Framework para la creación y ejecución de pruebas automatizadas sobre servicios REST

Maximiliano Agustín Mascheroni[1], Emanuel Irrazábal[1,2],

[1] Universidad Nacional Del Nordeste. Facultad de Ciencias Exactas y Naturales y Agrimensura. Departamento de Informática
[2] Universidad de la Cuenca del Plata. Facultad de Ingeniería y Tecnología.
{mascheroni, eirrazabal}@exa.unne.edu.ar

Abstract. La arquitectura REST emerge como alternativa al diseño de servicios web, con mayor simpleza que SOAP y los servicios basados en WSDL, haciendo que las pruebas sobre esta arquitectura cobren mayor relevancia. Si bien existen muchas herramientas disponibles, muy pocas pueden integrarse a un proceso de desarrollo continuo de software, donde el tiempo es un factor clave. En este trabajo, partiendo de los principios de esta arquitectura y los fundamentos de pruebas unitarias, se propone un framework para pruebas sobre servicios REST que puede ser integrado a este tipo de entornos. El mismo ha sido implementado en una empresa de desarrollo software multinacional con gran éxito. Se ha comprobado la disminución en un 90% del tiempo para realizar regresiones automatizadas y refactorizaciones en un entorno de integración continua. Finalmente, la utilización de herramientas para ejecutar pruebas automáticas, ha permitido otro tipo de mejoras, como la generación automática de reportes.

Keywords: Pruebas de servicios web, REST, desarrollo continuo de software

1 Introducción

La transferencia de estado representacional (REST - Representational State Transfer) fue ganando adeptos como una alternativa más simple al protocolo de acceso a objetos simples (SOAP - Simple Object Access Protocol) y a los servicios web basados en el lenguaje de descripción de servicios web (WSDL - Web Services Descripcion Language) [1]. La arquitectura REST define un conjunto de principios por los cuales se diseñan servicios web haciendo foco en los recursos del sistema, incluyendo: cómo acceder al estado de dichos recursos y cómo se transfieren por HTTP hacia clientes escritos en diversos lenguajes [2].

La adopción de REST por grandes proveedores de servicios de aplicaciones Web como Google, Yahoo y Facebook o por compañías como Wal-Mart, indican su creciente aceptación [3]. Estas organizaciones consideraron a las tecnologías SOAP e interfaces basadas en WSDL como obsoletas, y decidieron implementar REST como un modelo orientado a los recursos más fácil de usar [4]. En este sentido, la calidad cobra una gran importancia; un servicio web que no ha sido probado puede enviar al

cliente contenido no deseado, malicioso o con errores, en lugar de las estructuras de datos que esperadas [5].

En la actualidad, existe un gran número de herramientas como, por ejemplo, Postman [6], RestClient [7] o Swagger [8], que permiten realizar pruebas sobre servicios REST. Sin embargo, estas herramientas no permiten el desarrollo de pruebas automatizadas ni su integración a servidores de integración continua. Por ello, en este trabajo se propone un framework que utiliza los mismos principios del desarrollo de servicios REST y los combina con los principios de pruebas unitarias. Además de esta sección introductoria, el trabajo se compone en 5 secciones. En la sección 2, se listan los fundamentos y principios de la arquitectura REST. En la sección 3, se describen las herramientas más utilizadas para realizar pruebas de servicios web. En la sección 4 se presenta el framework propuesto y los resultados de su implementación se muestran en la 5. Finalmente, las conclusiones son mencionadas en la sección 6.

2 REST

Un servicio web se define como una aplicación software identificada por un URI cuyas interfaces se pueden definir, describir y descubrir mediante documentos XML [9]. Los servicios web permiten la interoperación de sistemas distribuidos heterogéneos con independencia de las plataformas hardware y software empleadas. Por tanto, puede pensarse en ellos como en una arquitectura, conceptual y tecnológica, haciendo posible que distintos servicios se describan, publiquen, descubran y utilicen a través de sistemas distribuidos, empleando la infraestructura proporcionada por Internet [10].

REST emergió en los últimos años como el modelo predominante para el diseño de servicios. Ha logrado un gran impacto en la web desplazando a SOAP y a las interfaces basadas en WSDL por tener un estilo más simple de usar [2]. Fielding define a REST como “un estilo de arquitectura de software para sistemas hipermedia distribuidos tales como la World Wide Web” [1]. Para otros autores, es un conjunto de principios para el diseño de redes, que es utilizado comúnmente para definir una interfaz de transmisión sobre HTTP de manera análoga a como lo hace SOAP [11].

Una implementación de un servicio REST los siguientes principios [4]:

- **Verbos HTTP**: los más comunes son GET, POST, PUT y DELETE [12].
- **Sin estado**: ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.
- **Identificadores de recursos uniforme (URI)**: cada recurso es accedido únicamente a través de su URI.
- **Hipermedios**: permite transferir los formatos HTML, JSON y XML.

3 Pruebas de servicios REST

Según Johnson, la mayoría de las organizaciones han migrado o comenzado a utilizar arquitecturas basadas en servicios REST [13]. Esto ha generado la aparición de un

gran número de herramientas para realizar pruebas sobre estos sistemas, buscando asegurar la calidad de los mismos. A continuación se describen las más utilizadas:

- **Postman** [6]: es un cliente REST que puede integrarse con el navegador Google Chrome. Es una herramienta que permite crear peticiones de manera muy sencilla, para luego poder enviarlas a un servidor y visualizar la respuesta.
- **Rest-Client** [7]: al igual que Postman, es un cliente REST que se integra con el navegador Mozilla Firefox. Permite la construcción de peticiones para probarlas directamente contra un servidor.
- **Swagger** [8]: permite la representación de las APIs REST de un sistema. De este modo, es posible realizar pruebas mediante la generación de peticiones y verificación de las respuestas del servidor a través de su interfaz.
- **SOAP-UI** [14]: es una herramienta multi-plataforma de código abierto para realizar pruebas funcionales sobre servicios REST y principalmente SOAP. Permite crear y ejecutar regresiones automatizadas sobre servicios y también medir los tiempos de respuesta.

Sin embargo, la principal limitación de todas ellas es que requieren de un esfuerzo manual para realizar las verificaciones sobre los datos obtenidos del servidor, para corroborar que sean los esperados. Asimismo, este esfuerzo manual conlleva a que estas herramientas no puedan ser integradas en un entorno de desarrollo continuo de software, donde los elementos principales son las pruebas automatizadas y los servidores de integración continua. Para buscar una solución a estos inconvenientes, surgen propuestas en Git-hub como Restfuse [15] o Rest-assured [16], que aún se encuentran en etapas de desarrollo y solo disponen de versiones beta.

4 Framework Propuesto

Partiendo de las propias herramientas utilizadas para la creación de servicios REST, como, por ejemplo, RestTemplate de Spring [17] y tomando los principios de las pruebas unitarias [18], es posible crear un framework de pruebas para este tipo de servicios. Por ejemplo, los mismos creadores de Spring, lanzan el Spring MVC Test Framework [19], que permite la creación de pruebas unitarias de servicios REST. Al ser pruebas unitarias, tienen la ventaja de integrarse en un servidor de integración continua junto con el código fuente. El problema que presentan estas herramientas es la dependencia: las pruebas se acoplan a la herramienta utilizada, y esto, a su vez, limita su alcance.

El framework propuesto incluye:

1. La creación de una herramienta para interactuar con servicios REST, sin depender de un cliente en particular.
2. El manejo de objetos de transferencia de datos en las peticiones y las respuestas.
3. Un modelo de dos capas para la creación de pruebas.

4.1 Interacción con los servicios REST

En primer lugar, se busca resolver el problema de la dependencia a partir del uso de interfaces .Con la creación de una interfaz es posible brindar métodos para crear las peticiones y utilizar los verbos HTTP, interactuando con los servicios sin importar la implementación. En la Fig. 1, se muestra un ejemplo de una interfaz HTTPClient. Esta interfaz brinda los verbos HTTP con diferentes variaciones y métodos para preparar las peticiones. Las variaciones de cada verbo HTTP sirven para determinar el formato tanto de la petición, como el de la respuesta que se desea obtener. Los formatos permitidos son: un texto plano o un objeto simple con atributos.

```
7  public interface HTTPClient {
8
9      public void setHeader(String field, String value);
10
11     public void setExpectedObjectType(Class<?> type);
12
13     public void addURLQueryParameter(String field, String value);
14
15     public void addBodyParameter(String key, String value);
16
17     public void setCredentialsForBasicAuthentication(String user, String password);
18
19     public HTTPResponse get(String url);
20
21     public HTTPResponse get(String url, Class<?> type);
22
23     public HTTPResponse post(String url);
24
25     public HTTPResponse post(String url, Class<?> type);
26
27     public HTTPResponse post(String url, Object requestBody);
28
29     public HTTPResponse post(String url, Object requestBody, Class<?> type);
30
31     public HTTPResponse post(String url, String requestBody);
32
33     public HTTPResponse post(String url, String requestBody, Class<?> type);
34
35     public HTTPResponse delete(String url);
36
37     public HTTPResponse delete(String url, String item);
```

Fig. 1. Parte de la Interfaz HTTPClient

Cada clase que implementará esta interfaz, tendrá el código para interactuar con los servicios REST de acuerdo al cliente REST utilizado. Habrá tantas implementaciones de la interfaz como clientes REST se quieran utilizar (ver Fig. 2). En el artículo de Bhandari [20], pueden encontrarse una lista de diferentes clientes REST para implementarse con el lenguaje Java. De este modo, cada clase será un envoltorio de un cliente REST. En la Fig. 3 puede verse la implementación de un método GET con sus variaciones, utilizando el cliente RestTemplate de Spring [17].

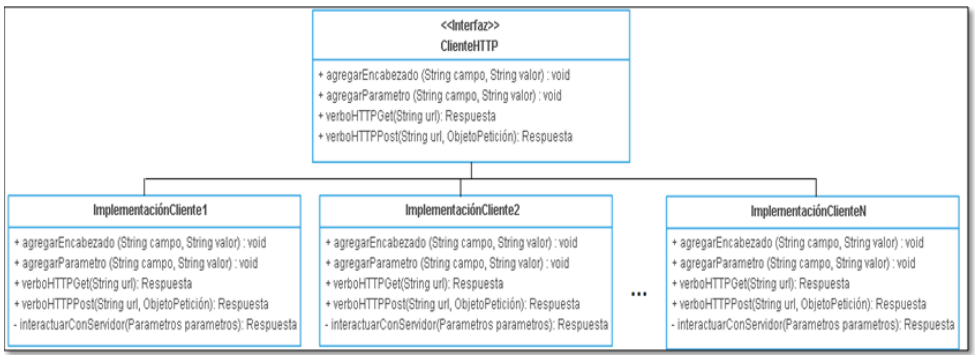

Fig. 2. Diagrama de clases de la implementación de los clientes HTTP

Como puede apreciarse en la Fig. 1 y en la Fig. 3, los métodos que representan verbos HTTP devuelven un objeto HTTPResponse. Esta clase es una representación de una respuesta del servidor. Como cada cliente HTTP maneja su propio tipo de objeto "respuesta de servidor", es necesario hacer una generalización de las mismas para ser utilizada de la misma forma por todas las implementaciones de los clientes. La Clase HTTPResponse contiene métodos para obtener de la respuesta: una lista de los encabezados, el código de estado de la petición y el resultado de la interacción con el servidor.

```java
public class RestTemplateHTTPClient implements HTTPClient {

    //Otras implementaciones

    @Override
    public HTTPResponse get(String url) {
        return this.getMethod(url, String.class);
    }

    @Override
    public HTTPResponse get(String url, Class<?> type) {
        return this.getMethod(url, type);
    }

    private HTTPResponse getMethod(String url, Class<?> type) {
        //Obtiene la URL con todos los parámetros
        URI uri = this.getURIWithURLQueryParameters(url);
        //Crea una entidad-petición RestTemplate usando los encabezados definidos
        HttpEntity<String> entity = new HttpEntity<String>(this.headers);
        //Llama a la ejecución del método executeHTTPAction con los parámetros
        return this.executeHTTPAction(uri, HttpMethod.GET, entity, type);
    }

    private HTTPResponse executeHTTPAction(URI uri, HttpMethod method,
                HttpEntity<?> requestEntity, Class<?> responseType) {
        //Ejecuta la acción solicitada contra el servidor
        ResponseEntity<Object> response = (ResponseEntity<Object>) RestTemplate
                                .exchange(uri, method, requestEntity, responseType);
        //Obtiene los encabezados de la respuesta del servidor
        HTTPHeaders responseHeaders = new HTTPHeaders(this.getHeaders(response.getHeaders()));

        //Genera un objeto HTTPResponse con los datos de la respuesta del servidor
        return this.createResponse(response.getStatusCode().value(), "OK", response.getBody(), responseHeaders);
    }

}
```

Fig. 3. Implementación de un método GET, en una clase de implementación de la interfaz.

4.2 Manejo del Objeto de Transferencia de Datos

Como se menciona anteriormente, tanto las peticiones como las respuestas aceptan dos formatos en sus cuerpos: texto plano o cualquier objeto simple. Una de las maneras de hacerlo es mediante el uso del patrón DTO [21], [22]. Para trabajar con este patrón, es necesario modelar cada entidad que participa en una petición o en una respuesta como un objeto simple.

Para la deserialización, es necesario utilizar alguna herramienta que permita el procesamiento de los formatos que soporta HTTP, como por ejemplo Jackson [23].

La petición puede ser cualquier objeto simple, que a través de una serialización será transformado al formato solicitado por el servidor. Por otro lado, la respuesta también puede ser cualquier objeto simple. De esta manera, se utilizan objetos (DTOs), que contienen en su interior los tipos solicitados.

4.3 Modelo para la creación de pruebas

El modelo propuesto, consiste en separar la lógica de interacción con los servicios web de la capa de pruebas. Una de las principales razones es cumplir con el patrón AAA [24], el cual indica que en primer lugar, las precondiciones y datos de entrada deben prepararse al principio, luego operar sobre el componente bajo prueba y por último verificar que se cumplan los resultados esperados. Por otro lado, se busca mejorar el mantenimiento de las pruebas, permitiendo reducir el esfuerzo requerido para conservar el correcto funcionamiento de ellas. Por esta razón se presentan dos capas: capa de servicio y capa de pruebas.

Capa de Servicio

Contiene clases especiales denominadas servicios. Así como el patrón PageObject [25] permite separar la implementación de una página web de las pruebas, de la misma manera, la capa de servicios aísla de las pruebas las siguientes implementaciones:

- La especificación del cliente a utilizar
- La configuración de los parámetros para generar la petición
- La creación de la petición
- La obtención de la respuesta

Cada método en una clase de servicio, representará una operación disponible de ese servicio. De esta manera, para la capa de pruebas, así como en un navegador web, la interacción con el servidor es transparente. En la Fig. 4 se implementa un servicio que tiene una operación para obtener una lista de países según un continente.

```
1  pubic class CountryService {
2
3      private HTTPClient client = new RestTemplateClient();
4
5      private String username = PropertyReader.getProperty("Username");
6      private String password = PropertyReader.getProperty("Password");
7
8      private String countriesURL = PropertyReader.getProperty("url.countries");
9
10     public HTTPResponse getCountries(String continent) {
11         client.setCredentialsForBasicAuthentication(username, password);
12         client.addURLParameter("continent", continent);
13         return client.get(countriesURL, Country[].class);
14     }
15
16 }
```

Fig. 4. Ejemplo de la Implementación de una clase de servicio.

Capa de Pruebas

En esta capa estarán las pruebas. Cada prueba deberá tener los siguientes pasos:

1. Creación del servicio que va a ser utilizado.
2. Invocación a la operación del servicio
3. Verificar si el servicio respondió correctamente, mediante su código de estado.

4. Realizar las verificaciones correspondientes a la prueba en sí.

Los pasos 1 y 2, consisten en definir e instanciar la clase del servicio a utilizar, e invocar al método que representa la operación que se desea. El resultado de esto será un objeto HTTPResponse que contiene los métodos para acceder a las variables para realizar las verificaciones de los pasos 3 y 4. Las verificaciones y la ejecución de las pruebas puede realizarse utilizando herramientas de ejecución de pruebas automáticas como JUnit [26] o TestNG [27]. Esto puede verse en la Fig. 5.

```
1
2   @Test(description="Esta prueba verifica que la lista de paises no esté vacia")
3   public void countryTest(String continent) {
4       //Paso 1
5       CountryService service = new CountryService();
6       //Paso 2
7       HTTPResponse response = service.getCountries(continent);
8       //Paso 3
9       Assert.assertEquals(response.getStatusCode(), 200, "El servidor no respondió correctamente");
10      //Paso 4
11      Country[] countries = (Country[]) response.getBodyResponse();
12      int cant = countries.lenght;
13      for(int i=0, i<cant; i++) {
14          SoftAssert.assertNotNull(countries[i]);
15      }
16      SoftAssert.assertAll();
17  }
18
```

Fig. 5. Ejemplo de una prueba de un servicio web.

5 Resultados obtenidos

En las primeras etapas de pruebas sobre servicios web, esta actividad se realizó manualmente. La empresa constaba de un equipo que constaba de 15 personas realizando pruebas manuales utilizando clientes REST como Postman y Swagger, y el proceso tomaba un total de 3 horas. Por esta razón, se decidió invertir en la utilización de un cliente REST para realizar las pruebas. Esto redujo significativamente el tiempo para realizar las pruebas. Sin embargo, se perdía mucho tiempo en comprender la manera en manipular la herramienta y en aplicar cada actualización de la misma sobre cada una de las pruebas. El tiempo de creación de una prueba promedio era de entre 1 a 2 horas. Las refactorizaciones podrían tomar un 1 día entero de trabajo de una persona. Finalmente, se optó por una solución que consistió en la implementación del framework descrito en este trabajo.

En la Fig. 6 se pueden apreciar los resultados (expresados en minutos) de los 3 enfoques según 4 características:

- Tiempo para crear un caso de prueba
- Tiempo para ejecutar un lote de caso de pruebas
- Tiempo requerido para realizar cambios varios (refactorización)
- Tiempo requerido para generar la documentación

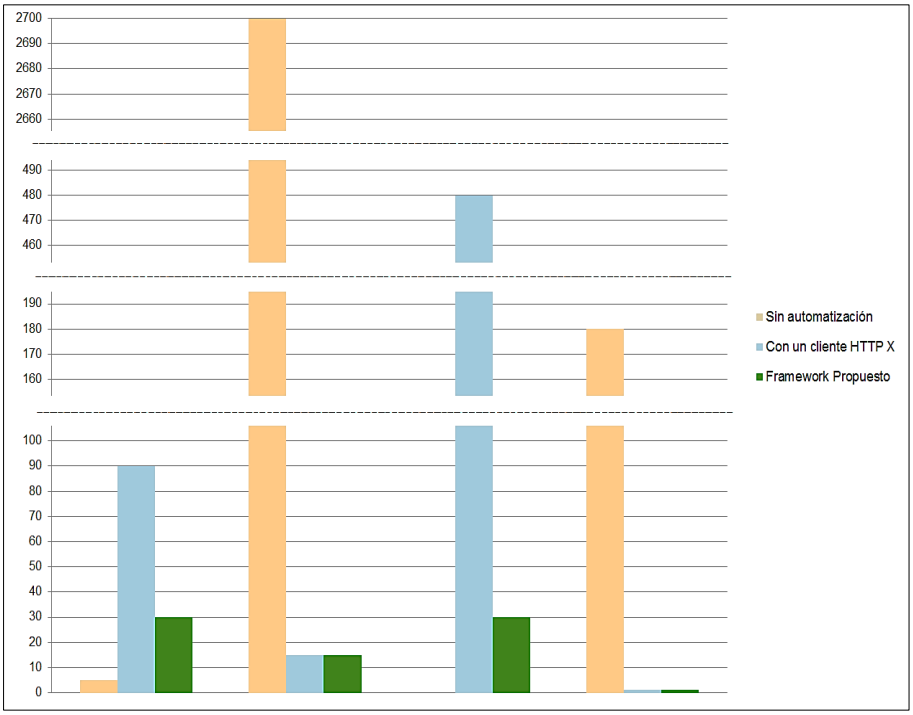

Fig. 6. Comparación de características asociadas a las pruebas según los diferentes enfoques.

El análisis de los resultados mostrados en la Fig. 6, puede verse en la Tabla 1. Se ha comprobado que tanto con el cliente como con el framework, se obtiene una disminución en un 90% del tiempo para realizar regresiones automatizadas con un servidor de integración continua, así como también la generación de reportes es automática. Con el Framework, el tiempo para crear un caso de prueba pasó a ser de 30 minutos. Además, se redujo la cantidad de refactorizaciones, con un tiempo promedio de 30 minutos.

Tabla 1. Comparación de características asociadas a las pruebas según los diferentes enfoques.

Característica	Sin automatización	Con un cliente HTTP "X"	Framework Propuesto
Creación de un caso de prueba	5 min	90 min	30 min
Ejecución de un lote de pruebas	2700 min	15 min	15 min
Refactorización	N/A	480 min	30 min
Documentación	180 min	1 min	1 min

6 Conclusión

Son muchas las empresas que están migrando a las tecnologías de servicios REST y las pruebas sobre ellos cobran relevancia. Si bien existen muchas herramientas, muy pocas pueden integrarse a un proceso de desarrollo continuo de software, donde el tiempo es un factor fundamental.

Mediante la integración de los principios de la arquitectura REST utilizados para el desarrollo de los mismos, y los fundamentos de pruebas unitarias, fue posible la creación de un framework que permite el desarrollo de pruebas automatizadas para servicios REST.

El tiempo de ejecución de un lote de pruebas que llevaba 3 horas, realizado manualmente con el equipo de 15 personas, pasó a ser de 15 minutos en un servidor de integración continua 100% automatizado. Este equipo ahora utiliza esas 3 horas para realizar otro tipo de pruebas complementarias y brindar soporte a otras áreas. El tiempo para crear un caso de prueba pasó a ser de 30 minutos. Además, se redujo la cantidad de refactorizaciones, con un tiempo promedio de 30 minutos.

La generación de reportes también es una tarea totalmente automatizada, a través de las herramientas de ejecución de pruebas automáticas.

Finalmente, los responsables del área de calidad manifiestan que con la implementación del framework, la confiabilidad en las pruebas aumentó considerablemente, ya que algunos errores podían pasar como desapercibidos con las pruebas manuales.

Referencias

1. R. Fielding, "Architectural styles and the design of network-based software architectures," University of California, Irvine, Tesis Doctoral 2000.
2. L. Deseta. (2008, Noviembre) Introducción a los servicios web RESTful. Dos Ideas (En Ideas Ágiles).
3. J. Sharma and M. Singh, "Web Services Oriented Architecture for DPI based Network Forensics Grid," International Journal of Energy, Information and Communications, vol. 6, no. 3, pp. 19-28, 2015.
4. L. Richardson and S. Ruby, RESTful Web Services. California, USA: O'Really Media, Inc., 2008.
5. L. Richardson and S. Ruby, "Writing Web Service Clients," in RESTful Web Services. California, USA: O'Really Media, Inc., 2008, ch. 2, p. 45.
6. Postman. [Online]. https://www.getpostman.com/
7. C. Zhou. RESTClient, a debugger for RESTful web services. [Online]. https://addons.mozilla.org/es/firefox/addon/restclient/
8. Swagger. APIS.Guru: A wikipedia for web APIs. [Online]. http://swagger.io/
9. World Wide Web Consortium (W3C). (2004, Febrero) Web Services. [Online]. https://www.w3.org/TR/ws-arch/
10. J. J. Domínguez Jiménez, A. Estero Botaro, I. Medina Bulo, M. Palomo Duarte, and F. Palomo Lozano, "El reto de los servicios Web para el software libre," in Proceedings of the FLOSS International Conference, 2007, pp. 117 - 132.
11. C. Morales Machuca, "Estado del arte: Servicios Web," Universidad Nacional de Colombia, Tesis de Maestría 2010.

12.R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. Fielding, Hypertext transfer protocol--HTTP/1.1, 1999.
13.R. Johnson, "REST and Web Services: In Theory and in Practice," Research Gate, Diciembre 2010.
14.SoapUI. SMARTBEAR. [Online]. https://www.soapui.org/
15.Restfuse. [Online]. https://github.com/eclipsesource/restfuse
16.Rest-assured. [Online]. https://github.com/rest-assured/rest-assured
17.Pivotal. Spring. [Online]. https://spring.io/
18.P. Tahchiev, F. Leme, V. Massol, and G. Gregory, JUnit in action, 2nd ed., ACM Digital Library, Ed.: Manning Publications Co, 2010.
19.Spring. Spring MVC Test Framework. [Online] http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/htmlsingle/#spring-mvc-test-framework
20.S. Bhandari. (2011, Enero) List of tools for Rest Web Services in Java. DZone. [Online]. https://dzone.com/articles/list-tools-rest-web-services
21.M. Fowler. (2010) Data Transfer Object. [Online]. http://martinfowler.com/eaaCatalog/dataTransferObject.html
22.M. Gulden and S. Kugele, "A concept for generating simplified restful interfaces," in Proceedings of the 22nd international conference on World Wide Web companion, International World Wide Web Conferences Steering Committee., 2013, pp. 1391-1398.
23.D. Peng, L. D. Cao, and W. J. Xu, "Using JSON for data exchanging in web service applications," Journal of Computational Information Systems, vol. 7, no. 16, pp. 5883-5890, 2011.
24.Microsoft Developer Network. (2016) Unit Test Basics - AAA Pattern. [Online]. https://msdn.microsoft.com/en-us/library/hh694602.aspx
25.M. Fowler. (2013, Septiembre) PageObject. [Online]. http://martinfowler.com/bliki/PageObject.html
26.JUnit. (2002) JUnit. [Online]. http://junit.org/junit4/
27.TestNG. (2004) TestNG.org. [Online]. http://testng.org