No API? No problem!

API mocking with WireMock

An open source workshop by …

Originally created by Bas Dijkstra - bas@ontestautomation.com - http://www.testautomation.com - @_basdijkstra

What are we going to do?

_Stubbing, mocking and service virtualization

_WireMock

_Get your hands dirty

Preparation

_Install IntelliJ IDEA (or any other IDE)

_Download or clone project

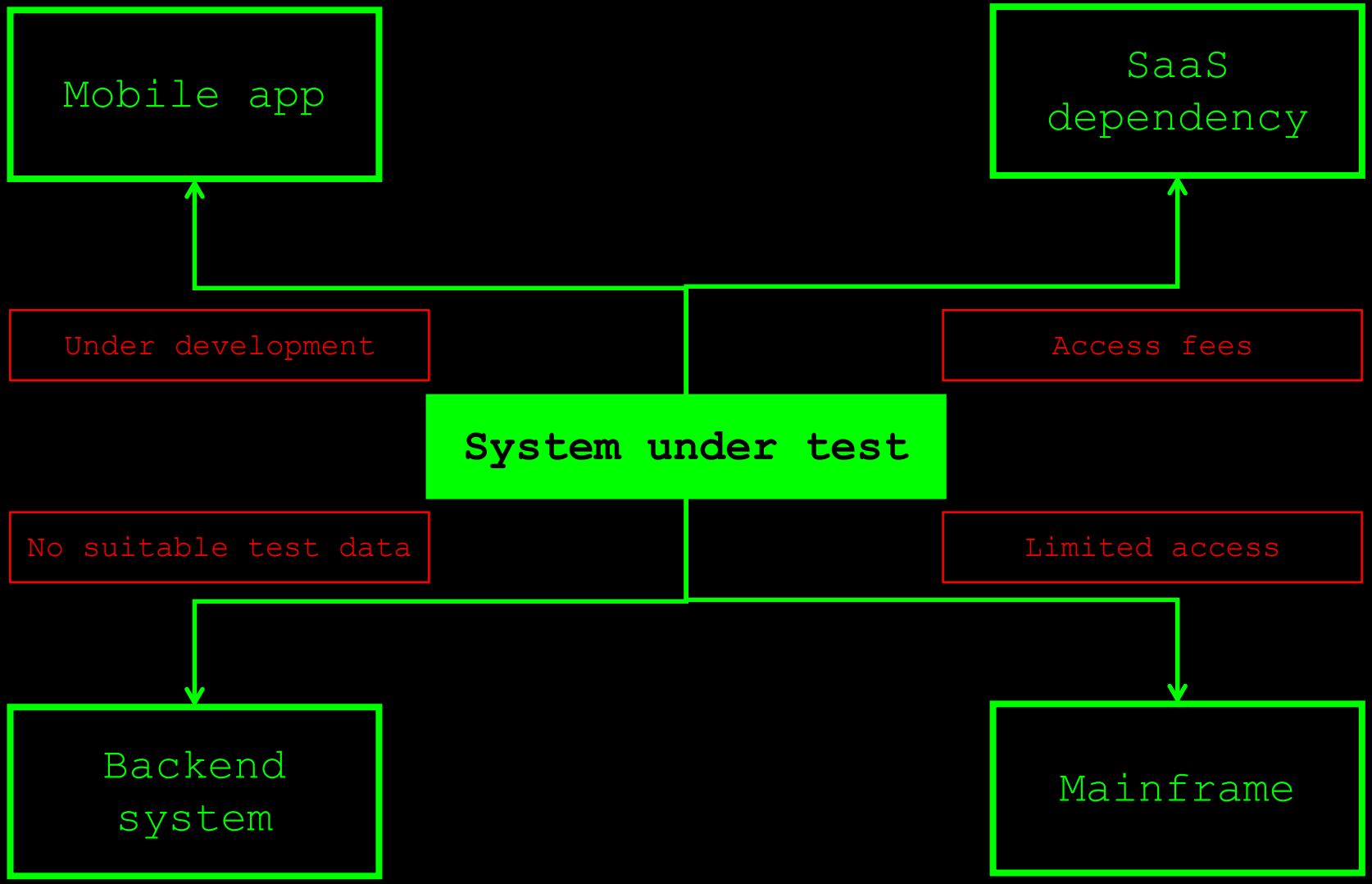
_Import Maven project in IDE

Problems in test environments

- Systems are constructed out of of many different components

- Not all of these components are always available for testing
  - Parallel development
  - No control over testdata
  - Fees required for using third party component
  - …

Problems in test environments



Simulation during test execution

_Simulate dependency **behavior**

_Regain full control over test environment
  _Available on demand
  _Full control over test data (edge cases!)
  _No third party component usage fees
  _…

Stubbing

_Predefined responses

_No flexibility

_Status verification

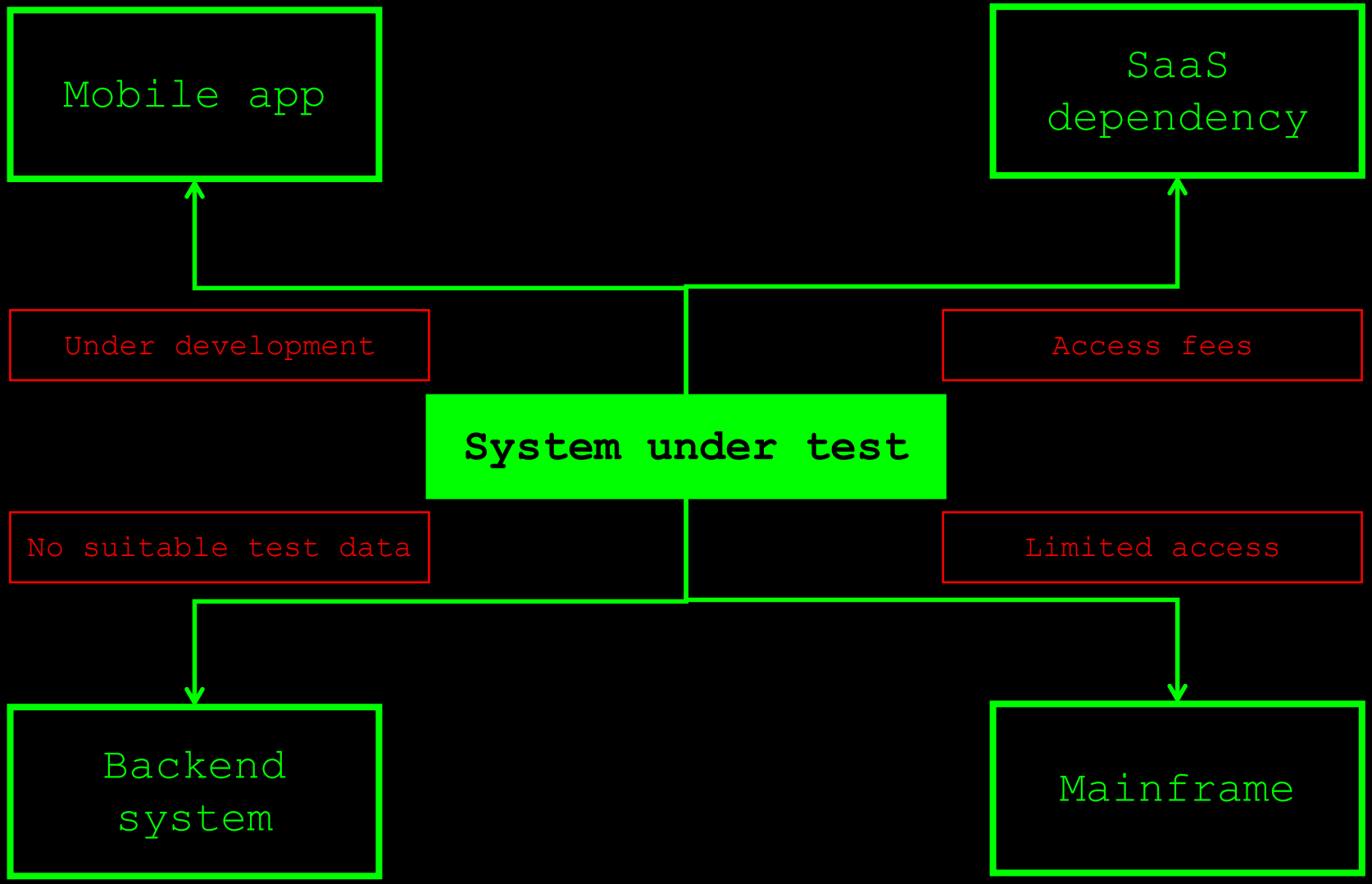
Mocking

- Define mock behavior during test initialization
- (Somewhat) more flexible
- Behavior verification

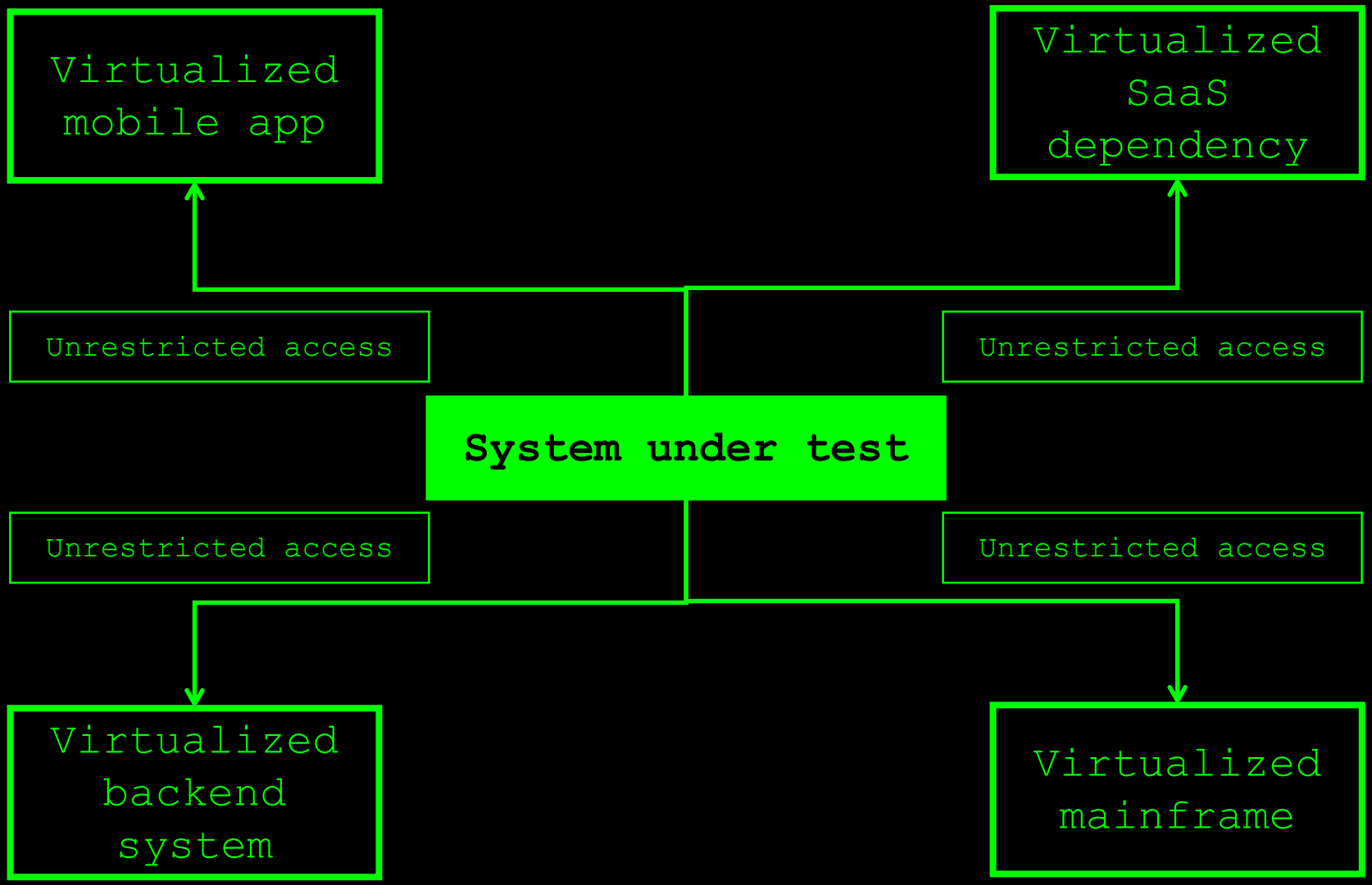
Service virtualization

- Simulate complex dependency behavior
- ‘Enterprise level’ stubbing / mocking
- Support for many different protocols and message formats
- Data driven

Problems in test environments



Simulation in test environments



WireMock

_http://wiremock.org

_Java

_HTTP mock server
  _only supports HTTP(S)

_open source
  _developed and maintained by Tom Akehurst

Install WireMock

_Maven

```
<dependency>
    <groupId>com.github.tomakehurst</groupId>
    <artifactId>wiremock</artifactId>
    <version>2.18.0</version>
</dependency>
```

```
<dependency>
    <groupId>com.github.tomakehurst</groupId>
    <artifactId>wiremock-standalone</artifactId>
    <version>2.18.0</version>
</dependency>
```

_Gradle

```
testCompile "com.github.tomakehurst:wiremock:2.18.0"

testCompile "com.github.tomakehurst:wiremock-standalone:2.18.0"
```

Run WireMock

_In Java (via JUnit @Rule)

```
@Rule
public WireMockRule wireMockRule = new WireMockRule(9876);
```

_In Java (without using JUnit)

```
WireMockServer wireMockServer = new WireMockServer(wireMockConfig().port(9876));
wireMockServer.start();
```

_Standalone

java -jar wiremock-standalone-2.18.0.jar --port 9876

Configure responses

_In (Java) code

_Using JSON mapping files

An example mock

In Java

```
stubFor(
    post(
        urlEqualTo( testUrl: "/pingpong")
    )
    .withRequestBody(
        equalToXml( value: "<input>PING</input>")
    )
    .willReturn(
        aResponse()
        .withStatus(200)
        .withHeader(
            key: "Content-Type",
            ...values: "application/xml"
        )
        .withBody("<output>PONG</output>")));
```

In JSON

```
{
    "request": {
        "method": "POST",
        "url": "/pingpong",
        "bodyPatterns" : [ {
            "equalToXml" : "<input>PING</input>"
        } ]
    },
    "response": {
        "status": 200,
        "body": "<output>PONG</output>",
        "headers": {
            "Content-Type": "application/xml"
        }
    }
}
```

Syntax

```
stubFor(
    post(
        urlEqualTo( testUrl: "/pingpong")
    )

    .withRequestBody(
        equalToXml( value: "<input>PING</input>")
    )

    .willReturn(
        aResponse()
        .withStatus(200)
        .withHeader(
            key: "Content-Type",
            ...values: "application/xml"
        )
        .withBody("<output>PONG</output>")));
```

This stub responds to:

- A HTTP POST to */pingpong*
- With body *<input>PING</input>*

With an answer having:

- HTTP status code *200*
- content type *application/xml*
- body *<output>PONG</output>*

Useful WireMock features

- Verification
  - Verify that certain requests are sent by application under test
- Record and playback
  - Generate mocks based on request-response pairs (traffic)
- Fault simulation
- …
- Full documentation at http://wiremock.org/docs/

Running WireMock standalone

- Start WireMock server
  - Options: port, keystore, ...
- Make mocks permanently available
  - For example for multiple teams
- Reconfigure mocks via JSON

java -jar wiremock-standalone-2.18.0.jar --port 9876

Starting and stopping WireMock during test execution

_Integration in test execution

_Mocks in version control (Git, etc.)

_JUnit integration using @Rule annotation

```
@Rule
public WireMockRule wireMockRule = new WireMockRule(9876);
```

_Can be used without having to use JUnit as well

Demo

Running and using WireMock in standalone mode

Use of WireMock in the exercises

Writing your very first mock

Exercise time!

_WireMockExercises1.java

_Create a number of simple mocks

_Exercises are defined in the comments

_Verify your solution by running the tests

Request matching

- Send a response only when certain properties in the request are matched

- Options for request matching:
  - URL
  - HTTP method
  - Query parameters
  - Headers
  - Request body elements
  - …

Example: URL matching

```
public void setupStubURLMatching() {

    stubFor(get(urlEqualTo("/urlmatching"))
        .willReturn(aResponse()
            .withBody("URL matching")
    ));
}
```

_Other URL options:

- _urlPathEqualTo (using exact values)
- _urlMatching (using regular expressions)
- _urlPathMatching (using regular expressions)

Example: body element matching

```
public void setupStubRequestBodyMatching() {

    stubFor(post(urlEqualTo("/requestbodymatching"))
        .withRequestBody(containing("RequestBody"))
        .willReturn(aResponse()
            .withBody("Request body matching")
    ));
}
```

_Other request body matching options:

- _equalTo (using exact values)
- _matching, notMatching (using regular expressions)

Example: header matching

```
public void setupStubHeaderMatching() {

    stubFor(get(urlEqualTo("/headermatching"))
        .withHeader("Content-Type", containing("application/json"))
        .withHeader("DoesntExist", absent())
        .willReturn(aResponse()
            .withBody("Header matching")
    ));
}
```

_absent(): check that parameter is not in request

Example: basic authentication matching

```
public void setupStubAuthorizationMatching() {

    stubFor(get(urlEqualTo("/authorizationmatching"))
        .withBasicAuth("username", "password")
        .willReturn(aResponse()
            .withBody("Authorization matching")
    ));
}
```

- HTTPS is supported
- Oauth(2) can be simulated using header matching

Exercise time!

_WireMockExercises2

_Use request matching

_Exercises are defined in the comments

_Verify your solution by running the tests

Fault simulation

_Extend test coverage by simulating faults

_Often hard to do in real systems

_Easy to do using stubs or mocks

_Used to test the exception handling of your application under test

Example: HTTP status code

```
public void setupStubReturningErrorCode() {

    stubFor(get(urlEqualTo("/errorcode"))
        .willReturn(aResponse()
            .withStatus(500)
    ));
}
```

_Often used HTTP status codes:

Client error	**Server error**
403 (Forbidden)	500 (Internal server error)
404 (Not found)	503 (Service unavailable)

Example: timeout

```
public void setupStubFixedDelay() {

    stubFor(get(urlEqualTo("/fixeddelay"))
        .willReturn(aResponse()
            .withFixedDelay(2000)
    ));
}
```

- Random delay can also be used
  - Uniform, lognormal, chunked dribble distribution options

- Can be configured on a per-stub basis as well as globally

Example: bad responses

```
public void setupStubBadResponse() {

    stubFor(get(urlEqualTo("/badresponse"))
        .willReturn(aResponse()
            .withFault(Fault.MALFORMED_RESPONSE_CHUNK)
    ));
}
```

- HTTP status code 200, but garbage in response body

- Other options:
  - RANDOM_DATA_THEN_CLOSE (as above, without HTTP 200)
  - EMPTY_RESPONSE (does what it says on the tin)
  - CONNECTION_RESET_BY_PEER (close connection, no response)

Exercise time!

_WireMockExercises3

_Use fault simulation

_Exercises are defined in the comments

_Verify your solution by running the tests

Stateful mocks

- The mocks we created until now have been stateless
  - Order of calling mocks does not influence behavior

- Not always true in the real world

- Request A > request B might differ from request B > request A

Stateful mocks in WireMock

- Supported through the concept of a Scenario
- Essentially a finite state machine (FSM)
  - States and state transitions
- Combination of current state and incoming request determines the response being sent
  - Before now, it was only the incoming request

Stateful mocks: an example

```
public void setupStubStateful() {

    stubFor(get(urlEqualTo("/order")).inScenario("Order processing")
        .whenScenarioStateIs(Scenario.STARTED)
        .willReturn(aResponse()
            .withBody("Your shopping cart is empty")
    ));

    stubFor(post(urlEqualTo("/order")).inScenario("Order processing")
        .whenScenarioStateIs(Scenario.STARTED)
        .withRequestBody(equalTo("Ordering 1 item"))
        .willReturn(aResponse()
            .withBody("Item placed in shopping cart"))
        .willSetStateTo("ORDER_PLACED")
    );

    stubFor(get(urlEqualTo("/order")).inScenario("Order processing")
        .whenScenarioStateIs("ORDER_PLACED")
        .willReturn(aResponse()
            .withBody("There is 1 item in your shopping cart")
    ));
}
```

Exercise time!

_WireMockExercises4

_Use stateful mocks

_Exercises are defined in the comments

_Verify your solution by running the tests

Response templating

- Often, you want to reuse elements from the request in the response
  - Request ID header
  - Unique body elements (client ID, etc.)
  - Cookie values

- WireMock supports this through response templating

Setup response templating

_In code: through the JUnit rule

```
@Rule
public WireMockRule wm = new WireMockRule(wireMockConfig()
        .port(9876)
        .extensions(new ResponseTemplateTransformer( global: false))
);
```

_Global == false: response templating transformer has to be enabled for individual stubs

Enable/apply response templating

_This template reads the HTTP request method (GET/POST/PUT/…) and returns it as the response body

```
public void setupStubResponseTemplatingHttpMethod() {

    stubFor(any(urlEqualTo( testUrl: "/template-http-method"))
            .willReturn(aResponse()
                    .withBody("{{request.requestLine.method}}")
                    .withTransformers("response-template")
            ));
}
```

Request attributes

- Many different request attributes available for use
  - `request.requestLine.method` : HTTP method (example)
  - `request.requestLine.path.[<n>]` : n^{th} path segment
  - `request.requestLine.scheme` : protocol (e.g. HTTPS)
  - …

- All available attributes listed at

http://wiremock.org/docs/response-templating/

Request attributes (cont'd)

_Extracting and reusing body elements

_In case of a JSON request body:

{{jsonPath request.body '$.path.to.element'}}

_In case of an XML request body:

{{xPath request.body '/path/to/element/text()'}}

JSON extraction example

_When sent this JSON request body:

```
{
    "book": {
        "author": "Ken Follett",
        "title": "Pillars of the Earth",
        "published": 2002
    }
}
```

_This stub returns a response with body “Pillars of the Earth”:

```
public void setupStubResponseTemplatingJsonBody() {

    stubFor(post(urlEqualTo( testUrl: "/template-json-body"))
            .willReturn(aResponse().
                    withBody("{{jsonPath request.body '$.book.title'}}").
                    withTransformers("response-template")
            ));
}
```

Exercise time!

_WireMockExercises5

_Use request templating

_Exercises are defined in the comments

_Verify your solution by running the tests

Mock specification via JSON

- So far, we've only specified mock behaviour in Java code
- Mocks live for the duration of the test run
- Want longer living mocks? Use JSON mapping files
- WireMock can be run as a standalone process

Running WireMock standalone

java -jar wiremock-standalone-2.18.0.jar --port 9876

- ‘mappings’ subfolder should contain JSON mapping definitions

- ‘__files’ subfolder contains additional files

JSON mapping files

_All features available in Java also available through JSON mappings

_Example:

```
{
    "request": {
        "method": "POST",
        "url": "/pingpong",
        "bodyPatterns" : [ {
            "equalToXml" : "<input>PING</input>"
        } ]
    },
    "response": {
        "status": 200,
        "body": "<output>PONG</output>",
        "headers": {
            "Content-Type": "application/xml"
        }
    }
}
```

JSON mapping files

_Documentation of all features, along with examples on how to implement them through JSON mapping files can be found at

http://wiremock.org/docs/

Demo

Using JSON mapping files to configure stubs

Record and playback options

_Use WireMock as a proxy

_Record request-response pairs (traffic)

_Genererate mock from recorded traffic

Demo

Using record and playback in WireMock

Pros and cons of record and playback

_Pros:
  _Easy creation of mocks
  _Analyse traffic of which there are no specifications

_Cons:
  _Rerecording necessary when interface changes
  _Mocks are not flexible
  _Mocks are hard to extend

_Similar to record and playback in test automation

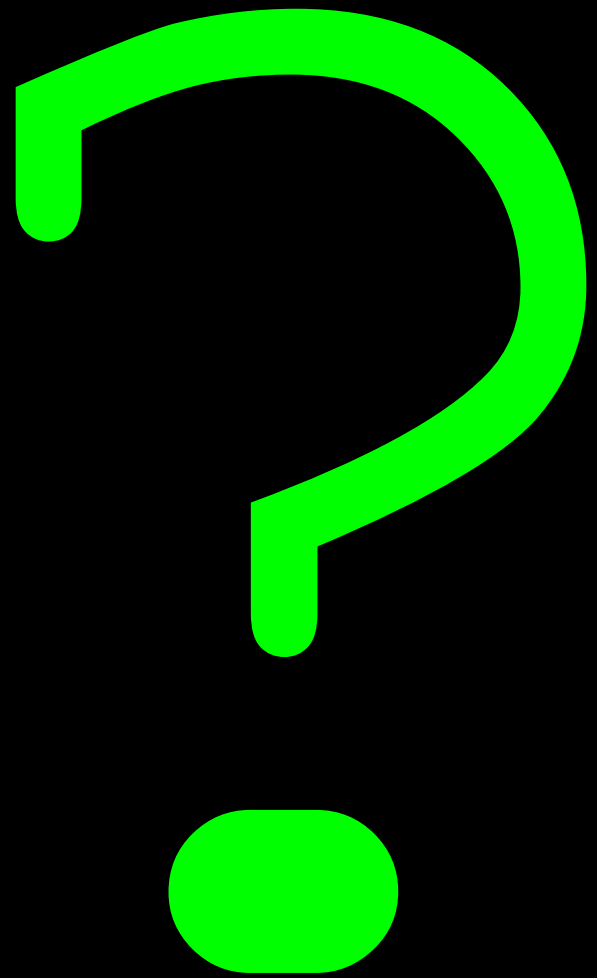
Other useful features

_Verification (was a given message sent ?)

_Response transformations (via extensions)

_Integration into a CI / CD pipeline

_Documentation: http://wiremock.org/docs/



Contact

_Email:

_Blog:

_LinkedIn:

_Twitter: