# Automatizacion de Pruebas de Integración en Arquitecturas Orientadas a Servicios

Lic. Facundo Chambó, Mg. Patricia Bazán, Lic. Juan Ramón Cortabitarte

Universidad Nacional de la Plata, Facultad de Informática

**Resumen** En arquitecturas 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. Este artículo 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.

**Keywords:** calidad, automatizacion, http, arquitecturas orientadas a servicios, framework

## 1. Introducción

Este artículo fue escrito como resultado de mi tesina de grado correspondiente a la carrera de Licenciatura en Sistemas en la Facultad de Informática de la Universidad de La Plata. La misma fue dirigida por Mg. Patricia Bazán y mi asesor profesional fue Lic. Juan Ramón Cortabitarte.

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.

El aporte de este trabajo es construir un framework para dar soporte a la automatización de pruebas de integración en el desarrollo de software 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 y la dependencia entre diferentes equipos de desarrollo.

Para lograr este objetivo, será necesario utilizar los componentes de infraestructura existentes para rutear las invocaciones HTTP e instalar una aplicación web con sus repositorios de persistencia.

## 2. Framework TestIA (Testing Information by Automation)

### 2.1. 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 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:

- 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 3, 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 y se genere un loop infinito.

[1] También llamados *Criterios de Aceptación*

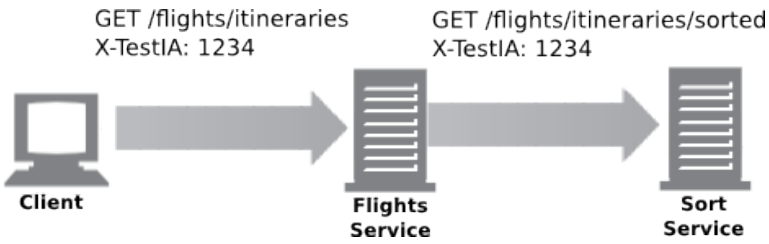


**Figura 1.** Cada servicio debe reenviar el header recibido a todos los servicios que éste invoque.

- 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.
- 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.

## 3. 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 2):

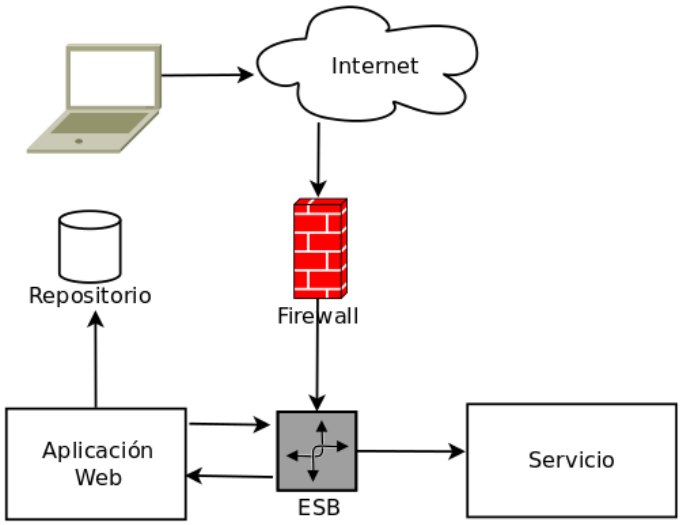


**Figura 2.** Arquitectura general.

- *ESB*: su función principal es la de discernir cuáles invocaciones corresponden a ejecuciones de casos de prueba y cuáles no.[2]
- *Aplicación web*: su función es identificar a qué caso de 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 (ver la Figura 3) [3]
- Todas las aplicaciones o servicios que consuman otros servicios deben incluir en sus invocaciones el header recibido en la invocación original.

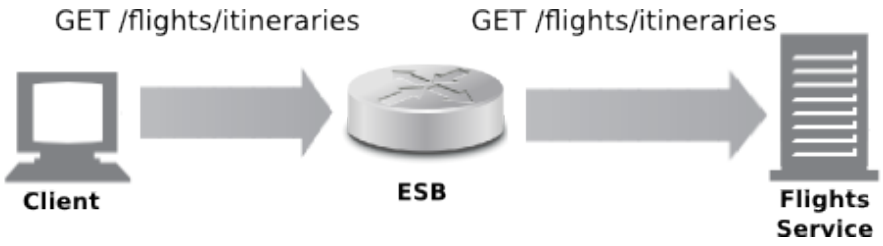


**Figura 3.** Comportamiento normal del ESB.

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.

### 3.1. 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.

```
{
  ‘‘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.** Ejemplo de la representación de un scenario.

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 el escenario de prueba en el repositorio.

```
6

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

**Figura 5.** Ejemplo de la representación de un servicio.

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.

```
POST /scenario/1/execution

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

HTTP/1.1 201 CREATED

{‘‘execution_id’’: 123}
```

**Figura 6.** Ejemplo de la creación de una ejecución de un escenario de prueba.

### 3.2. Ejecución

Durante la ejecución de un escenario de prueba, 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 HTTP *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. Por ejemplo, si la configuración del escenario define que es necesario ejecutar validaciones sobre la respuesta real del servicio, el componente web invoca al servicio, 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 7, 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 7 se puede observar la responsabilidad de cada componente dentro del flujo antes descrito.

## 4. Caso de Estudio: Despegar.com

### 4.1. 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.

8

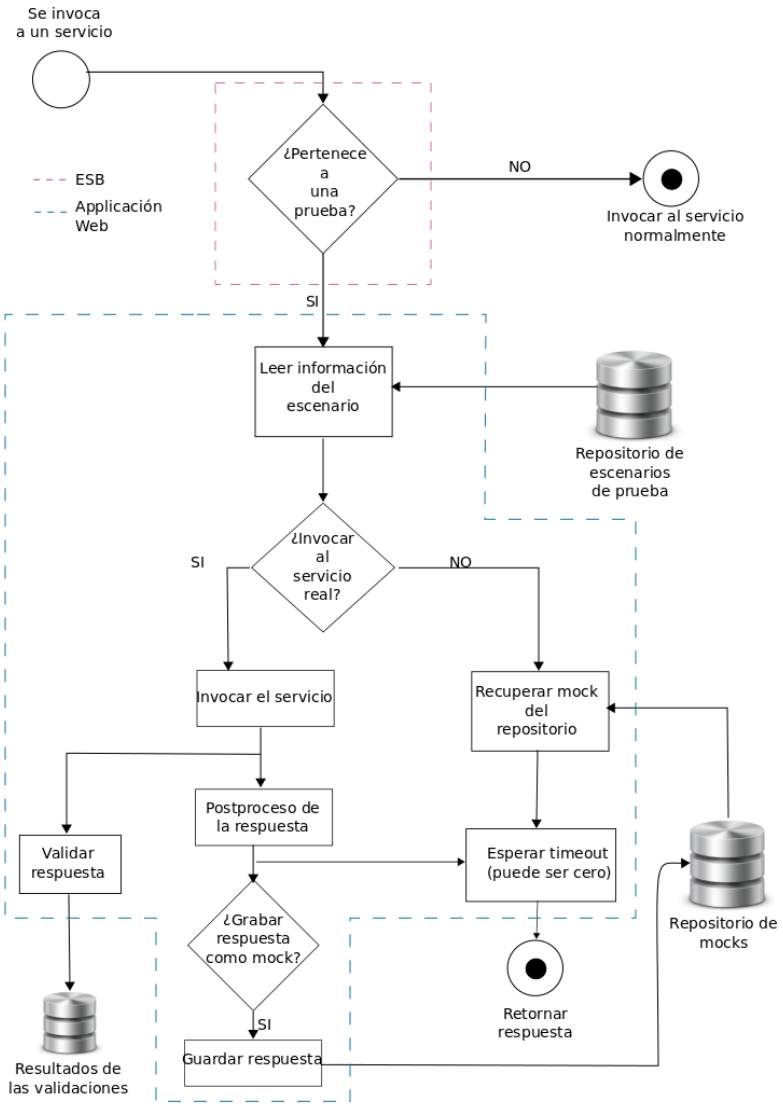


**Figura 7.** Separación de componentes dentro del flujo.

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 automática-

mente 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:

- No pueden comenzar el desarrollo de las automatizaciones mientras el desarrollo de la funcionalidad no se encuentre avanzado e instalado en 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 siguientes fragmentos de código.

```
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
}
```

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.

### 4.2. 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.

### 4.3. 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.

En la Figura 8 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 son:

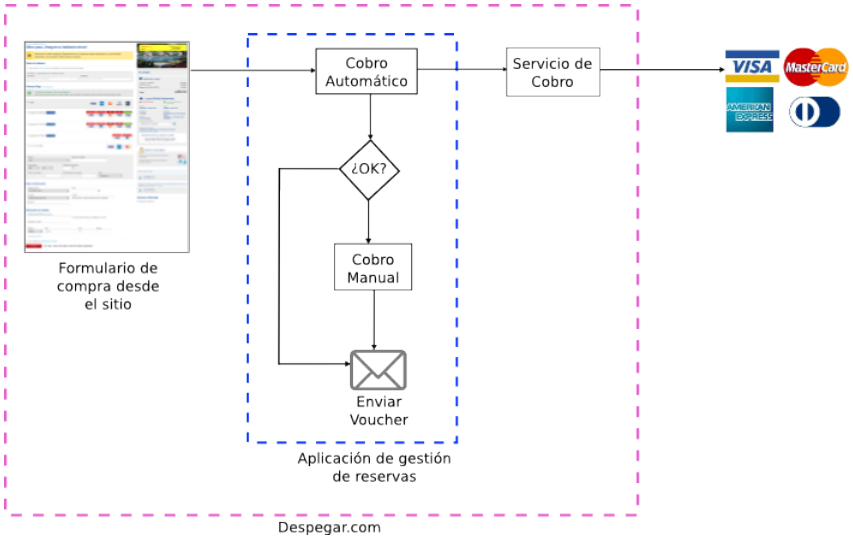


**Figura 8.** Descripción del proceso de negocio de compra.

```
{
  ‘‘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’’:
```

12

```
  [{
    ‘‘service_id’’: 1,
    ‘‘actions’’:
    [
     {
        ‘‘type’’: ‘‘VALIDATION’’,
        ‘‘expression’’: ‘‘http_status == 200’’
      }
    ]
  }]
}
```

Ambos escenarios de prueba hacen referencia al service_id 1, el cual corresponde a un servicio con la estructura.

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

**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 9, 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.

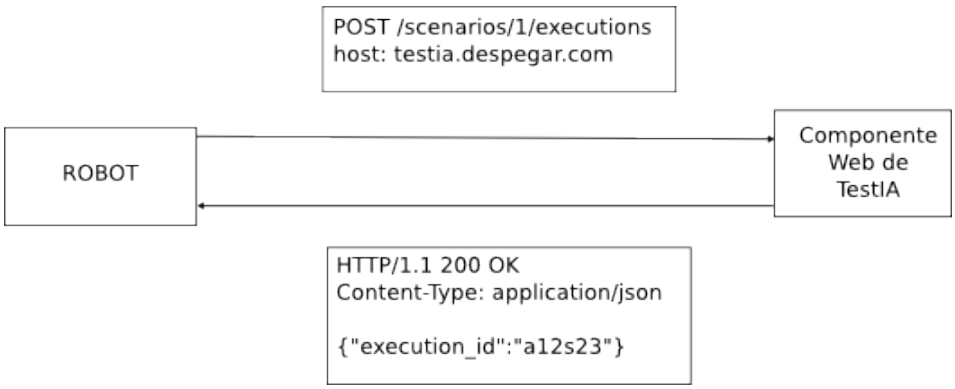


**Figura 9.** El robot obtiene un identificador para la ejecución del escenario.

**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 la Figura 10).

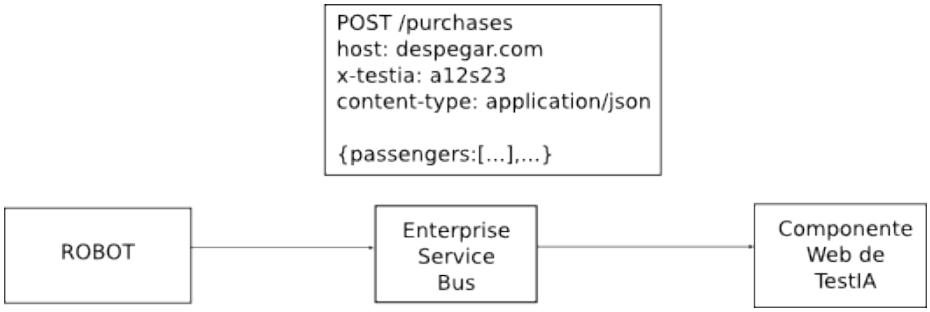


**Figura 10.** El robot invoca el servicio de compra.

**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 11.

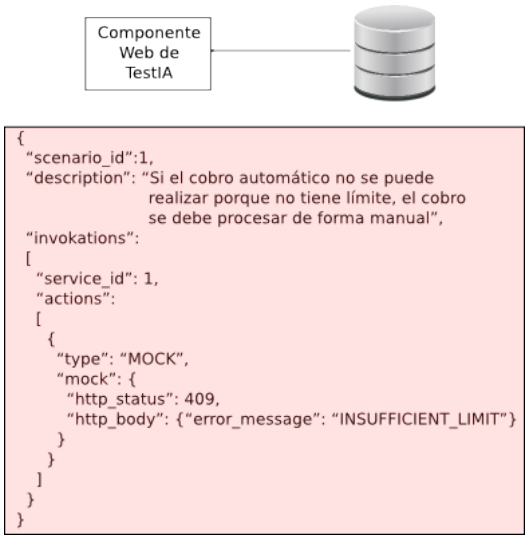


**Figura 11.** El componente web recupera la configuración del escenario de prueba.

**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 la Figura 12).

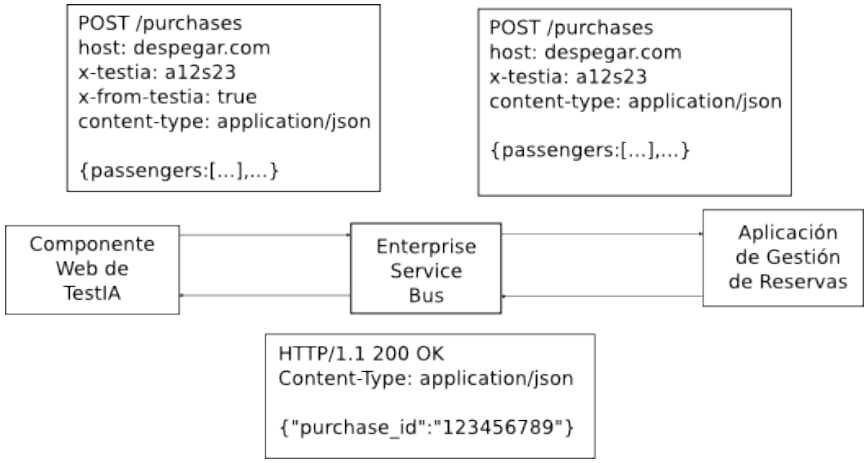


**Figura 12.** El componente web deja pasar la invocación que no pertenece al escenario de prueba en ejecución.

**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. Conclusión

En este trabajo se implementó 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 que la respuesta del servicio invocado sea la esperada.
- Interceptar y devolver una respuesta, simulando la ejecución del servicio invocado.

El framework desarrollado 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 dentro de la empresa Despegar.com. Los resultados obtenidos luego de esta implementación fueron muy satisfactorios generando los siguientes beneficios.

- 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.

**Cuadro 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. |

## Referencias

1. Soumya Simanta, Edwin Morris, Grace A. Lewis y Dennis B. Smith: A Framework for Assurance in Service-Oriented Environments 1-6 (Abril 2010)
2. David Chappell Enterprise Service Bus O'Reilly Media (Junio 2004)
3. Arnon Rotem-Gal-Oz SOA Patterns Manning Publications (Septiembre 2012)