Testing RESTful services

Ciências ULisboa

slides: João Neto

REST

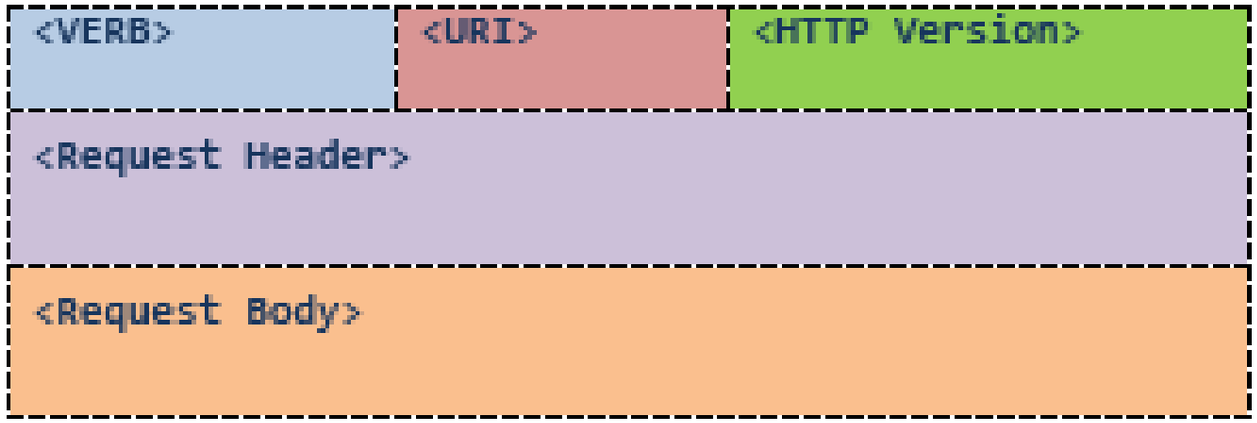
- REST stands for Representational State Transfer
  - REST is not dependent on any protocol, but almost every REST based service uses HTTP
  - HTTP functions as a request–response protocol in a client/server model (eg: browser/web-server)
  - Architectural style primarily used to build Web services
- A service based on REST is called a RESTful service
  - The focus of a RESTful service is on resources and how to provide access to these resources
  - A resource can be a webpage, a video, a Java object, …

REST Modelling

- While designing a system we need to identify what are the resources and how they relate to each other
- Need to represent these resources for transfer purposes
  - Typical choices are XML and JSON
  - Both client and server need to comprehend the format

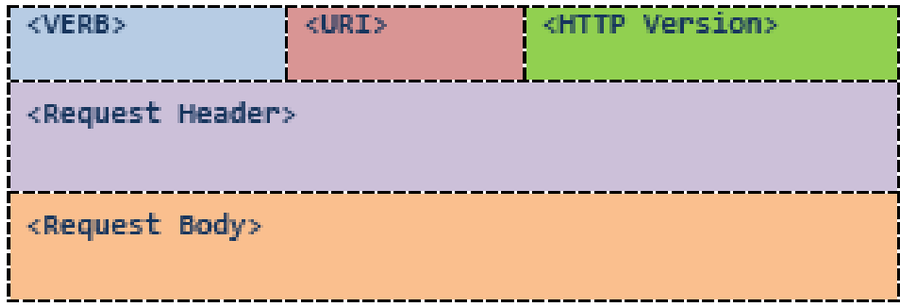
Messaging

- The client and server communicate via messages
- The client send a request, the server reply with a response
  - This is the format of a HTTP request:



  - `<VERB>` is a HTTP method: `GET`, `PUT`, `POST`, `DELETE`, `OPTIONS`, `HEAD`
  - `<URI>` is the URI resource on which the operation is to be performed
  - `<Request Header>` contains the metadata as a collection of key-value pairs of headers and their values
  - `<Request Body>` is the actual message content

POST and GET requests

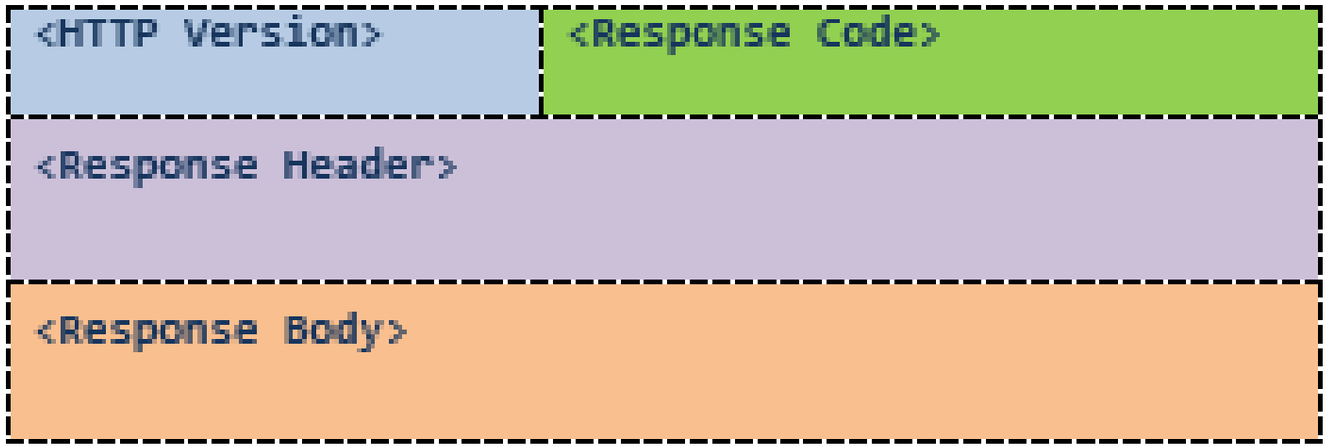



```
POST http://MyService/Person/
Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

```
GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org
Accept: text/html,application/xhtml+xml,application/xml; …
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 …
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,hi;q=0.6
```

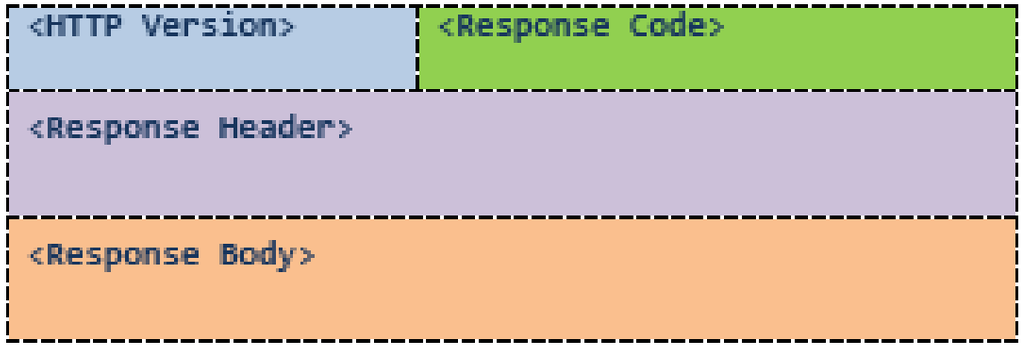
Messaging

- This is the format of a HTTP response:



- `<response code>` contains the status of the request.
  - Usually a HTTP status code (200=OK, 404=not found…)
- `<Response Header>` contains the metadata and settings about the response message.
- `<Response Body>` contains the representation if the request was successful.

GET response




```
HTTP/1.1 200 OK
Date: Sat, 23 Aug 2014 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
Cache-Control: max-age=21600, must-revalidate
Expires: Sun, 24 Aug 2014 00:31:04 GMT
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns='http://www.w3.org/1999/xhtml'>
<head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head>
<body>
...
```

Inspect HTTP requests and responses with Fiddler

Addressing Resources

- REST requires each resource to have at least one URI
- A RESTful service uses a directory hierarchy like human readable URIs to address its resources
- A URI identifies a resource or collection of resources
- The actual operation is determined by an HTTP verb
- The URI should not say anything about the operation or action
  - So, the same URI with different HTTP verbs can perform different operations

Statelessness

- A RESTful service is stateless, i.e., it does not maintain the application state for any client
  - A request cannot be depended of a previous request
  - HTTP is stateless by design
- To implement state some extra information must be added (e.g., in headers or cookies)

Best Practices

- Locators (resource identifiers) should make part of the URL, while filters (parameters) go into the query string
- General content should go into request/response body
- Use `http://service/resource/id` if you want to return a 404 error when the parameter value does not correspond to an existing resource
- However, to return an empty response template it's better to use query strings `http://service/resource?param=id`

cf. `www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api`

Testing RESTful services

- Testing these web services can be awkward given the low-level details of the HTTP protocol
  - The best way is to use available libraries
- **Rest-Assured** allows to create HTTP requests and assert their responses; it has an expressive yet simple API for this purpose
  - Uses Java Hamcrest matchers (like `equalTo`, `hasItems`)
  - Usage examples at
    `github.com/rest-assured/rest-assured/wiki/Usage`

- Import maven project `vvs_rest` for the next egs
  - This project includes simple services implemented with the JAX-RS API (it uses Jersey's implementation)
  - Run it on your Wildfly server

Defining a RESTful service

- An eg of a service using JAX-RS annotations:


```java
@Path("/c2f")
public class CtoFService {
  @Path("{c}")
  @GET
  @Produces(MediaType.APPLICATION_XML)
  public String convertCtoFfromInput(@PathParam("c") Double celsius) {
    Double fahrenheit = ((celsius * 9) / 5) + 32;

    String result = "celsius to fahrenheit converter -- REST service";
    return "<ctofservice>" +
            "<celsius>" + celsius + "</celsius>" +
            "<fahrenheit>" + fahrenheit + "</fahrenheit>" +
            "<report>" + result + "</report>" +
          "</ctofservice>";
  }
}
```

Using Rest-Assured

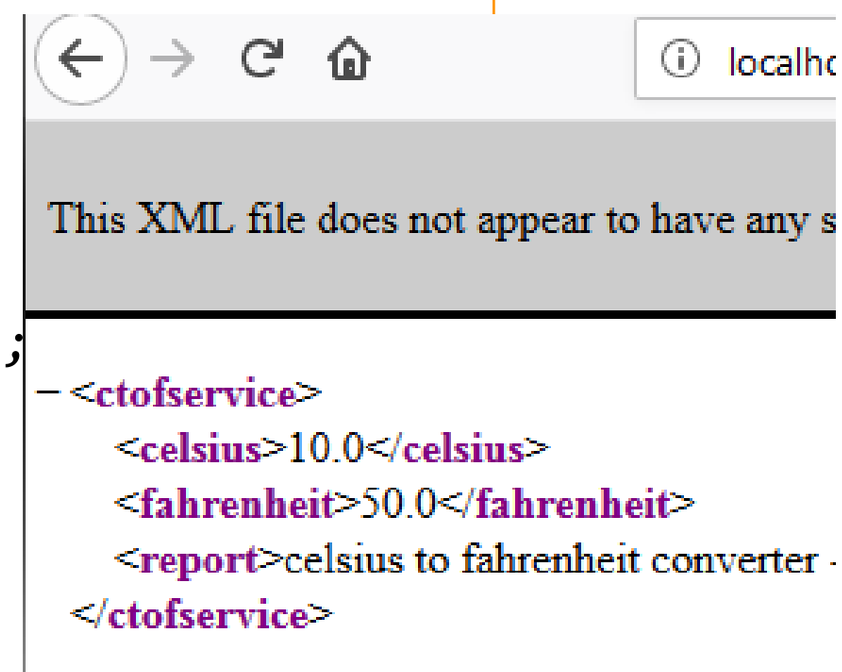
- Rest-Assured's API generate requests and analyze their HTTP responses

```
public class TemperatureTest {

  @Before
  public void setup() {
    RestAssured.baseURI = "http://localhost:8080/vvs_rest";
  }

  @Test
  public void celsiusTest() {
    get("/services/c2f/10")
        .then()
        .body("ctofservice.celsius", equalTo("10.0"));
  }

  @Test
  public void fahrenheitTest() {
    get("/services/c2f/10")
        .then()
        .body("ctofservice.fahrenheit", equalTo("50.0"));
    }
}
```



Using Rest-Assured

- Consider another service:

```
@Path("/people")
public class PeopleService {
  @GET
  @Produces("application/xml")
  public String getPeopleReport(@QueryParam("id") String user) {
    …
    switch (user) {
      case "joao":
        report = "<info>" +
                   "<firstname>" + "joao" + "</firstname>" +
                   "<lastname>" + "neto" + "</lastname>" +
                   "<hobbies>" +
                     "<hobby>" + "abs
                     "<hobby>" + "mov
                     "<hobby>" + "rea
                   "</hobbies>" +
                 "</info>";
        break;
      …
    }
    return report;
}
```

localhost:8080/vvs_rest/services/people?id=joao

```
<info>
  <firstname>joao</firstname>
  <lastname>neto</lastname>
- <hobbies>
    <hobby>abstract games</hobby>
    <hobby>movies</hobby>
    <hobby>reading</hobby>
  </hobbies>
</info>
```

Using Rest-Assured

```
public class PeopleTest {

  @Before
  public void setup() {
    RestAssured.baseURI = "http://localhost:8080/vvs_rest";
  }

  @Test
  public void pingTest() {
    when().request("GET", "/services/people?id=")
      .then()
      .assertThat()
      .statusCode(200);
  }

  @Test
  public void xmlTest() {
    given().param("id", "joao")
        .when()
        .get("/services/people")
        .then()
        .assertThat()
        .contentType(ContentType.XML);
  }
```

Using Rest-Assured

localhost:8080/vvs_rest/services/people?id=joao

```
- <info>
    <firstname>joao</firstname>
    <lastname>neto</lastname>
  - <hobbies>
      <hobby>abstract games</hobby>
      <hobby>movies</hobby>
      <hobby>reading</hobby>
    </hobbies>
  </info>
```

```java
@Test
public void responseTimeTest() {
  given().param("id", "joao")
          .when()
          .get("/services/people")
          .then()
          .time(lessThan(1000L), TimeUnit.MILLISECONDS);
}

@Test
public void firstNameTest() {
  given().param("id", "joao")
          .when()
          .get("/services/people")
          .then()
          .body("info.firstname", equalTo("joao"));
}
```

Using Rest-Assured

localhost:8080/vvs_rest/services/people?id=joao

```
- <info>
    <firstname>joao</firstname>
    <lastname>neto</lastname>
  - <hobbies>
      <hobby>abstract games</hobby>
      <hobby>movies</hobby>
      <hobby>reading</hobby>
    </hobbies>
  </info>
```

```java
@Test
public void namesTest() {
  given().param("id", "joao")
          .when()
          .get("/services/people")
          .then()
          .root("info")  // no need to repeat tag info below
          .body("firstname", equalTo("joao"))
          .body("lastname",  equalTo("neto"));
}

@Test
public void hobbiesTest() {
  given().param("id", "joao")
          .when()
          .get("/services/people")
          .then()
          .body("info.hobbies.hobby", hasItems("movies", "reading"));
}
```

Using Rest-Assured

localhost:8080/vvs_rest/services/people?id=joao

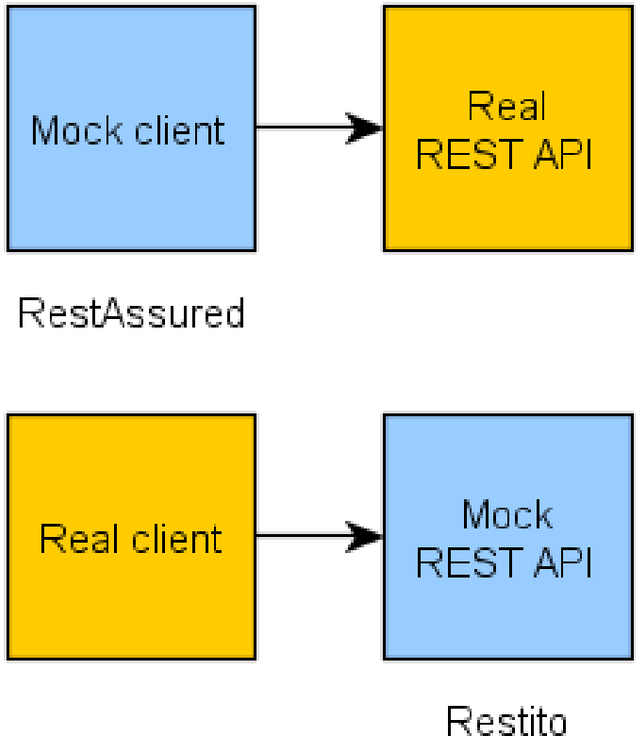
```
- <info>
    <firstname>joao</firstname>
    <lastname>neto</lastname>
  - <hobbies>
      <hobby>abstract games</hobby>
      <hobby>movies</hobby>
      <hobby>reading</hobby>
    </hobbies>
  </info>
```

```
@Test
public void hobbiesCountTest() {
  given().param("id", "joao")
          .when()
          .get("/services/people")
          .andReturn()
          .xmlPath()
          .getNode("//hobby")
          .equals(3); // it should produce three hobbies
  }

} // end test class
```

Mocking a webserver

- Restito is a tool inspired by mockito and functionally opposite to the Rest-Assured.
  - Rest-Assured simulates a client,
  - Restito mimics/stubs a rest server behavior
- Helps testing an application which makes calls to some HTTP service
- Defined conditions that need to be matched, and if so, execute certain actions (a kind of if-then conditional behavior)




```
whenHttp(stubServer)
    .match(endsWithUri("/demo"))
    .then(status(HttpStatus.OK_200),
          stringContent("Hello world!"));
```

If a HTTP request (either POST, GET,…) is sent to a URI ending with `/demo` then the stub server outputs "Hello world!" with HTTP status 200

Mocking a webserver

```
public class TodoRestitoTest {

  private static StubServer server;

  private static String todoEg =
      "<todoElement>"
    + "<id>1</id>"
    + "<summary>ArticleX</summary>"
    + "<description>Read article.html</description>"
    + "</todoElement>";

  @Before
  public void start() {
    server = new StubServer().run();
    RestAssured.port = server.getPort();

    whenHttp(server)
      .match(get("/services/todos/check"), parameter("id", "1"))
      .then(status(HttpStatus.OK_200),
            contentType("application/xml"),
            stringContent(todoEg));
  }
```

Mocking a webserver

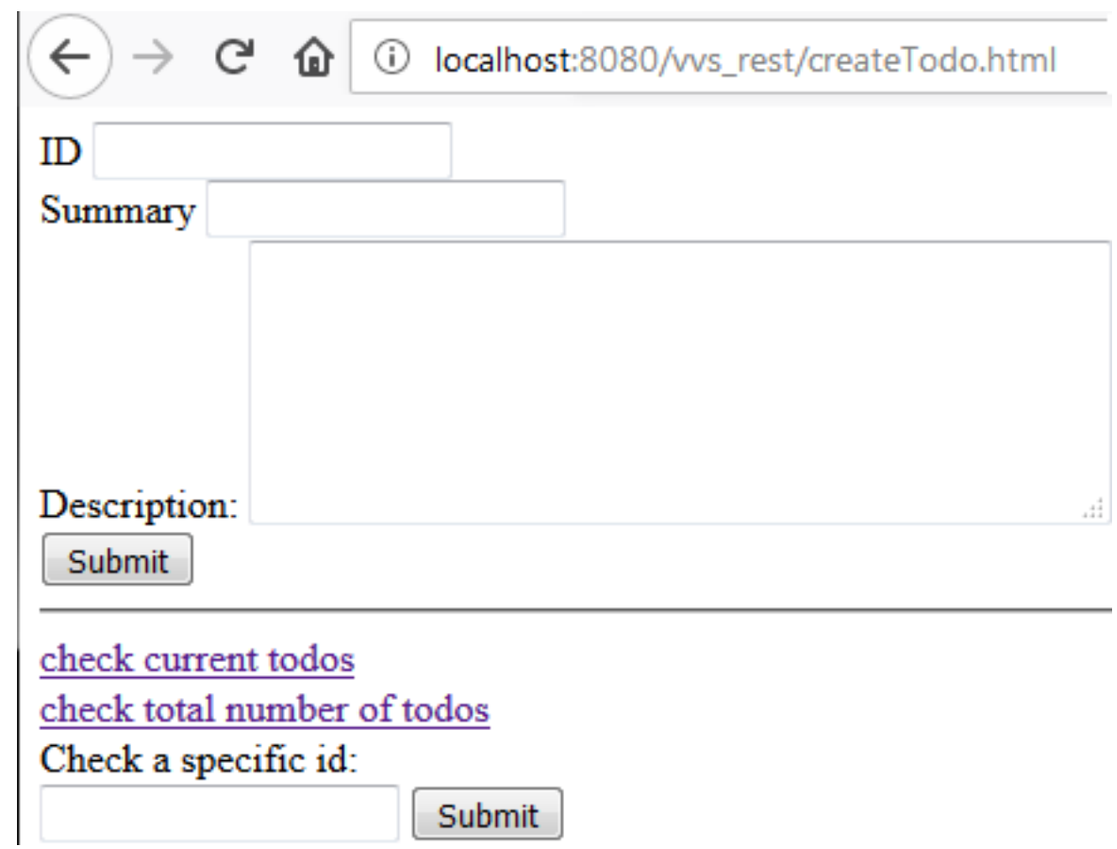
```
    @Test
    public void todoTest() {
      RestAssured
            .given()
            .param("id", "1")
            .when()
            .get("/services/todos/check")
            .then()
            .body("todoElement.summary", equalTo("ArticleX"));

      // verify that the GET request has happened just once
      verifyHttp(server)
         .once(method(Method.GET),
               uri("/services/todos/check"),
               parameter("id", "1")
          );
    }

    @After
    public void stop() {
        server.stop();
    }
}
```

Exercises

- Project `vvs_rest` includes two more RESTful services
  - `CountryService` that given a country code returns some information
  - `AddTodoService` that manages todo notes



- Explore and understand the code
- Include a remove todo service
- Use Rest-Assured to test all services

Exercise (partial) solution

```
@Before
public void setup() {
  RestAssured.baseURI = "http://localhost:8080/vvs_rest";
}

@Test
public void insertTodoTest() {
  // insert a todo
  given().param("id", "5")
         .param("summary", "summary")
         .param("description", "descriptionXPTO")
         .when()
         .post("/services/todos");

  // check if it's there
  get("/services/todos")
     .then()
     .body("todoElements.todoElement.description",
           hasItems("descriptionXPTO"));
}
```