Test Design and Automation for REST API

Ivan Katunou
About myself

• Software Testing Team Leader and Resource Manager at Epam Systems
• 11 years of experience in IT, 8 years in automated testing
• Organizer of “Morning Coffee with Automation engineers” meetups

• Past projects:
  • Epam Systems – Hyperion-Oracle
  • CompatibL – Sberbank, RMB
  • Viber Media
Agenda

1. What is special about RESTful API applications?
2. Test design and coverage
3. Automation
Out of scope

• Web services basics
• Unit, performance, security testing
• How to use tools for REST API testing
Agenda

1. What is special about RESTful API applications?
2. Test design and coverage
3. Automation
Client - Server
No UI

• Requires additional tools to interact with
Message Format

- **XML**

```xml
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

- **JSON**

```json
{  
  "empinfo":  
    {  
      "employees": [  
        {  
          "name": "James Kirk",
          "age": 40,
        },  
        {  
          "name": "Jean-Luc Picard",
          "age": 45,
        },  
        {  
          "name": "Wesley Crusher",
          "age": 27,
        }  
      ]  
    }  
}
```
Resources

RESOURCES

RESOURCES EVERYWHERE
Requirements Except Business Ones

- XSD
- JSON Schema
- WADL (rarely used)
Stateless

In RESTful applications, each request must contain all of the information necessary to be understood by the server, rather than be dependent on the server remembering prior requests.

Storing session state on the server violates the stateless constraint of the REST architecture. So the session state must be handled entirely by the client.
Transfer Protocol

• Transfer protocol for RESTful API applications in majority of the cases is HTTP(S).
• However it can also use SNMP, SMTP and others
Implementations Differ

REST is an architectural style, implementations might differ

- Roy Fielding - Representational State Transfer (REST)
- Richardson Maturity Model
- What is the Richardson Maturity Model?
Agenda

1. What is special about RESTful API applications?
2. Test design and coverage
3. Automation
Test Design and Coverage

What to test?

How to test?

What coverage is good enough?
RESTful Services by Functionality

1. Provides data

2. Provides data based on some manipulation with data provided to the service

3. Complex operations
Business Requirements

A forecast service
Business Requirements

A calculation service
Business Requirements

An Internet shop
Business Requirements – Summary

• It is crucial to verify business requirements for RESTful applications.
• Use common test design approaches (divide into submodules, boundary value analysis, equivalence partitioning, state transition testing, etc.)
Components
Requests, Responses
Endpoints

/api/devices

/api/devices

/api/devices/1

/api/devices/1

REST CLIENT

REST SERVER
Endpoints

Examples:

http://example.com/api/devices
http://example.com/api/devices/sonyz3
http://example.com/api/users
http://example.com/api/products
http://example.com/api/products/12345
http://example.com/api/products/search?q=name:Keyboard
Endpoints

• Find out which endpoints your web service provides
• Each collection endpoint needs to be tested
• At least one single resource endpoint needs to be tested for each resource type

• Negative tests:
  • Try to request endpoint that does not exist
Search, Filtering, Sorting

Endpoints that support search, filtering, sorting, etc. need to be verified with supported parameters separately and in combinations. Use boundary values, equality partitioning, pairwise testing, check special characters, max length parameters, etc.

http://example.com/api/products?q=name:Keyboard&maxPrice:200
http://example.com/api/products?year:2018
http://example.com/api/products?sort:name,asc

- Negative tests:
  - Try to request search/filtering/sorting with wrong parameter/value
Pagination,Cursor

• Endpoints that support pagination, cursors need to be verified with supported parameters separately and in combinations. Use boundary values, equality partitioning, pairwise testing.

• For negative tests try to use limit more than max one, offset out of bounds, incorrect values

• [http://example.com/api/products?limit=20&offset=100](http://example.com/api/products?limit=20&offset=100)
# Versioning

<table>
<thead>
<tr>
<th>Type</th>
<th>Sample</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td><code>{host}/api/v2/...</code></td>
<td>Minimum</td>
</tr>
<tr>
<td>Custom Header</td>
<td><code>api-version:2</code></td>
<td>Average</td>
</tr>
<tr>
<td>Custom Accept Header</td>
<td><code>Accept:application/vnd.trainmodel.v2+json</code></td>
<td>Maximum</td>
</tr>
</tbody>
</table>
Request Methods

HTTP POST /api/devices

HTTP GET /api/devices

HTTP PUT /api/devices/1

HTTP DELETE /api/devices/1
Request Methods

• GET
• POST
• PUT
• DELETE
• OPTIONS
• HEAD
• PATCH
• TRACE
• CONNECT
Request Methods

• Positive tests
  • For each collection endpoint and at least one single resource endpoint for each resource type verify supported request methods
  • Verify concurrent access to resources (DELETE and GET for example)

• Negative tests
  • For each collection endpoint and at least one single resource endpoint for each resource type try to verify behavior for not supported request methods
Request Headers
Request Headers

Headers carry information for:
- Request body (charset, content type)
- Request authorization
- etc.
Request Headers

• Positive
  • For each collection endpoint and at least one single resource endpoint verify all supported request methods with correct header values - with required headers only first, then with verifying optional ones one at a time and combinations.
  • Use boundary values, equivalence partitioning, pairwise testing. Verify special characters, Unicode text for headers, max length values.

• Negative
  • Verify behavior in case of missing required header, one at a time
  • Verify behavior in case of wrong/unsupported/empty header value
  • Verify behavior in case of not supported header
Request Body

- HTTP POST /api/devices + payload
- HTTP GET /api/devices
- HTTP PUT /api/devices/1 + payload
- HTTP DELETE /api/devices/1
Request Body

Adding user by sending POST request to http://example.com/api/users

```json
{
    "id": 1,
    "name": "Ivan",
    "surname": "Ivanov",
    "username": "Ivanovich",
    "email": "ivan.ivanov@gmail.com",
    "address": {
        "street": "Kolasas",
        "house": 5,
        "apt": 67,
        "city": "Minsk",
        "zipcode": "220005"
    },
    "phone": "+375297777777",
    "website": "ivanivanov.org",
    "company": {
        "name": "IT Tech",
        "address": {
            "street": "Melezha",
            "house": 1,
            "apt": 89,
            "city": "Minsk",
            "zipcode": "220013"
        }
    }
}
```
Request Body

• Test for endpoints and methods that support sending request body (e.g. POST, PUT)
Possible negative tests for POST/PUT requests:

- Field contains invalid value (not equals allowed value, out of bounds, etc)
- Field of wrong data type
- Field value is empty object/string
- Field value is null
- Required field is absent
- Redundant field ("Add to customFields" example)
- Empty object {}
Possible negative tests for DELETE requests:
  • Delete non-existing resource
Response Codes
# Response Codes

<table>
<thead>
<tr>
<th>1xx Informational</th>
<th>100 Continue</th>
<th>101 Switching Protocols</th>
<th>102 Processing (WebDAV)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2xx Success</strong></td>
<td>200 OK</td>
<td>201 Created</td>
<td>202 Accepted</td>
</tr>
<tr>
<td></td>
<td>202 Non-Authoritative Information</td>
<td>204 No Content</td>
<td>205 Reset Content</td>
</tr>
<tr>
<td></td>
<td>203 Partial Content</td>
<td>207 Multi-Status (WebDAV)</td>
<td>206 Already Reported (WebDAV)</td>
</tr>
<tr>
<td><strong>3xx Redirection</strong></td>
<td>300 Multiple Choices</td>
<td>301 Moved Permanently</td>
<td>302 Found</td>
</tr>
<tr>
<td></td>
<td>302 See Other</td>
<td>303 Temporary Redirect</td>
<td>304 Not Modified</td>
</tr>
<tr>
<td></td>
<td>305 (Unused)</td>
<td></td>
<td>306 Use Proxy</td>
</tr>
<tr>
<td><strong>4xx Client Error</strong></td>
<td>400 Bad Request</td>
<td>401 Unauthorized</td>
<td>402 Payment Required</td>
</tr>
<tr>
<td></td>
<td>401 Forbidden</td>
<td>403 Unauthorized</td>
<td>405 Method Not Allowed</td>
</tr>
<tr>
<td></td>
<td>404 Not Found</td>
<td>405 Method Not Allowed</td>
<td>406 Request Timeout</td>
</tr>
<tr>
<td></td>
<td>409 Conflict</td>
<td>410 Request Timeout</td>
<td>408 Request-URI Too Long</td>
</tr>
<tr>
<td></td>
<td>411 Requested Media Type</td>
<td>412 Precondition Failed</td>
<td>415 Request-URI Too Long</td>
</tr>
<tr>
<td></td>
<td>414 Request-URI Too Long</td>
<td>416 Requested Range Not Satisfiable</td>
<td>417 Expectation Failed</td>
</tr>
<tr>
<td></td>
<td>418 Requested Range Not Satisfiable</td>
<td>420 Enhance Your Calm (Twitter)</td>
<td>422 Unprocessable Entity (WebDAV)</td>
</tr>
<tr>
<td></td>
<td>423 Locked (WebDAV)</td>
<td>424 Failed Dependency (WebDAV)</td>
<td>425 Reserved for WebDAV</td>
</tr>
<tr>
<td></td>
<td>426 Upgrade Required</td>
<td>428 Precondition Required</td>
<td>429 Too Many Requests</td>
</tr>
<tr>
<td></td>
<td>431 Request Header Fields Too Large</td>
<td>444 No Response (Nginx)</td>
<td>499 Retry With (Microsoft)</td>
</tr>
<tr>
<td></td>
<td>450 Blocked by Windows Parental Controls (Microsoft)</td>
<td>451 Unavailable For Legal Reasons</td>
<td>499 Client Closed Request (Nginx)</td>
</tr>
<tr>
<td><strong>5xx Server Error</strong></td>
<td>500 Internal Server Error</td>
<td>501 Not Implemented</td>
<td>502 Bad Gateway</td>
</tr>
<tr>
<td></td>
<td>503 Service Unavailable</td>
<td>504 Gateway Timeout</td>
<td>505 HTTP Version Not Supported</td>
</tr>
<tr>
<td></td>
<td>506 Variant Also Negotiates (Experimental)</td>
<td>507 Insufficient Storage (WebDAV)</td>
<td>507 Too Many Requests</td>
</tr>
<tr>
<td></td>
<td>509 Bandwidth Limit Exceeded (Apache)</td>
<td>510 Not Extended</td>
<td>511 Network Authentication Required</td>
</tr>
<tr>
<td></td>
<td>599 Network request timeout error</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* "Top 10" HTTP Status Code. More REST service-specific information is contained in the entry.
Response Codes

• Find out which response codes might be returned by your service in which cases
• Make sure they are logically related with events (404 for resource not found, 200-201 for resource created)
• Try to reproduce such events and verify response codes are correct
• Verifying server configuration errors usually out of scope
Response Headers

REST CLIENT

HTTP POST /api/devices + payload
201

HTTP GET /api/devices
200

HTTP PUT /api/devices/1 + payload
200

HTTP DELETE /api/devices/1
200

REST SERVER
Response Headers

Find out which headers might be returned by your service. Test those ones that are related to your service.
Response Body
Response Body

Receive information on particular user using GET request to http://example.com/api/users/1

```json
{
  "id": 1,
  "name": "Ivan",
  "surname": "Ivanov",
  "username": "Ivanovich",
  "email": "ivan.ivanov@gmail.com",
  "address": {
    "street": "Kolasar",
    "house": 5,
    "apt": 67,
    "city": "Minsk",
    "zipcode": "220005"
  },
  "phone": "+375297777777",
  "website": "ivanivanov.org",
  "company": {
    "name": "IT Tech",
    "address": {
      "street": "Melezha",
      "house": 1,
      "apt": 89,
      "city": "Minsk",
      "zipcode": "220013"
    }
  }
}
```
Response Body

Verify
  • Structure of a response
  • Fields
  • Values
  • Data types
Minimal Positive Test

For adding a new user

```json
{
  "id": 1,
  "name": "Ivan",
  "surname": "Ivanov",
  "username": "Ivanovich",
  "email": "ivan.ivanov@gmail.com",
  "address": {
    "street": "Kolasa",
    "house": 5,
    "apt": 67,
    "city": "Minsk",
    "zipcode": "220005"
  },
  "phone": "+375297777777",
  "website": "ivanivanov.org",
  "company": {
    "name": "IT Tech",
    "address": {
      "street": "Melezha",
      "house": 1,
      "apt": 89,
      "city": "Minsk",
      "zipcode": "220013"
    }
  }
}
```
Minimal Positive Test

- Add only required request headers
- Add only required fields in the request with some correct values
- Send POST request
- Verify the response code
- Verify the body of the response
- Verify service-specific headers
- *In case response body is not returned on POST, send GET request for the new item or get from DB
Other Positive Tests

• Required fields only in request body
• Test values for each of the fields in separate tests. Verify full response body

{  
  "id": 1,  
  "name": "Ivan",  
  "email": "ivan.ivanov@gmail.com"  
}
Other Positive Tests

• Required fields plus one optional

• Test values for the optional field. Verify full response body!

```json
{
    "id": 1,
    "name": "Ivan Ivanov",
    "email": "ivan.ivanov@gmail.com",
    "surname": "Ivanov"
}
```
Complex Request Body

• Makes sense to add at least one test with all possible fields
• For testing combinations of fields pairwise testing might be leveraged
Boundary Values for Field Values

Find out boundary values - might depend on XML/JSON restrictions, DB, other components
Transformations

{  
  "id": 1,  
  "firstName": "Ivan",  
  "lastName": "Ivanov",  
  "email": "ivan.ivanov@gmail.com"  
}

{  
  "id": 1,  
  "name": "Ivan Ivanov",  
  "email": "ivan.ivanov@gmail.com"  
}
State Transition Testing
Caching and Rate Limits
Rate Limits and Caching

• Verify that cached responses received faster
• Verify cache expiration time (right before and after)
• Find out rate limits for different methods/endpoints
• Try to reproduce max allowed rate limit
• Try to reproduce more than max allowed rate limit
Integration with Third Party

Stubs can be used to make testing easier and faster, not dependent on actual services
Coverage

I will test our RESTful API service thoroughly.
There are billions of combinations to verify. And we have release in a month.

I will leverage automation.

and it will take only half a year.
Project Management Triangle

- Schedule
- Quality
- Cost
- Scope
Third-Party Components

No need to test third-party components. Test your application only!
Risks
Risks

• 1. No testing at all
• 2. No requests headers testing
• 3. No separate fields of request body testing
• 4. No field values, types, number of occurrence in request body testing
• 5. No status code testing
• 6. No response body testing
• 7. Testing only specific fields in the response body
• 8. No response headers testing
Trusted Boundaries
Trusted Boundaries

- Find out trusted boundaries. “Trust CMS” example
- Add notes for trusted boundaries for tracking
- Verify that those boundaries are still relevant
Using XSD, JSON Schema

• In case you have XSD/JSON Schema/WADL, you can validate messages against them and limit coverage in your tests accordingly
• In case web service uses XSD/JSON Schema/WADL validation itself additional verification might be minimized. You still need to make sure that application actually does it (check logs for example)
Difficult to Verify Values

• E. g. hash values, current time, random generated values
Test Automation

Including parallel tests
Test Pyramid

UI
Service
Unit

Turtle

Rabbit

$\$$

$\$$
Test Coverage - Summary

• Main challenge - great variety of combinations parameters and values
• Depends on time/resources you have and quality you need
• No need to test third-party components. Test you application only!
• Depends on risks you and the PO are agreed on. How service is used
• Depends on trusted boundaries
• Depends on additional verification using XSD, JSON Schema by the service
• Number of field values difficult to verify
• Depends on automated testing, parallel tests execution
• Depends on unit, UI tests coverage
Test Strategy
Agenda

1. What is special about RESTful API applications?
2. Test design and coverage
3. Automation
Postman

• Use standalone version
• Functionality to reduce routine:
  1. Pre-request scripts
  2. Test scripts
  3. Variables, environments, globals
  4. Sharing collections
  5. Collection runs
  6. Mock server
  7. Generate code for cUrl, Java, Python, C#, etc.
  8. Newman
Java Libraries

- JUnit/TestNG
- RestAssured, Apache HTTP Client, other tools and libraries
- Gson/Jackson (choose the one that is not used by your team’s developers)
- JsonAssert
RestAssured

- **Usage**
- **Not thread safe** (there is a [PR](https://github.com) that might fix it)
Test Data

Where should it be located?

In which format?
Text

• Can be saved as regular text/JSON/XML files at resources folder of your test project or in DB
• Hard to change
• Can be used for a limited number of tests with all fields and values pre-defined
• Maintenance might become a pain in the neck
Text + Templates

Apache FreeMarker

Values can be stored in CSV file

Still requires much effort for maintenance
Object Mapping

JSON

{"message": "My message"}

POJO

```java
public class Message {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Deserialization

```java
Message message = get("/message").as(Message.class);
```
Object Mapping

Serialize

```java
Message message = new Message();
message.setMessage("My message");
given().
    contentType("application/json").
    body(message).
when().
    post("/message");
```
Object Mapping

{  
  "id": 1,  
  "name": "Ivan",  
  "surname": "Ivanov",  
  "username": "Ivanovich",  
  "email": "ivan.ivanov@gmail.com",  
  "address": {  
    "street": "Kolasa",  
    "house": 5,  
    "apt": 67,  
    "city": "Minsk",  
    "zipcode": "220005"  
  },  
  "phone": "+375297777777",  
  "website": "ivanivanov.org",  
  "company": {  
    "name": "IT Tech",  
    "address": {  
      "street": "Melezha",  
      "house": 1,  
      "apt": 89,  
      "city": "Minsk",  
      "zipcode": "220013"  
    }  
  }  
}
Object Mapping

jsonschema2pojo

```java
public class User {
    private Integer id;
    private String name;
    private String surname;
    private String username;
    private String email;
    private Address address;
    private String phone;
    private String website;
    private Company company;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public User withId(Integer id) {
        this.id = id;
        return this;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

public class Company {
    private String name;
    private Address address;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Company withName(String name) {
        this.name = name;
        return this;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public Company withAddress(Address address) {
        this.address = address;
        return this;
    }
}

public class Address {
    private String street;
    private Integer house;
    private Integer apt;
    private String city;
    private String zipcode;

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public Address withStreet(String street) {
        this.street = street;
        return this;
    }

    public Integer getHouse() {
        return house;
    }

    public void setHouse(Integer house) {
        this.house = house;
    }

    public Address withHouse(Integer house) {
        this.house = house;
        return this;
    }
}
```
Object Mapping

```java
Address userAddress = new Address();
userAddress.setCity("Minsk");
userAddress.setZipcode("220005");

Address companyAddress = new Address();
companyAddress.setCity("Minsk");
companyAddress.setZipcode("220013");

Company company = new Company();
company.setName("IT Tech");
company.setAddress(companyAddress);

User user = new User();
user.setId(1);
user.setName("Ivan");
user.setEmail("ivan.ivanov@gmail.com");
user.setAddress(userAddress);
user.setCompany(company);
```
public class UserBuilder {

    User user = new User();

    public UserBuilder withRequiredFieldsOnly() {
        user.setId(generateRandomId());
        user.setName("Ivan");
        user.setEmail("ivan.ivanov@gmail.com");
        return this;
    }

    public UserBuilder withAllFields() {
        withRequiredFieldsOnly();
        // add the rest of the fields. Use builders for Address and Company to define corresponding fields
        // ...
        return this;
    }

    // builder methods that add one field at a time.
    // ...

    public User build() {
        return user;
    }
}
Object Mapping + Builder

```java
User user1 = new UserBuilder().withRequiredFieldsOnly().build();
System.out.println(user1.getName());

User user2 = new UserBuilder().withAllFields().build();
System.out.println(user2.getName());
```
Object Mapping + Builder

• The most convenient approach to work with test data
• Working with business objects in tests. Technical details are encapsulated
• Saving test data in files/DB is no longer needed
• Builder pattern allows chaining adding details for business objects
Object Mapping + Builder

Cannot be (easily) used in the following cases:

- Custom fields
- Adding null values for specific fields (but not all)
- Wrong data type
Gson/Jackson Objects

```java
JsonObject userAddress = new JsonObject();
userAddress.addProperty( property: "street", value: "Kolasan" );
userAddress.addProperty( property: "house", value: 5 );
userAddress.addProperty( property: "unknownProperty", value: "value" );
userAddress.addProperty( property: "house", value: null );

System.out.println(userAddress.get("street").getAsString());
System.out.println(userAddress.get("house").getInt());
```
Gson/Jackson Objects

```java
public class Address extends DomainObject {
    public static final String STREET_FIELD = "street";

    public boolean hasStreet() {
        return jsonObject.has(STREET_FIELD);
    }

    public String getStreet() {
        return jsonObject.get(STREET_FIELD).isNull() ? null : jsonObject.get(STREET_FIELD).getAsString();
    }

    public void setStreet(String street) {
        jsonObject.addProperty(STREET_FIELD, street);
    }

    // other getters and setters for Address fields
    // ...
}

public abstract class DomainObject {

    protected JSONObject jsonObject = new JSONObject();

    public DomainObject() {
        return jsonObject;
    }

    public JSONObject getJsonObject() {
        return jsonObject;
    }

    public void setJsonObject(JSONObject jsonObject) {
        thisJsonObject = jsonObject;
    }

    public boolean hasCustomStringField(String fieldName) {
        return jsonObject.has(fieldName);
    }

    public String getCustomStringField(String fieldName) {
        return jsonObject.get(fieldName).isNull() ? null : jsonObject.get(fieldName).getAsString();
    }

    public void setCustomStringField(String fieldName, String value) {
        jsonObject.addProperty(fieldName, value);
    }

    Address address = new Address();
    address.setStreet(null);
    address.setCustomStringField(fieldName: "unknownProperty", value: "value");
    address.setStreet("Kolama");

    if (address.hasStreet()) {
        System.out.println(address.getStreet());
    }

    if (address.hasCustomStringField(fieldName: "unknownProperty")) {
        System.out.println(address.getCustomStringField(fieldName: "unknownProperty");
    }
```
public class AddressBuilder {

    Address address = new Address();

    public AddressBuilder withRequiredFieldsOnly() {
        address.setStreet("Kolasa");
        return this;
    }

    public AddressBuilder withAllFields() {
        withRequiredFieldsOnly();
        // add the rest of the fields
        // ...
        return this;
    }

    // builder methods that add one field at a time.
    // ...

    public Address build() { return address; }

    Address address2 = new AddressBuilder().withRequiredFieldsOnly().build();
    System.out.println(address2.getStreet());

    Address address3 = new AddressBuilder().withAllFields().build();
    System.out.println(address3.getStreet());
Gson/Jackson Objects

• The most flexible approach for working with JSON/XML in Java
• Can be used where Object Mapping cannot be (easily) applied
• Requires more effort on developing business objects comparing to Object Mapping
• Business object cannot be used for serialization/deserialization. Use wrapped JsonObject instead with its setter and getter
JSON Schema Validation

RestAssured JSON Schema Validation

get("/products") .then() .assertThat() .body(matchesJsonSchemaInClasspath("products-schema.json"));
@Test
public void exampleTest() {
    stubFor(get(urlEqualTo("/my/resource"))
        .withHeader("Accept", equalTo("text/xml"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "text/xml")
            .withBody("<response>Some content</response>")));

    Result result = myHttpServiceCallingObject.doSomething();

    assertTrue(result.wasSuccessful());

    verify(postRequestedFor(urlMatching("/my/resource/[a-z0-9]+"))
        .withRequestBody(matching(".*<message>1234</message>.*")
        .withHeader("Content-Type", notMatching("application/json")));
}
Using DB

- Update data and make verifications in DB
- Disadvantages: too coupled with DB schemas that might change often and significantly
Independent Tests

Bad Examples:

• POST object in one test, GET and verify in another one
• Add embedded object in one test, verify it in another one
• Shared data with not thread safe access (test data, configs, connection to DB, etc)
Using Java 8 Features

Lambdas and Stream API are rather useful for working with collections of data
Kotlin

*Kotlin and API tests* presentation by Roman Marinsky
Automation – Tools to Investigate

- **SoapUI** – REST and SOAP testing tool
- **RAML/Swagger** - for documentation and contract testing
- **Epam JDI HTTP** - Web services functional testing, RestAssured wrapper
- **Karate** - Web services functional testing using BDD style, based on Cucumber - JVM
- **AWS Lambda** - might be used to run API tests in parallel
Useful links

- Presentation samples
- Подходы к проектированию RESTful API
- Testing RESTful Services in Java: Best Practices
- REST CookBook
- Status codes
- The JavaScript Object Notation (JSON) Data Interchange Format
- Introducing JSON
- OData
- Тестирование API используя TypeScript. Пример технологического стека и архитектуры
Thank You! Any Questions?
Contacts

- Ivan Katunou, Software Testing Team Leader / Resource Manager at Epam Systems (Coconut Palm test automation team)

- ivan.katunou@gmail.com
- @IvanKatunou (Telegram)
- +375 29 259 56 42 (Viber, GSM)