveedor, desarrollar una solución SaaS en lugar de un producto, le trae varias ventajas, por ejemplo:

- Le permite tener un mejor control y administración del software, dado que el mismo reside en sus instalaciones de forma completa, y no esta diseminado en

todos los clientes. Esto le permite al proveedor modificar, actualizar y corregir lo que sea necesario sin la necesidad de liberar versiones de forma periódica o parches.

- El proveedor solo debe codificar el software para una única plataforma, y al ser accedido vía internet, llegar a plataformas heterogéneas.

- Dado que la aplicación está alojada en la infraestructura del proveedor, le permite tener un feedback rápido de los clientes y de los *bugs* que se producen en la aplicación, en lugar de esperar reportes de los clientes en forma esporádica.

- Con un producto, es el cliente el que decide siempre migrar de una versión a la otra, y el proveedor tiene la obligación de ofrecer mantenimiento y soporte para versiones anteriores. En cambio, controlar y administrar un software alojado en servidores controlados por el proveedor, supone una mayor flexibilidad para aplicar actualizaciones y correcciones que fácilmente sean accedidas por todos los clientes.

Para el cliente que consume un Software como Servicio en lugar de un producto, también le trae varias ventajas, por ejemplo:

- Tener un modelo de pago flexible y a un menor costo: El mayor costo para una empresa que necesita de software está en las licencias y en el hardware que necesita para correr dicho software. Normalmente las licencias son por un período de tiempo determinado y para cierta cantidad de usuarios o máquinas. Esta inversión inicial es costosa, ya que la mayoría de los vendedores de software otorgan licencias de un año o más. En cambio, las aplicaciones SaaS no se licencian, sino que el cliente se suscribe al servicio por un tiempo determinado, de meses a años, pudiendo dejar de utilizar el servicio cuando lo desee, ya que en general no hay contratos de permanencia, dándole al cliente una gran libertad económica.

- Tener siempre la última versión del servicio: el cliente tiene la garantía de que siempre utilizará la última versión de la solución, con las actualizaciones de seguridad, correcciones y mejoras correspondientes, sin la necesidad de esperar al lanzamiento de una versión estable, y sin la necesidad de hacer migraciones de una versión a la otra.

En la Tabla 2-2, se muestra un resumen de la diferencia entre un software ofrecido como producto y un software ofrecido como servicio.

Tabla 2-2 - Comparativa entre SaaS y SaaP

	Software como producto	**Software como servicio**
Delivery	Diseñado para que el	Diseñado para ser utilizado

	cliente lo instale, administre y mantenga	desde internet
Desarrollo	Ciclo de desarrollo largo, múltiples codificaciones para la misma aplicación	Diseñado para correr miles de clientes sobre un único código fuente.
Política de precios	Licencia + Mantenimiento	Suscripción (Todo Incluido)
Costos adicionales	Instalación, mantenimiento, personalización, actualizaciones	Solo para características adicionales
Plataforma	Múltiples versiones para diferentes plataformas	Una única versión para diferentes plataformas
Bugs	Mecanismos de actualización lentos y con diferentes tiempos para cada cliente	Se arregla un problema para un cliente, y se arregla para todos
Demo de la aplicación	Requiere instalación	Online, no requiere instalación
Actualizaciones	Lentas, dependen del cliente.	Rápidas, las controla el proveedor
Feedback del cliente	Lento y dependen del cliente el envío de las mismas	Rápido y online, se pueden monitorear estadísticas de uso.

Las características principales de SaaS, que están más bien relacionadas con el modelo de negocio, y la diferencia con otras alternativas a SaaS como SaaP, ASP y S+S, se deben implementar siguiendo ciertos requisitos técnicos. En la siguiente sección se explayan éstos requisitos técnicos que se necesitan para construir una solución SaaS.

REQUISITOS TÉCNICOS DE LAS APLICACIONES SAAS

Para el proveedor, construir una solución SaaS, implica considerar ciertas requisitos técnicos, que ayudarán a cumplir las características principales de una solución SaaS. Estos requisitos técnicos son esenciales, y deben ser tenidos muy en cuenta al momento de diseñar y construir una solución SaaS. A continuación se enumeran dichos requisitos.

La aplicación debe ser multi-tenant

El concepto de multi-tenant o multi-cliente refiere a un principio de arquitectura de software en donde una única instancia de un producto de software corre en un servidor, atendiendo a múltiples organizaciones o clientes. Cada cliente u organización dentro de la misma instancia de la aplicación tiene su propio ambiente o partición, es decir, puede personalizar la aplicación definiendo sus propios usuarios, mecanismos de seguridad, parámetros y configuraciones visuales sin interferir a las otras organizaciones y de forma

totalmente transparente. El objetivo primario de la arquitectura multi-tenant es maximizar recursos de hardware y de software.

La arquitectura multi-tenant se opone de alguna forma a la arquitectura multi instancia, en donde para cada cliente u organización se necesita instalar una nueva instancia de la aplicación con sus respectivos recursos de hardware y software dedicados para cada organización.

El middleware de una plataforma como servicio es esencialmente multi-tenant, ya que un desarrollador, o una organización con muchos desarrolladores, utiliza el middleware de la plataforma de forma aislada y transparente de otras organizaciones o desarrolladores.

Por otro lado, las aplicaciones alojadas en una plataforma como servicio pueden ser o no multi-tenant. Para que una aplicación sea multi-tenant es necesario diseñarla para tal fin, y además la plataforma como servicio debe brindar el soporte específico para que se puedan construir sobre ella aplicaciones multi-tenant.

La aplicación debe ser escalable y proveer mecanismos de balaceo de carga

Las aplicaciones SaaS deben estar preparadas para soportar una gran cantidad de clientes. En aplicaciones single-tenant la escalabilidad se logra instalando un nuevo servidor web con la misma aplicación y balanceando la carga. Eso para soluciones multi-tenant es una solución de grano grueso, por lo que es necesario definir mecanismos de escalabilidad que contemplen el uso de recursos por tenant y la posibilidad de asignar recursos de grano fino.

La aplicación debe ser personalizable y configurable

Cada cliente que se suscribe al servicio, utiliza la aplicación como si fuera el único cliente de la misma, por lo que la aplicación necesita diseñarse de tal forma que permita a cada cliente personalizarla según sus necesidades sin interferir a los otros clientes.

La aplicación debe proveer mecanismos de suscripción, monitoreo, monetización y facturación

El modelo de pago por uso implica que en la aplicación se tenga que diseñar específicamente los mecanismos de suscripción, monetización y facturación, por lo que se necesita monitorear constantemente el uso que cada tenant le da a la misma. Es prioritario ofrecer al cliente una buena variedad de precios, y que el cliente sepa exactamente qué se le está cobrando.

La aplicación debe dar soporte para el aprovisionamiento de recursos

Debido al modelo de suscripción dinámico de las aplicaciones SaaS deben estar preparadas para reservar y dedicar recursos a cada nuevo suscriptor. Además cuanto la cantidad de suscriptores es muy grande se hace inviable que las tareas de aprovisionamiento se realicen de forma manual, y es necesario tener un proceso automático.

Estos requisitos técnicos son de suma importancia para construir una verdadera solución SaaS, y son los que guiarán el presente trabajo a lo largo de los siguientes capítulos. Estos requisitos técnicos derivaron de una constante evolución en la forma de construir soluciones SaaS. En la siguiente sección se mencionarán los diferentes modelos de madurez de una solución SaaS de acuerdo a los requisitos técnicos que adopte.

CONCLUSIÓN

En este capítulo se definió de forma precisa el modelo de servicio SaaS, diferenciándolo de otros modelos como S+S y ASP, y del modelo Software como Producto. Se enumeraron los requisitos técnicos que involucran la construcción de una solución SaaS. Estos requisitos técnicos son fundamentales, ya que de acuerdo a qué requisitos se implementen, definen el modelo de madurez de la solución SaaS. En el siguiente capítulo se ahondará en éstos requisitos técnicos, que guiarán el presente trabajo, manteniendo una trazabilidad y homogeneidad temática con los subsiguientes capítulos.

CAPÍTULO 3 REQUISITOS TÉCNICOS DE LAS APLICACIONES SAAS

El objetivo de este capítulo es explorar en detalle los requisitos técnicos de una aplicación SaaS, de forma de evidenciar las cuestiones de diseño y arquitectura que se deben tener en cuenta a la hora de construir una aplicación de este tipo.

MULTI TENANT

Según [15] una aplicación *multi tenant* permite a los clientes (tenants) compartir los mismos recursos de hardware, ofreciéndoles una instancia de aplicación y base de datos compartida, permitiendo a cada cliente configurar la aplicación según sus necesidades y correrla como si estuviera en un ambiente dedicado y aislado de los demás clientes.

También según [15] un tenant o cliente es la entidad organizacional que renta una solución SaaS multi-tenant. Típicamente un tenant agrupa a un conjunto de usuarios, que son los que utilizan la aplicación dentro de la organización.

Desde el punto de vista del desarrollo de software, multi-tenant es un principio de arquitectura cuyo objetivo es brindar lineamientos de diseño para que la solución SaaS sea escalable, personalizable y maximice la utilización de los recursos.

Según [8] existen diferentes niveles de uso compartido en las aplicaciones

Aplicaciones Multi-usuario

La definición de multi-usuario corresponde a los sistemas que permiten que múltiples usuarios utilicen concurrentemente una misma instancia de aplicación. Aquí el usuario es la persona física que accede al sistema, con un usuario y contraseña, pertenezca a una organización o no. Vale aclarar que algunas aplicaciones que se clasifican como SaaS no necesariamente tienen una arquitectura multi-tenant. Por ejemplo, los servicios de mail como Hotmail y Yahoo Mail, son aplicaciones SaaS multi-usuario, pero estos usuarios son usuarios individuales para estos servicios, ya que no están asociados a ninguna organización.

Aplicaciones Multi-instancia

Las aplicaciones multi-instancia son aquellas que se instalan en un servidor de aplicaciones o servidor web. Por lo general estos servidores permiten alojar diferentes aplicaciones y diferentes instancias de la misma aplicación. Para el cliente, esto es transparente, y puede simular una aplicación multi-tenant, pero la

realidad es que si bien se comparten recursos, la arquitectura multi-tenant aprovecha muchísimo mejor el uso de un servidor.

Multi-tenant vs Single-tenant

Como grafica la Figura 3-1, las aplicaciones multi-tenant y single-tenant pueden estar basadas en sistemas multi-usuario y multi-instancia, pero la principal diferencia es que las aplicaciones single-tenant están preparadas para atender a una sola organización o a usuarios individuales sin relación con alguna organización. Si se decide ofrecer una solución single-tenant a otra organización, por lo general es necesario que el proveedor instale un nuevo servidor y una nueva base de datos, de forma de aislar la aplicación y los datos del otro cliente, y que no interfiera la actividad de uno con el otro.

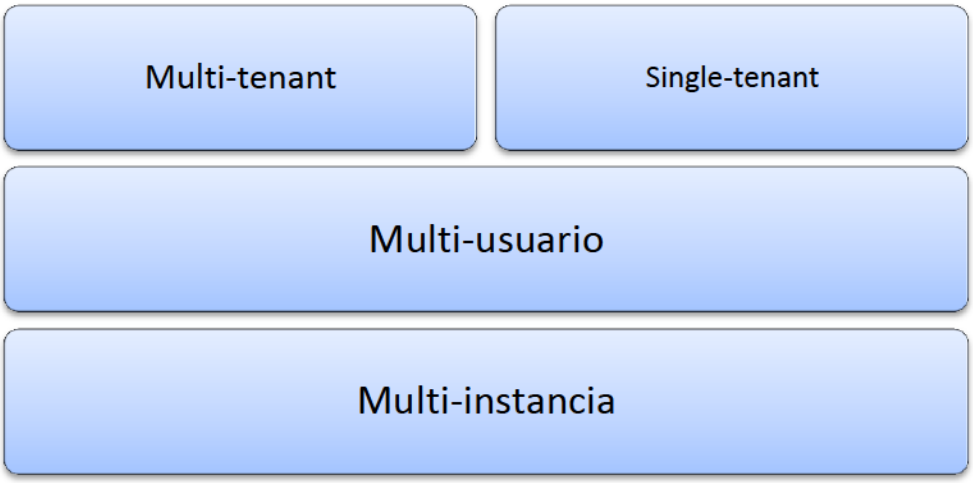

Figura 3-1- Relación entre multi y single tenant, multi-usuario y multi-instancia

Por otro lado, las aplicaciones multi-tenant representan el máximo nivel de uso compartido de una aplicación. Un sistema multi-tenant puede implementarse con una única instancia de aplicación o con múltiples instancias. La cantidad de instancias no está relacionada directamente con la cantidad de tenants. Todos los tenants pueden correr en una única instancia, o un grupo de tenants pueden correr en una instancia y otro grupo de tenants puede correr en otra, o bien, un tenant puede tener asignada más de una instancia de aplicación. La capacidad de adecuar la cantidad de instancias de la aplicación según el uso y la cantidad de tenants es una característica importante de las aplicaciones SaaS.

GRADOS DE MULTI-TENANT

El objetivo de una arquitectura multi-tenant es hacer que el sistema maximice la utilización de recursos, y la mejor forma de hacerlo es compartiendo la misma solución SaaS entre los diferentes tenants. Ahora bien, qué cosas se pueden compartir en una solución SaaS.

Desde un punto de vista de alto nivel, una solución SaaS está compuesta por la aplicación ejecutable y por la base de datos. Así podemos encontrar dos combinaciones:

Aplicación multi-tenant, base de datos single-tenant

Esta opción asume que los tenant solo se van a compartirla aplicación y cada tenant tendrá su propia base de datos. Ahora bien, compartir la aplicación para diferentes tenants tiene implicancias de diseño muy importante, ya que cada tenant necesitará sus propias configuraciones, reglas de negocio, diferentes mecanismos de seguridad, y por supuesto, que a cada tenant se lo puede aplicar una tarifa diferente según su configuración, por lo que es necesario diseñar la aplicación correctamente para mantener aislado estas cuestiones de los diferentes tenants. El impacto en el diseño es muy importante, por ejemplo, una aplicación cuya lógica de negocio se implemente con procedimientos almacenados en la base de datos, tendrá que codificar diferentes lógicas según el tenant, con el consiguiente costo adicional de mantenimiento.

Aplicación multi-tenant, base de datos multi-tenant

Esta opción utiliza la misma aplicación y la misma base de datos para todos los tenants. Con esta opción es necesario diseñar cuidadosamente no solo la aplicación sino también la base de datos para que soporte diferentes tenants. Para diseñar una base de datos compartida por tenants, existen diferentes alternativas, como replicar las mismas tablas con nombres diferentes para cada tenant, o conservar las mismas tablas, y agregar un campo adicional a cada tabla para identificar los registros de cada tenant. Usando el mismo ejemplo anterior, si la lógica de la aplicación se encuentra en la base de datos, será mucho más costoso codificar en los mismos procedimientos las diferentes lógicas de los tenants.

La elección de una u otra alternativa, dependerá de la ecuación costo-beneficio del proveedor que desarrolle la solución. En los siguientes capítulos se darán lineamientos y consejos para tomar la mejor decisión.

El desarrollo de aplicación multi-tenant debe enfocarse en dos aspectos muy importantes:

En mantener un único código fuente base

> Para el vendedor de una solución SaaS mantener un único código base es mucho más simple y barato, lo que redunda en actualizaciones más rápidas para el cliente, y a menor costo.
> Mantener un único código base para toda la aplicación y para todos los tenants conlleva una importante tarea de planificación, diseño y desarrollo, ya que hay que pensar que el mismo código se debe adaptar a las necesidades de cada tenant, a esta variabilidad del código en la jerga se la denomina *configurabilidad*.
> A menudo, a este tipo de aplicaciones se las denomina *“polimórficas”*.

En [7] también se menciona el concepto de versionamiento. Dado que es menester mantener un único código base, es necesario que el software pueda correr concurrentemente diferentes versiones de los mismos servicios.
Este tipo de circunstancias ocurre cuando se necesita mantener compatibilidad hacia atrás del software, cuando un tenant por alguna razón desea conservar cierta versión y no quiere actualizarse, o cuando en cierto país existe una legislación que obligue al software a operar de forma diferente.

En mantener el aislamiento entre los diferentes tenants

Al utilizar un ambiente compartido, es necesario siempre asegurar que cada tenant acceda a su ambiente correcto, asegurando que la actividad de un tenant no interfiera con la de otro tenant.
Según [7] el concepto de aislamiento alude a que la actividad de un tenant no debe interrumpir ni interferir la actividad de otro tenant, a lo que propone dos tipos de aislamiento:

Aislamiento de datos: dado que en una aplicación multi-tenant todos los tenants utilizan la misma instancia de aplicación y base de datos, es necesario implementar mecanismos para que un tenant no acceda a datos de otro tenant. A lo que propone diferentes soluciones, como agregar una capa más entre la capa de negocio y la capa de datos, o utilizar diferentes niveles de permisos a nivel del DBMS.

Aislamiento de ejecución: en una aplicación multi-tenant, cada tenant puede operar con una lógica de negocio diferente, según su configuración, por lo que es necesario asegurar que cada uno corra con su propia configuración.

En [23] también se menciona el aislamiento de rendimiento, ya que la falta de garantías del mismo es uno de los mayores obstáculos para los potenciales usuarios de soluciones SaaS.
Los problemas de rendimiento por lo general se deben a una carga de trabajo excesiva de unos pocos tenants, en perjuicio de los otros tenants. Según [23] existe el aislamiento de rendimiento si se cumplen para cada tenant el acuerdo de nivel de servicio asignado a cada uno, independientemente de si algún tenant se excede en la utilización de los recursos. El acuerdo de nivel de servicio especifica las garantías que el proveedor ofrece a sus clientes, entre las cuales puede estar la garantía de un umbral de rendimiento.
A su vez, [23] menciona el aislamiento de recursos, donde es posible asignar a algún tenant particular cierto cuota de CPU y memoria predefinida, de forma tal que este tenant no compita por estos recursos con otros tentants.

ESCALABILIDAD

El concepto de escalabilidad se podría definir como la capacidad de un sistema informático de cambiar de tamaño o configuración para adaptarse a las circunstancias

cambiantes, sin perder calidad y manejando el crecimiento o decrecimiento de forma continua, sin interrupciones del servicio y de forma homogénea.

La escalabilidad hacia arriba es cuando hay un mayor volumen de uso de la aplicación y se necesita agregar mayor poder de procesamiento, de memoria, de ancho de banda, etc., para seguir brindando el servicio en óptimas condiciones, siempre de forma continua y sin interrupciones.

La escalabilidad hacia abajo es cuando hay un volumen de uso menor de la aplicación y existe capacidad ociosa de los recursos, por lo que se necesita desafectar esos recursos de la solución SaaS, siempre de forma continua y sin interrupciones.

Debido a que un tenant se puede suscribir y dar de baja en cualquier momento, la solución SaaS tiene que estar prepara para escalar en recursos de hardware y software hacia arriba o hacia abajo de forma dinámica.

La escalabilidad de una solución SaaS se logra diseñando cuidadosamente la aplicación para tal fin, de forma que el servicio se brinde sin interrupciones.

Actualmente las plataformas de cloud ofrecen opciones de escalabilidad de grano fino, ya que en lugar de ofrecer instalar nuestra aplicación en un servidor dedicado, implementan una capa intermedia, donde la aplicación reside en una especie de contenedor. Aunque para cada plataforma este contenedor tenga nombres y especificaciones diferentes, el contenedor define el marco computacional de esa aplicación, como ser cantidad y tipo de procesadores, cantidad de memoria, cantidad de espacio de disco, etc. En este tipo de plataformas, la escalabilidad se logra aumentando la cantidad de contenedores.

Una característica importante es que las aplicaciones puedan escalar de forma automática e ininterrumpida. Para lograrlo la plataforma debe proveer métricas de utilización de los recursos en tiempo real, y mecanismos automáticos para incrementar o decrementar la cantidad de contenedores. Algunas plataformas permiten administrar estos parámetros desde su panel de administración, y otras proveen APIs para que el desarrollador implemente su propio algoritmo de escalabilidad. Para las aplicaciones multi-tenant será necesario tener siempre en mente que cualquier implementación de escalabilidad impactará en el diseño de la aplicación.

Las aplicaciones SaaS deben estar preparadas para un alto volumen de tráfico de usuarios, para ello podemos emplear la técnica de balancear la carga de trabajo de la aplicación. El balanceo de carga puede operar en tres niveles, a nivel de red, a nivel de servidor y a nivel de aplicación. En una aplicación web tradicional el balanceo de carga se hace a nivel de red o a nivel del servidor web. Por lo general se dispone una granja de servidores relacionados, el propio sistema operativo o servidor web se encarga de monitorear la carga de trabajo y redirigir las peticiones al servidor correspondiente. Ahora bien, en aplicaciones SaaS multi-tenant necesitamos balancear la carga a nivel de aplicación, ya que para cada tenant podemos tener diferentes políticas de balanceo de carga. Este concepto implica tener en cuenta que es necesario diseñar la aplicación para

soportar diferentes instancias de la misma en el mismo server, y diferentes instancias en diferentes servers, identificando siempre qué tenant está haciendo el requerimiento.

También es posible balancear la carga de trabajo de la base de datos. Este punto es importante ya que dependiendo de cómo hayamos implementado la técnica de multi-tenant, tendremos que tener en cuenta de qué tenant es el requerimiento para saber a qué base de datos dirigirlo.

CONFIGURABILIDAD

Según [20] la configurabilidad permite proveer a los clientes/tenants de una multitud de opciones y variaciones de la aplicación, usando un único código base, de tal forma que es posible que cada tenant tenga su propia configuración de software.

En una solución SaaS podemos tener *configurabilidad* a 2 niveles

> **A nivel de interface de usuario**: la configurabilidad de la interface de usuario implica poder adaptar el "look and feel" según las necesidades del usuario y del tenant en cuestión, tales como íconos, colores, fuentes, títulos, etc.
>
> **A nivel funcional**: la configurabilidad a nivel funcional implica que para cada tenant puede variar el comportamiento de la aplicación, donde ciertas funciones del sistema pueden tener una lógica de negocios distinta para diferentes tenants o hasta flujos de trabajo totalmente diferentes. Además, la variabilidad de cierta funcionalidad puede ocasionar que también varíe la interface.

Para ambos niveles de configurabilidad necesitaremos almacenar los datos de configuración para poder implementar esta característica en la aplicación. Los datos de configuración son el corazón de una aplicación SaaS y es necesario diseñar su almacenamiento y recupero con soporte para múltiples tenants.

APROVISIONAMIENTO

El aprovisionamiento es el proceso por el cual se reservan recursos de hardware y de software en la solución SaaS para un nuevo tenant o cliente.

Permitir a un nuevo tenant utilizar una aplicación SaaS puede involucrar, por ejemplo, importar cuentas de usuario y datos, aplicar configuraciones, crear tablas o nuevas bases de datos, asignar una cantidad fija de procesadores, habilitar permisos de ejecución y reglas de acceso del firewall, etc.

El aprovisionamiento puede ser manual o automático. Es manual cuando la cantidad de tenant es muy poca, y las tareas de aprovisionamiento las puede hacer el administrador

de la solución SaaS. En este caso, el cliente solicita o se suscribe al servicio SaaS, y espera un tiempo de 24 hs por ejemplo, hasta que el administrador aprovisiona los recursos necesarios para que el tenant este operativo en la solución SaaS.

Es automático, cuando la cantidad de tenants comienza a crecer y se hace imposible dedicar una persona a las tareas de aprovisionamiento. En este caso es necesario que la solución SaaS disponga de un módulo de software específico para tareas de aprovisionamiento automático, con la posibilidad de que el administrador monitoree las mismas.

El aprovisionamiento es una cuestión fundamental ya que necesita de un soporte específico en la PaaS elegida, en forma de APIs, para poder reservar y liberar recursos y monitorear la utilización de los mismos.

SUSCRIPCIÓN, MONETIZACIÓN Y FACTURACIÓN

Los mecanismos tradicionales de cobro son por cantidad de usuarios y por acceso a ciertos módulos de la aplicación, pero en el modelo SaaS se puede cobrar por tiempo de uso de la aplicación, por la cantidad y por el tiempo de los recursos de hardware que se desea utilizar, o modelos de cobro relacionados con el negocio concreto de la solución (monetización), como por ejemplo en el caso de un CRM, donde se puede relacionar el precio con el volumen de operaciones de venta. Introducir estos mecanismos de cobro en una solución de software implica que deben ser tenidos en cuenta en el diseño y en la codificación de la aplicación.

CONCLUSIÓN

En este capítulo se hizo hincapié en la problemática de diseño que implican los requisitos de diseño de una aplicación SaaS. Se explicaron cabalmente los conceptos de multi-tenant, escalabilidad, configurabilidad, aprovisionamiento y monetización, que distinguen a una solución SaaS de otro tipo de soluciones.

CAPÍTULO 4 TECNICAS DE DISEÑO Y PROGRAMACION PARA IMPLEMENTAR LA CONFIGURABILIDAD

Ciertos requisitos técnicos de las aplicaciones SaaS *multi-tenant*, como la *configurabilidad a nivel funcional* y la *configurabilidad nivel de UI*, necesitan implementarse con el soporte de técnicas modernas de diseño y programación. En este capítulo se analizaran diferentes técnicas de programación para llevar a cabo estos dos requisitos y para poder entender cuál se ajusta a las necesidades de la aplicación a desarrollar, y así implementar soluciones SaaS de forma óptima, evitando diseños difíciles de mantener.

CONFIGURABILIDAD A NIVEL FUNCIONAL

A continuación se presentan las diferentes técnicas de diseño y programación que facilitan la implementación del requisito técnico *Configurabilidad a Nivel Funcional*.

PROGRAMACION ORIENTADA A OBJETOS (OOP)

El paradigma OOP es uno de los pilares para construir aplicaciones SaaS *multi-tenant* altamente configurables, debido a sus principios de diseño de reusabilidad, modularidad, abstracción, encapsulamiento, ocultación de información, herencia y polimorfismo.
La mayoría de los lenguajes modernos que se utilizan para desarrollar aplicaciones web, ya sean de front-end o de back-end son orientados a objetos o tienen pleno soporte del paradigma. Así, lenguajes de back-end como java, c#, visual basic, ruby, pyton, php, y de front-end como javascript por nombrar algunos, son indicados para desarrollar soluciones SaaS multi-tenant. Características como uso de programación paralela y multithreading, binding dinámico y tipado dinámico de los lenguajes OOP permiten escribir software altamente adaptable y configurable.

ARQUITECTURA ORIENTADA A SERVICIOS (SOA)

SOA es ideal para diseñar sistemas débilmente acoplados, una característica muy importante si tenemos en cuenta cuestiones como la alta configurabilidad necesaria en una aplicación multi-tenant.
Una solución SOA, con soporte multi-tenant, trae aparejado particularidades técnicas en la definición de la arquitectura que no aparecen en aplicaciones single-tenant, las cuales serán necesarias resolver para implementar con éxito tal solución, como por ejemplo, el concepto de multi-tenant utilizando un ESB, todo lo concerniente a orquestación y motores de reglas de negocio, y las diferentes técnicas de integración de aplicaciones, mecanismos de instanciación de servicios, etc.

Si bien los principios de diseño de SOA, como bajo acoplamiento, composición, interoperabilidad y federación de servicios, son afines a los principios de diseño de una aplicación SaaS multi-tenant, ésta no necesariamente debe tener una arquitectura SOA, y

si bien es posible plantear la lógica de negocios del backend de la aplicación como una colección de servicios, no necesariamente significa que tengamos una arquitectura SOA.

En [12], [19] y [21] se analizan diferentes patrones que pueden ser utilizados para diseñar y desarrollar aplicaciones SaaS orientadas a servicios y a procesos BPEL.

CONTEXT ORIENTED PROGRAMMING

La programación orientada al contexto, COP su abreviatura en inglés, permite definir comportamiento variable dependiendo del contexto, lo cual lo hace atractivo para utilizarlo en el desarrollo de aplicaciones multi-tenant configurables.

Los conceptos principales de COP son:

Variaciones de comportamiento: las variaciones típicamente consisten en un nuevo o modificado comportamiento. Estos pueden ser expresados como definiciones parciales de módulos en el modelo de programación subyacente, como procedimientos o clases.

Layers o capas: los layers agrupan variaciones de comportamiento relacionadas. Las layers son las entidades principales, por lo que pueden ser explícitamente referenciadas desde el modelo de programación subyacente.

Activación: las layers pueden ser activadas o desactivadas dinámicamente en tiempo de ejecución. Por código se puede decidir habilitar o deshabilitar las layers basados en el contexto actual.

Context: Cualquier información que es accesible computacionalmente puede formar parte del contexto que activará las variaciones de comportamiento que dependen de él.

Alcance: el alcance dentro de las cuales las layers pueden ser activadas o desactivadas pueden ser controladas explícitamente.

Las layers son construidas de modo que reaccionen a la información contextual. Basado en la información disponible del contexto de ejecución actual, layers específicas pueden ser activadas o desactivadas. Los lenguajes COP o extensiones a lenguajes para soportar COP para expresar, activar y componer layers en tiempo de ejecución, pero es la aplicación la que determina cuál es la información contextual relevante.

Los mecanismos de abstracción existentes en los lenguajes orientados a objetos son suficientes para modelar el contexto.

En [27] se menciona cómo se puede utilizar COP para implementar la configurabilidad de una aplicación multi-tenant.

INVERSIÓN DE CONTROL E INYECCIÓN DE DEPENDENCIA

La inversión de control es un patrón de diseño de OOP, en el que una clase A no depende directamente de otra clase B, sino a través de un intermediario, cediendo o invirtiendo el control de la clase al intermediario, al contrario del enfoque tradicional, donde el código de la clase A, controla o define exactamente con qué otras clases necesita interactuar, definiendo una relación fuerte entre las mismas. Para esto, la clase A no utiliza las clases relacionadas directamente, sino que declara las variables utilizando interfaces o clases abstractas.
El intermediario es el encargado de instanciar, por ejemplo la clase B, e inyectar la dependencia en la clase A.
Estos patrones son ideales para implementar la característica de configurabilidad de una aplicación SaaS, ya que se pueden inyectar diferentes implementaciones de las clases, dependiendo del tenant en cuestión, permitiendo adaptar en tiempo de ejecución la lógica de negocios, la lógica de presentación, y muchas otras cuestiones, de acuerdo a cada tenant. Esta técnica permitirá escribir código limpio, orientado a objetos y componentizable, evitando el código “spaguetti” y un sinnúmero de sentencias “if” que ocasionaría las diferentes configuraciones de cada tenant.

ASPECT ORIENTED PROGRAMMING

AOP complementa a OOP añadiendo otro nivel de modularidad y de desacoplamiento, ya que encapsula comportamiento transversal a todas las capas de la aplicación, como ser validaciones sensibles al contexto, manejo de errores, logging, optimizaciones, etc. Con AOP podemos abstraernos de estas cuestiones, concentrando todos estos aspectos en diferentes unidades de código, y aplicándolas en tiempo de ejecución de acuerdo a cierto contexto de ejecución.
En [25] se menciona a AOP como una buena práctica, para construir aplicaciones SaaS. De este modo, AOP contribuye a mejorar la mantenibilidad, adaptabilidad y reutilización del código, lo que lo hace ideal para construir productos cuyas características se ofrecen bajo demanda. Así, AOP puede ayudar a mejorar las implementaciones de aplicaciones SaaS multi-tenant en cuestiones como la configurabilidad de la solución multi-tenant, monitoreo, seguridad y caching.

EL PATTERN PLUG-IN

Este pattern en [25] es mencionado como una buena práctica, cuya función es la de permitir agregar funcionalidad adicional una vez terminado y entregado el producto. Para ello es necesario preparar la aplicación y dejar codificado puntos de extensibilidad mediante interfaces, de forma que los componentes que respeten e implementen dichas

interfaces puedan ser agregados como plug-in a la aplicación, sin la necesidad de recompilarla, y hasta sin siquiera detener su ejecución. Este tipo de patrón es ideal para ofrecer diferentes funcionalidades para diferentes tenants, y activarlos en tiempo de ejecución de acuerdo al tenant que este accediendo a la aplicación en ese momento. También este patrón contribuye a hacer el código de la aplicación mantenible en el tiempo, ya que cada funcionalidad particular es separada en un componente diferente, y no forma parte del núcleo principal de la aplicación.

CONFIGURABILIDAD A NIVEL DE USUARIO

A continuación se presentan las diferentes técnicas de diseño y programación que facilitan la implementación del requisito técnico *Configurabilidad a Nivel de UI*.

SPIAR

SPIAR es el acrónimo de Single Page Internet Application aRchitecture, o en castellano Arquitectura de aplicación de Internet de una Sola Página. Este tipo de arquitectura se popularizó con la aparición de Ajax, permitiendo que las páginas html puedan hacer invocaciones asincrónicas de javascript sin refrescar la misma. Actualmente esta tendencia se acentuó y es muy común ver aplicaciones que solo están conformadas por una única página html, que se modifica dinámicamente utilizando javascript y haciendo invocaciones al servidor de forma asincrónica.

Si bien las aplicaciones que se construyen con este tipo de arquitectura tienen un mejor tiempo de respuesta, para construir aplicaciones multi-tenant con interfaces de usuario altamente configurables será necesario replantear y tener en cuenta algunas cuestiones, como no llevar lógica de presentación por tenant al browser, ya que puede involucrar un problema de seguridad importante, y para esto se necesitará una programación mucho más cuidadosa.

MVC

El patrón MVC es clave para implementar una UI altamente configurable. La mayoría de los frameworks que implementan el patrón MVC permiten definir en ejecución el mapeo entre la vista y el controlador y el mapeo entre una url y una vista. Con estas dos características, estos mapeos pueden ser almacenados en una base de datos, y diferentes vistas y controladores pueden ser invocados según el tenant que este accediendo a la aplicación, permitiendo que la UI se adapte al mismo y que la lógica de presentación para cada tenant quede del lado del servidor.

ARQUITECTURA DIRIGIDA POR METADATOS

Las características de una aplicación SaaS multi-tenant, como la configurabilidad pueden ser alcanzadas basando el producto en una arquitectura dirigida por metadatos.

Almacenando la disposición de la UI, las validaciones del cliente, las propiedades de los controles visuales, en un repositorio de metadatos, y luego interpretando estos metadatos en tiempo de ejecución hará que la aplicación tenga diferentes aspectos según las necesidades de cada tenant. La metadata puede ser usada para representar flujos de trabajo, actividades, políticas, lógica y datos. La metadata puede ser de negocio, técnica u operacional, y usualmente es almacenada en formato xml o en una base de datos. La principal ventaja de construir productos dirigidos por metadatos es la reusabilidad, la consistencia, la mantenibilidad y la alta adaptabilidad del software. Un ejemplo de aplicación enteramente basada en una arquitectura dirigida por metadatos la podemos encontrar en SalesForce.com, en [2] y en [14] se realiza una buena explicación sobre su arquitectura.

CONCLUSIÓN

Las técnicas de programación y de diseño facilitan la construcción de aplicaciones SaaS *multi-tenant*, facilitando la implementación de los requisitos de configurabilidad de una solución SaaS. En la Tabla 4-1 se resume lo analizado en el capítulo.

Tabla 4-1 - Relación de características técnicas de saas y técnicas de programación

Característica Técnica SaaS	**Técnica de diseño y programación**
Configurabilidad a nivel funcional	OOP, SOA, Context Oriented Programming, Inyección de Dependencia, AOP, Plugin Pattern
Configurabilidad a nivel de usuario	Uso de MVC, SPIAR, Arquitectura dirigida por Metadatos

En el siguiente capítulo se mostrarán diferentes implementaciones posibles que utilizan éstas técnicas de diseño y programación.

CAPÍTULO 5 IMPLEMENTACION DE LA CONFIGURABILIDAD

Al diseñar una solución web *multi-tenant*, la implementación de la configurabilidad involucra tres aspectos fundamentales. El primero es la identificación del tenant en la aplicación web, ya que dado que la misma instancia de la aplicación atenderá los requerimientos http de todos los clientes, es necesario implementar algún mecanismo para diferenciar los requerimientos de un tenant y de otro. El segundo aspecto es como diseñar y codificar la aplicación para soportar la configurabilidad tanto a nivel funcional como a nivel de UI. El tercer aspecto es cómo diseñar la base de datos de la aplicación para que soporte múltiples clientes, con diferentes datos y configuraciones. En este capítulo se abordarán éstos tres aspectos, y se ofrecerán diferentes enfoques para implementarlos.

IDENTIFICACIÓN DEL TENANT EN LA APLICACIÓN

Lo primero que hay que especificar para diseñar una aplicación web *multi-tenant* es cómo se van a identificar los tenant, y cómo se mantendrá esa información a lo largo de la ejecución de una determinada acción. Este es el punto de partida, y la forma en que lo diseñemos impactará en el diseño de los demás aspectos de la configurabilidad.

Como toda aplicación web, el acceso a la misma se realiza utilizando requerimientos http a través de una URI. En [10] se define al TenantId como el identificador único global para identificar a un tenant, el cual se obtiene a través de la URI.

El TenantId puede ser un número, un código alfanumérico, o bien puede ser el dominio de internet del cliente. Para codificar el TenantId en la URI tenemos 3 alternativas.

UTILIZAR UN PARÁMETRO DE LA URI

Se utilizaría para todos los tenant el mismo dominio, pero cada uno entraría al sistema con un parámetro de id de tenant distinto. Por ejemplo, si el dominio del sistema de alertas fuera *http://alertasonline.com*, se podrían codificar de la siguiente forma *http://alertasonline.com?TenantId=1* o *http://alertasonline.com/Tenant1*

En este caso, cada requerimiento a las siguientes páginas tienen que ser precedidas de la identificación del tenant*, http://alertasonline.com/Tenant1/Pautas* o *http://alertasonline.com/Pautas?TenantId=1*

UTILIZAR UN SUBDOMINIO DEL DOMINIO DEL SISTEMA DE ALERTAS

Cada tenant podría identificarse como http://primercliente.alertasonline.com y http://segundocliente.alertasonline.com

Cada requerimiento individual siempre se realiza utilizando el subdominio, por ejemplo http://primercliente.alertasonline.com/Pautas y http://segundocliente.alertasonline.com/Pautas

UTILIZAR EL DOMINIO DEL CLIENTE.

Cada cliente que posea su propio dominio, puede redirigir los requerimientos de cierta url a los servidores donde está alojado el sistema. Por ejemplo, se puede configurar en el servidor de internet que la misma aplicación acepte más de un dominio, de esta manera, la misma aplicación puede aceptar requerimientos del dominio www.primercliente.com y requerimientos del dominio www.segundocliente.com

En tiempo de ejecución, antes de procesar el requerimiento, se puede analizar la URI y según el dominio determinar el Tenant.

Cualquier de estas tres alternativas es válida para identificar el tenant, la elección particular de una de ellas, dependerá de las características de la plataforma como servicio donde se implemente la aplicación y de las facilidades del framework web que se utilice.

CONFIGURABILIDAD DE LA APLICACIÓN

Como se mencionó en el capítulo anterior, la configurabilidad la podemos aplicar a nivel funcional y a nivel de UI. A continuación se mostrarán dos enfoques para implementar la configurabilidad a nivel funcional. Por ejemplo [11] utiliza el enfoque basado en *features* o características y en [27] se utiliza el enfoque basado programación orientada al contexto.

ENFOQUE BASADO EN *FEATURES*

Un feature, es una característica de funcionamiento específico del sistema, cuya implementación es variable y donde cada tenant puede elegir entre un conjunto predefinido de las mismas, o bien, puede optar entre utilizarla o no.

Según [11], en la Figura 5-1, desde el punto de vista del desarrollo, cada *feature* es una unidad de código la cual se puede componer con otras y con la aplicación base para lograr una funcionalidad completa, en tal caso será necesario establecer los puntos de variación donde se enlazarán estas features, pudiendo utilizarse consistentemente en las diferentes capas de la aplicación.

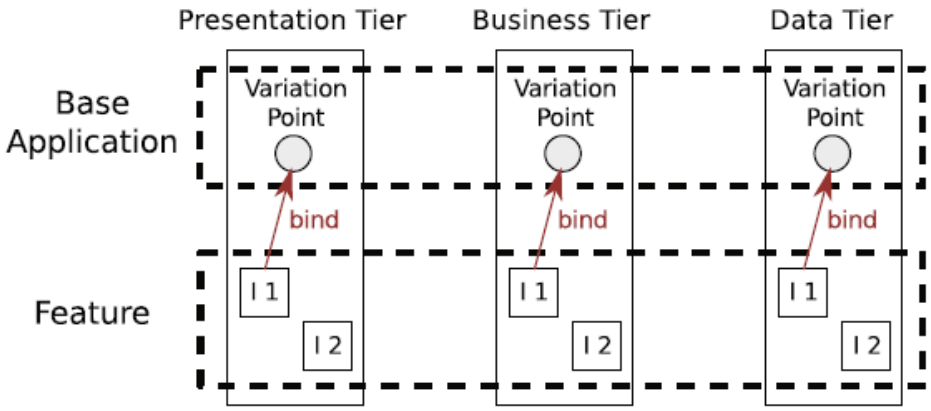

Figura 5-1 - Modelo basado en Features

En la Figura 5-2 se muestra el modelo de componentes que propone [11] para implementar un framework de features.

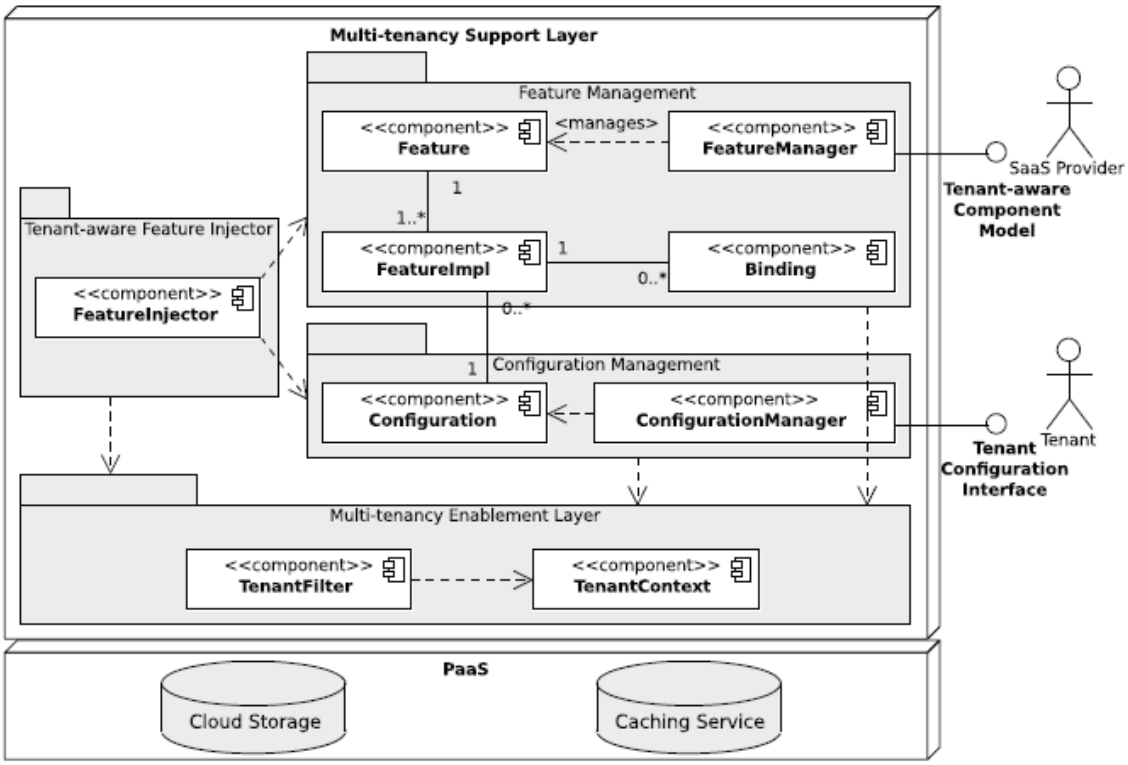

Figura 5-2 - Modelo de componentes para un framework basado en Features

1) El módulo Feature Management provee una API para manejar la variabilidad de la aplicación y la disponibilidad de las implementaciones de las features.

   El componente FeatureManager administra el conjunto de Features disponibles y sus diferentes implementaciones. Una Feature especifica al menos la siguiente información: un identificador único, por ej., el nombre de la feature, una descripción de la feature, y un conjunto de diferentes implementaciones para cada feature.
   Una FeatureImpl contiene la descripción de la implementación de la Feature, un conjunto de enlaces, y una referencia a la interface de configuración para esta implementación.
   Cada Binding especifica el mapeo de un punto de variación a un componente de software específico. La metadata relacionada con los features es accesible globalmente, tanto por el proveedor de SaaS como por los tenant, por lo que esta información no debe ser aislada.
   El componente FeatureManager ofrece una API de desarrollo para permitir al proveedor SaaS crear y registrar características junto con sus diferentes implementaciones.

2) El módulo Configuration Management administra las configuraciones por default y las específicas de cada tenant.
   Puesto que cada Feature puede tener diferentes implementaciones, cada tenant puede especificar su preferencia por una implementación de feature específica, utilizando la interface de configuración del tenant. El componente Configuration define el mapeo entre una feature y una implementación específica, más específicamente entre un ID de Feature y una FeatureImpl. El módulo ConfigurationManager maneja las diferentes configuraciones específicas de cada tenant. Al contrario de las descripciones de las features, las configuraciones específicas de cada tenant son almacenadas aisladamente para cada tenant. Además el proveedor SaaS debe especificar para cada feature una implementación por default, de modo que si el tenant no elige ninguna, automáticamente se cargue la implementación default.

3) El módulo Feature Injection se encarga de inyectar dinámicamente las variaciones de software requeridas de acuerdo a las configuraciones específicas de cada tenant.
   Basado en las features registradas en el componente FeatureManager, la capa de soporte de multi-tenant tiene que activar la implementación de feature apropiada cuando es requerido. Para lograr esto es necesario utilizar el patrón Dependency Injection. En lugar de instanciar las implementaciones de las features directamente desde el código de la aplicación, el flujo de control es invertido: la administración de ciclo de vida de las implementaciones de las features, es controlado por un proveedor inyector de dependencias. Este inyector enlaza las dependencias definidas en la aplicación con un archivo de implementación. Este enlace o binding, es tradicionalmente entre un tipo (una clase abstracta o una interface) con un tipo de implementación (una clase concreta)

Para cada punto de variación en la aplicación, el FeatureInjection decide en tiempo de ejecución cuál implementación necesita utilizar, basado en la configuración. Primero, el FeatureInjector intercepta el requerimiento a una dependencia y consulta el ConfigurationManager. Este componente utiliza el TenantId para recuperar los datos específicos del Tenant, y obtiene la clase que eligió el tenant que implementa la funcionalidad específica de la feature. Luego, esta clase es instanciada y se ejecuta la funcionalidad específica. Adicionalmente, la instancia inyectada puede almacenarse en una caché para que los próximos requerimientos no se incurra en el costo de instanciación.

ENFOQUE UTILIZANDO CONTEXT ORIENTED PROGRAMMING

En [27] se muestra cómo se puede utilizar COP para implementar la configurabilidad de una aplicación multi-tenant. COP esta específicamente dirigido para construir software altamente adaptable, el cual permite escribir aplicaciones que pueden variar su comportamiento dinámicamente acorde a su contexto de ejecución.
En COP cada porción variable de código se la define como una layer. Cada layer puede ser activada dependiendo por ejemplo del TenantId y de su contexto asociado.

Tomando el ejemplo de la referencia [27], que utiliza el framework ContextJ para aplicaciones desarrolladas en Java, se muestra a continuación como se podría utilizar COP para implementar la funcionalidad variable de un algoritmo de búsqueda simple y otro avanzado.

```
publicstatic final Layer BusquedaSimple = new Layer();
publicstatic final Layer BusquedaAvanzada = new Layer();

publicclass ClasificadorNotas {

public Nota[] ejecutarAlgoritmoClasificacion(string palabrasClaves)
   {
//Algoritmo default
   }

   layer BusquedaSimple {
public Nota[] ejecutarAlgoritmoClasificacion(string palabrasClaves)
      {
//Algoritmo de búsqueda simple
      }
   }

   layer BusquedaAvanzada {
public Nota[] ejecutarAlgoritmoClasificacion(string palabrasClaves)
      {
```

```
//Algoritmo de búsqueda avanzada
    }
  }
}
```

Según el Layer seleccionado, que puede depender el tenant actual, se ejecutará el método ejecutarAlgoritmoClasificacion correspondiente, el cual para cada Layer tiene una diferente implementación.

El enfoque de COP propone un cambio radical en la forma de manejar el flujo de ejecución de una aplicación y representa una estrategia muy interesante para escenarios de gran variabilidad.

Encarar una aplicación web multi-tenant sin un framework concreto para soportar la configurabilidad puede tornarse una tarea de codificación y mantenimiento insostenible. Por su puesto será necesario analizar el nivel de configurabilidad para poder optar entre las diferentes soluciones. Si el nivel de configurabilidad es bajo, la mejor opción es utilizar directamente MVC, si tenemos un nivel de configurabilidad medio podemos optar por el enfoque de Features, ya que con pocas clases podemos tener un framework potente. Para escenarios de alta configurabilidad podemos optar por el uso de COP, ya que es el framework con mayor grado de flexibilidad.

DISEÑO DE LA BASE DE DATOS

El soporte de la configurabilidad para diferentes tenants, implica diseñar la base de datos específicamente para soportar ésta característica. A continuación se muestran diferentes técnicas para alcanzar este objetivo.

BASE DE DATOS DEDICADA PARA CADA TENANT

Esta aproximación consiste en dar a cada tenant su propia base de datos, donde cada tenant puede extender de acuerdo a sus necesidades.

De esta forma, cada vez que un nuevo tenant es creado, se crea una nueva base de datos default, como parte del proceso de aprovisionamiento. Una vez que es creada, el tenant es libre de modificarla o extenderla, de acuerdo a sus necesidades particulares de negocio o de interface, permitiendo crear nuevos campos, nuevas consultas, o incluso nuevas tablas y relaciones.

Esta puede ser una aproximación interesante si el costo de alocar una nueva base de datos por cada tenant no es importante para el proveedor, ya que es el modelo más simple de construir y se ofrece a los clientes la máxima libertad de extender el modelo de datos default. Además para clientes que manejan información bancaria o registros médicos, donde se necesita asegurar al cien por ciento el aislamiento de datos, esta puede ser la única opción viable.

La desventaja es que esta aproximación para una cantidad considerable de tenant puede consumir muchos recursos, y los costos de infraestructura serán bastante altos, además esta solución no es escalable ya que cada server soporta una cantidad fija de bases de datos.

Por otro lado, el código de la aplicación debe estar preparado para afrontar diferentes bases de datos, con esquemas diferentes, y si no se programa específicamente la aplicación para soportar diferentes funcionalidades de acuerdo al esquema de la base de datos, se puede terminar desarrollando versiones diferentes del mismo software, con el consiguiente aumento de la complejidad de mantener versiones distintas del mismo software para diferentes tenants.

BASE DE DATOS COMPARTIDA, DIFERENTES TABLAS POR TENANT

Esta aproximación utiliza la misma base de datos para todos los tenant. Cada vez que se crea un nuevo tenant, el proceso de aprovisionamiento crea las tablas específicas para el nuevo tenant, nombrando a cada tabla con un prefijo o sufijo para poder identificar a qué tenant pertenece esta tabla.

Esta opción reduce los costos de administración, puesto que se comparte la misma instancia de base de datos para todos los tenant. Además permite que cada tenant pueda extender sus tablas, agregando nuevos campos y relaciones, sin interferir con otros tenants.

Por ejemplo:

Select * from Pautas_Tenant_1

Select * from Pautas_Tenant_2

La desventaja es que hay que tener en cuenta que el código de la aplicación deberá generar todos los SQL de forma dinámica, ya que antes de hacer cualquier consulta, inserción, modificación o eliminación deberá armar el nombre de la tabla de acuerdo al tenant que ejecute la sentencia. Esta aproximación además tiene el costo de que generará un catálogo de la base de datos bastante grande, ya que se debe almacenar para cada tabla su log de transacciones, sus índices, etc.

También si se necesita información estadística de todos los tenant, se necesitarán procesos de consolidación que unifiquen la información.

BASE DE DATOS COMPARTIDA, MISMAS TABLAS, NUEVO CAMPO PARA DISTINGUIR LOS REGISTROS DE CADA TENANT

Esta aproximación utiliza una misma instancia de base de datos y las mismas tablas para todos los tenants. Para distinguir en cada tabla, las filas correspondientes a un tenant específico, en cada tabla se agrega un campo adicional que represente el id del tenant.

Por ejemplo

Select * from Pautas where TenantId = 1

En el proceso de aprovisionamiento no hará falta crear nuevas bases de datos ni nuevas tablas, sino que solo se harán inserciones en tablas ya existentes, teniendo siempre presente de relacionar todas las entidades con el TenantId

Desde el punto de vista del código de la aplicación, siempre que se necesita ejecutar una sentencia SQL contra la base de datos, será necesario incluir una cláusula where adicional para filtrar por el ID del tenant, sin esta cláusula se estaría rompiendo el asilamiento entre los diferentes tenants.

Para esta solución necesitaremos definir una nueva tabla, que contendrá la información relacionada con cada tenant.

FEDERACIÓN DE BASE DE DATOS

El concepto de Federación es una manera de alcanzar gran escalabilidad y performance, ya que se particiona la base de datos de forma horizontal, lo que significa que una o más tablas de la base de datos tendrán grupos de filas en múltiples bases de datos, que forman parte de la Federación como miembros de la misma. Este tipo de particionamiento horizontal es referido como Sharding.

Una Federación es una colección particionada de bases de datos que son definidas bajo un esquema de distribución, conocido como Esquema de Federación. Este esquema define una clave de distribución, el cual determina cómo se distribuirá la información en las diferentes particiones. Las particiones de base de datos dentro de una Federación son conocidas como miembros de la federación.

En [28] se puede encontrar una descripción más detallada sobre cómo funciona la Federación en Windows Azure y cómo se puede implementar para soportar una aplicación multi-tenant

Una alternativa más sofisticada es el soporte nativo del RDBMS para multi-tenant. Para un análisis del mismo consultar [9].

USO DE OBJECT RELATIONAL MAPPINGS (ORMS)

Un ORM u Object Relational Mapping es el componente de software que permite utilizar las entidades de una base de datos relacional como si fueran los objetos de un modelo de

objetos. Este tipo de software permite abstraerse del DBMS que se utilizando, y permite que el desarrollador se concentre en consultar, insertar, modificar y eliminar objetos, utilizando un lenguaje de objetos, en lugar de SQL. Por lo general, un ORM es un framework independiente de la aplicación, el cual implementa toda la funcionalidad necesaria para interactuar con diferentes DBMSs y trae su propio lenguaje de consultas de objetos.
Además de que el desarrollador no tiene que preocuparse de escribir código SQL, la ventaja principal es que se genera el SQL de forma dinámica, permitiendo agregar filtros antes y después de la generación del SQL. Esta característica facilita enormemente trabajar con consultas que necesitan acceder a información de cierto tenant, ya sea que estén implementadas con tablas diferentes para cada tenant, o que haya una columna en cada tabla para identificar el tenant. Para el código de la aplicación es transparente el método por el cual este particionada la base de datos, y será tarea del ORM, a través de alguna configuración o extensión, obtener el id del tenant actual, y agregar los filtros necesarios o determinar en tiempo de ejecución el nombre de la tabla a acceder según el tenant. En [24] podemos encontrar una buena discusión sobre el uso de ORMs.

El método para particionar la base datos a elegir dependerá de la complejidad que se quiera afrontar. En la Tabla 5-1 *se muestra un resumen de las ventajas y desventajas.*

Tabla 5-1 - Técnicas para particionar la base de datos

Método	***Ventaja***	***Desventaja***
Diferentes bases de daos	*Fácil de implementar*	*Consume muchos recursos de infraestructura*
Misma base, diferentes tablas	*Consume menos recursos, seguridad y aislamiento a nivel base de datos*	*Difícil para consolidar información*
Misma base, mismas tablas, campo discriminador	*Consume mínimos recursos. Fácil para consolidar información*	*Seguridad y aislamiento a nivel de aplicación.*
Federación de base de datos	*Partición a nivel de DMBS. Mejor performance.*	*Lo debe soportar el motor de DBMS. Es un servicio más caro.*

CONCLUSIÓN

La configurabilidad es uno de los requisitos técnicos más importantes e involucra resolver varias cuestiones tanto a nivel de aplicación como a nivel de datos, que no aparecen en aplicaciones *single-tenant*. Los requisitos de escalabilidad, aprovisionamiento y monitoreo

que también se exigen para aplicaciones SaaS *multi-tenant*, no se alcanzan directamente con técnicas de diseño y programación, sino que es necesario que la plataforma subyacente donde se aloje la aplicación brinde servicios específicos para tal fin. En el siguiente capítulo se analizarán las plataformas como servicio, y cómo los servicios que ofrecen pueden cubrir este tipo de requisitos.

CAPÍTULO 6 DESARROLLO DE APLICACIONES SAAS SOBRE PAAS

Las características técnicas de las soluciones SaaS como escalabilidad, aprovisionamiento y monitoreo, deben ser cubiertas por la infraestructura donde se aloje la solución. Las plataformas como servicio vienen preparadas para ofrecer estos servicios a las aplicaciones SaaS de modo que puedan crecer en infraestructura según la demanda, sin degradar la performance del sistema. En este capítulo se van a identificar y describir los servicios ofrecidos de dos principales proveedores de PaaS en función requisitos técnicos para construir una aplicación SaaS *multi-tenant*.

INTRODUCCIÓN

Según [1] podemos encontrar tres tipos diferentes de plataforma como servicio

- Plataforma orientada a la colaboración
  Estas plataformas están orientadas al desarrollo colaborativo de software, proveyendo herramientas de desarrollo, testing y deploy de forma integrada, ofreciendo IDEs de desarrollo basados en Web.
  Algunas plataformas incluyen hasta herramientas de planeamiento de proyecto y herramientas de comunicación.
  Por lo general soportan un único tipo de lenguaje y están orientadas para desarrollar aplicaciones web single-tenant o mono-cliente.

- Plataforma de propósito general
  Este tipo de plataformas por lo general soportan más de un lenguaje de desarrollo y tienen una arquitectura específica que es necesario aprender para desarrollar las aplicaciones.
  Están pensadas para dar soporte y mecanismos de escalabilidad automática.
  Permiten desarrollar aplicaciones multi-tenant o multi-cliente, necesarias para ofrecer aplicaciones SaaS.
  Por lo general es necesario bajar un SDK y utilizar una IDE de desarrollo de escritorio.

- Plataforma orientada a un producto particular.
  Este tipo de plataformas ponen a disposición del desarrollador un marco de trabajo para extender un producto web ya existente.
  El caso emblemático es Force.com, que es la plataforma que permite desarrollar extensiones para el producto web Salesforce CRM.

Según esta clasificación, el tipo de plataforma que necesitamos para desarrollar soluciones SaaS multi-tenant son las plataformas de propósito general, las dos

plataformas elegidas para el caso de estudio son *Google App Engine* y *Windows Azure*, por una cuestión de popularidad y madurez en el mercado.

Para no extender el análisis, se descartó el servicio de PaaS *Amazon BeansTalk*, ya que su arquitectura es muy similar a la de *App Engine*, y si bien *Amazon Web Services*, es un proveedor de IaaS muy conocido, el servicio de PaaS es aún reciente, y no tiene aún un tiempo de maduración importante en el mercado.

HERRAMIENTAS Y SERVICIOS ESENCIALES

Para poder llevar a cabo un buen diseño de una aplicación SaaS multi-tenant necesariamente la plataforma debe dar soporte a una serie de cuestiones, sin las cuales no podríamos diseñar una aplicación multi-tenant, o bien, sería mucho más complejo y hasta inviable. Estas cuestiones son las que se analizarán para cada plataforma en el siguiente apartado.

Portal Web para administración

Existen muchas tareas administrativas relacionadas con la plataforma que necesitan realizarse de forma manual, como configuración de dominios, accesos ftp, creación de bases de datos, monitoreo gráfico de uso de recursos, creación de aplicaciones, subir los ejecutables de la aplicación, etc. Para todas estas tareas es necesario que la plataforma ofrezca un portal web donde poder llevar a cabo todas estas acciones.

Lenguajes, frameworks, API, SDK del middleware

Como vimos, los lenguajes y frameworks que se utilicen tienen una gran importancia a la hora de construir una aplicación SaaS multi-tenant, y las diferentes plataformas pueden soportar más de un lenguaje para desarrollarlas. Será necesario entonces evaluar cuál se ajusta a las necesidades particulares, y qué frameworks para dichos lenguajes ya vienen soportados por la plataforma.

Ya que cada plataforma tiene su propio middleware y que las aplicaciones que se desarrollen se van a montar sobre dicho middleware, la plataforma necesita exponer una API y un SDK para que el programador pueda acceder a todos los servicios de la misma y desarrollar su aplicación en su ambiente local.

Acceso a datos

La mayoría de las aplicaciones de negocio necesitan utilizar una base de datos relacional. Alojar una aplicación en la nube, pero accediendo a una base de datos local o en otro servidor de internet, es muy poco performante, por lo que es muy útil acceder a una base de datos relacional alojada en el mismo datacenter y administrada por la misma plataforma.

Mecanismos de escalabilidad

La plataforma debe brindar algún mecanismo explícito para que la aplicación pueda escalar. La plataforma debe definir de qué forma y utilizando qué parámetros proveerá este mecanismo.

Soporte para multi-tenant

La plataforma específicamente debe dar soporte para construir este tipo de aplicaciones, ya que será necesario aislar la utilización de los recursos de acuerdo a cada tenant. Existen varias técnicas para identificar a los diferentes tenants dentro de una aplicación, pero siempre será necesario que la plataforma brinde mecanismos que faciliten esta cuestión.

Servicio de monitoreo

Debido a que una plataforma como servicio, es un servicio cloud, funciona bajo la modalidad pay per use o pago por uso, por lo que siempre es necesario que el proveedor pueda visualizar el consumo de los recursos en cualquier momento, de modo de saber exactamente lo que hay que pagar.

Ejecución de procesos en background

Para facilidad la escalabilidad y la performance es necesario que la plataforma provea mecanismos para ejecutar tareas en segundo plano. Sin esta funcionalidad todo el trabajo lo tendría que realizar la aplicación online, y debido a la alta carga de trabajo que tendrá con los todos los tenants, se verá degradada la performance.

En las siguientes dos secciones se mostrarán los servicios de las dos plataformas Google App Engine y Windows Azure, que cubren estas características técnicas.

GOOGLE APP ENGINE

Google App Engine es la plataforma como servicio que ofrece Google para alojar aplicaciones web en los datacenter de Google. App Engine suscribe al modelo de pago por uso. No existen costos de configuración ni tarifas recurrentes. Los recursos que utiliza la aplicación como, por ejemplo, el almacenamiento y el ancho de banda, se miden por gigabytes y se facturan de acuerdo al consumo. App Engine permite controlar la cantidad máxima de recursos que consume la aplicación, de modo que siempre permanezcan dentro del presupuesto.

App Engine se puede empezar a utilizar de forma totalmente gratuita. Todas las aplicaciones pueden utilizar hasta 500 MB de almacenamiento y suficiente CPU y ancho de banda como para permitir un servicio eficaz de alrededor de cinco millones de visitas al mes, sin costo alguno. El desarrollador cuando lo desee puede habilitar la facturación para su aplicación, por lo cual se incrementan los límites gratuitos y solo se pagará aquellos recursos que se utilizan por encima de los niveles gratuitos.

ENTORNO DE EJECUCIÓN, LENGUAJES, SDKS

En App Engine, la unidad de computación se llama *Instancia*. Estas *instancias* son similares a las máquinas virtuales que ofrecen los proveedores de IaaS, pero la diferencia es que éstas se ejecutan en un entorno seguro, llamado *sandbox*, que proporciona acceso limitado al sistema operativo subyacente. El entorno seguro, aísla la aplicación en un entorno de confianza, totalmente independiente del hardware, del sistema operativo y de la ubicación física del servidor web. Estas limitaciones permiten a App Engine distribuir balancear la carga de trabajo en diferentes *instancias* y en varios servidores e iniciar y detener las *instancias* según la demanda.

App Engine ofrece dos tipos de instancias

Instancias de FrontEnd: son instancias que corren el código de la aplicación y escalan dinámicamente en base a los requerimientos, pero están limitadas en el tiempo que puede durar este requerimiento, por ej., un requerimiento no puede demorar más de 30 segundos en ejecutarse. Las instancias de FrontEnd manejan todos los requerimientos por default, incluidos las tareas que corren en background.

Tabla 6-1 - Tipos de FrontEnd con sus diferentes capacidades

Tipo de FrontEnd	**Límite de Memoria**	**Límite de CPU**	**Costo Por Hora**
F1 (default)	128MB	600MHz	U$S 0.08
F2	256MB	1.2GHz	U$S 0.16
F4	512MB	2.4GHz	U$S 0.32
F4_1G	1024MB	2.4GHz	U$S 0.48

Instancias de Backend: Es un tipo de instancia cuya duración y escalabilidad es determinada por la configuración. Este tipo de instancias se utilizan con tareas asincrónicas solamente y su principal ventaja es que no tienen un límite de tiempo de ejecución.

Tabla 6-2 - Tipos de BackEnd con sus diferentes capacidades

Tipo de BackEnd	Límite de Memoria	Límite de CPU	Costo Por Hora
B1	128MB	600MHz	U$S 0.08
B2 (default)	256MB	1.2GHz	U$S 0.16
B4	512MB	2.4GHz	U$S 0.32
B4_1G	1024MB	2.4GHz	U$S 0.48
B8	1024MB	4.8GHz	U$S 0.64

Algunos ejemplos de las limitaciones del entorno seguro son:

- Una aplicación solo podrá acceder a otros equipos de Internet a través de los servicios de correo electrónico y extracción de URL proporcionados. Otros equipos solo se podrán conectar a la aplicación mediante solicitudes HTTP (o HTTPS) en los puertos estándar.

- Una aplicación no podrá escribir en el sistema de archivos. Una aplicación podrá leer archivos, pero solo aquellos subidos con el código de la aplicación. La aplicación deberá utilizar el almacén de datos de App Engine, Memcache u otros servicios para todos los datos que permanezcan entre las solicitudes.

- El código de aplicación solo se ejecuta en respuesta a una solicitud web, a una tarea en cola o a una tarea programada y debe devolver datos de respuesta en un periodo de 30 segundos en cualquier caso. Un controlador de solicitudes no podrá generar un subproceso ni ejecutar código después de haber enviado la respuesta.

La plataforma permite a los desarrolladores escribir sus aplicaciones en Java o Python.

El entorno de Java proporciona un JVM Java 6, una interfaz de Servlets Java y la compatibilidad de interfaces estándar con los servicios y el almacén de datos escalable de App Engine como, por ejemplo, JDO, JPA, JavaMail y JCache.

La JVM se ejecuta en un entorno seguro para aislar la aplicación por servicio y seguridad. El espacio aislado garantiza que la aplicación solo pueda realizar acciones que no interfieran con el rendimiento ni con la escalabilidad de otras aplicaciones. Por ejemplo, una aplicación no puede generar cadenas, escribir datos en el sistema de archivos local ni establecer conexiones de red arbitrarias. Una aplicación tampoco puede utilizar JNI ni ningún otro código nativo. La JVM puede ejecutar cualquier código de bytes de Java que opere dentro de las restricciones del entorno seguro.

El SDK Java de App Engine incluye herramientas para probar la aplicación, subir archivos de la aplicación y descargar datos de registro. El SDK también incluye un componente para Apache Ant que permite simplificar tareas comunes en proyectos de App Engine. El complemento de Google App Engine para Eclipse añade funciones al IDE de Eclipse para desarrollar, probar e implementar App Engine y, además, incluye el SDK completo de App Engine. El complemento de Eclipse también facilita el desarrollo de las aplicaciones de Google Web Toolkit y las ejecuta en App Engine.

El SDK incluye un servidor de desarrollo que ejecuta la aplicación en un equipo local para desarrollarla y probarla, simulando el *sandbox* de App Engine. El servidor además simula el almacén de datos, los servicios y las restricciones del espacio aislado de App Engine.

Normalmente, los desarrolladores de Java utilizan el lenguaje de programación Java y las API para implementar aplicaciones web para JVM, pero también se pueden utilizar intérpretes o compiladores compatibles con JVM, como por ejemplo, JavaScript, Ruby o Scala.

TECNOLOGIAS, ARQUITECTURAS Y FRAMEWORKS WEB SOPORTADOS

Las tecnologías que forman parte de Java Enterprise Edition no son 100 % compatibles con App Engine, debido a las restricciones que impone el *sandbox*.

Entre los componentes de la especificación Java EE que sí soporta App Engine están incluidos los siguientes:

- Java Data Objects (JDO)
- Java Persistence API (JPA)
- Java Server Faces (JSF) 1.1 - 2.0 2
- Java Server Pages (JSP) + JSTL 3
- Java Servlet API 2.4
- JavaBeans™ Activation Framework (JAF)
- Java Architecture for XML Binding (JAXB)
- Java API for XML Web Services (JAX-WS) 4
- JavaMail
- XML processing APIs including DOM, SAX, and XSLT

Entre los que no están incluidos tenemos los siguientes:

- Enterprise Java Beans (EJB)
- JAX-RPC
- Java Database Connectivity (JDBC)
- Java EE™ Connector Architecture (JCA)
- Java Management Extensions (JMX)
- Java Message Service (JMS)
- Java Naming and Directory Interface (JNDI)
- Remote Method Invocation (RMI)

Para desarrollar aplicaciones web con Java en App Engine podemos utilizar el estándar Java Servlet. App Engine permite alojar los servlets de la aplicación, las páginas JSP,

archivos estáticos y archivos de datos, junto con el descriptor de implementación (el archivo web.xml) y otros archivos de configuración en una estructura de directorio WAR estándar.

Google Web Toolkit

La librería GWT son un conjunto de APIs escritas en Java. Estas APIs permiten escribir aplicaciones Ajax utilizando java, pero cuya salida es código Javascript multi-browser. El desarrollador no necesita escribir una sola línea en javascript, ya que las APIs se encargan de generar todo el código necesario.

MULTI-TENANT DE APLICACIÓN Y DE DATOS

Google App Engine da pleno soporte para construir aplicaciones multi-tenant a través de una API de espacios de nombres. Esta API permite definir un espacio de nombre único para cada tenant de forma programática. De esta forma, cada requerimiento web hacia la aplicación lleva asociado el espacio de nombres correspondiente, permitiendo identificar al tenant y acceder a los servicios de GAE correspondientes según cada tenant.

Los servicios de App Engine que soportan namespace son: Datastore, Memcache, Task Queues.

PORTAL WEB DE ADMINISTRACIÓN

Google App Engine dispone de una consola de administración para administrar, monitorear y configurar las aplicaciones alojadas en la plataforma. Desde esta consola se puede además crear nuevos IDs de aplicaciones, invitar a otros desarrolladores a contribuir con el desarrollo de la aplicación, ver información de acceso y los log de errores, analizar el tráfico, inspeccionar el DataStore, administrar las tareas programadas, y mucho más.

Utilizando la cuenta personal de Google, la que se usa para acceder a servicios como Gmail, se puede ingresar a la consola de administración a través del link http://appengine.google.com

ESCALABILIDAD

Google App Engine provee un mecanismo de escalabilidad automática, a medida que el tráfico a la aplicación crece, se van creando más instancias para manejar la carga de trabajo excesiva, sin la necesidad de monitorear y requerir nuevos recursos manualmente. Permite definir en tiempo de diseño y de ejecución la cantidad mínima y máxima de instancias FrontEnd y Backend.

EJECUCIÓN DE PROCESOS EN BACKGROUND

En App Engine las tareas en background las pueden llevar a cabo tanto las instancias de tipo FrontEnd como las instancias de tipo BackEnd. La diferencia radica en que las instancias FrontEnd tiene un límite de tiempo acotado, según la especificación actual de 30 segundos, que las instancias de tipo BackEnd no las tienen, por lo que en este último tipo de instancias se pueden llevar a cabo tareas que lleven más tiempo.

Las tareas en background se pueden disparar a través de eventos temporales o bien utilizando colas de mensajes.

Para eventos temporales *Google App Engine* ofrece el servicio de *Planificación de Tareas*, más conocido como *"Servicio Cron"*. Este servicio permite a las aplicaciones ejecutar tareas que no estén relacionadas con un requerimiento web, sino que se ejecutan a intervalos regulares de tiempo o a una hora especificada. En otras palabras, las aplicaciones pueden crear y correr tareas en background mientras atienden los requerimientos web.

El servicio de colas de mensajes de *App Engine* se llama *Task Queues Service*, y permite realizar una comunicación asincrónica entre instancias de tipo *FrontEnd* y *BackEnd*, o entre diferentes instancias de tipo *BackEnd* para disparar tareas en background. En el capítulo 13 de [4] se puede encontrar más información sobre el manejo del servicio de colas.

ACCESO A DATOS

Almacenamiento relacional

App Engine ofrece el servicio Google Cloud SQL, que es un servicio que utiliza base de datos MySQL que viven en la nube de Google. Tiene todas las capacidades y funcionalidades de un motor MySQL, con algunas pocas características adicionales, y algunas características no soportadas. Google Cloud SQL automáticamente mantiene y provisiona las bases de datos.

Características

- Permite alojar bases de datos MySQL existentes en la nube.
- Replicación geográfica sincrónica y asincrónica.
- Importar y exportar bases de datos usando mysqldump.
- Compatibilidad con Java y Python
- Herramienta de línea de comandos.

Restricciones

- El límite de tamaño para instancias individuales es de 100 GB
- Las funciones definidas por el usuario no son soportadas
- La replicación de MySQL no está soportada
- Las siguientes sentencias SQL no están soportadas:

- LOAD DATA INFILE
- SELECT ... INTO OUTFILE/DUMPFILE
- INSTALL/UNINSTALL PLUGIN ...
- CREATE FUNCTION ...
- LOAD_FILE()

Almacenamiento NoSQL

Google App Engine ofrece un servicio de almacenamiento NoSQL llamado DataStore, para almacenar y consultar datos estructurados no relacionales. El servicio DataStore es un servicio de almacenamiento de objetos sin esquema, basado en el mecanismo de recupero clave/valor, con un motor de consultas y de transacciones atómicas solo a nivel de filas. El servicio DataStore está construido sobre BigTable, en cual además está construido sobre Google File System, por lo que escala perfectamente bien cuando la cantidad de datos crece y tiene además alta disponibilidad y confiabilidad. Una ventaja es que se el servicio puede accederse vía JDO y JPA. Para más información sobre como manipular del DataStore se puede consultar el capítulo 6 de [3].

WINDOWS AZURE

*Windows Azure*s es el servicio de *Cloud Computing* de *Microsoft*. Actualmente ofrece servicios de *IaaS* y de *PaaS*. Como proveedor de *PaaS* la plataforma permite alojar aplicaciones web en los datacenter de *Microsoft*. *Windows Azure*, al igual que *App Engine* suscribe al modelo de pago por uso. Tampoco existen costos de configuración ni tarifas recurrentes. Los recursos que utiliza la aplicación como, por ejemplo, el almacenamiento y el ancho de banda, se miden por gigabytes y se facturan de acuerdo al consumo.

Cloud Services, es el servicio de *Windows Azure* que ofrece un entorno de cómputo para alojar aplicaciones web escalables y distribuidas, sin preocuparse por la infraestructura subyacente. Si bien la infraestructura de *Cloud Services* está basada en servidores de *Windows Servers*, las aplicaciones interactúan con un middleware que abstrae muchas de las particularidades del Sistema Operativo, permitiendo desarrollar aplicaciones en .NET, Java, PHP, C++, etc.

ENTORNO DE EJECUCION, LENGUAJES Y SDKS

El entorno de ejecución de *Windows Azure Cloud Services* consiste contenedores virtuales llamados Roles. Un Rol es similar a una máquina virtual, en el sentido de que tiene asignado una cierta cantidad de memoria y velocidad de CPU, pero tiene restringido el acceso al Sistema Operativo subyacente.

Los tipos de roles son:

Web Role: es un tipo de contendor virtual para alojar aplicaciones web. Cada *Web Role* puede definir una o más aplicaciones web, aunque si el sitio tiene mucho tráfico, por lo general un *Web Role* tendrá una única aplicación para poder dedicar los recursos completos a la misma.

En la Tabla 6-3 se muestran los diferentes tamaños de Web Roles:

Tabla 6-3 - Tamaños de Web Role

Tamaño	Cantidad Cores	Velocidad CPU	Memoria	Costo Por Hora
Extra Small	Compartido	1.0 GHz	768 MB	U$S 0.02
Small	1	1.6 GHz	1.75 GB	U$S 0.12
Medium	2	1.6 GHz	3.5 GB	U$S 0.24
Large	4	1.6 GHz	7 GB	U$S 0.48
Extra Large	8	1.6 GHz	14 GB	U$S 0.96

Worker Role: es un tipo de contenedor virtual para realizar tareas en background y dar soporte a los *Web Role* para tareas asincrónicas.

Los tipos o tamaños de Worker Role son los mismos que para los web role:

Tabla 6-4 - Tamaños de Worker Role

Tamaño	Cantidad Cores	Velocidad CPU	Memoria	Costo Por Hora
Extra Small	Compartido	1.0 GHz	768 MB	U$S 0.02
Small	1	1.6 GHz	1.75 GB	U$S 0.12
Medium	2	1.6 GHz	3.5 GB	U$S 0.24
Large	4	1.6 GHz	7 GB	U$S 0.48
Extra Large	8	1.6 GHz	14 GB	U$S 0.96

Si bien la plataforma permite desarrollar aplicaciones en diferentes lenguajes, el estándar para desarrollar aplicaciones Web sobre Windows Azure Cloud Services es ASP.NET MVC.

ACCESO A DATOS

Almacenamiento relacional

Para almacenamiento relacional, Windows Azure provee el servicio SQL Database, anteriormente llamado SQL Azure. SQL Database provee todas las características de un RDBMS, incluyendo transacciones atómicas, acceso a datos concurrentes para múltiples usuarios con integridad de datos, consultas en ANSI SQL. Como con SQL Server, SQL Database puede ser accedido con Entity Framework, ADO.NET, JDBC, y otras tecnologías familiares de acceso a datos. También soporta plenamente T-SQL, aunque con algunas pequeñas restricciones.

SQL Database no es solo un RDBMS en la nube. SQL Database permite que el cliente tenga el control de los datos y de quién accede a ellos, pero se encarga de la infraestructura subyacente y de las tareas administrativas, como replicación automática de datos, actualizaciones del software subyacente, etc.

Almacenamiento NoSQL

Windows Azure ofrece el servicio Azure Table Service como mecanismo de almacenamiento estructurado no relacional, basado en el mecanismo clave/valor que utilizan otros motores NoSQL.
Este servicio se ofrece como la forma más rápida de recuperación de datos. Aunque estos datos deben ser simples, ya que el servicio tiene algunas limitaciones como como no soportar joins entre tablas, una de las ventajas es que es un servicio mucho más barato que SQL Database.

Almacenamiento no estructurado

Windows Azure ofrece el servicio de almacenamientos de Blobs (Binary Large Object) que se almacenan en forma de archivos, ya sea de texto, de imágenes, html, etc. Aunque no es un sistema de archivos propiamente dicho, ya que tiene una estructura interna muy diferente, sí permite emular un sistema de archivos utilizando el servicio de Windows Azure Drives, que utiliza el servicio de Blobs montar un sistema de archivos virtual.

SOPORTE MULTI-TENANT

Windows Azure no trae un soporte multi-tenant explícito para acceder a los diferentes servicios como el servicio de Namespaces de App Engine. Por lo que obliga a que las aplicaciones se encarguen de implementar la configurabilidad y el asilamiento necesario para permitir que múltiples clientes utilicen la misma instancia de la aplicación.

EJECUCIÓN DE PROCESOS EN BACKGROUND

Las tareas en background se pueden disparar a través de eventos temporales o bien utilizando colas de mensajes.

Windows Azure a través del servicio Mobile Services provee un mecanismo de Job Schedulling para poder programar eventos temporales.
Estos Jobs permiten definir a qué hora y qué día se ejecuta una cierta tarea o cada cuanto tiempo. Cada Job tiene asociado un Script que se ejecuta cuando ocurre el evento temporal, y es través de este script que se pueden invocar al Servicio de Cola de Mensajes para delegar tareas de forma asincrónica a los Worker Role.

ESCALABILIDAD

Windows Azure Cloud Services a través de una API del SDK, permite crear múltiples instancias del mismo Web Role y Worker Role para manejar un alto volumen de requerimientos. También permite definir en tiempo de diseño la cantidad mínima y máxima de instancias, pudiendo cambiar estos valores también en tiempo de ejecución.
Windows Azure se encarga automáticamente de balancear la carga entre las diferentes instancias.
Windows Azure constantemente monitorea las instancias activas, para que en caso de fallas volver a reiniciarlas para que vuelvan a estar activas.
Estas características hacen de Windows Azure Cloud Services un servicio escalable y elástico

CONCLUSIÓN

Cada plataforma tiene sus ventajas y desventajas, y utilizan diferentes tecnologías, pero ambas apuntan a resolver las necesidades técnicas de las soluciones SaaS.
A continuación la Tabla 6-5 muestra un resumen de los requisitos técnicos según la plataforma.

Tabla 6-5 - Resumen de requisitos técnicos según la plataforma

	App Engine	**Windows Azure**
ENTORNO DE EJECUCION	Instancias FrontEnd e instancias Backend	Web Roles y Worker Roles
LENGUAJES	Java y Python	Plataforma .NET
TECNOLOGIAS, ARQUITECTURAS Y FRAMEWORKS WEB SOPORTADOS	Servlets, Struts2, Spring	ASP.NET MVC
ACCESO A DATOS	Relacional: Cloud SQL No Relacional: DataStore	Relacional: SQL Database No Relacional: Azure Tables
SOPORTE MULTI-TENANT	A través de una API de Namespaces	No tiene soporte explícito.
EJECUCIÓN DE PROCESOS EN BACKGROUND	Cron Services y Task Queues	Job Schedulling
ESCALABILIDAD	Escalabilidad automática	Escalabilidad manual y programática
Portal Web	Si	Si

.

CAPÍTULO 7 CASO DE ESTUDIO

En este capítulo se planteará un caso de estudio para unificar los conceptos vistos en los capítulos anteriores.

INTRODUCCIÓN

El caso de estudio se enfocará desde el punto de vista de un proveedor ficticio, que necesita realizar una propuesta comercial y técnica para construir una solución SaaS para uno de sus clientes. Este proveedor no tiene experiencia construyendo aplicaciones SaaS sobre plataformas como servicio, por lo que necesita realizar un análisis técnico que le permita saber si está en condiciones de construir este tipo de soluciones, el esfuerzo que le demandará y contar con lineamientos de diseño que le permitan armar una propuesta consistente y factible. El presente capítulo oficiará del análisis técnico que necesita este proveedor ficticio para armar su propuesta.

Con el aporte del análisis técnico, el proveedor estará en condiciones de:

- Minimizar los riesgos de construcción, ya que contará con alternativas de diseño de los puntos críticos del sistema.
- Estimar tiempo y esfuerzo de construcción.
- Elegir correctamente una plataforma como servicio de acuerdo a las necesidades del proyecto y del cliente.
- Implementar ciertas características específicas del proyecto en diferentes plataformas como servicio.

El caso de estudio no solo mostrará cómo se implementan en una plataforma como servicio los requisitos técnicos de una solución SaaS descriptos en los capítulos anteriores, sino que también mostrará cómo se pueden implementar ciertas características específicas del proyecto que van más allá de los requisitos técnicos propios de las soluciones SaaS.

A continuación se presenta el sistema que necesita el cliente del proveedor ficticio.

SISTEMA DE ALERTA DE NOTICIAS

Descripción general

Una empresa de análisis de medios necesita desarrollar un sistema de alerta de noticias para sus clientes. Los clientes son empresas u organismos que necesitan ser alertados sobre las noticias que se publican en los distintos medios online cuyos temas estén relacionados con diferentes pautas definidas por el cliente.

Una pauta es un conjunto de reglas de concordancia, que pueden incluir temas (política, economía, salud), palabras claves (estado, gobierno, unlp), medios (clarín, la nación, página 12) y otros datos relevantes para refinar la búsqueda.
Las noticias que coincidan con las pautas serán enviadas por email a los diferentes destinatarios dentro de la misma organización.
Este sistema debe presentar una interfaz web de modo tal de permitir al cliente acceder desde sus oficinas. Los destinatarios pueden acceder a la web y ver las noticias online.
Cada cliente u organización puede personalizar el sitio web asociado a su cuenta según sus necesidades. Por cada cliente u organización habrá un usuario administrador que será el encargado de la personalización del sitio, la definición de pautas y palabras claves, la definición los diferentes destinatarios dentro de la misma organización.
La extracción de la información online se realizará utilizando un proceso automático de scrapping a los diferentes medios de internet.
El proceso completo de extracción debe generar una única base de datos con las noticias.
Se debe tener en cuenta que el motor de base de datos elegido debe soportar mecanismos de full text search.
La facturación a cada organización dependerá de la cantidad de medios a analizar, por la cantidad de noticias enviadas, frecuencia de búsqueda, por lo que la aplicación deberá registrar estos datos y tener un proceso mensual para generar esta información de facturación.

Requerimientos de la aplicación

A continuación se enumeran los requerimientos del proyecto, de acuerdo a las necesidades del cliente:

- Debe ser una aplicación web.
- Debe ser una aplicación *multi-tenant* para maximizar la utilización de recursos.
- La aplicación debe ser escalable y flexible, ya que inicialmente serán pocos clientes, pero si la aplicación tiene éxito debe poder adaptarse fácilmente a cientos y miles de clientes.
- Se debe utilizar una base de datos transaccional con soporte multi-tenant para almacenar la información del sistema.
- Debe soportar la ejecución de procesos en background.
- Debe poder descargar páginas completas de los medios en internet.
- Se debe utilizar un motor de full text search para la búsqueda de las noticias.
- Debe poder contabilizarse el uso de ciertas partes del sistema con el fin de poder realizar una facturación mensual.
- Se debe contar con un servicio de envío de emails confiable.
- Se debe poder monitorear en todo momento la performance y el grado de utilización de los recursos del sistema.
- El portal web para cada tenant debe ser configurable.
- Diseñar la solución de modo que se puedan aprovisionar recursos a medida que se incremente la cantidad de clientes.

El análisis técnico estará dividido en dos secciones. La primera sección abordará requerimientos técnicos de base que conformarán la arquitectura de la aplicación, relacionados específicamente con los requisitos técnicos de una solución SaaS, como configurabilidad, escalabilidad, etc. La segunda sección abordará requerimientos técnicos relacionados con funcionalidad crítica del sistema de alertas, como el envío de emails, mecanismos de full text search, etc.

LINEAMIENTOS PARA LA ARQUITECTURA DEL SISTEMA

De acuerdo al análisis técnico que necesita el proveedor ficticio para confeccionar una propuesta comercial y técnica para su cliente, en esta sección se analizarán posibles implementaciones de los requisitos técnicos de una solución SaaS tanto en Google App Engine como en Windows Azure.

Se eligieron estas plataformas por una cuestión de popularidad y de tiempo de maduración en el mercado. Los ejemplos de *Google App Engine* se harán utilizando la tecnología *Java*, ya que en este lenguaje es donde existen la mayor cantidad de frameworks que permitirán desarrollar la aplicación sin inconvenientes. Los ejemplos en *Windows Azure* se harán utilizando la tecnología *ASP.NET*.

Para mantener una trazabilidad con los requerimientos del sistema de alertas, la Tabla 7-1 muestra la relación entre las siguientes subsecciones y los requerimientos técnicos que conformarán la arquitectura de la aplicación.

Tabla 7-1 - Trazabilidad entre las secciones y los requisitos técnicos

Secciones	**Requerimientos técnicos del sistema**
Implementación de la configurabilidad	Debe ser una aplicación web
	El portal web para cada tenant debe ser configurable
	Debe ser una aplicación *multi-tenant*
Elección del esquema de base de datos	Se debe utilizar una base de datos transaccional con soporte multi-tenant
Diseño de la escalabilidad	La aplicación debe ser escalable y flexible
Implementación de tareas en background	Debe soportar la ejecución de procesos en background

IMPLEMENTACIÓN DE LA CONFIGURABILIDAD

Tanto la opción de Feature como la de COP involucran desarrollar un framework específico para manejarlas, y toda la aplicación termina quedando atado a este

framework, y dado que el nivel de configurabilidad del Sistema de Alertas es relativamente bajo. En estos casos la mejor opción es utilizar MVC.

En lo que respecta al diseño de base de datos, además de las tablas relacionadas con el modelo de negocio del Sistema de Alertas serán necesarias tablas para modelar las diferentes configuraciones.

Las diferentes configuraciones a nivel de UI en el Sistema de Alertas se dan en la visualización de las noticias en cada portal individual asignado para cada tenant.

APPENGINE

Si bien existen números frameworks MVC compatibles con GAE, entre los que están los dos más conocidos Struts 2 y Spring MVC, ninguno trae una integración para soporte multi-tenant. En estos ejemplos se tomará el camino más sencillo que es utilizar directamente JSP y Servlets, ya que es el soporte nativo de App Engine para desarrollar aplicaciones web en lenguaje Java.

También se utilizará el framework de inyección de dependencia Google Guice, ya que App Engine tiene pleno soporte para el mismo.

La idea de la solución es utilizar la API de Namespaces que provee App Engine para implementar la configurabilidad tanto a nivel de UI como de Negocio, en conjunto con el inyector de dependencia Guice.

Cada tenant se identificará con un parámetro de la url. Guice trae una integración para Servlets. Utilizando esta integración, se interceptará el requerimiento con un Filter de Servlet, para obtener el parámetro del tenant y establecer el namespace actual.

Para cada tenant, se tendrá una instancia de Injector, que es la clase que se encarga de resolver dependencias.

Utilizando una clase proveedora de Injector

Con Guice podemos modificar el mecanismo de instanciamiento de los Servlets y JSP para poder inyectarle dependencias según el tenant.

Definición del filtro que se encargará de tomar el parámetro del tenant y establecer el Namespace actual. En este caso se hereda de la clase GuiceFilter de Guice para aprovechar la integración con el mecanismo de los Servlets. Cada request http que maneje la aplicación tendrá su propio Namespace actual.

```
package com.alertasonline;
```

```
import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

import com.google.appengine.api.NamespaceManager;
import com.google.inject.servlet.GuiceFilter;

publicclass NamespaceGuiceFilter extends GuiceFilter {

    @Override
publicvoid doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {

        String url = "";
        String queryString = "";

if (NamespaceManager.get() == null)
        {
if (request instanceof HttpServletRequest) {
                url = ((HttpServletRequest)request).getRequestURL().toString();
                queryString = ((HttpServletRequest)request).getQueryString();
               }

            String tenantName = request.getParameter("tenantName");

            NamespaceManager.set(tenantName);
        }

super.doFilter(request, response, chain);
    }
}
```

Código Fuente 7-1

Al momento de arrancar la aplicación y antes de cualquier requerimiento, será necesario inicializar el Proveedor de Injectors.

```
package com.alertasonline;

import java.io.IOException;
import javax.servlet.http.*;

import com.google.appengine.api.NamespaceManager;

@SuppressWarnings("serial")
publicclass StartupServlet extends HttpServlet {

publicvoid init() {
```

```
          InjectorProvider.InitProvider();
          }
}
```

Código Fuente 7-2

Abajo se muestra la claseInjectorProvider, la cual se encarga de buscar la información de la base de datos con los tenant actuales, crear una instancia de Injector para cada tenant, y armar una Hashtable con los mismos, utilizando como clave el id o nombre del tenant.

A los fines de simplificar el ejemplo, se utilizará un proveedor de Injectors falso, que carga dos injectors asociados al Tenant1 y al Tenant2.

```
package com.alertasonline;
import java.util.Hashtable;

import com.google.appengine.api.NamespaceManager;
import com.google.inject.Guice;
import com.google.inject.Injector;


publicclass InjectorProvider {

static Hashtable<String, Injector> injectors;

publicstaticvoid InitProvider()
    {
        injectors = new Hashtable<String, Injector>();

        injectors.put("Tenant1", Guice.createInjector(new
MultiTenantGuiceModule("Tenant1")));

        injectors.put("Tenant2", Guice.createInjector(new
MultiTenantGuiceModule("Tenant2")));
    }

publicstatic Injector getInjector()
    {
        String tenantName = NamespaceManager.get();
return injectors.get(tenantName);
    }
}
```

Código Fuente 7-3

Abajo se muestra el código de la clase MultiTenantGuiceModule. Esta clase es la que se encarga de relacionar las interfaces o clases abstractas con las implementaciones correspondientes según el tenant indicado en el constructor y de crear su contexto.

Esta clase también puede acceder a la base de datos para obtener la información de las diferentes implementaciones para cada tenant, y la información completa para armar el contexto. En este ejemplo, a los fines de simplificar, se muestra con información fija de prueba.

```
package com.alertasonline;

import com.google.inject.*;
import com.google.inject.name.Names;

publicclass MultiTenantGuiceModule extends AbstractModule
{
    String _tenantName;

public MultiTenantGuiceModule(String tenantName)
    {
        _tenantName = tenantName;
    }

    @Override
protectedvoid configure() {
if ( tenantName == "Tenant1")
        {
            bind(TenantContext.class)
                .annotatedWith(Names.named(_tenantName))
                .toInstance(new TenantContext("Tenant1", "orange.css"))
                ;
        }
elseif (_tenantName == "Tenant2")
        {
            bind(TenantContext.class)
                .annotatedWith(Names.named(_tenantName))
                .toInstance(new TenantContext("Tenant2", "green.css"))
                ;
        }
    }
}
```

Código Fuente 7-4

Esta clase contendrá información relacionada con cada Tenant, como el nombre o id, el archivo css de estilos, imagen de logo, etc. El contexto puede tener tanta información como el nivel de variabilidad lo requiera.

```
packagecom.alertasonline;

publicclass TenantContext implements ITenantContext {
```

```
public TenantContext(String name, String cssFile)
    {
this.name = name;
this.cssFile = cssFile;
    }

private String name;

private String cssFile;

public String getName() {
return name;
    }

publicvoid setName(String name) {
this.name = name;
    }

public String getCssFile() {
return cssFile;
    }

publicvoid setCssFile(String cssFile) {
this.cssFile = cssFile;
    }
}
```

Código Fuente 7-5

Abajo se muestra el archivo web.xml, donde se aplica el filtro NamespaceGuiceFilter para todos los requerimientos y se define el Servlet inicial StartupServlet. Los demás Servlet de la aplicación se definen normalmente, como el Servlet PortalPrincipalServlet.

```
<?xmlversion="1.0"encoding="utf-8"standalone="no"?><web-
appxmlns="http://java.sun.com/xml/ns/javaee"xmlns:web="http://java.sun.com/xml/ns/javaee/web
-app_2_5.xsd"xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"version="2.5"xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app 2 5.xsd">
<filter>
<filter-name>guiceFilter</filter-name>
<filter-class>com.alertasonline.NamespaceGuiceFilter</filter-class>
</filter>

<filter-mapping>
<filter-name>guiceFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
<servlet-name>StarupServlet</servlet-name>
```

```
<servlet-class>com.alertasonline.StartupServlet</servlet-class>
<load-on-startup>0</load-on-startup>
</servlet>


<servlet>
<servlet-name>PortalPrincipalServlet</servlet-name>
<servlet-class>com.alertasonline.PortalPrincipalServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>PortalPrincipalServlet</servlet-name>
<url-pattern>/Portal</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

Código Fuente 7-6

Abajo se muestra un ejemplo básico de uso del contexto del tenant.

```
package com.alertasonline;

import java.io.IOException;
import javax.servlet.http.*;

import com.google.appengine.api.NamespaceManager;
import com.google.inject.Injector;
import com.google.inject.Key;
import com.google.inject.name.Names;

@SuppressWarnings("serial")
publicclass PortalPrincipalServlet extends HttpServlet {
publicvoid doGet(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
        resp.setContentType("text/plain");
        resp.getWriter().println("Hello, world");
        resp.getWriter().println(NamespaceManager.get());
        String tenantName = NamespaceManager.get();

// Se obtiene el injector actual de acuerdo al tenant.
Injector injector = InjectorProvider.getInjector();

// Aqui se obtiene el contexto en base al id del tenant.
TenantContext tenantContext = injector.getInstance(Key.get(TenantContext.class,
Names.named(tenantName)));

        resp.getWriter().println(tenantContext.getName());
        resp.getWriter().println(tenantContext.getCssFile());
```

```
    }
}
```

Código Fuente 7-7

WINDOWS AZURE

En Windows Azure la solución para identificar los tenant y su configurabilidad asociada, se implementará de una forma un poco diferente.

Antes que nada, hay que saber que Windows Azure no tiene un mecanismo para identificar un tenant a lo largo de un requerimiento, como si lo tiene App Engine con la API de Namespaces.

En la implementación de App Engine se utilizó el modelo de JSP y Servlets, que no es un modelo MVC puro. En cambio para la implementación de Windows Azure se utilizará el framework ASP.NET MVC, porque es en la actualidad el framework por default para hacer aplicaciones web para Windows Azure.

Como en la implementación de App Engine, aquí también se utilizará un inyector de dependencia. Para .NET existen varios frameworks open source para realizar esto. En este caso utilizaremos uno llamado StructureMap, ya que se integra sin problemas con la plataforma de Windows Azure. Para trazar una analogía, la clase Injector de Guice, tiene su contrapartida en StructureMap, llamada Container. A lo largo de la explicación de la implementación, se utilizará este último nombre.

La idea de la solución es controlar la instanciación de cada Controller asociado a una Vista. Para ello se crearán algunas clases adicionales al modelo MVC, de modo de sea el inyector de dependencia el encargado de controlar la instanciación de los Controller.

El modelo se basa en que cada tenant tenga asociado su propio Container de inyección de dependencia, de modo que al ejecutarse los métodos del Controller, estos accedan a las implementaciones de las clases que le correspondan.

Esto no solo permitirá actualizar la vista según la información contextual del tenant, sino que permitirá también elegir diferentes implementaciones para la lógica de negocio, y a su vez se podrá trasladar el contexto del tenant a la implementación de la misma.

Lo primero que se necesita definir es el contexto del tenant. Cada contexto de tenant tendrá asociado su nombre, el theme que especifica su configuración css, la lista de urls asociadas al tenant, y el Container de inyección de dependencia.

```
publicclass TenantContext : ITenantContext {

    IContainer _container;

publicstring Name { get; set; }
publicstring Theme { get; set; }
public IEnumerable<string> Urls { get; set; }
```

```
public IContainer DependencyContainer { get { return _container; } }

publicvoid InitializeContainer(Action<ConfigurationExpression> customExpression = null) {
//Se inicializa el container de injección de dependencia de StructureMap
        _container = new Container();
        _container.Configure(config => {
            config.For<ITenantContext>().Singleton().Use(this);
if (customExpression != null)
                customExpression(config);
        });
    }
```

Código Fuente 7-8

Para el ejemplo, se creó un proveedor de tenants falso, que devuelve una lista de tenants fijos. Para el caso productivo, se necesitará acceder a la base de datos para cargar la lista de tenants actuales.

```
publicclass FakeContextProvider : ITenantContextProvider {
public IList<ITenantContext> Tenants {
        get
        {
            var tenants = new List<TenantContext>() {
new TenantContext {
                    Name = "Tenant1",
                    Theme = "Default",
                    Url = "http://tenant1.alertasonline.com:99999"
                },
new TenantContext {
                    Name = "Tenant2",
                    Theme = "Custom",
                    Url = "http://tenant2.alertasonline.com:99999/"
                }
            };
return tenants;
        }
    }
}
```

Código Fuente 7-9

El proveedor de Container, se encarga de identificar al tenant actual, y retornar el Container de Inyección de Dependencia correspondiente:

```
publicclass TenantContainerProvider : IContainerProvider {

public IEnumerable<ITenantContext> Tenants { get; private set; }

public TenantContainerProvider (IEnumerable<ITenantContext> tenants) {
```

```
this.Tenants = tenants;
    }

public IContainer Resolve(ITenantIdentifier identifier) {
return identifier.Identify(this.Tenants).DependencyContainer;
    }
}
```

Código Fuente 7-10

El identificador de tenant se basa en la url, analiza la url del requerimiento actual, y verifica si parte de la url del requerimiento coincide con el nombre de alguno de los tenant disponibles.
En el caso de encontrar una coincidencia, devuelve el contexto correspondiente.

```
publicclass UrlTenantIdentifier : ITenantIdentifier {
private RequestContext _context;

public UrlTenantIdentifier(RequestContext requestContext) {
        _context = requestContext;
    }

public ITenantContext Identify(IEnumerable<ITenantContext> tenants) {

string baseurl = _context.HttpContext.BaseUrl().TrimEnd('/');

        var valid = from tenant in tenants
                    from url in tenant.Urls
where url.Trim().TrimEnd('/').Equals(baseurl, StringComparison.OrdinalIgnoreCase)
                    select tenant;

if (!valid.Any())
thrownew TenantNotFoundException();
return valid.First();
    }
}
```

Código Fuente 7-11

Abajo se muestra el código de la clase que se encargará de proveer las instancias de los Controller para las vistas. Esta clase hereda de una clase del framework MVC de ASP.NET y es la que permite cambiar el modelo de instanciación de los Controllers.
Esta modificación permite que la instanciación del Controller pase por el Injector de Dependencia, y le inyecte el contexto del tenant correspondiente.

```
publicclass ContainerControllerFactory : DefaultControllerFactory {
public IContainerProvider ContainerProvider { get; private set; }

public ContainerControllerFactory(IContainerProvider provider) {
this.ContainerProvider = provider;
    }
```

```
protectedoverride IController GetControllerInstance(RequestContext requestContext, Type
controllerType) {
        IController result = null;
if (controllerType != null) {

// Se obtiene el Container del tenant, en base a la Url.
var container = this.ContainerProvider.Resolve(new UrlTenantIdentifier(requestContext));
try {
          // Crea una instancia de Controller a través del container.
result = container.GetInstance(controllerType) as Controller;
            } catch (StructureMapException) {
                Debug.WriteLine(container.WhatDoIHave());
throw;
            }
        }
return result;
    }
}
```

Código Fuente 7-12

Para activar todo el mecanismo el siguiente código se debe insertar en el evento Application_Start del Global.asax

```
var topContainer = new Container();
topContainer.Configure(config => {
    config.For<ITenantContextProvider>().Use<FakeContextProvider>();
});

var tenants = new FakeContextProvider().Tenants;


var containerProvider = new TenantContainerProvider(tenants);

ControllerBuilder.Current.SetControllerFactory(new
ContainerControllerFactory(containerProvider))
```

Código Fuente 7-13

En el constructor de cada clase Controller, será necesario agregar un parámetro para recibir el contexto del tenant, que será inyectado por el Inyector de Dependencia

```
[HandleError]
publicclass HomeController : Controller
{
    ITenantContext  currentTenantContext;

public HomeController(ITenantContext tenantContext)
    {
        _currentTenantContext = tenantContext;
```

```
    }


public ActionResult Index()
    {
        ViewData["TenantName"] = _currentTenantContext.Name;

return View();
    }

public ActionResult About()
    {
return View();
    }
}
```

Código Fuente 7-14

Para ambas plataformas haciendo adaptaciones en los frameworks web que proveen se pueden implementar el mecanismo de configurabilidad de una aplicación multi-tenant. El uso de Namespaces en App Engine para diferenciar a los tenants supone una ventaja respecto de Windows Azure ya que están integrados con los servicios adicionales de App Engine como el uso de colas, url Fetch y MemCaché, por nombrar algunos, y no harán falta mecanismos externos para diferenciar el uso de los mismos.

ELECCIÓN DEL ESQUEMA DE BASE DE DATOS

En el capítulo anterior se mostró las diferentes opciones disponibles a la hora de diseñar un esquema de base de datos para una aplicación multi-tenant.

En resumen tenemos:

1. Una base de datos diferente por cada tenant
2. Una única base de datos, diferentes tablas para cada tenant
3. Una única base de datos, mismas tablas, diferentes registros diferenciados por un campo discriminador para cada tenant.

La elección en este caso dependerá de los siguientes factores:

- Las facilidades que ofrezca la plataforma para cada opción
- El costo monetario de cada opción
- Las restricciones que imponga la aplicación a desarrollar
- La complejidad que estemos dispuestos a trasladar el código de la aplicación

Comencemos el análisis desde lo que ofrecen las plataformas.

Google App Engine

En esta plataforma tenemos dos opciones, DataStore o Google Cloud SQL.

DataStore

Una de las restricciones de la aplicación es que la base de datos sea transaccional. DataStore es una base de datos No-SQL, y si bien trae algunas características técnicas para alojar una base de datos multi-tenant, tiene algunas restricciones, como no permitir joins y no permitir transacciones entre filas de más de una tabla.

De todas formas, no hay que descartar esta opción completamente, ya que puede ser un buen complemento para almacenar cierta información, y puede llegarse a implementar un mix junto con el motor de base de datos transaccional.

Google Cloud SQL

Como se mencionó en el capítulo 4, el servicio de Cloud SQL es proveer bases de datos MySQL. Dado que MySQL es un RDBMS completo, es ideal para la almacenar la información del Sistema de Alertas.

Windows Azure

En Windows Azure también tenemos dos opciones similares a las que ofrece App Engine, que son Azure Tables y SQL Azure Database.

Azure Tables

La opción de Azure Tables es también una especie de base Non-SQL, y tiene restricciones similares al servicio DataStore de App Engine, como no permitir joins y no permitir transacciones entre filas de más de una tabla. Para el sistema de Alertas de Noticias esta opción también no alcanza a cubrir los requerimientos, pero tampoco hay que descartarla completamente, y al igual que con el servicio DataStore, tiene opciones de escalabilidad y performance que pueden ser necesarias para implementar ciertos componentes de la aplicación.

SQL Azure Database

Este servicio cumple con todos los requisitos de un RDBMS, y como el servicio de Cloud SQL de App Engine cumple con los requerimientos que necesita el Sistema de Alertas.

Análisis de los diferentes esquemas

Opción una base de datos por tenant

Tanto el servicio SQL Azure Database como Google Cloud SQL permiten una cantidad limitada de bases de datos por cuenta, por lo que esta opción no sería escalable, ya que el Sistema de Alertas puede tener desde cientos a miles de clientes.

Además, dado que en el Sistema de Alertas, no se va a cobrar el uso físico de recursos, solo se cobrarán el uso de unidades funcionales del sistema, como cantidad de mails enviados, cantidad de medios a analizar, el objetivo del Sistema de Alertas es maximizar el uso de recursos de la plataforma, por lo que dedicar una base de datos para cada tenant incrementaría los costos significativamente.

Una única base de datos, diferentes tablas para cada tenant o una única base de datos, mismas tablas, diferentes filas diferenciadas por un campo.

En cuanto al costo monetario, ambas opciones son viables para el Sistema de Alertas ya que los servicios de Cloud SQL y SQL Azure Database calculan el costo de acuerdo al tamaño de la base de datos, y no por la cantidad de tablas o registros, por lo que tener diferentes tablas o diferentes filas para cada tenant no modifica el tamaño de la base de datos significativamente.

Tanto App Engine como Windows Azure ofrecen almacenamiento NoSQL, pero no satisface los requerimientos de ser una base de datos transaccional, que es la que necesita el Sistema de Alertas. El servicio de Cloud SQL de App Engine tiene todas las características de un RDBMS y es el que necesita el Sistema de Alertas para almacenar los datos. Windows Azure ofrece el servicio de SQL Azure Database como RDBMS y también cumple con los requisitos para almacenar la base de datos del Sistema de Alertas.

DISEÑO DE LA ESCALABILIDAD

La principal característica de escalabilidad que ofrecen las plataformas como servicio, es que las aplicaciones no se despliegan en servidores web directamente, sino que se empaquetan en contenedores virtuales, donde cada contenedor virtual está definido por una cierta cantidad de memoria disponible y cierto poder de procesamiento. Esto permite a la plataforma como servicio, crear muchas instancias del mismo contenedor de aplicación, y alojar estas instancias en diferentes servidores físicos, sin que la aplicación tenga noción del servidor real.

Por lo general las plataformas como servicio ofrecen contenedores para aplicaciones web y contenedores para tareas en background. El Sistema de Alertas va a necesitar ambos tipos de contenedores.

APP ENGINE

Contenedores Web

Los contendores web, son los contenedores por default de App Engine, llamados FrontEnd Instances. Con una cuenta en App Engine podemos crear más de una aplicación web, donde cada aplicación web puede tener múltiples instancias FrontEnd.

Una aplicación Web en App Engine se define por nombre y título, donde el nombre identifica a la aplicación como subdominio de appengine.google.com, por ejemplo:

Nombre: alertaspormail.appengine.google.com

Título: Sistema de Alertas

Cuando subimos una aplicación web, no se crea directamente una instancia para la aplicación, sino que la misma es almacenada en un repositorio del server desde donde puede ser instanciada dinámicamente cuando es necesario.

El instanciamiento se produce recién cuando un requerimiento es recibido desde uno de los front-end y siempre a través de un requerimiento HTTP.

App Engine inicia o detiene dinámicamente las diferentes instancias de acuerdo a la cantidad actual de requerimientos. Las aplicaciones no solo responden a la carga de trabajo sino también dependen del origen del requerimiento. Si la aplicación es internacional y por ejemplo es muy accedida desde Japón, App Engine inicia una instancia de la aplicación en el DataCenter de Japón o el DataCenter más cercano.

App Engine usa un algoritmo de planificación para decidir cuantas instancias son necesarias para atender la cantidad de tráfico que reciba una aplicación. Con cada request que la aplicación recibe, el planificador hace una decisión de si atiende el requerimiento con una de las instancias disponibles o si crea una nueva instancia para ese requerimiento. La decisión se toma en base a la disponibilidad de las instancias actuales, a cuán rápido la aplicación haya estado atendiendo los requerimientos, es decir, la latencia de la aplicación, y de cuánto tiempo toma arrancar e inicializar una nueva instancia de la aplicación para que empiece a tomar requerimientos.

El planificador también monitorea la inactividad de las instancias de la aplicación, si hay muchas instancias sin actividad, el planificador comienza a darlas de baja hasta que la aplicación alcanza una actividad promedio.

En la consola de administración de la aplicación, existe la posibilidad de ajustar manualmente las variables del planificador.

Lower Max Idle Instances: esta opción permite controlar el número máximo de instancias inactivas disponibles de la aplicación. Estableciendo este valor, se le está diciendo a App Engine que debe apagar cualquier instancia inactiva debajo de este valor, por ejemplo si

hay 4 instancias inactivas, y el valor de este parámetro es 3, App Engine debe apagar una instancia. También este valor se utiliza para indicarle a App Engine que debe iniciar una nueva instancia si la cantidad de instancias inactivas es menor a la actual, por ejemplo, si la cantidad de instancias inactivas es 2, App Engine debe crear una más para llegar a 3.

Raise Min Pending Latency: este parámetro le dice a App Engine que no cree nuevas instancias a menos que un requerimiento haya estado pendiente más del tiempo especificado en este parámetro.

Contenedores Background

Los contendores BackEnd no escalan en respuesta al volumen de requerimientos, sino que se especifica la cantidad de instancias para cada backend desde la consola de administración de App Engine.

Las instancias de tipo Backend están preparadas para ejecutar tareas de mayor duración y de mayor volumen de procesamiento.

WINDOWS AZURE

Contenedores WebEn Windows Azure los contenedores web se denominan Web Roles. Al igual que en App Engine, por cada cuenta o suscripción en Windows Azure, se puede crear múltiples aplicaciones.

Una aplicación se identifica en Windows Azure como un *Cloud Service*. Un *Cloud Service* puede tener asociado múltiples Web Roles, a su vez, en tiempo de ejecución cada Web Role puede tener múltiples instancias.

Como en App Engine, un Cloud Service se identifica con un nombre y un prefijo de url dentro del dominio cloudapp.net perteneciente a Windows Azure.

Prefijo Url: alertaspormail.cloudapp.net

Nombre: Sistema de Alertas

Utilizando los archivos de configuración correspondientes se puede indicar tanto el tamaño del Web Role, como la cantidad de instancias a utilizar.

Ejemplo del archivo ServiceDefinition.csdef que define el nombre del Web Role y el tamaño del mismo:

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="alertaspormail"xmlns="http://schemas.microsoft.com/Servi
ceHosting/2008/10/ServiceDefinition">
```

```
  <WebRole name="WebRoleAlertas" vmsize="Small">
    <Sites>
      <Site name="Web">
        <Bindings>
          <Binding name="Endpoint1" endpointName="Endpoint1" />
        </Bindings>
      </Site>
    </Sites>
    <Endpoints>
      <InputEndpoint name="Endpoint1" protocol="http" port="80" />
    </Endpoints>
    <Imports>
      <Import moduleName="Diagnostics" />
    </Imports>
  </WebRole>
</ServiceDefinition>
```

Ejemplo del archivo ServiceConfiguration.cscfg que se utiliza para indicar la cantidad de instancias del Web Role.

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceConfiguration serviceName="
alertaspormail"xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/Service
Configuration" osFamily="1"osVersion="*">
  <Role name="WebRoleAlertas">
    <Instances count="1" />
    <ConfigurationSettings>
      <Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"val
ue="UseDevelopmentStorage=true" />
    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

Contenedores Background

En Windows Azure los contenedores background se llaman Worker Roles. Al igual que en App Engine este contendor no responde a requerimientos http, sino que está preparado para ejecutar tareas asincrónicas en forma batch. Una aplicación puede tener uno o muchos Worker role y por cada Worker role puede haber múltiples instancias del mismo. Como en el Web Role, hay que diferenciar el Worker Role de las instancias del mismo. El Worker role es la definición estática de las tareas a realizar.

A diferencia de App Engine, Windows Azure no maneja escalabilidad automática por defecto, es decir que si se especificó que el web role utilizará solo 2 instancias, y por cuestiones de tráfico se incrementa la carga de trabajo, será necesario agregar más instancias desde la consola de administración. Esto es una clara desventaja respecto de App Engine.

De todas formas, utilizando las Windows Azure Management APIs que provee la plataforma, es posible monitorear la carga de trabajo y crear nuevas instancias de forma programática. Por supuesto que esto es un costo adicional de desarrollo.

Para aliviar el costo de desarrollo existen algunas herramientas de código libre y otras pagas, para facilitar la especificación de reglas de escalabilidad. La más popular de código libre es el "Autoscaling Application Block", conocido también por el nombre clave "Wasabi", que forma parte de los "Application Blocks" del framework "Enterprise Library", desarrollado por el grupo de Microsoft Pattern and Practices.

Este es un ejemplo de configuración xml del "Autoscaling Application Block"

```
<rules
xmlns=http://schemas.microsoft.com/practices/2011/entlib/autoscaling/rules
enabled="true">
<constraintRules>
<rulename="Default"description="Siempre Activa"
enabled="true"rank="1">
<actions>
<rangemin="2"max="5"target="WebRoleAlertas"/>
</actions>
</rule>
</constraintRules>
<reactiveRules>
<rulename="ScaleUp"description="Incrementa la cantidad de instancias"
enabled="true"rank="10">
<when>
<greateroperand="Avg_CPU_RoleA"than="80"/>
</when>
<actions>
<scaletarget="WebRoleAlertas"by="1"/>
</actions>
</rule>
</reactiveRules>
<operands>
<performanceCounteralias="Avg_CPU_RoleA"
performanceCounterName="\Processor(_Total)\% Processor Time"
aggregate="Average"source="WebRoleAlertas"timespan="00:45:00"/>
</operands>
</rules>
```

Código Fuente 7-15

Esta herramienta permite definir reglas fijas y reglas reactivas. En el ejemplo se define la regla fija de que como mínimo existan siempre activas como mínimo 2 instancias y como máximo 5 instancias. También se define una regla reactiva para que si el uso de CPU supera el 80 % se incremente en 1 la cantidad de instancias del Web Role.

IMPLEMENTACIÓN DE TAREAS EN BACKGROUND

En esta sección se muestran las diferentes alternativas para realizar tareas en background en ambas plataformas, detallando particularidades de diseño y de implementación.

APP ENGINE

Las tareas en background se pueden disparar ante un evento temporal o ante el cambio de una condición en el sistema.

Para disparar tareas en background utilizando un evento temporal, se pueden utilizar el servicio Cron Jobs.

Un *Cron Job* se define utilizando el archivo cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/url_target</url>
    <description>Una Descripcion</description>
    <schedule>qué día, a que hora, cada cuanto</schedule>
  </cron>
</cronentries>
```

Código Fuente 7-16

Cada *Cron Job*, apunta un Servlet determinado a través de la url. Los *Cron Job* se pueden utilizar tanto en las instancias backend como en las instancias front-end. No hay diferencia en el código de un Servlet entre una instancia FrontEnd o una instancia Backend. Como se mencionó en el capítulo anterior, la diferencia radica en el límite de capacidad de procesamiento de uno y de otro.

Un Backend se define utilizando el archivo Backend.xml

```
<backends>
```

```
<backend name="backend1">
<class>B1</class>
<instances>1</instances>
<options>
<dynamic>true</dynamic>
</options>
</backend>
</backends>
```

Código Fuente 7-17

Un backend puede ser dinámico o residente. Un Backend dinámico permanece inactivo, hasta que se activa ante un evento, como la ejecución de un Cron Job, y una vez que realiza la tarea, vuelve a estar inactivo. En cambio un Backend residente, una vez que se activa permanece activo, hasta que se desactiva utilizando la consola de administración.

El tipo de instancia Backend no está destinada a hacer una única tarea, sino que se pueden programar para que se ejecuten múltiples Jobs sobre la misma. Para ello solo es necesario que diferentes *Cron Job* apunten al mismo backend utilizando el tag *"target"*.

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/url_target1</url>
    <description>Una Descripcion</description>
    <schedule>qué día, a que hora, cada cuanto</schedule>
<target>backend1</target>
  </cron>

  <cron>
    <url>/url_target2</url>
    <description>Otra Descripcion</description>
    <schedule>qué día, a que hora, cada cuanto</schedule>
<target>backend1</target>
  </cron>
</cronentries>
```

Código Fuente 7-18

La cantidad de tareas a asignarle a un Backend dependerá de que éste no se sobrecargue de modo que no pueda atender todas las tareas. En tal caso, se pueden incrementar la cantidad de instancias para el mismo Backend, o bien agrupar lógicamente ciertas tareas en un Backend, y otras tareas en otro Backend.

En definitiva, la decisión de usar las instancias backend o FrontEnd para tareas en background siempre estará relacionada con el tiempo de procesamiento que lleve la tarea en cuestión.

Para tareas cortas, se puede utilizar las instancias de FrontEnd perfectamente, para tareas que lleven mayor tiempo de procesamiento, y mayor poder de procesamiento, sin dudas las instancias BackEnd serán las que hay que utilizar.

WINDOWS AZURE

Al igual que en App Engine, las tareas se pueden disparar ante un evento temporal o ante el cambio de una condición del sistema.

Para disparar eventos temporales en Windows Azure se puede utilizar el servicio Job Scheduller disponible desde la Consola de Administración, Figura 7-1.

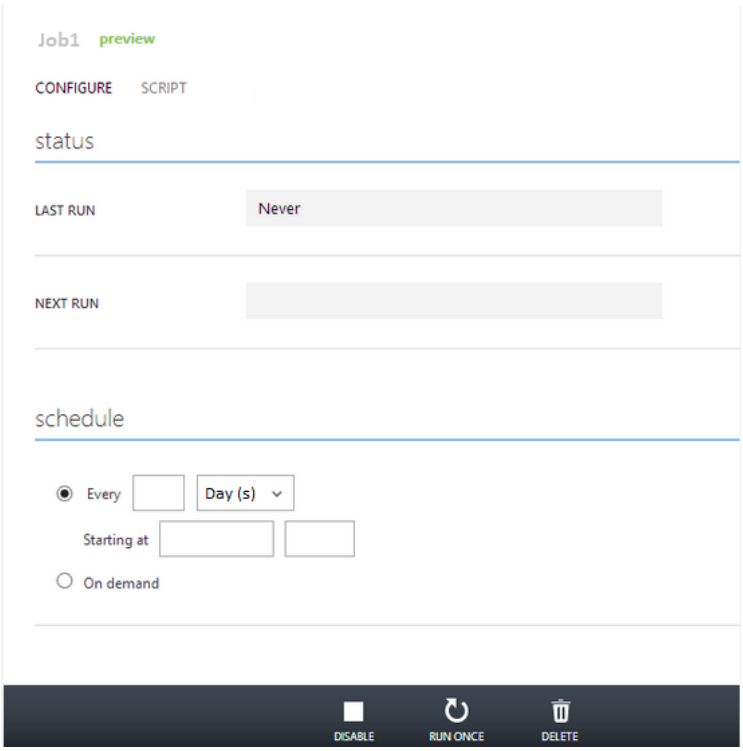

Figura 7-1 - Configuración de un job en Windows Azure

Cada *Cron Job* programado al llegar el momento de ejecutarse, invoca a un script asociado, el cual puede delegar tareas utilizando el servicio de colas de mensajes.

```
functionscript_job() {
var azure = require('azure');
var queueService = azure.createQueueService("myaccount", "mykey");
queueService.createQueueIfNotExists("queue", function(error){ });
queueService.createMessage("queue", "TareaParaHacer", function(error){});
}
```

Código Fuente 7-19

Para realizar las tareas lanzadas por el Job Scheduller podemos utilizar un Worker Role.

Para crear un Worker Role solo es necesario crear una clase que herede de *RoleEntryPoint*.

Esta clase tiene 3 métodos principales

OnStart: es un método que se ejecuta cuando se activa el Worker Role

Run: es el método principal donde se deberá programar el trabajo a realizar por el Worker Role

OnStop: es un método que se ejecuta cuando se desactiva el Worker Role.

Los 3 métodos describen el ciclo de vida de un *Worker Role*, es decir, un comienzo, algo que hacer, y el final. Un Worker Role por defecto no corre indefinidamente. Una vez que se termina de ejecutar el método Run, el Worker Role se desactiva. Para evitar esto, se utiliza un *while(true)* con una llamada al método *Sleep* para dormirlo por un tiempo determinado y volver a chequear luego si tiene trabajo para realizar.

Ejemplo de Worker Role:

```
publicclass WorkerRole : RoleEntryPoint
{
public override bool OnStart()
    {
return base.OnStart();
    }

public override void  Run()
    {
while (true)
        {
// Aca va el código a ejecutar de una tarea concreta.
//  .......
//
            Thread.Sleep(60000);
        }
    }

public override bool OnStop()
    {
return base.OnStop();
    }
```

```
}
```

Código Fuente 7-20

Si bien, el objetivo del *Worker Role* es análogo al tipo de instancias *Backend* de *App Engine*, tienen un modelo de activación y ejecución diferente, que obliga a programarlos de otra forma. El código del *Worker Role* no se activa ante requerimientos, sino que se activa al momento de subirlo junto con toda la aplicación, o bien se puede activar desde la consola de administración de Windows Azure una vez subido.

Se puede utilizar un tipo de *Worker Role* separado para cada tipo de tarea, donde cada instancia de *Worker Role* utilizará recursos dedicados, o bien, para mejorar la utilización de los recursos existe la posibilidad de utilizar el mismo *Worker Role* para múltiples tareas. Para poder dividir el trabajo en un *Worker Role*, la mejor opción es utilizar una cola de trabajo. Para realizar esto el *Worker Role* cada cierto intervalo de tiempo, por ejemplo cada 5 minutos, verifica si hay elementos en la cola trabajo, si lo hay, extrae el elemento, que por lo general es un *string* indicando el nombre de la tarea a realizar, y mediante un *switch*, se dispone a ejecutar la tarea correspondiente.

```
Public class WorkerRole : RoleEntryPoint
    {
Public override void Run()
        {
           var account =
CloudStorageAccount.Parse(ConfigurationManager.AppSettings["StorageConnection"]);
           var queue = account.CreateCloudQueueClient().GetQueueReference("queue");
           queue.CreateIfNotExists();

           while (true)
            {
               var msg = queue.GetMessage();
               if (msg != null)
                   {
                       switch (msg.AsString)
                       {
                          Case "xxx":

                          Case "yyy":

                          default:
                              break;
                    }
```

```
                }

                Thread.Sleep(10000);
            }
        }

        ...
    }
```

Código Fuente 7-21

La utilización de procesos en background facilita la realización de tareas asincrónicas, y permite aislar el comportamiento que más tiempo de CPU conlleva, para que la aplicación online tenga mejores tiempos de respuesta. Tanto App Engine como Windows Azure ofrecen estos mecanismos para ejecutar este tipo de tareas, junto con los mecanismos de sincronización, como las colas de trabajo.

En esta sección se analizaron diferentes cuestiones relacionadas con la arquitectura de la aplicación del Sistema de Alertas, que permiten entender la complejidad técnica de una solución SaaS *multi-tenant*. La implementación de la configurabilidad, la elección del esquema de la base de datos, el diseño de la escalabilidad y la implementación de tareas en background son los requisitos técnicos más importantes y los que primeros que se deben evaluar y analizar como paso previo a la construcción de una solución SaaS *multi-tenant.*

En la siguiente sección se analizarán diferentes cuestiones técnicas relacionadas con cierta funcionalidad crítica del Sistema de Alertas y cómo estas se pueden implementar sobre una plataforma como servicio.

LINEAMIENTOS PARA LA FUCIONALIDAD CRÍTICA DEL SISTEMA

En esta sección se darán lineamientos técnicos para resolver la funcionalidad crítica del sistema de alertas. Estos lineamientos técnicos permitirán minimizar los riesgos de construcción de la solución final, ya que se contará con alternativas de solución concretas. Estos requerimientos no están relacionados directamente con una solución SaaS pero son importantes de analizar ya que la aplicación estará alojada en una plataforma como servicio, y será necesario tener presente otras cuestiones que no se presentan al utilizar un servidor web tradicional.

Para mantener una trazabilidad con los requerimientos funcionales críticos del sistema de alertas, la Tabla 7-2 muestra la relación con las siguientes subsecciones.

Tabla 7-2 - Trazabilidad entre las secciones y los requisitos técnicos del sistema

Secciones	Requerimientos técnicos del sistema
Mecanismos de Extracción	Debe soportar la ejecución de procesos en background.
	Debe poder descargar páginas completas de los medios en internet.
Mecanismos de Full Text Search	Se debe utilizar un motor de full text search para la búsqueda de las noticias.
Envío de aleras a través de emails	Se debe contar con un servicio de envío de emails confiable.
Implementación de la facturación	Debe poder contabilizarse el uso de ciertas partes del sistema con el fin de poder realizar una facturación mensual.

MECANISMOS DE EXTRACCIÓN

El mecanismo de extracción se debe encargar de obtener el contenido de medios online haciendo peticiones http a los sitios web correspondientes. Además cada medio online tiene su frecuencia de actualización, por lo que este mecanismo se debe ejecutar de forma regular. Para esto es necesario que cada cierto intervalo de tiempo se acceda a la página principal del medio online y se comiencen a extraer las últimas noticias. Este proceso es independiente de cada tenant, por lo tanto no es necesario que la codificación sea *multi-tenant*.

APP ENGINE

En App Engine podemos utilizar el servicio URL Fecth para obtener el contenido de los medios online. Este servicio se utiliza con la librería estándar java.net para hacer peticiones http.

Dado que las aplicaciones alojadas en App Engine no pueden abrir sockets, cuando una aplicación realiza una petición http utilizando el servicio URL Fetch, este se redirige a servidores internos de Google que realizan la petición efectivamente.

Abajo se muestra un ejemplo de código de uso del servicio

```
import java.io.IOException;
import java.net.URL;
import java.util.List;

import com.google.appengine.api.urlfetch.HTTPHeader;
import com.google.appengine.api.urlfetch.HTTPResponse;
import com.google.appengine.api.urlfetch.URLFetchService;
import com.google.appengine.api.urlfetch.URLFetchServiceFactory;
```

```
public class WebPageFetcher
{
public void ObtenerPaginaWeb()
    {
        URLFetchService fetcher = URLFetchServiceFactory.getURLFetchService();
try {
            URL url = new URL("http://someurl.com");
            HTTPResponse response = fetcher.fetch(url);

byte[] content = response.getContent();
// if redirects are followed, this returns the final URL we are redirected to
            URL finalUrl = response.getFinalUrl();

// 200, 404, 500, etc
int responseCode = response.getResponseCode();
            List headers = response.getHeaders();

for(HTTPHeader header : headers) {
                String headerName = header.getName();
                String headerValue = header.getValue();
            }

        } catch (IOException e) {
// new URL throws MalformedUrlException, which is impossible for us here
        }
    }
}
```

Código Fuente 7-22

Se puede establecer un límite de tiempo para el requerimiento, es decir, la mayor cantidad de tiempo que el servicio esperará por una respuesta antes de arrojar una excepción. Por default, cada requerimiento tiene un límite de tiempo de 5 segundos. El máximo límite de tiempo es de 60 segundos y el límite de tamaño para las respuestas es de 32 MB

Para implementar la frecuencia de actualización de cada medio, una solución sería que al crear un nuevo medio en la base de datos, con su propia frecuencia de actualización, automáticamente se cree un Cron Job con la frecuencia especificada. Lamentablemente App Engine no ofrece una API para crear Cron Jobs desde las aplicaciones, sino que se crean en tiempo de diseño utilizando el archivo cron.xml y se activan al subirlo en App Engine.

Otra alternativa podría ser utilizar el servicio de Task Queues, pero este servicio solo permite indicar un retraso de *x* cantidad de tiempo para ejecutar una tarea, pero no permite encolar tareas para que se ejecuten en intervalos regulares, por ejemplo cada 2 horas.

La solución que adoptaremos es utilizar un Cron Job que se ejecute cada cierto intervalo de tiempo fijo, por ejemplo cada 10 minutos, y será tarea del Servlet asociado verificar si existen medios que tengan extracciones pendientes, de acuerdo a la última extracción y a la cantidad de tiempo transcurrido.

Definición del Servlet en el archivo web.xml

```
<servlet>
<servlet-name>ProcesoExtraccionServlet</servlet-name>
<servlet-class>com.example.alertasonline.cron.ProcesoExtraccionServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>ProcesoExtraccionServlet</servlet-name>
<url-pattern>/ProcesoExtraccion</url-pattern>
</servlet-mapping>
</servlet>
```

Código Fuente 7-23

Otra cosa que es necesario realizar es indicarle al proceso de clasificación de que hay nuevas notas para clasificar. Para ello se utilizará el servicio de Task Queues, por lo que luego de extraer las notas de los medios, se agregará un elemento en la cola ClasificacionQueue, para que lo procese el Servlet asociado.

Abajo se muestra la definición de la cola ClasificacionQueue en el archivo queues.xml

```
<queue-entries>
<queue>
<name>ClasificacionQueue</name>
<max-concurrent-requests>2</max-concurrent-requests>
</queue>

</queue-entries>
```

Código Fuente 7-24

Servlet que procesa las extracciones pendientes

```
package com.example.alertasonline;

import java.io.IOException;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
publicclassProcesoExtraccionServlet extends HttpServlet {
```

```
privatestaticfinal Logger _logger =
Logger.getLogger(ProcesoExtraccionServlet.class.getName());

publicvoid doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException
    {

try {
                if (extracionService.HayExtraccionesPendientes())
         {
                   extracionService.ProcesarExtraccionesPendientes();
this.dispararClasificacion();
               }
        }
catch (Exception ex) {
//Log any exceptions in your Cron Job
        }
    }

    @Override
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        doGet(req, resp);
    }

Public void dispararClasificacion(){
Queue queue = QueueFactory.getQueue("ClasificacionQueue");
TaskOptions taskOptions = TaskOptions.Builder.withUrl("/clasificacion")
                          .header("Host",
BackendServiceFactory.getBackendService().getBackendAddress("alertas backend"))
                          .method(Method.POST);
queue.add(taskOptions);
    }
}
```

Código Fuente 7-25

En el código de arriba, luego de procesar las extracciones pendientes, se invoca al método *dispararClasificacion(),* donde se obtiene la cola de tareas *ClasificacionQueue*, y se inserta un elemento para que sea procesado por el Servlet asociado a la Url *"/clasificacion"* y dentro del marco de la instancia del Backend *"alertas_backend". Esta implementación se verá en la siguiente sección.*

Para terminar la implementación del mecanismo de extracción, solo falta definir el Cron Job para que se ejecute por intervalos de 10 minutos.

Código del cron en el archive cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/ProcesoExtraccion </url>
    <description>Procesa las extraciones pendientes</description>
    <schedule>every 10 minutes</schedule>
  </cron>
</cronentries>
```

Código Fuente 7-26

WINDOWS AZURE

En Windows Azure no hay un servicio especial para realizar peticiones http a sitios de internet. Para realizar este tipo de peticiones se puede utilizar directamente la clase *WebRequest* de las librerías de .NET, a través del namespace *System.NET*.

Abajo se muestra un ejemplo de código de cómo utilizar la clase WebRequest

```
public class WebPageFetcher
{
public string ObtenerPaginaWeb()
    {
        HttpWebRequest req = WebRequest.Create(url) as HttpWebRequest;
        HTTPResponse response = fetcher.fetch(url);
        String result = null;
        using (HttpWebResponse resp = req.GetResponse() as HttpWebResponse)
        {
            StreamReader reader = new StreamReader(resp.GetResponseStream());
            result = reader.ReadToEnd();
        }
        return result;
    }
}
```

Código Fuente 7-27

En Windows Azure podemos utilizar el servicio Job Scheduling. Como en App Engine, en Windows Azure no se pueden crear Jobs programáticamente, sino que se crean en tiempo de diseño a través del portal web de la plataforma, por lo que en este caso se adoptará la solución de programar una tarea para que corra cada 10 minutos y utilizar un Worker Role para que verifique efectivamente si hay extracciones pendientes y ejecute la tarea de Extracción efectivamente.

En la Figura 7-2 *se muestra la creación del Job "Extracción" desde la consola de administración de Windows Azure:*

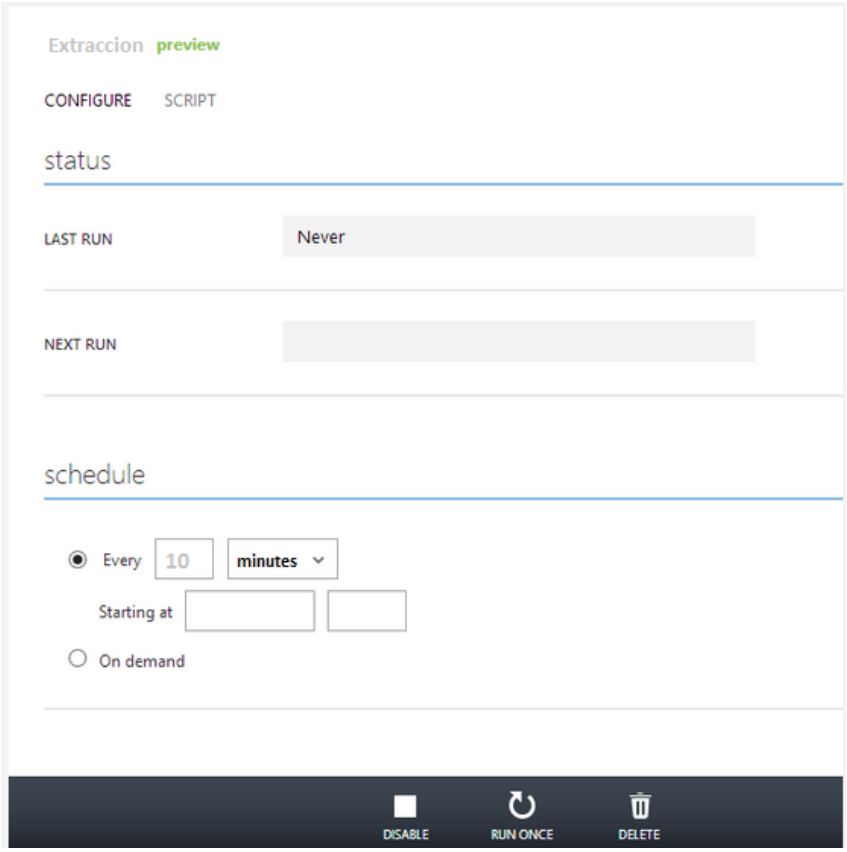

Figura 7-2 - Configuración del Job de Extracción

En el script asociado al Job, se utilizará una cola de trabajo para transferirle la tarea al Worker Role.

Este es el ejemplo del script del Job que agrega el elemento "Extracción" en la cola de trabajo:

```
Function extraccion() {
var azure = require('azure');
var queueService = azure.createQueueService("myaccount", "mykey");
queueService.createQueueIfNotExists("work", function(error){ });
queueService.createMessage("work", "Extraccion", function(error){});
}
```

Código Fuente 7-28

El Worker Role quedaría programado de esta forma:

```
Public class WorkerRole : RoleEntryPoint
    {
Public override void Run()
```

```
        {
            var account =
CloudStorageAccount.Parse(ConfigurationManager.AppSettings["StorageConnection"]);
             var queue = account.CreateCloudQueueClient().GetQueueReference("work");
             queue.CreateIfNotExists();

            while (true)
            {
                var msg = queue.GetMessage();
                if (msg != null)
                {
                    switch (msg.AsString)
                    {
                       Case "Extraccion":
                           If (ExtraccionService.ExtraccionesPendientes())
                           {
                              ExtraccionService.EjecutarExtraccion();
                              queue.DeleteMessage(msg);
                            }
                       default:
                            break;
                   }
               }

               Thread.Sleep(10000);
           }
       }

       ...
   }
```

Código Fuente 7-29

Tanto App Engine como Windows Azure proveen mecanismos para acceder al contenido de los medios online a través de APIs, lo que facilitará la tarea de extracción de la información de los medios online. Ambas plataformas, a través del servicio de tareas programadas, permiten implementar el mecanismo de extracción regular, que necesita el sistema para mantener actualizada su base de datos de noticias.

A continuación se analizará los mecanismos de full text search de ambas plataformas.

MECANISMOS DE FULL TEXT SEARCH

El proceso de clasificación se encarga de recorrer las notas extraídas que aún no fueron clasificadas, y de acuerdo a las pautas vigentes, clasificarlas como noticias relevantes.

Para realizar esta tarea es necesario contar con un motor de full text search para que las búsquedas en la base de datos sean óptimas y no tengan problemas de performance. Por lo tanto, es necesario identificar los diferentes motores de full text search que ofrecen los proveedores de cloud, y de qué forma se pueden utilizar para resolver los mecanismos de búsqueda.

APP ENGINE

El servicio Cloud SQL de App Engine al estar basado en MySQL, permite utilizar las capacidades de full text search de este motor.

Tanto las notas como las noticias, no están relacionadas con ningún tenant en particular, por lo que las tablas no necesitan el campo IdTenant para identificar el tenant. Las noticias se relacionan con un tenant al momento de informar una alerta.

Abajo se muestra un ejemplo de creación de la tabla Nota, con FullTextSearch en MySQL:

```
CREATETABLE Nota  (
  IdNota integer,
  Titulo text,
  TextoCompleto text,
IdExtraccion integer,
IdTema integer,
PRIMARYKEY RESULTS PK (IdNota),
  FULLTEXT (Titulo,TextoCompleto)
) ENGINE=MyISAM DEFAUTextoCompletoLT CHARSET=utf8;
```

Código Fuente 7-30

En la figura de abajo se muestra el código Java para obtener los Id de las notas que coinciden con el patrón de búsqueda:

```
public Iterable<int> searchIdsNotas(String query) {
  List <int> results = new ArrayList<int>();
  Connection conn = null;
try {
   DriverManager.registerDriver(new AppEngineDriver());
   conn = DriverManager.getConnection("jdbc:google:rdbms://[your db
instance]/search_values");
   String statement = "SELECT IdNota FROM Nota WHERE MATCH (Titulo,TextoCompleto) AGAINST (?
WITH QUERY EXPANSION);";
   PreparedStatement stmt = conn.prepareStatement(statement);
   stmt.setString(1, query);
   ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    results.add(rs.getInt(1));
   }
```

```
  } catch (SQLException e) {
   e.printStackTrace();
  } catch (EntityNotFoundException e) {
   e.printStackTrace();
  } finally {
if (conn != null)
try {
     conn.close();
    } catch (SQLException ignore) {}
  }
return results;
 }
```

Código Fuente 7-31

El proceso de clasificación se ejecutará desde un Servlet, y este será lanzado cuando se inserte un elemento en la cola ClasificacionQueue definida en el proceso de extracción.

Abajo se muestra la definición de la cola ClasificacionQueue en el archivo queues.xml

```
<queue-entries>
<queue>
<name>ClasificacionQueue</name>
<max-concurrent-requests>2</max-concurrent-requests>
</queue>

</queue-entries>
```

. Código Fuente 7-32

Definición del ServletClasificacionServlet en el archivo web.xml

```
<servlet>
<servlet-name>ClasificacionServlet</servlet-name>
<servlet-class>com.example.alertasonline.cron.ClasificacionServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>ClasificacionServlet</servlet-name>
<url-pattern>/clasificacion</url-pattern>
</servlet-mapping>
</servlet>
```

Código Fuente 7-33

El proceso de clasificación se debe ejecutar después de que haya ocurrido el proceso de extracción. Para ello, en App Engine, podemos utilizar una cola de trabajo, para disparar la ejecución del proceso de clasificación.

```
package com.example.alertasonline;

import java.io.IOException;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
publicclassClasificacionServlet extends HttpServlet {
privatestaticfinal Logger _logger = Logger.getLogger(ClasificacionServlet.class.getName());

publicvoid doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException
    {

try {

                    clasificacionService.clasificarNotas();

        }
catch (Exception ex) {
//Log any exceptions in your Cron Job
        }
    }

    @Override
publicvoiddoPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        doGet(req, resp);
    }

}
```

Código Fuente 7-34

Dado que el proceso de clasificación puede demorar más tiempo que el límite permitido de un request para una instancia FrontEnd, se utilizará el mismo backend definido para el módulo de facturación.

```
<backends>
<backend name="alertas_backend">
<class>B1</class>
<instances>1</instances>
<options>
<dynamic>true</dynamic>
</options>
</backend>
</backends>
```

Código Fuente 7-35

El enlace entre el módulo de extracción y el módulo de clasificación, queda en evidencia en el código que se ejecutará luego de hacer la extracción y que se transcribe abajo:

```
publicvoid dispararClasificacion(){
Queue queue = QueueFactory.getQueue("ClasificacionQueue");
TaskOptions taskOptions = TaskOptions.Builder.withUrl("/clasificacion")
                          .header("Host",
BackendServiceFactory.getBackendService().getBackendAddress("alertas backend"))
                          .method(Method.POST);
queue.add(taskOptions);
}
```

Código Fuente 7-36

WINDOWS AZURE

El servicio SQL Azure Database si bien comparte muchas de las características de la versión SQL Server, la capacidad de Full Text Search es una de las funcionalidades que no es compatible, aunque existen alternativas open source para poder implementar esta funcionalidad. Uno de los productos open source que más apoyo tienen de la comunidad es Lucene.NET.

Lucene.NET es la implementación en C# para .net del producto Lucene escrito en java. Lucene.NET no está integrado directamente con el motor de SQL Azure Database, sino que tiene una arquitectura propia para manejar Full Text Search, por lo que es necesario realizar un proceso de extracción de los datos de la tabla de la base de datos para insertarlos e indexarlos en los archivos que utiliza Lucene.NET.

Abajo se muestra el código que accede a la base de datos SQL Azure Database e indexa los datos en la base de datos de Lucene.NET

```
publicvoid IndexarNotas()
{
// Create the AzureDirectory against blob storage and create a catalog named 'Catalog'
    AzureDirectory azureDirectory= new
AzureDirectory(CloudStorageAccount.FromConfigurationSetting("Microsoft.WindowsAzure.Plugins.
Diagnostics.ConnectionString"), "Catalog");

    IndexWriter indexWriter = new IndexWriter(azureDirectory, new StandardAnalyzer(), true);
    indexWriter.SetRAMBufferSizeMB(10.0);
    indexWriter.SetUseCompoundFile(false);
    indexWriter.SetMaxMergeDocs(10000);
    indexWriter.SetMergeFactor(100);
```

```
// Create a DataSet and fill it from SQL Database
    DataSet ds = new DataSet();
using(SqlConnection sqlCon = new SqlConnection(sqlConnString))
    {
        sqlCon.Open();
        SqlCommand sqlCmd = new SqlCommand();
        sqlCmd.Connection = sqlCon;
        sqlCmd.CommandType = CommandType.Text;

        sqlCmd.CommandText = "select IdNota, TextoCompleto from Nota where indexada =
false";
        SqlDataAdapter sqlAdap = new SqlDataAdapter(sqlCmd);
        sqlAdap.Fill(ds);
    }
if (ds.Tables[0] != null)
    {
        DataTable dt = ds.Tables[0];
if (dt.Rows.Count > 0)
        {
foreach (DataRow dr in dt.Rows)
            {
// Create the Document object
                Document doc = new Document();
foreach (DataColumn dc in dt.Columns)
                {
// Populate the document with the column name and value from our query
                    doc.Add(new Field(
                           dc.ColumnName,
                           dr[dc.ColumnName].ToString(),
                           Field.Store.YES,
                           Field.Index.TOKENIZED));
                }
// Write the Document to the catalog
                indexWriter.AddDocument(doc);
            }
        }
    }
// Close the writer
    indexWriter.Close();

this.MarcarNotasIndexadas();
}
```

Código Fuente 7-37

Abajo se muestra el proceso de búsqueda de FullTextSearch de Lucene

```
public List<int> searchIdsNotas()
{
```

```
    List<int> result = new List<int>();
// Create the AzureDirectory for blob storage
    AzureDirectory azureDirectory = new
AzureDirectory(CloudStorageAccount.FromConfigurationSetting("Microsoft.WindowsAzure.Plugins.
Diagnostics.ConnectionString"), "Catalog");
// Create the IndexSearcher
    IndexSearcher indexSearcher = new IndexSearcher(azureDirectory);
// Create the QueryParser, setting the default search field to 'Bio'
    QueryParser parser = new QueryParser("TextoCompleto", new StandardAnalyzer());
// Create a query from the Parser
    Query query = parser.Parse(searchString);
// Retrieve matching hits
    Hits hits = indexSearcher.Search(query);
// Loop through the matching hits, retrieving the document
for (int i = 0; i < hits.Length(); i++)
    {
//Retrieve the string value of the 'Id' field from the
        result.add(hits.Doc(i).GetField("IdNota").IntValue());
    }
}
```

Código Fuente 7-38

Tanto App Engine como Windows Azure ofrecen alternativas para implementar Full Text Search. App Engine tiene una ventaja que al usar las características de Full Text Search de MySQL sin la necesidad de utilizar un producto aparte, como Windows Azure con Lucene, lo que puede facilitar el desarrollo de esta funcionalidad y el mantenimiento a futuro.

En la siguiente subsección se analizarán los mecanismos para enviar alertas a través de emails.

ENVIO DE ALERTAS A TRAVÉS DE EMAILS

El envío de alertas utilizando mails es fundamental para el Sistema de Alertas, por lo que debe ser un servicio 100% fiable. Además se espera un volumen de emails por día y mensual considerable, por lo que contar con un servicio de email escalable es una característica importante. Cada configuración de alerta define la periodicidad con la que se enviarán las alertas. Este comportamiento se puede implementar utilizando las tareas en background que ofrecen las plataformas. Ahora bien, al igual que en el módulo de extracción, cada tenant puede definir una cantidad indeterminada de alertas con diferentes periodicidades, por lo que utilizar los mecanismos de planificación de las plataformas no es conveniente. En su lugar se puede planificar que una tarea corra cada cierto intervalo de tiempo fijo, por ejemplo cada 5 minutos, y verifique si hay alertas pendientes para enviar.

Para verificar si hay alertas que enviar, solo es necesario verificar la fecha y hora de la última corrida con la fecha y hora actual, y determinar si el tiempo transcurrido supera el valor especificado en la periodicidad.

Dado que las noticias ya están preseleccionadas de acuerdo a las pautas establecidas por cada cliente, el envío de alertas solo tiene que recorrer aquellas noticias aún no enviadas, por lo que esta tarea no consume mucho tiempo de procesamiento, luego confeccionar el cuerpo del mail con las noticias correspondientes, y enviar los mails a los destinatarios de la alerta.

APP ENGINE

El servicio de mail utilizado por App Engine es Gmail, que además de ser un servicio de mail personal, forma parte de la suite de Google Apps for Business, en donde se puede utilizar Gmail como servidor de mail corporativo. El servicio de Gmail corporativo permite asociar un dominio propio y tiene un límite de cuota de uso más alto que la versión personal, que va desde un límite gratuito de 100 emails por día, y con la versión paga, se puede escalar hasta 20000 emails por día.

El servicio de email de App Engine soporta la librería JavaMail (javax.mail) para enviar mails.

La cuenta del remitente del mensaje debe ser un mail registrado y validado en la cuenta de App Engine y debe ser una cuenta registrada en el servicio de Google Apps.

Abajo se muestra un ejemplo en Java, de uso de la librería JavaMail:

```
import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

// ...
      Properties props =newProperties();
      Session session =Session.getDefaultInstance(props,null);

      String msgBody ="...";

      try{
         Message msg =newMimeMessage(session);
         msg.setFrom(newInternetAddress("admin@example.com","Example.com Admin"));
         msg.addRecipient(Message.RecipientType.TO,
                  newInternetAddress("user@example.com","Mr. User"));
```

```
        msg.setSubject("Your Example.com account has been activated");
        msg.setText(msgBody);
        Transport.send(msg);

    }catch(AddressException e){
        // ...
    }catch(MessagingException e){
        // ...
    }
```

Código Fuente 7-39

El proceso de alertas será llevado a cabo por un Servlet, el cual será llamado desde un Cron Job que se ejecutará cada 10 minutos.

Configuración del Servlet en el archivo web.xml

```
<servlet>
<servlet-name>EnvioAlertasServlet</servlet-name>
<servlet-class>com.example.alertasonline.cron.EnvioAlertasServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>EnvioAlertasServlet</servlet-name>
<url-pattern>/VerificarYEnviarAlertas</url-pattern>
</servlet-mapping>
</servlet>
```

Código Fuente 7-40

Código del Servlet que llama a los métodos correspondientes para chequear si hay alertas para enviar y para enviarlas efectivamente

```
package com.example.alertasonline;

import java.io.IOException;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
publicclass EnvioAlertasServlet extends HttpServlet {
privatestaticfinal Logger _logger = Logger.getLogger(EnvioAlertasServlet.class.getName());

publicvoid doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException
    {
```

```
try {
                if (envioAlertasService.HayAlertas())
          {
                   envioAlertasService.ProcesarAlertasPendientes();
                 }
        }
catch (Exception ex) {
//Log any exceptions in your Cron Job
        }
    }

    @Override
publicvoiddoPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        doGet(req, resp);
    }
}
```

Código Fuente 7-41

Código del cron en el archive cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/VerificarYEnviarAlertas</url>
    <description> Verifica si hay alertas para enviar, y las envía</description>
    <schedule>every 10 minutes</schedule>
  </cron>
</cronentries>
```

Código Fuente 7-42

En este Cron Job no se especifica el target, por lo que la ejecución del Servlet la llevará a cabo una instancia del tipo FrontEnd.

WINDOWS AZURE

Windows Azure no dispone de un servicio de email propio como App Engine. Si ya se posee una cuenta en algún servidor externo de Web Hosting, que ofrezca un servidor SMTP, desde Windows Azure se pueden enviar email utilizando este protocolo. Enviar emails utilizando un servidor SMTP de un proveedor externo a Windows Azure, no requiere ningún tipo de configuración especial de ninguna de las partes, solo basta con usar el puerto 25, ya que Windows Azure, al contrario de App Engine, tiene abiertos todos los puertos de salida.

El siguiente es un ejemplo de código en C# de envío de email básico utilizando una cuenta SMTP:

```
SmtpClient MySMTPClient;
MailMessage myEmail;
MySMTPClient = new SmtpClient("smtp.secureserver.net", 25);
MySMTPClient.Credentials = new NetworkCredential("<MailID>", "<Password>");
myEmail = new MailMessage(new MailAddress("<sender>"), new MailAddress("<receiver>"));
myEmail.Body = "Email from Windows Azure Application";
myEmail.Subject = "Email from Windows Azure";
try
{
  MySMTPClient.Send(myEmail);
}
catch (Exception ex)
{
// Display Exception Details
}
```

Código Fuente 7-43

Ahora bien, los servidores SMTP tradicionales que ofrecen los Web Hosting no siempre son fiables, y poseen una infraestructura fija que no es capaz de escalar para un gran volumen de emails. Además, por lo general las cuentas SMTP se obtienen como parte del paquete de Web Hosting, lo que encarece el servicio si solo se necesita la cuenta SMTP.

La otra opción es que el proveedor SaaS tenga un servidor SMTP en las propias instalaciones, pero tiene que encargarse él mismo de garantizar performance, escalabilidad y confiabilidad, con el costo adicional que eso conlleva.

Windows Azure tiene un convenio con el servicio de email basado en la nube llamado SendGrid, que provee entrega confiable de emails, escalabilidad y análisis de métricas en tiempo real. Este servicio provee APIs flexibles que facilitan la integración con las aplicaciones alojadas en Windows Azure.

La principal ventaja de utilizar un servicio como SendGrid es que el costo es muy bajo, y hasta hay planes gratuitos con un cupo de envío de email por día y por mes.

Con el servicio de SendGrid tenemos la posibilidad de enviar desde 100 emails hasta millones de emails por mes, con la garantía de confiabilidad, escalabilidad y performance.

SendGrid permite utilizar el protocolo SMPT o una Web API, basada en REST.

Ejemplo de uso de la librería de SendGrid para C# utilizando la Web API

```
// Create the email object first, then add the properties.
```

```
SendGrid myMessage =SendGrid.GenerateInstance();
myMessage.AddTo("anna@contoso.com");
myMessage.From=newMailAddress("john@contoso.com","John Smith");
myMessage.Subject="Testing the SendGrid Library";
myMessage.Text="Hello World!";

// Create credentials, specifying your user name and password.
var credentials =newNetworkCredential("username","password");

// Create an REST transport for sending email.
var transportREST = REST.GetInstance(credentials);

// Send the email.
transportREST.Deliver(myMessage);
```

Código Fuente 7-44

Ejemplo de uso de la librería de SendGrid para C# utilizando SMTP

```
// Create the email object first, then add the properties.
SendGrid myMessage =SendGrid.GenerateInstance();
myMessage.AddTo("anna@contoso.com");
myMessage.From=newMailAddress("john@contoso.com","John Smith");
myMessage.Subject="Testing the SendGrid Library";
myMessage.Text="Hello World!";

// Create credentials, specifying your user name and password.
var credentials =newNetworkCredential("username","password");

// Create an SMTP transport for sending email.
var transportSMTP = SMTP.GenerateInstance(credentials);

// Send the email.
transportSMTP.Deliver(myMessage);
```

Código Fuente 7-45

Abajo se muestra un ejemplo de Windows Azure utilizando el mismo Worker Role que para el Proceso de Extracción:

```
Public class WorkerRole : RoleEntryPoint
    {
        public override void Run()
        {
            while (true)
            {
                   If (envioAlertasService.HayAlertas())
                    {
                            envioAlertasService.ProcesarAlertasPendientes();
                    }

                Thread.Sleep(30000);
            }
        }

        ...
    }
```

Código Fuente 7-46

Como se comentó en la sección anterior, el Worker Role utiliza un while (true) para iterar indefinidamente y así poder chequear si tiene tareas por realizar. Sin este while(true) el Worker Role haría solo un chequeo la primera vez, y se terminaría la instancia.

Tanto App Engine como Windows Azure disponen de un servicio de email confiable y escalable. En App Engine, la utilización de un Cron Job permite que se puedan enviar los mails de forma periódica. En Windows Azure

En la siguiente subsección se analizará una posible implementación para la facturación del sistema.

IMPLEMENTACIÓN DE LA FACTURACIÓN

Este módulo se encargará de calcular la facturación mensual de cada cliente de acuerdo al consumo.

El Sistema de Alertas manejará el siguiente esquema de facturación

Tabla 7-3 - Esquema de facturación del Sistema de Alertas

Plan	Cantidad de Emails de Alertas	Cantidad de medios	Tipo de Precio
Básico	Hasta 100 emails	Hasta 3 medios	Precio mensual
Profesional	Hasta 500 emails	Hasta 6 medios	Precio mensual
Libre	Libre	Libre	Precio por unidad

Para el plan libre, se cobrará un precio por unidad de cada mail enviado y un precio por unidad de cada medio analizado.

Para cada plan puede elegir la opción de si utiliza la búsqueda simple o la búsqueda avanzada. Utilizar la búsqueda avanzada tiene un costo extra, y puede cambiarse de un mes para el siguiente.

El módulo de Ejecución de Alertas será el encargado de contabilizar la cantidad de emails enviados y de ir actualizando la tabla de Facturación Mensual.

El cálculo del total a pagar por el cliente se hará una vez por mes, los días 27. Para los que tienen el plan básico y profesional no se necesitará calcular nada, puesto que el precio es fijo, para los que tienen el plan libre, será necesario hacer la cuenta del precio total.

Para implementarlo podemos hacer uso de las tareas programadas que ofrecen tanto App Engine como Azure.

Es importante destacar que este proceso no necesita aislar los diferentes tenants, ya que se realizará el cálculo para todos por igual.

APP ENGINE

En App Engine podemos dejar programado un Cron Job para que se ejecute todos los 27 de cada mes.

Definición del Servlet en el archivo web.xml

```
<servlet>
<servlet-name>FacturacionServlet</servlet-name>
<servlet-class>com.example.alertasonline.cron.FacturacionServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>FacturacionServlet</servlet-name>
<url-pattern>/calcularFacturacion</url-pattern>
</servlet-mapping>
</servlet>
```

Código Fuente 7-47

Definición del Cron Job en el archivo cron.xml

La url apunta a la dirección del Servlet.

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/calcularFacturacion</url>
    <description>Calcula la facturación mensual</description>
    <schedule>27 of month 01:00</schedule>
  </cron>
</cronentries>
```

Código Fuente 7-48

Código java del Servlet encargado de ejecutar la tarea de Facturación

```
package com.example.alertasonline;

import java.io.IOException;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
publicclass FacturacionServlet extends HttpServlet {
privatestaticfinal Logger _logger = Logger.getLogger(FacturacionServlet.class.getName());

publicvoid doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException
    {

try {
                FacturacionService.CalcularFacturacion();
        }
catch (Exception ex) {
//Log any exceptions in your Cron Job
        }
    }

    @Override
publicvoid doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        doGet(req, resp);
    }
}
```

Código Fuente 7-49

Los request lanzados a las instancias FrontEnd desde un Cron Job, tienen las mismas limitaciones que un request HTTP hecho desde un navegador, por lo que el proceso de facturación debe poder ejecutarse dentro de los 30 segundos asignados al request. Si el

volumen de facturación no es muy alto, probablemente este tiempo alcance perfectamente, aunque será necesario monitorear las tareas realizadas por este *Cron Job* para saber si no se está llegando al límite de tiempo.

Si la ejecución se está llevando mucho tiempo, podemos hacer que el Cron Job se ejecute en una instancia Backend.

Definición de una instancia de backend en el archivo Backend.xml
El nombre asignado al backend es muy importante, ya que es el que se utilizará para relacionarlo con el Cron Job

```
<backends>
<backend name="alertas_backend">
<class>B1</class>
<instances>1</instances>
<options>
<dynamic>true</dynamic>
</options>
</backend>
</backends>
```

Código Fuente 7-50

En el archivo cron.xml es necesario agregar el tag "target" para indicar que el Cron Job se ejecute en el Backend "alertas_backend"

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/calcularFacturacion</url>
    <description>Calcula la facturación mensual</description>
    <schedule>27 of month 01:00</schedule>
    <target>alertas_backend</target>
  </cron>
</cronentries>
```

Código Fuente 7-51

La configuración del Servlet en el web.xml y el código del Servlet quedan sin modificaciones.

En el archivo Backend.xml, se definió que el tipo de backend sea dinámico *<dynamic>true</dynamic>*, esto significa que la instancia estará inactiva y se activará al primer requerimiento, en nuestro caso, solo se activará los días 27 de cada mes. Una vez terminado la tarea, esta instancia volverá a estar inactiva. Esto supone un ahorro en la cantidad de horas de uso de la instancia de backend, ya que App Engine solo cobrará por la cantidad de horas activas.

En Windows Azure podemos utilizar el servicio Job Scheduling, que nos permite programar una tarea para que corra un día determinado, todos los meses y utilizar un Worker Role para que ejecute la tarea de Facturación.

En la Figura 7-3 *se muestra la creación del Job "Facturación" desde la consola de administración de Windows Azure:*

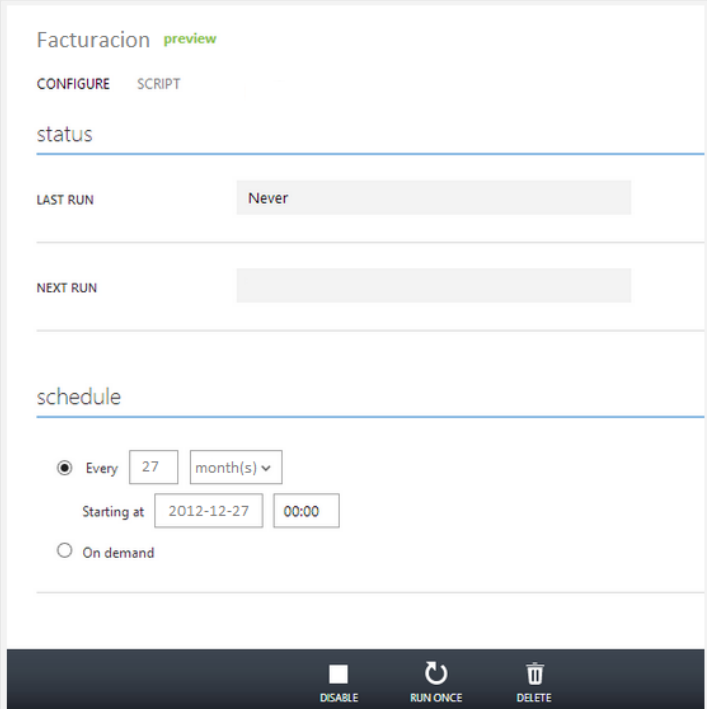

Figura 7-3 - Configuración del job de Facturación

En el script asociado al Job, se utilizará una cola de trabajo para transferirle la tarea al Worker Role.

Este es el ejemplo del script del Job que agrega el elemento "Facturacion" en la cola de trabajo:

```
functionfacturacion() {
var azure = require('azure');
var queueService = azure.createQueueService("myaccount", "mykey");
queueService.createQueueIfNotExists("work", function(error){ });
```

```
queueService.createMessage("work", "Facturacion ", function(error){});
}
```

Código Fuente 7-52

El Worker Role quedaría programado de esta forma:

```
Public class WorkerRole : RoleEntryPoint
    {
Public override void Run()
        {
           var account =
CloudStorageAccount.Parse(ConfigurationManager.AppSettings["StorageConnection"]);
             var queue = account.CreateCloudQueueClient().GetQueueReference("work");
             queue.CreateIfNotExists();

            while (true)
            {
                var msg = queue.GetMessage();
                if (msg != null)
                {
                    switch (msg.AsString)
                    {
                       Case "Facturacion":
                            FacturacionService.CalcularFacturacion();
                            queue.DeleteMessage(msg);
                            default:
                            break;
                    }
                }

                Thread.Sleep(10000);
            }
        }

        ...
    }
```

Código Fuente 7-53

En ambas plataformas es vital el uso de las tareas programadas, ya que la facturación debe ejecutarse una vez al mes, todos los meses. El uso de una instancia de tipo Backend en App Engine permite ejecutar la tarea de facturación sin restricciones de tiempo. El Windows Azure la combinación de un Job Scheduling, el uso de colas de

mensajes y un Worker Role permiten implementar la tarea de facturación sin inconvenientes.

CONCLUSIÓN

A lo largo de este capítulo se relacionaron los temas teóricos planteados en los capítulos anteriores, con un caso de estudio real y práctico, esbozando posibles soluciones de diseño de los principales características del sistema, sobre plataformas como servicio concretas y productivas, dejando en evidencia la complejidad inherente de una solución SaaS multi-tenant altamente escalable.

A continuación en la Tabla 7-4 se muestra un resumen de los puntos analizados en ambas plataformas.

Tabla 7-4 - Trazabilidad entre las secciones y los requisitos técnicos

	App Engine	**Windows Azure**
Implementación de la configurabilidad	Uso de Servlets en combinación el inyector de dependencia Google Guice y la API Namespaces de App Engine para identificar los tenants	Uso de ASP.NET MVC con modificaciones para soportar múltiples tenants.
Elección del esquema de base de datos	Se eligió el esquema de utilizar un campo discriminador por tabla y como motor de base de datos el servicio Cloud SQL basado en MySQL	Se eligió el esquema de utilizar un campo discriminador por tabla y como motor de base de datos el servicio SQL Azure Database basado en SQL Server.
Diseño de la escalabilidad	Uso de contenedores FrontEnd y Backend	Uso de Web Roles y Worker Roles
Implementación de tareas en background	Uso de Cron Jobs y Backend	Uso del servicio Job Schedduling y Worker Roles
Mecanismos de extracción	Uso del servicio Url Fecth y del servicio Cron Jobs	Uso de la clase WebRequest y del servicio Job Scheduling
Mecanismos de Full Text Search	Uso del mecanismo de full text search de Cloud SQL	Uso de la librería Lucene de Full Text Search
Envío de alertas a través de emails	Uso del servicio de Gmail a través de javax.mail y del servicio de Cron Jobs	Uso del servicio SendGrid y del servicio de Job Scheduling

Implementación de la facturación	Uso del servicio Cron Jobs	Uso del servicio de Job Scheduling

Ambas plataformas presentan diferentes soluciones para la construcción de la solución del Sistema de Alertas. Como resultado del análisis técnico quedó en evidencia que es necesario analizar cada punto crítico de la aplicación a construir, ya que cada plataforma como servicio tiene sus particularidades, que la diferencian entre ellas, y que las diferencian de un servidor web tradicional. La decisión final sobre qué plataforma construir el sistema de alertas dependerá de la experiencia del proveedor en cada lenguaje. Cualquier proveedor que se embarque en construir una solución SaaS sobre una plataforma como servicio, deberá analizar cuidadosamente las características técnicas del sistema a desarrollar, ya que las plataformas presentan nuevos desafíos técnicos que no se presentan en entornos locales o en servidores web tradicionales.

CAPÍTULO 8 CONCLUSIONES

APORTES PRINCIPALES DEL TRABAJO

Estimo que este trabajo ayudará a aquellos profesionales de la informática que se interesen en construir aplicaciones SaaS sobre plataformas como servicio, brindándole herramientas para tomar mejores decisiones.

Entre los aportes podemos mencionar:

- Se logró enmarcar el desarrollo de soluciones SaaS, diferenciándola de otro tipo de soluciones, identificando correctamente la necesidad de construir una solución de este tipo y sus principales beneficios.

- Se puso de manifiesto las principales características de diseño de una aplicación SaaS

- Se esbozaron las características técnicas necesarias que debe cumplir un proveedor de plataforma como servicio para alojar una aplicación SaaS

- Se enumeraron y explicaron las herramientas de desarrollo y codificación necesarios para construir una aplicación SaaS multi-tenant de calidad y con menor esfuerzo.

- Utilizando el caso de estudio propuesto se logró mostrar cómo ciertos aspectos de una aplicación, que en un entorno local y para un único cliente tienen una solución trivial, en un entorno cloud y multi-tenant necesitan un esfuerzo mayor de diseño y codificación.

- Utilizando dos proveedores de PaaS como Google App Engine y Windows Azure, se logró mostrar que ciertas cuestiones de diseño son independientes de la plataforma. También quedaron en evidencia las diferencias y similitudes de las plataformas, sus ventajas y desventajas, y las diferencias técnicas en cuanto a la implementación.

- Quedó en evidencia el gran esfuerzo de ingeniería y de diseño, que será necesario para construir una solución de este tipo, y que necesariamente deberá contar con recursos humanos muy bien calificados.

LINEAS FUTURAS DE INVESTIGACIÓN

La construcción de aplicaciones SaaS multi-tenant es un tema de investigación muy amplio, principalmente la característica *multi-tenant* es un tema en constante debate ya que atraviesa todos los componentes de una aplicación, y siempre existen mejores diseños para alcanzar un mayor grado de optimización.

Algunos temas que quedaron en el tintero para analizar más a fondo, a mi juicio son los siguientes:

- Uso de identidad federada en entornos *multi-tenant*. Tanto *Google App Engine* como *Windows Azure* ofrecen servicios de identidad federada, ahora bien, la implementación para una aplicación *multi-tenant*, donde cada tenant puede optar por diferentes mecanismos de autenticación y autorización, no es trivial.

- Plantear una arquitectura *SOA multi-tenant*. Diseñar una arquitectura *SOA multi-tenant* tampoco es trivial, ya que se necesita analizar cómo operaría en un entorno *multi-tenant* el motor *BPEL*, el *ESB*, la instanciación de los servicios, y muchas otras cuestiones más. En [13] se propone una arquitectura para lograr una aplicación *multi-tenant* utilizando SOA, en donde se discuten los problemas encontrados, decisiones de diseño y potenciales soluciones.

- Soporte nativo de *multi-tenant* en motores de bases de datos. Si bien, el concepto de sharding y federación de base de datos se puede utilizar para implementar multi-tenant a nivel de base de datos, existen otra serie de cuestiones que son necesarias contemplar. En [9] se hace una aproximación al tema.

CONCLUSIÓN FINAL

La construcción de una aplicación web para ofrecerla como un SaaS implica entender correctamente los conceptos de multi-tenant, configurabilidad y escalabilidad, ya que estas características requieren conocimientos avanzados de diseño y programación, que no se presentan en aplicaciones web tradicionales. Además, es necesario conocer a fondo la Plataforma como Servicio donde se alojará la aplicación, ya que estas plataformas tienen particularidades que las diferencian de los servidores web tradicionales.

A largo del desarrollo de esta tesina fui avanzando por diferentes etapas. Mi investigación inicial comenzó con la plataforma de Windows Azure, y sobre cómo desarrollar aplicaciones web para la misma, pero rápidamente apareció el concepto de *multi-tenant* y cómo este concepto permite construir aplicaciones como servicio maximizando la utilización de los recursos para reducir costos de mantenimiento y desarrollo y así poder ofrecer soluciones *SaaS* a un mejor precio para el cliente.

Si bien antes de cloud computing, y en particular de las plataformas como servicio, ya existían aplicaciones para un cliente final que se ofrecían como servicio, la unión de ambas tendencias era relativamente nueva, y representaba un desafío importante de investigación.

Los conceptos técnicos de *multi-tenant*, escalabilidad, configurabilidad y aprovisionamiento fueron los faros que guiaron esta investigación y los que le dieron forma a esta tesina y de los que aprendí muchísimo.

Para enriquecer el trabajo, me propuse también investigar una plataforma alternativa, como Google App Engine, de forma tal de poder mostrar que la problemática del diseño de este tipo de aplicaciones es independiente de la plataforma.

Ambas plataformas, junto con otras que están en el mercado, están constantemente ofreciendo actualizaciones y servicios nuevos, y hay una puja incesante por quién se convertirá en el proveedor por defecto. Esta tesina me ayudó a identificar los servicios necesarios y básicos que deben cumplir las mismas, y a estar atento a las mejoras de los servicios para así construir aplicaciones de mejor calidad y cumpliendo con las premisas de una aplicación como servicio.

El caso de estudio surgió de una necesidad concreta en mi entorno laboral, y me permitió aportar soluciones y recomendaciones concretas, ayudando a delinear la propuesta técnica del proyecto.

En lo profesional, el desarrollo de esta tesina enriqueció mi visión del desarrollo de software de altas prestaciones, consolidó conocimientos aprendidos en la facultad como el diseño orientado a objetos, el uso de patrones de diseño y de arquitectura, el diseño de base de datos, el uso de programación concurrente y paralela y muchas más.

Referencias Bibliográficas

[1] Ben Kepes "UNDERSTANDING The Cloud Computing Stack SaaS, Paas, IaaS". Diversity Limited 2011.

[2] Craig D Weissman, Steve Bobrowski, "The Design of the Force.com Multitenant Internet Application Development Platform".

[3] Eugene Ciurana."Developing with Google App Engine". First Press.

[4] Dan Sanderson. "Programming Google App Engine".O'Reilly Media.

[5] Information Technology Security Council (ITSC) and Physical Security Council. "*Cloud Computing and Software as a Service (SaaS) - An Overview for Security Professionals".Febrero 2010.*

[6] The Art of Service. "Cloud Computing Certification Kit - Software as a Service and Web Applications".

[7] Irene S. Harris y Zubair Ahmed*"An Open Multi-Tenant Architecture to Leverage SMEs".*

[8] Cor-Paul Bezemer y Andy Zaidman*"Multi-Tenant SaaS Applications - Maintenance Dream or Nightmare".*

[9] Oliver Schiller, Benjamin Schiller, Andreas Brodt, Bernhard Mitschang*"Native Support of Multi-tenancy in RDBMS for Software as a Service".*

[10] Hong Cai, Ke Zhang, MingJun Zhou, Wei Gong, JunJie Cai, XinSheng Mao*"An End-to-End Methodology and Toolkit for fine granularity SaaS-ization".*
*[11] "*Stefan Walraven, Eddy Truyen, and Wouter Joosen" *A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications.*

[12] Milinda Pathirage, Srinath Perera, Indika Kumara, Sanjiva Weerawarana*"A Multi-tenant Architecture for Business Process Executions".*

[13] Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, Paul Fremantle*"Multi-Tenant SOA Middleware for Cloud Computing".*

[14] SalesForce "The multitenant, metadata-driven architecture of Databasedotcom". Agosto 2011.

[15] Cor-Paul Bezemer, Andy Zaidman*"Challenges of Reengineering into Multi-Tenant SaaS Applications".*

[16] Michael Miller*"Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online".*

[17] Patterns & Practices *Group"Microsoft Application Architecture Guide 2da Edition" ISBN: 9780735627109.*

[18] Peter Mell, Timothy Grance "The NIST Definition of Cloud Computing".

[19] Ralph Mietzner, Tobias Unger, Robert Titze, Frank Leymann *"Combining Different Multi-Tenancy Patterns in Service-Oriented Applications".*

[20] PRADEEP KUMAR ARYA, V.VENKATESAKUMAR, S.PALANISWAMI*"Configurability in SaaS for an Electronic Contract Management Application".*

[21] Ralph Mietzner, Frank Leymann*"Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows andServices to Realize a Unified Provisioning Infrastructure for SaaS Applications".*

[22] Frederick Chong y Gianpaolo Carraro*"Architecture Strategies for Catching the Long Tail".*

[23] Rouven Krebs, Christof Momm y Samuel Kounev "Architectural Concerns in Multi-Tenant SaaS Applications".

[24] Arsalan Shahid, Muhammad Naeem Ahmed Khan "Object-Relational Mapping Framework to Enable Multi-Tenancy Attributes in SaaS Applications".

[25] Karthik Viswanathan "Right Engineering SaaS".

[26] David Carew – IT Architect IBM "Building multi-tenancy applications with IBM middleware".

[27] Nicolás Cardozo, Jorge Vallejos, Sebastian Günther"Context-oriented Programming for Customizable SaaS Applications".

*[28] Microsoft MSDN "*Federation Guidelines and Limitation" http://msdn.microsoft.com/en-us/library/windowsazure/hh597469.aspx. Ultimo Acceso: marzo 2013.