

# 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

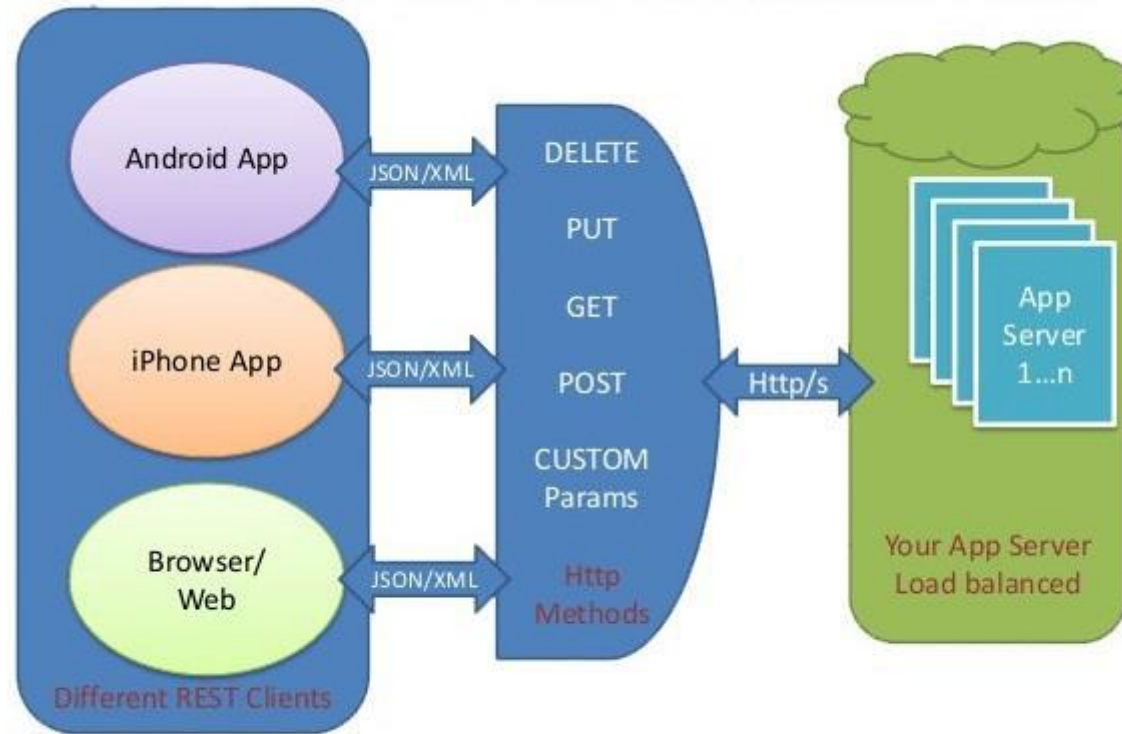
# Out of scope

- 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](#)
- [JSON](#)
- Others

## 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

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



# Resources



# Requirements Except Business Ones

- XSD
- JSON Schema
- WADL (rarely used)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="brightstar">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="magnitude" type="xs:decimal"/>
        <xs:element name="distance" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
{
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "age": {
      "description": "Age in years",
      "type": "integer",
      "minimum": 0
    }
  },
  "required": ["firstName", "lastName"]
}
```

# 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?



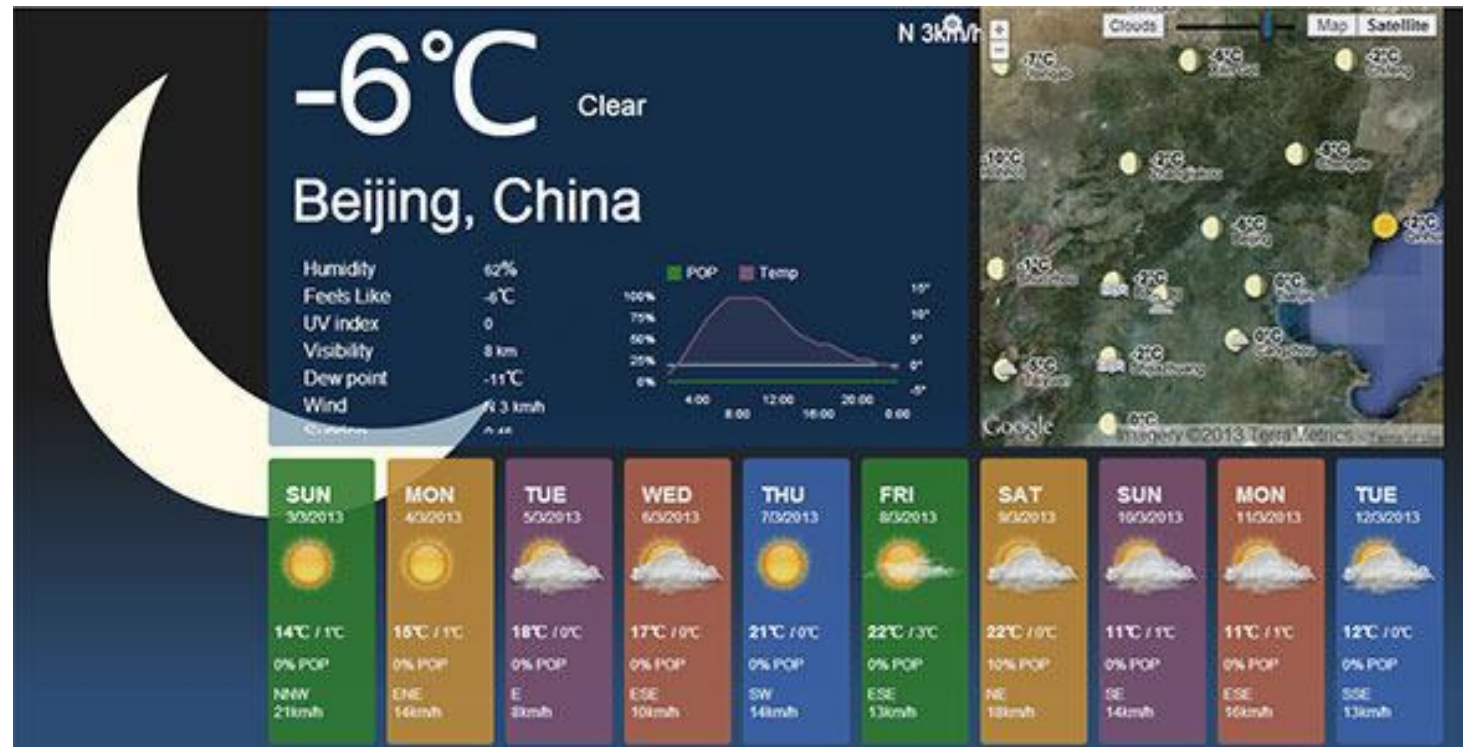
# RESTful Services by Functionality

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

5+5

About 25,270,000,000 results (0.35 seconds)

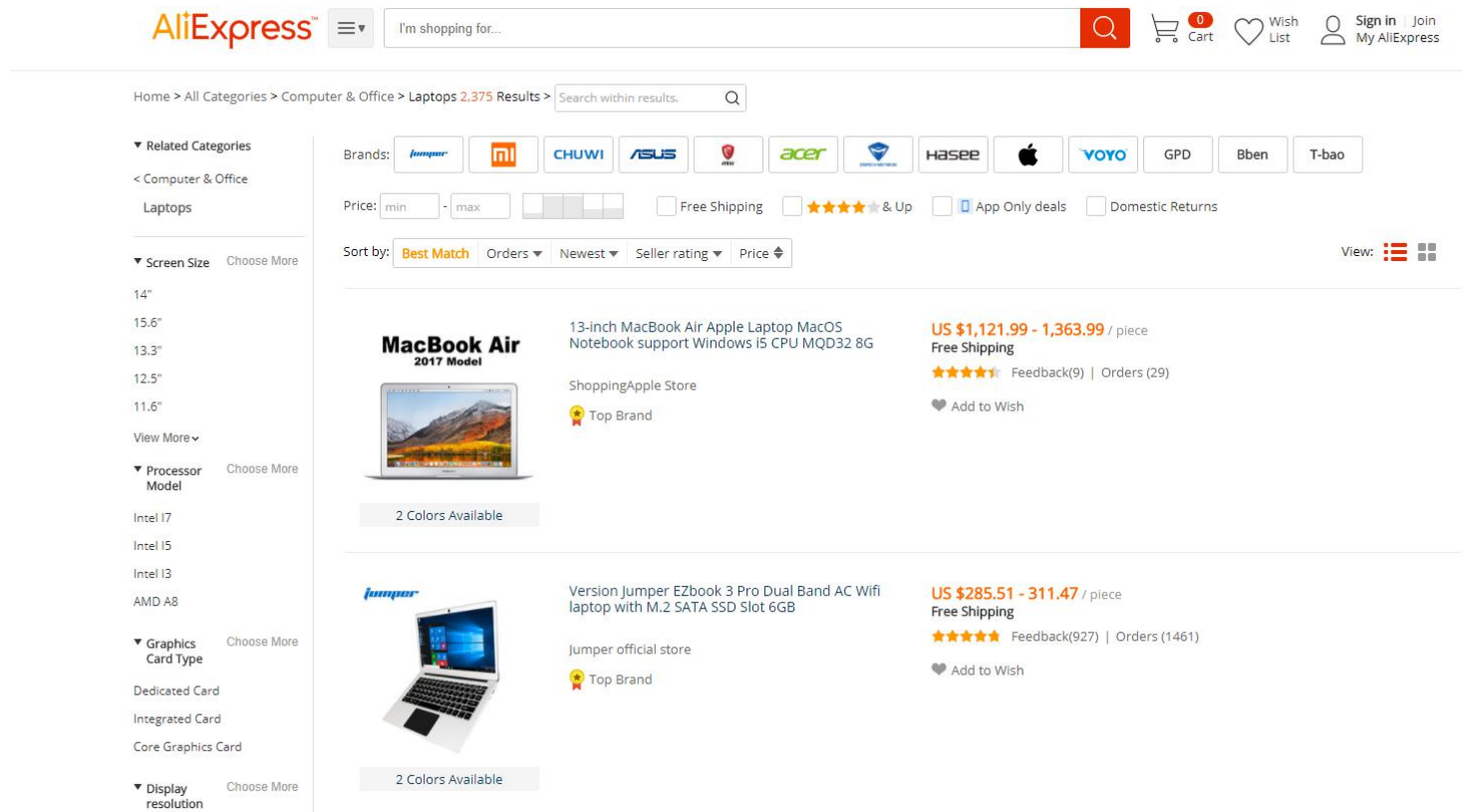
5 + 5 =

10

Rad		x!	(	)	%	CLEAR
sin <sup>-1</sup>	sin	√	7	8	9	÷
cos <sup>-1</sup>	cos	ln	4	5	6	×
tan <sup>-1</sup>	tan	log	1	2	3	-
π	e	x <sup>y</sup>	0	.	=	+

# Business Requirements

## An Internet shop



The screenshot displays the AliExpress website interface for laptop searches. The top navigation bar includes the AliExpress logo, a search bar with the placeholder "I'm shopping for...", and icons for a shopping cart (0 items), wish list, and user account (Sign in / Join My AliExpress).

The breadcrumb trail indicates the current location: Home > All Categories > Computer & Office > Laptops 2,375 Results >. A search bar within the results area is also present.

**Left Sidebar (Filters):**

- Related Categories:** Computer & Office > Laptops
- Screen Size:** Choose More. Options: 14", 15.6", 13.3", 12.5", 11.6". View More.
- Processor Model:** Choose More. Options: Intel i7, Intel i5, Intel i3, AMD A8.
- Graphics Card Type:** Choose More. Options: Dedicated Card, Integrated Card, Core Graphics Card.
- Display resolution:** Choose More.

**Main Content Area (Filters and Results):**

**Brands:** Jumper, Chuwi, ASUS, Acer, Hasee, Apple, Voyo, GPD, Bben, T-bao.

**Price:** min - max. ☐ Free Shipping. ☐ ★★★★★ & Up. ☐ App Only deals. ☐ Domestic Returns.

**Sort by:** Best Match (selected), Orders, Newest, Seller rating, Price. **View:** List/Grid.

**Product Listings:**

- MacBook Air 2017 Model:** 13-inch MacBook Air Apple Laptop MacOS Notebook support Windows i5 CPU MQD32 8G. Price: US \$1,121.99 - 1,363.99 / piece. Free Shipping. ★★★★★ Feedback(9) | Orders (29). Add to Wish. 2 Colors Available.
- Jumper EZbook 3 Pro:** Version Jumper EZbook 3 Pro Dual Band AC Wifi laptop with M.2 SATA SSD Slot 6GB. Price: US \$285.51 - 311.47 / piece. Free Shipping. ★★★★★ Feedback(927) | Orders (1461). Add to Wish. 2 Colors Available.

# 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



REST  
CLIENT

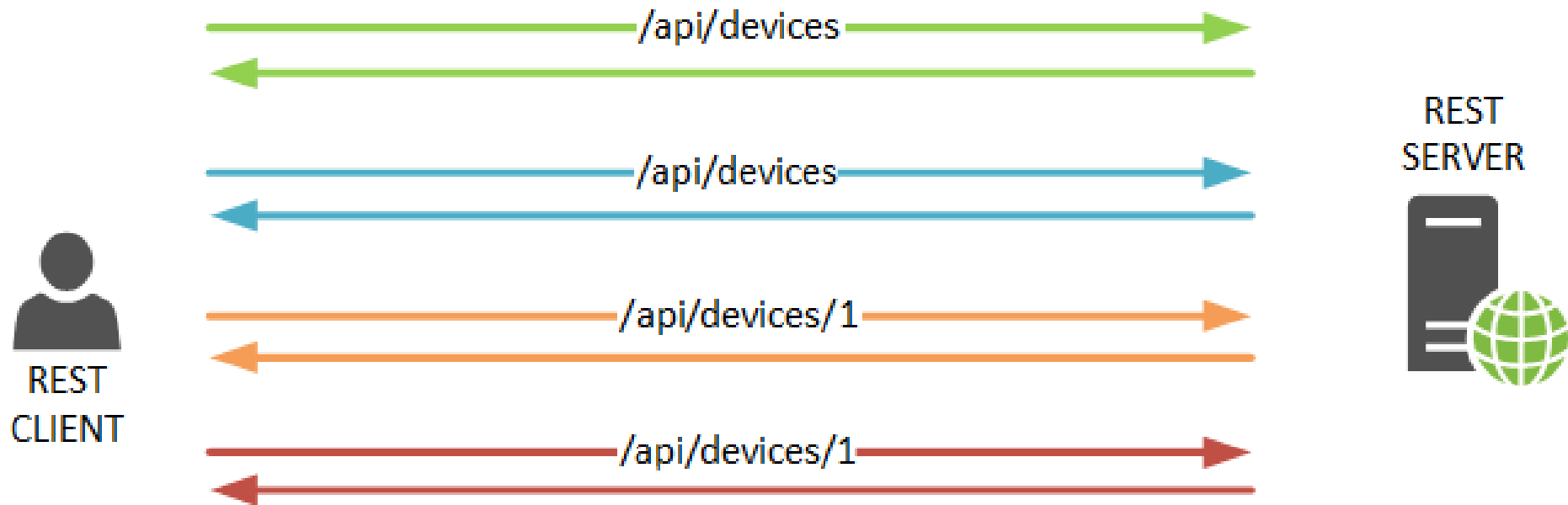
REST  
SERVER



# Requests, Responses



# Endpoints



# 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

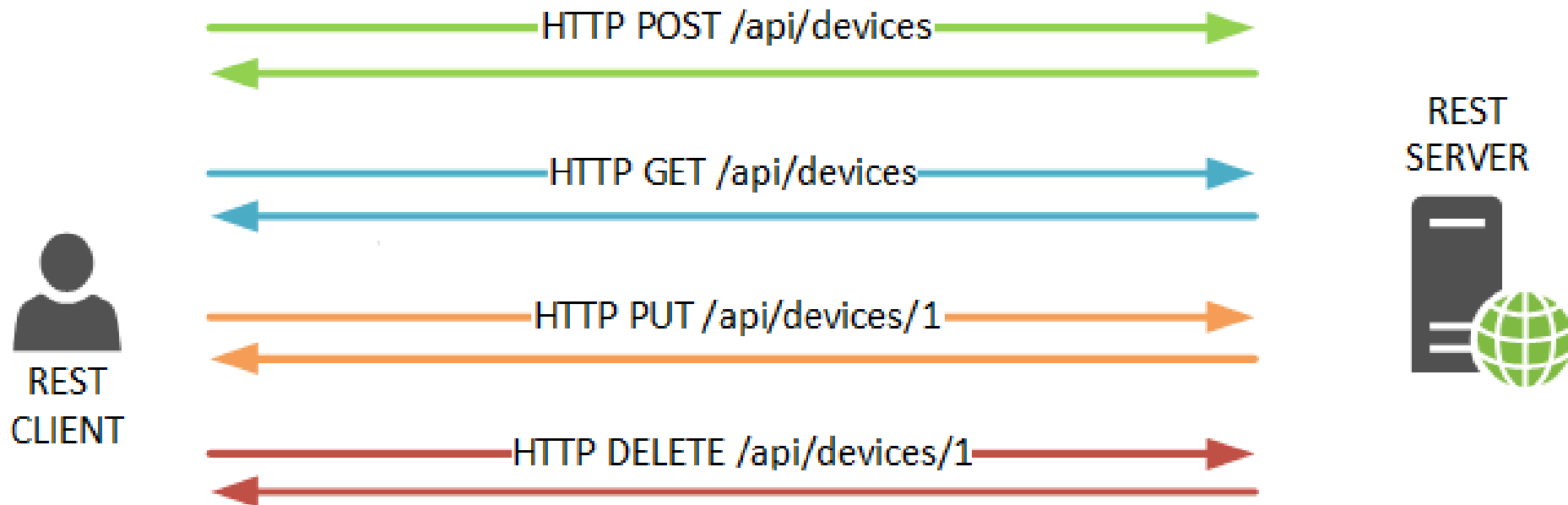
# Pagination, Cursors

- 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>

# Versioning

Type	Sample	Complexity
URL	<code>{host}/api/v2/...</code>	Minimum
Custom Header	<code>api-version:2</code>	Average
Custom Accept Header	<code>Accept:application/vnd.trainmodel.v2+json</code>	Maximum

# Request Methods



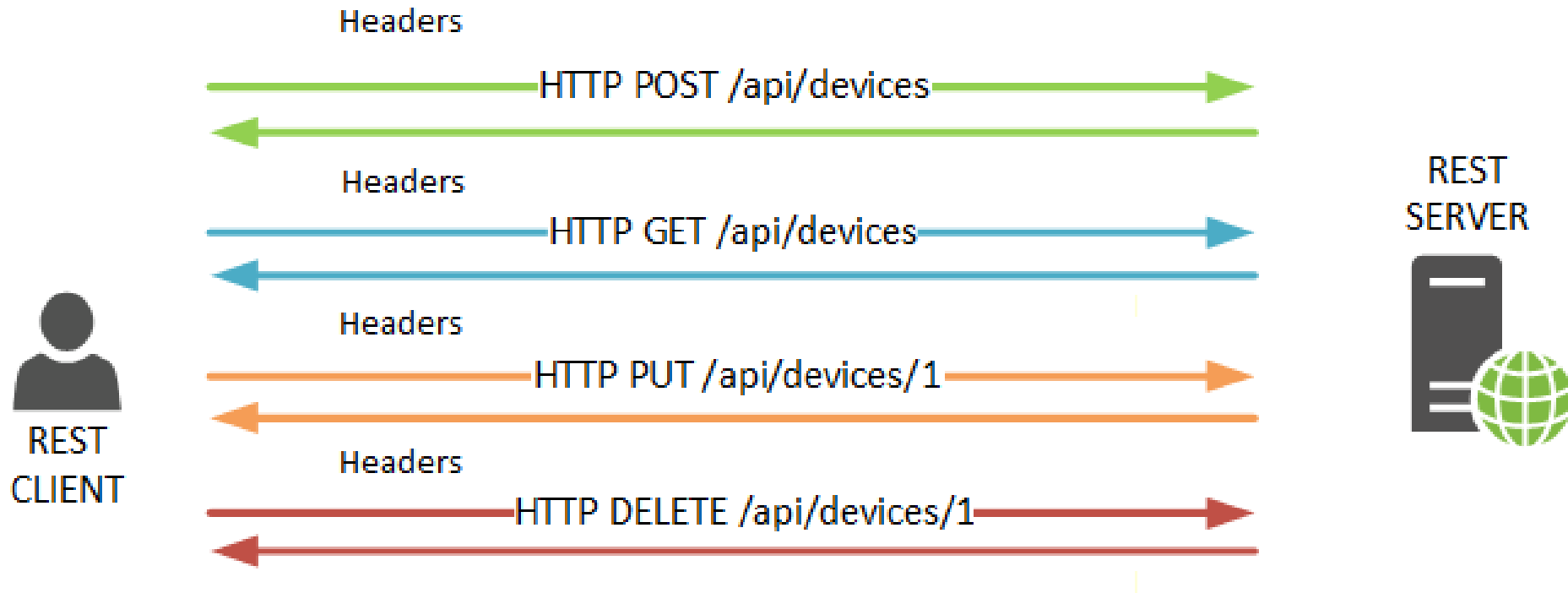
# Request Methods

- 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

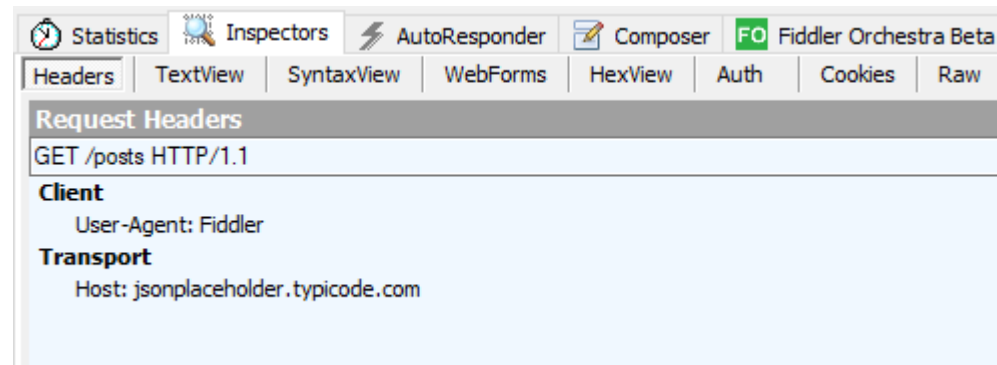




# Request Headers

Headers carry information for:

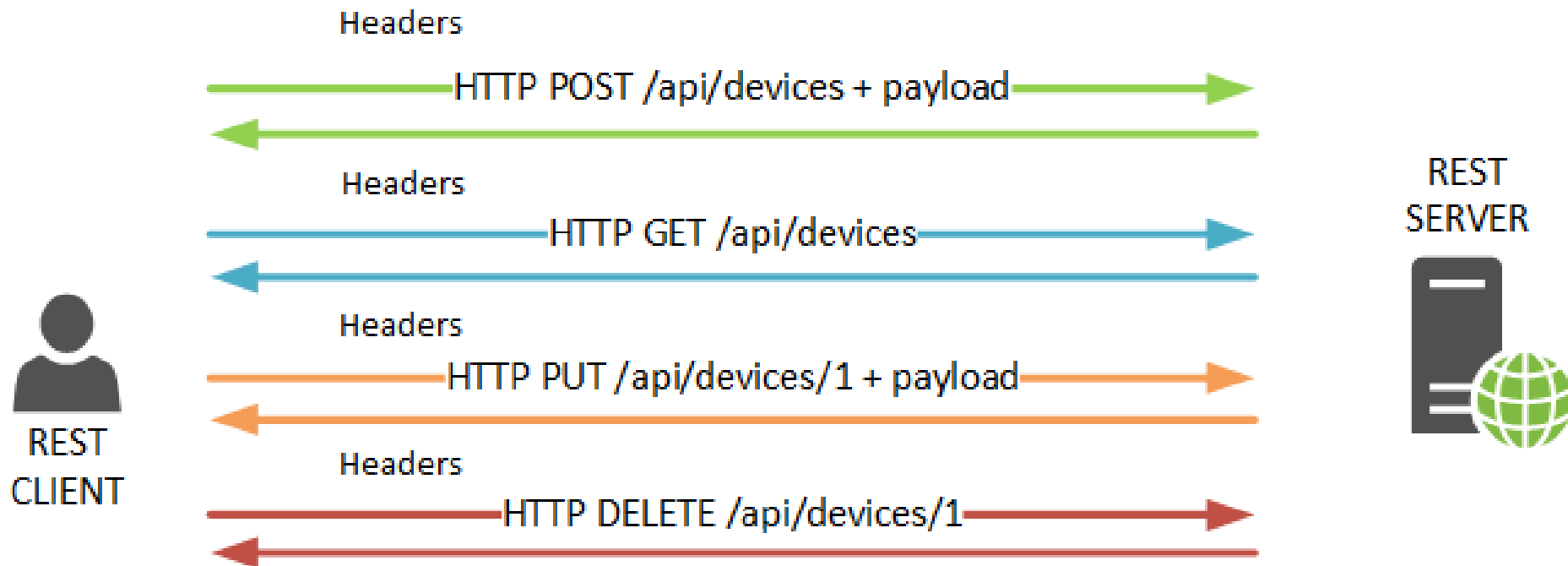
- Request body (charset, content type)
- Request authorization
- etc.



# 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



# Request Body

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



```
{
  "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"
    }
  }
}
```

# Request Body

- Test for endpoints and methods that support sending request body (e. g. POST, PUT)

# Request Body

Possible negative tests for POST/PUT requests:

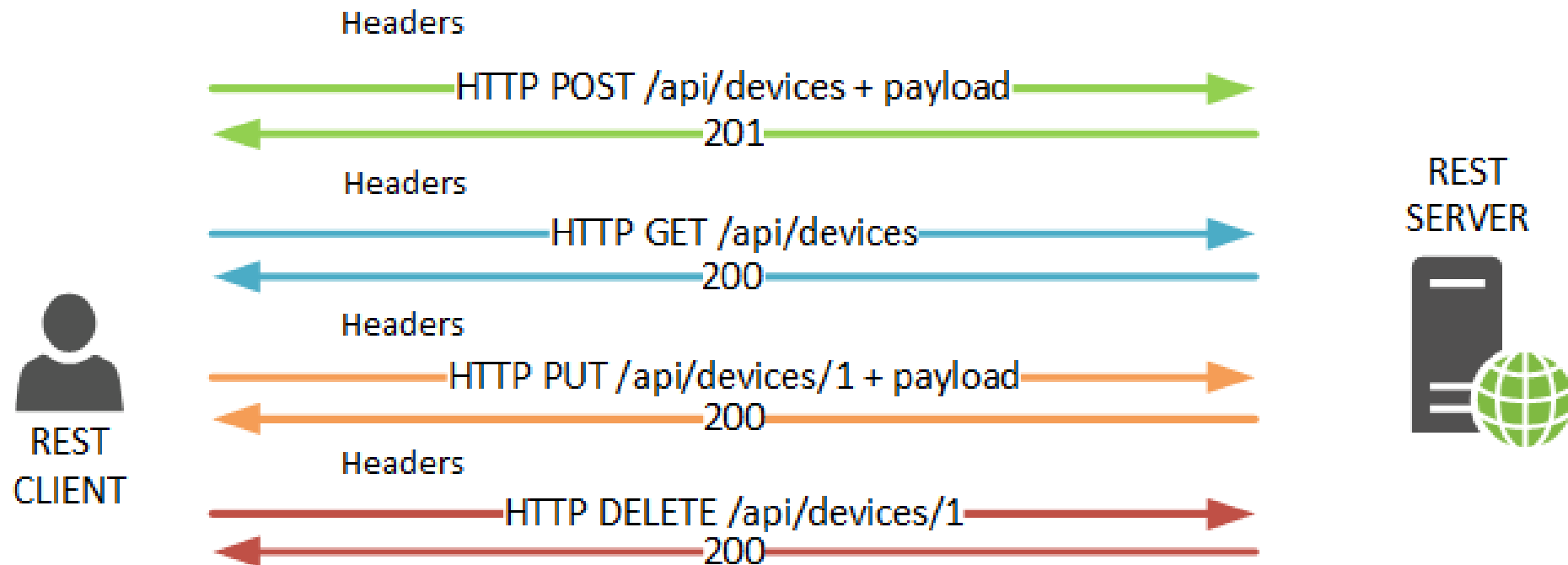
- Field contains invalid value (not equals allowed value, out of bounds, etc)
- Field of wrong data type
- Field value is empty object/string
- Field value is null
- Required field is absent
- Redundant field (“Add to customFields” example)
- Empty object {}
- Invalid XML/JSON
- No data
- Post already existing resource

# Request Body

Possible negative tests for DELETE requests:

- Delete non-existing resource

# Response Codes





# Response Codes

## 1xx Informational

100 Continue

## 2xx Success

★ 200 OK

203 Non-Authoritative Information

206 Partial Content

226 IM Used

## 3xx Redirection

300 Multiple Choices

303 See Other

306 (Unused)

## 4xx Client Error

★ 400 Bad Request

★ 403 Forbidden

406 Not Acceptable

★ 409 Conflict

412 Precondition Failed

415 Unsupported Media Type

418 I'm a teapot (RFC 2324)

423 Locked (WebDAV)

426 Upgrade Required

431 Request Header Fields Too Large

450 Blocked by Windows Parental Controls (Microsoft)

101 Switching Protocols

★ 201 Created

★ 204 No Content

207 Multi-Status (WebDAV)

301 Moved Permanently

★ 304 Not Modified

307 Temporary Redirect

★ 401 Unauthorized

★ 404 Not Found

407 Proxy Authentication Required

410 Gone

413 Request Entity Too Large

416 Requested Range Not Satisfiable

420 Enhance Your Calm (Twitter)

424 Failed Dependency (WebDAV)

428 Precondition Required

444 No Response (Nginx)

451 Unavailable For Legal Reasons

102 Processing (WebDAV)

202 Accepted

205 Reset Content

208 Already Reported (WebDAV)

302 Found

305 Use Proxy

308 Permanent Redirect (experimental)

402 Payment Required

405 Method Not Allowed

408 Request Timeout

411 Length Required

414 Request-URI Too Long

417 Expectation Failed

422 Unprocessable Entity (WebDAV)

425 Reserved for WebDAV

429 Too Many Requests

449 Retry With (Microsoft)

499 Client Closed Request (Nginx)

## 5xx Server Error

★ 500 Internal Server Error

503 Service Unavailable

506 Variant Also Negotiates (Experimental)

509 Bandwidth Limit Exceeded (Apache)

598 Network read timeout error

501 Not Implemented

504 Gateway Timeout

507 Insufficient Storage (WebDAV)

510 Not Extended

599 Network connect timeout error

502 Bad Gateway

505 HTTP Version Not Supported

508 Loop Detected (WebDAV)

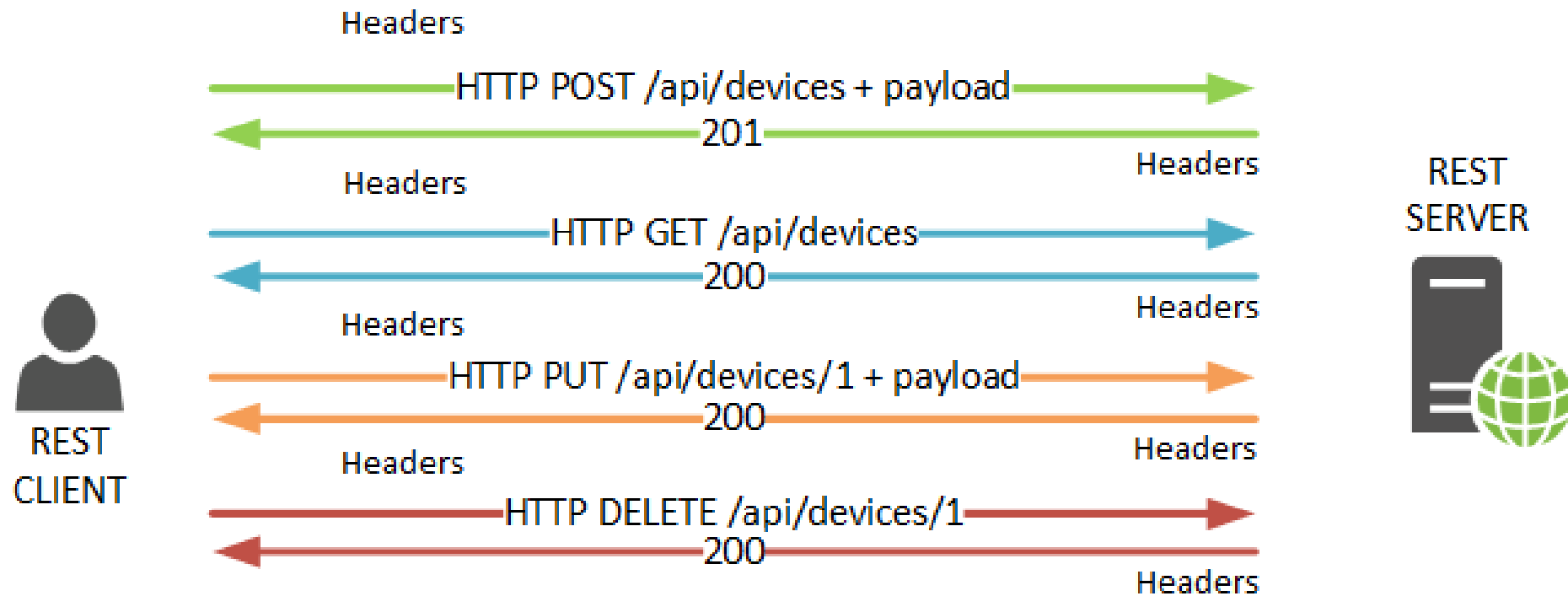
511 Network Authentication Required

★ "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



# Response Headers

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

Transformer | **Headers** | TextView | SyntaxView | ImageView | HexView | WebView | Auth | Caching | Cookies | Raw | JSON | XML

**Response Headers**

HTTP/1.1 200 OK

**Cache**

- Cache-Control: public, max-age=14400
- Date: Sat, 24 Feb 2018 18:25:52 GMT
- Expires: Sat, 24 Feb 2018 22:25:52 GMT
- Pragma: no-cache
- Vary: Origin, Accept-Encoding

**Cookies / Login**

- Set-Cookie: \_\_cfduid=dbfd8e8623f64e1c6d73064cc3ad327cc1519496752; expires=Sun, 24-Feb-19 18:25:52 GMT; path=/; domain=.typicode.com; HttpOnly

**Entity**

- Content-Type: application/json; charset=utf-8
- Etag: W/"6b80-Ybsq/K6GwwqrYkAsFqxqDXGC7DoM"

**Miscellaneous**

- CF-Cache-Status: HIT
- CF-RAY: 3f2478510cad871b-ARN
- Expect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
- Server: cloudflare
- X-Powered-By: Express

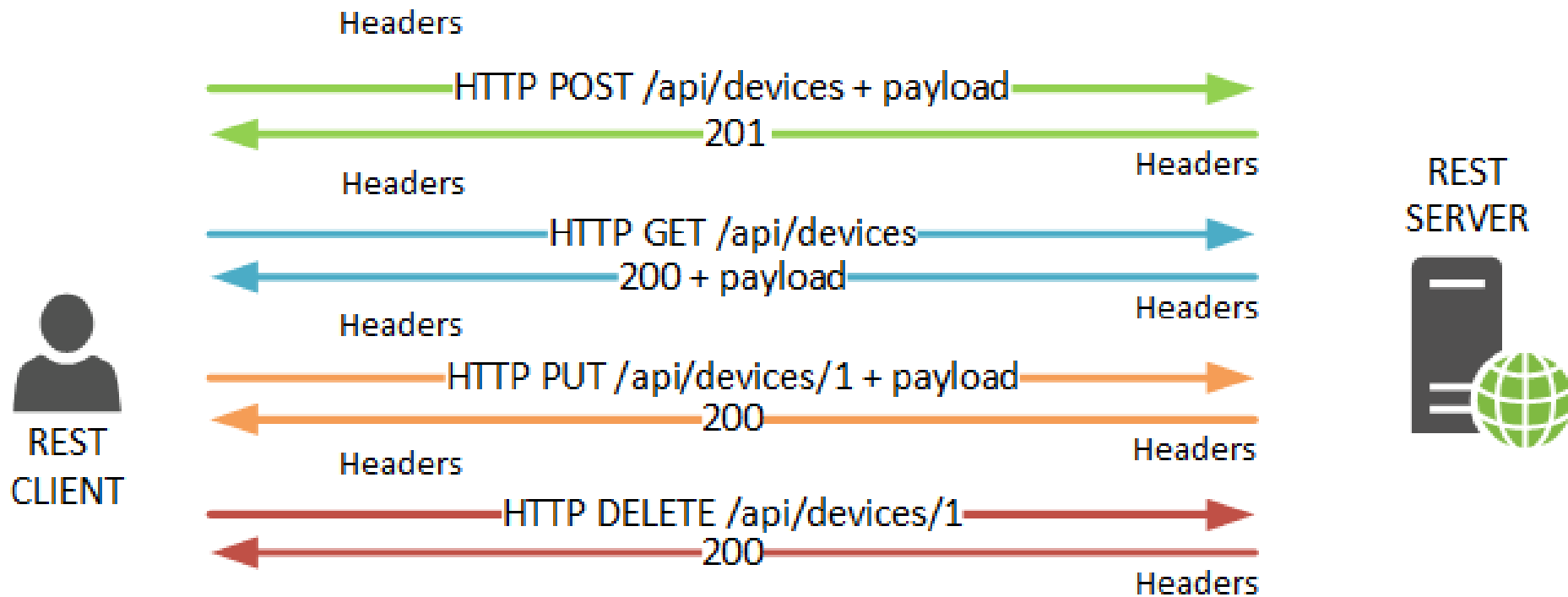
**Security**

- Access-Control-Allow-Credentials: true
- X-Content-Type-Options: nosniff

**Transport**

- Connection: keep-alive
- Transfer-Encoding: chunked
- Via: 1.1 vegur

# Response Body



# Response Body

Receive information on particular user  
using GET request to

<http://example.com/api/users/1>



```
{
  "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"
    }
  }
}
```

# Response Body

## Verify

- Structure of a response
- Fields
- Values
- Data types

# Minimal Positive Test

For adding a new user



```
{
  "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

<input checked="" type="checkbox"/>	Authorization	Basic dGVzdDE6dGVzdDI=
<input checked="" type="checkbox"/>	Content-Type	application/json

```
{  
    Request  
    "id": 1,  
    "name": "Ivan",  
    "email": "ivan.ivanov@gmail.com"  
}
```

Send

Status: 200 OK

```
{  
    Response  
    "id": 1,  
    "name": "Ivan",  
    "email": "ivan.ivanov@gmail.com"  
}
```

Etag → W/"6b80-Ybsq/K6GwwqrYkAsFxqDXGC7DoM"

Expect-CT → max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"

# Other Positive Tests

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

```
{  
    Request  
    "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!

```
{  
  "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



**Товар закончился**

Дизайн привычных вещей

Дональд Норман

Букинистика

[Отписаться](#)



Андреас Мюллер, Сара Гвидо

**Бestseller**

**2 529 ₽**

Введение в машинное обучение с

помощью Python. Руководство

Андреас Мюллер, Сара Гвидо

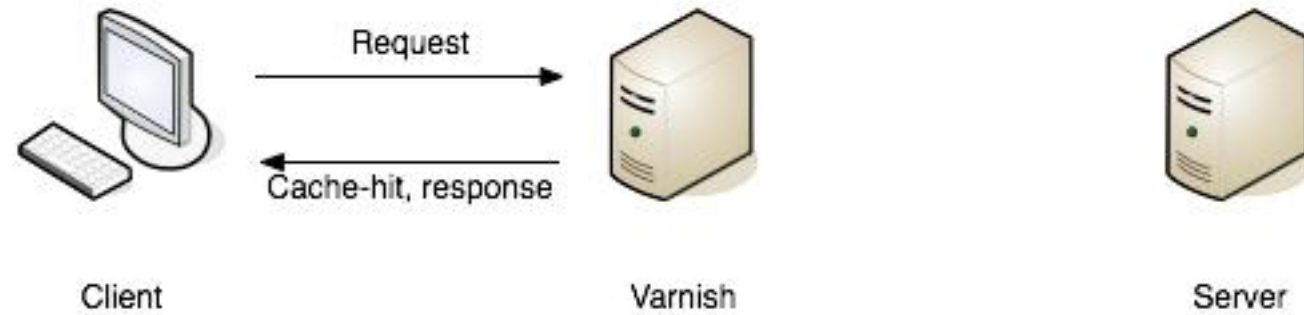
★★★★★ 3

В наличии

[В корзину](#)



# 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

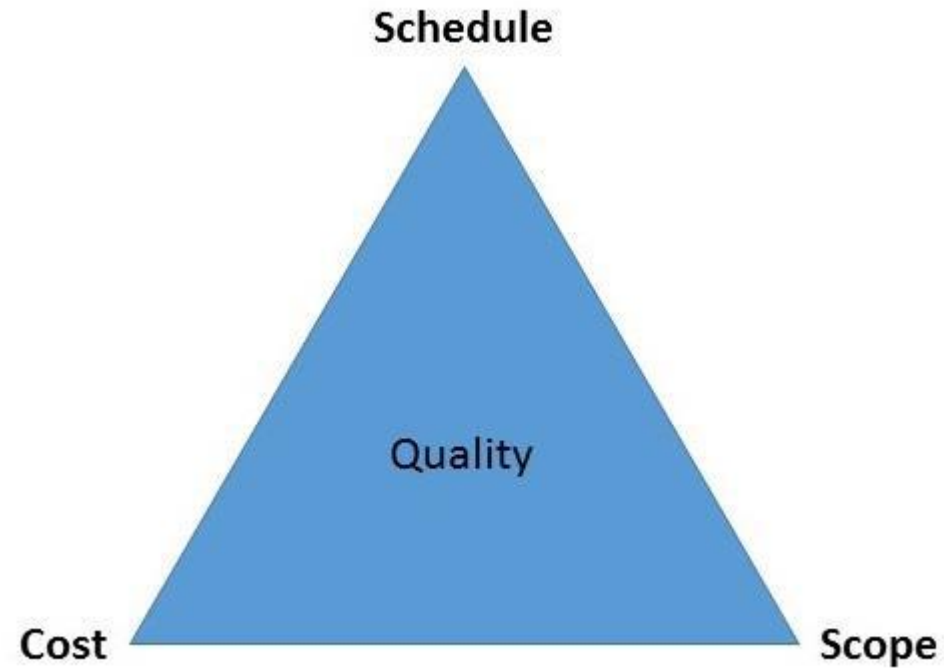
# Integration with Third Party

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

# Coverage



# Project Management Triangle



# Third-Party Components

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



# Risks

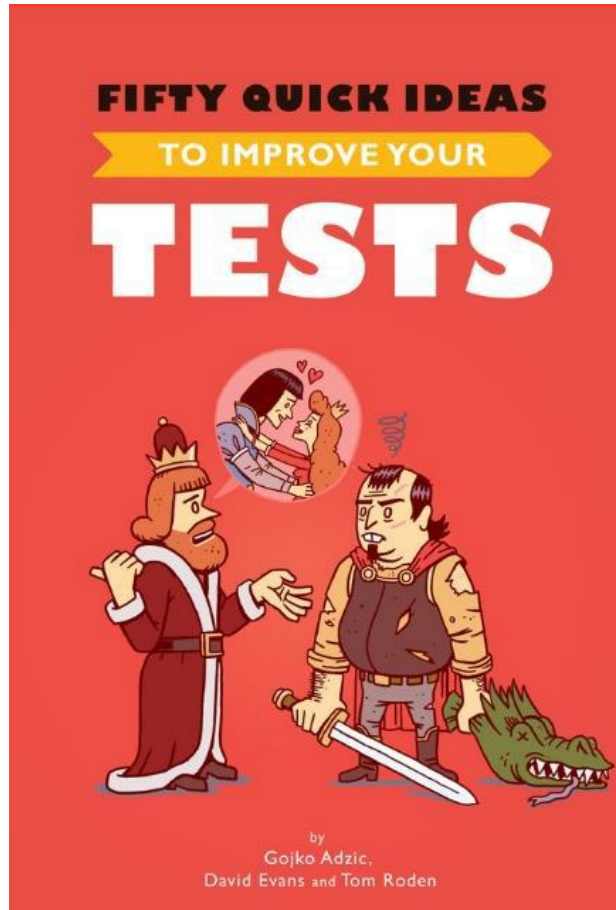


# Risks

- 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

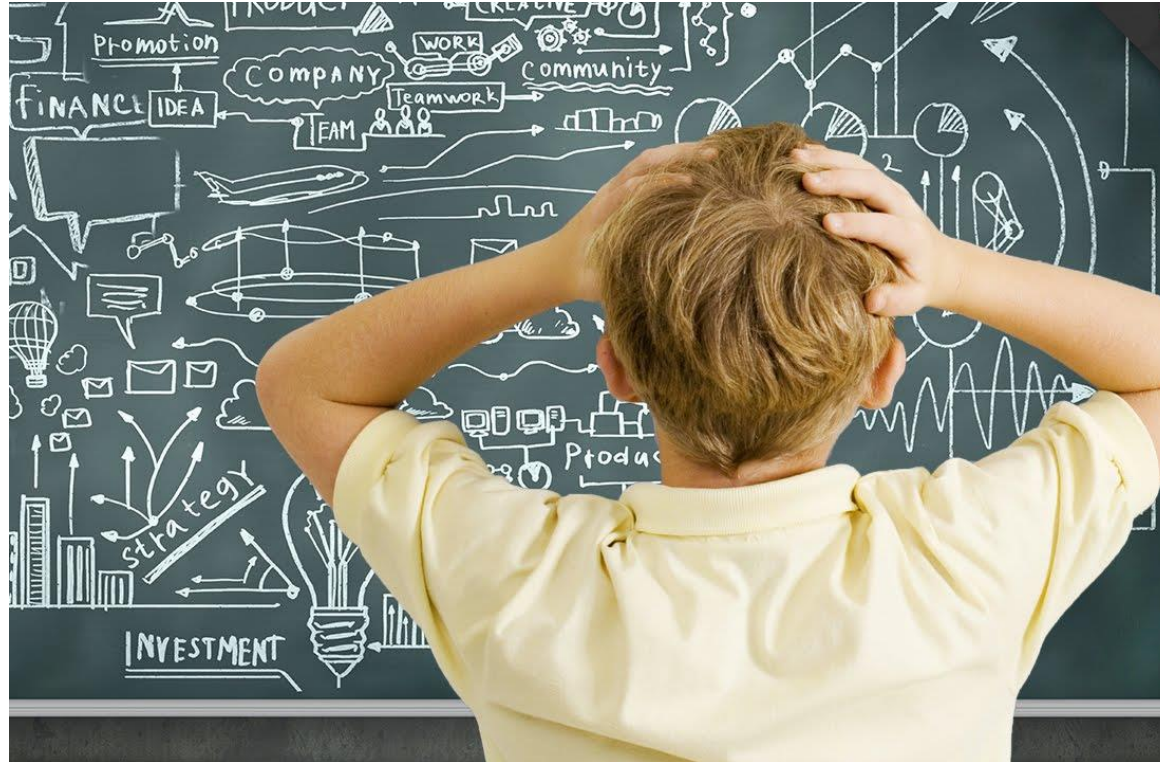
- Find out trusted boundaries. “Trust CMS” example
- Add notes for trusted boundaries for tracking
- Verify that those boundaries are still relevant

# 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

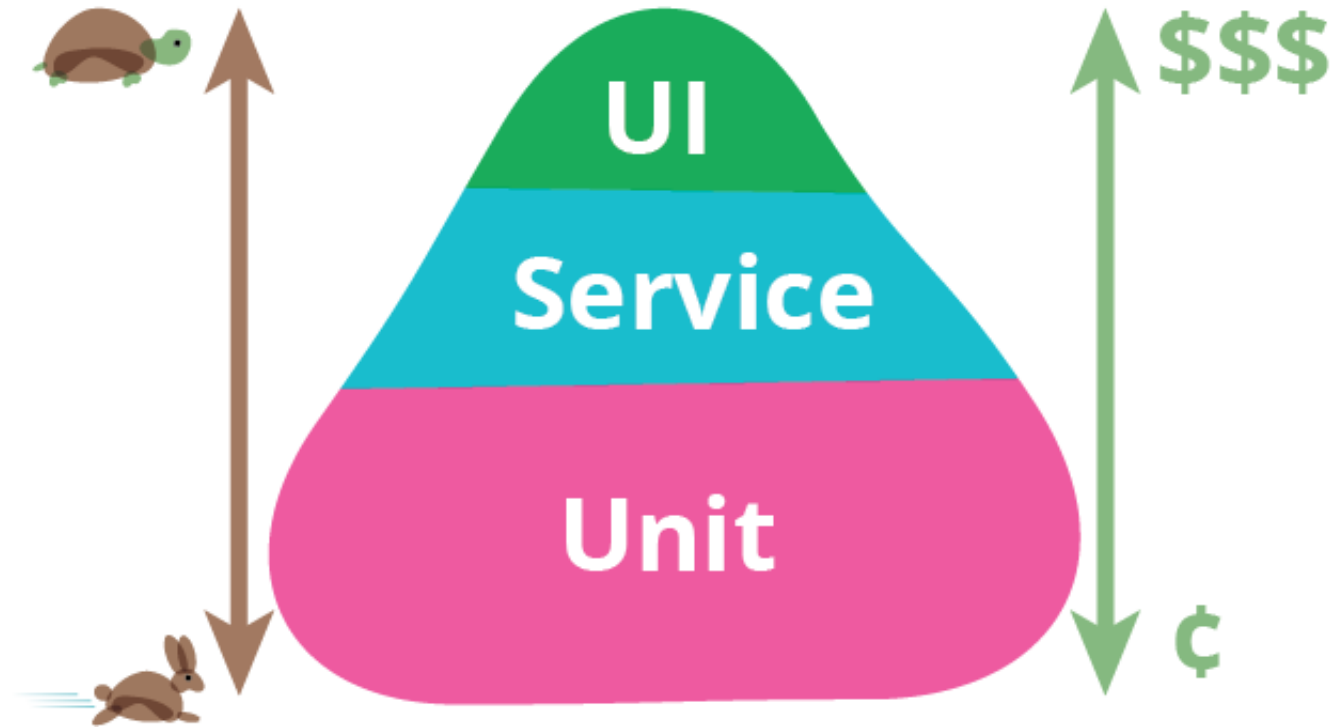


# Test Automation

Including parallel tests



# Test Pyramid



# 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



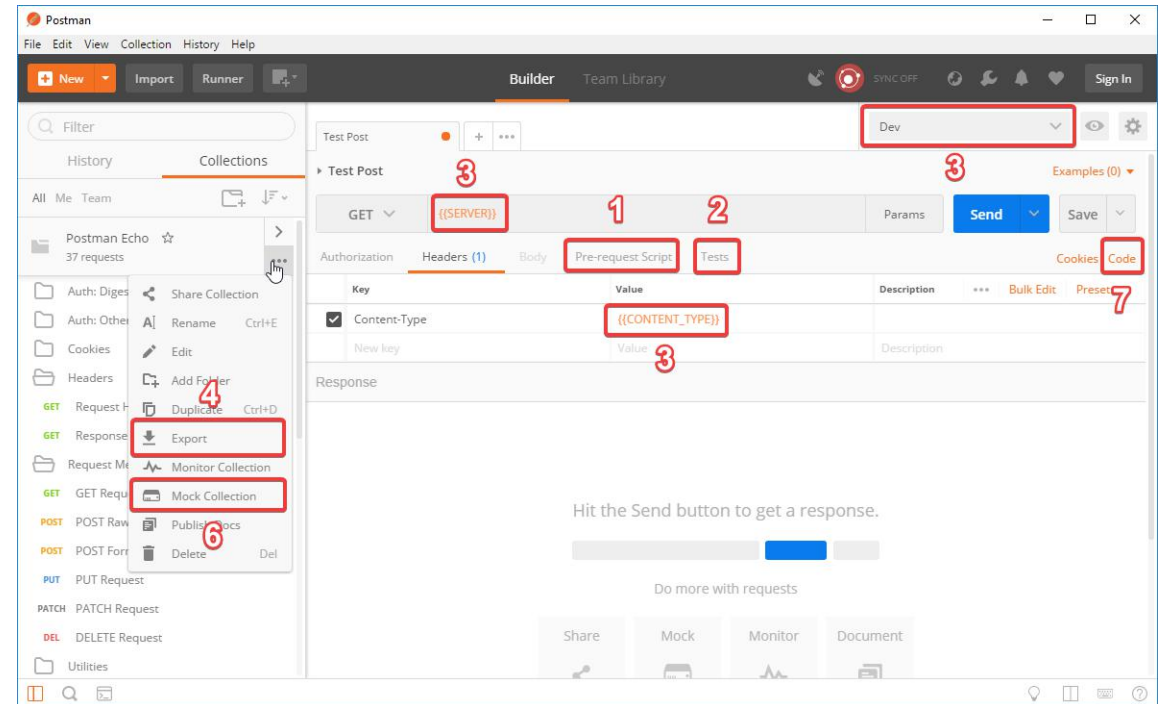
# Agenda

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](#) 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

```
{
  "id": ${id},
  "name": ${name},
  "surname": ${surname},
  "username": ${username},
  "email": ${email},
  "address": {
    "street": ${adr_street},
    "house": ${adr_house},
    "apt": ${adr_apt},
    "city": ${adr_city},
    "zipcode": ${adr_zip_code}
  },
  "phone": ${phone},
  "website": ${website},
  "company": {
    "name": ${comp_name},
    "address": {
      "street": ${comp_adr_street},
      "house": ${comp_adr_house},
      "apt": ${comp_adr_apt},
      "city": ${comp_adr_city},
      "zipcode": ${comp_adr_zipcode}
    }
  }
}
```

# Object Mapping

JSON

```
{"message": "My message"}
```

POJO

```
public class Message {  
    private String message;  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
}
```

Deserialize

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

# Object Mapping

## Serialize

```
Message message = new Message();  
message.setMessage("My messagee");  
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

```
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;  
    }  
}
```

# Object Mapping

```
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);
```

# Object Mapping + Builder

```
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

```
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

```
{  
  "id": 1,  
  "name": "Ivan",  
  "unknownfield": "someValue",  
  "surname": "Ivanov",  
  "username": "Ivanovich",  
  "email": "ivan.ivanov@gmail.com"  
}
```

```
{  
  "id": 1,  
  "name": "Ivan",  
  "surname": null,  
  "username": "Ivanovich",  
  "email": "ivan.ivanov@gmail.com"  
}
```

```
{  
  "id": 1,  
  "name": 42,  
  "surname": "Ivanov",  
  "username": "Ivanovich",  
  "email": "ivan.ivanov@gmail.com"  
}
```

# Gson/Jackson Objects

```
JsonObject userAddress = new JsonObject();  
userAddress.addProperty( property: "street", value: "Kolasa");  
userAddress.addProperty( property: "house", value: 5);  
userAddress.addProperty( property: "unknownProperty", value: "value");  
userAddress.add( property: "house", value: null);  
  
System.out.println(userAddress.get("street").getAsString());  
System.out.println(userAddress.get("house").getAsInt());
```



# Gson/Jackson Objects

```
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).isJsonNull() ? 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 JsonObject getJsonObject() { return jsonObject; }  
  
    public void setJsonObject(JsonObject jsonObject) { this.jsonObject = jsonObject; }  
  
    public boolean hasCustomStringField(String fieldName) {  
        return jsonObject.has(fieldName);  
    }  
  
    public String getCustomStringField(String fieldName) {  
        return jsonObject.get(fieldName).isJsonNull() ? 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("Kolasa");
```

```
if (address.hasStreet()) {  
    System.out.println(address.getStreet());  
}  
if (address.hasCustomStringField( fieldName: "unknownProperty")) {  
    System.out.println(address.getCustomStringField( fieldName: "unknownProperty"));  
}
```

# Gson/Jackson Objects + Builder

```
public class AddressBuilder {  
  
    Address address = new Address();  
  
    public AddressBuilder withRequiredFieldsOnly() {  
        address.setStreet("Kolasas");  
        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"));
```

# Stubs

## WireMock

```
@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.isSuccessful());

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