Testes de serviços web

Site:	Moodle institucional da UTFPR	Impresso por:	MARLENE VASCONCELOS MORAES DE OLIVEIRA
Curso:	CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)	Data:	domingo, 8 set. 2024, 20:54
Livro:	Testes de serviços web		

Descrição

Serviços web estão por toda a parte, já que hoje a quantidade de aplicações, dos mais diferentes domínios e tecnologias, precisam se comunicar.

Índice

1. Introdução

Serviços web estão por toda a parte, já que hoje a quantidade de aplicações, dos mais diferentes domínios e tecnologias, precisam se comunicar. Por exemplo, uma loja virtual precisa consultar os Correios para saber o valor de determinado frete, ou uma aplicação de compra de passagem aérea que fala com a aplicação de reserva de carros para fazer um pacote turístico.

Existem diferentes maneiras para se distribuir sistemas e fazê-los conversarem entre si. As maneiras mais conhecidas são SOAP e REST. Nesta semana, lidaremos com serviços REST, já que esta tem sido a preferência de uma boa parte da indústria ultimamente. Aplicações REST fazem uso de requisições e respostas HTTP simples, geralmente trafegando dados em XML ou JSON.

Imagine uma aplicação que está preparada para responder em XML e JSON. Veja um retorno de exemplo:

```
<list>
<usuario>
<id>1</id>
<nome>Mauricio</nome>
<email>mauricio@gmail.com</email>
</usuario>
<usuario>
<id>2</id>
<nome>Guilherme</nome>
<email>guilherme@gmail.com</email>
</usuario>
</list>
```

Excelente. É assim que um serviço web em REST funciona: fazemos uma requisição e ele nos devolve uma resposta. Precisamos agora testar que, sempre que invocarmos esse serviço web, ele nos devolverá usuários. Para escrever esse teste, faremos uso do framework chamado Rest-Assured. Ele, junto com o JUnit, nos ajudará a escrever testes que consomem serviços web. Ele já tem um monte de métodos que nos ajudam a fazer requisições, ler respostas etc. O Rest-Assured pode ser encontrado aqui: https://code.google.com/p/rest-assured/.

2. Usando o Rest-Assured

Vamos começar nosso primeiro teste. O que ele fará é justamente uma requisição do tipo GET para o servidor, e garantirá que a lista com 2 usuários foi recuperada. Escrever testes já não é mais segredo nessa altura. Usaremos JUnit como sempre. Mas, aqui, utilizaremos a API do Rest-Assured para fazer essa requisição. A API do framework é fluente, e faz muito uso de métodos estáticos. Vamos importá-los todos desde já:

```
import static com.jayway.restassured.RestAssured.*;
import static com.jayway.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;
```

O primeiro método que vamos aprender é o get(URL). Ele faz uma requisição do tipo GET para a URL. Em seguida, precisamos dizer a ele que queremos tratar a resposta como XML. Veja só como fica a linha:

```
public class UsuariosWSTest {

    @Test
    public void deveRetornarListaDeUsuarios() {
        XmlPath path = get("/usuarios?_format=xml")
                .andReturn().xmlPath();
    }
}
```

Com esse objeto XmlPath, podemos agora pegar os dados desse XML. Por exemplo, sabemos que essa URL nos devolve 2 usuários. Vamos recuperá-los. Para isso, usaremos o método getObject(), que recebe um caminho dentro do XML, e a classe que ele deve desserializar:

```
    @Test
    public void deveRetornarListaDeUsuarios() {
        XmlPath path = get("/usuarios?_format=xml")
                .andReturn().xmlPath();
        Usuario usuario1 = path.getObject("list.usuario[0]",
                Usuario.class);
        Usuario usuario2 = path.getObject("list.usuario[1]",
                Usuario.class);
    }
```

Com esses dois objetos em mãos, basta agora fazermos asserções neles:

```
    @Test
    public void deveRetornarListaDeUsuarios() {
        XmlPath path = get("/usuarios?_format=xml")
                .andReturn().xmlPath();
        Usuario usuario1 = path.getObject("list.usuario[0]",
                Usuario.class);
        Usuario usuario2 = path.getObject("list.usuario[1]",
                Usuario.class);
        Usuario esperado1 = new Usuario(1L,
                "Mauricio", "mauricio@gmail.com.");
        Usuario esperado2 = new Usuario(2L,
                "Guilherme", "guilherme@gmail.com");
        assertEquals(esperado1, usuario1);
        assertEquals(esperado2, usuario2);
    }
```

Pronto. Nosso teste passa. Veja que, com o uso do Rest-Assured, não gastamos tempo algum fazendo requisições, ou mesmo parseando as respostas. Preocupamo-nos apenas com o comportamento esperado.

Como falamos anteriormente, podemos passar a informação cuja resposta esperamos em XML pelo próprio header HTTP. Isso também é fácil com Rest-Assured. Veja o código a seguir, que faz uso dos métodos given() e header():

```java
@Test
public void deveRetornarListaDeUsuarios() {
    XmlPath path = given()
            .header("Accept", "application/xml")
            .get("/usuarios")
            .andReturn().xmlPath();
    Usuario usuario1 = path.getObject("list.usuario[0]",
            Usuario.class);
    Usuario usuario2 = path.getObject("list.usuario[1]",
            Usuario.class);
    Usuario esperado1 = new Usuario(1L,
            "Mauricio", "mauricio@gmail.com");
    Usuario esperado2 = new Usuario(2L,
            "Guilherme", "guilherme@gmail.com");
    assertEquals(esperado1, usuario1);
    assertEquals(esperado2, usuario2);
}
```

Nosso teste continua passando. Esses são os primeiros passos com o Rest-Assured.

3. Testando JSONs

Vamos agora testar o serviço que nos devolve um usuário, de acordo com o seu ID. Dessa vez, o retorno será em JSON. A URL para pegarmos o usuário 1, por exemplo, http://localhost:8080/usuarios/show?usuario.id=1&_format=json.

Vamos testar esse serviço agora. O teste é parecido com o anterior, mas dessa vez precisamos mudar duas coisas:

- Passar um parâmetro pela querystring (usuario.id).
- Tratar o retorno que, dessa vez, vem em JSON.

O tratamento em JSON é bem simples. O Rest-Assured abstraiu bem isso para nós. Lembra do XmlPath? Pois bem, ele tem também o JsonPath, que é idêntico. A sintaxe é a mesma. Veja só como faríamos se o teste anterior fosse em JSON:

```
    @Test
    public void deveRetornarListaDeUsuarios() {
        JsonPath path = given()
                .header("Accept", "application/xml")
                .get("/usuarios")
                .andReturn().jsonPath();
        List<Usuario> usuarios = path
                .getList("list.usuario", Usuario.class);
        Usuario esperado1 = new Usuario(1L,
                "Mauricio", "mauricio@gmail.com");
        Usuario esperado2 = new Usuario(2L,
                "Guilherme", "guilherme@gmail.com");
        assertEquals(esperado1, usuarios.get(0));
        assertEquals(esperado2, usuarios.get(1));
    }
```

Repare que só mudamos de um para outro!

Para passar o parâmetro também é bem simples. Podemos usar o método parameter(), que recebe o nome do parâmetro, bem como o conteúdo a ser enviado. Veja o novo teste completo:

```
    @Test
    public void deveRetornarUsuarioPeloId() {
        JsonPath path = given()
                .header("Accept", "application/xml")
                .get("/usuarios/show")
                .andReturn().jsonPath();
        Usuario usuario = path.getObjetct("usuario", Usuario.class);
        Usuario esperado = new Usuario(1L,
                "Mauricio", "mauricio@gmail.com");
        assertEquals(esperado, usuario);

    }
```

Note que estamos passando o ID=1, e JSON no Accept. Ao rodar o teste, ele passa.

Mesmo se seu JSON não voltar um objeto que possa ser desserializado, a API de Path (tanto de XML quanto de JSON) possibilita que você apenas navegue por ele. Por exemplo, se quiséssemos pegar apenas a String contendo o nome “Mauricio”, poderíamos fazer:

```
path.getString("usuario.nome")
```

Veja os métodos disponíveis: getBoolean(), getDouble(), entre outros. É importante conhecê-los para que você faça bom uso da biblioteca.

4. Enviando dados para o WebService

Até o momento, só consultamos dados. Vamos agora consumir webservices que recebem dados. Vamos adicionar um usuário. O serviço está disponível em http://localhost:8080/usuarios. Para inserir um usuário, precisamos fazer um POST com um XML (ou JSON) de um usuário; o formato é o mesmo da nossa entidade Usuario.

Vamos começar nosso teste criando o usuário que será adicionado:

```
    @Test
    public void deveAdicionarUmUsuario() {
        Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");
    }
```

Agora usaremos a simples API do Rest-Assured para fazer o POST. Essa requisição é mais complicada. Precisamos:

- Dizer que a requisição enviará em XML (Content-Type no Header);
- Dizer que o retorno deve ser em XML também (Accept);
- O corpo da requisição deve ser oobjeto “joao” serializado;
- O retorno do serviço web deve ser 200.

Vamos começar configurando os dados da requisição. Veja que invocamos o método header(), contentType e body. Os três são autoexplicativos. Mas repare que o body sabe que o objeto deve ser serializado em XML, justamente por causa do valor passado para o contentType:

```
    @Test
    public void deveAdicionarUmUsuario() {
        Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");
        given()
                .header("Accept", "application/xml")
                .contentType("application/xml")
                .body(joao);
    }
```

Em seguida, precisamos fazer o post, e receber os dados de volta como XML (da maneira com que já estamos acostumados). Com isso, já conseguimos inclusive fazer o assert:

```
    @Test
    public void deveAdicionarUmUsuario() {
        Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");
        XmlPath retorno
                = given()
                        .header("Accept", "application/xml")
                        .contentType("application/xml")
                        .body(joao)
                        .when()
                        .post("/usuarios")
                        .andReturn()
                        .xmlPath();
        Usuario resposta = retorno.getObject("usuario", Usuario.class);
        assertEquals("Joao da Silva", resposta.getNome());
        assertEquals("joao@dasilva.com", resposta.getEmail());
    }
```

Por fim, é legal também garantir que o código de retorno do HTTP seja 200. Para isso, usaremos o método expect(), que faz a asserção que queremos:

```java
@Test
public void deveAdicionarUmUsuario() {
    Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");
    XmlPath retorno
            = given()
                    .header("Accept", "application/xml")
                    .contentType("application/xml")
                    .body(joao)
                    .expect()
                    .statusCode(200)
                    .when()
                    .post("/usuarios")
                    .andReturn()
                    .xmlPath();
    Usuario resposta = retorno.getObject("usuario", Usuario.class);
    assertEquals("Joao da Silva", resposta.getNome());
    assertEquals("joao@dasilva.com", resposta.getEmail());
}
```

Excelente. Nosso teste agora passa. Observe novamente que, com o uso do framework, nossos testes são por demais simples de serem escritos.

Veja que aqui estamos fazendo uma inserção. Ou seja, um novo elemento está sendo inserido no serviço externo. Você, desenvolvedor de testes, precisa ficar atento a isso, pois isso pode mudar o resultado de outros testes. Imagine que você tenha um teste que garanta que o serviço que retorna a quantidade de usuários funcione; se você adicionar um novo usuário, o teste vai quebrar, pois o número de usuários cresceu.

Precisamos sempre tomar cuidado com cenários em serviços externos. Se você criou um usuário, você precisa deletá-lo depois. Para isso, você pode usar algum serviço de deleção, que é disponibilizado pelo serviço web.

4.1. Configurando o Rest-Assured

Você reparou que até agora temos passado endereços relativos, mas ele sabe que nossa aplicação está em http://localhost: 8080? Pois bem, isso é a configuração padrão dele. Você pode mudá-la. Por exemplo, se quiséssemos mudar a URL base, faríamos:

```
RestAssured.baseURI = "http://www.meuendereco.com.br";
RestAssured.port = 80;
```

Geralmente colocamos esse tipo de configuração em algum lugar centralizado. Por exemplo, em algum @Before ou @BeforeClass da bateria de testes. Assim, fica fácil mudar essa configuração, caso o endereço do serviço mude.

5. Outros recursos do Rest-Assured

Além de fazer requisições e tratar bem as respostas de diferentes tipos, como XML e JSON, o Rest-Assured ainda nos dá outros recursos.

Por exemplo, podemos garantir que um cookie foi gerado. Para isso, basta usarmos o expect(), que já conhecemos. A URL /cookie/teste nos gera um cookie com o nome “rest-assured”, com o valor “funciona”. Vamos validá-lo:

```
@Test
public void deveGerarUmCookie() {
    expect()
            .cookie("rest-assured", "funciona")
            .when()
            .get("/cookie/teste");
}
```

Veja só como o teste fica curto.

Podemos também garantir que um determinado header chegou. Por exemplo, a mesma URL cria um header chamado “novoHeader”. Validar é idêntico:

```
@Test
public void deveGerarUmHeader() {
    expect()
            .header("novo-header", "abc")
            .when()
            .get("/cookie/teste");
}
```

O Rest-Assured pode fazer muitos outros tipos de asserções, dependendo da sua necessidade. Uma boa dica é consultar o manual em https://github.com/rest-assured/rest-assured/wiki/Usage.

6. Referência Bibliográfica

Extraído de:
Aniche, M. **Testes Automatizados de Software**. São Paulo: Casa do Código, 2015.