UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Framework para la automatización de las pruebas de integración en arquitecturas orientadas a servicios

Autores: Facundo Chambó
Director: Lic. Patricia Bazán
Asesor profesional: Lic. Juan Ramón Cortabitarte
Carrera: Licenciatura en Sistemas

Resumen

En arquitectureas orientadas a servicios, las pruebas de integración cumplen un papel muy importante ya que es necesario validar tanto los escenarios de prueba nuevos como los ya existentes para evitar que los nuevos desarrollos generen fallos en escenarios que ya funcionaban.
Estas pruebas pueden ser manuales o automáticas. El problema con las pruebas de integración automáticas es que requieren ser desarrolladas y no es posible comenzar con su desarrollo hasta que esté avanzado el desarrollo de la funcionalidad. Esto hace que la automatización retrase la puesta en producción del nuevo desarrollo, generándole un perjuicio al negocio.
Esta tesis propone un framework el cual permita automatizar las pruebas de integración de aplicaciones en arquitecturas orientadas a servicios sin necesidad de que la funcionalidad de la misma se haya comenzado a desarrollar y así reducir el tiempo de la puesta en producción.

Palabras Claves

Prueba, automatización, framework, SOA, HTTP, TDD

Conclusiones

En este trabajo se realizó un marco teórico con los lineamientos necesarios para implementar una solución que permite realizar pruebas de integracóin automatizadas.
Se validó el marco teórico implementándolo en la empresa Despegar.com, obteniendo resultados satisfactorios generando beneficios para la calidad del producto y agilidad para responder a las necesidades del negocio.

Trabajos Realizados

Como validación del marco teórico desarrollado, se implementó un framework sobre el protocolo HTTP que permite mejorar la velocidad del desarrollo de la automatización de las pruebas en una arquitectura orientada a servicios independizando las pruebas de la implementación de la funcionalidad.

Trabajos Futuros

Si bien esta tesina se centró en la implementación del framework sobre el protocolo HTTP, se deja abierta la posibilidad de que se implemente también en otros protocolos de comunicación, como por ejemplo SOAP.

Fecha de la presentación: MARZO 2015

Universidad Nacional de La Plata

Facultad de Informática

Tesina de Grado

Framework para la automatización de las pruebas de integración en aplicaciones orientadas a servicios

Director:
Lic. Patricia Bazán

Autor:
Facundo Chambó

Asesor Profesional:
Juan Ramón Cortabitarte

5 de marzo de 2015

Índice general

Índice de figuras

Índice de tablas

Abreviaciones

SOA	**S**ervice **O**riented **A**rquitecture
ESB	**E**nterprise **S**ervice **B**us
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
TDD	**T**est **D**riven **D**evelopment
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
JSON	**J**ava**S**cript **O**bject **N**otation
XML	e**X**tensible **M**arkup **L**anguage
HTML	**H**yper**T**ext **M**arkup **L**anguage
URI	**U**niform **R**esource **I**dentifier
URL	**U**niform **R**esource **L**ocator
URN	**U**niform **R**esource **N**ame
QA	**Q**uality **A**ssurance

Agradecimientos

Esta tesina realizada en la Facultad de Informática de la Universidad de La Plata es un esfuerzo en el cual, directa o indirectamente, participaron distintas personas opinando, corrigiendo, teniéndome paciencia, dándome ánimo, acompañando en los momentos de crisis y en los momentos de felicidad. Quiero aprovechar este apartado para agradecer a todas esas personas. En primer lugar, a mi directora, Lic. Patricia Bazán, mi más amplio agradecimiento por su paciencia ante mi inconsistencia, por su valiosa dirección y apoyo para seguir y llegar a la conclusión del mismo.

Mis agradecimientos a la gran colaboración de mi asesor profesional, Lic. Juan Ramón Cortabitarte, por su acompañamiento, sus importantes comentarios y correcciones a lo largo de la preparación de este trabajo.

A la Facultad de Informática de la Universidad de La Plata, por brindarme la formación y darme las herramientas necesarias para poder desempeñarme en un futuro como profesional.

A la empresa Despegar.com por darme la libertad necesaria para llevar adelante este trabajo dentro del ambiente laboral.

Desde luego, mis agradecimientos a mis padres y familiares porque me brindaron su apoyo moral y económico para seguir estudiando y lograr el objetivo trazado para un futuro mejor y ser orgullo para ellos y de toda la familia.

Quiero dedicarle un especial agradecimiento mi esposa por su paciencia y comprensión durante todos estos años de estudio.

Capítulo 1

Introducción

1.1. Motivación

Cada día más empresas desarrollan sus sistemas utilizando la Arquitectura Orientada a Servicios (SOA) y utilizan como metodología de desarrollo "Test Driven Development" (TDD). El primer concepto consiste en partir el software en pequeñas aplicaciones que exponen sus funcionalidades como servicios y a su vez consumen los servicios expuestos por otras aplicaciones para cumplir una funcionalidad mayor. La metodología TDD propone primero entender lo que se quiere desarrollar, luego crear pruebas unitarias y, por último, desarrollar la funcionalidad y utilizar las pruebas unitarias para validar el desarrollo. En ambientes SOA, las pruebas de integración cumplen un papel muy importante ya que es necesario validar tanto los escenarios de prueba nuevos como los ya existentes para evitar que los nuevos desarrollos generen fallos en escenarios que ya funcionaban. Estas pruebas pueden ser manuales o automáticas. En la Tabla 1.1 se listan las características de uno y otro método.

Tabla 1.1: Comparación de las pruebas manuales y automáticas

Pruebas Manuales	**Pruebas Automáticas**
No son confiables en pruebas que son repetitivas, debido a que se pueden cometer errores humanos.	Son confiables en pruebas que son repetitivas, ya que se ejecutan siempre de la misma forma.
El costo es menor que la prueba automática, pero una prueba manual es útil una sola vez ya que no es posible reutilizarla.	El costo de la automatización es mayor que la prueba manual, pero útil para las futuras pruebas de regresión en donde el código cambia frecuentemente.
Son útiles para pruebas donde se requiere una apreciación subjetiva.	Muchas veces no son útiles para probar interfaces gráficas. Por ejemplo, no pueden determinar la usabilidad de una interfaz.
Algunas tareas son difíciles de realizar manualmente. Por ejemplo, pruebas de muchas combinaciones de datos de entrada.	Las herramientas utilizadas para automatizar tienen limitaciones que hacen que determinados escenarios no puedan ser automatizados.

El costo inicial de automatizar las pruebas puede hacer que se retrase la puesta en producción de una nueva funcionalidad, generando un perjuicio al negocio aventajando a la competencia. Por

ejemplo, en fechas festivas donde aumenta la cantidad de clientes, una promoción o una nueva funcionalidad que no se activa en los ambientes productivos puede significar una gran ventaja para la competencia.

Por lo expuesto anteriormente, esta tesis propone un framework para automatizar las pruebas de integración de aplicaciones en ambientes SOA sin necesidad de que la funcionalidad de la misma se haya comenzado a desarrollar y así reducir el tiempo de la puesta en producción.

1.2. Introducción

Asegurar la calidad de una aplicación desarrollada sobre una arquitectura SOA requiere de al menos tres tipos de prueba [1]:

- *Pruebas Unitarias*: aseguran que cada componente trabaja correctamente asumiendo que la integración con otros servicios funciona correctamente. Esto se realiza utilizando mocks de todos los componentes externos (invocaciones a otros servicios, acceso a las bases de datos, etcétera). El objetivo de estas pruebas es probar de forma aislada componentes internos del servicio como métodos, clases o procedimientos.
- *Pruebas de Integración*: aseguran que la comunicación entre servicios funciona correctamente consumiendo la aplicación y analizando los resultados sin validar su comportamiento interno.
- *Pruebas de Aceptación*: aseguran que la aplicación cumple correctamente con los *objetivos para los cuales fue creada*[1]. Estos objetivos no siempre pueden ser probados de forma automática (o bien su implementación es muy costosa), por lo que es común ver que estas pruebas se realicen de forma manual. Este tipo de prueba se utiliza para validar el comportamiento de los servicios creados (o modificados) dentro del sistema completo en un escenario de prueba integral.

Esta tesina se concentrará en los últimos dos tipos de pruebas (Pruebas de Integración y Pruebas de Aceptación), resolviendo los siguientes problemas para asegurar la calidad de las aplicaciones sobre arquitecturas SOA y que se encuentran enunciados en [2]:

- Debido a que la lógica de la aplicación se encuentra distribuida en distintos componentes, *no existe un solo cliente que tenga un control completo sobre todos los componentes.*
- Una prueba funcional[2] adecuada requiere de la disponibilidad de todos los servicios involucrados, *lo cual no siempre es posible.*
- Según el dominio del sistema, es posible que no existan casos de prueba funcionales para todos los potenciales usos y combinaciones de servicios, porque generalmente estos son desconocidos al momento de diseñar e implementar la funcionalidad.

[1] También llamados *Criterios de Aceptación*

[2] A lo largo de este documento se utilizarán los términos prueba funcional y prueba de aceptación indistintamente como sinónimos.

Otras contribuciones de este trabajo son: obtener una solución que permita *agilizar el desarrollo de las pruebas* para poder lograr una rápida puesta en producción de la funcionalidad sin entorpecer la metodología de desarrollo utilizada por el equipo, y que la solución sea lo menos invasiva posible en el desarrollo de la funcionalidad. Esto es, evitar incluir código de rastreo de casos excepcionales como por ejemplo:

```
// código ejecutado en condiciones normales
if(mail == 'test@info.unlp.edu.ar'){
  //codigo que se ejecuta solo cuando se utiiza un mail de prueba
}
```

1.3. Estructura de la tesina

Esta tesina está organizada en siete capítulos, siendo este primero el correspondiente con la introducción y presentación del tema.

En el capítulo 2 se definen conceptos generales acerca de SOA y TDD y el contexto en el cual es aplicable la solución propuesta.

En el capítulo 3 se analizan algunos trabajos relacionados que fueron analizados para la concreción de este trabajo.

En el capítulo 4 se explica de forma detallada el framework propuesto y en el capítulo 5 se muestra un caso de estudio implementando dentro de la empresa Despegar.com.

En el capítulo 6 se presentan posibles líneas de trabajo futuro y finalmente en capítulo 7 se muestran las conclusiones a las cuales se ha arribado.

Capítulo 2

Conceptos generales

2.1. Introducción

El término SOA se utilizó por primera vez en 1996 cuando Roy Schulte y Yeffim V. Natiz lo definieron como "un estilo de computación multicapa que ayuda a las organizaciones a compartir lógica y datos entre múltiples aplicaciones y modos de uso". Pero con su uso y sus diferentes implementaciones, la definición de SOA se ha ido transformando. A lo largo de esta tesina se utilizará la siguiente definición [3]:

Arquitectura Orientada a Servicios Es un estilo arquitectónico para construir sistemas basados en interacciones de componentes autónomos, de grano grueso y bajo acoplamiento llamados servicios. Cada servicio expone procesos y comportamiento mediante contratos, los cuales están compuestos por mensajes en direcciones descubribles llamadas endpoints. Un servicio está regulado por políticas que son externas a él. Los contratos y mensajes son utilizados por componentes externos llamados consumidores de servicios.

Esto significa que el término SOA define componentes, relaciones y restricciones sobre cómo utilizar e interactuar con estos componentes. La Figura 2.1 ilustra la relación entre los componentes definidos por una Arquitectura Orientada a Servicios.

Las arquitecturas orientadas a servicios son utilizadas en ambientes con problemas que se pueden dividir en dos grandes categorías [4]:

- La incompatibilidad entre el negocio y la tecnología.
- Duplicación de funcionalidad y la existencia de procesos de negocio sin comunicación entre sí.

Hoy en día las organizaciones son tan dependientes de la tecnología, que es muy importante que la tecnología que provee esa información y que es utilizada para dar soporte a los procesos de negocio estén alineados con las necesidades de la organización.

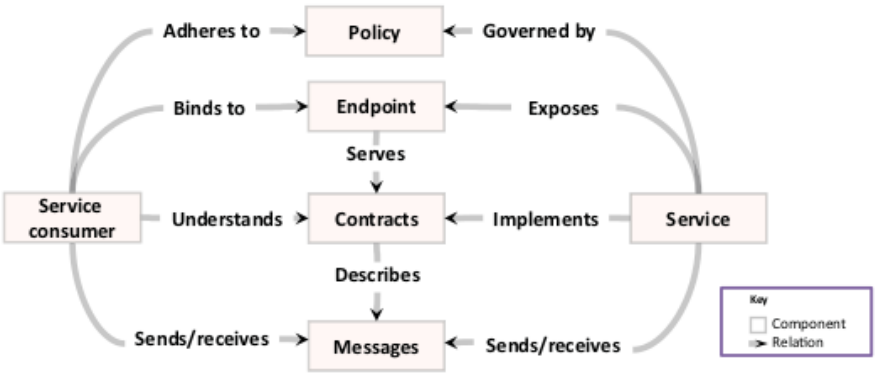

FIGURA 2.1: Diferentes componentes de una Arquitectura Orientada a Servicios.

Los síntomas más comúnmente encontrados de esta incompatibilidad entre la tecnología y el negocio son [4]:

- La tecnología no es capaz de cambiar al mismo ritmo que el negocio.
- La funcionalidad implementada no resuelve correctamente los problemas del negocio.

El otro gran problema es la duplicación de funcionalidad y la existencia de procesos de negocio sin comunicación entre sí. Tradicionalmente, las organizaciones están divididas funcionalmente en departamentos. Cada uno de estos departamentos tienen sus propios sistemas informáticos para tener un registro de la información que manejan. Esto último hace que la información se encuentre duplicada, lo cual genera inconsistencias, ya que la información no solo es almacenada, sino que también sufre modificaciones dentro de estos sistemas. Además de duplicarse la información, también se duplica aquella funcionalidad que es común en más de una unidad de negocio.

Aquellos departamentos que son autosuficientes y están aislados de otros departamentos son llamados silos organizacionales. Estos silos no solo provocan que exista información y funcionalidad duplicada, pero también ejecuciones ineficientes de los procesos de negocio. Esto se debe a que los procesos fueron creados según la división de la estructura de la organización y no teniendo en cuenta el proceso de negocio de punta a punta independientemente del departamento por el cual pasa en un momento determinado. [4]

Existen al menos tres grandes categorías de componentes que componen una arquitectura orientada a servicios [5]:

- *Componentes de infraestructura*: son elementos de software que permiten la correcta comunicación entre los diferentes componentes de la arquitectura. Los que se encuentran comúnmente son: ESB, firewall y registro de servicios.
- *Servicios*: consisten de aplicaciones pequeñas que exponen funcionalidad a través de uno o más protocolos de red. Explicado desde una visión más abstracta se puede definir como "algo útil que un proveedor hace para un consumidor" [4]
- *Procesos de negocio*: son aplicaciones que componen los servicios expuestos por otras aplicaciones para cumplir un objetivo determinado.

2.2. Enterprise Service Bus

Asumir que una red es homogénea es una de las falacias de los sistemas distribuidos[6]. Teniendo esto en cuenta, es conveniente encontrar la forma de que una aplicación no requiera implementar una forma de comunicación diferente para cada servicio que necesita consumir o cliente que requiera consumir sus servicios.

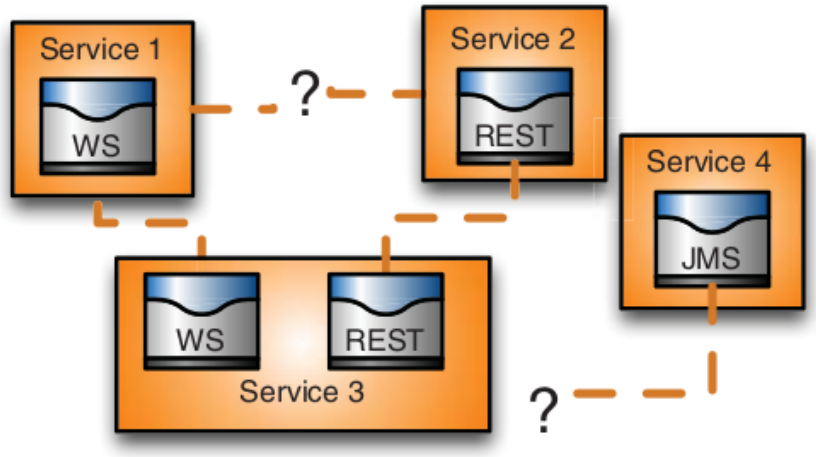

FIGURA 2.2: Situación en la que cada servicio expone y consume operaciones utilizando tecnologías y/o protocolos diferentes.

El ESB es un componente de infraestructura que desacopla las aplicaciones clientes de aquellos servicios que consume. La encapsulación de los servicios por el ESB significa que la aplicación cliente no necesita conocer nada acerca de la ubicación de los servicios o los diferentes protocolos de comunicación utilizados para invocarlos (ver Figura 2.3). El ESB brinda la posibilidad de compartir el uso de los servicios entre aplicaciones de todos los sectores de la organización e incluso entre organizaciones. [7]

El ESB procesa y redirige todas las invocaciones a servicios. Siempre que un componente necesita invocar un servicio de otro componente lo hace a través del ESB. Esto hace que cada componente sólo requiere desarrollar, probar y mantener la interfaz con el ESB, en lugar de integrarse de forma diferente con cada una de las aplicaciones que consumen.

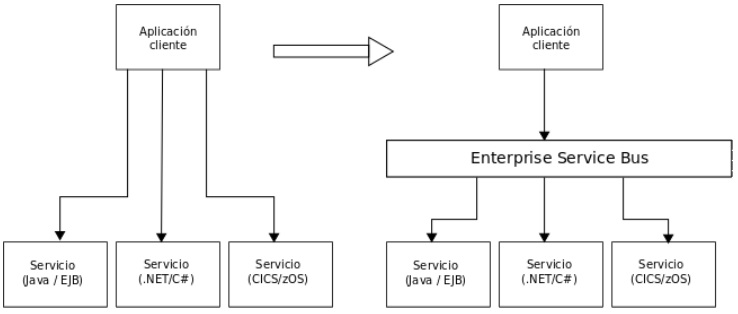

FIGURA 2.3: El ESB permite abstraer la comunicación entre aplicaciones clientes y servicios

Para poder implementar esta abstracción, el ESB provee funcionalidades especiales, ilustradas en la Figura 2.4, las cuales se describirán a continuación.

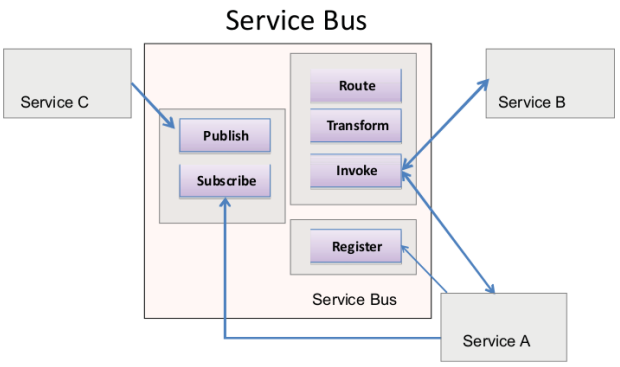

FIGURA 2.4: Funcionalidades comunes en un ESB.

- *Ruteo*: el ESB se encarga de hacer llegar la invocación al servicio de destino. De esta forma, el cliente se abstrae de la ubicación del servicio.
- *Transformación del mensaje*: el ESB provee la posibilidad de transformar el cuerpo o la metadata de los mensajes. Estas transformaciones pueden implicar una pequeña modificación o cambiar completamente el mensaje de un formato a otro (por ejemplo, pasar de JSON a XML, o convertir un XML en un HTML)
- *Invocación de servicios*: permite invocar servicios utilizando protocolos de red de la capa de transporte de forma sincrónica o asincrónica.
- *Registro de servicios*: el ESB es capaz de registrar servicios para que luego puedan ser invocados utilizando un nombre único.
- *Publicación/Subscripción de eventos*: el ESB permite intercambiar mensajes, habilitando a un servicio a subscribirse a los eventos que publican otros servicios.

2.3. Servicios

Construir una arquitectura orientada a servicios implica exponer la funcionalidad del negocio en interfaces claras y concretas. Estas interfaces son, para quienes las consumen, los servicios.

La definición más pura de un servicio es *"algo útil que un proveedor hace para sus consumidores"*. Un servicio puede ser algo tangible como la compra de una botella de leche, un auto, una nueva casa, pero también algo intangible como una asistencia al viajero o un tratamiento médico. Los servicios pueden ser simples o complejos y, además, pueden componerse de forma tal que la invocación de un servicio implique la invocación de otro u otros servicios más simples.

Todo servicio debe tener ciertas características para que esté bien definido y sea utilizable. Si un servicio no está bien definido puede no ser claro para los consumidores cuál es el valor agregado que ofrece, cuanto cuesta utilizarlo, o cómo utilizar el servicio. Esto puede llevar a confusiones

por parte del usuario e incluso a que los usuarios no lo utilicen. Los tres componentes esenciales de un servicio son [7]:

- *Contrato*: especifica qué puede esperar el consumidor de un servicio basándose en sus necesidades y lo que necesita ofrecer el proveedor del servicio. El contrato responde a preguntas como por ejemplo: ¿En qué horarios estará accesible el servicio? ¿Qué cantidad de tráfico soporta el servicio? ¿Tiene algún costo para el consumidor la utilización del servicio?
- *Interfaz*: define cómo se puede acceder y utilizar el servicio. Las interfaces describen las operaciones que el servicio es capaz de realizar, los datos de entrada, salida y las posibles condiciones de error que pudieran ocurrir.
- *Implementación*: es el desarrollo en un lenguaje de programación concreto (por ejemplo: Java, .Net, etc). La elección del lenguaje de programación en el cual se implementará el servicio es libre para el proveedor, no así la interfaz y el contrato, los cuales tienen que estar basados en protocolos y estándares globalmente aceptados para que el consumidor pueda utilizar el servicio.

Mientras que el contrato y la interfaz son visibles hacia el consumidor del servicio, la implementación permanece oculta. De esta forma, el proveedor del servicio puede realizar cambios en la implementación del servicio sin necesidad de que sus consumidores se vean afectados por los mismos, siempre y cuando no se modifique el contrato y la interfaz del mismo.

El proveedor y el consumidor del servicio no tienen, necesariamente, que pertenecer a la misma organización, ya que la utilización de estándares en la interfaz permite que diferentes aplicaciones puedan utilizar el servicio sin conocer su implementación. Esto será un punto muy importante para el framework propuesto en el Capítulo 4.

La reutilización de los servicios no es el objetivo de las arquitecturas orientadas a servicios, sino el medio por el cual se logran los objetivos. Los objetivos de los servicios y las arquitecturas orientadas a servicios son [4]:

- *Flexibilidad*: los servicios son pequeños "bloques de construcción" con un conjunto de capacidades claras y limitadas. Esto significa que es más fácil soportar los cambios de requerimientos de la organización.
- *Uso de estándares*: los servicios ocultan los detalles y complejidades técnicas utilizando estándares abiertos. Esto independiza a los consumidores del servicio de un proveedor o tecnología particular.
- *Reducción de costos*: Reutilizar servicios es más barato que construir, mantener y hospedar funcionalidad duplicada. Menor tiempo de salida al mercado: la reutilización de servicios permite desarrollar más rápidamente la funcionalidad, lo cual reduce el tiempo de salida al mercado.

- *Incremento de calidad*: debido a que más consumidores consumen el servicio, éste se encuentra mejor testeado. Además, no hay duplicidad de datos, por lo que la calidad de los datos también mejora.

2.4. Protocolos de Red

En esta sección se explicarán los protocolos de red más utilizados dentro de las arquitecturas orientadas a servicios: Hypertext Transfer Protocol y Simple Object Access Protocol

2.4.1. Hypertext Transfer Protocol (HTTP)

HTTP es un protocolo de aplicación sin estado utilizado comúnmente para aplicaciones cliente/-servidor. Una característica importante de HTTP es que está basado en transacciones, es decir que el cliente envía una petición al servidor y se queda esperando la respuesta.[8]

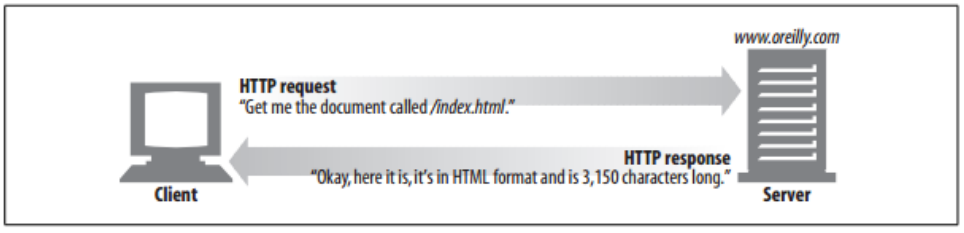

Figura 2.5: Comunicación HTTP entre cliente y servidor.

Cada acción HTTP que realiza un cliente, lo hace sobre un recurso determinado. Cada recurso tiene un nombre, de forma tal que los clientes sean capaces de hacer referencia a aquel recurso del cual están interesados. El nombre de un recurso es también llamado URI. Dada una URI, HTTP es capaz de identificar un recurso. Existen dos formas de expresar las URI: como URL o como URN. Las URN todavía están en un estado experimental y no han sido masivamente adoptadas como es el caso de las URL.

Una URL tiene tres partes:

- La primer parte es llamada esquema, y describe el protocolo utilizado para acceder al recurso.
- La segunda parte es indica el host y puerto del server donde se encuentra alojado el recurso.
- La parte que resta especifica el nombre del recurso en el servidor.

Las transacciones HTTP consisten de una petición HTTP invocada por el cliente y una respuesta recibida del servidor. Esta comunicación se realiza en bloques de datos llamados mensajes HTTP, como se ilustra en la Figura 2.7.

HTTP soporta diferentes comandos (también llamados métodos) para sus peticiones. Toda petición HTTP debe tener un método, éste le indica al servidor qué acción debe realizar sobre el recurso. La siguiente tabla grafica los métodos más utilizados por el protocolo HTTP.

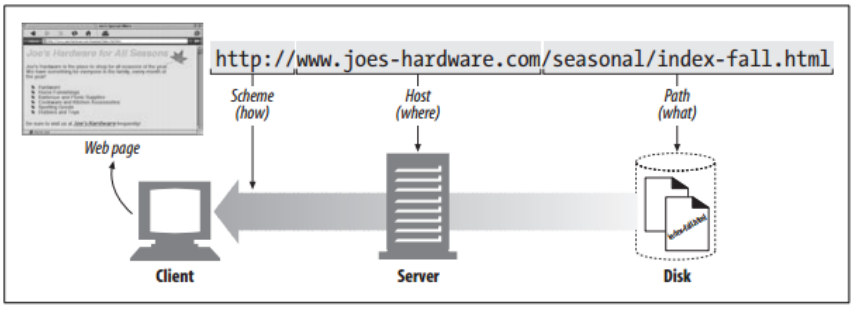

Figura 2.6: Composición de una URL.

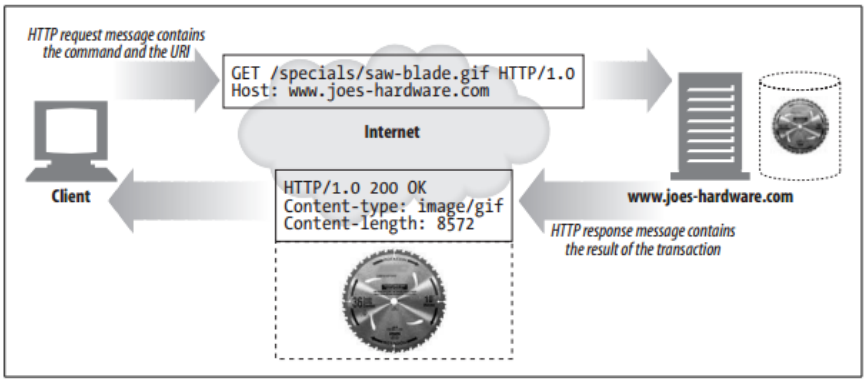

Figura 2.7: Descripción de una transacción HTTP.

Tabla 2.1: Descripción de los métodos HTTP

Método HTTP	Descripción
GET	Recupera el recurso y lo envía del servidor hacia el cliente.
PUT	Guarda los datos enviados por el cliente dentro del recurso existente en el servidor.
DELETE	Elimina el recurso del servidor.
POST	Envía los datos enviados por el cliente al servidor. Generalmente, esto implica que se cree un nuevo recurso.
HEAD	Devuelve únicamente el encabezado de la respuesta (sin el cuerpo del mensaje HTTP)

Cada respuesta HTTP viene acompañada por un código numérico de tres dígitos que representa el estado final de la transacción. Esta información es útil para el cliente para entender la respuesta y determinar la acción siguiente a realizar. Algunos de los códigos de estado más comunes son:

Tabla 2.2: Estados de respuesta HTTP

Código de estado	Descripción
200	OK
201	CREATED
404	NOT FOUND
403	FORBIDDEN
405	METHOD NOT SUPPORTED
500	INTERNAL ERROR

La estructura de los mensajes HTTP son simples de leer ya que consisten en líneas de texto plano. La Figura 2.8 muestra los mensajes HTTP involucrados en una transacción HTTP simple.

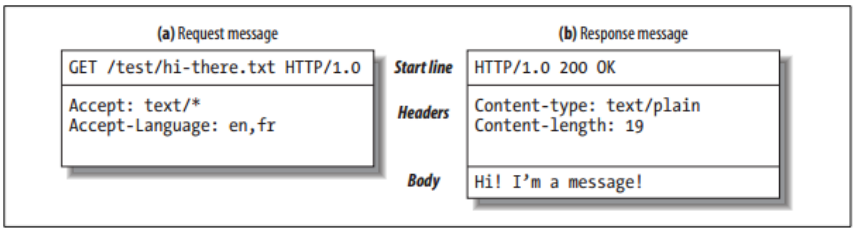

FIGURA 2.8: Mensajes HTTP involucrados en una transacción.

Los mensajes enviados del cliente hacia el servidor se llaman request messages y los mensajes retornados por el servidor devuelta al cliente son llamados response messages.

Los mensajes HTTP consisten de tres partes:

- *Línea de comienzo (Start line)*: La primer línea del mensaje indica qué tiene que hacer el servidor o cual fue el resultado de la acción que realizó el servidor.
- *Campos de encabezado (Header fields)*: A la primer línea le siguen cero o más campos de encabezado. Cada campo consiste de un nombre y un valor, separados por ':'. El encabezado termina con una línea en blanco. Es posible agregar encabezados simplemente agregando nuevas líneas. Esto es de mucha utilidad para enviar información de metadatos al servidor sin necesidad de modificar el cuerpo del mensaje.
- *Cuerpo (Body)*: luego de la línea en blanco que marca el fin del encabezado el mensaje continúa con un campo opcional que puede tener cualquier formato (imágenes, videos, audio, texto, etcétera).

2.4.2. Simple Object Access Protocol (SOAP)

SOAP es un protocolo de aplicación ya estandarizado en el desarrollo de web services que se utiliza para compartir mensajes entre aplicaciones. La especificación del protocolo no especifica más que un simple encabezado basado en XML para la información que va a ser transferida y un conjunto de reglas para traducir los tipos de datos específicos de las aplicaciones y la plataforma en formato XML. [9]

Debido a que el formato XML no está atado a ninguna aplicación, sistema operativo o lenguaje de programación, los mensajes XML se pueden utilizar en cualquier ambiente. Esta es la base sobre la cual se apoya la característica de interoperabilidad del protocolo. La idea fundamental es que dos aplicaciones, independientemente de la plataforma u otro detalle de sus implementaciones técnicas son capaces de compartir información con la simple transmisión de un mensaje encodeado de tal forma que ambas implementaciones puedan entenderlo. SOAP provee una forma estándar de estructurar los mensajes XML.

La estructura de un mensaje SOAP está formada por un contenedor (llamado SOAP envelope) que contiene el encabezado y el cuerpo del mensaje. El encabezado contiene bloques de información relevante que se utiliza para saber cómo se debe procesar el cuerpo del mensaje. Esta

FIGURA 2.9: SOAP provee una forma estándar de estructurar los mensajes XML.

información incluye configuración de ruteo y entrega, validaciones de autenticación y autorización y contextos de transacción. El cuerpo del mensaje contiene el mensaje real a ser entregado y procesado. Cualquier información que pueda ser expresada como XML puede ir en el cuerpo del mensaje.

FIGURA 2.10: Estructura de un mensaje SOAP.

La sintaxis XML para expresar un mensaje SOAP está basada en el espacio de nombres `http://www.w3.org/2001/06/soap-envelope`. Este espacio de nombres apunta a un esquema XML que define la estructura que debe tener un mensaje SOAP.

Según este esquema, un contenedor SOAP debe tener exactamente un elemento Body. El elemento Body puede tener todos los nodos que sean necesarios. El elemento Body está definido de tal forma que pueda contener cualquier XML válido y bien formado que haya sido definido por un espacio de nombres y que no contenga información de procesamiento del mensaje o referencias a definiciones de tipos de documentos (DTD). Si un contenedor SOAP contiene un elemento Header, no puede contener más de uno y tiene que aparecer como el primer hijo del elemento Envelope, antes del elemento Body. El encabezado, al igual que el cuerpo, puede contener cualquier XML válido, bien formado y cuyo espacio de nombres haya sido definido en el elemento Envelope.

El propósito de cada bloque de información que contiene el encabezado es comunicar información contextual relevante al procesamiento de mensaje SOAP. Por ejemplo, puede contener las credenciales de autenticación, o información de cómo rutear el mensaje.

Existen tipos de mensajes SOAP especiales que se utilizan para indicar errores o fallas al procesar el mensaje. La información contenida en un mensaje de falla SOAP está estructurada de la siguiente forma.

```
<s:Envelope
 xmlns:s="http://www.w3.org/2001/06/soap-envelope">
    <s:Header>
       <m:transaction xmlns:m="soap-transaction"
                      s:mustUnderstand="true">
           <transactionID>1234</transactionID>
       </m:transaction>
    </s:Header>
    <s:Body>
       <n:purchaseOrder xmlns:n="urn:OrderService">
          <from><person>Christopher Robin</person>
                <dept>Accounting</dept></from>
          <to><person>Pooh Bear</person>
               <dept>Honey</dept></to>
          <order><quantity>1</quantity>
                  <item>Pooh Stick</item></order>
       </n:purchaseOrder>
    </s:Body>
</s:Envelope>
```

FIGURA 2.11: Ejemplo de un mensaje SOAP.

```
<s:Envelope xmlns:s="...">
   <s:Body>
      <s:Fault>
       <faultcode>Client.Authentication</faultcode>
       <faultstring>
          Invalid credentials
       </faultstring>
       <faultactor>http://acme.com</faultactor>
       <details>
          <!-- application specific details -->
       </details>
      </s:Fault>
   </s:Body>
</s:Envelope>
```

FIGURA 2.12: Ejemplo de un mensaje de falla SOAP.

2.4.3. Test Driven Development (TDD)

TDD es una metodología para construir aplicaciones guiando el desarrollo mediante las pruebas. Es decir, que los escenarios de prueba deben ser escritos antes del código fuente [10].

Esta metodología fue desarrollada por Kent Beck en 1990, su esencia se basa en repetir continuamente estos tres simples pasos:

- Escribir una prueba para la próxima pequeña funcionalidad a desarrollar.
- Escribir el código fuente necesario para que la prueba sea exitosa.
- Modificar el código (tanto el nuevo como el previamente existente) para mantenerlo bien estructurado.

Se debe continuar iterando sobre estos tres pasos, de a una prueba a la vez, hasta que la funcionalidad del sistema quede completa.

Utilizando esta metodología es más probable que se creen pruebas para todo el código y funcionalidad que se desarrolle. Dirigiendo el desarrollo mediante pruebas automáticas, y luego eliminando código duplicado, cualquier desarrollador puede escribir código confiable y libre de errores sin importar su nivel de complejidad. TDD es más comúnmente aplicado a pruebas de unidad, pero también puede funcionar con pruebas de performance, integración o aceptación.

Para un proyecto en el cual se utilizan metodologías ágiles, el foco de las pruebas de integración se debe poner sobre las interfaces externas que no pertenecen al proyecto, es decir, la forma en la

que nuestro componente se va a comunicar con otros servicios. Las pruebas de integración deben ser automatizadas e incluidas en las compilaciones continuas del sistema, cualquier prueba que falle implica que la compilación debe fallar.

Con respecto a la utilización de TDD en las pruebas de aceptación, éstas se deben realizar a partir de los criterios de aceptación definidos por el analista de negocio. Una vez definidos los criterios de aceptación y sin necesidad de esperar a que finalice el desarrollo de la funcionalidad, se pueden escribir los escenarios de pruebas que los validen. Las pruebas de aceptación se deben ejecutar cuando se pasa de un ambiente a otro, en donde una falla en una prueba de aceptación implica que la funcionalidad no puede pasar al siguiente ambiente.

2.4.4. Conclusiones del capítulo

A lo largo de este capítulo se definieron conceptos generales que están estrechamente relacionados con la solución propuesta en los siguientes capítulos.

El framework propuesto como solución en el Capítulo 4 se aplica en un contexto de arquitecturas orientadas a servicios, por lo que es importante conocer las definiciones y características comunes que tienen estos tipos de arquitectura, ya que ayudará a contextualizar la aplicación del framework.

Del término SOA surgen otros conceptos relacionados como Enterprise Service Bus y Servicios. Éstos son una parte integral del framework: el primero por ser quien rutea las invocaciones y será capaz de diferenciar entre invocaciones reales o invocaciones de prueba; el último porque serán los servicios los componentes a probar durante las ejecuciones.

Luego se detallaron los protocolos de red más comúnmente utilizados en arquitecturas orientadas a servicios. Es importante conocer la estructura y detalles de comunicación de cada protocolo, ya que el framework se apoyará en estos para transmitir información vital para su correcta ejecución.

Por último, se expuso una metodología de desarrollo ágil que consta de escribir las pruebas antes de escribir el código fuente. La idea con la que fue creado el framework coincide con lo expuesto por esta metodología, ya que intenta quitar las barreras que existen para poder realizar la automatización de las pruebas en etapas tempranas del desarrollo a fin de paralelizarlos al máximo.

Capítulo 3

Trabajos previos relacionados

3.1. Introducción

En este capítulo se presentarán investigaciones previas que tienen una fuerte relación con el tema de esta tesina y en los cuales se apoyan algunas de las ideas del framework propuesto en el Capítulo 4.

3.2. Jia Zhang, 2004

En el año 2004, Jia Zhang presentó un paper [11] en el cual aborda el tema de probar la factibilidad de los servicios web utilizando agentes móviles para ejecutar escenarios de prueba y así elegir el mejor servicio web a consumir.

La solución presentada por Jia Zhang, se basa en una arquitectura de servicios web en la cual existen varios proveedores, un intermediario y un cliente. El cliente solicita al intermediario la ejecución de un proveedor para ejecutar un determinado servicio. El intermediario debe determinar cuál es el proveedor que puede responder al requerimiento de la forma más eficiente y efectiva posible.

Para encontrar este servicio, Zhang propone, enviar en un agente móvil a cada candidato para que ejecute una batería de pruebas y retorne los resultados para poder tomar la determinación.

Un agente móvil es un software autónomo y autocontenido que se puede mover de una máquina a la otra y actuar en nombre de usuarios u otras entidades. A fin de probar un servicio web remoto, un agente móvil es creado y movido al servicio web remoto, llevando consigo casos de prueba.

A continuación se explicará un escenario de ejecución de lo presentado por Zhang.

Paso 1: El cliente crea un agente móvil. Una base de datos está preparada para el agente móvil, la cual contiene dos conjuntos de información: los datos para la prueba y las validaciones.

Paso 2: El cliente envía el agente móvil al servicio web remoto.

Paso 3: El agente móvil ejecuta las distintas pruebas utilizando los datos para la prueba obtenidos de la base de datos para generar mensajes SOAP con los cuales probar el servicio. Dentro de las diferentes pruebas que realiza, busca generar fallas en servicio web.

Paso 4: El agente móvil va almacenando los resultados de sus pruebas en su base de datos. Si alguna de las pruebas falla, no considerará al servicio web confiable y lo eliminará de la lista de candidatos.

Paso 5: Cuando todas las pruebas finalizan, el agente móvil almacena los registros de los resultados en la base de datos y migra de vuelta hacia el cliente.

Esta tesis se apoya en el concepto planteado por Jia Zhang de que las pruebas y las validaciones no se ejecuten del lado del cliente, sino que se ejecuten de forma remota. Una diferencia importante con el framework anteriormente expuesto con el planteado en esta tesis es que en el último caso se trata de un ambiente controlado, mientras que Zhang presenta un contexto en el cual el servicio remoto puede estar en cualquier lugar de internet montado sobre una infraestructura desconocida por el cliente.

3.3. Hehui Liu, 2009

En el año 2009, el Laboratorio de Investigaciones de IBM de China presentó un framework en el cual aborda el problema de realizar pruebas de integración en ambientes de integración continua. [1]

El framework planteado consiste en 2 premisas bien claras, como se ilustra en la Figura 3.1 y la Figura 3.2.

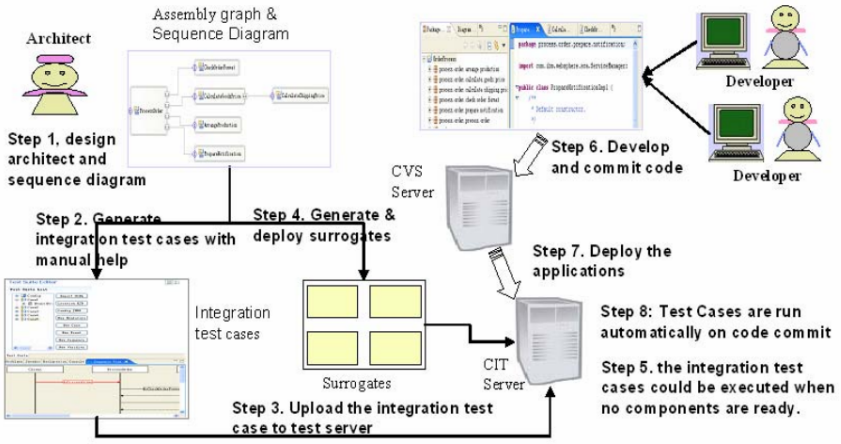

FIGURA 3.1: Premisa 1: Separar el desarrollo de las pruebas del desarrollo de la aplicación.

El framework presentado por Liu, propone que la aplicación que ejecuta las pruebas inyecte un identificador del escenario de prueba que permita rastrearlo durante toda la ejecución. Cada servidor de aplicaciones donde se ejecutan los servicios contiene un agente del framework que

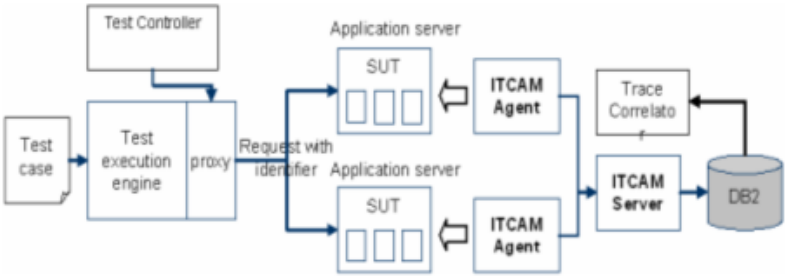

Figura 3.2: Premisa 2: Mantener una correlación de las pruebas con el escenario de prueba a pesar de que se ejecuten en servidores de aplicaciones diferentes.

intercepta los requerimientos que contienen el identificador y registra los resultados en la base de datos, manteniendo la correlación del escenario de prueba.

El framework propuesto en esta tesis se apoya fuertemente en la presentación del Laboratorio de Investigaciones de IBM de China para todo lo relacionado con el rastreo y la correlación de los escenarios de prueba. Pero una diferencia bien marcada que existe entre ambos frameworks es que el framework presentado en esta tesis no requiere que se instale ningún software en las máquinas que contienen los servidores de aplicaciones (que en ambientes a gran escala pueden ser muchas), sino que la correlación de los escenarios de prueba se realiza en un componente de infraestructura central como es el Enterprise Service Bus.

3.4. Conclusiones del capítulo

En la Tabla 3.1 se puede observar una comparación de las características de cada investigación junto con las del framework planteado en el siguiente capítulo.

Tabla 3.1: Características de los distintos frameworks

Característica	**Jia Zhang**	**Hehui Liu**	**TestIA**
Injección de datos de prueba	Los agentes móviles realizan las pruebas utilizando los datos almacenados en la base de datos.	El paper propone injectar un identificador del escenario de prueba para poder rastrear las ejecuciones, pero no especifica si es posible injectar datos en las invocaciones o respuestas.	Se pueden injectar datos que estan guardados en el repositorio en las invocaciones o en las respuestas de los servicios.
Resultados de las pruebas	Luego de la ejecución de los agentes móviles, los resultados se almacenan en la base de datos para poder ser consultados posteriormente.	Los resultados de las pruebas quedan guardados en la base de datos del lado servidor del framework	Los resultados de las pruebas pueden ser consultados desde el repositorio de resultados.
Escalabilidad	Cada agente móvil se ejecuta localmente en el servidor del servicio que se quiere probar.	El agente del framework que se ejecuta en cada servidor de aplicaciones depende de la plataforma donde se ejecute. Por lo que mientras más diversas sean las tecnologías del ambiente, más agentes diferentes habrá que implementar.	No se requieren más componentes del framework a medida que los servicios aumentan. La implementación es la misma independientemente de las tecnologías o plataformas en las que se ejecuten los servicios.

Luego del análisis y comparación de los trabajos previos relacionados, se puede concluir que existe un interés claro por encontrar una solución integral a las pruebas de integración de servicios en arquitecturas orientadas a servicios.

A diferencia de los trabajos previos, el framework propuesto plantea una mejora en términos de escalabilidad y flexibilidad. Escala mejor porque independientemente de la cantidad de servicios o servidores y la variedad de las plataformas en sobre las que se ejecute cada uno de ellos, la infraestructura necesaria se sigue manteniendo; lo único que sí se debe mantener es el protocolo de comunicación entre los servicios. Es más flexible, porque permite: injectar datos tanto en las invocaciones como en las respuestas, simular errores y manipular los tiempos de respuesta.

Una similitud de los tres frameworks es que ninguno requiere modificaciones del código de la funcionalidad que se está probando.

Capítulo 4

El Framework TestIA

4.1. Objetivo

El framework TestIA busca dar soporte a la automatización de las pruebas aportando las siguientes herramientas:

- Permitir un control completo sobre la ejecución de un determinado escenario sin necesidad de depender de que en la implementación de las funcionalidades exista código especial para una prueba.
- Brindar independencia al desarrollo de las pruebas de la implementación de la funcionalidad. Esto ayuda a que la automatización de las pruebas pueda seguir el ritmo del equipo que implementa la funcionalidad.
- Auditar los servicios involucrados en la ejecución de un escenario de prueba.

La ejecución de un escenario de prueba dentro de la arquitectura del framework, está basado en cuatro grandes pasos:

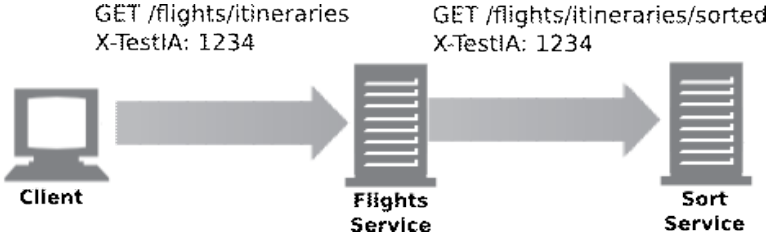

Figura 4.1: Cada servicio debe reenviar el header recibido a todos los servicios que éste invoque.

- Cada aplicación debe mantener, mediante un header HTTP, un dato que identifica el escenario de prueba actual y debe propagarlo por cada invocación que realiza.

- Como se ilustra en la Figura 4.2, el ESB busca ese dato en el encabezado de la invocación y, si lo encuentra, redirige al componente web del framework. En caso de que el ESB deba invocar al servicio de destino, se agrega un header, que el ESB luego remueve, para evitar que vuelva a redirigir al componente web del framework (ver la Figura 4.3) y así se genere un loop infinito.

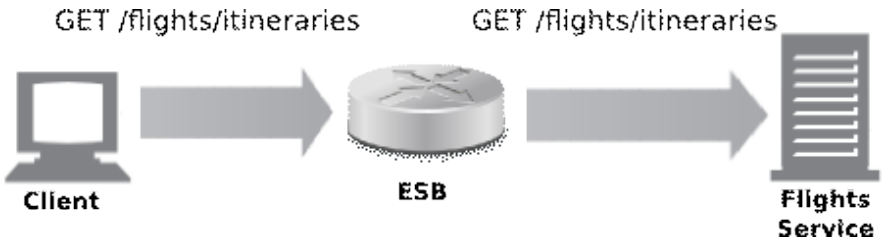

Figura 4.2: Comportamiento normal del ESB.

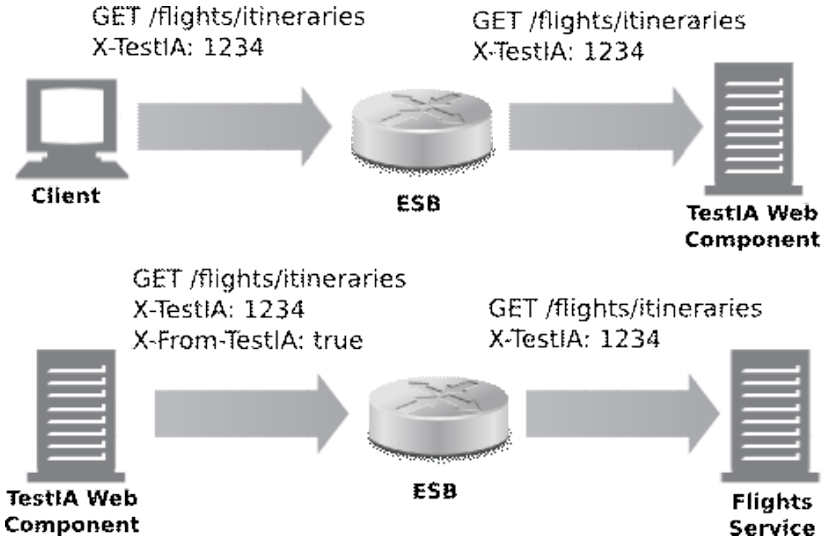

Figura 4.3: Comportamiento del ESB ante la presencia del header.

- El componente web, en base al origen y destino de la invocación y al escenario que se está probando (el cual se identifica por el header HTTP), decide qué hacer con la invocación, como se observa en la Figura 4.4.
- Cada escenario de prueba puede requerir un determinado conjunto de validaciones. El componente web contiene un conjunto estándar de validaciones, pero, en caso de ser necesario, contiene un mecanismo de plugins para que se añadan otras validaciones.

4.2. Arquitectura

Se comenzará por definir la arquitectura del framework, es decir, sus componentes y la relación que deberá existir entre los mismos (ver la Figura 4.5):

- *ESB*: su función principal es la de discernir cuáles invocaciones corresponden a ejecuciones de casos de prueba y cuáles no. Aplicación web: su función es identificar a qué caso de

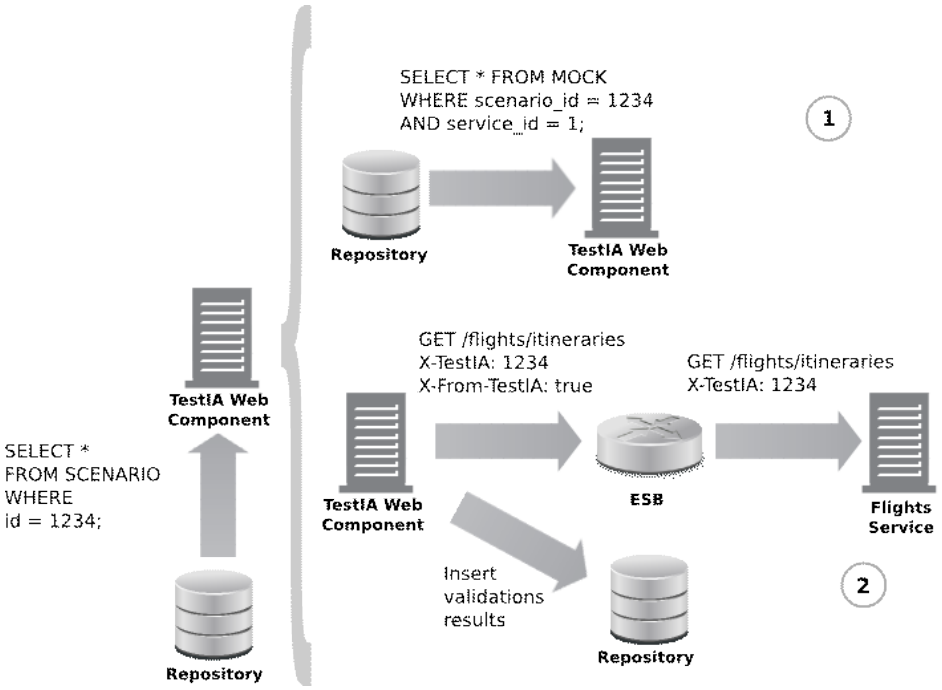

FIGURA 4.4: Ante una invocación, el componente web puede devolver una respuesta predefinida (opción 1), o invocar al servicio y ejecutar validaciones sobre la respuesta (opción 2).

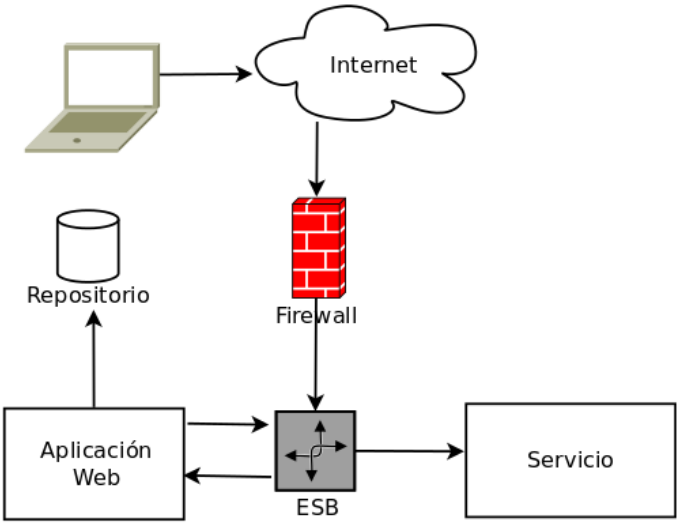

FIGURA 4.5: Arquitectura general.

prueba corresponde una invocación y, en base a la configuración del mismo, tomar las acciones necesarias.

- *Repositorio de casos de prueba*: almacena los casos de prueba y las configuraciones a realizar para cada interacción entre componentes.

- *Repositorio de resultados*: almacena los resultados de las ejecuciones de los casos de prueba. Aquí es posible consultar: tiempos de respuesta, resultado de las validaciones, respuestas de los servicios (originales o no).

- *Repositorio de mocks*: almacena las respuestas precargadas que la aplicación puede devolver como respuesta a la invocación de un servicio.

Esta arquitectura implica que se cumplan las siguientes precondiciones para la correcta ejecución de las pruebas:

- Todas las invocaciones a servicios deben pasar por el ESB.
- Todas las aplicaciones o servicios que consuman otros servicios deben incluir en sus invocaciones el header recibido en la invocación original.

Teniendo esto en cuenta y ayudado por el protocolo de red, el ESB, es capaz de diferenciar aquellas invocaciones que pertenecen a la ejecución de un caso de prueba de aquellas que pertenecen a una ejecución real.

Existe un repositorio de escenarios de prueba identificados unívocamente. Este repositorio contiene las distintas invocaciones a los servicios y la operación a realizar ante cada invocación.

4.3. Estructuras de datos

Dependiendo del dominio de aplicación, los datos requeridos para representar la ejecución de un escenario de prueba pueden variar, pero este framework define la siguiente estructura mínima:

- *Identificador del escenario de prueba*: este es el dato que identifica unívocamente al escenario.
- *Invocaciones*: consiste en una lista de invocaciones a servicios. Cada invocación a un servicio debe contener, al menos, una identificación del servicio y la acción que se debe realizar.
- *Servicios*: esta estructura de datos describe cómo se debe invocar a un determinado servicio. Por ejemplo, se debe especificar el método HTTP que se debe utilizar, la ubicación del servicio (o su URL) y la estructura de datos que debemos enviar.
- *Acciones*: definen qué acción se realizará con la invocación. Las acciones deben estar tipificadas y cada tipo de acción contiene una estructura diferente. Si bien es posible agregar nuevos tipos de acciones, los dos tipos de acciones básicos definidos por el framework son:
    - Validación de los datos de respuesta: esta acción se ejecutará cuando el servicio de destino haya devuelto su respuesta y se ejecutarán validaciones sobre los datos de la misma.
    - Retornar una respuesta predefinida: en lugar de invocar al servicio, se retorna una respuesta del repositorio, simulando una respuesta real del servicio.

Antes de iniciar la ejecución de un escenario de prueba el robot de automatización debe obtener un identificador de dicha ejecución. Este dato es el que utilizará el componente web para buscar

```
{
  ‘‘scenario’’: {
    ‘‘id’’:1,
    ‘‘invokations’’: [{
      ‘‘service_id’’:123,
  ‘‘actions’’: [{
      ‘‘type’’: ‘‘VALIDATION’’,
  ‘‘expression’’: ‘‘some_field == true’’
      }]
    },{
  ‘‘service_id’’:1234,
  ‘‘actions’’:[{
  ‘‘type’’: ’’MOCK’’,
  ‘‘mock_id’’: 1432
  }]
    }]
  }
}
```

Figura 4.6: Ejemplo de la representación de un scenario.

```
{
  ‘‘service’’: {
  ‘‘id’’:123,
  ‘‘host’’: ‘‘info.unlp.edu.ar’’,
  ‘‘port’’: ‘‘8080’’,
  ‘‘method’’: ‘‘GET’’,
  ‘‘content_type’’: ‘‘application/json’’,
  ‘‘accept’’: ‘‘application/json’’
}
```

Figura 4.7: Ejemplo de la representación de un servicio.

```
POST /scenario/1/execution

--------------------------

HTTP/1.1 201 CREATED

{‘‘execution_id’’: 123}
```

Figura 4.8: Ejemplo de la creación de una ejecución de un escenario de prueba.

el escenario de prueba en el repositorio. A partir de este identificador es posible obtener del repositorio tanto la configuración del escenario de prueba como los resultados de las validaciones de las invocaciones a los servicios.

4.4. Ejecución

En la Figura 4.9 se detalla una ejecución de la automatización de un escenario de prueba para reservar un hotel.

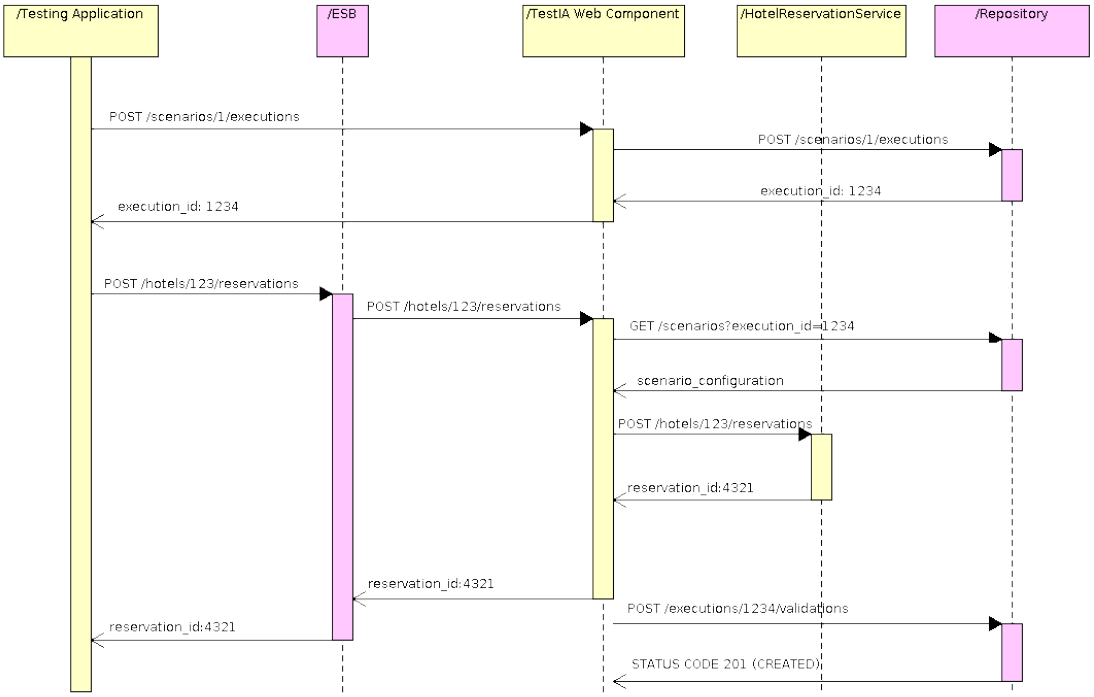

FIGURA 4.9: Ejecución de la creación de la reserva de un hotel. Los elementos de color rosa corresponden a componentes de infraestructura, mientras que los amarillos corresponden a aplicaciones o servicios.

Para comenzar la ejecución, la aplicación de prueba solicita un identificador de ejecución del escenario de prueba. Este dato será el que se envíe en el header HTTP.

Luego, la aplicación de prueba invoca el servicio al igual que lo hace la aplicación productiva, pero agrega el identificador de ejecución del escenario de prueba como valor del header *X-TestIA*.

Este agregado en la metadata del protocolo HTTP hace que el ESB detecte que se trata de una prueba y redirija el tráfico al componente web del framework. Éste busca en el repositorio la configuración del caso de prueba. En el ejemplo de la Figura 4.9, la acción a ejecutar para este servicio son una serie de validaciones sobre la respuesta real del servicio. Por lo que se invoca al servicio, se retorna la respuesta y, de forma asincrónica, se ejecutan una serie de validaciones cuyos resultados se almacenan en el repositorio para su posterior consulta.

Ahora se entrará más en detalle en lo que sucede cada vez que se realiza una invocación al ESB y qué responsabilidad tiene cada componente en cada paso de la ejecución. En la Figura 4.10, se muestra el flujo por el que atraviesa una invocación a un servicio dentro de esta arquitectura.

El ESB identifica que una invocación está dentro de la ejecución de un caso de prueba buscando en el encabezado HTTP por el valor del campo X-TestIA, si lo encuentra, redirige dicha invocación al componente web del framework.

El componente web verifica el identificador del caso de prueba, identifica el servicio que se está invocando y decide, en base a la configuración del caso de prueba, la acción a realizar. Dicha acción puede ser:

- Simular la invocación al servicio y devolver una respuesta previamente guardada. Esta respuesta puede ser exitosa o una respuesta de error.
- Invocar normalmente al servicio y realizar validaciones sobre la respuesta del mismo. Estas validaciones se realizan de forma asincrónica, luego de retornar la respuesta, para no alterar demasiado los tiempos de respuesta del servicio. Los resultados de las validaciones pueden ser consultados luego desde el repositorio de resultados.

En ambos casos, se puede esperar un cierto tiempo antes de retornar la respuesta. Esto es útil cuando se quiere simular que el servicio demora en responder.

En el caso de que la acción a realizar sea invocar al servicio, es posible configurar uno o más post-procesadores de la respuesta. Esto se utiliza para modificar la respuesta y simular comportamientos del servicio que de otra forma serían muy difíciles de obtener. Una vez terminado el post-proceso se puede optar por guardar la respuesta para que luego se pueda utilizar como respuesta en este u otro escenario.

En la Figura 4.10 se puede observar la responsabilidad de cada componente dentro del flujo antes descrito.

4.5. Seguridad

Es importante securizar la ejecución y la configuración de los escenarios de prueba, ya que éstos pueden alterar el funcionamiento de los servicios abriendo la puerta a alguna eventual vulnerabilidad.

Dentro de la arquitectura planteada por el framework, existen dos puntos críticos a securizar.

1. Acceso al servicio de creación de identificadores de ejecución de escenarios. Esto evitará que se pueda utilizar maliciosamente la definición de un escenario de prueba existente. El acceso de usuarios externos es posible restringirlo cambiando la configuración del firewall para que sólo se pueda ejecutar ese servicio desde dentro de la intranet. Dependiendo del dominio de la aplicación y de la infraestructura es posible que se deba exponer el servicio a internet, en este caso y también a modo de auditoría, puede ser necesario requerir datos de autenticación en la ejecución del servicio.
2. Codificación y creación de los escenarios de prueba. Dejar abierto el acceso a esta funcionalidad puede permitir que se manipule la comunicación y la lógica interna de los servicios. Por ejemplo: retornando una respuesta predefinida para un servicio que realice una validación de seguridad en la aplicación productiva.

El acceso, creación y modificación de los escenarios de prueba en su repositorio se debe exponer a través de servicios. Estos servicios se pueden securizar de la misma forma que se aseguró el servicio de creación de identificadores de ejecución.

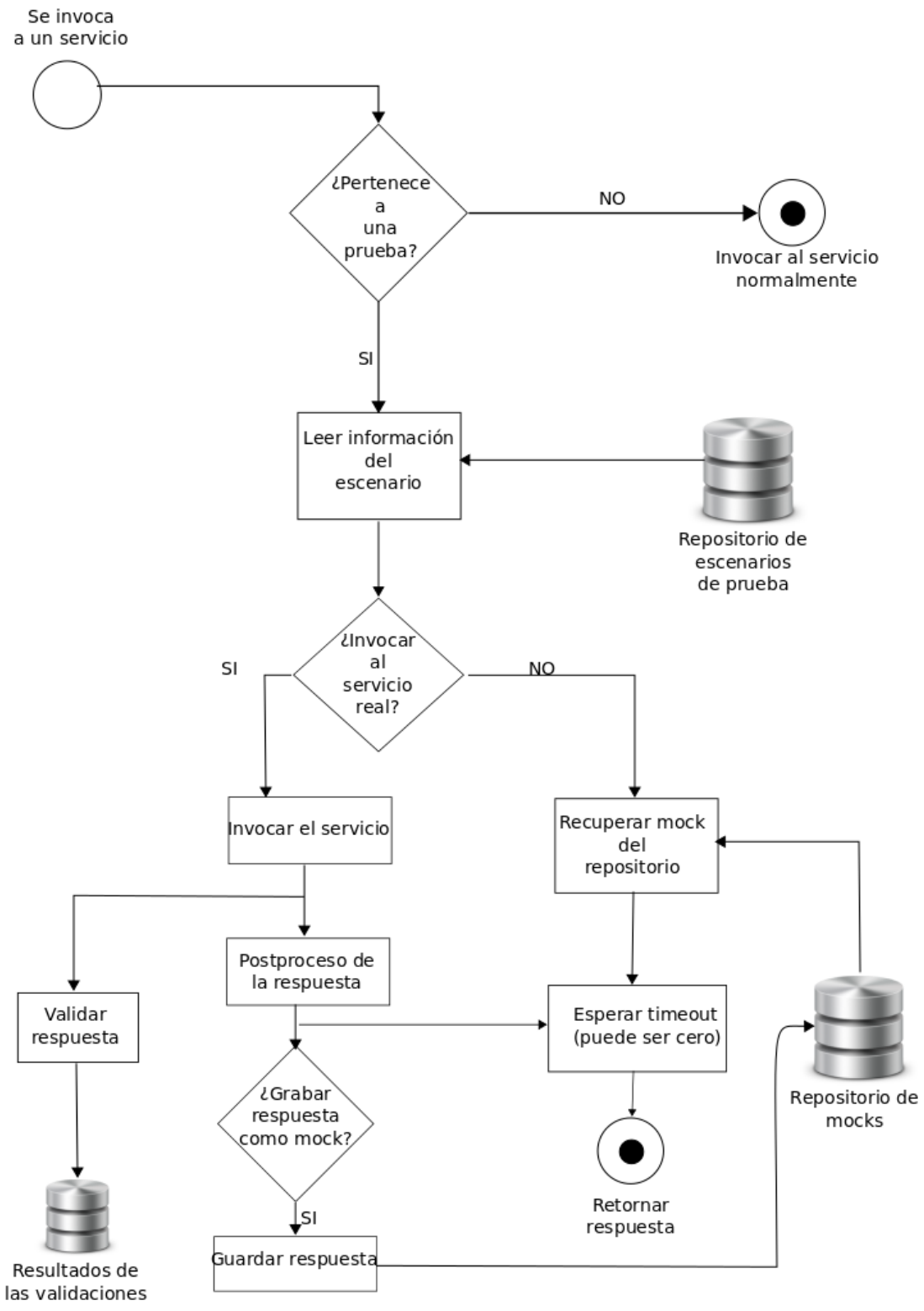

Figura 4.10: Flujo de una invocación a un servicio.

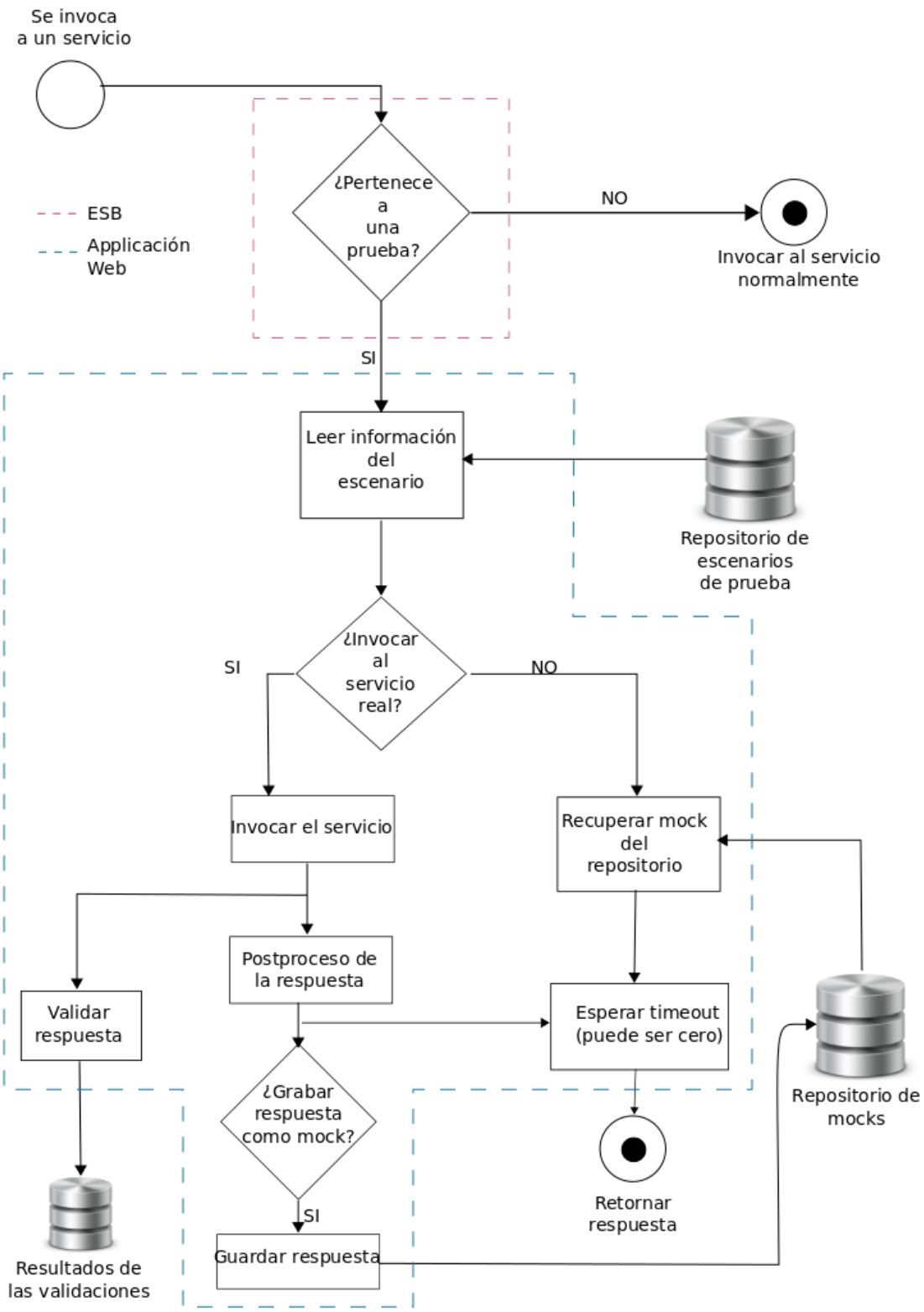

Figura 4.11: Separación de componentes dentro del flujo.

4.6. Conclusiones del capítulo

A lo largo del Capítulo 4 se presentó el framework TestIA, que provee al equipo de automatización de las herramientas necesarias para poder realizar las automatizaciones de las pruebas incluso

antes de que comience el desarrollo de la funcionalidad que se quiere probar.

El framework utiliza los componentes de infraestructura existentes para rutear las invocaciones HTTP, esto hace que para su implementación no se requieran grandes cambios en la arquitectura. Se requiere instalar el componente web del framework que analiza el valor del header HTTP (el cual identifica la ejecución del escenario de prueba) y realiza la acción previamente configurada para ese servicio.

Luego se presentaron las estructuras de datos mínimas que se requieren para representar un escenario de prueba, un servicio y una ejecución. También se explicó cómo se ejecuta un escenario de prueba dentro del framework y cómo se relacionan los componente de la arquitectura en cada etapa de la ejecución.

Por último, se mencionan puntos a tener en cuenta y posibles soluciones para que la implementación del framework no deje abierta la posibilidad de un ataque a la seguridad del sistema.

Capítulo 5

Implementación en Despegar.com

5.1. Contexto del problema

Despegar.com es una agencia de turismo online en constante crecimiento con presencia en 21 países, la mayoría de ellos en Latinoamérica. Al desarrollar su negocio a través de Internet, es muy importante que la plataforma tecnológica acompañe la creciente demanda del negocio. Para lograr este objetivo, la empresa implementó una arquitectura orientada a servicios que consta de los siguientes componentes:

- Un ESB que es responsable, principalmente, de rutear el tráfico entre aplicaciones y servicios.
- Servicios que ofrecen funcionalidades concretas y pueden ser implementados por equipos de desarrollo y en lenguajes de programación diferentes.
- Aplicaciones que consumen y orquestan los servicios con el fin de cumplir con los objetivo del negocio.

Si bien las implementaciones son diferentes, todos los servicios se consumen utilizando HTTP como protocolo de red.

La cantidad de equipos involucrados en cada requerimiento de negocio y la diversidad de tecnologías utilizadas plantean un desafío importante por mantener la calidad de las implementaciones sin generar retrasos en las puestas en producción de las mismas. Con el fin de superar ese desafío, Despegar.com eligió automatizar las pruebas por sobre realizar pruebas manuales. De esta forma, ante cada puesta en producción de una nueva funcionalidad, se ejecutan automáticamente todas las pruebas de regresión para garantizar el buen funcionamiento de lo que previamente funcionaba correctamente y, además, se agregan las pruebas de la nueva funcionalidad desarrollada.

En este contexto, los equipos responsables de desarrollar las automatizaciones de las pruebas (equipo de QA) enfrentan dos problemas:

```
if (nro_tarjeta == 1111111111111111) {
  //la tarjeta es fraudulenta
}

if (nro_tarjeta == 2222222222222222) {
  // la tarjeta no tiene límite disponible
}

if (request.header['X-Fail-Search']) {
  // no devolver resultados
}
```

Figura 5.1: Fragmentos de código usualmente encontrados en funcionalidades que requieren pruebas complejas

- No pueden comenzar el desarrollo de las automatizaciones mientras el desarrollo de la funcionalidad no se encuentre avanzado y subido a algún ambiente de prueba. Lo cual conlleva que el equipo esté ocioso durante ese tiempo, corriendo el riesgo de retrasar la subida a producción.
- Existen casos de prueba que requieren simular errores o determinadas respuestas de los servicios que se consumen. Por ejemplo: el rechazo de una tarjeta por límite insuficiente, tarjetas fraudulentas, devoluciones sobre cobros que deben existir previamente, etc.

El primer punto se resolvió consensuando con el equipo de desarrollo que lo primero a diseñar son las interfaces de los servicios y, mediante un desarrollo rápido, se suben al ambiente de prueba servicios que cumplen dichas interfaces, pero devuelven siempre las mismas respuestas. Eso le da algo con lo cual avanzar al equipo de QA mientras se avanza con la implementación de la funcionalidad.

El segundo problema, se resolvió acordando con el equipo de desarrollo determinados datos ficticios, headers, etc que modifican el comportamiento de los servicios para simular los distintos casos.

Algunos ejemplos:

- Si la tarjeta ingresada en la compra es igual 1111111111111111, el sistema lo toma como una tarjeta fraudulenta.
- Si la tarjeta ingresada en la compra es igual a 2222222222222222, el sistema lo toma como una tarjeta sin límite.
- Si el servicio de búsqueda recibe un determinado header, devuelve o no resultados.

El framework propuesto pretende mejorar estas soluciones para que no exista en el código fuente de la funcionalidad partes que sólo aplican para casos de prueba. En los ejemplos anteriores, en algún lugar del código fuente es inevitable encontrar los fragmentos de código ilustrados en la Figura 5.1.

Además, con la implementación del framework se busca que el equipo de QA logre tener independencia del equipo de desarrollo durante la automatización de las pruebas.

5.2. Implementación

Se realizó una implementación del framework TestIA para poder identificar el escenario de prueba y tomar decisiones en base a ello.

Los pasos a seguir fueron:

- Se creó un repositorio de escenarios de prueba.
- Se agregó un ruteo en el ESB para que, si está presente el header X-TestIA, redirija ese request hacia el componente web del framework.
- Se desarrolló el componente web para que en base al valor del header X-TestIA (que contiene el identificador de la ejecución de un escenario de prueba determinado), el método HTTP y la URL del servicio al que se está invocando se pueda identificar el servicio y realizar la operación definida para el escenario de prueba que se está ejecutando.

De esta forma, una vez que el equipo de desarrollo definió las interfaces de los servicios, el equipo de QA puede configurar el caso de prueba y especificar las respuestas de cada uno de los servicios. Esto le permite desarrollar las automatizaciones sin necesidad de esperar que el equipo de desarrollo haya implementado funcionalidad alguna.

Por otro lado, le permite al equipo de QA probar escenarios complejos y simular respuestas de error sin tener que depender de un desarrollo especial para las pruebas.

5.3. Ejemplo de una ejecución

A continuación se presentará un ejemplo de un escenario de prueba real en donde se quiere probar el proceso de negocio mediante el cual se realiza el cobro de una compra.

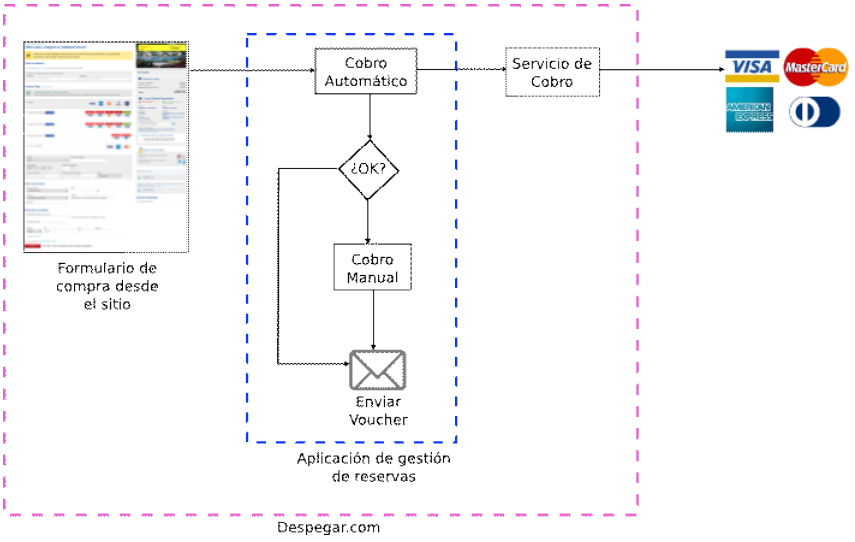

Figura 5.2: Descripción del proceso de negocio de compra.

En la Figura 5.2 se ilustra el proceso de negocio que se ejecutará a lo largo de este ejemplo. El objetivo de las pruebas es validar que, para las diferentes combinaciones de datos de entrada de una compra, se ejecute correctamente el proceso de negocio de cobro. Los resultados posibles son:

- Si el cobro automático se efectúa correctamente, se debe enviar el email al cliente con su voucher.
- Si el cobro automático retorna un error, el cobro se debe procesar de forma manual.

Para automatizar estas pruebas se definieron dos escenarios de prueba dentro del repositorio. Sus estructuras están descriptas en la Figura 5.3.

```
{
  ‘‘scenario_id’’:1,
  ‘‘description’’: ‘‘Si el cobro automático no se puede realizar porque
                  no tiene límite, el cobro se debe procesar de forma manual’’,
  ‘‘invokations’’:
  [{
    ‘‘service_id’’: 1,
    ‘‘actions’’:
    [
      {
        ‘‘type’’: ‘‘MOCK’’,
        ‘‘mock’’: {
          ‘‘http_status’’: 409,
          ‘‘http_body’’: {‘‘error_message’’: ‘‘INSUFFICIENT_LIMIT’’}
        }
      }
    ]
  }]
}

{
  ‘‘scenario_id’’:2,
  ‘‘description’’: ‘‘Si el cobro automático se puede realizar correctamente,
                  se debe enviar el mail al cliente con el voucher.’’,
  ‘‘invokations’’:
  [{
    ‘‘service_id’’: 1,
    ‘‘actions’’:
    [
     {
        ‘‘type’’: ‘‘VALIDATION’’,
        ‘‘expression’’: ‘‘http_status == 200’’
      }
    ]
  }]
}
```

Figura 5.3: Escenarios de prueba utilizados en este ejemplo.

Ambos escenarios de prueba hacen referencia al service_id 1, el cual corresponde a un servicio con la estructura que se puede observar en la Figura 5.4.

```
{
  ‘‘service_id’’: 1,
  ‘‘service_name’’: ‘‘Servicio de cobro’’,
  ‘‘service_url_pattern’’: ‘‘/collection-service/.*’’,
  ‘‘service_method’’: ‘‘POST’’,
}
```

Figura 5.4: Representación de un servicio dentro del repositorio

Paso 1: Para comenzar la ejecución del escenario de prueba, el robot crea una ejecución del escenario en el repositorio, tal como se observa en la Figura 5.5, invocando un servicio provisto por el componente web de TestIA. La respuesta de esta invocación contiene un identificador que luego será utilizado durante el resto de la ejecución.

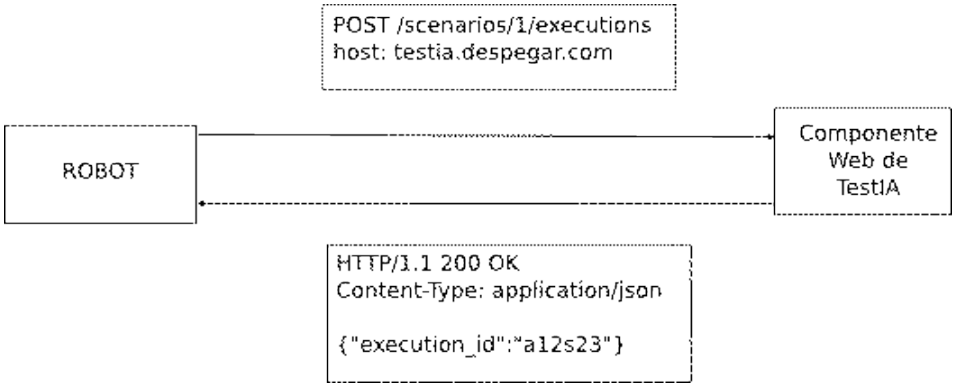

Figura 5.5: El robot obtiene un identificador para la ejecución del escenario.

Paso 2: El robot completa el formulario e invoca el servicio de Compra injectando el header X-TestIA con el valor recibido en el paso anterior (ver Figura 5.6).

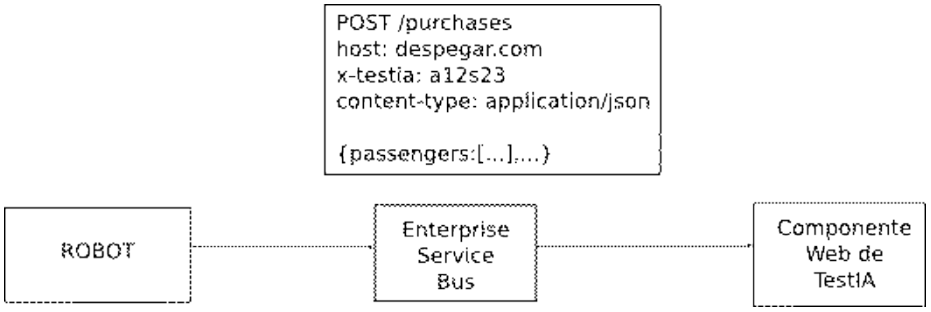

Figura 5.6: El robot invoca el servicio de compra.

Paso 3: La invocación pasa por el ESB que, al detectar el header X-TestIA, la redirige al componente web del framework. Éste último, recupera la configuración del escenario del repositorio a partir del identificador de ejecución recibido como valor del header HTTP. Este paso está ilustrado en la Figura 5.7.

Paso 4: Dado que el servicio de compra no aparece dentro de las invocaciones a interceptar por el framework, el componente web agrega el header X-From-TestIA y vuelve a invocar el servicio a través del ESB. Este header HTTP evita que el ESB vuelva a redirigir la invocación al componente web (ver Figura 5.8).

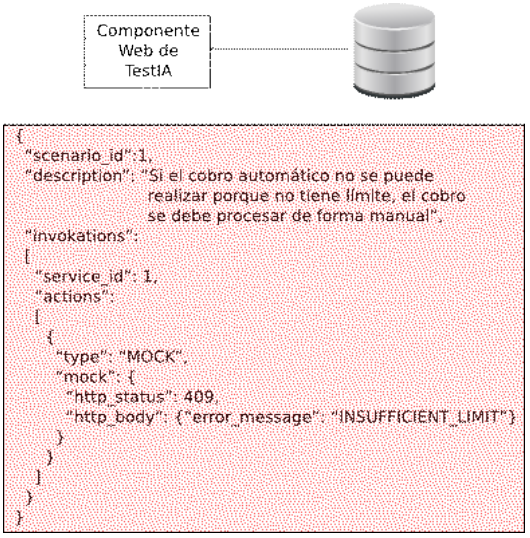

Figura 5.7: El componente web recupera la configuración del escenario de prueba.

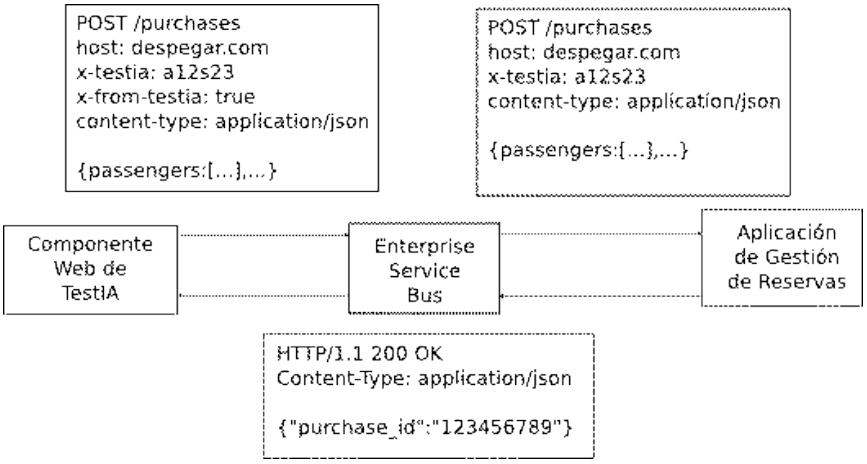

Figura 5.8: El componente web deja pasar la invocación que no pertenece al escenario de prueba en ejecución.

Paso 5: La aplicación de gestión de reservas invoca al servicio de cobro a través del ESB, manteniendo el header X-TestIA que recibió. El ESB detecta el header HTTP y lo redirige al componente web del framework. Debido a que la URL del servicio de cobro coincide con la expresión regular del servicio con id 1 que está en la configuración del escenario de prueba que se está ejecutando, éste realiza las acciones que corresponden. En caso de que se trate de la ejecución del escenario con id 1, retornará una respuesta de error. En caso de que se trate de la ejecución del escenario con id 2, invocará al servicio y luego de recibir la respuesta validará que el código de respuesta HTTP sea igual a 200.

Paso 6: El robot contrasta el resultado del proceso de negocio con los resultados de las validaciones y las acciones del componente web que fueron quedando registradas en el repositorio de resultados.

5.4. Conclusiones del capítulo

Con la implementación del framework el equipo de QA obtuvo dos beneficios importantes:

- Independencia en las pruebas ya que no dependen de que la implementación de los servicios tenga en cuenta casos especiales complejos para determinados escenarios.
- Velocidad de desarrollo, porque no es necesario esperar a que el equipo de desarrollo tenga publicadas las interfaces en los ambientes de prueba.

Tabla 5.1: Antes y después de TestIA en Despegar.com

Característica	**Antes de TestIA**	**Después de TestIA**
Tiempo	El equipo de automatización debe esperar a que la funcionalidad esté terminada e instalada en algún ambiente de prueba para poder comenzar con las pruebas.	El equipo de automatización puede comenzar el desarrollo una vez diseñadas las interfaces de los servicios.
Auditoría	El resultado de las automatizaciones se determina por el resultado final de las pruebas. No es posible evaluar resultados internos para verificar cómo se llega a ese resultado final.	Se pueden agregar validaciones a las respuestas de aquellas invocaciones a servicios que pasen por el ESB y contengan el identificador de la ejecución del escenario de prueba.
Dependencia de la implementación	Es necesario alterar el funcionamiento normal de la implementación para generar los contextos en los que se ejecutará la automatización. Por ejemplo: simular que se realizó un cobro.	Al poder predefinir las respuestas de los servicios, es posible simular respuestas de los servicios para generar diferentes escenarios de prueba sin alterar la implementación de la funcionalidad.

Capítulo 6

Puntos de extensión

En este capítulo se describen las líneas futuras de trabajo que se desprenden o tienen relación con los conceptos presentados en esta tesis. Estos temas exceden de momento el alcance de este trabajo, pero son complementos que enriquecerán los lineamientos presentados. Es útil poder avanzar en estos temas para potenciar el desarrollo futuro del framework propuesto.

6.1. Carga de la configuración los escenarios de prueba

En esta primera versión del framework, no se profundizó sobre la carga de nuevos escenarios de prueba y la modificación de la configuración de los escenarios existentes. Insertar y configurar los escenarios de prueba directamente en el repositorio puede implicar un alto costo en tiempo.

Para resolver esto, se propone buscar la forma de crear y configurar escenarios de prueba en el repositorio con el fin de reducir los tiempos necesarios para preparar el entorno para ejecutar las automatizaciones.

6.2. Implementar el framework para otros protocolos de red

Si bien esta tesina se centró en la implementación del framework sobre el protocolo HTTP, se deja abierta la posibilidad de que se implemente también en otros protocolos de comunicación que permitan transmitir metadatos.

Por ejemplo, en el caso de SOAP, es posible aplicar la misma metodología planteada en esta tesina utilizando un campo en el encabezado del mensaje SOAP definido en el sección 2.4 de esta tesis.

6.3. Análisis de datos de respuesta binarios

A lo largo de esta tesina se implementaron ejemplos donde el cuerpo del mensaje es de texto plano. Queda pendiente profundizar el análisis de las respuestas y la ejecución de posibles validaciones sobre servicios que reciban y retornen datos en formato binario.

Capítulo 7

Conclusión

En este trabajo se desarrolló un marco teórico con los lineamientos necesarios para implementar una solución que permite realizar pruebas de integración automatizadas utilizando una o más de las siguientes funcionalidades:

- Definir escenarios de prueba que pueden involucrar a dos o más servicios.
- Validar el orden en que se invocan los servicios.
- Validar que la respuesta del servicio invocado sea la esperada.
- Interceptar y devolver una respuesta, simulando la ejecución del servicio invocado.

Como validación del marco teórico desarrollado, se implementó un framework que permite mejorar la velocidad de la automatización de las pruebas en una arquitectura orientada a servicios independizando las pruebas de la implementación de la funcionalidad. El camino elegido para independizar las pruebas de la implementación fue apoyarse en los componentes de infraestructura y en los protocolos de red de capa de aplicación utilizados.

Además, se aplicó dicho framework en una arquitectura orientada a servicios que utiliza HTTP como protocolo de comunicación. Los resultados obtenidos luego de esta implementación fueron muy satisfactorios generando los siguientes beneficios:

- Se logró una alta independencia entre la automatización de las pruebas y la implementación de la funcionalidad. Es decir, no se requiere de ningún código fuente especial en la implementación de la funcionalidad para poder automatizar escenarios de prueba complejos.
- Esta independencia es la que permite al equipo de automatización comenzar con la misma sin necesidad de esperar al equipo de desarrollo.

Adicionalmente, el framework obliga al equipo de automatización a conocer cómo se comunica el módulo que se está probando con el resto de los módulos involucrados en el escenario de prueba. Esto permite que ante un error, se pueda informar con mayor detalle al equipo de desarrollo

cual fue causa del problema o, al menos, describir y analizar con mayor detalle la ejecución del escenario de prueba.

Bibliografía

[1] Jun Zhu Huafang Tan y Heyuan Huang Hehui Liu, Zhongjie Li. A unified test framework for continuous integration testing of soa solutions. 2009.

[2] Grace A. Lewis y Dennis B. Smith Soumya Simanta, Edwin Morris. A framework for assurance in service-oriented environments. pages 1–6, Abril 2010. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5482443.

[3] Arnon Rotem-Gal-Oz. *SOA Patterns*. 2012.

[4] Lonneke Dikmans y Ronald van Luttikhuizen. *SOA Made Simple*. 2012.

[5] Jason Arbon y Jeff Carollo James A. Whittaker. *How Google Tests Software*. 2012.

[6] Roy W. Schulte and Yefim V. Natis. 'service oriented' architectures, part 1. 1996.

[7] S. Hong J. Lee, K. Siau. Enterprise integration with erp and eai, communications of the acm. Febrero 2003.

[8] David Gourley y Brian Totty. *HTTP: The Definitive Guide*. 2002.

[9] James Snell. *Programming Web Services with SOAP*. 2001.

[10] Kristan Vingrys. *ThoughtWorks Anthology (Chapter 13)*. 2008.

[11] Jia Zhang. An approach to facilitate reliability testing of web services components. Noviembre 2004.

[12] Sebastian Wieczorek y Alin Stefanescu. Soa test governance: enabling service integration testing across organization and technology borders. Octubre 2009.

[13] Felipe Meneses Besson. A framework for automated testing of web service choreographies. Octubre 2012.

[14] Jia Zhang. Fault injection-based test case generation for soa-oriented software. Junio 2006.

[15] Sujit Kumar Chakrabarti. Test-the-rest: An approach to testing restful web-services. Noviembre 2009.

[16] Youngkon Lee. Semi-automatic test assertion transformation scheme for soa. Octubre 2011.

[17] Youngkon Lee. An implementation case study: Business oriented soa execution test framework. Agosto 2011.

[18] Ching-Seh Wu y Yen-Ting Lee. Automatic saas test cases generation based on soa in the cloud service. 2012.

[19] Guido Schmutz y Daniel Liebhart y Peter Welkenbach. *Service-Oriented Architecture: An Integration Blueprint*. 2010.

[20] D.A. Chappell. *Enterprise Service Bus*. 2004.