Testing APIs

Best practices
Version 0.1

Reviews History

Date	Version	Description	Author
08/06/2016	0.1	Initial draft of the document	Juan Krzemien

Table of Contents

About this document

Given the increase in Quality Assurance projects Globant is experiencing, there is an unquestionable need to standardize how automated solutions should be designed and developed in order to meet (and even surpass) clients expectations.

This document focuses on automated API testing; but the root concepts detailed here can be extrapolated to different software development areas and, possibly, languages. It covers frequently used technologies and defines certain rules, considered as good practices:

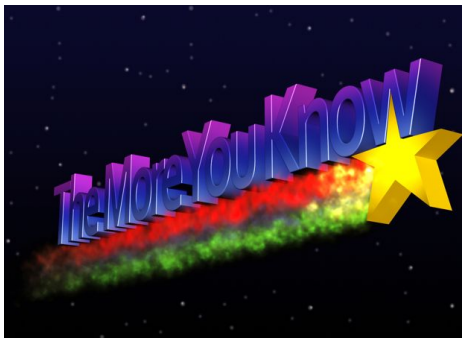
1. Adhere to DRY (Don't Repeat Yourself) principle.
2. Increase test script development speed as the project advances.
3. Make test code more robust, readable and reusable.
4. Decrease maintenance costs of test code.
5. Increase the quality of test design and its code.
6. Increase the Return on Investment (ROI) of tests.

Prerequisites for this document

Knowledge and experience in software testing is a mandatory requirement, but also you will benefit from having some degree of expertise in:

- Object Oriented Programming paradigm
- Java (or similar) Object Oriented Programming language

Glossary



- API - Application Programming Interface. This document focuses particularly in Web APIs.

- SUT - System Under Test

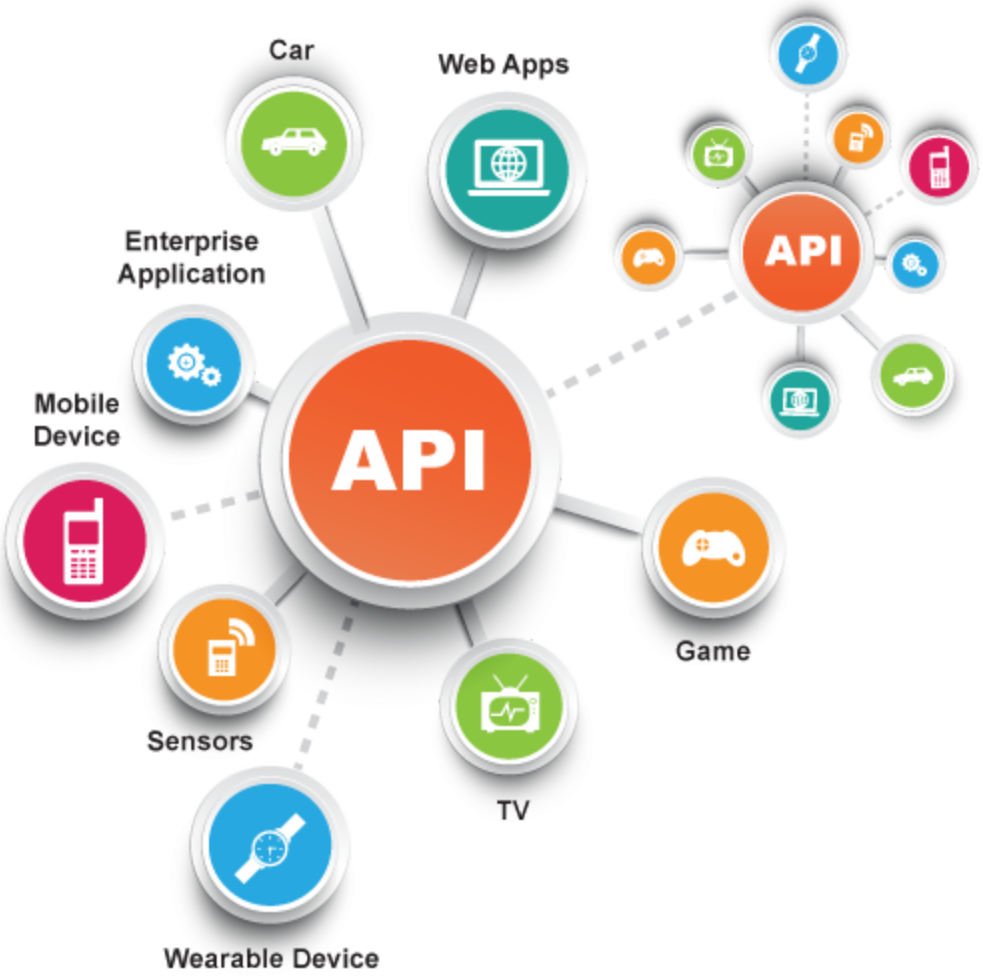
- DSL - Domain Specific Language

- URL - Uniform Resource Locator
  Is a reference to a resource that specifies the location of the resource on a computer network and a mechanism for retrieving it. A URL is a specific type of uniform resource identifier (URI). A URL implies the means to access an indicated resource, which is not true of every URI. URLs occur most commonly to reference web pages (http), but are also used for file transfer (ftp), email, database access, and many other applications.

- URI - Uniform Resource Identifier
  Is a string of characters used to identify a name of a resource. Such identification enables interaction with representations of the resource over a network, typically the World Wide Web.

- POJO - Plain Old Java Object

- HATEOAS - Hypermedia as the Engine of Application State
  Special implementation of a subset of REST. Client actions are discovered within resource representations returned from the server. The client transitions through application states by selecting from the links within a representation. In this way, RESTful interaction is driven by hypermedia, rather than out-of-band information.



About Service Testing

Also referred to as API testing, its purpose is to determine if services meet expectations on functionality, reliability, performance, and security. APIs do not have a GUI, so testing has to be performed at the message layer.

Nowadays, API testing is considered a critical part of automated testing because APIs serve as the primary interface to application logic and because GUI tests are difficult to maintain due to short release cycles and frequent changes commonly introduced by Agile software development methodologies.

Most of the time, we will be dealing with HTTP based APIs, but know that there are lots of different ways to communicate data between endpoints and they are also considered API testing.

API testing coverage

Must cover:

- ✓ **Functional tests:** Valid requests to test different combination of arguments and parameters, prepare test scenario that simulate clients requests.
- ✓ **Negative tests:** Invalid (bad) requests should be tested to check that there are handled properly and won't crash your application.

Could cover:

- 👻 **Security tests:** Check that requests from different clients don't influence each other.
- 👻 **Performance tests:** Measure response times from endpoints.

HTTP Protocol

For now, let's focus on HTTP. As you may already know, an HTTP request message is basically composed of:

- 👉 An URL/URI: the "resource" we are interested in
- 👉 A verb/action over the previous URL/URI/resource, representing the *intention* of the query
    - 🔁 **POST** - Creates the specified resource
    - 🔁 **GET** - Retrieves the specified resource. It should have no other effect
    - 🔁 **PUT** - If exist, updates the specified resource, if not, it creates it
    - 🔁 **DELETE** - Deletes the specified resource
    - 🔁 **HEAD**, **TRACE**, **OPTIONS**, **CONNECT** and **PATCH**
- 👉 Headers: meta-information that define the operating parameters of an HTTP transaction
- 👉 A payload: the actual data to send/receive. Its format can vary. *Usually* it is JSON, XML or YAML, but it *can* be *anything*.

In HTTP responses, the first line of the HTTP response is called the *status line* and includes a numeric 3 digits *status code* (such as "404") and a textual *reason phrase* (such as "Not Found")

- 👆 1xx Informational
- 👍 2xx Success
- 👇 3xx Redirection
- ✋ 4xx Client Error
- 👊 5xx Server Error

Aside from the HTTP code and reason status line, the response also has headers and a payload.

Service Testing Best Practices

Extract string literals and magic numbers to a constants class

When writing automated API tests, it is pretty common to have to deal with numbers, strings and some other values to check for the expected response values.

Here are some tips for when those situations arise:

🚫 **Do not** hardcode values in tests nor framework.

✓ Have a *public* class that groups nested static classes with public static members representing the constants in order to have a single place to maintain the test values.

```
public class Constants {
   public static class Errors {
       public static String NoData = "The request returned no data";
       public static String NoValidApi = "/ruso was not found in this server";
   }

   public static class Http {
       public static int Ok = 200;
       public static int Created = 201;
       public static int NotFound = 404;
       public static int InternalServerError = 500;
   }
}
```

An use it like this:

```
@Test
public void aTest() {
   // Invoke API and store response
   // …
   assertEquals("Response code", Constants.Http.InternalServerError, response.code);

   assertEquals("Response body", Constants.Errors.NoValidApi, response.data);
}
```

⚠ **Most HTTP libraries already include an enumeration with HTTP status codes. Use it whenever possible, so you do not need to define them as constants.**

✓ Externalize data into configuration files. Read them *once* at framework initialization and store data in immutable structures. Make public getter methods for useful values with clear and concise names.

Choose the right library for the job

In *most* projects, API testing relies **a lot** on HTTP as data transport mechanism. In order for you to work with it, you will need an HTTP client. There are **lots** of HTTP clients out there...but some are better than others.

What makes an HTTP client good? Its features! Some may include:

- 🐞 Support for multiple, cached, pooled and/or concurrent connections
- 🐞 Support for many/customized authentication mechanisms
- 🐞 Easy interception/monitoring/logging
- 🐞 Easy initialization, abstraction, usage or code readability
- 🐞 Support for embedded or pluggable marshalling/serialization to/from JSON, XML, etc
- 🐞 Speed

- 🚫 **Do not** use low level classes (such as *URL* or *URLConnection*) directly in your framework. The level of expertise required to *correctly* use them is high if you do not know what you are doing. You would have to deal with buffers, bytes, strings, parsing, (un)marshalling, sockets, etc.
- 🚫 **Avoid** directly using Apache's HttpClient too, for reasons similar to the ones described in previous item.

My recommendation would be to use one the following libraries:

- Retrofit 2

```
<dependency>
   <groupId>com.squareup.retrofit2</groupId>
   <artifactId>retrofit</artifactId>
   <version>2.0.2</version>
</dependency>
<dependency>
   <groupId>com.squareup.retrofit2</groupId>
   <artifactId>converter-jackson</artifactId>
   <version>2.1.0</version>
</dependency>
```

Retrofit 2 Maven dependencies

- RestEasy Proxy Framework - a subset of RestEasy (client side only)

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-client</artifactId>
   <version>3.0.17.Final</version>
</dependency>
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-jaxrs</artifactId>
   <version>3.0.17.Final</version>
</dependency>
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-jackson-provider</artifactId>
   <version>3.0.17.Final</version>
</dependency>
```

RestEasy Maven dependencies

Why? Well, both libraries allow you to define the API you are trying to test as Java interfaces, which is clean, simple and urges you to define entities/POJOs to store and send the data, enforcing a minimum number of DSL models, which is good. It also allows you to keep working in an OOP fashion.

The way you work with them is quite similar.

First you define an interface which will represent a *particular* API:

```
public interface Groups {
   @GET("group/{id}/users")
   Call<List<User>> groupList(@Path("id") int groupId, @QueryMap Map<String, String> options);
}

public interface Users {
   @Headers("Cache-Control: max-age=640000")
   @POST("users/new")
   Call<User> createUser(@Body User user);
}

public interface User {
   @GET("user")
   Call<User> getUser(@Header("Authorization") String authorization);
}
```

Retrofit 2 API definition example

```
public interface Groups {
   @GET
   @Path("group/{id}/users")
   List<User> groupList(@PathParam("id") int groupId, @QueryMap Map<String, String> options);
}
public interface Users {
   @POST
   @Path("users/new")
   User createUser(User user);
}
public interface User {
   @GET
   @Path("user")
   User getUser(@HeaderParam("Authorization") String authorization);
}
```

RestEasy API definition example

🚫 **Do not** define all APIs in a *single interface*!. Each endpoint could have dozens of parameters and options so your interface would become easily unreadable due to the amount of code it would contain. It is much better to create one interface file for each (ex: Group, Users and User) under an `api` package that groups them.

✓ Define as many interfaces as APIs you have, and declare all related endpoints inside them.

Then, create a dynamic instance of that interface (automatically embedding an HTTP client in it) that can be used to perform the request as defined by the contract interface:

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com/")
        .build();

User userApi = retrofit.create(User.class);
userApi.createUser(aUserObject);
```

Retrofit 2 API client instantiation example

```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target("https://api.github.com/");

User userApi = target.proxy(User.class);
userApi.createUser(aUserObject);
```

RestEasy API client instantiation example

🚫 **Do not** hardcode the base URL in your tests nor framework. You will *likely* want to run the same tests against multiple environments.

I favor Retrofit 2 usage because:

- ✓ It is cleaner, uses less lines of code and allows to describe API endpoints almost completely via annotations decorated interfaces. Please note that there are several annotations (like `@Headers` and `@Body`) that are not present in RestEasy.
- ✓ OkHttp (its internal Http client engine) is lightweight and the fastest client currently available.
- ✓ It is *thread safe*

> ⚠ **Warning**
>
> RestEasy proxy client uses underneath the Apache Commons Http Client, which **by default** relies on *SimpleHttpConnectionManager* which is **not thread safe**.
>
> Switch to *MultiThreadedHttpConnectionManager* to achieve thread safety.

On the other hand, RestEasy does not force you to return a *Call*<> wrapper object like Retrofit does. This renders calling of the API and handling of its return type a lot *cleaner*.
You can *mask* (as in, hide away) Retrofit cumbersome *try/catch* block for *.execute()* calls inside a protected method in your test base classes.

The reason behind Retrofit decision for this “inconvenience” is that by always wrapping the result of the API method call, it allows synchronous **and** asynchronous handling of this Call instance while *still* using an interface proxy to model the API. RestEasy *cannot handle asynchronous invocations* when using the interface declaration together with its Proxy framework.

You can *still* perform async requests using RestEasy, but you loose the nice API modeling via interfaces:

```
ClientBuilder.newClient().target("http://www.anUrl.com/api/user/1")
        .request()
        .async()
        .get(new InvocationCallback<User>() {
            @Override
            public void completed(User user) {
                // Async request completed
            }

            @Override
            public void failed(Throwable throwable) {
                // Async request failed
            }
        });
```

RestEasy async request sample

Use POJOs to communicate with APIs

- 🚫 **Do not** declare nor use objects as JSON/XML/YAML strings.

- ✓ Use DSL POJOs, set them values, and let Retrofit/RestEasy (or whatever serious library you want to use) marshall them back and forth from POJO ↔ JSON/XML/YAML automatically while sending/receiving HTTP transfers.

- ✓ Take advantage of free online tools to generate your initial entities/POJOs from JSON or XML quickly. *Always treat auto-generated code with a grain of salt*. Not only humans screw things up.

    - 🔗 http://www.jsonschema2pojo.org/
    - 🔗 http://pojo.sodhanalibrary.com/

✓ Whenever possible, try to work with immutable entities.

Entities (de)serialization

🚫 **Do not** create custom (de)serializers for *each* entity.

🚫 **Avoid** implementing `Serializable` interface. Implementing `Serializable` breaks encapsulation and information hiding; all private and package-private members become part of the class's exported API. This introduces potential security vulnerabilities and may make the class more fragile. The decision to implement `Serializable` should be made with great care and the serialized form should be carefully designed.

✓ Whenever possible, try to work with Jackson or Gson libraries to (de)serialize your data.

✓ Consider having a generic abstract “base entity” class for all your entities. You can put there common code for all entities: common fields (id?), override methods (toString?), etc

Introduce DSL object factories

- ✓ Once you have your POJOs, it is useful to create also a simple static factory method in them to be able to create generic (random?) instances of them and not pollute tests with long object instantiation lines.

- ✓ Factory methods are static methods that return an instance of the native class:

    - ➢ Have names, unlike constructors, which can clarify code.

    - ➢ Do not need to create a new object upon each invocation - objects can be cached and reused, if necessary.

    - ➢ Can return a subtype of their return type - in particular, can return an object whose implementation class is unknown to the caller. This is a very valuable and widely used feature in many frameworks which use interfaces as the return type of static factory methods.

  Examples in the JDK:

    - LogManager.getLogManager
    - Pattern.compile
    - Collections.unmodifiableCollection, and so on
    - Calendar.getInstance

Use assertions correctly

✓ Try to use Hamcrest, which is a framework for writing matcher objects allowing 'match' rules to be defined declaratively. It works with JUnit and TestNG. You can include it in your Maven project by simply adding its dependency in your *pom.xml* file:

```
<dependency>
   <groupId>org.hamcrest</groupId>
   <artifactId>hamcrest-all</artifactId>
   <version>1.3</version>
</dependency>
```

🚫 **Do not** evaluate/assert everything with the *same* assertion method. There are more methods than just *assertTrue*, *assertNull* and *assertNotNull*:

Instead of doing this:

```
assertTrue(service.id > 0, "Service id should be assigned");
```

Do something like this:

```
assertThat("Service ID", service.id, greaterThan(0));
```

🚫 **Do not** repeat the content of the assertions over and over again. Cache the response once and then use it many times.

Instead of doing this:

```
assertNotNull("Service id", response.body().data.id);
assertNotNull("Service name", response.body().data.name);
assertNotNull("Service code", response.body().data.code);
assertNotNull("Service xxxx", response.body().data.xxxx);
assertNotNull("Service yyyy", response.body().data.yyyy);
```

Do something like this:

```
Matcher<Collection<?>> isNotNullNorEmpty = allOf(is(notNullValue()), is(not(empty())));
…
Service service = response.body().data;
assertThat("Service", service, is(notNullValue()));
assertThat("Service ID", service.id, is(notNullValue()));
assertThat("Service ID", service.id, greaterThan(0));
assertThat("Service Name", service.name, isNotNullNorEmpty);
assertThat("Service Code", service.code, isNotNullNorEmpty);
assertThat("Service xxxx", service.xxxx, isNotNullNorEmpty);
assertThat("Service yyyy", service.yyyy, isNotNullNorEmpty);
```

🚫 **Avoid** repeating expensive operations in tests if there is no *real* need to do that. For example, querying databases on every assertion. Query it *once*, assert on the *result*.

🚫 **Do not** perform assertions in methods that perform the API call or inside the framework. When dealing with APIs you want to know if the response code is the adequate one for different queries. Not everything will return a 200/201 code. Leave validations and assertions to tests.

Logging

🚫 **Do not** log actions manually during test scripts, this render tests longer and unreadable.

🚫 **Do not** use *System.out*. You cannot easily change log levels, turn it off, customize it, etc. Also, it is a blocking IO-operation and, therefore, it is time consuming. The problem with using it is that your program will wait until the `println` has finished. This may not be a problem with small programs but as soon as you get load or many iterations, you'll feel the pain.

🚫 Mind the log type. If it is not really an error **do not** log it as such.

✓ Prefer the Simple Logging Facade for Java (SLF4J) which serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing you to plug in the desired logging framework at *deployment time*.

✓ Use a private static Logger class defined in base POM class and in the testing framework itself to log to a file, console, etc.

✓ Another way to extend your logging is to add a listener to the test harness framework that you may be already using (TestNG/jUnit). You'll gain access to event from test suites & methods. At those times, it is a good idea to attach a listener to get screenshots and/or request / responses dumps upon test failures.

✓ When you define assertions to check for expected conditions, be very descriptive on the error message you will log. This will save you a lot of time during the analysis of the failures.

✓ If you are using Java 8, an alternative may be to create an interface with a default implementation for the Logger. It "feels" like a single method abstract class, but it is different: it does not force you to extend from a class (given that Java does not support multiple inheritance) and you can easily add it to any class that requires logging without breaking existing code. Here is an example:

```
package ruso.testing.common.logging;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public interface Logging {
   default Logger getLogger() {
       return
LoggerFactory.getLogger(getClass());
   }
}
```

Sample logger as interface with default implementation

Test Scripts

- 🚫 **Do not** perform many validations per tests. It is preferable to have 100 tests with 1 validation each than 1 test with 100 validations in it. Keep it simple. If there is an error during the first validation then you won't know what happens with the other 99 remaining validations.

- 🚫 **Do not** hard code dependencies on external accounts or data. Development and testing environments can change significantly in the time between the writing of your test scripts and when they run, especially if you have a standard set of tests that you run as part of your overall testing cycle. Instead, use API requests to dynamically provide the external inputs you need for your tests.

- 🚫 **Avoid** dependencies between tests. Dependencies between tests prevent tests from being able to run in parallel. Running tests in parallel is by far the best way to speed up the execution of your entire test suite. It's much easier to add a virtual machine than to try to figure out how to squeeze out another second of performance from a single test. This includes avoiding

- ✓ For client authentication, set your credentials as environment variables that can be referenced from within your framework.

- ✓ Use Setup and Teardown. If there are any "prerequisite" tasks that need to be taken care of before your test runs, you should include a setup section in your script that executes them before the actual testing begins.

- ✓ Have the ability to tag the tests and/or suites to be able to run a sub set of the tests you have. Most current test runners supports this feature. Nobody likes to spend an hour waiting for a whole suite of unrelated tests to finish running just because they changed an error message.

- ✓ There is no rule saying that there should be a 1:1 relation between the manual test case steps and its automated version, if it keeps the tests cleaner / more readable, split a single manual test case into many automated tests.

- ✓ Tests should be synchronous and deterministic. This means, that:

    - ✓ Test should have a well defined beginning and end

- ✓ Each and every step performed in the test script should translate to a reaction on the SUT

- ✓ Test should be able to validate such reaction

If this cannot be done, you will have to find a way to make it happen. This may be achievable by adjusting SUT configuration on demand and/or via mock objects. You can't wait for the SUT to actually deliver that email to your test account till midnight because there were network congestions or the scheduled job wasn't triggered till later on.

✓ Organize/structure your tests in a coherent manner. For example, group same API related tests into the same class, so each class represents a suite for a particular API.

⚠ You can use retry rules in your tests. If waiting patiently doesn't resolve your test flakiness, you can use JUnit `TestRule` class or TestNG `IRetryAnalyzer` interface. These will rerun tests that have failed without interrupting your test flow. However, you should **use these only as a last resort**, and very carefully, as rerunning failed tests can mask both flaky tests and flaky product features. Examples can be found here and here, along with their respective tests: here and here.

⚠ If you are developing a new test script and you start mumbling about *"how difficult is to do this!"* or *"Every time I create a new test script I have to do this, or declare that, or copy and paste this block of code"*. Stop right there, because you are probably right and there is something more to be done for the sake of simplifying the script development. Ask your TL or a co-worker for advice and/or insights.

Documentation

- ✓ Include a brief documentation for every test script. Mention its purpose and reference the corresponding manual test case.

Test Reports

- ✓ API tests should generate report with information on which requests passed / failed.
- ✓ In case of failure, the raw request and response should be logged for further investigation.

Configuration

Externalize configuration

🚫 **Do not** hardcode values into your framework.

🚫 **Do not** have a public class with public static fields manually mapped to properties. This implies that the more the properties you have, the more this class needs to be manually changed.

✓ Use external files to define properties that can change the behaviour of your code. The format can be, in order of preference:

- YAML
- JSON
- Simple Java properties file
- XML

✓ Model your configuration as a class, and deserialize the file into it. Use *Jackson* library preferably, which is clean, well known and supports both JSON and YAML marshalling, or any other library for the same purpose that you like.

🔗 **Link:** An example of externalized configuration can be found here.

Auto detect configuration when possible

🚫 For settings that can be autodetected within Java code, **do not** request them as properties. If you need your code to behave differently depending if you are on Windows or Linux, have a helper class to figure it out. Do not request users to put *another* property in your configuration file. No one likes having to deal with a config file with 1000 parameters.

```
public class Environment {

   private static final String OS = getProperty("os.name").toLowerCase();
   private static final String ARCH = getProperty("os.arch", "");

   private Environment() {}

   public static boolean isWindows() {
       return OS.contains("win");
   }

   public static boolean isMac() {
       return OS.contains("mac");
   }

   public static boolean isUnix() {
       return OS.contains("nix") || OS.contains("nux") || OS.contains("aix");
   }

   public static boolean is64Bits() {
       return ARCH.contains("64");
   }
}
```

Example for OS and architecture verification

Java Proxy Settings

Many times, a Java app needs to connect to the Internet. If you are having connectivity troubles because you are behind a corporate proxy, set the following JVM flags accordingly when starting your JVM on the command line. This is usually done in a shell script (in Unix) or bat file (in Windows), but you can also define them in your IDE settings for each test run configuration.

JVM variable	Example value	Type
`http.proxyHost`	proxy.corp.globant.com	String
`http.proxyPort`	3128	Integer
`http.nonProxyHosts`	localhost|127.0.0.1|10.*.*.*|*.foo.com	String

Or, if you have your system settings already configured, you can try:

JVM variable	Example value	Type
`java.net.useSystemProxies`	true	Boolean

JVM variables are defined prepending a **`-D`** when calling the java command from your system's terminal/console.

For example: `java -Djava.net.useSystemProxies=true some_file.jar`

You can also set them programmatically, like:

```
System.setProperty("java.net.useSystemProxies", "true");
```

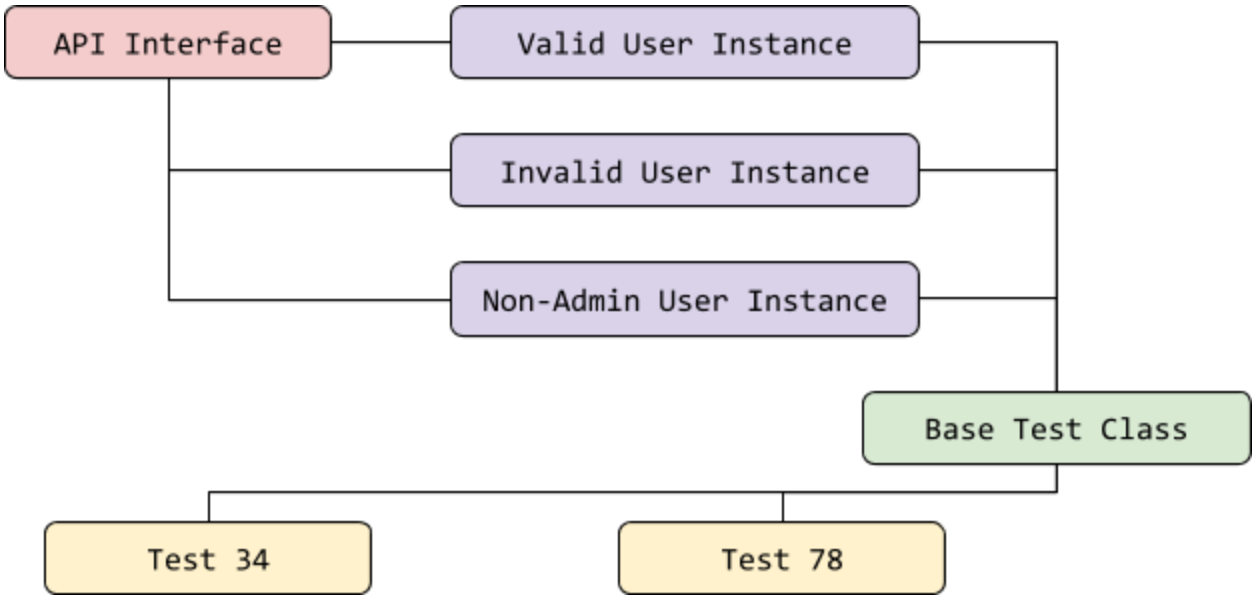
Authorization & Authentication

🚫 **Do not** pass around tokens and/or hashes to your API definition via parameters. If you do that, you'll end up polluting the readability of the API interface and the tests.

If the API you are testing *only* accepts a token/hash as *a parameter in the URL* or *some other ugly way* that forces you to break this rule:

1. I don't believe you
2. Talk with the dev team about the security issues that may arise from sending credentials and/or sensitive information in the URL (not encrypted). Do not let them convince you easily.
3. *As a last resort*, create and expose to tests an adapter/interceptor for the API instances (usually, the HTTP client) so APIs do not take the token/hash as an argument but it is provided to the actual API call either way.

✓ Consider having different API instances for different credentials/privileges levels that the API may support:



✓ Define and use *interceptors* for the HTTP client that you chose. Embed the corresponding *Authorization* header for every request of *a particular API instance*.

Database testing - Beyond the API endpoint

So, you just sent a POST request to your shinny API but, since you've been reading this document and you have some pride, you *know* that you can't just perform a couple of assertions on the response and assume everything went fine behind the API endpoint...right? *RIGHT?*

Now you want to *make sure* that the newly created resource was *actually* persisted into the database (*yes...you want to*).

There is an underlying testing principle here that I did articulate yet, so here it is: if your system under test communicates with the outside world (e.g. a database or a network connection), and you **only** test the system using its *own* operations (API), all you have verified is that the system is consistent with itself.

Your application uses an API to interact with the database. It is possible to write that API in such a way that it presents correct results to the application and yet still uses the database in the wrong way.

For example, imagine a database with an EMPLOYEE table and a MANAGER table. The tables are alike (ex: each contains a first name, last name, company email address, and salary) but the EMPLOYEE table is only for non-managers and the MANAGER table is only for managers.

Now imagine a database API with two classes, one per table. The programmer writes the employee code first. The manager code is almost the same, so the programmer makes a copy of the employee code and then edits it. He changes the name of the class, and he changes the method names. But he's distracted and forgets to change the name of the table from EMPLOYEE to MANAGER. Code compiles, unit tests pass and the programmer declares victory.

Of course, there's a bug in the code: manager records are written to the EMPLOYEE table.

Also, you need to consider that not **all** fields in the database are exposed to the front end/API. Calculated fields, timestamps and the like all *need to be checked*.

Another example is that you **need** to check for "logical" deletes. Usually, data is not actually deleted from DB, but *marked* as deleted. GUI/APIs should no longer retrieve/show this data but the date *should be there in the DB*.

Now that you know all this stuff, you may say “Understood, let's test those database entries out” so, in your tests, you start typing `String query = "SELECT * FROM …";` ***sighs***

- 🚫 **Do not** perform SQL queries manually.

  The reasons are:

  - 💩 TAEs (and sometimes devs too) *usually* are not good at writing SQL statements.
  - 💩 If you write good queries, then it most likely will be useful for a *particular* test, because it will have *just* the required fields and *exact* JOINs between tables that you *actually* need to work with. So you can’t reuse it in other tests. Leading to changes in multiple places if queries need to be updated.
  - 💩 If you write poor queries, then performance and tests run times explode automatically (SELECT * FROM .. UNION/JOIN <6 other tables>).
  - 💩 If you are writing *all* your SQL statements, then I’d say there is a *good chance* that you are *also* manually:
    - 😐 Creating connections to databases
    - 😐 Executing the prepared statement
    - 😐 Extracting resultsets into (hopefully) local variables in every test
    - 😐 Forgetting to close connections
    - 😐 And so on...

- ✓ Use ORM libraries like Hibernate, Sormula or pBeans[2] to delegate the dirty work.
- ✓ This allows you to keep working in an OOP way.
- ✓ Improves the readability of your tests.

Despite Hibernate being a mammoth framework - that we mortals will likely never use more than 0.1% of what it does - I suggest using it because:

1. It will fulfil your needs
2. It works
3. You will find help in forums all over the internet

Simple Hibernate stand alone usage

1. Include the following dependency in your Maven project:

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>5.2.1.Final</version>
</dependency>
```

Hibernate Maven dependency

2. You are going to need a driver for the DB you are going to connect to. For simplicity's sake, I've used H2, an in-memory database, but you can choose whatever you want.

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <version>1.4.191</version>
</dependency>
```

H2 Maven dependency

3. Create a *hibernate.properties* file in your resource folder that will hold the connection information for Hibernate:

```
hibernate.connection.driver_class=org.h2.Driver
hibernate.connection.url=jdbc:h2:file:./tmp_db_test
hibernate.connection.username=sa
hibernate.connection.password=
hibernate.dialect=org.hibernate.dialect.H2Dialect
hibernate.hbm2ddl.auto=create-drop
```

hibernate.properties file contents

4. Decorate a little bit any existing POJO that you may have with some Hibernate annotations:

```
@Entity
@Table(name = "POJOS")
private class MyPojo {

   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   @Column(name = "ID")
   private Long id;

   @Column(name = "NAME")
   private String name;

   @Temporal(TemporalType.TIMESTAMP)
   @Column(name = "DATE")
   private Date date;
```

```
    public MyPojo(String name, Date date) {
        this.name = name;
        this.date = date;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Date getDate() {
        return date;
    }
}
```

Simple POJO with some Hibernate annotations

5. And test it out!

```
SessionFactory sessionFactory = new Configuration()
        .addAnnotatedClass(MyPojo.class)
        .buildSessionFactory();

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

session.save(new MyPojo("Ruso", new Date()));

session.flush();

tx.commit();

session.close();
```

Simple Hibernate stand alone usage

> ⚠ **Warning**
>
> Hibernate's Session is **not thread safe**.
> If you are going to use it in multithreaded environments a possible solution would be to contain it in a ThreadLocal<T>.

Service Testing with Spring

Externalize configuration values and constants

🚫 **Do not** hardcode values into your framework.

🚫 **Do not** have a public class with public static fields manually mapped to properties. This implies that the more the properties you have, the more this class needs to be manually changed.

✓ Take advantage of Spring Profiles feature, which allows to define beans and configurations per profile (environment). This means that Spring will automatically filter out object instances and configurations that have nothing to do with the current profile and leave them out. So you can have defined in its context everything you will ever need, but just use whatever you may need at a time.

In practical teams, this means that you can make Spring load the properties file of your choice with a @Configuration class such as the following one:

```
@Configuration
public class Properties {

    @Profile({"dev", "default"})
    @Configuration
    @PropertySource("classpath:dev.properties")
    public static class ConfigDev {
    }

    @Profile("qa")
    @Configuration
    @PropertySource("classpath:qa.properties")
    public static class ConfigQA {
    }

    @Profile("staging")
    @Configuration
    @PropertySource("classpath:staging.properties")
    public static class ConfigStaging {
    }
}
```

Now, if you run your tests passing a JVM parameter such as `-Dspring.profiles.active=qa`, your tests will take the configuration properties from a *qa.properties* file, for example.

Access databases via Spring JPA

✓ Use Spring (Core + JPA modules) in your project in order to transparently create, read,

update and delete (CRUD) entities from DBs without any pain.

1. Define a Java interface to model operations that can be performed on DAO. By extending CrudRepository interface you gain basic CRUD operations. You can also extend from PagingAndSortingRepository for additional operations.

```
public interface UsersRepository extends PagingAndSortingRepository<User, Long> {

   User findByEmail(String email);

}
```

Automagic Spring JPA repository interface for dealing with User entities

2. In your data source related `@Configuration` class, make sure to define also `@EntityScan(basePackages = <package_entities>)` and `@EnableJpaRepositories(basePackages = <package_repositories>)` annotations. You can also use the class based (type safe) version via their `basePackageClasses` attribute.

```
@EntityScan(basePackageClasses = BaseEntity.class)
@Configuration
public class DataSources {

   @EnableJpaRepositories(basePackageClasses = UsersRepository.class)
   @Configuration
   public static class UsersDataSource {
       @Autowired
       Environment env;

       @Bean
       public DataSource getDataSource() {
           DriverManagerDataSource dataSource = new DriverManagerDataSource();
           dataSource.setDriverClassName("com.mysql.jdbc.Driver");
           dataSource.setUrl(env.getRequiredProperty("db.users.url"));
           dataSource.setUsername(env.getRequiredProperty("db.users.user"));
           dataSource.setPassword(env.getRequiredProperty("db.users.password"));
           return dataSource;
       }
   }
}
```

About HATEOAS

Test Cases samples

API Test Case Sample

Test Case #	175	**Test Case Name**	Edit Category Group
System	QDBX API	**Sub System**	Category Groups
Designed By	Juan Krzemien	**Design Date**	March 31th 2016

Preconditions
The following scenarios must be fulfilled before running the test case:

- A known existing Category Group should be available or must be created as part of this test fixture
- At least, one Category should exist or must be created as part of this test fixture

Step	Action	Expectation

1	Populate a CategoryGroup object with:
	● Category Type
	● Category Group
	● At least, one category Id is defined in Categories list
		{
		"CategoryType": {
		"Name": "**Super**"
		},
		"Categories": [
		{
		"Id": "FBMN",
		},
		...
		],
		"Name": "**Oral Care**"
		}

2	**PUT** against */categoryGroups* using **CategoryGroup** from **step 1** as body content.	HTTP response code should be **201** (CREATED)
3	Verify **Location** response header	Should exist Should not be empty/null Should match format https://environment/WAPI/v1/categoryGroups/Super/Oral%20Care
4	Verify **Errors** in response body	Should not be any error present in response
5	**GET** using the content of **Location** header (*/categoryGroups/{type}/{group}*) received in **step 3**.	- Response should be HTTP 200 (OK).

6	Verify returned **CategoryGroup**	- Should not be null - Should match **CategoryGroup** from **step 1** in every field, including amount of Categories.

Postconditions

- Remove any entity created during test fixture from environment's DB, if any.
- Remove created **CategoryGroup** from environment's DB.